Modeling Concurrent Data Rendering and Uploading for Graphics Hardware

Dissertation von Markus Wiedemann

München 2021

Modeling Concurrent Data Rendering and Uploading for Graphics Hardware

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

eingereicht von
Markus Wiedemann

aus
München

29.01.2021

Erstgutachter/in:	Prof. Dr. Dieter Kranzlmüller
Zweitgutachter/in:	Prof. Hank Childs, PH.D.
Tag der mündlichen Prüfung:	13.04.2021

Abstract

Graphics rendering hardware often contains specialized components to maximize utilization of its Graphical Processing Unit (GPU). Examples are dedicated memory and copy engines to directly access the graphics memory without blocking GPU processing. Considering that graphics memory is always limited, while the size of datasets produced today is ever increasing, optimized methods for fast data exchange between host memory and graphics memory are needed for real-time visualization.

Understanding the complex relations involved in concurrently rendering and uploading requires sophisticated models describing the hardware and the respective data flow. Input for such models includes the size and structure of the dataset under consideration, the number of parallel threads for processing it, and modular and exchangeable functions performing the data transfers as well as parameters for optimal data access. The models include additional system parameters, such as the concrete hardware used and the various settable clock rates. In this work we describe a methodical approach to derive such models in a systematic fashion. The results are two Models for Asynchronous Rendering and K-time Uploading (MARKU), one for rendering and one for uploading.

Our methodical approach relies on measuring the effects on the host and the graphics hardware, while data movement and processing are executed concurrently. A broad set of experiments is performed, disconnecting inter-process and data dependencies as basis for a statistical evaluation. This allows to identify individual as well as combinatorial influences of the evaluated control variables. Using design of experiments approaches reduces the number of necessary measurements and allows to quickly derive a mathematical description of the underlying processes. With this, we are able to predict performance expectations for specific use cases. Finally, we evaluate our approach in a multi-step process to gain a broad understanding of accuracy and precision of the two MARKUs for rendering and uploading.

Based on our models, future development of memory transfer optimizations are possible to balance the predicted impact on rendering with the performance of the data transfer, leading to improved real-time realizations even for large data-sets.

Kurzfassung

Grafik-Rendering-Hardware enthält oft spezialisierte Komponenten, um die Auslastung der grafischen Verarbeitungseinheit (GPU) zu maximieren. Beispiele sind dedizierte Bauteile zum Speicherzugriff und Datentransfer, die direkten Zugriff auf den Grafikspeicher, ohne die Verarbeitung in der GPU zu blockieren, ermöglichen. Da einerseits Grafikspeicher üblicherweise begrenzt ist, andererseits die Größe der heute erzeugten Datensätze weiter zunimmt, werden optimierte Methoden für den schnellen Datenaustausch zwischen Host-Speicher und Grafikspeicher zur Echtzeit-Visualisierung benötigt.

Das Verständnis der komplexen Zusammenhänge beim gleichzeitigen Darstellen und Ausführen von Datentransfers erfordert anspruchsvolle Modelle, die die Hardware und den jeweiligen Datenfluss beschreiben. Die Eingabe für solche Modelle umfasst die Größe und Struktur des betrachteten Datensatzes, die Anzahl der parallelen Threads für den Datentransfer und modulare und austauschbare Funktionen, die diese Datentransfers durchführen, sowie Treiberhinweise für den optimalen Datenzugriff. In die Modelle fließen weitere Systemparameter ein, wie die konkret verwendete Hardware und die verschiedenen einstellbaren Taktraten. In dieser Arbeit beschreiben wir einen methodischen Ansatz, um solche Modelle systematisch herzuleiten. Das Ergebnis sind zwei *Modelle für asynchrones Darstellen und k-fachen Datentransfer* (MARKU), jeweils eines für das Darstellen und eines für den Datentransfer.

Unsere Methodik beruht auf Messungen der Auswirkung auf den Host und die Grafikhardware, während Datentransfer und Darstellung gleichzeitig ausgeführt werden. Es wird eine Reihe von Experimenten durchgeführt, wobei die Abhängigkeiten zwischen den Prozessen und den Daten getrennt werden, um eine statistische Auswertung zu ermöglichen. Dadurch können sowohl individuelle als auch kombinatorische Einflüsse der ausgewerteten Steuergrößen identifiziert werden. Durch Methoden der Versuchsplanung wird die Anzahl der notwendigen Messungen reduziert und die schnelle Herleitung einer mathematischen Beschreibung der zugrundeliegenden Prozesse ermöglicht. Damit sind wir in der Lage, Performanzerwartungen für bestimmte Anwendungsfälle vorherzusagen. Schließlich evaluieren wir unsere Methodik mehrstufig, um die Genauigkeit und Präzision der beiden MARKUs für Darstellung und Datentransfer umfassend zu verstehen.

Basierend auf unseren Modellen ist eine zukünftige Optimierung von Datentransfers möglich, um die Auswirkung auf das Darstellen mit der Geschwindigkeit des Datentransfers auszubalancieren, was zu verbesserten Echtzeitvisualisierungen auch für große Datenmengen führt.

Eidesstattliche Versicherung

(Siehe Promotionsordnung vom 12.07.11, §8, Abs. 2, Pkt. 5)

Hiermit erkläre ich an Eidesstatt, dass die Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist.

Wiedemann, Markus

Name, Vorname

München, 07.05.21 Markus Wiedemann

Ort, Datum Unterschrift Doktorand

Contents

Chapter 1

Introduction

"The important thing is not to stop questioning. Curiosity has its own reason for existing. One cannot help but be in awe when he contemplates the mysteries of eternity, of life, of the marvelous structure of reality. It is enough if one tries merely to comprehend a little of this mystery every day" [Albert Einstein, Life magazine, May 2, 1955].

Science tries to satisfy curiosity by describing reality, often by using formulas. Such formulas describe how planets revolve around the sun, where electrons might be around atomic cores and why matter and energy are the same. However, in many cases formulas do not draw a complete picture as they usually abstract reality. In their quest for truth, scientists challenge such formulas constantly by comparing simulations with real-world experiments. Usually, the finer the resolution of both, simulations and measurements of real-world experiments, the better the match with reality. With just the right level of fine-grained resolutions, we might detect even the smallest deviations which, in turn, might allow us to adapt or even rewrite the formulas to better fit real behavior.

However, before comparing simulations with real-world experiments, we first need to understand the results of the simulations. These results often comprise several time dependent datasets, which represent the evolution of the simulation in certain time intervals. Resolution itself comprises temporal and spacial resolution. Considering that using small datasets can easily mean having millions or more data points, even *small* simulation runs can be impossible to understand by just looking at the numbers.

Therefore, we create images of datasets to help us with understanding them. This process of creating images out of datasets and the result thereof – the images – is called visualization, c.f. [Han05, p.xiv] and [War08, p.20]. For creating these images, we distinguish two processes: *rendering* and *uploading*.

Rendering describes the computation to gain the pixels for the images. *Uploading* is the process of making datasets available for *rendering*, usually by moving it to graphics hardware (reserved) memory. In addition, we have to do rendering and uploading multiple times to be able to change perspective onto the dataset or change parts of the dataset.

When visualizing datasets to explore them, every additional millisecond spent for rendering an image slows down its examination; the longer we have to wait for the next

view point, the longer it takes to sift through the whole dataset. We might miss connections or relationships just by waiting too long for the next image to be shown. Even worse, waiting constantly for the data to be shown can be annoying for users and halt exploration altogether.

By chance, entertainment in general and computer gaming in particular faces similar challenges, e.g. balancing large and highly detailed scenes with short rendering times, thus leading to the development of complex and optimized rendering techniques to better exploit the capabilities of graphics hardware with the goal to minimize rendering times.

However, when datasets become bigger than memory available on graphics hardware, meaning they can not be stored completely in memory on graphics hardware anymore, other solutions must be found. Simulation results may consist of multiple time steps that include the simulation state at a certain point in time. Such datasets can easily be split into their time steps and while the whole dataset might be bigger than available graphics memory, single time steps might easily fit. Having a time step available in graphics memory usually suffices for rendering that particular time step.

Yet, when we want to see the evolution of the simulation over time (like a movie), the time steps in graphics memory need to be constantly exchanged to be able to render them consecutively. Therefore, we need to load data from host memory and copy them to graphics memory – which denotes the process of *uploading*; in the best case, while concurrently rendering.

To achieve this, various sophisticated approaches were developed for a range of use cases. These studies show that data uploading can substantially affect rendering. Furthermore, they analyze how to find an optimized solution for both together, the time needed for rendering and uploading.

Unfortunately, this leaves out many visualization use cases subject to different performance requirements. These performance requirements constitute different optimization targets. In this work we analyze the following three distinct optimization targets:

OT1 High priority data rendering

OT2 High priority data uploading

OT3 Balancing both activities

OT1: High priority data rendering means that we prioritize data rendering over data uploading finishing on time. An example typically prioritizing rendering is using Virtual Reality (VR) technology for visualizations. VR hardware allows to use natural body movements for changing the perspective: turning your head to look around or walking to explore the virtual scene. Instead of spending time learning how to interact with a computer to navigate through a dataset, users can utilize this time to explore the same dataset and to improve their understanding of it. This also requires the application to maintain low rendering times at any moment to make this interaction as natural as possible. If a virtual movement lags noticeably behind real movement, users might slow down interaction [LH14] or feel discomfort in form of cybersickness [SNL20] which can halt exploration completely.

Hence, when optimizing either rendering or data transfer, we need to prioritize rendering, possibly at the cost of longer data transfer times. However, this might not lead to an optimal solution as most displays have a fixed minimum frame time by design. This means, if rendering is finished earlier than this time slot, we can use the remaining frame time to improve data transfer rates.

OT2: High priority data uploading means that we prioritize data uploading over data rendering finishing on time. This can mean that data uploading must be finished before data rendering can start. An example is to render a video to visualize a dataset. We want to show a certain part of a dataset at a particular point in the video. Consequently, at the time of rendering, this part of the dataset has to reside in graphics memory. Although we can measure overall performance based on total rendering and uploading time, if the datasets to be rendered are absent for a particular frame, the process has failed.

OT3: Balancing both activities connects the former two. Typical examples are real-time desktop visualizations. On the one hand, high frame rates play an important role to ensure real-time interactivity, but having some frames take significantly longer usually does not halt the whole exploration process. On the other hand, showing the next part of a dataset when requested also ensures real-time interactivity, but waiting two or more frames longer still allows you to explore the whole dataset without failing the overall objective. In this case, we might just aim to optimize both rendering and uploading time together, without any particular priority.

1.1 Research Question

For optimization it is crucial to understand the different parameters involved. Nonetheless, before we can fully enhance the performance of visualization applications – and thus improve understanding of simulations and their datasets – we need to figure out what can be tuned and how it affects rendering and uploading. This leads to the main research question **RQ** of this work:

RQ *How to model concurrent data rendering and uploading using graphics hardware?*

This work lays the groundwork to understand the two processes, rendering and uploading, for subsequently optimizing them. We develop a methodical approach that allows to identify and quantify the parameters that influence concurrent rendering and uploading.

Our work is based on the hypothesis that there is a linear mathematical model for each of both processes called *Model for Asynchronous Rendering and concurrent K-time Uploading* (MARKU). The two resulting MARKUs describe the relationship between the identified and quantified parameters and their influence on performance of each of the corresponding processes.

Therefore the research question **RQ** above can be rephrased to: *How to obtain the two MARKUs?*. This question is built upon more detailed sub research questions (SRQ) that illuminate the problem space and outline our methodical approach. They are as follows:

SRQ1 What are use cases for visualization and their performance requirements addressing **RQ**?

SRQ2 How can data be uploaded to graphics hardware while concurrently rendering earlier data?

SRQ3 What parameters can be controlled in these two processes and what are other influences on performance?

SRQ4 How to design experiments to derive the MARKUs addressing **RQ**?

SRQ5 How strong are the identified influences of the two MARKUs for the two processes rendering and uploading?

SRQ6 How to evaluate the two MARKUs and how to find a more optimal solution for the outlined optimization targets?

SRQ1 describes the whole range of requirements that visualization use cases may need to fulfill. By answering **SRQ2** we describe how to optimize for any of the performance requirements for the outlined use cases. The questions **SRQ1** and **SRQ2** combined lead to **SRQ3**. The answer to this question describes what we consider to influence the two processes of uploading and rendering and therefore, what can be tuned in order to optimize for different performance targets. **SRQ4** aims at removing redundancy when experimenting and deriving models from them. The field of statistics has a long tradition of gaining the most information of a small amount of experiments with similar or equal performance compared to doing all possible combinations for given information requirements. **SRQ5** analyzes the derived MARKUs and **SRQ6** describes an methodical approach to evaluate them. Additionally, **SRQ6** evaluates the derived use cases that can help to improve real-world data visualization applications.

1.2 Methodology

This work builds on three typical visualization use cases that describe a continuum of possible use cases. For answering **RQ** the following steps are applied:

1) We systematically analyze selected use cases and derive their performance requirements. From that we derive optimization targets for either uploading data from host memory to graphics memory, for rendering data, or both.

2) We systematically analyze how data uploading and rendering can be implemented, both on hardware and software, and what choices software developers have when designing visualization applications. All these parameters form a set of possible configurations that allow to fine tune for a given scenario.

3) We empirically derive a mathematical model by applying the statistical methods D-optimal design and linear regression to gain an understanding of the underlying processes involved and to quantify influences of changing parameters. The derived models are analyzed to gain an understanding of the included parameters regarding form and strength of their influences on performance.

4) We evaluate the obtained models using the original experimental data for deriving the mathematical model, using additional random configuration experiments, using configurations based on the models and optimization strategies, and using a real-world dataset. This allows to gain an overview on the models accuracy and precision in predicting performance of configurations.

Steps 1) and 2) carry out a qualitative analysis of the problem space. Step 3) performs a quantitative analysis based on the knowledge gained from the first two steps to gain a model of the problem space. Step 4) makes predictions based on the gained models to gain an understanding on their performance and challenge them through experimentation.
The 4 steps of our methodology are described in Chapter 2 for step 1) and 2), Chapters 4 and 6 for step 3) and Chapters 7 and 8 for step 4).

1.3 Contributions

The contributions of this thesis are the following:

- We examine typical visualization scenarios and deduce from them different requirements regarding performance. This allows us to define concrete optimization targets depending on the visualization scenario and their individual performance requirements.
- We examine how data rendering and data transfer can be implemented in hardware and in software and how these two processes can be decoupled. Together with the described individual optimization targets, this allows us to fine tune the optimization of performance.
- We describe a methodical approach on how to obtain a mathematical model describing the influences on concurrent rendering and uploading. By applying approaches from the field of statistics adapted to the challenge at hand, we obtain the mathematical descriptions in form of the two MARKUs and therefore, a description of the involved processes. This helps us to understand and predict the two processes for optimization. With the resulting MARKUs, we are able to guide future visualizations towards better performance.

- Using the two resulting MARKUs, we give an overview on the parameters involved for uploading and rendering datasets and how they affect performance of each.

1.3.1 Associated Publications

The following publications are directly associated with this thesis. They provide either parts of the methodical approach in our work, describe groundwork or frameworks implemented for this thesis, or apply the methodical approach of this thesis.

- Markus Wiedemann, Bernhard S.A. Schuberth, Lorenzo Colli, Hans-Peter Bunge, and Dieter Kranzlmüller: "Visualising large-scale geodynamic simulations: How to Dive into Earth's Mantle with Virtual Reality". In EGU General Assembly 2020, Online, 2020, doi: 10.5194/egusphere-egu2020-5714

  Summary: This work presents the particular result of applying parts of the methodical approach presented in this thesis. A dataset from the field of geophysics that is beyond fitting into available graphics memory is visualized using VR technology in real time.

  Own Contribution: The dataset is time dependent and needs to be uploaded to graphics memory while concurrently rendering it. Parts of the methodical approach of this thesis are applied. This enables to render the full time range of the dataset in real time.

  Other Contributors: Bernhard S.A. Schuberth, Lorenzo Colli and Hans-Peter Bunge provided the dataset. Fine tuning of visualization parameters for visual exploration was a collaboration effort between all authors.

- Markus Wiedemann and Dieter Kranzlmüller. "Statistical Analysis of Parallel Data Uploading using OpenGL". In Proceedings of the 2019th Eurographics Symposium on Parallel Graphics and Visualization, Porto, 2019, pp. 101 - 108, doi: 10.2312/pgv.20191114

  Summary: This work presents mechanisms described in this thesis that allow to decouple the two processes data rendering and data uploading. Furthermore, a statistical analysis that shows the influences of choosing different parameters on performance for the two processes.

- Markus Wiedemann, Christoph Anthes, Hans-Peter Bunge, Bernhard S.A. Schuberth, Dieter Kranzlmüller. "Transforming Geodata for Immersive Visualisation". In Proceedings of the 2015 IEEE 11th International Conference on e-Science, Munich, 2015, pp. 249-254, doi: 10.1109/eScience.2015.80

  Summary: This work presents one workflow for reducing a volumetric dataset to make it real time render-able and a real-time visualization framework that allows the use of various displaying technologies.

Own Contribution: This work lays the groundwork of the visualization framework used in this thesis. By dividing the dataset into smaller parts and implementing concurrent uploading schemes, visualization of datasets bigger than available graphics memory become possible

Other Contributors: Bernhard S.A. Schuberth and Hans-Peter Bunge provided the dataset for the visualization. Christoph Anthes provided the idea for isosurfacing the dataset to help reduce the amount of information.

1.3.2 Publications Not Directly Associated to the Thesis

The following publications are not directly associated with this thesis, but were created as efforts alongside the creation of this thesis in the field of scientific visualization. Some of the techniques developed in the course of the thesis have been applied to new applications.

- Salvatore Cielo, Luigi Iapichino, Johannes Günther, Christoph Federrath, Elisabeth Mayer and Markus Wiedemann. "Visualizing the world's largest turbulence simulation", submitted to Parallel Computing.
- Thomas Odaker, Markus Wiedemann, Christoph Anthes, and Dieter Kranzlmüller. "Texture analysis and repacking for improved storage efficiency". In Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology (VRST 2016), Munich, 2016,
  doi: 10.1145/2993369.2996332
- Christoph Anthes, Rubén Jesús García-Hernández, Markus Wiedemann and Dieter Kranzlmüller, "State of the art of virtual reality technology", In Proceedings of the 2016 IEEE Aerospace Conference, Big Sky, 2016, pp. 1-19,
  doi: 10.1109/AERO.2016.7500674
- R. J. García-Hernández, C. Anthes, M. Wiedemann and D. Kranzlmüller, "Perspectives for using virtual reality to extend visual data mining in information visualization," In Proceedings of the 2016 IEEE Aerospace Conference, Big Sky, 2016, pp. 1-11, doi: 10.1109/AERO.2016.7500608

1.4 Thesis Outline

This thesis provides a description of the process of uploading data to graphics hardware while concurrently rendering data on the associated GPU as well as their performance characteristics. For this, we first analyze how the processes can be implemented in Chapter 2. We start with a high level overview followed by a description of how the graphics Application Programming Interface (API) OpenGL specifies it. We discuss challenges in different implementations. Additionally, we need to define our understanding of performance. As there are different possible scenarios for visualization applications, we discuss

what might be prioritized by which scenario. From this we derive exclusion and soft criteria that model the different requirements for such an application. The rest of the chapter brings these requirements together in one framework. Our framework allows to design uploading and rendering mostly independent from each other. The framework can therefore fulfill all of these requirements and allows to model the processes involved individually. Finally, we describe possible variables and influences involved for both hardware and software parameters.

Relevant related work is discussed in Chapter 3. We first take a look at approaches analyzing OpenGL and how they describe performance for data uploading and rendering. Secondly, we widen the topic by including other non-graphics related APIs that work on the same hardware and, on a higher level, describe a similar process where data is uploaded to hardware and processed with that hardware. We discuss in both cases how they addressed both, modeling data transfers to graphics hardware and using the GPU to operate on the transferred data. We outline their shortcomings and describe our contribution to this topic.

Chapter 4 describes our methodical approach on modeling concurrent uploading and rendering by describing which variables are analyzed, by introducing statistical methods for designing a sufficient number of experiments for our purposes and how to obtain a model from those experiments.

Next we apply these concepts on a target system configuration. Chapter 5 describes the parameters of the involved system. This includes hardware parameters, the included controllable variables (or shorter control variables) and their settings as well as how the experimental data is processed to prepare it for model deduction.

Chapter 6 describes the results of deriving a mathematical model for concurrently rendering and uploading data for the given system. We analyze influences and interactions of control variables in the deduced models as well as which control variables are removed from the models.

In Chapter 7 we describe how we challenge the deduced models. This is done by comparing it to three different experimental configuration sets. The first is the data used for creating the model. As second we randomly create a configuration set to get a broad picture on the models of the whole parameter space. As third we optimize for the outlined three use cases of movie, desktop and VR visualization. For that we measure the best predicted configurations and compare it to the prediction.

In Chapter 8 we use a real-world dataset and compare measured performance with predicted performance. With this we discuss the obtained MARKUs in terms of applicability and intended usage.

Chapter 9 summarizes our findings and outlines possible future work.

Chapter 2

Problem and Requirements Analysis

This chapter provides an overview of the problem at hand and analyzes the requirements for a possible solution. The overall goal of this thesis is to optimize performance of uploading and rendering datasets. For this to be achieved, we first need to analyze and define what performance means in our context. Once performance targets are defined, the next step is to analyze what components are involved when rendering datasets. This includes the analysis of respective hardware and software. In the third step we discuss how to overlap rendering and uploading in order to optimize performance. The fourth step collects all possible parameters described in the steps before, leading to an overview on what can be controlled and what can have an influence on performance. Finally, we summarize all findings of the chapter and describe the connection to the research question **RQ** and its sub research questions **SRQ1-3**.

2.1 Performance

Depending on the scenario, performance for uploading and rendering data can mean many things. In this section we analyze the following three use cases that describe typical visualization applications:

Use case 1 Movie visualizations

Use case 2 Desktop visualization

Use case 3 Virtual Reality visualizations.

Each of these use cases adhere to a variety of priorities or requirements. On the one hand, there are some critical criteria that, when not met, fail the whole process:

- Maximum frame time
- Finishing an upload before rendering

On the other hand, there are soft criteria that help to optimize both rendering and uploading of data to maximize individual priorities:

- Minimize rendering time
- Minimize uploading time

Each of them highly depends on the visualization scenario as well as the context, i.e. how high the frame rate or the uploading performance already is. Before describing what factors can influence performance, we will first take a closer look at these scenarios, categorize them on a priority continuum and then derive requirements or priorities from them. Finally, we combine the findings to derive the optimization targets introduced in Chapter 1:

OT1 High priority data rendering

OT2 High priority data uploading

OT3 Balancing both activities

2.1.1 Use Case Analysis

Naturally, there is a range of use cases and individual priorities possible for each of them. In this work, we focus on the three previously introduced use cases that span a whole continuum of priorities.

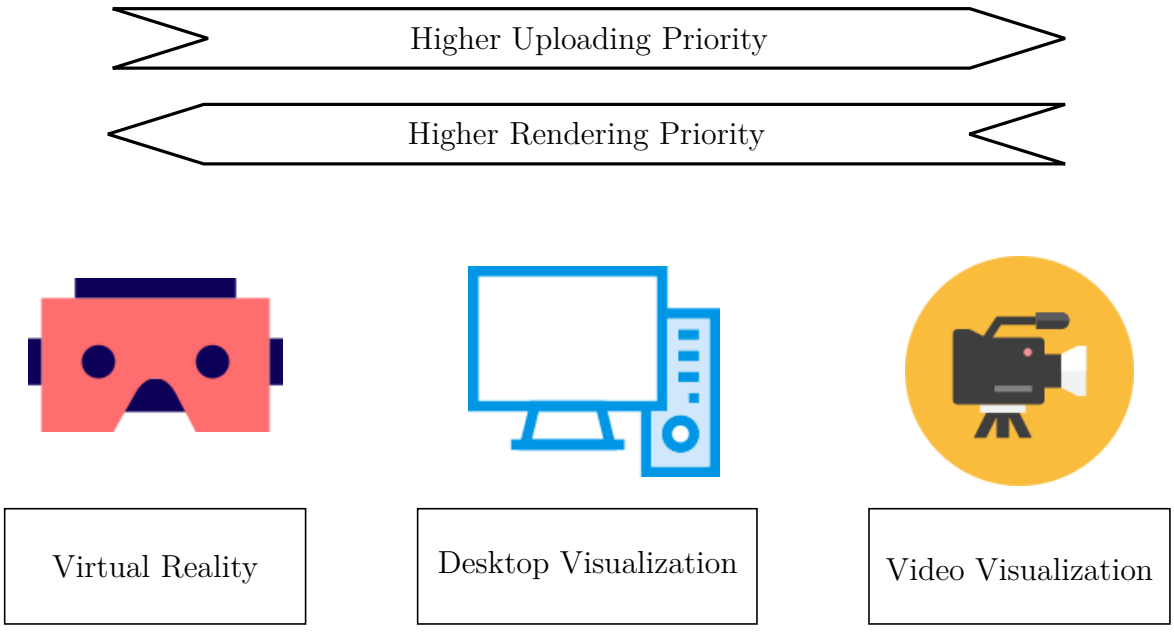


Figure 2.1: Priority continuum for different types of visualizations

Fig. 2.1 shows how we classify each of them on the whole continuum of priorities. On the left side we have VR visualization applications. Usually, when using VR we want to

minimize the latency between users interacting and the image they see adapting. An interaction can be as simple as turning the head which requires updating the virtual perspective to the real pose of the user. Highly sensitive users are able to detect latencies smaller than 5 ms [Jer10] between interaction and an update of the image seen. Higher latencies can induce Oscillopsia [AHJ+01], reduce performance in pointing [WBK+20], searching and reaching tasks [CMG19] or even induce cybersickness [CMG19]. All of this could force a user to stop using the application altogether. So this side of the continuum can easily have a strict deadline for how much time we have to render the next frame. However, although waiting a couple of milliseconds longer for the next timestep of a data set might bother people, it will not necessarily prevent a user from further using the application as cybersickness might.

Rendering a movie to visualize a dataset limits this continuum in Fig. 2.1 on the other side. Usually, when rendering a movie, rendering a frame can take seconds, minutes or even longer. Here, the focus lies on visual detail and not on being able to see it in real-time. However, when we render a frame we want the image to be complete and to show what was defined before. This means that the dataset needs to be complete in the graphics memory, when it is rendered. Consequently, the influence of data transfers on rendering is irrelevant as long as no dataset is present, as the data must be available to render it.

While VR and movie visualization both have these criteria for exclusion, namely having a minimum frame rate or having an upload finished before starting to render, desktop visualization can be classified somewhere on the whole continuum between VR and movie visualization in Fig. 2.1. However, this only covers parts of the whole truth. As soon as the criteria for exclusion for VR and movie visualization have been satisfied, different additional priorities can be set.

Criteria for Use Case 1 – Movie Visualization

For movie visualizations we most certainly need the data to be in memory, for being able to render it. However, this does not mean that priority lies completely on optimizing uploading times. Rendering a movie includes both timings for rendering and uploading. This means, choosing a slower uploading strategy for gaining performance in rendering can also improve overall performance. Fig. 2.2 illustrates this by an example. Assuming there are three possibilities on how to configure both processes: Setting a) balances both the times needed for rendering ($t_{r,n}, n \in 1, 2$) and uploading ($t_{u,n}, n \in 1, 2$). Setting b) reduces the time needed for uploading and with that increases the time needed for rendering. Setting c) increases the time needed for uploading but disproportionally reduces the time needed for rendering. This means that the overall time needed for both processes ($t_{o,n}, n \in 1, 2$) is the smallest for setting c).

Often, rendering for movie visualization includes a high set of visual details, and therefore, rendering times are disproportionally larger than uploading times. This means, if we can overlap uploading with rendering, choosing a slower uploading strategy might not even be noticeable.

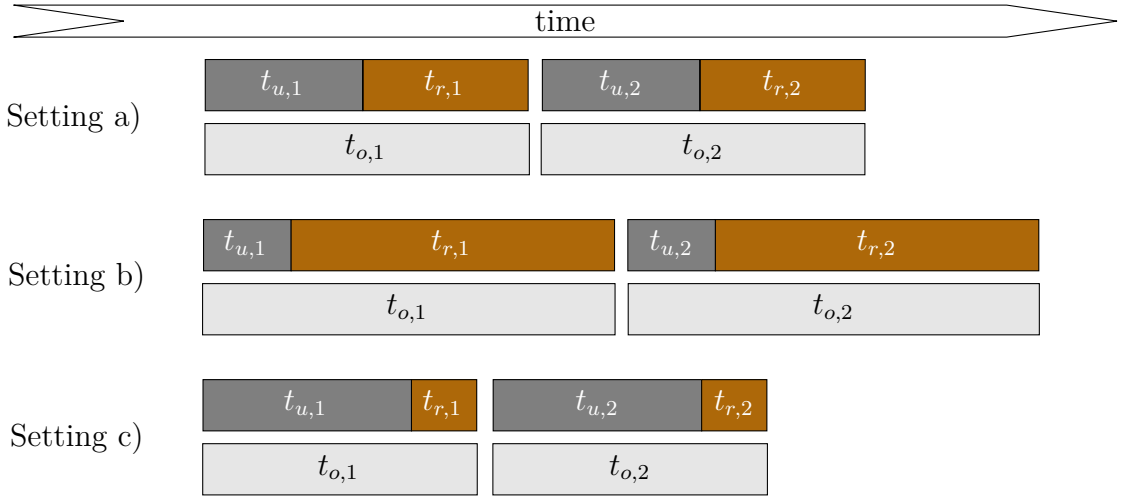


Figure 2.2: Three different strategies applied to a video visualization scenario. Setting a) shows balanced uploading ($t_{u,n}, n \in 1, 2$) and rendering ($t_{r,n}, n \in 1, 2$) times. Setting b) decreases the time needed for uploading but increases the time needed for rendering. Setting c) needs overall the smallest amount of time ($t_{o,n}, n \in 1, 2$) by reducing the time needed for rendering and increasing the time needed for uploading.

Criteria for Use Case 2 – Desktop Visualization

Desktop visualizations are the most variable. Here, application developers have to balance between the performance of both, rendering and uploading. Both, having a high frame rate as well as having low latency between changing a dataset and seeing it being changed is desirable. Often, a developer additionally has to balance between spatial interaction response time and dataset interaction response time, which is represented in balancing between uploading and rendering.

On the one hand, we want real-time interactivity to be able to quickly spatially explore a data set. If rendering times are too high, and motions feel jerky, users might easily get bothered and stop exploration. On the other hand, if there are some frames with high latency, they won't get as cybersick as when using VR visualizations. This means, a developer can play around to get the optimal performance as long as rendering times are not too high.

For uploading performance, the picture is similar. If it takes seconds every time we change parts of the data before it gets displayed, it might interfere with the ability to explore datasets, especially if we want to explore the time dimension. But, if only some uploads have high latency, the impact on a user exploring a dataset can be minimal.

Fig. 2.3 shows how performance for desktop visualization is highly depending on how an application is used. While in scenario a), user interaction only in some frames issues an upload of a new dataset, scenario b) requires constant uploading and increases the total time $t_{o,i}$ needed for rendering for all frames.

Another addition to complexity comes from how images can be displayed on a device.

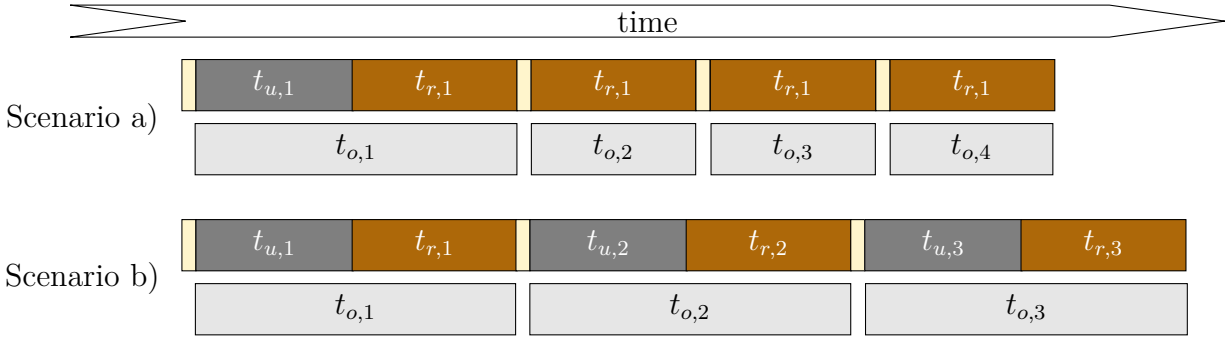


Figure 2.3: Performance for desktop visualization highly depends on usage. For scenario a) only some frames need a new dataset to be uploaded. Consequently, the overall time needed ($t_{o,n}, n \in 1, 2, 3, 4$) mostly depends on rendering time ($t_{r,1}$). For scenario b), many frames need uploading of a new dataset. Here, overall time needed depends on both processes.

Most displays have a fixed built-in refresh rate at which they can display new frames. This refresh rate is also signaled to the graphics card with the so-called vertical synchronization signal or vsync. A newly rendered frame is only shown in the next displaying cycle, when rendering of it is finished before vsync is signaled. If a frame is not finished when vsync is signaled, for the next cycle the frame, that was already shown in the last displaying cycle, is shown again and consequently, the rate of newly rendered and shown frames, or frame rate, is dropping. For uploading and rendering this can mean that splitting uploading in two parts can increase the frame rate as shown in Fig. 2.4. Here, scenario a) shows the case where uploading and rendering together take longer than the time available for one frame. In this case, rendering of dataset d_1 (which is done in time $t_{r,1}$) finishes after vsync is signaled and therefore will be shown after the third vsync (in the third shown frame). Splitting the process of uploading dataset d_1 (which is done in time $t_{u,1} = t_{u,1,1} + t_{u,1,2}$) in two parts as in scenario b) allows to show a new rendered frame every cycle but prolongs the time needed to show the next dataset. Consequently, in the first frame, an old dataset d_0 is shown which is rendered in $t_{r,0}$ and dataset d_1 is also only shown in the third frame after the third vsync. However, in this case, possible perspective changes are shown with lower latency, as they can be included for rendering and are shown in every frame cycle of the display.

Different requirements make it more complicated to find an optimal solution. While usage influences the focus on which process to optimize, vsync can influence how data needs to be uploaded in order to maintain high frame rates. For all cases it is necessary to predict how long one of the processes takes in order to improve performance.

Criteria for Use Case 3 – VR Visualization

VR visualizations have an even more complex set of requirements and can require to meet a certain maximum time between two consecutive frames, usually the same as the refresh time of the displaying device. This also includes the necessity to finish rendering

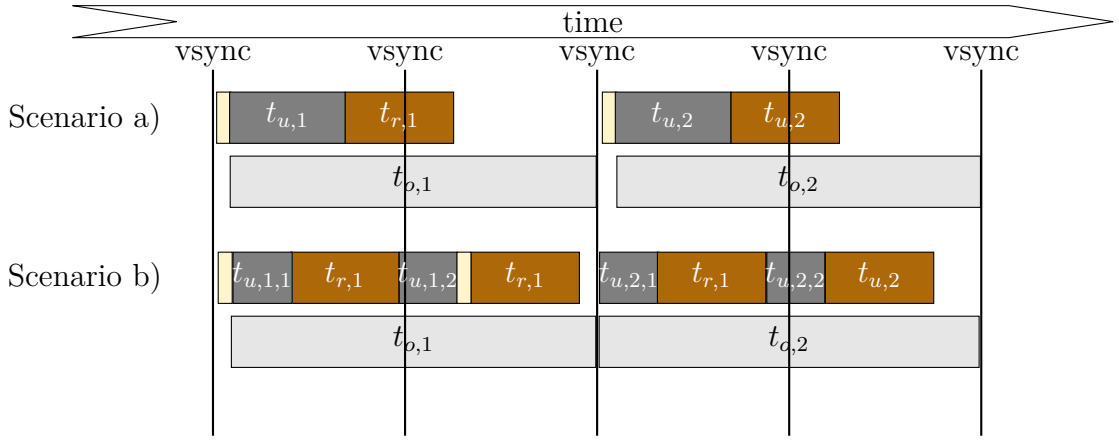


Figure 2.4: Performance for desktop visualization also depends on the refresh rate of the displaying device. For scenario a) the frame rate drops as rendering finishes after vsync. Scenario b) circumnavigates this by splitting up the process of uploading of a dataset.

a frame before vsync is signaled and as such including the same challenges of desktop visualizations. However, we also want to reduce the latency between physical movement and virtual movement. Some implementations therefore move the rendering of a frame to the latest possible time of the frame time. This is illustrated in Fig. 2.5.

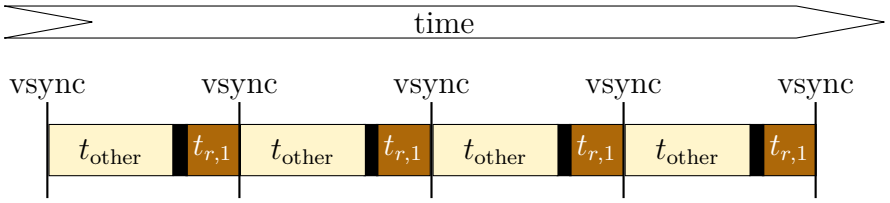


Figure 2.5: Some VR frameworks require to put rendering at the last possible timeslot between two frames. This means that newest tracking information (shown as black rectangle) is used for rendering the next frame to reduce latency between movements and displaying them.

In the beginning of the frame, the timeslot t_{other} can be used to perform other tasks by the GPU. For instance, this can be a physics simulation or setting the OpenGL states. A certain time slot is allocated for actually rendering the frame. This time slot is moved to the end of the frame time and just before it starts, the latest tracking information (black rectangle) is gathered. This allows to keep the latency of physical motion to displaying it fixed. It is also known as *Running Start*, usually 2 to 3 ms before vsync [Vla15, APLK17]. However, the first part of the frame time, namely t_{other}, can also be used to upload data to graphics memory without worrying about impact on rendering, if we are sure the upload is finished before rendering starts.

This can also mean that increasing the aggregated time of uploading and rendering can improve performance. Fig. 2.6 illustrates this issue. While in scenario a) the aggregated

time for uploading and rendering is shorter, rendering takes longer and therefore the time between a physical motion and it being translated to the VR display is longer than in scenario b). However, it is important to notice that here in both scenarios the aggregated time fits into the time allocated for one frame.

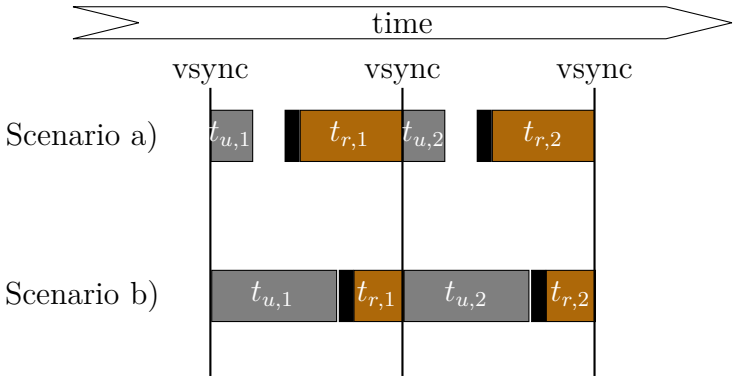


Figure 2.6: In some cases, increasing the time needed for uploading ($t_{u,n}, n \in 1, 2$) can boost the overall performance even though the aggregated time for uploading and rendering is larger. For VR, when all other requirements are met, reducing latency between movement and displaying and therefore reducing rendering time often takes priority.

On the other hand, this strategy is designed for VR games and possibly games that require fast physical reactions. For studying a dataset, these requirements are far above the bare minimum for acceptable experiences. If we have the possibility to choose between a frame rate of 80 Hz and 90 Hz or to have smooth constant uploads instead of some parts popping up and some come not being uploaded at all, we might choose to reduce the frame rate to 80 Hz to enable a smooth uploading rate. This, however, changes completely, when the choices are between 20 Hz with smooth uploading and 30 Hz with discontinuous uploads. Here the visual difference can lie between seeing single images or moving pictures.

2.1.2 Matching Optimization Targets with Use Cases

We see that it highly depends on the situation of each individual use case for which process to prioritize. For use case 1, movie visualization, data must be completely in the graphics memory for rendering. If this is not the case for a particular frame, then at this moment uploading takes priority, which is described by **OT2**. In the remaining situations for movie visualization, the total time for both, rendering and uploading together, needs to be reduced. Therefore, this situation is described by **OT3**.

For use case 2, desktop visualization, a couple of situations depending on user interaction can define which process to prioritize. Usually, high responsiveness and with that short rendering times are preferable. This is described by **OT1**. However, as we saw in the section before, the used displays often work with **fixed** frame times and therefore a time slot for rendering. If rendering finishes early, uploading can be prioritized which is described by **OT2**. Additionally, depending on the type of interaction, situations can occur

where datasets need to be constantly uploaded. In this case, balancing between rendering and uploading might be necessary and therefore, priority lies at the total time for rendering and uploading, **OT3**.

Additionally to the situations described for use case 2, for use case 3, VR visualization, usually, low latency between interaction and showing the result is preferable. This means, rendering times should be as low as possible. As most VR devices also work with fixed frame times and rendering is pushed to the end of these time slots, there are time slots available in the beginning that can be used to prioritize uploading, **OT2**.

Hence, most situations can be described by either one of these three optimization targets **OT1**, **OT2** or **OT3**.

2.2 Rendering Images

In this section we describe how an image can be rendered using graphics hardware. For rendering an image, data needs to be uploaded from host memory to graphics hardware. Once there, it can be rendered and stored as an image or shown on a computer screen.

To achieve these two steps, uploading and rendering, several hardware components and software concepts are involved. Therefore, in this section we take a look on uploading and rendering from the following two abstraction layers:

- Hardware
- Software

The first abstraction layer, the hardware, describes what possibilities are there for data to be uploaded from host memory to graphics hardware, where it is rendered. For this purpose we also analyze the involved components and in which of the possibilities they are involved.

For the second abstraction layer, the software, we analyze which steps are required for uploading and rendering to happen. Additionally, we take a look at specifics related to the actual software implementation, to what details special attention needs to be paid and how overlapping of rendering and uploading can improve performance.

2.2.1 Hardware

Graphics hardware is usually designed to process lots of data using single instructions. This means that the same execution workload is applied to multiple data in parallel, also known as single-instruction stream – multiple-data stream (SIMD) [Fly72]. One feature thereof is to render polygon meshes, where each vertex can be processed using the same instructions. This processing can happen in parallel using hundreds or thousands of small processing cores at the same time and therefore, many vertices can be computed at the same time using the same instructions. Further, graphics hardware usually works asynchronously from the rest of the computer they are attached to. They usually have their own processing unit, the GPU, and their own dedicated memory, the graphics memory or video RAM (VRAM).

Assumptions

We base the following analysis on two assumptions:

Triangle meshes	We assume that the dataset to be rendered consists of meshes that are constructed from triangles.
Data in Host Memory	We assume that data, which ought to be rendered, already resides in main memory.

Triangle Meshes Instead of using triangle meshes for rendering, other approaches exist as well, for example volume rendering techniques, but are not part of the focus of this thesis. For more information on those techniques see e.g [Bar93] for an overview on several rendering techniques for medical data or [Han05, chap. 7] for a general overview on volume rendering for visualization. When using triangles for rendering, this means, we have at least a list of vertices describing the position of the triangles in space. Each vertex can contain more information than just the position, for example its color, its normal or its texture coordinates, which further help to describe visual properties of the surface the triangle describes.

Data in Host Memory We require that data is already present in host memory, for rendering it. This assumption is based on two reasons:

1. Main memory is easier expandable than graphics memory as available hardware allows it. In many cases, CPUs and mainboards allow to address and use large amounts of host memory and easily add new host memory by just sticking it into the memory slots. For graphics memory this is usually not the case and to add new graphics memory to graphics hardware, soldering it to the graphics board is required. This shows that the amount of host memory can easily exceed graphics memory for a given system[1]. Consequently, if a dataset exceeds available graphics memory but fits into main memory, we can preload the dataset into main memory and repeatedly use it for visualizing it. This, however, means we need to repeatedly copy it to graphics memory.

2. If data does not fit into main memory, we can use multiple Solid State Disks (SSDs) in RAID 0 configuration. RAID 0 allows to use multiple hard disks or SSDs together to increase write and read bandwidth onto them. This in turn can increase reading performance up to the point where a transfer from main memory to graphics memory is slower than from storage to main memory [GRE09].

[1] At the moment of writing this thesis, the highest available amount of graphics memory is 48 GB for the NVIDIA Quadro RTX 8000. Yet, the Intel Xeon Platinum 8260L CPU can theoretically address up to 4.5TB of main memory and the AMD Epyc 7742 CPU up to 4TB.

Involved Hardware

An abstract overview of the involved hardware components valid for most rendering hardware available today is shown in Fig. 2.7. For this and the following figures, when referring to components of those figures, words in the text are written italic to mark them visually.

The involved hardware includes (from bottom to top):

- Host memory or *RAM* (Random-Access Memory)
- *CPU*
- The connection between *CPU* and graphics hardware or in this case the *PCI Express* (PCIe)
- Graphics hardware, in this case a *Graphics Card* with (from left to right):
  - *VRAM* as graphics memory
  - *Host Interface*
  - *GPU* with:
    - *Memory Controller*
    - *DMA* (Direct Memory Access) *Engine(s)*
    - *GPU Cores*

RAM is connected to *CPU* which is connected via the *PCI Express* to the *Host Interface* of the *Graphics Card*. The *Host Interface* is connected to *VRAM* and *GPU*. Within the *GPU*, there is a connection from *DMA Engine(s)* to the *Memory Controller* and externally to the *VRAM*.

Following the work of [KMMB12, KAB13, FAN+13], multiple paths for data to be rendered using the *GPU* exist:

- **Direct CPU Transfers**
- **Pinned Transfers**
  - **Direct**
  - **DMA Based**
  - **Microcontroller Based**

For all paths, the general data movement is as follows: Data is moved from *RAM* via the *CPU* (and integrated chips/interconnects) and *PCI Express* to the *Graphics Card*. On the *Graphics Card*, data can either be transferred via the host interface directly into *VRAM*, via *DMA Engine(s)* or other microcontrollers and the memory controller into *VRAM*, or directly be processed by the *GPU*. However, how this is achieved and who is actively performing the transfer depends on the implementation of the graphics hardware, its driver and the particular visualization application.

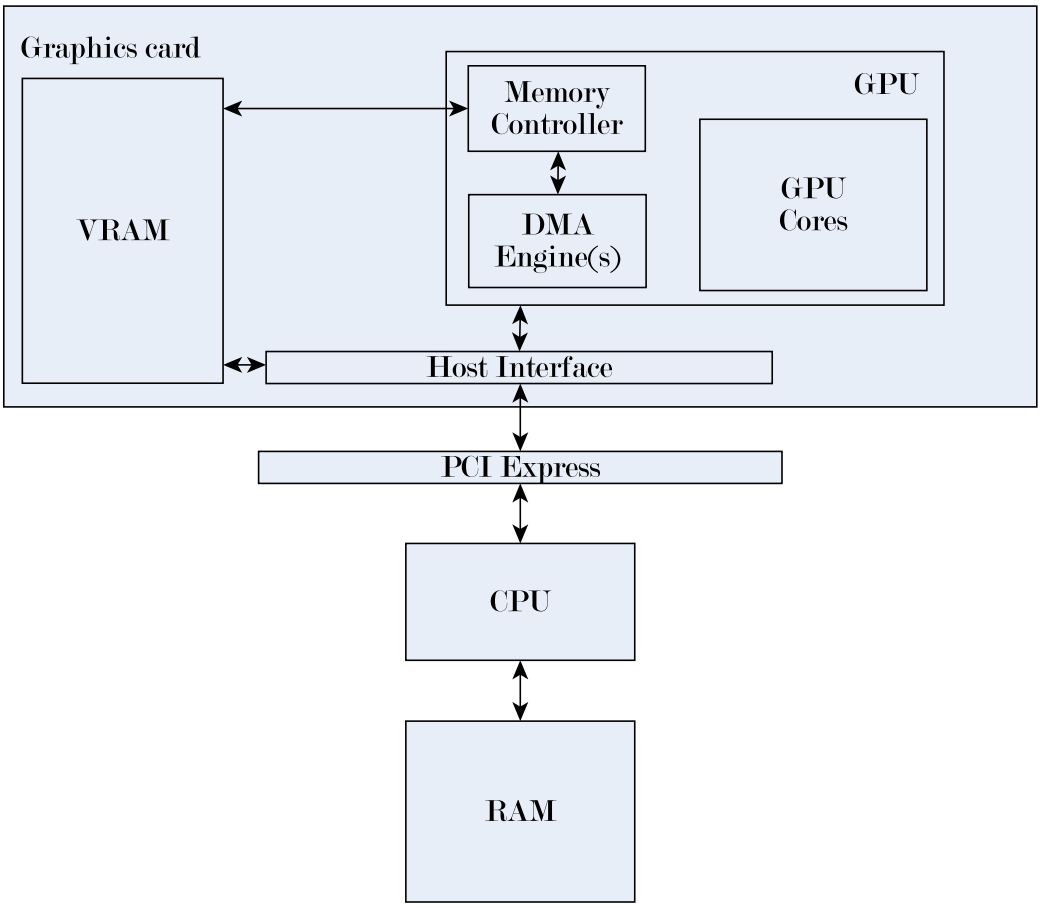


Figure 2.7: Schematic overview of involved hardware components, based on [KAB13]

Direct CPU Transfer The *CPU* takes the leading role for **Direct CPU Transfers**; it loads the data from main memory and copies it to the *Graphics Card*. Through PCIe base address registers (BARs), it can directly write into device memory. The path taken is illustrated in Fig. 2.8, where the data is moved from *RAM* via *CPU*, *PCI Express* and *Host Interface* to *VRAM*. Yet, we do not know what other parts of the *Graphics Card* might be involved when taking this approach.

Pinned Transfers Another possibility using PCIe BARs operates on host memory. Here, the *GPU* is granted direct access to pinned (host) memory to allow **Pinned Transfers**. This part of the host memory is page-locked so it cannot be swapped out [HM12]. Using pinned memory allows two paths to take as illustrated in Fig. 2.9: (2a) **Direct** – Either the *GPU* reads directly from *RAM* via *Host interface*, *PCI Express*, and *CPU* or (2b) **DMA Based** – the *DMA Engine(s)* transfer data via the same components to *VRAM*.

For the latter so-called copy engines are used. The DMA transfers can either happen

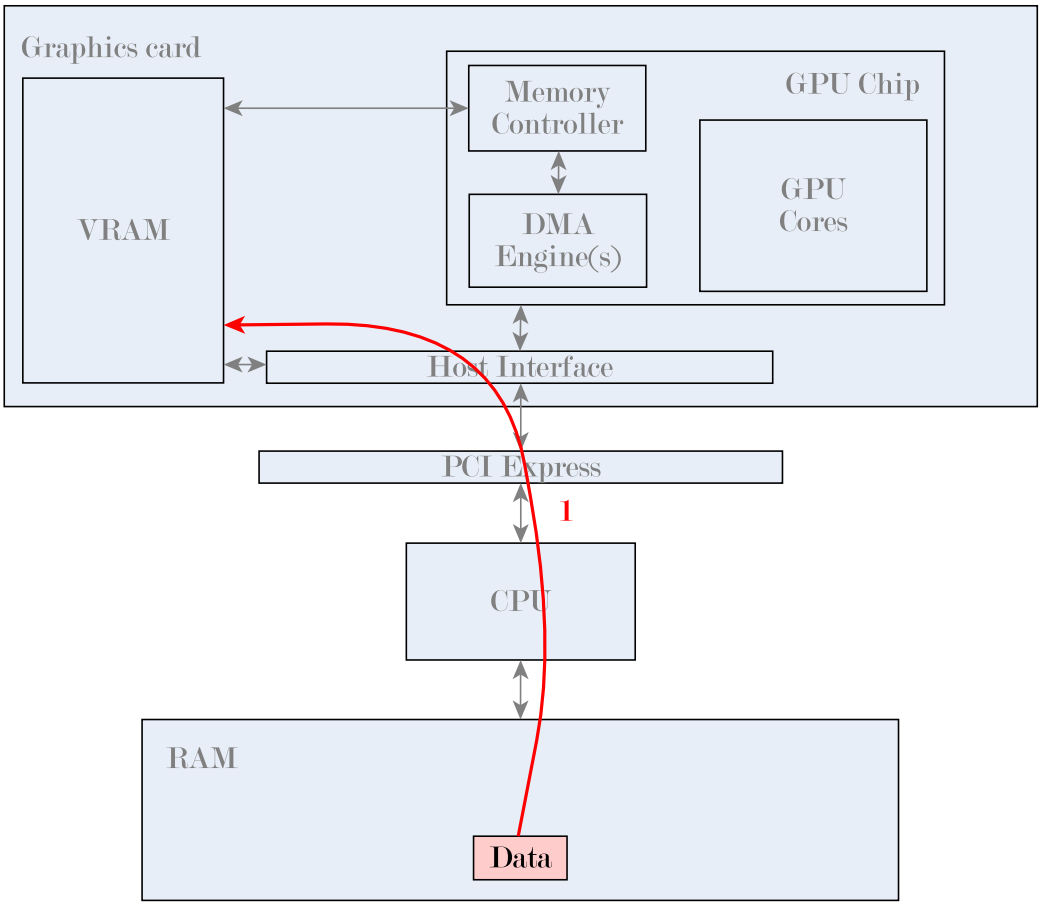


Figure 2.8: Schematic of direct *CPU* to *VRAM* data transfer

asynchronously from the *GPU* or halt *GPU* processing for the duration of the transfer, depending on the *DMA Engine(s)* involved [HM12, Ven10]. However, for this to work, data must already reside in pinned memory, which usually requires a transfer from pageable host memory to pinned memory, see also path (1) in Fig. 2.9.

A third possibility using pinned memory with various manifestations – **Microcontroller Based** – uses other microcontrollers residing in the *GPU* for copying data. Here, data is read by these microcontrollers from pinned memory and in a second step written into device memory. Fujii et al. [FAN+13] however mention that this part is usually limited to a few hundred bytes and therefore, for our cases, is not relevant.

Unless the *GPU* directly reads data from host memory, data is transferred from device memory via a high bandwidth onboard bus from *VRAM* to the *GPU* and there processed to render the final image.

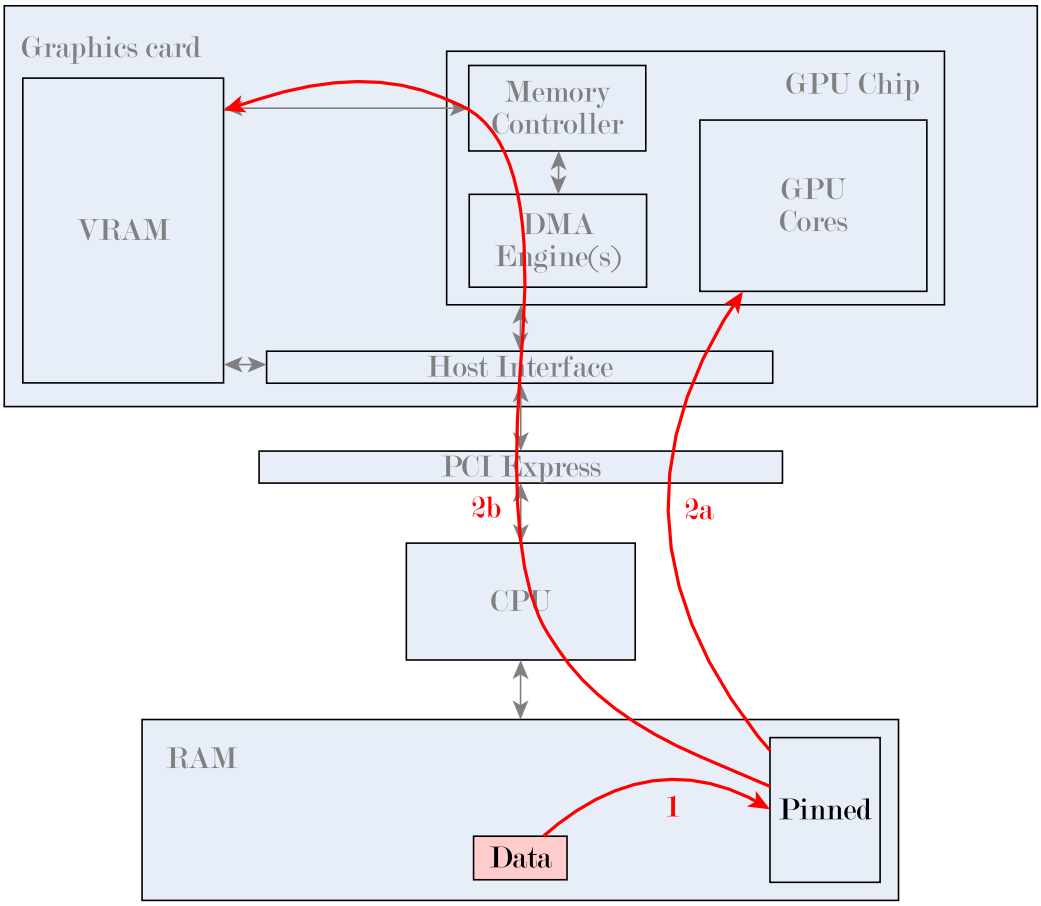


Figure 2.9: Schematic overview of involved hardware for data transfer from host memory to pinned memory. From there, data can be either directly accessed by the *GPU* or copied to *VRAM*

2.2.2 Software

To use either of the paths, we need to communicate with the graphics hardware and be able to design applications that make use of graphics hardware. Graphics APIs, such as Vulkan[2], DirectX[3] or OpenGL[4] can be used for this purpose. In this work we focus on OpenGL for the following reasons: DirectX is a Windows only API while OpenGL is usable on Windows, Linux and MacOS systems. Vulkan is similar to OpenGL but more explicit and allows to have more control over the rendering pipeline [Guh18]. However, Vulkan is not natively available for MacOS systems, it can be used only via an interface layer with possible additional overhead. As OpenGL is available for all three main computer

[2] `https://www.khronos.org/vulkan/`
[3] `https://docs.microsoft.com/de-de/windows/win32/directx`
[4] `https://www.khronos.org/opengl/`

platforms, we focus on OpenGL in this work.

Uploading and Rendering Steps

For OpenGL, there are several possibilities on how a data upload from main memory to graphics memory can be programmed. Each of them requires a couple of steps to be taken:

Generation For all cases we first need to generate a name for a so-called vertex buffer object (VBO). This VBO, or simply buffer, is a generic memory object that holds data in memory that is available to the GPU (usually in graphics memory).

Allocation Generating the name however does not actually create a data object. This is done in a second step, the allocation step. Here we can choose to issue an asynchronous (for the calling thread on the CPU) data transfer or just to allocate memory.

Upload If memory is only allocated, there are two further options: Either getting a pointer to the allocated memory, copying it using the `memcpy` function and then freeing the pointer, or issuing an asynchronous transfer (asynchronous for the calling thread on the CPU).

Rendering is issued by first specifying which shaders to use for both vertices and fragments, specifying vertices to render and specifying where to write the finished pixels. Shaders are programs executed, depending on the type, for each vertex, primitive or fragment that define how it should be processed. Primitives can be triangles, quads or other geometric shapes that describe a surface to be rendered. Fragments themselves are rasterized primitives, in other words pixels to be. This means that fragments can become pixels of the image shown, but not necessarily, as other fragments could be drawn over them. More information about shaders can be found in [Guh18, p. 470ff], about fragments in [Sel13, p. 41f] and primitives in [Sel13, p. 10f].

OpenGL Specifics

Next we take a more detailed look on how OpenGL itself is specified and how we assume it is implemented in driver and hardware by graphics card vendors.

The Command Queue A graphics card executes commands asynchronously from the rest of the computer [SKS16, p. 589]. Therefore, OpenGL implements a so-called command buffer or command queue. This queue provides an interface between the CPU and GPU. When the CPU issues commands to be processed by the GPU, those commands a pushed into the command queue. This helps to reduce waiting of CPU and GPU as they do not need to wait on each other in many cases. An example is issuing a drawing command, meaning, for example, rendering a triangle mesh. After finishing preparation of the mesh for rendering, only a couple of non blocking (for the calling CPU thread) calls are executed

on the CPU to start drawing this mesh. This means that while the GPU processes the commands in the command queue, the CPU is free to execute other tasks.

However, using a queue forces the GPU to process all commands in this queue sequentially [HM12]. This means individual drawing commands can be processed sequentially. While the vertices of one mesh are processed in parallel, several meshes in different drawing commands can be processed like they were issued – one after the other.

The Rendering Pipeline OpenGL specifies a rendering pipeline as an abstract model to describe how rendering happens. This rendering pipeline describes how vertices are processed, combined to primitives, rasterized to fragments (if necessary), and then how the fragments are processed to final pixels of the rendered image. There are many more details to be discovered in this pipeline, such as various rendering layers, different programmable stages or optimization techniques. However, most of them are not relevant for this thesis and are therefore skipped. The interested reader is referred to [Guh18, Sel13, SKS16] for more information about the rendering pipeline.

The OpenGL Context A variety of different states define how certain steps or which steps of the rendering pipeline are executed. States are a set of variables that can be changed and queried via the OpenGL API. The whole set of states is defined in a so-called context. An OpenGL context is needed for issuing OpenGL commands and it is created via windowing libraries. There exist different windowing libraries e.g for Windows and Linux based systems. They allow to create windows that have an OpenGL context associated with them. For more information on context and window creation, the interested reader is referred to [Sel13, p. 623ff].

An OpenGL context can only be *current* in one CPU thread, which means that at one certain point in time, only one thread can issue an OpenGL command in this context. However, when the *CPU* is too busy doing other tasks and not able to fill the command buffer for the GPU fast enough, it can make sense to use multiple threads to avoid that the GPU is waiting for new tasks and thus avoiding slowing down performance. Hrabcak and Masserann have shown in [HM12] that it is possible to gain performance if data transfers are overlapped with rendering.

Another possibility also detailed in [HM12] is to use multiple contexts that share **some** part of the whole state set. An example for shared states are memory objects. This means that we can outsource data transfers to a different thread to avoid stalling of the GPU while the *CPU* is copying data to and from graphics memory.

In all cases, each application only has one command buffer, irregardless of the number of contexts. However, some commands can be processed concurrently on graphics hardware which in turn, possibly allows overlapping of data uploading and rendering.

Complications However, either way, there are a couple of details involved that make data uploads using OpenGL more complicated:

- When we issue an asynchronous transfer, we need to make sure that the transfer is finished before we use it for a draw call. If not, OpenGL implicitly synchronizes transfer and rendering by letting the rendering command wait until the transfer is finished [HM12]. We can make sure that a transfer is finished by using synch objects. Synch objects are inserted into the command buffer. When all commands are executed that were inserted into the command buffer before the synch object was inserted, then the synch object signals the CPU, implying that all commands before are executed. This allows to synchronize the part of the application being executed on GPU with the part of the application executed on the CPU.
- When we issue a data transfer, via either option, implementations can copy the data to pinned memory. As pinned memory is memory in host memory reserved for the GPU and accessible by the GPU, this makes the issued transfer a host to host copy. If we want to use this part of memory for rendering, a second data transfer is issued that copies the data either from host memory to graphics memory or directly to the GPU before it is used for rendering [HM12].

Timing Considerations and Overlapping

Overall, there exist multiple possibilities to design data transfers in parallel to rendering, e.g. using the before described possibilities of the steps necessary to upload and render. Depending on which we choose, in the worst case we can get an overall time t_o for rendering a dataset or uploading a dataset as the sum of time needed for the transfer t_u and the time needed to render it t_r, i.e. $t_o = t_u + t_r$. The reason for this can be that either we need to wait for the data transfer to be finished before rendering can start, or a draw call blocks the GPU and consequently, stalls any uploading process (which means that the uploading process needs to wait for the draw call to finish). Fig. 2.10 illustrates this with the first row of blocks showing an uploading process waiting for rendering to finish and the second row vice versa. In both cases, the total time t_o, shown as third row, is the combination of both times t_r and t_u.

For Fig. 2.10 (and the following figures), dark gray blocks symbolize the time needed for uploading and brown blocks symbolize the time needed for rendering a particular part of a dataset. Additionally, light gray blocks below the brown and/or gray blocks illustrate the total time of either both processes combined or until the next frame is displayed to a user, which is in most cases the time combined of both processes. In some cases, user interaction or other tasks are symbolized by light yellow blocks, pose tracking events by black blocks. Pose tracking means a technology that allows to determine the position and orientation – the pose – of an object in the real-world and track it. Usually, the poses can then be accessed by the application or are signaled by an event system to the application.

A better solution will overlap both uploading and rendering, so in the best case the total time for rendering and uploading equals the maximum time needed of either of the processes: $t_o = \max(t_r, t_u)$. See also Fig. 2.11, where the first row shows time needed for rendering and the second row time needed for uploading. The third row shows for both

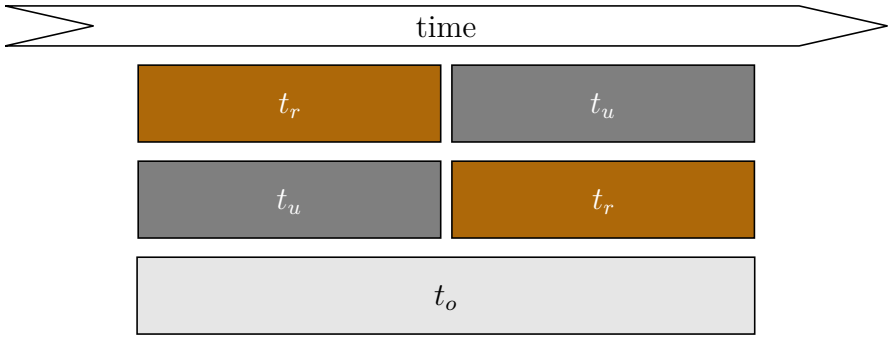


Figure 2.10: Sequentializing uploading and rendering leads to a total time $t_o = t_r + t_u$

steps (left and right column) the maximum time of either of them. We assume that the maximum time is used as the worst case scenario. This is due to the nature of OpenGL. We can only use sync objects for getting a signal if processes are finished. The synch objects possibly require GPU involvement, which means that we have to wait for both processes to finish, when we issue them at the same time and insert a sync object after each (as they use the same command buffer).

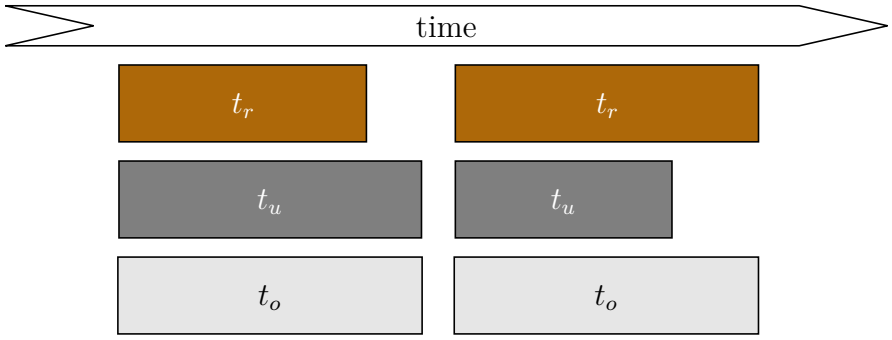


Figure 2.11: Overlapping uploading and rendering leads in an ideal case to a total time $t_o = \max(t_r, t_u)$

2.3 Decoupling Rendering and Uploading

For uploading and rendering a dataset it is not necessary that the steps involved are performed one after another. Furthermore, while a dataset needs to be present in graphics memory (this can also mean pinned memory) to be rendered, this does not mean that both uploading and rendering need to be alternated. If we decouple the two processes, we are able to plan more flexible how the two processes are performed.

Consequently, we analyze in this section how to decouple both, rendering and uploading, on the CPU side of the application as well as on the GPU side of an application. This is done in the following four steps:

(1) Conceptual Framework

(2) Implicit Synchronization

(3) Decoupling Using Threads

(4) Decoupling Using Copy Engines

Conceptual Framework gives a brief overview on the implemented framework that allows us to decouple rendering and uploading. **Implicit Synchronization** describes the concept of implicit synchronization and how to avoid it to decouple dependencies on the GPU side of the application. **Decoupling Using Threads** analyzes how to decouple dependencies on the CPU side of the application. This is achieved by using several CPU threads for the two processes, rendering and uploading, and a strict separation (as far as possible) of data in graphics memory and data in main memory. Both, avoiding implicit synchronization and decoupling using threads, only decouple rendering from uploading when the two processes are executed in parallel. Therefore, in **Decoupling Using Copy Engines**, we analyze the usage of copy engines und how they allow to decouple any further dependencies.

(1) Conceptual Framework

The general idea in our framework is two have two separate processes that depict the two stages of the data to image process: *Rendering* and *uploading*. Every part of a dataset passes through these two processes. This technique, to the best of our knowledge and excluding our own work, has not been scholarly documented.

Using these two processes, we can render a dataset that is already present in graphics memory more than once, while another part of the application uploads the next dataset. This way, the visualization application can still be responsive to user input for moving within a virtual scene. On the other hand, when uploading is done quicker than rendering, we can also preload parts of a dataset that might be needed in the future, to avoid urgency for uploading and along with that, possible performance losses for rendering.

(2) Implicit Synchronization

Implicit synchronization occurs when a dataset is both issued to be uploaded and rendered. In the case that uploading is not finished at the time the rendering should take place, rendering has to wait until the upload is finished.

An additional potential conflict is that both, rendering and uploading, access graphics memory. The access pattern however is different for both processes. Rendering usually (excluding more advanced rendering techniques) only reads from graphics memory while uploading writes to it. This means, concurrent access might be possible.

To decouple any dependency caused by implicit synchronization between both, rendering and uploading, we need to make sure that an upload is finished, before we use that part of the memory for rendering. This can be done, as aforementioned, with synch objects. If and only if an upload is finished (as far as the driver lets us know this; the driver might

hide this using pinned-memory and perform a copy when needed to graphics memory), the reference to the now in graphics memory residing dataset is given to rendering. In other words, already present datasets are reused by rendering as long as new ones are not finished uploading. Once uploading is finished, rendering replaces the before used datasets with the new ones. It then uses these datasets until the next dataset upload is issued and finished.

This makes sure that rendering is unaware of any data uploads while uploading is unaware of any data rendering. The latter is achieved by designing the processes to be in a pipeline.

(3) Decoupling Using Threads

To achieve a decoupling also on CPU side, both processes have their own thread. Uploading does only need to know about which datasets to upload and where they are in host memory. Rendering only needs to get the reference to the data in graphics memory.

The question what needs to be uploaded usually is depending on virtual spatial position, temporal position or on user input. All of them can require prediction of what positions are needed next or what a user might input. As these can be computational expensive and require an interface to both rendering and uploading, we need to separate all tasks related to finding out what needs to be uploaded next into a separate thread to minimize interference on rendering and uploading.

(4) Decoupling Using Copy Engines

Apart from separating tasks on CPU side, one critical bottleneck is the command buffer for the GPU. As all commands issued for execution on one graphics card for one OpenGL application can be sequentially executed, both rendering and uploading can not be separated completely. However, as we try to make use of the copy engines on the graphics card, the actual copy call can be outsourced so that it does not block subsequent commands in the command buffer.

If we have more than one upload pending and more than one copy engine available, this can mean that we can also parallelize uploading. However, before we can upload data, we need to prepare memory in graphics memory, which are commands that are processed via the command queue. Yet, there might be drawing calls being processed and blocking further commands in the pipeline. On the other hand, creation of buffers is usually quicker than the actual uploading process. To maximize uploading performance, we can also separate the preparation of buffers from uploading using an additional thread. If we combine both ideas, parallelizing and separation of threads, this leaves us with 1 to k threads for uploading data and one thread preparing buffers.

Having more threads for uploading than available copy engines might also allow us to use all available copy engines, while some threads are waiting for a synch object that was issued after a draw call (due to possible racing conditions).

Once uploads are finished, only the reference to the buffer needs to be given to rendering. This reference is usually an unsigned integer and therefore, the influence of this task on performance is negligible.

2.4 Parameters and Input Space

From the sections before, we see there are many components involved in uploading data from host memory to graphics memory and in rendering data using the GPU. In order to optimize for either of the optimization targets we need to know what can be adjusted. In this section we analyze the involved hardware and software components and describe the possible influences of them on the two processes rendering and uploading.

We start by analyzing hardware related bottlenecks and controls. As of next we discuss possible influences of software design.

2.4.1 Hardware

As described in Section 2.2, a couple of hardware components are involved for rendering an image of a dataset. This part analyzes at first the **Data Bandwidth** between and of the individual components, to find out which components represent bottlenecks.

This is followed by **Variable Factors**, which describes possibilities to adjust hardware and thus can have an influence on performance. Both parts are structured in the order of data movement, starting from host memory up to the GPU as final component:

- Host memory
- CPU
- PCI Express
- Grapics card, with:
  - Graphics memory
  - GPU

Data Bandwidth

The components involved for data transfers are sketched in Fig. 2.7 in Section 2.2.

Host memory (*RAM* in Fig. 2.7) is the first component involved. The type, latency timings and clock rate can play an important role for read rate and therefore limit uploading performance. However, read capability of the used host memory usually exceeds theoretical bandwidth of the PCIe bus (*PCI Express* in Fig. 2.7), i.e. for PCIe version 3 with 16 lanes this means that the CPU can read data from host memory with a higher rate than 15.75GB/s. This is theoretically already achieved using DDR3-2000 host memory with an access rate of 16 GB/s. We further disregard access latency as it usually lies in the range of a couple of ns, while we focus on transfer times in the range of µs to ms.

The second station is the CPU itself which usually is bound in read rate only by the main memory and therefore also not a bottleneck for the whole transfer.

The most limiting component in the host system is the PCIe bus. Here, factors include the type or version and the number of lanes used. Graphics hardware is in the best case (currently) connected with a PCIe 16x slot. At the time of writing this thesis, only one graphics card supports PCIe version 4 with double the bandwidth of its predecessor, PCIe version 3. Most available graphics cards use the older standard which has a transfer rate of about 15.75 GB/s.

The next steps are the graphics hardware (*Graphics Card* in Fig. 2.7) and its components. One important factor can be the number of copy engines and the memory controllers. The used graphics memory (*VRAM* in Fig. 2.7) is depending on architecture, vendor and version of a graphics card. This translates to which GDDR version is used, how big the capacity of the memory is and at which clock rate it can be or is operated. The total memory bandwidth specifies at which rate data can be read or written to graphics memory. Usually, this number is a lot higher than the bandwidth of the PCIe bus. For instance, one of the graphics cards used in the Chapters 5, 6 and 7 is the NVIDIA Quadro RTX 4000. Its memory bandwidth is up to 416 GB/s, which is more than 25 times the bandwidth of the maximal possible bandwidth of 15.75 GB/s for PCIe Version 3. The GPU read rate is only bound to the read rate of graphics memory.

Variable Factors

The obvious variability for hardware comes by choosing different components. Using a different CPU, host memory, PCIe Version (e.g. mainboard) or graphics card can have a significant impact. However, there are more options available besides changing components. For all involved hardware components, clock cycles play an important role for data transfers.

Often the same main memory can be used with different clocks. Using different clocks directly translates into different data access rates. For example, operating DDR4 with a bus clock of 1500 MHz (i.e. 375 MHz internal clock) give a theoretical bandwidth of 24 GB/s while the same module with a bus clock of 2600 MHz gives 41.6 GB/s of theoretical bandwidth. However, changing the clock rate is in most cases not possible via software solutions but only via BIOS options.

Changing the clock of the CPU is nowadays done automatically by the operating system or micro code depending on the current work load. Another option is to change the clock rate via software, which is an option that the operating system can allow. The clock rate can have an important impact on data uploading and rendering, especially when many CPU instructions are involved for issuing them. An example might be to have many small data datasets (instead of a few large) that all need to be uploaded and rendered. Each of these requires the CPU to put commands in the command buffer for allocating memory on the graphics card, set up the different states for each data set and issue a draw call.

Changing the version of the PCIe bus, as it is the most limiting factor, has an

important impact on transfer rates and therefore on rendering and uploading performance. However, this is, if available, only available via BIOS options.

The graphics card itself has usually two options for clock rates: GPU and graphics memory clocks. Both can have an impact on rendering and uploading and can be changed via software for some of the currently available (and in this work used) graphics cards.

2.4.2 Software

The most optimized hardware cannot produce the best performance, if used incorrectly. Therefore, software design plays an important role. Besides the used system software, i.e. the operating system, the drivers and the used graphics API, there are many choices when designing an application that allows to transfer data from main memory to graphics memory and render it there. In this subsection we describe these choices in the following order:

- Multi-Threading
- Uploading Method
- Buffer Usage Hints
- Direct State Access
- Pixel Buffer Objects
- Dataset

Multi-Threading One of the factors already mentioned before is the choice of using multi-threading or not. Hrabcak and Masserann [HM12] have already shown that using a multi-threaded approach can boost performance. To also make use of this performance boost, we use the before introduced multi-threading approach for this work. However, for our approach we additionally have the possibility to specify how many concurrent data uploads can happen at the same time, i.e. how many threads are used for uploading.

Uploading Method Furthermore, OpenGL itself provides a set of possible factors to tune. There are several uploading methods comprised of non deprecated functions in modern OpenGL to upload data to graphics memory:

- `glBufferData`
- `glBufferSubData`
- `glMapBuffer + memcpy + glUnmapBuffer`
- `glMapBufferRange + memcpy + glUnmapBuffer`

Apart from `glBufferData`, each of them requires that a buffer is allocated before the actual upload. This can be done by using `glBufferData` without uploading any data. The function signature of `glBufferData` is as follows:

```
void glBufferData( GLenum target, GLsizeiptr size,
                   const void * data, GLenum usage );
```

`GLenum target` is an enumerated type that defines the binding point. We will take a closer look at binding points for Pixel Buffer Objects below. `GLsizeiptr size` and `const void * data` define the size of a dataset and a pointer to that dataset, respectively. The pointer to the dataset is only needed to be not NULL if a data transfer should be issued with a call to `glBufferData`. `GLenum usage` is an enumerated type to specify buffer usage hints, which will be discussed in the following.

Buffer Usage Hints The function `glBufferData` itself allows to specify buffer usage hints. These buffer usage hints can be specified to indicate, how we plan to use the buffer. In total there are 9 different possibilities generated by the combination of usage and frequency. Usage describes how the application intends to use the buffer. The possibilities are `READ` for reading data from a buffer to the application, `DRAW` for the buffer is modified by the application and used for rendering, meaning being processed by the GPU, and `COPY` for copying data within the realm of the graphics card. An example for the latter would be the GPU generating data, writing it into a buffer and then using it for rendering. The frequency describes how often a buffer is intended to be used. It can be `STATIC`, for a buffer being modified once and used often, `STREAM`, for a buffer being modified once and only used a couple of times, and `DYNAMIC`, for a buffer being modified and used repeatedly [Sel13, p. 92f].

Direct State Access Furthermore, using binding points or not can have an impact on performance. This can be implemented by using either the function `glBindBuffer` or, available since OpenGL version 4.5, by using *named* functions of the four uploading methods above:

- `glNamedBufferData`
- `glNamedBufferSubData`
- `glMapNamedBuffer + memcpy + glUnmapNamedBuffer`
- `glMapNamedBufferRange + memcpy + glUnmapNamedBuffer`.

Using the *named* version is also known as direct state access (DSA).

Pixel Buffer Objects When using `glBindBuffer` the binding point or buffer binding target itself can play an important role. For vertex data, usually `GL_ARRAY_BUFFER` is used. For asynchronous pixel data transfers (texture transfers), the so-called pixel buffer objects (PBO) were introduced. These can also be used instead of `GL_ARRAY_BUFFER` when modifying buffers.

Dataset Another important factor is the dataset itself. While we usually cannot change the size of a dataset (unless we remove or add information), it will have an impact on performance. What we can change, however, is how this dataset is structured: Is it one big dataset or is it subdivided into many small ones? Especially when more than just position information is given, e.g. colour information of the vertices, we need to decide if these different properties are saved in individual buffers or together in one big buffer.

2.5 Chapter Summary

In this chapter we describe how data can be uploaded from host memory to graphics memory and from there to the *GPU*, where it is rendered into an image. We can see that this process consists of two stages. These are *uploading*, which describes the transfer from host memory to graphics memory, and *rendering*, that describes transferring the data from graphics memory to the *GPU*, where it is processed into an image.

This structure can impact performance goals for several visualization scenarios. We describe two extreme ends of a visualization priority continuum, VR visualization and movie visualization. A third instantiation are desktop visualizations, which in turn reside somewhere between the former two. These three visualization types also describe three use cases and their performance goals and priority requirements. With that we answer **SRQ1**: *What are use cases for visualization and their performance requirements addressing **RQ**?*

However, graphics hardware usually works asynchronously from the CPU. OpenGL and other graphics APIs use command buffers, to collect *work* for the GPU. Yet, as OpenGL only implements one, this enforces a sequential execution of all commands in this command buffer [Sel13, p. 612]. But, we can overlap *Rendering* and *Uploading* as graphics hardware can have copy engines. Yet, we need to avoid rendering a not yet fully uploaded data set, as this would force implicit synchronization, i.e. the GPU waiting for the upload to finish. We solve this by using old buffers as long as new ones are not finished being uploaded yet and only replace after the uploads are finished. Furthermore, as some of the commands initiating a data upload block the CPU, we can use multiple threads to allow the use of more than one copy engine if available. This answers **SRQ2**: *How can data be uploaded to graphics hardware while concurrently rendering earlier data?*

All these steps and the chosen system allow for a variety of tuning parameters. At first the choice of the involved hardware components:

- CPU
- RAM
- Main board and with it the PCIe Bus, the number of lanes and its version
- Graphics card/graphics hardware

Furthermore, there are customizable parameters via software for the hardware:

- CPU clock rate

- GPU clock rate
- Graphics memory clock rate

On the software side, first we have the system software that can influence performance:

- Operating system
- Drivers of the graphics hardware (Open/Closed source and version)
- Used API

Once these are chosen and using the described framework, there are a couple of parameters that can be configured

- Number of threads used for uploading
- Uploading method used for data transfer
- Buffer usage hints
- Using binding points or not

All these choices describe an answer to **SRQ3**: *What parameters can be controlled and what are other influences?*

The choices themself create an enormous input space for possible fine tuning if we construct all possible permutations of them. Before we discuss how to design experiments to reduce the number of measurements, we will first take a look at related work and their approaches to quantify the influences of this input space in the next chapter.

Chapter 3

Related Work

Transferring data to and from graphics hardware can be achieved in two ways: using APIs aimed at graphics rendering like Vulkan, DirectX or OpenGL or using APIs aimed at general purpose computation on graphics processing units (GPGPU) like CUDA or OpenCL. Both approaches require fine tuning of various factors to achieve the best performance.

In this chapter we analyze relevant related work focusing on data transfers for graphics hardware. This analysis is structured in

- Related work using GPGPU APIs
- Related work using graphics APIs.

Subsequent, we summarize our findings and discuss them in the context of this work's goal.

3.1 Performance of Data Transfers of GPGPU APIs

Although not directly related to visualization, relevant related work can also be found for GPGPU APIs. While compute performance of these APIs is studied in a wide range of approaches and application examples, Boyer et al. [BMK13] are one of the first to also consider transfer times for predicting compute performance using GPGPU APIs with GPUs. For that, they expand their performance prediction framework GROPHECY [MMK+11] to analyze data usage and the need for data transfers. Given a projection of how much data needs to be transferred from and to graphics memory, they are able to approximate transfer times using the linear equation

$$T(d) = \alpha + \beta d \tag{3.1}$$

with $T(d)$ as the expected transfer time for d bytes, α as overhead for PCIe transfers and β as the inverse of transfer bandwidth (of the PCIe bus). With this expansion, Boyer et al. are able to better predict how much porting an application to a GPU implementation speeds up execution of said application. In numbers, they are able to reduce the error for the speedup prediction for four examples from 255% to 9% .

While Boyer et al. modeled the transfer of data as individual process, newer graphics cards, drivers and GPGPU APIs (or versions thereof) allow data transfers to overlap with calculation, similar to overlapping uploading with rendering, as illustrated before. Gómez-Luna et al. [GLGLBG12] analyze this kind of behavior for CUDA streams. In their analysis they deduct two kind of experiments: At first, they use a fixed data size for transfers to and from graphics memory with a variable computation time; as second, they use a variable data size for transfers to and from graphics memory with a fixed computation time. For both experiments they analyze the resulting total time (for upload, download, and computation combined) and deduce a performance model. This model is based on measured parameters such as fixed offsets, execution time, and timings for uploading and downloading data and a particular compute capability, which describes the set of compute features that are available for the given GPU. For hardware with compute capability 1.x they describe the following formula as performance model:

$$t = t_T + t_{oh}, \qquad \text{if } t_T > t_E + \frac{t_T}{nStreams} \tag{3.2}$$

$$t = t_E + \frac{t_T}{nStreams} + t_{oh}, \qquad \text{if } t_T < t_E + \frac{t_T}{nStreams}, \tag{3.3}$$

with t being the resulting execution time, t_T as time needed to transfer the data, t_{oh} as overhead for each stream in the total number of streams used $nStreams$ and t_E as time needed to process the data.

For hardware with compute capability 2.x their model changes to

$$t = t_{Thd} + \frac{t_E}{nStreams} + t_{Tdh} + t_{oh}, \qquad \text{if } t_{Thd} > t_E \tag{3.4}$$

$$t = \frac{t_{Thd}}{nStreams} + t_E + t_{Tdh} + t_{oh}, \qquad \text{if } t_{Thd} < t_E, \tag{3.5}$$

here additionally with t_{Thd} as time needed to transfer data from host to device, t_{Tdh} to transfer it back. Additionally, the authors show how they are able to deduce the optimal number of streams for $nStreams$ and validate their model using three example applications based on the CUDA SDK.

Van Werkhoven et al. [vWMSB14] extend this type of model with more parameters: They include the number of copy engines, if implicit synchronization happens or not, and the PCIe Bus version. This allows them to more accurately predict performance for a number of example applications and to estimate the optimal number of streams more precisely.

Fuji et al. [FAN+13] get a step closer to the hardware: Instead of testing different CUDA functions or frameworks on top of CUDA, they adapt how CUDA functions themselves are transferring data. This is achieved by adapting an open source driver. In total Fuji et al. investigate four different types of data transfers, where one has three different sub variants based on so-called graphics processing clusters (GPCs):

1. DMA engine based

2. I/O remapping based
3. Indirect memory-mapped based
4. Microcontroller based
    a) One GPC individually
    b) Four GPCs in parallel individually
    c) All GPCs with broadcasting

Type 1 of data transfers issues GPU commands that cause the DMA engine(s) to copy data from or to device memory. Type 2 uses the I/O remapping function, after the GPU has allocated the necessary memory region. Type 3 uses indirect access of memory-mapped I/O space. Here, the CPU can write and read to and from host memory, that itself is implicitly mapped to device memory. This in turn means that the CPU can indirectly read and write from and to device memory. Type 4 uses the microcontrollers that control a certain number of CUDA cores. The used graphics card, a NVIDIA GTX 480, has 4 of these GPCs. For the first of the sub variants, a), this microcontroller is used individually. For the second, b), four of them are used in parallel and for the third, c), another microcontroller is used that broadcasts commands to them. For all those three sub variants, in contrast to type 1,2 and 3, the microcontrollers are performing the actual data transfer.

Fuji et al. analyze these four types of data transfer mechanisms by varying a couple of parameters and averaging the required time to transfer data of 1000 measurements. The parameters tested are:

- Data size, ranging from 16B to 64MByte
- Use of real-time tasks within the possibilities of the used Linux kernel
- Stress tests, with another process either allocating memory, performing I/O system calls with pipes or creating high CPU workload
- Single transfers and double transfers

The authors conclude that either standard DMA (the second type) or I/O remapping (the third type) perform best depending on the data size.

These previous approaches on analyzing data transfers for GPGPU, show us another possibility on how to approach this task. The main goal of their analysis is to make data transfers predictable, so that the time needed for data transfers can be included in considerations about whether using GPUs for general purpose calculations is worth porting an application.

3.2 Performance of Data Transfers of Graphics APIs

One of the earliest systematic approaches to analyze data transfers to graphics memory was performed by Buck et al. [Buc04]. They provide a benchmarking framework to measure

a multitude of processes relevant to GPGPU, such as memory bandwidth, data upload or numerical precision. Although the purpose of their work is GPGPU, they are using OpenGL as no other APIs were widely supported or available at that time. They analyze how textures can be copied from and to graphics memory. For these data transfers, they compare the performance of fixed and floating point representations, color component ordering, and the performance of choosing one, two, three, or four color components.

As OpenGL has undergone numerous changes and redevelopments with various new versions and changes in its API – the newest version as of today, 4.6, was introduced 2017 – newer works needed to re-evaluate data transfers using newly introduced concepts or functionality to give a more complete picture on performance of data transfers. Therefore, Grottel et al. [GRE09] started to systematically benchmark various factors for a scenario, where parts of a dataset have to be uploaded to the graphics card in order to render it for the current frame. The benchmarked factors in [GRE09] consisted of the following:

- OpenGL function used:
  - Immediate mode using `glBegin` and `glVertex*` functions (deprecated since OpenGL 3.0)
  - Vertex Arrays using `glVertexPointer` (deprecated since OpenGL 3.0)
  - `glBufferData` with 3 different usage hints: `GL_STATIC_DRAW`, `GL_DYNAMIC_DRAW`, `GL_STREAM_DRAW`
  - `glMapBuffer`, with only positional data, with color data as an additional buffer, and with color data interleaved in the position array
- Four computer systems with different CPUs
- Five different graphics cards with different GPUs

The authors experiment with these factors in various combinations to see which perform best. For this work, performance is measured by the time needed for both uploading and rendering a dataset together. This means that the authors measure the time starting from uploading the data to graphics memory until the rendering of this data finishes. Evaluating their experiments, the authors conclude that for most cases, the deprecated functionality using `glVertexPointer` yields the best results.

In [FGKR16], Falk et al. extend the work in [GRE09] with shader storage buffers and come to similar conclusions.

Hrabcak and Masserann show and evaluate in [HM12] several ways of uploading and downloading data to and from graphics memory. They compare various factors:

- OpenGL function used for uploading to multiple already allocated buffers:
  - `glMapBufferRange` with the flags `GL_MAP_WRITE_BIT` and `GL_MAP_INVALIDATE_BUFFER_BIT`
  - `glMapBufferRange` with the flags `GL_MAP_WRITE_BIT` and `GL_MAP_FLUSH_EXPLICIT_BIT`

  - `glMapBufferRange` with the flags `GL_MAP_WRITE_BIT` and `GL_MAP_UNSYNCHRONIZED_BIT`
  - `glMapBufferRange` with the flag `GL_MAP_WRITE_BIT`
  - `glBufferData` to orphan an old buffer and `glBufferSubData` to upload data
  - `glBufferSubData` to upload data (without orphaning)
- Three computer systems with different CPUs
- Two AMD, one Intel and three NVIDIA graphics cards
- Three scenarios: Single-threaded with one OpenGL context, multi-threaded with one OpenGL context, and multi-threaded with two shared OpenGL contexts

Hrabcak and Masserann [HM12] conduct experiments to gain data on how the different OpenGL functions perform for the three scenarios on 6 different combinations of graphics card and computer systems. For that they also measure the time needed for uploading data and rendering together and compare the resulting times. They point out the differences of the used graphics cards and recommend to specifically profile applications on the used hardware and to adapt it to the used hardware to gain optimal performance.

While all of these works give a great overview on the different possibilities for uploading data while rendering, they illuminate only isolated cases and a particular set of all possible combinations of factors. Even when we combine all their results, possible factors, various configurations, and the range of scenarios are missing.

3.3 Discussion of Related Work

There exist multiple works describing performance modeling of data transfers and using graphics hardware at the same time, preceding this work. Those approaches search for an optimal time for both, transferring data and processing the data combined. They test different configurations, different hardware and multiple programmable paths for actually transferring the data.

Our goal is to broaden the view on this topic and include multiple different use case where different performance requirements and optimization targets are included. Furthermore, we describe how to systematically experiment and deduce a mathematical description for a given target system that allows to individually optimize for given optimization targets.

In Table 3.1 we compare several key concepts of related work. These key concepts are shown as columns and consist of the following:

API	Which API is analyzed?
Analyze Data Transfers	Are data transfers part of the analysis?

Multi-threaded Are multiple threads used for parallelizing transfers and processing and are structures decoupled?

Individual Process Analysis Are transfer and processing analyzed individually?

Statistical Experimental Design Is the design of experiments planned using statistical concepts?

The rows of Table 3.1 represent the previously in this chapter described approaches.

None of the presented works uses a statistical approach on planning experiments in an efficient manner. Works analyzing GPGPU APIs mainly focus on deriving a mathematical description of how transferring data to graphics hardware, processing it and transferring it back. In most cases they analyze the processes individually, meaning how long does the data transfer take and how long processing. For works focusing on graphics APIs, neither of those two aspects are analyzed. However, one work of those includes an analysis of using multi-threading for graphics applications, contrary to GPGPU focused works, where multi-threading is described to happen only on graphics hardware.

	API	Analyze Data Transfers	Multi-threaded	Mathematical Model	Individual Process Analysis	Statistical Experimental Design
[BMK13]	CUDA	✓	–	✓	–	–
[GLGLBG12]	CUDA	✓	–	✓	✓	–
[vWMSB14]	CUDA	✓	–	✓	✓	–
[FAN+13]	Driver Adaptation	✓	–	✓	only transfers	–
[Buc04]	OpenGL	✓	–	–	–	–
[GRE09]	OpenGL	✓	–	–	–	–
[HM12]	OpenGL	✓	✓	–	–	–
[FGKR16]	OpenGL	✓	–	–	–	–

Table 3.1: Comparison of key concepts of related work

Chapter 4

Experimental Design

This chapter describes how to design the necessary experiments for analyzing the performance of involved components. Performing measurements for all possible permutations is not necessary and not even feasible, as statistics provides approaches to increase the information gained when not measuring all possible permutations.
This chapter is structured as follows:

- Mathematical Model
- D-Optimal Design and Linear Regression
- Model Selection using Information Criteria
- Summary of all Findings

We first detail the mathematical model, that is assumed to describe the processes rendering and uploading. This, additionally to the general formula, includes what variables and how variables can have an influence on performance. As second, we discuss how to design experiments efficiently using D-optimal design and the concept of linear regression for model deduction. As third we discuss concepts that help to decide which model is a good fit for a number of experiments given a set of different models. Finally, we summarize and relate the findings of this chapter to **SRQ4**.

4.1 Mathematical Model

This section describes the mathematical model and the various choices that feed into it. This is structured by the following three aspects:

Model Formula	This describes the general formula we consider as foundation for the mathematical model.
Control Variable Choices	This describes which variables are considered to be part of the formula and that are checked whether they have an effect on performance.

Control Variable Variations	This describes the different levels or values considered when parameterizing the chosen control variables.

4.1.1 Model Formula

For this work we assume only the time needed for either uploading or rendering or both needs to be modeled. This means, the observed variable Y is the time necessary to perform either uploading or rendering. All other variables, such as GPU clock rate or data size, are variables we can control, and are therefore the control variables.

We further assume a linear model. Eq. (4.1) gives the general linear function, that describes the assumed relationship between our observed variable Y and the control variables. The control variables are included in X_i, where i is a placeholder to identify each of these control variables individually. As described, we assume a linear model. This means that the later estimated parameters β_i are linear. However, this can also mean that control variables are modeled nonlinear.

$$Y = \beta_0 X_0 + \beta_1 X_1 + \cdots + \beta_k X_k \tag{4.1}$$

When performing the experiments, we measure the time Y, that results when setting the control variables in X_i. Yet, we do not know the parameters β_i and the form of X_i. While β_i is linear, the form of X_i could have a linear, quadratic or other nonlinear forms of contribution.

Another point not known before is interactions between variables. For example, let us assume that both, using a certain driver version as well as choosing a certain uploading method, have each a specific impact on the observed time needed. However, they can also interact with each other, meaning that when choosing both together, their impact can be different than just adding up both. These kind of interactions are also modeled in the described function in Eq. (4.1).

As the variables X_i can also describe interactions of more than one variable, these interactions have their own parameter β_i that can be estimated. This means that k describes the number of all possible impact factors, including the number of control variables, their form variations (i.e. linear, quadratic, etc.) and all possible interactions thereof. All of them are called *explaining variables* below. Yet, before we come to finding solutions, we first need to decide which control variables to include.

4.1.2 Control Variable Choices

In Chapter 2, we see that many variables can potentially play an important role when considering concurrent rendering and uploading. However, we do not consider all variables for our experimental design, either because of their trivial influence or because they require a complete new analysis which is beyond the purpose of this thesis. In the following we detail which variables are included and which are not as well as the reasoning for that decision. This is done in the following order: hardware parameters, software parameters and dataset parameters.

Hardware Parameters

For the experiments and the evaluation, described in subsequent chapters, we include the following control variables related to hardware and assumptions of their performance effects, given in Tab. 4.1.

CPU clock rate	The CPU clock rate can be changed via software. When the GPU processes the commands in the command buffer faster than the CPU can fill it, we assume that rendering performance will decrease.
GPU clock rate	The GPU clock rate can be changed via software. Reducing the clock rate of the GPU reduces the amount of commands that can be processed in a certain time interval. We assume this has direct effects on rendering performance.
Graphics memory clock rate	The graphics memory clock rate can be changed via software. Reducing the clock rate of the graphics memory reduces access rate of graphics memory. We assume that this has effects on both rendering and uploading performance.
Graphics card	We include two distinct NVIDIA graphics cards that allow to set GPU and graphics memory clock rate. We assume, as their GPUs have different microarchitectures, that they differ in performance.

Table 4.1: Included control variables related to hardware

Software Parameters

For software parameters we decided to stick to operating system, driver and the used API. While we expect that the introduced methodical approach can be applied to a variation of them, doing so would require another in-depth analysis as well as extensive adaptation of our code basis which is beyond the scope of this work and serves no other purpose than re-applying the presented methodical approach. The variables included in the subsequent experiments related to software design, are shown in Tab. 4.2.

Number of threads for uploading	As the tested graphics cards have more than one copy engine, we assume the number of threads used for uploading data to the graphics card can greatly affect uploading performance. We also assume that this influences rendering.
Function used for data transfer	We suspect that different OpenGL function use different subroutines for the data transfer. While one might allow asynchronous uploading for the CPU, another could allow asynchronicity for the GPU.
Buffer usage hints	While there is no guarantee that buffer usage hints actually affect the used memory, we assume that vendors and therefore driver developers include the specified hints for deciding which type of memory on the graphics card to use. This would reflect in the performance of uploading, rendering or both.
Using binding points	If using binding points or not has an effect on the performance highly depends on the implementation of OpenGL. As not using binding points or, in other words, using direct state access (DSA), is rather newly specified, one trivial way of implementing this for driver developers would be to use the old implementation and binding buffers behind the scenes. Including this factor in our evaluation can inform about such hidden implementation details.

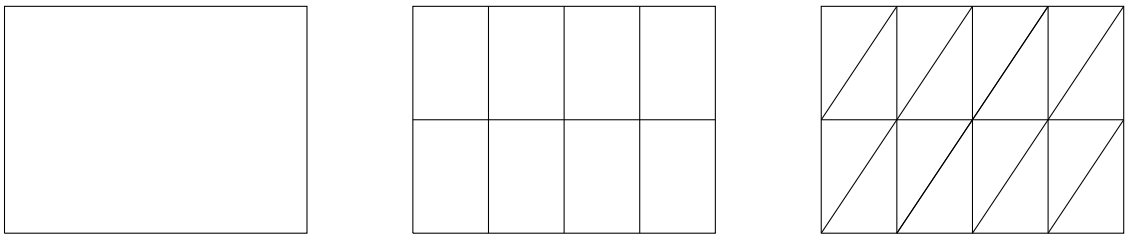
Table 4.2: Included control variables related to software design

Dataset Parameters

One important aspect is the dataset itself. For this work we create an artificial dataset that allows to fine tune certain parameters. The generated dataset is designed to fill the screen fully by rendering a plane with the same size. This means that such a full screen quad is constructed by using a plane that completely fills a display if it is shown on that display.

For increasing the dataset size, we split up this plane in rows and columns that themselves are each build up of two triangles. This is also depicted in Fig. 4.1, which shows in a) the full screen quad, in b) the quad split up in rows and columns and in c) the elements further split up in two triangles each. This allows us to balance dataset size with work for rendering. Additionally, this plane is replicated and all depth tests are turned off to further increase dataset size and increase necessary work for rendering.

Usually, rendering pipelines for graphics cards have built-in optimizations to maximize rendering performance. One example is called culling where parts of a dataset that are not visible are cut out. An example are fragments, that are already obscured by other



(a) Full-screen quad (b) Split into rows and columns (c) Split into triangles

Figure 4.1: Schematic drawing of how the dataset is generated. (a) shows a full screen quad, (b) the quad split up in rows and columns and (c) further split up in triangles

surfaces in front of them (with respect to the virtual camera) or virtually outside of the screen. These fragments need not to be processes – as they cannot be seen – and some implementations skip processing of them.

By turning off depth tests and using a dataset that is fixed to produce only fragments of the screen, we avoid having optimizations reducing workload on graphics hardware by canceling out the replicated planes or having fragments that can be skipped for processing.

This dataset gives us two more control variables that influence performance, listed in Tab. 4.3:

Dataset size	We expect the size of a dataset to directly influence both rendering and uploading performance. However, we do not vary workload without changing dataset size, meaning changing the size of the individual triangles to produce more or less fragments and therefore varying the workload on the GPU. We assume that behavior related to this can be simulated by changing the clock rate of the GPU.
Dataset partitioning	We can vary how many triangles are in one buffer and therefore how one dataset is split up in several sub datasets. While this does not change the overall size of the dataset it varies the number of buffers used. We assume that this variable has several impacts on the overall performance. On the one hand, having many buffers requires many state changes for rendering and many allocations for uploading. However, if it is possible to parallelize multiple uploads and increase used bandwidth, having at least two buffers can increase upload performance. On the other side, if one of the two processes uploading or rendering blocks the other, having multiple buffers instead of a few might easier allow to intertwine the individual calls and might also allow parallel execution of some parts of the two processes.

Table 4.3: Included control variables related to the dataset

To reduce the possible search space, we exclude altering the following hardware shown in Tab. 4.4:

CPU architecture	Changing the clock rate of the CPU has similar effects for the sought of model. Having a *slower* CPU architecture only reduces the amount of commands that can be put in the command buffer, which has the same consequences as reducing the CPU clock rate. For the experiments run in this work we use a CPU with 4 logical cores, which suffices for having multiple uploading threads as well as a distinct rendering thread. The reasoning for this is that most at the moment available graphics cards are only equipped with two distinct copy engines. We assume that using more than 3 parallel threads for uploading has negligible influence on performance, as they will only be waiting on each other.
DRAM clock rate and architecture	Changing the clock rate or using different DRAM modules only reduces the possible rate at which data can be transferred. This means that the maximal possible transfer rate is either limited by this hardware or not, which is trivial to model.
Mainbord, PCIe bus version, number of lanes	Changing the number of lanes or the PCIe version (to increase the version number would require a new mainboard as well as a graphics card that supports it) only affects the maximal possible transfer rate. Therefore, its influence is trivial to model.

Table 4.4: Excluded parameters related to hardware

Furthermore, our work aims to provide an methodical approach that models concurrent rendering and uploading. Our goal is to allow developers to optimize their applications. Consequently, this means that we focus on variables a developer actually can control without forcing the user to change hardware.

4.1.3 Control Variable Variations

After determining which control variables are part of this analysis, it is necessary to define what levels we analyze. This in consequence requires us to determine what kind of variables we are looking at. We distinguish between categorical variables and quantitative variables.

Categorical Variables An example for a categorical variable is what kind of graphics card is used. A categorical variable takes only certain values which do not need any ordering and can just describe a certain value. Out of the in subsection 4.1.2 described control variables, the following fall in that category:

- Used uploading method
- Buffer usage hint
- Using binding points or not
- Used graphics card

These variables possibly provide an offset and interaction to our mathematical model. For the experiments, all variations of these variables are used for modeling and experimenting.

Quantitative Variables In contrast, quantitative variables take a range of values. An example is clock rates. However, we cannot set arbitrary clock rates but only particular ones. Yet, for our mathematical model we assume them to be continual and therefore their influence is modeled continual. The remaining control variables in this category are:

- Dataset size
- Dataset partitioning
- CPU, GPU and graphics memory clock rates
- Number of uploading threads

For the quantitative variables we limit the number of possibilities by restricting the highest order polynomial to a degree of two. We assume that the influence of most quantitative control variables is linear, except the influence of dataset partitioning.

The latter is based on the number of copy engines of the used graphics cards. For example, if we assume to have two copy engines and each of them allows to use half of the maximum possible bandwidth, then dividing one dataset in two allows to increase the usage of bandwidth and thus performance. Dividing it further however will eventually increase overhead as more buffers need to be allocated and consequently reduce performance. Hence, we assume the influence of the number of partitions to be quadratic.

The choice of this, however, can be based on assumed a-priori knowledge or exploration experiments and only functions as a starting point. Assuming higher polynomials also increases the number of necessary experiments, which means we have to balance amount of experiments with information. In turn however, using higher polynomials and performing more experiments does not necessary result in models that provide a more *correct* view on the processes modeled.

To make sure that we catch all possible effects to test our assumptions of control variable influence forms, may they be linear or quadratic or higher, we use more levels as necessary. This means that we experiment with more than three variations for dataset partitioning, in particular five levels to see a broad range of possible effects for modeling.

For each of the linear modeled parameters we use three points, one low, one medium and one high value, to be able to distinguish between linear and quadratic behavior or other nonlinear behavior.

4.2 D-Optimal Design and Linear Regression

Having the control variables and their variations, the next step is to design an experiment that allows to obtain the necessary information for deducing a mathematical model. The naive solution is to use all possible permutations of control variable variations which can become a large number of single experiments to run.

To illustrate this lets assume that we have a computer, that allows to set 3 different CPU, 3 GPU and 3 memory clock rates. Further, we assume that we use 3 different data sizes and 4 different ways of partitioning it. Additionally, we are able to set 9 buffer usage hints, use 4 different uploading methods and 1, 2 or 3 threads. When we want to test all possible permutations, this would require to perform $3 \cdot 3 \cdot 3 \cdot 3 \cdot 4 \cdot 9 \cdot 4 \cdot 3 = 34,992$ experiments. Each of the single runs requires some time; when we assume just 1 minute on average, this would result in 24.3 days of performing experiments.

The sheer number of possible permutations for all control variables alone requires to plan experiments efficiently. We apply the so-called *Fedorov exchange Algorithm for D-optimal Design* [MN94] or in short D-optimal design. This sections gives a brief overview on the mathematical background for this algorithm and linear regression to estimate the parameters β_i of our modeling function Eq. (4.1).

The sought for model can use linear, quadratic or higher degree polynomial influences. Therefore, we combine them in a function vector

$$f_i(x_i) = (x_i, x_i^2, \dots), \tag{4.2}$$

where x_i are the modeled explaining variables. They can be control variables as well as an interaction term that includes multiple individual control variables, such as GPU clock rate ($x_{\text{gpu-clk}}$) and CPU clock rate ($x_{\text{cpu-clk}}$) together. The interaction $x_{\text{cpu\&gpu-clk}}$ of $x_{\text{cpu-clk}}$ and $x_{\text{gpu-clk}}$ would be defined as

$$x_{\text{cpu\&gpu-clk}} = x_{\text{cpu-clk}} * x_{\text{gpu-clk}} \tag{4.3}$$

Each of the mutations of x_i (or in other words the different functions used to model the influence a particular control variable) in $f_i(x_i)$ can take the possible levels or values of the defined control variables, e.g. the set clock rate of the CPU.

The l-th observation (or time measurement) Y_l equals then

$$Y_l = f_i(x_{i,l})B_i + \varepsilon_l \tag{4.4}$$

with ε_l being an error term collecting not modeled effects and measurement noise. B_i consists of the individual parameters $\beta_{i,v}$, with v identifying the element of $f_i(x_i)$:

$$B_i = (\beta_{i,1}, \beta_{i,2}, \dots)^T \tag{4.5}$$

We define for the following Y_l to be the l-th measurement of time and $x_{l,i}$ to be the setting for the l-th measurement for the i-th control variable. Considering 3 different control

variables and 4 measurements, this means $l \in 1,2,3,4$ and $i \in 1,2,3$. The modeling function then would be:

$$\begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \end{bmatrix} = \begin{bmatrix} f_1(x_{1,1}) & f_2(x_{1,2}) & f_3(x_{1,3}) \\ f_1(x_{2,1}) & f_2(x_{2,2}) & f_3(x_{2,3}) \\ f_1(x_{3,1}) & f_2(x_{3,2}) & f_3(x_{3,3}) \\ f_1(x_{3,1}) & f_2(x_{3,2}) & f_3(x_{3,3}) \end{bmatrix} \cdot \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \varepsilon_4 \end{bmatrix} \tag{4.6}$$

or

$$\boldsymbol{Y} = \boldsymbol{X} \cdot \boldsymbol{B} + \boldsymbol{E} \tag{4.7}$$

Our goal is to find $\boldsymbol{B}$. For this we can use the Moore-Penrose inverse $\boldsymbol{X}^+$ of $\boldsymbol{X}$:

$$\boldsymbol{X}^+ = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T, \tag{4.8}$$

which is also known as the least squares solution and also used for linear regression algorithms. Using $\boldsymbol{X}^+$ we can estimate coming from Eq. (4.6) the parameter vector $\boldsymbol{B}$:

$$\tilde{\boldsymbol{B}} = \boldsymbol{X}^+\boldsymbol{Y}. \tag{4.9}$$

The idea behind D-optimal design is to minimize the determinant of $(\boldsymbol{X}^T\boldsymbol{X})^{-1}$ or maximize the determinant of $\boldsymbol{X}^T\boldsymbol{X}$ in order to minimize the variance $\tilde{\boldsymbol{B}}$. This is done by setting random values (out of the possible values) for each of the explaining variables in $\boldsymbol{X}$. The values are iteratively exchanged and only those are kept, that yield a better design, until no improvement can be found. This process is repeated a couple of times with different starting conditions in order to possibly find the global optimum. However, it is not guaranteed that the global optimum is found. (For more details regarding D-optimal design as well as statistical implications refer to [Fed72].)

4.3 Model Selection using Information Criteria

This section describes how to assess the worth of including or excluding individual control variables and their forms and an algorithms to automate that process for finding an optimal model.

We start with an a priori model based on our understanding of the system. However, we want to derive a mathematical model from measurements, that allows to understand performance of the analyzed processes. This means we need a way to assess the worth of a variable for a given model. This can be achieved by calculating Akaike's Information Criterion (AIC) or a similar version of that, the Bayesian Information Criterion (BIC).

The value of a model is estimated by the AIC or BIC. Both criteria themselves rely on the log likelihood estimation. Following [BA04, p. 12, p. 60f], for a least squares estimation, the maximized log likelihood estimation is

$$\ln(\mathcal{L}(\hat{\theta})) = -\frac{1}{2}n\ln(\hat{\sigma}^2) - \frac{n}{2}\ln(2\pi) - \frac{n}{2}, \tag{4.10}$$

with $\hat{\theta}$ as the estimation for the model parameters (the estimations of β_i in Eq. (4.1)), n the number of samples, $\hat{\sigma}^2$ the maximum likelihood estimate of the variance and ln being the natural logarithm. The AIC and BIC are used to compare different models. This means that constant terms independent of the model can be neglected, which leaves us with

$$\ln(\mathcal{L}(\hat{\theta})) \approx -\frac{1}{2} n \ln(\hat{\sigma}^2) \tag{4.11}$$

The AIC can then be calculated with

$$\text{AIC} = -2\ln(\mathcal{L}(\hat{\theta})) + 2K = n\ln(\hat{\sigma}^2) + 2K \tag{4.12}$$

with K being the total number of explaining variables, including the intercept.

Intuitively speaking, the AIC tries to get the best fit of the defined set of possible models while on the same time keeping the number of explaining variables as small as possible. The BIC goes a step further and penalizes more explaining variables stronger by using the natural logarithm of the sample size $\ln(n)$ as factor instead of 2.

$$\text{BIC} = n\ln(\hat{\sigma}^2) + \ln(n)K \tag{4.13}$$

For estimating the value of adding an variable, the AIC or BIC is calculated before and after the addition. Getting a smaller AIC or BIC after the addition means that the additional variable improves the model given the conducted experiments. When we iteratively subtract or add a variable to a given starting model, and only hold the result of such an operation that improves the value of model, we usually obtain a better fitted model. For more information about AIC or BIC, the interested reader is referred to [BA04, p. 60ff].

Adding or subtracting a variable does not only meaning adding/subtracting a control variable, such as GPU clock rate or size. As described above, explaining variables can take different forms. On the one hand, there are combinations of different control variables that can result in interaction effects. On the other hand, also the function describing the form of the influence of a control variable can lead to different results. Therefore, we include for all quantitative control variable their assumed polynomial degree (e.g. linear or quadratic) and also one order higher as defined for obtaining the screening design. By using the BIC with adding and removing variables, this keeps the polynomial degree that better represents the data and removes unnecessary polynomial degrees.

4.4 Chapter Summary

In this chapter we detail our approach on designing the necessary experiments. We include the following control variables for finding a mathematical model describing concurrent uploading and rendering:

- CPU, GPU and graphics memory clock rates

- Two distinct graphics cards
- Different uploading methods for data transfer
- Buffer usage hints
- Usage of binding points (whether and which)
- Dataset size and partitioning

The total number of all permutations for all levels of the control variables would require an huge amount of experiments and time for performing the experiments. Yet, we reduce the number by applying *Fedorov's Exchange Algorithm for D-optimal Design* in order to maximize the amount of information gained by single experiments. Linear regression is applied in combination with BIC in order to find a suitable model. This answers **SRQ4**: *How to design experiments to derive the MARKUs addressing **RQ**?*

Chapter 5

Experimental Setup and Data Preparation

This chapter describes the experimental setup and the steps necessary to prepare the data, that is gained from conducting experiments, for analysis and for deducing the two MARKUs.

5.1 Experimental Setup

System Parameters

For the experiments we use two structural identical systems with the following specification:

CPU	Intel Xeon CPU E5-1607 v3 @ 3.10GHz
Host Memory	4 x 16GB DDR3 @ 1866Mhz (from 8 available slots)
PCIe	Graphics card are connected with PCIe Version 3 x16

Graphics Cards Only the graphics cards are different; we use the following two:

- NVIDIA Quadro RTX 4000 with 8GB GDDR6 graphics memory (in the following RTX)
- NVIDIA GeForce GTX Titan X with 12GB GDDR5 graphics memory (in the following TITAN)

CPU The CPU itself allows to read and write with a maximum rate of 59GB/s, supports the configured clock rate of main memory and can be connected with up to 40 PCIe lanes[1]. This means the CPU can be excluded as a bottleneck for data transfers as it exceeds the maximal bandwidth of the used PCIe v3 x16.

Host Memory For host memory, twice as many slots as needed are available. This means that the configured DIMMs can be used in dual channel mode and possible access rate exceeds the maximal bandwidth of the used PCIe v3 x16. A benchmark using Intel Memory Latency Checker[2] version 3.8 confirms this for the test system as it measures an access rate of up to 32 GB/s.

Clock Rates CPU, GPU and graphics memory clock rate can be adjusted for the given system. For the CPU we first turn off Intel's pstates via a kernel argument. For the following experiments, we use three specific settings: 1200, 2000 and 3100 MHz. The two graphics cards allow to set GPU and graphics memory clock rate via NVIDIA's Management Library (NVML). We include the following settings:

Card	GPU clock rates [MHz]	Graphics memory clock rate [MHz]
RTX	300, 1200, 2100	810, 5001, 6501
TITAN	595, 1063, 1519	810, 3304, 3505

Please note that for the graphics cards and their GPUs two different architectures are used, namely Turing and Maxwell for RTX (upper row) and TITAN (lower row), respectively.

Operating System and Driver Configuration Ubuntu 18.04.06 LTS is used as operating system with NVIDIA's proprietary driver 440.64 prebuilt from the graphics drivers Personal Package Archive (ppa)[3]. For the driver we turn off *threaded optimizations* via the program *nvidia-settings* as this prevented freezing and crashing the system when using multi-threaded OpenGL-application.

Graphics Interface A separate off-screen Xserver (which is a graphical surface of linux that allows to create windows) is started for the experiments in order to avoid any influences

[1] see also `https://ark.intel.com/content/www/de/de/ark/products/82762/intel-xeon-processor-e5-1607-v3-10m-cache-3-10-ghz.html`, accessed 21.04.2020

[2] `https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html`, accessed 21.06.2020

[3] `https://launchpad.net/~graphics-drivers/+archive/ubuntu/ppa`, accessed 21.04.2020

from monitor or input devices. Additionally, the application itself creates an off-screen framebuffer as renderbuffer in order to further avoid such influences. Creation of OpenGL contexts and off-screen windows is implemented using the library GLFW 3.2 from the official Ubuntu repositories. With this library, contexts are created with a hidden window and specified to use OpenGL 4.5 with the core profile.

Dataset Generation and Parameters

Rendering performance and GPU workload highly depend on the scene rendered and the current perspective, partially because the following two reasons:

(1) Invisible parts of a scene

(2) Distance to visible parts of a scene

(1) Often, parts of a scene, although ordered to be rendered, are not visible from the current perspective and therefore can be skipped early in the rendering pipeline and reduce workload for rendering.
(2) Another correlation between perspective and workload is caused by distance to single triangles. A triangle that covers most of the screen produces more fragments that are then processed to pixels in comparison to a triangle far away that only covers a few pixels. Consequently, less processing is needed for the latter and workload on the GPU is smaller.

Dataset Sizes To avoid such perspective influences, we generate an artificial dataset as described in Chapter 4, using $50 \cdot 50 = 2500$ rectangle to divide the full screen quad. As we always want to pick complete triangles for rendering, we are limited to a multiple of 3 vertices times 3 floating-point numbers for each vertex times 4 Byte for each floating-point number, resulting in $3 \cdot 3 \cdot 4 = 36$B for each triangle. This means that the used datasets are always a multiple of 36B. To gain sizes close to full MB (i.e. $1024 \cdot 1024$B) we use a factor of 29128 ($\approx 1024 \cdot 1024/36$).

For the experiments we use datasets with the size of roughly 16, 64 and 128MB; the exact amount is $\{16, 64, 128\} \cdot 29128 \cdot 36$B. Due to the nature of our uploading scheme, larger datasets can easily result in more graphics memory needed than available. However, assuming a real dataset with only 200 timesteps, each with 128MB in size, easily shows that this *small* dataset easily is more than most current graphics card can store. In total, this dataset has 25GB while only the top tier professional series of graphics cards have 32GB of graphics memory. On the lower end of the sizes, smaller dataset sizes get small enough to be hold in memory.

Dataset Partitioning Furthermore, due to splitting up datasets in smaller subsets we can also easily observe and measure the performance for smaller sizes. Again, we require that only complete triangles are in the resulting datasets after splitting them apart. Therefore, we use $1, 2, 4, 11, 22, 44$ as divisors. These are chose as they are the divisors of 29128 and therefore guarantee to produce partitions with complete triangles. While using one

partition is one extreme, using 11, 22, or 44 should give a glance on the effect of using more and more numbers of partitions on the time needed for both uploading and rendering. Using two or four partitions allows to see the effect of using two copy engines (that are present on the tested graphics cards) in parallel and, in the case of four partitions, possibly overlapping of preparing uploading and uploading as well as keeping the copy engines busy by having already prepared buffers to fill.

Software Parameters

For the following experiments, we vary the four aforementioned uploading methods without and with DSA:

Without DSA	With DSA
`glBufferData`	`glNamedBufferData`
`glBufferSubData`	`glNamedBufferSubData`
`glMapBuffer + memcpy + glUnmapuffer`	`glMapNamedBuffer + memcpy + glUnmapNamedBuffer`
`glMapBufferRange + memcpy + glUnmapuffer`	`glMapNamedBufferRange + memcpy + glUnmapNamedBuffer`

Table 5.1: Used uploading methods, with and without DSA

Further the possible 9 different buffer usage hints are varied:

- `GL_STREAM_DRAW`
- `GL_STREAM_COPY`
- `GL_STREAM_READ`
- `GL_STATIC_DRAW`
- `GL_STATIC_COPY`
- `GL_STATIC_READ`
- `GL_DYNAMIC_DRAW`
- `GL_DYNAMIC_COPY`
- `GL_DYNAMIC_READ`

We also test using PBOs or not by either binding to `GL_ARRAY_BUFFER` or `GL_PIXEL_UNPACK_BUFFER`. The number of concurrent data uploads is varied by using 1, 2 or 3 threads, each with their own OpenGL context.

5.1.1 Experimental Design

For the experimental design we use the following formula as input for the D-optimal design algorithm:

$$\begin{aligned} time = (&method + bufferhint + pbo + dsa + card + poly(size, 2) + \\ &poly(gpu - clock, 2) + poly(mem - clock, 2) + poly(numberthreads, 2) + \\ &poly(partition, 3) + poly(cpu - clock, 2))^2. \end{aligned} \tag{5.1}$$

Equation (5.1) is interpreted as follows: For each of the variables, e.g. method or partition, we try to estimate a regression coefficient β_i.

Categorical Variables Some of the variables are categorical variables and have only a certain set of possibilities. These variables can have more than one regression coefficient. For example, in our case, method can take 4 different values for the 4 different uploading methods, described before. For each of the values we want to estimate its influence which is expressed by its regression coefficient. This means, we need at least 3 regression coefficients β_i, with which we express the deviation from a specific default value. The default value is just one of the 4 possibilities. This means that for each categorical variable, we need one less regression coefficient as the variable has possible levels. The categorical variables are:

method	Which uploading method is used to upload data?
bufferhint	Which buffer usage hint is used?
pbo	Are pixel buffer objects used as buffer binding point?
dsa	Is direct state access used (i.e. without `glBindBuffer`)?
card	Which card is used?

Quantitative Variables The remaining variables are quantitative variables. For them we estimate only one regression coefficient. The operator *poly* is from the `stats` package, which itself is part of the core of the programming language R. More information about this package can be found in [R C19]. This operator describes that we assume that this variable has polynomial influence with a degree of the second parameter. An example is $poly(mem - clock, 2)$ in Eq. (5.1). This means that we assume that the influence of the graphics memory clock rate is quadratic.

The formula given in Eq. (5.1) reflects our assumptions described in Subsection 4.1.2. For the quantitative variables, we describe a higher degree polynomial to be able to test our hypotheses on their degree.

Interactions Another important part of Eq. (5.1) is the $(\cdot)^2$ operator. In this case it describes that we also want to test for all two way interactions. This means that we test

for all possible pairs of variables. For the categorical variable this means that all levels of them are variables. An example for this are interactions of the control variables method and card. For the interaction we additionally test for the combined influence of the first card and the first uploading method, the first card and the second uploading method, the first card and the third uploading method and so on. Combined means that we estimate the influence of both together while we also estimate their influence individually. Theoretically this can mean, that the first uploading method has a certain influence regardless of all other variables that is completely annulled by its interaction with other variables for a certain case. Each of these interaction terms again gets their own regression coefficient β_i.

Number of Configurations The minimum number of configurations (this means also the number of experiments) is defined by the number of regression coefficients, which is in our case 340. To gain even more information and to be able to vary the formula in the evaluation, we use more than double this amount, in particular 800 experimental configurations.

5.1.2 Experimental Parameters

The experiments are conducted in the following steps:

1. Configure uploading and rendering processes for given configuration. This includes that all uploading threads are killed and newly created with new OpenGL contexts.

2. Upload dataset l times without rendering. For each time, wait until the upload is finished before the next upload is issued. This allows to get a clean measurement of uploading performance without GPU workload. This measurement set is called *uploading only* in the following.

3. Render dataset l times without uploading. Each time we wait until rendering is finished before the next upload is issued. This allows to get a clean measurement of rendering without the influence uploading in parallel. This measurement set is called *rendering only* in the following.

4. Render dataset **at least** l times while uploading (and also exchanging the rendered dataset) **at least** 30 times. As rendering and uploading are not necessarily taking the same time, either the dataset is rendered or uploaded exactly 30 times. The respective other usually is performed more than 30 times. Another thread is generated to issue and therefore start the process of uploading. This thread is also used to determine when a data upload is finished by exchanging the buffer on CPU side for the rendering process. These measurement sets are called *rendering concurrently* and *uploading concurrently* in the following.

We use $l = 30$ to have a large enough sample size for further analysis.

Measurements

For all measurements, the C++ `std::chrono::high_resolution_clock` class is used for measuring the time. Measurements start right before uploading or rendering is issued and include the calls needed for issuing. The CPU thread performing rendering is halted until these commands are finished and then the time is measured.

The end time for uploading a dataset is determined when it can be exchanged for the rendering process (which is not necessarily needed for this to happen). In comparison to the time needed for uploading, the additional time for exchanging the buffer on CPU side is negligible.

Measurements Example

Fig. 5.1 illustrates this process with an example. Here, one uploading thread (and only the uploading thread, no preparation thread is illustrated) is shown with the the rendering thread.

This figure shows from left to right beginning and end of measurements and the individual calls needed for uploading and rendering. The first and third row show what is done on CPU side, for rendering and uploading, respectively. The second row illustrates what is done on GPU side. In other words, the first and third row illustrate how the CPU fills the command queue. The third row illustrates how the GPU processes the commands in the command queue.

Rendering Calls The necessary calls involved for rendering and their denotations in Fig. 5.1 are:

1. Start measurement: s_R
2. `glBindBuffer` (bind the buffer): bc_R
3. `glDraw` (draw the buffer): dc_R
4. `glBindBuffer(0)` (unbind the buffer): ub_R
5. Insert synchobject: so_R
6. Wait for synchobject: w_R
7. Stop measurement: o_R

Uploading Calls In this example, the uploading method using the OpenGL function `glBufferData` without DSA is used. The necessary calls involved for the corresponding uploading method and their denotations in Fig. 5.1 are:

1. Start measurement: s_U

2. `glBindBuffer` (bind the buffer): bc_U
3. `glBufferData` (allocate and upload to the buffer): bd_U
4. `glBindBuffer(0)` (unbind the buffer):ub_U
5. Insert synchobject: so_U
6. Wait for synchobject: w_U
7. Stop measurement: o_U

For both cases are waiting times on CPU side involved, denoted w_R for rendering and w_U for uploading.

The measured times are shown in the fourth and fifth row and denoted as mt_R for rendering and mt_U for uploading.

Fig. 5.1 shows a possible measurement error for each rendering and uploading. We see that steps 2, 3, and 4 of uploading are inserted into the command queue before the synchobject of rendering so_R can be inserted. Therefore, the measured time for rendering mt_R includes the time it takes to perform steps 2,3, and 4 of uploading.

For uploading we see a similar error. Here, additionally to the time needed for uploading, also parts of the time for performing the draw call issued by rendering dc_R is included.

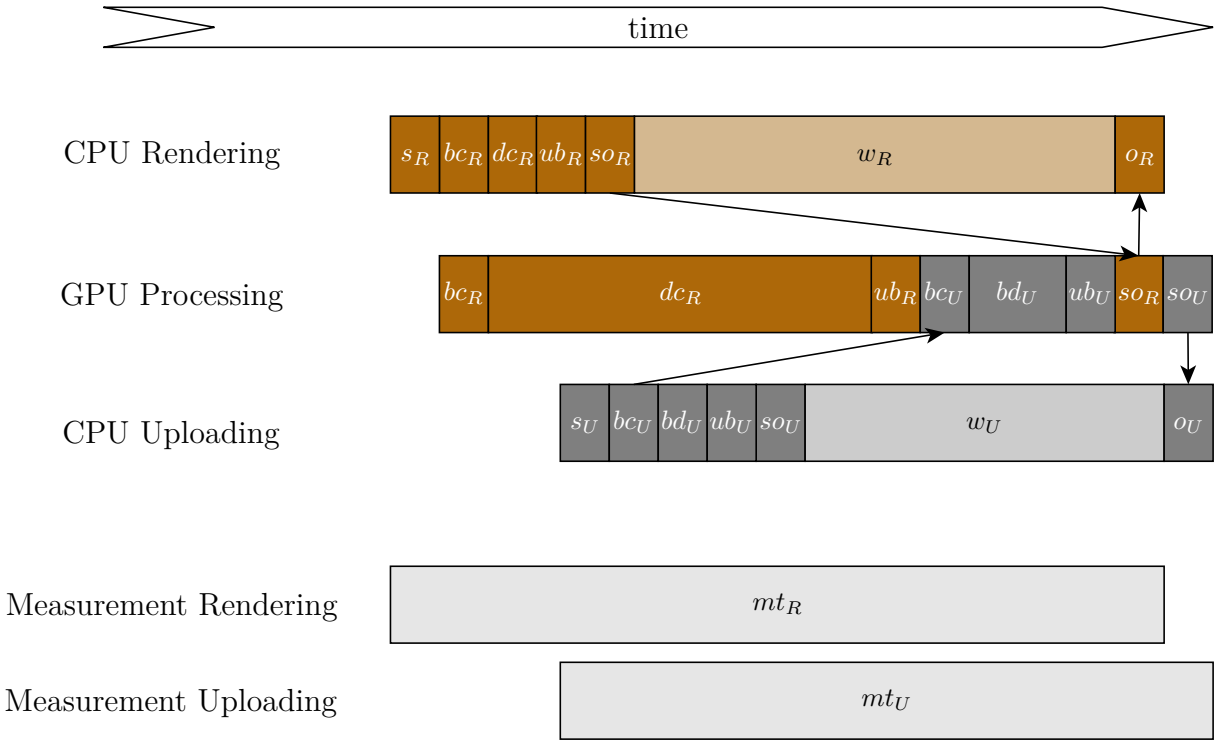


Figure 5.1: Schematic showing how uploading and rendering are measured. A possible measurements error for each rendering and uploading is illustrated.

5.2 Data Preparation

For evaluation of the measurements, the measured data first needs to be preprocessed. For this, three steps are involved:

- Data Fusion
- Outlier Removal
- Data Aggregation

5.2.1 Data Fusion

The deviance of set clock rates to queried clock rate might be due to energy saving algorithms of the driver. Therefore, we implemented on another thread periodical clock rate queries. The clock rates are queried with a forced 1ms delay between each query. In most cases, the time needed for either process takes longer than one ms, so that every uploading/rendering measurement should have at least one query in that time frame.

Yet, some queries are getting delayed. We assume this is caused by scheduling or the driver being busy working on other tasks. For these cases, we search for the closest queries and use these. For the cases where more than one query is performed while executing

uploading or rendering, the arithmetic mean of all queries in the measured time period is used.

5.2.2 Outlier Removal

In some cases, there are some serious delays in the measurements. One cause can be system processes that take precedence over the measurement application. In order to filter them out we use the outlier detection available from the R function `boxplot.stats` of the core package *grDevices*, see also [R C19] for more information. In order to be as conservative as possible, we only filter out extreme outliers (or *far out*, as Tukey [Tuk77, p. 44f] labelled them). This is achieved by using double the default coefficient (which is 1.5) that includes measurements as non outliers. The range is determined by the size of the range in which 50% of the measurements are, meaning the range between the 25th and 75th percentile. In our case we use 3 times the size of this range. We only exclude measurements, and therefore classify them as outliers, that are outside this extended range.

For an example let's assume that this range, i.e. the range between the 25th and 75th percentile, spans from 10 to 15 ms. This means that 50% of all measurements lie between 10 and 15 ms. The lower and upper boundaries for measurements being classified as outliers would be $5\text{ms} \cdot 3 = 15\text{ms}$ lower or greater than the 25th and 75th percentiles, i.e. -5ms and 30ms.

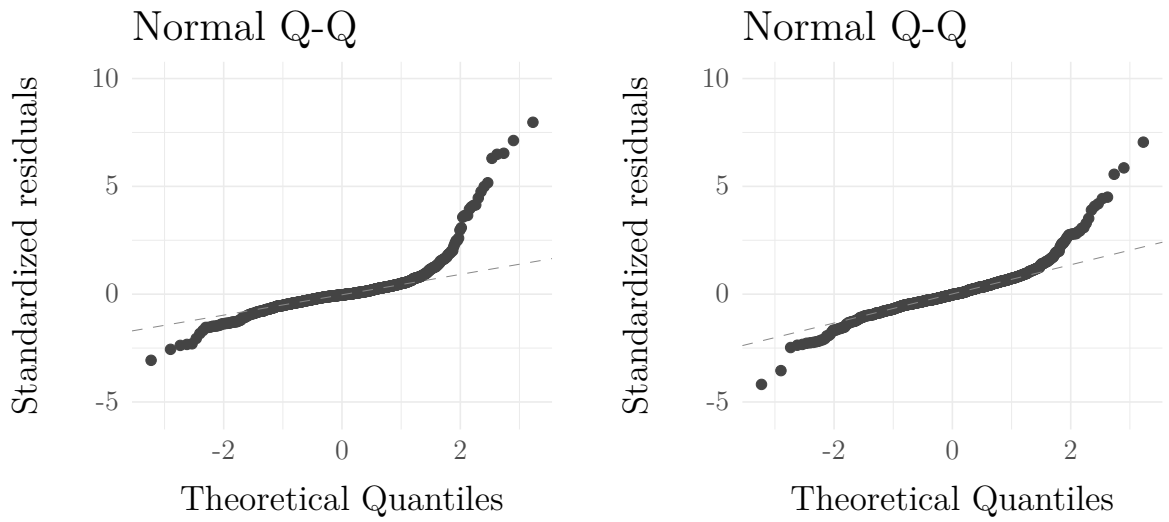
5.2.3 Data Aggregation

For the *uploading only* and *rendering only* experiments, we test 800 different configurations and measure for each 30 data points, resulting in 24,000 data points in total. For the *concurrently* scenario, we measured about 240,000 data points for rendering and 42,000 data points for uploading. Additionally, for some configurations, a high variation can be observed in consecutive measurements due to the nature of the uploading and rendering processes. The sheer number of data points increase the time for analysis immensely, while information gain is rather small. Therefore, we first summarize the datasets and analyze the summarized results. The aggregation of the data points is handled the following way:

- Calculate the natural logarithm of each measurement
- Calculate the mean of all measurements for each configuration

Transforming the Dataset We transform the dataset using the natural logarithm as we cannot be sure that the distribution of measurements is normal. This is due to each process requiring a minimum time to be performed if it is undisturbed. Noise, in the sense of system interruptions, performance throttling or other processes holding back the measured process, only add a time delay on each measurement and therefore, skew the distribution positively. Although disputed in the statistics community, transforming skewed data using the logarithm is widely used [Kee95, MM01, CHN+14]. After this, the arithmetic mean of the transformed measurements is calculated.

Normality Analysis Transforming the measurements results in visually more normal looking residuals, as can be seen in Fig. 5.2. The figure shows a QQ-Plot for deducing a model as described for all measurements for the concurrent rendering case. The x-axis describes the theoretical quantiles of a normal distribution, the y-axis shows the standardized residuals of a given model. In (a) the data is not transformed by using the natural logarithm, in (b) it is. In (a) and (b) we see that the standardized residuals in the range from about -1.5 to 1.5 of theoretical quantiles mostly follow a straight line. Above and below 1.5 and -1.5, respectively, the standardized residuals deviate from that straight line. An indicator for normality is that residuals follow mostly a straight line as illustrated by the dashed line. We see that transforming the data leads to residuals being closer to the dashed line, especially for more extreme values of theoretical quantiles in the range below -1.5 and above 1.5.



(a) QQ-Plot for residuals for untransformed measurements

(b) QQ-Plot for residuals of log-transformed measurements

Figure 5.2: Comparison of QQ-Plots for (a) untransformed and (b) log-transformed measurements

5.3 Chapter Summary

In this chapter, we determine the levels used for the performed experiments. Additionally, we understand how to transform the gained measurements using the natural logarithm to get more normal distributed measurements and summarize them using the arithmetic mean to quicken further analysis.

Chapter 6

Experimental Data and MARKUs Analysis

This chapter describes the analysis of the conducted experiments and the derived MARKUs. Firstly, we take a look at the queried variables, meaning graphics memory clock rate and GPU clock rate. Secondly, we discuss the differences between rendering and uploading without the respective other compared to how they perform if executed concurrently. Thirdly, we discuss the MARKUs derived from the experiments. Furthermore, we discuss what this means for the processes themselves: Which configuration is best for rendering, which for uploading and what are their impact on the respective other.

Analysed Cases

We distinguish between *rendering/uploading only* and *rendering/uploading concurrently*.

Rendering/Uploading only means that only one of the two processes is performed. For *rendering only*, this means that data is uploaded to graphics memory once before measurements start and then rendering is performed with static data. For *uploading only* data is uploaded to graphics memory but not further processed on graphics card.

Rendering/Uploading concurrently means that both processes are performed within the same time period and affect each other, as the uploading process replaces the data used for the rendering process. The optimization targets **OT1**, **OT2** and **OT3** only apply to the two *concurrently* cases, as without either rendering or uploading, prioritizing the other is given.

This means that we take a look at these four cases:

- *Rendering only* (static data in graphics memory is rendered)
- *Uploading only* (data is not rendered)

- *Rendering concurrently* (of dynamic data; rendering and uploading happen concurrently)
- *Uploading concurrently* (rendering and uploading happen concurrently)

The analysis of the four cases can be found in the following subsections:

	rendering	*uploading*
only	6.1.1	6.1.2
concurrently	6.1.1, 6.2.2	6.1.2, 6.1.2

6.1 *Only* vs. *Concurrently*

In this section, we compare *only* cases with *concurrently* cases. We start with *rendering only* versus *rendering concurrently* and then compare *uploading only* with *uploading concurrently*.

Configurations For all four cases the same configurations are compared. This means, that all control variables are set to the same values. However, as mentioned before, the queried GPU and graphics memory clock rates can differ for either case and we specifically discuss deviations in these variables, if applicable. For simplicity, we select 5 out of the 800 measured configurations that illustrate similarities and differences and represent the whole dataset.

The 5 selected configurations are depicted in Table 6.1. They are numbered from 1 to

#	size MB	number partitions	method	pbo	dsa	buffer hint	number threads	CPU GHz	GPU GHz	VRAM GHz	card
1	16	1	2	yes	no	6	1	1.2	1.2	5.001	RTX
2	16	44	1	no	yes	4	1	2.0	0.3	5.001	RTX
3	64	22	3	yes	yes	5	3	3.1	0.595	3.505	TITAN
4	128	4	0	yes	no	8	2	2.0	1.519	3.304	TITAN
5	64	4	0	no	no	7	3	1.2	2.1	0.81	RTX

Table 6.1: Five selected configurations used for the comparison of the *rendering only* and *uploading only* with the corresponding *concurrently* cases.

5 in column #. The 5 configurations include 3 different sized datasets, in particular 16 MB, 64 MB and 128 MB, and 4 different numbers of partitions: 1, 4, 22, and 44. Furthermore, all 4 before described methods for uploading are used, with or without PBOs, with or without DSA, 5 different buffer usage hints – 4, 5, 6, 7, and 8 –, the use of 1, 2 or 3 threads

for uploading as well as 3 different set CPU clock rates – 1.2 GHz, 2.0 GHz, 3.1 GHz–, 5 different set GPU clock rates – 0.3 GHz, 0.595 GHz, 1.2 GHz, 1.519 GHz, and 2.1 –, 4 different set graphics memory clock rates – including 0.81 GHz, 3.304 GHz, 3.505 GHz and 5.001 GHz–, and both graphics cards, RTX and TITAN.
The values for buffer usage hints (buffer hint in Tab. 6.1) correspond to:

Value	Buffer usage hint
4	`GL_STATIC_COPY`
5	`GL_DYNAMIC_COPY`
6	`GL_STREAM_READ`
7	`GL_STATIC_READ`
8	`GL_DYNAMIC_READ`

The values for the used uploading method (method in Tab. 6.1) correspond to:

Value	Uploading method
0	`glBufferData`
1	`glMapBuffer + memcpy + glUnmapBuffer`
2	`glMapBufferRange + memcpy + glUnmapBuffer`
3	`glBufferSubData`

6.1.1 *Rendering Only* vs. *Rendering Concurrently*

For comparing *rendering only* with *rendering concurrently*, we first analyze differences in set and queried clock rates and subsequently differences in times needed for rendering.

Clock Rates

The comparison of the set GPU and graphics memory clock rate with queried clock rates is shown in Fig. 6.1. The image shows for *rendering only* (top row) as well as *rendering concurrently* (bottom row) the comparison of queried clock rate against set clock rate for graphics memory (left) and GPU (right) clock rates. In all four images, the configurations are distributed on the x-axis, the y-axis shows the respective clock rate in GHz from 0 GHz to 6 GHz for graphics memory and 0 GHz to 2.5 GHz for GPU clock rates. For each configuration, both the set clock rate as well as the mean queried clock rate is shown.

We see that in some cases, e.g for configurations 3 and 4 for graphics memory, or configuration 1, 2 and 3 for the GPU, the mean queried clock rate is nearly the same as the set clock rate. For other cases, e.g. configuration 1, 2 and 5 for graphics memory and 4 and 5 for GPU clock, the mean queried differs from the set clock rate. For configuration

5 for graphics memory clock rate, this difference is quite large. Here the set clock rate is 0.81 GHz, but the queried is 6.5 GHz. Both, *rendering concurrently* and *rendering only* have similar characteristics.

All Configurations If we further compare the queried data for all configurations, we see that there is quite a difference between the two used graphics cards as well. Fig. 6.2 shows for all 800 configurations (x-axis) the mean queried clock rate (y-axis), in red for the RTX and in blue for the TITAN graphics card. We see that the RTX card has a greater variety for the GPU clock rate, ranging from 300 MHz to 2.1 GHz. The TITAN card ranges from about 600 MHz to 1.33 GHz.

However, the difference for graphics memory clock rate is pivotal. Fig. 6.3 gives an overview on all mean queried graphics memory clock rates, again red for the RTX and blue for the TITAN graphics card.

In comparison to graphics memory clock rates, the GPU clock rates are more evenly distributed. For the RTX card higher and lower clock rates are queried than for the TITAN card. For the graphics memory clock rates, we see a more or less divided picture. Most mean queried graphics memory clock rates for the RTX card are above 5 GHz, only 4 mean queried are lower than those queried using the TITAN card. Most mean queried graphics memory clock rates are at 6.5 GHz. For the TITAN card, we see that no mean queried graphics memory clock rate is above 3.6 GHz and the majority of them are distributed in the range between 3.3 GHz and 3.5 GHz and at 810 MHz. This needs to be taken into account for further analysis when we take a look at the influence of the control variables. Default values need to be used for control variables that are not the focus of the particular analysis. As the mean queried graphics memory clock rates are highly different for both cards, we also need to set different default values for each card in order to be within the prediction boundaries of the deducted models. For the following, when not otherwise stated, we use the maximum settable clock rate of 6.5 GHz for the RTX card, as there are the most measurements, and 3.505 GHz for the TITAN card, which is the maximum settable clock rate for this card, so it is comparable to the RTX card.

Rendering Times

In Fig. 6.4 the time needed for rendering with the same configuration is compared for the two cases *rendering only* and *concurrently*. The 5 configurations are spread on the x-axis and the y-axis shows the time needed for rendering with a range from 0 ms to about 120 ms. Configuration 1 needs about 14.9 ms for both cases. The remaining configurations all increase the time needed for rendering and configuration 5 shows a doubling of the time needed from about 37.9 ms to 80.8 ms. We see that the *rendering only* case can serve as a minimal time needed for rendering with a particular configuration. In most cases, *rendering concurrently* needs the same time or more than *rendering only*.

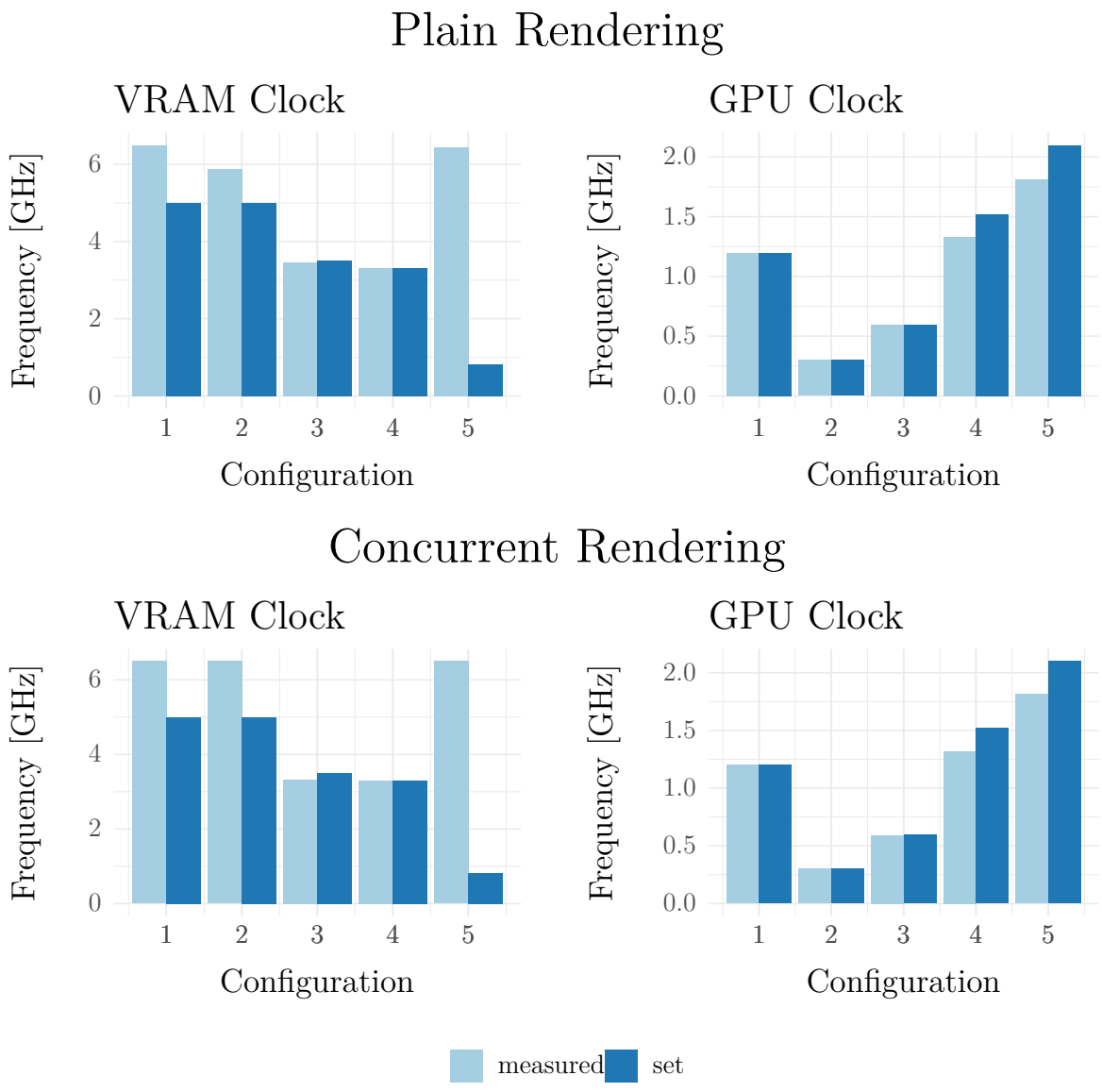


Figure 6.1: Comparison of graphics memory and GPU clock rates for *rendering only* and *rendering concurrently*.

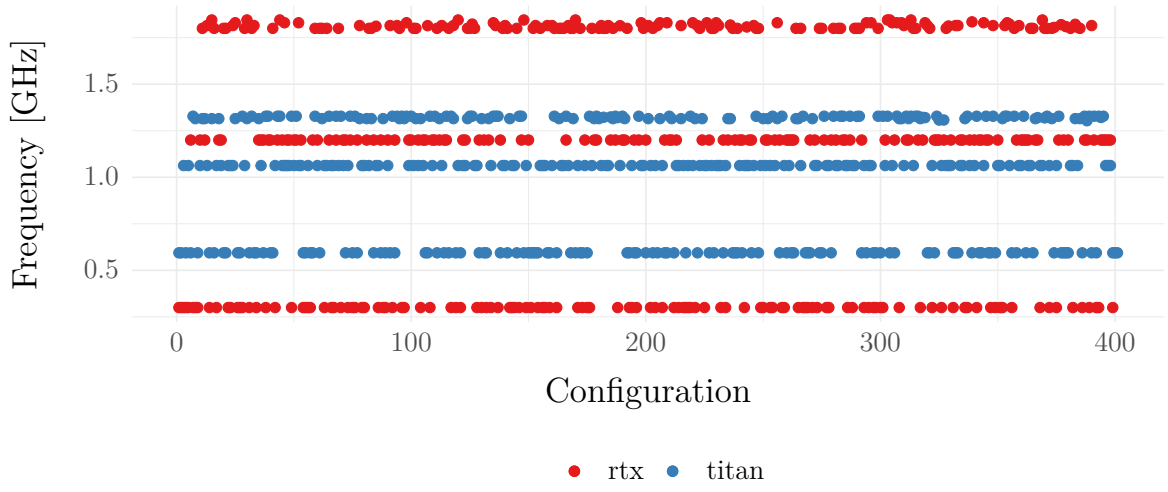


Figure 6.2: Queried mean GPU clock rates for all tested 800 configurations for RTX (red) and the TITAN (blue) graphics card

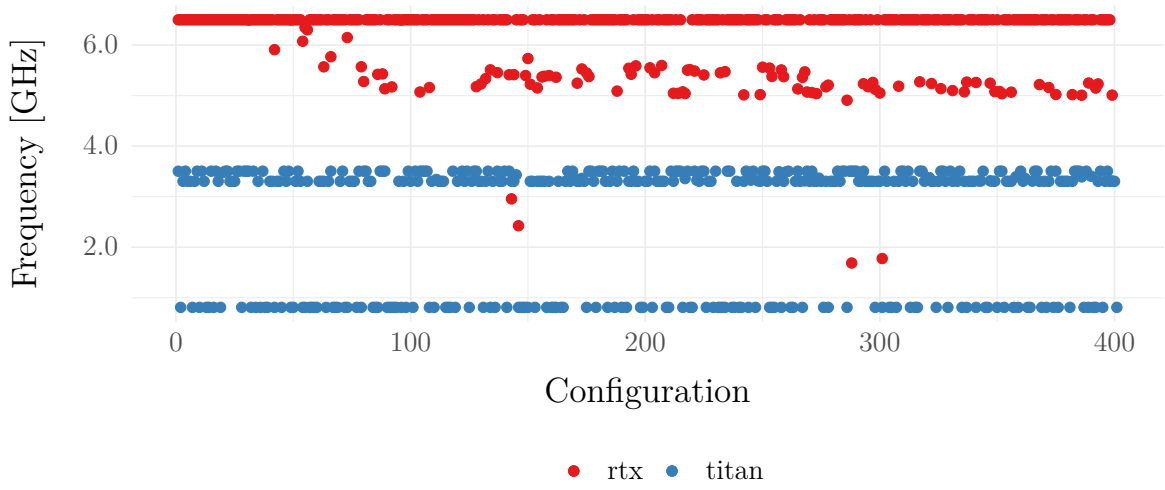


Figure 6.3: Queried mean graphics memory clock rates for all tested 800 configurations for RTX (red) and the TITAN (blue) graphics card

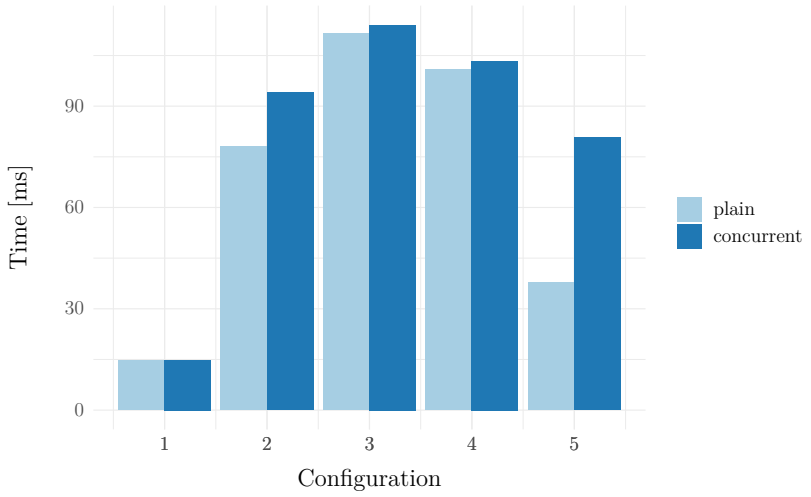


Figure 6.4: Comparison of needed time for five selected configurations for *rendering only* and *rendering concurrently*.

6.1.2 *Uploading Only* vs. *Uploading Concurrently*

For comparing *uploading only* with *uploading concurrently* we again analyze first differences in set and queried clock rates and then differences of times needed for both cases.

Clock Rates

The comparison of the set clock rates with the mean queried clock rates for the uploading cases is shown in Fig. 6.5. The mean queried clock rates show almost the same deviations as for the rendering cases.

Uploading Times

Fig. 6.6 shows the time needed for uploading with the same configuration for both *uploading only* and *uploading concurrently*. Again, the 5 selected configurations are distributed on the x-axis. The y-axis shows the mean time needed for uploading from 0 to about 950 ms. For each configuration, we compare *uploading only* (light blue) with *uploading concurrently* case (darker blue). We see that the time needed for uploading for configurations 3 and 4, respectively highly deviates from one to the other case. While *uploading only* needs about 100 ms to 110 ms, for *uploading concurrently* the average time is 810 ms to 910 ms. However, not all configurations expose such an increase for the time needed when comparing *uploading only* with *uploading concurrently*. Configuration 1, 2 and 3 change the time needed from about 20.3 ms, 48.8 ms, and 95 ms to about 19.9 ms, 57.2 ms and 137.8 ms, respectively. For configuration 1 there is even a decrease in the mean time needed. We assume that this decrease is caused by noise and outlier removal and both means should be about equal.

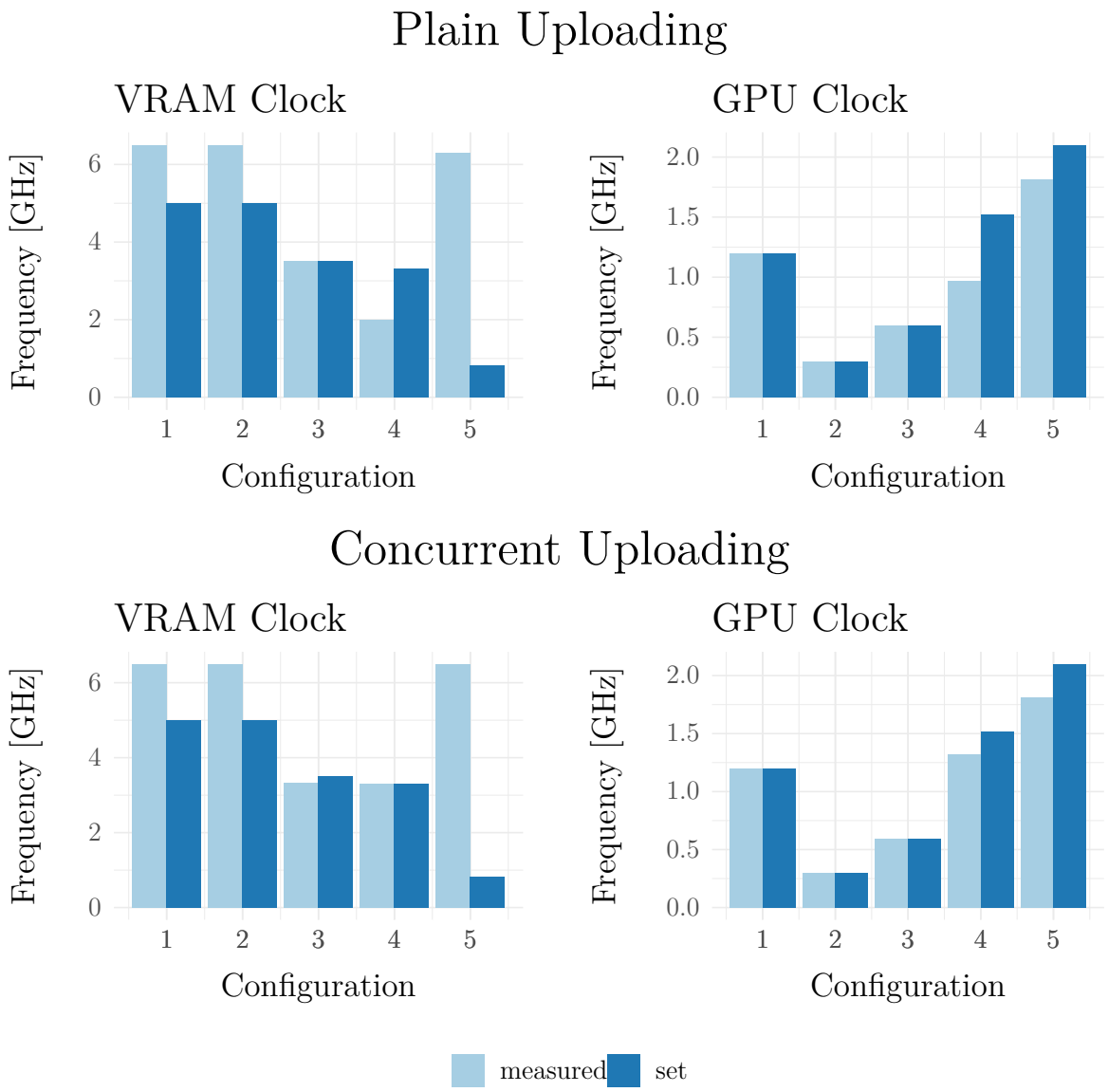


Figure 6.5: Comparison of graphics memory and GPU clock rates for *uploading only* and uploading concurrently.

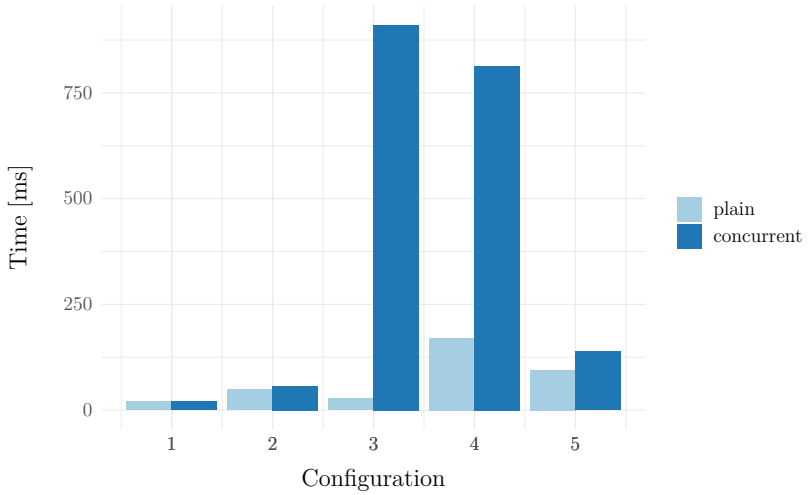


Figure 6.6: Comparison of needed time for *uploading only* and *uploading concurrently* cases.

Once more, we see that *uploading only* can serve as a bottom line for the time needed to complete a particular process. For most configurations, the time needed for uploading is equal or higher for *uploading concurrently* compared to *uploading only*. This means that modeling *concurrently* can give us an upper limit on the time needed for both rendering and uploading. We also see that while the increase for rendering for configuration 3 and 4 was moderate, the increase for uploading for those two configurations is immensely. *Uploading only* (without rendering) is done on average in 27.5 ms and 170 ms. For *uploading concurrently*, the time needed increases to 910.7 ms and 813.5 ms. This further motivates modeling both processes as concurrent processes instead of using the two *only* cases.

6.2 MARKUs Analysis

We discuss in this section the two MARKUs derived from the performed experiments divided in the following parts:

- MARKU-R: MARKU for *Rendering Concurrently* (in Subsection 6.2.2)
- MARKU-U: MARKU for *Uploading Concurrently*: (in Subsection 6.2.3)
- Merged MARKUs (in Subsection 6.2.4)

After describing generalities about the following plots and variables, we start with the MARKU-R which is followed by the MARKU-U. We conclude this section by discussing differences and similarities of the MARKUs and the effect of them on possible optimizations.

6.2.1 Generalities about MARKUs

The plots shown in this section are, if not otherwise noted, split for the two graphics cards. The top row represents the TITAN card, the bottom row the RTX card. The plots visualize the effects of one control variable, displayed on the x-axis, on the time needed in ms for performing the task at hand, displayed on the y-axis. A blue line represents the predicted form of the modeled effect. Gray dots visualize partial residuals to give an idea on the precision of the model. The partial residuals show for the given configurations the residual errors when removing the influence of other effects than the shown one(s). The more distributed these points are, the less accurate the model predicts that configuration. Furthermore, if present, the gray band around the blue line visualizes the confidence bands around the prediction. The visualizations are produced using the R package `visreg`. Additionally, the response variable, i.e. the time needed for rendering or uploading, is back-transformed from the log transformation to get easier understandable values. Further information about this package, partial residuals, how they are calculated, how the confidence bands are calculated can be found in [BB17].

Interactions

Most control variables have interactions modeled with other control variables. This means that analyzing them alone can be misleading. Therefore, in most cases we will analyze the effect of one control variable with their corresponding interactions. For these cases, each row consists of more than one plot, representing different settings for the interaction variable. The set value of the additional variables are written on top of each plot.

Interactions are described in the following formulas with an asterisk $*$ (please note that not all formulas include the asterisk). An interaction can occur between two or more control variables. If we have an interaction, this means for the formula that each of the set values for the control variables are multiplied together. Each term, may it be a single term, e.g. size, or an interaction term, e.g. size$*$gpuClock, has an estimated coefficient which is multiplied with the values set for the variable.

Nomenclature of Control Variables

The different control variables are as follows:

time	Time needed for completing the task at hand (in ms)
size	Size of the dataset in MB
cpuClock	CPU clock rate in GHz
nPartition	Numbers of partitions
gpuClock	GPU clock rate in GHz

memClock Graphics memory clock rate in GHz

method Used method as a number

bufferUsageHint The used buffer usage hint as a number

pbo Using or not using PBOs (encoded as either 1 or 0; 1 stands for using PBOs)

dsa Using or not using DSA (encoded as either 1 or 0; 1 stands for using DSA)

card The used graphics card, either TITAN or RTX

The used uploading method for uploading is encoded as the following values:

Value	Uploading method
0	`glBufferData`
1	`glMapBuffer + memcpy + glUnmapBuffer`
2	`glMapBufferRange + memcpy + glUnmapBuffer`
3	`glBufferSubData`

or, if applicable, the *named* versions as described in Chapter 6 for DSA usage with the same numbers. The buffer usage hints are encoded as the following values:

Value	Buffer usage hint	Value	Buffer usage hint	Value	Buffer usage hint
0	`GL_STREAM_DRAW`	1	`GL_STATIC_DRAW`	2	`GL_DYNAMIC_DRAW`
3	`GL_STREAM_COPY`	4	`GL_STATIC_COPY`	5	`GL_DYNAMIC_COPY`
6	`GL_STREAM_READ`	7	`GL_STATIC_READ`	8	`GL_DYNAMIC_READ`

Derivation of the MARKUs

The MARKUs are derived using the R function `lm` for linear regression and the function `step` for iteratively adding and subtracting terms and evaluating the BIC, to decide which terms to include. Both functions are part of the R package `stats`, which is part of the core of R. More information about this package can be found in [R C19]. For the `lm` function we use the following formula as input argument:

$$\begin{aligned}\text{time} = {} & method + bufferUsageHint + dsa + pbo + card + size \\ & + poly(size, 2) + gpuClock + poly(gpuClock, 2) + memClock \\ & + poly(memClock, 2) + nThread + poly(nThread, 2) + nPartition \\ & + poly(nPartition, 2) + poly(nPartition, 3) + cpuClock + poly(cpuClock, 2)\end{aligned} \tag{6.1}$$

The operator *poly* stands again for creating a polynomial of the first argument with degree of the second argument. Additionally, the remaining amount of measurements after outlier removal is used as weights for the summarized data points. For example, if one configuration has 30 measurements and 5 are removed as outliers, the weight of the summarized measurements is 25. For another configuration with 30 measurements and only 1 outlier, the weight is 29.

The `step` function uses the resulting model obtained from the `lm` function. Additionally, we define as upper model limit all four way interaction of the included terms from the `lm` model. This means that for example an interaction between *method*, *bufferUsageHint*, *poly(gpuClock,2)* and *poly(nPartition,3)* can also be part of the final model and is tested for change of the BIC.

6.2.2 MARKU-R: MARKU for *Rendering Concurrently*

In this subsection we take a look at the MARKU-R. This subsection is is divided into the following parts:

- Size
- Number of Partitions
- Clock Rates
- Software Design

The MARKU-R takes the form of the following formula:

$$\begin{aligned}\text{time} = {} & method + dsa + card + poly(size, 2) + poly(gpuClock, 2) \\ & + poly(memClock, 2) + poly(nThread, 2) + poly(nPartition, 3) \\ & + poly(cpuClock, 2) + poly(gpuClock, 2) * poly(memClock, 2) \\ & + method * dsa + poly(gpuClock, 2) * poly(nPartition, 3) \\ & + poly(gpuClock, 2) * poly(cpuClock, 2) + card * poly(nThread, 2) \\ & + card * poly(memClock, 2) + card * poly(gpuClock, 2) + method * card \\ & + card * poly(nPartition, 3)\end{aligned} \tag{6.2}$$

As before, the operator *poly* stands for creating a polynomial of the first argument with degree of the second argument. We start by analyzing the dataset parameters, namely size and number of partitions. As next we analyze the effect of the different settable clock rates on performance. This is followed by the analysis of the software design influences, namely the used method, buffer usage hint, usage of DSA and usage of PBOs or not, and the number of threads used for uploading.

Size

For rendering data, obviously the amount of work to do has an important impact on how long it takes. This is also reflected in the MARKU-R. Size is modeled alone as polynomial with degree 2 and without any interaction. Fig. 6.7 shows the expected influence of size on rendering time. We see that increasing the size of the dataset also increases the time needed to render it.

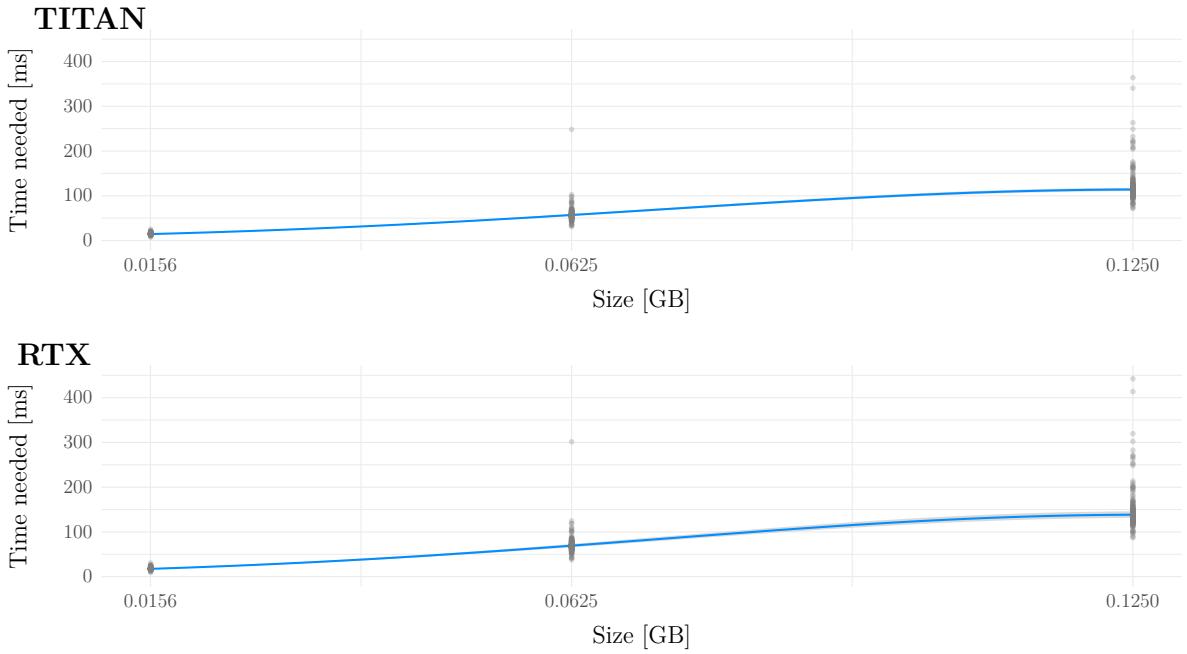


Figure 6.7: Predicted influence of size on rendering times shown for three different numbers of partitions.

Number of Partitions

The influence on rendering times of the number of partitions is modeled as third degree polynomial and with an interaction with both, the GPU clock rate (as quadratic polynomial) and the card used, individually. Fig. 6.8 shows this interaction with three plots for each card for the clock rates 595 MHz, 1.063 GHz and 1.519 GHz for the TITAN card and 300 MHz, 1.2 GHz and 1.845 GHz for the RTX card. The general tendency for both cards is that more partitions improve the performance for rendering. Another tendency shown is that the higher the GPU clock rate, the lower the time needed for rendering. The interaction between GPU clock rate and number of partitions is visible when we compare the slope and curvature of the blue prediction line for the individual GPU clock rates. The higher the GPU clock rate, the stronger the influence of the number of partitions and consequently, the lower the time needed for rendering. The interaction with the card is visible when we look at the difference between the low GPU clocks of both cards. While

for the TITAN card, using more partitions in combination with the low GPU clock can increase the time needed, for the RTX card, using more partitions always results in lower rendering times.

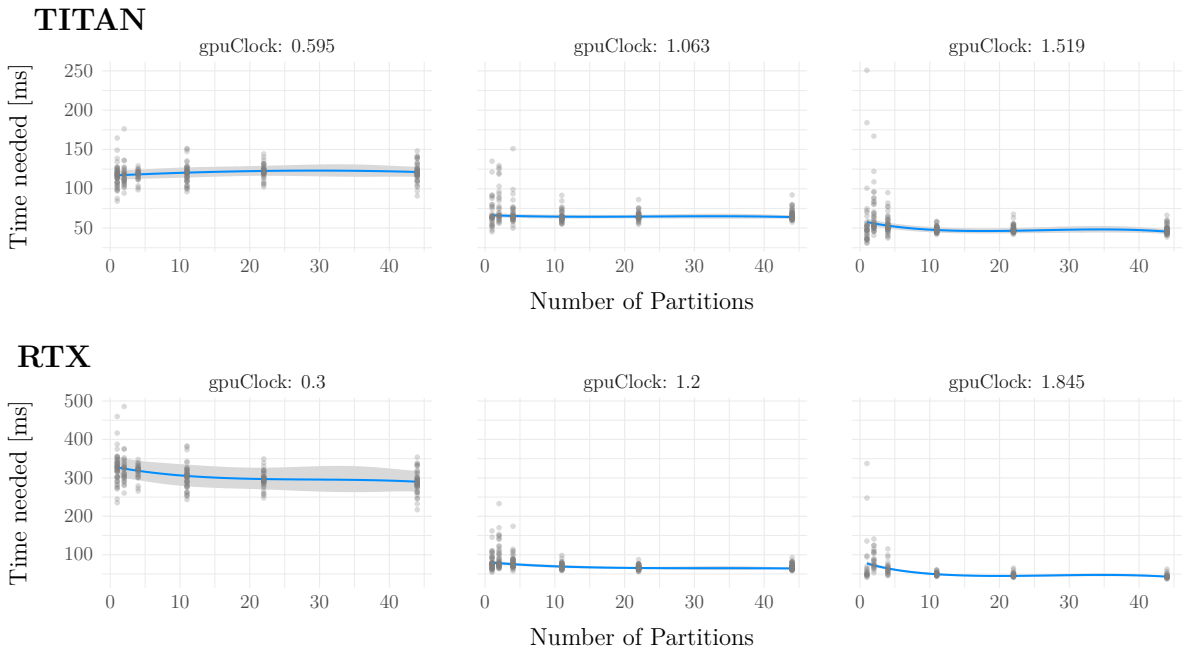


Figure 6.8: Predicted influence of number of partitions on rendering times shown for three different GPU clock rates.

Clock Rates

If we analyze the influence of the clock rates of hardware components on the graphics card, namely the GPU and graphics memory clock rates, we can also observe an interaction between both as quadratic polynomial. Graphics memory clock rate is modeled with that interaction, alone as quadratic polynomial and additionally with an interaction with the used card. Fig. 6.9 shows the corresponding plots for its influence. The left plots show the effect of changing the GPU clock rate on the needed time for rendering for the low graphics memory clock rates and the right plots for the high graphics memory clock rates. For both cards, we see that having a high graphics memory clock rate alone does not necessarily improve rendering times, yet together with a high GPU clock rate, they outperform high GPU clock rates alone. The interaction with the card is clearly visible for low graphics memory clock rates: While increasing GPU clock rates for the RTX card always increases the performance, for the TITAN card this is not necessarily true. Increasing the GPU clock further than 1.5 GHz theoretically decreases the performance for the low graphics memory clock rate. However, that part is without measurements as these clock rates are not tested. Therefore, these predictions can be without any meaning.

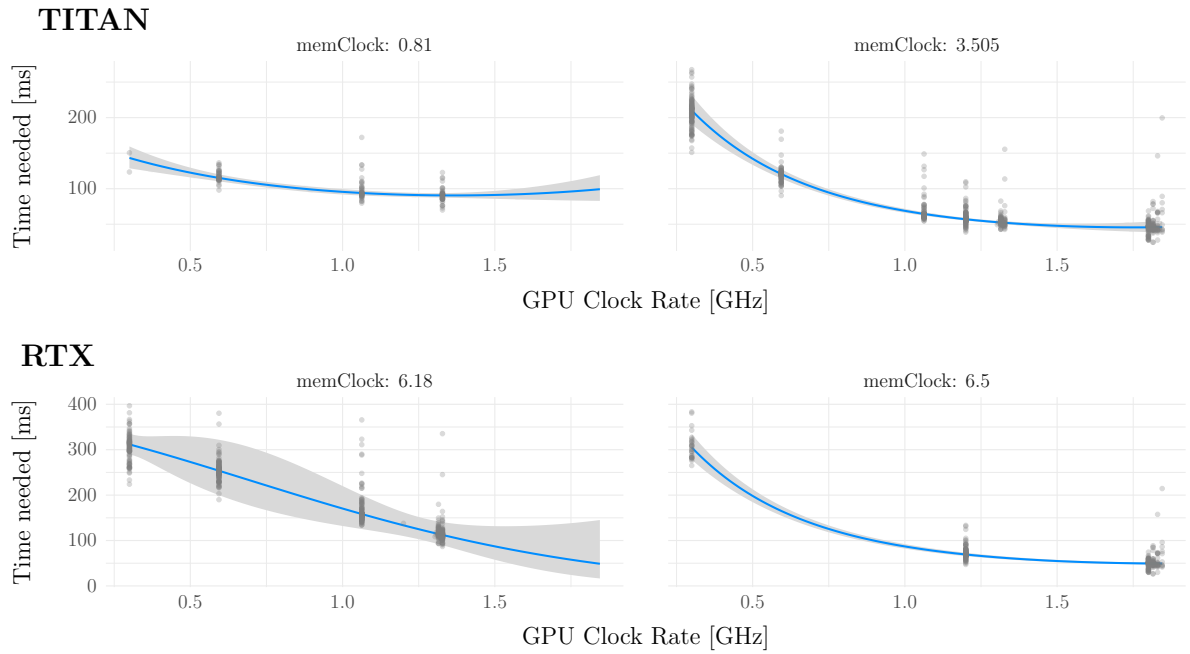


Figure 6.9: Predicted influence of GPU clock rate on rendering times shown for two different graphics memory clock rates.

The GPU clock rate additionally to the before shown interactions and its influence alone has an interaction with the CPU clock rate and the used graphics card individually.

The influences are shown Fig. 6.10 for three different GPU clock rates for each card. We see that while increasing the CPU clock rate for the lowest GPU clock rate can slightly increase the time needed for rendering (e.g. from 1.2 GHz to 2.0 GHz for the TITAN card), this effect is turned around for higher clock rates. For the highest GPU clock rate the performance increase is visible the clearest.

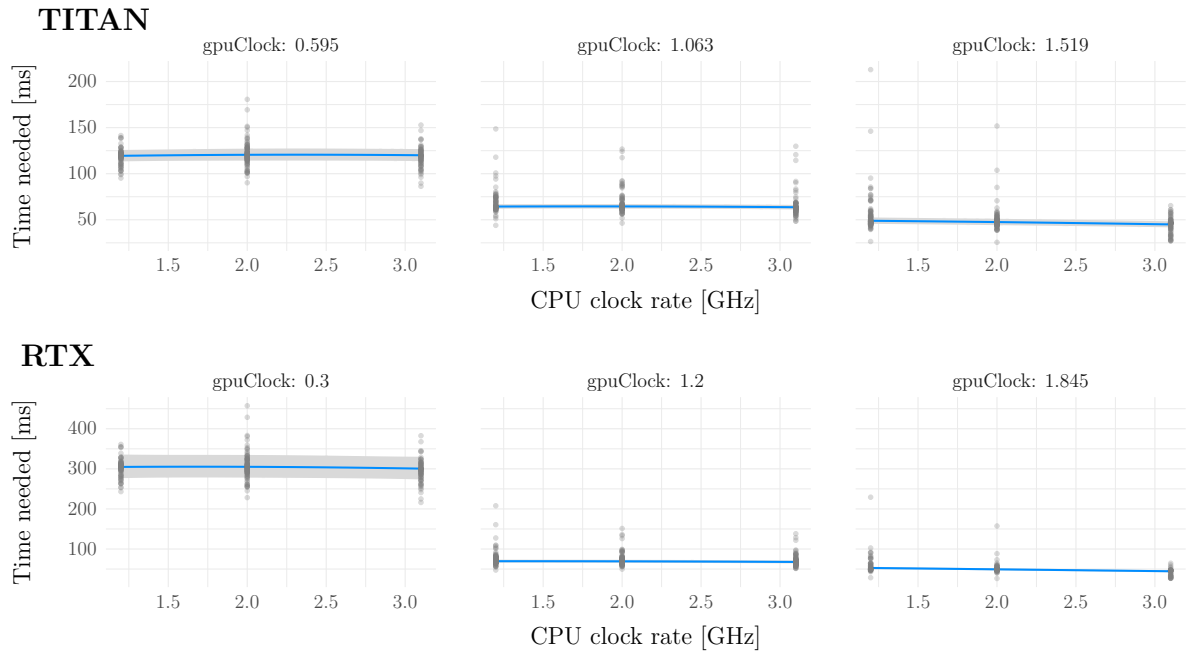


Figure 6.10: Predicted influence of CPU clock rate on rendering times shown for three different GPU clock rates.

Software Design

The influence of the used method is modeled alone, together with using DSA or not and the used graphics card. Their influences are visualized in Fig. 6.11. Left shows not using DSA and right shows using DSA. The numbers on the x-axis represent the method used. Please note that we limited the range of the y-axis to a range of 0 ms to 150 ms. This caused the exclusion of 7 partial residual for the TITAN and 10 partial residuals for the RTX card due to them being outside that range.

The difference between method 1, 2 to 0 and 3 for non DSA usage is almost not visible in comparison to using DSA; for the latter it is significantly increased. While using DSA has similar behavior for both cards, not using DSA is different for each card. For the card TITAN, methods 1 and 2 need slightly more time for rendering than method 0 and 3; for the card RTX this is turned around. Here method 1 and 2 are slightly faster than method 0 and 3.

The influence of the number of used uploading threads is shown in Fig. 6.12. The MARKU-R contains this variable alone in quadratic form and with an interaction with the used graphics card. Wee see that using more threads increases the time needed for rendering. The difference between the cards is a difference in the slope.

The remaining software design variables buffer usage hint and usage of PBO or not are not part of the model and can be expected to only have a minor or no impact on rendering times.

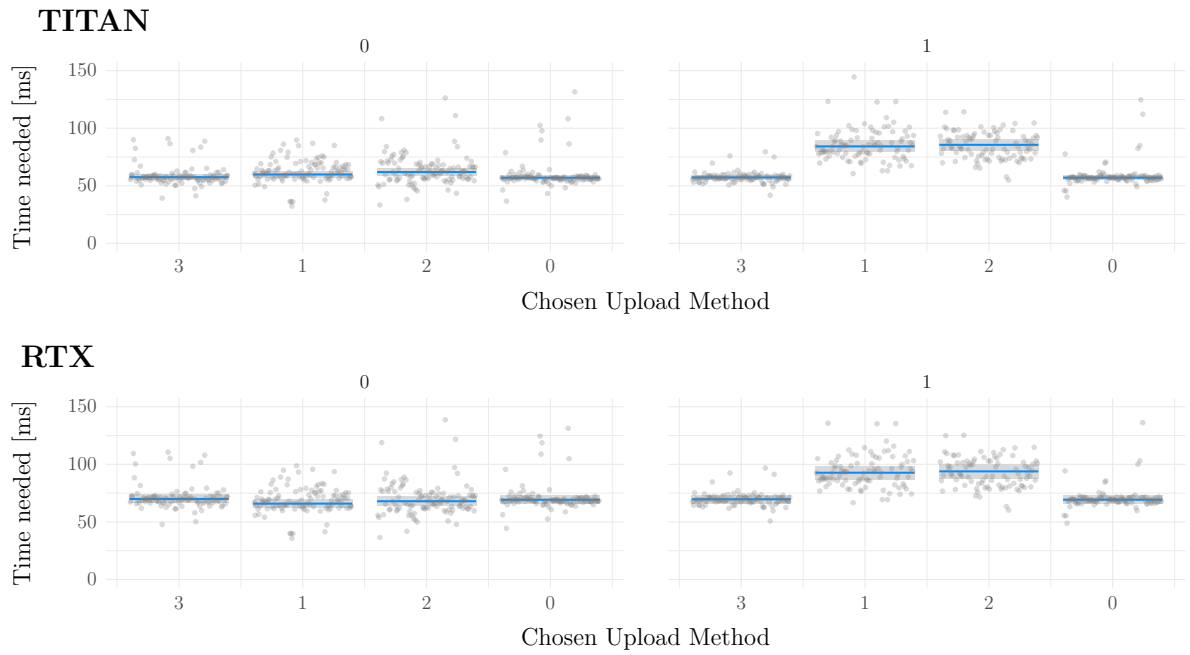


Figure 6.11: Predicted influence of used method on rendering times shown for either not using (left) or using (right) DSA.

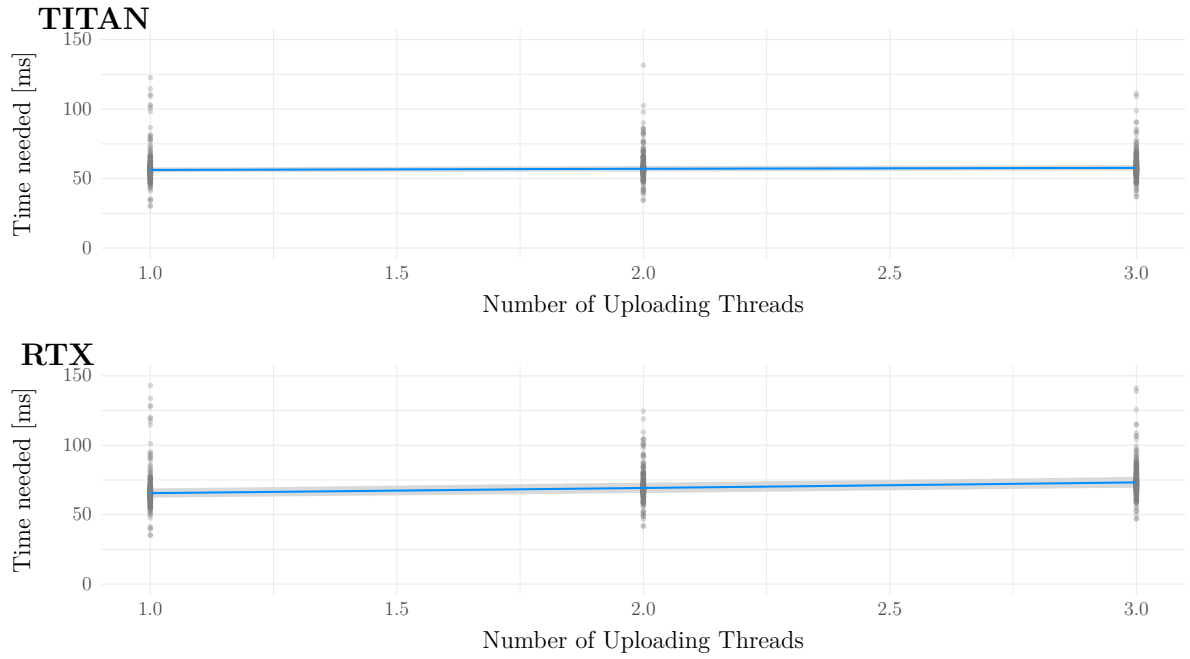


Figure 6.12: Predicted influence of the number of used uploading threads on rendering times.

Summary of MARKU-R

From the subsections before, we see there are quite a number of possible influences and interactions of the influences depicted by the MARKU-R. All quantitative variables, except number of partitions, are modeled as second degree polynomials; number of partitions is modeled as third degree polynomial. While size plays an important role in determining how fast rendering can be done, other variables determine performance of rendering as well.

First of all, the different clock rates all positively influence the performance and thus reduce the time needed for rendering . However, if the graphics card cannot process fast enough the commands issued by the CPU, increasing the CPU clock rate does not necessarily increase performance. Even worse, with the lowest settable GPU clock rates, performance can be decreased for the TITAN card. Partitioning of the dataset itself, without changing the size of the dataset, also reduces rendering times unless lower GPU clock rates are used; in that case the time can be increased for the TITAN card. Software choices also significantly influence the performance for rendering while concurrently uploading data. However, using different buffer usage hints as well as using PBOs or not, do not, according to the MARKU-R, influence the outcome. On the other hand, changing the method used for uploading increases slightly the time for rendering when using method 1 or 2 for the TITAN card or decreases it slightly for the RTX card. In combination with DSA, for both cards an increase from method 1 or 2 to method 0 and 3 is modeled. Using more threads for uploading slightly increases rendering times.

6.2.3 MARKU-U: MARKU for *Uploading Concurrently*

In this subsection we discuss the MARKU-U divided into the following parts:

- Size
- Number of Partitions
- Clock Rates
- Software Design

Eq. (6.3) shows the formula that constitutes the MARKU-U for the time needed for performing an upload of data while concurrently rendering. For the analysis, we start again with dataset parameters, namely size and numbers of partitions. This is followed by describing the influence of the settable clock rates and the influence of the software design variables. For all we include interactions with other control variables in the order of their first appearance within the order of the main effects described here.

$$
\begin{aligned}
\text{time} = {} & method + bufferUsageHint + dsa + card + poly(size, 2) \\
& + poly(gpuClock, 2) + poly(memClock, 2) + poly(nThread, 2) \\
& + poly(nPartition, 2) + poly(cpuClock, 2) + bufferUsageHint * dsa \\
& + method * poly(nPartition, 2) + method * dsa + method * poly(gpuClock, 2) \\
& + method * poly(memClock, 2) + card * poly(size, 2) \\
& + poly(size, 2) * poly(nPartition, 2) + dsa * poly(nThread, 2) \\
& + method * poly(cpuClock, 2) + method * bufferUsageHint \\
& + method * poly(nThread, 2) + card * poly(cpuClock, 2) \\
& + dsa * poly(nPartition, 2) + card * poly(memClock, 2) \\
& + method * bufferUsageHint * dsa
\end{aligned}
\tag{6.3}
$$

Size

The size of the dataset is modeled as polynomial with degree 2. It has interactions with the number of partitions and the used card. Fig. 6.13 shows the influence of size on the x-axis for different numbers of partitions (left, middle and right plot). Please note, the y-axis is fixed to a range between 0 ms and 3000 ms to clarify the image. In this process, 7 partial residual for the TITAN card and 3 for the RTX card are removed as they are outside that range.

We can observe that increasing the size of the dataset to be uploaded increases the time needed for finishing the upload. This effect is increased by increasing the number of partitions into which the dataset is split into. This means that using more partitions for one dataset increases the impact of the size of that dataset. Having a large size with many partitions constitutes the worst performance for this relationship. Additionally, the confidence bands become broader the more partitions are used. The interaction with the used card can be seen when we compare the slopes of both rows. For the TITAN card, there is a stronger increase in uploading times when using bigger in size datasets.

Number of Partitions

The MARKU-U includes an interaction of number of partitions with the chosen upload method, the size (as seen before) and using DSA or not. Its influence is modeled as second degree polynomial and also alone part of the formula. Numbering of the method is the same as in Subsection 6.2.2. Fig. 6.14 shows the influence of the numbers of partitions (x-axis) on uploading times (y-axis) for the four different uploading methods (left to right: 3, 1, 2, 0). While the numbers of partitions are modeled to have a strong increasing effect on the time needed for method 3 and 0, the effect for method 1 and 2 in comparison is almost negligible. For method 1 for the TITAN card, using 1 partition instead of 44 decreases the time needed from about 12 ms to 10 ms, while for method 3 the times needed for 44 partitions is about 613 ms which is decreased to 83 ms for 1 partition.

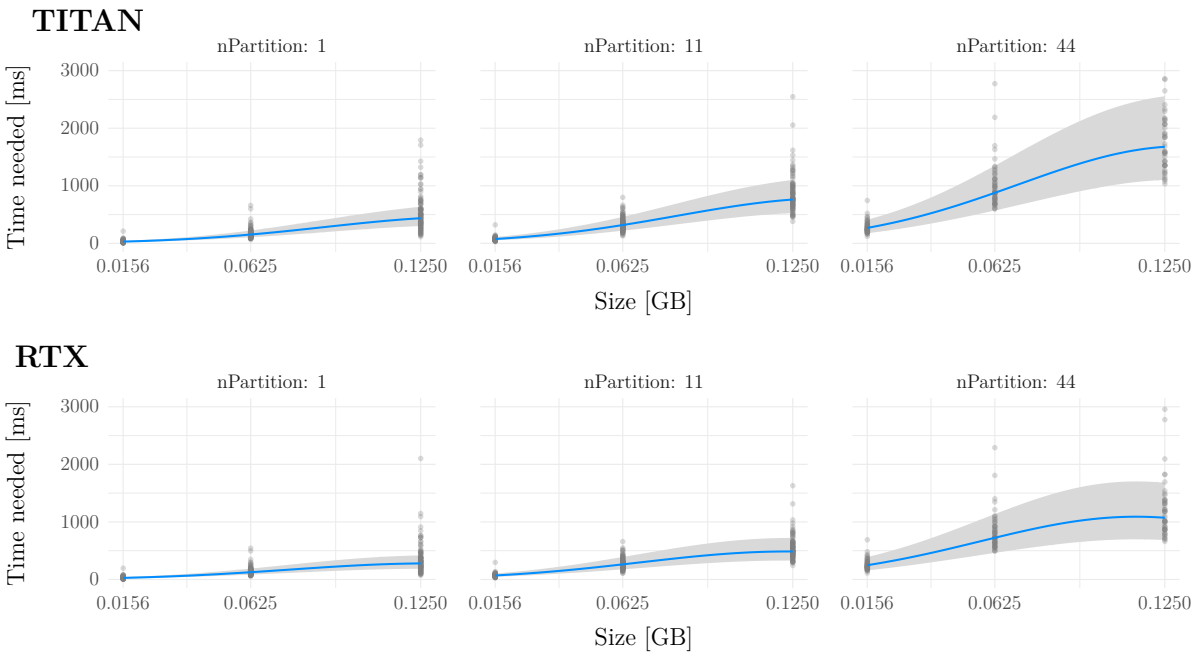


Figure 6.13: Predicted influence of dataset size on uploading times shown for three different numbers of partitions.

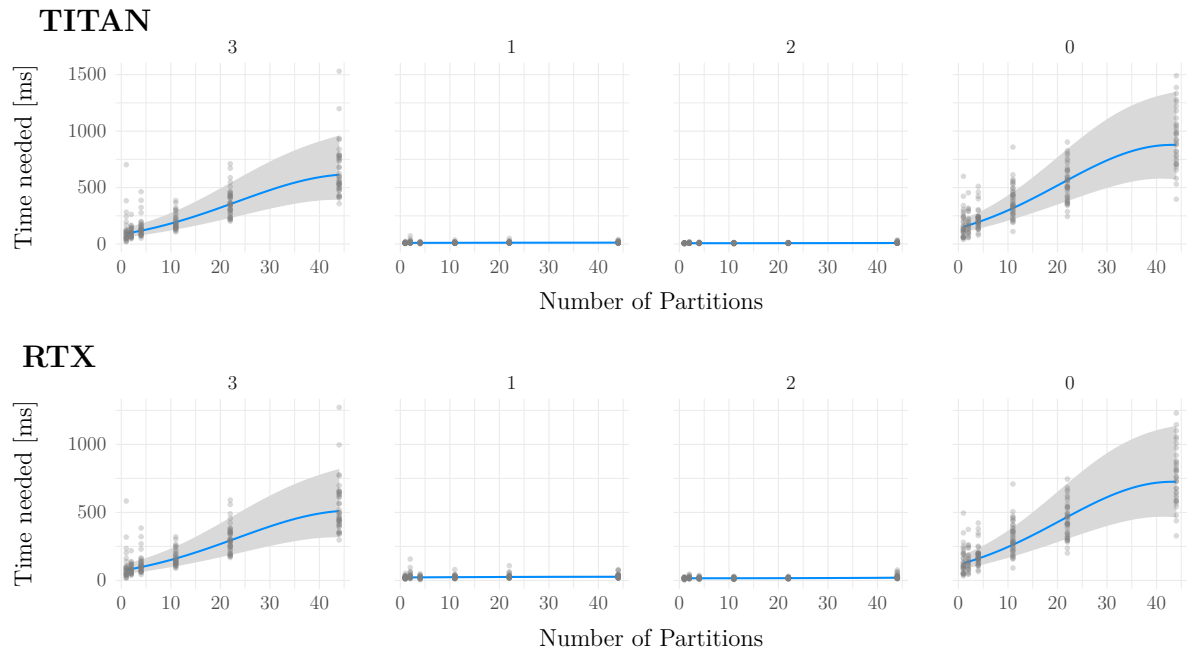


Figure 6.14: Predicted influence of number of partitions on uploading times shown for the four different used methods.

The interaction with using or not using DSA is shown in Fig. 6.15. The interaction has only a small effect. Using DSA decreases the time needed when using 44 partitions from 877 ms for the TITAN card to 844 ms, for the RTX card from 724 ms to 697 ms.

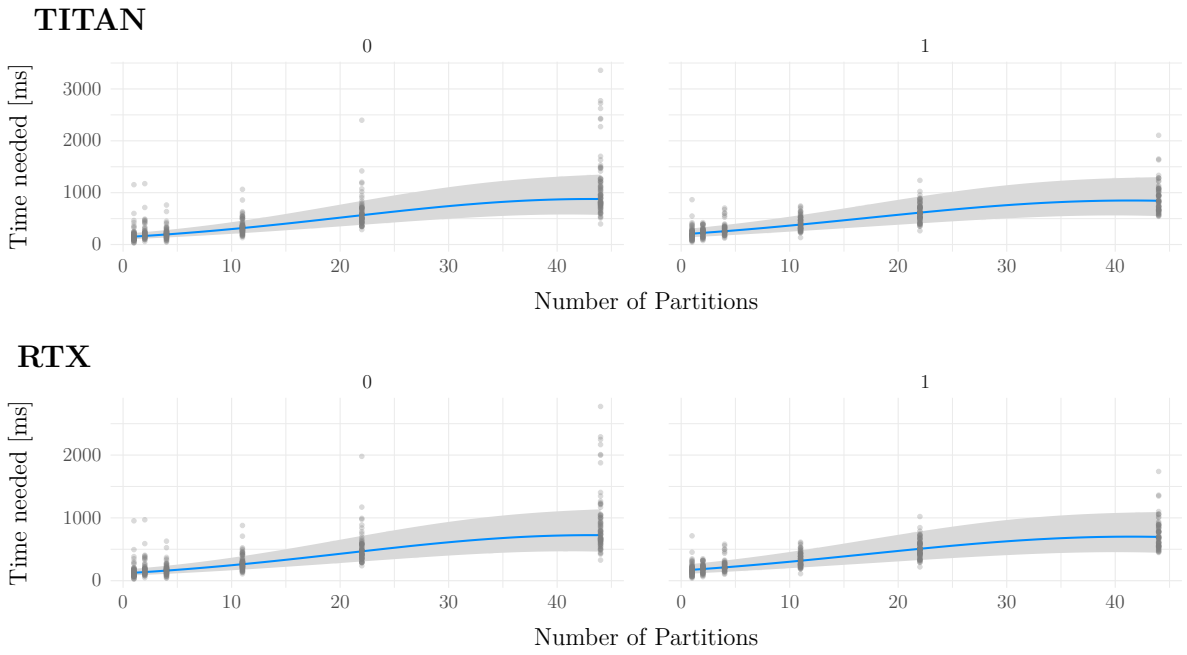


Figure 6.15: Predicted influence of number of partitions on uploading times shown for not using (left) and using (right) DSA.

Clock Rates

For the queried graphics memory clock rates, an interaction with the used uploading method and the used card is modeled. Its influence has the form of a second degree polynomial and is also alone part of the formula. Fig. 6.16 shows the influence of graphics memory clock rate on needed uploading time for the four different uploading methods from left to right.

Again, we see that method 0 and 3 have a different behavior than 1 and 2. Using higher graphics memory clock rates usually decreases the time needed for uploading a dataset, but for method 0 and 3 the overall time and overall reduction is a lot higher than for method 1 and 2. For example, changing the graphics memory clock rate for the TITAN card using method 0 from 6.5 GHz to 0.81 GHz increases the time needed for uploading from about 82 ms to 451 ms, while for method 1 the predicted times are changed from 15.2 ms to 19.0 ms. For the TITAN card the most time is needed for methods 3 and 0 when using a graphics memory clock rate of about 1.8 GHz to 2 GHz. Please note, that these values might be fitting errors as we do not have any measurements in that area, which is also illustrated by the large confidence band around the prediction line. For the RTX card

we see for these two methods that there is also a lot of uncertainty in graphics memory clock rates lower than 3 GHz. This again is caused by not having many measurements for these clock rates.

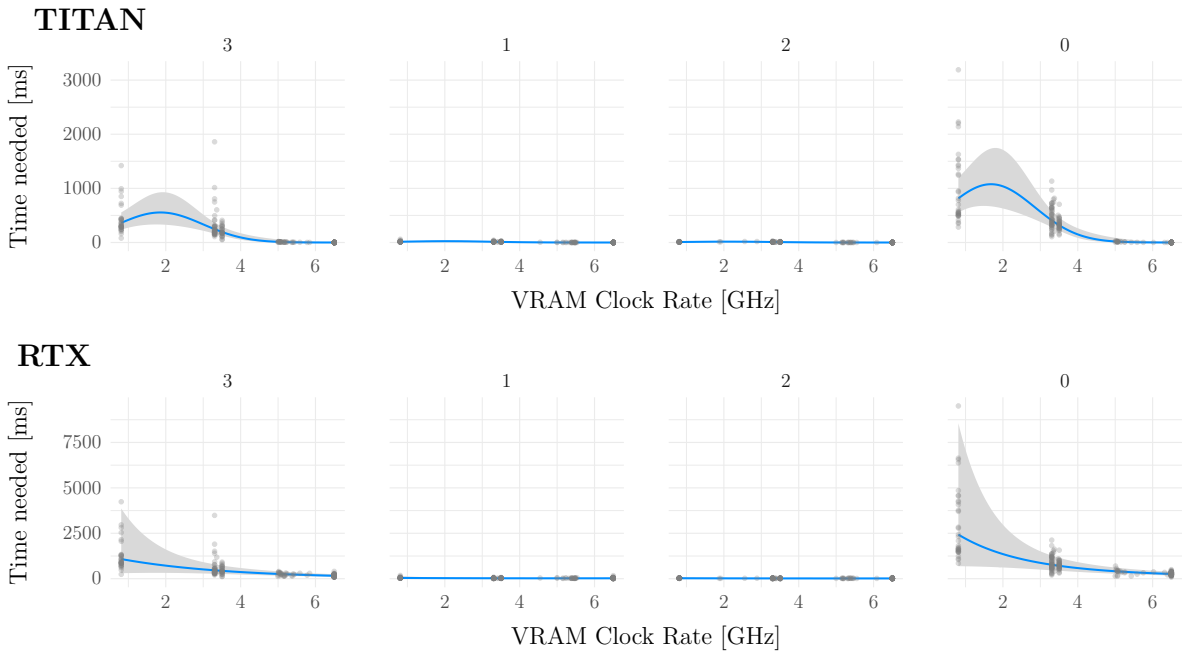


Figure 6.16: Predicted influence of graphics memory clock rate on uploading times shown for the four different used methods.

A similar behavior can be seen for GPU clock rates, as shown in Fig. 6.17. Increasing the GPU clock rate decreases the time needed for uploading a dataset. However, for method 0 and 3, this effect is a lot stronger than for method 1 and 2. For the latter, there is almost no effect for both cards visible. We also see that method 3 and 0 have a lot of uncertainty, visualized by the grey confidence bands, in comparison to method 1 and 2.

The CPU clock rate is modeled as second degree polynomial, alone and with an interaction with the used method and the used graphics card. Fig. 6.18 shows the corresponding plots. We can again see that the methods 1 and 2 have a different behavior than method 0 and 3. For method 0 and 3, increasing the CPU clock rate decreases the time needed for uploading for clock rates lower than 2.0 GHz; using 3.0 GHz however can increase the time needed. This effect in not visible for method 1 and 2.

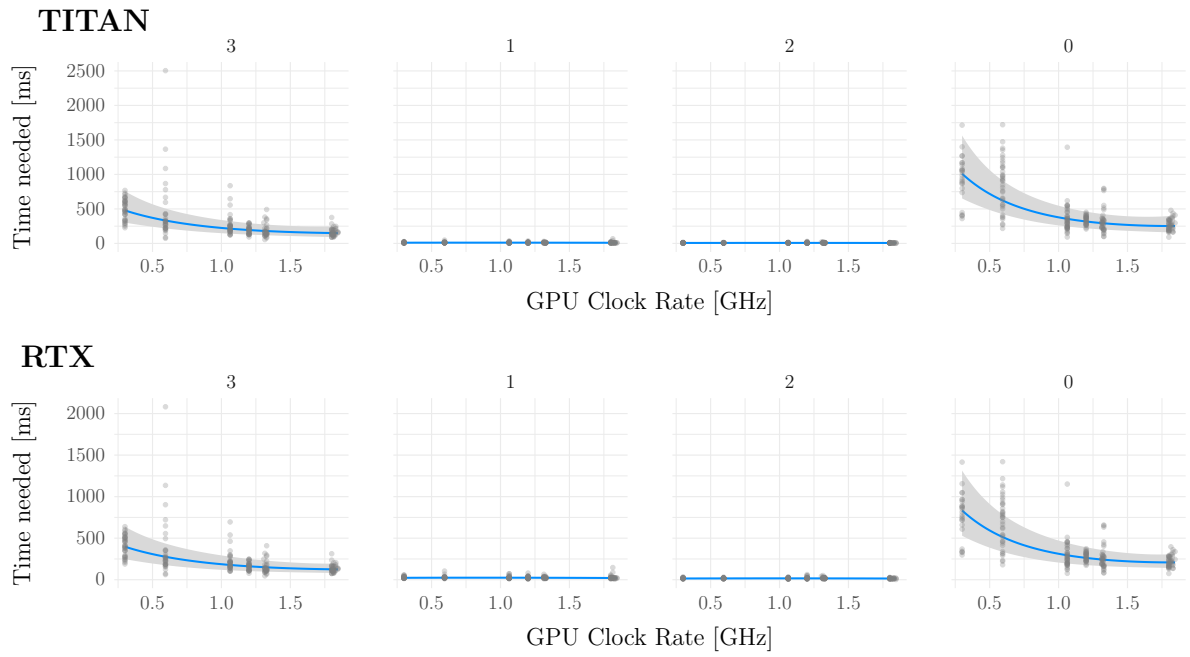


Figure 6.17: Predicted influence of GPU clock rate on uploading times shown for the four different used methods.

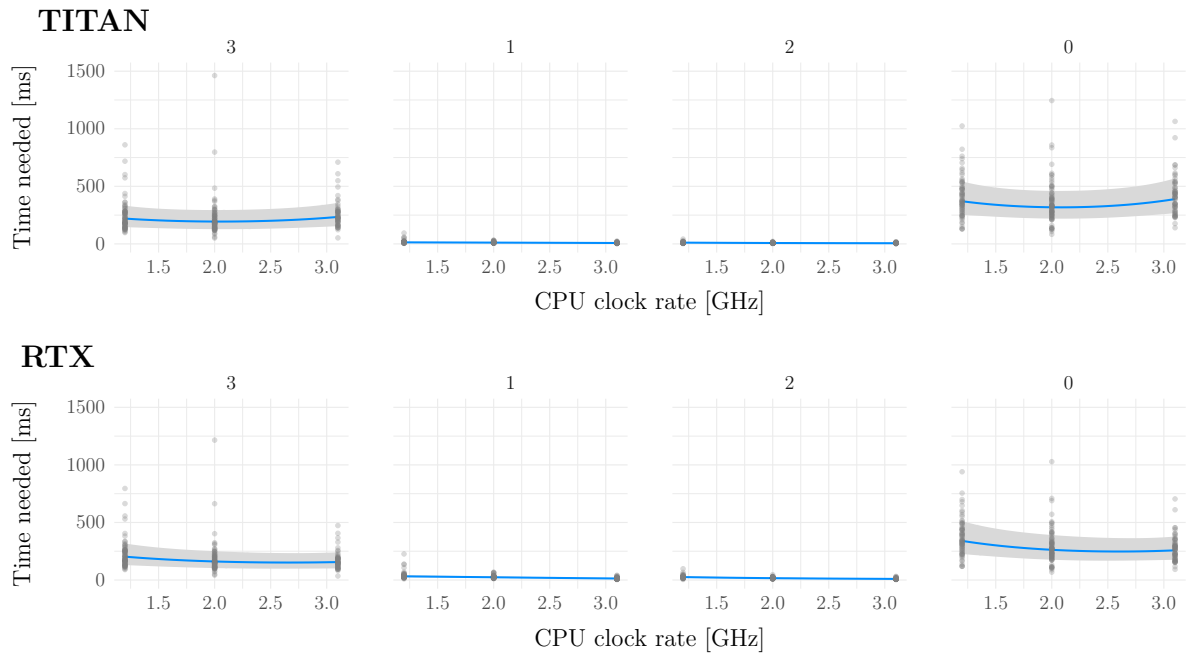


Figure 6.18: Predicted influence of CPU clock rate on uploading times shown for the four used methods.

Software Design

Apart from the already shown interactions, MARKU-U has the used uploading method also as an interaction with using DSA or not, the buffer usage hint and the number of threads for uploading. We also have a three way interaction between method, buffer usage hint, and using DSA or not.

Method Fig. 6.19 shows the influence of the used method on uploading times for using DSA (right) and not using DSA (left). For all methods, using DSA offsets positively the time needed for uploading. Furthermore, we see that method 1 and 2 need in all cases less time for uploading.

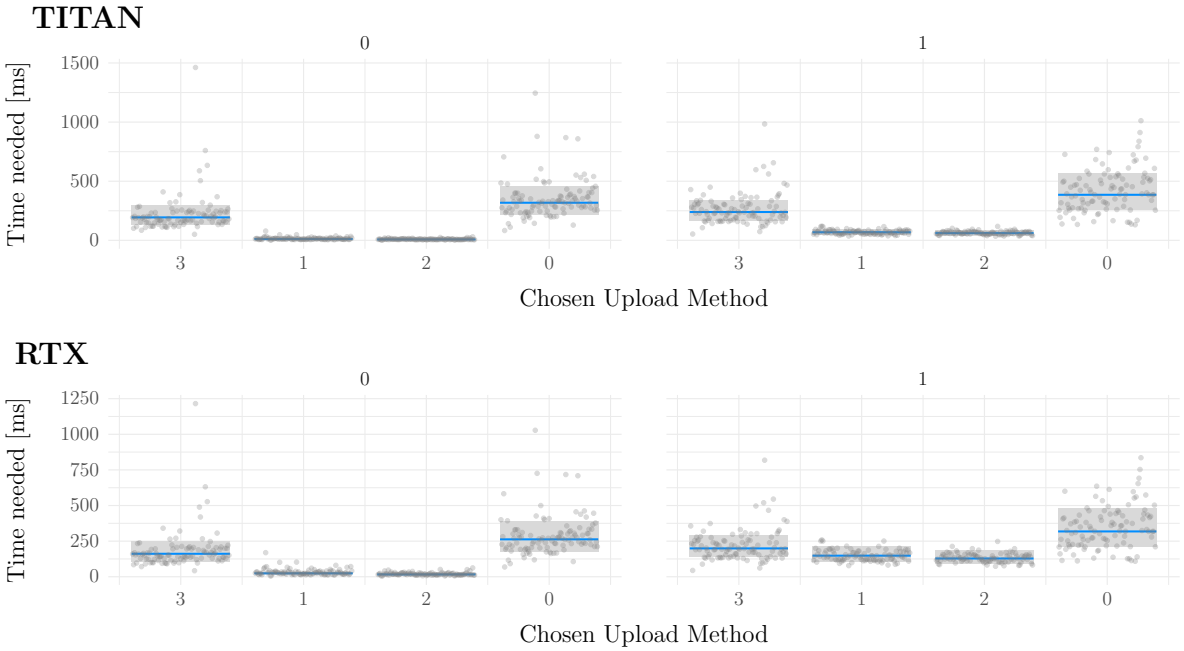


Figure 6.19: Predicted influence of the used method on uploading times shown for not using (left) and using DSA (right).

The interaction between method and the number of threads is shown in Fig. 6.20, for using 1 (left), 2 (middle) and 3 (right) threads for uploading. We can see that using more threads decreases the overall time needed for uploading and decreases the difference between method 3 and 0 to method 1 and 2.

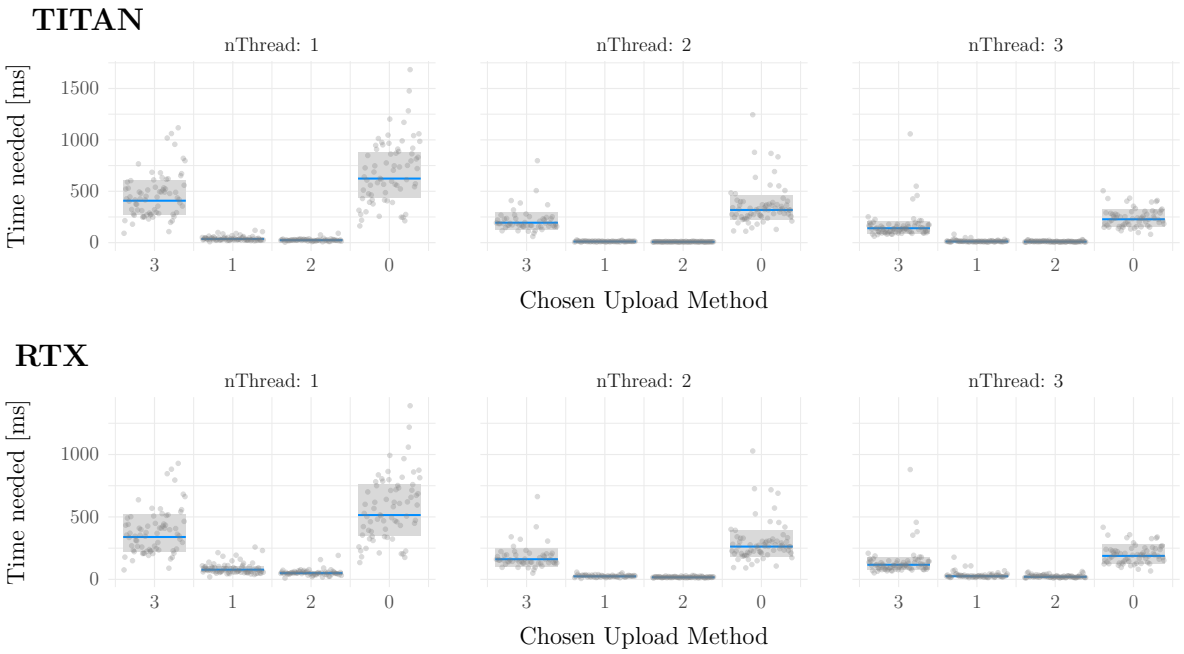


Figure 6.20: Predicted influence of the used method on uploading times shown for using 1 (left), 2 (middle) or 3 (right) threads.

Buffer Usage Hints The influence of using DSA or not on the buffer usage hint is shown in Fig. 6.21. For buffer usage hint 0 and 7 and 8, the time needed for uploading is decreased if we use DSA. However, this is not the case for buffer usage hint 4, 5, 6, 1, 2, and 3. For these hints, the time needed is decreased when we are **not** using DSA. However, we see that almost all prediction lines are within the range of the confidence bands, so the actual effect might be negligible. Please note that in this plots, no partial residuals are plotted for clarity. Furthermore, some of the partial residuals would be outside of the range of the shown y-axes.

The interaction of buffer usage hint and method is shown in Fig. 6.22. For method 3 buffer usage hints 5, 1, 2, 3, 7 are very similar. Buffer usage hints 6, 0, and 8 have the lowest times. For method 0 the buffer usage hints are quite similar, with 1, 2, and 3 a little lower than the others. However, for method 1 and 2, buffer usage hints 5 and 3 lead to distinctly higher times for uploading than the remaining.

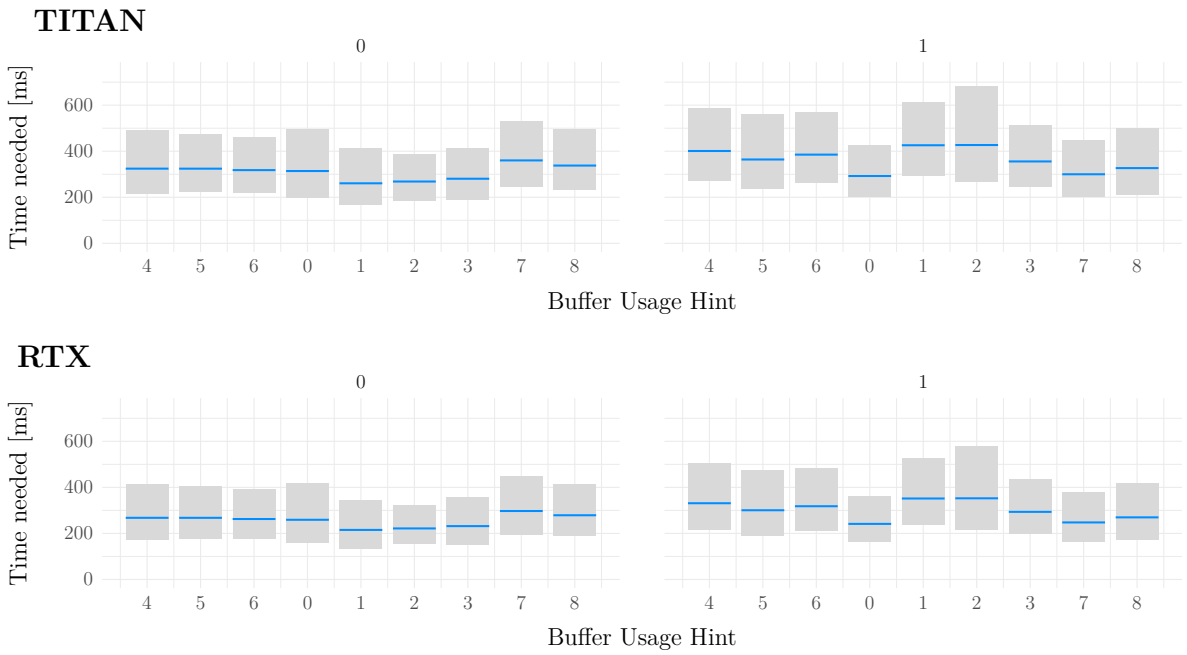


Figure 6.21: Predicted influence of specified buffer usage hints on uploading times shown for not using (left) or using (right) DSA.

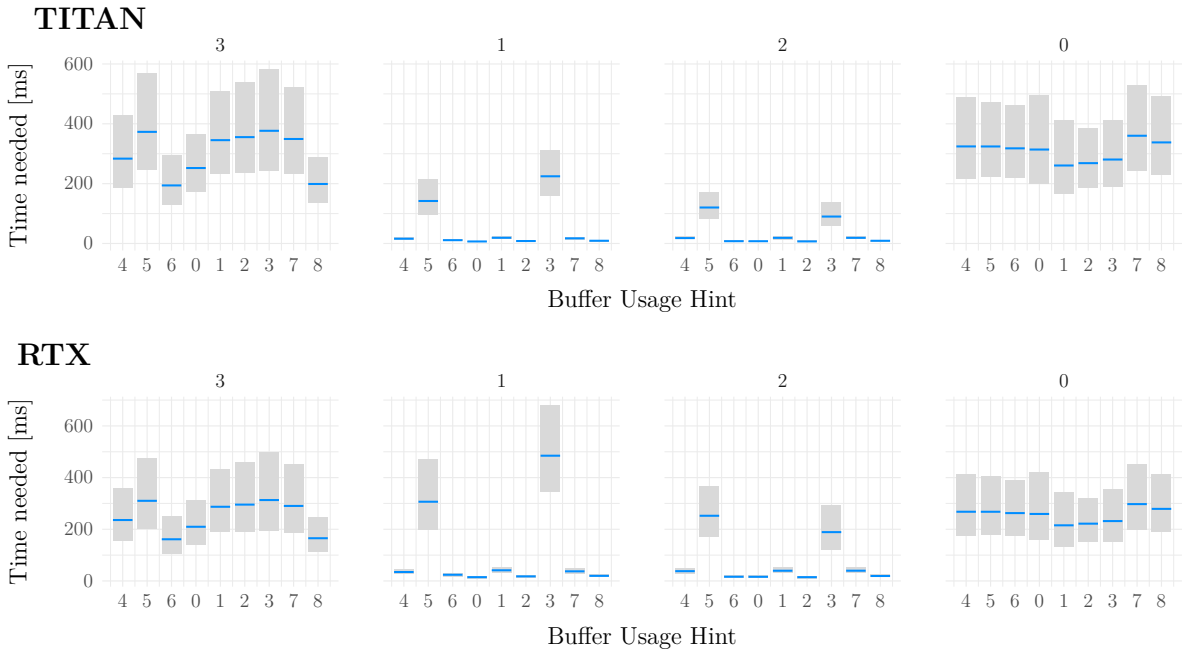


Figure 6.22: Predicted influence of specified buffer usage hints on uploading times shown for the four used methods.

Number of Threads The number of threads used for uploading is additionally modeled with an interaction with using DSA or not using DSA and alone as second degree polynomial. A visualization of that interaction is shown in Fig. 6.23. For these plots, the range for the y-axis is fixed between 0 ms and 1000 ms. Following that, 31 partial residuals for the TITAN card and 16 for the RTX card are not plotted as they are outside that range. We see that increasing the number of threads from 1 over 2 to 3 for both cards reduces the time needed for uploading. Using DSA or not has an effect on the curvature of the prediction line. As usually no half threads can be used, this difference is negligible. Additionally, the slopes are different for using DSA or not. While using 1 thread without DSA results for the TITAN card in 623 ms and for not using DSA in 489 ms, using three threads requires an estimated 227 ms for not using DSA and 266 ms for using DSA. We see that when we use multiple threads, not using DSA is the better choice regarding low uploading times. When we can only use one thread, using DSA results in lower times. This is true for both cards.

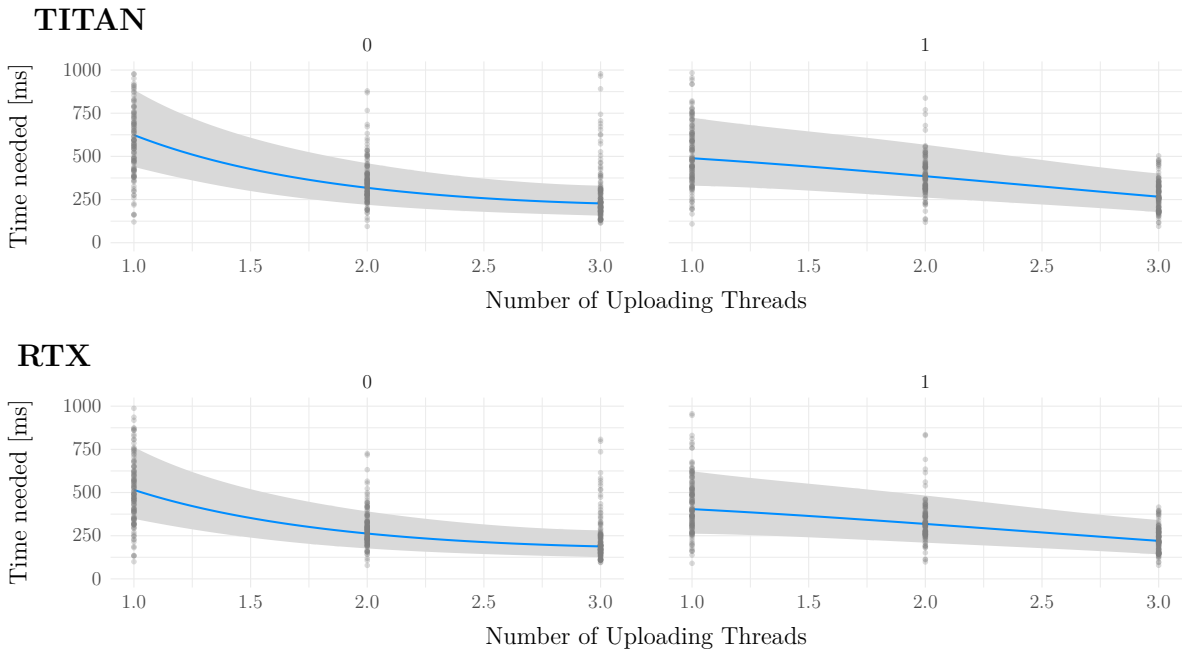


Figure 6.23: Predicted influence of number of the number of threads used shown for not using (left) and using (right) DSA for uploading on uploading times.

Finally, the usage of PBOs is not modeled and we assume its influence to be negligible.

Summary of MARKU-U

The MARKU-U describes influences by almost all tested control variables for uploading data while concurrently rendering. All included quantitative control variables are in the form of a second degree polynomial. All control variables except using PBOs or not are

included as main effects. The remaining terms of the formula are, except for one which is a three way interaction, all two way interactions.

The size of an dataset increases the time needed for uploading it. Dividing that dataset into more partitions also increases the time needed and widens the confidence bands of the prediction. The effect of using method 0 and 3 on the effect of using more partitions is an increase of that effect with a broader confidence band. In comparison to that, the effect of using DSA or not interacting with the number of partitions is almost negligible. Increasing the graphics memory clock rate usually reduces the time needed for uploading. The GPU clock reduces the time needed the higher it is. Method 0 and 3 again increase this effect but also add uncertainty. In comparison to that, for both, graphics memory and GPU clock rates, almost no effect is visible for method 1 and 2. Using values lower for the CPU clock rate (from 1 GHz to 2 GHz) can increase performance when increased, but also also reduce performance when increased at higher values (from 2 GHz to 3 GHz), when done in combination with method 3 and 0.

For software choices almost all effects influence the time needed for uploading data. Using DSA is modeled to be worse than not for all 4 uploading methods. However, the effect is not so clear in combination with the 9 different buffer usage hints. In most cases, not using DSA produces lower needed times, but for buffer usage hint 0, 7 and 8 using DSA is the better choice when time should be minimal. The effect of using DSA or not on the number of threads is also depending on the number of threads. When using only 1 thread, choosing to use DSA produces lower times; for 3 threads, not using DSA is better. Using more threads however usually results in lower times needed for uploading and decreases the difference of choosing one of the four methods.

The combination of buffer usage hints and methods highly depends on the method. While method 3 and 0 have similar behavior regardless the buffer usage hint, for method 1 and 2 buffer usage hint 5 and 7 produces significantly larger times.

6.2.4 Merged MARKUs

The two MARKUs, MARKU-R and MARKU-U, show several similar characteristics for uploading and rendering.

Clock Rates Increasing the size of a dataset increases the time needed for both, uploading and rendering. Increasing the clock rate of GPU, graphics memory and CPU, usually decreases the time needed for rendering and uploading. The highest CPU clock rates are an exception to that for default values; higher values can decrease performance.

DSA and PBOs Using DSA or not highly depends on other variables. For both cases, using or not using PBOs has no modeled effect on the outcome.

Opportunities for Optimization

The two MARKUs show significant differences that allow to optimize depending on the scenario.

Number of Partitions Increasing the number of partitions allows to better predict the time needed for rendering while also decreasing that time; the contrary effect is visible for uploading. Here, using more partitions increases the time needed. We assume this effect is caused by two reasons:

- More partitions need more allocations and more uploads. Each additional upload possibly introduces an additional overhead which results in longer uploading times.
- Buffer allocations and preparation possibly require GPU processing. When the GPU is busy rendering, uploading needs to wait for rendering to be finished before it can perform its task. This naturally increases the overall measured time needed for uploading (but not necessarily increases the time for uploading on its own).

Uploading Methods Another difference is visible for the four different uploading methods used. While method 0 and 3 model rendering with lower times, these methods increase the time needed for uploading. In a scenario where rendering needs to be done as quick as possible, these methods might be the best way to go. Method 1 and 2 outperform the other when uploading needs to be done fast, but increase the time needed for concurrently rendering.

Number of Threads Additionally, for uploading, using more threads results in less time needed, for rendering, more threads increase the time needed.

Buffer Usage Hints One control variables is modeled to have no effect on rendering but on uploading. Buffer usage hints are without effect on rendering, but can have different times for uploading.

Graphics Cards In all cases, the two different graphics cards show mostly similar characteristics but with some different offsets. This means that slopes or curves usually look similar for most modeled explaining variables but might have different offsets.

6.3 Chapter Summary

In this chapter we analyze the data gained from performing the configured set of experiments. We see that the two *only* cases can serve as best case scenarios, while using the *concurrently* cases can serve as the worst case. Additionally, clock rates set are not necessarily the same being used, assuming queried clock rates mirror actually used ones. For the

two MARKUs, we see there are similarities between both models that help to reduce overhead and performance loss for both processes. Additionally, there are differences for the two MARKUs that allow to optimize for a given use case. For example, using method 0 or 3 allow to optimize for lower rendering times but increases the time needed for uploading. Using the remaining methods 1 and 2 allows to do the opposite. We also identified in both cases control variables that statistically do not influence the outcome and consequently, can be ignored in the design stage of an visualization application. For all control variables we also gained insight in how the influence rendering and uploading times. This can be mathematical positive effects, meaning increasing the time, as well as reducing effects. In all cases we need to pay attention to possible interaction between several control variables in order to get full knowledge of the modeled effects. With all this we answer **SRQ5** *What and how strong are the identified influences of the two MARKUs for the two processes rendering and uploading?*.

Chapter 7

Evaluating the MARKUs

We use the Bayesian Information Criterion (BIC) with the iterative algorithm to determine which explaining variables provide a good fit for the performed measurements. However, this will only find a good solution given all its restrictions, including what control variables are used, how they are used and in what combination. Naturally, the next step is to evaluate how the two MARKUs predict performance of either of the analyzed processes.

The full picture on that task could be drawn by measuring all possible configurations and comparing them to their predictions. Yet, due to the sheer number of possibilities, this is not feasible. Additionally, since "all models are wrong" [Box76] the goal of our MARKUs is not to correctly estimate all possible configurations. Our goal is to give an overview on the parameters involved and how they affect the outcome. This can give a good starting point for finding a good configuration for a given scenario.

Hence, in this chapter we focus on giving an overview on the performance of the MARKUs by performing the following three steps:

1. Comparing predicted performance with measured performance for the MARKUs generating experiments. This allows to understand how well the MARKUs fits the experiments. In the following this set of configurations is denoted as Model Generating Configurations (**MGC**).

2. Performing additional random experiments (different to those already performed) and comparing predicted performance with measured performance. This gives us an overview on how well the MARKUs describe the underlying processes and not only how well the MARKUs fit the data. This step can detect if MARKU is more likely to only describe the data used for deriving the MARKUs and not the underlying processes. In the following this set of configurations is denoted as Random Configurations (**RC**)

3. Performing additional experiments for configurations that are predicted to have the best performance for the described scenarios and compare predicted performance with measured performance. This is closely related to the overarching objective of the thesis: Finding a good configuration for a given scenario. In the following the

set of best predicted performance configurations of configurations is denoted as Best Predicted Configurations (**BPC**).

7.1 Evaluation Approach

Comparing single measurements with single predictions would give the most complete overview, yet can easily overwhelm when the number of predictions and measurements increases. Therefore, we describe model fitness by three performance indicators:

1. How big is the confidence interval r_p compared to the time t_p predicted? – $p_1 = \frac{1}{n}\sum_{i=1}^{n}\frac{r_{p,i}}{t_{p,i}}$, smaller is better. This describes the precision and variability of the prediction.

2. How many measurements n_i lie within the confidence interval compared to the total number n_t of measurements? – $p_2 = \frac{n_i}{n_t}$, higher is better. This describes the accuracy of the predictions.

3. How far off are predicted times t_p from measured times t_m on average, also known as Mean Absolute Percentage Error (MAPE)? – $p_3 = \frac{1}{n}\sum_{i=1}^{n}\frac{t_{p,i}-t_{m,i}}{t_{m,i}}$.

Performance indicator p_1 gives an overview on how precisely the MARKUs predicts the time needed for uploading or rendering. Predictions are calculated using the R function `predict.lm` including prediction intervals. This function is part of the `stats` package which itself is part of the core of R. More information about this package can be found in [R C19].

Performance indicator p_2 hints on how well the underlying processes are modeled. It is highly dependent on the width of the estimated confidence intervals. Please note that these confidence intervals are based on the tested configurations and not necessarily provide a realistic estimation for the range, where a measurement might be. This indicator is intended to help compare the different tested sets of configurations but not to give an absolute estimate on how well the MARKUs perform.

Performance indicator p_3 describes the absolute deviation of the prediction from measurement and is calculated using the `MAPE` function which is part of the R `DescTools` package [ema20].

Please note that we describe the performance indicators for logarithmic transformed measurements that are summarized by the mean as described in Chapter 6. This means that actual times need to be back transformed and will result in different performance indicators. However, as the MARKUs are fitted to that transformation, the evaluation of the models is best described by using the transformed data without back transforming.

7.2 MGC Evaluation

In this section we compare the measured timings of **MGC** with the predicted performance of the MARKUs. We start by looking at the MARKU-R, continue with the MARKU-U and conclude with a summary. This allows us to see how well the MARKUs fit the experimental data.

7.2.1 MGC MARKU-R

The experimental data consists of 800 configurations. The three performance indicators rounded to three decimals are as follows:

$$p_1 = 0.026 \qquad p_2 = \frac{396}{800} = 0.495 \qquad p_3 = 0.024$$

This means that the confidence interval is 2.6% on average of the predicted time needed. For example, if a value of 5 is predicted, the model predicts summarized measurements to be in the interval of 5 ± 0.0625 (note for this and the following that no units are given to avoid confusion as 5 here is the expected average of log transformed times).

Furthermore, about 50.5% fall within the confidence intervals. On average, predictions are off by about 2.4%. Performance indicators p_1 and p_3 provide a context for the low accuracy of 50%. As the confidence intervals are very narrow around the predicted times, there is not much room for error. In total however, most predictions are very close to measurements.

7.2.2 MGC MARKU-U

As *uploading concurrently* is the same experiment as *rendering concurrently* but a different measured process, the number of experimental configurations is the same. The three performance indicators rounded to three decimals are as follows:

$$p_1 = 0.161 \qquad p_2 = \frac{533}{800} = 0.666 \qquad p_3 = 0.071$$

In this case, the confidence intervals are larger than for rendering. They span about 16.1% around the predicted time. The accuracy of the MARKU-U is higher than the accuracy of the MARKU-R; about 66.6% of all measured configurations are within the confidence intervals. On average, predictions are within 7.1% of measured times. The bigger confidence intervals explain the higher accuracy of the MARKU-U.

7.2.3 MGC Evaluation Summary

In both cases the MARKUs produce narrow confidence intervals and a close prediction to the measurements. The MARKU-R has a smaller error than the MARKU-U although the latter has a higher accuracy. This is due to the bigger confidence intervals for the MARKU-U.

7.3 RC Evaluation

In this section we evaluate the two MARKUs using the **RC**. We start with evaluating MARKU-R, continue with MARKU-U and conclude with a summary.

One common challenge for deducing models from data is overfitting, which means the model fits closer the data than the underlying system. To test for overfitting, we use randomly picked configurations to see how well the MARKUs can predict them. These configurations are not part of the first subset of configurations used for creating the MARKUs and also not part of the configurations used for the next section. Using randomly picked configurations allows us to get a broader view on the whole range of configurations.

For finding out if the MARKUs are overfitted to the experimental data, at least one configuration outside the experimental data needs to be used. Using more configurations than the experimental data would overreach its usefulness of finding out if the MARKUs are overfitted and giving an overview on the whole range. Therefore, we meet in the middle of the two, using 1 configuration and using 800, and pick 400 random configurations to evaluate the MARKUs. This set of configurations (the **RC**) is used to perform experiments the same way the first experiments are performed.

The variable settings are also adjusted to see how the MARKUs behave between the tested settings. For this we use roughly 32 MB and 96 MB as dataset size, 1, 8, 18, 36 for numbers of partitions and 1.5 and 2.5 GHz as CPU clock rates. Graphics memory clock rates are not altered as only 4 possible choices are available. The lowest graphics memory clock rate is excluded as it restricts the possible GPU clock rates to only very low values.

For GPU clock rates we use 0.823 and 1.291 GHz for the TITAN and 0.750 and 1.650 GHz for the RTX card.

7.3.1 RC MARKU-R

The three performance indicators for **RC** for MARKU-R are rounded to three decimals as follows:

$$p_1 = 0.027 \qquad p_2 = \frac{45}{400} = 0.113 \qquad p_3 = 0.047$$

The confidence intervals span about 2.7% around the predicted values which is similar to the **MGC** data. This also reflects on the not previously used set values for the quantitative variables CPU clock, number of partitions and size. We used for these three variables values different to the **MGC**, but within the extrema of the already set values. This means that the MARKU-R is able to also map these values with a high precision.

Accuracy is only at 11.3%, but the mean absolute error is still low at only 4.7%. This means, although most measurements are outside the predicted confidence intervals, the MARKU-R is able to predict them with only a small deviation.

7.3.2 RC MARKU-U

For *uploading concurrently* using **RC**, we get the following performance indicators:

$$p_1 = 0.151 \qquad p_2 = \frac{235}{400} = 0.588 \qquad p_3 = 0.079$$

p_1 is smaller than for the **MGC** data. In consequence, also accuracy is lower at 58.8% as indicated by p_2. The overall error p_3 is similar to the error of the **MGC** data.

7.3.3 RC Evaluation Summary

We can see the MARKUs perform in a similar fashion as before. The MARKU-R has a lower accuracy and a higher overall error, but at a very low level of under 5%. For the MARKU-U, the overall error is also small at under 8% and with that very similar to the **MGC**. We see no decisive evidence for overfitting.

7.4 BPC Evaluation

One goal for modeling performance of concurrently data uploading and rendering is to find an optimal configuration for a given scenario. The scenarios can have different requirements on what task needs to be prioritized and what to optimize. Therefore, we analyze in this section how to find possible candidates of configurations for achieving the targeted prioritization and afterwards evaluate their performance.

The following is structured as follows:

- Description of how the configurations are generated
- Evaluation of prediction performance for
  - OT1: High priority data rendering
  - OT2: High priority data uploading
  - OT3: Balancing both activities.

7.4.1 Configuration Generation

Exemplary for the whole priority continuum introduced in Chapter 2, we use the following priorities for evaluation:

Movie Visualization	Priority lies in minimizing the aggregated time for rendering and uploading together.
Desktop Visualization	Priority lies in minimizing either rendering or uploading time.

VR Visualization Priority lies in minimizing time for rendering. Uploading time minimization has lower priority.

Based on these priorities we want to find the best configurations. This means, using the two MARKUs, we predict for all possible configurations the time needed to perform uploading or rendering a dataset and search for the minimal times. Therefore, we use the optimization targets **OT1**, **OT2** and **OT3** derived in Chapter 2 and predict the performance for either MARKU-R, MARKU-U or both:

OT1 High priority data rendering – MARKU-R

OT2 High priority data uploading – MARKU-U

OT3 Balancing both activities – Both MARKUs.

For **OT1** we search for configurations that are predicted to produce the shortest rendering times using MARKU-R, for **OT2** the shortest uploading times using MARKU-U and for **OT3** to shortest time for both, rendering and uploading, together using both MARKUs. The procedure to search for configurations is as follows:

1. Define which levels of the control variables to include
2. Limit the number of configurations
3. Reduce redundancies

Included Control Variable Levels

For real use cases, usually the dataset is fixed to a certain point or cannot be controlled. As the idea is to also evaluate the MARKUs with different parameters than the trained, we use about 32MB (exact calculation of size is given in Chapter 5.1) as dataset size. This size stands exemplary for a time dependent dataset where each time step has 32MB of data which is right between 16MB and 64MB, the lower two sizes used for creating the MARKUs. We also vary the possible set of numbers of partitions to 1, 8 ,18 and 36. Additionally, as the goal for this task is to find the fastest configurations, we restrict the CPU, GPU and graphics memory clock to their highest values, given in Section 5.1.

Number of Configurations

As we cannot be certain that the MARKUs predict the needed time accurately, we need to include a range of variety of configurations close to the best predicted configuration. This allows to find the best configuration within the given uncertainty of the MARKUs. Furthermore, we could have secondary optimization targets with different priorities as described in Chapter 2. This means that including worse performing configuration than the configuration with the best timings for the highest priority target, can overall be better

when a second priority target can be optimized as well; while the timings for the highest priority target are close to the optimum.

The next step is to decide how many configurations to include. On the one hand, using more configurations for this task can allow to also optimize a second priority goal or to find the best solution for the actual system. Yet, using too many configurations includes worse and worse predictions. This can mean that configurations are measured that are likely to miss the first priority goal and therefore unnecessary to even consider.

This requires to tune the number of configurations based on the task at hand. For this work, we include all predictions within 2σ (standard errors of the residuals, see also the package `stats` which is part of the core of R [R C19]) of the MARKUs. With this we account for the variability of the MARKUs and their residuals, in the hope of catching all "best" configurations. We do this for each card individually as they differ in peak performance. For the following we use the RTX card to extract configurations exemplary.

Reduced Redundancies

For both MARKUs, there are control variables that are modeled to not affect performance. For MARKU-U this is using or not using PBOs; for MARKU-R using or not using PBOs as well as changing the buffer usage hint are modeled to have no effect. For prediction of performance we include these two control variables to not exclude not modeled behavior and consequently hide possible errors of the models. However, this increases the number of configurations to be tested without having different predicted performance and leads to large numbers of not necessary experiments.

Therefore, for all predicted configurations within 2σ of the best predicted time that only vary in these control variables (that are modeled to have no effect), we randomly sample one of them and remove the rest of the set of configurations.

Exemplary for **OT1** using the all configurations within 2σ of the best predicted time results in 1728 configurations for the RTX card. Out of those 1728 configurations only 96 differ in predicted times as buffer usage hint and using or not using PBOs are not modeled to have an effect on MARKU-R.

Resulting Configurations for Both MARKUs

After applying this procedure, the resulting configurations for the three optimization targets are for **OT1** 192 configurations, for **OT2** 1678 configurations and for **OT3** 1219 configurations.

Performance Indicators for MARKU-R

Table 7.1 shows the three performance indicators rounded to three decimals as columns for MARKU-R for the best predicted configurations for each optimization target as an individual row. We can see that for all configurations, that the confidence intervals are slightly higher than before at around 3.5%. The accuracy is at a very low level and between

4.2% and 9.3%, but, the overall error, although slightly increased is still low at around 6%.

Optimization Target	p_1	p_2	p_3
OT1	0.036	$\frac{8}{192} = 0.042$	0.061
OT2	0.035	$\frac{94}{1678} = 0.042$	0.060
OT3	0.036	$\frac{113}{1219} = 0.093$	0.059

Table 7.1: Performance indicators for MARKU-R of the BPC configurations

Performance Indicators for MARKU-U

Table 7.2 shows the three performance indicators rounded to three decimals as columns for MARKU-U for the best predicted configurations for each optimization target as an individual row. In all cases, the confidence intervals are increase to 20.9% to 23.4% and the accuracy is similar to the previous configurations of MARKU-U and is between 62.3% and 68.3%. The overall error is also slightly increased to a range of 9.3% to 11%.

Optimization Target	p_1	p_2	p_3
OT1	0.209	$\frac{124}{192} = 0.646$	0.096
OT2	0.212	$\frac{1146}{1678} = 0.683$	0.093
OT3	0.234	$\frac{759}{1219} = 0.623$	0.110

Table 7.2: Performance indicators for MARKU-U of the BPC configurations

7.4.2 BPC Evaluation Summary

For the tested **BPC**, we see similar but worse performance than before. One aspect that can cause this are the ”extreme” targets and therefore extreme values for the observed variables. However, the models behave very similar for all three optimizations targets and have a low overall error between 5 and 11 %.

7.5 Discussion

Overall we see that the errors are below 11% on average. The confidence intervals for the MARKU-R are very narrow and cause a low accuracy for predicted times. For the MARKU-U, the confidence intervals are much larger and consequently, more measurements are within the intervals. However, the average error for the MARKU-U is also higher than for the MARKU-R.

Logarithmic Transformation An important aspect is that we use logarithmic transformed observations. This needs to be taken into account when trying to understand errors and ranges. For example, if we take a closer look at the confidence intervals and transform them back with the exponential function, we see how big they actually can be. An example measured time is a mean measured time from the uploading optimization target for uploading of 43.93 ms. Please note that this and the following numbers are round to 2 decimals. 43.93 ms transformed using the natural logarithm is 3.78. For this value, the MARKU-U predicted a value of 3.52 or non logarithmic of 33.80 ms, with a prediction interval from 3.17 to 3.87, or non logarithmic from 23.88 to 47.85 ms. We see that the prediction intervals are asymmetric, caused by non linear transformation, and range with roughly 24 ms, which can make it difficult for fine tuning depending on rendering load.

Reduced Search Space However, the different settings can also have a huge effect on rendering or uploading times. For example, while using the same dataset size, CPU, GPU and graphics memory clock rate, the mean time needed for uploading is for one configuration 6.84 ms, for another 517.49 ms. This is also reflected in the prediction intervals for both configurations. The first ranges from 4.66 to 7.54 ms, the second from 255.67 to 579.28 ms after back transformation. Consequently, our approach can help to find configurations that have a high likelihood of being within the best performing for a given optimization target, without testing all possible combinations of control variables.

Scope While our methodical approach is applied to a great variety of parameters, the scope of this work does not allow to test all possible choices. For this work, we focus on the OpenGL programming interface and on a Linux based operating system. Other APIs and other operating system might have completely different behavior and require a remodeling step. This is also true for different hardware setups. Attention needs also be paid to the validity of the model, especially for quantitative variables, which can produce an unlimited amount of permutations. This means the derived MARKUs are only valid within the boundaries of the settings. For example using significantly smaller sizes than 16 MB or larger than 128 MB would possibly require re-performing the model generating process in order to get a reliable model. Yet, we tested two distinct sizes between 16 MB and 128 MB not used before for deducing the MARKUs and see that these sizes are predicted similarly to the original sizes.

Missing Details Furthermore, not all details of the processes are modeled. As we summarize and filter out outliers the measurements we lose smaller details. Especially outliers can have a highly distracting impact on VR visualizations: If some frames are significantly delayed, data exploration can stop as users might feel unsafe. Furthermore, a preliminary analysis of the data shows that there might be effects depending on the iteration of the measurement: We perform for each configuration at least 30 measurements or iterations; some of them gain performance while others lose performance to the last measurements of these 30.

7.6 Chapter Summary

In this chapter we discuss how to evaluate the two MARKUs. We analyze them using three performance indicators that describe how predictions compare to measured performance and therefore a method that allows to evaluate the MARKUs. We further describe three classes of experimental data that help to give an overview on how the MARKUs perform in various situations. The third class leans at an real-world application of the presented methodology of this thesis: How to optimize for a given use case? We see that while accuracy is for most cases far from optimal, overall errors for predictions are at a low level. This mostly is caused by the confidence intervals having only narrow widths.

With a methodology for evaluation and the third set of evaluation configurations, we answer **SRQ6** *How to evaluate the two MARKUs and how to find a more optimal solution for the outlined optimization targets?*.

Chapter 8

Application to Real-World Use Case

In this chapter we describe the application of the two MARKUs on a real-world dataset. We start by giving a brief description on the dataset itself, which is followed by a comparison of predicted times for both uploading and rendering. We conclude this chapter with a discussion of the applicability of the MARKUs for prediction.

8.1 Dataset Description

The dataset used in this chapter is from the domain of geophysics. It is the result of a simulation using the parallel finite element code TERRA [BB95, BRB96, BR96, BRB97] and carried out on the supercomputer SuperMUC[1] of the Leibniz Supercomputing Centre (LRZ) of the Bavarian Academy of Sciences and Humanities. The simulation describes a simulation of Earth's Mantle convection over 200 million years. For further details on the model setup refer to [NCG+16]. Every 50.000 years the current status is dumped and stored to disk resulting in 4000 timesteps. Out of these 4000 timesteps, we use only the first 20 timesteps for the evaluation as this suffices for performance comparison.

These data dumps are preprocessed similarly as described in [WAB+15]. The details of the resulting dataset are:

Structure	20 timesteps with two triangle meshes each
Size	About 55 MB per timestep
Partitioning	6 buffers/partitions per timestep

The 6 partitions are structured as 2 buffers for positions of the vertices, 2 buffers for normal vectors of the surfaces and 2 buffers for indices. Normal vectors describe the surface of the triangles and are used for shading effects. Indices are used to save storage by reusing positions of vertices that are shared for two triangles.

[1] `https://doku.lrz.de/display/PUBLIC/Decommissioned+SuperMUC`

Differences to Evaluation Datasets

Please note that this differs from the datasets used for performing the experiments and for the evaluation in three properties:

- Vertices per unit of storage size
- Fragments per triangle
- Used shader

Vertices per unit of storage size In this case there are less vertices processed (and therefore rendered) per unit of storage size as additional properties are included. However, the usage of indices allows to store more triangles using less storage. This means that while the first timestep of the dataset has only about 1.17 million vertices, a total of 2.34 million triangles are rendered. For the datasets generated for the chapters before, a size of about 55 MB results in 4.81 million vertices and 1.60 million triangles.

Fragments per triangle Additionally to a varying number of vertices and triangles, the number of produced fragments highly deviates for both datasets. While the dataset, which is generated obtaining the models, produces with two triangles a full screen quad and consequently 1920 times 1080 fragments, the triangles of the real-world data is not bound to full screen quads. These triangles only cover a fraction of the screen and therefore require less workload as less fragments are produced and require processing. The used dataset is part of a larger dataset including more details, such as a visual depiction of the core mantle boundary and Earth's surface [SVS+05]. A rendering of this larger dataset is shown in Fig. 8.1. A rendering of the dataset used for this evaluation from the used perspective is shown in Fig. 8.2. It can be seen that the surfaces do not cover the full screen. Hence, the single triangles making up the surfaces cover even less. The datasets used in the chapters before all were made of planes filling the whole screen and would result in a rectangle if rendered and printed here.

Used shader Another difference to the previously used datasets is that a different shader is used. This time shading effects are calculated as normal vectors are included in the dataset and would be used for visualization.

Therefore, a correct prediction of rendering and uploading can not be expected. However, the two MARKUs should give a starting point for finding good configurations for a given task.

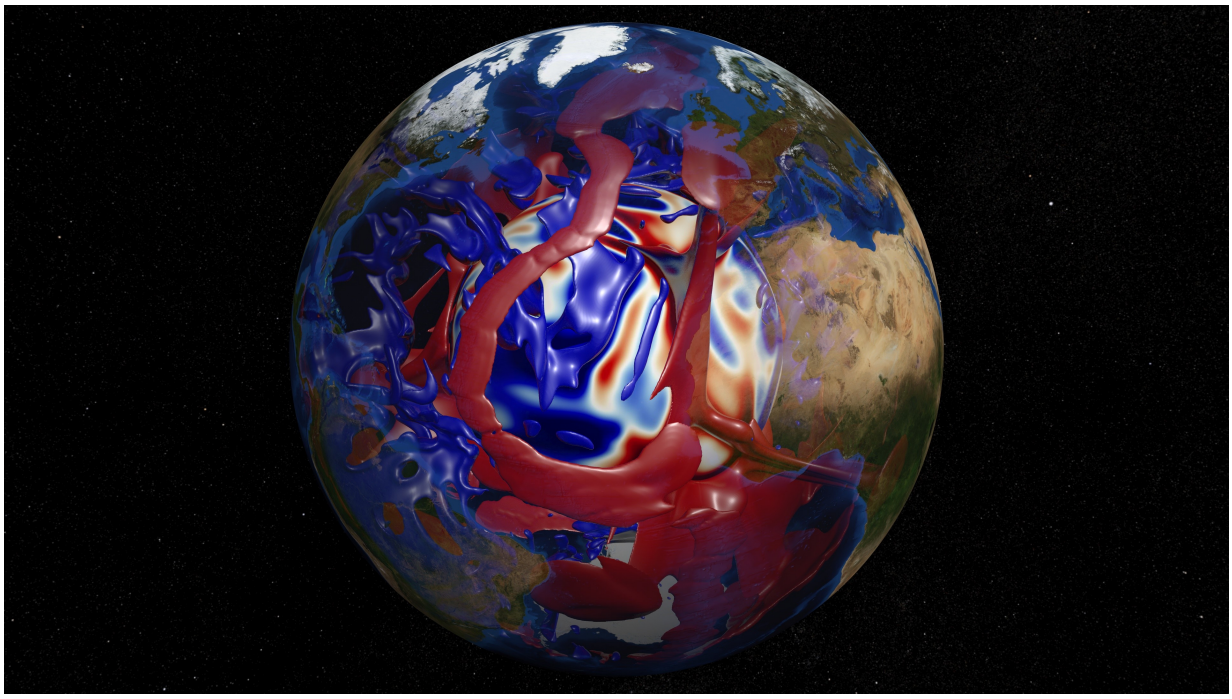


Figure 8.1: Rendering of the complete visualization dataset

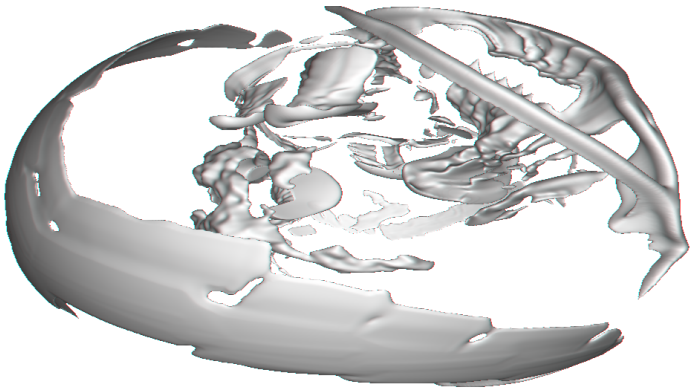


Figure 8.2: Rendering of the used example dataset

8.2 Prediction vs. Measurements

For testing the dataset we use again only the maximum settable clock rates for CPU, GPU and graphics memory. The size of the dataset and the number of partitions is fixed. All possible configurations for the remaining control variables – buffer usage hint, using DSA, using PBO, number of threads and uploading method – are generated. These configurations are tested the same way as the experiments before. This totals in 432 different configurations for each of the used graphics cards. The measurements are then processed as described before.

Below, we compare predictions of MARKU-R and MARKU-U with measurements, respectively.

8.2.1 MARKU-R

Fig. 8.3 shows a comparison of the predicted times using the MARKU-R vs. the measured time needed for the real-world dataset. The measured/predicted time is shown on the y-axis and ranges from 0 to about 65 ms.

In Fig. 8.3 all configurations are sorted ascending by the time predicted and assigned a consecutive number from 0 to 864, the configuration number. The x-axis shows this configuration number and also ranges from 0 to 864. Consequently, configuration number does not denote a certain configuration but only the position in the sorted sequence from shortest to longest predicted time for rendering.

Please note, in Fig. 8.3 the time needed and predicted are back-transformed.

We can clearly see in Fig. 8.3 that measurements and prediction highly deviate from each other. All measurements stay below the predicted time. However, although the configurations are ordered by predicted times, the best measured times (at nearly 0 ms) are roughly within the best 600 (from 864) configurations. We can also see that worse measurements, also near to the worst measured of around 35 to 65 ms, are within this range. Starting from around configuration 600, all measurements stay above 9 ms.

8.2.2 MARKU-U

Fig. 8.4 shows a comparison of predicted times using the MARKU-U vs. the measured time needed for the real-world dataset. The measured/predicted time is shown on the y-axis and ranges from 0 to about 600 ms.

As before, all configurations are sorted ascending by the time predicted and given a number of 0 to 864. The x-axis shows this number.

Again, we can see that prediction and measurement deviate from each other. The best measured times in the range from about 8 to 26 ms are within the first 50 configurations and reflect the prediction. However, the worst measured times, above about 170 ms, are within the configurations in the range from about 400 to 600.

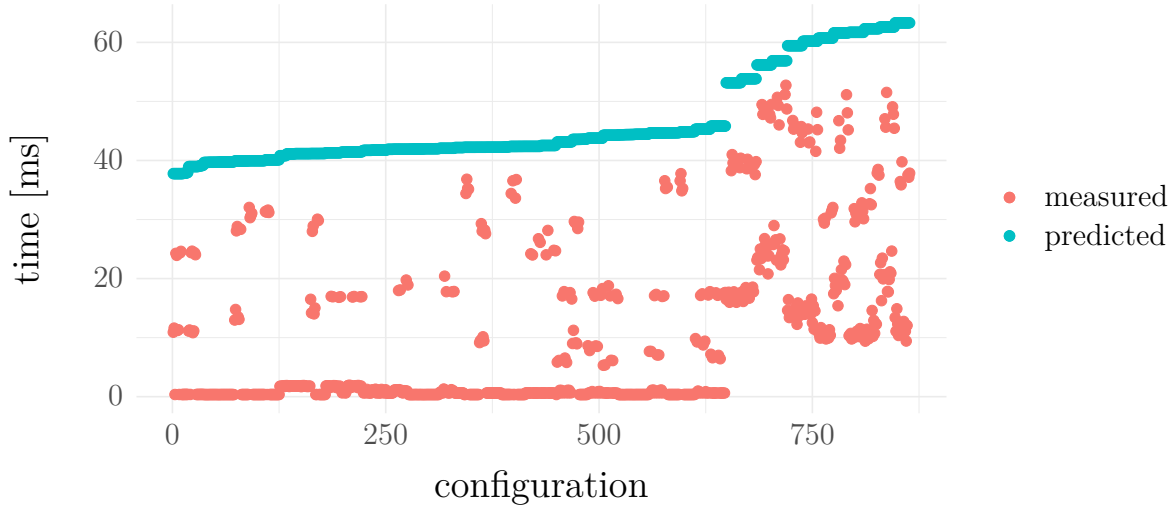


Figure 8.3: Comparison of predicted time needed vs. measured time for rendering

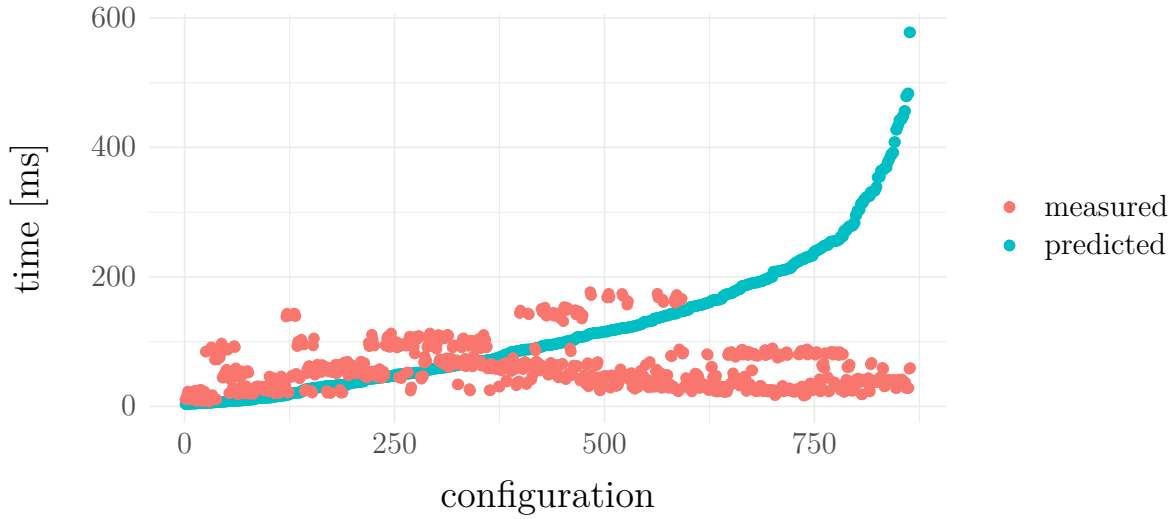


Figure 8.4: Comparison of predicted time needed vs. measured time for uploading

8.3 Discussion of MARKUs

For both MARKUs, the predictions deviate from measured times. While we can find the best configurations within the best predicted times, not all good configurations are predicted correctly.

Differences for rendering On the one hand, workload of rendering highly deviates from the generated datasets used to obtain the models. After back-transformation, best rendering times are around 0.3 ms for a dataset of about 55 MB. Compared to that, the generated dataset with 32 MB used in Chapter 7 that needed in the best cases around 30 ms shows that there is a great difference in workload. This great difference in processing time also reflects to worse configurations. This short processing time for the real-world dataset can increase the impact of effects not modeled.

Differences for uploading This also creates a strong impact for the MARKU-U. In many cases, allocation, binding or even copying of the data itself is processed sequentially with rendering. This means when rendering times increase, the wait time and also the total time needed for uploading increases. The same is true vice versa which affects prediction precision negatively.

MARKUs for optimization Nevertheless, for both models, MARKU-R and MARKU-U, using the shorter predicted times allows to find good solutions to optimize for a given use case and to avoid inefficient configurations.

Chapter 9

Conclusions and Future Work

Visualization of datasets often requires uploading data to graphics hardware while concurrently rendering. This work analyses the involved hardware and software components and possible paths data can take from host memory to the final image. Based on that we describe several possibilities to define performance in the context of visualization.

Visualization can take a broad range of forms which we outline on a priority continuum for rendering and uploading. This continuum shows VR visualization as an extreme on the one side and movie visualization as an extreme on the other side. All types of visualization in between these two extremes on the priority continuum can have a variety of priorities that need to be fulfilled in one or another way.

Based on distinct priorities for those types of visualization, we derive three optimization targets **OT1**, **OT2**, and **OT3** for the two processes of rendering and uploading. **OT1** prioritizes rendering over uploading. An example for this is VR visualizations, where consistently low rendering times are needed. **OT2** prioritizes uploading over rendering. Exemplary for this are desktop visualizations, where fixed time slots can be assigned for rendering the next image shown on a computer screen. If there is time left over in this time slot, uploading can easily be maximized by prioritizing it. **OT3** prioritizes the total time of both, rendering and uploading, together. An example visualization form of this are movie visualizations, where the total time for rendering a movie (including all the uploads necessary for rendering) should be as minimal as possible.

An implementation of a framework for rendering while concurrently uploading data, which allows to fulfill all these requirements, needs to be as flexible as possible. To achieve this, we describe how we decouple possible dependencies of data structures on the host system and on the graphics hardware as well as inter process dependencies of uploading and rendering, in order to find a solution that fits all use case scenarios. This approach introduces the number of uploading threads as an additional design choice. The number of uploading threads, the chosen system, hardware, and other software parameters, form together an enormous input space of control variables. All of them possibly influence performance of rendering and or uploading.

In order to quantify these influences, we use D-optimal design and linear regression in combination with the Bayesian Information Criterion. This allows us to perform a small

number (compared to all permutations) of experiments and derive for each, rendering and uploading, a Model for Asynchronous Rendering and K-time concurrent Uploading (MARKU), that describes form and weight of these influences. We apply this approach for a given system setup and analyse the resulting MARKUs.

Furthermore we provide an evaluation scheme to challenge performance of the MARKUs. By predicting the times needed for uploading and rendering using the two MARKUs and by comparing these times against the MARKUs generating experiments, we see how well the MARKUs fits that data. Additionally, using in-between range quantitative variables, such as sizes between the tested sizes, we compare predictions of the MARKUs against measurements of randomly chosen configurations and for best predicted times for several use case scenarios. While the prediction interval increases for the latter cases, the MARKUs are able to predict the needed times similarly compared to the MARKUs generating measurements with a low average error of under 11%.

Using the presented methodical approach allows to analyse and model data rendering and uploading for graphics hardware in a general fashion. Instead of only testing for specific configurations, it enables us to systematically search for optimization targets. From this point on, targeted testing can easily verify if the found configurations fit to a given use case.

Finally, we compare predictions of the obtained MARKUs with measurements of a real-world dataset. While measured and predicted performance deviate from each other, due to the different structure of the dataset and with this different workload, both extreme ends of predictions (worst and best) provide valuable insight into finding good configurations and avoiding inefficient ones.

Table 9.1 amends Table 3.1 adding this work and its characteristics to the last row. The columns of Table 9.1 outline several key concepts for a comparison of related work and this work. Related work is structured as individual rows.

	API	Analyze Data Transfers	Multi-threaded	Mathematical Model	Individual Process Analysis	Statistical Experimental Design
[BMK13]	CUDA	✓	–	✓	–	–
[GLGLBG12]	CUDA	✓	–	✓	✓	–
[vWMSB14]	CUDA	✓	–	✓	✓	–
[FAN+13]	Driver Adaptation	✓	–	✓	only transfers	–
[Buc04]	OpenGL	✓	–	–	–	–
[GRE09]	OpenGL	✓	–	–	–	–
[HM12]	OpenGL	✓	✓	–	–	–
[FGKR16]	OpenGL	✓	–	–	–	–
MARKUs (this work)	OpenGL	✓	✓	✓	✓	✓

Table 9.1: Comparison of key concepts of related and this work

Future Work

Rendering images for games, visualizations or movies is highly dependent on the data and the scene rendered as well as the perspective onto the scene. In this work, we focus on getting a starting point for optimization. We use the worst case scenario for rendering where the GPU needs to render everything and is kept busy the whole time. In a real scenario, some perspectives or scenes might finish a lot sooner than expected as some parts are excluded due to occlusion. These early finishes allow even more possibilities for improving performance of uploading or other tasks at hand. Future work might include models of rendering time depending on scene and perspective.

Texture images can also be an important part of the rendering process. However, they are processed differently compared to vertex data. This means that their performance can be depending on different parameters and consequently, deviate vastly from our results on vertex data. An in-depth analysis of texture uploading and rendering can shed light on this topic and could allow to describe their performance in a mathematical model as well.

The presented methodical approach relies on a lot of hand tuning and step by step work in order to get high performing configurations. Future work would build upon our work and implement fully automated tuning based on defined performance requirements for a given use case scenario. On-the-fly evaluation while rendering could fine tune and balance uploading and rendering, depending on remaining frame time or other criteria.

Our two MARKUs and this thesis will help future visualizations to make better use of hardware capabilities and improve scientific tools. Hence, they will help to understand a little bit more of the "*mysteries of eternity, of life, of the marvelous structure of reality*" [Albert Einstein, Life magazine, May 2, 1955].

List of Figures

List of Tables

Bibliography

[AHJ+01] R.S. Allison, L.R. Harris, M. Jenkin, U. Jasiobedzka, and J.E. Zacher. Tolerance of temporal delay in virtual environments. In *Proceedings IEEE Virtual Reality 2001*, pages 247–254. IEEE Comput. Soc, 2001.

[APLK17] Rachel Albert, Anjul Patney, David Luebke, and Joohwan Kim. Latency requirements for foveated rendering in virtual reality. *ACM Transactions on Applied Perception*, 14(4):1–13, sep 2017.

[BA04] Kenneth P Burnham and David R Anderson. *Model Selection and Multimodel Inference*. Springer New York, 2 edition, 2004.

[Bar93] C. Barillot. Surface and volume rendering techniques to display 3-d data. *IEEE Engineering in Medicine and Biology Magazine*, 12(1):111–119, mar 1993.

[BB95] Hans-Peter Bunge and John R. Baumgardner. Mantle convection modeling on parallel virtual machines. *Computers in Physics*, 9(2):207, 1995.

[BB17] Patrick Breheny and Woodrow Burchett. Visualization of regression models using visreg. *The R Journal*, 9(2):56, 2017.

[BMK13] Michael Boyer, Jiayuan Meng, and Kalyan Kumaran. Improving GPU performance prediction with data transfer modeling. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1097–1106. IEEE, IEEE, may 2013.

[Box76] George E. P. Box. Science and statistics. *Journal of the American Statistical Association*, 71(356):791–799, dec 1976.

[BR96] Hans-Peter Bunge and Mark A. Richards. The origin of large scale structure in mantle convection: Effects of plate motions and viscosity stratification. *Geophysical Research Letters*, 23(21):2987–2990, oct 1996.

[BRB96] Hans-Peter Bunge, Mark A. Richards, and John R. Baumgardner. Effect of depth-dependent viscosity on the planform of mantle convection. *Nature*, 379(6564):436–438, feb 1996.

[BRB97] Hans-Peter Bunge, Mark A. Richards, and John R. Baumgardner. A sensitivity study of three-dimensional spherical mantle convection at 108rayleigh number: Effects of depth-dependent viscosity, heating mode, and an endothermic phase change. *Journal of Geophysical Research: Solid Earth*, 102(B6):11991–12007, jun 1997.

[Buc04] Ian Buck. Gpubench: Evaluating gpu performance for numerical and scientific application. In *Proceedings 1st ACM Workshop General-Purpose Computing on Graphics Processors (GPˆ 2'04)*, 2004.

[CHN+14] FENG Changyong, WANG Hongyue, LU Naiji, CHEN Tian, HE Hua, LU Ying, et al. Log-transformation and its implications for data analysis. *Shanghai archives of psychiatry*, 26(2):105, 2014.

[CMG19] Polona Caserman, Michelle Martinussen, and Stefan Göbel. Effects of end-to-end latency on user experience and performance in immersive virtual reality applications. In Erik van der Spek, Stefan Göbel, Ellen Yi-Luen Do, Esteban Clua, and Jannicke Baalsrud Hauge, editors, *Entertainment Computing and Serious Games*, pages 57–69, Cham, 2019. Springer International Publishing.

[ema20] Andri Signorell et mult. al. *DescTools: Tools for Descriptive Statistics*, 2020. R package version 0.99.36.

[FAN+13] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Edahiro. Data transfer matters for GPU computing. In *2013 International Conference on Parallel and Distributed Systems*, pages 275–282. IEEE, IEEE, dec 2013.

[Fed72] Valerii Vadimovich Fedorov. *Theory of optimal experiments*. Academic Press, New York, 1972.

[FGKR16] Martin Falk, Sebastian Grottel, Michael Krone, and Guido Reina. Interactive GPU-based visualization of large dynamic particle data. *Synthesis Lectures on Visualization*, 4(3):1–121, oct 2016.

[Fly72] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[GLGLBG12] Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. Performance models for asynchronous data transfers on consumer graphics processing units. *Journal of Parallel and Distributed Computing*, 72(9):1117–1126, sep 2012.

[GRE09] S. Grottel, G. Reina, and T. Ertl. Optimized data transfer for time-dependent, GPU-based glyphs. In *2009 IEEE Pacific Visualization Symposium*, pages 65–72. IEEE, IEEE, apr 2009.

[Guh18] Sumanta Guha. *Computer Graphics Through OpenGL*. Chapman and Hall/CRC, dec 2018.

[Han05] Charles Hansen. *The visualization handbook*. Elsevier Butterworth-Heinemann, Burlington, MA, 2005.

[HM12] Ladislav Hrabcak and Arnaud Masserann. Asynchronous buffer transfers. In Patrick Cozzi and Christophe Riccio, editors, *OpenGL Insights*, pages 391–414. A K Peters/CRC Press, jul 2012.

[Jer10] Jason J Jerald. *Scene-motion-and latency-perception thresholds for head-mounted displays*. PhD thesis, University of North Carolina at Chapel Hill, 2010.

[KAB13] Shinpei Kato, Jason Aumiller, and Scott Brandt. Zero-copy i/o processing for low-latency GPU computing. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems - ICCPS '13*, pages 170–178. ACM Press, 2013.

[Kee95] Oliver N. Keene. The log transformation is special. *Statistics in Medicine*, 14(8):811–819, apr 1995.

[KMMB12] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class gpu resource management in the operating system. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 37, USA, 2012. USENIX Association.

[LH14] Zhicheng Liu and Jeffrey Heer. The effects of interactive latency on exploratory visual analysis. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2122–2131, dec 2014.

[MM01] Willard G Manning and John Mullahy. Estimating log models: to transform or not to transform? *Journal of Health Economics*, 20(4):461–494, jul 2001.

[MMK+11] Jiayuan Meng, Vitali A. Morozov, Kalyan Kumaran, Venkatram Vishwanath, and Thomas D. Uram. GROPHECY. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*, pages 1–11. ACM Press, 2011.

[MN94] Alan J. Miller and Nam-Ky Nguyen. Algorithm AS 295: A fedorov exchange algorithm for d-optimal design. *Applied Statistics*, 43(4):669, 1994.

[NCG+16] Rainer Nerlich, Lorenzo Colli, Siavash Ghelichkhan, Bernhard Schuberth, and Hans-Peter Bunge. Constraining central neo-tethys ocean reconstructions with mantle convection models. *Geophysical Research Letters*, 43(18):9595–9603, sep 2016.

[R C19] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2019.

[Sel13] Graham Sellers. *OpenGL superbible : comprehensive tutorial and reference*. Addison-Wesley, Upper Saddle River, NJ, 2013.

[SKS16] Dave Shreiner, John M. Kessenich, and Graham M. Sellers. *OpenGL Programming Guide*. Pearson Education (US), 9 edition, 2016.

[SNL20] Jan-Philipp Stauffert, Florian Niebling, and Marc Erich Latoschik. Latency and cybersickness: Impact, causes, and measures. a review. *Frontiers in Virtual Reality*, 1:31, nov 2020.

[SVS+05] Reto Stöckli, Eric Vermote, Nazmi Saleous, Robert Simmon, and David Herring. The blue marble next generation-a true color earth dataset including seasonal dynamics from modis. *Published by the NASA Earth Observatory*, 2005.

[Tuk77] John Tukey. *Exploratory data analysis*, volume 2. Addison-Wesley Pub. Co, Reading, Mass, 1977.

[Ven10] Shalini Venkataraman. Nvidia quadro dual copy engines, 2010.

[Vla15] Alex Vlachos. Advanced vr rendering. Online, 2015.

[vWMSB14] B. van Werkhoven, J. Maassen, F.J. Seinstra, and H.E. Bal. Performance models for CPU-GPU data transfers. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 11 – 20. IEEE, IEEE, may 2014.

[WAB+15] Markus Wiedemann, Christoph Anthes, Hans-Peter Bunge, Bernhard S.A. Schuberth, and Dieter Kranzlmulle. Transforming geodata for immersive visualisation. In *2015 IEEE 11th International Conference on e-Science*, pages 249–254. IEEE, aug 2015.

[War08] Colin Ware. *Visual Thinking*. Elsevier LTD, Oxford, 2008.

[WBK+20] Michael L. Wilson, Sarah C. Beadle, Amelia J. Kinsella, Ryan S. Mattfeld, Adam W. Hoover, and Eric R. Muth. Task performance in a head-mounted display: The impacts of varying latency. *Displays*, 61:101930, jan 2020.

Acknowledgments

I am deeply grateful for all the support and feedback I got while preparing and writing this thesis. First and foremost, I would like to express my gratitude to Prof. Dr. Dieter Kranzlmüller for providing me with the opportunity to write this thesis and his support in every stage of this endeavor. I would also like to offer my special thanks to Prof. Dr. Göran Kauermann for his help and advice for the necessary statistical concepts. I would like to extend my thanks to Prof. Dr. Christoph Anthes, who tutored me during the first part of this work and got me started for the research done. I am sincerely grateful to Prof. Dr. Hans-Peter Bunge and Dr. Bernhard S.A. Schuberth for providing me with the application for my research and their constant support and motivation. I would like to thank all the colleagues at the Leibniz Supercomputing Centre (LRZ) and especially the Virtual Reality and Visualization Group that accompanied me for the most part of this academic adventure. I am also very grateful to have worked alongside my doctoral brothers and sisters and all other colleagues at Ludwig-Maximilians-Universität München, who endured me during the final months writing. A special thanks to Matthias Maiterth, Maximilian Höb and Dr. Nils Otto vor dem Gentschen Felde for the countless discussions that challenged me and my research, that helped me sort my thoughts, and that helped me to consistently improve me and my work. I would like to thank my family and friends for their consistent support and for accompanying me in my life. I would like to specifically thank Ross Jones for his time and effort proofreading the final version of this thesis. Finally, I am deeply grateful for my partner in life, Eva-Maria Gelfert, who constantly challenges, criticizes, motivates, and supports me, sometimes even all at the same time, and who always encourages me to strive for my best.