Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 63

Predicting the GPU Execution Time of 3D Rendering Commands using Machine Learning Concepts

Robin Keller

Course of Study:	Informatik
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Dipl.-Inf. Stephan Schnitzer
Commenced:	November 2, 2015
Completed:	May 3, 2016
CR-Classification:	I.2.6, I.3.4, C.4

Abstract

3D rendering gets more and more important in embedded environments like automotive industry. To reduce hardware costs, power consumptions, and installation space for Electronic Control Units (ECUs), the ECUs including their GPU are consolidated. Thus, multiple applications with diverse requirements use the same GPU. For example, the speedometer has safety-critical requirements, whereby third party application does not. To execute multiple applications on the GPU a GPU scheduler is necessary. So far, GPUs do not support preemption and thus, a non preemptive real-time GPU scheduling is required. Such a real-time GPU scheduler needs the execution time of GPU commands beforehand. Predicting the GPU execution time is complex and hence, we tackle this problem by using machine learning concepts to predict the execution time of GPUs.

In this thesis, influence factors for the execution time of 3D rendering commands on the GPU are analyzed and how these factors are processed to use them as features for the online machine learning algorithm. This leads to a linear regression problem that is tackled by using stochastic gradient descent with online normalization of the input data and an adaptive learning rate calculation. Furthermore, the shaders are analyzed during runtime and the feature vector contains the GPU instructions. This allows the algorithm to gain knowledge about the execution time for so far unseen shaders. To validate the concept, it is implemented and evaluated on a workstation computer and on an embedded board. We show for the workstation computer that the newly developed concept has higher accuracy in glmark2-es2 benchmarks than the previous approach. For the embedded board is shown that the execution times for GPU instructions can be learned accurately. However, the developed concept provides room for improvement at constant scenes, where only the execution time changes.

Kurzfassung

3D-Rendering wird in eingebetteten Umgebungen, wie zum Beispiel der Automobilindustrie, zunehmend wichtiger. Um die Hardwarekosten, den Energieverbrauch und den Einbauraum von Steuergeräten zu reduzieren, sollen die Steuergeräte einschließlich ihrer GPU konsolidiert werden. So verwenden mehrere Anwendungen mit unterschiedlichen Anforderungen dieselbe GPU. Zum Beispiel weist der Tachometer sicherheitskritische Anforderungen auf, wobei Drittanbieter-Anwendungen diese nicht besitzen. Zur Ausführung mehrerer Anwendungen auf der GPU ist ein GPU-Scheduler notwendig. Bisher unterstützen GPUs keine Preemption, sodass ein preemptiver GPU-Scheduler, welcher Echtzeitanforderungen erfüllt, erforderlich ist. Ein solcher Echtzeit-GPU-Scheduler benötigt im Voraus die Ausführungszeit der GPU-Befehle. Um die GPU-Ausführungszeit vorherzusagen, wird in dieser Arbeit ein Vorhersage-Framework entworfen, welches maschinelles Lernen verwendet.

In dieser Arbeit werden Einflussfaktoren auf die Ausführungszeit von 3D-Rendering-Befehlen auf der GPU analysiert. In diesem Zusammenhang wird aufgezeigt, wie diese Faktoren verarbeitet werden, um sie als Features für den Algorithmus zu verwenden, der zur Laufzeit maschinelles Lernen einsetzt. Dieses führt zu einer linearen Regressionsanalyse, welche durch die Verwendung des Gradientenverfahrens mit Normalisierung der Eingabedaten zur Laufzeit und adaptiver Lernratenberechnung angegangen wird. Außerdem werden die verwendeten Shader zur Laufzeit analysiert und die ermittelten GPU-Instruktionen in den Feature Vektor aufgenommen. Das erlaubt dem Algorithmus, Wissen über die Ausführungszeit von bisher unbekannten Shadern zu erlangen. Um das Konzept zu validieren, wurde es auf einer Workstation und einem eingebetteten Board implementiert und evaluiert. In der Evaluation wird dargestellt, dass für die Workstation das neu entwickelte Konzept eine höhere Genauigkeit in glmark2-es2 Benchmark-Tests besitzt als in dem bisherigen Ansatz. Für das eingebettete Board wird gezeigt, dass die Ausführungszeiten für GPU-Instruktionen präzise gelernt werden können. Jedoch bietet das entwickelte Konzept Raum für Verbesserungen bei konstanten Szenen, in denen sich nur die Ausführungszeiten verändern.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

In this chapter, the work at this topic is motivated, the contribution in this research area and the organization of this thesis is presented.

1.1 Motivation

Nowadays, graphics processing units (GPU) are integrated into embedded systems to use high resolution 3D rendering. The number of displays in embedded systems increases, which leads to a greater demand of GPUs, that are required to control the displays. In embedded systems, usually, each electronic control unit (ECU) contains one GPU. Therefore, with a higher number of display, the number of ECUs increases as well. In the automotive industry, the cars are using more and more displays to show high resolution 3D graphics.

To reduce cost, energy and space requirements in a setup with multiple ECUs, the ECUs can be consolidated. However, that means, that multiple 3D rendering tasks need to run on one ECU in parallel. That leads to the fact, that multiple 3D rendering tasks must run on one GPU. To run multiple 3D rendering tasks on one GPU, a GPU scheduler is necessary.

Another aspect that needs to be considered for consolidating ECUs is that, 3D rendering tasks can have real-time requirements. For example, in the case of the automotive domain, in high class cars the instrument cluster containing the tachometer is displayed with 3D rendering commands and has real-time requirements. Other 3D graphics applications, which run on the same GPU, can be infotainment or third party applications. If there is at least one 3D rendering task with real-time requirements running on the GPU, the system needs to handle real-time requirements.

This leads to a setup, in which a real-time GPU scheduler is required. Because of the fact that the GPU is not a preemptive system, a started 3D rendering job on the GPU cannot be interrupted and needs to be finished before another 3D rendering job can be started. Therefore, such a real-time GPU scheduler needs the execution time of each task beforehand. Existing real-time GPU schedulers such as [22] depend on accurate execution time prediction.

However, prediction of the execution time of 3D rendering commands is complex. To determine the execution time of 3D rendering commands, the rendering can be emulated on the central processor unit (CPU). The predicted execution time is accurate, but the calculation of the execution time takes much more time than executing the 3D rendering job on the GPU. On the other hand, the execution time can be estimated by only using the raw input data of the 3D rendering command. Thus, the time to estimate the execution time is reasonable, but the accuracy of the prediction is insufficient. Thus, a trade-off between calculating some parts of the 3D rendering command and estimating other parts needs to be find.

Kato et al. [3] and Yu et al. [26] proposed history based approaches to predict the execution time of GPU command groups. If the GPU command group data varies, the results of the history based approach are insufficient accurate. Schnitzer et al. [21] use the tracked OpenGL context to predict the execution time of GPU command groups. A downside for this approach is, that for each shader a calibration is needed. A prediction cannot be performed during the calibration, because the calibrated values are uncertain and the predicted execution time would be inaccurate.

This motivates us to propose a concept which improves the previous approaches. This can be done by getting rid of the calibration for new shaders and being adaptive to varying environments using a self-tuning machine learning. Thus, our machine learning algorithm leads to higher flexibility and faster adaption of the real world. However, OpenGL is a complex state based graphics rendering environment, which does not provide all necessary information for machine learning. Another challenge is the embedded system setting, in which less computation power is provided. Therefore, stochastic gradient descent is applied which leads to more challenges.

1.2 Contribution

In this section, an overview of the main contributions of this work is presented. In general, to predict the execution time, the influence factors need to be analyzed. Using the measured execution time and the influence factors of past 3D rendering commands, machine learning is applied to learn the underlying model parameters.

1. Using the influence factors of the execution time of 3D rendering commands using OpenGL, the features for machine learning are determined and presented. Furthermore, a linear function which describes the execution time is presented.

2. Learning on GPU instruction level is introduced. Therefore, a linear model, which is independent on the shader, is presented.
3. The stochastic gradient descent method is applied on a data set collected from the state based OpenGL context. To this end, online learning with feature normalization and adaption of the learning rate is utilized.
4. To evaluate the concept, two platforms are used. On the one hand, a workstation graphics card, on the other hand, an embedded graphics card.

1.3 Outline

This section comprises an outline of the thesis.

Chapter 2 – Related Work: In this chapter, the previous publications in the area of GPU execution time prediction of 3D rendering commands using machine learning concepts are listed.

Chapter 3 – Background: This chapter introduces the background of the following work. The two majors are the OpenGL rendering pipeline and machine learning using online learning with gradient descent.

Chapter 4 – Concept: In this chapter, the concept using machine learning is presented. That contains beyond others, a detailed analysis of the OpenGL rendering pipeline and the derived linear model. Furthermore, the learning on GPU instruction level is explained in detail and the resulting stochastic gradient descent algorithm is presented.

Chapter 5 – Implementation: In this chapter, the platform independent implementation of the concept is introduced.

Chapter 6 – Evaluation: The evaluation compares the existing bounding box solution for predicting the execution time and the concept of this work. In addition, the execution time of each GPU instruction is presented.

Chapter 7 – Summary and Future Work: In this chapter, the summary and the future work are presented.

2 Related Work

The related work of this thesis is split into two scopes. First, this work is part of real-time GPU scheduling and thus, execution time prediction of 3D rending commands. In order that the real-time GPU scheduler can schedule the GPU task with respect to the deadlines, the execution time prediction has to be precise. The second, the prediction is made using online machine learning concepts, in particular stochastic gradient descent.

2.1 GPU Scheduling and Execution Time Prediction

So far tasks on GPUs are not preemptive. Microsoft designed a GPU preemption model, which needs at minimum WDDM version 1.2 and is available since Windows 8. However, although all WDDM 1.2 display miniport drivers must support this feature, they might reject it [14].

To set up fine-grained sharing of GPUs, a GPU command scheduler can be implemented in the device driver. One approach from Kato et al. [9] is called TimeGraph, which is a GPU scheduler for real-time multi-tasking environments. The TimeGraph scheduler supports the posterior and apriori enforcement policy. The posterior enforcement submits a task to the GPU and calculates afterwards an overrun penalty, if the task occupied the GPU for more time than expected. This has advantages for throughput, but not performable with real-time requirements. The apriori enforcement policy only allows to submit GPU command groups if the prediction cost is smaller or equal the budget.

Kato et al. [9] propose a history-based prediction approach for the execution time, where only the graphics card method and the size of data are considered. The information to predict the execution time are gained out of the GPU command profiler. If no previous execution time for the exact same method and size of data exist, the worst-case execution time over all entries is taken into account. With this approach it is not possible to predict complex scenes, in which GPU commands change.

Yu et al. [26] propose for one of their scheduling policies a similar history-based approach like Kato et al. [9]. Instead of taking the entire history into account, the average execution time of the last twenty executions are considered. In this work, the

OpenGL context as well as the actual execution time of previous commands are included to calculate the predicted execution time.

Bautin et al. [3] describe a GPU scheduler based on Deficit Round Robin scheduling. Irrespective of the demand of a process, it distributes the share of the GPU identical. The efficient utilization of the GPU is the goal of the scheduler called GERM, which does not fulfill real-time requirements.

Schnitzer et al. [22] propose a real-time GPU scheduler for 3D GPU rendering. The scheduler handles different priorities and a desired number of frames per second per application. Thus, the GPU utilization is optimized. The observing of the real-time requirements of the GPU scheduler highly depends on the predicted execution time for each application.

A method to predict the execution time of GPU command groups considering the OpenGL context is proposed by Schnitzer et al. as well [21]. A prediction model for the main OpenGL ES 2.0 GPU commands flush, clear, and draw is designed. The presented framework uses the OpenGL context and the semantic of the OpenGL commands to predict the execution time. Therefore, the OpenGL API calls are intercepted and the context is tracked. Additionally, a heuristic to calculate the number of fragments is presented. The number of fragments is calculated using the 3D bounding box of the rendered model and the vertex shader projection. This leads to better results than in history-based prediction. However, the approach needs to calibrate the underlying model parameters at the beginning of each new shader.

This thesis uses the framework presented by Schnitzer et al. [21] as basis, adding history-based information and adaption using online machine learning.

2.2 Online Learning: Stochastic Gradient Descent

The previous work in the area of GPU execution time prediction of 3D rendering commands uses manual model driven approaches so far [21]. To get a more adaptive setup, online machine learning is used. This arises challenges that are explained in the following.

One of the challenges is the scale of the feature data. It is unknown and can change depending on the OpenGL ES 2.0 application. Thus, per-feature scaling needs to be applied to the data. Ross et al. [20] proposed normalized online learning to be independent of features scales. Therefore, no pre-normalization of the input data is required.

Another challenge that we are facing is that, the optimal parameter of unknown input data cannot be calculated, but have strong impacts on the convergence speed of the algorithm. Thus, online parameter adaptation is appropriate to achieve reasonable parameters for the stochastic gradient descent algorithm. The learning rate is one of the parameters which can be adapted. Almeida et al. [1] proposed parameter adaption in scenarios with stochastic optimization. A per-feature learning rate is suggested. The adaption of the learning rate leads to an optimization of the gradient length. In [1] the learning rate mainly depends on the last two gradients, but does not cover solutions for sparse data. Another approach called ADAGRAD [5] leads to a more robust gradient length calculation. However, in this approach the learning rate needs to be set manually. An extension of the ADAGRAD algorithm is proposed by Zeiler et al. [27]. It is called ADADELTA and calculates the learning rate on its own. Another advantage is that the monotonically decreasing learning rate term of ADAGRAD is replaced by a decaying average. Thus, the learning rate will not end up being infinitesimal small.

To obtain sparse model parameters, the update rule for the model weights contains a regularization term. Langford et al. [10] proposed a method to implement the regularization term in online stochastic gradient descent. The main approach is to truncate the gradient, if it points close to zero. Therefore, it is more likely that unimportant weights close to zero end up in zero weights.

3 Background

In this chapter, the background of this thesis is explained. On the one hand, background information about OpenGL is required. In order to find impacts for the execution time of OpenGL rendering commands, the OpenGL ES 2.0 Pipeline is explained. On the other hand, to figure out which machine learning algorithm is suitable for the obtained influence factors, machine learning approaches are presented.

3.1 OpenGL ES 2.0

Open Graphics Library for Embedded Systems 2.0 (OpenGL ES 2.0) is an Application Programmable Interface (API) in the second version to interact with graphics hardware. OpenGL ES 2.0 is based on OpenGL 2.0, but is designed primarily for graphics hardware running on embedded and mobile devices. Substantially, the user first opens a window with a framebuffer, in which the program will draw later. The framebuffer will be displayed on a function call. Then, to operate with OpenGL, a context must be created. Subsequently, the user can specify two- or three-dimensional geometric objects that are drawn into the framebuffer. Geometric objects are mainly points, line segments, and polygons. The objects can be modified though additional calls, e.g. lighting, color or mapping in either two- or three-dimensional space [8, pp. 1, 2].

In the following, the dominating factors with respect to execution time are presented. Therefore, the OpenGL rendering pipeline is introduced.

3.1.1 Primitives

Primitives in OpenGL ES 2.0 are represented in a generic way using vertex arrays. With this representation seven geometric objects can be drawn: points, connected line segments, line segment loops, separate line segments, triangle strips, triangle fans, and separated triangles. Each of these primitives is defined via one or more vertices. Each vertex can have attributes, e.g. color, normal, texture coordinates, etc., which are used in further steps. [8, pp. 4, 15]

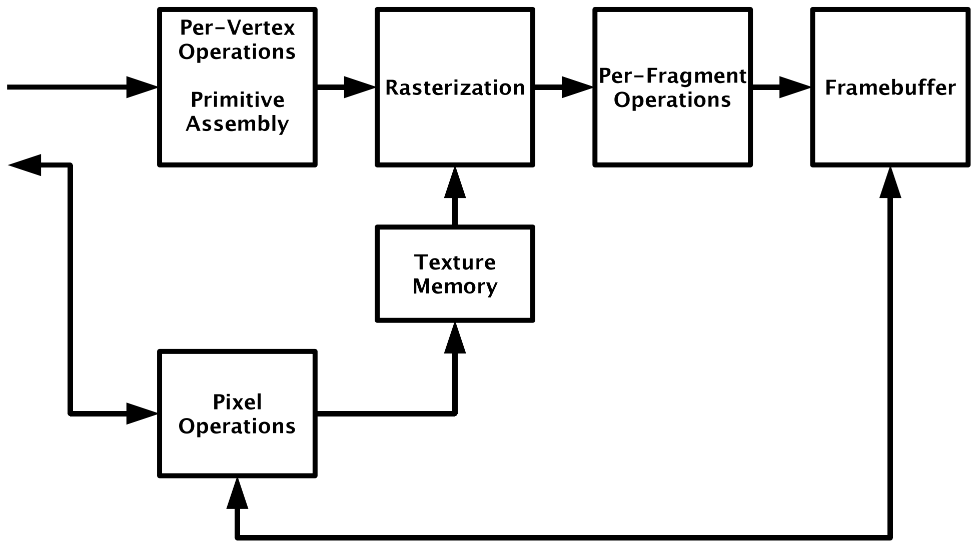

Figure 3.1: OpenGL ES 2.0 Processing Pipeline [8, p. 13]

3.1.2 Rendering Pipeline

Figure 3.1 shows the graphics pipeline implemented by OpenGL ES 2.0. Each command is passed from left to right through the pipeline and the result is written to the framebuffer. Some commands are used for creating geometric objects, others are used for changing the state of the respective stages. Using pixel operations, either the texture memory can be written or the framebuffer can be read or written [8, p. 11].

The three operations, shown in Figure 3.1, are the main functions and they will be discussed in the following.

3.1.3 Vertex Processing and Primitive Assembly

This first stage consists of four fine-grained stages, that are processed in the following sequence. In general, each vertex is processed on its own. In case of clipping, new vertices might be created and obtain modified values, which is described below [8, 15].

Vertex Shader. Each vertex, which is specified in the API calls `DrawArrays` or `DrawElements`, is processed by the vertex shader. After running the vertex shader, the vertices are passed on to primitive assembly [8, p. 26].

Primitive Assembly. In this stage, the vertices are assembled into primitives, e.g., line strip, triangle fan.

Coordinate Transformations. OpenGL ES 2.0 has different coordinate systems. In the first place, the vertices are in clip coordinates, i.e., are four-dimensional homogeneous vectors. After processing this fine-grained stage, each vertex needs to be in viewport coordinates. The viewport coordinates are three-dimensional, where the first two coordinates define the two-dimensional position on the framebuffer and the third coordinate defines the depth information. To transform the position of the vertices from clip to viewport coordinates, one intermediate coordinate system is used: normalized device coordinates, which do not contain the homogeneous component of the clip coordinates [8, pp. 44, 45].

Primitive Clipping. When the vertices are in the viewport coordinate system, the primitives can be clipped. Therefore, a clip volume is created. Only primitives inside the clip volume end up in the framebuffer. However, primitives can be completely inside, completely outside, or partly inside the volume. In the first two cases, it is obvious that the vertices can either be further considered or dropped. In the case that the primitive is partly inside the clip volume, the primitive needs to be shrunken to the edges of the clip volume [8, p. 46].

After the first stage is finished, all primitives are inside the clip volume in viewport coordinates.

3.1.4 Rasterization

The inputs of this stage are primitives in viewport coordinates which are transformed into a two-dimensional image. The result of this second stage is the color and depth information of each point of the image. The primitive rasterization determines if a point of the image is occupied by a primitive, the texturing obtains the color of a fragment by sampling the texture image and the fragment shader calculates the color and depth information [8, pp. 28, 65, 66].

Primitive Rasterization. This fine-grained stage creates fragments for each point in the image, which is covered by a primitive. The rasterization can be split into three parts: point, line, and triangle rasterization. To determine the affected fragments of a point or a line, the radius or line width is considered respectively. However, the detection of the area of polygons is more challenging. Polygons are three dimensional objects and have a direction to which they are facing. OpenGL offers the possibility to render only the front or back face. The other face is not rendered respectively, which is called culling [8, pp. 48, 49, 57].

Texturing. Texturing is not a stage of its own. It is used by the fragment. The texturing maps an image onto a fragment, which is realized by the fragment shader. The fragment shader samples the texture image to obtain the color for the fragment. This is the typical use case, but the texture can also be filled with other information, which is supposed to be passed to the fragment shader [8, pp. 65, 66].

Fragment Shader. The fragment shader is a program, which runs for each fragment. The fragments were created by the rasterization from primitives like points, lines, and polygons. These fragments are passed to this stage. The fragment shader computes the color of the fragment [8, pp. 86, 87].

The result of the rasterization stage are fragments with color information.

3.1.5 Per-Fragment Operations

The per-fragment operations perform tests and modifications for each window coordinate. Therefore, the fragments at each window coordinate are considered. The tests are executed in the order listed below. The result is a filled framebuffer, which is affected by the fragments processed in this stage.

1. **Pixel Ownership Test.** The pixel ownership test verifies if the pixel at location (x_w, y_w) in the framebuffer is currently owned by the current GL context. If the pixel is not owned by the current GL context, the fragment is discarded [8, p. 91].

2. **Scissor Test.** The scissor test determines if the pixel location is within a rectangle defined by `void` **Scissor**(`int` *left*, `int` *bottom*, `int` *width*, `int` *height*). If the pixel is inside this rectangle, $left \leq x_w < left + width$ and $bottom \leq y_w < bottom + height$, the fragment passes the test, otherwise, the fragment is discarded. The scissor test is optional and can be either enabled or disabled [8, p. 93].

3. **Multisample Fragment Operations.** Each pixel has an alpha and a coverage value. These values can be modified by the Multisample Fragment Operations. Multisample Fragment Operations is an antialiasing technique that works on a subfragment level. Therefore, every pixel is divided into several samples, which are used for rendering. Thus, there exists a higher resolution for rendering. Afterwards, the samples are resolved to achieve the original number of pixels [8, p. 93] [15, pp. 234,249].

4. **Stencil Test** The stencil test determines if a fragment gets discarded or not. The first step is to initialize the stencil buffer with a per-pixel mask by drawing primitives and the second step is to use the stencil operations (e.g., `GL_KEEP, GL_REPLACE` etc.), the stencil function (e.g., `GL_EQUAL, GL_LESS` etc.) and a compare value to determine if a fragment update is processed. The stencil test is optional and can be either enabled or disabled [15, p. 240] [8, p. 95].

5. **Depth Buffer Test** Basically, the depth buffer test avoids drawing fragments that are in the background and covered by other fragments. Therefore, for each fragment, the actual depth buffer value is compared to the fragment's depth value. Possible comparison operators are for example `ALWAYS, LESS, EQUAL`, etc. If the fragment passes the test, the depth buffer is updated. Otherwise, the fragment is discarded. The depth buffer test is optional and can be either enabled or disabled [8, p. 96].

6. **Blending** Blending modifies the R, G, B and A values of the framebuffer. Therefore, an incoming fragment at position (x_w, y_w) is combined with the framebuffer at the same position. How the values are combined is defined by a blend equation [8, pp. 96, 97].

7. **Dithering** Dithering modifies the color in the framebuffer using a dithering algorithm. The aim of the algorithm is to simulate greater color depth, when only a limited color depth is available. However, OpenGL ES 2.0 does not specify an algorithm. Thus, it is implementation-dependent. The dithering test is optional and can be either enabled or disabled [15, p. 249].

After this last stage, all the fragments are processed and the framebuffer is filled with the generated scene.

3.1.6 Special Functions

OpenGL ES 2.0 uses a command stream for all commands. For synchronization purpose the `Flush` and `Finish` commands are used. The `clear` command is a framebuffer manipulation and provides the functionality to reset the framebuffer to a particular color [8, pp. 103,122].

Flush The command `void` **Flush**`(void)`; indicates that all previously sent commands must be completed in finite time.

Finish The command `void` **Finish**`(void)`; forces that all previously sent commands have to be complete. This command is blocking and only returns when all commands are fully propagated.

Clear The command `void` **Clear**(`bitfield` *buf*); sets every pixel to the same value. The parameter *buf* determines which values are set. Possible bits, that can be set, are `COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT,` and `STENCIL_BUFFER_BIT`. Respectively to the bits, the buffer values are set to the predefined values.

3.2 Machine Learning

Machine learning is composed out of machine and learning. In Oxford Dictionary, “to learn” is defined by “to gain knowledge or skill by studying, from experience, from being taught, etc.” [17, p. 886], which means in the case of machine learning that we transfer experience into knowledge. Experience is input data which is processed by the machine and, as a result, we gain knowledge out of it. Knowledge is data which can be interpreted in a certain way by the machine.

The machine in our case is an algorithm, which uses the input data to learn from it and change its internal state respectively. Every time the machine gets new input data it will take the data into account and learn from it. Thus, the machine gains experience gradually, can improve, and will perform better in the future.

It is often possible to find a software solution for problems without machine learning. Anyhow, there exist machine learning algorithms to get better results. For this thesis, the important reasons to use machine learning algorithms are listed in the following [16].

- A function cannot be described, because the underlying model is unknown. The only information are input and output data. The machine learning method can identify the unknown model by matching input and output data and therefore, adjust its internal state and hence, predict output data for so far unseen input data.
- Structure and correlations in data are not known. This information can often be identified by machine learning methods.
- For a changing environment, it is more convenient to have a program which can adapt itself. Machine learning methods provide this possibility.

3.2.1 Classification of Machine Learning Algorithms

In this subsection, a brief overview about different machine learning methods is given. Thus, the approach which suits best to our problem statement can be determined.

Supervised versus Unsupervised Learning

First we want to distinguish between Supervised and Unsupervised Learning [23]. While supervised learning uses an expert to gain information, unsupervised learning does not. The expert or supervisor provides the correct output information of the function that should be learned. In unsupervised learning this information is not provided and in unsupervised learning algorithms, this functionality is obtained by examining the structure of the data.

For example, a function which marks an email dependent on the content as spam or not. In supervised learning, the supervisor provides for each email the label spam or not-spam. Therefore, the function can correlate the email content with the provided label. In case of unsupervised learning, the unsupervised algorithm needs to figure out similar structures in emails and separate the emails in two sets: spam and not-spam [23].

In our problem we have a so-called supervisor. That means that we get labeled input data with corresponding output data to learn the function to predict the execution time of a 3D GPU rendering command.

Active versus Passive Learning

Another distinction is the type of learning which can be active or passive. This aspect is considering at which situation a supervisor is providing the correct output data. An active learning algorithm can ask the learner for the correct output data anytime, whether a passive learning algorithm can only get the information by observing the environment [23].

In the scope of this work, we only can observe the environment and use the provided information. Thus, we consider the use of passive learning further.

Statistical Classification versus Regression Analysis

There are multiple machine learning algorithms, which have the property of supervisors and passive learning. The two major categories are statistical classification and regression analysis.

Classification is to categorize input data into groups. For instance, optical character recognition uses an image as input and classifies the image into a character. Thus, each character is a group. This can be extended to character strings by running the program consecutively for each character [2].

Regression is a method to create a mapping from input variables to an output variable. For example, in automotive market used cars have different attributes, e.g. mileage. If we want to predict the prices of newly offered cars on the second hand market and we already got the data from previous sales, a regression analysis can be made. A linear function can be used to fit as best as possible for the given data. Then, the price for the car can be predicted using the function which is gained through the regression analysis [2].

Because of our problem statement, where we get a continuous output value provided, the regression analysis is our choice as algorithm class.

3.2.2 Linear Regression

Linear regression is a regression analysis with a function that is linear with respect to the input parameters. That means that the output variable depends linear on each input parameter. A set of pairs of input and output variables are necessary to learn the model. The input variables are also called predictors, independent variables or features. By contrast, the output variables are called responses or dependent variables [7, pp. 9,10].

Input variables are denoted as X and output variables either as Y, if they are quantitative, or G, if they are qualitative. We only consider quantitative outputs, because our aim is to determine a function which has a quantitative output variable. Thus, we are using Y for the output variables. The jth component of a vector X is denoted X_j. The observed values, also called samples, are written in lowercase: x_i for the ith sample of X [7, p. 10].

To learn the function which leads to our prediction of the output $\hat{Y}$, we use training data to build prediction rules. Training data is a set of tuples, which are measurements (x_i, y_i), $i = 1, \ldots, N$. A simple approach to construct the prediction rules is the linear model fit by least squares. The linear model has been used for the last 30 years and is still one of the most important tools [7, p. 11].

The linear model uses a given vector of input values $X^T = (X_1, X_2, \ldots, X_p)$ to predict the output Y using the model

$$\hat{Y} = w_0 + \sum_{j=1}^{p} X_j w_j. \tag{3.1}$$

Values w are coefficients of the linear model. w_0 is a constant summand which is called bias in machine learning. In terms of math, w_0 specifies the point $(0, w_0)$ at which the Y-axis is cut. For convenience, it is common to set X_0 to 1 and include w_0 into the vector w . Therefore, one can come up with a more general model and denote the prediction with the inner product

$$\hat{Y} = X^T w = \sum_{j=0}^{p} X_j w_j \text{ with } X_0 = 1. \tag{3.2}$$

In general, X can be a vector or a matrix and therefore $\hat{Y}$ can be a scalar or a vector. In our case, X is a vector and hence, $\hat{Y}$ is a scalar [7, p. 12].

The prediction can be viewed as a linear function $f(X) = X^T w$. To obtain the gradient of f, we derive f, which leads to $f'(X) = w$. This gradient points to the direction of ascent and can be used to improve the weights w of the linear model to get a better approximation of the real world. The most popular approach of fitting the linear model is the method of least squares [7, p. 12].

The main idea of the method of least squares is to find the coefficients w to minimize the residual sum of squares

$$\text{RSS}(w) = \sum_{i=1}^{N} (y_i - x_i^T w)^2. \tag{3.3}$$

For the minimum of RSS(w) exists exactly one solution, because it is a quadratic function. To get the parameter w for which RSS(w) is minimal, Equation (3.3) has to be differentiated and solved for w. It is simpler to describe the solution in matrix notation [7, p. 12]. Thus, the matrix notation is

$$\text{RSS}(w) = (\mathbf{y} - \mathbf{X}w)^T (\mathbf{y} - \mathbf{X}w), \tag{3.4}$$

where $\mathbf{y}$ is the vector of output values and $\mathbf{X}$ is a matrix containing one sample in each row. To minimize Equation (3.4), it needs to be differentiated w.r.t. w which leads to

$$\mathbf{X}^T (\mathbf{y} - \mathbf{X}w) = 0. \tag{3.5}$$

Hence, there exists a unique solution for w, if $\mathbf{X}^T\mathbf{X}$ is nonsingular:

$$w = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \tag{3.6}$$

If all the input and output data is given in advance, this can be solved by matrix inversion and multiplication. If the equation can be solved, then it is optimal for the given training data. However, in the case of this work, the data is not given in advance but continuously in a stream. For this reason, online machine learning instead of batch learning for the linear regression will be further considered.

3.2.3 Online Learning with Stochastic Gradient Descent

In general, online machine learning is to learn a function $f : X \to Y$, where X are input values and Y are output values. Because of the problem we are facing, we are using a linear map from multiple input values to one output value. Thus, the input value x is a vector and the output value y is a scalar. This leads us to linear regression using online learning, in particular, stochastic gradient descent. Stochastic gradient descent is to improve the coefficients w of the linear model step by step using a loss function. The loss function uses an approximate gradient that points in the direction of improvement.

In batch learning, as described in Section 3.2.2, the loss function is a sum over the entire training data set and hence, the optimal global solution can be calculated. The loss function is an error function, which returns the error between the current and actual solution. In online learning, only one sample per time is considered to improve the weights of the linear model, which leads to a more adaptive algorithm.

The most commonly employed update rule to sequentially update the weights of the linear model $f(x) = x^T w$ is

$$w_{t+1} = w_t - \eta_t \nabla L(y, x), \tag{3.7}$$

where η is the so-called learning rate and $\nabla L(y, x)$ denotes the gradient of the loss function with its parameters y and x. t is the current and $t + 1$ the next time step. To obtain the improved weights, the previous weights are required. This method is called stochastic gradient descent [2, p. 219].

In general, the gradient descent method calculates the optimal gradient with respect to the entire trainings set. In stochastic gradient descent, the gradient is calculated only considering the error of the current sample, not the entire trainings set [4].

Next, the parameter learning rate and loss function are discussed further. The learning rate and loss function are essential to establish a fast convergence and robust update rule. The learning rate scales the gradient of the loss function, whereby the gradient points towards the direction of the minimum at point of the derivation.

Loss Function

A widely used loss function is the sum of least squares

$$L(y, x) = \frac{1}{2}(y - x^T w)^2, \tag{3.8}$$

where y is the actual desired value, x the training input data and w the weights of the linear model. The gradient of the loss function

$$\nabla L(y, x) = \frac{\partial L(y, x)}{\partial w} = (y - x^T w)x \tag{3.9}$$

is used to improve the weights w of the linear model.

Learning Rate

The learning rate η is a factor to scale the gradient of the loss function. For higher η, a faster convergence is gained, but overshooting is more likely as well. Overshooting means that the update of the weights w was too strong and the optimal goal is passed. If the learning rate η is chosen too high, it is also possible that the overshooting leads to divergence.

A common simple approach is to set η in respect to the time. Usually, η_t decays over time, which is called annealing, e.g.

$$\eta_t = \frac{1}{t}, \tag{3.10}$$

where t is the time step or current number of iteration. Robbins and Monro [19] showed that a convergence is ensured, when $\eta \to 0$, $\sum_t \eta_t = \infty$, and $\sum_t \eta_t^2 < \infty$. These requirements are satisfied for Equation 3.10.

3.2.4 Assumptions of Linear Regression

The linear regression model has several assumptions on the input data set. The key assumptions are listed in the following [18].

No Measurement Error. The observation (x, y) must be measured without an error. This can be partially relaxed. Then only x must be measured without an error. Thus, w_0 includes the measurement error of y.

Linear Relationship. The output variable y must be linear dependent on each input variable x_i. This does not mean, that a nonlinear data set cannot be used by the linear model. It is satisfying if there exists a function which transforms the nonlinear data set to a linear. Thus, $x_i^2 w_i$ is a valid term for the linear model.

Independence. The input variables x_i must be independent of each other. Otherwise multicollinearity is present.

Homoscedasticity. All input variables must have the same constant variance of the error.

Normally Distributed. The error of each input variable x_i must be normally distributed.

However, all these assumptions can be further relaxed, with the drawback that the learned parameters of the linear model are inaccurate or the learning process takes longer.

4 Concept

In this chapter, a concept to predict the execution time of 3D rendering commands using machine learning is presented. At first, the selection of the features and how to obtain them is discussed. Further, the required linear model and their persistent storage is presented. This is followed by the learning on GPU instruction level, online machine learning and their problems.

4.1 Selection of Features

The selection of the features is one important aspect, because the duration for the learning progress and the representation of the real world depend on the features. Thus, the given information need to be analyzed with respect to the execution time. To determine the impacts on the execution time, we are looking on influence factors in the OpenGL ES 2.0 processing pipeline. The pipeline was presented in Chapter 3. To predict the execution time a linear model is applied. The selection of the features will lead to the parameter of the linear model.

In OpenGL, there are two different type of commands. On the one hand, commands that are using the GPU and on the other hand, commands that only change the internal state of OpenGL. Only OpenGL commands which are using the GPU are considered for the execution time prediction.

Draw Command (glDrawArrays and glDrawElements)

The draw commands are processed by the OpenGL pipeline, which passes several steps.

1. Per-Vertex Operations. This stage processes each vertex using the vertex shader on its own. Therefore, the execution time of this step depends on the number of vertices and the execution time of the vertex shader.
2. Rasterization. The primitives are transformed to fragments and processed by the fragment Shader. The execution time of this step depends on the number of fragments and the execution time of the fragment shader.

3. Per-Fragment Operations. At this stage, each fragment is processed in order to obtain the color for the framebuffer. The tests can be enabled or disabled. Therefore, the execution time depends, on the one hand, on the number of fragments and on the other hand, the execution time for each test.

Modern graphics cards support unified shaders. That means, that the graphics card provides several threads to which the shader can be assigned. Thus, the number of parallel executions of shaders can vary [11].

The following equation shows the execution time of the draw command considering the OpenGL pipeline. t denotes the execution time, and N the number of shader instances. The index vs denotes the vertex shader and the index fs the fragment shader. $tests$ are the tests, which are processed for each fragment. In t_{test}, the execution times of all tests are accumulated.

$$t_{draw} = t_{vs} \cdot N_{vs} + t_{fs} \cdot N_{fs} + t_{tests} \cdot N_{fs} \tag{4.1}$$

Clear Command (glClear)

The clear command sets all pixels in the framebuffer to an initial configured value. The execution time of this command depends on the number of written pixels in the framebuffer. The framebuffer contains the color, depth, and stencil information. This information can be cleared either on its own, in combinations or all at once. Therefore, it is a difference, if first the color and then the depth information is cleared, because clearing color and depth information at the same time requires less execution time.

$$t_{clear} = (t_c + t_d + t_s + t_{cd} + t_{cs} + t_{ds} + t_{cds}) \cdot N_{pixels} \tag{4.2}$$

Where the sum is the execution time to clear one pixel and N_{pixels} is the number of pixels, that are effected by the clear command. The indexes c, d, and s denote color, depth, and stencil respectively. For example, the color and depth information is cleared, thus, the variable t_{cd} is taken into account.

Flush Command (glFlush)

The flush command states that all previously sent commands must be completed. This is an operation which is propagated to the graphics card and takes a constant execution time c_{flush}.

$$t_{flush} = c_{flush} \tag{4.3}$$

Swap Buffer Command (eglSwapBuffer)

The swap buffer command is not part of the OpenGL specification and is an improvement in performance. Instead of rendering directly to the device buffer, at first, the object is rendered to a renderbuffer that will be linked to the device with the swap buffer command. The swap buffer command is swapping the two buffers and copies the memory from the renderbuffer into the framebuffer. A swap buffer commands implies a flush command on the context before copying the renderbuffer into the framebuffer. The execution time of the swap buffer command depends on the number of pixels and the duration to copy one pixel.

$$t_{swap} = t_{pixel} \cdot N_{pixels} \tag{4.4}$$

In this context, t_{pixel} is the execution time to copy one pixel and N_{pixels} the total number of pixels that are swapped.

4.2 Learning on GPU Instruction Level

In this section, the use of GPU instructions for machine learning is discussed. When using a shader identifier as feature, the learning process is only done for this particular shader. When the online learning runs with a program, to which a so far unknown shader is linked, the learning process starts without knowledge and needs to learn the weights from scratch for the new shader. Thus, the prediction of the execution time for new shader leads to high prediction errors in the beginning. Another disadvantage is, that each shader needs a new weight, which leads first to a higher memory usage and second, the feature vector gets more entries and therefore, the execution time for online learning increases.

However, when learning on GPU instruction level is applied, the learning process is done for each GPU instruction. The number of GPU instructions has in comparison to the number of possible shaders a smaller upper bound. The amount of instructions depends on the GPU. The number of required features for shaders is reduced by the use of GPU instructions.

Another advantage for learning on GPU instruction level is, that jump commands in the shader can be detected. Because of jumps, the number of instructions cannot be predicted anymore. Jumps are generated by if-clauses or loops. In the case of loops, it is possible that the optimizer unrolls the loops. If the number of iterations is unknown or the internal memory has an insufficient size to unroll the loop, the GPU instructions contain a jump. In this case, the number of actual GPU instructions cannot be determined and hence, no worst case execution time can be predicted. Without knowing the worst case execution time, the real-time GPU scheduler cannot hold the deadlines of all tasks.

In the following the GPU instruction code of the used graphics cards is presented.

4.2.1 Nvidia Instruction Set Architecture

As graphics card the Nvidia Quadro 400 (GT216GL) is used. This graphics card belongs to the NV50 family, which has the Nvidia 3D object codename Tesla. The NV50 family has subcategories where the Quadro 400 is part of. The subcategory is NVA5 and is also called GT216 [12] .

The Tesla family uses the Tesla CUDA ISA, which stands for Completely Unified Device Architecture Instruction Set Architecture. All types of shaders use nearly the same ISA and therefore, it is possible to execute the shaders on the same streaming multiprocessors [24].

Tesla ISA is stored in 32-bit little-endian words. Each instruction can be either short (1 word) or long (2 words). To distinguish which instruction is a short and which is a long one, the bit 0 of the first word is either set or not. In Table 4.1, it can be seen, that there are two short and five long instruction types.

Table 4.1: Tesla ISA instruction type [24]

Word 0 Bits 0-1	Word 1 Bits 0-1	Instruction Type
0	-	short normal
1	0	long normal
1	1	long normal with join
1	2	long normal with exit
1	3	long immediate
2	-	short control
3	any	long control

In word 0, the first word, the bits 28-31 define the primary opcode field. If the instruction is a long instruction, the secondary opcode field is in word 1, the second word, bits 29-31. Respectively this information, an opcode can be disassembled in its instruction. A map from the opcode to instruction name can be seen in the Appendix A.2. To determine the instruction of a given opcode, the instruction type, the primary opcode and depending on the instruction type, the secondary opcode is necessary [24].

In the following, it will be shown, how the instruction of the given opcode `0xe0850605 0x00204780` is obtained. At first, the instruction type is determined. Therefore, bits 0-1 of word 0 are checked. In the case that word 0 is `0xe0850605`, the bit 0 is set and thus, this is a long instruction. To specify the long instruction bits 0-1 of word 1 are checked. These are 0 and thus, the instruction type is long normal. The primary opcode is given by the four highest bits of word 0 `0xe0850605` and because we are facing a long instruction, the secondary opcode is given by the three highest bits of word 1 `0x00204780`. This leads to a primary opcode of `0xe` and a secondary opcode of `0x0`, which finish up in the instruction *fmul+fadd*. This instruction name can be taken from Table A.1 out of row 15 column *long normal, secondary 0*.

4.2.2 Vivante GC2000

As graphics card on the embedded board i.MX6Quad manufactured by Freescale, a Vivante GC2000 GPU is integrated. For this GPU we are using the opcode, which is used by the proprietary driver. An enumeration of the opcodes is depicted in Section A.1.

4.3 Linear Model

In this section, the linear model using the obtained features is explained. First, the linearity of the features needs to be checked. That means, that each feature has a linear dependency on the execution time. In Section 4.1, the execution time model of the important commands draw, clear, flush, and swap is listed. This model does not consider different shaders and is improved by the concept which is explained in Section 4.2, where the level of learning is transferred from shader level to GPU instruction level. In the following, the figured out features are denoted in the form, that can be used be machine learning.

Our basic linear model without naming the features is

$$\text{ET}(x) = \sum_{i=0}^{N} x_i \cdot w_i \text{ with } x_0 = 1, \tag{4.5}$$

where the ET (Execution Time) can be obtained by the model parameters w and the independent variables x. Which independent variables are picked is discussed in the following for each command separate.

Draw Command

The draw command gets, because of the more fine-grained learning level from the previous section, more complex. The number of variables increased through the improvement explained in Section 4.2. Therefore, the part of the linear model, which implies from the draw command is the following.

$$\mathrm{ET}_{draw}(vs, fs, t) = \sum_{i=1}^{N}(N_{vs} \cdot vs_i + N_{fs} \cdot fs_i) \cdot w_i + \sum_{j=N+1}^{N+M}(N_{fs} \cdot t_j) \cdot w_j \tag{4.6}$$

N is the number of GPU instructions, N_{vs} is the number of vertex shader instances, N_{fs} is the number of fragment shader instances and fs and vs are the distribution of commands of the current vertex shader and fragment shader. Thus, vs_i is the number of GPU instruction i in the used vertex shader. For example, $vs_0 = 5$ means, that 5 GPU instructions of type 0, e.g. `MOV`, per vertex shader exist. The second sum models the execution time for the per-fragment operations, where M is the number of per-fragment tests and t_i is set 0 or 1, respectively if the is activated or not.

Clear Command

The execution time of the clear command depends on the combination of buffers that are cleared and the number of pixels that are effected. In this case each combination gets one weight assigned, which leads to the equation

$$\mathrm{ET}_{clear}(N_{pixels}, c) = (N_{pixels} \cdot c_i) \cdot w_i \tag{4.7}$$

Where N_{pixels} is the number of pixels, which are effected by combination i. If the combination i is activated, c_i is 1, otherwise 0.

Flush and Swap Buffer Command

The flush command has a constant execution time, which yields to a factor f, which is either 0 or 1. The swap buffer command depends on the number of pixels N_{pixels}, that are swapped. Thus, the following two equations for the flush and swap buffer command are denoted.

$$\mathrm{ET}_{flush}(f) = f \cdot w \tag{4.8}$$

$$\mathrm{ET}_{swap}(N_{pixels}) = N_{pixels} \cdot w \tag{4.9}$$

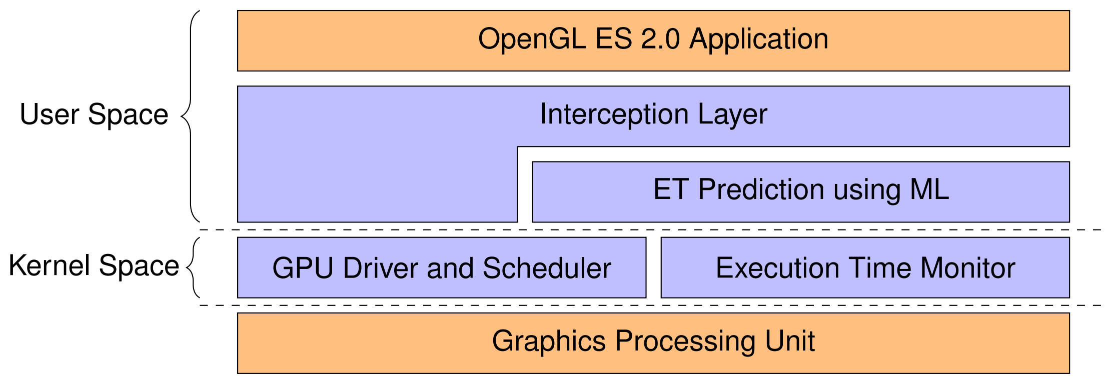

Figure 4.1: Architecture of the Prediction Framework

Summarize

To get the entire linear model, the single commands need to be summed up to accomplish the entire model, which leads to the following equation. Thus, for each occurrence of one of the commands explained before, the execution time model sums up all the input variables to predict the execution time.

$$\text{ET}_{cg} = \sum_{c \in cg} \text{ET}_c \tag{4.10}$$

Where the sum is adding up all execution times of command c from the command group cg. To obtain all the needed parameters, the OpenGL state needs to be tracked. This is explained in Section 4.4.

4.4 Prediction Framework

In this section, the prediction framework is presented. Therefore, the architecture of the prediction framework, how the required parameters for the linear model are obtained, the measuring of the execution time of command groups, and the communication between the single tasks are introduced.

4.4.1 Architecture

The architecture of the prediction framework can be seen in Figure 4.1. The framework bases on the work from Schnitzer et al. [21] and is extended by the functionality of the prediction using machine learning. The two boxes, which are marked with an orange background, are untouched by the architecture and are the boundaries of the prediction framework. The boxes with blue background are modified or inserted for the sake of the prediction framework.

The architecture, see Figure 4.1, contains out of three main areas. First, the user space, where the OpenGL ES 2.0 application and the execution time prediction is running. Second, the kernel space, in which the GPU driver and scheduler, and the execution time monitor is running. Third, the hardware, which is the graphics processing unit itself.

Flow of Action

In this subsection, the general procedure of the prediction framework is explained. The OpenGL ES 2.0 application, which is programmed by a user, is using OpenGL API calls to render a scene. The application itself stays how it is and does not need to be modified for the framework. The API calls from the application are intercepted by the Interception Layer. The Interception Layer is the first stage of the framework. It parses the parameter of the OpenGL ES 2.0 API calls and build up its own OpenGL state. Therefore, at any time, the local OpenGL state can be requested and used for prediction.

At the time of interception, the prediction library is called and uses the current parameter of the API call and the local collected OpenGL state to predict the execution time for the intercepted GPU command. The predicted execution time is stored, thus, the GPU scheduler can use it to schedule the GPU command groups, so that the task can comply its requirements and no deadlines are violated.

After intercepting the API call, the original OpenGL API is called to do not manipulate the conventional execution of the API call. Thus, the call is forwarded to the GPU Driver and Scheduler.

When using machine learning, the real execution time is required. Therefore, the Execution Time Monitor measures the execution time for the GPU commands. GPU commands are grouped into GPU command groups. A GPU command group is written into the GPU command buffer and will be executed not preemptively by the GPU. Reverse, the measured execution time can contain several GPU commands.

4.4.2 Execution Time Prediction using Machine Learning

The execution time prediction using machine learning estimates the execution time of 3D rendering commands. The internal structure can be seen in Figure 4.2. The outer box Execution Time Prediction using Machine Learning is the main component, which contains three smaller components: Linear Regression, Shader Analyzer and Number of Fragments Estimator.

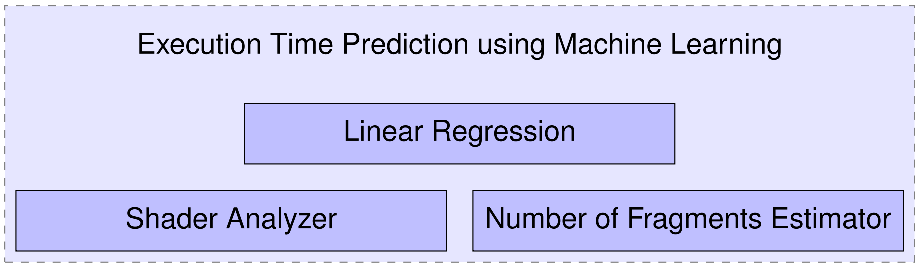
Execution Time Prediction using Machine Learning
Linear Regression
Shader Analyzer
Number of Fragments Estimator

Figure 4.2: Execution Time Prediction using Machine Learning

To improve the weights of the linear model, the real execution time needs to be acquired. Because the GPU buffer holds usually a bunch of GPU commands, the measured execution time is the execution time for the entire GPU command group. Therefore, for learning the linear model, a mapping of multiple single GPU commands to a GPU command group has to be created.

The Execution Time Prediction using Machine Learning stage contains of four substages.

Machine Learning. This is the interface to the boundaries. It communications with the Interception Layer to collect all necessary information for predicting, contacts the Execution Time Monitor to obtain the real execution time and is organizing the mapping of single GPU commands to GPU command groups.

Linear Regression. This module contains the mathematical solution for the linear regression problem. An improved version of the stochastic gradient descent algorithm is applied. This algorithm is explained in Section 4.5.

Shader Analyzer. The Shader Analyzer examines the shader and creates on the one hand a distribution of the GPU instruction types the shader is using and on the other hand, a jump detection is done. This concept is explained in Section 4.2.

Number of Fragments Estimator. One unknown parameter in this system is the number of fragments. The number of fragments is necessary to determine the execution time for draw calls. To estimate the number of fragments using the vertex data and the vertex shader, Schnitzer et al. [21] proposed the method bounding box. This method to calculate the number of fragments is also used in this work.

4.5 Online Learning

When using online learning, there are requirements for the trainings data set to have a convergence of the algorithm. In the following, problems which need to be solved are listed.

Normalization. Each feature can be scaled differently. In machine learning each scale should be in the same range. In batch learning all data is known beforehand and therefore, the scaling of the trainings data for each feature can be done in advance. However, in online learning no trainings data is given in the beginning and hence, no scaling can be done beforehand [20].

Adaption of Learning Rate. In the simple approach, listed above, the learning rate depends only on the time. To boost the convergence of the error function to zero, the learning rate can be adapted with respect to the gradient [1].

Regularization. To avoid overfitting, regularization can be applied. Overfitting means, that the model fits perfect for the trained samples, but has poor generalization power. This can be avoided using regularization. In this work, regularization is done with the truncated gradient approach [10] and explained in Section 4.5.4;

Sparsity. A data set is sparse, if the most elements are zero. Linear regression is used for applications, where the number of independent variables are larger than the number of dependent variables. In this work, we have several independent variables and only one dependent variable. Often, not all independent variables are influencing the depend variable. Thus, the goal is to get sparse weights, where only the relevant weights are set. The truncated gradient approach [10] is combining regularization and sparsity of the weights.

How these problems are tackled is discussed in the following.

4.5.1 Conceptional Algorithm

In this subsection, the conceptional algorithm is introduced. The parts of this algorithm are explained in the following subsections. The algorithm can be seen in the listing Algorithm 4.1. The main loop starting in line 2 is processing all observed samples, where x are the independent variables, y are the dependent variables, and t the time step. In this problem statement, one observed sample is one GPU command group with the feature values x and the real execution time y. In lines 3 and 4 the predicted execution time $\hat{y}$ and the gradient g is calculated. The gradient represents the error of the prediction. After that, in line 5 the normalization of the observed feature values x is performed.

The loop over all feature starting in line 6 calculates a reasonable learning rate η_i and updates the weight w_i for each feature. The functions NORMALIZATION, LEARNINGRATE and WEIGHTUPDATE are explained in the following sections. The rescale of the learning rate η_i is applied, because it shifts the influence of the input parameter x on the update to the learning rate η.

Algorithm 4.1 Stochastic gradient descent

```
1: $w_i \leftarrow 0$
2: for all time steps $t$ observe sample $(x, y)$ do
3:     $\hat{y} \leftarrow \sum_i w_i x_i$
4:     $g \leftarrow \frac{\partial L(\hat{y}, y)}{\partial w}$
5:     $(w, g, N) \leftarrow$ NORMALIZATION$(w, g, x)$
6:     for all feature $i$ do
7:         $\eta_i \leftarrow \begin{cases} \text{LEARNINGRATEALMEIDA}(\eta_i, g_i, t), & \text{for } activeAlg = ALMEIDA \\ \text{LEARNINGRATEADADELTA}(\eta_i, g_i, t), & \text{for } activeAlg = ADADELTA \end{cases}$
8:         $\eta_i \leftarrow \eta_i \frac{t}{N}$
9:         $w_i \leftarrow$ WEIGHTUPDATE$(w_i, \eta_i, g_i)$
10:    end for
11: end for
```

4.5.2 Normalization of Features

In this section, it is discussed how normalization is applied in our stochastic gradient descent setting. Like written before, normalization is crucial to converge to the optimum. Without normalization the feature scale would dominate the importance of a feature. Ideally, the learning rate η is the only impact on the importance of the current sample. To put this into effect, the normalization is necessary. The data set we are using is generated at run time. Therefore, the samples need to be normalized at runtime as well.

Ross et al. [20] proposed the Normalized Gradient Descent (NG) algorithm, which has scale invariance in every feature for online learning. The normalization part of the NG algorithm can be seen in the listing Algorithm 4.2. In the first line, the global variables are initialized. The maximum value for each feature value s_i. In plain gradient descent, the influence of the update of the model weights w depend on the input variable x. To shift this influence of x to the learning rate η_t, the variable N is used. The main idea, to obtain scale invariance for each feature, is that the maximum magnitude of each feature is calculated. With this maximum magnitude, the input value for the respective feature is rescaled and a normalized update can be applied.

Algorithm 4.2 Normalization used by stochastic gradient descent

```
1:  $s_i \leftarrow 0, N \leftarrow 0$
2:  function NORMALIZATION($w, g, x$)
3:      for all features $i$ do
4:          if $|x_i| > s_i$ then
5:              $w_i \leftarrow \frac{w_i s_i^2}{|x_i|^2}$
6:              $s_i \leftarrow |x_i|$
7:          end if
8:          $g_i \leftarrow \frac{g_i}{s_i^2}$
9:      end for
10:     $N \leftarrow N + \sum_i \frac{x_i^2}{s_i^2}$
11:     return $(w, g, N)$
12: end function
```

The main loop in line 3 of Algorithm 4.2 is executed for all features. As first step from line 4 to 7 the maximum magnitude s is calculated and the weights w are adapted respectively. Second, in line 8, the rescaling of the gradient g is performed. Third, in line 10, the change in prediction N is calculated. Using the calculated N, the gradient will be rescaled and the influence of the input variable x is shifted to the learning rate η. At last step, the adjusted weight w and gradient g are returned.

4.5.3 Adaption of Learning Rate

In this section, we discuss how the learning rate is determined. The learning rate specifies how strong the current weight update is taken into account. As simplest, the learning rate is decreasing over time, like shown in Subsection 3.2.3. The effect of this annealing learning rate is, that the ability of adaptation after numerous iterations is reduced. If the trainings data set is randomly distributed, no problem comes up. However, our trainings data set is not random distributed and it can happen that one of the features show up very late. To still have the opportunity to react on the new seen feature and be adaptive, each feature can get its own learning rate.

Almeida

Almeida et al. [1] proposed a general method of parameter adaption in stochastic optimization, which is applicable to stochastic gradient descent. The main idea is to consider the last two gradients, instead of using the time step to determine the new learning rate $\eta_i^{(t)}$, where i is the index of the feature vector and subscript (t) is the tth iteration.

To update the new learning rate $\eta_i^{(t)}$ the update rule

$$\eta_i^{(t)} = \eta_i^{(t-1)} \cdot \left[1 + k \frac{d_i^{(t)} d_i^{(t-1)}}{v_i^{(t)}} \right] \tag{4.11}$$

is applied, where k is a step size factor, d is the gradient of the loss function $\nabla L(y, x)$ and v an exponential average of the square of $d_i^{(t)}$, determined by

$$v_i^{(t)} = \gamma v_i^{(t-1)} + (1 - \gamma) \cdot \left[d_i^{(t)} \right]^2, \tag{4.12}$$

where γ is the weight of the average. A high value for γ leads to low pass filter and a low value for γ to a high pass filter.

This method of Almeida et al. follows two approaches. First, if the last two gradients point into the same direction, the learning rate η will increase. This is realized by multiply then two deviations. If the sign of the deviations is the same, the result is positive, if not, it is negative and the learning rate will decrease. Second, the change of the learning rate is proportional to the gradients magnitude. For this reason, this rule is also called normalized update rule [1].

In Algorithm 4.3, the function LEARNINGRATEALMEIDA is listed. The function contains the calculation of the moving average v and the update of the learning rate η for each feature index i.

Algorithm 4.3 Learning rate calculation based on Almeida et al.

```
   Global Parameters: $\gamma$, $k$
   Global Initialization: $v_i \leftarrow 0$
1: function LEARNINGRATEALMEIDA($\eta_i, g_i, t$)
2:     $v_i \leftarrow \gamma v_i + (1 - \gamma) \left[ g_i^{(t)} \right]^2$
3:     $\eta_i \leftarrow \eta_i (1 + k \frac{g_i^{(t-1)} g_i^{(t)}}{v_i})$
4:     return $\eta_i$
5: end function
```

ADADELTA

Zeiler proposed the method ADADELTA to adapt the learning [27]. This method results from an improvement of ADAGRAD [5]. ADADELTA follows two main ideas, first, the denominator is a local estimate and it not an infinity sum, like for the simple approach for calculating the learning rate (see Section 3.2.3). The other idea is to use an approximation of the Hessian matrix to obtain the same units for numerator and denominator, and have second order derivative information.

In order to find the step Δw to improve the weights, the following equation is applied.

$$\Delta w = \frac{RMS(\Delta w)^{(t-1)}}{RMS(g)^{(t)}} g^{(t)}, \tag{4.13}$$

where g is gradient, and the tth iteration is denoted by subscript (t). RMS is the root mean square and is defined by

$$RMS(x) = \sqrt{E(x) + \epsilon}. \tag{4.14}$$

The root mean square takes an exponential decaying average, which is calculated over a window.

$$E(x)^{(t)} = \gamma E(x)^{(t-1)} + (1-\gamma)x^2 \tag{4.15}$$

The learning rate can then be determined by dividing the descent step by the gradient:

$$\eta = \frac{\Delta w}{g} \tag{4.16}$$

The applied algorithm is listed in Algorithm 4.4. In line 2 the exponential decaying average of g_i is calculated. In the following line, the step Δw is calculated. Therefore, the root mean square of Δw_i and g_i are considered. For $RMS(\Delta w_i^2)$ only all previous Δw_i are taken, because the step size of the current step is not set so far. After getting the step size Δw_i, the exponential decaying average of Δw_i is calculated in line 4. As last step, the learning rate is return.

Algorithm 4.4 Learning rate calculation based on ADADELTA

```
   Global Parameters: $\gamma, \epsilon$
   Global Initialization: $E[g_i^2] \leftarrow 0, E[\Delta w_i^2] \leftarrow 0$
1: function LearningRateADADELTA($g_i$)
2:     $E[g_i^2] \leftarrow \gamma E[g_i^2] + (1-\gamma) g_i^2$
3:     $\Delta w_i \leftarrow \frac{RMS(\Delta w_i^2)}{RMS(g_i^2)} g_i$
4:     $E[\Delta w_i^2] \leftarrow \gamma E[\Delta w_i^2] + (1-\gamma)\Delta w_i^2$
5:     return $\frac{\Delta w_i}{g_i}$
6: end function
```

4.5.4 Regularization and Sparsity

Regularization and sparsity are correlated to each other. Regularization adds penalization to the loss function. The simple loss function in Section 3.2.3 can be extended by regularization term. In the following equation an L_1-regularization can be seen.

$$L(y,x) = \frac{1}{2}(y - x^T w)^2 + \lambda |w|_1 \tag{4.17}$$

Through this regularization term, non-zero weights are penalized and larger weights are even more penalized. Therefore, it is more likely that a weight is zero and the weight vector is sparse. Another aspect is, that the regularization term is used to decrease the chance of over fitting.

Langford et al. proposed sparse online learning via truncated gradient, which covers an improved version of the L_1-regularization [10]. The basic concept is to truncate the gradient, if the weight update would result into a weight, which lies in the interval $[-\theta, \theta]$. Instead of rounding the weight directly to zero, a smoother approach is applied. The equation

$$w_i = T_1(w_i - \eta_i g_i, \eta_i \eta r, \theta) \tag{4.18}$$

shows the usage of the truncated gradient function T_1. The variables r and θ are added by this approach, where r is a gravity parameter and θ the interval range. By rising r and θ the chance, that the solution is sparse, increases.

$$T_1(v_j, \alpha, \theta) = \begin{cases} \max(0, v_j - \alpha) & \text{if } v_j \in [0, \theta] \\ \min(0, v_j + \alpha) & \text{if } v_j \in [-\theta, 0] \\ 0 & \text{otherwise} \end{cases} \tag{4.19}$$

The truncation can only be performed every K iterations, to gain a more aggressive truncation with gravity parameter Kg. Thus, it is more likely to get better sparsity.

The algorithm for the weight update using the truncated gradient function is listed in Algorithm 4.5. The variable g_i is already the gradient. Thus, the gravity parameter g gets the name r in the algorithm description.

Algorithm 4.5 Weight update for online learning

```
    Global Parameters: $K, \theta, r$
1: function WEIGHTUPDATE($w_i, \eta_i, g_i, t$)
2:     $w_i \leftarrow \begin{cases} T_1(w_i - \eta_i g_i, \eta_i r K, \theta) & \text{if } \frac{t}{K} \in \mathbb{N} \\ w_i - \eta_i g_i & \text{otherwise} \end{cases}$
3:     return $w_i$
4: end function
```

4.6 Store the Model Parameters

In this section, the persistent storage of the model parameters is explained. Without a persistent storage, the model is untrained at every new start of an application. Thus, the prediction of the execution time in the beginning is inaccurate. Because of real-time requirements, an inaccurate prediction cannot be used for the GPU scheduler and is pointless. A wrong estimation can violate tasks with real-time restrictions. That implies, that a persistent storage for the parameters is necessary.

Choice of Parameters

The parameters of the linear model are the weights w. On the basis of the facts that the features are normalized and the learning rate is adapted during online learning, the variables for both methods need to be reviewed for a persistent storage as well.

The Normalized Gradient Descent algorithm, which is described in Section 4.5.2, uses the variables w, s, t, $\hat{y}$. The meaning of the variables is described in that section as well. The prediction $\hat{y}$ is calculated before the value is used and hence, a persistent storage of $\hat{y}$ is redundant. All other values are stored persistent.

The calculation of the learning rate proposed by Almeida et al. [1], which is presented in Section 4.5.3, uses v and η for each feature as variables. The learning rate η and the moving average v is changing over time and needs to be stored persistent.

To calculate the learning rate using ADADELTA, the learning rate η does not need to be stored persistent, because this method is determining the learning rate using the exponential decaying average of the gradient g and the step width Δw. These both moving averages need to be stored to have initialized values and have a smooth adaption of the learning rate in the beginning for a new program.

Method to Store Model Parameters

The method to store the model parameter is described in the following. At the initialization of the machine learning module, the model parameters are initialized. After that it is checked, if a previous configuration file with model parameters exist. If previous values for the model exist, they are loaded into the module and are used for further learning. After finishing learning, the values are written to the configuration file. As file format Extensible Markup Language (XML) is chosen. Basically, all variables get a tag assigned and are written consecutive to the file.

5 Implementation

In this chapter, the implementation of the concepts from the previous chapter is explained. First, the used setup and then, the implementation containing the interception layer, machine learning module, and stochastic gradient descent module is presented.

5.1 Setup

The implemented software is designed for two platforms with two different graphics cards. On the one hand, a workstation and on the other hand, an embedded computer which are listed below.

1. Workstation

   Operating System: Linux distribution Fedora with a fully preemptive kernel.

   Graphics Card: Nvidia GT216GL (Quadro 400)

2. Embedded Computer

   Operating System: Linux distribution with a fully preemptive kernel.

   Graphics Card: Vivante GC2000

The software is written in the programming language C and written in a general way to support both graphics cards as best as possible. As 3D rendering pipeline we are using OpenGL ES 2.0. The main differences in the implementation of the two different systems are the following.

Swap Buffer Command. After drawing the primitives into the renderbuffer, the renderbuffer is swapped to display the scene. Depending on the graphics card, the prediction of the execution time for swapping the buffer is implemented in a different way.

Obtaining GPU Code. The GPU instructions of shaders are received on different ways. The information is obtained from the graphics card driver which differs on both systems.

5.2 Interception Layer

The implementation of the interception layer is presented in this section. The implementation of this layer is taken from Schnitzer et al. [21] and extended by the machine learning calls and state tracing. The interception is the first stage of the system. In this stage, the OpenGL API calls are intercepted and used to change the internal stored context. The interception layer contains two layers. One is the general interception of the OpenGL calls. The other is designed to trace the OpenGL state and to forward the correct data to the machine learning layer.

5.2.1 General Interception

The interception of the OpenGL API calls is depicted in Algorithm 5.1. The original OpenGL API call is overwritten with an own function. The function declaration describes a function which is called instead of the real OpenGL API call. In this self-defined function, the real OpenGL API function is called. But the prediction library is called beforehand to be able to trace the state properly. Some OpenGL API calls have parameters, others do not have parameters. If the original function has parameters, they are passed to the prediction library and the real OpenGL API function in the same way. If the OpenGL API call has a return value, it is passed as well.

Algorithm 5.1 General interception of OpenGL calls

```
1: function OPENGLCALL(...)                          // Overwritten openGlCall
2:     ETP_OPENGLCALL(...)
3:     return REAL_OPENGLCALL(...)                   // real openGlCall
4: end function
```

5.2.2 Prediction State Tracing

The state tracing and the forwarding of all necessary information to the machine learning module is described in this subsection. Only the relevant functionality to understand the crucial implementation is explained.

Initialization

At the beginning, applications create an OpenGL context which is used to render the objects afterwards. At this point, the internal context is created and initialized by setting all variables to their default values. At last, the prediction state is initialized as well. This is illustrated in Algorithm 5.2.

Algorithm 5.2 Interception of eglCreateContext

```
1: procedure ETP_EGLCREATECONTEXT
2:     INITIALIZECONTEXT( )
3:     INITIALIZEPREDICTIONSTATE(state)
4: end procedure
```

The OpenGL API calls glCreateProgram, glAttachShader and glUseProgram are implemented respectively, such that the internal state contains the program and the attached vertex and fragment shader after the initialization.

Draw Commands

The execution time model for the draw command of Chapter 4 requires the number of vertices, number of fragments, and the GPU instructions of the vertex and fragment shader. At this point, we obtain the number of vertices and number of fragments. The draw commands are split into glDrawArrays and glDrawElements. Both of these commands provide the number of vertices as parameter count. Algorithm 5.3 line 2 shows the implementation of summing up the number of vertices. To obtain the number of fragments, the bounding box algorithm [21] is used. The bounding box algorithm calculates a bounding box that contains all vertices and uses the vertex shader projection to calculate the framebuffer area which is covered by the bounding box. The bounding box is calculated for all draw calls in one command group, except the Mode-View-Matrix is changed or glFlush is called. Therefore, the bounding box is iteratively constructed in each draw call and can by extended by the data of the vertices. This functionality is called by the function in line 3 of Algorithm 5.3.

Algorithm 5.3 Interception of glDrawArrays

```
1: procedure ETP_GLDRAWARRAY(mode, first, count)
2:     state.numVertices $\leftarrow$ state.numVertices + count
3:     BOUNDINGBOXPREPERATION( )
4: end procedure
```

After constructing the bounding box, the vertex shader projection is applied to obtain the number of fragments. The function ETP_PREDICTDRAWEXECUTIONTIME, listed in Algorithm 5.4, is called when the end of a command group is reached or the Mode-View-Matrix changes. The bounding box is reset in line 3. Finally, the current command group index is obtained and the collected state is passed to the machine learning stage. The machine learning stage is explained in the following section.

Algorithm 5.4 Predicting the execution time at the end of each command group

```
1: procedure ETP_PREDICTDRAWEXECUTIONTIME
2:     state.numFragments ← BOUNDINGBOXFRAGMENTESTIMATION( )
3:     BOUNDINGBOXRESET( )
4:     cg_idx ← GETCURRENTCOMMANDGROUPINDEX( )
5:     ML_ADD_SAMPLE(cg_idx, state)
6: end procedure
```

Clear Command

The clear command requires the number of pixels and the type of buffer that is cleared. The number of pixels are determined by the window dimensions. This is implemented by a multiplication in line 2 of Algorithm 5.5. The window width and height are set at initialization time. The kind of buffer is determined by the bitmask which is passed as a parameter. A mapping from the mask to state variables needs to be created. This is done by adding up an index over the mask such that each index is identifying one combination. This combination is used as switch criteria to assign it to the correct state variable.

Flush Command

The flush command depends only on the number of flush commands. Thus, every time glFLush is called, a state variable is incremented. Therefore, this trivial algorithm is not listed.

Swap Buffer Command

The implementation of the swap buffer command is not included in the graphics card driver itself because the command is not part of the OpenGL specification. On the workstation a X-Server is responsible for the memory transfer. The execution time of the memory transfer has a high variance, because of this, a reasonable prediction with the swap buffer command is not possible. Therefore, the swap buffer command is suppressed and the prediction is not implemented.

However, on the Vivante GPU the execution time can be predicted. The analysis of the swap buffer command is presented in Chapter 4 and considers the number of pixels that

Algorithm 5.5 Interception of glClear

```
procedure ETP_GLCLEAR(mask)
    numPixels ← state.windowWidth * state.windowHight
    index ← 0
    if mask & GL_COLOR_BUFFER_BIT then                    // & ≙ bitwise "and"
        index ← index + 1
    end if
    if mask & GL_DEPTH_BUFFER_BIT then
        index ← index + 2
    end if
    if mask & GL_STENCIL_BUFFER_BIT then
        index ← index + 4
    end if
    switch index do
        case 1: state.clear_color ← pixels
        case 2: state.clear_depth ← pixels
        case 3: state.clear_stencil ← pixels
        case 4: state.clear_color_depth ← pixels
        case 5: state.clear_color_stencil ← pixels
        case 6: state.clear_depth_stencil ← pixels
        case 7: state.clear_color_depth_stencil ← pixels
end procedure
```

are affected by the swap buffer command. Usually, the entire renderbuffer is swapped, but the Vivante GPU driver sends only areas to the GPU that are affected. Analyzing the binary code that is sent to the GPU buffer, the width and height of the area that is swapped is determined. Thus, the state variable is calculated as follows: state.swap ← width * height.

Get Predicted Execution Time

The GPU scheduler requires the predicted execution time when the GPU buffer is filled. At this time, the framework calls the function ETP_PREDICTEXECUTIONTIME with the command group index as a parameter. The function returns the predicted execution time of the passed command group index and is explained in Algorithm 5.6. In line 2, the current state is passed to the machine learning. At this time, all the information that influence the execution time of the current command group are passed to the machine learning stage. In the next step, the machine learning module predicts the execution time for the current command group (see in line 3).

The second part of this function contains the learning process of already passed command groups. The measured execution time is required to learn. From line 5 to 8, the while loop iterates over all command groups that are already sent to the GPU, but were not considered for learning. If such a command group exists, the function TRYGETMEASUREDET tries to get the execution time of the command group. If the measured execution is set by the function, the loop condition is true and the learning of the command group is executed in line 7.

Algorithm 5.6 Predicting the execution time for one command group

```
 1: procedure ETP_PREDICTEXECUTIONTIME(current_cg_index)
 2:     ML_ADD_SAMPLE(current_cg_index, state)
 3:     predictedET ← ML_PREDICT_COMMAND_GROUP(current_cg_index)
 4:     cg_idx ← ML_GET_NEXT_LEARNING_INDEX( )
 5:     while cg_idx != -1 and TRYGETMEASUREDET(cg_idx, measured_ET) = 0 do
 6:         ML_LEARN(cg_idx, state)
 7:         cg_idx ← ML_GET_NEXT_LEARNING_INDEX( )
 8:     end while
 9:     return predictedET
10: end procedure
```

5.3 Machine Learning

The machine learning module is a piece of software that connects the surrounding software components: interception layer, shader analyzer and linear regression. The interception layer is described in the previous section and is calling the machine learning module. The machine learning module is split into three parts. Adding new samples, learning the new added samples and predicting the execution time of command group. These three parts are explained in the following.

Add New Samples

To eventually learn the parameters of the linear model, data needs to be collected. This is done by collecting the state which is depicted in Algorithm 5.7. This function is called by the interception layer. In line 2 the state is transformed to the feature vector which is used by the linear regression. This transformation is described in Algorithm 5.8 and is explained afterwards. After the feature vector is obtained, the feature vector is added to the buffer. The buffer is a module variable and is visible inside the machine learning module. Usually, the function ML_ADD_SAMPLE is called more than once per command group. Then, the buffer contains more than one feature vectors with the same command group index. In line 4, the state is reset and all used state variables are cleared.

Algorithm 5.7 Add new samples to the machine learning module

```
1: procedure ML_ADD_SAMPLE(cg_index, state)
2:     ML_STATE_TO_FEATURE_VECTOR(state, feature_vector)
3:     buffer.add(cg_index, feature_vector)
4:     ML_RESET_STATE(state)                // Initializing values in the state
5: end procedure
```

Algorithm 5.8 shows how the state is transformed to the feature vector. The state is a structured data type, whereby, the feature vector is an array of floating numbers. To create the feature vector, the distribution of the GPU instructions of the vertex and fragment shader are required. The distribution is represented as an array, where the index identifies a GPU instruction and the stored value identifies the number of occurrences in the current shader, e.g., dist[1] = 4 means that the GPU shader instruction 1 (e.g. ADD) exists four times in the shader. The distribution is calculated by the shader analyzer and has to be done once per program. Therefore, in line 2 is a check which checks if the distribution of the current program has not been created. If that is true, the GPU instruction distribution for the program is created.

At this point all information for the feature vector are available. The values for the feature vector are assigned in the following lines of code. If a draw call arose, the constant bias for draw calls is activated by writing a one to the feature vector with the index 0. The following lines of code contain the set of the clear, swap, and flush command up to index 9. The for loop in the end of the function calculates the number of GPU instructions that occur in the draw call. After finishing, the feature vector is set completely.

Predicting the Execution Time

The prediction of the execution time is done by summing up all seen commands in the command group and asking the linear regression module for the prediction. This is depicted in Algorithm 5.9. The function buffer.sumUp in line 2 sums up all feature vectors for the given command group index. In the following line, the prediction of the command group is forwarded to the linear regression module.

Algorithm 5.8 Transformation of the state into the feature vector

```
procedure ML_STATE_TO_FEATURE_VECTOR(state, vector)
    if program.instructionDist = 0 then
        SHADERANALYZER_CREATE_DIST(program)
        program.instructionDist ← 1
    end if
    if state.numVertices > 0 then
        vector[0] ← 1.0                                  // Bias for draw command
    end if
    vector[1] ← state.clear_color
    ...
    vector[7] ← state.clear_color_depth_stencil
    vector[8] ← state.swap
    vector[9] ← state.flush
    for i ← 0 to NUM_GPU_INSTRUCTIONS do
        vector[10 + i] ← state.numVertices * program.vs.dist[i] + state.numFragments * program.fs.dist[i]
    end for
end procedure
```

Algorithm 5.9 Predicting the execution time

```
procedure ML_PREDICT(cg_index)
    feature_vector ← buffer.sumUp(cg_index)
    LR_PREDICT(feature_vector)
end procedure
```

Learning the Model Parameters

The learning of the model parameters requires the collected feature vectors. The collection of feature vectors is done by the call of Algorithm 5.7. Another requirement is the measured execution time from the kernel space. Every time the execution time for one command group is predicted (see Algorithm 5.6), the learning is activated. This requires that the measured execution time is available. Algorithm 5.10 depicts, how the learning is forwarded to the linear regression module. At first, the feature vector for the given command group is summed up. The feature vector and the measured execution time are required to improve the weights of the linear model. In line 4, the just now learned elements from the current command group are removed from the buffer.

Algorithm 5.10 Using the measured execution time to learn the model parameters

```
procedure ML_LEARN(cg_index, measured_ET)
    feature_vector ← buffer.sumUp(cg_index)
    LR_LEARN(feature_vector, measured_ET)
    buffer.removeAll(cg_index)
end procedure
```

5.4 Shader Analyzer

In this section, the shader analyzer is explained. It has two main purposes. First, it creates a distribution over the shader instructions and second, it detects jumps. The functionality is depicted in Algorithm 5.11. The function SHADERANALYZER_CREATE_DIST creates the distribution of the GPU instructions for the fragment and vertex shader. The implementation for the vertex and fragment shader is equivalent. Thus, it is outsourced in an additional function. This function starts in line 5. The for loop iterates over all GPU instructions inside the shader. It first obtains the type of instruction in line 7. Then, it is checked if the instruction is of the type jump. If that is true, the execution time of this shader cannot be predicted for sure. In the second step in line 11, the distribution of the instruction types is created. Thus, each instruction type in the array has a number of occurrences in the given shader assigned.

Algorithm 5.11 Creating the distribution of GPU instruction for a program

```
 1: procedure SHADERANALYZER_CREATE_DIST(program)
 2:     SHADERANALYZER_CREATE_SHADER_DIST(program.vs)
 3:     SHADERANALYZER_CREATE_SHADER_DIST(program.fs)
 4: end procedure
 5: procedure SHADERANALYZER_CREATE_SHADER_DIST(shader)
 6:     for i ← 0 to shader.binary_length do
 7:         instruction ← shader.binary_code[i]
 8:         if instruction is jump then
 9:             ABORT( )
10:         end if
11:         shader.dist[instruction] ← shader.dist[instruction] + 1
12:     end for
13: end procedure
```

5.5 Linear Regression

In the linear regression module, the stochastic gradient descent algorithm is implemented. To hold the variables beyond one function call, they are placed in a global way. Thus, all linear regression function can access them. All variables are hold in the structured data type `LRstate` that is listed in the following.

```
typedef struct {
      double w[LINEAR_REGRESSION_NUM_FEATURES]; // weights
      double s[LINEAR_REGRESSION_NUM_FEATURES]; // normalization: max value of sample
      double eta[LINEAR_REGRESSION_NUM_FEATURES]; // Almeida: learning rate
      double v[LINEAR_REGRESSION_NUM_FEATURES]; // Almeida: exponential average of g^2
      double g_last[LINEAR_REGRESSION_NUM_FEATURES]; // Almeida: g of t-1
      double E_g2[LINEAR_REGRESSION_NUM_FEATURES]; // ADADELTA: exponential average of g^2
      double E_delta_w2[LINEAR_REGRESSION_NUM_FEATURES]; // ADADELTA: exponential average
          of delta w^2
} LRstate;
```

On the one hand, the data type is holding the weights w and the variables for the normalization, on the other hand, it holds variables for the calculation of the learning rate η.

The linear regression module consists of two main functions. The first implements the prediction of a given feature vector and the second is to improve the model parameters. Both are described in the following.

Predicting the Execution Time

The prediction of the execution time is listed in Algorithm 5.12. It simply implements the multiplication of the transposed feature vector x and the weights vector w.

Algorithm 5.12 Predicting the execution time

```
1: function LR_PREDICT(x)
2:     res ← 0
3:     for for i ← 0 to LR_NUM_FEATURES do
4:         res ← res + x[i] * LRstate.w[i]
5:     end for
6:     return res
7: end function
```

Learning the Model Parameters

Learning the model parameters using stochastic gradient descent is depicted in Algorithm 5.13. This function is called by ML_LEARN and provides the feature vector for one group x and the measured execution time y. This function is split into three parts. The first, where the normalization is done (Subsection 4.5.2), the second contains the calculation of the learning rate η (Subsection 4.5.3) and third, the update of the weight vector w (Subsection 4.5.4).

Algorithm 5.13 Learning the model parameters using stochastic gradient descent

```
 1: procedure LR_LEARN(x, y)
 2:     for i ← 0 to LR_NUM_FEATURES do
 3:         if |x[i]| > LRstate.s[i] then
 4:             LRstate.w[i] ← LRstate.w[i] * $\frac{\text{LRstate.s[i] * LRstate.s[i]}}{\text{LRstate.x[i] * LRstate.x[i]}}$
 5:             LRstate.s[i] ← |LRstate.x[i]|
 6:         end if
 7:     end for
 8:     y_hat ← LR_PREDICT(x)
 9:     for i ← 0 to LR_NUM_FEATURES do
10:         if LRstate.s > 0 then
11:             gradient_loss ← (y_hat - y) * x[i]
12:             gradient[i] ← gradient_loss / (LRstate.s[i] * LRstate.s[i])
13:             switch activeLearningRateAlgorithm do
14:                 case Almeida: LRstate.eta[i] ← LR_ALMEIDA(gradient, i)
15:                 case ADADELTA: LRstate.eta[i] ← LR_ADADELTA(gradient, i)
16:             if truncatedGradient = 1 then
17:                 ML_TRUNCATEDGRADIENT(gradient, i)
18:             else
19:                 LRstate.w[i] ← LRstate.w[i] - LRstate.eta[i] * gradient[i]
20:             end if
21:         end if
22:     end for
23: end procedure
```

The first loop is part of the normalization and rescales the weight, if a higher absolute input value is seen. The maximum absolute value s is stored respectively. In line 8, the prediction of the passed feature vector x is calculated. This is required to calculate the gradient in the next steps. The second loop, starting in line 9, is processing each feature on its own and updates its weights in the end. Line 10 let only input values pass, that are set. Thus, the learning is only applied to set features. In line 11 and 12 is the gradient with respect to the feature calculated and normalized.

The second part, the normalization, is implemented in the switch. The linear regression module provides two different variants to calculate the learning rate. One is the method proposed by Almeida et al. [1] and the other one is proposed by Zeiler [27]. Depending on which learning rate algorithm is chosen, the calculate of the learning rate η differs.

After calculating the learning rate, the update phase is reached. If the truncateGradient flag is set, the truncated gradient algorithm [10] is used to perform the weight update, otherwise, the update is performed with the default update rule.

In Algorithm 5.14 the implementation of the learning calculation using the method of Almeida et al. is shown. This method has two parameters γ and k. The parameter γ influences the exponential average of the gradient. The step size of increasing η, if the last and the current gradient point in the same direction, is hold by k. In line 2 of the algorithm listing, the exponential average of the gradient is calculated and in the following line the learning rate η is calculated. To provide the gradient of the previous round, it is stored in g_last at the end of the function before returning the learning rate.

Algorithm 5.14 Implementation of the learning rate calculation based on Almeida et al.

```
   **Global Parameters:** $\gamma$, $k$
1: **function** ML_ALMEIDA(gradient, i)
2:     LRstate.v[i] ← $\gamma$ * LRstate.v[i] + (1 - $\gamma$) * gradient[i] * gradient[i]
3:     LRstate.eta[i] ← LRstate.eta[i] * (1 + $k$ * $\frac{\text{LRstate.g\_last[i] * gradient[i]}}{\text{LRstate.v[i]}}$)
4:     LRstate.g_last[i] ← gradient[i]
5:     **return** LRstate.eta[i]
6: **end function**
```

In Algorithm 5.15 the implementation of the calculation of the learning rate based on the ADADELTA method is presented. The method uses the parameter γ and ϵ, where γ has the same purpose as in the algorithm proposed by Almeida et al. and influences the exponential average of the gradient. The algorithm description of the implementation only slightly differs from the conceptional algorithm. Thus, the algorithm is only explained briefly at this place. RMS represents the root mean square of the variable, e.g., RMS_delta_w is the root mean square of delta_w.

The sparse online learning via truncated gradient is listed in Algorithm 5.16. The code contains a main procedure and the threshold function T1 that is called by the main procedure. The if statement in line 2 guarantees that the truncated gradient function is only called every K time steps. The threshold function only affects the update if the new weight w is between $-\theta$ and θ.

Algorithm 5.15 Implementation of the learning rate calculation based on ADADELTA

```
    Global Parameters: $\gamma, \epsilon$
 1: function ML_ADADELTA(gradient, i)
 2:     RMS_delta_w ← 0, RMS_gradient ← 0, delta_w ← 0, eta ← 0
 3:     RMS_delta_w ← $\sqrt{\text{LRstate.E\_delta\_w2[i]} + \epsilon}$
 4:     RMS_gradient ← $\sqrt{\text{LRstate.E\_g2[i]} + \epsilon}$
 5:     delta_w ← RMS_deta_w / RMS_gradient * gradient[i]
 6:     LRstate.E_delta_w2[i] ← $\gamma$ * LRstate.E_delta_w2[i] + (1 - $\gamma$) * delta_w * delta_w
 7:     eta ← delta_w / gradient[i]
 8:     return eta
 9: end function
```

Algorithm 5.16 Implementation of the truncated gradient algorithm

```
    Global Parameters: $K, \theta, r$
 1: procedure LR_TRUNCATEDGRADIENT(gradient, i)
 2:     if LRstate.t mod K = 0 then
 3:         LRstate.w[i] ← T1(LRstate.w[i] - LRstate.eta[i] * gradient[i], LRstate.eta[i]
    * $K$ * $r$, $\theta$)
 4:     else
 5:         LRstate.w[i] ← LRstate.w[i] - LRstate.eta[i] * gradient[i]
 6:     end if
 7: end procedure
 8: function T1(v, alpha, theta)
 9:     if v > 0 and v < theta then
10:         return MAX(0, v - alpha)
11:     else if v > -theta and v < 0 then
12:         return MIN(0, v + alpha)
13:     else
14:         return v
15:     end if
16: end function
```

6 Evaluation

In this chapter, we evaluate the effectivity of the newly developed machine learning algorithm. Therefore, the evaluation setup is introduced. Afterwards the choice of the machine learning algorithm is evaluated and a comparison of the bounding box algorithm and machine learning algorithm is presented.

6.1 Setup

In this section, the setup for the evaluation is introduced. We are using two different setups to evaluate our algorithm. First, a workstation computer and second, an embedded board.

Setup 1 Workstation computer with an Intel Core i7-3770K CPU running with 3.50 GHz and Nvidia GT216GL (Quadro 400) revision a2 as graphics card using the nouveau GPU drivers. As operating system, the Linux distribution Fedora with a fully preemptive kernel is running.

Setup 2 Embedded Freescale SABRE Board for Smart Devices with an i.MX6Q rev1.2 processor equipped. The processor runs with 792 MHz. The board uses 1 GiB of DRAM and has the GPU Vivante GC2000. As graphics card driver we are using the proprietary driver provided by Freescale. The operating system is a fully preemptive Linux distribution.

6.2 Choice of Stochastic Gradient Descent Setup

In Chapter 4 are different options for the stochastic gradient descent algorithms presented. The learning rate is either calculated using the method of Almeida et al. or using the ADADELTA method. Further, there is the option to use truncated gradient to gain regularization. To evaluate which of the options suit best for the developed feature set, a scenario is required in which all features come into effect. The most of the features are caused by adding the number of each GPU instruction to the feature vector (4.2).

Thus, a scenario with changing shaders gives the best possibility to evaluate with which options we get best results. In the following, the four options are listed and in the next subsection, the method to create a scenario with changing shaders is explained.

Almeida. Using the method of Almeida et al. to determine the learning rate. Do not use regularization.

Almeida and Truncated Gradient. Using the method of Almeida et al. to determine the learning rate and use regularization with the truncated gradient method to gain sparse resulting model weights.

ADADELTA. Using ADADELTA to determine the learning rate. Do not use regularization.

ADADELTA and Truncated Gradient. Using ADADELTA to determine the learning rate and use regularization with the truncated gradient method to gain sparse resulting model weights.

We use $k = 0.01$ and $\gamma = 0.9$ for the algorithm proposed by Almeida et al. For ADADELTA we used $\epsilon = 1E - 5$ and $\gamma = 0.9$ to evaluate our solution. The truncated gradient algorithm uses the parameter K, r and θ, where we set $\theta = \infty$ such it is proposed in the publication [10], $K = 10$ and the gravity parameter r is set to 0.001.

6.2.1 Generation of Shaders

In this subsection, we describe which scenario we create to decide which algorithm suits best. Therefore, we developed an application that draws a random number of vertices. The number of vertices is chosen between $300\,000$ and $600\,000$ randomly. To gain a single draw call in one command group, a flush is sent immediately after the draw call. Thus, the command group only contains a draw and a flush call. Additionally, with a probability of 20%, a separate flush command is sent. And with a probability of 10% a clear command that clears a random buffer combination is sent.

We want to avoid to generate fragments through our scenario, because an error in the heuristic to estimate the number of fragments would distort the evaluation. To achieve this, we only create a vertex shader that only generates vertices that are outside the clip volume. Thus, no fragments are created. For each draw call a separate vertex shader is generated that contains one or two different OpenGL Shading Language (GLSL) commands. So far only eleven different GLSL commands are chosen, but this can be extended easily. Each of the GLSL shader commands occurs 10 to 20 times in the shader. For further reference, we call this application "shadergen".

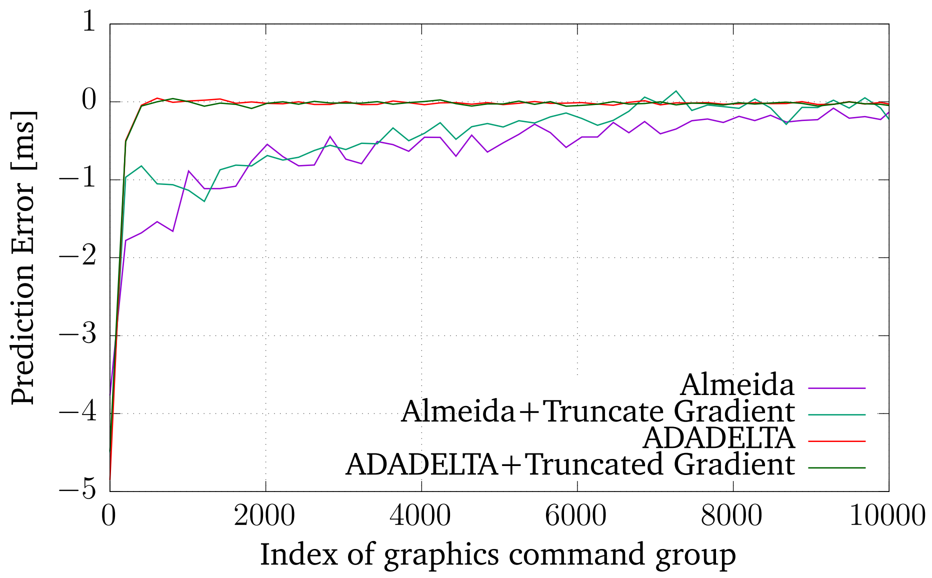

Figure 6.1: Convergence of machine learning algorithms using the application "shadergen" on the Nvidia GPU.

6.2.2 Convergence of Machine Learning Algorithms

Using the just introduced application shadergen we can evaluate how strong the different machine learning algorithms converge. We let the four different options of algorithms run the application for 10k command groups and measured the prediction error. In Figure 6.1 and Figure 6.2 is the Bézier curve over the error for each algorithm depicted. Figure 6.1 shows the convergence on the Nvidia GPU and Figure 6.2 for the Vivante GPU respectively. The ADADELTA algorithm leads on both platforms to better performance in respect to convergence. This approach of determining the learning rate is more stable as well.

From Figure 6.1 and Figure 6.2 it cannot be determined whether the truncated gradient algorithm comes with advantages or not. Further it is not possible to see how accurate the ADADELTA algorithm is. This is considered in the next subsection.

6.2.3 Accuracy

The accuracy reflects how precise the prediction of the execution time is. To show the accuracy we use the cumulative distribution function (CDF) of the prediction error. The scenario is the following: In opposite to the previous evaluation, we use an initialized weight vector. The weights are taken from the ADADELTA algorithm of the previous evaluation.

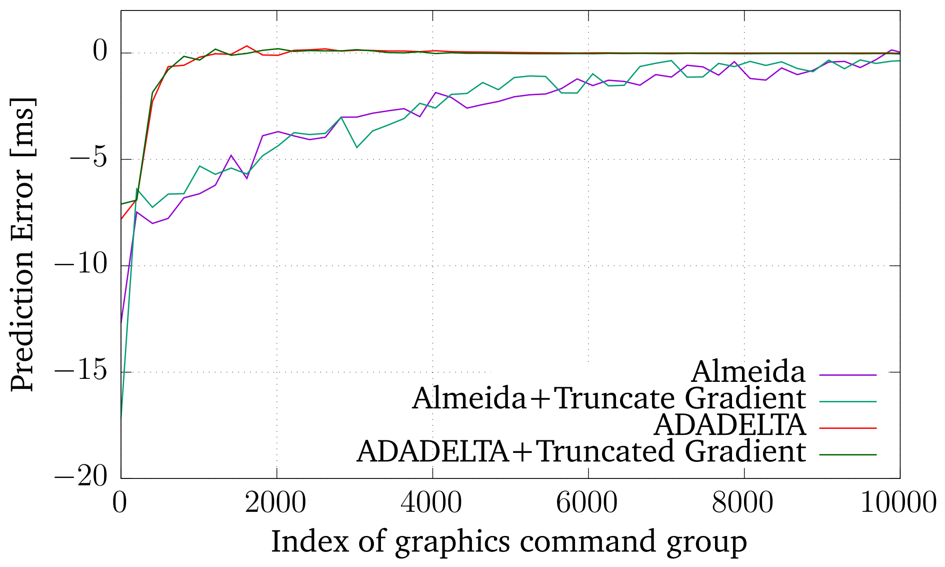

Figure 6.2: Convergence of machine learning algorithms using the application "shadergen" on the Vivante GPU.

Figure 6.3 and Figure 6.4 depict the CDF of the Nvidia and Vivante GPU respectively. It can be seen, that the truncated gradient algorithm reduces the ability of the ADAGRAD algorithm. Especially on the Vivante GPU is the impact large. A reasonable explanation for that is, that our weights vector is already sparse and we only consider weights that are influencing the predicted execution time. Thus, we drop the L_1-regularization preliminary in this evaluation.

Furthermore, we can observe that the prediction error on the Nvidia GPU is larger than on the Vivante GPU, although the execution time of each command group is shorter on average. On the Nvidia GPU, one command group has an execution time of around $1ms$ to $13ms$ and on the Vivante GPU, one command group needs around $16ms$ to $42ms$. The differences are the underlying hardware and the graphics card driver that creates the shader binary code. We observed differences in how instructions are mapped between GLSL and shader binary code. One of the reasons for worse results on Nvidia than on Vivante could be caused by the mapping between GLSL and shader binary code. This can induce linear dependencies between shader instructions and violates the assumption of linear regression. Thus, it is possible that the stochastic gradient descent algorithm stucks in a local minimum. Further, the hardware architecture of the Nvidia graphics card is a black box where techniques to improve the execution time could such as buffers are implemented. These effects could also harm the linear model assumptions.

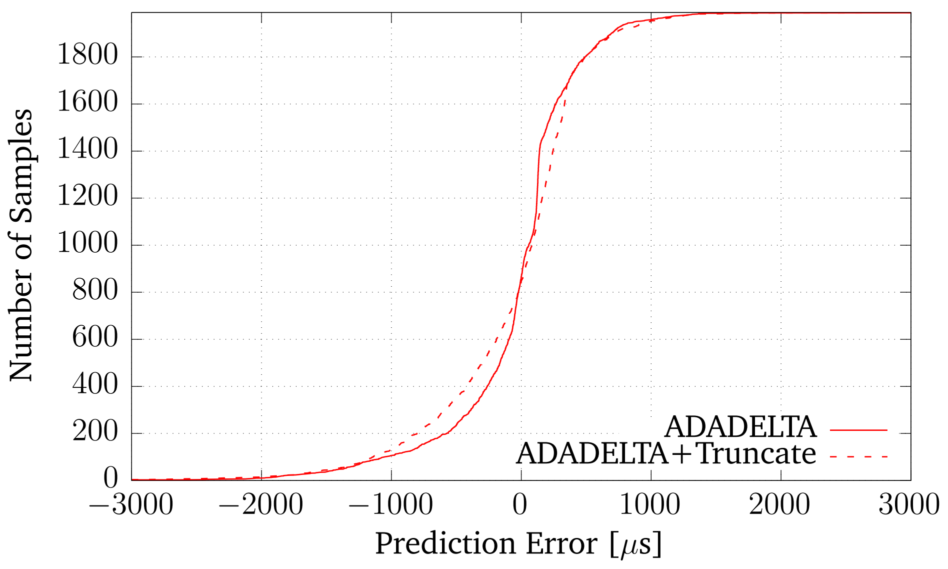

Figure 6.3: Nvidia: Comparing accuracy of machine learning algorithms using the program "shadergen".

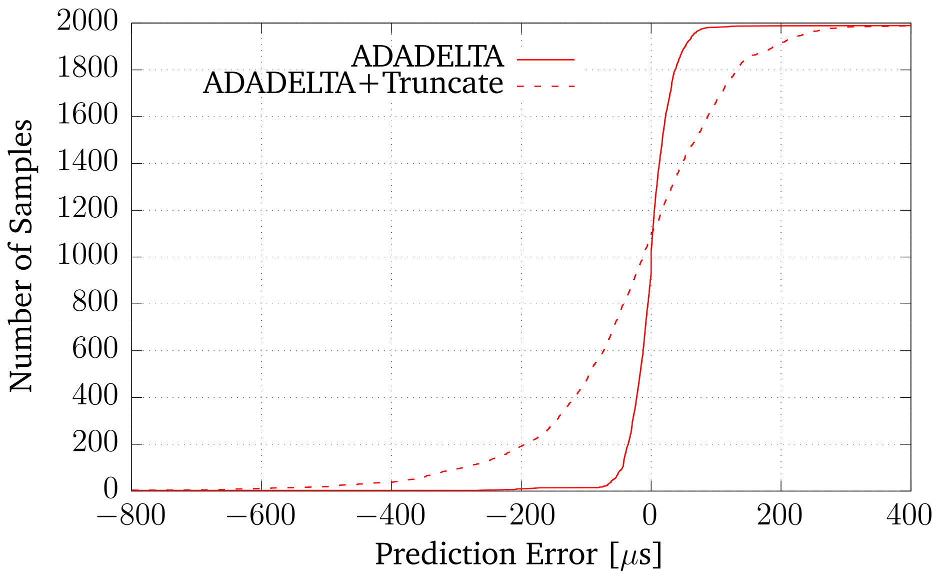

Figure 6.4: Vivante: Comparing accuracy of machine learning algorithms using the program "shadergen".

6.2.4 Execution Time of GPU Instructions

Using the evaluation results from previous subsection we can obtain the execution time of the GPU instructions. Each weight in the weight vector represents the execution time for this feature. Because we are using the GPU instructions as features, we can extract the weights after executing the previous evaluation. The figured out execution time for the GPU instructions of the Nvidia GPU are listed in Table 6.1. Whether it needs to be considered, that these execution times are uncertain, because the CDF in Figure 6.3 shows a not negligible prediction error.

Table 6.1: Estimated Execution Time of Nvidia GPU Instructions

Instruction	Execution Time [μs]	Instruction	Execution Time [μs]
MOV	0.413202	EX2	0.053226
FADD	0.023050	FMIN	0.039195
FMUL	0.017729	FMAX	0.036597
FMADD	0.052592	LOGIC_OP	0.031032
CVT_F2F	0.044275	RSQRT	0.089171
PRESIN_PREEX2	0.053226	FSET	0.033656
LG2	0.080294	CVT_I2F	0.033656

For the Vivante GPU, however, the prediction error is much smaller and therefore, the estimated execution time is more reliable. The estimated execution time for the Vivante GPU is depicted in Table 6.2. Using this information and assuming that the shortest execution time correspond to one GPU cycle, the most of the GPU instructions take one cycle. But RSQ and EXP take 7, and LOG takes 9 cycles.

Table 6.2: Estimated Execution Time of Vivante GPU Instructions

Instruction	Execution Time [μs]	Instruction	Execution Time [μs]
MOV	0.208379	LOG	1.895198
ADD	0.210483	FLOOR	0.209549
MUL	0.214238	EXP	1.473430
MAX	0.213748	SIGN	0.210650
MIN	0.208210	STEP	0.210643
RSQ	1.472668		

6.3 Machine Learning versus Bounding Box

In this section, the machine learning algorithm is compared to the bounding box algorithm. We are not able to use the introduced application "shadergen" from the previous section. So far, the bounding box implementation requires the position transformation of the vertex shader. Therefore, we are going to use different scenarios which are explained in the following.

6.3.1 Scenarios

To generate scenarios for the execution time prediction, different benchmarks are used. On the one hand, we show an evaluation for glmark2-es2 benchmark "buffer" and on the other hand, we use glmark2-es2 benchmark "build, model horse" to compare the machine learning and bounding box algorithm. glmark2-es2 is a benchmark program, which uses OpenGL ES 2.0 API-calls and can create various 3D-scenes [6]. It is primary used to test the speed of the graphics card. For this evaluation, we are using it to measure the error of the execution time prediction.

We focus on two different benchmark scenes. In the following the properties of the scenes are explained. On the contrary to the application "shadergen" the glmark2-es2 benchmark program uses a nearly constant setup of shaders, vertices and number of fragments.

Buffer. The scene "buffer" uses two different command group calls alternating. The one contains a draw and a clear command, and the other contains a swap and flush command. The number of vertices is constant for the entire scene. The number of fragments changes over time slowly. However, the fragment estimation does not realize this change. Thus, the machine learning and bounding box algorithm do not now that the number of fragments changes. Hence, the execution time of each draw command has only a small fluctuation.

Build. The scene "build, model horse" has the same command group setup, but the number of fragments changes. The heuristic to calculate the number of fragments shows a strong deviation to the real number of fragments in this setup.

Before we present the evaluation results, we first explain the machine learning and the bounding box approach shortly.

Machine Learning (ML). This approach is the implementation of the concept from Chapter 4. It uses the bounding box algorithm to estimate the number of fragments.

Bounding Box (BB). This approach is the reference implementation from Schnitzer et al. [21]. It pre-calibrates the used shaders and uses the calibrated values to predict the execution time. To still be adaptive to calibration errors and changing environments, an adaption is implemented. To estimate the number of fragments the bounding box algorithm is applied, which is presented in the same publication.

The bounding box algorithm needs to calibrate each shader in advance, whether the machine learning algorithm adapts over time. Thus, the expectation is, that the bounding box algorithm has advantages in the beginning, but after certain learning steps the machine learning algorithm takes the lead.

6.3.2 Adaptivity

In this section, the adaptivity of online machine learning algorithms in comparison to the bounding box algorithm is shown. Adaptivity means to adjust the parameter of the used linear model to fit to the current environment. In general, the parameters are untrained in the beginning and can be chosen randomly. Over time the parameters of the algorithm adjust continuously to fit to the actual execution time in the end. However, we want to create a realistic setup for the both algorithms. Therefore, the bounding box algorithm already run and could calibrate the system parameters, and the machine learning application run with the application "shadergen" to pre-calibrate the weights. We are using the pre-calibrated weights from the previous section.

In Figure 6.5 the adaption of the parameters on the Nvidia graphics is shown. The glmark2-es2 program using the scene "buffer" is used to generate this scenario. In general, the first predictions are made with the start parameters, because the prediction and measurement of the execution time is performed at different times. It exists an asynchronous behavior in the prediction framework. The prediction is made before the graphics commands are executed, the measurement of the real execution time is finished after the GPU completed its execution. Because the prediction is in user space and the measurement of the execution time is in kernel space, it can lead to further delays.

In the first 100 command groups the bounding box algorithm needs to calibrate itself and is not processing the buffer benchmark. During this calibration no prediction is made and the error corresponds to the execution time of the calibration. Thus, for the bounding box algorithm the benchmark "buffer" is delayed and not executed for the first 100 command groups. After 100 command groups the convergence of the bounding box algorithm is faster than for the machine learning algorithm.

In contrast, the machine learning algorithm has two primary curves. The reason is, that two different command groups are executed alternated: the overestimation, the flush command and the underestimation, the draw command. The execution time of the flush

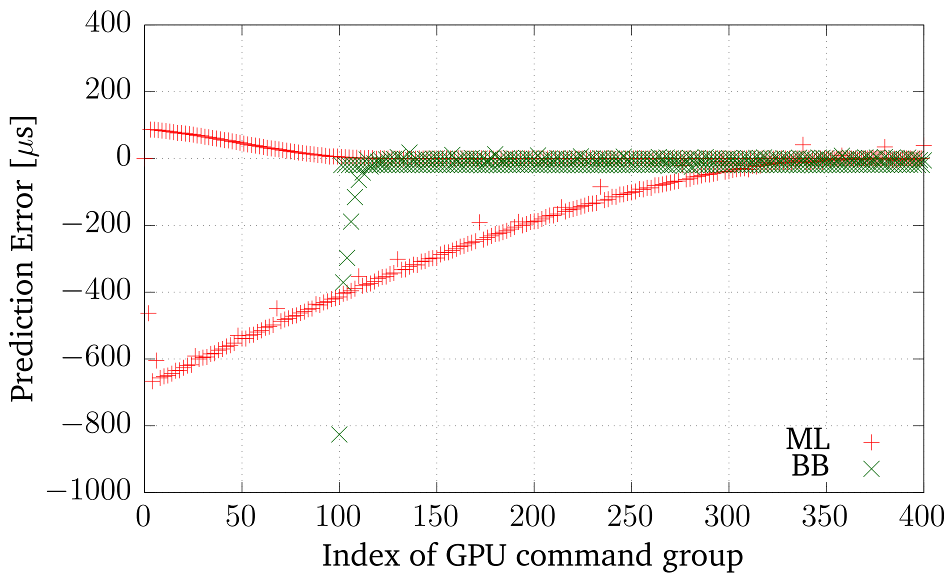

Figure 6.5: Adaptivity of the bounding box and machine learning algorithm using glmark2-es2 "buffer" on the Nvidia graphics card.

command varies from application to application. The flush command is not calibrated properly by the application "shadergen" and has an estimated execution time of around $100\mu s$ whether the execution time for the flush command in this scenario is around $40\mu s$. The curve that underestimates in the beginning is caused by the draw command. The execution time of the draw command is around $1ms$. Without calibration that would be the prediction error at start time. Because of the calibration, the draw command is around $-630\mu s$ which leads to an improvement of around $370\mu s$ in the beginning.

Overall we can see that the calibration for the command group of the draw command is better for the machine learning than for the bounding box. The worst prediction of the bounding box algorithm is around $-800\mu s$ whether $-630\mu s$ the worst prediction for the machine learning algorithm is. However, the convergence of the bounding box algorithm to this static scene is much faster.

In Figure 6.6 the adaption of the parameter on the Vivante graphics card can be seen. To be able to compare the Nvidia and Vivante graphics cards, the same scenario, using the glmark2-es2 program with the scene "buffer", is also applied on the Vivante graphics card. The main difference is, that the flush command is not implemented on the Vivante platform and the swap buffer command is implemented. The bounding box algorithm calibrates the program for the first 100 command groups and cannot make any predictions. After that the algorithm is converging faster than the machine learning algorithm.

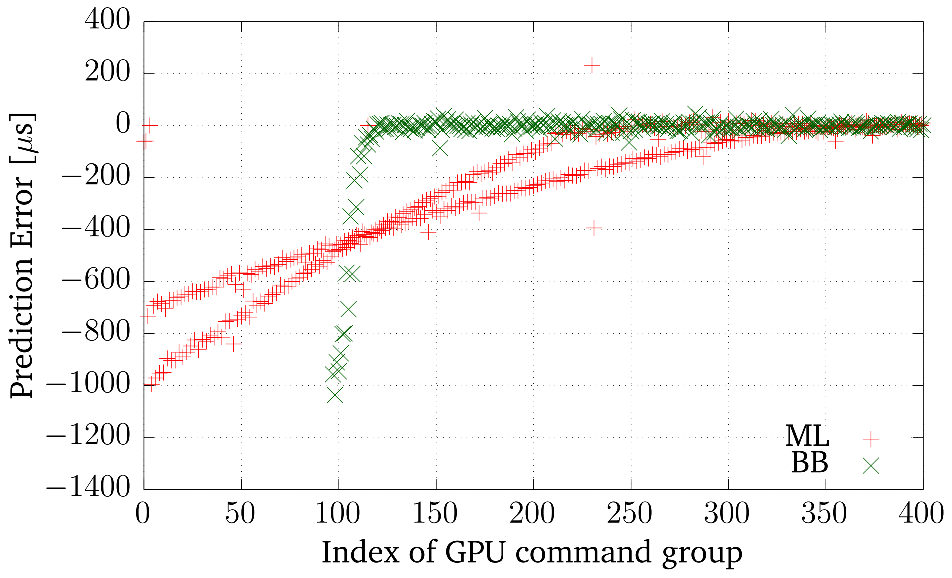

Figure 6.6: Adaptivity of the bounding box and machine learning algorithm using glmark2-es2 "buffer" on the Vivante graphics card.

In comparison to the Workstation graphics cards driver implementation, the implementation on the Freescale platform differs. The main commands in the two curves are the swap buffer command and the draw command. In this scenario, both commands take similar execution time usage. The execution time of the draw command is slightly higher, than the execution time of the swap buffer command. Using the application "shadergen" it was not possible to calibrate the swap buffer command, because the application does not create fragments. Therefore, the prediction of the swap buffer command starts as lowest at around $-1000\mu s$. Whether for the draw command exist initial weights and we observe a start prediction error of around $-690\mu s$ instead of $-1270\mu s$, which is an improvement of more than 45% of the real execution time. Further it can be observed, that the first predictions of the machine learning algorithm are better than of the bounding box algorithm.

Overall, we observe for the Nvidia and Vivante GPU, that the system calibration is more accurate for the machine learning algorithm than for the bounding box algorithm. Further, the adaptivity of the applied stochastic gradient descent algorithm is slower than the bounding box algorithm. Nevertheless, it is possible to determine a learning rate, which is optimal in this scenario, but would lead to overshooting in other scenarios.

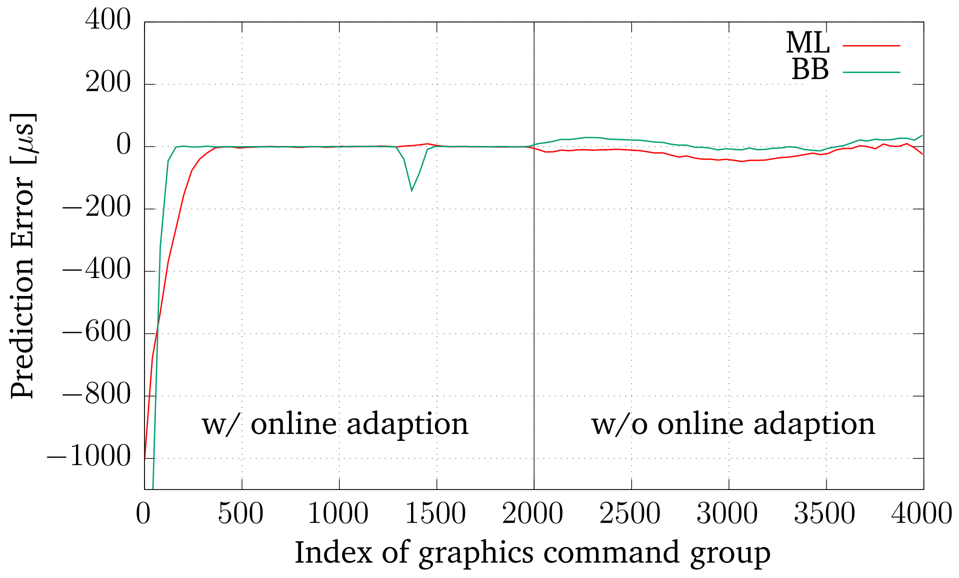

Figure 6.7: Vivante: Learning the model parameters using glmark2-es2 "buffer".

6.3.3 Learning the Parameters of the Execution Time Model

In this scenario, we evaluated if the algorithms can learn the scene and still predict reasonable execution times if the online learning is turned off. In Figure 6.7 a Bézier curve over the prediction error is shown. The figure shows the evaluation for the Vivante graphics card. This evaluation is split into two phases. First, the trainings phase and second, the evaluation of the trained parameters.

The first phase runs for 2000 graphics command groups. First, both algorithms are improving their prediction. The bounding box algorithm improves via calibrating the program and adapting their parameters online. The online machine learning algorithm by training the model parameters. After finishing the first phase, the adaption is turned off. Both algorithms were able to adapt their parameters and the parameter adaption is evaluated. In the first 100 command groups, the bounding box algorithm is calibrating itself and is not predicting the benchmark scene "buffer". The machine learning algorithm continuously improves its weights and is better on average in the first phase.

In the second phase, beginning with the 2001st graphics command group, the prediction is running without online adaption. It only uses the adapted parameters from the previous phase. Both algorithms reach similar results. Because the scene "buffer" generates nearly constant input and output parameters, the prediction should not have any variation after the training is finished. However, the estimation of the fragment is a complex heuristic, which has an error. This estimation error of the heuristics can be seen in phase two. Although all parameters are constant, the error of the number of fragment estimation propagates to the prediction error for both algorithms.

6.3.4 Accuracy of Execution Time Prediction

In this section, the accuracy of the execution time prediction is presented. Therefore, the bounding box algorithm and the proposed concept in this thesis are compared, when the parameters of the algorithms are already trained. The scenario is the following: Beforehand, both algorithm train their parameters. That means, that they can improve their parameters and adapt them to the scene. This is done by running the benchmark with each algorithm and store the obtained parameters persistent. After that, the evaluation starts with another execution of the benchmark program and the prediction error is measured. To show the accuracy a cumulative distribution function over the error is presented. In this scenario we use the glmark-es2 program with the "buffer" and "build, model horse" scene.

In Figure 6.8 the cumulative distribution function of the error for the Nvidia graphics card is depicted. The results for the scene "build" are similar for both algorithms. The machine learning algorithm is better to prevent underestimation in this scene. For the scene "buffer" the machine learning algorithm has a bigger advantage. The bounding box algorithm has an overestimation for around 66% of the command groups and a larger prediction error overall. In general, the machine learning algorithm shows in comparison to the bounding box algorithm better or similar results on many of the glmark2-es2 scenes.

Figure 6.9 depicts the CDF of the same scenes "buffer" and "build" for the Vivante graphics card. By comparing the CDFs of the scene "build" it can be seen that the bounding box algorithm is slightly better, especially for underestimation. However, for the scene "build" the bounding box algorithm has a clear benefit. As previously mentioned, the scene "build" consists of two alternating command groups. One contains mainly a draw command and the other mainly the swap buffer command. Both of them have strong prediction errors. In the following we analyze why they have a strong prediction error. First we consider the draw command. Analyzing the scene "build" shows us that the entire environment is constant except the number of fragments. The number of fragments do not show any linearity to the execution time. Thus, it is not possible to find a linear mapping for this parameter. However, analyzing the real number of fragments afterwards shows that a linear dependency between the real number of fragments and the execution time exists. Hence, a better heuristic to estimate the number of fragments would probably lead to better results for the draw command. Second, we analyze the cause of the error for the command group containing the swap buffer command. This command group has a constant number of pixels that are swapped, but

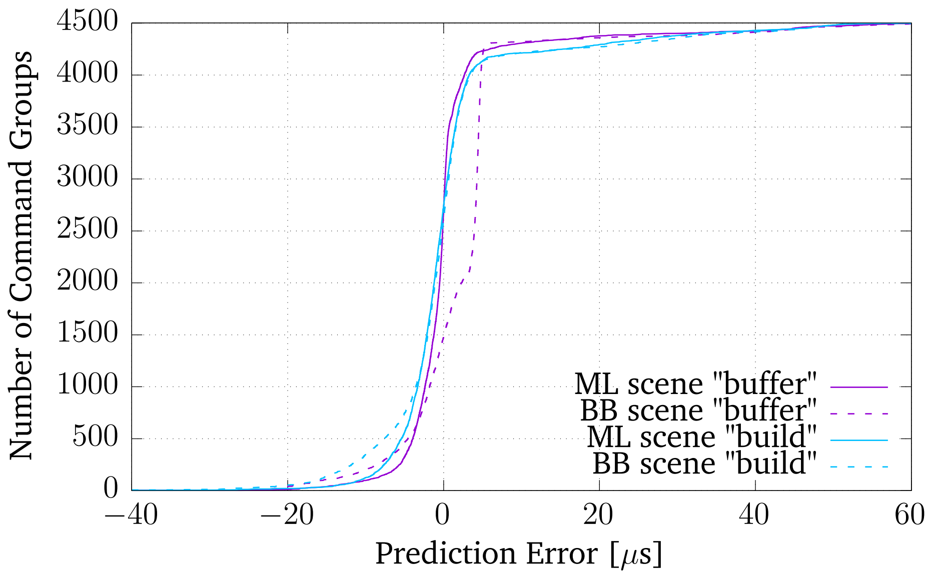

Figure 6.8: Cumulative distribution function of the error of machine learning and bounding box algorithm for the Nvidia graphics card.

a high variance in execution time. The range is from $560\mu s$ to $740\mu s$, which is caused by a tiled-based rasterization that is used by the Vivante GPU [13, p. 22]. The machine learning algorithm tries to find one weight for this feature, which leads to an average prediction of around $640\mu s$. This implies a high error in many predictions.

We still need to consider why the bounding box algorithm gets better results although both effects apply to it as well. The adaption algorithm of the bounding box does not try to fit a model in comparison to the machine learning algorithm. It is adaptive to the current scene, only considers the recent values and is adaptive to them. Hence, as long as the scene has only minor changes from one command group to the next command group, the bounding box algorithm has advantages in predicting the execution time. However, when the scene has major or has noise

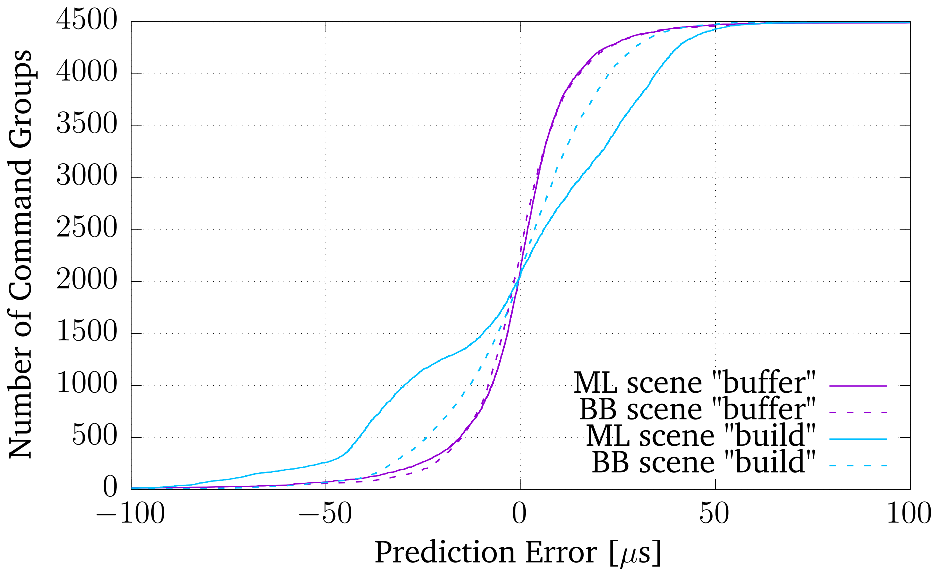

Figure 6.9: Cumulative distribution function of the error of machine learning and bounding box algorithm for the Vivante graphics card.

7 Summary and Future Work

In this chapter, a short summary and the future work is presented.

7.1 Summary

As we presented in the motivation, the use of high resolution 3D rendering in embedded systems is increasing. Through consolidating of GPUs, a real-time GPU scheduler is needed that needs the accurate GPU execution time of GPU commands to schedule the GPU commands.

In this thesis, a framework for predicting the GPU execution time using machine learning was presented. At first, background information about OpenGL ES 2.0 and machine learning were introduced. After that, the influence factors for the GPU execution time of the 3D rendering Pipeline of OpenGL ES 2.0 were analyzed. Using this information, a linear model which allows us to predict the execution time was created. The linear model contains the GPU instructions of the shaders as features. After that, the architecture of the prediction framework including the machine learning module was introduced. As online machine learning algorithm, we presented the stochastic gradient descent algorithm using normalization, adaption of the learning rate, and regularization. Then, we presented our implementation of the prediction framework, generalized and implemented on a workstation computer and an embedded board.

The evaluation of the concept was performed on both platforms. We showed, considering the shader binary code in the feature vector yields to better predictions for so far unseen shaders. For the Vivante GPU it was possible to determine precise execution times of GPU instructions. Furthermore, we observed a slower convergence of the machine learning algorithm. However, this depends on the learning rate which is still tunable. The accuracy of the machine learning algorithm in comparison to the bounding box algorithm evaluated with the glmark2-es2 benchmark shows better results on the Nvidia GPU than on the Vivante GPU. Although the accuracy for random shaders was more precise on the embedded board, the accuracy in glmark2-es2 benchmarks on the embedded board has space for improvement. However, the evaluated scenes are highly repetitive with minor changes in consecutive command groups. In conclusion, the bounding box algorithm adapts to recent changes, whereas, the machine learning algorithm tries to fit a model.

7.2 Future Work

In this section, we present topics for future work in this research area.

One point of improvement is to avoid underestimation. The prediction was designed to predict accurately. However, errors are not avoidable and are punished independently if they are overestimations or underestimations in the same way. For a real-time scheduler an underestimation can lead to miss deadlines of a task. On the contrary, overestimation cannot harm the real-time system. There are two approaches to deal with it. First, it is possible to punish an underestimation and temporarily increase the learning rate η to do not predict with an underestimation again. The second option deals with an overestimation in the current state where the algorithm slows down in continuously approaching the optimum. This can be implemented by decreasing the learning rate η, if the gradient points towards a reduction of the predicted execution time.

Furthermore, we do not support jumps in shaders yet. Generally, a precise estimation cannot be made, if a jump command exists in a shader. However, jumps which go forward and skip GPU commands lead only to overestimation. An upper bound of execution time can still be determined. Overestimation of a command group can only make a command group not to be executed.

Moreover, the loading time of a shader depends among others on the shader attributes. We did not consider the loading time so far. However, including the shader attributes in the feature vector could be another idea to improve the prediction of the execution time even further.

A Appendix

A.1 Vivante Opcodes

In this section, we present the opcode of the graphics card Vivante GC2000. The opcode is provided by the GPU driver version 5.0.11.p4.25762 published by Freescale [25]. The following enumeration holds the opcodes of the Vivante GC2000.

```
typedef enum _gcSL_OPCODE {
  NOP, MOV, SAT, DP3, DP4, ABS, JMP, ADD, MUL, RCP, SUB, KILL, TEXLD, CALL,
  RET, NORM, MAX, MIN, POW, RSQ, LOG, FRAC, FLOOR, CEIL, CROSS, TEXLDPROJ, TEXBIAS,
  TEXGRAD, TEXLOD, SIN, COS, TAN, EXP, SIGN, STEP, SQRT, ACOS, ASIN, ATAN, SET, DSX,
  DSY, FWIDTH, DIV, MOD, AND_BITWISE, OR_BITWISE, XOR_BITWISE, NOT_BITWISE,
  LSHIFT, RSHIFT, ROTATE, BITSEL, LEADZERO, LOAD, STORE, BARRIER, STORE1,
  ATOMADD, ATOMSUB, ATOMXCHG, ATOMCMPXCHG, ATOMMIN, ATOMMAX, ATOMOR, ATOMAND,
  ATOMXOR, TEXLDPCF, TEXLDPCFPROJ, SINPI = 80, COSPI, TANPI, ADDLO, MULLO, CONV,
  GETEXP, GETMANT, MULHI, CMP, I2F, F2I, ADDSAT, SUBSAT, MULSAT, DP2, UNPACK,
  IMAGE_WR, SAMPLER_ADD, MOVA, IMAGE_RD, IMAGE_SAMPLER, NORM_MUL,
  NORM_DP2, NORM_DP3, NORM_DP4, PRE_DIV, PRE_LOG2, MAXOPCODE
} gcSL_OPCODE;
```

A.2 Tesla Instruction Set Architecture

In this section, the binary code of the Tesla Instruction Set Architecture is depicted. Table A.1 and Table A.2 depict the mapping between opcode and instruction name. Each instruction has a primary code which can be seen in the first column. Further, each instruction is of a type that is encoded in the first 32 bits. Depending on the type, the column is selected respectively. For the type long normal exists a secondary code that is written in the header column as well.

Table A.1: Tesla opcode map part 1/2 adapted from [24].

Primary opcode	**short normal**	**long immediate**	**long normal, secondary 0**	**long normal, secondary 1**	**long normal, secondary 2**	**long normal, secondary 3**
0x0	-	-	ld a[]	mov from $c	mov from $a	mov from $sr
0x1	mov	mov	mov	ld c[]	ld s[]	vote
0x2	add/sub	add/sub	add/sub	-	-	-
0x3	add/sub	add/sub	add/sub	-	-	set
0x4	mul	mul	mul	-	-	-
0x5	sad	-	sad	-	-	-
0x6	mul+add	mul+add	mul+add	mul+add	mul+add	mul+add
0x7	mul+add	mul+add	mul+add	mul+add	mul+add	mul+add
0x8	interp	-	interp	-	-	-
0x9	rcp	-	rcp	-	rsqrt	lg2
0xa	-	-	cvt i2i	cvt i2i	cvt i2f	cvt i2f
0xb	fadd	fadd	fadd	fadd	-	fset
0xc	fmul	fmul	fmul	-	fslct	fslct
0xd	-	logic op	logic op	add $a	ld l[]	st l[]
0xe	fmul+fadd	fmul+fadd	fmul+fadd	fmul+fadd	dfma	dadd
0xf	texauto/fetch	-	texauto/fetch	texbias	texlod	tex misc

Table A.2: Tesla opcode map part 2/2 adapted from [24].

Primary opcode	**long normal, secondary 4**	**long normal, secondary 5**	**long normal, secondary 6**	**long normal, secondary 7**	**short control**	**long control**
0x0	st o[]	mov to $c	shl to $a	st s[]	-	discard
0x1	-	-	-	-	-	bra
0x2	-	-	-	-	-	call
0x3	max	min	shl	shr	-	ret
0x4	-	-	-	-	-	prebrk
0x5	-	-	-	-	-	brk
0x6	mul+add	mul+add	mul+add	mul+add	-	quadon
0x7	mul+add	mul+add	mul+add	mul+add	-	quadpop
0x8	-	-	-	-	-	bar
0x9	sin	cos	ex2	-	trap	trap
0xa	cvt f2i	cvt f2i	cvt f2f	cvt f2f	-	joinat
0xb	fmax	fmin	presin/preex2	-	brkpt	brkpt
0xc	quadop	-	-	-	-	bra c[]
0xd	ld g[]	st g[]	red g[]	atomic g[]	-	preret
0xe	dmul	dmin	dmax	dset	-	-
0xf	texcsaa/gather	(*unknown*)	emit/restart	nop/pmevent	-	-

Bibliography

[1] L. B. Almeida, T. Langlois, J. D. Amaral, A. Plakhov. "Parameter Adaptation in Stochastic Optimization." In: *On-Line Learning in Neural Networks, Publications of the Newton Institute* (1998), pp. 111–134 (cit. on pp. 7, 30, 32, 33, 36, 50).

[2] E. Alpaydin. *Introduction to Machine Learning*. 2nd ed. The MIT Press, 2010 (cit. on pp. 15, 16, 18).

[3] M. Bautin, A. Dwarakinath, T.-c. Chiueh. "Graphic Engine Resource Management." In: *Electronic Imaging 2008*. International Society for Optics and Photonics. 2008 (cit. on pp. 2, 6).

[4] L. Bottou. "Stochastic Gradient Descent Tricks." In: *Neural Networks: Tricks of the Trade*. Springer, 2012, pp. 421–436 (cit. on p. 18).

[5] J. Duchi, E. Hazan, Y. Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." In: *The Journal of Machine Learning Research* 12 (2011), pp. 2121–2159 (cit. on pp. 7, 33).

[6] A. Frantzis, J. Barker. *OpenGL 2.0 and ES 2.0 benchmark*. URL: https://github.com/glmark2/glmark2 (visited on 03/01/2016) (cit. on p. 59).

[7] T. Hastie, R. Tibshirani, J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. 2nd ed. Springer, 2011 (cit. on pp. 16, 17).

[8] T. K. G. Inc. *OpenGL® ES Common Profile Specification Version 2.0.25 (Full Specification)*. Nov. 2, 2010. URL: https://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf (visited on 03/08/2016) (cit. on pp. 9–13).

[9] S. Kato, K. Lakshmanan, R. Rajkumar, Y. Ishikawa. "TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments." In: *USENIX Annual Technical Conference (USENIX ATC'11)*. 2011, p. 17 (cit. on p. 5).

[10] J. Langford, L. Li, T. Zhang. "Sparse Online Learning via Truncated Gradient." In: *Advances in Neural Information Processing Systems*. 2009, pp. 905–912 (cit. on pp. 7, 30, 35, 50, 54).

[11] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym. "NVIDIA Tesla: A Unified Graphics and Computing Architecture." In: *IEEE Micro 28(2)* (2008), pp. 39–55 (cit. on p. 22).

[12] *List of NVIDIA Chip Code Names*. URL: https://nouveau.freedesktop.org/wiki/CodeNames/ (visited on 03/25/2016) (cit. on p. 24).

[13] H. Ma. "Concepts and Metrics for Measurement and Prediction of the Execution Time of GPU Rendering Commands." Master's Thesis. University of Stuttgart, 2014 (cit. on p. 65).

[14] Microsoft. *Microsoft WDDM GPU preemption*. URL: https://msdn.microsoft.com/en-us/library/windows/hardware/jj553428.aspx (visited on 03/01/2016) (cit. on p. 5).

[15] A. Munshi, D. Ginsburg, D. Shreiner. *OpenGL ES 2.0 Programming Guide*. Pearson Education, Inc., 2009 (cit. on pp. 10, 12, 13).

[16] N. J. Nilsson. *Introduction to Machine Learning*. 1998. URL: http://ai.stanford.edu/~nilsson/MLBOOK.pdf (cit. on p. 14).

[17] *Oxford Advanced Learner's Dictionary*. Oxford Univ. Press (OELT), 2015 (cit. on p. 14).

[18] M. A. Poole, P. N. O'Farrell. "The Assumptions Of The Linear Regression Model." In: *Transactions of the Institute of British Geographers* (1971), pp. 145–158 (cit. on p. 19).

[19] H. Robbins, S. Monro. "A Stochastic Approximation Method." In: *The Annals of Mathematical Statistics* (1951), pp. 400–407 (cit. on p. 19).

[20] S. Ross, P. Mineiro, J. Langford. "Normalized Online Learning." In: *arXiv preprint arXiv:1305.6646* (2013) (cit. on pp. 6, 30, 31).

[21] S. Schnitzer, S. Gansel, F. Dürr, K. Rothermel. "Concepts for Execution Time Prediction of 3D GPU Rendering." In: *Industrial Embedded Systems (SIES), 9th IEEE International Symposium*. 2014, pp. 160–169 (cit. on pp. 2, 6, 27, 29, 40, 41, 60).

[22] S. Schnitzer, S. Gansel, F. Dürr, K. Rothermel. "Real-time Scheduling for 3D GPU Rendering." In: *Industrial Embedded Systems (SIES), 11th IEEE International Symposium*. 2016, Accepted for publication (cit. on pp. 2, 6).

[23] S. Shalev-Shwartz, S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014 (cit. on pp. 14, 15).

[24] *Tesla CUDA ISA*. URL: http://envytools.readthedocs.org/en/latest/hw/graph/tesla/cuda/isa.html (visited on 03/25/2016) (cit. on pp. 24, 25, 70, 71).

[25] *Website of Freescale*. URL: http://www.freescale.com (visited on 04/29/2016) (cit. on p. 69).

[26] M. Yu, C. Zhang, Z. Qi, J. Yao, Y. Wang, H. Guan. "VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming." In: *ACM Transactions on Architecture and Code Optimization (TACO)* 11.2 (2014), p. 17 (cit. on pp. 2, 5).

[27] M. D. Zeiler. “ADADELTA: An Adaptive Learning Rate Method.” In: *arXiv preprint arXiv:1212.5701* (2012) (cit. on pp. 7, 33, 50).

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