Institute for Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis Nr. 43

Performance Quantification of Volume Visualization

Valentin Bruder

Course of Study:	Informatik
Examiner:	Prof. Dr. Thomas Ertl
Supervisor:	Dr. rer. nat. Steffen Frey
Commenced:	August 15, 2015
Completed:	January 12, 2016
CR-Classification:	I.3.8

Abstract

This thesis presents machine learning models to predict the performance of volume visualization applications. The work focuses on two aspects of performance prediction related to volume visualization: the prediction of the execution time of an upcoming frame during runtime of an interactive volume visualization application and the prediction of the average performance of rendering volume data sets on arbitrary graphics cards.

For dynamic frame time prediction, a volume ray caster with acceleration techniques is implemented, which allows user interactions. Data from the corresponding acceleration algorithms is used for the creation of a linear regression machine learning model, among other features. This model enables a real-time prediction of execution times needed for upcoming frames with a coefficient of determination between 0.67 and 0.96 for tested data sets.

Predicting the average execution times on different GPUs is approached from two different directions. In the first one, a machine learning model is employed that allows the prediction of average execution times of an unevaluated data set. In addition to attributes describing the GPUs on which the performance is to be predicted, it only uses one feature specifying the volume, namely its file size. For tested data sets with high resolutions, predictions with a coefficient of determination between 0.56 and 0.83 could be made. In the second approach, a linear regression model is used which can predict the average execution time of a volume data set on an unevaluated system. Thereby, relative prediction errors between 5.33% and 22.22% on average could be achieved on different evaluated NVIDIA GPUs.

Kurzfassung

In dieser Arbeit werden verschiedene Modelle aus dem Bereich des machinellen Lernens vorgestellt, mit denen die Leistungsfähigkeit von Anwendungen zur Volumenvisualisierung vorhergesagt werden kann. Der Fokus der Arbeit liegt dabei auf zwei verscheidenen Aspekten der Leistungsfähigkeitsvorhersage bei dreidimensionaler Visualisierung von Volumina: der Vorhersage der Zeit, die benötigt wird um das nächste Bild zur Laufzeit einer interaktiven Volumenvisualisierungsanwendung zu berechen, sowie derer, die im Durchschnitt benötigt wird, um Volumina auf beliebigen Grafikprozessoren zu berechnen.

Für die dynamische Bildvorhersage findet ein Volumen-Raycaster mit Beschleunigungstechniken Verwendung, welcher auch Benutzerinteraktion ermöglicht. Daten aus den Algorithen für die Beschleunigungstechniken werden zusammen mit anderen als Features für die Generierung eines linearen Regressionsmodells zum maschinellen Lernen verwendet. Das entwickelte Modell ermöglicht eine Echtzeitvorhersage der Zeiten, die für die Berechnung der nächten Bilder benötigt werden. Der Determinationskoeffizient für die untersuchten Datensätze liegt dabei zwischen 0.67 und 0.96.

Zwei unterschiedliche Herangehensweisen werden verwendet, um die durchschnittlichen Berechnungszeiten auf verschiedenen GPUs vorherzusagen. In der ersten Variante wird ein Modell verwendet, das die Vorhersage der durchschnittlichen Berechnungszeiten eines unevaluierten Volumens ermöglicht. Während das Modell verschiedene Attribute inkludiert, welche die jeweilige GPU spezifizieren, ist die Dateigröße das einzige Attribute, das der Beschreibung von Volumina dient. Für getestete Volumina mit hohen Auflösungen konnten Vorhersagen mit einem Determinationskoeffizienten zwischen 0.56 und 0.83 gemacht werden. In der zweiten Herangehensweise wurde ein lineares Regressionsmodell verwendet, das die durchschnittliche Berechnungszeit eines Volumendatensatzes auf einem unevaluierten System vorhersagen kann. In Abhänigkeit vom Testdatensatz konnten durchschnittlich Vorhersagen mit einem realtiven Fehler zwischen 5.33% und 22.22% auf verschiedenen, getesteten NVIDIA Grafikkarten erreicht werden.

Contents

List of Figures

List of Tables

1 Introduction

Volume visualization is an important application in science, medicine, engineering, and other fields. It is used to gain insight and better understanding of measured or simulated data of various phenomena. For explorative analysis of data sets, it is often desirable to be able to interact with the displayed volume. For a good experience during user interactions, low latencies and high frame rates are crucial. To accomplish those, hardware capable of massively parallel execution is often used nowadays. Predicting the performance of such an application on this hardware, however, is a challenging task due to many influencing factors. Those include the architectural complexity, software implementation details, and performance impact through user interactions such as change of camera position or transfer function.

The goal of this work is to create models which are capable of accurately predicting the performance of a volume rendering application. Two different directions in performance prediction are investigated. One deals with predicting the execution time of the next frame during runtime; the other one with predicting the average performance of rendering volume data sets on different systems. Thereby, the focus lies on predicting the performance on different graphics processing units (GPUs).

1.1 Motivation and Background

Generating images from volume data has gained a lot in importance over the past years. Especially scientific visualization, where numerical data from measurements and simulations is visualized, indicates its relevance. The need to facilitate the understanding of three-dimensional data in this and other fields such as medical applications or virtual engineering, for instance, has contributed to a growing importance of volume rendering.

An oftentimes central feature to such applications is the possibility to interactively explore the three-dimensional data. The user interaction thereby normally includes the change of camera configuration as well as adjustments to the transfer function. A fluent experience, which normally means at least five to twenty-five frames per seconds in the context of volume visualization (depending on user requirements and the form of application) is mandatory for most applications.

Interaction capabilities, the demand for high quality renderings that arises from the need to identify or distinguish small details in the data, and the increasing resolution of the data due to the improvement of measurement equipment, all contribute to a requirement of hardware providing high processing power. Because of the parallel nature of algorithms used for implementing volume visualization, this demand can be met by utilizing massively parallel hardware architectures or clusters. Acceleration devices enabling such parallel execution, which have greatly advanced in recent years, are graphics processing units. They can be used for real-time volume rendering.

To plan an infrastructure capable of executing high quality volume visualizations, it is advantageous to predict the performance that will be needed. Such a prediction can prevent later shortcomings in performance due to an underdesigned setup not meeting the demands, as well as unnecessarily high expenses caused by oversized hardware infrastructure.

If volume visualization is performed on a cluster or multiple acceleration devices, load balancing can be essential. Depending on the demands generated by the application, it may be necessary to perform load balancing to be able to process the rendering in a reasonable amount of time. Furthermore, a flexible occupation of computing power in form of variable amounts of processing nodes, for instance, can free computational capacity in a distributed system for other applications or save power. For a combination of a satisfying user experience, meaning a constant, basic amount of frames per second, and an efficient occupancy, however, it is important to predict the performance of an upcoming frame, so the computational load can be distributed accordingly.

A lot of research on performance prediction has been conducted in recent years. Most of it, however, targets scientific applications in high performance computing (HPC) environments. Performance prediction models focusing on visualization applications are surprisingly rare. The same holds true for models capable of dynamically predicting frame execution times during runtime.

1.2 Problem Statement

The problem addressed in this work, is to model, asses and predict the performance of volume visualization applications. Two different cases are focused on, thereby: One is predicting the upcoming frame during runtime of an interactive volume visualization application; the other one is predicting the performance on different hardware architectures.

Challenges for such a prediction arise because of high complexity in hardware and software in the context of parallel execution which is needed for high quality volume visualizations in real-time. Moreover, explorative interaction with volume data visualizations, such as transfer function changes and camera configuration adjustments, seriously impact the performance. While in the context of system performance prediction, the computation time of a model is of little importance, it is crucial for the runtime prediction of the execution time of the next frame. An advanced prediction model is needed to address all those challenges. To the best of the author's knowledge, such a model has not yet been proposed.

1.3 Contribution

During this work, performance prediction models based on machine learning techniques have been developed. They enable a user to predict the performance of a volume rendering application on GPUs in different ways. The first model can predict the execution time of a ray casting kernel during runtime. Crucially, the model is able to give reliable predictions during user interaction in form of camera position and transfer function changes. Special attention was paid to real-time capabilities of the model.

Two other machine learning models can make predictions on GPU capabilities for infrastructure planning, in the context of volume visualization. One predicts the average execution time performance of an unevaluated volume data set on different GPUs, the other one the average execution time of a volume visualization on an unevaluated graphics card.

1.4 Outline

The following chapters of this thesis are organized as follows:

Chapter 2 – Fundamentals and Related Work: The fundamentals of this work are presented in this chapter. This includes volume visualization in general and how it can be accomplished. As a technique commonly used for this task *volume ray casting* is discussed in further detail. It is also the method used in the volume renderer, implemented for this work. An overview of GPU architectures and the OpenCL parallel programming framework shall give a basic understanding of utilized hardware and programming techniques. The chapter continues with a general introduction to performance prediction and the field of machine learning as the specific method used in this work for implementing it. The last section of this chapter provides a summary of related work.

Chapter 3 – Performance Quantification - Methods and Implementation: This chapter starts with a description of the methods employed for implementing a volume ray caster, including used acceleration techniques. Thereby, the focus lies on those implementation details that are important for the model generation. In the following sections, the models for dynamic frame prediction and system performance prediction are presented.

Chapter 4 – Measurements and Results: This chapter covers the results gained from testing the performance and predictions of various model variations and different test cases. Testing procedures as well as measurements are presented and interpreted in detail. The chapter concludes with a discussion of the limitations of the developed models and offers some possible solution to the same.

Chapter 5 – Conclusion and Future Work A conclusion of the contributed work is drawn in this chapter. It closes with a proposition of possible future work.

2 Fundamentals and Related Work

This Chapter provides a short introduction to volume visualization in general (Section 2.1) and volume ray casting, the algorithm used at the core of this work in Section 2.2. Different visualization variants and acceleration techniques are also discussed in that section. In Section 2.3, an overview of the structure of current graphics cards architectures is given, as well as an introduction on how they can be used for parallel execution of ray casting. Performance prediction and machine learning basics are covered in Section 2.4. The chapter concludes with an overview of related work in Section 2.5.

2.1 Volume Visualization

Scientific visualization is mainly concerned with visualizing and rendering three-dimensional phenomena. These can have their origin in medicine, biology, physics, meteorology, architecture, engineering and other fields. The emphasis of the visualization is usually on a realistic, accurate, and detailed representation of the underlying data [Fri95]. This data is often generated either through measuring or simulating a phenomenon and is usually of a tree-dimensional and scalar form.

The goal of volume visualizations is to gain insight into the data and eventually get a better understanding of these phenomena. This is especially relevant, due to the fact that the amount of digital data, which is being generated and can be analyzed, has vastly grown over the last years.

A common way to render three-dimensional objects is the use of polygon meshes to represent the surface. The surface properties, such as color or light interaction, are then often modeled by using a shading algorithm, e.g. the Blinn-Phong shading model [Bli77]. However, these techniques usually only take surface-light-interaction into account and are therefore mostly unsuitable for visualizing light interactions inside three-dimensional objects.

Volume rendering techniques, on the contrary, focus on generating images from tree-dimensional, scalar data, directly. In most cases, this is more expansive than rendering the surface, but the quality of the generated images is usually superior.

In recent years, due to the evolution of volume rendering techniques and computing hardware, volumetric data is gaining importance in different scientific fields. In visual arts and 3D video games, fuzzy objects or phenomena can be represented using volumetric data as well. Examples are water, fire, steam, and clouds. In those fields, the volume data can also be generated synthetically with procedural methods, enabling new possibilities for artists.

2.1.1 Volume Data

The volume data used in this work for evaluation and measurement, is represented as a three dimensional array of voxels[1] [Kau94]. Each of these voxels is represented by a scalar value, taken from a continuous, three-dimensional signal

$$(2.1)\quad \Omega \subset \mathbb{R}^3 \to \mathbb{R}\ .$$

This value is obtained at a single point from the signal. Therefore, the data can be processed using a texture as data structure. According to the sampling theory, it should theoretically be possible to exactly reconstruct the signal. However, this is not practical due to the computational effort needed to convolve the sample points with the three-dimensional $sinc$-function: For an exact reconstruction of a single point, all sampling points have to be considered because of the infinite extent of this function. Furthermore, real-life data can contain step functions on sharp boundaries, which, in turn, have infinite extent in frequency domain.

In practice, either a box filter (nearest-neighbor interpolation) or a tent filter (trilinear interpolation) is used for sampling. The volume rendering application implemented for this work supports both.

2.1.2 Volume Rendering

The goal of volume rendering is usually, to map a discrete 3D data-set onto a 2D image plane. The generated image can then be displayed on a screen. [DCH88]

There are direct and indirect 3D volume rendering techniques. The latter convert the given data into an intermediate representation. This is usually a surface representation that can be rendered using traditional approaches. Iso-surface rendering is one of the indirect methods that is supported by the developed application. Marching cubes is

[1]**vol**ume **el**ements with cubic characteristic

another example of a well-known approach for implementing indirect volume rendering [LC87].

Direct volume rendering, on the contrary, considers the data as a semi-transparent material with physical properties. They include, but are not limited to reflection, emittance, scattering and absorption of light. Normally, an optical model describing these properties is evaluated. The scalar data value is mapped via a transfer function to physical quantities which, again, can be used for image synthesis. Based on the optical model, the light propagation through the volume is computed. This can be achieved by solving the *volume rendering integral* (Equation 2.5). For reasons of practicality, the integral is normally approximated numerically in real-world applications.

There exist several optical models for direct volume rendering [Max95]. The most common one assumes particles to emit light by factor q and absorb incoming light with coefficient κ. That means that the emitted radiant energy $q(t)$ at a certain distance $t = d$ from the observer is continuously absorbed along the viewing ray until it reaches the eye.

Assuming a constant absorption κ along the ray, the portion of radiant energy that actually reaches the eye q', can be calculated with

(2.2) $q' = q \cdot e^{-\kappa d}$.

If the absorption is not constant, however, the absorption coefficients have to be integrated along the distance d

(2.3) $q' = q \cdot e^{-\int_0^d \kappa(t)dt}$.

The second factor T is called the *transparency*, the integral over the absorption coefficients in the exponent the *optical depth*

(2.4) $T(d_1, d_2) = e^{-\int_{d_1}^{d_2} \kappa(t)dt}$.

Given an entry point s_0 of the ray into the volume and an exit point D, the volume integral defines the total amount of radiant energy I from this ray direction in the *volume rendering integral*

(2.5) $I(D) = I(s_0) \cdot T(s_0, D) + \int_{s_0}^{D} q(s) \cdot T(s, D)ds$.

2.2 Volume Ray Casting

Ray casting is a straight-forward approach to evaluating the volume rendering integral (Equation 2.5). It is an image-order, direct volume rendering method. [Lev88]

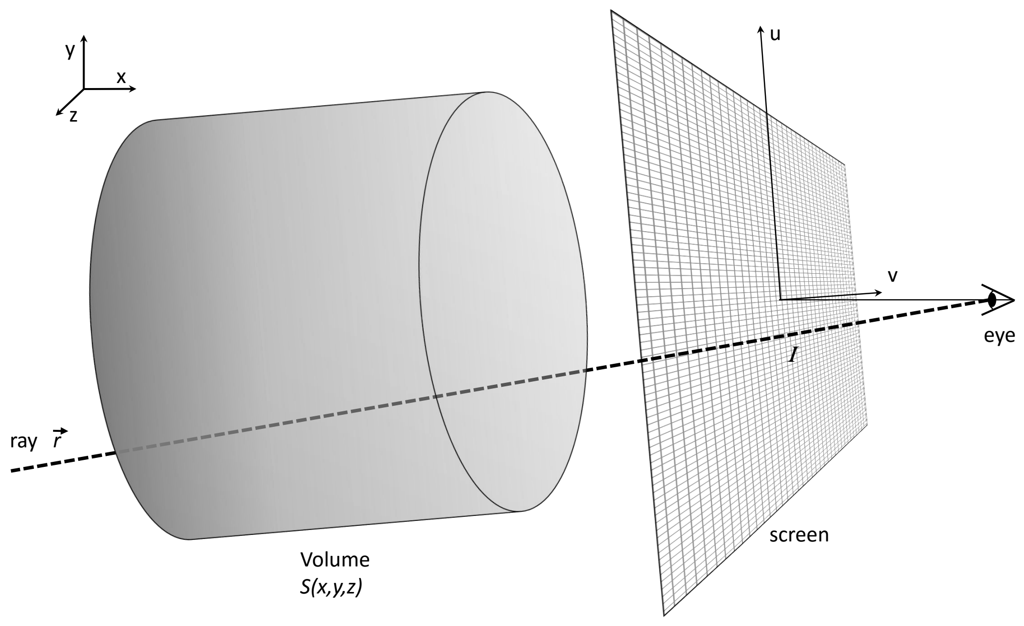


Figure 2.1: A schematic illustration of the volume ray casting approach: Rays are cast from the observers eye through the pixels on the image plain (screen). The scalar field representing the volume is sampled along the rays.

For each pixel on the 2D image plane, a viewing ray (the direction vector from the eye position through a pixel position) is shot into the scene as shown in Figure 2.1. The scalar values along each ray are sampled on equidistant steps from the volume data. Normally, trilinear or nearest neighbor interpolation is used for reconstruction of the signal during sampling. There are different approaches to visualizing the sampled data. The ones used in the application used in for this work, are described in Subsection 2.2.1.

2.2.1 Rendering Techniques

Four different rendering techniques for ray casting are implemented in the application. They are pictured in Figure 2.2 and include the direct volume rendering techniques line-of-sight integration, maximum intensity projection, and direct rendering using a custom transfer function. As an indirect technique, iso-surface visualization is supported.

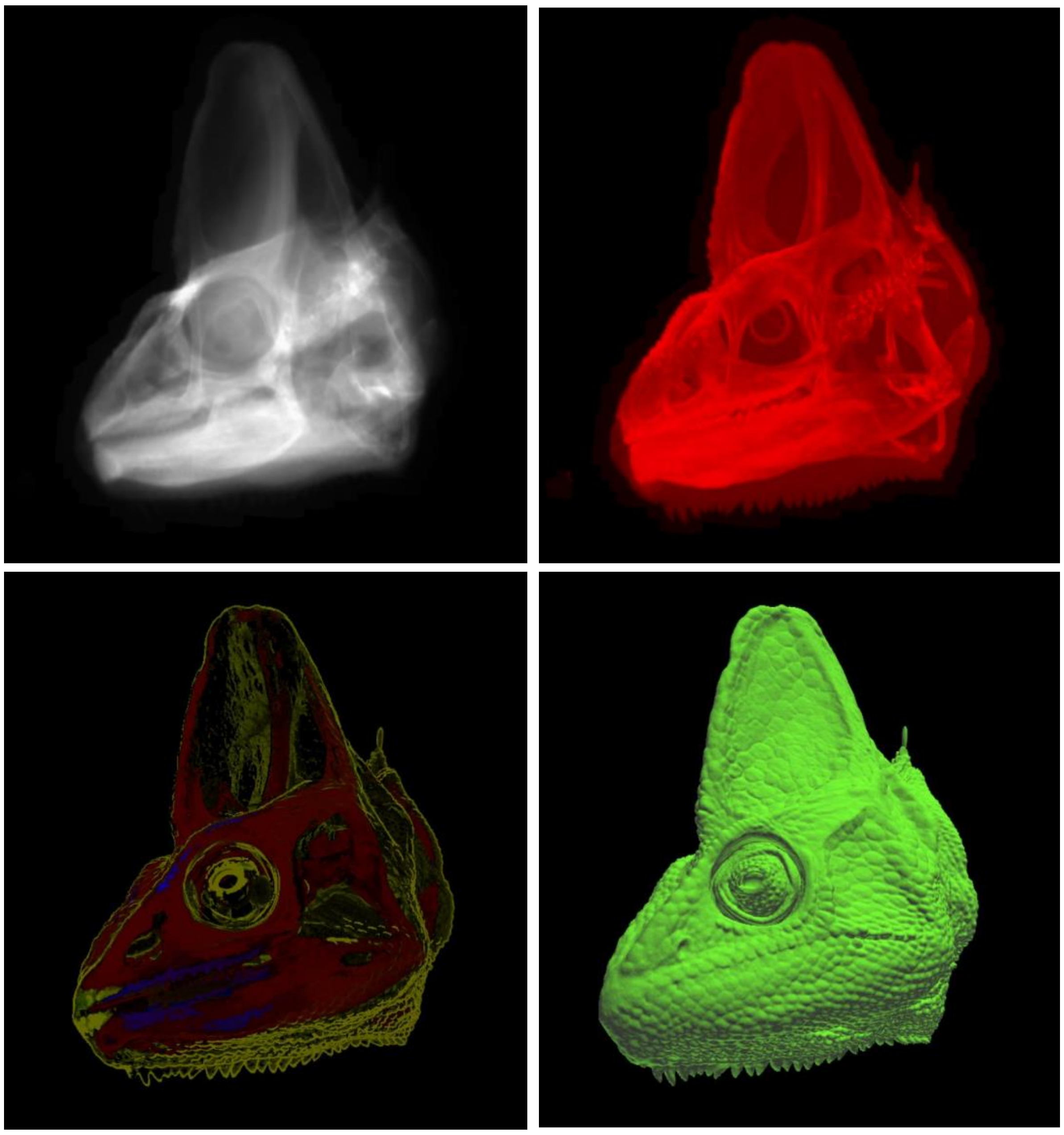


Figure 2.2: Different direct and indirect ray casting modes supported by the volume visualization application: Line-of-sight integration (top left), maximum intensity projection (top right), direct volume rendering with transfer function (bottom left), and iso-surface rendering as an indirect method (bottom right).

Line-of-Sight Integration

A simple form to implement ray casting is line-of-sight integration. The scalar values are integrated along the viewing ray and scaled with a global factor. This factor can be used to scale the intensity of the projection according to the user's liking. With a continuous scalar field $S(x, y, z)$ and scaling factor c, a pixel value $P(u, v)$ results from the scaled integral

$$P(u,v) = c \int S[\vec{x}(t)]dt \ , \tag{2.6}$$

where $\vec{x}(t) = (x(t), y(t), z(t))^T$ is the parametrized viewing ray. Due to the discrete characteristic of the data, the integral from equation 2.6 is approximated with a Riemann sum:

$$P(u,v) \approx c \sum_{n=0}^{N} S|\vec{p}|, \quad \text{with} \quad \vec{p} = \vec{s_0} + n \cdot \Delta\lambda \cdot \vec{r} \tag{2.7}$$

where $\vec{s_0}$ is the camera position, $\vec{r}$ the ray direction, and $\Delta\lambda$ the step width. The numerical solving of this sum for each pixel of the window, results in a volume visualization in the line-of-sight integration style. The resulting image of this form of volume rendering, is shown in the top left part of Figure 2.2.

Maximum Intensity Projection

In contrast to line-of-sight integration, the maximum intensity projection technique does not integrate the sampled values along a viewing ray but rather maps the maximum value found onto the respective screen pixel. There are also variants, were the projected values are scaled using the respective depth value to improve depth perception. An impression of the visual difference in comparison to line-of-sight integration can be gained from Figure 2.2 (maximum intensity projection at the top right).

Direct Volume Rendering with Transfer Function

Line-of-sight integration as well as maximum intensity projection only provide a limited range of interaction possibilities to the user. To enhance these, a *transfer function* can be employed which defines a color and transparency value for each scalar value that could be sampled from the volume data set being rendered. This enables the user to visualize different densities with varying colors, for instance.

Analogue to line-of-sight integration, a volume is traversed along viewing rays for each pixel. But instead of summing up the scalar values sampled along the way, the transfer function is used to map these values to a color and transparency. Given a color C_1 and transparency α_1 of the previous step and the color C_2 and transparency α_2 evaluated with the transfer function, the color is calculated with

(2.8) $C = C_1\alpha_1 + C_2\alpha_2(1 - \alpha_1)$,

and the transparency with

(2.9) $\alpha = \alpha_1 + \alpha_2(1 - \alpha_1)$.

The calculated values C and α are then used as C_1 respectively α_1 in the next integration step. The results can be examined in Figure 2.2 at the bottom left.

Iso-Surface Rendering

In contrast to the other three techniques discussed in this section, iso-surface rendering is an indirect volume rendering method because only a subset of the volume data is used to create the visualization.

The rays are traversed until a given iso-value S_{iso} is passed while sampling at two consecutive positions $\vec{p_1} = \vec{s_0} + n \cdot \Delta\lambda \cdot \vec{r}$ and $\vec{p_2} = \vec{s_0} + (n+1) \cdot \Delta\lambda \cdot \vec{r}$. That means the equation

(2.10) $(S(\vec{p_1}) - S_{iso}) \cdot (S(\vec{p_2}) - S_{iso}) < 0$

holds true for the points p_1 and p_2.

The actual surface point can then be obtained with linear interpolation. The related surface normal can be numerically evaluated with the gradient of the scalar field at the interpolated point: $\vec{n} = \vec{\nabla} S$. With surface positions and normals, a surface shading algorithm such as Blinn-Phong shading can be used to render the iso-surface. The result is shown in Figure 2.2 at the bottom right.

2.2.2 Acceleration Techniques

There are several techniques that are commonly used to accelerate the ray casting procedure. One of these aims at terminating rays early that generate no further information. It is therefore called *early ray termination*. Another technique that is often implemented in volume rendering applications, tries to speed up the algorithm by "leaping" over empty space in the volume data to avoid unnecessary sampling and is often referred to as *(empty) space leaping* or *skipping*. [HLSR09]

Early Ray Termination

Early ray termination only works with front-to-back volume ray casting, i.e. the rays are traversed from the observer's position in the direction of the volume. Using iso-surface rendering, the ray can be terminated immediately when the iso-value is passed over.

Applying either line-of-sight integration or transfer functions in the ray casting, a fixed value close to 1.0 is normally defined. When using the line-of-sight method, one can terminate all rays greater than or equal to the defined value. Similarly, using transfer functions, rays with an evaluated alpha value (see Equation 2.9) exceeding the defined value can be terminated early.

Empty Space Leaping

Oftentimes, only parts of the volumetric data contain values relevant for the rendering. The rest is either empty or noise generated by the data scanner, for instance. These irrelevant values are normally close to 0.0. The amount of empty parts can vary strongly depending on the shape of the object to be displayed. There are two different conceptional approaches of skipping these parts: image-order and object-order. Hybrid approaches have also been developed [SHN+06].

The image-order technique tries to avoid sampling empty space by leaping a predefined distance - which is greater than the sampling width - along the ray until relevant data is sampled. Usually, the last leap is backtracked and the data is sampled from this point, using the normal sampling width, at least until the former leaping point is reached. The algorithm can then restart leaping if empty space is hit again.

These possibly multiple leaping passages along a single viewing ray is an advantage for image-order empty space skipping, as empty space inside objects can be skipped without additional cost. Another advantage of this method is the almost complete lack of overhead in the calculation process. The major drawback, however, is that relevant data is missed if the width of consecutive relevant data is smaller than the leaping distance. This usually results in artifacts or moiré patterns at corners or thin parts of the displayed objects.

The object-order empty space leaping approach focuses on a pre-processed bounding geometry to determine more accurate entry and exit points of the viewing rays than the - otherwise normally used - bounding box. There are different approaches to generating a suitable bounding geometry. For volumetric data that is structured as a uniform grid, a forward approach consists in scaling the data in a down-sampling step. Afterwards, it can be determined for each of the low resolution voxels whether it is empty or not,

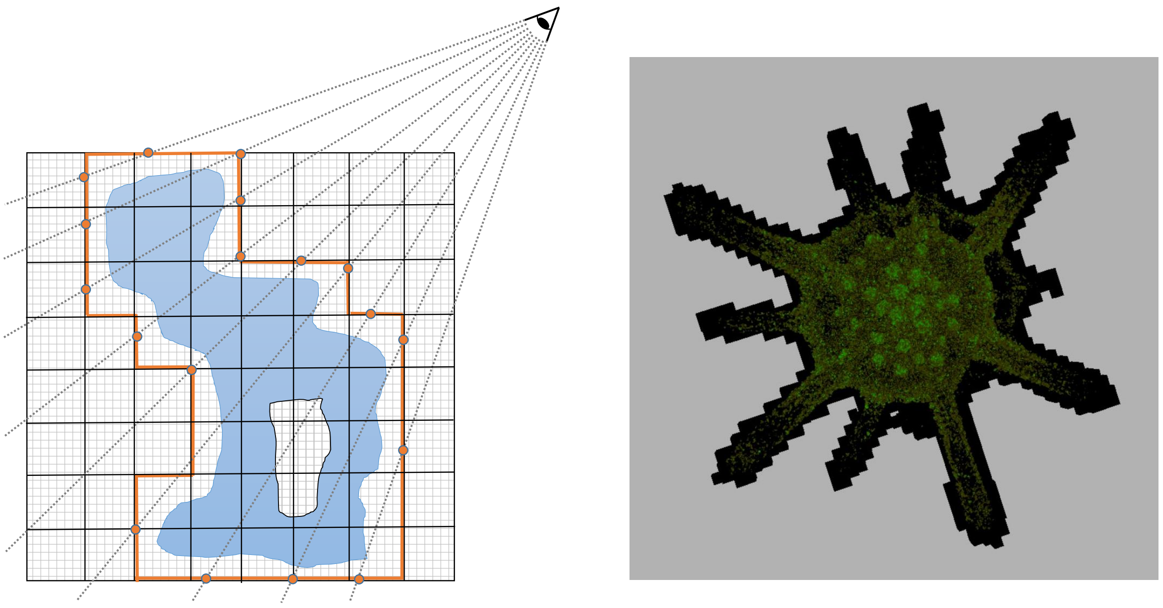


Figure 2.3: Left: Ray casting with object-order empty space leaping. A bounding geometry (orange) on a coarse grid defines ray entry and exit points. Right: Rendered bounding geometry (black) which was generated based on a transfer function.

i.e. whether the interpolated value is below a certain threshold. The use of a transfer function can further refine the visibility selection of the coarse voxels.

Generating front- and back-faces for each of the nonempty voxels yields a bounding geometry when rasterized. Figure 2.3 shows the schematic approach as well as a rendered bounding geometry. When storing only one entry and one exit point, empty space inside the volume (and, consequently, the bounding geometry) cannot be skipped. This is a major drawback of this method. One possible solution to the problem is using multiple input and output points for each ray which are defined only if the ray passes multiple empty parts. However, calculating multiple points per ray is usually more complex.

The extend of the performance improvements largely depends on the data set being rendered. For the visualization of volumetric data generated by computed tomography (CT) scans or magnetic resonance imaging (MRI), object-order empty space leaping usually improves rendering times significantly. That is because the visible parts in those kind of data sets are normally fairly localized. If other kinds of volumetric data, representing phenomena such as gases or fluids, where the relevant parts are often scattered over the entire space, are considered, this approach of space leaping can

Instruction streams	**data streams**	
	Single Data	**Multiple Data**
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

Table 2.1: The four categories of Flynn's taxonomy. [Fly66]

actually be counterproductive for the performance of the ray casting process. The reason for this disadvantageous reaction is that the overhead generated from computing the bounding geometry is larger than the performance gained by skipping the probably small empty spaces.

2.3 Ray Casting on the GPU

Modern graphics processing units (GPUs) are designed to process hundreds of similar tasks in parallel [AMD15a]. This makes GPUs well suited for ray casting algorithms because the viewing rays are independent from each other and can, therefore, be processed simultaneously.

Although it is possible to use other devices, such as central processing units (CPUs), for volume ray casting, this work focuses on ray casting on the GPU. To support different systems, however, the ray casting kernel of the application which was developed for testing during this work is realized with the *OpenCL* parallel programming framework. This allows running the application on heterogeneous systems without further modification of the code. A short overview of OpenCL and the underlying models is provided in Section 2.3.3.

2.3.1 Flynn's Taxonomy

Flynn's taxonomy is a rough classification system for computer architectures. In this classification, architectures are divided based on *data streams* and *instruction streams*. Furthermore, the access variant is used as a criterion, discriminating between single or sequential and multiple or parallel architectures.. Table 2.1 shows the four categories. [Fly66]

Single instruction, single data (SISD): A single processor accesses a single data element. From a single instruction memory, a single instruction stream is executed. SISD is normally associated with the "classical" *von Neumann* architecture.

Single instruction, multiple data (SIMD): From a common instruction stream, a single instruction stream operates multiple processors. They have access to different data. Data parallelism allows the synchronous execution of commands on multiple data elements. For SIMD, special hardware, such as array computers, is needed.

Multiple instruction, single data (MISD): Multiple processors with own instruction streams and instruction memory access single common data memory. Although *systolic arrays*[2] are often classified as MISD, there is some controversy due to the fact that a data swarm passing through the array is transformed. That means, the cells technically do not access the same data, which speaks against the MISD classification.

Multiple instruction, multiple data (MIMD): Multiple processors with their own instruction stream and instruction memory access their own data memory. Executing a single program multiple times can result in different instruction streams. This is called single program, multiple data (SPMD) and is a subset of MIMD. A classical multiprocessor system or cluster is a representative of the MIMD class.

2.3.2 GPU Architectures

Recently, graphics processors have gained a lot of flexibility and are now on the verge of becoming an essential part of the computing landscape. GPUs started out as special purpose hardware for fixed function processing of geometry transformation and lighting. They evolved from devices supporting simple shaders into such which feature fully programmable computing capabilities. Those enable GPUs nowadays, to perform demanding parallel tasks such as volume ray casting in real time.

The current architecture generations of the three main desktop GPU manufacturers AMD, Intel, and NVIDIA all feature a similar structure. [NVI14] [AMD15a] [Int15]

Using the terminology of AMD, they contain basic *Compute Units* that are loosely coupled in clusters. Those Compute Units contain *SIMD-Units* that, in turn, include the actual *Stream Processors*. The terms for comparable units used by NVIDIA and Intel are shown in Table 2.2.

At the lowest level, the streaming processor blocks (current generation NVIDIA: 32, AMD: 16, Intel: 7) execute the same instructions in lockstep. This is also referred to as the *Single Instruction, Multiple Threads* (SIMT) principle in the style of Flynn's taxonomy

[2]A network of homogeneous data processing units (DPUs) that are tightly coupled. Each node processes functions with the data received from its upstream neighbors and passes the result downstream.

NVIDIA Maxwell GM204		**AMD GCN 1.2 Fiji XT**		**Intel gen9 GT4/e**	
Graphics Processing Cluster (GPC)	4	Compute Unit Cluster	16	Slice	3
Streming Multi-processor (SM)	$4 \cdot 4 = 16$	Compute Unit (CU)	$16 \cdot 4 = 64$	Subslice	$3 \cdot 3 = 9$
CUDA Core processing block	$16 \cdot 4 = 64$	SIMD-Unit	$64 \cdot 4 = 256$	Execution Unit	$9 \cdot 8 = 72$
CUDA Core	$64 \cdot 32 = 2048$	Stream Processor	$256 \cdot 16 = 4096$	Thread	$72 \cdot 7 = 504$

Table 2.2: Current GPU architectures, their hierarchical design and numbers of the respective units of a selected implementation. [NVI14] [AMD15a] [Int15]

(see Section 2.3.1). That means instruction-level parallelism is realized, but neither out-of-order or speculative execution, nor branch prediction.

SIMT places itself between the two principles SIMD and MIMD of Flynn's taxonomy (see Section 2.3.1). It differs from classical SIMD systems for each thread disposes its own registers and a separate instruction address counter. Consequently, the instruction stream can diverge. Nevertheless, the coherence of the instruction set of thread groups is still controlled, so it cannot be classified as a MIMD process. In practice, this means that divergent streams are executed sequentially and the results of a thread group are partly discarded. A high thread divergence therefore leads to inefficiency.

Table 2.2 gives an overview of this hierarchical structure extended by vendor specific terminology. There are also some numbers listed for current realizations of the architectures in the consumer sector. Although the hierarchical layers are quite similar, one cannot directly compare them to each other because there are a lot of other different units (e.g. vector units, arithmetic-logical-unis, special-function-units, load/store-units etc.) in various numbers and connections included that are not listed in the table for reasons of clarity.

Each SIMD-unit has a certain amount of private memory through registers. The compute units have their own local memories that can only be accessed from inside these units. These local memories usually feature level one caches (L1). The globally visible device memory can be accessed through memory controllers and is cached through a level two (L2) layer.

Modern dedicated single chip GPUs support theoretical data transfer rates of up to 512 GByte/s and theoretical single precision processing power of up to 8.6 TFlops[3] [AMD15b].

2.3.3 OpenCL

There exist several application programming interfaces (API) respective languages to access and program the computing capabilities of modern GPUs and other devices capable of parallel program execution. Although - due to the success of CUDA (a framework specific for NVIDIA hardware) - it may not be the most popular, OpenCL makes a good choice as a parallel computing framework. It is supported by a wide range of heterogeneous devices as almost all major GPU, CPU, processor, and acceleration device manufacturers supply a conformant implementations of the open standard for their hardware [How15].

OpenCL consist of an API specification and *OpenCL C*, a programming language for computation kernels, the programs that can run in parallel on the devices. The API has to be supported by the acceleration device and therefore an implementation has to be supplied by the respective vendor. The OpenCL C programming language is based on the ISO C99 standard but lacks some of the features that are not reasonable to use in a parallel programming context, such as function pointers, recursion and arrays of variable length. On the other hand, language extensions for operations and data types that are often used in parallel programs and therefore often hardware accelerated by capable OpenCL devices, are provided. Those extensions include for instance, vector data types of fixed-length and vectorized operations on those types. In the following, a brief overview of the latest version (2.1 at the time of writing) of the OpenCL platform, memory, and execution model is given. [How15]

Platform Model

The OpenCL platform model requires a *Host* that is normally a CPU. Furthermore, there are one or more *Compute Devices* which are used for the actual computation, such as a GPU, for instance. The compute devices contain several *Compute Units* that, in turn, contain *Processing Elements*. Figure 2.4 depicts the structure of the platform model.

[3]**T**erra **fl**oating point **o**perations **p**er **s**econd

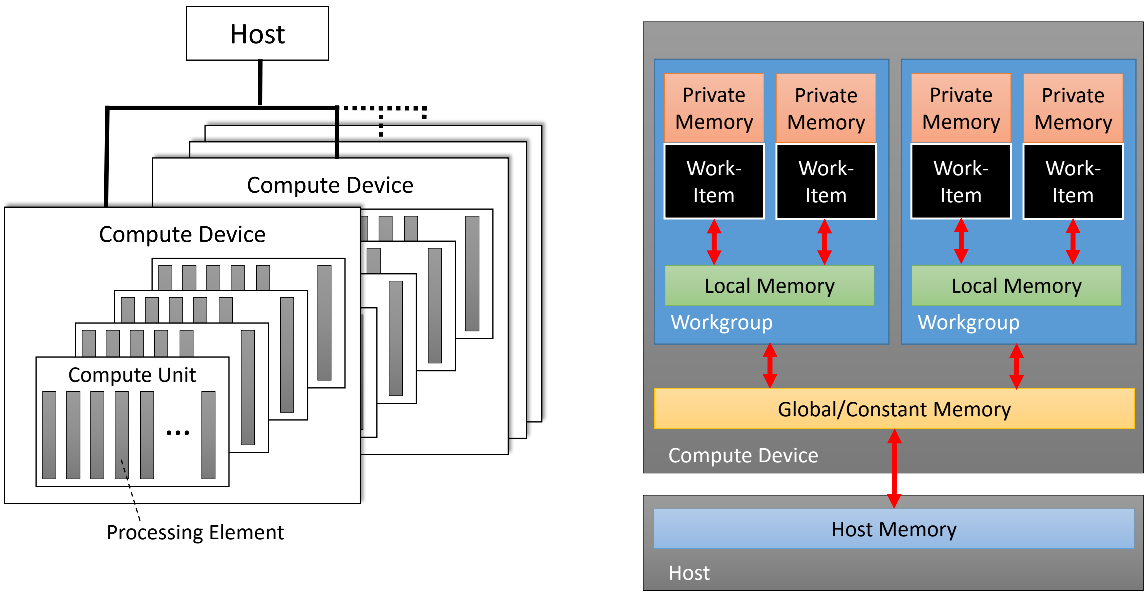


Figure 2.4: A schematic visualization of the OpenCL platform model (left) and the memory model (right) [How15].

Memory Model

The OpenCL memory model differentiates between four different memory scopes (see Figure 2.4). The *Host Memory* is only accessible by the host. On the compute device, there is a *Global/Constant Memory* that can be accessed by all workgroups. Each workgroup shares a *Local Memory* which can be accessed by all work-items inside this workgroup. The work-items themselves have each access to an own *Private Memory*. The memory management is explicit.

Execution Model

An OpenCL application runs on the host. This latter submits work to the compute devices. Work-items, which are the basic units of work on an OpenCL device, execute functions that are defined in *Kernels*. Various kernels are abstracted in a *Program* that runs in an environment on the device. This environment has to be defined by the *Context*. Kernel executions are queued in the *Command Queue* which can be run in-order or out-of-order.

2.4 Performance Prediction

In computer science, performance prediction is used to estimate the execution time of an application on a given system. It plays an important role in different fields, including for example design of new hardware architectures, machine code generation by compilers and software engineering. Performance prediction techniques usually abstract the target architecture by using a *model*. Different levels of abstraction of such a model normally go along with precision and efficiency of the prediction.

This work focuses on the performance prediction of a volume ray casting algorithm on a single graphics device able of general-purpose computing (GPGPU). There are three major factors that influence the execution time of a parallel application on this type of devices [Mic10]:

Arithmetic operations: Mathematical calculations. The theoretical limit of a device is often specified as floating point operations per second (FLOPS) or instructions per second (IPS).

Memory accesses: Read and write operations to the main memory. The theoretical bandwidth limit for a device is normally specified in bytes per second.

Latency: This factor is caused by poor overlap or balance between memory accesses and arithmetic operations.

A parallel algorithm is, therefore, either memory bound or computationally bounded. In addition, it can be latency bound if the machine balance is bad.

A straight forward approach in creating a model for performance prediction can be, to divide a program into a sequence of N basic blocks B that are connected via an execution path. A basic prediction can be made by profiling those blocks and summarizing the profiled times t, multiplied with the execution frequency f

$$(2.11)\quad T_{total} = \sum_{n=1}^{N} (t_{B_n} \cdot f_{B_n}) \; .$$

The efficiency of this method, however, is rather limited. That is, it may yield poor results when used for modern systems or parallel applications due the increased complexity that comes from features such as out-of-order execution, branch prediction, execution scheduling, and instruction pipelining.

It is assumed here that using a more complex model can produce more accurate predictions. A convenient way to generate such a model is by using machine learning methods, so that arbitrary input factors can be taken into account.

2.4.1 Machine Learning

Machine learning is a field of computer science related to computational statistics. Research in this area focuses on creating algorithms that learn from data and use it to make predictions. That means, historical data is used to create a *model*, which, in turn, allows the user to predict the behavior of the data in the future. The model can also be designed to change dynamically by adding more training data over time.

Learning Techniques and Validation

A machine learning model is trained on *features* or *attributes*. These are properties that influence the outcome of the data. Machine learning algorithms can roughly be divided into two categories, depending on the principle learning method: *supervised learning* or *unsupervised learning*.

In supervised learning, the input features as well as the expected output have to be defined explicitly. The main techniques are *regression* and *classification*. Classification tries to assign data to various categories based on the training data. Regression, on the contrary, predicts actual values for a set of features and is, therefore, much stronger than classification.

In unsupervised learning, the output is not exactly known beforehand. The goal is to find hidden patterns and structures in a data set. Principle component analysis (PCA) is an example of an unsupervised learning method that tries to reduce the number of features by combining them based on (hidden) correlations between those attributes.

There are several techniques to validate the generated model. A commonly used method is *cross validation*. In essence, the data set is split in two parts. One part is used for training the model, while the other is used for testing it. Comparing the actual values to the predictions, one gets both: a means to judge the performance of one's model as well as the used training data.

Machine Learning Terms

A selection of terms that are commonly used in connection with machine learning as well as in this work are shortly defined below.

Variance: An error caused by small-scale fluctuations or sensitivity in the training data. A high variance can lead to overfitting if random noise in the training data is modeled rather than the intended outputs.

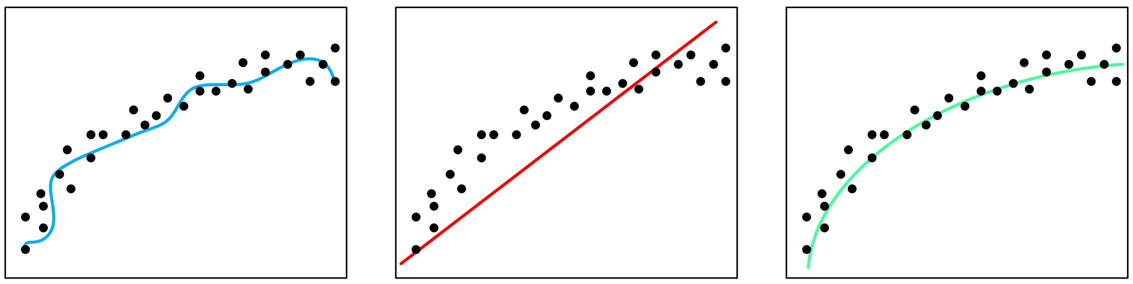


Figure 2.5: Illustration of an overfitted (left), an underfitted (mid), and a decent fit (right) regression line on a fictive data set.

Bias: An error caused by erroneous assumptions in the learning algorithm. A high bias can lead to underfitting if relevant relations between features and targets are missed.

Overfitting: A result of curve fitting that can occur if function is fitted too "closely" to the training data. This may happen if noise is incorporated into the model, for instance. An example of overfitting on a fictive data set is illustrated in Figure 2.5.

Underfitting: A result of curve fitting that can occur if too much statistical data is left out of the model. This may happen if the wrong regression type is used. For instance, applying linear regression to data with a non-linear structure can lead to underfitting. An example of underfitting is presented in Figure 2.5 as well.

2.4.2 Ordinary Least Squares

Ordinary least squares (OLS) is a form of linear regression and, thus, a supervised learning method. It tries to estimate unknown parameters in a linear regression model whereby it searches for a curve which is as close to the points of a data set as possible.

Basically, OLS tries to minimize differences between observed responses and responses predicted by the linear approximation. These differences are called *residuals*. Given a model curve $f(x_i)$ and observed response data y_i with the function parameter vector $\vec{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_m) \in \mathbb{R}^m$ and $\vec{f} = (f(x_1, \vec{\alpha}), \ldots, f(x_n, \vec{\alpha})) \in \mathbb{R}^n$, the sum of squared errors can be written as

$$\sum_{i=1}^{n}(f(x_i, \vec{\alpha}) - y_i)^2 = \|\vec{f} - \vec{y}\|_2^2 \,. \tag{2.12}$$

The parameters α_j which minimize the sum of the squared errors need to be chosen. Mathematically, this problem can be expressed as

$$\min_{\vec{\alpha}} \|\vec{f} - \vec{y}\|_2^2 \; . \tag{2.13}$$

Computationally this can be solved using a singular value decomposition of X, the design matrix of size $n \times m$. Assuming that $n \geq m$, this method has a complexity of $\mathcal{O}(n \cdot m^2)$.

The quality of a model generated with the ordinary least squares method can be estimated using the R^2 *score*. R^2 is the coefficient of determination of the prediction. With n being the number of samples, u being the regression sum of squares

$$u = \sum_{i=0}^{n-1} (y_{i,true} - y_{i,pred})^2 \tag{2.14}$$

and v the residual sum of squares

$$v = \sum_{i=0}^{n-1} (y_{i,true} - \bar{y})^2 \; , \tag{2.15}$$

the coefficient of determination can be defined as

$$R^2 = (1 - \frac{u}{v}) \; . \tag{2.16}$$

The best R^2 score is 1.0, smaller numbers indicate a worse score. It can also become negative because the model at use can be arbitrary bad.

2.5 Related Work

Performance prediction in the context of parallel systems has been an active field of research in recent years. Work targeting GPUs in particular is yet limited since graphics processors have only recently emerged as universal parallel computing devices.

In the high performance computing (HPC) context, however, a lot of research on performance prediction has been conducted. Often, but not always, the research in that area focuses on applications for numerical computations. In the following, an overview of the current state of research is provided.

Different methods have been presented to respond to the challenge of predicting application performance in the context of large parallel or heterogeneous systems in HPC. Bailey and Snavely [BS05; SCW+02] present a prediction framework based on three parts: application signatures, a compact representation of fundamental operations in an application, machine profiles which represent system capabilities and convolutions and means used to combine signatures and machine profiles.

Barnes et al. [BRL+08] introduce regression based techniques for performance prediction: one technique using execution time only, a second one based on processor information, and a third using the global critical path of an application. They focus on predicting parallel program scalability and build their regression model based on results gained from program executions on small subsets.

Yang et al. [YMM05] seek to predict performance based on relative performance: In their observations based approach, that uses no modeling, they try to make predictions on cross platform performance of an application through partly executing it. Tikir et al. [TCSS07] use benchmark results for performance prediction of HPC applications. Their approach includes the representation of memory bandwidth as a function of cache hit rates per machine which is done by means of a genetic algorithm.

Two different predictive models are presented by Lee et al. [LBdS+07]. One uses piecewise polynomial regression, the other artificial neural networks. A machine learning approach on performance prediction is proposed by Singh et al. [SİM+07]. They automatically build models based on results gained from executing samples. Multi-layer neural networks are employed for the prediction.

Research targeting graphics cards as computing devices capable of parallel execution, has also been contributed in recent time. Performing performance prediction with the application of a analytical model, is proposed by Baghsorkhi et al. [BDP+10]. Thereby, an auto-tuning compiler is provided with performance information so that it can assist in finding bottlenecks to improve the implementation. The model is validated only for NVIDIA GPUs using the CUDA parallel programming framework. Hong and Kim [HK09], too, use an analytical model and CUDA in their work. They predict kernel execution times based on the number of parallel memory requests.

Kothapalli et al. [KMR+09] developed a model specific to the CUDA parallel programming framework, which takes many architectural specifics into account, such as scheduling, memory hierarchy, and pipelining. Through an asymptotic analysis approach, code performance prediction is performed. Boyer et al. [BMK13] present a technique to port CPU code, to be able to run it on GPUs while making a prediction of the possibly gained speedup. Kernel execution times as well as data transfer times between CPU and GPU are, thereby, factored in. GROPHECY is a framework for GPU performance projection based on CPU code skeletons [MMK+11]. It was developed to enable the user to estimate the performance benefit gained from GPU acceleration without any actual GPU programming.

An approach on predicting the speedup from adding one or multiple GPUs to a system as well as changing the size of the input data set is proposed by Shaa and Kaeli in their work [SK09]. The authors investigate performance prediction of six different scientific applications, also using CUDA. They include a ray tracing application.

A performance model specific to NVIDIA GPUs of the 200-series was presented by Zhang et al. [ZO11]. It is microbenchmark-based and uses the instruction set of that GPU generation to make a quantitative analysis of GPU code performance and detect bottlenecks.

So far, the field of visual computing in particular has only been investigated by few researchers in regard to performance prediction. Work targeting the design of accelerator architectures for visual computing has been presented by Mahesri et al. [MJCP08]. It is, however, restricted mainly to benchmarks.

3 Performance Quantification - Methods and Implementation

This chapter provides detailed information about the methods used to predict the performance of a volume ray caster as well as their implementation. In Section 3.1, relevant parts of the application which was developed during this work and used for the performance analysis are described. The following two sections concentrate on the methods utilized for the actual performance predictions.

Section 3.2 focuses on dynamic frame time prediction in real time. The features used for the machine learning model that shall be able to make fast predictions during application runtime, are presented therein. In Section 3.3, the methods used to predict the performance of the volume visualization on different GPU architectures are discussed. Thereby, two approaches are followed: The development of methods enabling a performance prediction of unevaluated data sets on already known hardware, as well as the performance prediction of an already tested volume on an unevaluated system.

3.1 Ray Casting Application

As the basis of the predictions, an application for volume ray casting on uniform grids is implemented. It supports the four rendering modes described in Section 2.2.1, namely line-of-sight integration, maximum intensity projection, iso-surface rendering, and a ray casting based on user-defined transfer functions. The last mode features the most interaction possibilities for the user as he or she can interactively manipulate the transfer function in real time. That is the reason why this work focuses on the performance quantification of this rendering mode in particular. The approaches made and results gained while examining the transfer-function based rendering, can be transferred in most parts for the other methods without greater effort.

For performance improvements, both acceleration techniques described in Section 2.2.2, early ray termination and empty space skipping, are implemented. Since the use of image-order empty space skipping entails the risk of artifacts, an object-order version was implemented for the application. The two acceleration methods influence the performance significantly and are, thus, an essential part of the performance analysis and also the model generation. In Subsection 3.1.2, the specific implementation of object-order empty space leaping is described in further detail. The implemented version is loosely based on the technique delineated in the volume ray casting course notes from Hadwiger et al. [HLSR09].

3.1.1 Frameworks Used

The OpenCL parallel programming framework (see Section 2.3.3) in version 1.2, is used for the ray casting algorithms. This allows execution on GPUs as well as CPUs and possibly other parallel acceleration devices. However, this work focuses on performance predictions for GPUs.

For rendering purposes, the OpenGL application programming interface (API) in version 4.3, is used. In recent versions, it is possible to share resources between OpenGL and OpenCL without copying them. These capabilities are used to avoid unnecessary, possibly slow memory transfers between the host and the device. The applied API versions are supported by most dedicated graphics cards and CPUs released after 2010.

3.1.2 Implementation of Object-Order Empty Space Skipping

As described in Subsection 2.2.2, object-order empty space skipping needs a bounding geometry. If the user loads a new volume data set, an OpenGL compute shader performs a down-sampling of the data. The voxels of the original data are interpolated linearly and saved as a 3D-texture. A down-sampling factor of 16 results in good performance gains as well as fast build-up times and low sizes of bounding geometries for most data sets.

In the next step, the bounding geometry is generated. Because the geometry depends on a threshold value as well as on the transfer function, the generation has to be repeated every time the user changes either one of them. To speed up the creation time, a part of the OpenGL rendering pipeline is used.

An instanced draw call of `GL_POINTS` primitives is invoked, with the instance count being the number of voxels in the coarse grid. In the vertex shader, the corner points of the grid voxels are calculated based on the `gl_InstanceID` of every vertex and the positions passed to the geometry shader. In the geometry shader, the scalar voxel values are interpreted. That means, based on the transfer function and threshold, the algorithm decides whether a voxel is actually visible. The threshold value serves the purpose of filtering out noise (low values that have no significance for the volume rendering) in

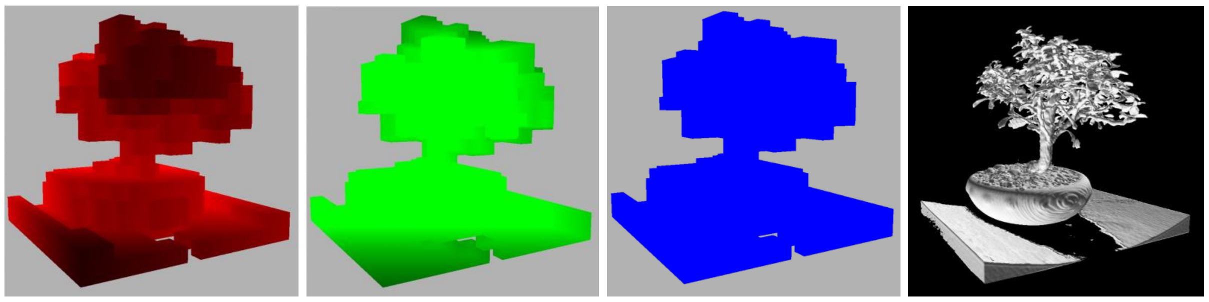


Figure 3.1: The color channels of the depth texture generated for the empty space skipping and prediction model input. From left to right: minimum depth value, maximum depth value, footprint, and an iso-surface rendering of the data set.

the data. For each visible voxel, the geometry shader invokes the creation of vertices for the six possible cube sides. Via transform feedback, the generated vertex data is, then, transferred back to the host so that the rasterizer in the OpenGL pipeline can be skipped. This contributes to short GPU processing time when generating a bounding geometry.

Every time the user changes the camera position, the bounding geometry is rendered to an off-screen buffer. For each fragment, the minimum and maximum depths are written to separate color channels as well as an identifier called *footprint*, which indicates whether the fragment is inside the bounding geometry or not. To obtain maximum precision, a 32-bit RGB-texture is used for saving the generated depth values. This texture is a shared resource between OpenCL and OpenGL. Figure 3.1 shows renderings of the separate color channels.

The approach of generating the minimum and maximum depth values in a single rendering pass was taken from the dual depth peeling method [Lou08]. In essence, the bounding geometry is rendered with blending enabled (`GL_BLEND`) while using `GL_MIN` as the blend equation. The vertex positions in eye space are calculated in the vertex shader and then passed to the fragment shader. After normalization, by means of dividing by the far plane distance, the z-value and the negated z-value of the fragment are directly written to the output channels. Due to the utilized blending equation, this results in the minimum depth value of the fragment as well as the negated maximum. Aside from this, the shader writes 0.0 or 1.0 to the third output channel depending on whether the fragment is located inside the bounding geometry or not.

In principle, this method can be extended to support multi-layer depth values. Such an extension enables the skipping of empty parts inside the volume but comes at the cost of an additional rendering pass per depth-layer.

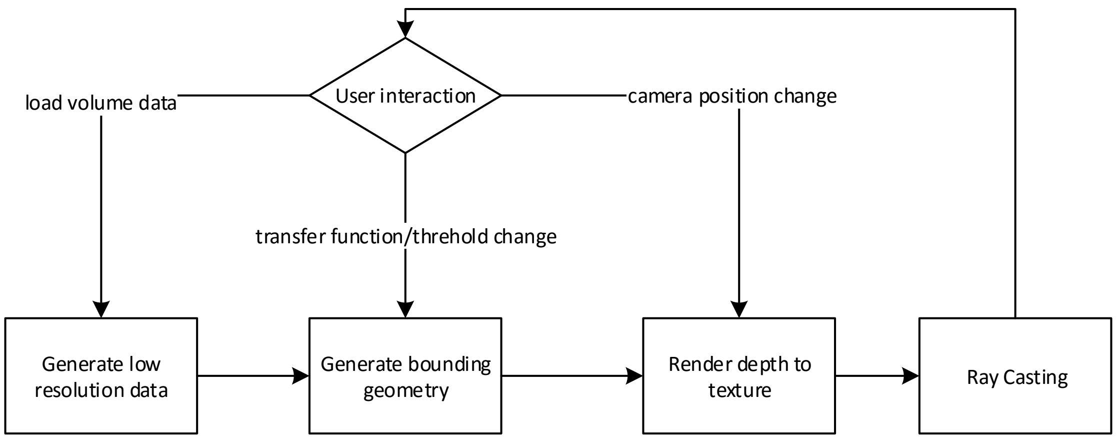


Figure 3.2: A flow diagram illustrating the functions involved in the empty space skipping algorithm, constrained to the user interaction.

The generated depth texture supports mipmaps[1]. From the mipmaps with the lowest possible resolution - in fact an image with 1×1 pixel - one can easily extract the average value of each color channel. Those are the mean of the minimum depth, the negated maximum depth, and the approximated percentage coverage of the footprint. Among other things, the footprint is used to reconstruct the actual, non-negative mean of the maximum depth values. This method of generating the average value has the advantage of being hardware accelerated by modern graphics cards. The values all flow into the frame prediction machine learning model (see Section 3.2.2).

Being a shared resource, the depth texture can be used by the volume ray casting kernel which is implemented in OpenCL C. After an appropriate transformation, the depth values sampled from the texture are taken as starting points of the viewing rays. In the same manner, the exit points are sampled from the color channel, where the maximum depth values are stored.

The sequence of the implemented empty space skipping algorithm is visualized in Figure 3.2. Notably, the bounding geometry is only build anew if the user changes the transfer function or the threshold. If the user changes the camera position, only the depth texture has to be rendered to the back-buffer in addition to the actual ray casting.

[1]A sequence of textures in progressively lower resolutions which represent the same image. [Wil83]

Even though the implementation requires a device supporting the OpenGL rendering pipeline for maximum performance, a software renderer could also be used for these tasks. Because of the OpenCL implementation, however, one can imagine a frontend which features a device capable of hardware supported rendering while the actual ray casting is performed on another device or set of devices, such as a cluster of CPUs, for instance. Although out of the scope of this work, it is also imaginable to let an integrated GPU carry out the additional rendering steps described and use a dedicated GPU as the computing device for maximum performance.

3.2 Dynamic Frame Prediction

The goal of dynamic frame prediction is to forecast the approximated rendering time of the upcoming frame. This prediction could, for instance, be used for load balancing. It is crucial that the calculations necessary for this prediction can be done in a fraction of the time needed to perform the actual ray casting. Hence, it was tried to avoid generation of additional features for the model that take additional processing time and rather use values which are already existent during runtime or can be generated or calculated relatively fast, that means in a fraction of the time needed to compute a single frame.

To predict the calculation time of the next frame, the influencing factors have to be identified. Through the OpenCL event profiling interface, accurate kernel execution times can be measured. These times are logged during runtime and can also be exported for further processing outside of the application.

3.2.1 Performance Factors

Naturally, during an explorative volume visualization, the user interaction influences the calculation time of the ray casting algorithm.

In order to identify possible performance factors, various interaction possibilities are implemented in the prototype application. If the user changes exactly one parameter during runtime and monitors the kernel execution times - which is supported by the application through kernel execution time profiling - he or she can get an impression of the performance impact of the manipulated parameter. The values that can be modified during runtime include:

- window resolution
- camera distance

- camera rotation
- volume resolution trough down-sampling
- step size of the ray casting integration
- transfer function manipulation via a histogram interface
- noise threshold.

As illustrated in Subsection 4.1.1, especially manipulation of camera parameters and changes of the transfer function are interesting parameters which influence the runtime performance.

3.2.2 Creation of the Prediction Model

For the creation of the model, the machine learning library *scikit-learn* (`scikit-lern.org`) is used. Based on the analysis of different performance factors, a linear regression model is created. Therefore, the ordinary least squares method described in Section 2.4.1 is employed.

The quality of the model can be assessed using the coefficient of determination, R^2, score, which is described in Section 2.4.1 as well. The different attributes are tested separately and in combination to be able to estimate their impact on the generated model.

An ideal model for the purpose of real time frame prediction should be created of a small set of attributes that are either already existent in the application or can be calculated without any great effort. Furthermore, a small amount of training data should be used to keep the learning time admissible for a potential user. At loading time of a volume data set, the model can then be generated, running a short sequence of possible, random user interactions. The respective execution time of the next frame can be predicted in the fraction of the time needed to process a frame accordingly, using the model.

Camera manipulation and change of transfer function being the most commonly used interaction techniques, the prediction model focuses on those two. With that in mind, five attributes are selected to create a feasible model for ordinary least squares prediction. These attributes are:

- **Camera rotation**: the rotation of the orbital camera in x-direction and y-direction as two attributes.

- **Average depth**: the approximated average depth of the volume, which is calculated directly from the values of the highest mipmaps of the minimum and maximum depth textures that are generated from the bounding geometry (see Section 3.1.2).
- **Footprint**: the approximated footprint of the volume (percentage image coverage). It is directly taken from the depth texture of the bounding geometry as well.
- **Average transfer function alpha value**: the average alpha value of the transfer function. This value is used to gain a rough estimate of the amount of early ray termination.

Although the the regression model is currently created and evaluated using Python scripts outside the volume visualization application, a direct integration in the provided volume rendering application, should not be particularly complex. An advanced machine learning library such as *mlpack* (`www.mlpack.org`) could be included and used with the same algorithms for that purpose. However, this is outside of the scope of this work.

3.2.3 Measurement Procedure

For cross validation purposes (see Section 2.4.1), the execution times of two different runs are logged. The data of the first run is used for the creation of the model, while the second is used for its validation. Different sets of volumetric data are examined.

The runs consist of a predefined pseudo random set of possible user interactions. Those include camera rotations in x-direction and y-direction as well as zooming and changing the transfer function. Totally random changes as well as small incremental changes, that are closer to real-life user interactions, are both evaluated. To give an impression of the discrepancies in such a sequence, the rendering results of some exemplary random steps are depicted in Figure 3.3.

Due to the model creation being time critical, it is also of interest, how different amounts of input factors influence the quality of the model. In the consequence, different quantities are taken form the training data set and used for model creation, to be able to judge the impact accordingly.

3.3 System Performance Prediction

In addition to the dynamic frame time prediction, the overall performance of the volume ray casting on different systems is examined. As in the dynamic prediction, the goal is to

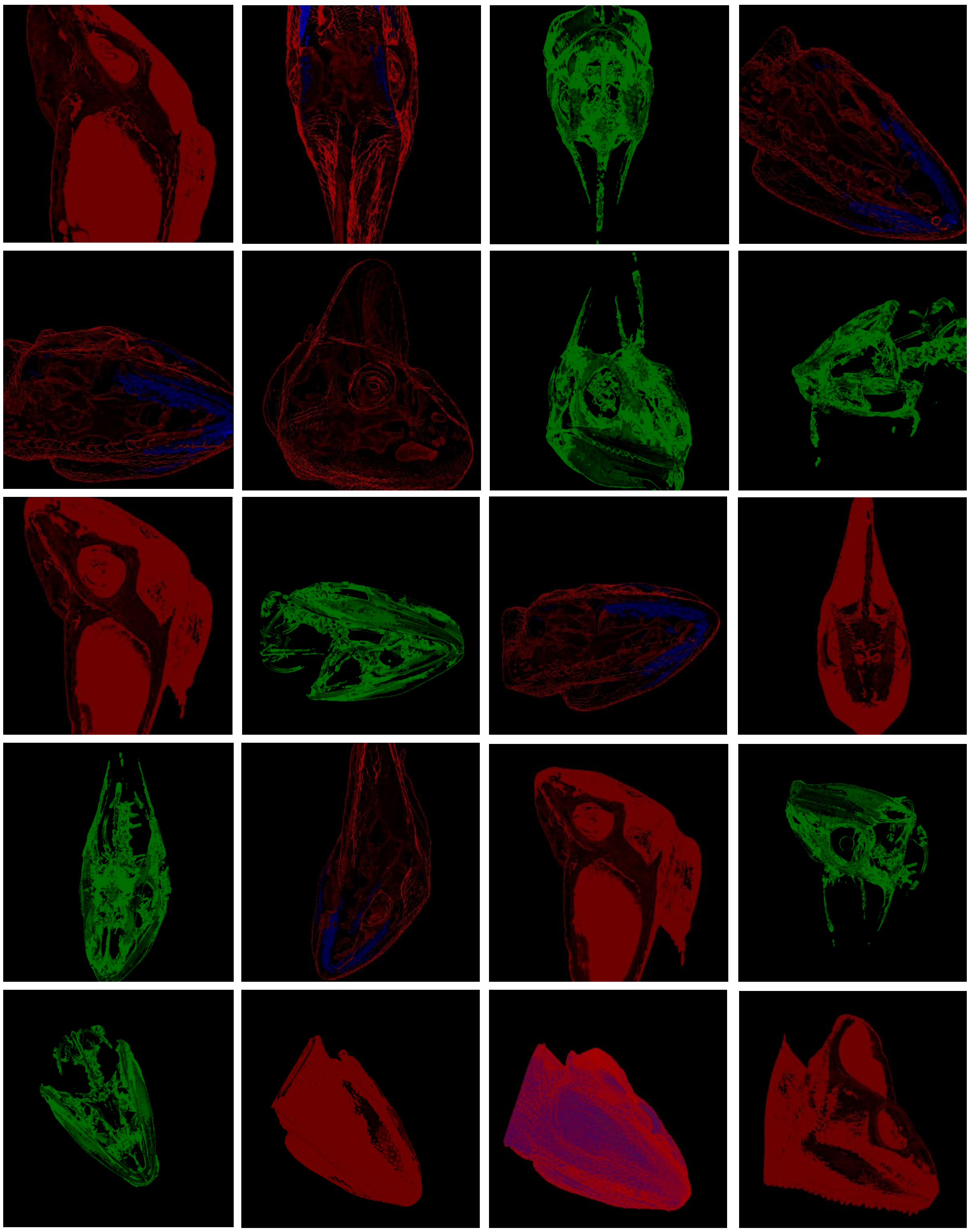


Figure 3.3: A sequence simulating random user interaction: rotation, zoom, and change of the transfer function. The data is used to train the model.

develop a, preferably general, model which can be used to predict the performance of volume visualization on current and possibly even future hardware architectures. This work focuses on the performance differences using various dedicated graphics cards. Therefor, GPUs from different vendors and recent generations of hardware architectures are examined.

Two forms of system performance prediction are investigated focusing on differing use cases. The first one deals with the case that the performance of an unevaluated volume data set shall be predicted for already tested systems. That means a machine learning model is created and trained for those systems, but with data sets of other volumes. Then, it is tried to predict the average execution time of an untested volume on those systems.

The second form deals with the case that the performance of an unevaluated system (GPU) shall be predicted for already tested volumes. A machine learning model is trained with a specific data set on different systems and employed to predict the execution time of the specific volume on the yet untested system.

3.3.1 Performance Factors

As described in Section 2.4, there are the three major factors - arithmetic operations, memory accesses, and latency - that influence the performance of a parallel application on a single GPU device. To generate a convenient machine learning model for predicting the system performance, these factors have to be evaluated. Integral to this process is especially, to identify which factors are limiting the ray casting procedure. This can be done by analyzing the ray casting kernel.

3.3.2 Analysis of the Ray Casting Kernel

There exist several tools maintained by the graphics cards vendors that can be used to analyze kernel runtime performance. CodeXL, a tool set provided by AMD, was used in version 1.9, to analyze the ray casting execution on an AMD Radeon HD 7950, the reference graphics card used during this work for implementation and testing. The GPU is based on the first *Graphics Core Next* (GCN) architecture generation and features 28 compute units and, thus, 1792 stream processors (refer to Subsection 2.3.2 for further details on GPU architectures).

After multiple executions of the kernel, which occur during normal runtime of the application, the CodeXL tool is able to provide logged data of diverse performance counters. These include kernel execution time and occupancy, but also details on memory accesses and arithmetic operations.

The mathematical operations are either processed by *vector arithmetic logic units* (VALU) or *scalar arithmetic logic units* (SALU). The average workload of those units is displayed after the kernel runtime analysis. In addition, a percentage VALU utilization is shown, indicating the thread divergence. That means, the closer it is to hundred, the less thread divergence occurs.

In terms of memory analysis, the CodeXL tool set is able to profile and display the fetches from the memory as well as the writes to it. The displayed overall percentage workload of the memory unit also includes all cache and memory effects. The cache performance is indicated with the cache hit percentage, a hundred percent indicates optimal performance. That would mean each value is accessed from the cache. Finally, a value of the extent to which the memory unit is stalled gives an indication of the latency: Zero percent implies insignificant latencies. From these performance counters, the used memory bandwidth M, too, can be derived. In order to do so, the the sum of the memory reads B_r and writes B_w is divided through the execution time t

$$M = \frac{B_r + B_w}{t} \ . \tag{3.1}$$

The performance counters of the ray casting kernel are monitored for multiple runs. The results prove that the implementation is primarily memory bound for the volumes tested on the investigated systems:

The Bonsai data set, that is specified in detail in the introduction to Chapter 4 and shown in Figure 3.1, is one of the smallest tested volume data sets in terms of resolution ($256 \times 256 \times 256$). The CodeXL analysis tool monitors a workload of more than 98% for the memory units on the HD 7950 graphics card; the latency of the memory units average below a non significant 0.2%. The ray casting of the volume was performed over the duration of several seconds, including user interactions, to get representative mean values of the performance counters. On the contrary, VALUs and SALUs are occupied below 50% respective 54% on average.

The Chameleon data set, which is detailed in the introduction to Chapter 4 as well, has one of the highest resolutions ($1024 \times 1024 \times 512$) among the volumes tested. Renderings are shown in Figure 3.3. An analysis similar to the one conducted with the Bonsai data set yields results of an average 92% memory unit occupation and less than 0.5% latency. VALUs are busy around 24%, SALUs 25% on average.

The detailed runtime information, which can be monitored, can also be used to optimize the ray casting kernel code towards a specific device. This is done exemplary with the

AMD Radeon HD 7950. The performance impact of the modified kernel is investigated for other graphics cards. The results are presented in Section 4.2.2.

3.3.3 Creation of Prediction Models

As in the dynamic frame prediction discussed in Section 3.2.2, the machine learning capabilities of the scikit-learn library are deployed to generate suitable models for system performance predictions. Two models are used for the different prediction tasks. Both make use of the ordinary least squares regression as the underlying machine learning technique.

To be able to give sufficiently good results, a model for predicting the performance of an unevaluated volume should contain features used to differentiate the systems for once, but also features specifying the data sets. The latter should not rely on runtime data because of the requirement to predict unknown volume data sets. Therefore, only such attributes which are directly visible from a definition file are used for the creation of the model. The investigated features include the closely correlated resolution, size of the data file, and precision of the data values.

Because of the great differences in resolution caused by the cubic nature of this parameter, a logarithmic function is applied to the resolutions during testing and the results investigated. The same is done with the file size. Measurements presented in Section 4.2.3 indicate that a logarithmic function of the file size as a model feature describing a volume data set gives the best results.

Due to the fact that the kernel is memory bound, as described in Section 3.3.2, the memory bandwidth of the GPU is used as the perhaps most important model feature. To give an indication of the overall architectural performance, the theoretical computational performance (given in flops by the vendor) as well as the amount of stream processors (see Section 2.3.2) are investigated as model features. As can be seen in Section 4.2.3, the number of stream processors generates the best results.

A similar yet not equal model is created for predicting the execution time of ray casting a volume on an unevaluated system. Training data is used from rendering the same volume data on other systems. That means, any attributes describing the data set are not needed for the model.

The memory bandwidth is used as the basic feature of the model because of its importance described in the previous subsection. The number of stream processors and the theoretical computational performance are examined as well. Since it generates the best results, as shown in Section 4.2.4, the former is used with a logarithmic function applied.

4 Measurements and Results

This chapter covers the results, analysis and interpretations of the execution time measurements that have been conducted during this work. It is divided into two main sections, the first one (Section 4.1) focusing on the results of the dynamic real time frame prediction, while in Section 4.2, two models for system performance prediction are discussed.

Dynamic frame prediction (Subsection 4.1.2), in this context, tries to predict the execution time needed for the ray casting computations of the upcoming frame. The prediction is thereby based on user interactions such as camera transformations and change of the transfer function.

In addition, the results of two forms of system performance prediction are analyzed. In the first phase, the average execution times on several different, but known GPUs for an unknown volume data set were predicted. The results of the developed model are discussed in Subsection 4.2.3. Different volume data sets were tested on those systems to train the model beforehand. In a second phase of system performance prediction, the execution time of a known volume on an unknown system was predicted. The volume to be evaluated was, therefore, tested on various GPUs and a model was trained based on those results. Subsection 4.2.4 summarizes the outcomes.

This chapter concludes with a brief discussion of the limitations of the presented methods in Section 4.3. Possible solutions to overcome those limitations are presented accordingly.

All data sets that were used for either the dynamic frame prediction or the system testing are listed in Table 4.1. The corresponding resolutions and precision of the data values are also specified there. Renderings of the volumes which are not shown in Figure 4.1 are pictured in Figure 4.6.

The determining factor for the selection of volumes was their fitness for use with the developed application on the hardware used for testing. Problems caused by volume data sets of bigger size can include driver errors or incorrect renderings. This manifests in missing parts or visible artifacts and occurs especially on GPUs with smaller video random access memory (VRAM) sizes.

Volume Name	Width	Height	Depth	Size [MVoxel]	Bit	Courtesy
Engine	256	256	128	8.39	8	General Electric
Bonsai	256	256	256	16.78	8	S. Roettger
Supernova	432	432	432	80.62	8	John M. Blondin
Stanford Bunny	512	512	361	94.63	8	Terry S. Yoo
Matamata	512	512	400	104.86	8	UTCT[1]
Vertebra	512	512	512	134.22	8	Phillips Research
Foraminifera	1024	1024	219	229.64	16	UTCT
Chameleon	1024	1024	512	536.87	16	UTCT
Woodpecker	1024	1024	515	540.02	8	UTCT

Table 4.1: The volume data sets used for measurements. Sizes are given in voxels and rounded to two decimal places.

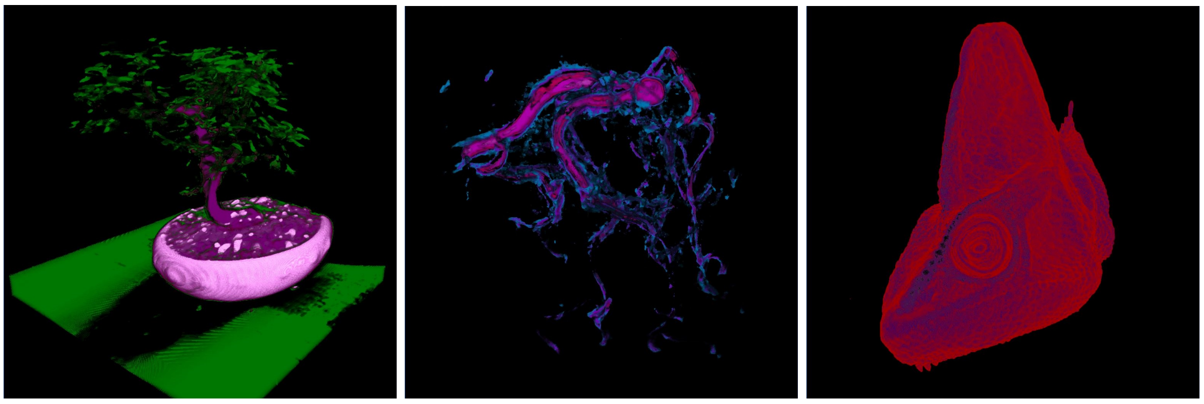


Figure 4.1: Renderings of the test volume subset used for dynamic frame prediction. The corresponding names from left to right: Bonsai, Vertebra, and Chameleon.

4.1 Dynamic Frame Prediction

This section deals with the results of the dynamic frame time prediction that should, per requirement, be real-time capable. First, the impact of different factors on the performance is shown. In Subsection 4.1.2, different features for the machine learning model are evaluated.

Although various different volume data sets have been measured and investigated during the course of this work, the three test sets shown in Figure 4.1 have been selected for

[1]University of Texas High-Resolution X-ray Computed Tomography Facility (www.ctlab.geo.utexas.edu)

presentation of results in this section. The volumes were specifically chosen because of their distinct differences in size, form and resolution. A short overview of the properties of these data sets is presented in Table 4.1.

4.1.1 Performance Factor Measurements

For a basic understanding of performance factors in a volume rendering application, the impact of interactively manipulable values on different data sets has been investigated. First of all, the factors presented in Section 3.2.1 have been examined without any acceleration techniques enabled. A selection of these results is depicted in the diagrams in Figure 4.2. To avoid distortion of the results caused by outliers, the kernel execution times of each configuration have been logged over a time period of one second. From these logs, the median has been taken and plotted against the y-axes. The x-axes feature the altered values, which are labeled under the corresponding graph.

As can be seen in Figure 4.2, some of these performance factors influence the execution time in approximately linear ways. These factors include image resolution, volume resolution, which was tested through down-sampling of the volume data sets, and integration step size. The latter is specified as a factor of the voxel depth.

Other interaction parameters, however, have a seemingly unpredictable impact on the execution times. This holds true especially for camera rotation and distance, as can be seen in Figure 4.2. The addition of acceleration techniques as presented in Section 2.2.2 seems to leads to more arbitrary results.

As a conclusion, an advanced approach is necessary in order to gain feasible prediction results. Since changes of camera position and transfer functions are the most commonly used user interactions and their performance impact is comparably unpredictable, the prediction model has been developed with focus on those parameters.

4.1.2 Model Features

As discussed in Section 3.2.2, the developed model features five attributes. These attributes have been tested for their impact on the prediction quality by evaluating the R^2 scores (R^2 is the coefficient of determination). The testing procedure is described in further detail in Section 3.2.3.

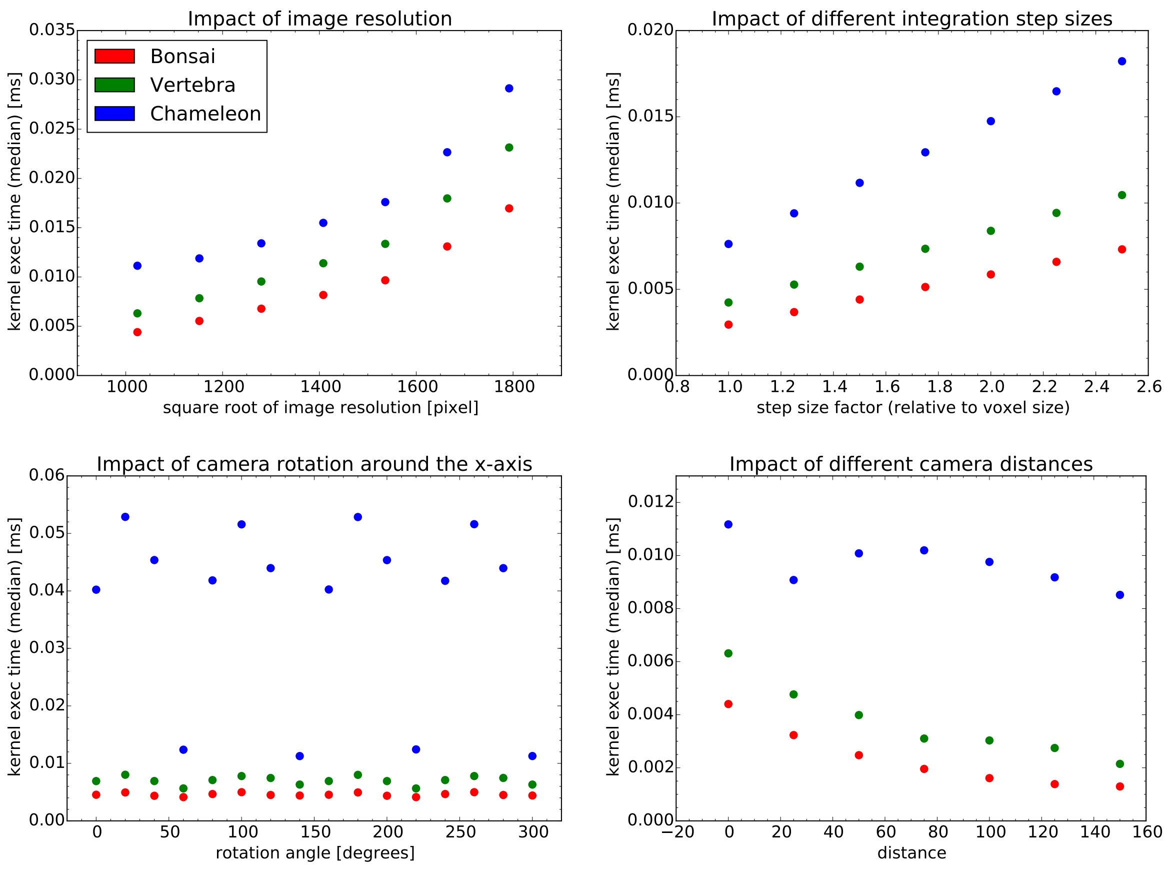


Figure 4.2: Impact on performance for different interactions without acceleration techniques: image resolution (top left), integration step size (top right), rotation around the x-axis (bottom left), and zoom (bottom right).

Early Ray Termination and Transfer Function Feature

To be able to judge the impact of the early ray termination (ERT) acceleration technique, the model performance has been evaluated both with ERT disabled and enabled. Furthermore, the average transfer function alpha value as a model attribute has been toggled during this test since it is used as the classifier for the estimated ERT impact. By adding this feature to the model and comparing the results, the quality improvements for the model can be derived. The results of the ERT specific test runs are listed in Table 4.2.

As discussed in Section 2.4.1, an R^2 score of 1.0 implies a perfect match of the predicted and measured values, while the lower the score, the worse is the overall prediction. R^2 scores of the training data set (labeled *train*) as well as the test data set (labeled *test*) are given in Table 4.2. As mentioned in Section 3.2.3, separate data sets are used

	without ERT		**with ERT**		**ERT, α-attribute**	
	train	**test**	**train**	**test**	**train**	**test**
Bonsai	0.99	0.97	0.91	0.76	0.90	0.82
Vertebra	0.98	0.70	0.98	0.70	0.99	0.97
Chameleon	0.85	0.51	0.15	-0.11	0.68	0.62

Table 4.2: Impact of early ray termination: R^2 scores of the results from training data and test data. The rightmost column shown the scores when using the average transparency of the transfer function as a model attribute.

excluded	**depth**		**footprint**		**depth/footp.**		**rotation**		**none**	
	train	**test**	**train**	**test**	**train**	**test**	**train**	**test**	**train**	**test**
Bonsai	0.81	0.73	0.91	0.80	0.21	-0.05	0.91	0.76	0.90	0.82
Vertebra	0.99	0.95	0.97	0.94	0.28	-0.02	0.99	0.97	0.99	0.97
Chamel.	0.63	-0.44	0.59	-0.25	0.51	-0.31	0.63	-0.47	0.68	0.62

Table 4.3: Impact of excluding a single feature at a time from the model: R^2 scores of the results from training data and test data are shown as indicators of the model quality.

for cross validation purposes. The R^2 score of the test data can be considered more suitable for the estimation of the model performance as the score of the training data judges the prediction performance only based on the training data itself. Consequently, those results are generally higher than the ones from the test data sets. All R^2 scores are rounded to two decimal places.

The data in Table 4.2 reveals how ERT has a high impact on the prediction results. This can be claimed especially for volumetric data sets of great sizes. Adding average transfer function values as a feature to the machine learning model improves the predictions significantly and can, therefore, be judged as a decent value for factoring ERT into the prediction model.

Depth, Footprint, and Rotation Features

To assess the impact of each feature of the model, all of them have been excluded successively from the model creation in numerous combinations. The various models have been tested on the data gained from the test runs of the three models. The results are summarized in Table 4.3.

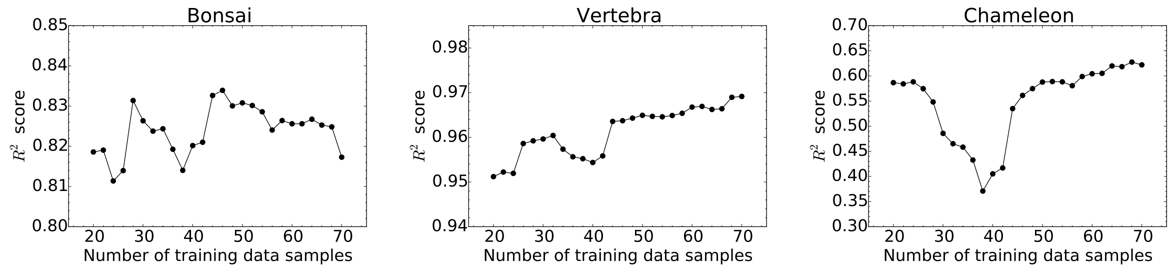


Figure 4.3: Impact of the training sample size on the R^2 scores of the predictions for the Bonsai, Vertebra and Chameleon data sets.

If the features footprint and average depth are excluded from the model, the results worsen the most. Leaving out either one of them solely, however, the impact is not as great. This phenomenon is probably caused by a correlation between those two features. The Bonsai volume is impacted least by the footprint, followed by the rotation angles. This order changes for the Chameleon data set, where the footprint has a bigger impact.

Excluding footprint, rotation, or average depth features from the Vertebra model creation does merely affect the outcome marginally, if at all. Interestingly, for the Chameleon data set, which is the biggest of the three, decent results for the R^2 score of the test data set are only achieved if all features are used. One can conclude that, depending on size and shape of the data set, all chosen features contribute to the improvement of the model.

Training Sample Size

The influence of the training data input size has been investigated as well. Therefore, increasing amounts of samples from the same data set (for each volume respectively) have been used to train the model. A selection of the numerical results of the R^2 score as a function of the training sample set size are shown in Table 4.4. A visual representation is given in Figure 4.3. Attention is invited to the fact that the linear interpolation between the discrete sampling points are just for visual guidance and shall by no means indicate a linear course between the samples. Also, the y-axis only shows a relevant extract which varies for each data set.

As can be seen, an increase of the sample size from 20 up to 70 has little impact on the Bonsai and Vertebra data sets. However, the model for the Chameleon volume improves visibly with the usage of more training data. This can probably be explained by the higher size of this volume.

number of training samples	**30**		**50**		**70**	
	train	**test**	**train**	**test**	**train**	**test**
Bonsai	0.91	0.83	0.92	0.83	0.92	0.82
Vertebra	0.99	0.96	0.99	0.96	0.99	0.97
Chameleon	0.68	0.49	0.62	0.59	0.63	0.62

Table 4.4: Impact of training data sample size on the R^2 scores. Scores are shown for the training data as well as test data.

Interestingly, the models of all three data sets show a decrease in score around the 40 samples mark. Whereas this drop in model performance is relatively low for the Bonsai and Vertebra data sets, it is huge for the Chameleon. The combination of configurations in the pseudo random training sequence at the 40 samples mark seem to distort the results of the model. As can be seen in Figure 4.4, a plot of the execution and prediction times, sequences with a transfer function that results in a rendering of only comparably small parts of the volumes are predicted badly. It should be pointed out that the linear interpolation between the discrete, measured points in the diagram is only used to give a guide for the eye. They shall by no means indicate linear transitions between the various configurations. In these special parts, the predicted execution times are generally very low; around 40 training samples, they even fall into the negative. It goes without saying that this last scenario is not realistic. A constraint guaranteeing positive prediction values could lead to better R^2 scores in this case. However, with the employed scikit-learn library, such a restriction is not usable on the ordinary least squares method without further ado. Increasing the number of training samples helps to smooth out the curve and so improves the the results.

A use of more than 70 data samples generally does not seem to improve the model performance significantly for any of the data sets tested. Accordingly, a training data size of 70 was used for all measurements.

4.1.3 Random Interactions Compared to Realistic Use Cases

The former tests have all been conducted with a simulated set of pseudo random user interactions. This ensures that even extreme interactions such as abrupt parameter changes, are included and can be handled by the model. Another reason for this approach is that in the context of load balancing, these rather unusual interactions are among the most interesting scenarios. So, with good predictions made for those cases, huge increases or decreases of performance can be evened out before they even occur.

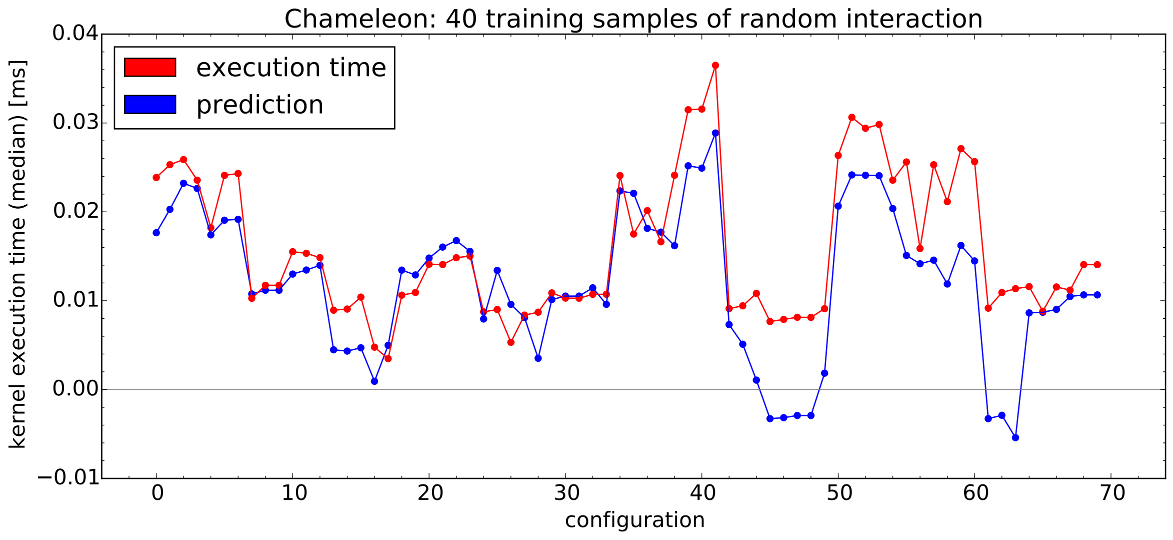


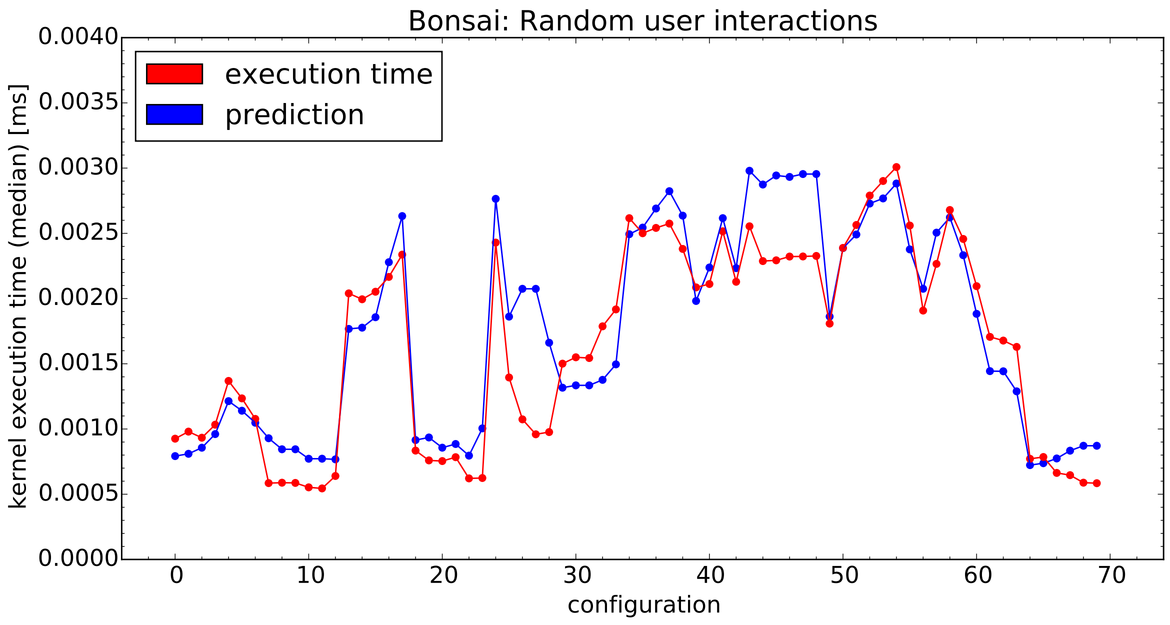
Figure 4.4: Execution time (red) and prediction (blue) of a random sequence of different configurations of the chameleon data set, using 40 training samples.

	random test data		**simulated use case**	
	train	**test**	**train**	**test**
Bonsai	0.90	0.82	0.92	0.96
Vertebra	0.99	0.97	0.99	0.91
Chameleon	0.68	0.62	0.75	0.67

Table 4.5: Random test data in comparison to a simulated realistic use case. Performance of the training and test data are shown in the form of R^2 scores.

Small changes, on the contrary, normally do not have that big of an impact on the overall performance. They rather manifest in modest execution time changes than in big leaps. Realistic use, however, usually includes this kind of small parameter changes. Consequently, it seemed important to investigate whether the developed model can be used to make decent predictions for such small changes as well. The R^2 score results gained from the prediction of a simulated "realistic" use case have been summarized in Table 4.5.

Figure 4.5 examplary shows the differences for the Bonsai data set. The plots of the other two data sets have likewise characteristics. The y-axes represent the execution time of the kernel, the x-axes the measured configuration, which, in this case, means 70 different configurations with random interactions and 144 different configurations with a simulated user interaction. Small incremental changes in camera position as well



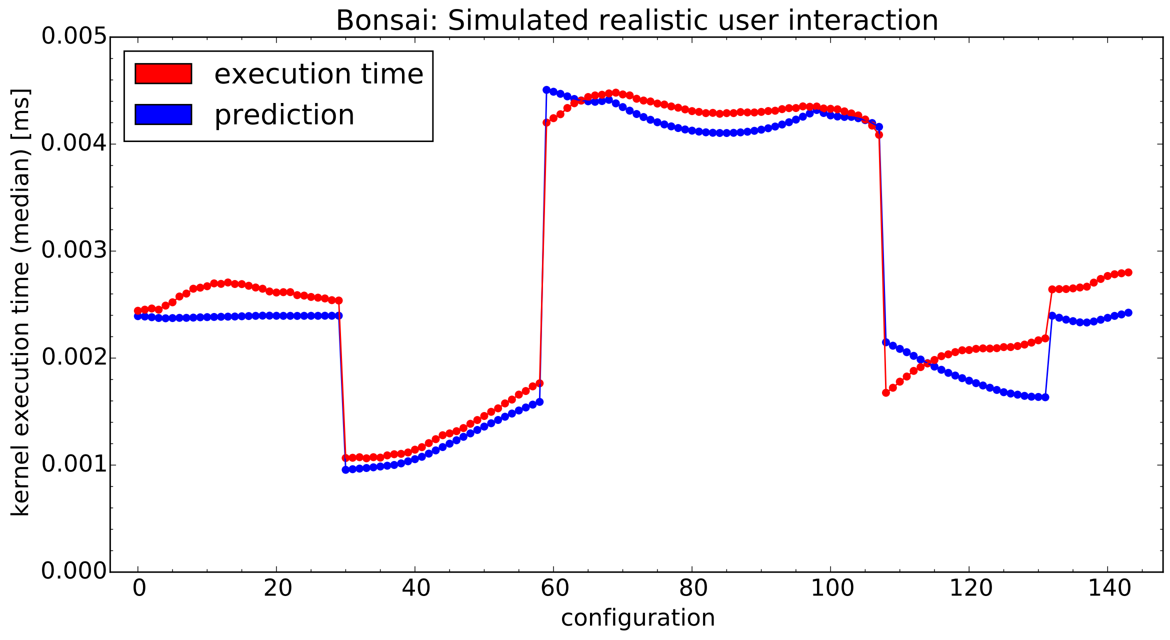


Figure 4.5: Comparison of a random interaction test set (top) and a simulated realistic use case (bottom). Shown are execution times in red and predictions in blue. The connecting lines between the points are intended as guide for the eye and shall not indicate linear dependencies between configurations.

as the loading of different transfer function were used to imitate a realistic use case. The blue dots depict the predicted values, while the red ones show the medians of the actual measured execution times for the same configuration respectively. The connecting lines between the dots are intended to give a visual guidance for the eye and shall not indicate a linear transition within the configuration.

The big leaps in execution time are changes to the transfer function, resulting in a changed bounding geometry. Thereby, they impact the empty space acceleration. This causes a major change in the processing time needed for the ray casting.

As the findings indicate, small incremental camera changes are often not sufficiently covered by the model. Big leaps in performance, however, are usually predicted properly. The bad performance on small increments could be facilitated by single floating point precision, which was used because of hardware limitations, as well as rounding errors that can be associated with this limitation.

What is more, in the section around the 120^{th} configuration in the simulated use case, the model even predicts a wrong trend for the incremental camera change. This is probably caused by unfavorable training data for this specific case.

A possible explanation for the behavior described above is that the training data set is still a random set of huge parameter changes compared to the modest changes in the simulated use case. That means, the model was primarily trained for those bigger changes. This, however, fulfills the intention of this work since the prediction of major leaps in performance is of high importance in regard to load balancing scenarios.

Overall, the model gives a fairly good prediction of the execution time that is to be expected. This especially holds true the case of major changes which are relevant for load balancing, for example.

4.2 System Performance Prediction

This Section deals with the overall system performance prediction. In this work, the focus lies on predicting the performance on different dedicated graphics cards. The different GPUs that were used for measurements are described in Section 4.2.1 in further detail.

In Section 4.2.3, the results of predicting the average performance on an unknown volume for diverse, known systems are discussed. Section 4.2.4 summarizes the outcome of the performance predictions of a known volume on an untested system.

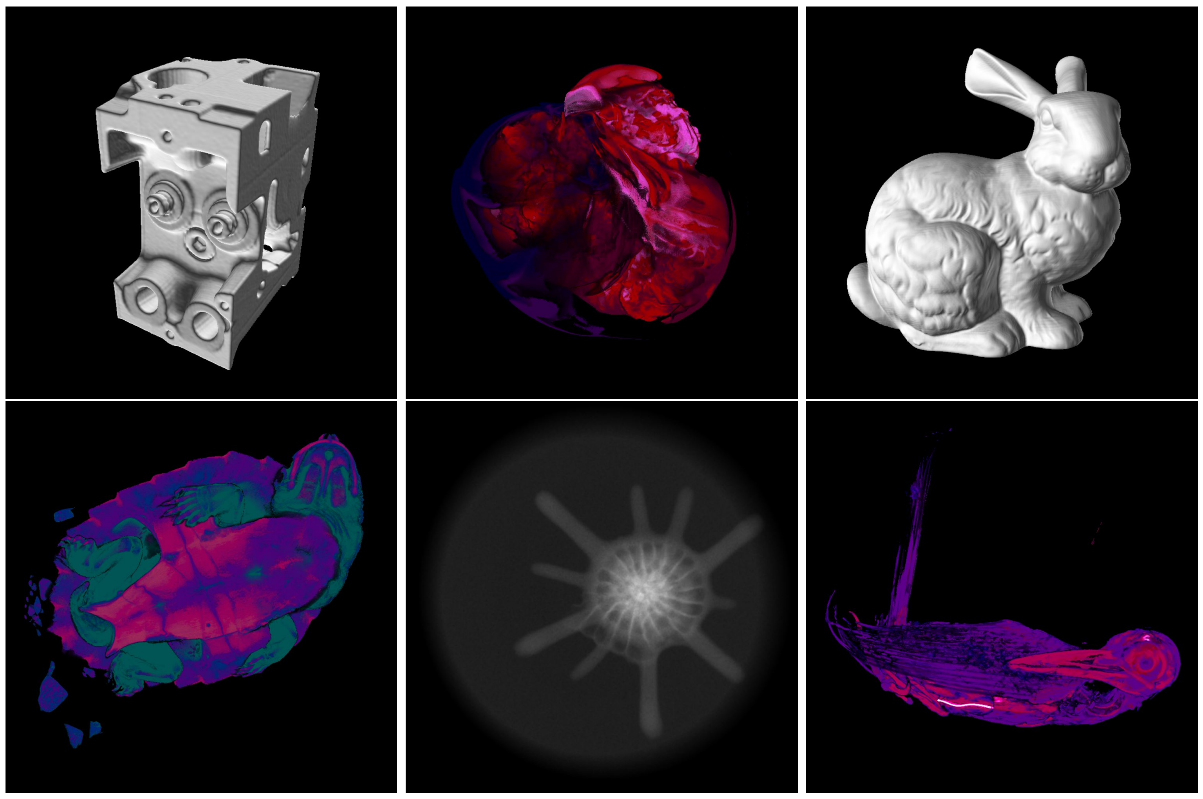


Figure 4.6: Volume data sets used for system tests, order by resolution from lowest to highest and rendered with different techniques. From left to right: Engine, time step of a simulated Supernova, Stanford Bunny (top row); Matamata, Foraminifera, and Woodpecker (bottom row). Specifications can be found in Table 4.1.

4.2.1 Systems used for Testing

A heterogeneous selection of GPUs of different hardware generations and vendors is used to investigate the system performance prediction. Because of limited access, however, only eight respective nine distinct graphics cards could be used for testing. The volume data sets chosen for the investigation are shown in Figures 4.6 and 4.1.

A listing of the different dedicated graphics cards used for measurements is given in Table 4.6. Relevant parameters of the hardware and their theoretical capabilities are specified therein. All GPUs support OpenCL 1.2, which is used in the volume rendering application. Discrete GPUs from NVIDIA as well as AMD are investigated.

Id	Vendor	Name	GPU Core	Architecture	VRAM [GB]	Bandwidth [GB/s]	GFLOPS	Stream Processors
1	NVIDIA	GeForce GTX 760	GK104	Kepler Refresh	2	192.26	2258	1152
2	NVIDIA	GeForce GTX 660	GK106	Kepler	2	144.19	1881.6	960
3	NVIDIA	GeForce GTX 660 Ti	GK104-300	Kepler	2	144.19	2460	1344
4	NVIDIA	GeForce GTX 680	GK104-400	Kepler	4	192.26	3090.4	1536
5	NVIDIA	GeForce GTX Titan	GK110	Kepler Refresh	6	288.38	4500.0	2688
6	AMD	Radeon R9 290	Hawai PRO	GCN 1.1	4	320.00	4848.6	2816
7	NVIDIA	Quadro 6000	GF100GL	Fermi	4	143.42	1030.4	448
8	NVIDIA	GeForce GTX 560 Ti	GF114	Fermi Refresh	1.28	152.00	1263.4	384
9	AMD	Radeon HD 7950	Tahiti PRO2	GCN 1.0	3	240.00	3315.2	1792

Table 4.6: The different GPUs used for testing and a selection of their vendor provided specifications.

4.2.2 Improving GPU Occupancy

For a first implementation of a ray casting kernel, the CodeXL tool set described in Section 3.3.2 estimates an average occupancy (number of running threads) of about 40% on the HD 7950. This is caused mainly by the usage of too many vector general purpose registers (VGPR), i.e. their number exceeds the number of registers available per work item. The consequence is register spilling into the local memory which is slower so delays can occur.

In an attempt to optimize the occupancy, the number of VGPRs used per work item were reduced. This was done by templating the kernel for each rendering technique (see 2.2.1), meaning that if the user changes the mode in the GUI, a specific kernel version is loaded which only supports this very rendering technique but is ,therefore, less complex. Adding some minor code adaptions, manly concerning variable usage, the estimated kernel occupancy could be improved to about 90% for the transfer function mode.

To receive an impression of the real performance improvement gained by this optimization, the execution time for several volume data sets were investigated. This has been done for various GPUs to be able to judge the impact on different architectures. The results are shown in Figure 4.7. Six volume data sets of different resolutions have been tested. For all of them, the diagram shows the execution time difference relative to the version without kernel occupancy improvements. Details on the tested systems and volume data sets are given in Tables 4.1 and 4.6.

As the graph depicts, the performance improvements range from small (below 10% for the Vertebra volume) to significant (more than 60% for the Matamata data set) on the reference GPU (AMD Radeon HD 7950). Although they are mainly positive, the performance changes vary on the other GPUs. The AMD R9 290, for instance, has an architecture which is closely related to the one used in the HD 7950, so the noticeable improvements for all six volumes are expectedly good. Improvements were also found for most of the NVIDIA GPUs, though those are not as huge on the graphics cards of older generations (Fermi and Kepler) and there is even a performance loss on the GTX 660. This is probably caused by bad memory access patterns or caching. Interestingly, the Vertebra data set performs worse on some of the tested NVIDIA GPUs. This could be caused by high latencies which again could be the result of unfavorable memory access patterns.

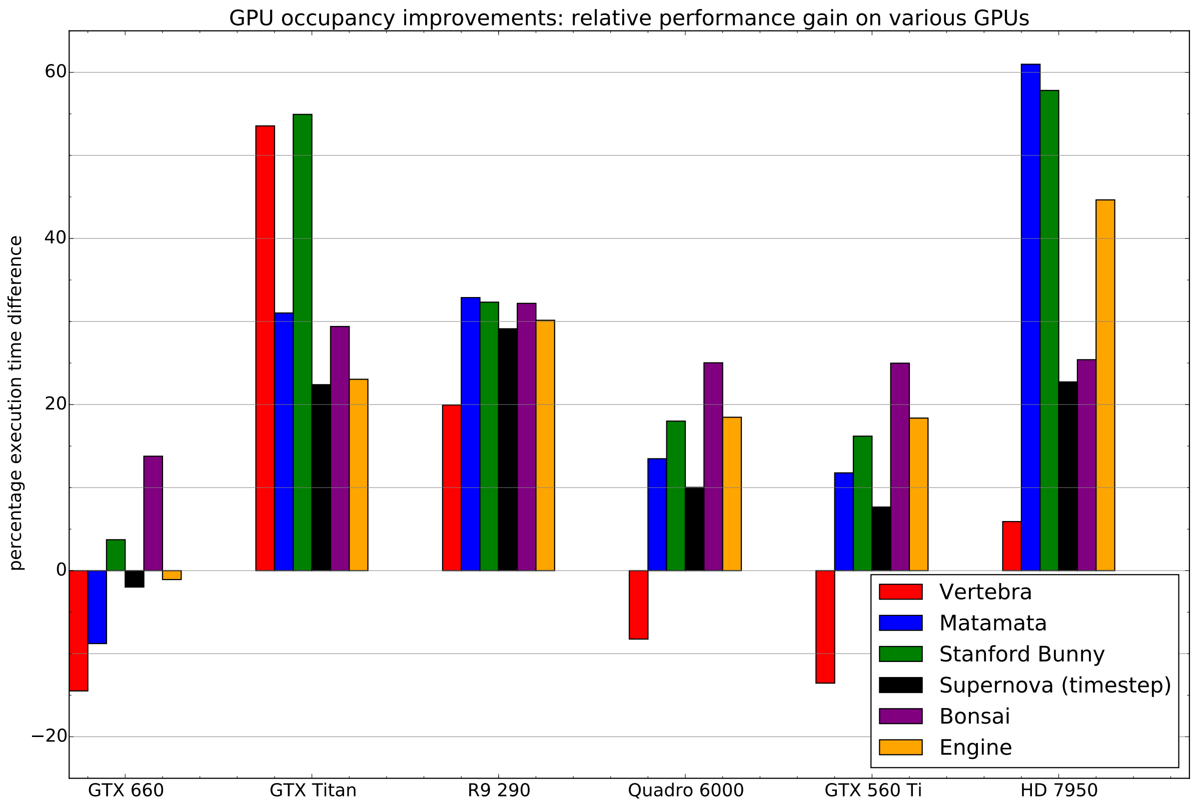


Figure 4.7: Impact of the GPU occupancy improvements on the performance of different systems and volume data sets. The bars represent the percentage improvements of the execution times compared to using the unimproved version.

4.2.3 Predicting the Performance of an Unevaluated Volume

This section deals with the question of how to predict the performance for a data set which has not been tested yet. The volumetric data used for this work is represented in two files: one containing measured or simulated data in a raw format, the other one containing information about the structure in the form of three-dimensional resolution, the slice thickness, and the format of the raw data.

The resolution contained in the descriptive information file is existent before actually rendering the volume and can, thus, be used in the model to predict the performance for the test systems. For hardware specification, memory bandwidth as well as the number of stream processors prove to be the best features among the tested and were, therefore, incorporated in the machine learning model.

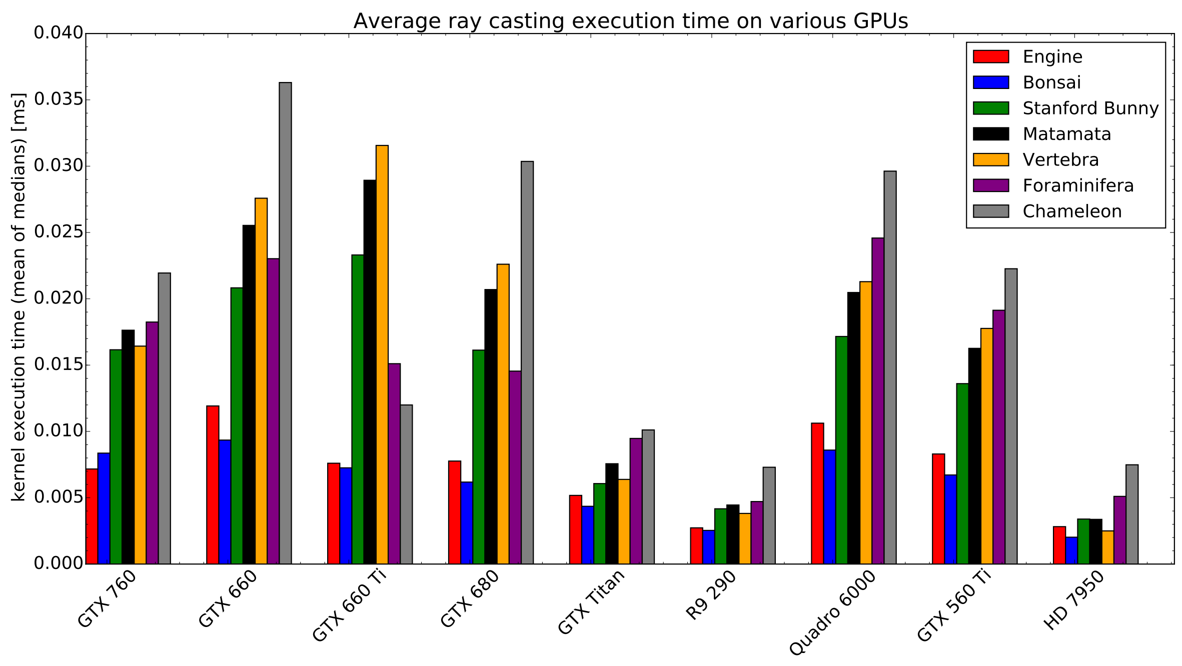


Figure 4.8: Average ray casting execution times for various GPUs. The measured volumes are sorted based on their resolution.

Average Execution Times

First, the model is trained with data gained from measurements of renderings of various data sets on all systems. That way, a sequence of different configurations is prepared and successively loaded on all test systems. These configurations include various camera positions and transfer functions: parameters significantly influencing the ray casting performance. Since the same sequence is used on all systems, this procedure guarantees comparability. All configurations are rendered for a period of half a second to assure multiple, complete kernel runs. The median of the logged kernel execution times is taken, to not incorporate outliers that can result from other tasks being requested and processed by the GPU during runtime of the volume rendering application. The mean of all medians of the various configurations is calculated to represent an average computation time for the tested volume.

Figure 4.8 depicts the average execution times of all volumes tested. The bars are ordered based on the resolution of the volume they are representing, the smallest being the leftmost. The order of the GPUs is random and has no meaning. One may expect that for higher resolutions, the execution times will rise accordingly. However, that is not the case for all configurations as can be seen in Figure 4.8. This circumstance complicates a prediction using only the resolution as model feature.

Features	**Resolution**		**Size**		**log(Size)**		**log(Res.) log(Size)**	
	train	**test**	**train**	**test**	**train**	**train**	**train**	**test**
Engine	0.76	-1.37	0.73	-2.14	0.79	-1.05	0.81	-2.77
Bonsai	0.78	-5.11	0.76	-6.43	0.80	-3.35	0.81	-3.14
Bunny	0.74	0.81	0.70	0.79	0.77	0.78	0.78	0.78
Matamata	0.75	0.62	0.72	0.55	0.78	0.83	0.81	0.81
Vertebra	0.75	0.67	0.72	0.58	0.78	0.77	0.79	0.79
Foraminifera	0.74	0.75	0.70	0.75	0.80	0.56	0.58	0.58
Chameleon	0.73	-0.13	0.65	0.31	0.75	0.67	0.61	0.61

Table 4.7: Impact of different features specifying the volume data set.

Model Features Specifying the Volume Data

Subsequently, a prediction of the mean execution time of a test volume - which has not been used as training data for the model - was made. The ordinary least squares method discussed in Section 2.4.2 was used as a supervised learning method. The R^2 score is employed to judge the quality of the model, a score of 1.0 indicates a perfect match of the prediction and the actual measurement while smaller numbers signify a worse performance of the trained model.

Different features have been tested for model generation. The ones defining the GPU are memory bandwidth, number of cores, and theoretical computational performance. Because of the requirement of predicting the performance before actually rendering the data set, the possible attributes specifying the volume are limited. Those investigated were the correlated resolution and file size. A logarithmic function has been employed on those two features and the impact examined. The R^2 scores of the volume data feature tests are listed in Table 4.7. All of the tests shown in Table 4.7 have been done using the bandwidth and number of cores as system defining features because with those the best overall model performances could be archived while rendering the tested data sets as is shown below.

As the values indicate, the size results in a better performance than the resolution for the tested data sets, albeit being close. Applying the logarithmic function improves those results further. An application of both the size and the resolution, both modified with the logarithmic function, hardly impacts the R^2 scores to the better. A possible explanation for this is the close relation between those two parameters.

Features	**Bandwidth, Flops**		**Bandwidth, Cores**		**Bandwidth, Cores, Flops**		**Bandwidth, Cores, Fill**	
	train	**test**	**train**	**test**	**train**	**train**	**train**	**test**
Engine	0.77	−1.09	0.79	−1.05	0.79	−0.99	0.81	−2.77
Bonsai	0.78	−3.35	0.80	−3.35	0.81	−3.28	0.81	−3.14
Bunny	0.76	0.81	0.77	0.78	0.78	0.84	0.78	0.78
Matamata	0.76	0.75	0.78	0.83	0.78	0.78	0.81	0.81
Vertebra	0.76	0.52	0.78	0.77	0.78	0.76	0.79	0.79
Foraminifera	0.78	0.52	0.80	0.56	0.80	0.58	0.58	0.58
Chameleon	0.74	0.65	0.75	0.67	0.76	0.67	0.61	0.61

Table 4.8: Impact of different features specifying the GPUs.

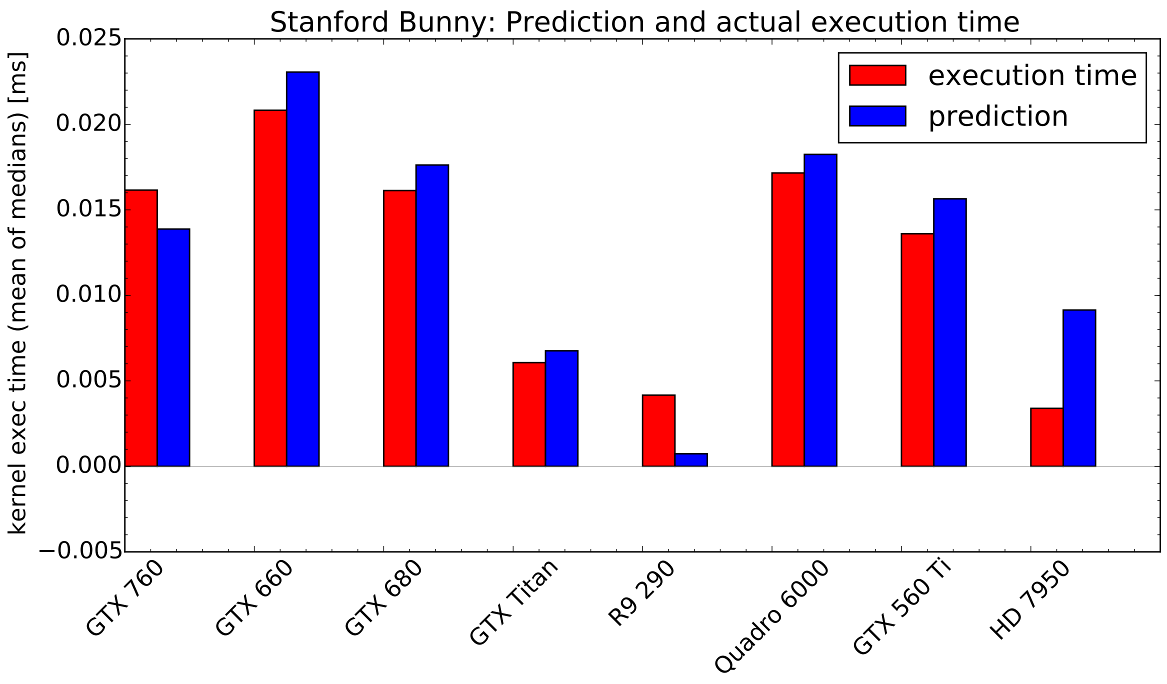
Model Features Specifying the GPUs

Table 4.8 lists the R^2 scores of measurements made to investigate the different model features defining GPUs which are used as given by the vendors: memory bandwidth, number of cores, and theoretical computational performance in flops. Since the application of a logarithmic function to the size specification of the volume proved effective before (see Table 4.7), it was used again to define the volume during these tests.

With some processing, the average empty space of the tested volumes, i.e. the percentage amount of values below a certain threshold, has been calculated and also used as a model attribute, further referred to as *fill* feature. With regard to the empty space leaping acceleration method used in the ray casting application, a model improvement through this percentage fill attribute could be expected. As can be taken from Table 4.8, the additional percentage fill feature does not bring any significant improvements to the model and can, thus, be left out to avoid the additional preprocessing.

Including the number of cores leads to a better performing model than including the theoretical computation attribute. Using both seems to have no significant impact on the results for the tested volumes. The models generally perform badly for the two smallest data sets (Bonsai and Engine). Figure 4.9 shows the predicted (blue) and the measured (red) average execution times of the Stanford Bunny data set and the Bonsai data set exemplary.

As illustrated by the diagrams, the model overestimates or underestimates the execution times of the Bonsai data set constantly. The same happens with the Engine data set. The prediction tendency is right, but the values are either too big or too small. This could be caused by a deficiency of training volumes used of a similar size and resolution. As



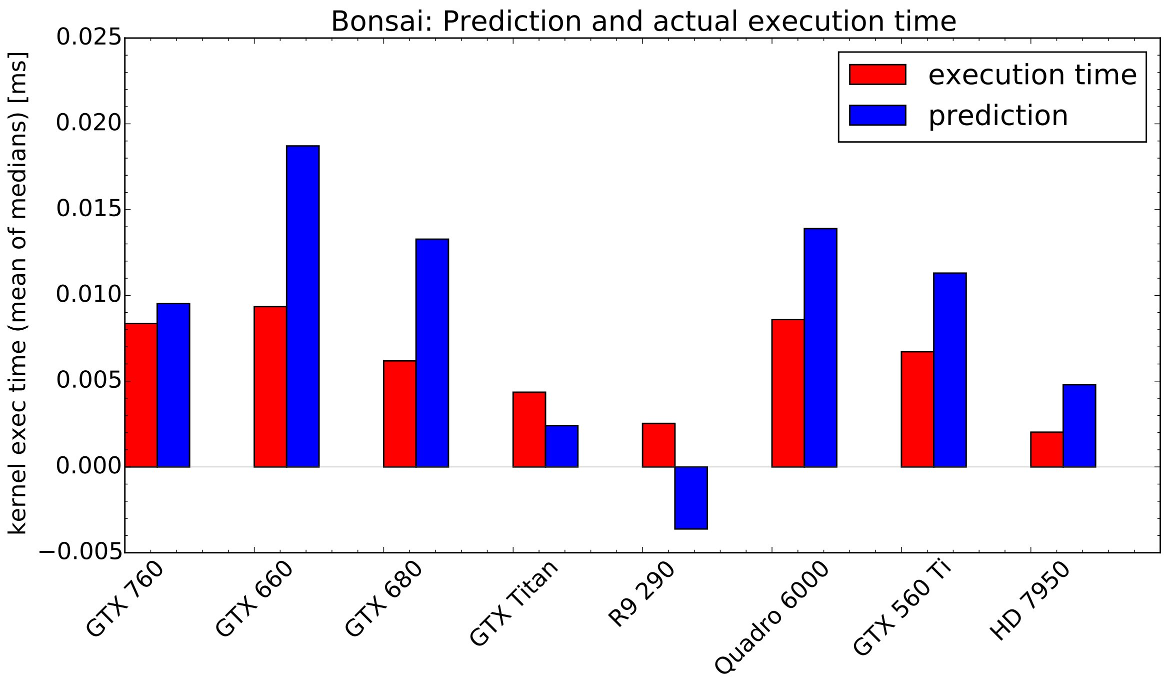


Figure 4.9: Comparison of prediction performance of Stanford Bunny (top) and the Bonsai data set (bottom) on different GPUs. The prediction of the Bunny data set is quite accurate, while the one for the Bonsai overestimates respectively underestimates the execution time for almost all systems.

can be seen in the diagram for the ray casting times used for the Bunny data set, the results are pretty accurate for this volume. This is probably the case for several volumes with similar resolutions and sizes were included in the training data sets, as shown in Table 4.1.

Furthermore, in the diagram of the Bonsai data, the execution time of the AMD R9 290 is actually predicted below 0.0 - a result which is not possible. However, with the scikit-learn machine learning library, that was used for the model creation, it is not possible to restrict the model prediction to positive numbers with the used ordinary least squares linear regression model. Supported linear regression models which feature such a restriction are variants of the LASSO[2] method, for example. Lasso is a regression analysis method that performs regularization with an $l1$ penalty term in addition to variable selection [Tib96]. Yet, the results of the Lasso method were much worse that with linear regression through OLS. Incorporating a non-negative constrained into the linear regression model could enable better predictions for small volumes.

4.2.4 Predicting the Performance of an Unevaluated System

In this subsection, the results of a machine learning process that deals with the prediction of the average execution time of a volume data set on a unknown system, are presented. The model is trained with GPU specifications and data gained from ray casting measurements on other systems. As for the other prediction variants, the ordinary least squares linear regression method (see Section 2.4.2) is used for model generation. The GPUs in the systems that were used for testing as well as the volume data sets are all listed in Section 4.2.1.

As explained in Section 4.2.3, the average execution time is calculated as a mean of the medians of different test configurations. Those configurations are the same for all systems and are rendered over a short period of time to gain multiple execution times and be able to exclude outliers.

Exactly one volume is predicted for one system at a time, i.e. the execution time is predicted for one system and the actually measured times used on the other eight systems as training data for their predictions . The evaluation of the model is done by calculating the relative error of the prediction, which is the percentage difference between the measurement and the prediction on the test system, where the measurement serves as reference.

[2]Least Absolute Shrinkage and Selection Operator

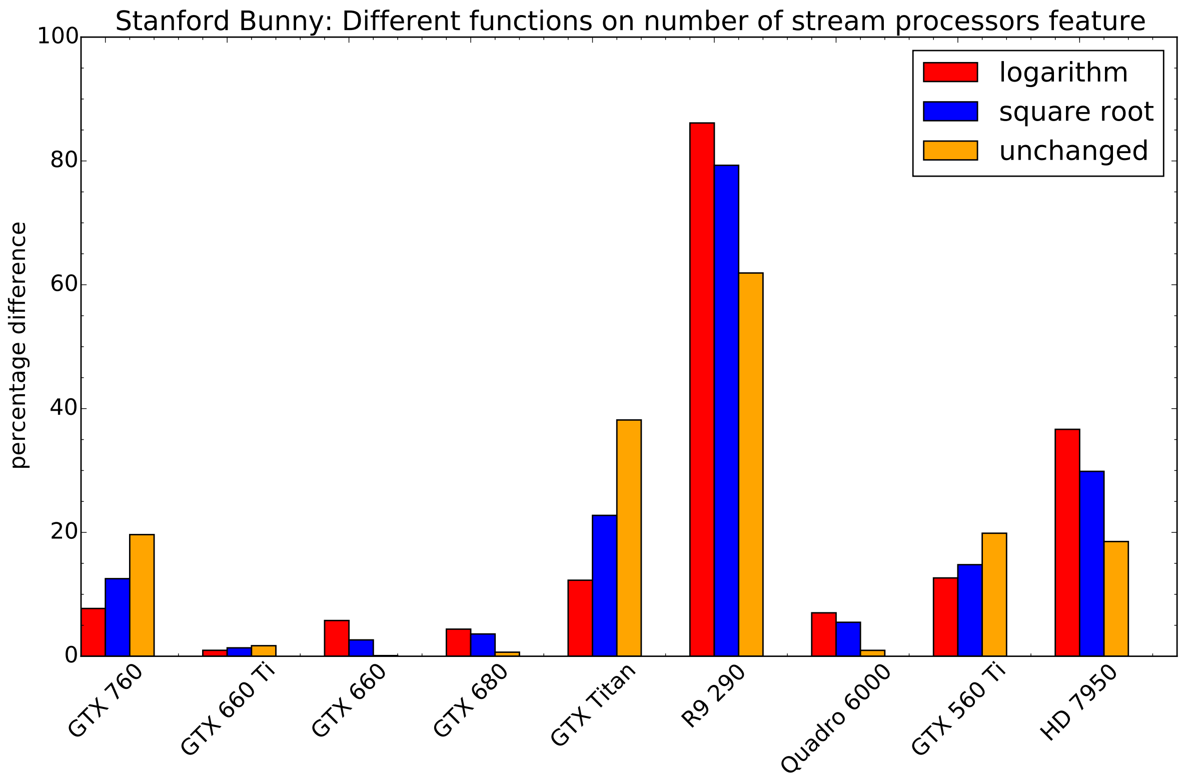


Figure 4.10: Prediction performance for the Matamata volume on different GPUs, relative to the function applied to the number of cores feature. A logarithmic function (red) and a square root function (blue) are applied.

Since only one volume is examined at a time, features specifying the volumes, as are used in the prediction described in Subsection 4.2.3, were not needed. Memory bandwidth, theoretical computational performance (flops), and the number of stream processors (cores) have been investigated as model features.

The usage of memory bandwidth in combination with the number of cores led to the best least prediction errors. Because of the high impact of the number of stream processors attribute on the results, the non-linear functions *(*square root) as well as *logarithm* have been applied to this number, followed by an investigation of the results. They are shown in Figure 4.10 for the Matamata data set. The other analyzed data sets behave in similar ways.

As the figure depicts, a square root function as well as a logarithmic function improves the prediction for almost all NVIDIA GPUs. However, the accuracy of the predictions on the AMD graphics cards, which is already lower in comparison, even decreases further when using one of those functions.

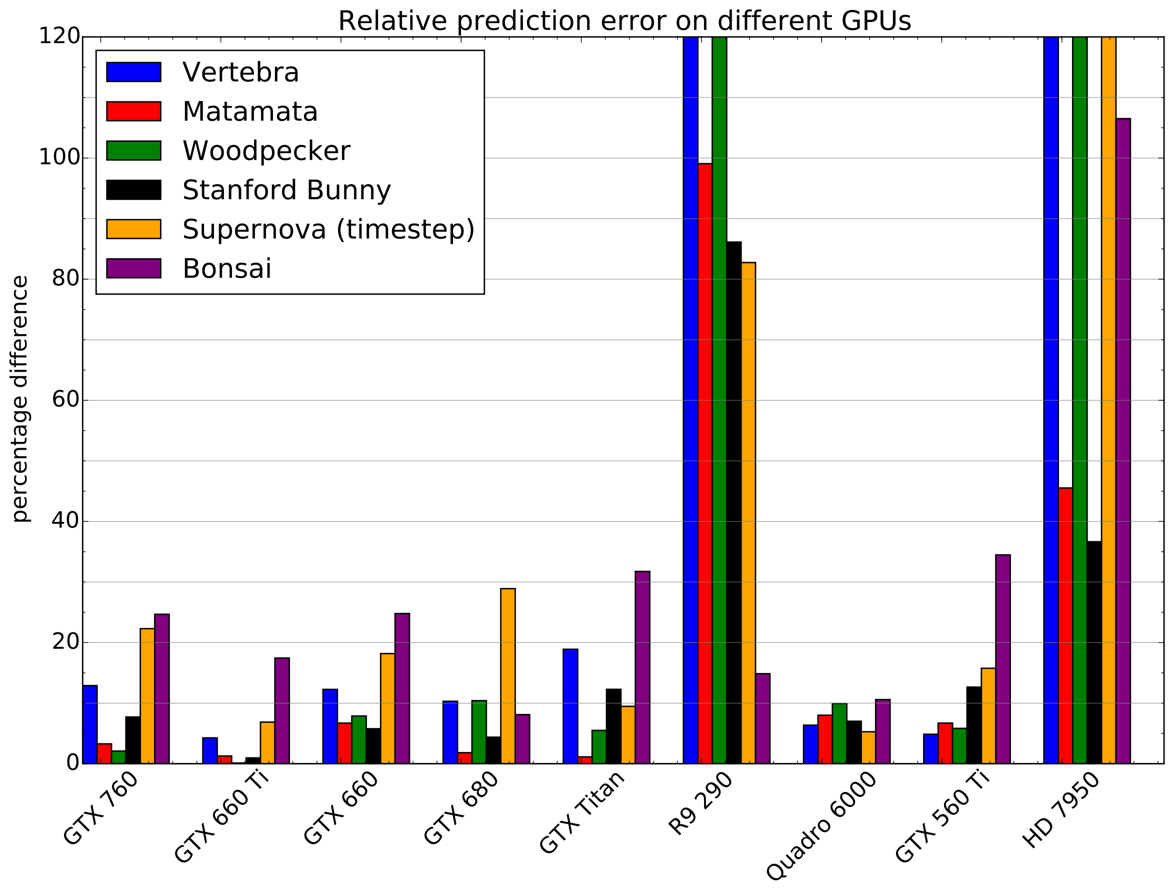


Figure 4.11: Relative prediction error for all tested volumes on diverse GPUs.

The relative differences between the predictions and actual execution times of all tested volumes are outlined in Figure 4.11. The logarithmic function was employed on the number of stream processors attribute.

The diagram shows that the results are decent for the NVIDIA graphics cards. Most volume data sets can even be predicted with a relative error below 10%. For the smaller data sets in the test field, the performance is generally worse, which can be expected considering that shorter execution times produce a higher error even for little discrepancies between prediction and measurement. A limitation to single floating point precision could be another negatively influencing factor in these cases.

The overall performance on the AMD GPUs is much worse compared to their counterparts, despite of decent results for some of the volumes. Notably, some of the relative differences even exceed the maximum 120% displayed in the diagram in Figure 4.11. The bad results on AMD GPUs are probably caused by an insufficient training data subset representing AMD hardware (only one in this case). Furthermore, it is hard to compare the vendor specific architectures based only on memory bandwidth and number of stream processors as many other attributes are obviously left out. Applying further input features which somehow specify the vendor architecture could improve the model in that regard. Additionally, the different implementations of the OpenCL specifications contribute to the performance diversity of GPUs from different manufacturers.

4.3 Current Limitations and Possible Solutions

There are some limitations to the developed models whose performance results are presented in this chapter. The model intended to enable dynamic predictions of the execution time of the next frame during user interaction in real-time, has two issues. A description of the model generation can be found in Section 3.2; results of the performance are discussed in Section 4.1.

For once, the R^2 score calculated from testing results decreases somewhat for larger volume data sets. That is mostly the case because some parameter configurations, that can occur during user interaction, are underestimated by the regression model. A possible solution to this issue could consist in the inclusion of a constraint in the model which maps a certain minimum execution time of the visualized volume data. This could help lower these underestimations and result in an overall improvement of the accuracy. The minimum execution time could be derived from the execution time measurements that are generated during the training sequence.

A second shortcoming occurs when dealing with sequences that are comply more with realistic user interaction (see Subsection 4.1.3): The developed model cannot predict small incremental parameter changes during user interaction as well as it predicts changes with higher impact on execution time.

A first approach on solving this issue could be to double the floating point precision. However, a general decrease in performance of the application would be a drawback. Furthermore, runtime environments - the utilized hardware in particular - have to support higher precision to avoid extensive and possibly expansive (in terms of runtime performance) implementation changes. Another possible solution could be the incorporation of some sort of error correction term into the model. That means, an error is tracked over the last several predictions and actual execution times and the prediction corrected based on that error. This correction, however, should only be applied if the prediction lies within a certain, comparably small range of the current execution time. The reasoning is that the mostly accurate predictions for the upcoming frame after greater changes of interaction parameters are not distorted.

A few limitations concerning the system predictions can be identified as well. One of the models was developed to predict the execution times of rendering an unknown volume data set on known systems. It is presented in Section 4.2 and evaluated in Section 4.2.3. The conducted measurements show a bad performance of the model for data sets with comparably small resolutions. This manifests in overestimation respective underestimation for almost all systems tested. Yet, the tendency of the prediction is normally correct. Adding more training data sets with a comparably small resolution as well as increasing the precision could already yield better results. A more advanced approach on solving these issues may be the introduction of limiting terms into the model, based on volume data resolution and training data gained from rendering comparable volumes. This could help smooth out the predictions and avoid occurrences of unrealistic predictions such as negative execution times.

A precision issue can be found while predicting the execution times of comparably small resolution volumes on unevaluated systems with the model described in Section 4.2, as well. The test data and a discussion of the results for that very model can be found in Subsection 4.2.4. Increasing the precision - if supported by hard- and software - could allow better results in this case as well. A far greater limitation of this model, however, are the big discrepancies between predictions and actual measurements for many data sets on AMD graphics cards specifically. A first straight forward approach to tackle this issue would be including more AMD GPUs in the training data. A probably more complex approach would be trying to add a form of "vendor term" to the model that incorporates a factor for comparability between GPUs from different manufacturers. This could be a term to make the number of stream processors of one architecture comparable to another, for instance. Also thinkable is the inclusion of more hardware specifics, enabling a more detailed mapping of hardware architectures and their characteristics in terms of computing performance into the model.

5 Conclusion and Future Work

High quality, interactive real-time volume visualizations demands a lot of hardware processing power to allow a satisfying user experience nowadays. Hardware capable of dealing with the high computational cost often needs to feature a massively parallel architecture. GPUs are used in this work as an acceleration device type suited for the required massively parallel execution.

The aim of this work was to develop a possibly general model to predict the performance on such devices. Different models from the field of machine learning have been developed and evaluated for performance prediction in two different use cases. One of them is predicting the execution time of the upcoming frame in a volume visualization application, the second one is predicting the average performance on different architectures.

User interactions, such as camera parameter adjustments and changes of transfer function, have the greatest influence on the performance of visualizing a volume data set via ray casting. It has been shown that acceleration techniques especially tend to influence execution times in a way that makes them hard to predict. This fact has been exploited in this work and parameters related to the acceleration techniques - namely empty space skipping and early ray termination - used in generating a model capable of making acceptable predictions for the execution time of an upcoming frame in real-time. As a machine learning technique, linear regression is used for making those predictions. In a sequence of random configurations with distinct camera positions and transfer functions, R^2 (coefficient of determination) scores of 0.62 up to 0.97 were reached for volume data sets of different sizes, with a score of 1.0 indicating a perfect match of predictions and execution time measurements. A more realistic sequence, the simulation of real user interactions, yields R^2 scores of 0.67 up to 0.96. The model can, therefore, be employed to give decent execution time predictions of an upcoming frame in real-time in a volume visualization application.

Two machine learning models for predicting the performance of volume visualization on different GPUs have been developed based on linear regression and evaluated. One is able to predict the performance of an unevaluated volume data set on different GPUs from the manufacturers NVIDIA as well as AMD using only the file size to specify the volume. The R^2 scores reached from 0.55 to 0.83 for various of the tested data sets.

However, the model has its limitations in predicting the performance of data sets with comparably small resolutions.

A second model targeting the prediction of performance on different architectures has been developed which can predict the performance of a volume data set on an unevaluated system. Using eight different GPUs to train the model, performance predictions could be made for a ninth system with a mean relative error ranging from 5.33% to 22.22% on NVIDIA graphics cards for the tested data sets. The prediction of average execution times by the model is generally more accurate for bigger volumes. A limitation of this model is its bad prediction performance for AMD GPUs. Both of the developed models can be used for basic performance prediction of volume visualization on different GPUs. These predictions could possibly be improved further by adding additional training data from more graphics cards and volume data sets.

Future Work

Addressing the current issues and limitations of the models could be a good starting point for future work. This would entail more tests on different graphics cards, especially from AMD, as well as including more volume data sets notably of smaller resolution, into the training data. Additionally, investigating the impact of higher floating point precision could be an approach in improving the performance of the developed models.

Extending the application to run on more diverse hardware architectures such as CPUs, mobile systems on a chip (SOC), Intel GPUs, and Many Integrated Core Architectures (MIC) as well as CPU clusters or multiple GPUs and therefore enable tests on them, could help in the process of generalizing the developed models. In using the OpenCL parallel programming framework, the groundwork for this task has already been done. Extending the application to run on distributed systems could enable the actual use of the real-time predictions for a load balancing scheme with dynamic work distribution in practice.

Other future work could consist of further refinement of the models including tests of alternative machine learning models. To be able to rank the developed models relatively to other performance prediction approaches, a direct comparison could be useful. Finally, an advancement of the volume renderer which allows processing unstructured grids and a validation of the machine learning models for those data types could be of interest.

Bibliography

[AMD15a] AMD. *Graphics Core Next Architecture, Generation 3, Reference Guide*. Tech. rep. Advanced Micro Devices, Inc., 2015. URL: amd-dev.wpengne.netdata-cdn.com/wordpress/media/2013/07/AMD_GCN3_Instruction_Set_Architecture.pdf (cit. on pp. 22–24).

[AMD15b] AMD. *R9 Fury Series Specs*. 2015. URL: http://www.amd.com/en-us/products/graphics/desktop/r9 (cit. on p. 25).

[BDP+10] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu. "An adaptive performance modeling tool for GPU architectures." In: *ACM Sigplan Notices*. Vol. 45. 5. ACM. 2010, pp. 105–114 (cit. on p. 31).

[Bli77] J. F. Blinn. "Models of Light Reflection for Computer Synthesized Pictures." In: *SIGGRAPH Comput. Graph.* 11.2 (July 1977), pp. 192–198. URL: http://doi.acm.org/10.1145/965141.563893 (cit. on p. 13).

[BMK13] M. Boyer, J. Meng, and K. Kumaran. "Improving GPU performance prediction with data transfer modeling." In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE. 2013, pp. 1097–1106 (cit. on p. 31).

[BRL+08] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. De Supinski, and M. Schulz. "A regression-based approach to scalability prediction." In: *Proceedings of the 22nd annual international conference on Supercomputing*. ACM. 2008, pp. 368–377 (cit. on p. 31).

[BS05] D. H. Bailey and A. Snavely. *Performance modeling: understanding the past and predicting the future*. Springer, 2005 (cit. on p. 30).

[DCH88] R. A. Drebin, L. Carpenter, and P. Hanrahan. "Volume rendering." In: *ACM Siggraph Computer Graphics*. Vol. 22. 4. ACM. 1988, pp. 65–74 (cit. on p. 14).

[Fly66] M. J. Flynn. "Very high-speed computing systems." In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909 (cit. on p. 22).

[Fri95] M. Friendly. “Milestones in the history of thematic cartography, statistical graphics, and data visualization.” In: *13TH INTERNATIONAL CONFERENCE ON DATABASE AND EXPERT SYSTEMS APPLICATIONS (DEXA 2002), AIX EN PROVENCE*. Press, 1995, pp. 59–66 (cit. on p. 13).

[HK09] S. Hong and H. Kim. “An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness.” In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA ’09. Austin, TX, USA: ACM, 2009, pp. 152–163. URL: http://doi.acm.org/10.1145/1555754.1555775 (cit. on p. 31).

[HLSR09] M. Hadwiger, P. Ljung, C. R. Salama, and T. Ropinski. “Advanced Illumination Techniques for GPU-based Volume Raycasting.” In: *ACM SIGGRAPH 2009 Courses*. SIGGRAPH ’09. New Orleans, Louisiana: ACM, 2009, 2:1–2:166. URL: http://doi.acm.org/10.1145/1667239.1667241 (cit. on pp. 19, 34).

[How15] L. Howes. *The OpenCL Specification Version 2.1*. Tech. rep. 23. The Khronos Group, Nov. 2015. URL: www.khronos.org/registry/cl/opencl-2.1.pdf (cit. on pp. 25, 26).

[Int15] Intel. *The Compute Architecture of Intel Processor Graphics Gen9*. Tech. rep. Intel Corp., 2015. URL: software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of_Intel-Processor_graphics-Gen9-v1d0.pdf (cit. on pp. 23, 24).

[Kau94] A. Kaufman. “Voxels as a computational representation of geometry.” In: *The computational representation of geometry. SIGGRAPH* 94 (1994) (cit. on p. 14).

[KMR+09] K. Kothapalli, R. Mukherjee, M. S. Rehman, S. Patidar, P. Narayanan, and K. Srinathan. “A performance prediction model for the CUDA GPGPU platform.” In: *High Performance Computing (HiPC), 2009 International Conference on*. IEEE. 2009, pp. 463–472 (cit. on p. 31).

[LBdS+07] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. “Methods of Inference and Learning for Performance Modeling of Parallel Applications.” In: *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’07. San Jose, California, USA: ACM, 2007, pp. 249–258. URL: http://doi.acm.org/10.1145/1229428.1229479 (cit. on p. 31).

[LC87] W. E. Lorensen and H. E. Cline. “Marching Cubes: A High Resolution 3D Surface Construction Algorithm.” In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH

’87. New York, NY, USA: ACM, 1987, pp. 163–169. URL: http://doi.acm.org/10.1145/37401.37422 (cit. on p. 15).

[Lev88] M. Levoy. “Display of surfaces from volume data.” In: *Computer Graphics and Applications, IEEE* 8.3 (1988), pp. 29–37 (cit. on p. 15).

[Lou08] K. M. Louis Bavoil. *Order Independent Transparency with Dual Depth Peeling*. Tech. rep. NVIDIA Corp., 2008. URL: http://developer.download.nvidia.com/SDK/10/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf (cit. on p. 35).

[Max95] N. Max. “Optical Models for Direct Volume Rendering.” In: *IEEE Transactions on Visualization and Computer Graphics* 1.2 (June 1995), pp. 99–108. URL: http://dx.doi.org/10.1109/2945.468400 (cit. on p. 15).

[Mic10] P. Micikevicius. *Analysis Analysis-Driven Optimization (GTC 2010)*. Tech. rep. NVIDIA, 2010. URL: http://www.nvidia.com/content/GTC-2010/pdfs/2012_GTC2010v2.pdf (cit. on p. 27).

[MJCP08] A. Mahesri, D. Johnson, N. Crago, and S. J. Patel. “Tradeoffs in Designing Accelerator Architectures for Visual Computing.” In: *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, pp. 164–175. URL: http://dx.doi.org/10.1109/MICRO.2008.4771788 (cit. on p. 32).

[MMK+11] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram. “GROPHECY: GPU Performance Projection from CPU Code Skeletons.” In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’11. Seattle, Washington: ACM, 2011, 14:1–14:11. URL: http://doi.acm.org/10.1145/2063384.2063402 (cit. on p. 31).

[NVI14] NVIDIA. *NVIDIA GeForce GTX 980, Whitepaper*. Tech. rep. NVIDIA Corp., 2014. URL: international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF (cit. on pp. 23, 24).

[SCW+02] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. “A framework for performance modeling and prediction.” In: *Supercomputing, ACM/IEEE 2002 Conference*. IEEE. 2002, pp. 21–21 (cit. on p. 30).

[SHN+06] H. Scharsach, M. Hadwiger, A. Neubauer, S. Wolfsberger, and K. Bühler. “Perspective isosurface and direct volume rendering for virtual endoscopy applications.” In: *Proceedings of the Eighth Joint Eurographics/IEEE VGTC conference on Visualization*. Eurographics Association. 2006, pp. 315–322 (cit. on p. 20).

[SİM+07] K. Singh, E. İpek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. “Predicting parallel application performance via machine learning approaches.” In: *Concurrency and Computation: Practice and Experience* 19.17 (2007), pp. 2219–2235 (cit. on p. 31).

[SK09] D. Schaa and D. Kaeli. “Exploring the multiple-GPU design space.” In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 1–12 (cit. on p. 32).

[TCSS07] M. M. Tikir, L. Carrington, E. Strohmaier, and A. Snavely. “A Genetic Algorithms Approach to Modeling the Performance of Memory-bound Computations.” In: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. SC ’07. Reno, Nevada: ACM, 2007, 47:1–47:12. URL: http://doi.acm.org/10.1145/1362622.1362686 (cit. on p. 31).

[Tib96] R. Tibshirani. “Regression shrinkage and selection via the lasso.” In: *Journal of the Royal Statistical Society. Series B (Methodological)* (1996), pp. 267–288 (cit. on p. 63).

[Wil83] L. Williams. “Pyramidal Parametrics.” In: *Computer Graphics (SIGGRAPH ’83 Proc.)* 17.3 (July 1983), pp. 1–11 (cit. on p. 36).

[YMM05] L. T. Yang, X. Ma, and F. Mueller. “Cross-platform performance prediction of parallel applications using partial execution.” In: *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*. IEEE. 2005, pp. 40–40 (cit. on p. 31).

[ZO11] Y. Zhang and J. D. Owens. “A quantitative performance analysis model for GPU architectures.” In: *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE. 2011, pp. 382–393 (cit. on p. 32).

All links were last followed in January 2016.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature