“GPU Debugging Tool - GPU SnapShot”

Submitted in Partial Fulfillment of the
Requirements for completion of

Major Project Report

By

Aniruddhsinh Parmar
22MECE04

Internal Guide:
Dr. Nagendra Gajjar

Department of Electronics and Communication Engineering,
Institute of Technology,
Nirma University,
Ahmedabad 382 481

May 2024

CERTIFICATE

This is to certify that the Comprehensive Evaluation Report entitled "GPU SnapShot" submitted by Mr. Parmar Aniruddhsinh Rajendrasinh (22MECE04) towards the partial fulfillment of for completion of Course on' is the record of work carried out by him/her individually.

Date: 24/04/2023
Faculty Coordinator: Dr. Nagendra Gajjar

Undertaking for Originality of the Work

I Parmar Aniruddhsinh Rajendrasinh (22MECE04) give undertaking that the Comprehensive Evaluation Report entitled "GPU SnapShot" submitted by me, towards the partial fulfillment of for completion of Course, is the original work carried out by me and I give assurance that no attempt of plagiarism has been made. I understand that in the event of any similarity found subsequently with any other published work or any report elsewhere; it will result in severe disciplinary action.

_____

Signature of the Student

Date: _____

Place: _____

Abstract

GPU snapshot debugging tool, a sophisticated utility meticulously crafted to capture a comprehensive array of GPU-related data vital for troubleshooting. It adeptly collects diverse information including ring buffers, registers, data structures, instruction buffers (PM4 packets), and page tables. Functioning seamlessly within a sysfs node, the tool orchestrates the creation of a binary file housing the gathered data. To render this information accessible and understandable, a Python script is enlisted to process the binary file, transforming it into a human-readable format. This streamlined process empowers developers to efficiently debug issues, thereby enhancing the robustness and stability of GPU-dependent systems.

INDEX

LIST OF FIGURES

1.1 Introduction:

1. Introduction GPU:
    - What is a GPU?
    - Use of GPU

- What is a GPU?

  A GPU, or Graphics Processing Unit, is a specialized electronic circuit designed to quickly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. Initially developed for rendering images, videos, and animations in computer games, GPUs have evolved to handle a wide range of complex computational tasks beyond graphics rendering.

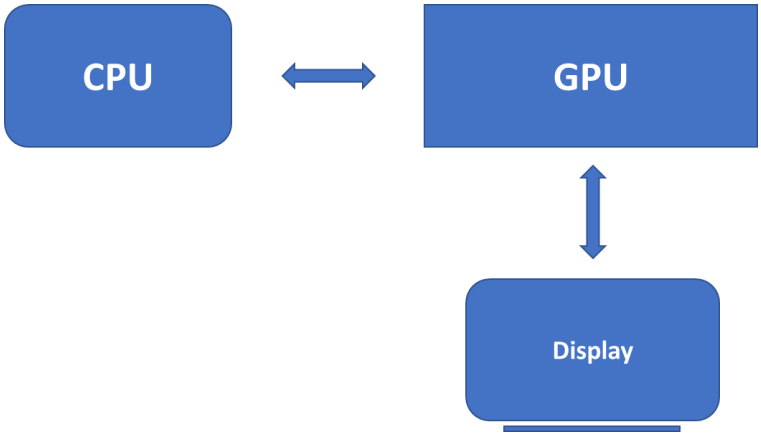

Fig. 1.1 – block overview of GPU

This is just an overview block diagram of the GPU in GPU pipeline part we understand much more about the GPU work and functionality.

- Use of GPU:

  1. Gaming and Entertainment:

    - Graphics Rendering: GPUs are crucial for rendering high-quality graphics in video games, providing immersive visual experiences and enabling smoother gameplay by handling complex rendering tasks.

    - Virtual Reality (VR) and Augmented Reality (AR): GPUs power VR and AR experiences by rendering realistic and immersive environments.

2. Scientific Computing and Research:

- Simulation and Modeling: GPUs accelerate scientific simulations and modeling in fields like physics, chemistry, biology, and engineering, allowing researchers to perform complex computations faster.

- Data Analysis: GPUs are used for processing large datasets and performing data analysis in scientific research and fields like astrophysics, climate modeling, and bioinformatics.

3. Artificial Intelligence (AI) and Machine Learning (ML):

- Deep Learning: GPUs excel in training and running deep neural networks, which are fundamental to various AI applications, including image and speech recognition, natural language processing, and autonomous vehicles.

- AI Research: Researchers use GPUs to experiment and develop new AI algorithms and models due to their high computational power.

4. Medical Imaging and Healthcare:

- Medical Imaging: GPUs are employed in medical imaging technologies such as MRI (Magnetic Resonance Imaging), CT (Computed Tomography), and ultrasound for faster image reconstruction and analysis.

- Drug Discovery: GPUs aid in computational tasks related to drug discovery, molecular dynamics simulations, and analyzing biological data.

5. Financial Services:

- Risk Analysis and Trading: GPUs are used in financial institutions for risk modeling, portfolio optimization, algorithmic trading, and performing complex financial calculations quickly.

6. Design and Creativity:

- Graphic Design and Animation: Professionals use GPUs for graphic design, animation, video editing, and rendering in industries like film production, advertising, and architecture, enabling faster rendering times and real-time previews.

7. Cryptocurrency Mining:

- Mining Operations: GPUs are utilized in mining cryptocurrencies like Bitcoin, Ethereum, and others, performing the necessary computations to verify transactions and secure blockchain networks.

1.2 GPU Pipeline Overview:

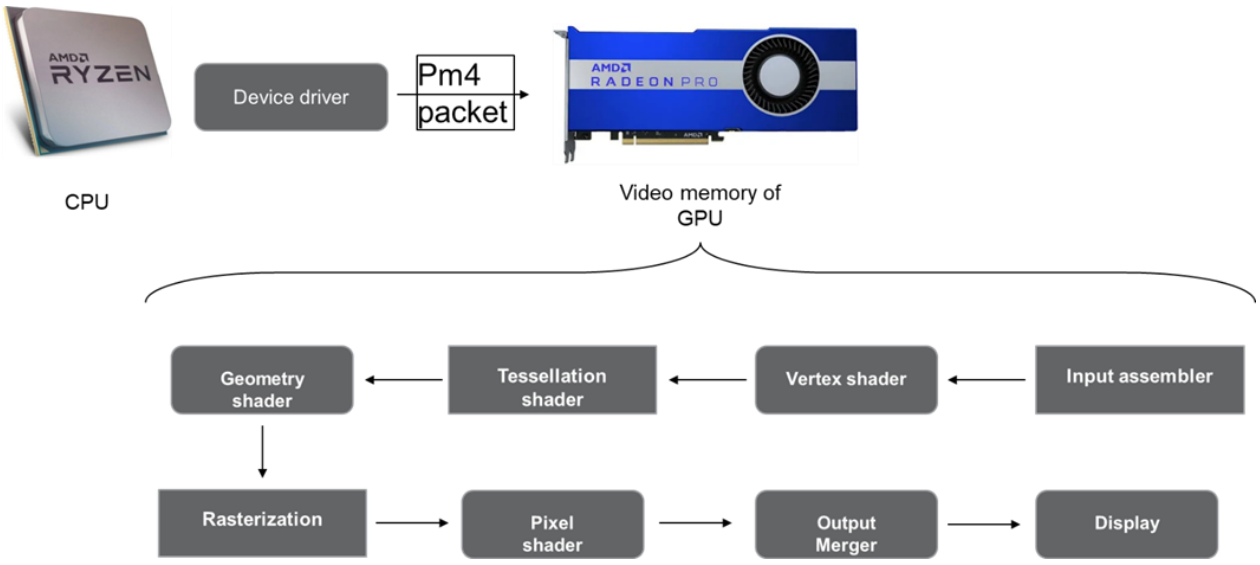

Fig 1.2 GPU pipeline block diagram

1. Application Stage:

Input Assembly:

- Vertex shaders receive input attributes for each vertex, like its position, color, texture coordinates, etc.

Vertex Shading:

- A vertex shader is a programmable component in modern graphics hardware that operates on individual vertices of geometric objects in 3D space. It's a fundamental stage in the graphics pipeline responsible for transforming vertices from their original positions in object space to their final positions in screen space.

- **Key Aspects of Vertex Shader:**

Purpose:

- Vertex shaders primarily handle transformations and manipulations of vertices.

- They perform operations on each vertex, such as translation, rotation, scaling, and other mathematical computations.

Execution:

- Vertex shaders are executed for every vertex of a 3D object before rasterization and pixel shading.
- They run in parallel on the GPU, allowing for efficient processing of multiple vertices simultaneously.

Customization and Programmability:

- Vertex shaders are programmable, allowing developers to write custom shader code in languages like GLSL (OpenGL Shading Language) or HLSL (High-Level Shading Language).
- Developers can define the specific transformations and operations they want to apply to vertices.

Input and Output:

- Inputs to a vertex shader include attributes associated with each vertex, such as position, color, texture coordinates, and normal.
- Outputs typically include the transformed vertex position and any other modified attributes.

Transformation and Projection:

- Vertex shaders handle transformations between different coordinate spaces, such as model space, world space, view space, and eventually screen space (or clip space).
- They transform vertices based on the model, view, and projection matrices to prepare them for rasterization and display.

Lighting and Effects (Optional):

- Vertex shaders can also handle basic lighting calculations, preparing data for the subsequent pixel shading stage.
- They can compute lighting effects like diffuse lighting, specular highlights, or other vertex-related visual effects.

- **Workflow and Operations:**

Input Assembly:

- Vertex shaders receive input attributes for each vertex, like its position, color, texture coordinates, etc.

Transformation:

- The shader code applies transformations to each vertex, manipulating its position and attributes based on the defined operations.

Coordinate Space Conversion:

- Vertices are transformed between various coordinate spaces to ensure correct positioning and orientation in the 3D scene.

Output:

- The vertex shader produces transformed vertices with updated attributes as output.
- These modified vertices proceed to subsequent stages of the graphics pipeline, such as geometry shading, clipping, and rasterization.

- **Example Scenario:**

For instance, in a 3D game:

- Vertex shaders can handle the transformation of vertices for character models, moving them from their original positions in 3D space to the final positions for rendering on the screen.
- They can also manipulate attributes like bone positions for skeletal animation or apply other effects like waving flags or deforming objects.
- In summary, vertex shaders play a crucial role in the graphics pipeline by transforming individual vertices, enabling the positioning and manipulation of 3D objects, and preparing them for subsequent stages of rendering in real-time graphics applications.

2. Tessellation shaders:

Tessellation shaders are a key component in modern graphics pipelines that allow for dynamic, on-the-fly subdivision of polygons. They enhance geometric detail and improve the visual quality of rendered surfaces by adding more vertices to existing primitives (such as triangles) in real-time.

Key Aspects of Tessellation Shaders:

Purpose:

- Tessellation shaders provide a way to dynamically refine or coarsen the level of detail (LOD) of surfaces in real-time.
- They enable the subdivision of basic geometric primitives (typically triangles) into smaller or more complex shapes.

Tessellation Stages:

- Tessellation shaders consist of three main stages: Control Shader, Tessellation Control Shader (TCS), Tessellation Evaluation Shader (TES).

Control Shader:

- The Control Shader (part of the pre-tessellation stage) sets tessellation parameters and computes per-vertex data.
- It determines how many new vertices will be generated between the original vertices, controlling the level of tessellation.

Tessellation Control Shader (TCS):

- The TCS receives input control points from the vertex shader.
- It calculates tessellation factors and other per-vertex data to control the tessellation level and determine the tessellation factors for patches.

Tessellation Evaluation Shader (TES):

- The TES is responsible for generating new vertices based on the tessellation factors.
- It evaluates the tessellated surface positions, interpolating and refining the vertices to create additional detail.
- Fixed Function Tessellation Pipeline:

- The tessellation stages are part of the fixed-function graphics pipeline and are processed on the GPU, enhancing geometric detail without requiring CPU intervention.

Adaptive Tessellation:

- Tessellation can be adaptive, allowing for varying levels of subdivision based on distance or other factors.
- This adaptive approach ensures that geometry is tessellated more in areas closer to the camera or where higher detail is required.

Workflow and Operations:

Control Point Generation:

- The original vertices generated by the vertex shader act as control points for tessellation.
- The control shader determines parameters for tessellation, such as how many new vertices to generate between these control points.

Tessellation Factors Calculation:

- The Tessellation Control Shader (TCS) computes tessellation factors based on the control points and other parameters.
- These factors dictate the level of tessellation and how much additional geometry should be generated.

Tessellation Execution:

- The Tessellation Evaluation Shader (TES) uses the tessellation factors to generate new vertices or points within each original primitive.
- These new vertices create additional detail, refining the surface or geometry.

Interpolation and Smoothing:

- The TES interpolates attributes (such as position, normal, texture coordinates) for the newly generated vertices, ensuring smooth transitions.

Use Cases:

- Terrain Rendering: Tessellation can be used to dynamically subdivide terrain meshes based on proximity to the viewer, allowing for increased detail where needed.
- Character Models: It can enhance the detail of character models in games by adding finer geometry to improve visual quality, especially for close-up shots.
- Realistic Surfaces: Tessellation can be applied to surfaces like water, rocks, or organic shapes to create more natural and detailed representations.

3. Geometry Stage:

The geometry shader is a programmable stage in modern graphics pipelines that operates on entire primitives (points, lines, triangles) rather than individual vertices. It's positioned between the vertex shader and the rasterization stage, allowing for the creation, deletion, or modification of primitives, which can significantly affect the geometry being rendered.

Key Aspects of Geometry Shader:

Operates on Primitives:

- Geometry shaders process entire primitives, which means they have access to multiple vertices that form a primitive.
- They can generate new primitives, discard existing ones, or transform/prune geometry.

Extends Primitives:

- Geometry shaders can augment existing primitives by creating additional vertices, lines, or triangles.
- This feature allows for tessellation (increasing geometric detail) or creating complex shapes from simpler primitives.

Dynamic Geometry Modification:

- They offer the ability to modify geometry dynamically during rendering, enabling procedural geometry generation or simplification.

Point, Line, and Triangle Processing:

- Geometry shaders can handle various input primitive types—points, lines, triangles—and generate different output primitive types.

Access to Adjacent Primitives:

- Geometry shaders have access to neighboring vertices and primitives, enabling operations that involve adjacent geometry.

Programmable and Customizable:

- Geometry shaders are programmable like other shader stages, allowing developers to write custom code using languages like GLSL or HLSL.
- This programmability allows for a wide range of effects and transformations.

Workflow and Operations:

Input Primitives:

- Geometry shaders receive input primitives generated by the vertex shader.
- These primitives can be points, lines, or triangles.

Geometry Processing:

- The shader code defines operations on these primitives—creating new vertices, modifying positions, or discarding primitives based on defined conditions.

Primitive Generation:

- Geometry shaders can emit new primitives based on the processed input.
- They can create additional vertices, lines, or triangles, altering the geometry stream.

Primitive Deletion or Modification:

- They have the capability to discard or modify primitives based on specified conditions, reducing or altering the geometry before rasterization.

Output:

- The output of the geometry shader includes the generated or modified primitives.
- These processed primitives move to the subsequent stages like rasterization and fragment shading.

Use Cases:

Tessellation and Detail Enhancement:

- Geometry shaders can augment geometry, adding extra vertices to enhance detail in real-time.
- For instance, they can turn simple geometry into more complex shapes or subdivide surfaces for increased detail.

Particle Systems:

- They can generate multiple vertices to simulate particle effects like smoke, fire, or sparks.

Procedural Generation:

- Geometry shaders facilitate the creation of procedural content by generating complex geometry based on simple primitives.

Silhouette Detection and Outline Rendering:

- They can identify silhouette edges and render object outlines by generating additional geometry around objects.

4. Rasterization Stage:

Rasterization is a crucial step in the graphics rendering pipeline where geometric primitives (such as triangles, lines, or points) are converted into pixels on the screen. This process determines which pixels should be filled or covered by the primitives to produce the final image that will be displayed on the screen.

Steps in Rasterization:

Clipping:

- Clipping ensures that only the portions of primitives that lie within the view frustum (the visible region) are considered for rendering.
- Primitives outside this region are discarded to optimize rendering performance.

Projection:

- Vertices of the primitives are transformed from 3D world space into 2D screen space.
- This transformation includes applying perspective projection to create the illusion of depth on a 2D screen.

Rasterization:

- After projection, the rasterizer takes the transformed primitives and converts them into fragments or pixels.
- Fragments represent the coverage areas of the primitives on the screen, determining which pixels are affected by the primitive.

Interpolation:

- Attributes such as color, texture coordinates, normals, or other vertex attributes are interpolated across the fragments within the primitive.
- This interpolation ensures smooth transitions between pixels, providing a realistic appearance to the rendered object.

Fragment Processing:

- Fragments generated by rasterization undergo further processing in subsequent stages, primarily in pixel shaders or fragment shaders.
- Pixel shaders compute final pixel colors, applying lighting, texture mapping, material properties, and other effects.

Detailed Example of Triangle Rasterization:

Consider rendering a simple 2D triangle on the screen:

Vertex Processing:

- Three vertices of the triangle in 3D space are defined, each with its position (x, y, z) and associated attributes (color, texture coordinates, etc.).
- These vertices undergo transformations (like translation, rotation, projection) in the vertex shader to bring them into the 2D screen space.

Clipping:

- Clipping determines which parts of the triangle lie within the visible region of the screen.
- Portions of the triangle outside the screen boundaries are discarded.

Projection and Rasterization:

- The transformed vertices of the triangle are passed to the rasterizer.
- The rasterizer calculates the fragments (pixels) covered by the triangle on the screen.
- It determines which pixels the triangle covers, generating fragments within the triangle's area.

Interpolation:

- For each fragment generated, attributes like color or texture coordinates are interpolated across the fragment.
- This interpolation ensures smooth color transitions across the triangle's surface.

Fragment Processing:

- Fragments are passed to the pixel shader for further processing.
- Pixel shader computes the final color for each fragment based on lighting, textures, or other effects applied to the triangle.

Display Output:

- The final computed pixel colors are stored in the frame buffer, and the resulting image is displayed on the screen.

5. Pixel Operations:

A pixel shader, also known as a fragment shader in some contexts, is a programmable stage in the graphics pipeline responsible for computing the final color and other attributes of individual pixels. It operates on fragments generated by the rasterization process and is crucial in determining the visual appearance of rendered objects.

Key Aspects of Pixel Shader:

Operates on Fragments:

- Pixel shaders process individual fragments generated by rasterization.
- Each fragment typically represents a pixel on the screen or a portion of it covered by geometry.

Customizable and Programmable:

- Pixel shaders are programmable, allowing developers to write custom shader code using languages like GLSL (OpenGL Shading Language) or HLSL (High-Level Shading Language).
- This programmability enables a wide range of visual effects and computations on a per-pixel basis.

Final Color Calculation:

- The primary task of the pixel shader is to calculate the final color of each fragment.
- It considers various factors like lighting, textures, materials, shadows, and other effects applied to the surface.

Additional Fragment Attributes:

- Apart from color, pixel shaders can compute other attributes like depth, transparency, specular highlights, reflections, or any other information needed for rendering.

Post-Processing Effects:

- Pixel shaders are often used for post-processing effects like blur, distortion, bloom, tone mapping, and more, enhancing the overall visual quality of the rendered scene.

Per-Pixel Computations:

- Each pixel can undergo unique computations within the pixel shader, allowing for complex and detailed rendering of surfaces.

Workflow and Operations:

Input Fragments:

- Fragments generated by the rasterization process (such as triangles, lines, or points) serve as input to the pixel shader.
- These fragments contain interpolated attributes like color, texture coordinates, normals, etc., calculated during rasterization.

Per-Pixel Computations:

- The pixel shader computes the final color and additional attributes for each fragment based on its properties and the defined shader logic.
- It performs operations like texture sampling, lighting calculations, shading models, and other effects.

Texture Sampling:

- Pixel shaders often sample textures to apply detailed surface information like color, patterns, or bump maps, enhancing the appearance of surfaces.

Lighting Calculations:

- The shader computes lighting effects (ambient, diffuse, specular, etc.) based on surface normal, light positions, and material properties.

Final Output:

- The calculated color and other attributes for each fragment are sent to the frame buffer, contributing to the final rendered image.

Use Cases:

Texturing and Material Effects:

- Pixel shaders handle texture mapping and apply material properties to surfaces, making them look realistic with detailed textures and visual effects.

Lighting Models:

- They compute lighting effects, creating realistic simulations of light interaction with surfaces, including reflections and shadows.

Special Effects:

- Pixel shaders are used to create a variety of visual effects like bloom, motion blur, depth of field, and many others, enhancing the visual appeal of rendered scenes.

Post-Processing:

- They execute post-processing effects to improve the final image quality, such as applying filters or adjusting color tones.

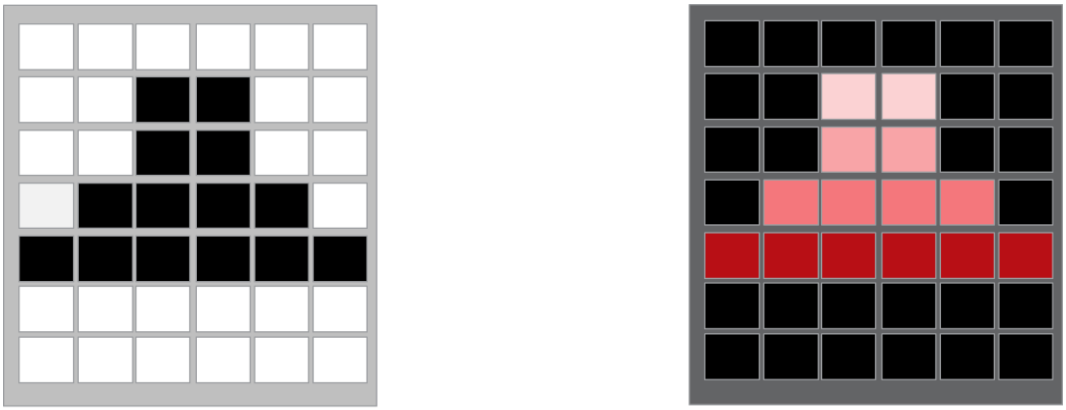

Fig 1.3 Pixel shader diagram

a. Output merger:

- The output merger stage in the GPU pipeline is the final phase before the rendered image is displayed on the screen. It combines all the fragments produced by the previous stages and determines the final color values for each pixel that will be written to the frame buffer. This stage involves operations like depth testing, blending, and writing the final pixel values.

Key Components and Operations of Output Merger:

Depth Testing:

- Fragments have depth values indicating their distance from the camera.
- Depth testing compares the depth values of fragments to determine which fragments are closer to the camera and should be displayed.
- Fragments that pass the depth test replace existing fragments in the depth buffer if they are closer.

Stencil Testing (Optional):

- Stencil testing allows fragments to pass or fail based on comparisons with a stencil buffer, enabling more complex masking and effects.

Blending:

- Fragments that pass depth and stencil tests may occupy the same pixel on the screen.

- Blending operations combine these fragments' color values to calculate the final color of the pixel that will be written to the frame buffer.
- Blending can involve operations like alpha blending, which combines colors based on transparency or other blending modes specified by the developer.

Output to Frame Buffer:

- Once depth testing, stencil testing, and blending operations are performed, the final fragment colors are written to the frame buffer.
- The frame buffer stores the pixel values of the rendered image that will be displayed on the screen.

Workflow and Operations in Output Merger Stage:

Depth Testing:

- Fragments' depth values are compared to the depth buffer's stored depth values.
- Fragments passing the depth test update the depth buffer with their depth values.

Stencil Testing (if enabled):

- Fragments undergo stencil testing, which determines whether they should be further processed based on comparisons with the stencil buffer.

Blending:

- Fragments that pass depth and stencil tests enter the blending stage.
- Blending operations compute the final color of each fragment based on its interaction with other fragments occupying the same pixel.

Writing to Frame Buffer:

- The computed final colors from blending are written to the frame buffer at the corresponding pixel locations.

Use Cases and Importance:

- Rendering Quality: The output merger stage ensures accurate rendering by handling depth testing, ensuring correct rendering order, and producing the final pixel colors.

- Transparency and Effects: Blending operations allow for transparency effects, softening edges, or implementing various visual effects.

- Visual Enhancements: Depth testing maintains proper occlusion, and stencil testing enables complex masking or effects.

- Performance Optimization: Depth testing helps in reducing unnecessary rendering of obscured objects, optimizing performance.

1.3 DRM (Direct Rendering Manager):

DRM (Direct Rendering Manager) in Linux is a subsystem within the Linux kernel responsible for managing graphics hardware and providing support for various display devices. It facilitates interactions between the kernel and graphics hardware, enabling efficient rendering and display of graphical content.

Key Components and Functionalities of DRM in Linux:

Graphics Drivers:

- DRM comprises graphics drivers specific to various GPU vendors (Intel, AMD, NVIDIA) and their specific hardware families.
- These drivers communicate with the hardware and expose interfaces for user-space applications to access GPU functionalities.

Kernel Mode Setting (KMS):

- KMS is a feature of DRM that sets up display resolutions, refresh rates, and other display-related settings within the kernel space.
- It initializes displays and graphics hardware during system boot-up, ensuring proper configuration before user-space components start.

Direct Rendering Infrastructure (DRI):

- DRI is a framework within DRM that enables direct access to the GPU hardware for user-space applications.
- It includes components like the DRM kernel module, the DRI2/DRI3 interface for user-space, and hardware-specific Direct Rendering Contexts (DRC) for context management.

Memory Management:

- DRM manages memory allocation, mapping, and sharing between the CPU and GPU.
- It provides APIs for allocating memory buffers, managing buffer objects (BOs), and handling memory synchronization between different processes or contexts.

Mode setting and Display Configuration:

- DRM manages mode setting operations, such as configuring display resolutions, framebuffers, and managing multiple displays (monitors).

- It supports features like hot-plugging, dynamic resolution changes, and extended display configurations.

Security and Access Control:

- DRM includes mechanisms for access control and security.
- It ensures that only authorized processes have access to the GPU and its resources, preventing unauthorized access or interference.

DRM Operation Flow:

Initialization and Hardware Probing:

- During system boot, the kernel loads relevant DRM modules based on detected hardware.
- DRM probes and initializes the connected GPUs and their associated devices.

Kernel Mode Setting (KMS):

- KMS is activated to configure display settings like resolution, refresh rate, etc., within the kernel space.
- It prepares the display hardware for user-space applications to utilize.

Direct Rendering:

- User-space libraries and APIs interact with the DRM drivers to perform graphics operations, such as rendering, displaying, and managing graphics resources.
- Direct Rendering capabilities enable applications to access GPU hardware directly for rendering, enhancing performance.

Resource Management:

- DRM manages GPU resources, including memory allocation and sharing, synchronization, and access control.
- It ensures efficient utilization of the GPU resources among different applications.

Importance and Functionality:

- DRM in the Linux kernel serves as a crucial interface bridging the gap between user-space applications and GPU hardware.
- It provides a standardized and efficient means for applications to utilize graphics capabilities in the Linux environment.
- DRM supports a wide range of graphics hardware and functionalities, enabling smooth rendering, display configurations, and access control for GPU resources.

2. Work Flow:

- The tool serves as a crucial aid in debugging GPU-related issues by streamlining the complexity of the debugging process.
- It's crafted to analyze and identify faults occurring within the GPU, including page faults, shader faults, and other potential issues.
- Operating within the Linux environment, the tool leverages a sysfs node to establish communication with the kernel, allowing it to extract essential GPU information efficiently.
- The workflow involves the creation of a directory within the sysfs node, named after the tool. Within this directory, a binary file is generated to encapsulate crucial GPU data.
- However, to make this data understandable and usable for analysis, a Python script comes into play. This script processes the information stored in the binary file, extracting, segregating, and formatting it into a human-readable form.

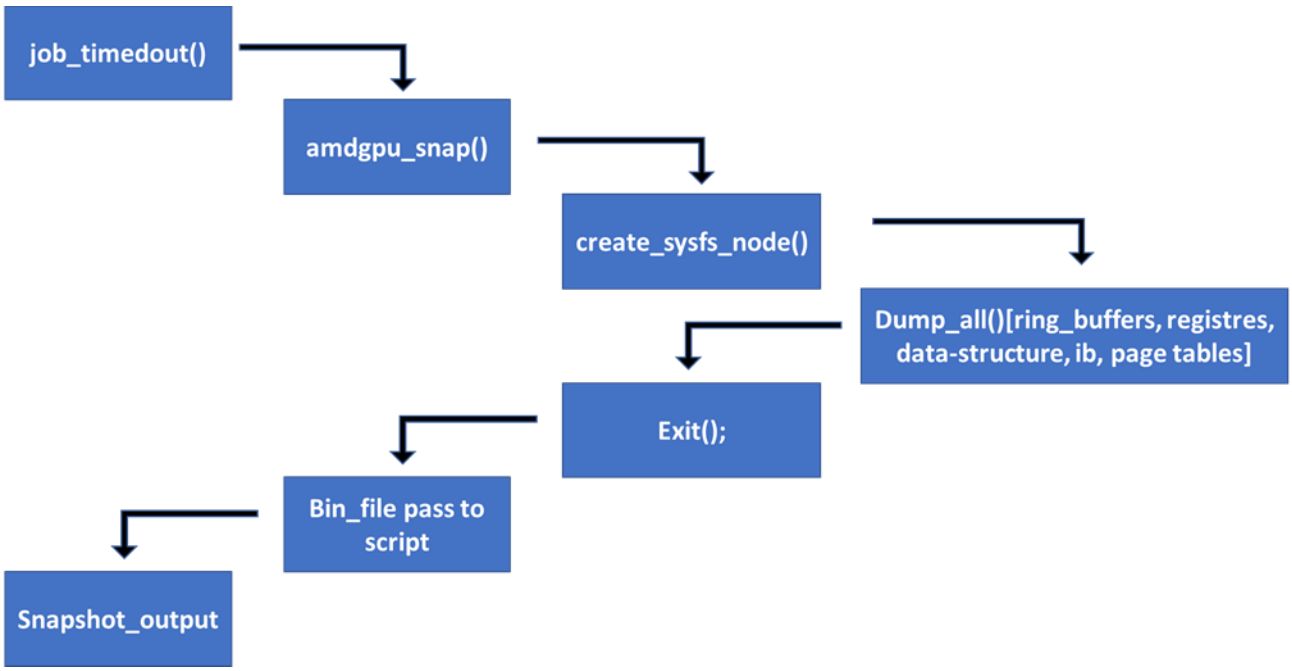

Fig 2.1 code flow diagram

The tool's workflow encompasses:

- Retrieving ring buffer values and subsequent post-processing on the Python side.
- Gathering and processing register values, also handled by the Python script.
- Collecting data structure values and performing post-processing using Python.
- Capturing page table values and subsequent processing through Python scripting.
- An essential aspect of this tool's design is its adaptability across different GPU versions. It's engineered to be flexible enough to work seamlessly with various GPU iterations, ensuring its usability and efficacy across diverse hardware configurations.

Tool working on this Cases:

In the context of job execution, the occurrence of a "Job-timedout" event signals a crucial aspect of job management. Each job is assigned a designated threshold limit for execution, typically set at 10 seconds. If a job exceeds this predefined time limit, indicating a potential performance bottleneck or issue, the "job-timeout" function is invoked. This function serves a critical role in maintaining system integrity by promptly removing the stalled job from execution and initiating necessary corrective measures.

Upon triggering the "job-timeout" function, the system takes proactive steps to mitigate the impact of the stalled job. One such measure involves the addition of a new job to the ring buffers, ensuring continuity in job processing and preventing system deadlock or stagnation. This dynamic adjustment mechanism optimizes resource utilization and fosters efficient task management within the system architecture.

Another vital aspect of system operation pertains to "Page fault" occurrences, which arise when jobs encounter memory-related challenges during execution. Each job is allocated a specific memory space for its execution requirements. However, if the available memory proves insufficient to accommodate all active jobs concurrently, the system resorts to swapping out certain job data from primary memory (VRAM) to secondary memory sources such as system RAM or even the hard drive.

This swapping process, while essential for managing memory resources, introduces the risk of encountering page faults. A page fault occurs when a process attempts to access a memory page that has been swapped out and is not currently resident in the primary memory (VRAM). Consequently, the system responds by triggering the "job-timeout" function, similar to the aforementioned scenario of exceeding the execution time limit.

In essence, the interplay between job timeout events and page faults underscores the dynamic nature of system resource management and the intricate balance required to ensure optimal performance and stability. By promptly addressing stalled jobs and managing memory resources effectively, the system can uphold responsiveness and reliability, thereby enhancing overall operational efficiency and user experience.

3. Linux Device driver overview:

1. Introduction of the driver:

A Linux device driver is a software component that facilitates communication between a hardware device and the Linux kernel. It serves as an intermediary layer that enables the operating system to interact with various hardware peripherals, such as input/output devices, storage devices, network interfaces, and more.

Types of Linux Device Drivers:

Character Device Drivers:

Character device drivers handle data transmission in a stream of bytes, typically representing individual characters or small data packets.
Examples include serial ports, keyboards, mice, and sound cards.
These drivers interact with the device through device nodes in the /dev directory and implement functions like open(), read(), write(), and close().

Block Device Drivers:

Block device drivers manage data in fixed-size blocks or sectors, enabling random access to data stored on the device.
Common examples include hard disk drives (HDDs), solid-state drives (SSDs), USB drives, and CD/DVD drives.
These drivers implement functions for reading and writing blocks of data, along with features like caching and buffering for performance optimization.

Network Device Drivers:

Network device drivers facilitate communication between the Linux kernel and network interface cards (NICs) or network adapters.
They handle tasks such as packet transmission and reception, interface configuration, and network protocol processing.
Examples include Ethernet drivers, Wi-Fi drivers, and Bluetooth drivers.

File System Drivers:

File system drivers provide support for various file systems, allowing the operating system to read from and write to storage devices formatted with different file system types.
Examples include drivers for ext4, NTFS, FAT32, and other file systems.
These drivers enable the mounting, accessing, and management of files and directories stored on storage media.

Miscellaneous Drivers:

Miscellaneous drivers cover a wide range of device types that do not fit neatly into the above categories.
Examples include device drivers for USB devices, input devices like touchscreens or joysticks, graphics cards, and miscellaneous hardware sensors.
These drivers often serve specialized or niche hardware functionalities.
Linux device drivers play a crucial role in enabling the Linux operating system to support a diverse array of hardware devices, contributing to its versatility, scalability, and widespread adoption across various computing platforms.

2. IOCTL calls:

IOCTL stands for "Input/Output Control". It's a system call in many operating systems that enables user-space programs to communicate directly with device drivers or the kernel. These calls are typically used for device-specific control operations that can't be performed through regular file operations like read and write.

Here's how it generally works:

User Program: Requests a specific operation on a device.
Device Driver: Receives the request from the user program.
Kernel: Processes the request and performs the necessary action.
Device: The action is carried out by the device.
For example, a user program might want to change the baud rate of a serial port, query the status of a hardware device, or perform other device-specific operations. These tasks are typically accomplished using IOCTL calls.

The specific IOCTL commands and their parameters are usually defined in the header files associated with the device drivers or in the operating system's documentation. They vary depending on the device and the operating system.

User Program Request:

The process begins with a user program making a request for a specific operation on a device. This request could be initiated through a system call or a library function provided by the operating system.
System Call Invocation:

When the user program invokes a system call related to device I/O control (such as ioctl() in Unix-like systems), the operating system's kernel takes control.
System Call Processing:

The kernel examines the system call parameters, including the file descriptor associated with the device and the specific control code (IOCTL command) passed by the user program.

File Descriptor Lookup:

The kernel looks up the file descriptor associated with the device in the process's file descriptor table. This table maintains information about all open files and their associated file descriptors.
File Structure and Device Driver Association:

Once the file descriptor is located, the kernel accesses the corresponding file structure. This structure contains information about the file, including a reference to the device driver responsible for handling I/O operations on that file.
Device Driver Interaction:

The kernel passes the IOCTL command and any associated data to the device driver's IOCTL handler function. This function is implemented by the device driver and is responsible for interpreting and executing the requested operation.
Device-specific Processing:

Within the device driver's IOCTL handler function, device-specific processing takes place. Depending on the IOCTL command received, the driver may perform various tasks such as configuring device parameters, retrieving device status, or initiating specific actions on the hardware.
Completion and Return:

Once the IOCTL operation is completed by the device driver, control returns to the kernel. If the IOCTL call was synchronous, the kernel may also return relevant data or status information to the user program.
User Program Response:

The user program receives the result of the IOCTL call and continues its execution based on the returned data or status. If necessary, the program may perform additional actions or issue further IOCTL calls.
Overall, the IOCTL mechanism provides a flexible and standardized way for user programs to communicate with device drivers and perform device-specific operations, enabling efficient management and control of hardware resources within the operating system.

3. Sysfs node:

In Linux, sysfs is a virtual filesystem that provides a view of the kernel's device model. It presents information about various kernel subsystems, devices, and their attributes in a hierarchical structure, accessible through the /sys directory. Each entry in sysfs represents a specific kernel object or attribute, known as a "node".

Here's a detailed explanation of what a sysfs node represents and how it functions:

Representation of Kernel Objects:

Each node in sysfs represents a kernel object, such as a device, bus, driver, or subsystem. For example, a device node might represent a physical device like a USB controller, while a driver node could represent a loaded device driver module.
Hierarchical Structure:

Sysfs organizes nodes in a hierarchical structure that mirrors the internal organization of the Linux kernel. This structure helps in navigating and locating specific kernel objects and their attributes.
Attributes and Properties:

Nodes in sysfs expose various attributes and properties of the associated kernel objects. These attributes can include information such as device status, configuration parameters, hardware capabilities, and more.
Read-Only Filesystem:

Sysfs is typically a read-only filesystem. While it's possible to write data to some nodes for certain operations (like changing device settings), the primary purpose of sysfs is to provide a view of kernel state rather than to allow direct manipulation of that state.
Dynamic Nature:

Sysfs nodes and their attributes are dynamic and can change based on the state of the kernel and its associated devices. Nodes may appear or disappear as devices are added or removed, drivers are loaded or unloaded, or as the system configuration changes.
User-Space Interaction:

Sysfs is primarily designed for interaction with user-space programs and tools. It allows user-space utilities to query and configure kernel objects and their attributes, facilitating tasks such as device discovery, monitoring, configuration, and troubleshooting.
Standardization and Documentation:

Sysfs follows a standardized layout and naming convention, making it easier for users and developers to understand and interact with. Additionally, documentation for sysfs nodes and their attributes is often provided in the kernel source code, helping users understand the meaning and usage of different attributes.
Integration with Other Subsystems:

Sysfs is closely integrated with other kernel subsystems such as udev, which is responsible for device management and dynamic device node creation. udev uses information from sysfs to create and manage device nodes in the /dev directory.
Overall, sysfs serves as a vital interface between the Linux kernel and user-space applications, providing visibility into the kernel's device model and enabling efficient management and interaction with kernel objects and their attributes.
initialization:

During system boot, the sysfs filesystem is initialized by the kernel. This involves setting up the directory structure and creating initial nodes to represent various kernel objects and subsystems.

Representation of Kernel Objects:

Sysfs represents kernel objects such as devices, buses, drivers, and other subsystems as directories and files within its filesystem hierarchy. Each object is represented by a directory, while its attributes are represented as files within that directory.

Dynamic Population:

As the system boots up and devices are detected, drivers are loaded, and subsystems are initialized, sysfs dynamically populates itself with nodes representing these objects and their attributes. This dynamic nature allows sysfs to reflect the current state of the system.

Hierarchical Structure:

Sysfs organizes objects in a hierarchical structure that reflects the relationships and dependencies between different kernel components. For example, devices are grouped under their corresponding buses, and drivers are associated with their respective devices.

Attributes and Properties:

Each node in sysfs represents a kernel object or attribute associated with a kernel object. Attributes are represented as files within the directories, and their contents provide information or configuration options related to the corresponding kernel object.

Read-Only Access:

Sysfs is primarily a read-only filesystem. While some attributes may support write access for configuration purposes, the filesystem as a whole is designed to provide visibility into the kernel state rather than allowing direct manipulation of that state.

User-Space Interaction:

Sysfs is designed for interaction with user-space programs and utilities. User-space applications and tools can read from sysfs to obtain information about system devices, drivers, and subsystems, and they can sometimes write to sysfs to configure certain attributes.

Integration with User-Space Tools:

Sysfs is closely integrated with user-space tools such as udev, which is responsible for device management and dynamic device node creation. udev uses information from sysfs to create and manage device nodes in the /dev directory based on the detected hardware.

Debugging and Troubleshooting:

Sysfs is a valuable tool for debugging and troubleshooting system configuration and hardware-related issues. System administrators and developers can use sysfs to inspect device properties, driver bindings, and other kernel-related information.
Overall, sysfs serves as a critical interface between the Linux kernel and user-space applications, providing a structured and dynamic view of the system's device model and kernel objects. Its dynamic nature, hierarchical organization, and integration with user-space tools make it an essential component of the Linux operating system.

SysFS in Linux Device Driver

There are several steps to creating and using sysfs.

Create a directory in /sys

Create Sysfs file

1. Create a directory in /sys

We can use this function (kobject_create_and_add) to create a directory.

struct kobject * kobject_create_and_add (const char * name, struct kobject * parent);

Were,

<name> – the name for the kobject

<parent> – the parent kobject of this kobject, if any.

If you pass kernel_kobj to the second argument, it will create the directory under /sys/kernel/. If you pass firmware_kobj to the second argument, it will create the directory under /sys/firmware/. If you pass fs_kobj to the second argument, it will create the directory under /sys/fs/. If you pass NULL to the second argument, it will create the directory under /sys/.

This function creates a kobject structure dynamically and registers it with sysfs. If the kobject was not able to be created, NULL will be returned.

When you are finished with this structure, call kobject_put and the structure will be dynamically freed when it is no longer being used.
Example

```
struct kobject *kobj_ref;

/*Creating a directory in /sys/kernel/ */
kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj); //sys/kernel/etx_sysfs
```

/*Freeing Kobj*/
kobject_put(kobj_ref);

a. Create Sysfs file

Using the above function we will create a directory in /sys. Now we need to create a sysfs file, which is used to interact user space with kernel space through sysfs. So we can create the sysfs file using sysfs attributes.

Attributes are represented as regular files in sysfs with one value per file. There are loads of helper functions that can be used to create the kobject attributes. They can be found in the header file sysfs.h

Create attribute:

Kobj_attribute is defined as,

```
struct kobj_attribute {
 struct attribute attr;
 ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr, char *buf);
 ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t
count);
};
```

Were,

attr – the attribute representing the file to be created,

show – the pointer to the function that will be called when the file is read in sysfs,

store – the pointer to the function which will be called when the file is written in sysfs.

We can create an attribute using __ATTR macro.

__ATTR(name, permission, show_ptr, store_ptr);

Store and Show functions
Then we need to write show and store functions.

```
ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr, char *buf);
ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t
count);
```

The store function will be called whenever we are writing something to the sysfs attribute. See the example.

The show function will be called whenever we are reading the sysfs attribute. See the example.

Create sysfs file:

To create a single file attribute we are going to use ‘sysfs_create_file’.

int sysfs_create_file (struct kobject * kobj, const struct attribute * attr);

Were,

kobj – object we’re creating for.

attr – attribute descriptor.

One can use another function ‘ sysfs_create_group ‘ to create a group of attributes.

Once you have done with the sysfs file, you should delete this file using sysfs_remove_file

void sysfs_remove_file (struct kobject * kobj, const struct attribute * attr);

Were,

kobj – object we’re creating for.

attr – attribute descriptor.

4. Page tables:

A "page" refers to a segment within the physical memory space. For instance, in a 1GB physical memory setup, it comprises numerous small chunks, each sized at 4096. Essentially, each page encompasses 4096 units, and this size remains consistent across all pages.

This topic delves into the intricacies of the page table walk, elucidating how the traversal occurs from Page Directory Entries (PDEs) to Page Table Entries (PTEs). In the realm of address spaces, two prominent types exist: virtual and physical. Virtual memory serves as an extension of the system's physical memory, offering a broader scope for memory utilization. Each process possesses its own page table, segregating its address space into user and kernel spaces.

The kernel space remains uniform across processes, while each process maintains its distinct user space. Page tables manifest in various forms, including 1-level, 2-level, 3-level, and 4-level page tables.

The terminal tier of the page table hierarchy is the Page Table Entry (PTE), housing the physical address pertinent to the process. Positioned above the PTEs are the Page Directory Entries (PDEs), which encompass both PTEs and additional PDEs.

A single PDE can encapsulate either 8 PDEs or 8 PTEs. Both PTEs and PDEs consist of 512 pages of entry, effectively accommodating 512 addresses. The diagram illustrates the page table walk process. Currently, the address length is 39 bits, with PDE-2 containing 512 entries. For each entry, validation is crucial. If the valid bit is set (1), traversal proceeds to PDE-1.

There, another 512 addresses are traversed, with each entry's validity assessed based on the offset. Upon encountering a valid address, progression continues to the PDE-0 level page. This iterative process persists until reaching the PTE, wherein the physical page address of the process is derived.

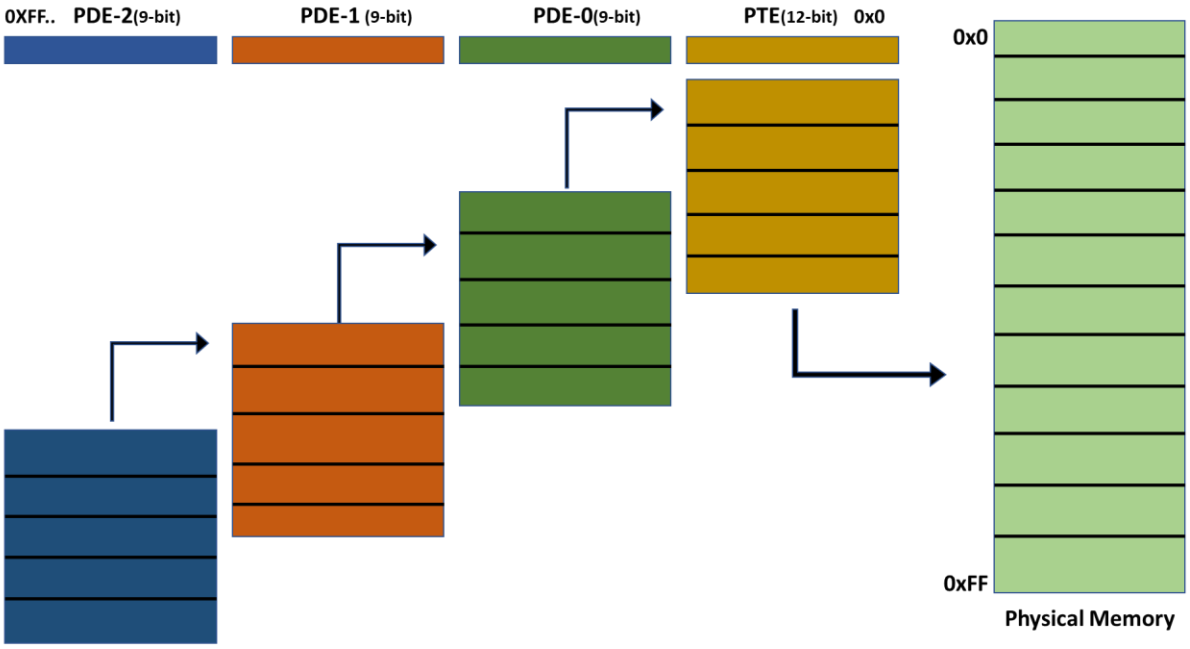

Fig 3.1-level page table

A 4-level page table walk is a hierarchical process used by modern operating systems, including Linux, to translate virtual addresses used by processes into physical memory addresses. This method allows for efficient memory management and facilitates the implementation of virtual memory systems. Let's delve into the steps involved in traversing a 4-level page table to reach the physical address of a process:

Virtual Address Breakdown:

The virtual address used by a process typically consists of several components, including the page directory pointer (PDP), the page directory (PD), the page table (PT), and the page offset. In a 4-level page table system, the virtual address is divided into four parts, with each part indicating an index into a specific level of the page table hierarchy.

PDP Level (Level 4):

The virtual address's most significant bits are used as an index into the Page Directory Pointer (PDP) level.
The PDP is a single-level page table that contains entries pointing to the Page Directories (PD) of each process.
Using the index obtained from the virtual address, the system locates the appropriate entry in the PDP to obtain the address of the corresponding Page Directory (PD).

PD Level (Level 3):

The address obtained from the PDP points to the Page Directory (PD) level.
The PD is another single-level page table containing entries that point to Page Tables (PT) or to a 2 MB or 1 GB page.
Using the index extracted from the virtual address, the system accesses the corresponding entry in the PD to retrieve the address of the Page Table (PT) or the page frame directly if using large pages.

PT Level (Level 2):

If the entry in the PD points to a Page Table (PT), the system proceeds to the Page Table level.
The PT is a multi-level page table containing entries pointing to the physical page frames in memory.
Using the index from the virtual address, the system accesses the appropriate entry in the PT to obtain the address of the page frame.

Page Offset:

The remaining bits of the virtual address represent the offset within the page frame, indicating the specific byte or word within the physical page frame.

Physical Address Construction:

Once the page frame address and the page offset are obtained, they are combined to form the physical address of the desired memory location.

Accessing the Data:

With the physical address determined, the system can access the data stored at that memory location, allowing the process to read from or write to the corresponding physical memory.
By following this hierarchical traversal process through the four levels of the page table, the operating system efficiently translates virtual addresses into physical memory addresses, enabling seamless memory management and access for processes.

5. Conclusion:

- This tool, through its systematic approach of data collection, storage, and Python-based processing, significantly streamlines the debugging process for GPU-related faults, making GPU troubleshooting more efficient and reliable in Linux environments.

6. Reference

1. Kernel_doc: https://www.kernel.org/doc/gorman/html/understand/understand004.html
2. Ldd (linux device driver): https://lwn.net/Kernel/LDD3/
3. Bootlin linux kernel doc: https://bootlin.com/docs/
4. Bootlin_linux_kernel_source_code: https://elixir.bootlin.com/linux/latest/source/kernel
5. GPU pipeline: https://www.youtube.com/watch?v=Y2KG_4OxDBg
6. Open GL doc: https://www.opengl.org/Documentation/Specs.html
7. Vulkan doc: https://docs.vulkan.org/spec/latest/index.html
8. Sysfs_doc: https://www.kernel.org/doc/html/next/filesystems/sysfs.html
9. Kernel build doc: https://kernelnewbies.org/KernelBuild
10. Kernel_API_doc: https://www.kernel.org/doc/html/latest/core-api/kernel-api.html?highlight=alloc_chrdev_region
11. DRM_memory_management_doc: https://www.kernel.org/doc/html/v4.20/gpu/drm-mm.html
12. Dmesg command doc: https://www.tecmint.com/dmesg-commands/
13. Windows drivers doc: https://learn.microsoft.com/en-us/windows-hardware/drivers