Universität Stuttgart

Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis Nr. 3635

Concepts and Metrics for Measurement and Prediction of the Execution Time of GPU Rendering Commands

Hua Ma

Course of Study:	INFOTECH
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Dipl.-Inf. Stephan Schnitzer
Commenced:	February 17, 2014
Completed:	August 19, 2014
CR-Classification:	I.7.2

Abstract

Graphic Processing Units (GPUs), wtih their highly parallel structure, is a powerful tool to manipulate computer graphics and process large amounts of data in parallel. GPUs are becoming increasingly popular in today's embedded system world. One typical example is the modern vehicle. Nowadays, modern vehicles are often required to run graphic applications such as instrumentation cluster, navigation system, media player or games with higher graphic quality concurrently on multiple displays. Compared to the CPU, GPU is more suitable to fulfil these requirements. In order to reduce the hardware cost and the power consumption at the same time, a good solution is to share a single GPU for multiple applications. Thus emerges the need of GPU real time scheduling. Since current GPUs do not support preemption, accurate prediction of the GPU command execution time becomes an indispensable precondition for scheduling.

In this work, the driver model and hardware features of a typical 3D embedded GPU is analysed. Concepts to measure GPU execution time and other statistics is present and implemented. The measured execution time and the relevant statistics of the GPU command group allow us to explore the factors and metrics that affect the execution time of GPU rendering commands, more specifically, OpenGL ES 2.0 commands/APIs. Different models are developed to predict the important time-consuming rendering commands: Flush, Swap, Clear, Draw and Texture Loading. As for the Draw command, a recent-history-based approach is proposed to estimate the number of fragments. Finally, the accuracy of the measurement and prediction is evaluated.

ACKNOWLEDGEMENTS

As I present the results of my Masters thesis, I would like to thank the people who helped me and supported me during this period of time.

First of all, I would like to express my sincere gratitude to my supervisor, Mr.Stephan Schnitzer, who has supported me, guided me and shared his valuable experience with me from the begin to the end. It was really a nice experience to have meetings and discussions with him.

I would like to thank Prof. Dr. Kurt Rothermel for giving me the opportunity to do the thesis in his department.

Finally, I am grateful to my family and my girl friend, for their support and encouragement. I am also thankful to all the colleges in DS department who created a pleasant working environment and especially to Patrik, who taught me how to use latex and proofread my thesis.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The parallel architecture gives GPU a big advantage to process graphic-related applications, especially applications with high resolution 3D animation, compared to the central processing unit (CPU). The efficiency in manipulating computer graphics makes GPU widely used in many areas including the embedded system domain. A particular compelling application for GPUs is that of automobiles[17].

In a modern vehicle system, the capabilities of the GPU are required and exploited by more and more automotive electronic subsystems. For instance, the GPU can be used by the instrument cluster. The instrument clusters are moving from traditional analog dials and gauges to digital dashboards rendered on instrument cluster screens. New specifications from leading automotive OEMs require advanced graphics rendering to be at the heart of the digital automotive experience[6]. As another example, the infotainment system, equipped with one or more screens, uses GPU to support higher resolution 3D navigation, latest multimedia and games. Figure 1.1 demonstrates the instrument cluster and the infotainment system of the Tesla Model S, as the winner of Motor Trend 2013's car of the year[3], Tesla Model S, equipped with a 17-inch touch screen infotainment system and a 12.3-inch instrument cluster LCD, benefits a lot from embedded GPU's properties of graphic acceleration and low power consumption[9].

Despite that, a GPU has great functionality and potential in the automotive domain, GPU can also leads to the growth of the cost, space requirement and energy consumption when multiple GPUs are integrated into the system. Therefore, the trend in new automotive designs toward consolidating(see Figure 1.2) multiple graphic applications into a single GPU to increase the GPU utilization, save power consumption and reduce the cost. There emerges a need of GPU scheduling.

1.2 Challenges

Generally, the operating system is responsible for scheduling applications. Graphic applications are also scheduled by the operating system. However, this does not mean the scheduling for the GPU is performed. For instance, in an OpenGL application, when a graphic API is called,

Figure 1.1: The Tesla Motors Model S [4]

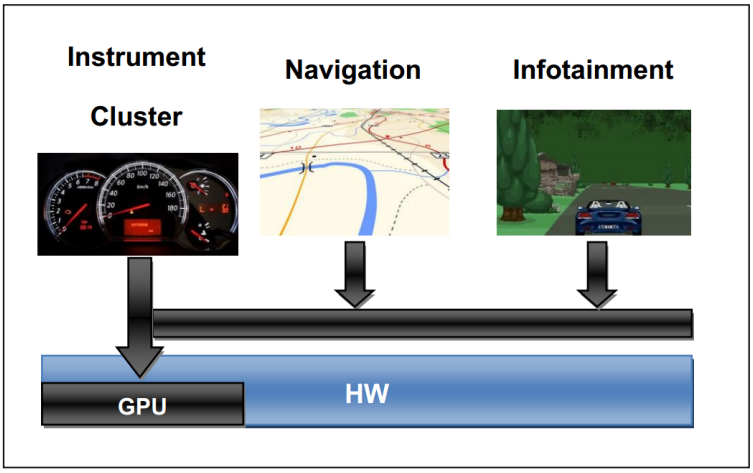

Figure 1.2: GPU Consolidation

normally the API only writes some GPU hardware commands into a temporary buffer. Nothing is actually executed on the GPU. Only under certain situations, the commands will be submitted to the GPU and executed on it. Hence a separate scheduler is required for GPU scheduling which is normally implemented by the GPU driver.

FIFO is often the scheduling policy for current GPUs, which means that the task submitted first will be served first. However, it is not sufficient for GPU applications such as the applications in the automotive domain that require real time performance. Scheduling real time graphic applications faces the following challenges:

1. Different GPU applications require different level of real-time performance. For instance, in a vehicle system, the instrument cluster is safety-critical and belong to real-time applications. 3D navigation can be grouped into soft real-time category while application like media player is sufficient with a lower real-time performance. Hence the scheduling policy should guarantee the execution of safety-critical applications while maximizing the utilization of the GPU.

2. GPU generally does not support preemption. That means during processing, the GPU cannot interrupt the low priority tasks and switch to high priority tasks. Without preemption, the GPU scheduler has to "know the future" before dispatching tasks with lower priority in order to guarantee that the safety-critical applications won't miss the deadline.

A possible solution for scheduling real-time GPU applications is to use non-preemptive deadline-driven scheduling. This requires the knowledge of the execution time of every dispatched GPU tasks. In [10] Bautin et al designed a priority based GPU scheduler with runtime prediction. In their work, the execution time of the GPU command group, which is a batch of GPU commands that form a certain task to be executed by the GPU, is predicted based on dynamically collected statistics. However, the method they proposed to estimate the GPU command execution time is in general inaccurate. Kato et al[24] proposed a history-based prediction approach which use history statistics to estimate the command group execution time, their approach only works for applications whose command groups are highly repetitive. Schnitzer et al[33] presented a fine-grained rendering command-based prediction approach that estimate the command group execution time based on every separated time-consuming rendering commands. In their prediction, the number of vertex number and fragment number are used for estimating the Draw command. However, the fragment number prediction model they used often leads to fragment number overestimation. Moreover, all of the works and concepts mentioned above are developed and evaluated only with desktop-GPUs which have many different features compared to embedded GPUs. All these problems mentioned above motivate us to conduct this work.

1.3 Contribution

In this thesis, concepts for measurement and prediction of the execution time of GPU rendering (OpenGL ES 2.0) commands are developed, implemented and evaluated on a typical embedded 3D GPU platform. The major contributions of this thesis are:

1. The hardware features and the driver model of typical embedded GPUs are explored and described.

2. The concept of an accurate measurement system is developed and implemented. The measurement system not only measures the GPU command group execution time, but also collects GPU statistics such as the size of each command group, the input vertices number,

fragments number etc. to help analyse the GPU rendering commands and improve the prediction accuracy.

3. Prediction models are built for estimating OpenGL ES 2.0/EGL commands including Flush, Swap, Clear, Draw and Texture Loading. As for the Draw command, a recent-history-based approach is proposed to estimate the fragment number generated by every GPU command group.

4. Evaluations using OpenGL ES 2.0 graphics benchmarks demonstrate that the recent-history-based approach can effectively predict the fragments number of the GPU command group (for the evaluated benchmark, the mean absolute error is less than 0.1%). Our commands prediction models also achieve good accuracy in estimating the execution time of different rendering commands. For the most critical rendering command, the Draw command, the mean absolute prediction error is limited from 1.77% to 2.6%.

5. For the Draw command, the affect of the texture size and the affect of a GPU optimization technique MASS(Multisample anti-aliasing) are evaluated.

1.4 Organization

Chapter 2 – Related Work: The previous publications in the field of GPU time measurement and prediction will be summarised here.

Chapter 3 – Background: The chapter describes the technical background of the thesis.

Chatper 4 – System Model: This chapter explores and describes the system model of the GPU driver.

Chapter 5 – Concept: This chapter gives a deep insight into the concept and metrics of the GPU rendering commands, prediction model for important GPU rendering commands as well as the framework of measurement and the prediction system.

Chapter 6 – Implementation: The implementation of our measurement and prediction system including run-time calibration is explained in detail in this chapter. Additionally, the modification in the original GPU kernel driver is mentioned.

Chapter 7 – Evaluation: The result of our measurement and prediction is evaluated in this chapter.

Chapter 8 – Summary and Future Work: The last chapter concludes the thesis and points out the possible future work originating from this thesis.

2 Related Work

It can be seen from the last chapter that the fundamental for scheduling real time GPU applications is to accurately predict the execution time of each GPU command group. In order to validate the precision of the prediction, an accurate measurement system is required. In this chapter, more details about GPU real time scheduling are presented. Then several previous works that measure and predict the execution time of GPU command groups for graphic (OpenGL/OpenGL ES) applications will be briefly described.

2.1 Real Time GPU Scheduling

Traditional real-time CPU schedulers often focus more on scheduling preemptable tasks [12]. However, GPU generally does not support preemption. Although windows system starts to support an advanced scheduling since DirectX 10 class WDDM(Windows Display Driver Model) drivers where the GPU can be interrupted within an individual primitive and within an individual shader program[32]. However, it is not a requirement for the hardware vendors to support the preemption. Moreover, to interrupt a shader code without introducing substantial overheads will be very challenging.

In [10] a fair-share GPU scheduler using weighted round robin policy is proposed. However, the purpose of the scheduler is to improve the utilization of the GPU resources. A real-time scheduler is not required. In [24], Kato et al proposed a scheduler using predictable-response-time (PRT) scheduling policy and apriori enforcement (AE) reservation policy. With PRT policy, GPU command groups are queued before dispatching to the GPU if the executions of the preceding command groups are not finished. After the preceding command groups are finished, the queued command groups are dispatched in a sequence based on their fixed priorities. AE policy assigns different values of budgets to different GPU applications and predicts the cost of each command group of these GPU applications. Then the scheduler compares the budget with the predicted cost. If the budget is smaller than the cost, the command group will go to sleep until the budget is replenished. With PRT and AE, prioritization and isolation (requirements of real time applications) are supported. However, the precondition for such a scheduler is to accurately estimate the execution time of each GPU command group.

2.2 GPU Execution Time Measurment

A very simple way to measure the GPU command group execution time is to insert an additional glFinish command after a command group is dispatched to the GPU [23]. Because glFinish will block the current rendering thread until all hardware commands in the GPU pipeline are finished. The time that glFinish returns can be used as the finishing time of the previous dispatched command group. However, because the measurement is done in the user space, high accuracy can hardly be achieved due to process scheduling and context switch. Moreover, this approach will unnecessarily block the GPU and introduce additional synchronization between user space and kernel space, which affect the accuracy of the measurement.

Based on the way to detect the command group finishing time, exemplary approaches of GPU execution time measurement can be divided into two categories: polling-based time measurement and interrupt-based time measurement. These two approaches are explained as follows:

Polling-based time measurement: Polling-based time measurement is often applied to GPUs that can not generate interrupts after the GPU completes a command group. Polling-based time measurement reads a specified GPU register frequently to detect the finishing time of the command group. [10, 16] explain the implementation of this approach in detail: at the end of every dispatched command group, a GPU command that increments the value of a particular register in the GPU is inserted. The register is often called "timing register". The inserted command will be executed when the dispatched command group is finished.

During the measurement, the value of the timing register is polled periodically to detect the change since the last polling. For instance, if the value is increased by one, then one command group is finished since the last polling. If more than one command groups have finished in one polling period, average execution time will be assigned to each command group. The accuracy of polling-based time measurement depends highly on the polling frequency and the latency of the polling thread (the polling thread might be blocked by other processes/threads and the interrupts in the operating system).

Interrupt-based time measurement: Many state of the art GPUs can use a special GPU command (such as NOTIFY) for generating interrupts. For these kinds of GPUs, interrupt-based time measurement can be a better option. This approach lets the GPU generate an interrupt when a command group is finished. The time when the interrupt comes is used as the finishing time of the command group. In [24, 33], the interrupt-based time measurement system is implemented on the NVIDIA GPU architecture. In their implementation, a SERIALIZE command and a NOTIFY command are appended to the end of each command group in sequence. The NOTIFY command lets the GPU generate an interrupt while the SERIALIZE command ensures that the NOTIFY command will be stalled until all the dispatched command group/groups are finished. Upon receipt of an interrupt, the operating system immediately halts the execution of other tasks and switches to execute the interrupt handler. This quick reaction significantly

increases the measurement accuracy. However, neither of their measurements are performed on embedded GPU platforms.

2.3 GPU Execution Time Prediction

Bautin[10] et al. introduced a method based on the number of vertices (or command group size when the number of vertices is zero) and the statistics of the already executed command groups to predict the command group execution time. To be more specifically, they used the elapsed time of the previous GPU command groups and the total number of vertices of these command groups to calculate the average per-vertex-process time. Then they use average per-vertex-processing time and the vertex number if the current command group to predict the command group execution time. If a command group contains no vertex number, they use the command size instead. In [16], a similar method is used, but except the number of vertices and the command group size, the author also takes texture size and swap buffer size into account and chooses one of these factors based on the predefined order to predict the command group execution time.

Both methods may work well in some situations but are in general inaccurate. The reasons are as follows:

1. It is hard to tell which stage in the GPU pipeline takes more time to execute. The author of [16] always assumes loading textures takes more time than other GPU operations, however, it is obviously not correct when the texture size is very small. In that case, loading textures might be much faster than other operations.

2. Even a single time consuming API inside the command group can dramatically affect the total execution time. For example, if glClear is contained in the command group, the GPU may need quite a long time to clear the frame buffer. The total GPU execution time is therefore dramatically increased. However, neither texture size nor the number of vertices are changed.

Kato et al proposed a history-based prediction approach to estimate the command group execution time[24]. In their approach, they assume that: 1. GPU applications tend to repeatedly create GPU command groups with the same methods (APIs) and data size. 2.GPU execution time depends highly on the methods and the data size. Based on these two assumptions, they created a time table to record the execution time for previous command groups with either different methods or data size. For an incoming command group, if the methods and the data size are both same as an existing record in the time table, then the recorded execution time will be assigned as the prediction value. Otherwise, a maximum execution time will be assigned. History-based prediction may achieve a good result when the command group is highly repeated. However, for applications with random inputs or dynamically generated data such as a 3D game, the size and methods inside the command group are often dynamic. In this

case, the history-based approach may never be able to find a predicted value in the recorded time table.

In [22], Raravi et al proposed an method to calculate the upper bound of the execution time by analysing every individual instruction in GPU thread. In their method, they analysed the instructions in the GPU thread conjectured the maximum clock cycles needed to execute every instruction. The method is however not suitable for graphic applications. First, unlike GPGPU applications, graphic applications comprise many APIs instead of using instructions directly in the program. APIs are converted to hardware instructions by the GPU driver. The process of the conversion and detailed information about the hardware instruction are often not available to users. Second, graphic applications are running on a GPU pipeline which involves the cooperation of many hardware modules. Instructions are executed concurrently in different hardware modules and only the module which is the bottleneck (depends on the workload in every modules) in the pipeline will affect the execution time. The instruction executed by other hardware modules may have no influence to the total execution time at all.

In [33], Schnitzer et al presented a fine-grained rendering command-based prediction approach. This approach interprets the semantics of the command group by intercepting the OpenGL and EGL API calls. Among all the APIs, the execution time of time-consuming rendering commands(APIs) are analysed, modelled and predicted individually and then summed up to predict the total execution time of a command group. This approach can predict command groups accurately as long as the prediction models of the time-related rendering commands are built correctly. It has been evaluated with the Nvidia GPU with less prediction error in many applications compared to Kato's approach.

However, hardware architectures and driver properties may differ from vendor to vendor and GPU to GPU. In other words, the implementations and properties of the same rendering commands for different GPUs can be very different. For instance, many embedded GPUs choose to use a tiled-based architecture, where the rendering is operated on the cache memory. Hence the prediction models need to be adjusted based on different GPUs.

Moreover, when predicting the Draw command (the most critical rendering command), the number of fragments generated by the command group is required. In order to predict the number of fragments, a bounding box model is proposed. However, the accuracy of the bounding box model depends highly on the shape of the rendering object.

3 Background

Because the goal our work is to develop concepts and metrics for measurement and prediction of the execution Time of GPU rendering(OpenGL ES 2.0) commands. we give some background knowledge about OpenGL ES 2.0 and embedded GPUs. In Section 3.1, We give an overview over OpenGL ES 2.0. The OpenGL ES 2.0 pipeline is described, emphasizing on the vertex shader and fragment shader. Section 3.2 illustrates the features and the architecture of general embedded GPUs. Section 3.3 describes more details about our target GPU, the Vivante 3D GPU.

3.1 OpenGL ES 2.0

OpenGL ES is a low-level, lightweight API for advanced embedded graphics using well-defined subset profiles of OpenGL for embedded platforms[8]. OpenGL ES 2.0 consists of two specifications including OpneGL ES 2.0 API specification and the OpenGL ES shading Language Specification. It implements a programmable 3D graphic pipeline with the ability to create shader and program objects and the ability to write vertex and fragment shaders in OpenGL ES Shading Language[8].

3.1.1 OpenGL ES 2.0 Pipeline

Figure 3.1 indicates the main stages of the OpenGL ES pipeline. As can be seen from the figure, stages of an OpenGL ES 2.0 pipeline includes vertex shader, primitive assembly, rasterization, fragment shader, per-fragment operations and storing the results into the framebuffer. Next we give a general overview of each pipeline stage.

3.1.2 Vertex Shader

In OpenGL ES, Objects on the display are composed of primitives including triangles, lines and points. The primitives are converted from vertices. Vertices specify the vertex attributes such as the position, color and texture coordinates[28]. The first stage in the GPU pipeline is to process the vertex-related data in the vertex shader. Before we discuss about the vertex shader, we first explain the data types used by the vertex shader[28]:

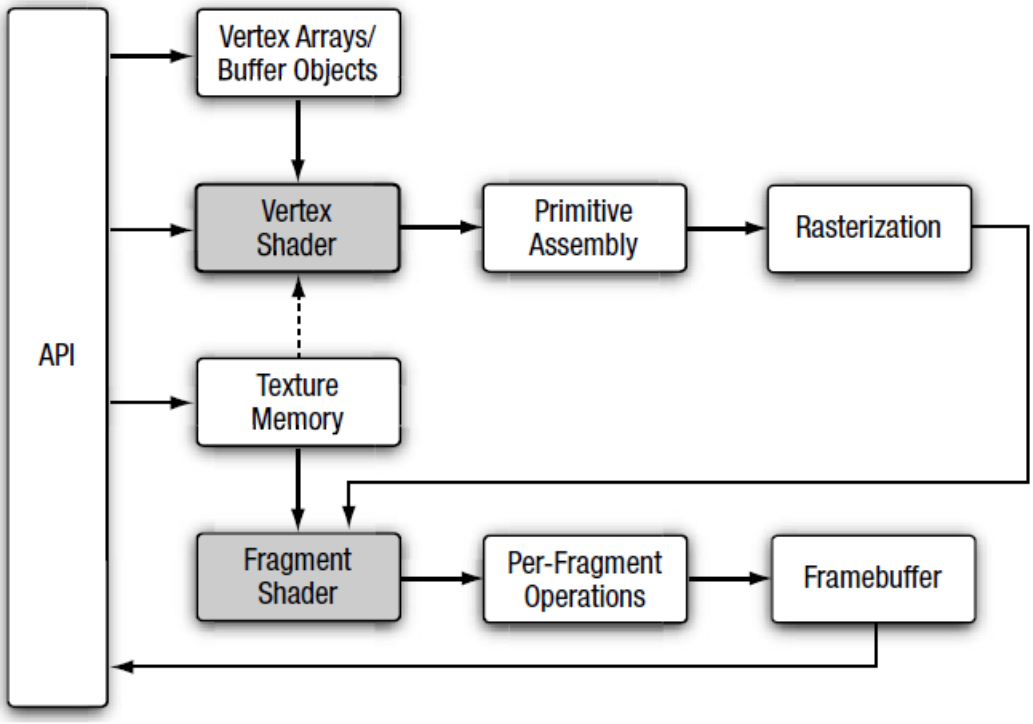

Figure 3.1: OpenGL ES 2.0 Pipeline [28]

- Attributes—Per-vertex data supplied using vertex arrays.
- Uniforms—Per-program variables that are constant during program execution.
- Samplers—A special form of uniform used for texturing.
- Shader program—Vertex shader program source code or executable that describes the operations that will be performed on the vertex.

The vertex shader takes vertices' attributes, uniforms, sometimes also samplers as the input. It is responsible for processing user-specified operations such as rotation, translation, projection and calculating the light equation on every vertex by using the shader programme. After all these operations, it outputs the result as varyings. The main calculation in the vertex shader is matrix multiplication. Equation 3.1 shows the way to convert the coordinates from the original object coordinates to the eye coordinates (with translation, transformation and rotation) by using a model view matrix. In the equation, x, y, z define the 3D location of the vertex, w is the homogeneous coordinate. The model view matrix is displayed in figure 3.2.

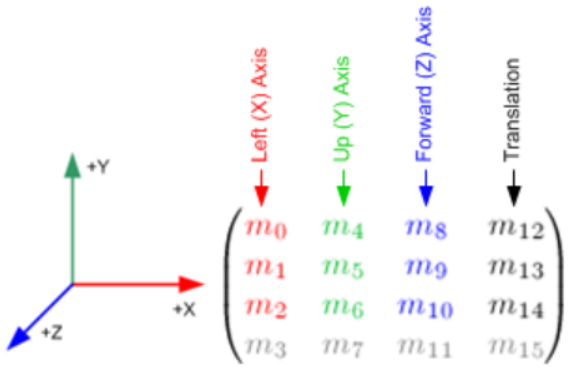

Figure 3.2: Four columns of the model view matrix

$$\begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix} = M_{modeView} \times \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix} \quad \textbf{(3.1)}$$

3.1.3 Primitive assembly

After the vertex shader, the processed vertices form into a set of primitives. Primitives are the basic elements that constitute a graphic. In OpenGL ES 2.0, the basic primitive types are: point, line and triangle. The primitives then go through the primitive assembly stage including clipping, perspective division and viewport transformation. With these operations, the eye coordinates are converted to screen coordinates in order to display the rendering content on the screen(monitor). Figure 3.3 illustrates the process to convert the eye coordinates to the screen coordinates. Next, we explain clipping, perspective division and viewport transformation in more detail.

Clipping: Sometimes, our rendering target is too big to be displayed on the screen. It is a waste of resources to render the object outside the range. That's why we need clipping. Clipping will drop the vertices that are out of the view volume. Before the clipping operation, eye coordinates need to be transformed to the clip coordinates with the help of the projection matrix (Equation 3.1). This is the process to place the object in our view volume. The projection matrix depends on the type of the viewing volume. For instance, if the viewing volume is a general frustum (Figure 3.4), which is normally used by prospective projection. The corresponding projection matrix is shown in equation 3.1 where (l,t,n), (r,t,n), (l,b,n), (r,b,n) are the coordinates of the top left corner, top right corner, bottom left corner and bottom right corner of the near plane in eye coordinates (see Figure 3.4), while f is the z-coordinate of the far plane. After the

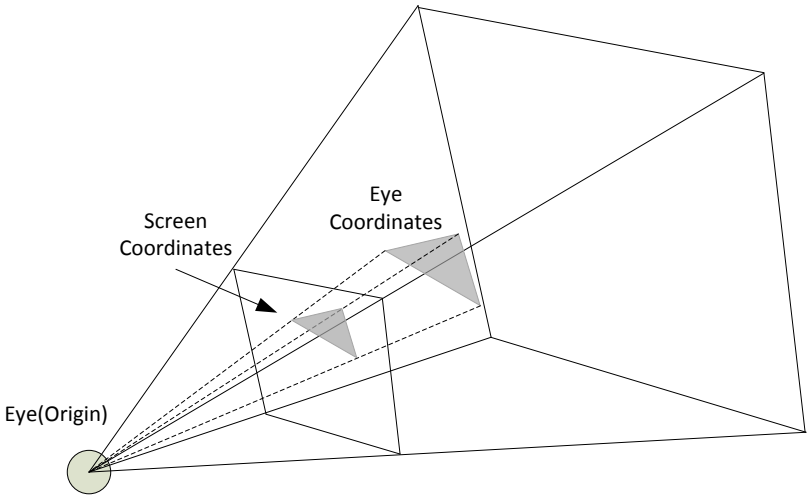

Figure 3.3: From Eye Coordinates to Screen Coordinates

coordinates transformations, the primitives get clipped based on Equation 3.1 and the type of the primitives. What need to be noted is that when primitives lie partially in the viewing volume, new vertices will be generated by the clipping operation.

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = M_{projection} \times \begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix} \quad \textbf{(3.2)}$$

$$M_{project} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad \textbf{(3.3)}$$

$$\begin{aligned} -w_{clip} <= x_{clip} <= w_{clip} \\ -w_{clip} <= y_{clip} <= w_{clip} \\ -w_{clip} <= z_{clip} <= w_{clip} \end{aligned} \quad \textbf{(3.4)}$$

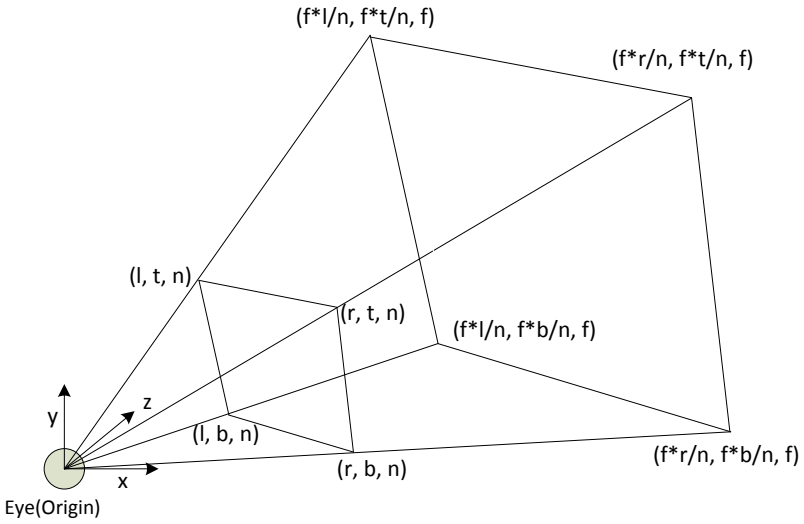

Figure 3.4: Viewing Volume

Perspective Division: The next stage is perspective division, where the clip coordinates are converted to normalized device coordinates:

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \end{pmatrix} \quad \textbf{(3.5)}$$

Viewport Transformation: The last stage of primitive assembly is the viewport transformation. This stage converts the normalized coordinates to the window coordinates which are the coordinates used by the display screen. The conversion is shown in equation 3.1. Parameters x, y, w, h are specified in function glViewport. Parameters x and y specify the screen coordinates of the viewport's lower left corner in pixels; Parameter w and h define the width and height of viewport in pixels. Parameters n and f are specified by function glDepthRange standing for the depth range of the screen.

$$\begin{pmatrix} x_{wind} \\ y_{wind} \\ z_{wind} \end{pmatrix} = \begin{pmatrix} \frac{w}{2}x_{ndc} + (x + \frac{w}{2}) \\ \frac{h}{2}y_{ndc} + (y + \frac{h}{2}) \\ (\frac{f-n}{2})z_{ndc} + (\frac{n+f}{2}) \end{pmatrix} \quad \textbf{(3.6)}$$

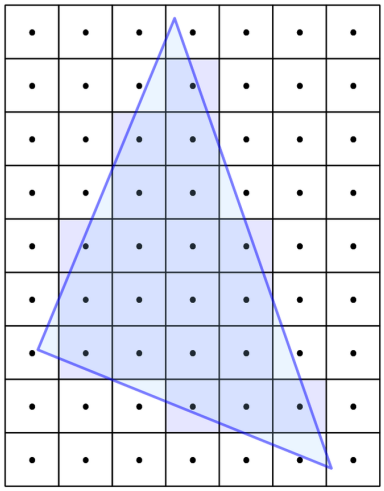

Figure 3.5: Rasterization[5]

3.1.4 Rasterization

After the primitive assembly stage, the primitives that pass the clipping test come to the rasterization stage. Rasterization takes the geometry information (the window coordinates) and the type (triangle, line, point) of each primitives as input, and outputs a set of fragments. Each fragment contains its pixel coordinates in the window and other optional interpolated data such as the color, depth value and texture coordinates of the pixel.

3.1.5 Fragment Shader

After primitives were rasterized, fragments are fed into the fragment shader. Similar to the vertex shader, the fragment shader is also programmable. The user can operate on every fragment, such as assigning a specified color or mapping a specified texture color by using the fragment shader program.

Texture Mapping Texture Mapping maps an image (the texture) onto the surface of the object in a scene. It is a powerful technique to add realism to a computer-generated scene[35]. In order to use textures in OpenGL ES applications, first, texture coordinates are used to specify which area in the image should be fetched. These texture coordinates are bound to vertices so each vertex has a correlated coordinate in the texture image. After the image warping (e.g. projection) and rasterization. The rasterizer will calculate the texture coordinates for every fragment. The texture coordinates are then fed into the fragment shader. When the fragment shader starts to work, it requires the texture colors for fragments based on the texture

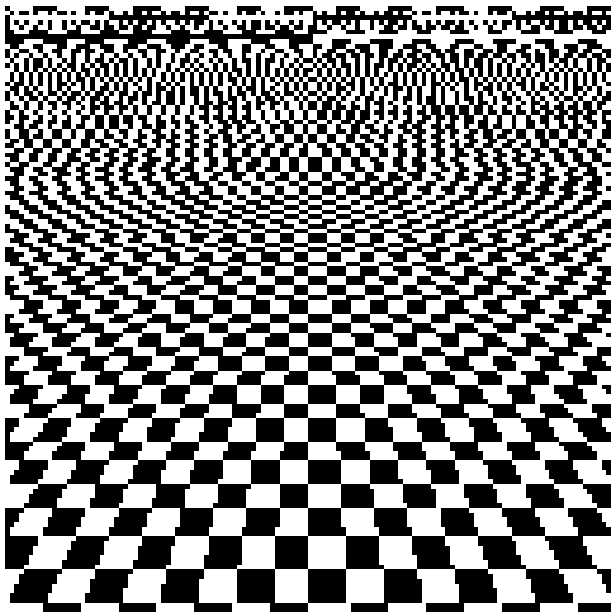

Figure 3.6: Aliasing [7]

coordinates. The texture colors are often offered by another hardware module in the GPU. For example, Vivante GPU has a module called texture engine which is used to fetch the texture from the main memory and calculates the texture color for every fragment. Normally the texture coordinates are not directly at the pixels in the texture, they are between the pixels. Hence texture sampling is required. The simplest texture sampling method is point sampling where the value of the nearest pixel is selected. However, this will often lead to aliasing problems when a small number of samplers are taken from a large texture area (minification). A better and more complicated sampling method that can be used in OpenGL ES 2.0 is bilinear sampling. Bilinear sampling uses the four corner pixel values around the texture coordinate to calculate the texture color. This helps reducing the aliasing problems and improving the image quality.

OpenGL ES also supports a filtering technique called mip mapping to improve the aliasing problem[36][28]. Mip mapping generates a chain of duplications of the original texture. The size of the duplication is always reduced to one-fourth of the previous one, until the size is only a single pixel. For instance, if an original texture size is 128x128 pixels, the size of the duplications will be 64x64, 32x32, 16x16, 8x8, 4x4, 2x2, 1x1. Texture sampling is performed using different levels of texture according to different levels of minification. Because anti-aliasing techniques are already applied during the process of texture compression (mipmap generation), the effect of aliasing is greatly reduced.

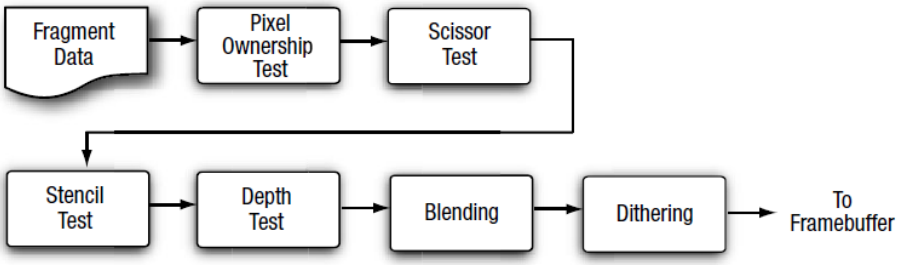

Figure 3.7: OpenGL ES 2.0 Per-Fragment Operations [28]

3.1.6 Per-Fragment Operations

Per-fragment operations are the operations after the execution of the fragment shader. It can affect the visibility and the final result of a pixel. Per-fragment operations include scissor box testing, stencil buffer testing, depth buffer testing, multisampling, blending and dithering[28]. They are all fixed functions.

- Scissor box test, Stencil buffer test and Depth buffer test: Scissor test specifies a rectangle region, pixels outside the region are not writeable. Stencil buffer test and depth buffer test are similar to each other. Stencil buffer test checks the stencil masks in stencil buffer to see if a pixel should be updated or not. Depth buffer test compares the depth value of the current fragment with the depth value stored in the depth buffer. If the depth value of the current fragment is smaller, then the color of the current fragment is written into the color buffer. If the depth value of the current fragment is larger than the depth value in the depth buffer, the fragment will be discarded. What we need to note is that both stencil buffer testing and depth buffer testing require additional memory for the application.
- Multisampling: Antialiasing is an important techniques for improving the graphic quality by trying to reduce the visual artefacts of rendering into discrete pixels[28].Good sampling algorithms and techniques, such as mipmap can reduce the texture aliasing problem. However, the problem still exists when the number of samples is not high enough. Moreover, rasterization can also lead to aliasing problems when the graphic resolution is not large enough. Many methods and algorithms were developed for anti-aliasing. Some common methods include super-sampling anti-aliasing (SSAA) and multi-sampling anti-aliasing (MSAA), the principle of these two methods are more or less the same: increasing the number of sampling to reduce the discrete. The algorithm of SSAA is quite straightforward: rendering is done on a scene whose resolution is higher than the resolution of the original screen. Then the rendering target is re-sampled (down-sampled) to the screen size using a reconstruction filter. Good results can be achieved but the disadvantage is also quite obvious: much more resource is consumed when SSAA is enabled. For example, when 2x2 SSAA is enabled, four times of the computation is

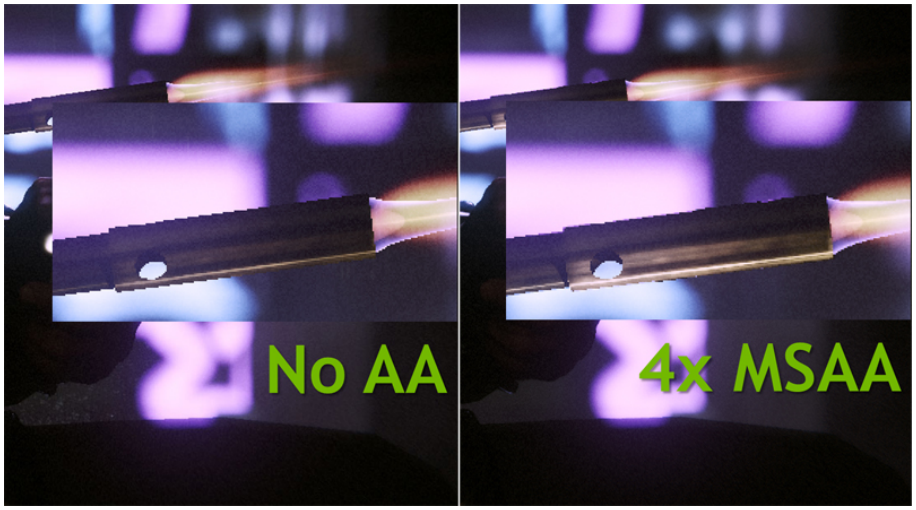

Figure 3.8: Graphic with No MSAA (Left) and with MSAA (Right)[1]

required for the fragment shader. Memory consumption is also increased a lot for storing the additional data (i.e. color,depth buffer of the sub-samples). MSAA, as an alternative, uses a similar algorithm, but for each pixel, the fragment shader is executed only once (that means only geometry is over-sampled, the color (texture) is sampled in the same way as the application without MSAA). The computational complexity is reduced. A similar performance is often achieved by using MSAA compared to using SSAA because pixel shading mostly consists of textures and thus doesn't suffer from aliasing problems (due to mip mapping)[27]. MSAA is widely used by common GPUs and is supported by OpenGL ES2.0.

- Blending and Dithering:

  Blending is a technique that combines the source color and destination color.The source color is the color value which is currently stored in the color buffer. Destination color means the output color of the fragment shader. Blending has effect only when the depth testing is disabled, otherwise, the source color will simply overwrite the destination color (if the new object is in front) or will be discarded (if the new object is in back).

  Dithering, which is also know as color quantization. Pixels with different colors are diffused so the diffused area seems to have a new color to the eyes of human beings. With dithering, more colors can be simulated to enrich the rendering scene. There's no standard dithering algorithm specified by OpenGL ES 2.0, thus the technique is very implementation dependent[28].

3.2 Embedded GPUs

Nowadays, the market of embedded devices such as smart phones and tablets is growing rapidly. Most of the devices integrate 2D and 3D GPUs for graphic acceleration. An estimated over 800 million mobile GPUs were shipped in 2012[34]. However, not a lot of information about the embedded GPUs is revealed to the public compared to desktop GPUs. For example, the Vivante corporation, as a GPU-IP vendor focused on design and licensing embedded GPUs, according to Jon Peddie Research, ranks second in GPU-IP business[30]. However, information about their GPUs are rarely known to the public. Because embedded GPUs are often the choise for embedded systems in order to save power consumption and reduce the cost. it is important for us to have some background knowledge of embedded GPUs. In this section, first, a general introduction to the embedded GPUs is given. Then we discuss more about the architectures of some typical embedded GPUs from different GPU vendors.

3.2.1 Features of embedded GPUs

Similar to the relationship between x86-architecture CPU and ARM-architecture, embedded GPUs also have some unique features compared to desktop GPUs. These features are listed below.

1). OpenGL ES2 is the standard graphic rendering API for embedded system GPUs:

Similar to the position of DirectX for desktop GPUs. OpenGL ES2 is the most popular graphic rendering API supported by embedded GPUs. Over 90 percent of the android devices and all the IOS devices since the iPhone 3GS support OpenGL ES2[39].

2). Power consumption:

Since embedded devices often run on battery, reducing the power consumption directly means a longer time of usage. For 3D GPU computing, large amounts of data are processed in parallel. The power consumption is quite considerable. Besides, the communication between the GPU and external memory consumes also a lot of power when the memory bandwidth is high. Hence increasing the power efficiency is one of the most important task for embedded GPUs.

3). Limited resources and chip size:

When a GPU is integrated into a single chip, the available system resources are very limited. Unlike desktop GPUs who have a large amount of internal GPU memory, embedded GPUs often have only a small amount of cache and have no internal GPU memory. Instead, embedded GPUs often reserve a part of system memory to store all the data they need. The silicon area the GPU consumes has a great influence on the cost since the whole system is integrated on a single die. The GPU designers also have to invest a lot of efforts to balance between the GPU computing performance and the silicon area.

3.2.2 Architecture of The Embedded GPU

In the previous subsection, we have already described some important features of embedded GPUs. These features lead to a hardware implementation which is different from the implementation of traditional desktop GPUs. Here we discuss two main differences in the architecture of embedded GPUs: the rendering mode and the shader.

Rendering Mode

The rendering mode used by embedded GPUs can be mainly divided into two types: immediate-mode rendering and tiled-based rendering. Tiled-based rendering can be again divided into two groups: tiled-based immediate mode rendering (TBIMR) and tiled-based deferred rendering (TBDR). Table 3.1 lists the products of the top embedded GPU vendors and their rendering mode. As can be seen from the table, tiled-based rendering is quite popular and is applied by many vendors such as ARM, Imagination, NVIDIA. However, the traditional architecture, immediate-mode rendering, also accounts for a large portion of the market. It is chosen by Vivante, Intel and AMD.

Immediate-mode rendering:

Traditionally, desktop GPUs trend to render as many frames per second as possible. Immediate-mode rendering, which renders the full scene in one pass, fits the goal. Although this rendering mode can achieve a high frame rate, it also has some drawbacks for embedded GPUs:

1). High memory bandwidth:

Because the full scene is rendered in one pass, the amount of data that needs to be transferred is very big. Besides, frequent memory access is needed, e.g. to read and write the depth/stencil buffer before and after the rendering. These lead to high memory bandwidth and as a result consumes a lot of power. Although complex compression algorithms are introduced to reduce the bandwidth, but the effect is still significant[14]. High bandwidth is a big challenge for embedded system devices since power consumption is a bottleneck as we have already mentioned in the last subsection.

2). Unnecessary work during rendering:

For immediate-mode rendering, overdraw is sometimes unavoidable. As an example, figure 3.9 illustrates a very simple rendering scene: two overlapped triangles. We assume the green triangle is rendered first, then the red triangle is rendered on top of the green one. Under this situation, the overlapped area must be rendered twice. This is a lot of unnecessary work for a complicated rendering scene.

Tiled-based rendering:

Tiled-based rendering GPUs, as an alternative, have good solutions for the drawbacks mentioned above. For tiled-based rendering, the full scene is divided into tiles. Every time, only

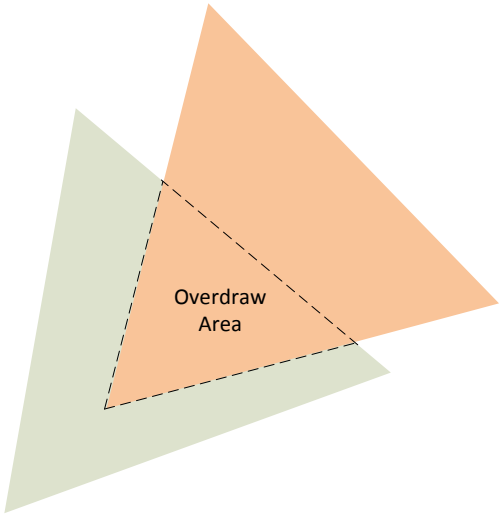

Figure 3.9: Overdraw for Traditional Rendering

one tile of the whole scene is rendered into the tile buffer. The tile buffer resides in the GPU cache which is a small amount of fast memory with very high bandwidth and very low latency. Operations like the depth/stencil testing and blending are directly executed in the cache which avoids a lot of communication with the external memory. Only the final result of the tile is resolved to the external memory which significantly reduces the bandwidth and the power consumption.

Of course it is possible to draw the full scene and let the vertices get clipped when rendering a single tile, but it will cost too much vertex shader performance and introduce a lot of overheads. Hence after the operations in the vertex shader, the GPU will sort triangles into bins for each tile. This is implemented differently by different hardware vendors[31]. Then only the triangles (primitives) that affect the tile will be rasterized. Until here, the unnecessary work during rendering (overdraw) is still not solved. We call this kind of rendering mode: Tiled-based immediate mode rendering (TBIMR). To solved the overdraw problem, tiled-based deferred rendering (TBDR) was developed. TBDR uses one of the hidden surface removal (HSR) algorithms[11] such as the z-buffer algorithm to check the depth value of each tile before the rendering and avoid the overdraw in the front-to-back case but not the back-to-front case[40]. PowerVR, as a unique, sorted the depth of each pixel before the rendering so only the visible pixel is rendered even for the back-to-front case. Again we take the two overlapped triangles as an example (Figure 3.10). For tile D, both the green triangle and the red triangle are involved

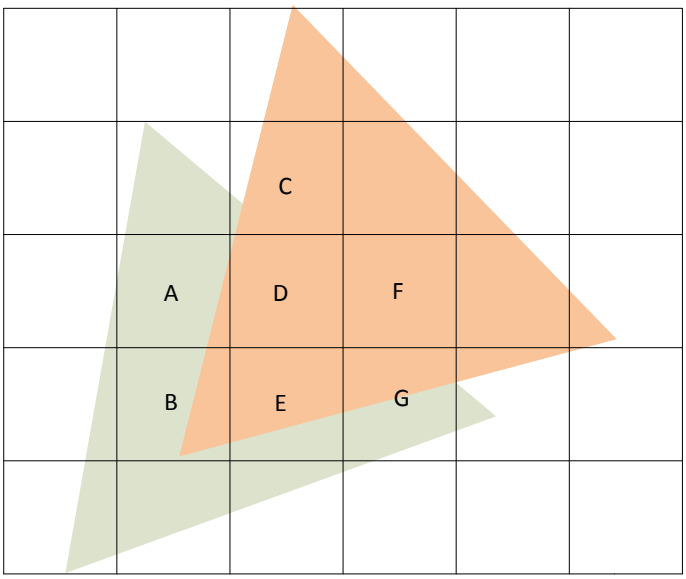

Figure 3.10: Less Overdraw for Tiled-based Rendering

in rendering. Before the rendering, the GPU sorts the depth value of each pixel that belongs to the green and red triangle in that tile. Then it discards all the green pixels that are covered by the red one. The overdraw problem is perfectly avoided. PowerVR's architecture works fine in low-poly-count case. However, it is possible that the performance drops dramatically for complex rendering scene where a large amount of primitives is involved[40].

Unified Shader Conventional GPU architectures often separate the vertex shader and fragment shader and implement them with different hardware. A different hardware implementation is to use a unified shader for computing both vertex shading and fragment shading on the same hardware. This is feasible because vertex shader and fragment shader have similar instruction set architecture (except for special instructions such as texture sampling)[38]. Because the chip size of the embedded GPUs is very limited and is critical to the cost, unified shader becomes a good solution for embedded GPUs since it can simplify the hardware complexity and thus reduce the size of the chip. Another benefit of unified shader is that it can balance the work between the vertex shader and fragment shader dynamically. For instance, in [38], the author points out that, during the fragment operations, if cache miss occurs when the fragment shader tries to fetch the texture, the unified shader can switch to vertex shader and do some vertex computation. Once the texture is available, the shader switches back and finishes the rest of

Company	Product	Pipeline	Notes
ARM	Mali	TBIMR	Unified shader, 2-4 math pipes per core
Imaginination	PowerVR	TBDR/HSR	Latest is Rogue (S6). Unified shader. DX11 support
Qualcomm	Adreno	FlexRender	Unified shader. "FlexRender" = automatic switching between direct render (IMR) and tile-based deferred rendering (TBDR).
NVIDIA	Tegra	TBDR TBIMR	Evolution 1. Tegra 1/2/3/4: non-unified TBDR architecture 2. Logan: Kepler-based GPU, TBIMR 3. Parker: Maxwell-based GPU, TBIMR
Vivante	ScalarMorphic	IMR	Unified Shader.
Intel	Gen/Atom	IMR	Market leader in integrated graphics. Atom-based devices using Imagination PowerVR
AMD	Radeon	IMR	Hondo/Temash pipes.

Table 3.1: Rendering Mode of different embedded GPUs [34]

the work on the fragment shader. The optimization of the scheduling saves a large amount of execution time.

3.3 The Vivante GPU

In the end we have a look into the some hardware features of the Vivante GPU, which is used as the evaluation platform of this work.

Although Vivante GPUs use immediate-mode rendering, we notice that the rendering is actually operated on small tiles which reside in the cache memory. Vivante GPUs use fully set associative cache for the rendering target buffer(rendering scene)[20]. During the rendering, if no cache miss happens, the data always resides in the cache. Once a cache miss happens, the GPU replaces a cache line and writes back the data to the external memory.

Vivante has invented an efficient rasterization technique called tile-based rasterization[15]. This also explains how the GPU does benefit from rendering on the cached tiles. Traditionally, scanline rasterization is one of the most commonly used rasterization techniques. Where a primitive is converted to pixels one horizontal row by the rasterizer at a time. The drawback of scanline rasterization is that it introduces a lot of redundant computation, since each edge is

checked for intersection with each successive scanline. For tiled-based rasterization, if the edge is in the tile, then it only needs to be checked for once. Hence tiled-based rasterization technique avoids these redundant computation. Further more, sophisticated pre-fetch algorithms were also developed to fetch the tiles in an optimized sequence to reduce the total fetch number and increase the cache utilization situations.

During preliminary test, we found out that Vivante GPUs also support early Z-test(similar to depth test) in the stage of rasterization. Normally, the depth test is done after the process of fragment shader. However, since most of the fragment shader does not change the pixel depth value. It would be a waste of GPU processing power to process the fragment which is overlapped by other fragments. Thus emerges the need of early Z-test. Early Z-test is performed in the stage of rasterization. It determines whether a fragment is visible based on the depth value. If the fragment is not visible, it is immediately dropped before it is sent to the fragment shader. In this way, unnecessary fragment processing is avoided.

4 System Model

In this chapter we discuss the system model of the GPU driver. Despite that the GPUs are designed and manufactured by different vendors, the system models of GPU drivers share many similarities. Since our work is implemented and evaluated on the Vivante GPU platform, we use the Vivante GPU as an example to explain the behaviour of the GPUs driver. We describe the system model of the GPU driver focus on the knowledge of GPU command execution flow.

As we have mentioned in the last chapter, almost no literature of the Vivante GPU is available. Although Vivante has opened their kernel space driver to the public, the user space driver is closed source. The knowledge of the user space driver and the hardware is derived from the related kernel space driver code, many tests, debugging and the reverse engineering project Etnaviv[26].

4.1 System Model Overview

Figure 4.1 illustrates the general system model of an OpenGL ES application. The system can be divided into four layers, they are: OpenGL ES API layer, user space driver, kernel space driver and GPU hardware.

In order to render some objects, OpenGL ES applications use OpenGL APIs to send rendering commands to the GPU. The rendering commands are converted into GPU hardware commands by the user space driver and passed to the kernel space driver. The kernel space driver is response for dispatching the commands to the GPU. Finally, the rendering commands are executed by the GPU.

4.2 User Space and Kernel Space Communication

The Vivante GPU driver, like many other drivers, can be divided into two parts: user space and kernel space. The communication between user space and kernel space is established only with the system call ioctl(). Specifically, when the user space wants to send a specified operation to the kernel space driver, the user space first builds a structure called DRIVER_ARGS, stores all the related data into the input buffer and passes the structure via ioctl() to the kernel.

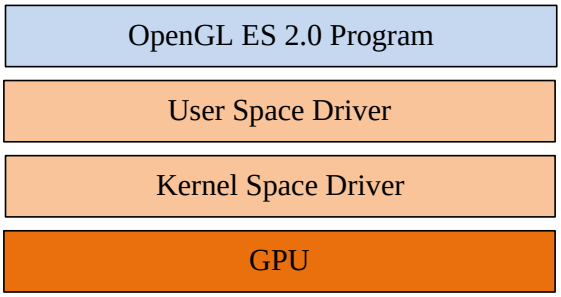

Figure 4.1: System Model

The kernel will then execute the operation, store the result in the output buffer (for some operations) which can be accessed in user space. The struct is declared in gc_hal_kernel_os.h:

DRIVER_ARGS

```
typedef struct _DRIVER_ARGS
{
   gctUINT64         InputBuffer;
   gctUINT64         InputBufferSize;
   gctUINT64         OutputBuffer;
   gctUINT64         OutputBufferSize;
}
DRIVER_ARGS;
```

InputBuffer The starting address of the input buffer. The buffer is filled in user space;

InputBufferSize Size of the input buffer;

OutputBuffer The starting address of the output buffer. The buffer is filled in kernel space;

OutputBufferSize Size of the output buffer;

The input buffer and output buffer use the same interface called gcsHAL_INTERFACE to pass parameters and the pointers of the data. gcsHAL_INTERFACE is declared in gc_hal_driver.h:

gcsHAL_INTERFACE

```
typedef struct _gcsHAL_INTERFACE
{
   /* Command code. */
   gceHAL_COMMAND_CODES command;

   /* Hardware type. */
   gceHARDWARE_TYPE     hardwareType;

   /* Status value. */
```

```
    gceSTATUS            status;

    /* Handle to this interface channel. */
    gctUINT64            handle;

    /* Pid of the client. */
    gctUINT32            pid;

    union _u{
             .....
    }u;
}gcsHAL_INTERFACE;
```

command Command code, it decides which operation the kernel space driver should execute. More command codes are listed in appendix A.2.

hardwareType The driver is compatible with different models of Vivante GPUs. This parameter specifies the hardware that you want to use.

union u The elements inside the union are based on the command codes you choose. For some command code, e.g. GET_BASE_ADDRESS, the union contains only parameters. For some other command codes, e.g. COMMIT, the union contains the parameters and pointers to the data.

The most relevant command code of our work is the command code "COMMIT". "COMMIT" is used to submit a specified task (a group of GPU commands) to the GPU. More details and concepts about GPU commands are described in the next section.

4.3 Command, Command Group, Command Buffer, Command Queue and Fetch Engine

In order to understand how GPU commands are generated, submitted to the GPU and how GPU fetches and executes the commands, we need to first understand some important concepts. They are GPU commands, command group, command buffer, command queue and the fetch engine. GPU commands are stored in command buffers and are submitted to the GPU in the form of groups. The submitted command groups are scheduled by the command queue and are fetched by GPU using the fetch engine. More details are explained below.

4.3.1 Command

GPU Command is binary code that is executable by the GPU. Graphic APIs such as OpenGL ES 2.0 APIs reside in an abstract layer and are independent to the GPU hardware. In order

to make the graphic API executable by the GPU, they have to be converted into specific hardware-executable GPU commands. The conversion is done in the user space driver.

The format of Vivante GPU command comprises two parts: the operation code and the parameters. Each operation code stands for a "basic" GPU operation. The operation code is followed by the corresponded parameters. The length of the parameters varies from command to command. Thanks to the effort by the Etnaviv project, the operation code has been interpreted and can be found in [25]. Among all the operation codes, "LOAD" is most frequently used. "LOAD" loads parameters into different GPU registers to configure and control the GPU hardware.

4.3.2 Command Group:

A command group is a batch of GPU commands that form a certain task to be executed by the GPU. Like most of the GPUs, Vivante GPU do not support preemption, command groups are non-interruptible. That means the next command group can be executed only when the previous dispatched command group is finished.

Vivante integrates different GPU cores (2D, 3D, Vector) into the same hardware and let the cores share the same memory control system. All the command groups submitted by the kernel space driver will first be fetched by a GPU hardware module: the command fetch engine (more details see 4.3.5). The fetch engine then decides the destination of the command group by checking the first 32-bytes of each command group, a "PIPE" command. The "PIPE" command specifies the pipe that should be used to transfer the data to the right destination. The last 64-bytes in every command group are a "LINK" command appended by the kernel space driver before the command group is submitted to the GPU. The "LINK" is used to let the execution be scheduled back to the kernel command queue (we will mention that soon).

Normally, the size of a command group is relatively small since the main data of the application, such as the vertex coordinates, texture data, have their separated memory space (section 4.7), only the pointers of the data are passed by the command group.

4.3.3 Command Buffer:

Command buffer is a block of contiguous memory that holds GPU command groups. In order to allocate memory for the command buffer, The user space driver sends a request to the kernel space driver asking for contiguous memory through ioctl() with the command code ALLOCATE_CONTIGUOUS_MEMORY.

It should be noted that, before a GPU command is pushed into the command buffer, the user space driver always checks if there's enough space left in the command buffer to fill the GPU command. Once the command buffer detects that there's not enough space left to fill the next

GPU command, the user space driver will instantly commit the current commands in the buffer as a command group to the kernel space. While normally, the commitment occurs only after special a API such as glFlush or gFinish is called. After committing the command group, a new command buffer is in use. An EVENT command is attached to the last command group in the command buffer. Once the command group finishes execution on the GPU, because of the EVENT command, the GPU will send an interrupt to the kernel space driver to let the kernel space driver know that the command buffer is completely executed and there's no need to hold the memory any more. The kernel space driver then notifies the user space via signals that the command buffer can be reused. Base on our experience, the command buffers are always associated with certain fixed signal numbers respectively (in our case signal number from 14 to 17).

4.3.4 Command Queue:

After the command group is committed by the user space, the kernel space driver takes over control. Command queue is used by the kernel to schedule the committed command groups. A command queue is also a block of contiguous memory similar to the command buffer. Struct _gcskCOMMAND_QUEUE keeps the related information of a command queue. The command queue contains the pointers to the physical addresses of all command groups, it also contains short commands such as EVENT, WAIT, LINK for synchronization and for scheduling. Once a command queue is created, a mutex will be set to indicate that the queue is in use. An EVENT command will be pushed into the head of the command queue. Once the EVENT arrives at the GPU, the GPU will inform the kernel that the last command queue has been fully executed. The kernel will then release the mutex of the last command queue hence it could be reused. The maximum number of command queues can be defined in file gc_hal_options.h. GPUs like NVIDIA's Fermi and Tesla support multiple channels for different applications. For Vivante GPUs, the same command queue is shared by different applications. For different GPU cores (like GPU 2D and 3D cores), the command queues are separated.

Figure 4.2 shows the relationship between the command group, command buffer and command queue as described above.

4.3.5 Fetch Engine

Before we explains the command execution flow, There's still one more concept needs to be discussed: the fetch engine.

Figure 4.3 shows the basic architecture of the Vivante 3D GPU. As we can see from the figure, FE (Graphics Pipeline Front End, also called Fetch Engine) has direct access to the memory controller. It takes care of loading state values into individual modules of the

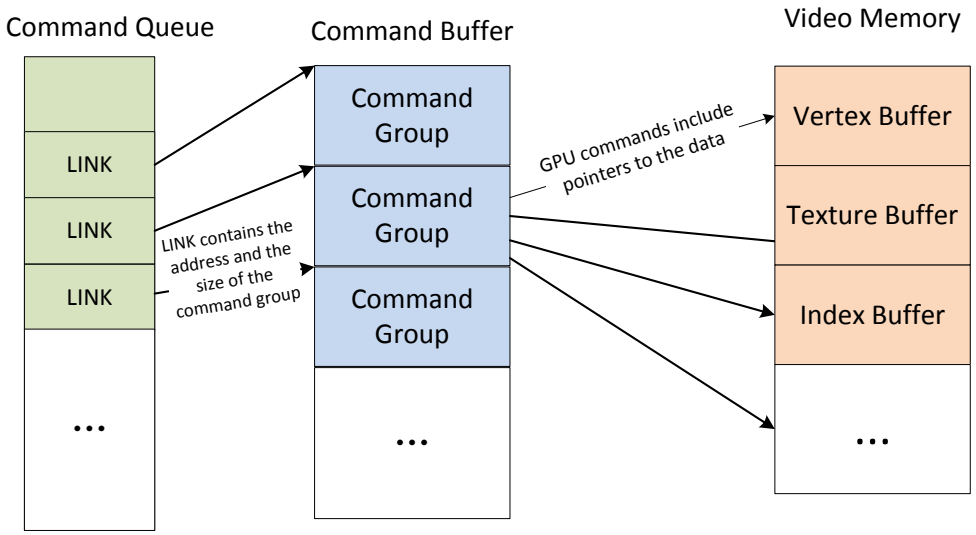

Figure 4.2: GPU Command Group, Command Buffer and Command Queue

GPU, kicking off primitives rendering, synchronization, and also supports basic flow control (branch,call,return)[26].

Since one of our goals is to measure the execution time of GPU command groups, we focus on the basic flow control of the FE. More specifically, we focus on how and when to fetch command group by the FE. For that, we need to understand some important GPU commands, these are WAIT,LINK and EVENT (LOAD).

Command WAIT: when command WAIT (beginning with operation code 0x38) is fetched by the FE, the fetch engine waits for a certain number of clock cycles. No further GPU commands are fetched during this time.

Command LINK: the LINK command (beginning with operation code 0x18) informs the FE to fetch a certain length of commands. It contains two parameters, one is the starting address of the target GPU commands, the other is the length of the target GPU commands. Normally, a LINK is always appended at the end of every command group, to ask the FE to fetch a new group of commands.

Command EVENT: the EVENT command belongs to the LOAD command. As we have mentioned in subsection 4.3.1, LOAD is the most frequently used command. It loads a certain value into a specified GPU state register. EVENT command is a LOAD command that loads value into the event register. When the event register is loaded, an interrupt will be triggered by a certain

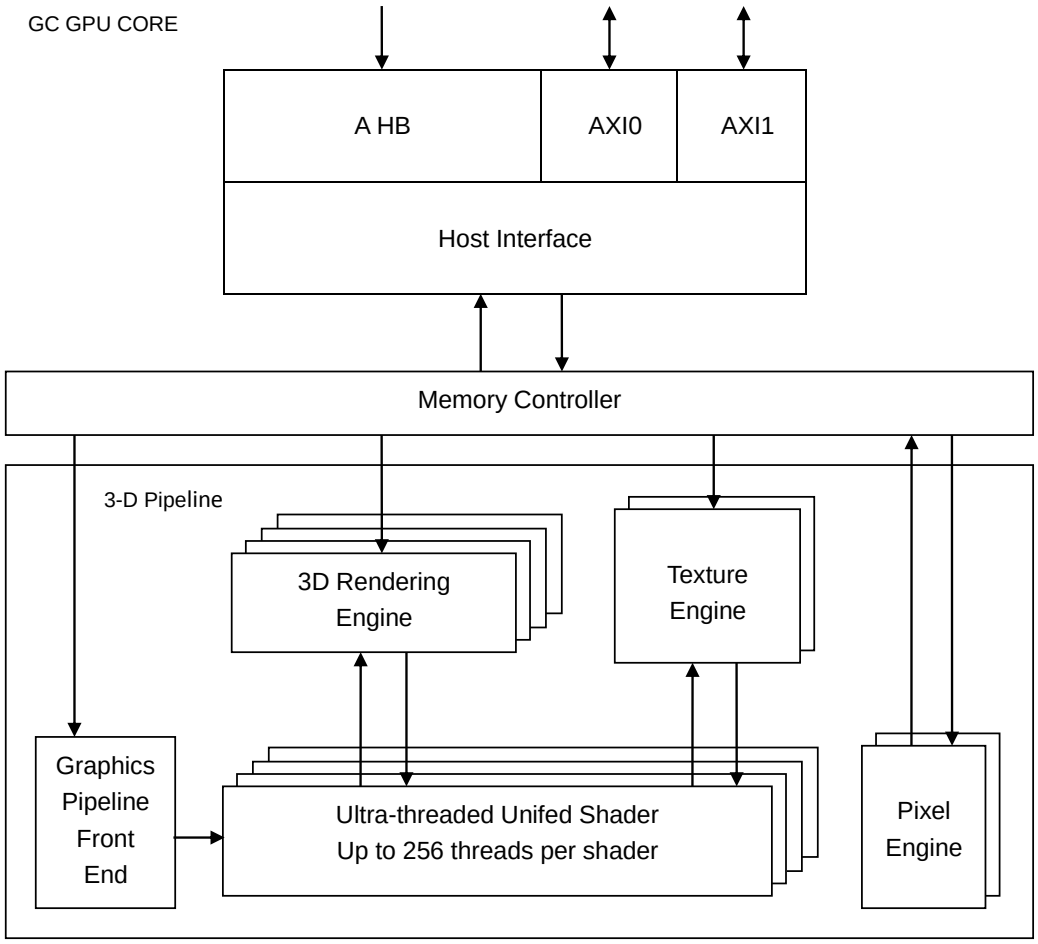

Figure 4.3: Vivante GPU Architecture 3D [20]

part of the GPU hardware. The part to generate the interrupt is also specified by the EVENT command. More details about EVENT and interrupt will be discussed in section 4.5.

Here are some example of the these commands:

Example WAIT: [380000C8 189FA038] Wait for 0xc8 clock cycles
Example LINK: [40000002 189FA028] Fetch the address starting from 0x189FA028. Fetch length is two words
Example EVENT (LOAD): [08010E01 00000044] Load value 1 into event register with offset 0x0E01»2, triggers an interrupt by the pixel engine

4.4 Commands Execution Flow

In the last section, all the required concepts for the commands execution flow are presented, next we put the pieces together and explain the whole execution flow of the GPU commands.

1. Fetch Engine Startup When the GPU kernel driver is installed by the system, function gckCOMMAND_Start will be called to power up the fetch engine. In order to do that, the function will first start a new command queue, then insert a WAIT/LINK pair which forms a loop in the new command queue. After that, the driver writes the starting address of this WAIT/LINK pair into the fetch address register through io-mapped memory to start the Fetch Engine. The Fetch Engine will then stay in the WAIT/LINK loop (see Figure 4.4) until a command group is committed from the user space.

2. Command Buffer Allocation: At the beginning of each application, four command buffers are allocated. To do that, the user space driver will send requests to the kernel driver asking for contiguous memory through ioctl() with the command code ALLOCATE_CONTIGUOUS_MEMORY. Based on our observation, the user space driver always requests four blocks of contiguous memory, each with a size of 128 KB, regardless of the content of the application.

3. Command Group Submitting: When a command group is committed by the user space driver, the kernel space driver first appends a new WAIT/LINK pair at the end of the command queue (step 1 in Figure 4.5) so once this command group has been executed, the FE can stay in the new loop. After that, the driver will append a LINK to the end of the command group to jump to the new WAIT/LINK pair (step 2 in Figure 4.5). Once the preparations are finished, the WAIT in the old WAIT/LINK pair will be replaced by a LINK (step 3 in Figure 4.6) pointing to the starting address of the command group and let the Fetch engine fetches the entire command group. The size and the starting address of each command group are calculated in user space and passed to the kernel space. At this time, the FE can jump out of the loop and start to fetch the command group. This is also the time that GPU takes over control of the task. After the execution, the GPU will stay in the new loop as long as no more command is submitted, because of the LINK we added in step 1.

Now we have explained how and when the command group fetched is by the FE, In order to know when the execution of the command group is finished, we need to understand more about the EVENT and interrupt mechanism. This will be discussed in the next section.

4.5 EVENT, EVENT Queue and Interrupt Mechanism

This section describes the concept of EVENT command, EVENT queue and presents the implementation of the GPU interrupt mechanism.

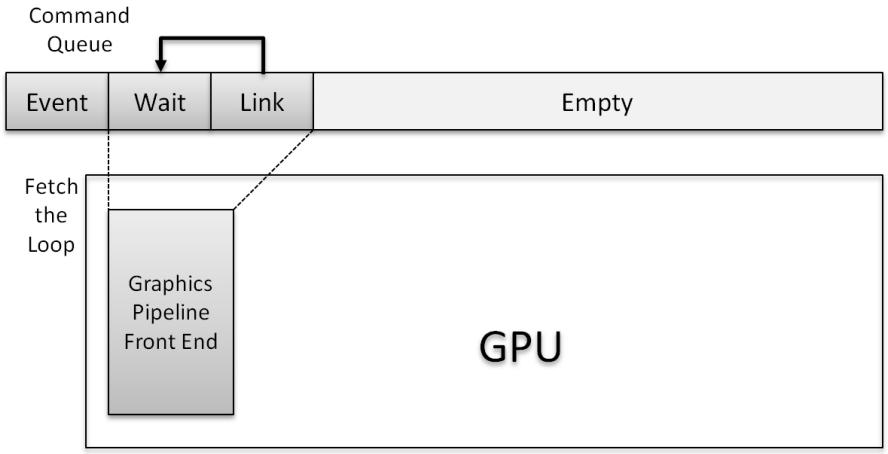

Figure 4.4: GPU idle

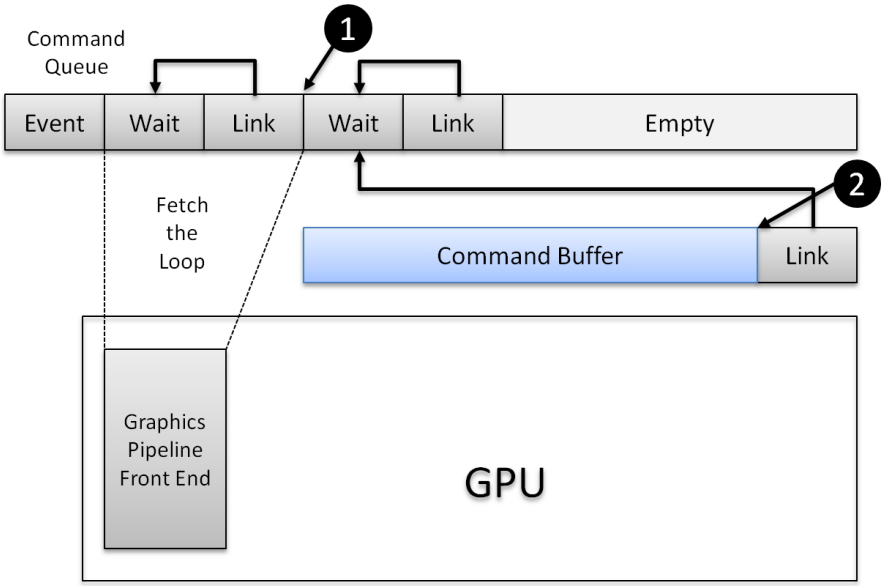

Figure 4.5: Submit Command Buffer 1

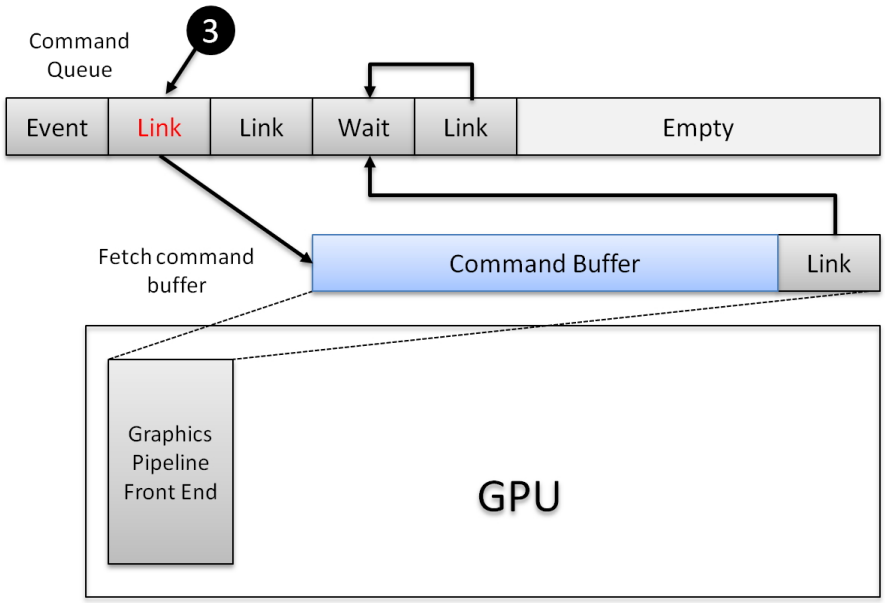

Figure 4.6: Submit Command Buffer 2

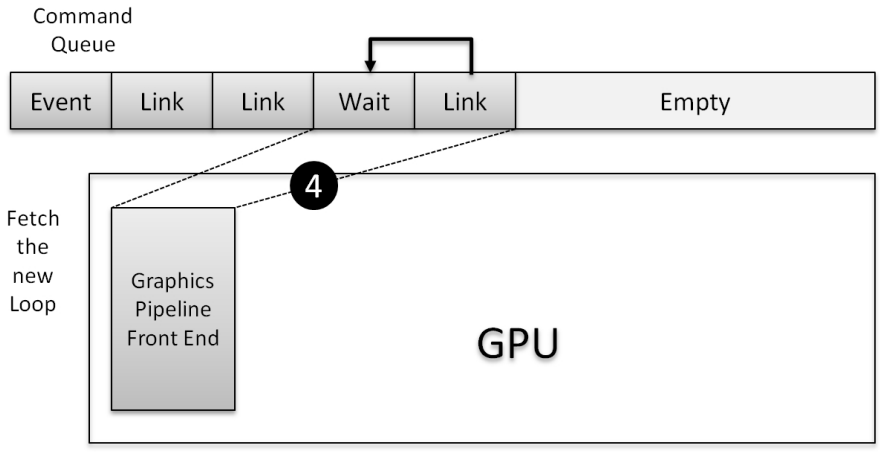

Figure 4.7: GPU idle after Command Execution

As we mentioned in section 4.3.5, Vivante provides an "EVENT" command to set a bit in a specified register (the event register) and to generate an interrupt from the GPU to the CPU (see Figure 4.8). The part of the GPU component that generates the interrupt can also be specified. For instance, when pixel engine is set as the place to generate the interrupt, if we append a "EVENT" command after the command group, the CPU will receive the interrupt generated by the "EVENT" when the previous command group finishes its execution on the pixel engine.

We need to remark that the execution of the EVENT command is very fast. If the GPU executes two EVENT commands within a short period of time, the GPU may write the event register during the interrupt service routine. We did some tests and did find out that once the event register is read, the value inside the register will be cleared automatically by the GPU hardware. That means, if the event register is written before the read operation of the interrupt service routine, we will find two or more bits set in the event register. If the event register is written after the read operation, we will only find one bit set but we will immediately get a second interrupt.

In the event register, bit 31 and bit 30 are reserved for special purposes. Bit 31 is reserved for AXI bus error, bit 30 is for dumping the MMU debug information on a MMU exception. These two bits in the event register are marked automatically by the GPU without sending the "EVENT" command. The other bits are associated with event queues.

An event queue, defined by structure gcsEVENT_QUEUE, contains an event list. Before an EVENT command is submitted to the GPU, the driver will try to obtain a free event queue and add all current events (tasks) to the event list inside the event queue. In the kernel space driver, there are in total of 30 queues available which are assigned with number 0 to 29. Each queue is associated to an unique bit in the event register. After the EVENT command is executed, the bit associated to the event queue number in the event register will be set. A separated event handling thread was started at driver initialization time. The thread is always running in a loop and waiting for the event notification. At the end of the interrupt service routine (ISR), after the ISR reads out the marked bit in the EVENT register, the ISR will notify the event handling thread, the thread will then process the task/tasks based on the bit/bits set in event register with function gckEVENT_Notify.

The "EVENT" command is applied in many situations which is demonstrated in the following list.

(1) When the AXI bus error occurs, bit 31 is set.

(2) Notify the CPU to dump the MMU debug information on a MMU exception. Bit 30 is set.

(3) Notify the CPU to free memory, to unlock memory and to unmap user memory.

(4) Notify the CPU that a command buffer has been executed.

(5) Notify the CPU that a new command queue is in use.

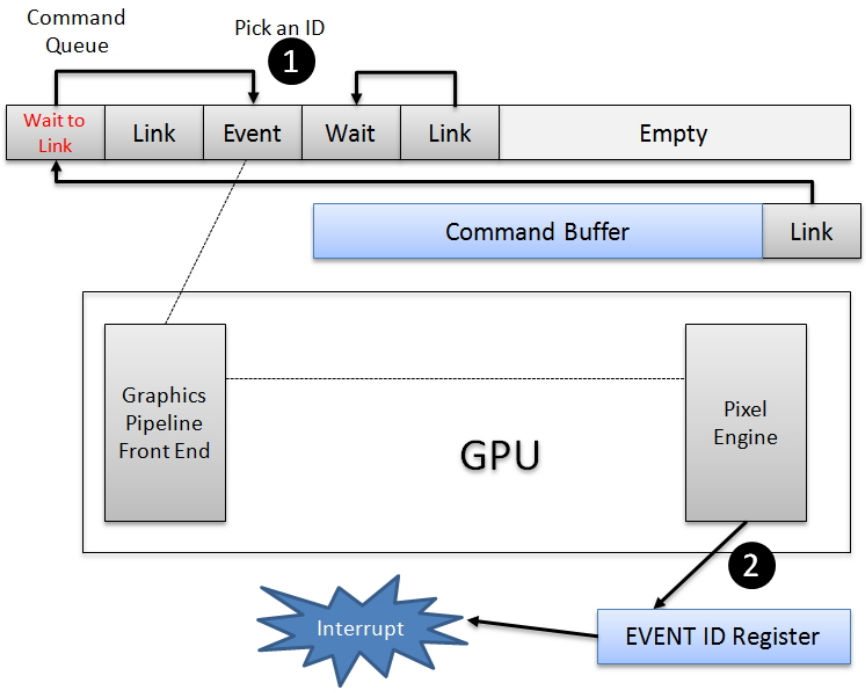

Figure 4.8: GPU receives Event and Send Interrupt

(6) Notify the CPU to signal the user space.

Now let us go back to our user command group execution flow: after a command buffer is submitted to the GPU, the kernel will automatically submit a small buffer which contains only an EVENT command to the GPU with event content: gcHAL_COMMIT_DONE. Once the associated interrupt is triggered, we know that the command group was fully executed by the GPU.

4.6 Signals and Synchronization

Figure 4.9 shows how the synchronization is realized between different layers. While the synchronization between kernel space and the GPU is realised by the EVENT command, the synchronization between user space and kernel space is implemented with signals. Every signal contains signal number so that when the user space is signalled, it can executed the corresponded tasks based on the signal number. The association between the signal and EVENT is established in kernel space with the event queue. That means signals are registered to an event queue and will be processed by the event handling thread.

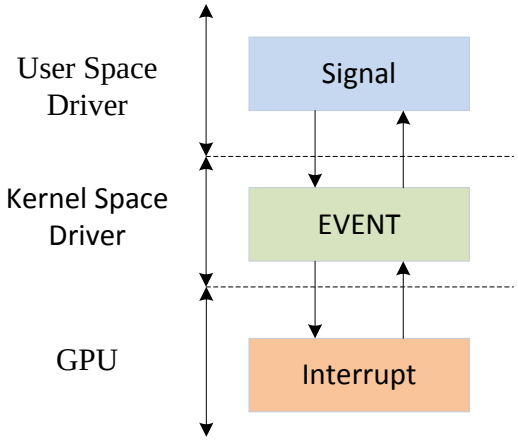

Figure 4.9: Synchronization

4.7 Linear Video Memory

From section 4.3, we know that each OpenGL ES 2.0 application uses command groups for passing commands to the GPU. What we need to note is that, most of the related data for rendering is not passed by command groups but by blocks of linear video memories in different memory spaces. For instance, when an OpenGL ES 2.0 application wants to pass the vertex attributes to the GPU, the user space driver will allocate a block of video memory through system call. Then the vertex attributes will be pushed into the allocated video memory. The starting address of the video memory is stored in the command group to inform the GPU the location of the video memory.

Video memories are allocated through ioclt() with the command code gcvHAL_LINEAR_VIDEO_MEMORY and are accessible by both the user space driver and the GPU. They are used for storing all kinds of data as rendering input or rendering output. According to different usages of the linear video memories, they are divided into different types (such as vertex buffer, depth buffer, color buffer) and named as different surfaces. That means every block of linear video memory is a surface. For simplicity and better memory management, the driver will allocate a default size of memory for some surfaces (such as the the vertex buffer) and use signals to change the states of these surfaces and informs the user space when these surfaces can be reused. This is similar to the signal used by the command buffer. For some other surfaces such as the texture buffer, image buffer, the size of the surface is adjusted by the data size. Normally, the surface

size is a little bit more than the data size. All available types of surfaces are enumerated in gceSURF_TYPE in file gc_hal_enum.h. Here we discuss some of the most important surfaces.

Render Target: The back framebuffer which is used during the rendering. It contains the pixel value (red, green, blue and alpha) of the object to be rendered. Before displaying the final image, the data in the render target will be resolved to the front color buffer. A few tiles of the render target are cached in the GPU. The render target is defined as gcvSURF_RENDER_TARGET in the driver.

Depth Buffer: For depth buffer, the memory is allocated only if the depth buffer is enabled by the program. A small portion of depth buffer is cached in the GPU depth cache. Depth surface is defined as gcvSURF_DEPTH in the driver.

Tile Status Buffer It is allocated along with the render target or the depth buffer. Tile status surface stores the tile information of the render target and the depth buffer. It is defined as gcvSURF_TILE_STATUS.

Vertex Buffer: The vertex buffer contains the attributes of the vertices. As default, five vertex buffers, each with size of 1 MB, are allocated at the beginning of each OpenGL ES application. Vertex Surface is defined as gcvSURF_VERTEX.

Texture Buffer: The buffer for the texture data. The memory is allocated only if texture is used in the program. The size of the texture buffer depends on the size of the texture. A small portion of texture is cached in the GPU texture cache to improve the speed of texture fetching. It is defined as gcvSURF_TEXTURE in the driver.

Image Buffer: Image Buffer, the front color buffer that stores the final image. Defined as gcvSURF_BITMAP in the driver.

4.8 Alignment of The Render Target

As we have mentioned in the previous section, render target is a block of video memory that contains pixel values (red, green, blue and alpha) of the rendering object. The render target is shown on the top of figure 4.10 in the form of a pixel array, where each of A1, A2... B1, B2.. stores the color of a pixel. The memory addresses of the pixels in the pixel array are contiguous. For instance, assume each pixel takes four bytes memory, if the memory address of A1 starts from 0x00, then the memory address of A2 starts from 0x04.

For efficiency and optimization, the render target is managed in different levels of tiles. The alignment of the tiles is demonstrated in figure 4.10. The top left corner of the figure shows an example of a 4x4 pixel tile, this is the smallest tile in the render target. The index on the tile grid indicates the sequence of the pixel in the pixel array. As can be seen from the figure, the memory addresses of the pixels in 4x4 tiles are contiguous. If tile size is larger than 4x4 pixels, the memory addresses of pixels are not all contiguous. The memory addresses of pixels are

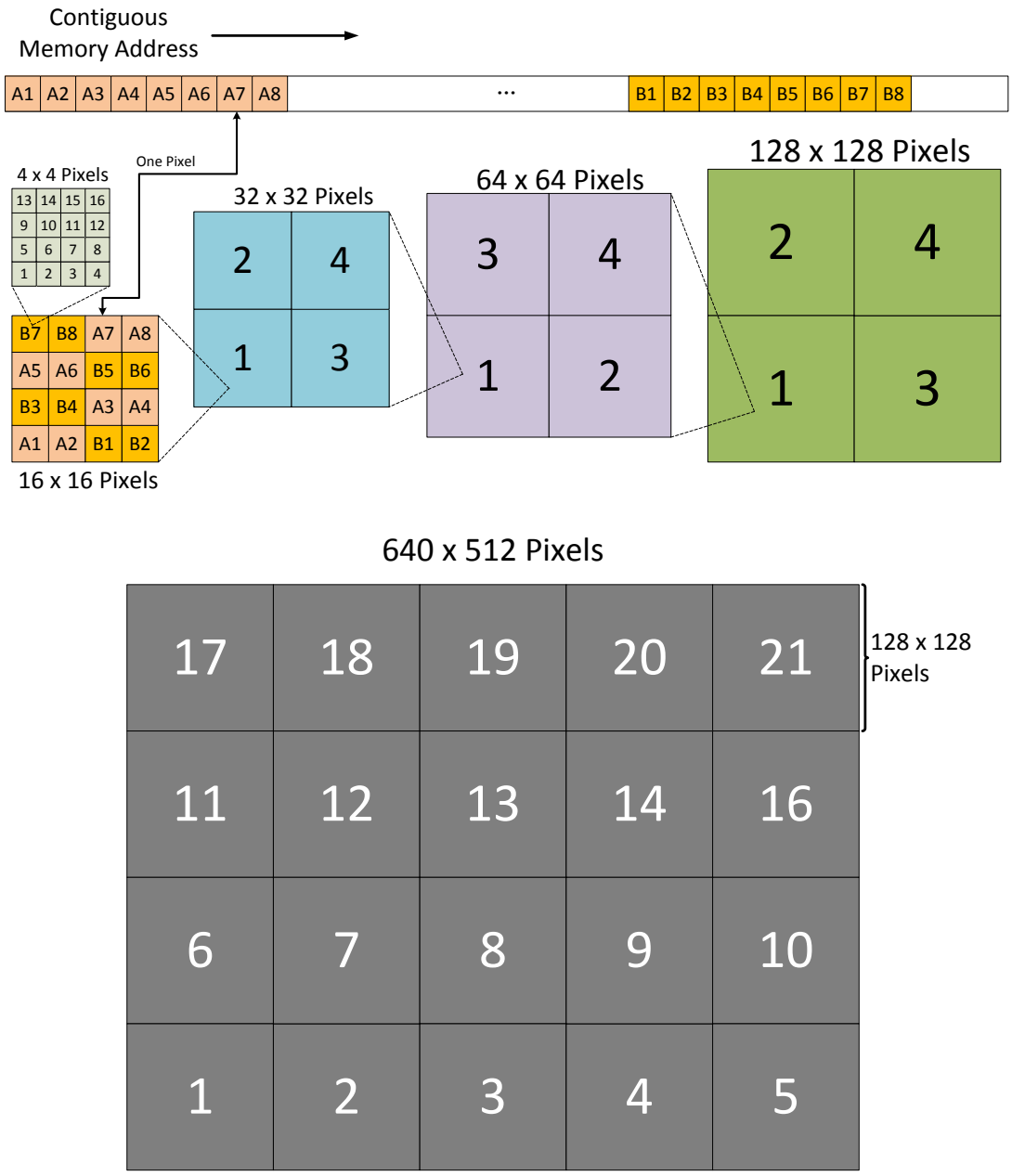

Figure 4.10: The Alignment of the Render Target

divided into two parts. Half of the pixels are stored in the first half of the pixel array, the other half pixels are stored in the second half of the pixel array. These two parts of pixel data are always interleaved. We take the 16x16 pixel tile in the figure as an example. In the tile, we can see that pixel A1, A2, B1, B2 are contiguous in the image. However, their memory address are not all contiguous. The memory addresses of A1 and A2 are contiguous. The memory addresses of B1 and B2 are contiguous. The memory addresses of A2 and B1 have a big offset which is half size of the whole render target.

The size of the largest tile is 128x128. The alignment for the 128x128 tiles is always from left to right, from bottom to top. In the lower half part of the figure, a complete render buffer with resolution 640x512 is demonstrated.

4.9 Fast Clear Technology

From the last section, we know that render target is organised in tiles for efficiency and optimization. Fast clear technology is exactly an example that benefits from using tiles in the render target. Vivante exploits a fast-clear technique which does not actually operate on the render target when Clear command is called. The tile in the render target that should be cleared will be marked in the tile status buffer, the render target is not touched. The real clear is performed only when the tiles of the render target are going to be rendered or when the render target is going to be displayed on the screen (in this case the tile in the color buffer are cleared). Marking the tiles is much faster compared to real clearing since only a very small portion of memories need to be accessed. The execution time of Clear is greatly reduced.

4.10 GPU Context Switch

When a new process (OpenGL ES 2.0 application) is created, it will first send a notification to the kernel space driver through ioclt() with the command code gcvHAL_ATTACH to let the kernel driver knows that a new process is attached. The kernel space driver will then initialize a context buffer for that process. When multiples processes are running concurrently on the GPU, before the GPU switch from the command group of one process to another process, many GPU states need to be changed. The states changes are done by the submitting the context buffer. The context buffer keeps all GPU states that should be preserved when a process is switched.

Every time when an process submits a command group to the GPU, the changes of the GPU states caused by the submitted command group will be maintained by the user space driver with the structure gcsSTATE_DELTA. This structure contains an array which is called recordArray to store the states changes. Each process attached to the GPU driver has its own recordArray. Once the driver switch from one process to a next process, the kernel space driver will update

the context buffer of the next process by merging the context buffer of that process and the changed GPU states in recordArray. Then the updated context buffer is dispatched to the GPU to load the GPU states required by the next process.

4.11 Profiling Registers

Vivante GPUs are equipped with a hardware profiler. GPU statistics such as the number of primitives after primitive assembly and the fragment number generated by the rasterizor, are collected by the profiler. The profiler is a valuable resource for analysing the GPU performance since this is almost the only way to directly extract GPU statistics from the GPU hardware. The profiling results are stored in some special registers which are accessible by the kernel space driver. By polling the value inside the profiling register, the data in the register is updated gradually during execution.

4.12 Summary

In this chapter, the system model of the GPU driver is presented. In general, the GPU driver can be divided into two parts: user space driver and kernel space driver, the communication between the user space driver and kernel space driver is established with an interface with system call ioctl(). The GPU command execution flow and the related concepts are explained in section 4.3 and 4.4, including how the command groups are committed to the kernel space, how they are submitted to the GPU and they are fetched by the GPU. The synchronization of the GPU including signals and interrupts are presented in section 4.5 and section 4.6. Besides, another features of the GPU drivers are discussed from section 4.7 to section 4.10.

5 Concept

From chapter 1 we know that a possible solution for scheduling real-time GPU applications is to use non-preemptive deadline-driven scheduling. This requires to predict the execution time of every dispatched GPU command group. In order to accurately predict the command group execution time, a precise measurement system is required. In this chapter, we explain our concepts of the measurement and prediction system. In the first section, the framework of the measurement and prediction system is presented. Then we explain how and when we take the timestamps and the profiler data in the measurement system in section 5.2. After that, we describe how to map the OpenGL ES 2.0 commands to the GPU command groups correctly in section 5.3. In section 5.4, how to find the correct interrupts generated by submitting command groups is discussed. In section 5.5, important OpenGL ES 2.0 rendering commands are analysed, modelled and predicted.

5.1 Measurement and Prediction Framework

In this section, we introduce the framework of our measurement and prediction system. Figure 5.1 demonstrates the system model for an OpenGL ES 2.0 application including our measurement and prediction system (the green modules). Compared to the system model illustrated in section 4.1, we added four components. They are OpenGL ES command monitor, execution time monitor, profiler and command predictor. The command monitor and command predictor are above the user space driver. The execution time monitor and the profiler reside in the same layer of the kernel driver.

The command monitor intercepts every OpenGL ES command that is called by the application and maps the OpenGL ES Commands to the correlated GPU command group/groups. That means the command monitor is aware of every OpenGL ES command inside every GPU command group. Mapping is needed because our prediction predicts the execution time of every time-consuming OpenGL ES commands instead of directly predicting the execution time of each GPU command group. Additionally, The command monitor also intercepts the ioctl function that commits the command group from the user space to the kernel space to collect statistics from the intercepted command group, such as the length, the hex content, the signal number(can be more than one signal number) attached to the command group. The statistics is used to analyse the driver model and improve the prediction model.

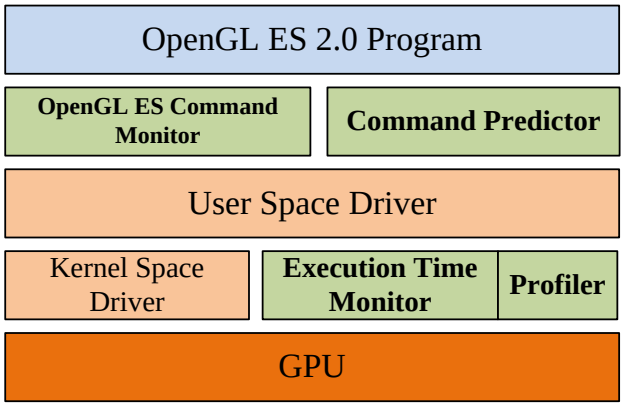

Figure 5.1: Measurement Framework

The execution time monitor and the profiler are integrated in the kernel space driver. The execution time monitor measures the execution time of each command group for different applications separately based on the process ID. The profiler works together with the execution monitor to collect statistics and information of each command group such as the number of fragments and the number of primitives.

The predictor predicts the execution time of every GPU command group before the command group is dispatched to the GPU using statistics collected by the command monitor, the execution time monitor, the profiler and some calibrated parameters.

5.2 Taking Starting and Finishing timestamps and Profiler Data

An accurate measurement system is fundamental for prediction. Because without an accurate measurement system, we cannot measure the error in our prediction model and improve the prediction model. Besides, the measurement system is also required to calibrate the parameters in the prediction model. The most critical component of the measurement system is the execution time monitor. In order to measure the command group execution time accurately by the execution time monitor, timestamps must be taken as close as possible to the real GPU starting execution time and finishing execution time. Here we explain how we chose the place to take the timestamps in order to optimize our measurement result.

Starting Timestamps: From section 4.3.5 we know that before the command group is fetched by the GPU, the fetch engine is waiting in a WAIT/LINK loop. Once the WAIT in the WAIT/LINK loop is modified to a LINK pointing to a new command group, the fetch engine will jump out

of the loop and start to fetch the new command group. This moment is the time when the command group is submitted to the GPU.

If the GPU is idle, the command group submitting time is the most suitable time to take the starting timestamps because at that moment, the GPU takes over the control of the task and can immediately start the execution. However, if the GPU is busy, after WAIT is converted to LINK, the command group has to wait in the command queue until other command groups are finished to start the execution. If we still take the timestamps in the same way, the time we take will be obviously too early. In that case, we use the same solution as described in [33], we compare the command submitting time of the current command group to the finishing time of the previous command group. If the submitting time of the current command group is earlier than the finishing time of the previous command group. We assume the command group starts to execute once the previous command group is finished and uses the finishing time of the previous command group as the starting execution time.

Finishing Timestamps: After submitting the command group, the kernel itself will push an EVENT command into the command queue, when the EVENT command is executed by the GPU, the GPU will mark a bit in the event register and trigger an interrupt (section 4.5). Once the interrupt arrives at the CPU, we immediately take a timestamp at the beginning of the interrupt service routine as the command finishing time since we know the EVENT command is executed only when all the previous command groups are finished. What needs to be noted is that some command groups are extremely short and consume almost no execution time. The GPU is possible to finish these command groups during the time of the interrupt service routine before the event register is read. Hence we also check the value inside the event register after we take the finishing timestamps. If more than one bit is set inside the event register, we assign the same finishing timestamp to all related command groups. For instance, if three bits are set in the event register, we then know three command groups are finished and we assign the same finishing time to these three command groups. In this case, during the measurement, it is possible to get command groups with zero execution time when the current command group's starting time and finishing time both equal to the finishing time of the last command group. However it is reasonable since we know the execution time for these command groups is extremely fast.

Profiler Data: From section 4.11, we know that Vivante GPU collects GPU statistics such as the valid fragment number, the primitives number after culling, and stores the statistics in certain profiling registers. Since the statistics are accumulated, in our profiler, we read the profiling registers in every GPU interrupt service routine and calculate the difference between every two interrupts. The difference is used as the profiling data for the current command group. We notice that the statistics we take might not be perfectly correct because the profiling data is always updating even during the interrupt service routine since the GPU is working in parallel with the CPU. But based on our observation, the statistics are updated gradually and will not change dramatically in short time. Hence the statistics we get are still quite close to the correct value in most of the cases.

5.3 Mapping OpenGL ES Commands to Command Groups

Based on the method described in last section, we can already measure the execution time of each command group in the kernel space driver. However, it is very difficult to predict the execution time of the command groups directly in kernel space. On one hand, the commands inside the command group are hardware-specific, detailed information of many GPU hardware commands are not available. On the other hand, GPU commands are binary codes. They have to be interpreted in order to be predicted. This obviously introduces much additional overheads.

A better solution to predict the execution time of the command group is to estimate the execution time of each individual OpenGL ES command inside the command group and then summed up the execution time as the command group execution time. This is feasible because all the OpenGL ES commands are known to the public and can be derived from intercepting the OpenGL ES API calls.

One obstacle of the solution is that due to different kinds of reasons(we will mention that in this section), a single OpenGL ES command can be split into two parts and committed by two command groups. In this situation we need to map the execution time of the split OpenGL ES command correctly to these two command groups. In order to do that, we need to first find our when a split occurs. We did many experiments and analysed the test results in order to find out all the split situations. The situations are enumerated below:

Situation 1. When the command buffer is full. During the test, we found out that, periodically, some command groups will be split into two parts in the user space and committed to the kernel space via two separate system calls. The split happens when a new block of command buffer is in use (More details regarding the split have already been mentioned in section 4.3). We notice that OpenGL ES command is possible to be split into two part. This is a critical problem for prediction because commands such as Draw, Clear(when fast clear is not applied) have a significant effect on the execution time of the total command group. We looked into the hex content of Draw and Clear commands and found that most contents of the these commands are doing settings and configurations. The real execution is in the last few bytes of the command. Based on our experience, they never split. Hence, we always map these commands to the second command group when a split occurs and assume these commands do not increase the execution time of the first command group.

Situation 2. When the vertex buffer is full and Draw(glDrawArrays, glDrawElements) is called. For each OpenGL ES application, besides the contiguous memory allocated at initialization time, Vivante GPU driver also allocates some vertex buffer (video memory) for storing the attributes of the vertex such as the vertex coordinates and the texture coordinates. As we have already mentioned in section 4.7, in the beginning of each application, five vertex buffers each with a size of 1 MB will be allocated. These vertex buffers are never freed till the end of the application. Once Draw is called and the vertex buffer object (VBO) is not applied, the

vertex attributes related to the Draw will be pushed into the vertex buffer. Before that, the user space driver will check if the current vertex buffer has enough space to fill the vertex attributes. Once the user space driver detects that there's not enough space left, the driver will load the vertex attributes into a new vertex buffer and immediately commit the current content in the command group to the kernel space and inform the GPU about the address of the new vertex buffer. The Draw command is therefore split into two parts. However, since we know that the real execution of the Draw is not contained in the first command group, we always map the Draw command to the second command group. Because the default size of each vertex buffer is only 1 MB, if the size of the vertex attributes is larger than 1 MB, the data cannot be filled into the vertex buffer anyway. In this case, a new vertex buffer with suitable size will be allocated before every Draw to fill all the vertex data. In this case, the command group will not split. This vertex buffer is instantly reclaimed by the driver after the Draw to save system memory. Additional, for Draw with VBO, split never occurs, because for VBO, the data of the vertex attributes are loaded only once and then stored persistently in the same memory space.

5.4 EVENT Submission

From section 4.5, we know that an EVENT command will always be submitted to the GPU after a command group is submitted. Beside that, EVENT command will also be submitted in some other situations and will also trigger interrupts. These interrupts are mixed with the interrupts caused by the submission of the command group. In order to calculate the execution time of a command group, we have to map the correct finishing timestamps (interrupts) to our command groups. That means, we need to distinguish the interrupts caused by command group submission from the other interrupts. Therefore, we have to find out all the situations where an EVENT command is submitted to the GPU to avoid interrupt mismapping. These situations are enumerated below:

Situation 1. The EVENT command is submitted after a command group is submitted. The interrupt generated by this EVENT is used to take the finishing timestamp of the command group.

Situation 2. System call with command code "gcvHAL_EVENT_COMMIT". Another approach to submit an EVENT to the GPU is to call ioctl with command code "gcvHAL_EVENT_COMMIT" in user space. This command code is normally used for synchronization. When the command code is passed to the kernel, the kernel will submit a command group with only an EVENT command to the GPU. Because the EVENT can be used for any purposes and only occurs rarely, it is difficult for us to cover all the situations where an EVENT is committed without the source code of the user space driver. However, since the EVENT from user space is always submitted with command code "gcvHAL_EVENT_COMMIT", we are aware of the submission and have no problem to distinguish them from the normal command group submission.

Since the execution time of EVENT command is extremely fast and is always constant for different applications, there's no need to take the execution time of EVENT command into consideration. We trace these kind of submissions only in kernel space and make the submissions transparent to the user space

Situation 3. When new command queue is in use. When a new command queue is in use, the kernel itself will submit an EVENT. This EVENT will finally trigger an interrupt to notify the kernel to release the lock on the last command queue. Hence, when a new command queue is in use during the submission a command group, two interrupts will be generated. If the new queue is in use before submitting the command group, then the second interrupt we get is the interrupt for the command group. If the new queue is in use after the submission of the command group, then we use the first interrupt as the interrupt for the command group. The execution time of the new queue EVENT is transparent to the user space just like the second situation.

5.5 Anatomy and Prediction of Important OpenGL ES Commands

After mapping OpenGL ES commands to the corresponding command groups, the next step is to analyse and predict the execution time of the OpenGL ES 2.0 commands. Although there are quite a large number of OpenGL ES 2.0 commands defined in the specification, most of the commands are either executed on the CPU or used only for configuration (like loading some values into some GPU state registers) which consumes almost no GPU time. Hence we only need to analyse and predict the commands that have measurable influence on the GPU execution time.

To the best of our current knowledge and experience, in general, the following commands are time-consuming on the GPU execution time during rendering: Flush, Finish, Clear and Swap. However, since OpenGL ES 2.0 is only a standard specification, the implementation varies from GPU to GPU. Even for the same rendering command, the effect can be distinct on different GPUs. For vivante GPU GC 2000,the command load the texture(glFramebufferTexture2D) will also significantly increase the GPU execution time. This command will also explained below. Commands only appear at the initialization time such as loading the shader code or generating the mipmaps are not taken into account since they are not used during rendering.

In order to predict the execution time of all time related OpenGL ES commands, we have to understand how they are implemented and which factors will actually affect the execution of these commands. Next, we describe our analysis of these time-consuming commands and explain our prediction models for them. As a reminder, all the discussions below are based on the tests on the Vivante GPU, for other GPUs, the results can be different.

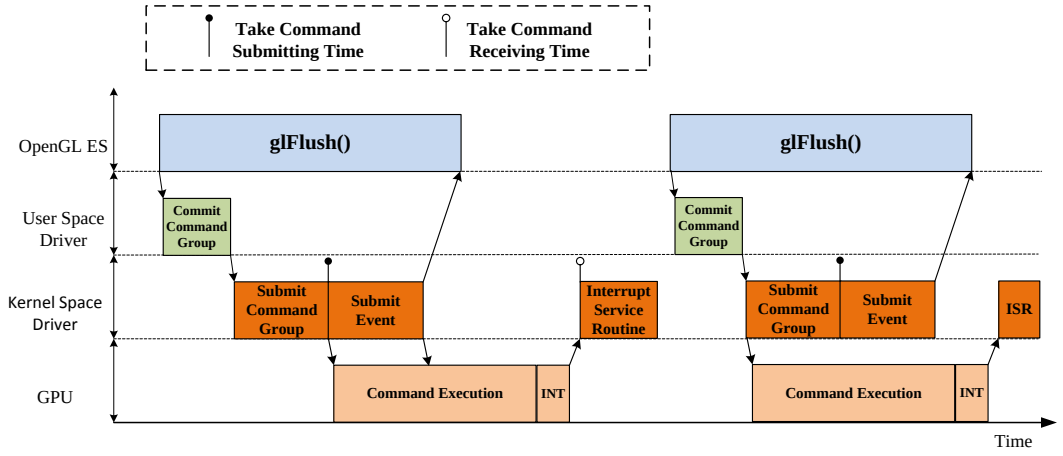

Figure 5.2: Execution Flow of glFlush

According to the way of prediction, we divide the rendering commands into three groups: 1. constant execution time; 2. buffer size based execution time. 3. vertex number and fragment number based execution time.

1.Command with Constant Execution Time: the commands in this group have constant execution time regardless of program's content and context.

Flush(glFlush): The first OpenGL ES command we want to discuss is Flush (used by methods such as glFlush,glFinish and eglMakeCurrent). Flush, as its name indicates, flushes the current commands inside the command buffer to the GPU. Figure 5.2 shows a simple application doing nothing except calling glFlush command. As can be seen from the figure, glFlush commits a short command group to the kernel from the user space. The content of the command group is a 16 bytes long GPU command to flush the GPU cache which is very fast compared to other rendering commands. After the kernel submits the command group and the EVENT command to the GPU, it will immediately return to the user space to continue the application.

Prediction of Flush: Since the time to execute Flush is always the same, all we need to do is to measure the average execution time of Flush.

Finish(glFinish): As is depicted in figure 5.3, glFinish commits two command groups to the kernel. The first command group submits all current buffered GPU commands to the GPU similar to glFlush. If the application does nothing except calls the glFinish command, then the content of the first command group is exactly the same as glFlush, submitting a 16 bytes long command to flush the GPU cache, an EVENT command is submitted separated after the submission of the first command group in user space. The second command group is always the same, independent to the content of the applications, it again flushes the cache of the GPU.

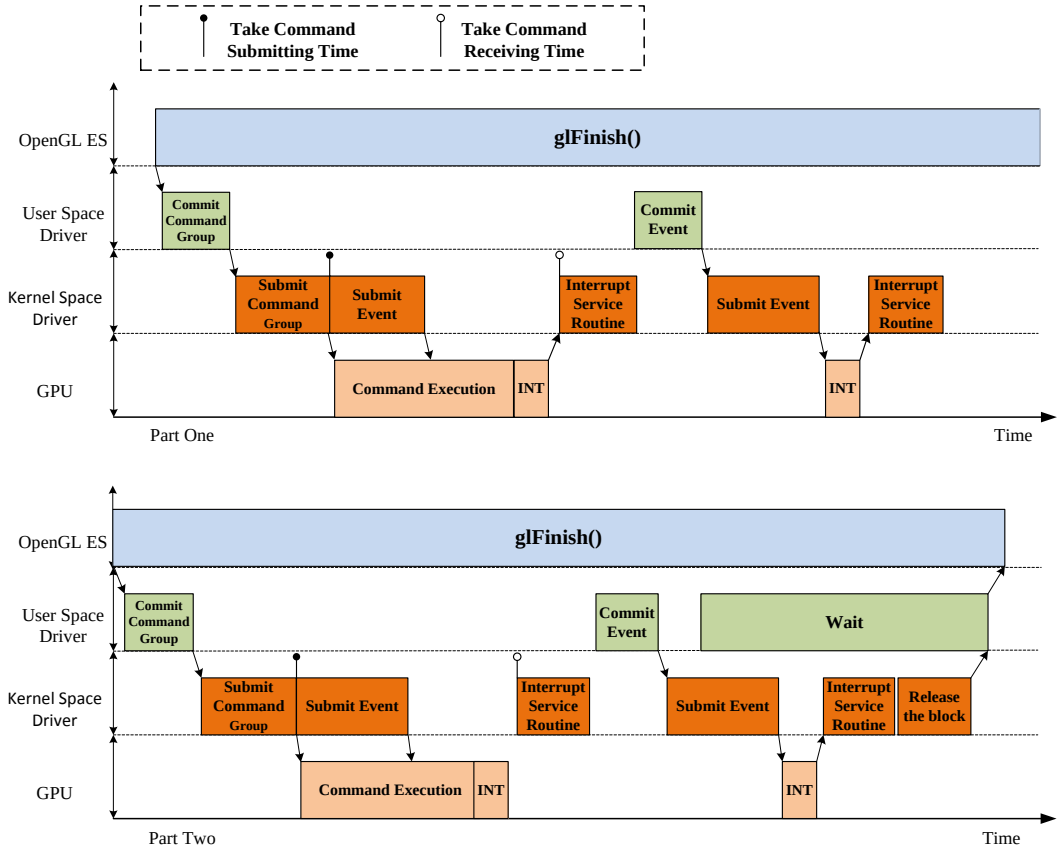

Figure 5.3: Execution Flow of glFinish

Hence the execution time of the second command group is always constant. After committing the second command group, one more EVENT is committed by the user space. The application is then blocked and waits for a specified signal from the kernel space. The application will be released by the kernel when it detects that the GPU has finished all the submitted commands. As for prediction, the execution time of each command group in glFinish is exactly the same as glFlush since the same content is committed.

Prediction of Finish: Similar as the prediction of Flush.

Clear(glClear): Clear is used to set portions of every pixel in a particular buffer to the same value[29]. For many GPUs, the Clear command will directly set every pixel in the render target to a specified color which involves a large amount of memory write operations[33].

For the Vivante GPU, the implementation of Clear is quite different. As we have mentioned in section 4.9, Vivante exploits a fast-clear technique, which does not actually operate on the render target when Clear is called. The execution of Clear can be divided into three steps:

Step 1. When glClear is called, the tile of the render target that should be cleared are marked in the tile status buffer. The rendering target is not touched.

Step 2. When the Draw command is called, tiles in the render target will be loaded into the GPU cache. The GPU will then check if the tile needs to be cleared based on the status inside the tile status buffer. If clear is marked, the tile in the cache will then be cleared. The rest of the tiles which are not touched by the rasterizer will still not be cleared. This approach works well since the render target is only a back buffer and will not be displayed on the screen directly.

Step 3. When Swap is called, the tiles in the render target that are rendered will be copied to the color buffer. The color buffer will clear the rest of the tiles which are not touched before the whole image is displayed on the screen.

What needs to be noted is that the fast clear technology is only used by the default render target. OpenGL ES 2.0 also supports framebuffer object to render on a specified non-default render target(or we can called it off-screen framebuffer) such as a texture buffer. In this case, the fast clear technology is not applied, the whole render target will be cleared immediately when Clear is executed by the GPU.

Prediction of Clear: The Prediction of the Clear command needs to be divided into two parts:

Situation 1: For the Clear of the default render target, the execution time is short and constant. Experiments show that it even has no impact on the execution time of Draw and Swap. This is reasonable, for Draw, if a tile is not cleared first, the GPU has to load the data from the rendering target to the cache which might take even longer time. For Swap, the color buffer will be cleared to black when Clear is not called which takes exactly the same amount of execution time.

Situation 2: For the Clear of the non-default rendering target, the execution time depends on the size of the non-default render target. This kind of Clear can be put into the next group: commands with buffer size based execution time.

2.Command with Buffer Size based Execution Time

Swap(eglSwapBuffers): Command eglSwapBuffers commits two command groups to the kernel. The first command group is same as glFlush which flushes the current commands in the command buffer to the GPU. The second command group copies the data from the render target to the front color buffer. To avoid unnecessary memory access, only the 4x4 pixel tiles that are updated (rendered) will be copied. The rest of the tiles in the color buffer are simply cleared.

Prediction of Swap: The prediction of the Swap command can be divided into two parts: memory reading time and memory writing time. Memory reading is involved in the copy operation, the execution time depends on the updated tile number. Memory writing is required by both Copy and Clear operations, therefore the execution time is always proportional to the size of the render target.

The prediction model of the Swap command can be presented by the following formula, where N_w stands for the total tile number in the default render target and N_r stands for the rendered tile number. $c_{writebyte}$ and $c_{readbyte}$ are the time to write a byte into the memory and the time to read a byte from the memory. They can be calculated through calibration. $c_{bytespertile}$ is the size of a tile. The rendered tile number can is estimated as one sixteenth of the total valid fragment number generated in each frame.

$$T_{swap} = c_{writebyte} \times c_{bytespertile} \times N_w + c_{readbyte} \times c_{bytespertile} \times N_r \quad \textbf{(5.1)}$$

Clear: To clear the non-default render target, the execution time depends on the size of the rendering target buffer. Generally, the size of the non-default render target is larger than the size we specified in the application. Experiments show that the Clear command will clear the whole render target even the part that is not used by the application. Hence the size of the real render target should be used for prediction instead of the size we specified in the application to predict the Clear command. The total execution time can be presented by the following formula. Parameter N in the formula stands for the total bytes of the non-default render target.

Prediction of Clear:

$$T_{clear} = c_{writebyte} \times N \quad \textbf{(5.2)}$$

Texture Loading(glFrameBufferTexture2D): Beside the commands above we noticed during the test that if an application switches from the default render target to some non-default render target(off-screen framebuffer), the execution time is significantly increased. This is due to the copy between the off-screen render target and the color/depth buffer which is done by the resolver(a GPU component). For instance, if an application wants to render on the texture(none-default render target), it will request the GPU driver to allocate some memory for the texture. The memory is only temporarily owned by the application and will be reclaimed after the rendering. This leads to a result: every time before the rendering, the data of the color/depth buffer needs to be loaded into the texture buffer(when API glFramebufferTexture2D is called). Such read operations consume a lot of GPU execution time.

Prediction of Texture Loading: The prediction of Texture-Loading (glFrameBufferTexture2D) is similar to the prediction of Clear since the principle is more or less the same. The execution

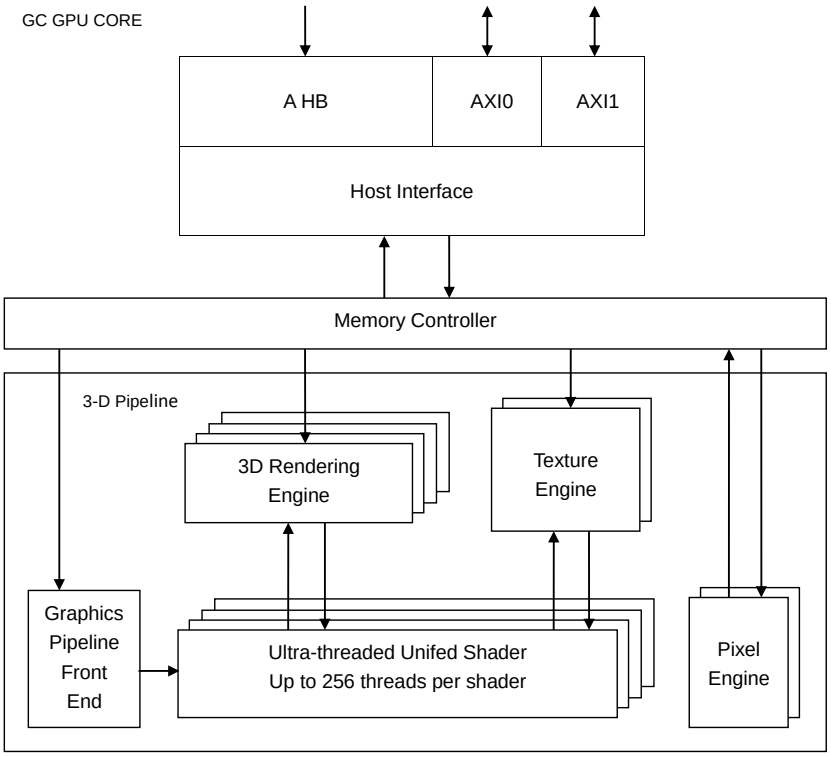

Figure 5.4: Vivante 3D GPU architecuture [20]

time depends on the size of the non-default render target and the memory read, write speed. However, during the experiments, we found out that memory read/write speed is faster when the size of the render target is below a threshold. Hence we use a piecewise linear formula to predict the execution time. In the following formula, c_1 and c_2 stand for the speed to load a byte into the render target when the size of the render target is below or above a certain range (N_s).

$$T_{glFrameBufferTexture2D} = \begin{cases} c_1 \times N, & if\ N <= N_s \\ c_1 \times N_s + c_2 \times (N - N_s), & if\ N > N_s \end{cases} \quad \textbf{(5.3)}$$

3.Command with Vertex Number and Fragment Number based Execution Time

Draw(glDrawArrays, glDrawElements): Draw is the most sophisticated and time consuming command. The execution of Draw needs the cooperation of different GPU modules and goes through every stage in the GPU pipeline. Now we explain the Draw commamnd is executed by the vivante 3D GPU (figure 5.4).

1. The command, uniforms, vertices' attributes and other related data needed by the vertex shader are fetched by the graphic pipeline front end(fetch engine). The execution time of this step depends on the vertex number and the size of the attributes.

2. Vertex shader (the unified shader threads) performs geometry transformation and lighting computation on each vertex. The execution time is related to the number of vertices and the complexity of the shader code.

3. The result of the vertex shader is sent to the rasterizer(3D rendering engine) where primitives are converted into fragments. The color and texture coordinates of each pixel are also calculated by the rasterizer. The execution time depends on the number of fragments.

4. After rasterization, the data of the fragments is sent back to the fragment shader (the unified shader threads). The fragment shader computes the color for every pixel. The texture engine retrieves texture information from memory when texture is requested by the fragment shader. The execution time depends on the number of fragments, the shader code and the size of the texture.

5. After the execution of the fragment shader, the pixel engine does further alpha blending and visible surface determination and writes the result into the memory. The execution time depends on the fragment number.

Prediction of Draw: In an optimized pipeline, the steps mentioned above should be able to execute concurrently in the GPU pipeline. However, this does not mean that the execution time of the Draw command is only defined by a single bottleneck. The reasons are as follows:

1.The execution time may depend on a single bottleneck at a moment. However, the bottleneck can change several times even during the rendering of a single object[37].

2. Nowadays, in order to support general computation and reduce the hardware complexity, vertex shader and fragment shader are often implemented on the same hardware as a unified shader. The unified shader comprise two stages of the OpenGL ES 2.0 pipelines. If more GPU threads in the unified shader are occupied by the vertex shader, then the processing speed of the fragment shader will be reduced and vice versa.

From the analysis of the Draw command, it can see that, if considering a single application with the same shader code and texture quality, the rendering time mainly depends on the number of vertices and the number of fragments. Hence the execution time of Draw may depend on 1. only the vertex number, 2. only the fragment number and 3. a combination of both vertex number and fragment number. Experiments show that a prediction model considering both vertex number and fragment number achieves the best accuracy. As a result we use the following formula to present the prediction of the Draw command.

$$\begin{aligned} T_{draw} &= T_{vs} + T_{fs} \\ &= c_v \times N_v + c_f \times N_f \end{aligned} \quad \textbf{(5.4)}$$

In the equation, the execution time of Draw consists of two parts: T_{vs} and T_{fs}. T_{vs} stands for the time the unified shader spends on vertex processing which linearly depends on the number of vertices N_v. T_{fs} stands for the time the unified shader takes for fragment processing, it is proportional to the valid fragments number N_f. A valid fragment is a fragment that passed the early-z test(see section 3.3). Fragments which are discarded by the rasterizer are not taken into account since they are not executed by the fragment shader. c_v and c_f are the per-vertex and per-fragment processing speed for specified shader and texture which are calculated through calibration. Next we explain how we calibrate c_v and c_f and how we get N_v and N_f.

Because the Draw command is intercepted by the OpenGL ES command monitor, N_v can be get by summing up the "count" parameter in every Draw command. To calibrate c_v for every different application, the glViewport function in the OpenGL ES application is also intercepted. Parameter height and width in glViewport (see subsection 3.1.3) are set to minimum so that no fragment will be generated after rasterization. This introduces no difference to the vertex shader since what we have changed is only the width and height value in the equation of the viewport transformation. The modified application is kept running for several frames. The total Draw execution time and number of vertices (this is known by intercepting the Draw commands) are then used to calculate the average per-vertex execution time c_v.

To calibrate c_f, we run the original OpenGL ES application without any modification for several frames. Since per-vertex processing speed is already calibrated, the total vertex-processing can be calculated. Then the total execution time for processing fragments can be calculated by subtracting the total vertex-processing time from the total execution time. By using the profiler(section 5.2), the fragment number of every command group can be measured. Hence we also have the total fragments number by summing up the fragments number in every command group. The per-fragment processing speed can then be calibrated at run time.

To get the number of fragments N_f of the current command group is more challenging. Unlike the vertex number that can be directly extracted from the Draw command, the fragment number depends the output of the rasterizer and needs to be predicted. We notice that in order to present fluent animation, certain amount of frame rate is required by the graphic applications. The difference of the image between the adjacent frames is often quite small, so does the number of the fragments. Based on this assumption, a recent-history based approach is developed to predict the fragment number of in a single frame.

In our approach, the fragment number in the last frame(or the most recent available frame, same for the following sentences) is used as the fragments number in the next frame. The fragments number in the last frame can be obtained by using the profiler. In order to get

the fragments number of each command group in a single frame, the recent-history based approach can be further divided into two types.

1. Fragments Number Prediction with Rencent-Vertex-Fragment Ratio

In the above paragraph, we assume the fragment number between two frames are similar to each other. in this approach, we further assume that the vertex-fragment ratios between two frames are similar to each other and predict the fragments number of the next command group by multiplying the vertex-fragment ratio of the recent available frame and the vertex number in the next command group.

2. Fragment Number Prediction with Same Command Group Sequence Number

We are aware that, for the approach we mentioned above, the success of the fragment number prediction requires a similar vertex-fragment ratio in every Draw command. Hence we also proposed an additional method to predict the fragment number when the vertex-fragment ratio in every Draw command is very different. In this approach, after each frame is finished, the fragments number of the executed Draw-contained command groups will be sequentially stored in an array which is called the recent fragments number array(the original data in the array will be cleared). A counter is used in the predictor to count the number of the Draw-contained command groups in a single frame. When the Nth Draw-contained command group comes, the predictor finds the Nth element in the recent fragments number array and uses it as the fragment number of the Nth Draw-contained command group. This approach is suitable for applications that use the same program repetitively and generate the same amount of command groups in every frame.

5.6 Summary

In this chapter, the framework of our measurement and prediction system is demonstrated. Then how to correctly mapping the OpenGL ES commands to the GPU command groups is explained. After that, the rendering commands are analysed and divided into three types. For each type of commands, prediction models are proposed.

6 Implementation

In this chapter, the implementation of the concepts of measurement and prediction is described. In the first section, the components in the measurement and prediction system are explained. In section 6.2, the algorithms we used to take timestamps and to calculate the command group execution time are present. In section 6.3 the algorithm to predict fragments number of the command group is presented. Finally, the modifications in the kernel drivers are enumerated in section 6.4.

The software requirements to implement our system are as follows:

- Linux Kernel version 3.035-r37.14
- Vivante GPU Graphic driver imported from Freescale BSP release 4.1.0.

6.1 Components of the Measurement and Prediction System

Figure 6.1 illustrates the components in of our measurement and prediction system. The components are marked in red and green. As we have already mentioned in chapter 5, our system is mainly comprised of three parts: the OpenGL command monitor, the command predictor and the execution time monitor(including the profiler). The command monitor and the command predictor reside in the interception & predicton layer between the OpenGL ES layer and the userspace driver. The execution time monitor is integrated in the kernel space driver. The user space driver is not touched. All the important data structure that is used by these components are stored in file gc_hal_kernel_ shared_mem.h and the file predict_data.h. The explaination of these data structures can be found in appendix A.1.

6.1.1 Interception & Prediction Layer:

The OpenGL command monitor and the predictor monitor reside in the interception & prediction layer. The command monitor intercepts every OpenGL ES APIs, EGL APIs and all also the ioctl system calls before they are executed. Every time when an OpenGL ES API or an EGL API is called by the application, related information is collected (such as the name of the API). When the ioctl system call comes, the command monitor checks the purpose of the system call. If the system call is used by the GPU user space driver to commit the current command group to

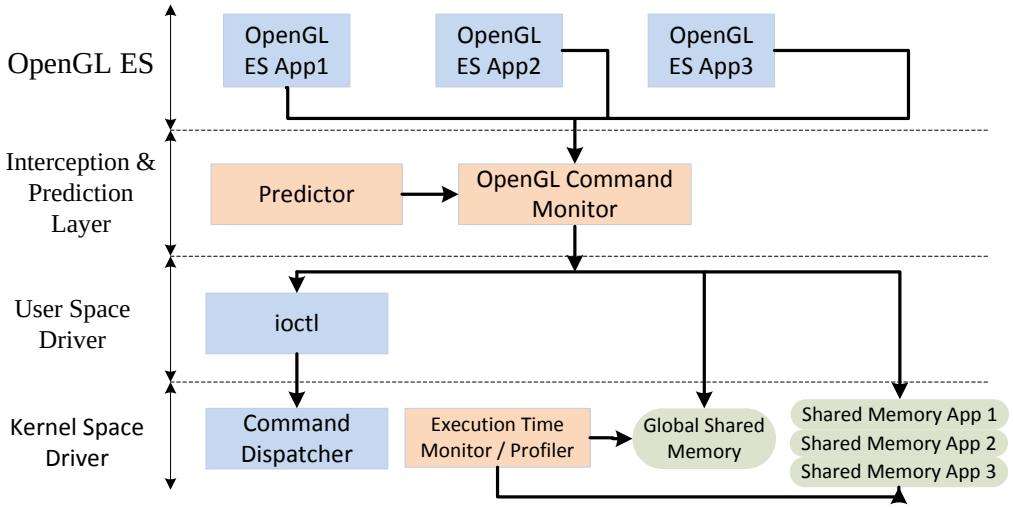

Figure 6.1: Measurement and Prediction System

the kernel space driver, then the command monitor will push all the information of the current command group into the application shared memory and then continue the real execution of the ioctl. Every time when a new OpenGL ES application(with the interception & prediction layer) is created. The interception & prediction layer will request the kernel space driver to allocate a block of application shared memory by using ioctl() system call. The application shared memory is then mapped to the user space using MMAP so that it is accessible by both kernel space driver and the interception & prediction layer. Once the application finishes, the exit handler of the main function will make a system call to inform the kernel driver to free the application shared memory. If the application is aborted, the kernel space itself will instantly free the memory to avoid the memory leaks.

The predictor traces all the time-consuming commands that have been discussed in section 5.5 and the parameters of the commands by intercepting related APIs. Every time a time-consuming command is called, the predictor will calculate the execution time of that command and add that to the total execution time of the current command group. As an exception, the execution time of Draw command (glDrawArrays or glDrawElements) is immediately predicted when Draw is called. This is because the prediction of Draw uses the total number of vertices and fragments, hence there's no need to predict every Draw command separately. Since our prediction method also exploits the data of the recent history, the command predictor updates the history data after every frame is finished. Finally, before the command group is committed, the predictor predicts the execution time of the Draw command and calculate the total predicted execution time of the current command group.

6.1.2 Execution Time Monitor:

The execution time monitor is integrated in the kernel space driver. It takes timestamps, profiles data and calculates the execution time of every GPU command group. After that, it writes the result to different application (process) shared memories based on the process ID. Besides, it allocates a large block of memory, the global shared memory, when the GPU driver module is loaded into the kernel. The global shared memory records the information of every command group that goes to the GPU regardless of the owner (application) of the command group, so that a prospective GPU scheduler can have a general view of what is happening on the GPU. The recorded information in the global shared memory can be queried in user space with an external application which has also been developed. This external application also supports other operations such as querying the current index of the command group, clearing the global shared memory and so on to make the measurement system more flexible. The global shared memory is implemented as a ring buffer to avoid the memory overflow.

6.2 Taking Timestamps and Calculating Execution Time

The submitting timestamps and the receiving timestamps are taken from different threads. They are mapped to each other via two array index numbers. Algorithm 6.1 shows the flow to take the starting timestamp of each command group. An index number is used to keep the order of the submitting timestamps. It increases every time when an EVENT command is submitted to the GPU. The timestamp of the EVENT submitting time will also be taken after the EVENT is submitted. The number of submitted EVENT commands after each command group depends on the number of the current event queues. An Event is submitted for each event queue (see the while loop). If no command group is submitted before the EVENT command, the command group submitting time is set to zero(default value).

The other index is used to keep the sequence of the receiving timestamps. It increases every time when an interrupt comes in order to map the finishing timestamps to the correct starting timestamps. Algorithm 6.2 shows the flow to take the finishing timestamps. After taking the finishing timestamps and calculating the command group execution time. The program will check whether the application uses application shared memory (only the applications start along with the interception & prediction layer use application shared memory). If it is true, then the data of the command group will be copied from the global shared memory to the application shared memory. Command groups contain only EVENT command will not be copied.

Algorithmus 6.1 Taking the starting timestamps

Command group/event from user space:
$lock\_the\_command\_queue()$
if $command\_group\_is\_committed$ **then**
    $cmd\_submitting\_time[i] \leftarrow current\_cpu\_time$
end if
$change\_the\_scheduler\_policy\_to\_fifo()$

$submit\_the\_command\_group()$

while $event\_queue \neq null$ **do**
    get_event_number()
    $cmd\_submitting\_pid[i] \leftarrow current\_process\_id$
    $submit\_the\_event\_command()$"
    $event\_submitting\_time[i] \leftarrow current\_cpu\_time$
    $i \leftarrow i + 1$
end while
$change\_the\_scheduler\_policy\_to\_normal()$
$release\_the\_command\_queue()$

Algorithmus 6.2 Taking the finishing timestamps and calculating the execution time

When GPU interrupt comes:
$finishing\_time \leftarrow current\_cpu\_time$
$event\_number \leftarrow getEventNumber()$
$first\_index \leftarrow j$
$n \leftarrow 0$
for $n \leqslant event\_number - 1$ **do**
    $cmd\_finishing\_time[j] \leftarrow current\_cpu\_time$
    $j \leftarrow j + 1$
    $n + 1$
end for

$n \leftarrow first\_index$
for $n \leqslant first\_index + event\_number - 1$ **do**
    if $cmd\_finishing\_time[n] > cmd\_finishing\_time[n-1]$ **then**
        $cmd\_execution\_time[n] \leftarrow cmd\_finishing\_time[n] - cmd\_submitting\_time[n]$
    else
        $cmd\_execution\_time[n] \leftarrow cmd\_finishing\_time[n] - cmd\_finishing\_time[n-1]$
    end if
    $check\_process\_shared\_memory\_list()$
    if $in\_process\_list$ **then**
        if $cmd\_submitting\_time[n] \neq 0$ **then**
            $copy\_data\_from\_global\_shared\_memory\_to\_process\_shared\_memory()$
        end if
    end if
end for

6.3 Fragment Number Prediction

The concept of predicting the fragments number of a command group was described in Section 5.5. Two approaches are proposed: they are prediction based on rencent-vertex-fragment ratio and prediction based on the same command group sequence number. Algorithm 6.3 depicts the implementation of the first approach: fragment number prediction based on the recent-fragment-vertex ratio. Every time when eglSwapBuffers is called, the program checks if the measurement of the last frame is finished by reading the the finishing timestamp of the last command group in the last frame. If value of the finishing timestamp is not zero, that means the last frame is finished and the measurement is done. Then the fragment-vertex ratio of the last frame can be calculated. If last frame is not finished yet, the vertex-fragment ratio of the frame before last frame will be used for the prediction. When a command group is submitted,

the fragments number of the command group is predicted by multiplying the fragment-vertex ratio and the vertex number of the current command group. After many tests, we find that it is always possible to get the fragment number of the frame before the last frame. The implementation of fragment prediction based on the same command group sequence number is similar to the first approach, the difference is already discussed in Section 5.5.

Algorithmus 6.3 Fragment Number Prediction

When eglSwapBuffers is called:
if $last\_frame\_is\_finished$ **then**
  while $command\_group\_in\_last\_frame \neq null$ **do**
    $totoal\_fragment\_number \leftarrow$
    $fragment\_number\_command\_group + totoal\_fragment\_number$
  end while
else
  if $frame\_before\_last\_frame\_is\_finished$ **then**
    while $command\_group\_in\_last\_frame \neq null$ **do**
      $totoal\_fragment\_number \leftarrow$
      $fragment\_number\_command\_group + totoal\_fragment\_number$
    end while
  else
    $totoal\_fragment\_number \leftarrow most\_recent\_available\_total\_fragment\_number$
  end if
end if
$fragment\_vertex\_ratio \leftarrow totoal\_fragment\_number/totoal\_vertex\_number$

Before dispatching the command group:
$fragment\_number \leftarrow fragment\_vertex\_ratio \times vertex\_number$

6.4 Measurement Optimization and Kernel Driver Modification For Time Measurement

The original driver is not intended for time measurement, a couple of original designs in the GPU kernel driver occur to have interference on the performance of the time measurement. Aside from the influences of the driver, non-real-time linux and multi-core system also affect the measurement stability. Therefore, in order to optimize our measurement framework, several adaptations with minimal influences on the original GPU driver are made in the kernel space. In this section, we present the measurement problems and the related adaptations.

Problem 1: Interference between CPU interrupts and GPU command submission. When an interrupt arrives at a CPU core, the processes on that core will be blocked. If the interrupt happens right after the GPU kernel driver submits the command buffer and before we take the starting timestamps, the starting time we take is delayed compared to the real one. The total execution is therefore shorter than the real execution time. On the other hand, if the interrupt arrives before the kernel driver submits the EVENT command(an EVENT is always submitted after the command group in order to get an interrupt4.5), the time we get the correlated interrupt generated by this EVENT is delayed. The total execution time is longer than the real one. For a single-core system it is difficult to avoid the problem, because the interrupts comes at any time and they should never be blocked. For multi-cores system, we introduced CPU affinity and ensure that the interrupts and our measurement process are located on different cores to avoid the concurrence.

Problem 2: Process Switching during command submitting. This is similar to the first problem, due to the fact that the command buffer and EVENT command are submitted at different time, when process switch happens, the measured execution time can be either longer or shorter than the real one. This problem can be fixed with a real-time linux kernel. However, because at moment real-time linux system is not yet available for our test platform, we bypass the problem by changing the linux scheduler policy from Normal to SCHED_FIFO before the kernel submits the command to the GPU and we change it back to Normal after submitting the EVENT command. The SCHED_FIFO scheduling class is a long-standing, POSIX-specified real-time feature. Processes in this class are given the CPU for as long as they want to fulfil the needs of higher-priority real-time processes. If more than two SCHED_FIFO processes with the same priority are attempting to gain the CPU resource, the one which is currently running on the CPU will continue until it gives back the CPU itself[13], which exactly fits our requirement.

Problem 3: Concurrent OpenGL ES applications and multi cores. Even though the scheduler policy is changed to SCHED_FIFO, in a multi-cores system, multiple OpenGL ES applications can still run simultaneously. This could lead to a wrong command group submission order. For instance, if we have two applications and each submits a command buffer and an EVENT, a possible submission sequence could be: command buffer, command buffer, EVENT, EVENT. In this case, we won't get any interrupt until both command buffers are executed. To avoid this problem, we add an extra Mutex for the command buffer and the EVENT submission to ensure that the submission sequence is always correct.

Problem 4: Mutex and Power Consumption. In 4.5, we mentioned all the tasks attached to a certain interrupt will finally be processed on a separated thread in function gckEVENT_Notify. After processing all the attached tasks this function will walk through all event queues to check if every event queue is empty. If it is true, the function then enquiries the GPU status to check if the GPU is idle. If the GPU is not idle, the function notifies the system to turn down the GPU frequency and make it run in idle mode in order to reduce power consumption.

The problem is that, before the function enquiries the GPU status, it acquires a mutex called powerMutex, this mutex is also acquired in the function that submits the command buffer to

the GPU. If the competition occurs, the GPU execution time we measured will be much longer than in the normal situation because a large amount time is spend on trying to acquire the powerMutex.

To fix the problem, one solution is to remove the function of hardware status enquiry and power consumption reduction, since the power consumption is not that critical to us at moment. This greatly increases the stability of the command execution time which is better for us to do the prediction. Because the chance of the mutex competition is not very high in the measurement, we could also choose to simply ignore some measurement values out of a reasonable range.

Problem 5: EVENT command is submitted to the GPU when a new command queue is in use, this EVENT will trigger the interrupt immediately when it comes to the GPU before the previous command group has been finished. This will disrupt the sequence of the receiving timestamps. Hence we modify this EVENT command and make it trigger the interrupt only after the previous command group is finished.

7 Evaluation

In the chapter, we evaluate the performance of our concepts. In section 7.1, the hardware feature of our evaluation platform is described. The measurement accuracy is evaluated in section 7.2. After that, a special evaluation is performed to verify the correctness our commands mapping concept. The accuracy of our prediction models of rendering commands is evaluated in section 7.4. Besides, the affect of texture size and MSAA on the Draw command is also evaluated in Section 7.5.

7.1 Hardware Setup

The board i.MX6Quad manufactured by Freescale was chosen for our evaluation. The I.MX6Quad application processor is a System-on-Chip focused on multimedia with high processing performance and low power consumption. It is an embedded platform which is suitable for automotive and industrial multimedia applications[19]. Since the future work is to introduce GPU scheduling to real time applications such as speedometer display, navigation system or medium players. The I.MX6Quad developing board is a suitable platform to satisfy the purpose.

The processor is integrated with 4 ARM Cortex A9 cores up to 0.996 GHz per core, 765416KB DDR3 memory, and a 3D OpenGL ES compatible GPU Vivante GC2000 which has the following features[18, 20]:

- 200Mtri/s triangle rate, 1000Mpxl/s fill rate.
- Ultra-threaded, unified vertex and fragment shaders.
- 8k x 8k texture size and 8k x 8k render target.
- 2 KB depth cache, 2KB render target cache, 2KB texture cache.

7.2 Measurement Accuracy

We first evaluated the accuracy of our measurement system in chapter 5 since measuring the execution time of each command group accurately is the precondition for us to do further analysis and to evaluate the prediction models for different rendering commands. For the evaluation of measurement accuracy, we did the following two tests. The Flush test is aimed to evaluate the fluctuation of the measurement system while the Cyclictest is focused on evaluating the max delay during rendering.

7.2.1 Flush test

In this test, we used an application that repetitively dispatches command groups which contain only a single Flush command and measured the execution time for 100000 groups . We chose Flush command because it is very simple and fast, hence the execution time is more sensitive to the fluctuation compared to the command group with a longer execution time.

Figure 7.1 depicts the distribution frequency of the execution time. The interval of every vertical bar is 600 ns. The execution time of Flush ranges from 6 μs to 13.2 μs. Around 92 % of the execution time lies in the interval between 8 μs to 12 μs. Based on the test result, we can say that the fluctuation is quite small. the accuracy of the measurement is sufficient.

7.2.2 Cyclictest

During rendering, large amounts of memory operations are involved. Factors such as hardware interrupts or DMA delay can introduce some latencies into our time measurement system. This test is aimed to find the max latency during rendering.

Cyclictest[21] is a benchmark for testing the latency of an operating system. The general idea of Cyclictest is to take two timestamps between a specified time interval and compare the difference between the specified time interval and the real time interval derived from two timestamps. The difference is used as the latency of the system. The latency can be increased by many factors such as the interrupts and the scheduler overheads.

In this test, we ran the cyclictest and an OpenGL ES application at the same time and gave Cyclictest the highest real-time priority 99 so that the latency we measured in the Cyclictest should be similar to the latency we get in the OpenGL ES application. After running the Cyclictest for 10000 loops with an interval of 10000 micro seconds, the maximum delay we get is 477 μs, this is quite a long time for our time measurement system. However, such a long time delay rarely occurs (in our case, it occurs only once). Such a long delay can introduce some error into our measurement, however, the influence is trivial. The average delay of the

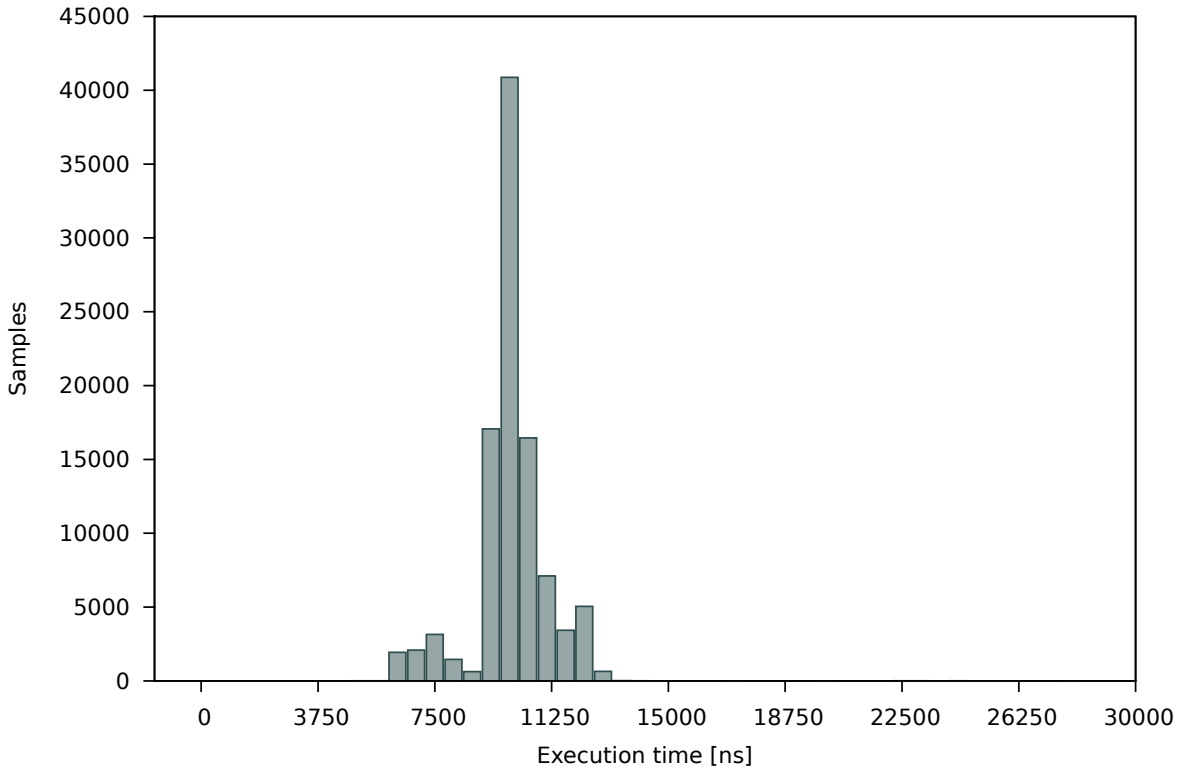

Figure 7.1: glflush Execution Time Distribution

Cyclictest is approximately 23 μs, this is fine since Cyclictest uses the nanosleep function which will introduce some overheads in the time measurement.

7.3 Draw Command Mapping Test

from section 5.3, we know that command groups are submitted in three situations:

- When a Flush command is called.
- When the command buffer is full.
- When the vertex buffer is full and Draw is called.

The Draw command might be split into two parts in the second or in the third situation where the first part of the Draw command is submitted along with the first command group and the second part of the Draw command is submitted with the next command group. In section 5.3, we have explained why we always map the Draw command to the second command group. In

this draw command mapping test, we verify that assumption. We measured the execution time of all scenes and all models of Glmark2-es2 (a benchmark for OpenGL ES 2.0 applications).

First, a python script is used to extract all the command groups that contain a Draw command. These command groups are sorted based on the amount of execution time. Then the average execution time and maximum execution time of the command groups are calculated. After these procedures, we manually analysed the results and make the conclusions whether the Draw command is mapped correctly in two steps:

- Check if the maximum execution time of the Draw command is reasonable. Normally, a command group with Draw in Glmark2-es2 takes a few microseconds or a few hundreds milliseconds to execute, if the maximum execution time of the command groups is very short, we may map all Draw commands incorrectly(considering split may occur in every command group). In this case, we have to look into the application and do further analysis.
- Compare the difference between the maximum execution time and the minimum execution time. Normally, maximum, average and minimum execution time of the Draw command for a same Glmark2-es2 application should have a similar magnitude. If the minimum execution time is far shorter than the maximum execution time and the average execution time, we might map some command groups incorrectly. Again, we have to look into the application and do further analysis.

If both the maximum execution time and the difference between maximun execution time and minimum execution time are reasonable, we conclude that the mapping is correct. After the test, we found out that our mapping is correct for all the applications in Glmark2-es2. Occasionally, we found some command groups with zero-execution time. However, this is due to the max latency in our operating system as we have already evaluated in 7.2.2. When the latency is very long, the GPU interrupt cannot be not processed in time, e.g. when the driver starts to process the first GPU interrupt, the second GPU interrupt also arrives. That's why zero-execution time command group is found. When the long latency occurs, the measured execution time of the command group submitted before the latency is much longer than the real execution time. While the measured execution time of the command group submitted after the latency is much shorter than the real execution time. We found that this is exactly the result that is observed from the experiment.

7.4 Prediction Accuracy

In this section, we evaluated the prediction accuracy of our prediction models for Clear, Swap, Loading Texture and Draw. For the Draw command, the prediction accuracy for the fragment number, the affect of texture size and MSAA optimization are also evaluated.

7.4.1 Prediction Accuracy for the Clear Command

From section 5.5, we know that the default render target and the non-default render target are cleared in a different way, since fast clear technology is only applied for clearing the default render target. Therefore, evaluations for these two types of Clear were also separated.

Evaluation for fast Clear:

To evaluate the execution time of fast Clear, we created an OpenGL ES application that calls glClear to clear the color of the default render target and then draws a big rectangle to fill the whole screen in every frame(the rectangle is rendered to force the execution of the Clear command). A glFlush is inserted after the the Clear command to make Clear a separated command group. We measured the execution time of Clear command with different window resolution (1x1, 320x216, 640x432). We found out that because Clear is very fast, in most of cases, we get no execution time(because two interrupts come together). Only occasionally, we get some measured execution time between 6 to 8 μs which might be raised by the fluctuation in the measurement system. Therefore, we predict the execution time of the fast Clear command as zero. Besides, we also evaluated the influence of Clear command on the Draw and Swap commands. We used the same program (rendering a rectangle on the whole screen) and compared the execution time of Draw and Swap with and without calling glClear before the rendering. We notice that Clear has actually no measurable influence on the execution time of both commands.

Evaluation for Clear of non-default render target:

To evaluate the prediction model of the Clear command for non-default render targets, we created a program which uses glClear to clear non-default render targets with different sizes. glFlush is called after each Clear command to separate the Clear command from other commands. For each size of the rendering target, 5000 groups of Clear commands were executed and measured. In Figure 7.2, the crosses depict the average execution times of the Clear command for the different sizes of the render targets, the line is our prediction value. In the test, the average execution time to clear one byte memory varies from 0.48 nano seconds to 0.53 nano seconds based on the size of the rendering target. An easily visible linear relationship between the size of the rendering target and the execution time can be observed.

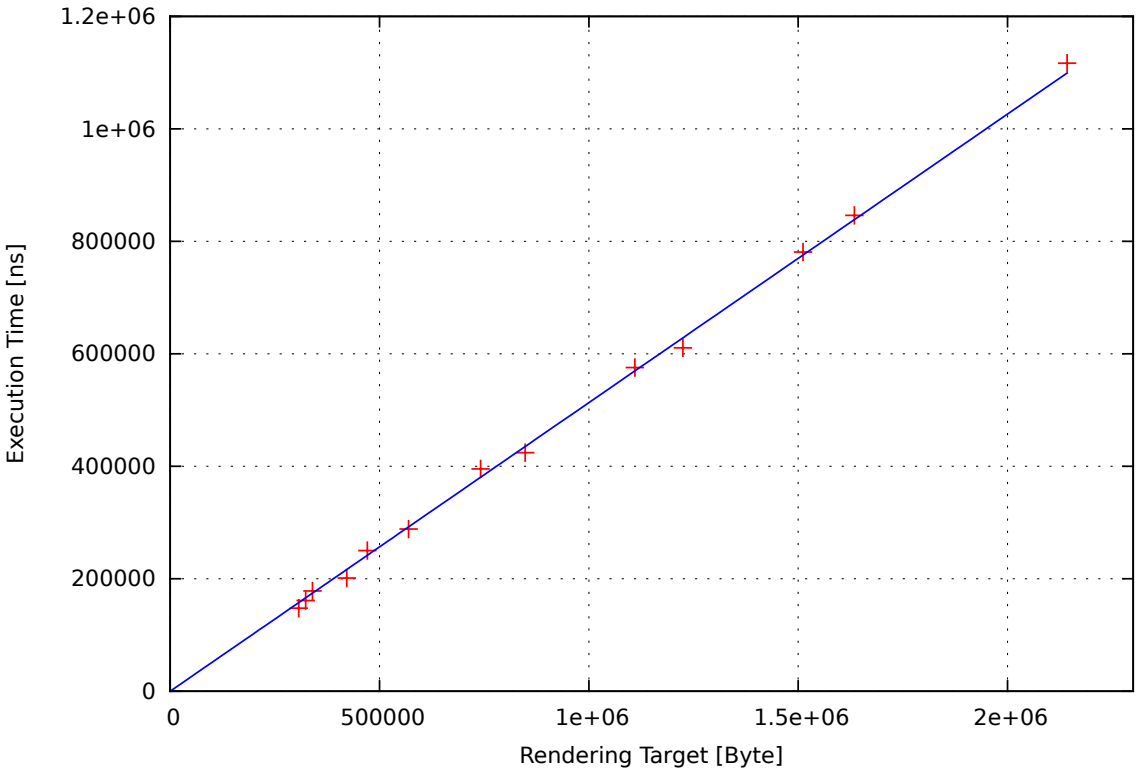

Figure 7.2: Prediction of Clear Command

7.4.2 Prediction Accuracy of the Swap Command

The Swap command posts the content of the back buffer to the front buffer. In section 5.5, we have given our prediction formula. In this test, we evaluated our prediction accuracy of the Swap command. In the test, we created a 640x432 pixel render target. In the first frame, we drew a rectangle covers 160 tiles, the size of every tile is 64 bytes(16 pixels x 4 bytes-per-pixel). Then we increased the size of the rectangle gradually every frame by 160 tiles until the whole rendering target is covered. After that, the rectangle size is again set to 160 tiles and the whole procedure is repeated. In total, 1000 frames are tested and the average execution time for different sizes of rectangle is calculated.

The result of the experiment is depicted in figure 7.3. It shows a strong linear relationship between the tile number and the execution time. The deviation between the prediction and the real execution time is always within 40 μs. Because the CPU and GPU share the same data bus, when a large number of memory is accessed, we found out that even for the command group with exactly the same content, the execution time can also have a deviation around 30 μs. Considering that, our prediction is already quite accurate.

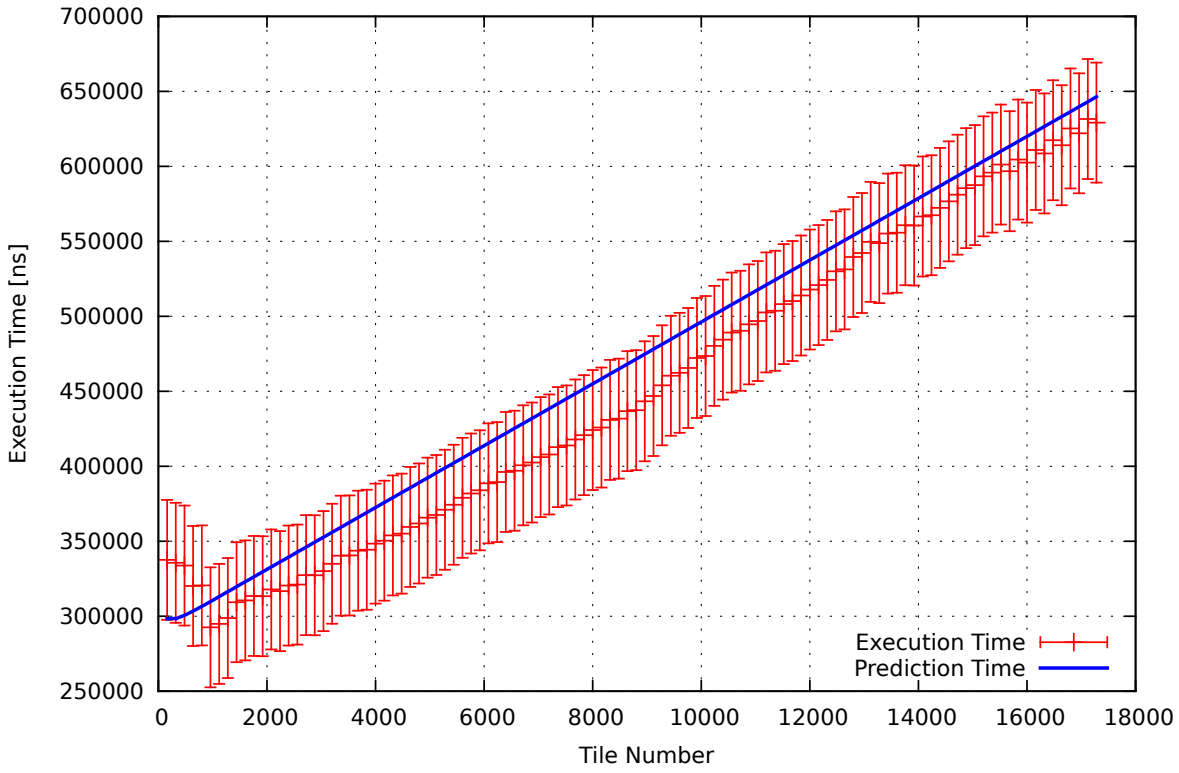

Figure 7.3: Prediction of Swap Command

7.4.3 Prediction Accuracy of Texture Loading(glFrameBufferTexture2D)

In this subsection, we evaluated the prediction accuracy of command glFrameBufferTexture2D to verify and calibrate our prediction model. In the experiment, we defined different size of non-default render target for texture rendering using glTexImage2D and called glFrameBuffer-Texture2D to load the color buffer into the render target. During the experiment, we found that the maximum resolution of the non-default render target is equal to the resolution of the front color buffer. Hence when the resolution of the non-default render target is larger than the resolution of the front color buffer. The execution time is always the same. The result of the evaluation is demonstrated in 7.4. The crosses stand for the real execution time while the lines are our prediction values. For this experiment, the prediction errors are always within 17 μs. Compared to the command execution time, the error is quite small.

7.4.4 Prediction Accuracy of the Draw Command

Prediction Accuracy of Fragments Number: Since our perdition model for the Draw command requires the fragment number of the command group, before we evaluate the prediction

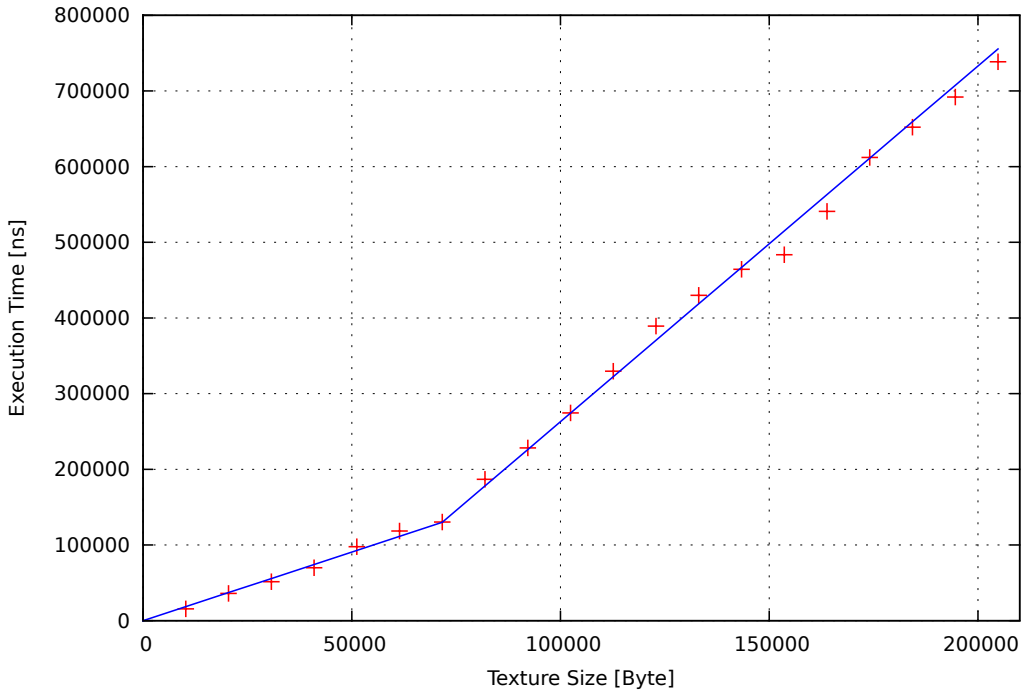

Figure 7.4: Prediction of Texture Loading

accuracy of Draw command, we first evaluate the accuracy of our concept for predicting the fragment number in a command group.

We evaluated the horse model with the scene build in Glmark2-es2 using a frame size of 640x432. In figure 7.5, the dashed curve shows the measured fragment number of each command group. The blue curve depicts the predicted fragments number using the bounding box model while the red curve illustrates the predicted fragments number using our recent-history approach. Figure 7.6 shows the prediction errors which are calculated by using the fragments number depicted in Figure 7.5. The blue curve shows the prediction error using the bounding box model while the red curve illustrates the prediction error using our recent-history approach. As can be seen from the figure, the prediction error of bounding box model is between 30.3% to 160%. This is due to the fact that the percentage of area covered by the horse model in the bounding box varies a lot during the rotation. Hence the prediction error for the bounding box model is quite apparent. The fragment number prediction error using our recent history approach is always within 1.28% while the mean absolute error is only 0.096% of the execution time. The result shows our estimation of the fragments number can achieve a high accuracy in a scenario without abrupt scene changes.

Prediction Accuracy of the Draw Command: We evaluated the predition accuracy of the most critical GPU command, the Draw command. Figure 7.7, 7.8 and 7.9 show the cumulative error distribution of the predicted execution time of the Draw command in different applications.

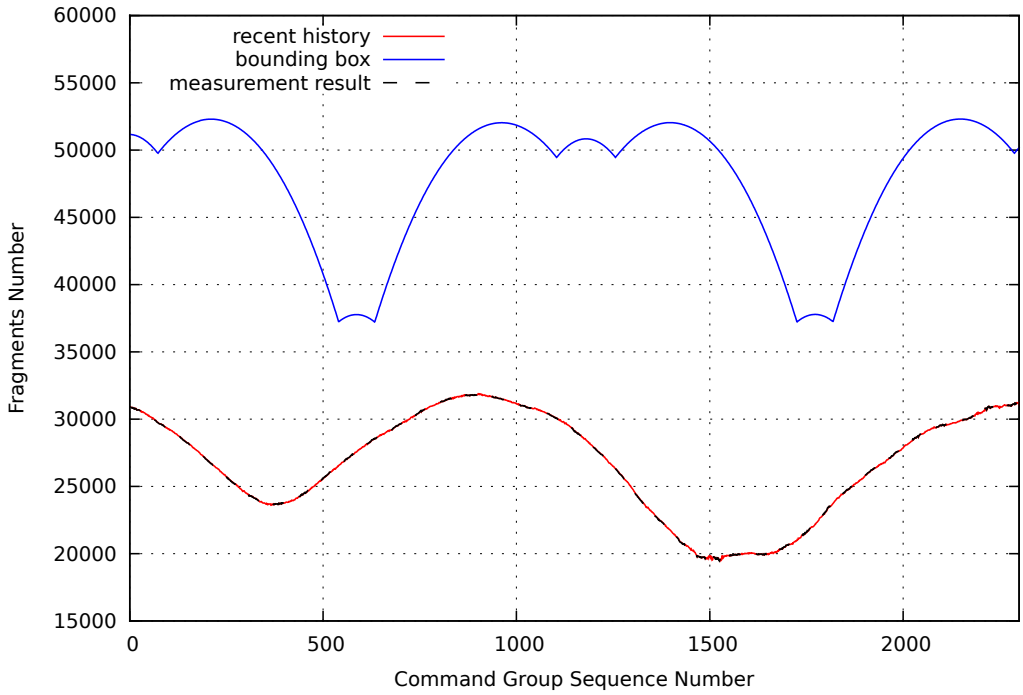

Figure 7.5: Fragment Number Prediction

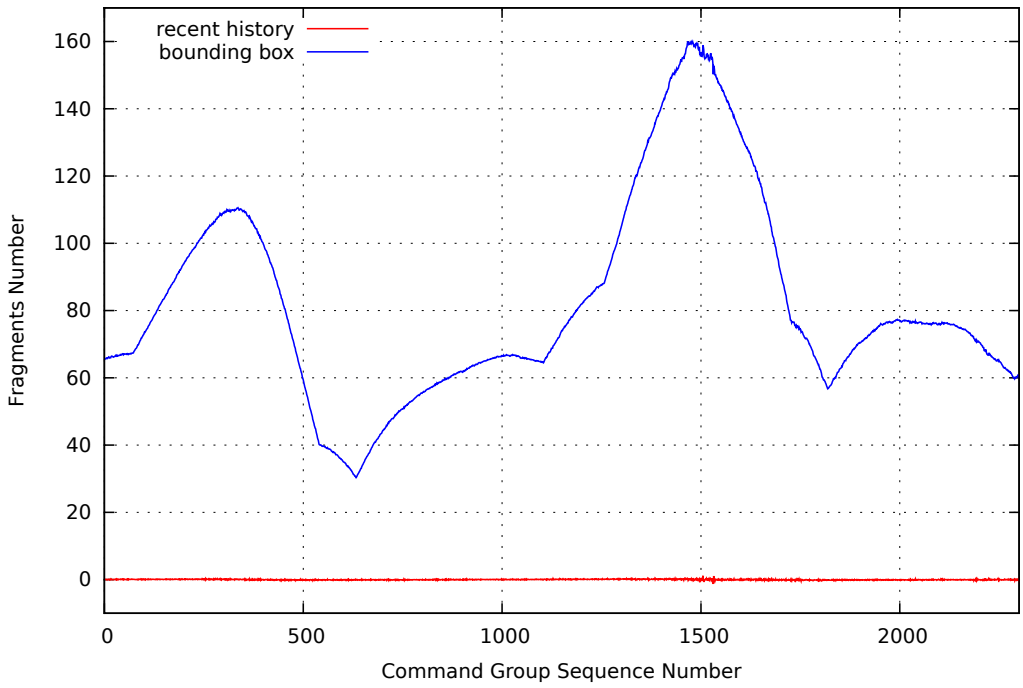

Figure 7.6: Fragment Number Prediction Error

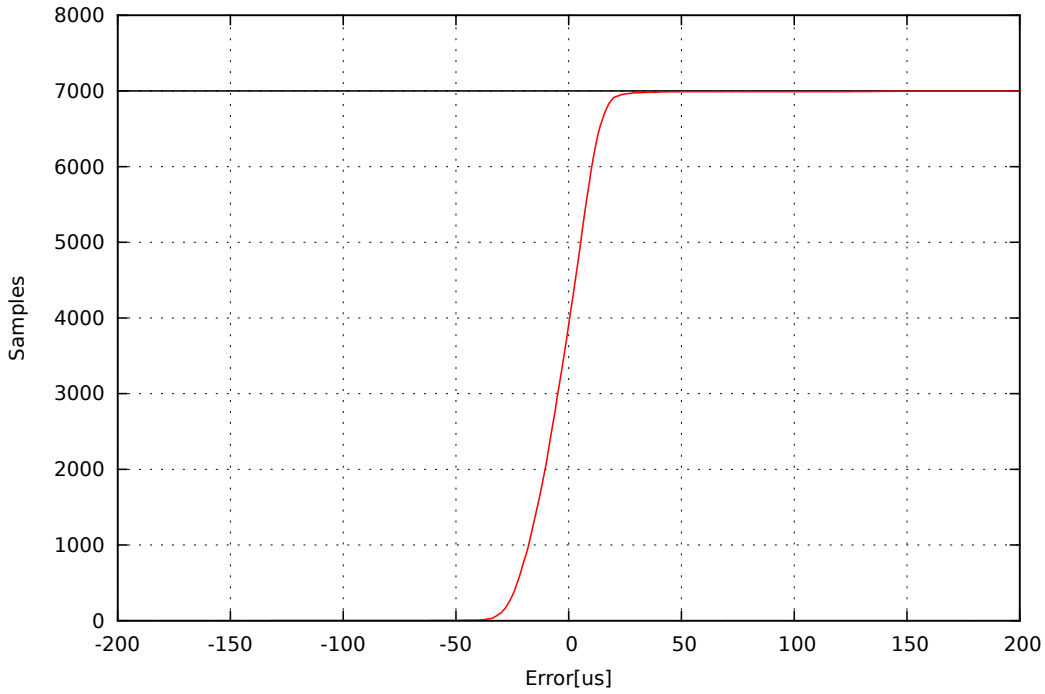

Figure 7.7: Glmark2-es "scene build, model horse"

The window resolution of all applications is set to 640x432. In our evaluations, the first three frames are not evaluated because the GPU pipeline is not stable in the beginning and we also need some initial data before we can predict the fragment number.

In figure 7.7, we evaluated the scene "build" in Glmark-es2 by rendering a rotating horse sculpture on the screen. This program contains only one Draw command in every frame. Every draw contains 21516 vertices. The result shows that an average, a draw command here takes 600.5 μs to execute. The mean absolute error in our prediction is 10.61 μs, which is only 1.77% of the total execution time. Besides, more than 99.4% of the predicted execution time has an absolute error within 25 μs seconds.

In figure 7.8, we evaluated the scene "shading" in Glmark-es2, where a rotating cat sculpture with shading effect is rendered. This program also contains only one Draw command in every frame. The cat model, comprised of 43044 vertices, is more complex than the horse model. On average it takes 1138 μs to execute a Draw command. This time, the mean absolute error is 30.5 μs, 2.6% of the real execution time. 99.87% of the samples has a prediction error within 80 μs. The result shows that our prediction is still quite accurate.

Next, we evaluated a different OpenGL ES benchmark application, es2gears[2], which is a popular OpenGL ES demo. Es2gears renders three rotating gears on the screen. Unlike the horse model and cat model, es2gears uses 280 Draw commands to compose a single frame.

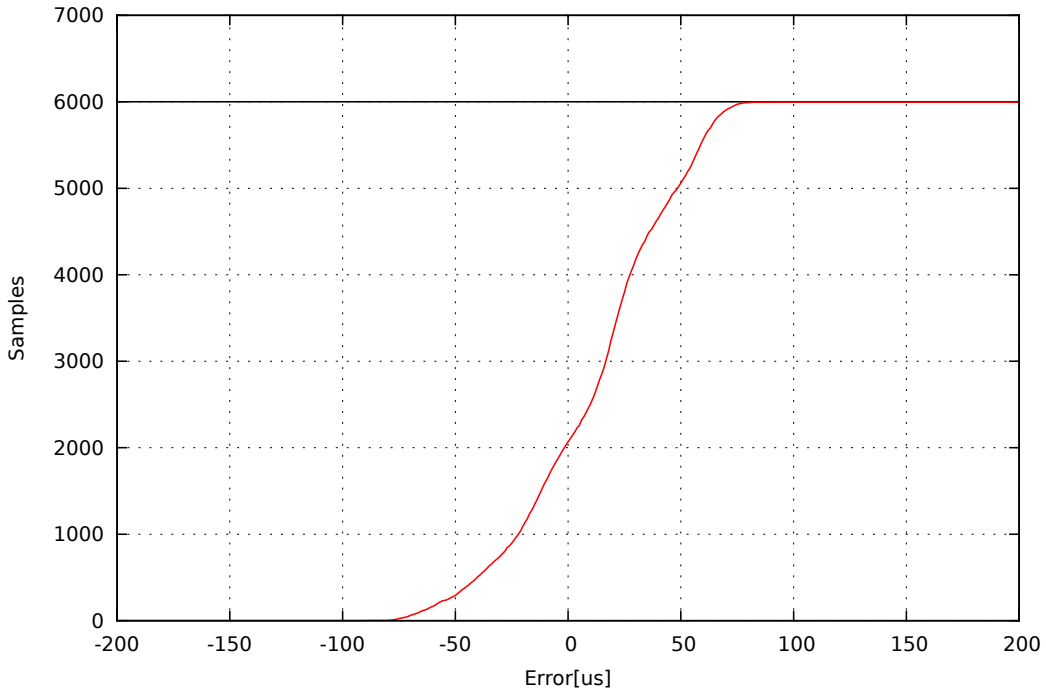

Figure 7.8: Glmark2-es "scene shading, model cat"

The size of a command group without split is 33.1 Kbytes. Since the size of the command buffer is only 128 Kbytes, there's a high chance that a command group can not be filled into a single command buffer and is split up into two command groups. When a split occurs, the number of Draw commands contained in the command group can be an arbitrary number between 0 to 280. Figure 7.9 illustrates that our approach can predict all command groups within an error of 54 μs. Additionally, 97.67% of the command groups have an error less than 15 μs. The result shows our prediction is still valid for command groups with random number of Draw commands.

7.5 Effects of Texture Size and MSAA on Execution of the Draw command

Until now, we have evaluated three different applications for the Draw command. The result shows the Draw prediction model described in section 5.5 can achieve good accuracy for different applications. It also shows that the number of vertices and fragments in a Draw can effectively be used to predict the execution time. Next, we evaluated some other factors that can also influence the execution of Draw.

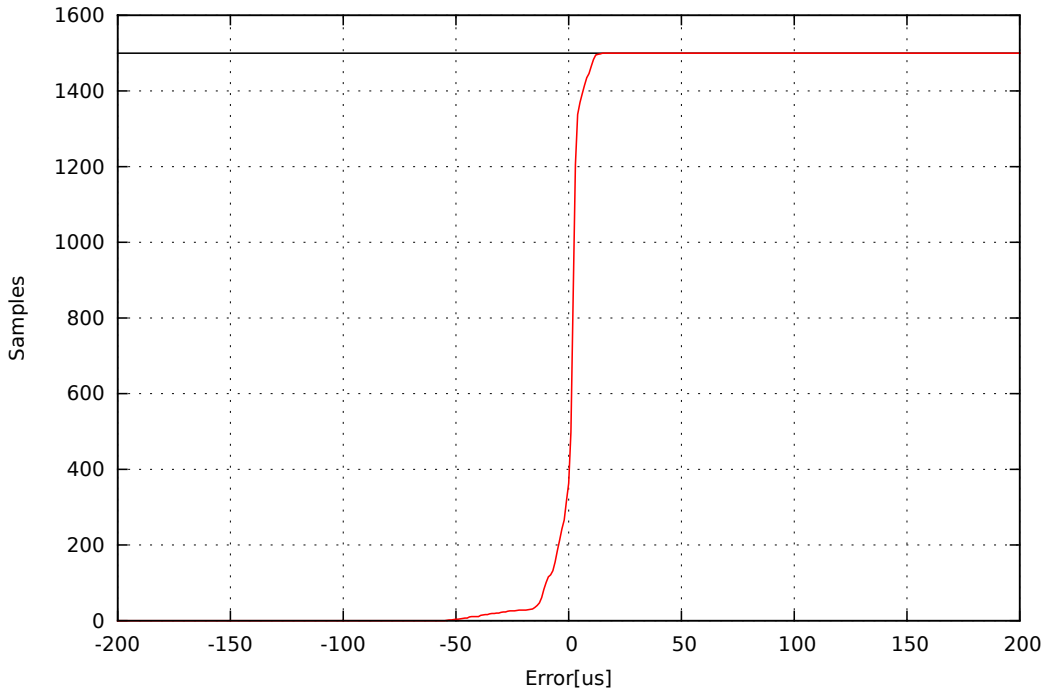

Figure 7.9: Prediction of Draw command: es2gears

Texture Size and Draw Command: In our prediction concept, we have pointed out that even for the same applications, when a different quality of texture is applied, the per-fragment processing speed needs to be re-calibrated. Here we demonstrate the influence of the texture quality (size) on the execution of the Draw command. In order to do that, we created a test program where a rectangle with 320x216 pixels is rendered. Then we map different resolutions of texture image on the rectangle using point sampling. Figure 7.10 demonstrates the relationship between texture size and the execution time for rendering the rectangle. In the figure, the crosses stand for the average execution time of the Draw commands using different size of textures. We can clearly see that different sizes of texture lead to different GPU execution time. There is a high probability that the difference in the execution time is the result of different texture cache prefetch hit rates. With the increase of the texture size, the prefetch hit rate is reduced and a fragment takes much longer to process when the texture is not in the cache memory.

MSAA and Draw Command: Finally, we evaluated the effect of MSAA on the Draw command. we evaluated an application which renders a single triangle on half of the whole screen(screen size: 640x432) by measuring the average execution time of the Draw in 2000 frames. Figure 7.11 shows that the draw without MSAA takes on average 224.7 μs to execute. When 4x MSAA is applied, a Draw takes 913.3 μs to execute, which is approximately 4.06 times of the Draw without MSAA. Since the Draw only contains 3 vertices, the execution time for vertex

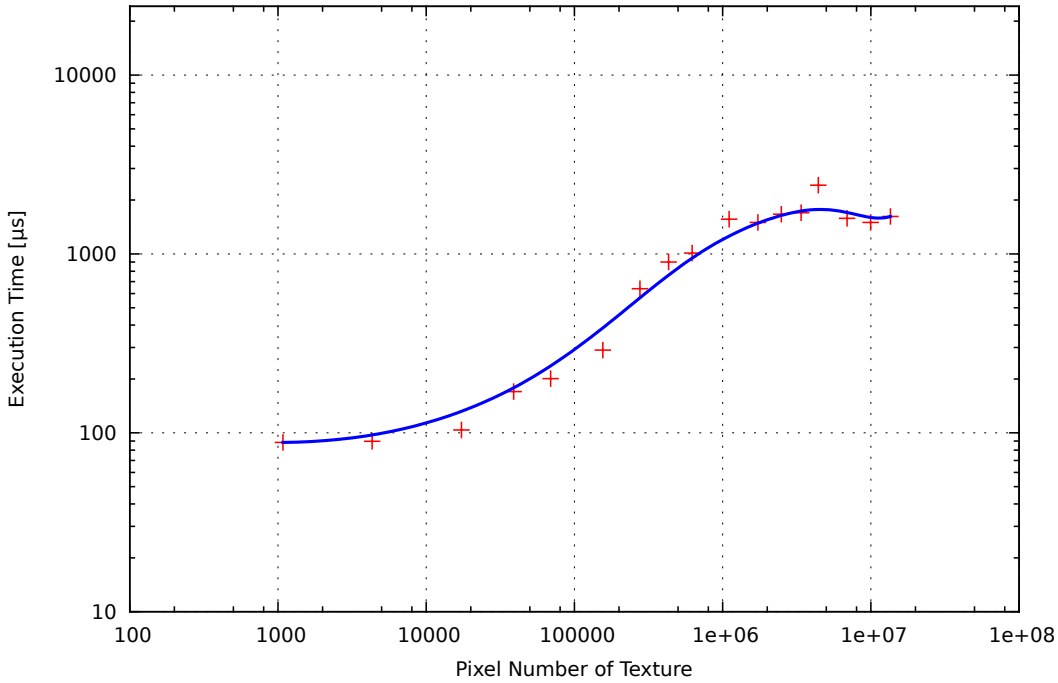

Figure 7.10: Texture Size and Execution Time

processing can be almost ignored. That means when 4x MSAA is applied, the per-fragment processing speed can be reduced to only one-fourth of the speed without MSAA.

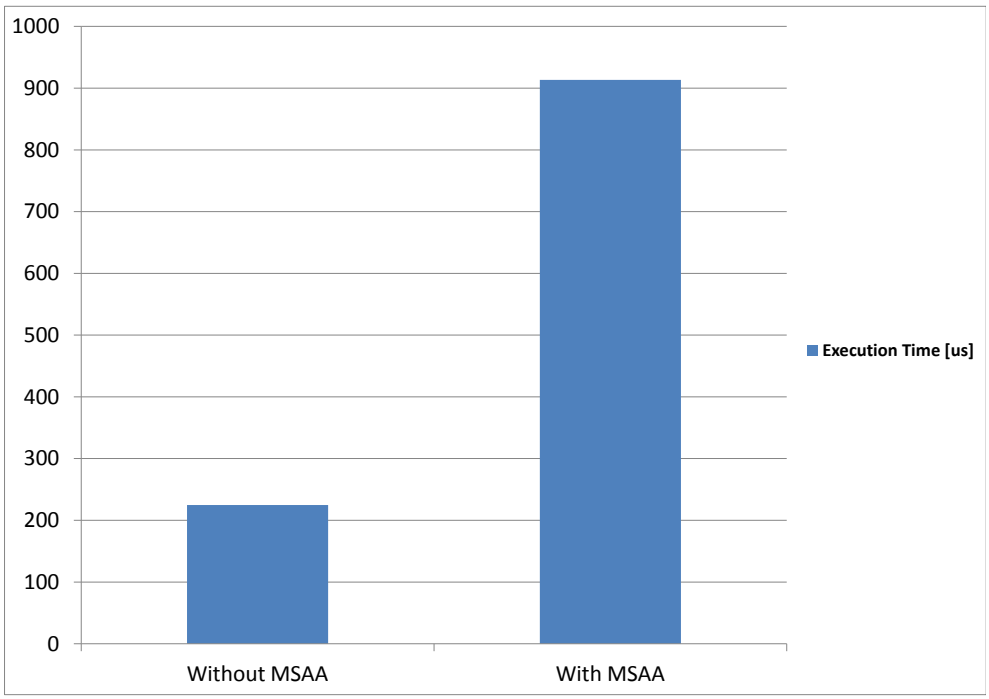

Figure 7.11: Execution Time without and with MSAA

8 Summary and Future Work

8.1 Summary

This work is motivated by the GPU real time scheduling for 3D rendering applications. Because current GPUs do not support preemption, a possible solution for scheduling real-time GPU applications is to use non-preemptive deadline-driven scheduling. Thus emerges the need of predicting the GPU execution time.

In this work, we explained the driver model of a typical embedded GPU in details. Many hardware features of embedded GPUs are also revealed. Based on the knowledge of the GPU driver and hardware, we designed and implemented a measurement and prediction system on the top of the existed system framework. In the prediction system, prediction models for time consuming rendering commands (Flush, Swap, Clear, Draw, Texture Loading) are built. Besides, a recent-history-based approach is proposed to estimate the fragments number of the GPU command groups which is required by predicting the execution time of Draw command.

Evaluations using OpenGL ES 2.0 graphics benchmarks demonstrate that our prediction system achieves good accuracy in estimating the execution time of different rendering commands. The recent-history-based approach is also effective for estimating the number of fragments in GPU command groups.

8.2 Future Work

One direction of the future work is to develop and implement a GPU real-time scheduler on the top of our measurement and prediction system. As another part of the future work, we will move our current work to a linux system with realtime-preempt kernel so that the latency in the measurement system can be reduced.

Moreover, because our prediction uses the data of previous frames, the prediction can be inaccurate at the time when the rendering scene is changed. Hence another extension of this work is to detect the scene changes and improve the prediction model during scene changes.

In the evaluation, we have revealed that GPU optimization techniques, such as MSAA, can significantly affect the execution time of the Draw command. Therefore another aspect that

can be explored is to improve the prediction models and make it adaptable to applications with GPU optimization techniques.

A Appendices

A.1 Important Data Structures of the Measurement and Prediction System

In this section, we explain the important data structure that is used in our system. These structures can be found in file gc_hal_kernel_ shared_mem.h and the file predict_data.h

gckProcessSharedCmdBuf: This structure is used by the application(process) shared memory.

```
typedef struct Process_Shared_Command_Buffer{
  gctUINT seq_no;           // The sequence number of the command group in the application.
  gctUINT global_seq_no;    // The sequence number of the command group in all applications.
  gctUINT buf_size;         // The size of the command name list
  gctINT8* buf_content;     // The content of the command name list
  gctUINT data_size;        // Size of the command group
  gctINT8* data;            // Hex content of the comnand group
  gctUINT64 time_submitted; // Command group submitting time
  gctUINT64 time_finished;  // Command group finishing time
  gctUINT64 time_execution; // Command group execution time
  gctUINT64 signal_no;      // The signals attached to the command groups
  gctUINT32 vertex_number;  // Total vertex number of the command group
  gctUINT32 fragment_number; // Total fragment number of the command group
  gctUINT32 predict_frag_number; // Predicted fragment number of the command group
  gctUINT64 time_prediction; // Predicted command group execution time
}gckProcessSharedCmdBuf;
```

gckGLobalSharedCmdBuf: This structure is used by the global shared memory. The information it contains is similar to information in the structure gckProcessSharedCmdBuf. The difference is that it also records the execution time of single EVENT command and the owner of the command group(Process ID).

gckProcessSharedMem: Each application has its own shared memory for storing the information of its command groups. The memory is managed by this structure.

```
typedef struct __gckProcessSharedMem{
  gctINT pid;           // The owner(process id) of the process-dependent shared memory.
  gctINT index;         // The current command group index of the process-dependent shared
    memory.
```

```
  gckProcessSharedCmdBuf *pt_mem; // Points to the address of the process-dependent shared
    memory.
  gckProcessSharedMem_PTR next;  // Points to the next gckProcessSharedMem struct.
  gctBOOL mem_is_allocated;
}gckProcessSharedMem;
```

gckProcessSharedMemManager: The measurement system uses this structure to access the shared memory of each application. It is also used to free the memory when the memory is no longer needed.

```
typedef struct __gckProcessSharedMemManager{
  gctUINT32 process_cnt;
  gckProcessSharedMem_PTR head;
  gckProcessSharedMem_PTR tail;
}gckProcessSharedMemManager;
```

A.2 Driver Interface Command Code

Command Code	**Description**
gcvHAL_QUERY_VIDEO_MEMORY	Query video memory size
gcvHAL_QUERY_CHIP_IDENTITY	Query chip identity
gcvHAL_ALLOCATE_NON_PAGED_MEMORY	Allocate non-paged memory
gcvHAL_FREE_NON_PAGED_MEMORY	Free non-paged memory
gcvHAL_ALLOCATE_CONTIGUOUS_MEMORY	Allocate contiguous memory for command groups
gcvHAL_FREE_CONTIGUOUS_MEMORY	Free contiguous memory for command groups
gcvHAL_ALLOCATE_VIDEO_MEMORY	Allocate video memory
gcvHAL_ALLOCATE_LINEAR_VIDEO_MEMORY	Allocate linear video memory
gcvHAL_FREE_VIDEO_MEMORY	Free video memory
gcvHAL_MAP_MEMORY	Physical-to-logical memory mapping
gcvHAL_UNMAP_MEMORY	Physical-to-logical memory unmapping
gcvHAL_MAP_USER_MEMORY	Logical-to-physical memory mapping
gcvHAL_UNMAP_USER_MEMORY	Logical-to-physical memory unmapping
gcvHAL_LOCK_VIDEO_MEMORY	Lock the video memory so they can be accessed
gcvHAL_UNLOCK_VIDEO_MEMORY	Unlock the video memory
gcvHAL_EVENT_COMMIT	Commit an event queue
gcvHAL_USER_SIGNAL	Dispatch depends on the user signal subcommands
gcvHAL_SIGNAL	Kernel Signal
gcvHAL_WRITE_DATA	Write data into specified address
gcvHAL_COMMIT	Commit a command and context buffer(if exists)
gcvHAL_READ_REGISTER	Read a GPU register
gcvHAL_WRITE_REGISTER	Write a GPU register
gcvHAL_ATTACH	Attach user process to the kernel driver
gcvHAL_DETACH	Detach user process from the kernel driver

Table A.1: Command Code

Bibliography

[1] URL http://forum.donanimhaber.com/m_71818465/tm.htm. (Cited on pages ix and 17)

[2] URL http://www.mesa3d.org/. (Cited on page 74)

[3] 2013 Motor Trend Car of the Year: Tesla Model S. URL http://www.motortrend.com/oftheyear/car/1301_2013_motor_trend_car_of_the_year_tesla_model_s/. (Cited on page 1)

[4] Enjoy the ride with NVIDIA IN-VEHICLE INFOTAINMENT. URL http://www.nvidia.com/object/automotive-infotainment-navigation.html. (Cited on pages ix and 2)

[5] GLSL Programming/Rasterization. URL http://en.wikibooks.org/wiki/GLSL_Programming/Rasterization. (Cited on pages ix and 14)

[6] Mobile Multimedia. URL http://www.vivantecorp.com/index.php/en/products/markets.html. (Cited on page 1)

[7] Spatial anti-aliasing. URL http://en.wikipedia.org/wiki/Spatial_anti-aliasing. (Cited on pages ix and 15)

[8] The Standard for Embedded Accelerated 3D Graphics. URL https://www.khronos.org/opengles/. (Cited on page 9)

[9] Tesla Model S to have 17-inch infotainment console powered by Tegra. URL http://www.engadget.com/2011/01/04/tesla-model-s-to-have-17-inch-infotainment-console-powered-by-t/. (Cited on page 1)

[10] M. Bautin, A. Dwarakinath, T. cker Chiueh. Graphic Engine Resource Management. *in Fifteenth Annual ACM/SPIE Multimedia Computing and Networking Conference (MMCN'08)*, 2008. (Cited on pages 3, 5, 6 and 7)

[11] K. Beets. PowerVR Tile Based Rendering, 2007. URL http://www.beyond3d.com/content/articles/38/1. (Cited on page 20)

[12] C.L.LIU, J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM 20*, 1973. (Cited on page 5)

[13] J. Corbet. SCHED_FIFO and realtime throttling, 2008. URL http://lwn.net/Articles/296419/. (Cited on page 63)

[14] P. Cozzi, C. Riccio. *OpenGL Insights*. CRC press, 2012. (Cited on page 19)

[15] A. U. Diril, F. Garritsen. EFFICIENT TILE-BASED RASTERIZATION. patent, 2012. URL http://www.faqs.org/patents/app/20120044245. (Cited on page 22)

[16] A. Dwarakinath. Master's thesis, Stony Brook University, 2008. (Cited on pages 6 and 7)

[17] G. A. Elliott, J. H. Anderson. Real-World Constraints of GPUs in Real-Time Systems. *Embedded and Real-Time Computing Systems and Applications(RTCSA)*, 2011. (Cited on page 1)

[18] Freeescale. URL http://cache.freescale.com/files/32bit/doc/fact_sheet/IMX6SRSFS.pdf. (Cited on page 65)

[19] Freescale. IMX6Q Reference Manual. URL http://www.freescale.com/infocenter/index.jsp?topic=%2FiMX%2Findex.html. (Cited on page 65)

[20] freescale. i.MX 6Dual/6Quad Applications Processor Reference Manual, 2013. URL http://cache.freescale.com/files/32bit/doc/ref_manual/IMX6DQRM.pdf?fasp=1&WT_TYPE=Reference%20Manuals&WT_VENDOR=FREESCALE&WT_FILE_FORMAT=pdf&WT_ASSET=Documentation&Parent_nodeId=13376371545356958310 62&Parent_pageType=product. (Cited on pages ix, 22, 31, 53 and 65)

[21] T. Gleixner. Cyclictest. URL https://rt.wiki.kernel.org/index.php/Cyclictest. (Cited on page 66)

[22] B. A. Gurulingesh Raravi. Calculating an upper bound on the finishing time of a group of threads executing on a GPU: A preliminary case study. *In Work-in-progress session of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 5–8, 2010. (Cited on page 8)

[23] B. Johnsson, T. Akenine-Möller. Measuring Per-Frame Energy Consumption of Real-Time Graphics Applications. *journal of Computer Graphics Techniques Vol. 3, No. 1*, 2014. (Cited on page 6)

[24] S. Kato, K. Lakshmanan, R. R. Rajkumar, Y. Ishikawa. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. *in Proceedings of USENIX conference on USENIX annual technical conference*, 2011. (Cited on pages 3, 5, 6 and 7)

[25] W. J. van der Laan. cmdstream. URL https://github.com/laanwj/etna_viv/blob/master/rnndb/cmdstream.xml. (Cited on page 28)

[26] W. J. van der Laan. Etnaviv. URL https://github.com/laanwj/etna_viv. (Cited on pages 25 and 30)

[27] Matt. A Quick Overview of MSAA, 2012. URL http://mynameismjp.wordpress.com/2012/10/24/msaa-overview/. (Cited on page 17)

[28] A. Munshi, D. Ginsburg, D. Shreiner. *OpenGL ES 2.0 programming Guide*. Addsion-Wesley, 2008. (Cited on pages ix, 9, 10, 15, 16 and 17)

[29] A. Munshi, J. Leech. OpenGL ES Common Profile Specification Version 2.0.25(Full Specification). 2010. (Cited on page 50)

[30] J. P. Research. Embedded graphics: Imagination Technologies supplies more GPU IP than all others combined, 2012. URL http://jonpeddie.com/press-releases/details/embedded-graphics-imagination-technologies-supplies-more-gpu-ip-then-all-ot/. (Cited on page 18)

[31] M. Ribble. Next-Gen Tile-Based GPUs, 2008. URL http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/gdc2008_ribble_maurice_TileBasedGpus.pdf. (Cited on page 20)

[32] G. Schechter. The role of the Windows Display Driver Model in the DWM, 2006. URL http://blogs.msdn.com/b/greg_schechter/archive/2006/04/02/566767.aspx. (Cited on page 5)

[33] S. Schnitzer, S. Gansel, F. Dürr, K. Rothermel. Concepts for Execution Time Prediction of 3D GPU Rendering. *in Proceedings of 9th IEEE International Symposium on Industrial Embedded System(SIES)*, pp. 160–169, 2014. (Cited on pages 3, 6, 8, 45 and 50)

[34] M. C. Shebanow. AN EVOLUTION OF MOBILE GRAPHICS, 2013. URL http://highperformancegraphics.org/wp-content/uploads/Shebanow-Keynote.pdf. (Cited on pages x, 18 and 22)

[35] P. S.Heckbert. Survey of Texture Mapping. *IEEE Computer Graphics and Applications*, pp. 56–67, 1986. (Cited on page 14)

[36] L. Williams. Pyramidal Parametrics. *Computer Graphics, SIGGRAPH '83 Proceedings*, pp. 1–11, 1983. (Cited on page 15)

[37] M. Wimmer, P. Wonka. Rendering Time Estimation for Real-Time Rendering. *Proceedings of the Eurographics Symposium on Rendering*, 2003. (Cited on page 54)

[38] J.-H. Woo, J.-H. Sohn, B.-G. Nam, H.-J. Yoo. *MOBILE 3D GRAPHICS SoC From Algorithm to Chip*. John Wiley and Sons(Aisa) Pte Ltd, 2010. (Cited on page 21)

[39] B. Zuehlke. Mobile GPUs : Introduction and Challenges, 2012. URL http://bastianzuehlke.wordpress.com/2011/10/18/mobile-gpus-introduction-challenges/. (Cited on page 18)

[40] B. Zuehlke. Mobile GPUs : Architectures, 2013. URL http://bastianzuehlke.wordpress.com/2012/04/05/mobile-gpus-architectures/. (Cited on pages 20 and 21)

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