FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

GPU-based Video Processing in the Context of TV Broadcasting

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Heinrich Fink
Matrikelnummer 0425503

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Dr. Anton Fuhrmann

Wien, 14.08.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

GPU-based Video Processing in the Context of TV Broadcasting

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Heinrich Fink
Registration Number 0425503

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Dr. Anton Fuhrmann

Vienna, 14.08.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Heinrich Fink
Schwarzingergasse 1/7, 1020 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I thank my supervisor Michael Wimmer. I was working with him for many years at the Institute of Computer Graphics, TU Vienna. His encouragement and exceptional support during this time was invaluable. While working as his teaching assistant, he trusted me with the task to redesign his practical course *Introduction to computer graphics* and helped me to publish this work as a journal paper. He has been a great mentor and teacher.

This thesis was written while I was working as a researcher at VRVis, Vienna. I thank my supervisor, Anton Fuhrmann. We had many inspiring conversations about the topics of this thesis, and he provided me with a lot of good advice.

I would also like to thank Walter Kuntner, CEO of ToolsOnAir, for providing me with the time and flexibility necessary to finish my thesis, and for his trust in my work. He made it possible for me to attend the SigGraph 2013 conference, which influenced my work in many aspects. Without him, this thesis would not have been possible.

I thank Thomas True at NVIDIA for taking time to discuss the results of this thesis with me, and for sharing his expertise in the area of GPU-based broadcast engineering.

NVIDIA and AMD generously provided hardware devices that I was able to test with the implementation of my thesis. I thank them for their support.

During the time of writing this thesis, I spent a lot of time at the Institute of Computer Graphics. I would like to thank the head of this institute, Werner Purgathofer, and everyone working there for their support. I have many great memories of my time studying there.

Finally, I would like to thank Lena and Lilja for being there.

Abstract

This thesis investigates GPU-based video processing in the context of a graphics system for live TV broadcasting. Upcoming TV standards like UHD-1 result in much higher data rates than existing formats. Processing such data rates while satisfying the real-time requirement of live TV poses a particular challenge for the implementation of a software-based broadcast graphics system. In order to reach the required data rates, the software needs to process image data concurrently on the central processing unit (*CPU*) and graphics processing unit (*GPU*) of the machine. In particular, the transfers of image data between main and graphics memory need to be overlapped with CPU-based and GPU-based executions in order to maximize data throughput. In this thesis, we therefore investigate the following questions: Which methods are available to a software implementation in order to reach this level of parallelism? Which data rates can actually be reached using these methods?

In order to answer these questions, we implement a prototype of a software for rendering TV graphics. To take advantage of the GPU's ability to efficiently process image data, we use the *OpenGL* application programming interface (*API*). We use advanced methods of OpenGL programming to render high-quality video and increase the level of employed parallelism of the GPU. We implement the transcoding between RGB and the professional video format *V210*, which is more complex to process than conventional consumer-oriented image formats. In our software, we apply the *pipeline* programming pattern in order to distribute stages of the video processing algorithm to different threads. As a result, those stages execute concurrently on different hardware units of the system. Our prototype exposes the applied degree of concurrency to the user as a collection of different optimization settings. In order to evaluate these optimizations, we integrate a profiling mechanism directly into the execution of the pipeline. This allows us to automatically create performance profiles while running our prototype with various test scenarios. The results of this thesis are based on the analysis of these traces.

Our prototype shows that the methods described in this thesis enable a software program to process high-resolution video in high quality. The results of our evaluations also show that there is no single best optimization setting for every GPU architecture. Different driver implementations and hardware features require our prototype to apply different optimization settings for each device. The ability of our software structure to dynamically change the degree of concurrency is therefore an important feature. For broadcasting software that is expected to perform well on a range of hardware devices, this is ultimately an essential feature.

Kurzfassung

Diese Arbeit beschäftigt sich mit GPU-basierter Verarbeitung von Video im Kontext des Grafiksystems eines Live-TV Senders. Kommende TV Spezifikationen, wie UHD-1, haben eine besonders hohe Datenrate an Bildinformationen zur Folge. Die Echtzeitverarbeitung solcher Datenraten stellt eine besondere Herausforderung für die Implementierung eines Software-basierten TV-Grafiksystems dar. Um die erforderten Datenraten zu erreichen, muss das Programm seine Berechnungen auf Haupt- und Grafikprozessor (*CPU* und *GPU*) parallel ausführen. Insbesondere müssen die Übertragungen der Videobilder zwischen Haupt- und Grafikspeicher über den PCIe-Bus mit den Berechnungen der CPU und GPU überlappt werden, um eine effiziente Ausführung zu garantieren. Diese Arbeit beschäftigt sich daher mit der Frage, welche Methoden für die Implementierung eines solchen Grafikprogramms zur Verfügung stehen, und welche Datenraten damit effektiv erzielt werden können.

Um diese Fragen zu beantworten, implementieren wir den Prototypen einer Software für das Rendering von TV-Grafiken. Dabei setzen wir die Programmierschnittstelle *OpenGL* ein, um die Fähigkeiten des Grafikprozessors für die effiziente Verarbeitung von Bilddaten auszunützen. Wir zeigen fortgeschrittene Methoden der OpenGL-Programmierung, welche die Bearbeitung von professionellem Videomaterial erleichtern, und helfen, den maximalen Grad an Parallelität in der Ausführung des Grafikprozessors zu erreichen. Insbesondere zeigen wir die GPU-basierte Verarbeitung des Studioformates *V210*, das im Vergleich zu herkömmlichen Bildformaten besondere Herausforderungen an die Implementierung stellt. Unser Prototyp basiert auf dem Softwaremodell einer Pipeline. Das Programm ist dadurch in der Lage, einzelne Schritte der Bildverarbeitung zu parallelisieren, und auf mehrere Prozessoren dynamisch zu verteilen. Dadurch können wir verschiedene Optimierungsverfahren einsetzen, um den Datendurchsatz des Programms zu maximieren. Um diese Verfahren und generell die Implementierung des Prototyps zu analysieren, integrieren wir die Messung des Laufzeitverhaltens direkt in unsere Software. Das ermöglicht die automatisierte Erstellung von Profilen verschiedener Testszenarios, deren Analyse die Basis für die Resultate dieser Arbeit bilden.

Unser Prototyp zeigt, dass die in dieser Arbeit vorgestellten Methoden die Echtzeitverarbeitung von hochauflösendem Videomaterial in hoher Qualität ermöglichen. Unsere Ergebnisse zeigen auch, dass für verschiedene GPU-Architekturen unterschiedliche Optimierungsverfahren eingesetzt werden müssen, um den optimalen Durchsatz zu erreichen. Die Fähigkeit unserer Software, die Optimierung der Videopipeline dynamisch anzupassen, ist also eine besonders wichtige Eigenschaft, und letztendlich Vorraussetzung für die Implementierung eines marktreifen Grafikproduktes, das auf unterschiedlichen Hardware-Konfigurationen effizient laufen soll.

Contents

CHAPTER 1

Introduction

This thesis implements and analyzes GPU-based video processing using the OpenGL API in the context of TV broadcasting.

1.1 Motivation

The production and delivery of a modern TV program is a fully digital process. The images we see on TV today are the result of a large chain of digital video processing components. In broadcasting studios, the last stage of the production chain is the *playout* server. It produces the final video frames of a TV program, just before they are sent on-air. The playout stage is usually also responsible for adding real-time graphics to the signal (see Figure 1.1). This rendering of broadcast graphics happens at a very critical time in the production chain. If it fails to deliver video frames at the real-time rate of the carrier signal, the program that is on air is interrupted. The playout server's renderer therefore has to operate reliably in real time.

Broadcasting studios used to employ highly specialized and expensive hardware to inscribe graphics into video material for live TV. As computers became more powerful and as file-based workflows became widely adopted, these systems have been replaced by software systems that run on commodity hardware. These systems are able to render video in real time when taking advantage of the graphics processing unit (GPU) of the machine. A GPU is optimized for processing image data and therefore ideal for video rendering.

The increase in resolutions and frame rates of recent video standards like UHD-1 or 1080p50/60 requires the playout software to process much higher data rates than with previous formats. Modern graphics processors provide the compute throughput that is necessary to render these new formats. However, in order to process video on the GPU, video frames first have to be transferred over the machine's PCI-Express (PCIe) bus to video memory. In a typical playout application, video frames also have to be transferred back to main memory for playout through the video output component. Because the PCIe bus is limited in bandwidth and shared between multiple processes on the system, the transfer to and from video memory introduces a

Figure 1.1: Broadcast graphics are essential to the experience of a TV program. They are added last in the production chain and have to be rendered reliably in the rate of the carrier signal. This screenshot shows demo graphics of the ToolsOnAir *just:out* playout system.

significant latency to the pipeline of the software. This makes it difficult for the playout software to process high-resolution video at the required real-time rates.

This problem can be addressed by parallelizing the pipeline of the playout process. Today's multi-core CPU systems enable the asynchronous handling of video I/O. Modern graphics cards contain an additional DMA hardware unit besides the GPU, which allows for parallel video memory transfer while executing the GPU for rendering. Such components are widely available to commodity-hardware machines. However, in order to take advantage of these components, the playout server's implementation needs to use a software model for its video processing pipeline that supports this highly parallelized execution model. The goal of this thesis is to describe such a model, and to provide a proof-of-concept implementation which demonstrates the benefits of a parallelized video processing engine that can be used within a playout software solution.

1.2 Research question

This thesis investigates the following questions:

- Which tasks of a video processing pipeline can be executed in parallel on different hardware?
- Which software patterns are needed to support the concurrent execution of these tasks?
- How to use a graphics API like OpenGL to render and transfer video concurrently?

- What are the maximum data rates that can be reached using commodity-based hardware and what are the limiting factors?

- Are these data rates high enough to process new high-resolution video formats in real time on commodity hardware?

1.3 Contributions

The main contribution of this thesis is the design, implementation and practical analysis of a software framework we call *FrameBender*. This framework provides a highly parallelized video processing pipeline. It implements a software model that allows the concurrent execution of CPU- and GPU-based processing tasks. In comparison to a conventional serialized solution, our implementation uses less time for the transfer of frames and therefore enables video rendering in real-time rates even for high resolutions like UHD-1.

Other contributions are:

- Profiling and visual analysis of the pipeline's performance characteristics.

- A comparison of the pipeline's runtime behavior on different GPU architectures using different configurations.

- Best practices for optimizing data throughput for OpenGL video processing.

- Design and implementation of a doubly-linked pipeline pattern that enables asynchronous two-way communication between individual stages of the pipeline.

- GPU-based algorithms to transcode between linearly coded RGB and the *V210* video format efficiently and in high quality using the *GLSL* OpenGL shading language.

- A comparison between various V210 transcoder implementations using different versions of GLSL.

Our framework shows that it is possible for OpenGL-based software to process new high-resolution video formats in real time with commodity hardware. As opposed to many other benchmarking applications, where only isolated tasks are being tested, our framework performs a complete video rendering task. The behavior and the optimizations of our program are therefore much more representative of what is required in a real-world video application than it is when measuring only an isolated part of the problem (e.g., measuring only the bandwidth of transfers between CPU and GPU). Through analysis of the captured and visualized runtime behavior, we were able to identify best practices for video processing in software design and the use of OpenGL. Our prototype further allowed us to compare the benefits of different GPU architectures in the context of real-time video processing. We also compared our GPU-based solution to process V210 with an existing CPU-based algorithm and showed that ours performs more efficiently while providing the same high level of image quality.

1.4 Overview

In Chapter 2, we provide background information on the theory and practices of video engineering relevant to this thesis. This chapter also summarizes GPU hardware architectures and provides an overview of the OpenGL features that were used in the implementation of the prototype. Chapter 2 concludes with an overview of related research and existing playout software systems. The general model and software design of our framework is described in Chapter 3. It focuses on the algorithms and concepts of the video pipeline, which are independent of their actual implementation. Chapter 4 then presents details about the prototype implementation of the framework. It describes the C++ infrastructure that supports the pipeline execution, explains how OpenGL has been used to implement the pipeline, and describes the optimization techniques that were implemented. The V210 encoding and decoding process is described in Chapter 5. This chapter explains how recent versions of GLSL benefit the design and performance of the GPU-based transcoding algorithm. Chapter 6 then discusses results in performance, quality and stability of various configurations of the framework running on different GPU architectures. The conclusion of these experiments is given in Chapter 7, which concludes the thesis and provides an outlook to further work.

CHAPTER 2

Background

2.1 Video engineering

In this chapter, we explain common video coding and conversion techniques that are relevant to the context of this thesis. We begin this chapter by describing the principal of perceptual coding, which is fundamental to image coding of video. We then explain the process to reproduce images of professionally captured video material and provide details of how to convert HD video to computer-RGB space for processing. The concepts described in this chapter are largely based on Poynton [26].

2.1.1 Perceptual coding

A picture element (*pixel*) of a digital image transports a property we intuitively call *brightness*, i.e. a sensation of an area emitting more or less light. The correct technical terms for this property are the following:

- *Intensity*: The physical quantity of radiant power into a particular direction.
- *Radiance*: Intensity per unit-projected area.
- *Luminance*: Radiance weighted by a sensitivity function that models the uneven response of human vision to different wavelengths of light.

The coding of an image is said to be *perceptually uniform* if a small change in the coding value is equally perceived across the full coding range. Human vision is only able to distinguish two luminance values if their ratio exceeds 1.01, this is based on the *Weber law*.

Let us consider an 8-bit coding of image values with values ranging from 0 to 255. If each code increment represented an increment in relative luminance, we would run into the so-called *code 100* problem (see Figure 2.1). This problem stems from the fact that at code 100, one increment to 101 represents exactly the Weber fraction, i.e. the least-noticeable difference. That

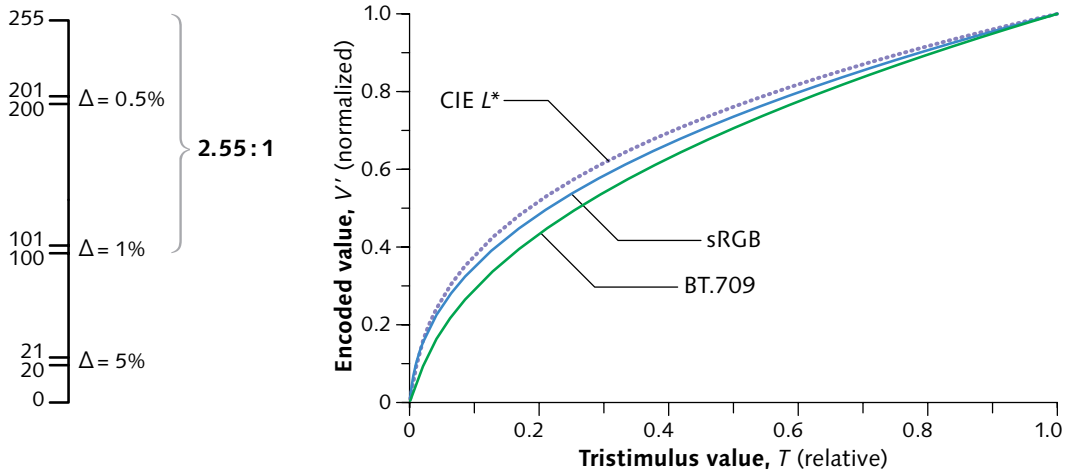


Figure 2.1: On the left side we show a visualization of the *code 100 problem*. The ratio between luminance values varies across the coding scale, which results in less coding efficiency and even visual artifacts in darker regions. On the right side, we see the plotting of three image transfers that relate to the human response curve of luminance. $CIE\ L*$ denotes the colorimetric quantity *lightness*, *sRGB* is largely used in computer imagery, and *BT.709* is the standard for encoding HD video. Both images are courtesy of Poynton [26].

means for all code values above, a single increment is not noticeable anymore, e.g. the increment between 200 and 201 is 0.5

A solution to this problem is to apply a non-linear transfer on the original signal that is adapted to the human response to luminance. This process is commonly called *gamma correction*.

The colorimetric model for the human response to luminance is defined by a quantity called *lightness*, which is defined as *the brightness of an area judged relative to the brightness of a similarly illuminated area that appears to be white or highly transmitting* [5]. Lightness is denoted using the symbol $L*$ and it is subject to a non-linear model that resembles the response function of human vision, adapted to a certain lighting environment. However, most digital imaging standards like sRGB or BT.709, use a power function to model this perceptual curve (see Figure 2.1).

The conversion of image intensities to a perceptual scale allows a much more effective coding. In order to reproduce the original image values on a display, the perceptual coding has to be reverted. The electron-gun of Cathode-ray-tube (CRT) monitors are subject to a response function between input voltage and light intensity which coincidentally models the inverse of the transfer function used for perceptual coding. Digital monitors do not use electron-guns and therefore need to apply the decoding function using signal processors.

2.1.2 Image reproduction

While shooting a scene, a camera captures the scene luminance and encodes it to digital images using the *opto-electrical-conversion-function* (OECF). As discussed in the previous chapter, gamma correction is performed by applying a power function using some fixed exponent in order to efficiently store the captured luminance. At the reproduction of the captured scene, the display applies an *electro-optical-conversion-function* (EOCF), which converts the image coding of the video into linear values used for displaying. We would intuitively expect the EOCF to be the inverse of the OECF. In practice, however, this is not the case. There are two additional factors that need to be weighted in:

- *Scene tone alteration*: The original scene is shot at surroundings of very high luminance (e.g. a scene in daylight has a surround luminance of 30000 nits[1]). However, the display on which the scene should be presented is usually located in a much dimmer environment, e.g. in about 300 nits. Displaying the same spectra with less luminance results in the so-called *Hunt effect* where the same colors look less colorful in a dimmer environment. In order for the display to faithfully reproduce the impression of the originally shot scene, the camera's OECF is usually designed to apply a form of *scene tone alteration*, which compensates for the *Hunt effect*. However, this results in a non-linear relationship of luminance captured at the scene and relative displayed luminance. E.g., the BT.709 standard for HD video implies an end-to-end power function with an exponent of 1.2 (see Chapter 2.1.2.1). The EOCF is therefore not just the inverse of the OECF.

- *Creative intent*: In a real production, the camera's OECF is usually tweaked to somehow manipulate the captured colors as desired in the creative process. In addition to that, post processing is performing color grading to achieve the desired look of a particular production. The final *approval* of video is done on a studio reference display. In order for any other display to faithfully reproduce the colors as approved by production, the EOCFs involved in the process need to be known. In this scenario of content creation, the OECF is therefore irrelevant, and we refer to our video data as being *display-referred* (as opposed to *scene-referred*).

2.1.2.1 BT.709

The ITU recommendation BT.709 [16] is a standard that intends to define the parameters for the HDTV display pipeline and programme exchange. It defines the transfer with an advertised exponent of 0.45 that is expected to be imposed at the camera (the OECF). Because the pure power function would have a infinite slope at 0, which causes artifacts during encoding, the function is split into a linear segment for low intensities and a power-segment for higher intensities (see Figure 2.2). The BT.709 OECF transfer is defined as

$$OECF_{709} = \begin{cases} 4.5T; & 0 \leq T \leq 0.018 \\ 1.099T^{0.45} - 0.099; & 0.018 \leq T \leq 1 \end{cases}$$

[1] candela per square meter

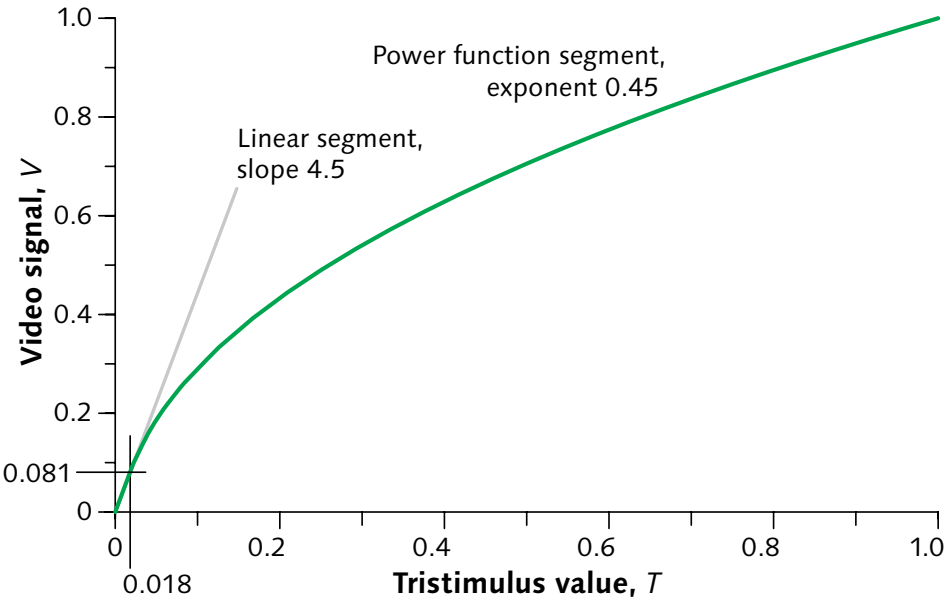


Figure 2.2: Image transfer function of BT.709-conforming video. For lower intensities, this function uses a linear segment in order to avoid noise. Image courtesy of Poynton [26].

where T is the linear light quantity.

This linearized adaptation is effectively closer to a power function using an exponent of 0.5 than the advertised gamma of 0.45 (see Poynton [26]). Unfortunately, BT.709 did not define the EOCF for the reference display that should be used when approving BT.709-conforming material. As we have discussed in the previous chapter, this is important for display-referred workflows in most of todays production scenarios. The recently approved BT.1886 standard [17] fixes this problem, and defines the reference display to use a power exponent of 2.4. Using BT.709's effective OECF exponent of 0.5, we can now see, that the end-to-end power exponent, from acquisition to display is 1.2. This is the implicitly defined rendering intent as we discussed previously.

Besides the above transfer function, BT.709 also defines the components to calculate *luma* (see Chapter 2.1.6). The color space of BT.709 is defined to use the same chromaticity primaries as sRGB.

2.1.3 sRGB

The sRGB [13] transfer function (see Figure 2.1) assumes a much brighter displaying environment than BT.709 transfer, as would be the case for an average office environment. The sRGB transfer is defined as

$$sRGB = \begin{cases} 12.92T; & 0 \leq T \leq 0.0031308 \\ 1.055T^{\frac{1}{2.4}} - 0.055; & 0.0031308 \leq T \leq 1 \end{cases}$$

The end-to-end power function is 1.1, i.e. much less than in BT.709. sRGB expects the display to apply a 2.2 power function for decoding.

2.1.4 BT.709 Y'CbCr coding

Digital video is encoded using a single component for lightness (*luma*) and two other components containing color information (*chroma*). One of the reasons for this type of coding is that human vision is less sensitive to changes in color than in lightness. *Chroma subsampling* reduces the spatial resolution of the chroma channels and takes advantage of this property (see Chapter 2.1.5).

The lightness component Y' (*luma*) was often falsely referred to as *luminance*. However, as opposed to luminance, the calculation of Y'CbCr is based on perceptually coded R'G'B' triples, i.e. gamma correction has already been applied [2]. The reason for this is related to the historical coincidence of the CRT display to naturally provide for the decoding function of the gamma-corrected video signal. More details on this engineering practice is given by Poynton [26], Chapter 10.

For BT.709-conforming video, luma is calculated as

$$^{709}Y' = 0.2126R' + 0.7152G' + 0.0722B' \tag{2.1}$$

The CbCr channels are based on the the following differences:

$$Cb = B' - Y' \tag{2.2}$$
$$Cr = R' = Y' \tag{2.3}$$

Actual formulae for the calculation of Y'CbCr will be given in Chapter 2.1.6.

2.1.4.1 Studio swing

In consumer imaging, all available code values are used to represent valid colors, e.g. RGB uses intensities from 0 to 255 in an 8-bit coding. This is called *full swing* coding. For professional video content, reference black and white values are placed at offsets of the interface. For example, in 8-bit coding, Y' defines reference black at level 16 and reference white at level 235. In processing range, these values map to 0 and 1, respectively. These regions are called foot- and headroom, the coding is referred to as *studio swing* coding. This extra coding space is partly due to historical reasons of analog transmission. Today, with processing chains that are

[2] We refer to a perceptually coded quantity using the apostrophe ' (e.g. R'G'B'), and to linearly coded values without using the apostrophe (e.g. RGB).

purely digital, head- and footroom is for example used to conserve transients of digital filters (e.g. undershoot and overshoot). The outermost interface values (0 and 255 in 8-bit coding) are reserved, they must not be used for storing image data. The other values within the head- and footroom regions, however, are allowed to be used as well. Values in the headroom are sometimes referred to as *superwhite* and basically provides for extended dynamic range. It is common practice in productions to use the headroom coding space for very bright highlights of a scene. Unfortunately, some video processing tools always clip the head and footroom, and thereby compromise the original material. Poynton therefore recommends to always keep values in the head and footroom [36].

BT.709 Y'CbCr coding always uses studio swing. In an 8-bit coding system, Y'CbCr has the following extent:

$$Y' = [1, 254] \tag{2.4}$$

Reference black is at level 16, reference white at level 235. In 8-bit coding, CbCr values have the extent of:

$$Cb, Cr = [16, 240] \tag{2.5}$$

Chroma values are stored with an offset of 128, i.e. they are actually represented by the range $[-112, 112]$.

For 10-bit coding, the same interface levels of 8-bit coding are used with the two extra bits appended as the least-significant bits, which provides for the extra precision.

2.1.5 Chroma subsampling

Because the visual acuity of human vision is less for color than for lightness, the chroma values of the Y'CbCr coding are commonly resampled to a lower resolution in order to save bandwidth during transmission. Figure 2.3 shows the 4:2:2 subsampling scheme for BT.601 (which is also often applied to BT.709 video). Here, the vertical resolution is the same for luma and chroma, but the horizontal resolution is half. The 4:2:2 subsampling schema is therefore a lossy compression with a compression rate of 1.5:1. For this particular scheme, the chroma samples are sampled at the same spatial location as the luma samples, they are *cosited*.

2.1.5.1 Reconstruction and decimation filters

The conversion from Y'CbCr 4:4:4 to 4:2:2 is a downsampling process of the chroma values to to half of their horizontal sampling frequency. According to the *sampling theorem*, at a sampling frequency of f_s signals in the range of $[DC, 0.5f_s]$ can be reconstructed. Therefore, when downsampling to half the resolution, only signals of frequencies in the range of $[DC, 0.25f_s]$ can be reconstructed. Low-end systems simply drop alternating chroma samples when downsampling to 4:2:2. For signal between $0.25f_s$ and $0.5f_s$ this will cause aliasing, because the new sampling is not able to represent these frequencies. Therefore, the aliasing frequencies need

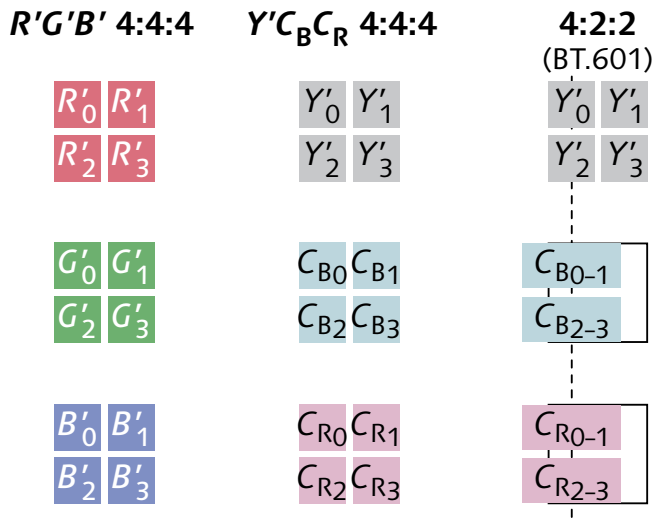


Figure 2.3: Y'CbCr 4:2:2 uses *cosited* chroma subsampling, i.e. the spatial location of the chroma samples is the same as the location of the luma sample. Image courtesy of Poynton [26].

to be removed using a low-pass filter on the original chroma values. This is usually achieved by applying a finite-impulse-response (*FIR*) filter on the original chroma values with a corner frequency of about one quarter of the original sampling frequency. A very basic approach would be a FIR filter with the components $[0.25, 0.5, 0.25]$ (*triangle* filter), which results in a cosited downsampling.

The upsampling from 4:2:2 subsampling to 4:4:4 also requires a reconstruction filter to be applied. Similar to above, a simple duplication of neighboring pixels (*nearest neighbor* or *box filter*) will cause aliasing. A better, although not optimal, approach is to average neighboring chroma values (this is also a *triangle* filter, or *linear interpolation*).

The theoretical optimal filter for downsampling and reconstruction would be the *sinc* filter, which in the frequency domain is a box filter, i.e. providing the perfect frequency cut-off. However, the unmodified sinc filter has an infinite frequency response, and can therefore not be applied in reality. There are several methods to *window* the sinc filter in order to limit its frequency response. Even then it is common to further manually tweak the filter coefficients for better performance. However, this discussion is beyond the scope of this thesis. Poynton [26] provides a very good overview of the theory of filtering and sampling in the context professional video. Further reading is provided by Mitra and Kaiser [23], and Rorabaugh [27].

2.1.6 Converting between R'G'B' and 10-bit BT.709 Y'CbCr

In the following, we summarize the formulas for converting back and forth between perceptually coded R'G'B', stored in floating point, and 10-bit Integer-encoded Y'CbCr. The source for these calculation is described by Poynton [26].

For a perceptually coded R'G'B' representation with its values in the unit range of $[0, 1]$, we can calculate the luma and chroma differences according to BT.709 as follows:

$$\begin{bmatrix} {}^{709}Y' \\ B' - {}^{709}Y' \\ R' - {}^{709}Y' \end{bmatrix} = {}^{709}M \bullet \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} \tag{2.6}$$

where

$${}^{709}M = \begin{bmatrix} 0.2126 & 0.7152 & 0.0722 \\ -0.2126 & -0.7152 & 0.9278 \\ 0.7874 & -0.7152 & -0.0722 \end{bmatrix} \tag{2.7}$$

In order to limit the excursion of the resulting chroma values to unit extent in the range of $[-0.5, 0.5]$, we further scale the rows of matrix ${}^{709}M$ by multiplying a scaling matrix:

$$S' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \dfrac{0.5}{0.9278} & 0 \\ 0 & 0 & \dfrac{0.5}{0.7874} \end{bmatrix} \tag{2.8}$$

$${}^{709}M' = S' \bullet {}^{709}M \tag{2.9}$$

The resulting matrix ${}^{709}M'$ now converts R'G'B' to Y'CbCr in the range of $(Y', Cb, Cr) = [[0, 1]; [-0.5, 0.5]; [-0.5, 0.5]]$. In order to map this range into the extent of $[0, 876]; [0, 896]; [0, 896]$ for 10-bit unsigned integer coding, we add another scaling matrix:

$${}_{876}S = \begin{bmatrix} 876 & 0 & 0 \\ 0 & 896 & 0 \\ 0 & 0 & 896 \end{bmatrix} \tag{2.10}$$

$${}^{709}_{876}M' = {}_{876}S \bullet {}^{709}M' \tag{2.11}$$

We now need to apply the interface offset for luma and chroma values, and finally calculate the 10-bit Y'CbCr coding as follows:

$$\begin{bmatrix} {}^{709}_{876}Y' \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 64 \\ 512 \\ 512 \end{bmatrix} + {}^{709}_{876}M' \bullet \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} \tag{2.12}$$

In order to reconstruct R'G'B' from BT.709 video, we apply the inverse of the above function:

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = {}^{709}_{876}M'^{-1} * \left(\begin{bmatrix} {}^{709}_{876}Y' \\ Cb \\ Cr \end{bmatrix} - \begin{bmatrix} 64 \\ 512 \\ 512 \end{bmatrix} \right) \tag{2.13}$$

R'G'B' values that are reconstructed from an Y'CbCr encoding can contain values outside the unit range. While negative R'G'B' values are not representing any real colors, a later conversion back to Y'CbCr is improved in quality if those values are kept during the process, e.g. by using a floating-point format for storing R'G'B'.

2.1.7 Mixing sRGB with BT.709 content

A typical scenario of rendering broadcast graphics is to overlay sRGB images on top of BT.709 video. In the previous chapter, we showed how to convert 10-bit BT.709 to R'G'B'. We now have to decide on a common coding space in which we can blend the sRGB and BT.709-originating materials. BT.709 and sRGB are defined to use the same chromaticity primaries, but their transfer function (*gamma* value) is defined differently.

A lot of video processing applications simply blend in perceptual R'G'B' space, ignoring the fact that this space is non-linear and that linear operations like blending can result in different results than expected.

BT.709/BT.1886 defines that the EOCF is a power function with an exponent of 2.4. As presented by Poynton, this function, however, assumes a dimly lit reference viewing environment of about 100 nits [36]. Under brighter viewing conditions, BT.709 should be displayed using a power function of 2.2, like sRGB. When mixing sRGB with BT.709, it is therefore sensible to apply the 2.2 power function of sRGB to *both* media. This is common in high-quality video rendering applications, and the recommended practice for this situation [36].

2.2 Hardware

2.2.1 Graphics processing units

The graphics processing unit (*GPU*) is a processor that is optimized to process data-parallel tasks using a large number of concurrently running cores. In most applications, a GPU is used to render graphics and to process image-like data. Rendering is achieved by the hardware-based graphics pipeline of a GPU. In several parallelized stages, this pipeline performs the conversion of abstract geometry data like triangles into actual pixel values on screen. Certain stages are programmable using so-called *shader* programs. These allow a software program to influence the overall algorithm of the pipeline. For example, a *vertex shader* can be used to define where input geometry is located in space, and a *fragment shader* can be used to calculate lighting equations for the pixels that are covered by the rendered geometry. A complete background on the graphics pipeline and the applied algorithms is given by Shirley and Marschner [30].

Today, GPUs contain a large number of *streaming multiprocessors* that are usually based on a wide SIMD architecture (single-instruction-multiple-data) architecture. These processors can be used to process other data-parallel computing problems than rendering graphics. There

is a strong trend towards solving generic computing problems on such streaming processors. Programming environments like CUDA [33] or OpenCL [24] exploit the architecture of a GPU for generic programming tasks. A tutorial and review of CUDA programming is given by Kirk [19]. An overview of OpenCL programming is given by Gaster [7].

In most currently available systems, high-performance GPUs are contained on a discrete graphics board, using dedicated video memory. In order for the GPU to process data, a DMA controller copies the data from main memory to the GPU's video memory over the PCI-Express (PCIe) bus. This bus is divided into lanes which can transfer data between two endpoints simultaneously (*full-duplex*). A PCIe slot is able to use multiple lanes at the same time, e.g. for graphics devices that require high-speed data transfers, 16 lanes are usually used. The bandwidth of a lane is calculated using *transactions per second*. Each transaction physically transfer 10 bits for one byte (8 bits) of actual data (encoding overhead). The PCIe 2.0 standard allows 5 Gigatransactions per seconds (GT/s) per lane. This results in a theoretical bandwidth of 500 MB/s for a single lane, full-duplex. A graphics device that is connected using a PCIe 2 slot using 16 lanes is therefore able to reach a theoretical bandwidth of 8 GB/s. In practice, a peak bandwidth of about 6.5 GB/s can be reached.

Another current trend in GPU hardware design is to engineer GPUs on the same silicon package as the CPU. For example, the Intel Haswell generation of CPUs provides a configuration with an accompanied GPU that uses on-chip embedded DRAM of 128 MB. This memory can be accessed by both the CPU and GPU, which eliminates the need to transfer data from main memory to video memory for GPGPU-based processing. This also allows for simpler programs using stream-processing languages like OpenCL. AMD offers similar hardware using their APU-line of products.

2.2.2 Video interfaces

The majority of professional video equipment transports video streams over the *serial digital interface* (SDI). SDI uses a coaxial cable and is widely used in existing broadcast studio infrastructure. The newest ratified standard of this interface, *3G-SDI* [31] provides data rates of 2.970 Gbit/s (~370 MB/s). This relatively little bandwidth requires to link together multiple SDI channels in order to operate at recent 4K-like resolutions. Blackmagic Design introduced a new version, 6G-SDI that allows to transport 4K in a single cable. This version yet has to be ratified by the SMPTE organization.

In order for a software program to access SDI-based equipment, several hardware vendors provide PCIe-connected devices that decode/encode SDI and provide the application with raw frame buffers. Most vendor SDKs of such devices provide frames in main memory, which could then be uploaded to the GPU for graphics rendering. In the same way, video frames have to be downloaded in order to be played out by the SDI-connecting device. The efficient implementation of such transfer is difficult. This thesis describes one solution to this problem. AMD and NVIDIA both cooperated with SDI-device manufacturers like AJA or Blackmagic Design to simplify this problem for software application programmers. Instead of the software having to explicitly manage the up- and download of video frames to the GPU, the video card's driver employs low-level operations of the graphics driver to transfer the video frame to the GPU

on behalf of the software application. NVIDIA calls their solution *GPUDirect for Video* [34], AMD provides a similar solution with the *SDI Link*[3] line of products.

SDI requires one or more dedicated cables per video stream. For large broadcasting studios, this results in high maintenance and complex logistics of cabling in order to connect routers, mixers, monitors, and so forth. A recent trend in video technology is to replace these legacy interface with networking-based technologies. For example, Audio-video bridging[4] is a standard in development that uses small extensions to layer 2 of the OSI networking stack in order to enable guaranteed latency for transfers of uncompressed video.

2.3 OpenGL

OpenGL provides software programs with an application programming interface (*API*) to graphics hardware in order to perform graphics-oriented tasks. The development of the OpenGL API is closely tied to the evolution of GPUs. GPUs evolved from specialized, purely graphics-oriented hardware to highly-programmable streaming processors. Over the last few years, the OpenGL API has similarly abandoned its original fixed-function programming model and replaced it with a shader-centric approach. Shaders are written in the GL shading language (*GLSL*) [18]. They are programs which are executed by the GPU's programmable hardware stages. They define how to process the data that is transferred through the graphics pipeline of the device. A typical OpenGL program uploads geometry data to the GPU, defines the geometry's position and orientation in space using a *vertex shader*, and defines the lighting and appearance of the geometry's visible surface in the *fragment shader*. These shaders are called by the GPUs *rasterizer*. The rasterizer processes the geometry and calculates which pixels on the screen are covered by the rendered geometry. A large part of the rasterizer is still implemented as fixed-function stages on the GPU. The *viewport* defines a rectangular portion of the screen which is backed by a *framebuffer*. By default, this area is the target of the drawing operation, and the state defining the viewport and framebuffer influences the execution of the rasterizer.

GLSL shaders are able to sample image data from *textures* and use these samples for further calculation. GPUs provide powerful sampling units that filter image data on behalf of the rendering application. These texturing units are usually also implemented by dedicated hardware units on the GPU.

In order to draw into an image rather than the screen itself, OpenGL allows to use an off-screen framebuffer which has a texture attached. The drawing operation is then writing into this texture. Shaders of following drawing operations able to read from this texture again.

A significant change in the evolution of OpenGL was to support the highly-programmable graphics pipeline of modern GPUs. This trend continued, and today in OpenGL 4 we have four programmable domains of the graphics pipeline: vertex, geometry, tesselation and fragment shaders. All of these stages are part of a graphics pipeline that somehow calculates pixel colors from more or less complex geometry. With the recent push to GPGPU-oriented programming environments, OpenGL also provided simpler interoperation with the OpenCL language. For

[3] http://www.amd.com/sdi-link

[4] http://www.ieee802.org/1/pages/avbridges.html

instance, OpenGL could be used to render a geometric scene into a texture. This texture is then read by an OpenCL kernel for applying advanced image-processing effects.

However, the OpenCL API is complex and interoperability with OpenCL-based executions also involves a certain cost in performance when synchronizing access to shared resources. In order to simplify the execution of generic computations related to graphics, OpenGL version 4.3 recently added a completely new shading stage, called *compute shaders*. This stage is completely independent of any pre-defined graphics-oriented semantics. However, compute shaders are able to read and modify OpenGL buffers and images. They are therefore able to modify data which are the input or output of the graphics pipeline execution. In comparison to OpenCL, compute shaders have the advantage of being written in GLSL. GLSL provides for more built-in functions, and existing GLSL sources (e.g. utility functions) can be re-used in compute shaders.

In the following, we are briefly summarizing the OpenGL features that are particularly relevant to our context of video processing. These features are going to be used in the implementation of our prototype video processing framework. We start with OpenGL 3.3 and then move onwards to OpenGL 4.3. A lot of features are not mentioned in this summary. For details about the OpenGL API we refer the reader to the official OpenGL and GLSL specifications [18, 29]. We always refer to the *core profile* of a specification, i.e. no legacy features are used.

2.3.1 Version 3.3

- **Framebuffer objects** (*FBO*s) for off-screen rendering are supported by default. Image processing algorithms can be executed by rendering a quad into an FBO having a texture attached. The fragment shader that is active during this rendering defines the image processing algorithm.

- **Blitting** operations enable fast copying between subregions of framebuffers. This is useful when an FBO is used for rendering into a texture, and when the result of this operation should be made visible on the screen's framebuffer.

- **Floating-point** textures are supported as a core feature. This is useful when high-precision video is processed and to avoid the clipping and quantization of integer rendering formats. Besides full-precision 32-bit floating-point, *half-precision* 16-bit floating-point formats are supported as well.

- **Timer queries** allow to measure the GPU-side execution times for executing commands originating from OpenGL. This can be used, for example, to accurately measure the overall execution time of a single render pass, without having to rely on external profiling tools. A good tutorial for this feature and an example use case is provided by Lux and Fuentes [6, 22].

- **Fence sync** objects are provided by OpenGL in order to synchronize the execution of OpenGL commands. This can be used to synchronize data access for OpenGL commands that are executed on multiple OpenGL contexts.

- **Integer textures** allow a shader to fetch the original integer coding of a texture, instead of the normalized range of $[0, 1]$. For video processing applications, this is useful when the

original Y'CbCr coding needs to be manually converted to RGB. This is often necessary, because video-related encoding standards like BT.709 (see Chapter 2.1.4) are not natively supported by OpenGL (as opposed to 8-bit sRGB-encoded images). OpenGL 3.3 also provides the `RGB10_A2UI` integer format which allocates 10 bits for three components packed into a single 32-bit word. This can be useful for representing 10-bit coded BT.709 video.

- **Non-power-of-two-textures**: This allows the user to define common 2D textures with dimensions that are not a power of two. Previously, this required the use of a specialized texture variant called *rectangle textures*. As GLSL 330 (the GLSL version defined by the OpenGL 3.3 specification) allows for texel-accurate fetching of any texture using integer coordinates with the command `texelFetch`, we can now use textures in the native dimensions of the video data without having to rely on specialized rectangular textures.

2.3.2 Version 4.0

- **GLSL 4.0** introduces built-in operations for integer bitfield-manipulations. It is common for 10-bit video coding, to store several 10-bit components in a single 32-bit word. New GLSL commands like `bitfieldInsert` and `bitfieldExtract` allow us to to extract these components directly in the shader from a sample of a `R32_UI` texture. This approach can be more efficient than using the format `RGB10_A2UI` to let OpenGL handle the extraction of the 10-bit words.

As we described in Chapters 2.1.4 and 2.1.6, the conversion between Y'CbCr-encoded video and RGB requires processing of the actual code words, e.g. applying the offset of the coding interface to accommodate head- and footroom. The ability of GLSL shaders (and therefore the ability of the GPU) to handle high-precision integer types natively is therefore crucial for accurate processing of video data.

2.3.3 Version 4.1

- **OpenGL debug output**: Instead of polling the GL for errors manually (using `glGetError`), this extension allows the host program to setup a callback that is notified when an error occurred. The user of OpenGL is able to specify whether this callback should happen asynchronously to the OpenGL API or synchronously. The latter allows to set a breakpoint directly in the callback function which then shows the stack trace into the program that caused the OpenGL error notification. The debug output extension also allows OpenGL drivers to provide performance hints, e.g. inefficient use of buffers, etc. This feature is crucial for modern OpenGL development and has been used extensively during the development of our prototype.

- **Improved interoperability with OpenCL**: While not used by our prototype, it is worth mentioning, that OpenGL 4.1 now allows to create an OpenGL sync object from an OpenCL sync object for efficient sharing of buffers.

2.3.4 Version 4.2

- **Image load/store** operations enable shaders to randomly read and write textures. This is a major change in the programming patterns for image processing shaders. They are not dependent anymore on the fragment shader's output to be written to an FBO-attached texture, but they can directly modify a texture's content. Most importantly, a single shader invocation is now able to write to multiple texture locations at the same time. This feature is also a step into the direction of compute shaders, which will be available in later versions of OpenGL. Image load and store operations require the shaders and host programs to explicitly synchronize read and write access, i.e. barriers need to be insert appropriately in order to make memory operations visible to other shader invocations and to synchronize with the client execution of OpenGL.

- **Immutable texture storage** allows the host program to specify that a texture's layout and type will not change during its lifetime (the actual content might change, though). For such textures, the OpenGL driver is able to improve their performance, because completeness checks are not necessary anymore.

- **Map-buffer alignment** ensures that a host-mapped pointer to OpenGL buffers is appropriately aligned to the specified multiples of bytes. This is required by certain instruction-sets of CPUs to operate efficiently (e.g. Intel's AVX instruction set).

2.3.5 Version 4.3

- **Compute shaders** provide a new independent processing stage to the OpenGL pipeline. This stage works independently of the rasterization pipeline, i.e. it is the only shader that is executed individually. Compute shaders are written using GLSL and like other shaders they are able to sample textures and randomly read/write texture layers via the previously described image load/store feature of OpenGL 4.2. While this already enables a large amount of image processing scenarios, OpenGL 4.3 also added a new block-buffer type called *storage buffer*, which allows a shader to modify a large amount of data which are not textures.

- **New internal format queries** enable the host program to assert whether the parameters of the format for OpenGL texture upload and download commands are the ones that are optimal for the the currently used OpenGL driver. For example, for some OpenGL internal RGBA formats, the preferred data layout could be BGRA, i.e. if this format is used by the host program to upload data to a texture, no format conversion needs to be done on the side of the driver before uploading. Before using this extension, this information was communicated informally, and often had to be guessed.

- **Framebuffer objects without attachment**: For any OpenGL drawing operation, a framebuffer has to be present. When rendering off screen, the use of a framebuffer object is mandatory. In GL 4.2, it was mandatory for a framebuffer to have a color buffer attached as the target of a drawing operation. However, because the image-store feature of GL 4.2 could result in fragment shaders that only write into images directly and do not return any

actual fragment output, the allocation and writing to an FBO's color attachment would be a waste of bandwidth and texture memory. Therefore, OpenGL 4.3 allows to use an FBO without any attached color buffer, which saves bandwidth and texture memory.

2.3.6 Transferring texture data

OpenGL uses the model of a client-server architecture. The client issues commands which are executed by the server. The server is able to use a different processor, memory space, or even a different machine for executing these commands. In most OpenGL applications, the client-side maps to the software program issuing OpenGL commands, i.e. client-memory is interpreted as CPU-accessible main memory, and the server-side maps to the GPU and GPU-accessible video memory.

In video processing applications, image data need to be transferred repeatedly from main memory to the GPU's video memory, and vice-versa. In the OpenGL model, this requires a constant *streaming* between the client's memory space and the server's memory space. A simplistic approach would be to constantly update the image data of texture objects with the pointer of a newly acquired frame:

1. Acquire new image from video stream.
2. For an existing OpenGL texture, command the OpenGL to update its content using the client-side memory location of the new image using `glTexSubImage2D`.

However, this method is not very efficient, because of the following reasons:

1. The update of the texture stalls the program execution, because a direct data update of a texture is a time-consuming task.
2. OpenGL drivers usually implement a pipeline model between client and server executions, i.e., when the client issues OpenGL commands, they are buffered internally and executed at a later time in larger batches on the GPU. The above version of using `glTexSubImage2D` stalls the internal pipeline, because the server-side upload needs to be finished when the client-side call returns, i.e. no OpenGL commands can be buffered in between.

The performance of the above scenario does not meet the requirements of high-performance video processing scenario.

A better way is to take advantage of *OpenGL buffers*. Such buffers represent a server-side data store that can be allocated, destroyed, read and modified by the client. In order for the client to read and write to this storage, the buffer first needs to *map* its server-storage to a client-side memory location. Once the client is done with its reading or writing of this memory region, it has to *unmap* the buffer again. The actual transfer of client-side memory to server-side memory is transparent to the user of OpenGL. The OpenGL implementation decides on how to map/unmap the buffers and when to actually transfer the data between client and server.

In order to transfer pixel data of images (i.e. video frames), we use *pixel buffer object* (PBOs). These buffers can be used as the input to a texture update. For example, in order to upload a frame we can now do the following:

1. Acquire new image from video stream.
2. Map an existing pixel buffer object (this buffer has at least the size of a single video frame).
3. Copy the video image into the mapped pixel buffer object.
4. Unmap the buffer.
5. Command the OpenGL to update the content of an existing texture using the server-side memory of the previously used PBO.

This approach is much more efficient, because the data source for the texture update is now managed by OpenGL. In the previous version, OpenGL had to make sure that the texture update is finished when `glTexSubImage2D` returns, because the host-side memory might be gone soon. In this version, the update is based on OpenGL-managed memory, therefore the call to update the texture can return immediately and the actual texture update can be done in the background.

For texture download, a similar approach can be used. We command the transfer of a texture's content into a PBO, map the PBO to client memory and copy this memory to our destination image.

In order for this approach to be really efficient, the software program needs to exploit the client-asynchronous texture updates. I.e., if a texture is used in a rendering operation directly after commanding the texture update on the basis of a PBO, the OpenGL execution again has to wait for this update to be finished. Instead, the program could use two PBOs to *interleave* updates with rendering operations. For example, while the current frame's PBO is used for a texture update, the previous frame's texture (which is already updated) is used for rendering.

The above description is only a brief summary of the basic principles of PBOs. In practice, there are a lot of other details to consider in order to reach optimal performance. These considerations also vary across different OpenGL implementations, which makes efficient buffer transfers a very difficult part of the OpenGL API. A good overview on this topic is given by the official OpenGL wiki article on pixel transfers [5]. A thorough review of OpenGL buffer transfer techniques is given by Hrabacek and Massermann [11].

2.3.7 Vendor-specific optimizations

2.3.7.1 NVIDIA asynchronous textures transfers

By default, PBO-based pixel transfer are only asynchronous to client-execution of their OpenGL commands. When the transfer to and from video memory is actually executed, the GPU (server-side) is not able to execute any other commands. NVIDIA graphics boards are equipped with

[5] http://www.opengl.org/wiki/Pixel_Transfer

additional DMA controllers, which can perform the GPU-side transfer asynchronously to other rendering operations. This allows for an overlap of transfer and rendering. The NVIDIA professional line of products is equipped with an additional DMA unit, hence upload, render and download can be overlapped. This thesis implements this approach and summarize the actual performance benefits in Chapter 6.

The asynchronous transfers with NVIDIA's DMA controllers is not enabled by a conventional OpenGL approach. For the driver to enable the overlapped transfers, the OpenGL application needs to use multiple OpenGL contexts which are executed on multiple threads. OpenGL commands between multiple contexts need to be synchronized explicitly. This is done by using the previously mentioned OpenGL fence sync objects.

Asynchronous texture transfers, as enabled by NVIDIA hardware, are described by Venkataraman [35].

2.3.7.2 AMD pinned memory

When data is transferred from main memory to video memory over the PCIe bus, it needs to reside in a so-called *pinned* region of memory. The term *pinned* refers to the property of memory residing in actual physical memory space as opposed to *pageable* memory regions which are allowed to be swapped in and out of the operating systems memory page-file on disk. Efficient PCIe transfers need to operate on pinned memory.

PBOs are either in a *mapped* or *unmapped* state. The client is only able to use a PBO when it is mapped, the server is only able to use PBOs in an unmapped state. A client-mapping is likely to point to *pinned* memory, but there is no guarantee. If it is not, the OpenGL driver needs to internally copy the client-mapped memory to pinned memory before transfering the data over the PCIe bus. Between multiple mappings of a buffer, the OpenGL implementation can also decide to use different client-side memory regions.

AMD provides an OpenGL extension which alleviates this problem by allowing the client to declare a memory region as *pinned*, and by allowing the client to use this memory as the backing storage for an OpenGL PBO. This has the following consequences:

- Mapping/Unmapping of a PBO using pinned memory as backing storage is not necessary anymore, because the client-side pointer is persistent.
- The driver does not need to internally copy data to pinned memory regions, because the used memory location is already pinned.
- The client program needs to manually synchronize the OpenGL driver when reading/writing to the pinned-memory backing of a pixel buffer object.

2.3.7.3 OpenGL 4.4 buffer storage

The recent release of OpenGL 4.4 introduced new approaches to dealing with buffer objects. First, a persistent client-side mapping, similar to AMD's pinned memory extension, is now available to any application using OpenGL 4.4. Second, it is now possible to explicitly specify whether a buffer's storage should favor client-side memory, or server-side memory. This, in

combination with other flags during creation, enables the use of memory regions which are accessible to both CPU and GPU. AMD's APU-line of processors, for example, provide for such shared memory, which eliminates the need for client-to-server transfers over the PCIe bus.

2.4 Related software

Several commercial broadcasting products provide integrated playout and broadcasting graphics using commodity hardware. Examples are ToolsOnAir's *just:live*[6], Softron's *OnTheAir* products[7] or Playbox's playout products[8]. CasparCG[9] is a popular open-source playout solution, providing graphics as well.

There are several open-source media processing frameworks. One of the most-used media libraries is *libav*[10], which allows to build complex media graphs with generic processing facilities. While most processing is based on CPU-based algorithms, OpenCL support has recently been added as well. A recent library, called *upipe*[11], provides a pipelined software model to process media data asynchronously on a computer. This library builds partly on libav for some transcoding and processing modules.

[6] http://www.toolsonair.com
[7] http://www.softron.tv
[8] https://playbox.tv
[9] http://www.casparcg.com
[10] http://libav.org
[11] http://upipe.org

CHAPTER 3

Design

This chapter describes the design of the *FrameBender* video processing framework. The goal of the framework is to implement a highly parallelized video processing pipeline. Developing concurrent programs that are both efficient and safe is a difficult task. In Chapter 3.1, we describe a software model that helps to address this problem in the context of a video processing pipeline. This model is based on the *pipeline pattern*, which enables our prototype to split the overall task of video processing into several stages, which can be executed concurrently. In Chapter 3.2, we then describe a specific video-processing algorithm that we use as our target scenario. It is implemented using the previously described software model. We finally describe the high-level application *FrameBender* in Chapter 3.3, which allows a user to execute the pipeline on actual video sequences and to collect data of its execution.

3.1 Pipeline model

In order to efficiently execute video processing, different hardware units in our system need to perform tasks of this algorithm in parallel. This can be done by letting the CPU, DMA controllers and GPU execute these tasks concurrently (see Figure 3.1). The *pipeline pattern* is a commonly used software model that allows parallelizing subsequent tasks of a larger processing algorithm like ours. Even though existing hardware architectures only allow parallelizing certain stages (see Chapter 4.3), it is beneficial to use a software model that in theory enables the parallel execution of any processing step. In the following we are going to describe the pipeline model that has been used for the implementation of our framework.

3.1.1 Stage

In our framework, a *stage* represents a single element of a generalized pipeline (see Figure 3.2). A stage executes an individual *task*, and several stages can be executed in parallel. The flow of video frames between stages is handled by FIFO queues. The pipeline processes data in order, which is necessary to keep the ordering of the frames for the processed video. Depending

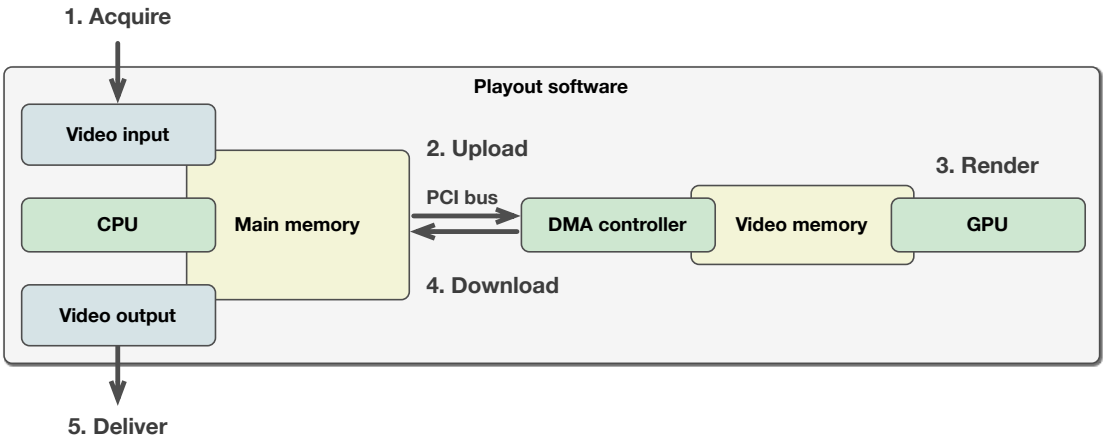


Figure 3.1: This diagram shows the components of a typical playout process using commodity hardware. The playout software first acquires a video frame from the video input module, which is stored in main memory and processed by the CPU. The DMA controller then uploads the video frame to the GPU's video memory via the PCI bus. The GPU then renders broadcast graphics on top of the original image. After rendering, the DMA controller downloads the frame from video memory to main memory. Finally, the CPU delivers the video frame to video output component. In order to optimize this process, the hardware units involved need to execute concurrently.

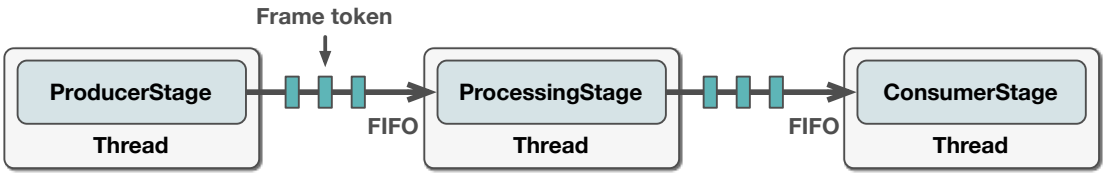


Figure 3.2: High-level concept of pipeline stages.

on whether input data, output data or both are required by a stage, we distinguish between *ProducerStages*, *ConsumerStages* and *ProcessingStages*.

In order to keep the stage executions thread-safe, the execution of a stage's task should not have any side-effects other than an input token to be consumed from the input queue, and an output token to be added to the output queue. If this can be ensured, then thread-safety of the stage executions can be guaranteed by making the FIFO queues themselves thread-safe. Otherwise, explicit locking would have to be performed by the task implementation. This is not only prone to errors, but can also result in worse performance when one execution has to wait for a lock to be released by another execution. Chapter 4.1.1 describes a lockless thread-safe FIFO implementation as it is used in the *FrameBender* framework.

The basic design shown in Figure 3.2 already allows us to parallelize tasks for a data flow into a single direction. However, for the tokens of the queues, we would like to cycle over a fixed

set of elements. Tokens of the same type should be sent back and forth between stage executions. This allows the stages to re-use data tokens of limited resources (e.g., regions of video memory). We model this feature by adding another FIFO which transports tokens from a stage back to the previous stage (see Figure 3.3). We now distinguish between data flow that goes *downstream* (from left to right in Fig. 3.3) and data flow that goes *upstream* (from right to left in Fig. 3.3). Tokens can now be passed in both directions and re-used between stage executions. While this semantic of a pool of re-usable tokens could be modeled by the tokens themselves, e.g., using a hypothetical *BufferToken* class, there are specific reasons why we chose to model this explicitly by adding *upstream* data flow semantics:

- The upstream model allows us to be more generic about the data type that is transferred via the queue. A hypothetical *BufferToken* class would already assume certain semantics (e.g., allocation/deallocation, locking/unlocking, etc). Our generic two-way communication pattern allows us to use any type of token. For example, if a stage fails to execute a task, it is able to attach an error state to the upstream token which is sent back to its previous stage. This would not really fit into the model of a strictly buffer-related token type.

- Tokens that flow upstream are able to have different state than those that flow downstream. This will be necessary, for example, when performing a complex two-way synchronization as it is shown in Figure 4.7 of Chapter 4.3.

- For some stages of our implementation, it will be necessary not only to cycle tokens between two stages, but over a range of stages where each execution changes the downstream and upstream state (see Figure 4.3 of Chapter 4.2). This requires the functionality of explicitly passing these tokens to the previous stage.

- A separate queue-based backwards communication path allows us to more strictly adhere to the rule of not performing any other state changes than addition and removal of queue tokens during stage execution. Downstream tokens can be thought of as the *input parameter* to the execution of a stage's task. Upstream tokens can be considered as the *return value* to this execution, which is being processed in a lazy way by the originating stage, i.e. the *caller*.

3.1.2 Simplified model of the video processing pipeline

To understand the benefits of our pipeline stage design, let us look at the execution of a simplified version of this framework's video processing application (see Figure 3.4). In this simple case, we have one producer stage (*In*) that acquires a video frame (e.g. from a video card), one processing stage (*Render*) that performs rendering from one source frame into one destination frame, and finally we have a consumer stage (*Out*) that outputs the result of the rendering operation (e.g. back to a video card). Data tokens represent video frames. Let us assume that memory resources are limited and that allocation and deallocation of such frame tokens should be avoided during execution of the rendering operations. We pre-allocate a limited set of tokens

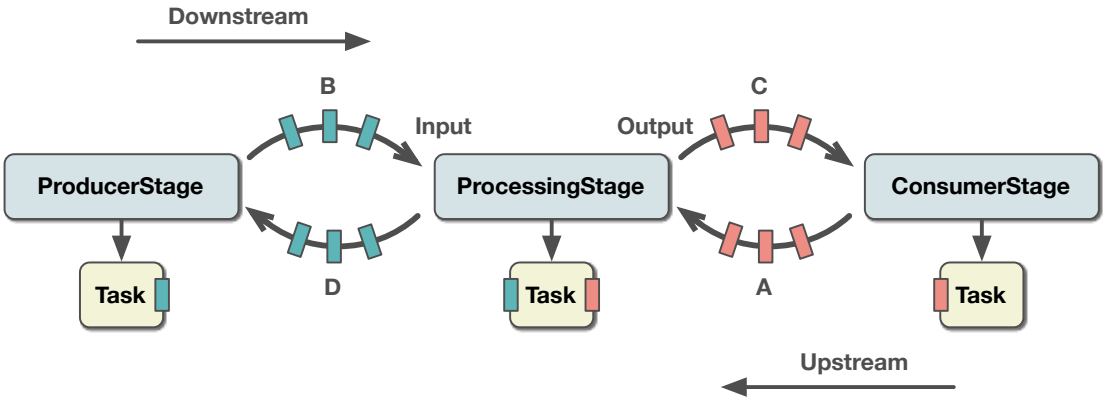


Figure 3.3: Using two bounded queues per stream direction for two-way communication. In order for the *ProcessingStage* to execute its task, an output token has to be fetched from the *upstream output queue* (A) and an input token has to be fetched from its *downstream input queue* (B). After execution, the output token holds the result and is added to the *downstream output queue* (C). The input token is not needed anymore and is added back to the *upstream input queue* (D). If processing had failed, this token could carry an error state, which is eventually read by the *ProducerStage* when it is fetched from the *ProducerStage*'s *upstream output queue* (D).

before the execution of the algorithm. In our example shown in Figure 3.4, we pre-allocate data tokens `1`, `2` to be used by stage *In* as output and by stage *Render* as input, and we pre-allocate data tokens `A`, `B` to be used by stage *Render* as output and by stage *Out* as input tokens.

In this example, each stage runs its own thread and all three stages can execute in parallel. Figure 3.4 shows the data flow of rendering three consecutive frames. Before start, at $t = 0$, no stage is executing, both upstream queues contain two frame tokens. After starting the pipeline at $t = 1$, stage *In* acquires the first video frame and writes it into token `1`, the other stages are still waiting for a frame to arrive. Once the input stage is done, it puts frame token `1`, which now contains the first actual video frame, onto the input stages' output downstream queue. This is shown at $t = 1.5$ in Figure 3.4. However, stage *Render* executes as soon as an input token is available to it, so the state shown at $t = 1.5$ only exists for a very short time. At $t = 2$, the render stage reads token `1` from its input downstream queue and reads token `A` from its upstream output queue. This stage is now rendering into frame token `A` using token `1` as the input frame. At the same time, the next frame is acquired by stage *In* and written into token `2`. Once the rendering operation and the second frame input operation are finished at $t = 2.5$, *In* and *Render* put their output tokens onto their respective downstream output queues. At this time, the render stage also puts its previous input token `1` onto its upstream input queue, so the next execution of stage *In* can use it as its again. In the next step, at $t = 3$, the first finished frame is now available to stage *Out*: Token *A* is fetched from stage *Out*'s downstream input queue and will be written to the output. At the same time, the second frame, now held by token `2` is rendered by stage *Render* into frame token `B`, and the third frame is acquired by stage *In*, now again written to token `1`. At this time, the pipeline is fully-loaded, i.e. stage *Out* outputs frame n at the same

time as stage *Render* renders frame $n-1$ while stage *In* acquires frame $n-2$ from input. In the example given in Figure 3.4, the third frame is our last frame, so from $t = 4$ onwards, the input stage has nothing to do anymore, and finally, at $t = 6$, all three frames were processed and written to the output.

Figure 3.5 shows the runtime chart of this example's parallelized execution compared with serial execution of the same algorithm. It is obvious that the parallel execution of the three stages results in a much shorter time of execution. The example that was given here only shows a very simplified model of the implemented pipeline, but it demonstrates the overall design of the pipeline stages and how the data-flow between stages is established in principle. Chapter 4.1 provides a complete graph of the pipeline stages that are implemented by the *FrameBender* video processing application.

3.1.3 Timing properties of the video processing pipeline

Figure 3.6 shows a generic pipeline with N stages executed in parallel. Let us assume that stage s_1 represent frame input, and that the final stage s_n represents frame output, everything in-between is a generic video-processing task.

Let t_i be the time required to execute the stage s_i, and let M be the total number of frames to process. Executing the stages serially would result in a total execution time T_s of

$$T_s = M * \sum_{i=1}^{N} t_i \tag{3.1}$$

Let us assume that the number of frames M to process is much larger than the number of stages N of the pipeline, which is usually the case. Then the *ramp-up* and *ramp-down* phases for parallel execution as shown in Figure 3.6 represent only a negligible amount of time. Therefore, if all stages s_i are executed in parallel, we can define the amortized total execution time T_p of processing M frames as

$$T_p \approx M * \max_{1 \leq i \leq N} t_i \quad | \quad N \ll M \tag{3.2}$$

It is important to note that because of the data dependencies between the stages, the longest executing stage will dominate the overall execution time. For example, if s_1 has an execution time of t_1 then the overall execution will be at least $M * t_1$, even if other stages take a shorter amount of time to execute.

If the execution time for each stage is the same, i.e.

$$t_i = d \quad | \quad i = 1..N \tag{3.3}$$

then the serialized execution time T_s of the pipeline is about N times larger than the parallelized execution time T_p:

$$T_p \approx \frac{T_s}{N} \tag{3.4}$$

In other words, the theoretical best-case performance improvement of parallelizing the execution of pipeline stages is N.

3.1.4 Live playback constraints

Until now, we have assumed that each stage s_i has a constant execution time of t_i. In reality, the execution time is likely to vary between executions. We therefore define the following function:

$$S(i, j) \tag{3.5}$$

represents the execution time of stage i when processing the jth frame. Let δ be the interval in seconds of subsequent frames of a video system, then δ is defined as

$$\delta = v_r^{-1} \tag{3.6}$$

where v_r is the video frame rate in Hz (e.g. 25 Hz for HDTV in Europe). The *live playback criterium* of our video processing, i.e. the ability of the output video rate of the pipeline to be at least as high as the input video rate, can now be defined as follows:

$$\max_{i=1..N,\, j=1..M} S(i, j) < \delta \tag{3.7}$$

No stage execution of any of the processed frames can take longer than the duration δ of a video frame. It is important to point out that the overall processing time t_j of a single frame j, which is

$$t_j = \sum_{i=1}^{N} S(i, j) \tag{3.8}$$

can take longer than the playback video duration δ, without breaking the *playback criterium*, because stage execution can be hidden behind stage executions of previous frames. However, longer frame processing times will contribute to the *latency* of a video system, which is defined as the difference of time between frame input and frame output (see Figure 3.6).

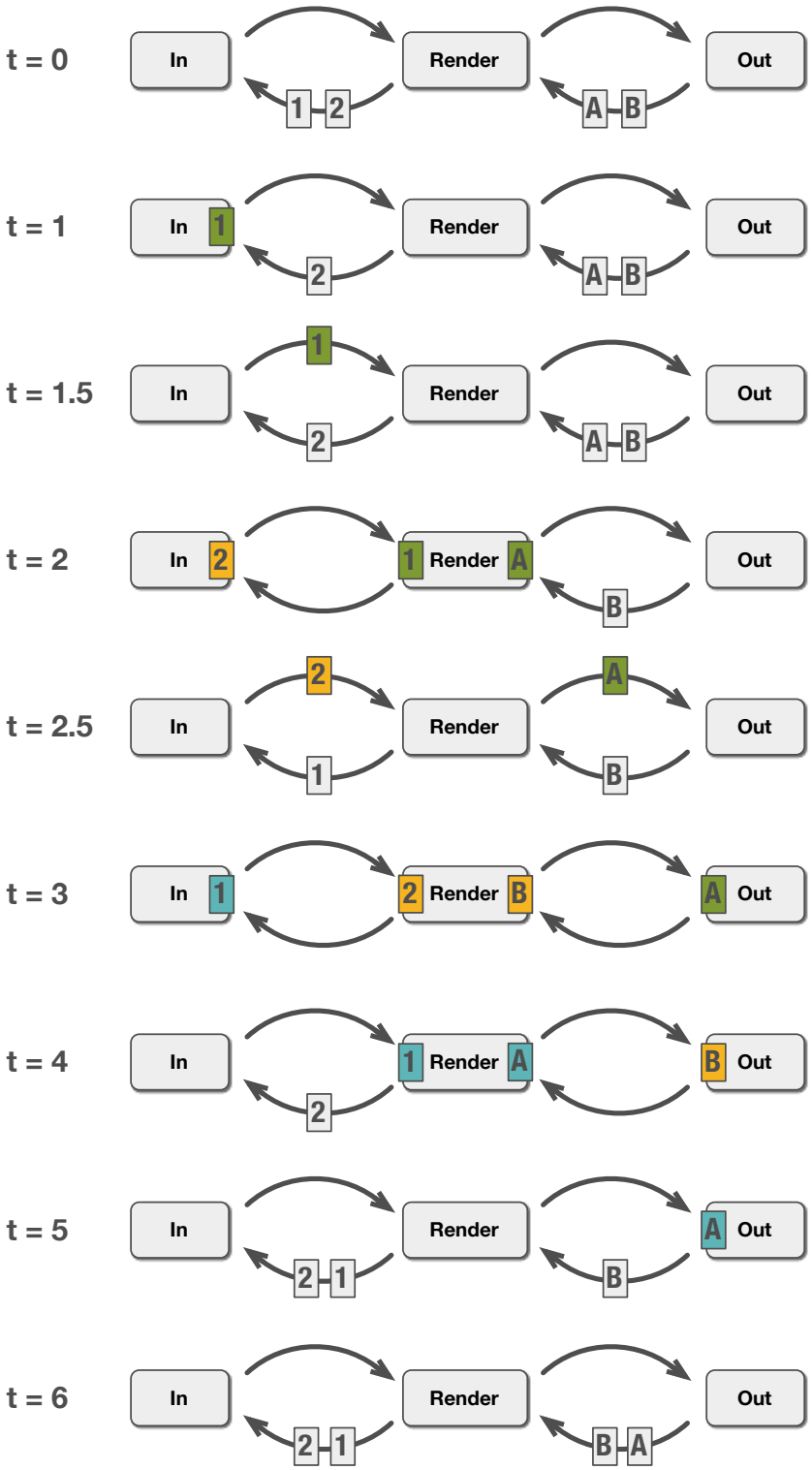


Figure 3.4: Example of data flow. The originating frames have different colors in the diagram.

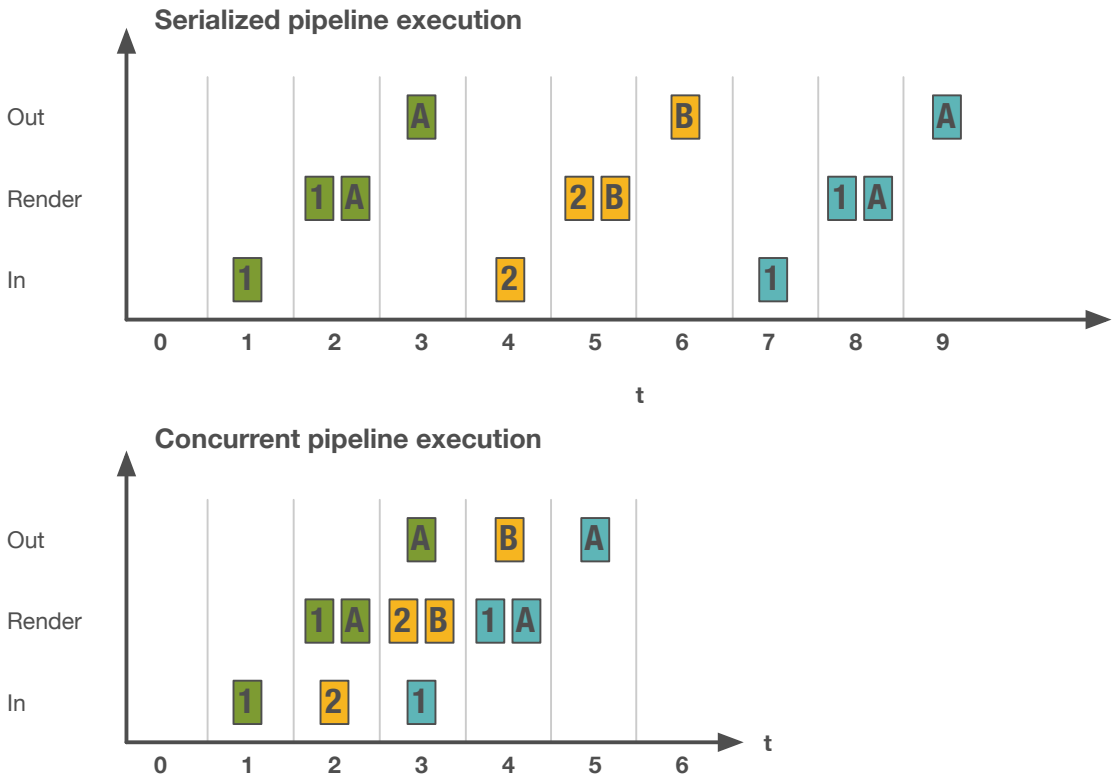


Figure 3.5: Runtime charts of the example run shown in Figure 3.4. The concurrent execution times show the best case scenario where all stages fully overlap each other.

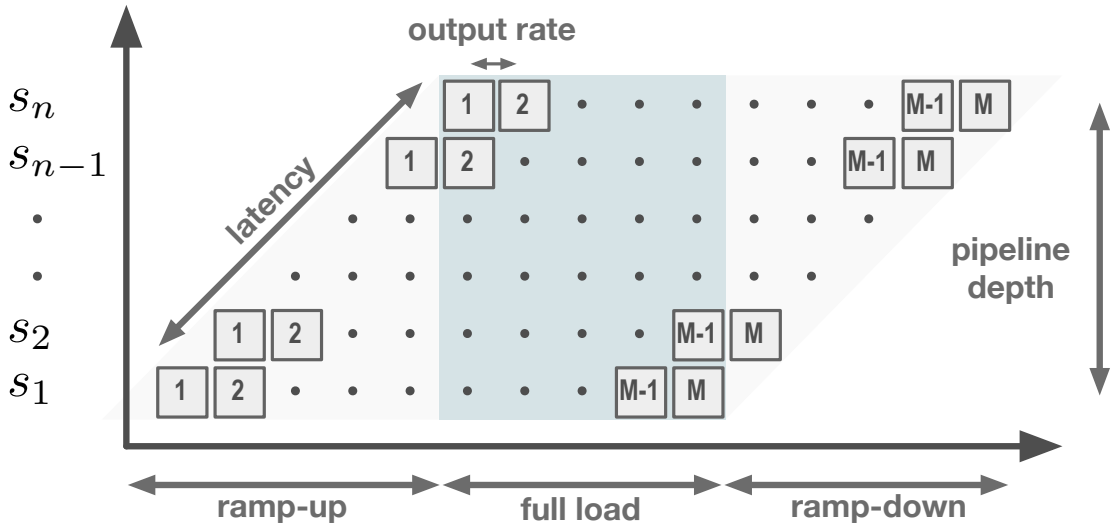


Figure 3.6: Timing properties of a generic processing pipeline with N stages processing M frames. The colored area of the pipeline diagram shows the executions which operate under a fully-loaded pipeline.

3.2 Targeted scenario

This framework is aimed at rendering sRGB graphics on top of a 10-bit Y'CbCr-encoded video sequence. We first acquire a frame from the Y'CbCr video sequence and then convert the frame to an RGB-format which is suitable for rendering. After overlaying graphics, we convert the frame back to Y'CbCr and write the to the output sequence.

The video format *V210* has been chosen as the 10-bit Y'CbCr format. It is used both as the input and output pixel format (see Chapter 5). This format is commonly used as the native pixel format by professional video capture cards. In order to render on top of the V210 frames, the Y'CbCr encoding first has to be converted to a linearly coded RGB pixel format. Typically, this conversion is done on the CPU before it is transferred to the GPU. The *FrameBender* framework directly transfers V210 to video memory, and performs the conversion from 10-bit Y'CbCr to RGB on the GPU. This has the following benefits:

- The GPU is considerably faster than the CPU in performing image processing tasks that can be executed in parallel. Chapter 6 provides a comparison with a state-of-the-art CPU-based conversion.
- V210 is a 4:2:2 chroma-subsampled format. It requires less data to represent a frame than its RGB representation and therefore reduces the amount of data that needs to be transferred over the PCIe bus.
- Converting the pixel format on the GPU allows the fusion of pixel format conversion with other necessary preprocessing steps, such as gamma correction or color-space conversions. While this could be done on the CPU as well, a faithful representation of gamma-corrected 10-bit video requires a high-precision intermediate representation which eventually needs to be transferred to the GPU for rendering. For example, a 1080p frame with 32-bit floating-point RGB components requires 24.9 MB of data, while the original V210 representation amounts to only 5.5 MB. For HD resolutions this is an unfortunate waste of bandwidth, but for UHD-1 where one 32-bit floating-point frame amounts to approximately 100 MB, this situation quickly becomes unfeasible.

Rendering in a linearly coded color space improves the quality of the rendered results (see Chapter 2.1.1). While this is common practice in the field of computer graphics, where lighting equations of the rendering algorithm assume a linear space, this is not always the case for video. Today it is considered as the recommended practice when rendering high-quality video[1].

Figure 3.7 shows a diagram of our framework's targeted image processing algorithm. The processing steps are:

A Upload an image of the input sequence from main memory to the GPU's video memory. The image format that is used for transfer should be as close as possible to the original uncompressed format in order to avoid too much processing on the CPU side. Latency

[1]This statement was supported by Charles Poynton during a personal conversation. Charles Poynton is the author of *Digital Video and HD, Morgan Kaufmann, 2012*, [26].

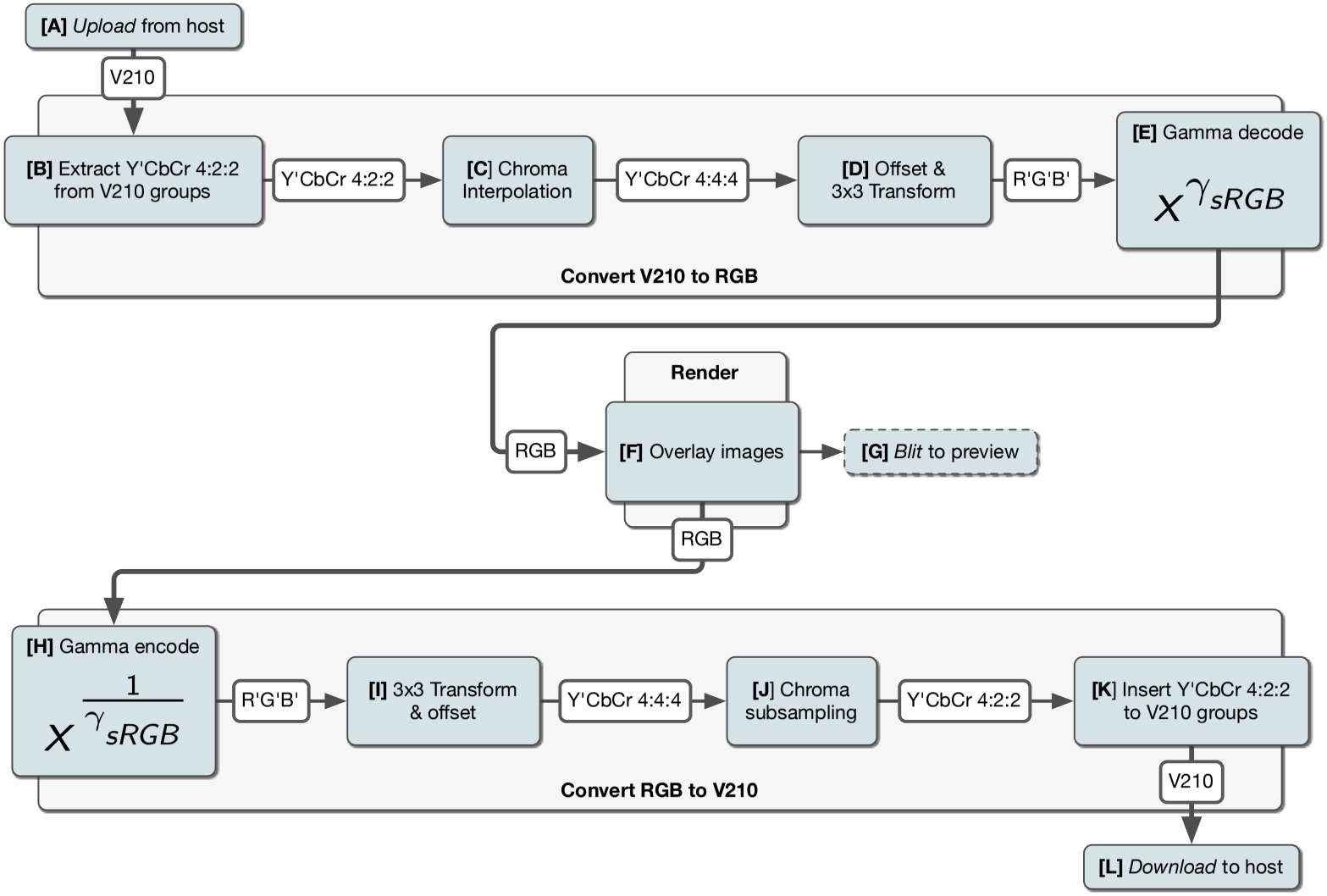


Figure 3.7: An overview of the image processing algorithm. All processing is done by the GPU.

hiding should be implemented to improve throughput and to avoid stalls for the following stages when waiting for input data.

B Extract Y'CbCr values from the V210-encoded frames (see Chapter 5). These values are still encoded in a 4:2:2 subsampled format.

C Reconstruct 4:4:4 Y'CbCr sampling from the subsampled chroma values. During reconstruction, high-quality filters should be used (see Chapter 2.1.5).

D Until this step, we are still processing in the domain of interface values, i.e., for 10-bit Y' components, reference black is at integer value 64, and reference white is at integer value 940. In order to convert to R'G'B' in unit range, we first offset the interface values and then multiply with a 3x3 matrix (see Equation 2.13). In Chapter 2.1.6, we explained how this matrix is derived for decoding BT.709-conforming video.

E Apply the sRGB transfer function in order to reconstruct linear RGB values from perceptually coded R'G'B' values. We apply sRGB transfer to both the video and the overlay images in order to convert to a common linearized space for mixing BT.709/BT.1886 video with sRGB graphics. Chapter 2.1.7 describes more details about mixing sRGB material with BT.709-conforming video.

F Overlay several sRGB images on top of the input video stream. The rendering algorithm should be swappable with any other video rendering algorithm (see Chapter 4.2.1.6 and 4.5).

G Display the rendered frame in real time for preview and monitoring purposes.

H Apply the inverse of step *E*, i.e. convert back into the original perceptual coding space.

I Convert R'G'B' to Y'CbCr using the inverse of step *D* (see Equation 2.12).

J Perform 4:2:2 chroma subsampling. Use high-quality filters for when applying the decimation chroma filter.

K Insert the Y'CbCr 10-bit values into 32-bit V210 words.

L Download the resulting V210 frame from video memory back to main memory.

3.3 FrameBender application

The C++ library *libFrameBender* is the center-piece of the framework. It contains the implementation of the video processing pipeline, shaders for format conversions and rendering, and general supporting data structures. The *FrameBender application* is built on top of the library. It configures the framework's input and output settings and its optimization parameters. The application can be used for collecting performance traces and for writing out the rendered output of the framework (see Chapter 4.6 and 6.1.3). The FrameBender application also provides a player window that allows the user to monitor the rendered video in real time.

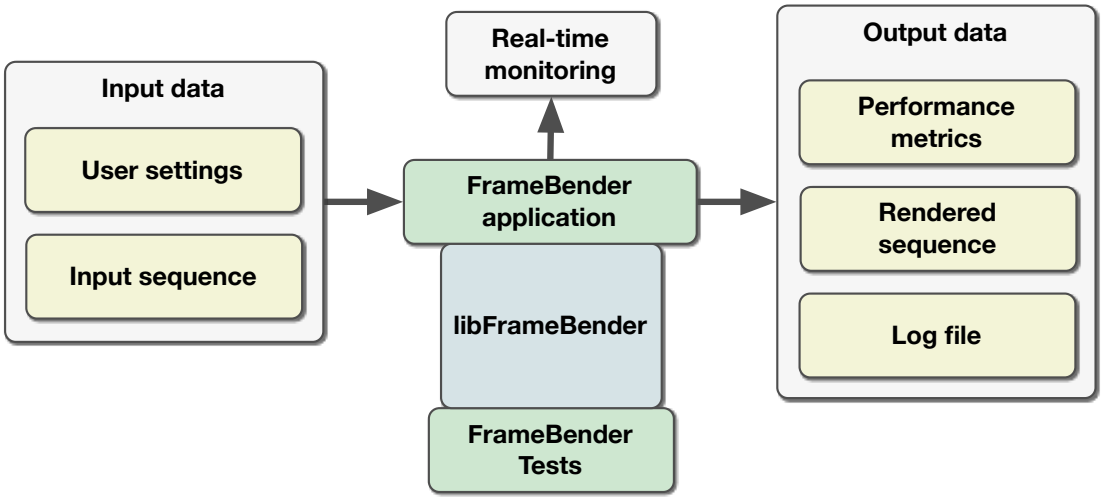


Figure 3.8: Overview of the *FrameBender* video processing framework. *libFrameBender* contains the video pipeline and implements the format conversion algorithms. The *FrameBender* application uses the library to render an input sequence of raw video frames. The application outputs the rendered sequence, performance traces of the video pipeline execution and a log file.

FrameBenderTests is a collection of unit tests. The tests are written against the interfaces of the *libFrameBender* library and therefore include the library in the same way the *FrameBender* application does. Besides testing the behavior of data structures and classes of the framework, the testing framework also includes *golden samples* of expected rendering results under a well-defined set of parameters. During the development of optimizations of the implementation, these golden samples are used to make sure that the rendered output is still correct, i.e., that using an added optimization does not introduce errors.

In summary, the following features are provided by the *FrameBender* framework:

- Studio-quality renderer for TV graphics
  - *V210* 10-bit Y'CbCr as raw input and output formats
  - Linear color coding (*gamma correct* rendering)
  - High-quality N-tap filters for chroma reconstruction and subsampling
  - Real-time preview of rendered output
- Optimized frame transfers and Y'CbCr conversion
  - High throughput rates between CPU and GPU by overlapping GPU upload, render and download
  - Multithreaded frame handling
  - Vendor-specific optimizations for AMD and NVIDIA

  - Multiple variations of Y'CbCr to RGB conversion shaders
  - Using OpenGL 4.2 and 4.3 features
- Integrated performance profiler
  - CPU- and GPU-based sampling
  - Writes traces into a well-defined file format
  - Visualization of traces for analysis
- File-based configuration
  - Allows to use different test sequences
  - Optimizations can be enabled/disabled selectively
  - Large set of debugging parameters
  - Can be used to re-run previous configurations for benchmarking purposes
- Unit-tested framework
  - Across shader variations, rendering results are kept consistent
  - Avoids regressions when optimizing the implementation

CHAPTER 4

Video Pipeline Implementation

The design and intent of our framework was described in Chapter 3. This chapter explains the actual implementation of the previously introduced concepts. We begin by demonstrating the C++ implementation of the pipeline infrastructure in Chapter 4.1. We then show how to apply this model on top of the OpenGL API to build the video processing pipeline (Chapter 4.2). The scheduling of the pipeline for hardware-concurrent execution is described in Chapter 4.3. The remaining chapters contain other details about OpenGL usage, profiling and debugging features of the framework.

4.1 C++ pipeline infrastructure

The general model and design of our pipeline model was described in Chapter 3.1. We use the C++ language to implement this model in several classes that provide a generic functionality of a pipeline model.

4.1.1 CircularFifo implementation

The most important data structure for the pipeline is the FIFO queue that passes tokens between stages (see Figure 3.3). As we intend to run the stages on different threads, access to those queues has to be thread-safe. Williams describes implementations of thread-safe queues using C++11 in his book [38]. Another simple implementation is described by Hedström [20], which was chosen as the basis for our framework's queue implementation. This type of queue is a lock-free, bounded circular queue providing thread-safe access to a single writer and a single reader. Since connections between stages are shared by exactly one stage that writes tokens, and another that reads them, this simple model of a thread-safe single-producer single-consumer queue is sufficient for our use case.

Listing 4.1 shows the C++ template interface to the `CircularFifo` of our framework. The queue is parameterized with the template type `Element`. During construction of an empty queue, an array to hold a number of `size` elements is allocated and stored in the `_array` data

```
1  template<typename Element>
2  class CircularFifo {
3  public:
4
5      CircularFifo(size_t size);
6      virtual ~CircularFifo();
7
8      bool push(Element&& item);
9      bool push(const Element& item);
10
11     bool pop(Element& item);
12
13     bool was_empty() const;
14     bool was_full() const;
15     size_t had_num_elements() const;
16
17     bool is_lock_free() const;
18
19     size_t size() const;
20
21 private:
22
23     template <typename InsertElement>
24     bool push_element(InsertElement&& item);
25
26     size_t increment(size_t idx) const;
27
28     const size_t _size;
29     const size_t _capacity;
30
31     std::atomic<size_t> _tail;
32     std::unique_ptr<Element[]> _array;
33     std::atomic<size_t> _head;
34 };
```

Listing 4.1: FIFO implementation as implemented in *libFrameBender*.

member. This queue is able to hold at most a number of `CircularFifo::_size` elements, hence it is *bounded*. The data members `_tail` and `_head` store indices to the previously allocated array and represent the current write and read position of the queue. When an element is written or read, these indices are incremented and potentially wrapped around the size of the queue. In order to make read/write access to the queue thread-safe (for a single reader and writer), it suffices to increment these indices *atomically*. Using C++11, this is easily done by using `std::atomic<size_t>` as the data members for the indices. More details on the new multiprocessor-aware memory model of C++11 is given by Williams [38].

The data type that is passed as the template parameter `Element` has to provide at least a default constructor. If the `Element` also provides a *move constructor* and *move assignment operator*, unnecessary copying can be avoided. *Move semantics* is a new feature introduced in C++11. It allows to transfer an object's ownership of all its data to another instance, the object in

```
enum class WaitingPolicy {
    SPIN,
    BLOCK
};

template <typename Element, WaitingPolicy policy>
class WaitingCircularFifo {

public:

    static const WaitingPolicy kWaitingPolicy = policy;

    // Same interface as CircularFifo<Element>

private:

    CircularFifo<Element> fifo_;
    std::mutex mutex_;
    std::condition_variable condition_;
};
```

Listing 4.2: Wrapper around CircularFifo for providing a waiting read operation.

its essence is *moved* somewhere else. This is not only useful to more strictly defined ownership of data, but it also improves performance by avoiding unnecessary copies. Therefore, if the data type `Element` supports it, reading and writing into the queue are reduced to *move* operations. A thorough description of *move semantics* is given by Stroustrup [32].

Another advantage of a circular buffer is that it allocates and deallocates its memory outside of read and write operations, i.e. the data member `_array` is allocated in the constructor. This is not only advantageous in terms of performance, but it also removes the responsibility of the read and write operations to handle errors during allocation.

4.1.2 WaitingCircularFifo wrapper

The implementation of `CircularFifo` in Listing 4.1 returns a boolean value for the `pop` method. It signals the caller whether an element could be fetched (true) or the queue was empty at the time of calling (false). In some situations, however, it would be more convenient for the `pop` method to wait until an element can be read, instead of just returning an unsuccessful state. In our framework, this is implemented using the wrapper `WaitingCircularFifo` around `CircularFifo` (Listing 4.2). This wrapper has the same interface as the `CircularFifo` class and adds an additional compile-time constant of type `WaitingPolicy` to the interface. There are two possible values for this policy:

- **`SPIN`**: Busy-wait in the calling thread until a value is available.
- **`BLOCK`**: A `std::condition_variable` is used to suspend the calling thread when no element is available. Once an item has been added, the thread is notified via the condition variable and is able to fetch the newly added element.

The choice between the two approaches should really be made based on performance measurements. In our case, the interval between queue manipulation is comparatively large. The overhead of suspending and resuming a thread between stage executions is therefore acceptable. Because suspending a thread allows other threads to take over CPU resources, the overall performance of system is improved. We therefore use the `BLOCK` strategy by default.

4.1.3 The Stage C++ template class

The pipeline of our frameworks consists of several *stages*. The design of a pipeline stage is shown in Chapter 3.1.1. Its implementation is described in the following.

A stage consists of a *task* that reads input tokens from the input queue and writes output tokens to the output queue. The `Stage` class (Listing 4.3) provides a convenient C++ class that is able to connect with a previous stage, manages the dispatch of tokens from its queues and passes the tokens to its task for execution. The actual execution of a stage's task is triggered from the outside using the `execute` method of the `Stage` interface. It allows the user of the class to decide how stage executions should be triggered. This could be done serially from a single thread or concurrently from multiple threads. This choice does not influence the output of the pipeline executions, i.e. the scheduling of the pipeline's executions is *orthogonal* to the definition of its algorithm, and therefore not part of the `Stage` classes' interface.

The `Stage` class is parameterized by two template type arguments: `InputElement` and `OutputElement`. These arguments define the input and output data types of a stage's *task*. While stages always process video frames in our pipeline, the data representation of a frame might vary between different stages. For example, one stage type might need to store the raw data of a frame with a token. For another stage, a frame might be represented only by an identifier to some OpenGL-specific resource, and only this identifier needs to be passed along the queue. It would be a waste of resources or unnecessarily complicated to map both of these frame representations to a single abstract data type for a queue's token. Since the types of the token are usually known during compile-time, it is better to explicitly pass the type parameters to the `Stage` template class. This has the following advantages:

- No runtime polymorphism is needed when using varying data types between different types of stages.
- Incompatible input and output types are reported by the compiler as an error.
- The definitions of tasks are easier to read and their implementation is more efficient because data types can be exactly tailored to the needs of a task.

A stage's task is defined as a `std::function` instance. This is a callable object that takes references to the current input and output token as function arguments (see line 31 in Listing 4.3). The task reads the data from the input token, processes it and writes the result back into the output token reference. The `std::function` wrapper is a new C++11 feature that wraps a callable object with a specified signature. For each execution, a task also has to return a status value of type `StageCommand` (see line 7 in Listing 4.3). This status value is processed by the generic stage class that originally called the task functor. Based on the value of the command, the

```
enum class PipelineStatus {
    INITIALIZING,
    READY_TO_EXECUTE,
    HAS_BEEN_STOPPED
};

enum class StageCommand {
    NO_CHANGE,
    STOP_EXECUTION
};

template <typename OutputElement>
struct OutputToken {
    typedef OutputElement TokenElementType;
    TokenElementType element;
    StageCommand command;
};

template <typename InputElement, typename OutputElement>
class Stage {

public:

    typedef OutputElement OutputType;
    typedef InputElement InputType;
    typedef OutputToken<InputElement> InputTokenType;
    typedef OutputToken<OutputElement> OutputTokenType;
    typedef WaitingCircularFifo<OutputTokenType, kDefaultWaitingPolicy>
        OutputFifoType;
    typedef WaitingCircularFifo<InputTokenType, kDefaultWaitingPolicy>
        InputFifoType;

    typedef std::function<StageCommand(InputType&, OutputType&)> Task;

    template <typename InputInputType>
    Stage(Task task,
          std::vector<OutputType> output_queue_initialization,
          const Stage<InputInputType, InputType>& input_stage);

    void execute();

    PipelineStatus status() const;

private:

    std::weak_ptr<InputFifoType> input_downstream_;
    std::weak_ptr<InputFifoType> input_upstream_;

    std::shared_ptr<OutputFifoType> output_downstream_;
    std::shared_ptr<OutputFifoType> output_upstream_;

    Task task_;

    std::atomic<PipelineStatus> status_;
};
```

Listing 4.3: Template interface to the pipeline stage class (some details are omitted).

stage performs additional managing routines. For example, a task is able to stop the execution of the pipeline by returning the constant `StageCommand::STOP_EXECUTION`. When its stage encounters this value, it will notify all other stages that the overall pipeline execution should be stopped.

The template arguments to the `Stage` interface only define the data types of the stage's task arguments. State that is common to all stages (e.g., flow control) also needs to be communicated over the queues' tokens. To allow this, the `OutputToken` structure (see line 13 in Listing 4.3) holds, in addition to the task-specific data types, also data that is common to every stage, like the `command` field. For example, if one stage's task returns `StageCommand::STOP_EXECUTION`, this command will be passed along the data stream to the following stage, which then knows at which point not to expect any more data from its preceding stage.

An instance of a `Stage` always owns its output queues (upstream and downstream). Input queues are only references to the input stage's output queues. As you can see in Listing 4.3 on line 44 and following, this is implemented using `std::shared_ptr` for owning queues, and `std::weak_ptr` for referenced queues. These classes are new to the C++11 STL library. They safely manage the lifetime of the queues while still being efficient enough.

Input and output queues both use the `WaitingCircularFifo` wrapper (see Listing 4.2), i.e. calls to `Stage::execute` block until data dependencies are met and the stage's task function has been executed.

4.1.4 Example of the execution of a simple video pipeline

Listing 4.4 shows a simplified code example of how to use three stage instances in order to create simple video display pipeline:

- In line 13 we create the stage `acquire_stage`. This is a *ProducerStage*, and we use the constant `NO_INPUT` as the input element type to specify that this stage has no input. In this example, we don't care about the actual implementation of this stage. All we need know is that this stage outputs a raw video frame of type `HostFrame` for each execution.

- In line 17 we define the upload stage. This stage is responsible for uploading a previously acquired raw frame into OpenGL-mapped memory. We call the constructor of the `Stage` class (see line 34 of Listing 4.3), and pass the following parameters:

  - The first parameter to the constructor is the *task* definition for this stage. In our example, this is a function pointer to the free function `upload_frame` defined in line 1. Note that this could also be a C++11 *lambda* expression or some bound expression using `std::bind`.

  - The second parameter of the constructor passes a vector of output tokens from which the *output upstream queue* is first initialized. The dimensions of this vector also define the maximum size of the output queues. In our example, we pass a vector of three default-constructed instances of `OpenGLFrame` for initialization.

```
1  StageCommand upload_frame(HostFrame& input, OpenGLFrame& output) {
2
3      if (input.marks_end_of_sequence())
4          return StageCommand::STOP_EXECUTION
5
6      output.upload(input.data());
7
8      return StageCommand::NO_CHANGE;
9  }
10
11 ...
12
13 Stage<NO_INPUT, HostFrame> acquire_stage = ...;
14
15 std::vector<OpenGLFrame> init{3};
16
17 Stage<HostFrame, OpenGLFrame> upload_stage{&upload_frame, init,
       acquire_stage};
18
19 Stage<OpenGLFrame, NO_OUTPUT> display_stage = ...;
20
21 while (display_stage.status() == PipelineStatus::READY_TO_EXECUTE) {
22
23     if (acquire_stage.status() == PipelineStatus::READY_TO_EXECUTE)
24         acquire_stage.execute();
25
26     if (upload_stage.status() == PipelineStatus::READY_TO_EXECUTE)
27         upload_stage.execute();
28
29     display_stage.execute();
30
31 }
```

Listing 4.4: Example usage of the stage class shown in Listing 4.3.

  - The third parameter is the input stage to the stage that is being constructed. This is the stage that acquires a frame. Note that a stage only stores references to the output queues of its input stage, i.e. stages communicate with each other *only* via their queue connections, and never by calling each other directly.

- The function defined in line 1 performs the task of the upload stage. If the input frame marks the end of a sequence, and pipeline execution should therefore be stopped, a proper flag is returned. If `input` represents a normal frame, then its data is uploaded into the OpenGL `output` token. Note that `HostFrame` and `OpenGLFrame` are classes that only exist for the purpose of demonstrating this example code. The actual OpenGL implementation of this framework is more complex and described in the next Chapter.

- In line 19, the stage that is responsible for displaying an OpenGL buffer is created. The details of its construction are hidden, the only thing that we need to know is that it is con-

```
1
2  ...
3
4  std::thread acquire_thread{[&]{
5
6      while (acquire_stage.status() == PipelineStatus::READY_TO_EXECUTE)
7          acquire_stage.execute();
8
9  }};
10
11 std::thread opengl_thread{[&]{
12
13     while (display_stage.status() == PipelineStatus::READY_TO_EXECUTE) {
14
15         if (upload_stage.status() == PipelineStatus::READY_TO_EXECUTE)
16             upload_stage.execute();
17
18         if (display_stage.status() == PipelineStatus::READY_TO_EXECUTE)
19             display_stage.execute();
20
21     }
22
23 }};
24
25 acquire_thread.join();
26 opengl_thread.join();
27
28 ...
```

Listing 4.5: Parallel execution of the pipeline defined in Listing 4.4.

nected with the upload stage and that it doesn't produce output token as its only purpose is to show frames on the display.

- The execution of the pipeline is triggered in line 21 and following. In this example, both stages are executed serially. Their stages are executed in a loop as long as there are still frames to be processed. The stage's `status()` method is checked repeatedly in order to determine whether a stage still has tokens to process.

Listing 4.4 shows the serial executions of a pipeline. Using the preliminaries of this example, Listing 4.5 shows how this pipeline could be executed in parallel:

- With the stage definitions of Listing 4.4 at hand, two threads are created in line 4 and line 11.
- Notice that the two OpenGL stages (upload and display) still execute serially and that only the stage to acquire the frame runs concurrently to the OpenGL stages. Let us assume that the acquire stage and the upload stage are both relatively heavy operations and that displaying a frame, once it is uploaded, is really quick. Then this implementation could be potentially twice as fast as the previous serial implementation (also see Chapter 3.1.3).

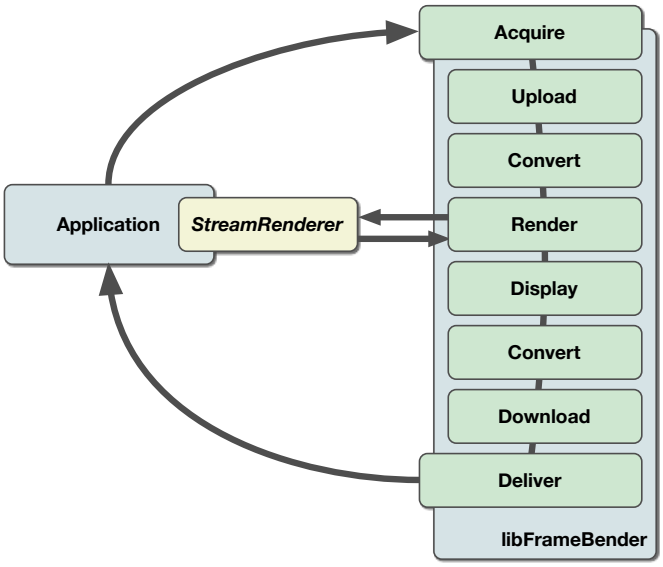


Figure 4.1: Overview of the general video processing steps. The library provides an interface to a generic video renderer, i.e. the actual rendering task can be defined by the application that uses the library.

- In line 25 both threads are joined, i.e. the calling thread waits until the stages have no more data to process.

4.2 OpenGL video render pipeline

In this chapter, we describe how the video render pipeline is implemented using the pipeline infrastructure of the previous chapter. We show how to use the OpenGL API to implement the video processing. We split this processing into several independent stages with the goal of executing them concurrently on different hardware units of the machine. The chapter begins by explaining the setup and structure of the video pipeline implementation. The scheduling of the stages is described by Chapter 4.2.3, and the hardware-concurrent execution of its stages is explained later in Chapter 4.3.

The video pipeline implementation is the heart of *libFrameBender*. It provides the client application of *libFrameBender* with a simple API to acquire, render and deliver video frames. Figure 4.1 shows the basic steps of the video processing engine. First a frame needs to be *acquired*, i.e. read from memory. It is the client application's responsibility to provide the pipeline with video frames that are stored in main memory. Once a frame is made available, it is uploaded to video memory of the graphics card. If necessary, the video frame is converted from its input native pixel format to the canonical render format of the pipeline (see Chapter 4.4). The video frame is then passed as the input to the *renderer*, which in turn stores its rendered result into some other video frame on the graphics card's memory. The *renderer* is a C++ interface. Its implementation defines the actual rendering operation, such as the one described in Chapter

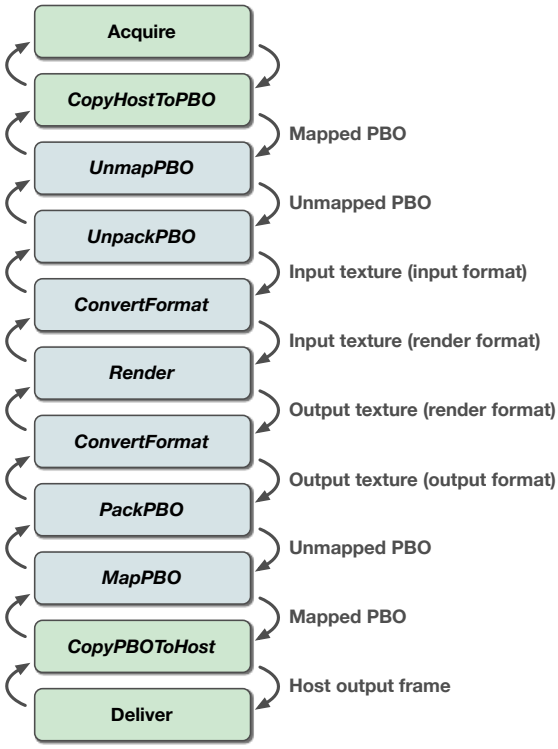


Figure 4.2: C++ stages of the video pipeline implementation. Green stages represent stages that only use the CPU for processing, blue stages use the OpenGL API to perform GPU-based tasks. No assumptions are yet made about the scheduling of the stages.

4.5. After rendering, the resulting video frame can be displayed in a player window. This is an optional step and can be bypassed completely in order to avoid any side effects on performance. In any case, the rendered frame is then converted from the canonical render format to its output native pixel format. Finally, the converted frame is downloaded from video memory to main memory from which it is delivered back to the client. These are the basic steps of rendering a video frame with our video pipeline.

4.2.1 Stage definitions

The pipeline uses the OpenGL API in order to fully take advantage of the GPU's performance in rendering and pixel processing. The logic of the pipeline uses the C++ infrastructure that was described in Chapter 4.1. I.e., the complete processing is split across several C++ stages. A diagram of all C++ stage implementations of the video render pipeline is given in Figure 4.2. The C++ infrastructure of a stage allows the execution of its task from any thread. In practice, however, we serialize some stage executions of our pipeline since they do not map to different hardware units, and therefore parallelization would not bring a benefit. From the software model's perspective, the definitions of a stage and the actual execution model are orthogonal.

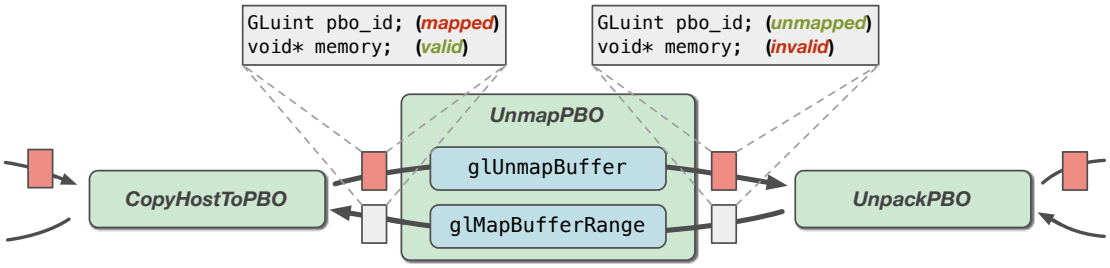


Figure 4.3: The *UnmapPBO* stage is the gateway of unmapping/mapping pixel buffer objects for client-side writing operations. This diagram shows the state of the frame token before and after execution of the *UnmapPBO* stage. The colored token represents a downstream token that carries a new video frame. The process described here depends on the ability to communicate tokens *upstream*, i.e., to the predecessor of the current stage (see Chapter 3.1.1).

In the following, we describe the stage definitions, without considering the actual threading of their executions yet, which is described later in Chapter 4.2.3.

4.2.1.1 *Acquire*

For each execution of this stage, a user-configurable callback is called that is expected to provide the pipeline with one video frame. The format of the video frame is well-defined, and the frame's image data is expected to be stored in main memory. The client of *libFrameBender* provides this callback. When the callback is called, a frame could be read from disk, or retrieved from a video capture card, for instance. The queues between this stage and the next one keep a constant number of frames circulated. The pipeline therefore buffers frames on behalf of the client application.

4.2.1.2 *CopyHostToPBO*

This stage takes a previously acquired frame and copies its data to a memory location that has been mapped by OpenGL for a particular *pixel buffer object* (PBO). A description of OpenGL PBO techniques is given in Chapter 2.3.6. While the identifier of the PBO has to be passed along the output tokens in order to be uniquely identified, no OpenGL calls actually need to be executed by this stage. This has the advantage of this stage being able to execute on a worker thread that is not associated with an OpenGL context (see Chapter 4.2.3).

4.2.1.3 *UnmapPBO*

This stage manages the mapping of OpenGL PBO objects to memory locations in host memory. Depending on whether data is passed *downstream* or *upstream*, it performs two different operations (see Figure 4.3):

- *Downstream*: The input token's mapped PBO location is unmapped using `glUnmapBuffer`. The identifier of the now-unmapped PBO is written into an output token which is passed further downstream to the following stage. This is necessary because the *UnpackPBO* stage will perform operations which require a PBO to be unmapped.
- *Upstream*: The output token's PBO is mapped to local memory using `glMapBufferRange`. The resulting memory location is written to an input token which is passed further upstream to the previous stage. Again, this relieves the stage *CopyHostToPBO* from the responsibility to have an OpenGL context bound during execution (and therefore is able to run concurrently).

Note how important it is for this stage to be able to pass data upstream as well as downstream. As hinted by Figure 4.2, PBO identifiers are passed from *CopyHostToPBO* via *UnmapPBO* to *UnpackPBO* and then circulated back. Most other stage connections circulate tokens only between two stages.

4.2.1.4 *UnpackPBO*

The input video frame has now been copied to OpenGL memory (PBO). This stage is responsible for copying (*unpacking*) this memory into an OpenGL texture. During execution, the input token's PBO is bound to the OpenGL *pixel unpack buffer* (see Chapter 2.3.6). An asynchronous upload to the designated output texture is then triggered by executing the OpenGL command `glTexSubImage2D`. Essentially, this triggers an asynchronous DMA transfer to video memory.

The output frame is now represented only by an OpenGL *texture* id. This stage and the following one exchange tokens over a fixed number of *input textures* via their connected queues. Note that even though the output of this stage references an OpenGL texture, its content is still stored in the original input pixel format of the video.

4.2.1.5 *ConvertFormat (1)*

In order for the next stage to use the video frame for rendering, it first has to be converted into the *canonical render pixel format* of the video pipeline (see Chapter 4.4). The output of this stage references an OpenGL texture that stores the video frame in the render pixel format.

The conversion is achieved by the class *ConvertFormat* which is a generic stage that allows to convert back and forth between two different formats. On execution, the following steps are executed:

- Assemble GLSL shaders that are able to convert the specified input format to the desired output format.
- Bind the input token's texture ID to the input *sampler uniform* of the previously compiled shader.
- Run the shader over the whole input texture image and write its fragment output into the output token's texture.

OpenGL provides multiple ways to write the output of a fragment shader to a texture. This framework implements several flavors for transcoding the V210 10-bit Y'CbCr format. These will be described later in Chapter 5.

4.2.1.6 *Render*

The input video frame is now available as an OpenGL texture and has already been converted to the canonical render format. The *Render* stage executes a rendering algorithm that takes this texture as input and provides its output as another texture. The output texture is again stored in the canonical render format. While the render stage is responsible for triggering the rendering operation, it does not contain the actual implementation. Instead, the *libFrameBender* library provides the client of the library with the C++ interface *StreamRenderer* (see Figure 4.1). The client is expected to implement the interface and to pass the implementation to the render stage during construction. This provides the client with a simple way of defining its own pluggable video processing algorithm, without having to take care of the frame transfers and format conversions.

If desired, this stage is able to display the output texture in a user-specified window. This can be used to create a simple player system or to provide *confidence monitoring* for the live broadcaster. However, the primary goal of this stage is to render a client-defined graphics algorithm (e.g., blending images over the video stream) into another texture. Displaying the result on screen is optional. As such, display timing (e.g. vsync) must not govern the timing of the rendering operation.

4.2.1.7 *ConvertFormat (2)*

The input token of this stage references a texture that stores the output frame of a previous rendering operation. In order to prepare the download of this frame to main memory, this stage converts the input frame from its canonical render format to the specified output pixel format. This conversion uses the same approach as *ConvertFormat (1)*, only with different pixel formats.

4.2.1.8 *PackPBO*

The output frame has been converted to the output pixel format, and this stage now triggers an asynchronous *download* from the input texture to the output token's OpenGL PBO. This is done by binding the PBO to the *pixel pack buffer* and by executing the `glGetTexImage` command for the input token's texture id.

4.2.1.9 *MapPBO*

Similar to the stage *UnmapPBO*, this stage performs two different operations, depending on whether data is passed downstream or upstream:

- The *downstream* input token's PBO is mapped to local memory using `glMapBufferRange`, and the resulting memory location is passed further down-

stream to the next stage. This relieves the stage *CopyPBOToHost* from the responsibility to have an OpenGL bound during execution.

- The *upstream* output token's mapped PBO location is unmapped using `glUnmapBuffer`. The identifier of the now-unmapped PBO is passed further upstream to the previous stage. This is necessary because a texture can only be packed into an *unbound* PBO.

4.2.1.10 *CopyPBOToHost*

The video frame that is referenced by the input token's mapped PBO memory location is copied to local memory of the output token's frame representation.

4.2.1.11 *Deliver*

In this final stage, the input token contains the frame that was copied by the previous stage. The client is expected to provide this stage with a callback that is called for each frame that arrives in this stage. In this callback, the client implementation could write the frame back to disk, or pass it back to a video card.

4.2.2 Data pools for queue elements

The previously described stages communicate via multiple circular queues (see Chapter 3.1). The data elements of those queues are always allocated during construction of the pipeline. During the execution of a stage's task, only references are passed along via tokens. This is the case for OpenGL buffers as well as host-side data structures. The following is a list of resource pools which are used by the pipeline:

- **Input frames**: Host-side data elements storing data frames in local memory.
- **Upload PBO buffers**: OpenGL pixel buffer objects that allocate OpenGL-controlled memory that is large enough to store multiple video frames in their input pixel format.
- **Upload textures**: OpenGL texture objects that are the destination when unpacking pixel buffer objects from the upload PBO buffers.
- **Input render textures**: OpenGL texture objects that store input video frames in the canonical render format in order for them to be passed to the renderer. Each texture is attached to an individual OpenGL framebuffer object, whose identifier is always passed along with the texture.
- **Output render textures**: OpenGL texture objects that store the results of the renderer in the canonical render format. Each texture is attached to an individual OpenGL framebuffer object, whose identifier is always passed along with the texture.
- **Download textures**: OpenGL texture objects that store the output frame in the output pixel format, i.e. they hold the result of conversion into the output pixel format.

- **Download PBO buffers**: OpenGL pixel buffer objects that are used as the destination for *packing* the download textures.
- **Output frames**: Host-side data elements storing video frames in local memory.

The size of each pool is configurable in the framework. It is important to set the size of each pool to a sensible number of elements. A queue with too few elements might cause exhaustion of resources, i.e. stages have to wait too long for tokens to become available. On the other end, very large pools might cause cache misses.

4.2.3 Scheduling of the video processing pipeline execution

Until now, we have not made any assumptions about the threading of the stage executions. In the simplest case, all stages are executed serially on the same thread. This approach does not use the resources of the system efficiently. The hardware units that actually perform the processing of our algorithm, i.e., CPU, GPU and DMA controllers (see Figure 3.1) have hardware-based pipelines of their own. If only one software stage runs at a time, these hardware pipelines do not have enough data available to run efficiently. We therefore need to execute as many software stages as possible in parallel in order for the hardware units to run efficiently, i.e., we assign stage execution to multiple threads. In principle, an instance of a *stage* can be executed from any thread. The call to execute a stage blocks until all data dependencies are met and the stage's task has been executed (see Chapter 4.1.3).

Consider the previous example given in Listing 4.5: As soon as the `acquire_thread` has first executed the `acquire_stage`, the other thread will immediately execute the `upload_stage` in line 16, because its data dependency is now met. At the same time, the `acquire_thread` is able to trigger the next execution of the `acquire_stage`, because its output queue is able to hold more than one output element. Therefore the first execution of `upload_stage` and the second execution of `acquire_stage` execute concurrently.

4.2.3.1 Concurrency of OpenGL-based stages

OpenGL was originally designed to provide a graphics API that is used by a client from a single thread. Calls to the OpenGL API require an *OpenGL context* to be active for the executing thread. The OpenGL API is not thread-safe, i.e., OpenGL-based calls can only be executed from the thread that is assigned to its context. Therefore, for those stages of our pipeline that use OpenGL calls, the following needs to be considered:

- The executing thread must have an active OpenGL context set.
- Stages that call the OpenGL API can only be executed concurrently if each concurrent execution uses its own dedicated OpenGL context.

These constraints apply to the stages *UnmapPBO*, *UnpackPBO*, *ConvertFormat*, *Render*, *PackPBO* and *MapPBO*. These stages need to be either executed by the same thread, or need to use an individual OpenGL context that shares resources with the other contexts for each thread.

```
1
2  ...
3
4  std::thread opengl_thread{[&]{
5
6      while (display_stage.status() == PipelineStatus::READY_TO_EXECUTE)
7      {
8
9          while (upload_stage.status() == PipelineStatus::READY_TO_EXECUTE
10                 && display_stage.input_queue_num_elements() < 2)
11         {
12             upload_stage.execute();
13         }
14
15         if (display_stage.status() == PipelineStatus::READY_TO_EXECUTE)
16             display_stage.execute();
17
18     }
19
20 }};
21
22 ...
```

Listing 4.6: *Interleaving* the OpenGL stage executions.

The decision of whether to use multiple contexts or to serialize OpenGL-based stages is not straightforward. Some OpenGL driver implementation are able to perform asynchronous transfers via separate GPU DMA controllers when the hardware supports it and when multiple contexts are used from multiple threads (see Chapter 4.3.2). In our example of Listing 4.5, we could then split the thread `opengl_thread` into two threads: `upload_opengl_thread` and `display_opengl_thread`. This would allow the DMA controller to transfer the frame asynchronously to the CPU *and* GPU.

For other implementations, the use of multiple contexts only creates a significant overhead of driver-side synchronization, and performs worse compared to serial execution using a single context. The best strategy for multithreaded OpenGL therefore depends both on hardware architecture and the particular implementation of the OpenGL driver. Because there is no single best choice, our framework uses a single thread for OpenGL stages by default, and provides multi-threaded OpenGL usage as one of several settings for pipeline scheduling (see Chapter 4.3).

4.2.3.2 Interleaved stage executions on a single thread

This chapter describes a feature of the pipeline scheduler to optimize single-threaded execution of several stages. For stages using OpenGL, this technique can be used as an alternative to using multiple OpenGL contexts from multiple threads.

In the example given in Listing 4.5, each thread executes the stages as fast as possible, immediately triggering the executions when data is available. Therefore, the overlapping of stage

executions between multiple threads happens naturally. However, in this example, the OpenGL-based stages are executed serially by a single thread. When two stages are executed serially in one thread, the data token that was produced by the first stage is processed immediately by the following stage. Let us look at Listing 4.5 again: When the `upload_stage` uploads frame `N`, the next execution of `display_stage` in line 19 will immediately display frame `N`, because they both run serially in the same thread `opengl_thread`. Commanding OpenGL to display a frame immediately after its upload might result in *implicit synchronization*, because the OpenGL pipeline has to wait until the upload is done before it can display the frame (see Chapter 2.3.6). This can result in undesired performance penalties and should be avoided. One possible solution would be to use multiple OpenGL contexts with multiple threads as described in the previous chapter. Alternatively, while still using a single thread, we could upload some frame `N` when executing `upload_stage` and have the following execution of `display_stage` display frame `N-1`. This has the following consequences:

- It should be less likely for the OpenGL driver to enforce an implicit synchronization point. As a result, the CPU-side execution is not blocked and we can use its resources for other tasks.

- The internal pipeline of the OpenGL driver is better saturated, which potentially allows the driver to parallelize its own pipeline executions.

Listing 4.6 shows a modified version of the `opengl_thread` of Listing 4.5 that implements this *interleaved* scheduling between the upload and display stages of our example. In line 10 of the example, the upload stage is now executed repeatedly until its downstream output queue holds two frames. We call this the *input constraint* of the following stage, i.e. the display stage of our example has an input constraint of two. In other words, the display stage waits until its input queue has a load of at least two elements. When starting the example, the upload stage is therefore executed twice before the display stage is executed. After the first round of executions, the upload and display stage will then execute alternately in lockstep. But the display stage always buffers an additional element in its input queue, and doesn't process the frame right after it has been uploaded. Notice that in our example, we might have improved the performance by avoiding implicit synchronization, but we have also added one frame of latency to the overall algorithm.

In summary, the purpose of interleaved executions and using multiple OpenGL contexts is to parallelize the execution of OpenGL commands. Using multiple contexts has the advantage that the client application itself can use different threads and therefore overlap at least the dispatch of OpenGL commands at the call-site of the client application. However, this method is required to explicitly synchronize between OpenGL contexts, which is complicated and adds overhead to the execution of their tasks. Using a single context with interleaved executions, this synchronization is not necessary, but we add a fixed number of frames to latency. Whether OpenGL commands are actually executed in parallel, ultimately depends on the OpenGL driver implementation. It is therefore important to implement multiple approaches and choose the one that shows the best performance.

4.2.3.3 User-controlled scheduling setup

The *libFrameBender* implementation allows the user of the library to manually configure large parts of the execution model that has been described in the previous chapters. This flexibility is provided by a scheduler that assigns one or more stages to multiple threads. The scheduler also supports the definition of the previously explained *input constraints* for their queues. Level of concurrency, queue sizes and input constraints can be set by the user from the outside. This provides the ability to define queue setups with different characteristics, e.g. deep pipelining vs. lower latency. Chapter 6.1 shows examples of possible pipeline configurations. Executing stages on multiple threads changes the performance characteristics of the pipeline drastically. These optimizations are described in the following chapter.

4.3 Concurrent hardware executions of the pipeline

In Chapter 4.2, we showed the structure of the OpenGL video pipeline and described details and limitations of how to schedule the execution of its stages. This chapter explains how this mechanism can be used to actually execute stages concurrently on different hardware units of the system. The client of the library can enable and disable these mechanisms. This can be used for measuring the effectiveness of the applied strategy (see Chapter 6).

The pipeline that we have described so far describes a software model that conceptually enables the concurrent execution of its stages. However, the actual degree of concurrency that can be reached depends on the availability of hardware resources of the system that the pipeline is executing on. Potentially, the following hardware units could execute stages in parallel:

- The **GPU** provides thousands of individual processing units that are able to process data-parallel tasks such as processing pixels in parallel while the rest of the system is performing other tasks.
- A **DMA controller** on the graphics board is able to asynchronously handle the transfer of data to and from video memory.
- Individual **CPU cores** are able to execute generic processing tasks concurrently, such as the host-side copying of video data.

By default, the framework executes the pipeline on a single thread serially. This is equivalent to a naive single-threaded implementation of the overall algorithm.

4.3.1 Asynchronous host copies

The first optimization technique is to offload the copying of frame data between host-side memory and OpenGL-mapped memory to worker threads. Instead of one thread for the overall pipeline execution, we now use three (see Figure 4.4):

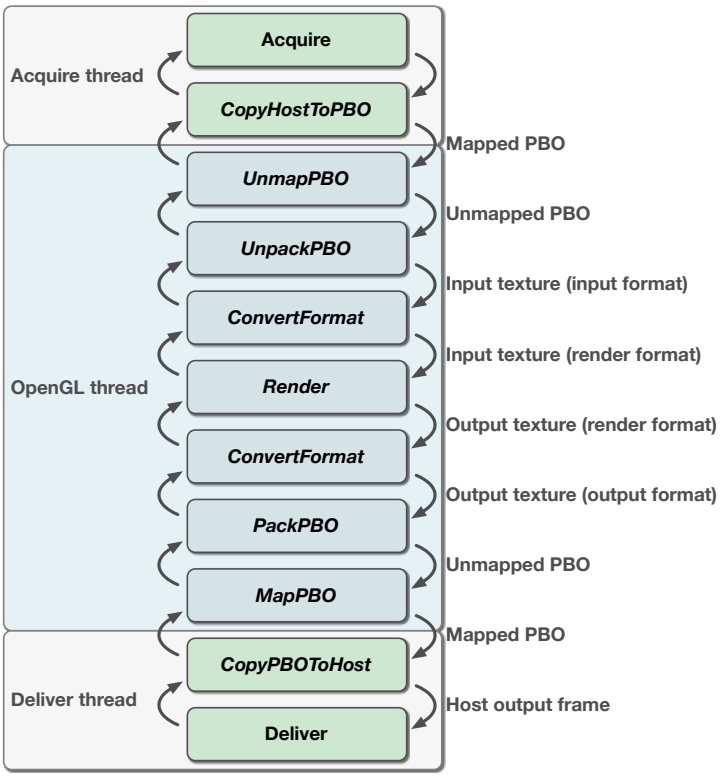


Figure 4.4: Running the host copies for input and output on separate threads. OpenGL-based tasks run on a single thread. Note that we could split the acquire and deliver thread again in order to overlap client-based frame handling with the copying operation.

1. **Acquire thread**: Acquires a frame and copies the input frame's memory into memory mapped by OpenGL for transfer. This thread executes the following stages[1]:
   - *Acquire*
   - *CopyHostToPBO*
2. **OpenGL thread**: Performs all stage executions which require an OpenGL context to be attached. This thread executes:
   - *UnmapPBO*
   - *UnpackPBO*
   - *ConvertFormat (1)*
   - *Render*

[1] In our prototype implementation, stages *Acquire* and *Deliver* use only a very small amount of CPU time. We therefore decided to execute these stages in the same threads as *CopyHostToPBO* and *CopyPBOToHost*. If they performed heavier tasks (e.g. reading files from hard disk) then these stages should have their own thread assigned.

- *ConvertFormat (2)*
- *PackPBO*
- *MapPBO*

3. **Deliver thread**: Copies memory mapped by OpenGL for downloaded frames into host-side memory of the output frames and execute the user-supplied callback for frame delivery. Executes the following stages[1]:
   - *CopyPBOToHost*
   - *Deliver*

This configuration therefore overlaps host-side copy operations with OpenGL-specific tasks, i.e., while the GPU is rendering. In comparison with the serialized execution of these stages, this approach has a much better overall performance. An exact measurement of these performance gains is given in Chapter 6.

4.3.1.1 Intel IPP - optimized memory copying

The Intel IPP library [15] is a library providing optimized algorithms for Intel CPUs. This library provides an optimized copying routine `ippiCopyManaged_8u_C1R`. When copying large data, Intel recommends to pass the flag `IPP_NONTEMPORAL_STORE` to this copying routine in order to bypass caching of the destination memory [14]. Otherwise copying large data would result in clearing the complete existing cache and might cause undesired cache misses in any subsequent memory read operations. We have experienced that using this routine in the stages *CopyHostToPBO* and *CopyPBOToHost* improves performance, especially in configurations that are bound to the bandwidth of the CPU.

4.3.1.2 AMD pinned memory

The stages *UnmapPBO* and *MapPBO* unmap and map OpenGL pixel buffer objects to host-side memory locations in order to copy video frames to and from these locations in other stages. Both stages perform an OpenGL *map* as well as an OpenGL *unmap* command in a single execution. The name of those stages denotes the downstream operation, e.g. *UnmapPBO* unmaps an already-mapped OpenGL buffer in the downstream direction because following stages require the OpenGL buffer to be unmapped (see Figure 4.3). At the same time this stage maps the upstream tokens of OpenGL buffers that are currently not used, and which are free to be filled with host-side video frames.

This rather complex setup is necessary because OpenGL forbids to use PBOs for texture upload in a mapped state. The OpenGL extension `GL_AMD_pinned_memory` [25] adds the possibility to use an existing host-side memory region as a persistent backing storage for OpenGL pixel buffer objects (see Chapter 2.3.7.2). When using this extension, the host-side memory location of a PBO is now persistent [1]. We can therefore simplify the *UnmapPBO* and *MapPBO*

[1] At the time of writing, the OpenGL 4.4 core specification has been published which provides a similar technique through the extension ARB_buffer_storage. However, OpenGL 4.4 was not available at the time of implementing the framework, and therefore is not considered in this thesis.

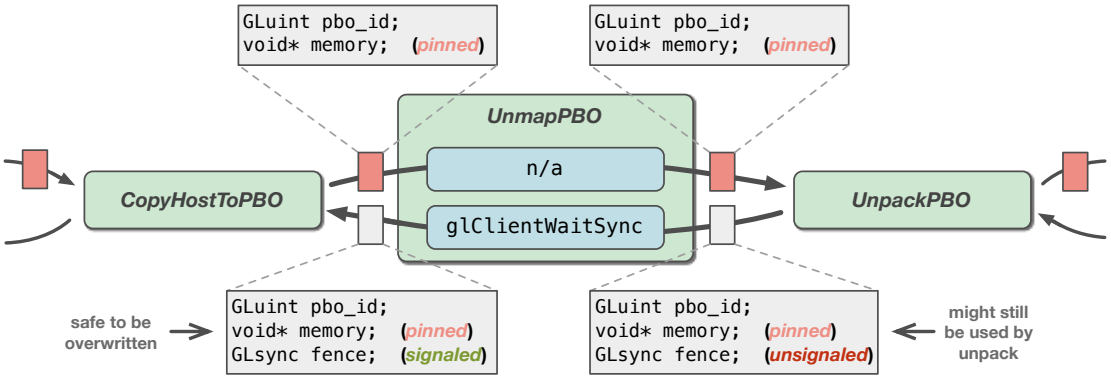


Figure 4.5: Using the AMD pinned memory extension for backing the OpenGL PBOs relieves the UnmapPBOStage of actually mapping OpenGL buffers, the memory is pinned (persistent). However, this approach requires explicit synchronization: *UnpackPBO* inserts a *fence* upstream, *UnmapPBO* waits until this fence has been signaled. It is then safe for *CopyHostToPBO* to overwrite the pinned memory of the token.

stages and skip the map/unmap OpenGL commands during their execution. However, access to these memory locations now has to be synchronized explicitly with OpenGL: *UnmapPBO* applies the OpenGL command `glClientWaitSync` on the upstream token's `fence` field, which was inserted by the *UnpackPBO* stage (see Figure 4.5). This ensures that the buffer is not used by any OpenGL command that has been triggered further downstream. When this token is then passed to *CopyHostToPBO*, this stage can safely copy data into the persistent backing storage of this PBO. This process is shown in the second scenario in Figure 4.5. Respectively, *MapPBO* has to synchronize downstream before *CopyPBOToHost* reads from its memory.

4.3.2 Multithreaded OpenGL for upload/render/download

Chapter 2.3.7.1 described a technique that allows concurrent execution of upload, render and download commands on the GPU using NVIDIA Quadro graphics cards. This is enabled by the two DMA units of this graphics card. The optimization can be implemented by using multiple OpenGL contexts for upload, render and download OpenGL commands. When each of these contexts runs on its own thread, the driver can automatically parallelize the GPU-side execution.

Our video pipeline can be configured to implement this scenario and to distribute the stages that execute OpenGL commands to three threads instead of just one (see Figure 4.6):

1. OpenGL upload thread:
   - *UnmapPBO*
   - *UnpackPBO*
2. OpenGL render thread:

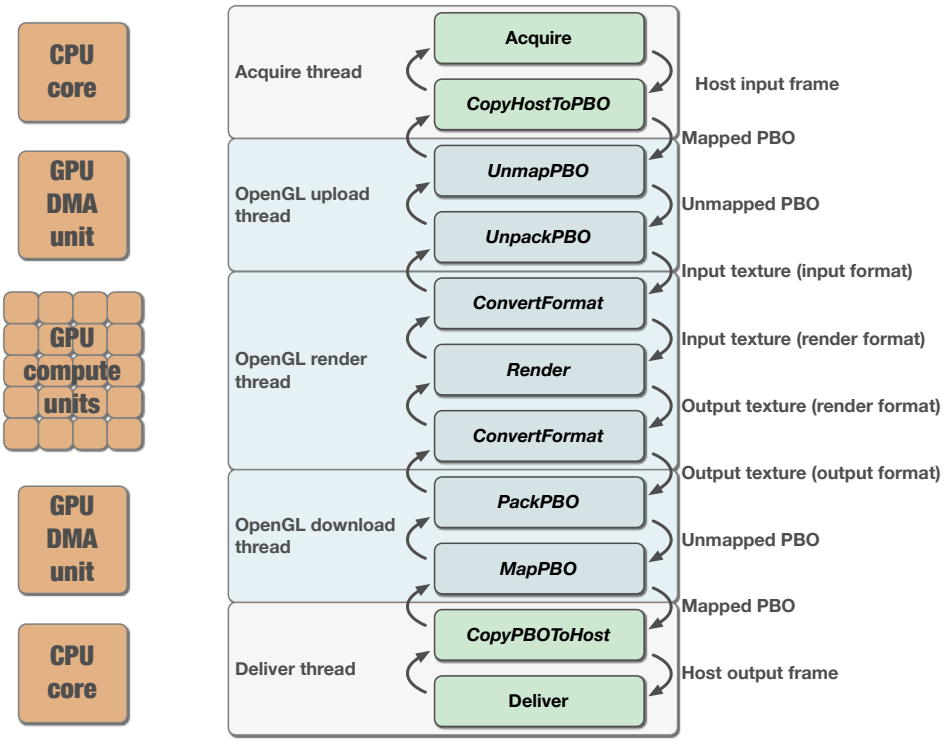


Figure 4.6: Using three different threads for the execution of OpenGL stages. On the left side of the diagram we show the hardware units that potentially execute the stages' task in parallel.

- *ConvertFormat (1)*
- *Render*
- *ConvertFormat (2)*

3. OpenGL download thread:
   - *PackPBO*
   - *MapPBO*

When sharing OpenGL data like textures and pixel buffer objects across multiple contexts, it is necessary to explicitly synchronize between the context boundaries using `ARB_sync` (see Section 4.1 in the OpenGL specification [29]). This is done with the following two commands:

- `GLsync fence = glFenceSync(...)`: Adds a new fence into the GL command queue, i.e. all previous OpenGL commands are added into the GL command queue before this fence.
- `glWaitSync(fence, ...)`: Adds a wait-command into the GL command queue for a particular fence, i.e. OpenGL commands that are executed after this wait command are

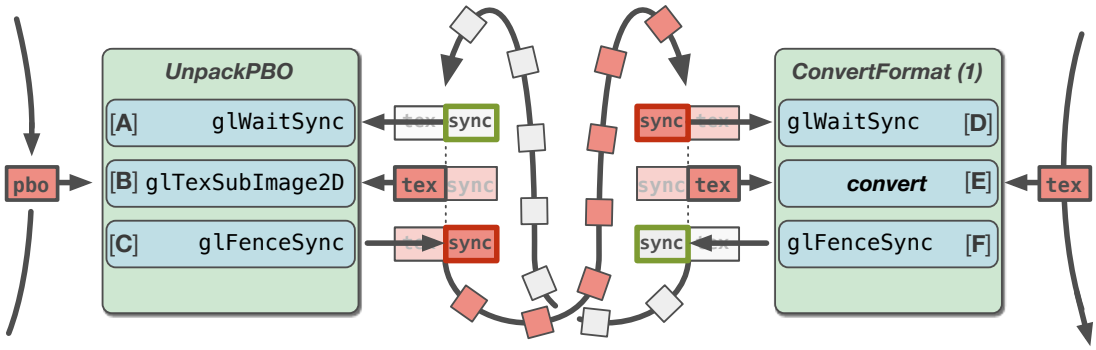


Figure 4.7: Stages *UnpackPBO* and *ConvertFormat (1)* use different OpenGL contexts. Therefore, they need to perform a two-way OpenGL synchronization to avoid the concurrent modification and rendering of a texture's image data. The red tokens in the diagram are tokens that carry new video frames, i.e., the data that is passed downstream.

guaranteed to be executed after the OpenGL commands that preceded the previous fence creation. Unlike `glClientWaitSync`, this command does not synchronize the client execution, i.e. the call does not block the calling thread.

In our pipeline, the following stages now use the above commands to sync explicitly:

- *UnpackPBO*: The target of this stage's unpacking operation is an OpenGL texture which is shared between this stage and the *ConvertFormat (1)* stage. Because *ConvertFormat (1)* uses a separate context in this optimization, explicit OpenGL synchronization is necessary. The textures between those two stages are cycled back and forth via the tokens of the output up- and downstream queues. Because the stage's queues only synchronize CPU-side execution, but do not know about the internal pipelining of the OpenGL implementation, the texture that has been retrieved by the upstream output queue might still be used by some OpenGL command of a previous *ConvertFormat (1)* execution. We therefore add a `GLsync` field next to the `GLuint` texture id field of the output queues' token data structure. Both stages are able to synchronize their OpenGL executions with each other using the sync fields of the tokens. *UnpackPBO* now synchronizes as follows (see Figure 4.7):

  A Synchronize with the output token's `GLsync` field using `glWaitSync`. This fence was created after a previous conversion operation that used the output token's texture id. Therefore syncing to this fence is equivalent to asking the OpenGL pipeline to wait for any previous format conversion that still depends on this texture's content to finish.

  B Perform the unpacking operation from the input token's PBO into the output token's texture using `glTexSubImage2D`.

C Create a new fence using `glFenceSync` and write the new fence into the output token's `GLsync` field. The following stage can sync with this this fence in order to make sure that the above unpacking operation is complete.

- *ConvertFormat (1)* needs to sync as well (see Figure 4.7):

  D Synchronize with the input token's `GLsync` field. This makes sure that the unpacking operation (i.e. the texture upload) has been finished for this texture before it is used as the input for the format conversion.

  E Perform the format conversion from the input token's source texture into some destination texture.

  F Create a new fence using `glFenceSync` and write the fence into the input upstream's token `GLsync` field. This allows the previous stage (i.e. *UnpackPBO*) to ask OpenGL to wait for this conversion to be finished before uploading another frame into this particular texture.

- Because *ConvertFormat (2)* is executed by the *render* OpenGL context and *PackPBO* is executed by the *download* OpenGL context, they synchronize their OpenGL execution in a similar way as *UnpackPBO* and *ConvertFormat (1)* did.

When using this multi-threaded approach with multiple OpenGL contexts, we are able to take advantage of the dual copy engines of recent NVIDIA Quadro hardware, which greatly improves the throughput of the video pipeline. Chapter 6 provides exact numbers for the actual improvement in throughput.

4.4 The canonical render format

The *ConvertFormat* stage of the video pipeline transcodes between the native video format and the format which is used for rendering. The precision and color space of the render format can be configured by the client of the video processing library. Higher precision and more accurate conversion of color spaces require more processing and greater bandwidth. By making some aspects of the render format configurable from outside the library, the user of the video processing framework is able to make a compromise between performance and quality, depending on the needs of the targeted use case. Chapter 6 provides a comparison between performance and image quality of various configurations of our prototype.

The render format always represents an RGBA pixel format. The specific configuration of the format has to be natively supported by OpenGL. The render format's RGB values can be configured to be stored as *linearly coded* intensity values (see Chapter 2.1.1). This results in higher quality of most rendering operations, but requires the format converters to perform image transfer between the perceptually encoded values and the linear RGB values of the render format. Chapter 5 provides an example of how to perform this conversion for ITU-709-encoded video.

The bit-depth and precision of the render format can be configured to be one of the following:

Figure 4.8: The *demo renderer* blends the logo and the lower-third images on top of the original video material. Image appearance is animated using ease-in and ease-out curves. The image in the background is a single frame of the EBU 1080P test sequence [4].

1. RGBA normalized 8-bit integer format
2. RGBA 16-bit floating point (*half* precision)
3. RGBA 32-bit floating point (*full* precision)

Depending on the requirements of the input and output video formats of the configuration, a different choice of the above formats has to be made. The first format provides the best performance but the least precision. As opposed to the other two formats, this format is also not able to store values outside the unit processing range $[0..1]$, which is necessary to avoid clipping of RGB triples that originate from a Y'CbCr representation (see Chapter 2.1.6). The other two formats store floating-point values, but with varying precision.

4.5 Demo renderer

The video pipeline itself does not define the rendering algorithm. Instead, the *Render* stage defines an interface for rendering, which is implemented by the client of the library and then used by the pipeline.

For the purpose of our prototype, we provide the implementation of a demo renderer, which performs a simple broadcast graphics scenario: Blending images on top of a video stream. In

particular, a logo image is shown in the upper right, and another image is shown in the lower third (see Figure 4.8). The images are encoded in the sRGB color space, versions in 8-bit and 16-bit color depth are available. The blending operation is performed in linearly coded RGB space, i.e. the video material and sRGB images have to be transferred to a common space (see Chapter 2.1.7).

The position and opacity of the overlaid images is animated using Bezier-interpolation. The animation is driven by the input video's time stamps, which are sent along the image data through the video pipeline. The complete rendered sequence is available on YouTube [2].

4.6 Internal profiling

In order to measure the performance of the video pipeline, a profiling mechanism has been added to the framework. While there exist several tools for external performance analysis, our own profiling mechanism operates directly in the domain of our pipeline. This allows us to more easily analyze bottlenecks and to evaluate the optimization techniques which have been implemented.

4.6.1 Sampling timestamps of pipeline stage executions

Each stage of the video pipeline is able to record timestamps for every frame that is processed. The C++ implementation of a stage (Chapter 4.1.3) provides a generic mechanism to record high-resolution timestamps of *events* during stage execution. The following events are recorded in chronological order:

- `EXECUTE_BEGIN`: A stage has entered its execution phase, i.e. the time at which `Stage::execute` (see Listing 4.3) is called.
- `INPUT_TOKEN_AVAILABLE`: During execution, a stage waits until input and output token are available, i.e. until data dependencies are met. This event is recorded when an input token was made available.
- `OUTPUT_TOKEN_AVAILABLE`: Similar to above, an output token was made available. The stage is now ready to actually execute its task.
- `TASK_BEGIN`: The stage's task function is being called.
- `TASK_END`: The stage's task function is finished.
- `EXECUTE_END`: This event is recorded when the stage has finished its task, added the result token to the output downstream queue, and moved the input token back to the upstream input stream queue. This event basically records the time at which the method `Stage::execute` returns.

[2] http://youtu.be/a8V2AacbPUY

The timestamps of these events are not stored in the stage class itself. This is done by the helper class `StageSampler`, which is called by the stage implementation when an event should be recorded. This class provides the following features:

- Store timestamps of stage events for a predefined number of frames.
- Preallocate containers for timestamp records in order to avoid the overhead of allocating memory during profiling.
- Provide statistical summaries of all recorded timestamps.

The `StageSampler` class is implemented using the `boost::chrono` library. Timestamps are recorded using the clock `boost::chrono::high_resolution_clock`. Timestamp records are therefore defined by `boost::chrono::high_resolution_clock::time_point`. While we could have used C++11's `std::chrono` library implementation, which provides the same functionality, the implementation of Visual Studio 2012 was broken at the time of writing, which made us fall back to the Boost-supplied library implementation.

4.6.1.1 Measuring GPU timing

The previously described stage events are captured with timestamps that are relative to the CPU timer of the machine. This is useful for measuring general performance characteristics of the pipeline, like overall data throughput, etc. However, many stages trigger the execution of GPU algorithms, e.g. rendering the current video frame. In order to evaluate the performance of these tasks, we would like to measure the time it actually took the GPU to process it. Because the OpenGL driver internally buffers commands for pipelining, CPU timers can only be used to measure the time it took to enqueue OpenGL command to the driver's internal command queue. In order to measure the GPU time of a task, we use OpenGL *timer queries* (see Chapter 2.3.1). A timer query is inserted into the OpenGL command stream and records the GPU time when it is actually processed by the hardware. The result of a query needs to be explicitly requested by the client of the OpenGL API. Due to the pipelined nature of OpenGL, this request needs to made at a time when it is likely that its results is already available, i.e. that hardware has already processed the query. Asking for a timer query's timestamp too early results in a stall of the OpenGL pipeline. Our framework therefore uses a helper class that insert timer queries and records their timestamps in a lazy fashion, i.e. several video frames later than when they were inserted. This helper class reuses the `StageSampler` class and stores the GPU-times in the events `GL_TASK_BEGIN` and `GL_TASK_END`.

While most stage events are recorded automatically by the generic stage class, GPU timer queries have to be inserted manually by the stage's task definition. The task decides which section of its OpenGL commands should be measured for the GPU's time of execution.

4.6.2 Trace format

The set of timestamps for all stage events of all used pipeline stages for a single frame is called a *trace*. A trace uniquely identifies the timing history of a single processed frame. While pro-

cessing a video stream, the trace for each frame is recorded in internal data structures. These data structures can written to a file for later analysis of this session's performance. *Google protobuf* [8] is a C++ library that allows the user to easily define a data format for file storage. Our framework uses this library to save all frame traces of processing a video stream to one file. Among other things, this file stores the following data:

- Human-readable name of the processing session
- Information about OpenGL renderer and vendor
- Date and time of the recorded session
- Traces of all stages for all processed frames

Using the library *protobuf* to store the traces in its own independent file representation also enables the processing of its data in other language like Python. In Chapter 6.1.3, we show further applications of this format.

All profiling features are optional during the execution of our video processing pipeline. While the overhead of the described profiling features is relatively low, the client of the *libFrameBender* framework can decide to turn it off completely to avoid any processing overhead of time sampling.

4.7 Debugging features

The *libFrameBender* library exposes several debugging options to the client of the framework:

- The framework can be configured to use an OpenGL *debug context* (see Chapter 2.3.4) which helps to find errors in OpenGL usage quickly, and which also reports performance warnings of the driver.
- Some parts of the pipeline can be bypassed or turned off in order to isolate performance bottlenecks:
  - Bypass *input* stages: *Acquire*, *CopyHostToPBO*, *UnmapPBO* and *UnpackPBO* will not be executed.
  - Bypass *output* stages: *PackPBO*, *MapPBO*, *CopyPBOToHost* and *Deliver* will not be executed.
  - Bypass *render* stages: *ConvertFormat(1)*, *Render* and *ConvertFormat(2)* will not be executed.
  - Disable host-side copying: Stages *CopyHostToPBO* and *CopyPBOToHost* will not perform host-side copying of frames. This is useful for isolating GPU-centric tasks.
  - Force pass-through renderer: Stage *Render* will use a pass-through renderer instead of the user-configured renderer.

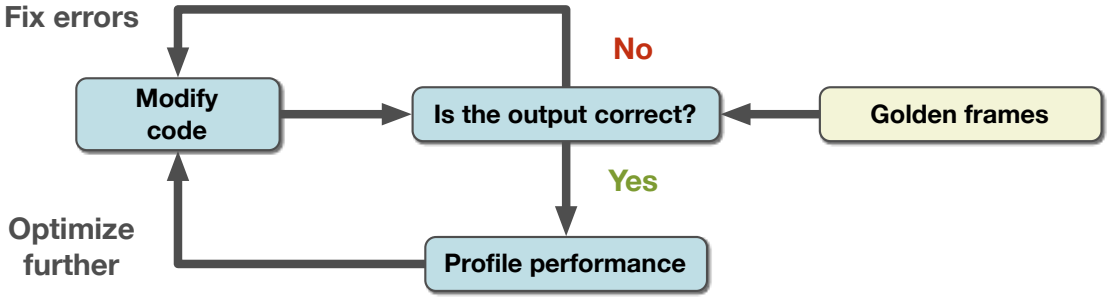


Figure 4.9: Once a correctly working implementation has been developed, optimizing the framework's performance was done in an iterative process: Based on a captured performance profile, modify code and then test for correctness of the video output by comparing with golden frames (expected output). If the output is not correct go back to fixing the code, otherwise continue profiling and optimize further.

- The framework verbosely logs events and errors of the pipeline. Logging is implemented using the `boost::log` framework. Log messages are automatically recorded to files. Printing log messages to the console can be enabled, but is disabled by default in order to avoid penalties in performance. If the framework is configured to use an OpenGL debug context (see Chapter 2.3.4), then the OpenGL debug output is also written into these log files.
- In order to monitor pipeline execution, a stage can be configured to log each encountered data token with its timestamp. This results in extremely verbose logging message and is usually turned off.

4.8 Testing

The framework uses several layers of testing. The most basic tests are unit tests of supporting infrastructure classes, like the C++ stage class. The library `boost::test` is used to implement a test runner. This framework provides convenient tools to register test scenarios and to build an executable that automatically executes these tests.

During development it was important to repeatedly test the correctness of the implemented image-processing algorithms. This was done by processing a well-known set of video frames, and testing the resulting video frames against *golden samples* which are known to represent the correct result for the tested algorithm. When using a pass-through renderer and when the format converters are configured to use the same format for source and destination, the output of the pipeline can be compared directly with the input frames. This proved to be one of the most useful test cases, where many errors could be found early during development.

While optimizing the framework, testing for correctness was a crucial part of the iterative process between development and performance measurement (see Figure 4.9).

CHAPTER 5

V210 Y'CbCr to RGB Transcoder

The V210 video format is widely used by professional video cards and camera interfaces. Our video framework natively supports the transcoding between this format and the canonical RGB render format (see Chapter 4.4). We have implemented the V210 transcoder in the *ConvertFormat* stage (Chapter 4.2.1.5) of our video processing pipeline.

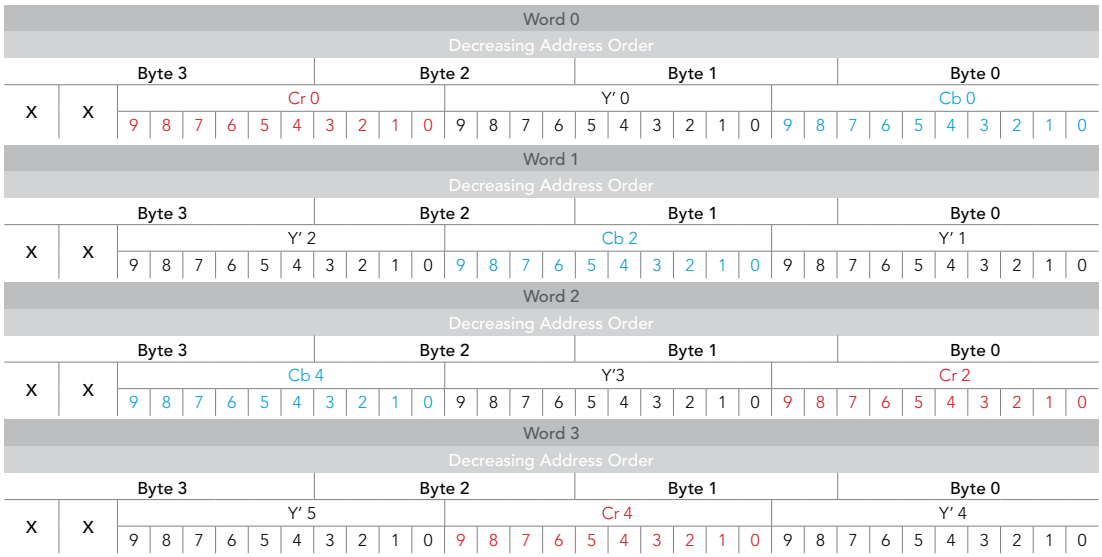


Figure 5.1: Data layout of a V210 group. Image courtesy of BlackMagic Design ®.

5.1 V210 structure

V210 is an interleaved 4:2:2 chroma-subsampled Y'CbCr format where each component has a precision of 10 bits. Three 10-bit components are packed in a single 32-bit little-endian word, where the last two bits are ignored. A group of four 32-bit words store 12 Y'CbCr components in the interleaved pattern $Cb_0, Y'_0, Cr_0, Y'_1, Cb_2, ..., Y'5$. Chroma values (CbCr) are sampled in half the resolution of luma (Y'). Each V210 group therefore stores 6 pixels. Figure 5.1 shows the V210 data layout.

The 4:2:2 chroma subsampling in V210 is *cosited*, e.g. `Cb2` and `Cr2` of the diagram in Figure 5.1 refer to the same spatial sample location as `Y'2`.

The V210 coding pads each image line to the next multiple of 128 bytes (48 pixels). The byte size $size_{v210}(w, h)$ of a video frame with width w and height h is therefore calculated as:

$$size_{v210}(w, h) = \left\lceil \frac{w}{48} \right\rceil * 128 * h \tag{5.1}$$

5.2 OpenGL V210 representation

The V210-encoded frames need to be transferred to and from the GPU using a natively supported OpenGL texture format before it can be processed by GLSL shaders. The transfer format should store the original data without loosing any precision: OpenGL 3 added native support for integer texture formats with a precision high enough to store the V210 frames. Integer textures are not normalized by OpenGL, i.e. the original code values of the texture elements (*texels*) can be accessed when sampling the texture in a shader.

Our framework stores V210 frames in OpenGL textures such that one texel represents a single V210 word. The framework implements two different approaches for storing V210 code words in 32-bit textures:

- `GL_RGB10_A2UI`: The components of a single V210 word (one line in the diagram of Figure 5.1) are extracted by the driver. When fetching a texel from the V210 texture in GLSL, the first three components (rgb) of the fetched `vec4` value directly represent the components of a single V210 word.

- `GL_R32UI`: A single V210 word is accessed by reading the first (`RED`) component of the fetched texel of the V210 texture. Extracting the 10-bit components needs to be done in the shader. This is done using the GLSL function `bitfieldExtract`. For inserting the V210 components back into a single 32-bit unsigned integer, the GLSL function `bitfieldInsert` is used. Bitfield operations were introduced in version 4.0 of GLSL.

While the former seems the be more convenient, the latter approach usually yields better performance. A comparison in performance between these two methods is given in Chapter 6. Our framework uses the OpenGL extension `ARB_internal_format_query2` to assert the optimal configuration of the OpenGL internal texture format and data type (see Chapter 2.3.4).

5.3 GLSL implementation

Our framework performs the conversion between V210 and RGB within a single OpenGL render pass using only GLSL shaders. The encoding and decoding process is implemented in three different versions of GLSL: 3.3, 4.2 and 4.3. Each version adds new features of the GLSL language and OpenGL API which allow more optimized implementations of the V210 transcoding process. The input and output data of the conversions are always the same:

- Decoder: Input is a texture containing a single V210 frame, output is an RGBA texture.
- Encoder: Input is an RGBA texture, output is a texture containing a V210 frame.

Each shader of the V210 encoders/decoders is also parameterized with the following options:

- **Flip origin (yes/no)**: OpenGL always operates with an image origin in the lower-left. If input/output data are represented using an upper-left origin, images need to be flipped during transcoding.
- **Chroma filter complexity**: For each shader, there are three levels of chroma filtering complexity available (described further below):
  - `NONE`
  - `BASIC`
  - `HIGH`
- **Extract/insert V210 components in shader (yes/no)**: Depending on the internal texture format of the V210 textures (see Chapter 5.2), this option is automatically enabled or disabled.
- **Convert to linearly coded RGB space (yes/no)**: This option is usually enabled, since rendering in linear space yields better results, but can be disabled for debugging purposes or performance measurements.

5.3.1 Chroma filters

Each shader variation of our V210 encoders/decoders is able to use filter kernels with a variable width for chroma decimation and reconstruction. Chapter 2.1.5 provides more background information on chroma filtering and the filter kernels that are used in this section. The previously described three levels of chroma filter complexity correspond to the following kernels:

- V210 encoder:
  - `NONE`: *drop*, i.e. no filtering is done, every second chroma sample is discarded.
  - `BASIC`: three-tap filter (tent) using the coefficients

$$[0.25, 0.5, 0.25]$$

  - `HIGH`: 13-tap filter as given by Poynton [3]:

$$[-0.00390625, 0.01171875, \\ -0.0234375, 0.046875, \\ -0.09375, 0.3125, \\ 0.5, \\ 0.3125, -0.09375, \\ 0.046875, -0.0234375, \\ 0.01171875, -0.00390625]$$

- V210 decoder:
  - `NONE`: *replicate*, i.e. neighboring chroma value is duplicated.
  - `BASIC`: two-tap filter (linear interpolation) using the coefficients

$$[0.5, 0.5]$$

  - `HIGH`: 12-tap filter as given by Poynton [2]:

$$[-0.0078125, 0.0234375, \\ -0.046875, 0.09375, \\ -0.1875, 0.625, \\ 0.625, -0.1875, \\ 0.09375, -0.046875, \\ 0.0234375, -0.0078125]$$

5.3.2 GLSL shader inclusion system

A lot of code is shared between the multiple variations of the shaders. Because GLSL does not natively provide a preprocessor command to include other sources, our framework provides a simple mechanism to include shader source files within other files. This is done by using the token `#fb_include<FILE.glsl>` in any shader. The framework's shader preprocessor looks for these token using C++11's `std::regex` library, parses the filename and looks for this file in a pre-configured location of GLSL sources. It then replaces the include token with the actual content of the referenced file. Indentation is done properly, recursive inclusion is possible and circular dependencies are avoided. This technique enables the framework to share existing code snippets between multiple shader variations.

5.3.3 Transcoding algorithm

All shader variations of the V210 encoders and decoders perform the same basic set of calculations. Let the *region of interest* (*ROI*) be the part of the image that is converted between the two formats in a single shader execution, then each shader variation performs the following steps with a single execution:

- Encoder:

    1. **Gather RGB ROI**: Fetch the RGB pixels covering the region of interest that should be encoded into V210. Also fetch enough neighboring pixels for applying the chosen decimation filter for chroma values later in the process.
    2. **Convert RGB to Y'CbCr**: Apply the inverse image transfer function to convert from RGB to R'G'B', then apply the 3x3 matrix to convert from R'G'B' to Y'CbCr (see Chapter 2.1.6).
    3. **Perform chroma subsampling**: Apply the chosen decimation filter on the previously converted Y'CbCr values in order to reach a 4:2:2 subsampling. Note that the filter only needs to be applied on those chroma values which are actually written out (every second chroma value).
    4. **Write out V210 ROI**: Write the resulting V210 groups covering the region of interest into the destination texture.

- Decoder:

    1. **Gather V210 ROI**: Fetch enough V210 groups covering the region of interest from the input texture sampler. Also fetch a neighborhood of V210 groups that contains enough chroma samples to accommodate the width of the chosen chroma reconstruction filter (see Chapter 5.3.1).
    2. **Apply chroma reconstruction filters**: Apply the reconstruction filters on the input chroma values in order to reconstruct the missing chroma values. After this step, we have a Y'CbCr triple for each spatial location of the output image (Y'CbCr *4:4:4* sampling).
    3. **Convert Y'CbCr to RGB**: Apply the 3x3 matrix to convert Y'CbCr to R'G'B' (see Chapter 2.1.6), then apply the image transfer function to convert perceptually encoded R'G'B' to linearly coded RGB.
    4. **Write out RGB ROI**: The resulting RGB values are written into the destination texture.

Figure 5.2 and 5.3 show the access patterns of the encoding and decoding process of a single V210 group (four 32-bit words). Table 5.1 shows the relation of dimensions and sizes between V210-encoded textures and RGBA textures.

5.3.4 GLSL 3.3

The encoding and decoding process in this version is implemented by a fragment shader that is active during the rendering of a quad. The quad's geometry covers the full range of the normalized device coordinates $[-1..1] \times [-1..1]$ in OpenGL. In order to execute the fragment shader for each element of the output texture, the dimension of the active viewport needs to be set to the dimension of the output texture using `glViewport`. The fragment shader derives the texel

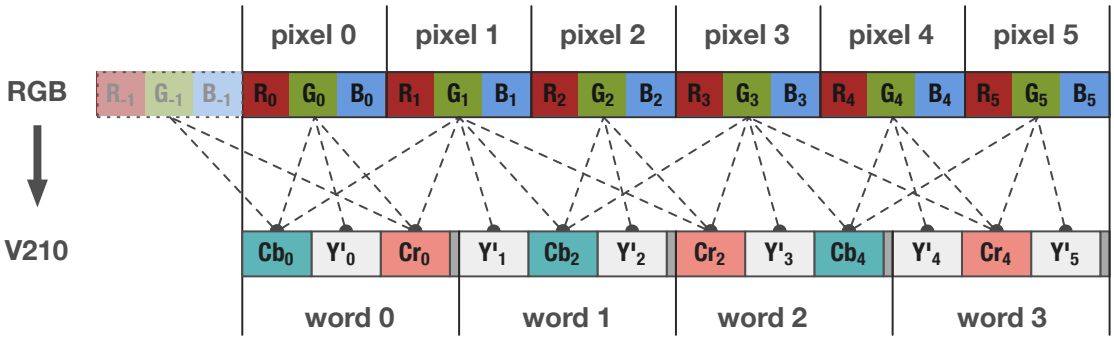


Figure 5.2: Access pattern for encoding six RGB pixels into a V210 group using a 3-tap decimation filter. Because Y'CbCr 4:2:2 encodes only half the chroma resolution, only even chroma samples need to be filtered and written out. This diagram uses a three-tap filter for chroma decimation.

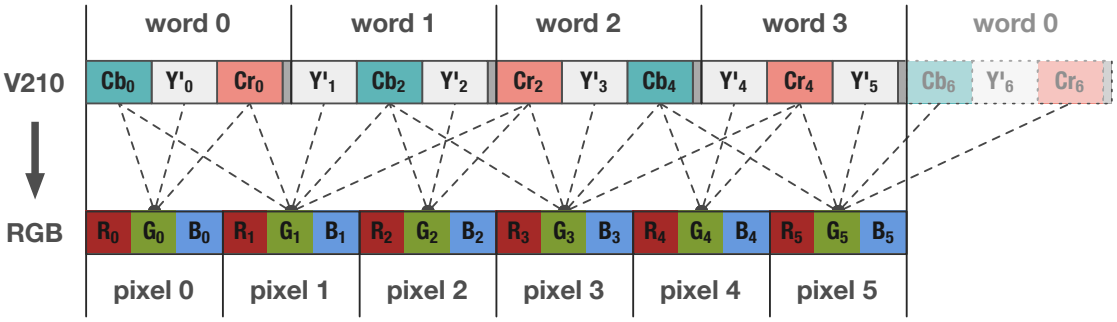


Figure 5.3: Access pattern for decoding a V210 group into a group of six RGB pixels using a 2-tap chroma reconstruction filter. Because V210 has *cosited* chroma samples, chroma filters need only be applied to odd pixels. This diagram uses a two-tap filter for chroma reconstruction.

location of the target texture using the built-in GLSL constant `gl_FragCoord` (see Figure 5.4). The fragment shader reads texture elements by using the GLSL command `texelFetch`. This command uses integer texture coordinates in order to fetch the texel at this exact location, bypassing any texture filtering. By default, `gl_FragCoord` returns the center of pixels in floating point coordinates (the lower-left corner would be $[0.5, 0.5]$ instead of $[0, 0]$). We can re-declare the constant in order to get the integer coordinates that we expect:

```
layout(pixel_center_integer) in vec4 gl_FragCoord;
```

If the image origin needs to be flipped, we change the signature again:

```
layout(pixel_center_integer, origin_upper_left) in vec4 gl_FragCoord;
```

In GLSL 3.3, the fragment shader can output only one pixel value per render target. We therefore have to execute the fragment shader for each texture element of the *destination* texture:

	V210 texture resolution	V210 texture size	RGBA texture resolution	RGBA 8-bit integer texture size	RGBA 16-bit float texture size	RGBA 32-bit float texture size
HD 1080p	1280x1080	5.5 MB	1920x1080	8.3 MB	16.6 MB	33.2 MB
UHD-1	2560x2160	22.1 MB	3840x2160	33.2 MB	66.3 MB	132.7 MB

Table 5.1: Resolutions and data sizes for V210 and RGBA textures. The V210 texture stores 6 RGB pixel values in 4 texels (V210 words). Therefore, the horizontal resolution of a V210-encoded texture is 2/3 of the RGB texture's width.

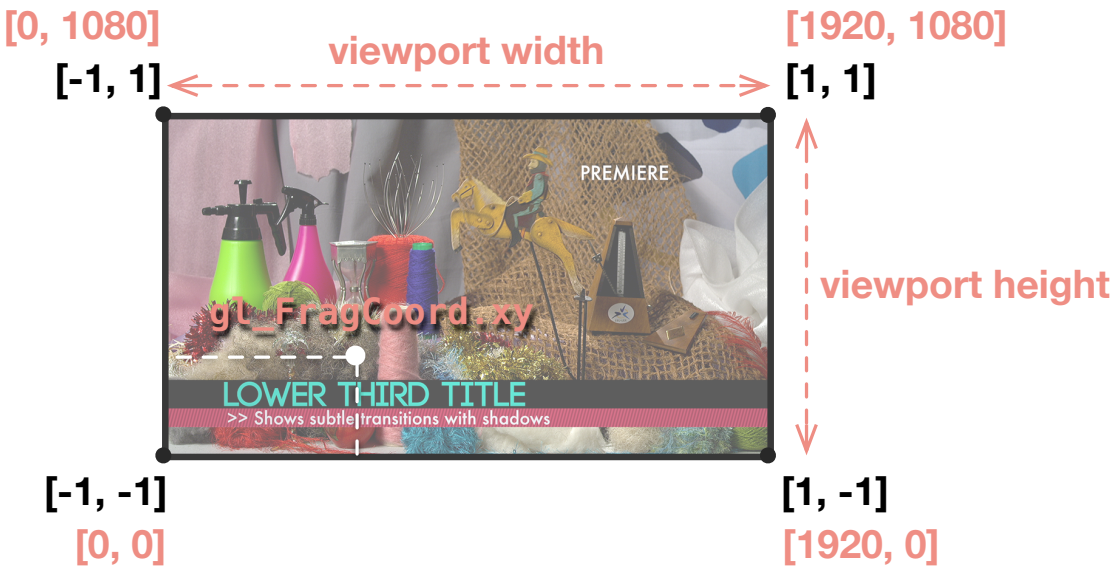


Figure 5.4: The fragment shader of the encoders and decoders is executed by rasterizing a quad that fully covers the range of the OpenGL *normalized device coordinates* (shown in black). In order to execute the fragment shader for each texel of the output texture, it is only necessary to set the viewport to the dimensions of the texture via glSetViewport. Texel locations of the currently executing fragment shader are derived by using the GLSL constant gl_FragCoord (shown in red).

The encoder executes its fragment shader per V210 output word, and the decoder executes its shader per output RGBA pixel (see Figure 5.5). Looking at the diagrams in Figure 5.2 and 5.3, we can see that the mapping between a single V210 word and an RGB pixel value is not simple, and that the constraint of running the fragment shaders for each output texel will either require the shader to use a lot of branching to distinguish between the different cases, or to accept some redundancy in reading more input data than actually necessary for the output texel. We chose the latter approach, e.g. the decoder always reads the full V210 neighborhood for every

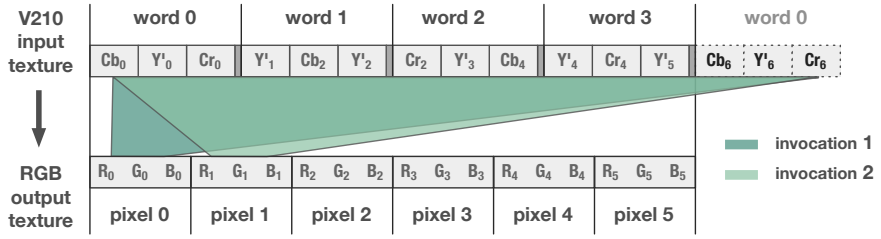


Figure 5.5: With the GLSL 3.3 implementation, the V210 decoder executes per output RGBA pixel and repeatedly has to read the same memory regions of the source image and perform a lot of redundant calculations.

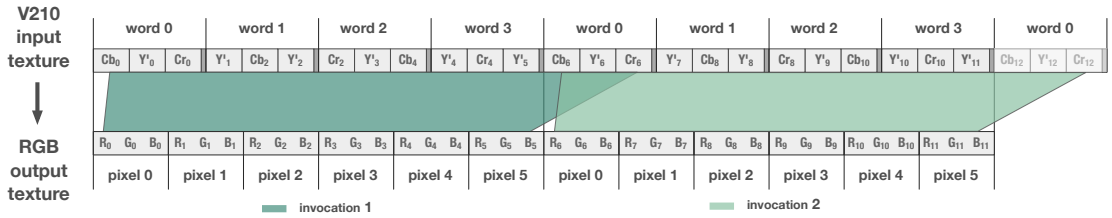


Figure 5.6: In GLSL 4.2, we can directly write 6 pixels for a single V210 group (4 bytes). This provides a very efficient data mapping. The input data region in this diagram is slightly larger in order to accomodate the neighborhood for the chroma filter.

output pixel, even though a V210 group actually covers six output pixels (see Figure 5.5). While functionally correct, this implementation's performance is suboptimal. By taking advantage of newer OpenGL features, we can improve the algorithm's performance drastically, as described by the next chapter.

5.3.5 GLSL 4.2

OpenGL 4.2 introduced a new method for shaders to write to *images* [1] via the GLSL function `imageStore` (see Chapter 2.3). A fragment shader is now able to write multiple values to a texture within a single invocation. Using this feature, the V210 encoder is able to output a complete V210 group of four 32-bit integer words for each input of six RGB pixels, and the V210 decoder is able to output six RGB pixels for each input of four V210 words (a complete V210 group, see Figure 5.6). When rendering the unit quad, we therefore set the width of the active viewport to the number of V210 groups (i.e. pixel width divided by six) instead of the number of output pixels. This results in less invocations of the fragment shader compared to the previous GLSL 3.3 implementation. For example, for a width of 1920 pixels, the viewport's width is set to 320, the number of V210 groups necessary to store 1920 pixel values. The

[1] An image in this context represents a single level of a texture.

fragment shader is therefore executed only 320 times for each line in the image, as opposed to 1920 times using the GLSL 3.3 implementation.

Because the GLSL image load/store operations operate under a relaxed memory model, after executing the fragment shader and writing into the destination image, a *barrier* has to be inserted by the host program in order to make these writes visible to the next operation. In our pipeline, the decoder of the stage *ConvertFormat (1)* insert a fence by calling the GL command `glMemoryBarrier(GL_TEXTURE_FETCH_BARRIER_BIT)`, because its output image will be used for texture reads while executing the *Render* stage. The encoder of *ConvertFormat (2)* insert a barrier using `glMemoryBarrier(GL_PIXEL_BUFFER_BARRIER_BIT)`, because the *PackPBO* stage will use its output image directly for pixel buffer transfers.

As opposed to the GLSL 3.3 implementation, the actual return value of the fragment shader is irrelevant, because all output values are written to the destination image via `imageStore`. However, in unextended OpenGL 4.2 we still have to attach a texture to the FBO that is bound while rendering the quad. Because we can not bind the same texture to the FBO that we write to via `imageStore`, we have to allocate and attach dummy textures, which is a waste of resources. If the extension `ARB_framebuffer_no_attachments`[2] is available, we can bind an FBO without attachments, which removes the necessity of using dummy textures. This also enables the OpenGL driver to ignore the output of the fragment shader when no texture is attached and saves the bandwidth of writing back the fragment shader's result to the framebuffer.

A complete listing of the GLSL 4.2 V210 decoder is given in Appendix A.

5.3.6 GLSL 4.3 compute shaders

While the previous two implementations rely on OpenGL's rasterizer to invoke the fragment shader, OpenGL 4.3 enables the host program to execute a shader independently of the traditional graphics pipeline. These shaders are called *compute shaders*. The principles of compute shaders are explained in Chapter 2.3.5.

Similar to the GLSL 4.2 variation, a single thread of the compute shader implementation operates on a single V210 group, i.e. four 32-bit words. However, these invocations are now grouped into several *work groups*, within they are able to share data. In our implementation, the size of a work group is defined as $[x_{v210}, 1, 1]$. The variable x_{v210} defines the number of V210 groups along a line of the image that should be processed within one work group. Our framework enables the user to tweak this number for optimizing compute shader executions. Chapter 6 shows results of how much the group size influences the overall performance on different architectures.

Because the shader invocations within a work group are able to share data, we can split the execution of our conversion algorithm into two phases:

1. Read input data for the complete work group and store in *shared memory*.

2. Read neighborhood from shared memory, convert to destination format, and store result in image.

[2] This extension is part of the OpenGL 4.3 core specification.

```
...

const int tile_v210_width = TILE_V210_WIDTH;

const int neighborhood_v210_size = (tile_v210_width +
    chroma_filter_v210_width);
const int neighborhood_pixel_size = neighborhood_v210_size * 6;

layout(binding = 0) uniform usampler2D v210_input_image;
layout(binding = 1) uniform restrict writeonly image2D rgba_output_image;

// Define the work group size
layout(local_size_x=TILE_V210_WIDTH) in;

// Shared memory for input luma and chroma
shared uint luma[neighborhood_pixel_size];
shared uvec2 chroma[neighborhood_pixel_size/2];

void main() {

    // Determine src/dst coordinates from
    // compute shader's built-in constants
    const ivec2 tile_xy = ivec2(gl_WorkGroupID);
    const uint thread_x = gl_LocalInvocationID.x;
    const ivec2 v210_coords = tile_xy*ivec2(tile_v210_width, 1) + ivec2(
        thread_x, 0);
    const int v210_y_line_num = v210_coords.y;
    const uint x = thread_x;

    const uvec2 rgba_pixel_base_coords = uvec2(v210_coords.x * 6,
        v210_coords.y);

    // We might have had to spawn more threads than which are actually
    // contributing to the image
    if (rgba_pixel_base_coords.x < ((v210_image_size.x/4)*6)) {

        // Phase 1: Read-in & decompose luma and chroma from V210 words
        for (int i=0; i < neighborhood_v210_size; i += tile_v210_width) {

            if ( (x + i) < neighborhood_v210_size) {

                // Extract Y'CbCr and store into shared memory
                ...
            }
        }

        // Barriers for synchronization threads and shared memory access
        memoryBarrierShared();
        barrier();

        // Phase 2:
        //     ] Apply chroma reconstruction filterconvert to RGB
        //     ] Convert to RGB
        //     ] Write results to rgba_output_image

        ...
    }
}
```

Listing 5.1: Skeleton of a GLSL 4.3 V210 decoder implementation using shared memory. Implementation is based on example code given by Kilgard [10]. Note that source code was skipped for the sake of brevity.

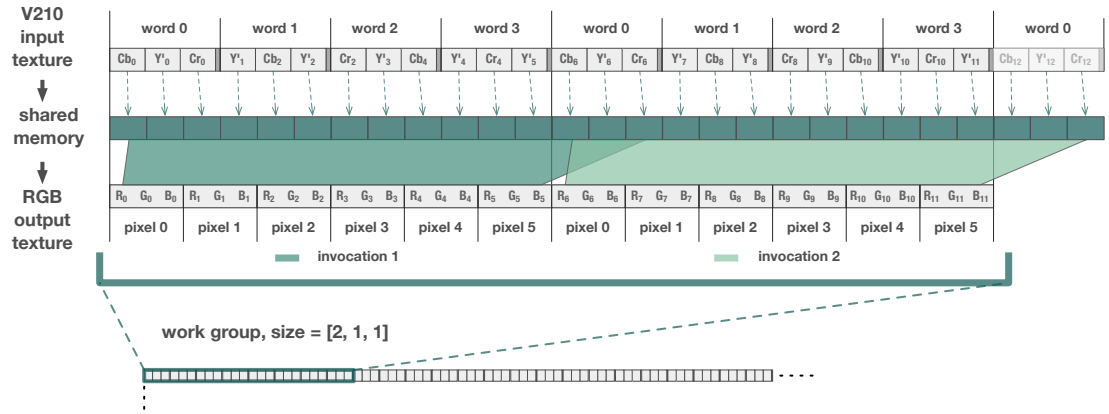


Figure 5.7: Using compute shaders, we can cache input data in shared memory. The data is then read by each thread of a work group. This requires less redundant texel-fetches when reading the neighborhood for chroma filtering. This diagram shows show a workgroup size of 2. Notice how the overlapping reads of the first and second invocation are now read from shared memory instead of fetched again from the texture unit.

This has the advantage that the neighborhood of chroma filter kernels only needs to be fetched from the texture once. Figure 5.7 shows the structure of a single work group. Work groups are distributed over each line of the image. A sensible work group size would be $[64, 1, 1]$. For an image of a pixel resolution of $[1920, 1080]$ where each row is stored in 320 V210 groups, this results in 5 work group executions for each line.

The skeleton of a GLSL 4.3 V210 decoder is shown in Listing 5.1:

- Line 8 and 9 declare the input texture and output image.
- The work group size is declared in line 12. `TILE_V210_WIDTH` is a constant that is defined at compile-time and denotes the number of threads within a work group. Each thread a work group operates on a full V210 group of four words.
- Shared memory is declared in line 15 and 16. The size of these arrays is defined as the number of V210 groups to process within this work group, plus the neighborhood that is necessary for the chosen chroma filter size.
- Each thread that enters the compute shader converts a single V210 group into six RGB pixels. The address and texture coordinates of this group is determined in line 22 and following. This is done using the GLSL built-in constants that are available to compute shaders.
- Depending on whether the chosen work group size is a multiple of the number of V210 groups available, there might be threads spawned which actually do not contribute to the image. These threads are excluded line 32.

- Line 35 and following reads in the neighborhood of the work group. Each thread reads at least one texture element. Some threads read more values in order to fetch the texture elements of the filter neighborhood.
- Line 45 insert a barrier to make sure that previous writes into shared memory are visible to all threads.
- Line 46 inserts a sync point between all threads of this work group, i.e. execution of the current threads blocks until all threads of this work group have entered the barrier.
- After the barrier is passed, each thread reads the Y'CbCr neighborhood of its assigned V210 group, converts it into RGB and writes the six resulting RGB pixels into the destination image.

5.3.6.1 Bypassing shared memory

In our implementation, the use of shared memory eliminates redundant reads of the neighborhood for chroma filters. This is particularly effective for very large filter kernels. However, because those texture reads are spatially coherent, they are likely to be cached by the texture units. It is therefore questionable whether the benefits of using shared memory outweigh the shader complexity of its implementation and the synchronization of inter-thread access. We have therefore also implemented a variation of our compute shader encoder/decoder which does not use shared memory. Instead, each thread individually reads all necessary input data and directly moves on to converting and writing out the results. This version is very similar to the GLSL 4.2 implementation, except that the shaders are not invoked by the rasterizer but directly by the host program's compute shader dispatch. Chapter 6 provides performance charts of both approaches.

5.3.7 Testing GLSL variations

During development, we made sure that each transcoder GLSL variation is functionally equivalent to the others, i.e., for the same configuration of chroma filters and render formats, the output has to be consistent across each GLSL implementation, only the performance characteristics should vary. The transcoders were cross-tested repeatedly for correctness (also see Chapter 4.8).

CHAPTER 6

Results

In this chapter, we show the results of testing the prototype implementation of the framework. We begin by describing the test setup and how the *FrameBender* application was used to provide performance data and quality measurements. We then describe the results of running different scenarios that influence performance and quality.

6.1 Test setup

We described a mechanism of our framework to capture performance profiles of particular configurations in Chapter 4.6 and 4.7. In this chapter, we employ this mechanism to measure the execution of the pipeline using different configurations. These configurations are described in Chapter 6.1.1. For each tested configuration, we ran the *FrameBender* application to execute a benchmarking scenario (Chapter 6.1.2) and analyzed the resulting performance trace files. In Chapter 6.1.3, we show our method to visualize the data that was captured. We further describe the statistical properties that can be extracted from a single trace of a benchmark run (Chapter 6.1.7) and explain how we measure the potential loss of image quality in our pipeline (Chapter 6.1.8).

6.1.1 Parameter space of benchmark configurations

Figure 6.1 provides an overview of the complete parameter space over which we define the configurations that are used for benchmarking. In the following, we provide additional information for each configuration parameter:

A **Device**: This parameter defines the device that was used for benchmarking with our test machine (see Chapter 6.1.5). The following devices were tested:

- NVIDIA Quadro K5000, 4 GiB GDDR5 VRAM, Driver ver. 320.27
- NVIDIA GeForce GTX 680, 4GiB GDDR5 VRAM, Driver ver. 320.18

A Device		**1** AMD FirePro W9000 (PCIe Gen-3)
		2 NVIDIA GeForce GTX 680 (PCIe Gen-3)
		3 NVIDIA Quadro 6000 (PCIe Gen-2)
		4 NVIDIA Quadro K5000 (PCIe Gen-2)
		5 NVIDIA Quadro K5000 (PCIe Gen-3)
B Pipeline depth		**1** Low latency
		2 High latency
		3 High latency + (un)map <-> (un)pack interleaved
		4 High latency + transfer <-> render interleaved
		5 High latency + (un)map <-> (un)pack & transfer <-> render interleaved
C Test sequences		**1** EBU Horse V210 HD 1080p50 (1920x1080)
		2 EBU Rain fruits V210 UHD-1p50 (3840x2160)
D Pipeline concurrency		**1** Single thread, single OpenGL thread + context
		2 Async host copies, single OpenGL thread + context
		3 Async host copies & AMD pinned memory, single OpenGL thread + context
		4 Async host copies, multiple OpenGL threads + contexts
		5 Async host copies & AMD pinned memory, multiple OpenGL threads + contexts
		6 Local best mode (AMD: 3, NVIDIA: 4)
E Active stages		**1** GPU upload
		2 GPU download
		3 GPU upload & download
		4 GPU upload & download & CPU host copying
		5 GPU upload & download & render
		6 GPU upload & download & render & CPU host copying (all)
F Render format		**1** RGBA 8-bit integer
		2 RGBA 16-bit integer
		3 RGBA 16-bit float
		4 RGBA 32-bit float
G V210 settings	**a** GLSL mode	**1** GLSL 3.3
		2 GLSL 4.2
		3 GLSL 4.2 & ARB_framebuffer_no_attachments
		4 GLSL 4.3 & caching reads in shared memory
		5 GLSL 4.3 & no shared memory
		6 Local best mode (AMD: 2, NVIDIA: 3)
	b Chroma filter	**1** None
		2 Basic
		3 High
	c Work group size	**1** 16
		2 32
		4 64
		5 96
		6 128
	d V210 mode	**1** RGB10_A2UI texture
		2 R32UI texture + shader bitops

Figure 6.1: This table shows the parameter space of the tests that were executed. Highlighted parameters denote values that also affect image quality, others influence performance only. Each configuration is uniquely identified by a tuple like $[A1, B1, C1, D1, E6, F3, G_{a2,b2,d2}]$.

- NVIDIA Quadro 6000, 6 GiB GDDR5 VRAM, Driver ver. 320.27
- AMD FirePro W9000, 6 GiB GDDR5 VRAM, Driver ver. 12.104.2.0

Note that not all of these devices take advantage of PCIe Gen-3. The Quadro 6000 is limited to PCIe Gen-2 by hardware. Even though the hardware of the Quadro K5000 is able to use PCIe Gen-3 transfers, the NVIDIA driver[1] disables PCIe Gen-3 by default in order to avoid compatibility issues with some motherboards. However, NVIDIA provides a patch to explicitly enable PCIe Gen-3 support[2]. We have tested the Quadro K5000 both with the default PCIe Gen-2 configuration and the patched PCIe Gen-3 support, hence it is listed twice in the device parameter values.

B **Pipeline depth**: Each configuration is a preset of queue sizes and interleaving settings. The following explains the effect of the used terms:

- **Low latency**: The queue sizes between stages are kept as small as possible (max. 2 elements per queue). This configuration never uses *single-threaded interleaving* (see Chapter 4.2.3.2). The goal of this configuration is to keep the latency low.
- **High latency**: The queue sizes between stages are larger so as to increase the potential of concurrency and interleaving (max. 8 elements per queue). This results in higher latency.
- **(Un)map <-> (un)pack interleaved**: The stages *UnpackPBO* and *MapPBO* use an *input constraint* of 2, i.e. their single-threaded execution is *interleaved* with the execution of their predecessors *UnmapPBO* and *PackPBO* (see Chapter 4.2.3.2). The goal here is to break the implicit dependency between the the pack and map operations.
- **Transfer <-> render interleaved**: Similar to above, the goal of this setting is to interleave the transfer-related stages with the rendering-related stages (this includes format conversions). Using this setting, stage *ConvertFormat (1)* uses an input constraint of 2, i.e. it is interleaved with stage *UnpackPBO*, and stage *PackPBO* uses an input constraint of 2 as well, it is interleaved with *ConvertFormat (2)*.
- The above two interleaving options can be combined as well.

C **Test sequences**: See Chapter 6.1.4.

D **Pipeline concurrency**: The listed possible values relate to the degree of concurrency that has been described in Chapter 4.3. Settings using AMD pinned memory are limited to AMD hardware and not tested by NVIDIA hardware. The *local best mode* option shown in the figure uses the settings that performed best for the chosen hardware. This setting is used when test results are compared between different hardware configurations and when this setting is not the focus of the comparison.

[1] As of NVIDIA ForceWare 320.27

[2] http://nvidia.custhelp.com/app/answers/detail/a_id/3135/~/geforce-600-series-gen3-support-on-x79-platform

E **Active stages**: These configurations allow isolating certain stage executions. They represent a preset of the debug configurations that were described previously in Chapter 4.7. We use the same terminology as in Chapter 4.7:

- **GPU upload**: Bypass output, bypass render stages and disable host-side copying, i.e. only GPU upload is performed.
- **GPU download**: Bypass input, bypass render stages and disable host-side copying, i.e. only GPU download is performed.
- **GPU upload & download**: Bypass render stages and disable host-side copying, i.e. GPU upload and download is performed.
- **GPU upload & download & CPU host copying**: Bypass render stages, i.e. up/-download including host-side copying is performed.
- **GPU upload & download & render**: Disable host-side copying, i.e. only GPU tasks are performed.
- **GPU upload & download & render & CPU host copying**: Disable nothing, execute complete pipeline. This is the only configuration that provides correct rendering results. All of the above are for debugging purposes only.

F **Render format**: Configures the *canonical render format* (see Chapter 4.4).

G **V210 settings**: These are the parameters of the V210 transcoding process (as implemented by the *ConvertFormat* stages).

a **GLSL mode**: These values map to the variation of the GLSL implementation as discussed in Chapter 5.3:

GLSL 3.3: See Chapter 5.3.4.

GLSL 4.2: This variation uses only OpenGL 4.2 core profile features, i.e. it has to allocate and attach dummy textures to FBOs (see Chapter 5.3.5).

GLSL 4.2 & ARB_framebuffer_no_attachments: This variation takes advantage of the extension `ARB_framebuffer_no_attachments` as discussed in Chapter 5.3.5.

GLSL 4.3 & caching reads in shared memory: This compute-shader variation caches the input reads for a complete workgroup in shared memory (see Chapter 5.3.6).

GLSL 4.3 & no shared memory: This compute-shader variation does not use shared memory for caching (see Chapter 5.3.6.1).

Local best mode: This option uses the best-performing mode for the chosen graphics hardware.

b **Chroma filter**: Maps to the chroma filter kernels as described in Chapter 5.3.1.

c **Work group size**: If a compute shader is used as the *GLSL mode*, this setting configures the used work group size (see Chapter 5.3.6). This setting has no effect on other GLSL modes.

d **V210 mode**: Configures the approach to store and access V210-encoded video frames (see Chapter 5.2):

6.1.1.1 Notation used for configuration identifiers

In order to uniquely identify the run configurations that are compared in our diagrams, we use a tuple of mappings between letters and numbers which refer to the entries of Figure 6.1. For example, $[A1, B1, C1, D1, E6, F3, G_{a2,b2,d2}]$ defines using the FirePro device to render HD test sequences in low-latency with a single thread using a 16-bit float render format and performing V210 transcoding using GLSL 4.2 with a basic chroma filter where V210 10-bit extraction and insertion is performed by shader bit operations.

We use a simple notation to describe the varying parameters of a chart:

$*$: The chart varies over all options for a parameter, e.g. $[\boldsymbol{A*}, B1, C1, D1, E6, F3, G_{a2,b2,d2}]$ varies over all tested devices, i.e., we want to compare how different devices perform in this scenario.

$\boldsymbol{x|y|...|z}$: The chart uses a subset of options for a parameter, e.g., $[\boldsymbol{A\{4|5\}}, B1, C1, D1, E6, F3, G_{a2,b2,d2}]$ shows how the Quadro K5000 PCIe Gen-2 compares to the Quadro K5000 PCIe Gen-3 for the same set of parameters.

Parameters which are of particular interest to the shown scenario are printed in bold face.

6.1.1.2 Common settings

While the parameters described above might vary between benchmark executions, the following settings are common to every test run that was used for capturing:

- The player window is disabled, i.e. previewing the rendered result should not influence the benchmarking data.
- The OpenGL context is not a debug-configured context. This results in slightly improved performance of OpenGL execution.

The binaries of the *FrameBender* executable and *libFrameBender* library that were used for benchmarking were compiled with Visual Studio 2012 in Release mode using the x64 compiler, i.e. using optimized code without debugging information.

6.1.2 Benchmarking scenario

For each test run, the *FrameBender* application performs the following tasks for benchmarking:

1. Load pipeline parameters (see Chapter 6.1.1)
2. Pre-fetch ~1 GB of V210 frames (200 1080p frames or 50 UHD-1 frames).
3. Enable CPU-based and GPU-based sampling.

4. Execute the video processing pipeline while looping ten times over this set of frames, i.e., render 2000 frames. For rendering, the demo renderer (see Chapter 4.5) is used.

5. Write the captured traces to a file.

The pre-loading of the frames excludes I/O operations of the hard disk from the benchmarking process. Looping the video processing pipeline over this set allows executing the pipeline over a larger number of frames without occupying a very large region of memory, which could have side-effects of its own. In a real-world application, this management of loading frames would be handled asynchronously by the acquire and deliver stages of the pipeline. However, for the purpose of this benchmark, the acquisition and delivery of frames is ignored, i.e., only the actual processing pipeline should be profiled.

For each test run, parameters of the pipeline configuration vary in order to evaluate their influence (see Chapter 6.1.1). In order to efficiently manage the benchmark execution, we have written a Python script that automates the process of launching the *FrameBender* application with the desired configurations. This is particularly convenient when the benchmarking process needs to be repeated for different hardware, or a re-run is necessary because of changes in the implementation of the pipeline. The script performs the following operations for each test configuration and for each tested graphics hardware device:

1. Assign a unique ID to the combination of parameters chosen for this configuration.

2. Create a folder with the unique ID of this run.

3. Assemble a human-readable string from the combination of parameters.

4. Run the benchmarking scenario using verbose logging and full debug parameters (i.e. OpenGL debug contexts). Dump all configuration parameters to a file. Store this file and the log file in the folder of this run.

5. Run the benchmarking scenario with normal logging and no debug parameters (i.e. normal OpenGL contexts). Store the trace file along with the used configuration and log file in the folder of this run.

Running the benchmarking with and without debugging parameters allows analyzing the OpenGL debug output (see Chapter 2.3.4) of a particular run, if necessary, while still capturing the trace of a non-debug configuration.

After all test runs are completed, the resulting trace files are automatically processed by the Python script for trace visualization (see Chapter 6.1.3). After this processing, a summarizing PDF of the CPU-sampled and GPU-sampled pipeline execution is available.

Each test session contains about 100 different test configurations and ran on five different hardware setups (see Chapter 6.1.1). In order to overview the captured data, we wrote another Python script that extracts summarizing statistics of all performed benchmarks and writes them into a single CSV table along with the configuration-ID and the human-readable string. This allowed us to quickly overview the data and to pick out interesting results.

6.1.3 Visualization of traces

In order to visualize the execution times of our pipeline, we have written a Python script that opens the captured trace file (see Chapter 4.6.2) and draws an execution graph using the *Cairo* vector drawing library [1]. Figure 6.2 shows an example and explains the components of the generated diagram.

6.1.4 Input sequences

For the input sequences of our testing scenarios, we have used two uncompressed standard test sequences:

1. V210-encoded 1080p50 EBU Horse sequence
2. V210-encoded UHD-1 (3840x2160) 50 hz EBU Rain Fruits sequence

Both sequences are provided by the European Broadcasting Union (*EBU*) and are publicly available [4]. Both sequences use a *gamma* conforming to ITU-R BT.709 (see Chapter 2.1.2.1). The UHD-1 sequence was originally stored in the DPX format in RGB 4:4:4 10-bit, which we transcoded to V210 separately using the *avconv* [21] tool. The HD sequence was already encoded in 10-bit 4:2:2 Y'CbCr. However, the container was *yuv10*, a format similar to V210 but storing the 10-bit components in *big-endian* words instead of *little-endian* [3]. We wrote our own converter tool that converts *yuv10* to V210, without losing any of its original information. Both sequences contain values in the head and footroom of the Y'CbCr encoding (see Chapter 2.1.6), which is common to real-world high-quality captured camera sequences.

6.1.5 Test machine specification

The following machine has been used for all configurations:

CPU: Intel® Xeon® CPU E3-1240 V2 @ 3.40GHz, 4 Cores, 8 Logical Processors

Mainboard: Asus P8C WS (incl. PCIe Gen-3 support)

RAM: DDR3 16 GiB (4x4 modules), 1600 MHz, Quad Channel, Corsair Vengeance Low Profile

OS: Windows 7 Professional 64-bit SP1

6.1.6 Limitations

Certain graphics card drivers caused limitations in the execution of some benchmarking configurations (see Chapter 6.1.1). The following is a list of these limitations:

[3] Ffmpeg/avconv support this format as the *v210x* codec. However, the transcoding of ffmpeg from v210x to v210 seems to lose some coding information in the process, therefore we have decided to write our own converter tool for this.

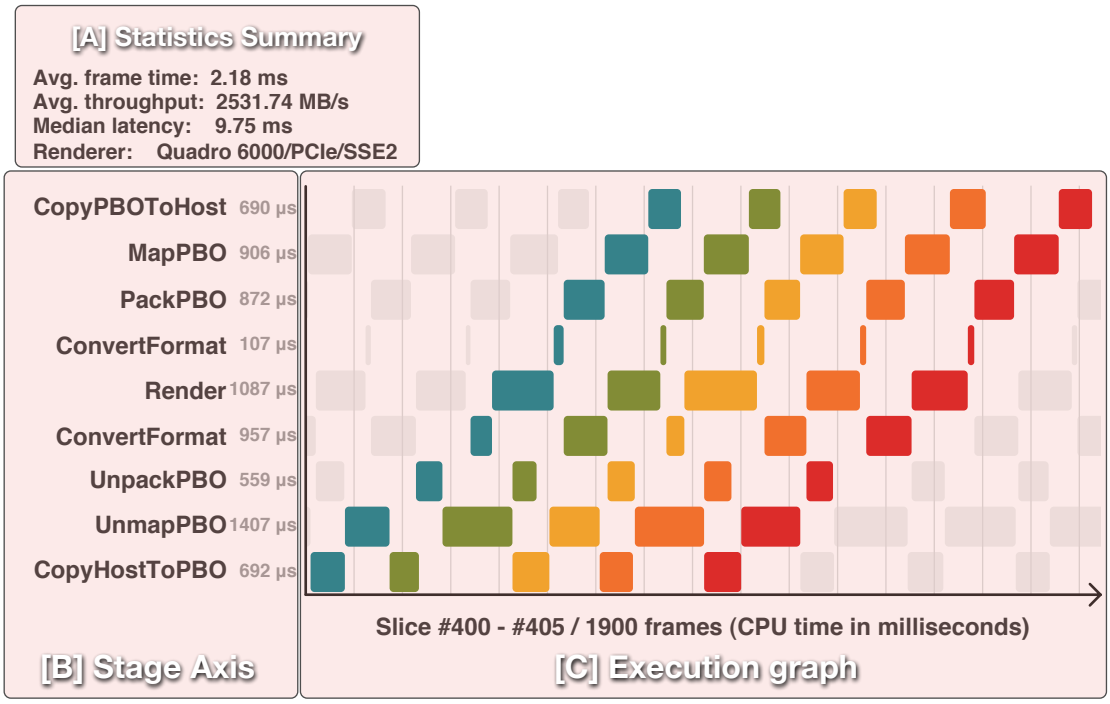


Figure 6.2: Visualization of processing 8 frames in our pipeline. The upper-left (*A*) shows a summary of the statistical markers, from top to bottom: average processing time per frame (see Chapter 6.1.7.2), average data throughput per second (see Chapter 6.1.7.3), median frame latency (see Chapter 6.1.7.4). It also includes the name of the OpenGL renderer to identify the tested GPU. On the left side (*B*), each stage of the pipeline is listed. Stage *Acquire* and *Deliver* are hidden for brevity. In our benchmarking scenarios, they only use a negligible amount of processing time. Stage execution order is upwards, i.e. the first relevant stage is *CopyHostToPbo*, the last is *CopyPBOToHost*. The median of stage execution time per stage (see Chapter 6.1.7.1) is shown on the right side of the stage name in microseconds in block *B*. In the middle of the graph (*C*), the traces for 8 frames are visualized. The horizontal axis represents processing time in milliseconds. Each drawn box visualizes the processing time of a single stage execution for a single frame, i.e. overlapping boxes denote concurrent stage executions. Individual frames use exchanging colors, i.e. in this diagram frame #400 is blue, frame #401 is green, etc. This diagram focuses on a *slice* of pipeline executions. Stage executions neighboring the focused area are drawn in grey. Grey vertical lines in the execution diagram show ticks of milliseconds. These diagrams are available for CPU-based sampling as well as GPU-based sampling. For GPU-based sampling, the stage execution time has been calculated using the approach described in Chapter 6.1.7.1.

Figure 6.3: UHD-1 (left) and HD (right) test sequences as provided by the EBU. Both sequences represent high-quality raw video captured from professional camera equipment.

- AMD Catalyst driver ver. 12.104.2.0:
  - Compute-shader support did not work as expected and crashed the OpenGL driver. Therefore, all GLSL 4.3-related tests were not executed.
  - The OpenGL 4.3 extension ARB_framebuffer_no_attachments is not available, i.e. for all AMD V210 tests, only the GLSL modes G_{a1} and G_{a2} were available.
  - The implementation for the OpenGL texture format GL_RGB10_A2UI is broken, i.e. for all AMD tests, only G_{d2} was tested.
  - The shader compiler of the AMD OpenGL implementation falsely reports an error when more than one memory qualifier is used for GLSL image2D uniform declarations. For all AMD tests, only the memory qualifier `writeonly` is used instead of `restrict writeonly` (as used when testing NVIDIA cards).
- NVIDIA ForceWare 320.27 / 320.18:
  - When using multiple GL contexts executed on multiple threads in order to achieve GPU-side overlap of executions, OpenGL timer queries were only allowed to be executed for the main context (the one executing *ConvertFormat (1)*, *Render*, *ConvertFormat (2)*). If timer queries were used in all contexts, the driver would report a performance warning that a fallback to serialized pixel transfers and rendering had occurred. The goal of observing a GPU-side overlap in our own captured GPU traces could therefore not be achieved.
  - AMD pinned memory extension is only supported on AMD hardware, therefore configurations using pinned memory were skipped when running NVIDIA hardware.

The drivers used here were the most recent driver available at the time of writing in 07/2013. Newer driver versions might have fixed these problems.

6.1.7 Statistical properties of traces

In the following, we describe the statistical properties that we calculate from the output traces of the executed benchmarking scenarios (see Chapter 6.1.2).

6.1.7.1 Stage execution time

Let $T_{CPUbegin}(i, j)$ be the the time in seconds at which the state `StageExecutionState::TASK_BEGIN` of the ith stage was sampled in CPU time for the jth frame, and let $T_{CPUend}(i, j)$ be the time in seconds at which `StageExecutionState::TASK_END` was captured.

We then calculate $D_{CPU}(i, j)$ as the time in seconds in which a single CPU core is busy executing the ith stage for the jth frame:

$$D_{CPU}(i, j) = T_{CPUend}(i, j) - T_{CPUbegin}(i, j) \tag{6.1}$$

Let $T_{GPUbegin}(i, j)$ be the result of an OpenGL timer query issued at the beginning of executing the ith stage for the jth frame. This value is stored in the trace file as the state `StageExecutionState::GL_TASK_BEGIN` of the ith stage for the jth frame. Let further be $T_{GPUend}(i, j)$ the result of an OpenGL timer query issued at the end of executing the ith stage for the jth frame. This value is stored in the trace file as the state `StageExecutionState::GL_TASK_END` of the ith stage for the jth frame. We then calculate $D_{GPU}(i, j)$ as the time in seconds in which the GPU is busy while executing the OpenGL commands that were issued by the ith stage for the jth frame:

$$D_{GPU}(i, j) = T_{GPUend}(i, j) - T_{GPUbegin}(i, j) \tag{6.2}$$

We use the median of the above two properties to describe the long-term processing time of the stage executions for CPU-based and GPU-based executions.

6.1.7.2 Average CPU processing time per frame

Let $i = 1$ denote the first stage *Acquire* (see Chapter 4.2.1.1) and $i = N$ denotes the last stage *Deliver* (see Chapter 4.2.1.11). Let M be the total number of frames processed. Then P_{avg}, the average processing time in seconds per frame, is defined as

$$P_{avg} = \frac{T_{CPUend}(N, M) - T_{CPUbegin}(1, 1)}{M} \tag{6.3}$$

6.1.7.3 Average data throughput

Let S_{v210} be the size of a single V210 video frame in Megabytes, then the average throughput D_{avg} (MB/sec) of a single session is calculated as

$$D_{avg} = \frac{S_{v210}}{P_{avg}} \tag{6.4}$$

Historically, a *Kilobyte* was calculated as 2^{10} bytes. As a result, a *Megabyte* was calculated as 2^{10} *Kilobytes*, i.e. 2^{20} bytes. This, however, is an incorrect use of the SI unit *Mega*. In 1999, the IEC therefore defined one Megabyte (MB) as exactly one million bytes (1 MB = 1 000 000 bytes) [12]. For the representation of 2^{20} bytes, the *binary prefix* Mebibyte (MiB) should to be used instead. The advertising of *Megabytes* which actually mean Mebibytes caused a lot of confusion among IT professionals and computer scientists [37]. For example, hardware RAM and cache sizes are labeled as MB, but actually mean MiB. On the other hand, the bandwidth of the PCI-Express bus is calculated using the SI prefix (see calculation in Chapter 2.2.1). In order to correctly express the amount of actual bytes transferred, we use the SI unit Megabyte, i.e. in this thesis, MB/sec means exactly 1 million bytes per second.

6.1.7.4 Latency

We further calculate the *latency* $L(j)$ of the jth frame as follows (also see Figure 3.6):

$$L(j) = T_{CPUend}(N, j) - T_{CPUbegin}(1, j) \tag{6.5}$$

We define L_{med} as the median of $L(j)$ over all captured frame traces. This is used as a marker for the frame latency within the pipeline.

6.1.8 Measuring image quality using PSNR

Using a high-quality set of input sequences, we are interested if and how much a roundtrip of our format converters of Y'CbCr to RGB to Y'CbCr corrupts the original image quality. We can measure this by using a pass-through renderer (i.e. no image overlays) and by calculating the peak-signal-to-noise-ratio (*PSNR*) of the output image to the input image. The PSNR describes the ratio in power of the clean unmodified image signal to the noise signal that has been introduced by some process. It is defined in decibel and calculated as follows:

$$MSE = \frac{1}{m * n} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i,j) - O(i,j)]^2 \tag{6.6}$$

$$RMSE = \sqrt{MSE} \tag{6.7}$$

$$PSNR = 20 * log_{10} \left(\frac{MAX_I}{RMSE} \right) \tag{6.8}$$

$$= 10 * log_{10} \left(\frac{MAX_I^2}{MSE} \right) \tag{6.9}$$

MAX_I represents the maximum possible component value. MSE is defined as the mean-squared-error, where $I(i,j)$ represents the input image, $O(i,j)$ represents the output image, and $[m,n]$ is the pixel-resolution of the image. $RMSE$ denotes the root of the mean-squared-error.

We measure the PSNR for each component of Y', Cb, Cr of our V210 input/output format separately. Each component has a bit-depth of 10-bit, MAX_I is therefore always 1024. Because of the 4:2:2 subsampling, MSE_{chroma}, the mean-squared-error for Cb and Cr channels, uses only half the horizontal resolution :

$$MSE_{chroma} = \frac{2}{m * n} \sum_{i=0}^{\frac{m}{2}-1} \sum_{j=0}^{n-1} [I(i,j) - O(i,j)]^2 \tag{6.10}$$

Higher PSNR values indicate higher quality of the processing algorithm. For example, for 10-bit Y' components, an RMSE of 102.4 (informally, an average pixel error of 10% of the coding extent), results in a PSNR of 20 dB. An RMSE of 10.24 (average pixel error of 1%) results in a PSNR of 40 dB. The PSNR value by itself is not very representative. It is rather used when comparing the outputs of different algorithms. The MPEG committee, for example, informally considers a PSNR delta of 0.5 dB for deciding whether an optimization of an encoder should be integrated or not [28].

6.2 Performance

In this chapter, we show the results of testing our prototype for performance. We use the test methodology and terminology that we described in the previous chapter.

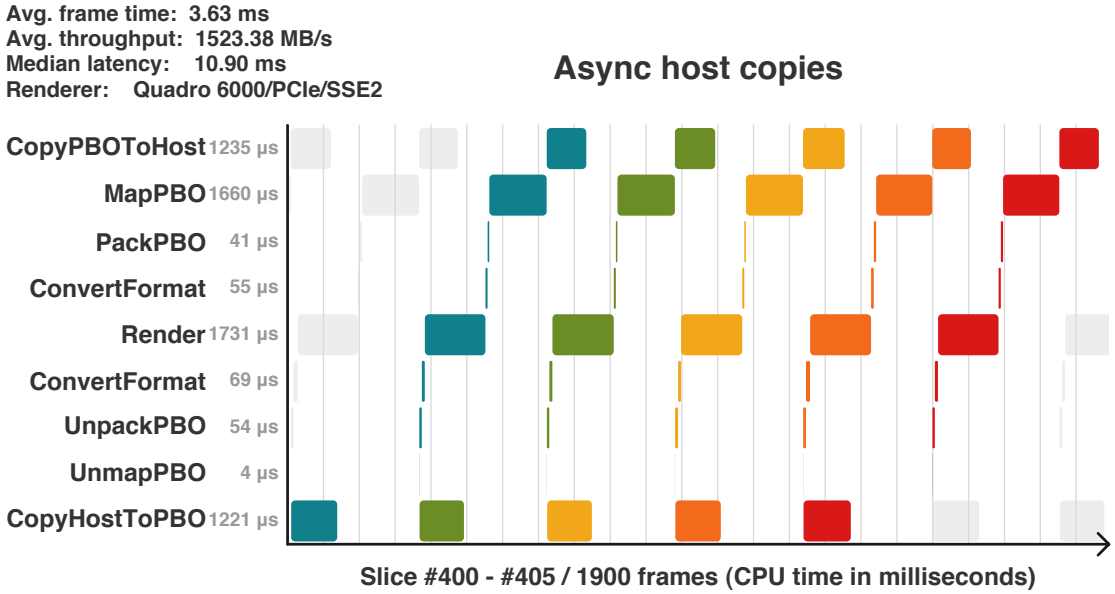
6.2.1 Parallelization of pipeline executions

The base configuration for the pipeline execution is to execute every stage serially. Figure 6.4(a) shows the CPU pipeline trace of a serially executed pipeline in a low-latency configuration. No execution of stages can be overlapped in this scenario, we reach an average throughput of 1174 MB/sec. The first improvement is to offload the host-side copies as described by Chapter 4.3.1. Figure 6.4(b) shows how *CopyPBOToHost* and *CopyHostToPBO* are now overlapped with the rest of the pipeline execution. This increases the throughput by approximately 400 MB/sec to 1520 MB/sec.

To further parallelize the hardware execution of the pipeline, we configure the pipeline to use multiple OpenGL contexts with multiple threads (see Chapter 4.3.2). Figure 6.6(a) shows the CPU trace of this scenario. We can observe that this optimization results in drastically improved performance (1523 MB/sec $\rightarrow$ 2531 MB/sec).

For throughput-oriented scenario, we can further interleave those pipeline executions that are constrained to single-threaded execution (e.g. *UnmapPBO* and *UnpackPBO*, see Chapter 4.2.3.2). We also increase the queue sizes of the connected stages. This results in a deeper pipeline, increases overlap of stage executions, but also results in higher latency. Such a scenario is shown in Figure 6.6(b). In the diagrams, the *interleaving* of two pipeline executions is shown

(a) Base configuration. All stages are executed serially. The highlighted area shows how each processing for a single frame is executed sequentially. Used configuration: $[A3, B1, C1, \boldsymbol{D1}, E6, F3, G_{a3,b2,d2}]$ (see Figure 6.1).



(b) Input and output host-side copy operation are now executed by different CPU cores (see Chapter 4.3.1). Stage executions of *CopyPBOToHost* and *CopyHostToPBO* are overlapping with each other and the OpenGL-related tasks. Throughput is about 1.3 times better compared to the basic configuration. This figure also shows an increase in latency. This latency is introduced because the only way to overlap the copying process of *CopyHostToPBO* is to overlap this execution with the OpenGL stage executions of a previous frame, this increases latency. Used configuration: $[A3, B1, C1, \boldsymbol{D2}, E6, F3, G_{a3,b2,d2}]$ (see Figure 6.1).

Figure 6.4: Parallelizing host-side copy operations. Based on rendering 1080p video.

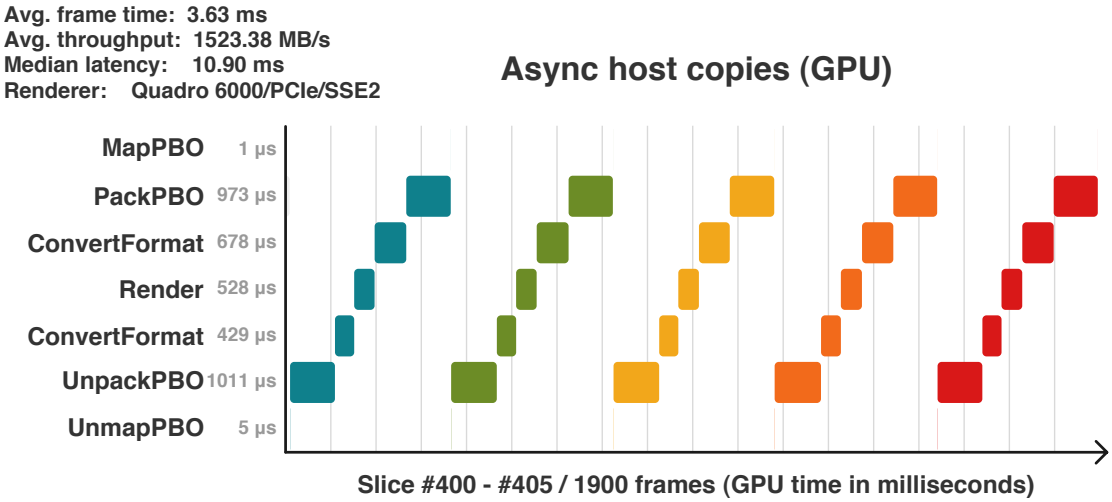
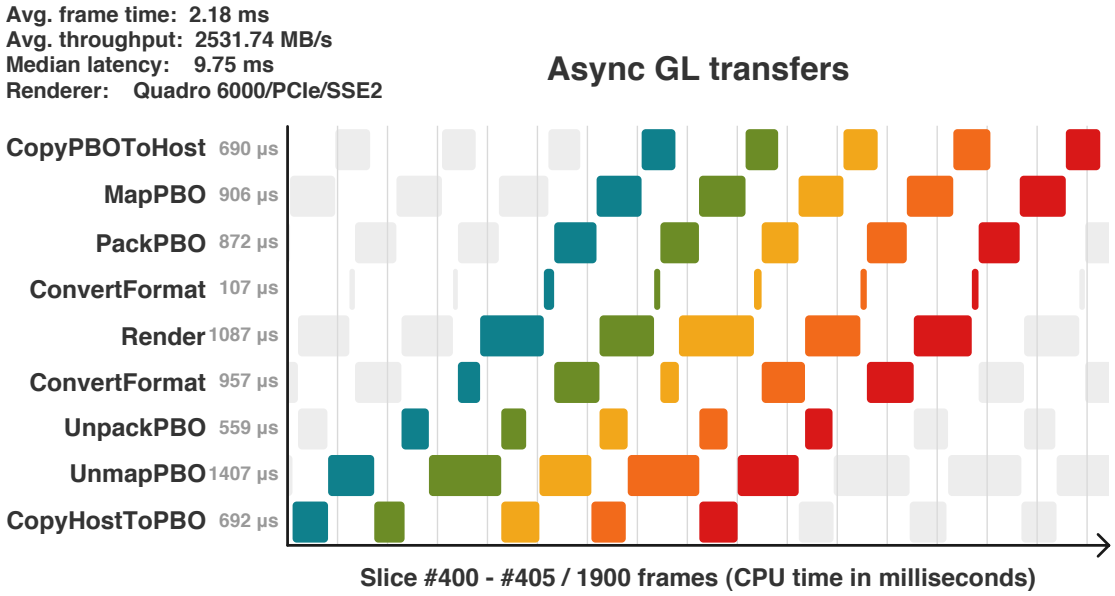
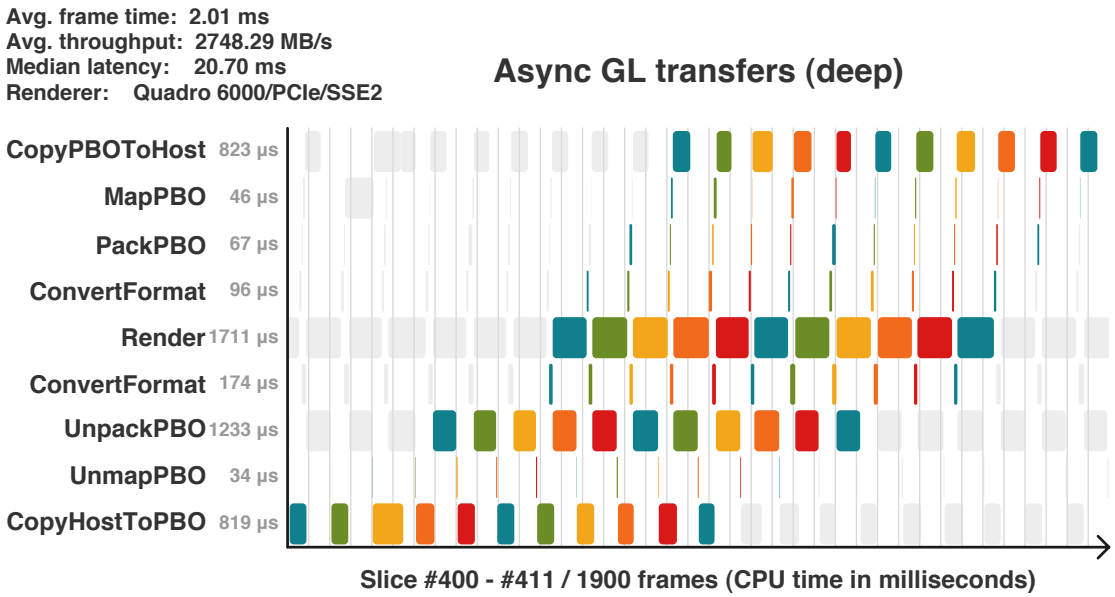


Figure 6.5: This figure shows the GPU trace of Figure 6.4(b) using OpenGL timer queries. We can see that the actual time spent on the GPU differs a lot from what we were able to measure in their respective CPU-sampled stages. For example, the median CPU-side execution time of stage *Render* does not reflect how much time the GPU actually spent converting the format. The GPU-sampled trace in this figure shows the exact time for each operation (which is about 0.5 ms for format conversions and rendering). Also, the actual transfers to and from the GPU can be observed by looking at the times shown in *UnpackPBO* and *PackPBO*. The median of the execution times shows values which would be expected when transferring a single V210 frame of 5.5 MB over the PCIe Gen-2 bus: At a hypothetical peak rate of 6 GB/sec, 5.5 MB of data would be transferred in 916 μs, which is close to the execution times shown for stage *UnpackPBO* and *PackPBO*. Used configuration: $[A3, B1, C1, \boldsymbol{D2}, E6, F3, G_{a3,b2,d2}]$ (same as in Figure 6.4(b)).

as interchanging colors between their following executions. For the rest of this chapter, we favor a throughput-oriented pipeline depth, unless specified otherwise.



(a) Enabling GPU-asynchronous frame transfers on NVIDIA hardware in addition to asynchronous host-side copies further overlaps OpenGL transfer-related stages with the other stage executions (see Chapter 4.3.2). In comparison to the trace shown in Figure 6.4(b), this improves the performance approximately 1.6 times. Latency slightly decreased, because a frame is able to earlier perform asynchronous tasks while the previous frames is still in the pipeline (*hiding latency*). Informally, this is shown in the diagram as a *better fit* of a single frame's trace to the trace of its preceding frame. Used configuration: $[A3, B1, C1, \boldsymbol{D4}, E6, F3, G_{a3,b2,d2}]$.



(b) Using an aggressive configuration of heavy interleaving and larger queues between stages results in a highly-overlapped execution. In comparison to Figure 6.6(a), performance could be improved slightly. Due to the heavy pipelining, latency was doubled. Notice how colors denoting frame numbers are further away and that stage executions of *MapPBO* and *PackPBO* are now much shorter, because *implicit synchronization* of OpenGL API calls could be avoided through heavy interleaving (see Chapter 4.2.3.2). Used configuration: $[A3, \boldsymbol{B5}, C1, \boldsymbol{D4}, E6, F3, G_{a3,b2,d2}]$.

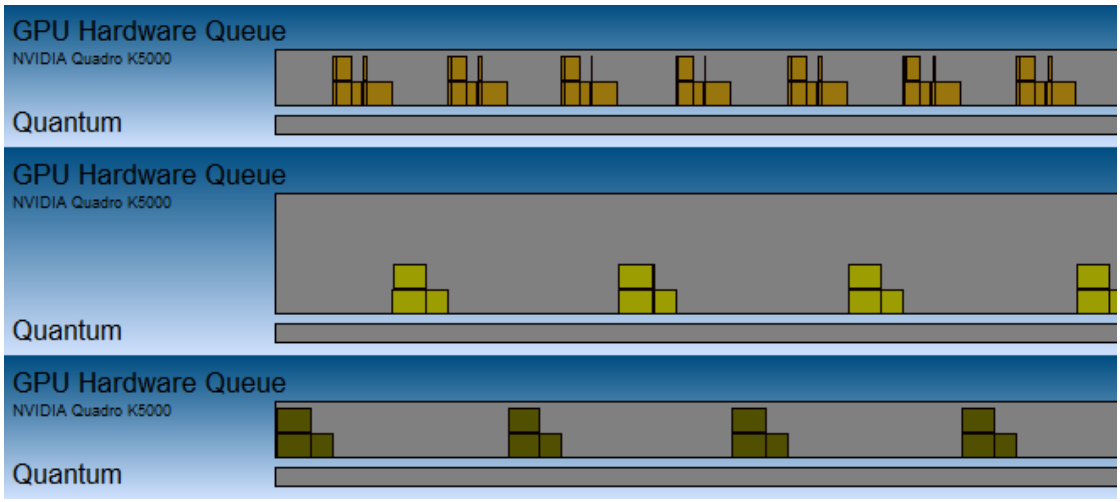
Figure 6.6

6.2.1.1 Visualizing GPU-asynchronous execution using GPUView

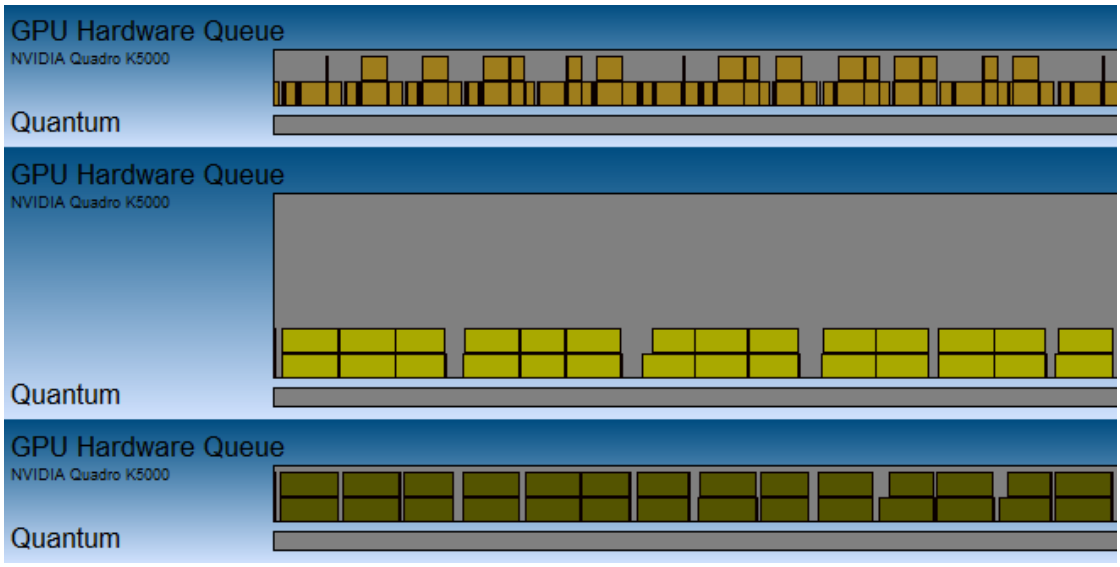
The pipeline execution shown in Figure 6.6(a) is based on CPU-based sampling. As such, the execution times of the OpenGL-based stages (e.g. *UnmapPBO*, *UnpackPBO*, *ConvertFormat (1)*, etc...) only show the CPU time occupied by the OpenGL API calls, but not the actual time spent by the GPU to actually execute those. While the degree of CPU-side concurrency in these stages is somewhat an indicator for the GPU-side concurrency, it does not provide any proof. Originally, we intended to visualize the GPU-side overlap of this scenario using our own GPU-trace using OpenGL timer queries (as shown in Figure 6.5 for a single OpenGL context). However, because of current limitations of the NVIDIA driver we were not able to capture such a trace using multiple GL contexts (see Chapter 6.1.6).

In order to observe the GPU-side overlap, we have used the tool *GPUView* [9][4]. GPUView directly instruments the Windows driver model. In this model, queues buffer commands that are executed next on the hardware units of the graphics card (GPU and DMA units). These queues are global to the system and managed by the kernel of the operating system. GPUView allows visualizing the load of the graphics hardware command queues. Figure 6.6(a) and 6.6(b) show how the scenarios shown in Figure 6.4(b) and 6.6(a) map to actual hardware execution.

[4]GPUView is now part of the Windows performance toolkit which can be downloaded with the Windows 8 SDK [39].



(a) Using a single OpenGL context for NVIDIA cards results in serialized hardware executions.This trace is based on executing our pipeline using the configuration as described in Chapter 4.4.



(b) Using multiple OpenGL contexts from multiple threads results in hardware-parallel execution of upload, render and download: Executions are overlapped as shown in this trace. This trace is based on executing our pipeline using the configuration as described in Chapter 4.3.2.

Figure 6.7: GPUView hardware queue visualization. Three hardware queues represent upload (bottom queue), render (upper queue) and download (middle queue) operations. Queue elements represent hardware command packets, e.g. DMA transfer or render operations. The bottom of each packet stack denotes the packet that is currently processed by the hardware.

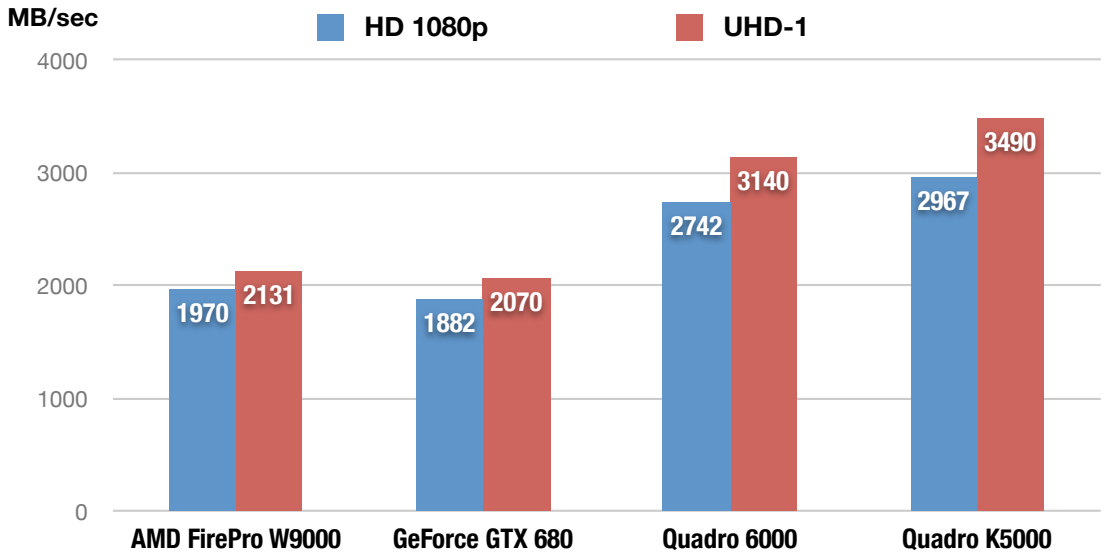


Figure 6.8: The larger data sizes of UHD-1 V210 frames improve the throughput of the overall pipeline. A single V210 frame in HD 1920x1080 uses 5.5 MB of data while it takes 22.1 MB to store V210 in UHD-1 resolution (see Table 5.1). Used configurations: $[A*, B5, C*, D6, E6, F2, G_{a6,b2,d2}]$ ($*$ denotes the varying parameters).

6.2.2 Varying resolution (HD vs. UHD-1)

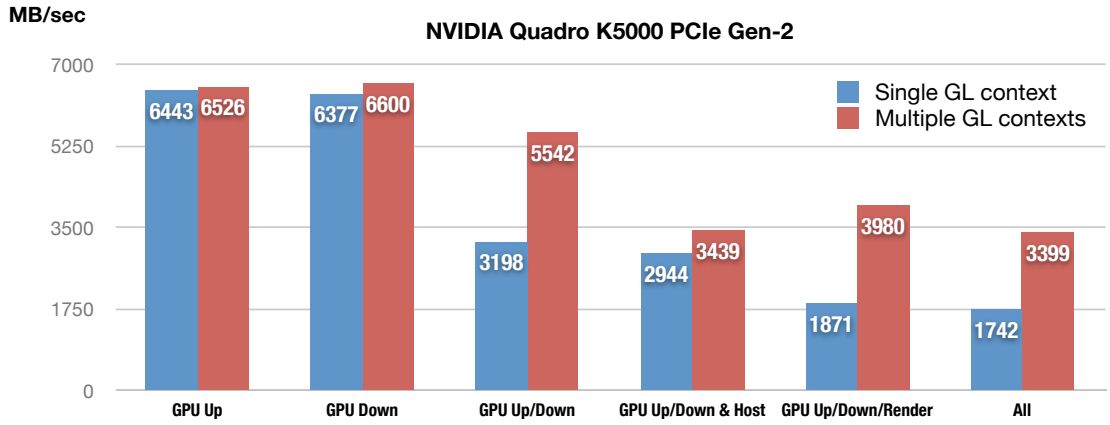
OpenGL PBO transfers are most efficient for data sizes of 8 MB and above (see Venkataraman [35]). Figure 6.8 shows the increase of throughput between transferring frames in HD resolution and UHD-1 resolution.

6.2.3 Isolating pipeline stages

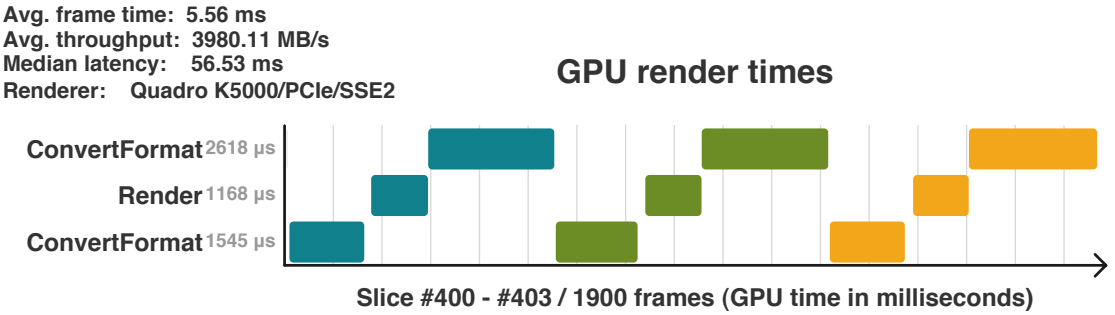
In order to identify bottlenecks in our pipeline execution, we configure the pipeline to bypass certain stages and then analyze the change in performance. This technique is described in Chapter 4.7.

Figure 6.9(a) shows the isolated pipeline stage executions using a single GL context versus using multiple GL contexts on NVIDIA Quadro hardware. As would be expected, this chart shows that GPU-side upload and download can be almost perfectly overlapped when using multiple GL contexts (as opposed to using a single GL context).

We can also observe that performance drops when activating host-side copying (from 5542 MB/sec to 3439 MB/sec). Because we already made sure for this test configuration that host-side copying could be overlapped with other executions of the pipeline (see Figure 6.4(b)), this indicates a bottleneck during the actual copying from and to main memory. We have used the



(a) This chart shows a comparison between execution isolated stages with a single GL context and multiple GL contexts using NVIDIA hardware. Isolated up- and downloads show the expected peak throughput of approximately 6 GB/sec. When performing both up- and download in the pipeline, throughput is cut in half when using a single GL context. Using multiple GL contexts, throughput stays almost the same, i.e. up- and download can be overlapped almost perfectly using two DMA controller units on the NVIDIA Quadro hardware. Adding host-side copies to the pipeline causes a drop in performance, i.e. we are limited by CPU's memory bandwidth. The right-most bar shows the throughput of the fully-enabled pipeline. Used configuration: $[A4, B5, C2, \boldsymbol{D\{2|4\}}, \boldsymbol{E*}, F2, G_{a6,b2,d2}]$.



(b) This figure shows the GPU trace of the stages executed by the render GL context for the scenario *GPU Up/-Down/Render* of the above figure. The sum of their median execution times is 5331 μs. For a frame data size of 22.1 MB, this is equivalent to a throughput of approximately 4145 MB/sec. In Figure (a) we can see that using multiple GL contexts for *GPU Up/Down/Render* results in a throughput of approximately 4000 MB/sec. Because this is very close to the pure render-related throughput of 4145 MB/sec, we can conclude that the GPU-side transfer times are completely hidden beneath the hardware execution of the render context, i.e. we can assume that on the GPU-side we are already bound by the render stages. Used configuration: $[A4, B5, C2, D4, \boldsymbol{E5}, F2, G_{a6,b2,d2}]$.

Figure 6.9: The upper diagram (a) shows the throughput of isolated stage executions in comparison between using a single GL context vs. multiple GL contexts on NVIDIA hardware. The diagram below (b) provides the GPU execution time of the *Render* stage which is common to the above scenarios.

Intel® VTune Amplifier XE 2013[5] in order to analyze this problem. We performed a CPU bandwidth analysis of the *GPU Up/Down & Host* scenario of Figure 6.9(a). We were able to identify the bottleneck as the CPU's memory controller operating constantly at 22.8 GB/sec, which is the peak bandwidth of our test machine's CPU model (see Chapter 6.1.5). This limitation has been confirmed by an NVIDIA engineer to be expected when using a single CPU machine in our scenario.

In Figure 6.9(a) we can further observe a drop in throughput between *GPU Up/Down* and *GPU Up/Down/Render*. Even though we are able to overlap GPU upload, render and download (see Figure 6.7(b)), the render-related tasks (which include stages for format conversion) seem to limit the overall GPU-side execution because render passes can not be overlapped when using a single OpenGL context. Figure 6.9(b) provides the GPU-side execution times of the rendering context. We can observe that the GPU-side rendering tasks alone would result in a hypothetical throughput of 4145 MB/sec. In general, the largest pipeline stage dominates the overall execution time (see Chapter 3.1.3), even for concurrent executions. In this case, the render-related GPU executions limit the GPU-side execution, even if GPU-side up and download can be overlapped.

In summary, we can conclude from Figure 6.9(a) that the GPU-side overlapping of GPU upload, render and download tasks provides almost twice the performance in comparison with serialized GPU-side execution of these tasks. With our test machine, the CPU memory controller is the bottleneck (causing a drop from 5542 MB/sec to 3439 MB/sec). However, this limit is not critical, because GPU-side render-related tasks limit the throughput either way to a maximum of 3980 MB/sec (when using a Quadro K5000), hence the actual limitation of the CPU memory bandwidth is less dramatic (a drop of 500 MB/sec).

While handling multiple GL contexts from multiple threads was worth the effort when using NVIDIA hardware, using multiple contexts on AMD hardware does not provide the same benefits. Figure 6.10 provides a comparison of using single GL contexts and multiple GL contexts in combination with AMD's pinned memory extension. The best-performing approach for AMD hardware is to use a single GL context in combination with the AMD pinned memory extension. This comparison shows us that the execution of our pipeline highly depends on the graphics vendor's implementation of OpenGL.

In order to summarize the behavior of different hardware architectures for isolated stage executions, Figure 6.11 provides an overview of each hardware including different generations of PCIe controllers. This diagram uses the *local best mode* for hardware-side parallelization, i.e. all NVIDIA hardware uses multiple GL contexts, while AMD hardware uses a single GL context. In this diagram we can observe the following:

- The GPU-only upload using a consumer-level GeForce GTX 680 card is extremely high (~12Gb/sec), while download transfer is about half of it (~6 GB/sec). Because the GeForce is primarily used for gaming where texture upload performance is far more important the download, this could be the result of an optimization of the consumer-level driver.

[5] http://software.intel.com/en-us/intel-vtune-amplifier-xe

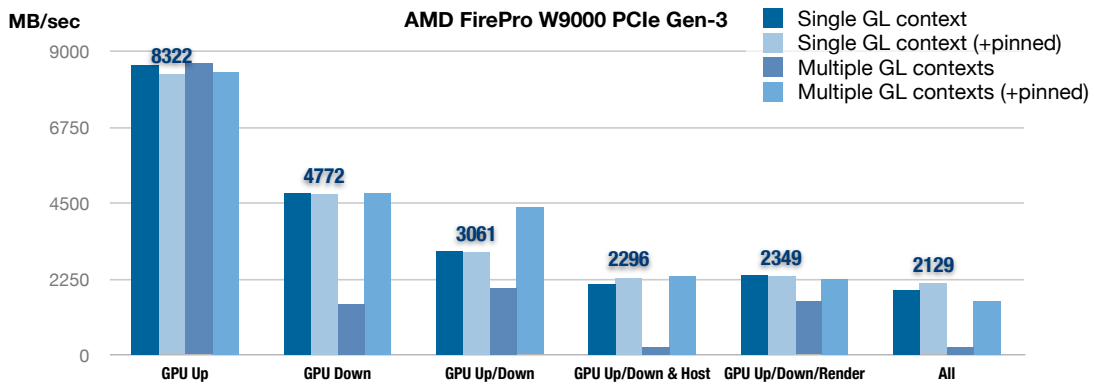


Figure 6.10: Using AMD hardware with different combinations of single/multiple GL contexts and the use of pinned memory extension. Only values of the *single GL context (+pinned)* are labeled. Using multiple GL contexts on AMD hardware without the use of the AMD pinned-memory extension results in very bad performance. The best-performing configuration is *Single GL context + pinned*. Used configurations: $[A1, B5, C2, \boldsymbol{D\{2|3|4|5\}}, \boldsymbol{E*}, F2, G_{a6,b2,d2}]$.

- The GeForce GTX 680 hosts a single on-board DMA controller, while the Quadro devices host two (*dual copy engines*). The GPU-side overlap between transfer and render of the GeForce GTX 680 can be observed by comparing *GPU Up/Down* (2455 MB/sec) with *GPU Up/Down/Render* (2468 MB/sec), i.e. the GPU-side rendering is completely hidden beneath the two (serial) GPU-side transfers. Because the GeForce GTX 680 hosts only a single DMA unit, the performance halves between *GPU Down* (6488 MB/sec) and *GPU Up/Down* (2455 MB/sec).

- While isolated GPU up- and downloads show very high transfer rates for PCIe Gen-3 enabled devices (e.g. ~10 GB/sec of *simultaneous* up- and download), PCIe Gen-3 does not have any measurable advantage in the execution of the actual pipeline. This is expected because of the previously mentioned bottlenecks in our pipeline. It is visible by comparing the throughput of the Quadro K5000 PCIe Gen-2 and Quadro K5000 PCIe Gen-3 in the right-most group (3399 MB/sec vs. 3452 MB/sec).

- Using our test machine, the best configuration of using a Quadro K5000 performs about 1.6 times better than the best configuration of using an AMD FirePro W9000. Considering only throughput, this is the same ratio between the NVIDIA Quadro K5000's performance and the performance of the consumer-grade GeForce GTX 680 graphics device.

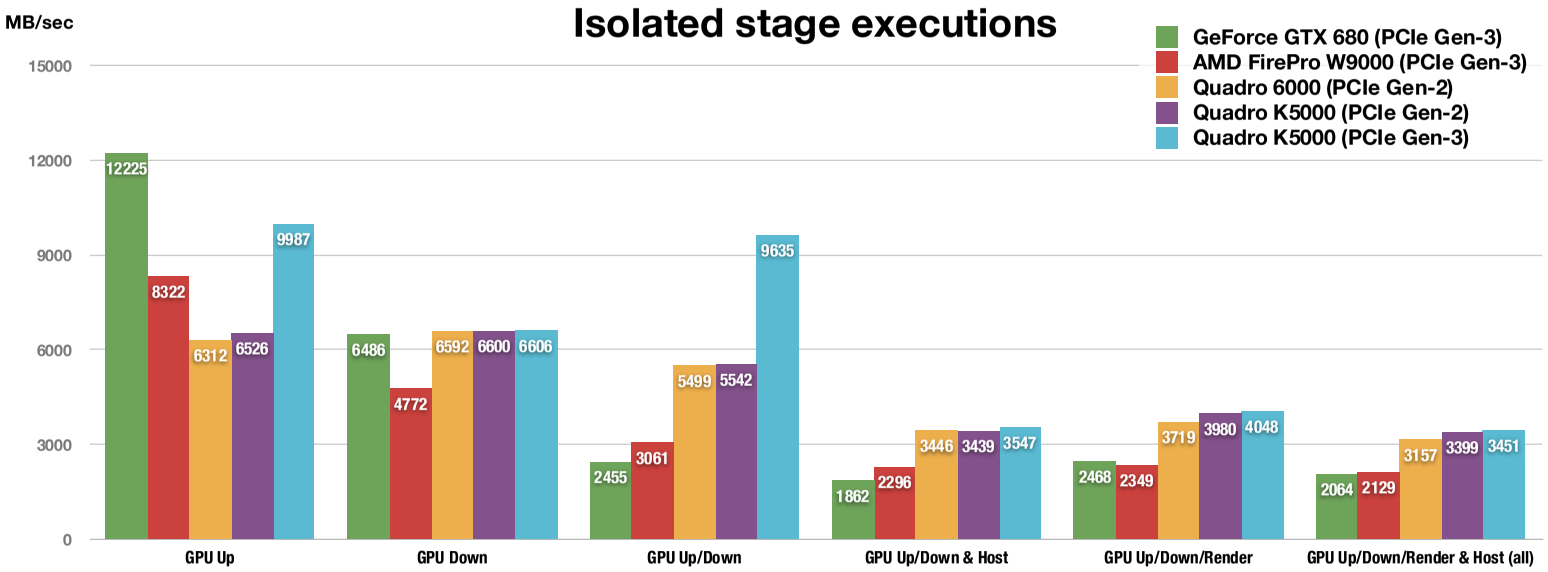


Figure 6.11: This figure shows the throughput of each tested hardware for each stage using its best-mode parallelization setting. Used configuration: $[\boldsymbol{A}*, B5, C2, D6, \boldsymbol{E}*, F2, G_{a6,b2,d2}]$.

6.2.4 V210 transcoder

This chapter shows performance charts of various V210 transcoder settings. We show how performance is influenced when GLSL implementation, chroma filters, V210 storage mode and render formats are varied.

6.2.4.1 GLSL variations

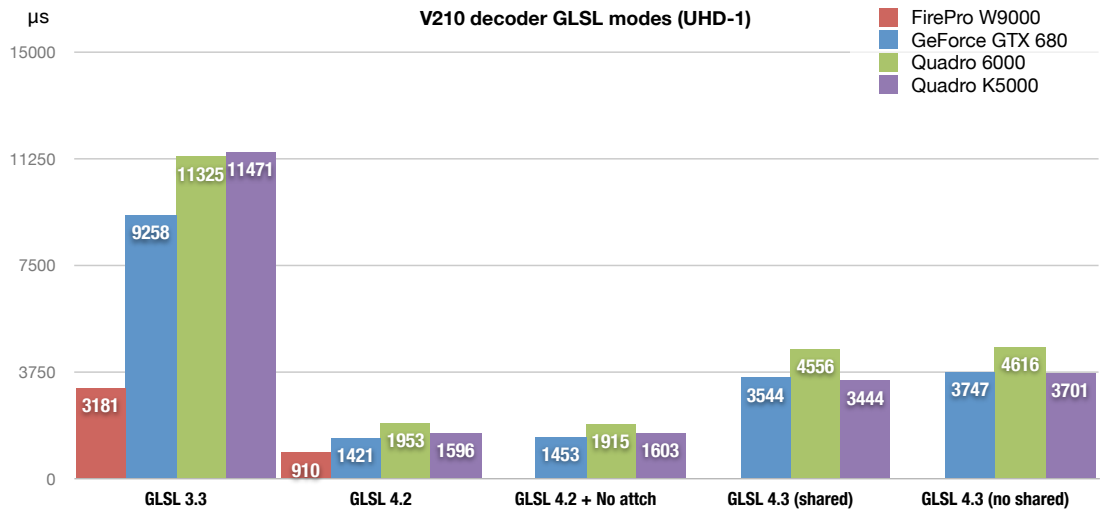
In Chapters 5.3.4, 5.3.5 and 5.3.6, we described three different implementations of the V210 transcoder where each variation uses different features of the GLSL language. Figures 6.12(a) and 6.12(b) show how these variations differ in performance for the decoder and encoder.

The GLSL 3.3 implementation performs worse for all test candidates. This is expected and confirms that the redundant calculations and unnecessarily complex access pattern for a single output fragment is not efficient (see Chapter 5.6).

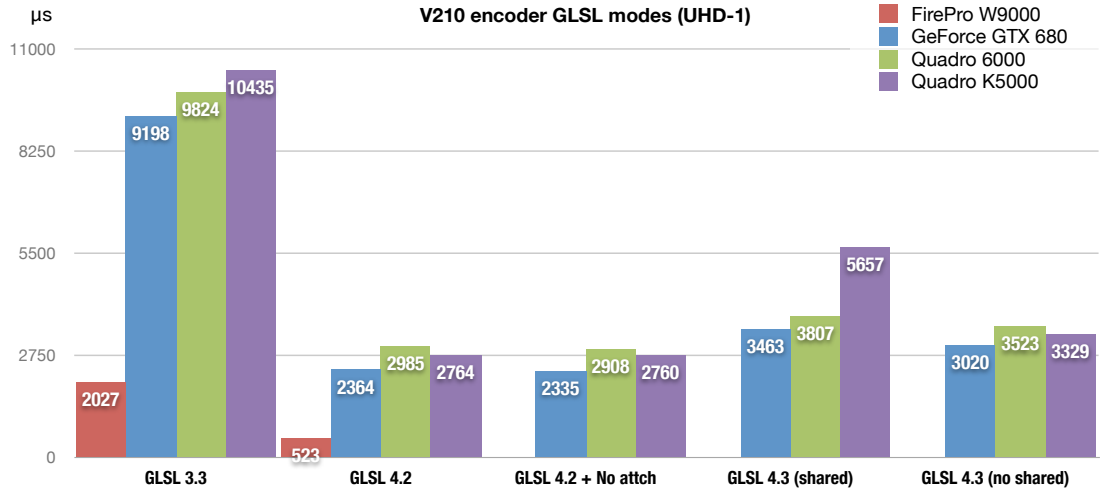
The best performance is achieved by using the GLSL 4.2 approach. Using NVIDIA cards, this approach performs approximately ten times better than the GLSL 3.3 implementation. For AMD cards, this approach is about three times faster. The direct mapping between V210 groups and a group of 6 RGB pixels seems to be very efficient (see Figure 5.6). If supported, the additional use of the `ARB_framebuffer_no_attachment` could slightly improve performance.

The compute shader (GLSL 4.3) implementation of the V210 transcoder shows worse performance than the GLSL 4.2 implementation. Also, caching reads from the input texture in shared memory (see Figure 5.7) performed worse than the compute shader variation that does not use shared memory (direct read/write). Without using a specialized shader profiling tool, we can only speculate about the observed differences in performance. Reasons could be:

- Compute shaders are a relatively new feature of the OpenGL pipeline. The performance difference between our GLSL 4.2 and 4.3 implementation might be due to the fact that GLSL 4.2 use cases are much better optimized by current drivers than GLSL 4.3 use cases.
- The overhead of writing, synchronizing and reading from shared memory is larger than the overhead of redundant reads from the input texture. Those redundant reads are necessary for chroma filtering, i.e. these fetches are spatially coherent and therefore likely to be cached by the texture unit.
- Compute shader performance could be worse than GLSL 4.2 because of memory bank conflicts or bad saturation of shader units. This can only be confirmed by a specialized profiler.



(a)



(b)

Figure 6.12: These figures show the medians of the GPU execution times D_{GPU} (see Chapter 6.1.7.1) for the stage *ConvertFormat (1)* (a), i.e. V210 decoder, and the stage *ConvertFormat (2)* (b). Processing is performed using UHD-1 resolution. Processing 1080p scales linearly to about one fourth of execution times. AMD hardware was only able to provide results for GLSL 3.3 and 4.2 due to current driver limitations (see Chapter 6.1.6). Used configurations: $[\boldsymbol{A*}, B5, C2, D6, E6, F2, \boldsymbol{G_{a*,b2,c4,d2}}]$.

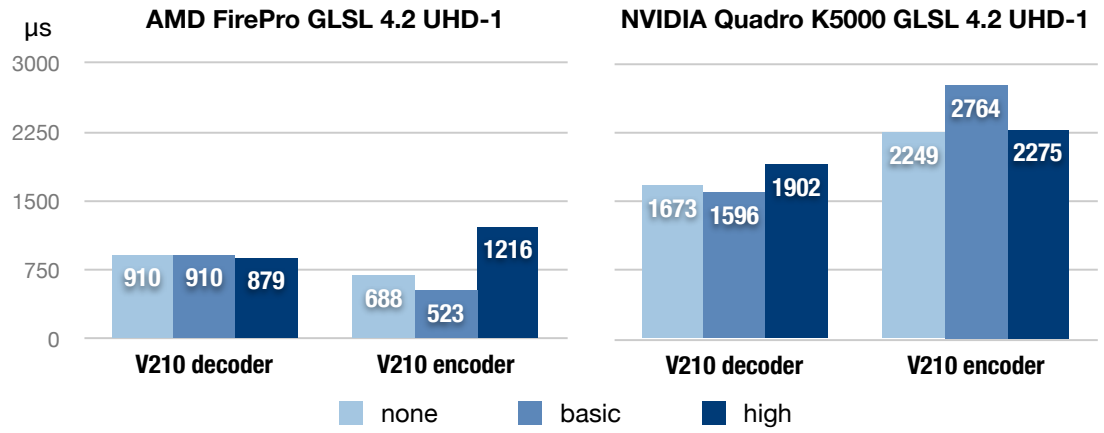


Figure 6.13: This figure shows the impact of varying the chroma filter kernel. The chart shows GPU executions times of the stage *ConvertFormat (1)* and *ConvertFormat (2)*. Used configurations: $[\boldsymbol{A\{1|5\}}, B5, C2, D6, E6, F2, \boldsymbol{G_{a6,b*,d2}}]$.

6.2.4.2 Chroma filters

Figure 6.13 shows the GPU execution times of the V210 decoder and encoder using varying chroma filter sizes. The chart shows inconclusive results between AMD and NVIDIA implementations. The difference in performance between the filter choices seems to be relatively low. We expected a higher difference between using the 13-tap filter for `high` and using no filtering at all (`none`). The V210 encoder shows even an increase in performance when using a larger filter kernel. Without profiling the actual shader, we can not explain this phenomenon. One reason could be the GLSL compiler being able to unroll the 13-tap loop more efficiently than using the 3-tap loop of the basic filter, although this seems unlikely.

6.2.4.3 V210 storage mode

In Chapter 5.2, we described two approaches to store and access V210-encoded video frames in OpenGL:

- Using the `GL_R32UI` internal texture format, a 32-bit integer texture with a single-channel and extracting V210 10-bit components in a shader.
- Using the `GL_RGB10_A2UI` internal texture format where the RGB 10-bit components are already extracted when sampling in the shader.

In Figure 6.14, we compare the average throughput and GPU executions times between those two approaches. The approach of using `GL_R32UI` provides a much better throughput than the other one, even though shader execution takes a bit longer due to the bit operations that need to performed in the shader. It seems that the overhead of the driver to provide the

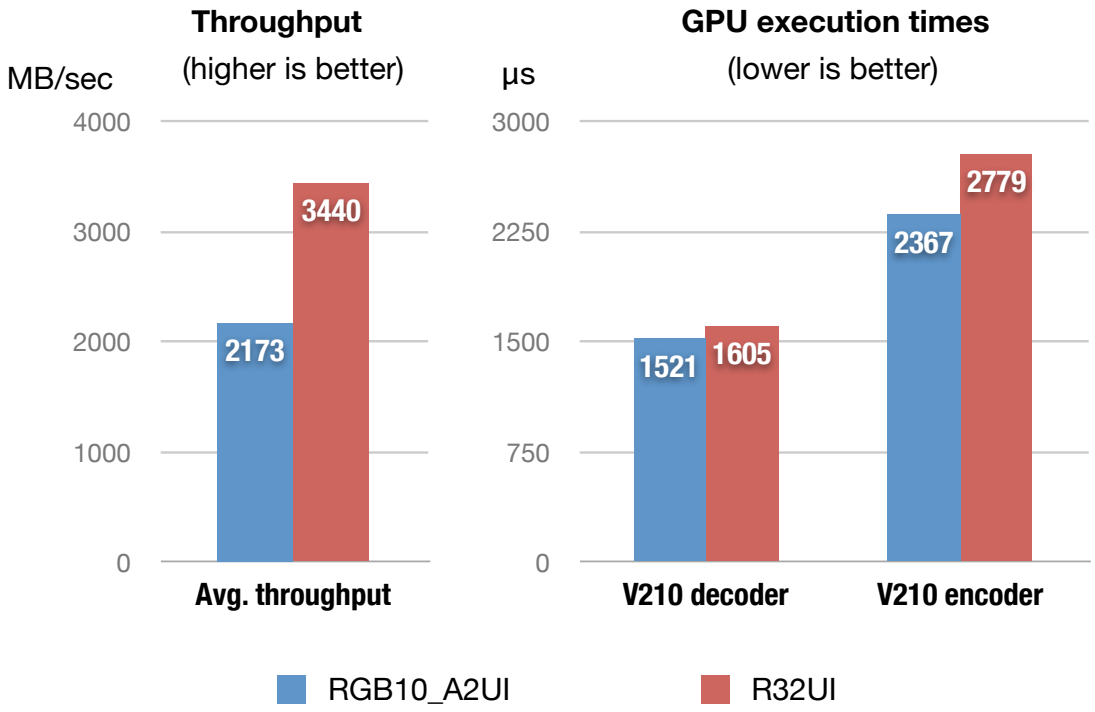


Figure 6.14: The left diagram shows the average throughput for both V210 OpenGL storage modes, while the right-side chart shows the median GPU-side execution times of *ConvertFormat (1)* (V210 decoder) and *ConvertFormat (2)* (V210 encoder). Due to current limitations of the AMD driver, we could only perform these tests on NVIDIA hardware (see Chapter 6.1.6). Used configurations: $[A5, B5, C2, D6, E6, F2, \boldsymbol{G_{a6,b2,d*}}]$

`GL_RGB10_A2UI` pixel format is larger than the overhead of performing additional bit operations in the shader. We can also speculate that this rather unusual pixel format enters a less optimized path during the pixel transfer operations of the driver.

6.2.4.4 Render formats

In Figure 6.15, we vary the precision of the *canonical render format* (see Chapter 4.4). We can read from the chart that our algorithm heavily depends on GPU memory bandwidth. For all devices except the GeForce GTX 680, throughput decreases when precision of the render format increases. The drop in performance between 8-bit integer and 16-bit float is acceptable and much less dramatic than the difference between 16-bit and 32-bit floating point. As it is shown in Chapter 6.3.1, the additional precision of 32-bit floating point vs. 16-bit floating point does not contribute to higher output quality.

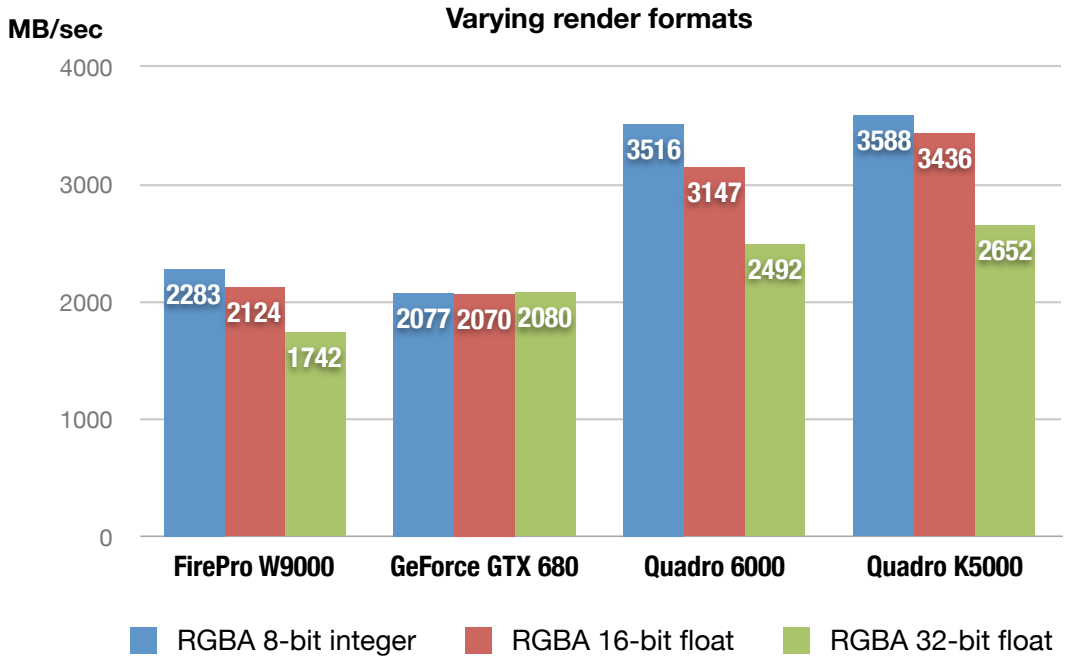
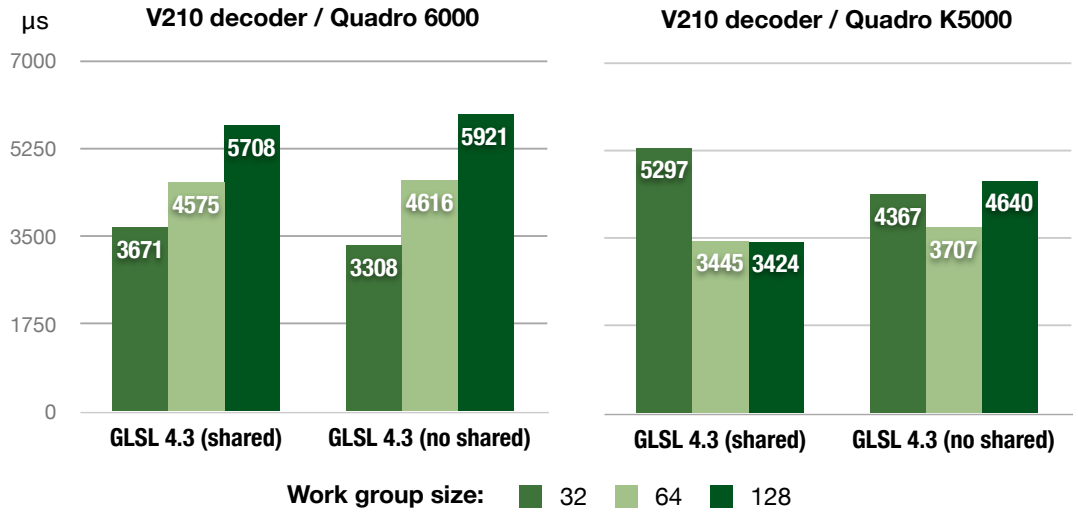


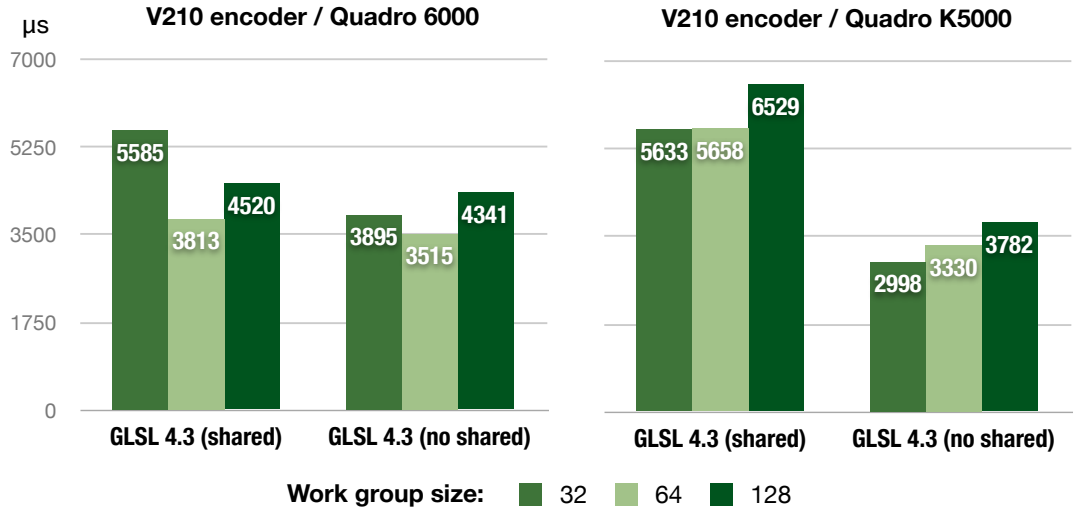
Figure 6.15: Our processing algorithm is largely limited by the GPU bandwidth. Increasing the bit-depth of the render format decreases performance. The GeForce GTX 680 is able to overlap rendering with a single transfer direction, hence the high render time of using a higher bit-depth is hidden beneath a transfer. Used configurations: $[A\{1|2|3|5\}, B5, C2, D6, E6, \boldsymbol{F\{1|3|4\}}, G_{a6,b2,d2}]$.

6.2.4.5 GLSL 4.3 compute shader workgroup sizes

Figures 6.16(a) and 6.16(b) show the influence of the workgroup size for the compute shader implementation of the V210 transcoders. While a work group size of 64 shows best results for the Quadro K5000, the results for the Quadro 6000 are different between the encoder and decoder implementation (decoder favors 32, encoder favors 64).



(a)



(b)

Figure 6.16: This figure shows the performance when varying the work group size for the compute shader implementations of the V210 encoders and decoders. Used configurations: $[A\{3|5\}, B5, C2, D6, E6, F3, G_{a6,b2,c\{2|4|6\},d2}]$.

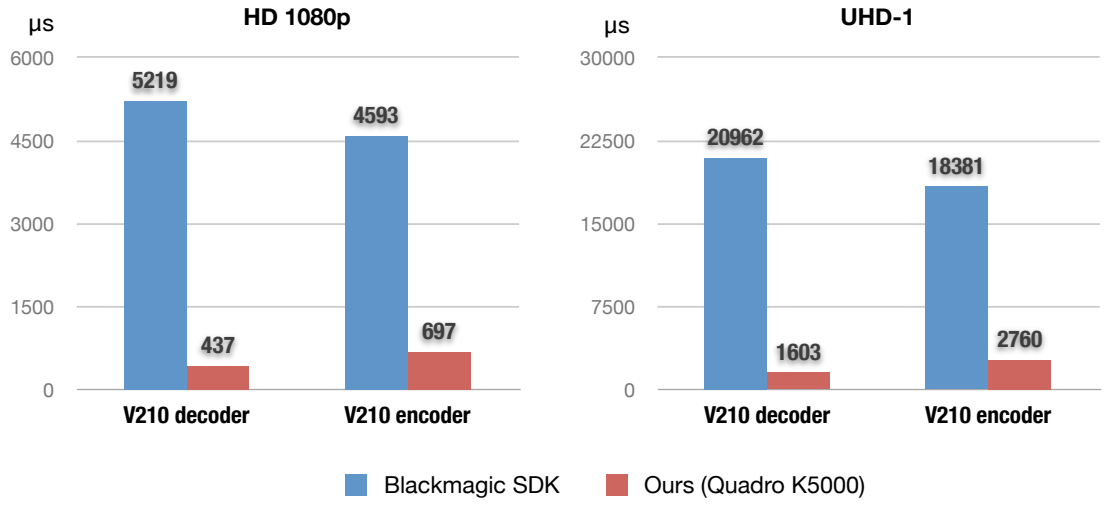
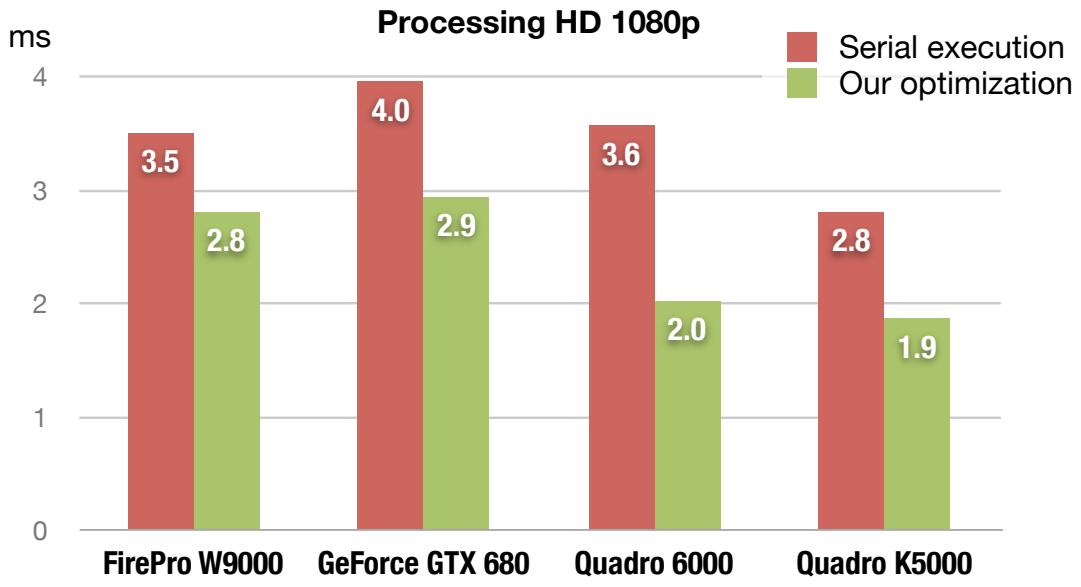


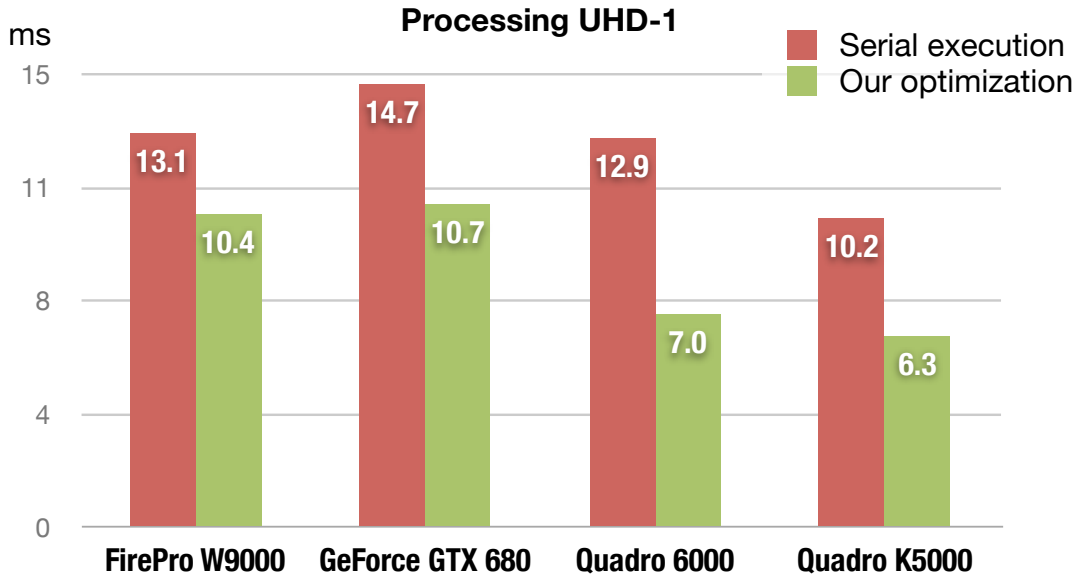
Figure 6.17: We compare the GPU execution time of our V210 transcoding algorithm to the CPU execution of state-of-the-art transcoders. The conversion of the Blackmagic SDK performed a roundtrip from V210 to R210 and back to V210. Our approach as shown in this diagram uses the following configuration: $[A5, B5, C\{1|2\}, D6, E6, F3, G_{a6,b2,d2}]$.

6.2.4.6 Comparing CPU-based V210 transcoding

We compare our GPU-based V210 transcoder with a state-of-the-art CPU-based conversion. Blackmagic Design is a company specialized in providing hardware for professional video solutions. Their software SDK comes with a suite of CPU-based pixel format transcoders. We have implemented a command-line utility which performs a roundtrip conversion from V210 to a 10-bit RGB integer format and then back to V210. Figure 6.17 compares the CPU execution times of these conversions compared with the GPU execution time of our algorithm. Our algorithm is an order of a magnitude faster while providing better image quality than the CPU-based solution (see Chapter 6.3.3). Especially when processing V210 in UHD-1 resolutions, the CPU-based approach will not be able to provide real-time rates. Our transcoding algorithm also performs additional operations like gamma correction, which is not provided by the Blackmagic SDK.



(a)



(b) When rendering UHD-1 in real time, an improvement of several milliseconds is significant, e.g. 60 Hz progressive requires a rendering time less than 16 milliseconds.

Figure 6.18: The serialized execution uses single-threaded interleaving as would a moderately advanced single-threaded OpenGL implementation. The optimized scenario uses the best performing settings for each tested devices.

6.2.5 Overview of speed improvements

In Figure 6.18, we summarize how the optimizations of our framework improve performance as compared to the conventional approach. This method is represented by the serialized execution of our pipeline with single-threaded interleaving (see Chapter 4.2.3.2). In a conventional serialized implementation, interleaving is a common technique.

Figure 6.18(b) shows that the applied optimizations reduce the render times from 12.9 ms down to 7 ms (Quadro 6000). When targeting real-time playout in 50 or 60 Hz, this improvement is significant because it provides more time for more advanced GPU rendering tasks than our demo renderer (see Chapter 4.5).

6.3 Image quality

In this chapter, we show how much the output image is affected when parameters of our pipeline are changed that influence quality (see shaded values in Figure 6.1). For each scenario, we use a renderer that copies the input image to the output image (*pass-through* renderer) instead of our demo renderer. The content of the output image is then supposed to be as close as possible to the original input. We calculate the PSNR (see Chapter 6.1.8) between the input image and the output image in order to quantify the loss in quality of our pipeline using a particular configuration.

6.3.1 Render formats

After decoding, a video frame is stored in the canonical render format (see Chapter 4.4) before it is rendered again, encoded and finally sent back to the host. The bit-depth and precision of this format therefore affects the image quality of the output frame. Figure 6.19 shows a comparison of using different render formats. Because the conversion from Y'CbCr to RGB results in negative pixel values (see Chapter 2.1.6), a floating-point format results in better PSNR values than using integer formats. In particular, by conserving negative pixel values throughput the process, we can restore Y' without any loss (Y' values are stored in the original resolution). We can also observe that using a 32-bit floating-point format shows the same PSNR values as compared to 16-bit floating point.

6.3.2 Chroma filters

In Figure 6.20(a), we show how different complexities of the used chroma filters influence the output quality of our algorithm.

6.3.3 CPU-based implementations

In Figure 6.13, we showed a performance comparison between our approach and a state-of-the-art CPU-based transcoder. Figure 6.21 provides a comparison in quality between our implementation and two CPU-based converters. In both CPU-based converters, we performed a roundtrip

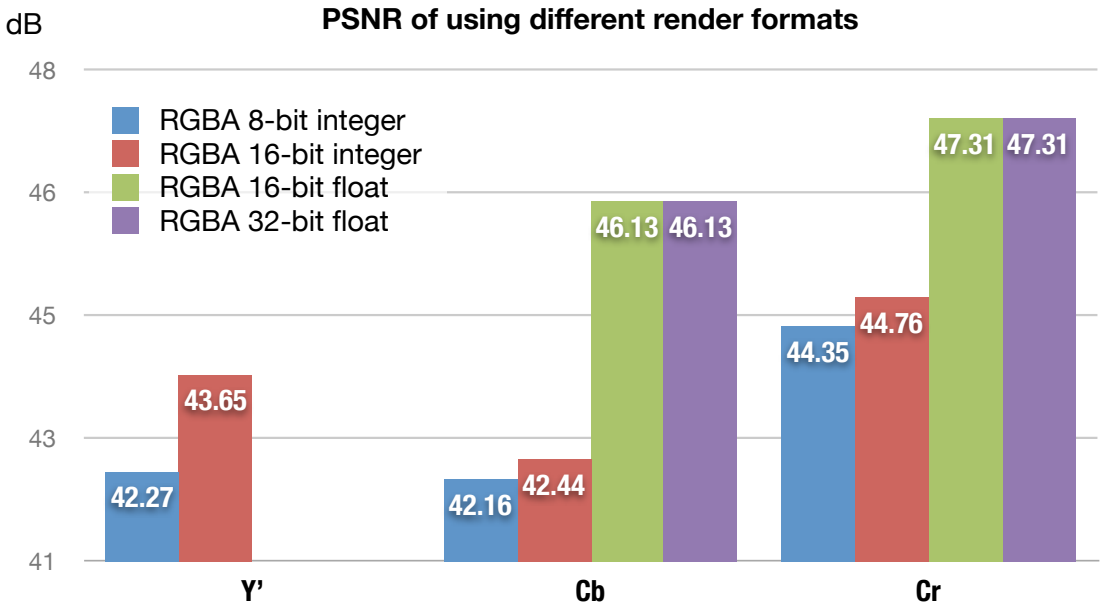
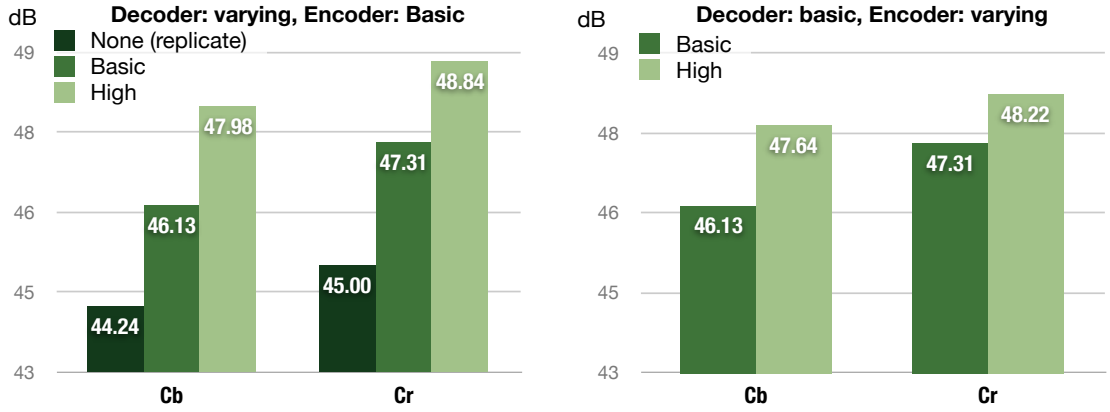


Figure 6.19: Using floating-point for the render format improves the overall quality of our algorithm. Using floating-point results in a mean-squared-error of 0 for Y', therefore the PSNR for Y' is not defined for these formats (no loss in quality).

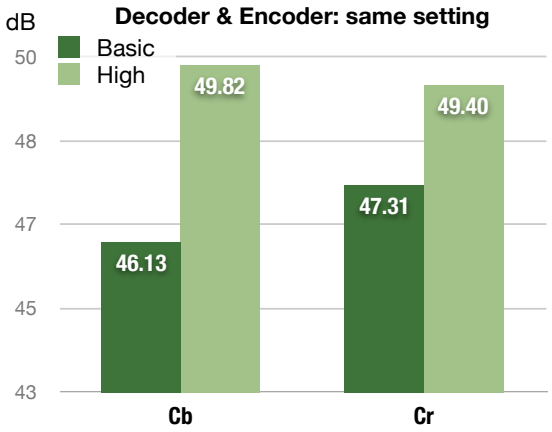
conversion from V210 to high-precision RGB and back to V210. These converters only provided integer-like formats. The charts in Figure 6.21 show that our approach yields superior PSNR values for the V210 roundtrip scenario.

Figure 6.22 further shows a subjective comparison of rendering a synthetic encoder test image [6] between our approach and the conversion performed by the Blackmagic SDK. The top row shows the RGB test image in its original resolution. The other images are the output of a roundtrip scenario between RGB $\rightarrow$ V210 $\rightarrow$ RGB. Our solution using its *basic* chroma filter configuration shows subjectively identical results to the output of the Blackmagic SDK. When no filter is applied (replicate/drop), the output image shows inferior quality. This setting should be avoided.

[6] http://codecs.onerivermedia.com



(a) The left-side figure varies the decoder while the encoder is fixed, the right-side figure varies the encoder.



(b) The best quality is reached by using *high* chroma filter complexity for decoder and encoder.

Figure 6.20: Varying the chroma filters significantly influences the PSNR values of the algorithm. A comparison in performance for these values is shown in Figure 6.13.

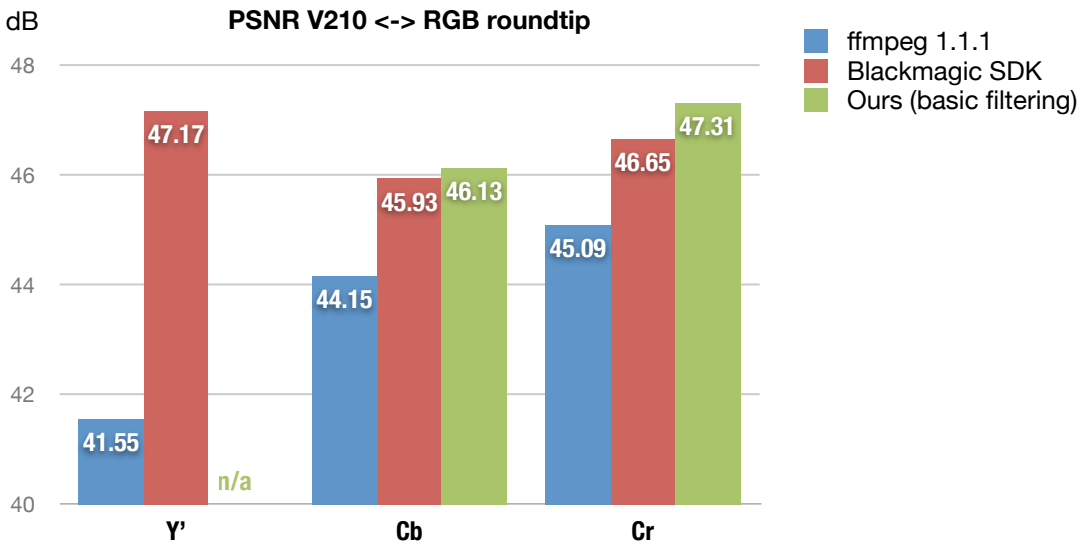


Figure 6.21: Our solution shows superior quality in comparison with two state-of-the-art converters of V210. The Blackmagic SDK used *R210*, a 10-bit integer RGB format, as the intermediate format, because there is no higher-precision RGB format available in their SDK. The ffmpeg conversion uses an integer 16-bit per-component RGB format as the intermediate format.

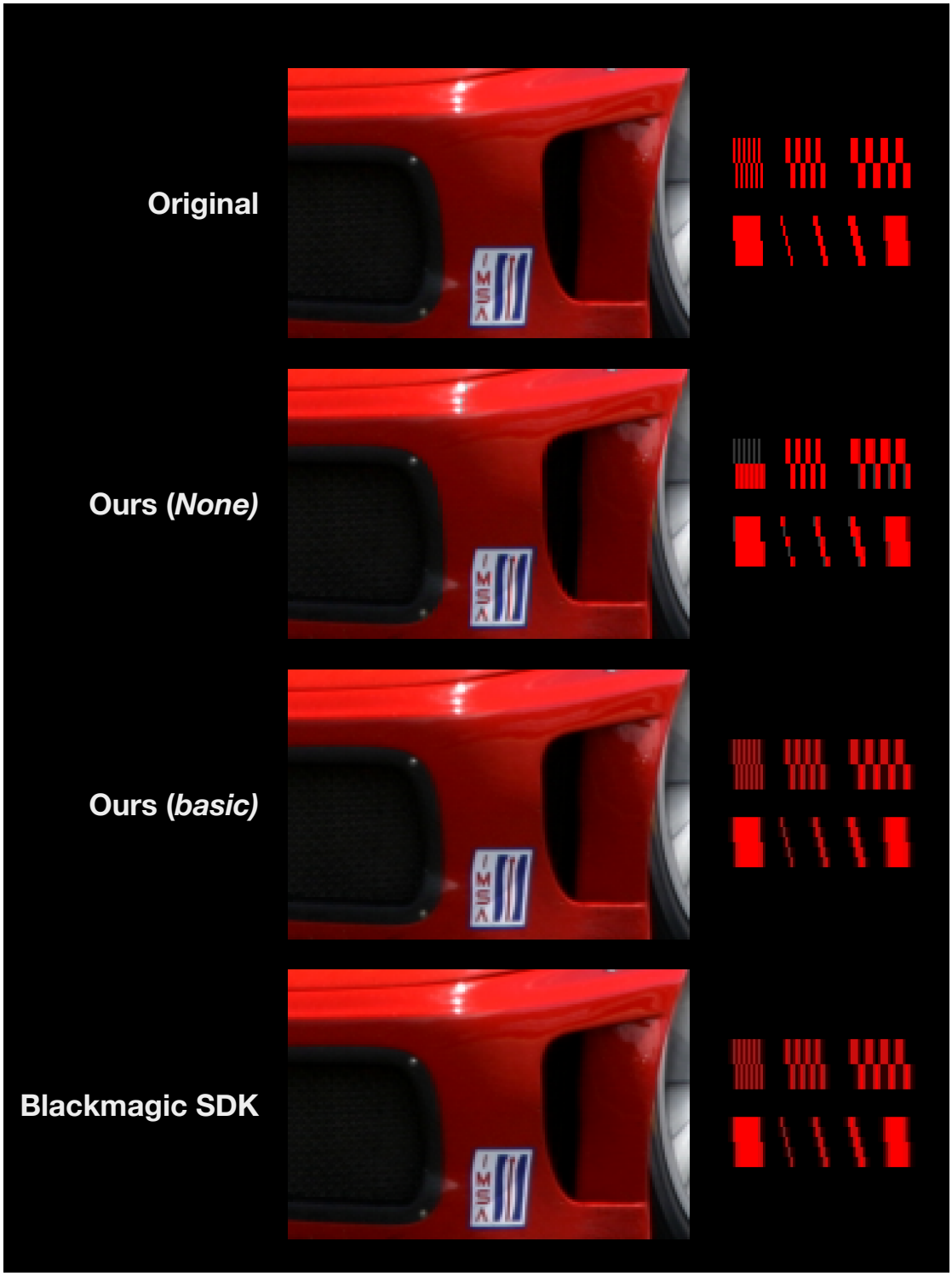


Figure 6.22: A comparison between our filters and using the Blackmagic SDK.

CHAPTER 7

Conclusion

We approached the problem of developing a broadcast-grade graphics renderer using a software pipeline model. We set our goal to improve the throughput of our solution by processing several video frames in parallel on different hardware units of the system. In order to achieve this level of concurrency, we split our OpenGL-based video processing algorithm into individual software stages, which can be executed by several software threads concurrently. Using our prototype implementation, we found out that the actual number of concurrently running stages is limited by the execution model of the OpenGL API and the availability of DMA controllers onboard the graphics hardware. Even though only few stages could be executed concurrently in hardware, we observed a significant performance improvement with our approach in comparison with a serialized conventional implementation. The largest performance improvements were the result of asynchronous frame transfers to and from GPU video memory via the PCIe bus. Our measurements showed that our best performing configuration fully saturates the CPU memory controller. This shows that our framework is able to build a high workload to efficiently process video data. It also means that the performance of our solution will automatically benefit from higher bandwidths of upcoming CPU architectures.

In our video processing solution, we used the OpenGL API to render graphics. In our software framework, we applied two methods to improve the concurrency of OpenGL-based operations:

1. Interleaving executions within a single thread allows the OpenGL driver to execute its own pipeline more efficiently and reduces the risk of enforcing implicit synchronization.

2. Using multiple OpenGL contexts from multiple threads enables hardware-concurrent execution of upload, rendering and download for some hardware devices. With this approach, the implementation has to employ complex synchronization methods.

We were able to implement both approaches using our two-way communication model between stages of our software pipeline. In our test scenarios, we could seamlessly activate either of those optimizations.

While testing our framework on different hardware, we found that there was no single configuration that performed best on each tested device. Our framework required different settings for each tested graphics driver in order to reach optimal performance. We found this to be particularly the case for buffer transfers in OpenGL, where the runtime behavior of the OpenGL drivers varies significantly between different implementations. Our framework's ability to provide the user with a wide range of settings for concurrency and OpenGL usage turned out to be important not only for testing, but also for adapting to the varying characteristics of OpenGL drivers. This is a critical feature for software that is expected to perform well on a range of different commodity hardware.

In summary, we showed that our software pipeline model provides the necessary infrastructure to run tasks concurrently on hardware, that it enables several OpenGL-based optimizations and that it is flexible enough to accommodate the varying needs of different hardware and software environments. We therefore conclude that our software pipeline is a good model for implementing a video processing solution.

In this thesis, we have also implemented a V210 to RGB transcoder using different versions of GLSL. After testing these variations, we conclude that the feature of GLSL 4.2 to perform random writes to image units simplified the GLSL algorithm and significantly improved the performance in comparison to legacy GLSL 3.3 implementations. For the implementation of the V210 transcoder, GLSL 4.3 compute shaders did not provide the advantages that we expected from their direct path in the OpenGL pipeline. The GLSL 4.2 implementation performed better in every test scenario. With compute shaders being a relatively new feature, this might improve in the future. From our experience of implementing and testing the V210 transcoder using different versions of OpenGL, we conclude that OpenGL 4.2 provides significantly better solutions for rendering broadcast video than legacy versions of OpenGL 2.x or 3.x.

We finally compared the performance of a GPU-based, but conventional serialized implementation (simulated by turning off all optimizations in our pipeline) with our advanced solution when processing high-resolution UHD-1 video streams. With our optimization, the video processing pipeline performed almost twice as fast. We were able to reach our goal of processing 60 Hz UHD-1 video at real-time rates on commodity hardware. With some hardware configurations, we could even render two UHD-1 streams simultaneously using one GPU. Processing times for rendering HD 1080 video scaled linearly with the size of the video frames. Our framework was therefore able to process about eight streams of HD 1080 50/60 Hz in real time. In comparison to an existing state-of-the-art CPU-based software SDK, our GPU-based video processing solution provides a higher image quality and performed better by an order of magnitude. We therefore conclude that with our approach, it is possible to build a software playout solution that is able to process new demanding TV formats in high quality and at real-time rates using commodity hardware.

7.1 Future work

The pipeline design of our framework allows us to replace certain stages with different implementations without requiring the rest of the pipeline to be changed. We would like to test using OpenCL or CUDA for implementing the buffer-transfer and format-conversion stages.

Both APIs provide interoperability with OpenGL and use synchronization mechanisms that our framework has already in place. However, interoperability often influences performance and might require the hardware pipelines to be flushed in-between. In future work, we would like to use our framework to accurately measure this impact and outline possible advantages or disadvantages in comparison to a purely OpenGL-based implementation.

OpenGL 4.4 introduced a new core feature with the extension `GL_ARB_buffer_storage`. This extension provides a mechanism to control a buffer's locality of storage. For example, graphics hardware that uses a unified memory space for GPU and CPU executions can benefit from allocating OpenGL buffers in that region, which eliminates the need for buffer transfers over the PCIe bus. This extension also allows defining persistent client-side mappings for OpenGL buffers, which is similar to AMD's pinned memory extension. We would like to integrate and test this new feature in our pipeline.

Finally, we would like to perform more advanced profiling on the execution of the pipeline. Some results of using GLSL 4.3 compute shaders could have been better analyzed using a dedicated tool such as NVIDIA's visual profiler. However, at the time, this tool did not yet support GLSL 4.3 shaders for profiling, but can be expected to do so in the future. We would also like to better understand the CPU memory bandwidth limitations. This would require a more detailed analysis of the OpenGL driver's internal processing and more advanced use of the Intel performance profiling tools.

APPENDIX A

GLSL 4.2 V210 decoder shader code

The following is a full listing of the GLSL 4.2 implementation of the V210 decoder. This implementation has been described in Chapter 5.3.5. Note that some preprocessor macros (e.g. `FB_GLSL_CHROMA_FILTER_NONE`) need to be set at compile time. These macros are written in upper-case letters and start with the prefix `FB_`.

```
1
2  #version 420 core
3
4  #if (FB_GLSL_CHROMA_FILTER_NONE == 1)
5
6  // Replicating filter
7
8  const int chroma_filter_pixel_width = 2; // must be an even number!
9  const int chroma_filter_pixel_offset = chroma_filter_pixel_width / 2;
10 const float[chroma_filter_pixel_width] chroma_filter_pixel_weights = {1.0f, .0f};
11
12 #elif (FB_GLSL_CHROMA_FILTER_BASIC == 1)
13
14 // Simple linear interpolation
15
16 const int chroma_filter_pixel_width = 2; // must be an even number!
17 const int chroma_filter_pixel_offset = chroma_filter_pixel_width / 2;
18 const float[chroma_filter_pixel_width] chroma_filter_pixel_weights = {0.5f, 0.5f};
19
20 #elif (FB_GLSL_CHROMA_FILTER_HIGH == 1)
21
22 // 12-tap filter from Poynton
23
24 const int chroma_filter_pixel_width = 12; // must be an even number!
25 const int chroma_filter_pixel_offset = chroma_filter_pixel_width / 2;
26 const float[chroma_filter_pixel_width] chroma_filter_pixel_weights = {-2.0/256.0,
       6.0/256.0, -12.0/256.0, 24.0/256.0, -48.0/256.0, 160.0/256.0, 160.0/256.0,
       -48.0/256.0, 24.0/256.0, -12.0/256.0, 6.0/256.0, -2.0/256.0};
27
28 #endif
29
30 // This is basically rounding up to a multiple of 6
31 // Note that we need to multiple the chroma_filter_pixel_offset by two, because
32 // the chroma_filter for the chroma values always "skips" one pixel value, e.g.
```

```
33  // in order to access 3 chroma values, you'll need to pass 6 luma values
34  // Note that the offsets are NOT symmetrical, because in the first V210 we
35  // already have one full CbCr pair.
36
37  const int chroma_filter_v210_offset_left = (chroma_filter_pixel_offset - 1 + 2) / 3;
38  const int chroma_filter_v210_offset_right = (chroma_filter_pixel_offset + 2) / 3;
39  const int chroma_filter_v210_width = chroma_filter_v210_offset_left +
        chroma_filter_v210_offset_right;
40
41  const ivec2 v210_image_size = ivec2(FB_INPUT_IMAGE_WIDTH, FB_INPUT_IMAGE_HEIGHT);
42  const ivec2 v210_input_image_bounds = ivec2(FB_INPUT_IMAGE_WIDTH - 1,
        FB_INPUT_IMAGE_HEIGHT - 1);
43  const ivec2 v210_input_image_group_bounds = ivec2(FB_INPUT_IMAGE_WIDTH - 4,
        FB_INPUT_IMAGE_HEIGHT - 1);
44
45  const mat3x3 bt709_rgb_to_ycbcr_inv = mat3x3(
46      0.0011415525114155253f,0.0011415525114155251f,0.0011415525114155255f,
47      -2.8740650732344318e-20f,-0.00020906726889581339f,0.00207098214285714 31f,
48      0.0017575892857142855f,-0.00052246012603867058f,-1.6263032587282567e-19f);
49
50  layout(binding = 0) uniform usampler2D v210_input_image;
51
52  layout(binding = 1) uniform restrict writeonly image2D rgba_output_image;
53
54  layout(pixel_center_integer) in vec4 gl_FragCoord;
55
56  in block
57  {
58      vec2 texcoord;
59  } frag_in;
60
61  layout(location = FRG_OUT_COLOR, index = 0) out vec4 out_color;
62
63  uvec3 uncompress_v210_components_from_word(uint word) {
64
65      uvec3 v210_components;
66
67      v210_components[0] = bitfieldExtract(word, 0, 10);
68      v210_components[1] = bitfieldExtract(word, 10, 10);
69      v210_components[2] = bitfieldExtract(word, 20, 10);
70
71      return v210_components;
72  }
73
74  // Optimized version of sRGB function without using branching
75  vec3 srgb_to_linear_component(vec3 c) {
76
77      vec3 linear_segment = c / 12.92f;
78      vec3 linear_segment_coeff = step(c, vec3(0.04045f));
79
80      vec3 power_segment = pow(max((c + 0.055f), vec3(0.0f)) / 1.055,vec3(2.4));
81      vec3 power_segment_coeff = step(vec3(0.04045f), c);
82
83      return linear_segment * linear_segment_coeff + power_segment *
            power_segment_coeff;
84
85  }
86
87  vec3 convert_YCbCr_to_rgb(uvec3 y_cb_cr) {
88
89      // See Poynton, Eq. 30.6 and 30.7
90      vec3 rgb_norm = bt709_rgb_to_ycbcr_inv * vec3((ivec3(y_cb_cr) - binary_offset));
```

```
91
92      // Apply EOCF
93      return srgb_to_linear_component(rgb_norm);
94
95  }
96
97  void main()
98  {
99
100     // Based on gl_FragCoord, get the group number
101     ivec2 target_coords = ivec2(gl_FragCoord.xy);
102
103     // Number of the compressed group
104     // e.g. x = 960 -> group_num = 160
105     int v210_x_group_base_num = target_coords.x;
106     int v210_y_line_num = target_coords.y;
107
108     // Note that the +1 here accomodates the special case of the assymetric
109     // filter (think it through with a filter_v210_width of 1 what happens
110     // when the pixel_width == 2.
111     uint[(1+chroma_filter_v210_width)*6] luma;
112     uvec2[(1+chroma_filter_v210_width)*3] chroma;
113
114     for (int i = 0; i<(chroma_filter_v210_width+1); ++i)
115     {
116
117         int v210_x_group_num = v210_x_group_base_num -
                chroma_filter_v210_offset_left + i;
118         const uint luma_base_index = i*6;
119         const uint chroma_base_index = luma_base_index/2;
120
121         const int v210_img_x = v210_x_group_num*4;
122
123         const ivec2 read_at_base = clamp(
124             ivec2(v210_img_x, v210_y_line_num),
125             ivec2(0, 0),
126             v210_input_image_group_bounds);
127
128         const bool clamp_right = v210_img_x > v210_input_image_group_bounds.x;
129         const bool clamp_left = v210_img_x < 0;
130
131         uvec3 v210_unpacked_words[4];
132         for (int v210_word_idx = 0; v210_word_idx < 4; ++v210_word_idx)
133         {
134
135             const ivec2 read_at = read_at_base + ivec2(v210_word_idx, 0);
136
137
138             const uint v210_packed = texelFetch(v210_input_image, read_at, 0).r;
139             v210_unpacked_words[v210_word_idx] =
                    uncompress_v210_components_from_word(v210_packed);
140
141         }
142
143         // This is the layout of the array 'words',
144         // with the indices for horizontal sample location
145         // | Cb_0, Y'_0, Cr_0 | Y'_1, Cb_2, Y'_2 | Cr_2, Y'_3, Cb_4 | Y'_4, Cr_4, Y'
                _5 |
146
147         luma[luma_base_index]     = v210_unpacked_words[0][1]; // Y'_0
148         luma[luma_base_index + 1] = v210_unpacked_words[1][0]; // Y'_1
149         luma[luma_base_index + 2] = v210_unpacked_words[1][2]; // Y'_2
```

```
150             luma[luma_base_index + 3] = v210_unpacked_words[2][1]; // Y'_3
151             luma[luma_base_index + 4] = v210_unpacked_words[3][0]; // Y'_4
152             luma[luma_base_index + 5] = v210_unpacked_words[3][2]; // Y'_5
153
154             if (clamp_left) {
155
156                 chroma[chroma_base_index][0]     = v210_unpacked_words[0][0]; // Cb_0
157                 chroma[chroma_base_index][1]     = v210_unpacked_words[0][2]; // Cr_0
158                 chroma[chroma_base_index + 1][0] = v210_unpacked_words[0][0]; // Cb_0
159                 chroma[chroma_base_index + 1][1] = v210_unpacked_words[0][2]; // Cr_0
160                 chroma[chroma_base_index + 2][0] = v210_unpacked_words[0][0]; // Cb_0
161                 chroma[chroma_base_index + 2][1] = v210_unpacked_words[0][2]; // Cr_0
162
163             } else if (clamp_right) {
164
165                 chroma[chroma_base_index][0]     = v210_unpacked_words[2][2]; // Cb_4
166                 chroma[chroma_base_index][1]     = v210_unpacked_words[3][1]; // Cr_4
167                 chroma[chroma_base_index + 1][0] = v210_unpacked_words[2][2]; // Cb_4
168                 chroma[chroma_base_index + 1][1] = v210_unpacked_words[3][1]; // Cr_4
169                 chroma[chroma_base_index + 2][0] = v210_unpacked_words[2][2]; // Cb_4
170                 chroma[chroma_base_index + 2][1] = v210_unpacked_words[3][1]; // Cr_4
171
172             } else {
173
174                 chroma[chroma_base_index][0]     = v210_unpacked_words[0][0]; // Cb_0
175                 chroma[chroma_base_index][1]     = v210_unpacked_words[0][2]; // Cr_0
176                 chroma[chroma_base_index + 1][0] = v210_unpacked_words[1][1]; // Cb_2
177                 chroma[chroma_base_index + 1][1] = v210_unpacked_words[2][0]; // Cr_2
178                 chroma[chroma_base_index + 2][0] = v210_unpacked_words[2][2]; // Cb_4
179                 chroma[chroma_base_index + 2][1] = v210_unpacked_words[3][1]; // Cr_4
180
181             }
182
183         }
184
185         // This is the layout of the array 'words',
186         // with the indices for horizontal sample location
187         // | Cb_0, Y'_0, Cr_0 | Y'_1, Cb_2, Y'_2 | Cr_2, Y'_3, Cb_4 | Y'_4, Cr_4, Y'_5 |
188
189         // This is the index into the accumulated array
190         const uint rgba_horiz_pixel_base_coord = chroma_filter_v210_offset_left*6;
191
192         const uvec2 rgba_pixel_base_coords = uvec2(v210_x_group_base_num*6,
                target_coords.y);
193
194         // First output the values where no interpolation is necessary
195         // (V210 is cosited)
196
197         for (uint rgba_horiz_pixel_offset = 0; rgba_horiz_pixel_offset < 6;
                rgba_horiz_pixel_offset += 2)
198         {
199
200             const uint pixel_idx = rgba_horiz_pixel_base_coord + rgba_horiz_pixel_offset
                    ;
201
202             uint luma_value = luma[pixel_idx];
203             uvec2 chroma_value = chroma[pixel_idx/2];
204
205             const uvec3 y_cb_cr = uvec3(luma_value, chroma_value.x, chroma_value.y);
206
207             vec3 rgb_linear = convert_YCbCr_to_rgb(y_cb_cr);
208
```

```
209             imageStore(
210                 rgba_output_image,
211                 ivec2(rgba_pixel_base_coords + ivec2(rgba_horiz_pixel_offset, 0)),
212                 vec4(rgb_linear, 1.0f));
213 
214         }
215 
216         // Now perform interpolation and write out
217         for (uint rgba_horiz_pixel_offset = 1; rgba_horiz_pixel_offset < 6;
                rgba_horiz_pixel_offset += 2)
218         {
219 
220             const uint pixel_idx = rgba_horiz_pixel_base_coord + rgba_horiz_pixel_offset
                    ;
221 
222             uint luma_value = luma[pixel_idx];
223 
224             // Need to reconstruct chroma values from neighbouring values.
225 
226             vec2 chroma_value = {.0f, .0f};
227 
228             for (int j = 0; j<chroma_filter_pixel_width; ++j) {
229 
230                 // Note that the pixel rgba_horiz_pixel_offset has to be shifted by one,
                        because
231                 // the chrom_value at int(pixel_idx) / 2) is already the one
232                 // left to the spatial position to be interpolated.
233                 int chroma_idx = (int(pixel_idx) / 2) - chroma_filter_pixel_offset + 1 +
                        j;
234                 chroma_value += chroma[chroma_idx] * chroma_filter_pixel_weights[j];
235 
236             }
237 
238             const uvec3 y_cb_cr = uvec3(luma_value, int(chroma_value.x), int(
                    chroma_value.y));
239 
240             vec3 rgb_linear = convert_YCbCr_to_rgb(y_cb_cr);
241 
242             imageStore(
243                 rgba_output_image,
244                 ivec2(rgba_pixel_base_coords + ivec2(rgba_horiz_pixel_offset, 0)),
245                 vec4(rgb_linear, 1.0f));
246 
247         }
248 
249     }
```

Bibliography

[1] Cairo Python bindings. http://cairographics.org/pycairo/, 07 2013.

[2] Charles Poynton's short notes: Converting Y'CbCr to R'G'B'. http://www.poynton.com/notes/short_subjects/video/ycbcr_to_rgb, 07 2013.

[3] Charles Poynton's short notes: ITU-R BT.601-4 Digital Filters. http://www.poynton.com/notes/short_subjects/video/itu-r_rec_601_digital_filter, 07 2013.

[4] EBU 4K & HD test sequences. http://tech.ebu.ch/testsequences, 07 2013.

[5] Mark D Fairchild. *Color appearance models*. John Wiley & Sons, 2005.

[6] Lionel Fuentes. A real-time profiling tool. In Patrick Cozzi and Christophe Riccio, editors, *OpenGL Insights*, pages 503–512. CRC Press, 2012.

[7] B. Gaster, D.R. Kaeli, L. Howes, and P. Mistry. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann Pub, 2011.

[8] Google Protocol Buffers Library. http://code.google.com/p/protobuf/, 07 2013.

[9] GPUView online description. http://graphics.stanford.edu/~mdfisher/GPUView.html, 07 2013.

[10] GTC 2012, Mark Kilgard, *OpenGL in 2012* presentation slides. http://developer.download.nvidia.com/gtc/pdf/gtc2012/presentationpdf/s0023-monday-nvidia-opengl-2012.pdf, 07 2013.

[11] Ladislav Hrabcak and Arnaud Masserann. Asynchronous buffer transfers. In Patrick Cozzi and Christophe Riccio, editors, *OpenGL Insights*, pages 391–414. CRC Press, 2012.

[12] IEC 60027-2-3:2005. *Quantities and units – Letter symbols to be used in electrical technology – Part 2: Telecommunications and electronics*. ISO, Geneva, Switzerland.

[13] IEC 61966-2-2:2003. *Multimedia systems and equipment – Colour measurement and management – Part 2-2: Colour management – Extended RGB colour space – scRGB*. ISO, Geneva, Switzerland.

[14] Intel blog: ippsCopy Vs. ippiCopyManaged. http://software.intel.com/en-us/articles/ippscopy-vs-ippicopymanaged, 07 2013.

[15] Intel® Integrated Performance Primitives. http://software.intel.com/en-us/intel-ipp, 07 2013.

[16] ITU. *Parameter values for the HDTV standards for production and international programme exchange (ITU-R Recommendation BT.709)*. International Telecommunications Union, April 2002.

[17] ITU. *Reference electro-optical transfer function for flat panel displays used in HDTV studio production (ITU-R Recommendation BT.1886)*. International Telecommunications Union, March 2011.

[18] John Kessenich, Dave Baldwin, and Randi Rost. The OpenGL® Shading Language, October 2012.

[19] David Kirk and Wen-mei Hwu. *Programming Massively Parallel Processors*. A Hands-On Approach. Morgan Kaufmann, January 2010.

[20] Kjell Hedström, Lock-Free Single-Producer - Single Consumer Circular Queue. *http://www.codeproject.com/Articles/43510/Lock-Free-Single-Producer-Single-Consumer-Circular*. CodeProject, 2012.

[21] libav - Open source audio and video processing tools. http://libav.org, 07 2013.

[22] Christopher Lux. The OpenGL Timer Query. In Patrick Cozzi and Christophe Riccio, editors, *OpenGL Insights*, pages 493–502. CRC Press, 2012.

[23] Sanjit K Mitra and James F Kaiser. *Handbook for digital signal processing*. John Wiley & Sons, Inc., 1993.

[24] OpenCL - The open standard for parallel programming and heterogeneous systems. http://www.khronos.org/opencl/, 07 2013.

[25] OpenGL extension AMD_pinned_memory. http://www.opengl.org/registry/specs/AMD/pinned_memory.txt, 07 2013.

[26] Charles A Poynton. *Digital Video and HD*. Algorithms and Interfaces. Morgan Kaufmann Pub, January 2012.

[27] C Britton Rorabaugh. *DSP primer*. McGraw Hill, 1999.

[28] David Salomon. *Data Compression.: The Complete Reference.* Springer, 2004.

[29] Mark Segal and Kurt Akeley. The OpenGL® Graphics System: A Specification, Version 4.3 (Core Profile), August 2012.

[30] Peter Shirley and Steve Marshner. *Fundamentals of Computer Graphics, 3rd edition*. AK Peters, June 2009.

[31] SMPTE. 3 Gb/s Signal/Data Serial Interface. In *ST 424:2012*. 2012-10-08.

[32] Bjarne Stroustrup. *The C++ Programming Language, 4th Edition*. Addison-Wesley Professional, 4 edition, May 2013.

[33] The NVIDIA CUDA programming language. http://www.nvidia.com/cuda, 07 2013.

[34] T. True, A. Reid, and J. Jones. Direct to GPU Video Transfers. In *SMPTE Conferences*, volume 2011, pages 1–10. Society of Motion Picture and Television Engineers, 2011.

[35] Shalini Venkataraman. Fermi asynchronous texture transfers. In Patrick Cozzi and Christophe Riccio, editors, *OpenGL Insights*, pages 415–430. CRC Press, 2012.

[36] Webinar *Colour Pipeline for HD*. http://www.poynton.com/notes/events/index.html, 07 2013.

[37] Wikipedia. Binary prefix — Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/w/index.php?title=Binary_prefix, 2013. [Online; accessed 24-August-2013].

[38] Anthony Williams. *C++ Concurrency in Action*. Manning, 2012.

[39] Win 8 SDK incl. GPUView. http://msdn.microsoft.com/en-us/windows/desktop/hh852363, 07 2013.