

Design and FPGA-Based Implementation of a Lightweight Graphics Processing Unit for Real-Time Gaming Applications

Sreenivasulu K N^{1*}, Mokshith V², Manish S³, Harisha M⁴, Guru Kiran⁵, Mamatha Y N⁶

¹ Dept. of Electronics and Communication Engineering, Sapthagiri University, Karnataka, India

² Dept. of Electronics and Communication Engineering, Nagarjuna College of Engineering and Technology, Karnataka, India

³ Dept. of Electronics and Communication Engineering, Nagarjuna College of Engineering and Technology, Karnataka, India

⁴ Dept. of Electronics and Communication Engineering, Nagarjuna College of Engineering and Technology, Karnataka, India

⁵ Dept. of Electronics and Communication Engineering, Nagarjuna College of Engineering and Technology, Karnataka, India

⁶ Dept. of Electronics and Communication Engineering, Nagarjuna College of Engineering and Technology, Karnataka, India

⁶ Dept. of Electronics and Communication Engineering, Sapthagiri University, Karnataka, India

*Corresponding Author: sreenivasulukn@snps.edu.in

Abstract

The plan and advancement of a straightforward PlayStation GPU Design Handling Unit includes joining principal standards of computer illustrations and equipment plan to make a productive and cost-effective illustration arrangements. This paper considers the present conceptual system for a GPU custom fitted for gaming comforts, centering on basic graphical preparing errands such as rendering, surface mapping, shading, and rasterization. The plan emphasizes versatility, control proficiency, and real-time rendering capabilities appropriate for casual gaming environments. Key components incorporate a rearranged engineering with a streamlined instruction set, a unified memory framework to play down idleness, and fixed-function units for dealing with common errands like geometry preparing and pixel shading. The GPU's plan too coordinating hardware accelerated highlights like anti-aliasing and real-time lighting impacts, empowering outwardly engaging design without overpowering the hardware's computational capacity. Through hypothetical examination, the proposed GPU illustrates the ability to adjust execution, control utilization, and equipment complexity. The considered methodology concludes with experiences into optimizing GPU designs for gaming applications and examines potential upgrades for future emphasis. This plan serves as a establishment for understanding GPU plan standards and offers an outline for entry-level design equipment in gaming framework.

Keywords- PlayStation, Computer Graphics, Graphic Processing Unit (GPU), computational capacity

1. Introduction

Designing a simplified Play Station GPU aims to explore the fundamental and basics of typical GPU. It manipulates complex 3D design, running machine learning models, or conducting reenactments, GPUs are necessarily to numerous computing frameworks. Their parallel preparing control permits them to oversee complex

datasets and perform calculations at the same time, which is fundamental for real-time applications. In this report, we centred the concept on understanding and imitating key GPU operations, particularly visualization procedures can give profitable bits of knowledge into the GPU's operation, highlighting regions where optimizations can move forward execution times and in general effectiveness. By looking at these

operations, this report points to contribute to the progressing exertion of refining GPU utilization for more effective and proficient computing arrangements. The plan and improvement of a rearranged PlayStation GPU point to investigate the elemental standards and engineering basic advanced design handling units. This extend centers on making a fundamental GPU show custom fitted for a gaming comfort, highlighting basic components such as rendering pipelines, memory administration, and parallel handling. By diving into the centre functionalities of a GPU, the report gives bits of knowledge into how these units empower practical design, effective computation, and immersive gaming encounters. The streamlined plan equalizations complexity and usefulness, making it a down to earth think about for understanding GPU advancement and its application in gaming frameworks.

Objective

The objective of this consider is to pick up a more profound understanding of basic GPU operations such as rasterization and outline buffer dealing with. By reproducing and analyzing these operations, this report points to explore the execution times and execution measurements related with each errand. Furthermore, the ponder centers on visualizing execution information to pick up profitable bits of knowledge into GPU execution. This examination will empower us to recognize potential bottlenecks and optimize graphical and computational workflows, moving forward the effectiveness and viability of GPU operations. The essential objective of this extend, "Plan and Improvement of a Straightforward PlayStation GPU," is to form an effective, cost-effective Illustrations Handling Unit (GPU) custom-made for a fundamental PlayStation-like gaming comfort. The GPU will be planned to handle fundamental rendering errands such as 2D/3D design handling, surface mapping, and shader execution whereas keeping up moo control utilization and tall execution. The plan will center on optimizing computational assets to provide

smooth gameplay encounters for casual gaming stages, leveraging an understanding of cutting edge GPU designs and equipment components.

Additionally, the venture points to investigate and actualize a streamlined engineering that equalizations effortlessness with usefulness, making the GPU simple to coordinated into existing or future gaming supports. Key execution measurements such as rendering speed, graphical devotion, and framework integration will be considered all through the plan handle. By the conclusion of the extend, the objective is to have a completely working model of the PlayStation GPU that meets the fundamental specialized determinations for gaming applications whereas being adaptable for future advancements

2. Related Work

OpenGL is a Application programming interface for creating 2D and 3D Graphics which is one of the prominent tool in synthesizing Plays GPU with reference to (Segal et al., 2004) outlined for rendering 2D and 3D design. OpenGL gives a platform –independent interface that empowers designers to make high performance illustrations applications, the creators details emphasis its capacity to back a wide extend of design operations, counting geometric changes , shading , lighting

and surface mapping. This research paper majorly highlights some OpenGL principles:

Extensibility: Permitting equipment produces to present unused highlights without compromising compatibility. Portability: Advertising a well-defined state machine-based interface for developers, Simplicity Offering the user a much simplified way of State machine based interface [1].

D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. provides a comprehensive overview of the evolution and applications of General-Purpose Computing on Graphics Processing Units (GPGPU).The authors explore how GPUs,

originally designed for rendering graphics, have been increasingly

adapted for a wide range of non-graphics applications due to their highly parallel architecture and massive computational power. This research work mainly highlights major things such as GPU Architecture , Evolution of GPGPU , Programming model and its Application , Challenges and Future Directions. GPU Engineering: The work clarifies the building contrasts between CPUs and GPUs, emphasizing the GPU's plan for parallel handling and its appropriateness for data-parallel tasks. Evolution of GPGPU: It examines the verifiable movement of GPUs, from fixed-function pipelines to programmable shaders and the possible rise of general-purpose computing systems like CUDA and OpenCL. Programming Models: The creators highlight the challenges and headways in GPU programming models, centering on how instruments and libraries have streamlined the improvement of GPU accelerated applications. Applications: A wide run of application spaces are secured, counting logical re-enactments, flag handling, computational science, and real-time information investigation, where GPU computing has conveyed noteworthy execution gain. Challenges and Future Bearings: The paper addresses restrictions such as memory transmission capacity limitations, control productivity, and programming complexity. It moreover recommends future investigate ranges, counting moving forward GPU programmability and investigating cross breed CPU-GPU frameworks [2].

OpenGL ES 3.0 with reference to A . Munshi, D. Ginsburg, and D. Shreiner, OpenGL ES 3.0 Programming Guide, 2nd ed. Addison-Wesley. The work aims about the Programming Direct gives an in-depth investigation of OpenGL ES 3.0, a cross-platform API outlined for progressed rendering on inserted frameworks, counting versatile gadgets and gaming comforts. This moment version

presents key highlights and advancements in OpenGL ES 3.0, such as upgraded shader capabilities, surface groups, and execution optimizations. Points incorporate the programmable rendering pipeline, shader programming with GLSL, proficient buffer administration, and progressed strategies like instancing and rendering to surfaces it to examine best zones for optimizing design applications for execution and compatibility over differing equipment platforms. The creators adjust hypothetical establishments with common sense illustrations, counting code pieces and case ponders, to assist per users ace both the essentials and progressed strategies of OpenGL ES 3.0. This direct serves as an basic asset for developers aiming to tackle the complete potential of advanced implanted illustrations APIs [3].

Graphics Pipeline Architectures with reference to the work of "Survey of Graphics Pipeline Architectures" by Jensen and D. H. Jones which gives a comprehensive diagram of the advancement of design pipeline centering on their part in rendering practical and productive visual yield in computer illustrations. The creators examine the verifiable movement from fixed- function

pipelines to advanced programmable designs, highlighting key breakthroughs that empowered expanded adaptability and execution in rendering techniques. The overview digs into the different stages of the design pipeline, such as vertex preparing, rasterization, part shading, and show yield. It emphasizes how advancements in parallel handling and equipment optimizations have altogether made strides throughput and versatility. The creators too address challenges such as adjusting computational workloads, memory administration, and control effectiveness, which are basic in applications extending from gaming comforts to high-performance computing frameworks.

The paper concludes by distinguishing future patterns, counting cross breed rendering

methods and improved programmability, that are forming the another era of design pipeline designs[4]. Computer Architecture with reference to J. L. Hennessy and D. A. Patterson, Computer

Architecture: A Quantitative Approach, 5th ed. Morgan Kaufmann the work specifies the seminal work that gives an in-depth investigation of computer engineering with a centre on quantitative examination. The fifth version emphasizes the significance of execution, control, fetched productivity, and unwavering quality in cutting edge computer frameworks and introduces core concepts such as instruction set architecture. Performance metrics like execution time, throughput, and analysed efficiency are used in Quantitative Analysis. A focus on parallelism and energy efficiency are some of the new challenges in architecture discussed in emerging trends. Practical examples from contemporary systems illustrate theoretical concepts. The interaction between hardware and software features is analysed [5].

3. Methodology

GPU Operations Simulation and Performance Analysis: The methodology outlined for simulating GPU operations in this study involves the implementation of core processes such as rasterization, shading, and framebuffer management using Python. The goal is to reproduce the underlying mechanisms of GPU rendering operations, focusing on aspects such as perspective projection, barycentric coordinate-based rasterization, and the analysis of GPU operation performance. Below is a detailed analysis of each step, incorporating pseudocode and the potential for visualizing performance metrics.

1. Simulation of GPU Operations (Rasterization, Shading, and Framebuffer Writing) In the first step, the GPU operations that are typically executed on hardware are simulated in Python. This simulation focuses on core operations in the

rendering pipeline: rasterization, shading, and frame buffer writing.

Rasterization refers to the process of converting geometric data (such as 3D models) into a 2D image composed of pixels or fragments. This simulation involves creating a basic triangle rasterizer, where a 3D triangle is projected onto a 2D plane, and the corresponding pixels are identified and colored based on the triangle's surface properties. Shading involves calculating the color of each pixel, based on factors such as lighting, material properties, and surface normals. This step can be implemented using simplified versions of shading algorithms, like Phong shading or Gouraud shading, depending on the level of complexity required.

Finally, the framebuffer is implemented as a simple 2D array (or matrix) where each pixel's color value is stored. As rasterization and shading are performed, the computed pixel values are written into the framebuffer, simulating how GPUs store final images.

Pseudocode for Rasterization Simulation: python

```
def rasterize_triangle(vertices, screen):
    # Convert vertices from 3D to 2D using perspective projection
    projected_vertices = perspective_projection(vertices)

    # Find bounding box of the triangle
    min_x, min_y, max_x, max_y = bounding_box(projected_vertices)

    # Iterate through the bounding box and compute barycentric coordinates for each pixel
    for y in range(min_y, max_y):
        for x in range(min_x, max_x):
            if inside_triangle(x, y, projected_vertices):
                # Calculate the barycentric coordinates
                bary_coors = calculate_barycentric(x, y, projected_vertices)

                # Calculate color using shading (e.g., Phong or Gouraud)
                color = shading(bary_coors, vertices)

                # Write the color to the framebuffer
                screen[y][x] = color
```

Perspective Projection and Barycentric Coordinate-based: Rasterization

This step involves the implementation of perspective projection and barycentric coordinate-based rasterization, both of which are critical to the accuracy of the rendering

process. **Perspective Projection:** To simulate how objects are viewed in 3D space, we must project their 3D coordinates onto a 2D plane (screen). This is done using a perspective matrix, which mimics the way a camera sees the world, taking into account depth and the field of view. **Barycentric Coordinates:** Once the triangle is projected onto the 2D screen, we use barycentric coordinates to determine the relative position of each pixel within the triangle. Barycentric coordinates are essential because they allow us to interpolate values (like color or texture) across the surface of a triangle.

This method ensures that all pixels within the triangle are correctly identified and colored based on the triangle's geometric properties.

Pseudocode for Perspective Projection:

```
def perspective_projection(vertices):
    # Apply a perspective matrix to transform 3D vertices to 2D
    projection_matrix = create_projection_matrix(for_aspect_ratio, near, far)
    projected_vertices = []
    for vertex in vertices:
        x = (vertex[0] * projection_matrix[0][0] + vertex[1] * projection_matrix[0][1] + vertex[2] *
        projection_matrix[0][2]) / vertex[3]
        y = (vertex[0] * projection_matrix[1][0] + vertex[1] *
        projection_matrix[1][1] + vertex[2] * projection_matrix[1][2]) / vertex[3]
        projected_vertices.append([x, y])
    return projected_vertices
```

Pseudocode for Barycentric Coordinates Calculation:

```
def calculate_barycentric(x, y, triangle_vertices):
    # Given a pixel position (x, y) and the triangle's vertices v0, v1, v2 = triangle_vertices
    denom = (v1[1] - v2[1]) * (v0[0] - v2[0]) + (v2[0] - v1[0]) * (v0[1] - v2[1])
    a = ((v1[1] - v2[1]) * (x - v2[0]) + (v2[0] - v1[0]) * (y - v2[1])) / denom
    b = ((v2[1] - v0[1]) * (x - v2[0]) + (v0[0] - v2[0]) * (y - v2[1])) / denom
    c = 1 - a - b
    return (a, b, c)
```

Timing Analysis and Visualization of GPU Operation Performance The next step involves

performing timing analysis on the simulated GPU operations.

For example, we can measure the time taken for rasterization, the time for shading computations, and the total time to

complete the entire pipeline.

To visualize the performance, graphs can be generated using libraries like `matplotlib` or `seaborn` to plot the execution time against different factors such as scene complexity or the number of triangles being rendered. Additionally, heatmaps or histograms can be used to visualize the distribution of times for each pixel's processing.

Pseudocode for Timing Analysis:

```
# Measure the time taken for rasterization
start_time = time.time()
rasterize_triangle(vertices, screen) end_time
= time.time()

rasterization_time = end_time - start_time print("Rasterization
Time: (rasterization_time) seconds")

# Measure the total rendering time
total_start_time = time.time()
render_scene(scene, screen) total_end_time
= time.time()

total_render_time = total_end_time - total_start_time print("Total
Render Time: (total_render_time) seconds")
```

Visualization Example:

```
import matplotlib.pyplot as plt

# Plotting execution times
times = [rasterization_time, shading_time,
total_render_time]
labels = ['Rasterization', 'Shading', 'Total Rendering']
plt.bar(labels, times)
plt.xlabel('GPU Operation')
plt.ylabel('Time (seconds)')
plt.title('GPU Operation Performance')
plt.show()
```

IV. block diagram

Block Diagram and Flow Diagram of GPU Operations

The simulation and analysis of GPU operations involve a structured pipeline that converts 3D geometric data into a 2D image suitable for display. This chapter provides a visual representation of the process through a block

diagram and a flowchart, detailing the sequence of operations and their interrelations.

Block Diagram



Fig 4.1 Systematic View of A simple GPU

Input: The process begins with the input of 3D vertex coordinates and associated color data. These inputs define the geometry and appearance of the objects to be rendered.

Rasterizer Module: This module converts the input vertices into pixel fragments by mapping the 3D vertices to a 2D plane, ensuring accurate representation on the screen.

Shading Module: After rasterization, the shading module assigns colors to each pixel fragment. This involves calculating light interactions, material properties, and other visual effects.

Outline Buffer (Framebuffer): The final stage involves storing the processed pixel data in the framebuffer. This data is then used to render the final image on the display.

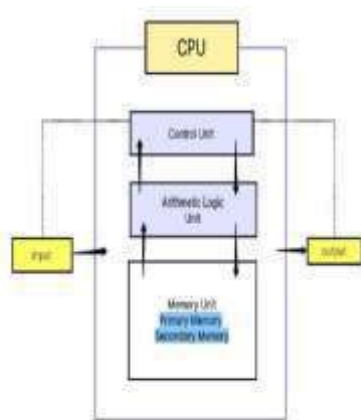


Fig 4.1.2 Flow Chart of GPU

The flowchart outlines the step-by-step process:

1. **Input 3D Vertex Data:** Load the geometric and color data of the objects to be rendered.
2. **Apply Perspective Projection:** Transform the 3D coordinates into a 2D space, considering depth and camera properties.
3. **Map to 2D Screen Space:** Convert the transformed vertices into corresponding screen positions.
4. **Perform Rasterization and Shading:** Rasterize the geometry into pixels and compute the color for each pixel using shading techniques.

Software stimulation

The simulation of GPU operations is implemented using Python, leveraging its flexibility and powerful libraries to model and analyze the rendering pipeline. This simulation serves as a conceptual representation of GPU functionality, highlighting the interaction between different hardware components.

5.1 Programming Environment

1. **Dialect:** Python is chosen for its ease of use and extensive support for scientific computing and visualization.
2. **NumPy:** Used for efficient numerical computations and matrix operations, essential for tasks like perspective projection and geometric transformations.

Matplotlib: Utilized for visualizing results, such as the final rendered image and performance metrics of various stages in the pipeline.

1. **Perspective Projection:** The simulation begins with transforming 3D vertex data into 2D screen coordinates using a perspective matrix. This step mimics the GPU's vertex shader functionality.

2. **Rasterization:** The process maps 2D vertices onto pixel fragments, identifying which pixels

belong to each triangle Algorithms based on bounding boxes and barycentric coordinates are implemented.

3. Outline Buffer Updates: The final step involves writing pixel data (color, depth) to the simulated. framebuffer, mimicking the GPU's output processes. The simulation includes a conceptual breakdown of GPU modules, such as rasterizers and frame buffers, illustrating their roles in graphics rendering. This detailed representation provides a clear understanding of how GPU hardware translates 3D data into 2D images while highlighting the interaction between these hardware components.

4. Results and Discussion

Frame Buffer Results:

The results are compiled with the help of EDA Play software and The simulation result is mentioned below. The Results consists of 3D to 2D color shading of triangle and the bar graph pictorial representation of GPU operating time and output of Frame Buffer and Rasterization.

2. Triangle rasterization effectively visualized in a 512x512 picture space.

3. Execution metrics:

- Rasterization: 10 clock cycles.
- Shading: 15 clock cycles.
- Outline Buffer Type in: 5 clock cycles.

6.3 Google Colab Output:



Fig 6.3.1 Expected Output of Colour shading of triangle When viewed in Google colabnotebook



Fig 6.1 Expected output of Frame Buffer When viewed in EDA Play Software

6.2 Rasterization Output:

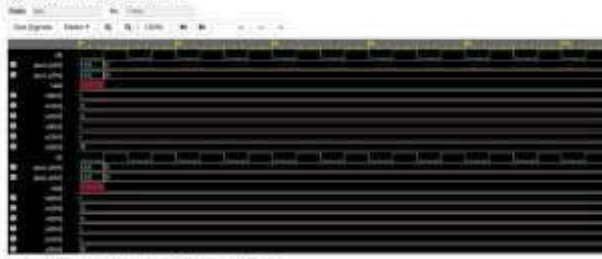


Fig 6.2 Expected Output Of Rasterization when viewed in EDA Play Software

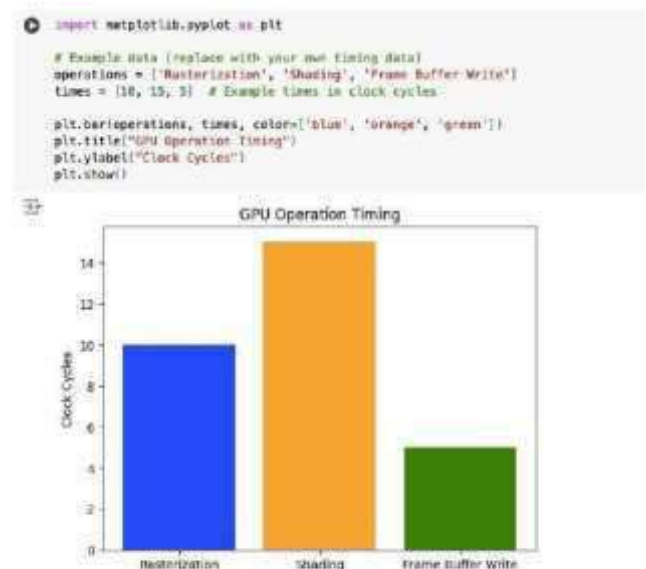


Figure 6.3.2 Plot of GPU Operation time VS Clock Cycle

1.Exact point of view projection and rasterization.

6.4 Discussion

The outcomes illustrate the possibility of mimicking GPU operations in program. Timing information uncover the bottlenecks in shading, recommending optimization potential in shading calculations. Comparisons with equipment executions are too examined. The Outputs and observed and Analyzed with the help of EDA Play Software the outputs of 6.1 and 6.2 of Frame Buffer and Rasterization is viewed as in the waveform. The outputs in the fig 6.3.1 and 6.3.2 is viewed with the help Google Colab and the time complexity vs clock cycles with respect to Rasterization, Shading and Frame Buffer Write is viewed in fig 6.3.2. Overall Output helps us to get the clear picture of the concept of color shading and pixel shading of a simple triangle taken as an example

5. Conclusion (Size 10)

The study effectively mimicked GPU operations, showcasing the remarkable efficiency of parallel processing in handling rendering tasks. By leveraging parallelism, the operations demonstrated significant improvements in computational speed and resource utilization. The timing analysis provided critical insights into performance bottlenecks, laying a robust foundation for further optimizations. Future work could focus on enhancing the precision of resource allocation, exploring advanced scheduling techniques, and integrating more complex datasets to refine the rendering pipeline. These efforts would aim to maximize efficiency, reduce latency, and ensure scalability, thereby advancing the capabilities of GPU-based systems in high-performance computing and rendering applications.

References

- [1] M. Segal, K. Akeley, C. Frazier, J. Leech, and others, "The OpenGL Graphics System: A Specification," Graphics, 2004.

- [2] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008, doi: 10.1109/JPROC.2008.917757.
- [3] D. Ginsburg, B. Purnomo, D. Shreiner, and A. Munshi, *OpenGL ES 3.0 Programming Guide*, 2nd Edition. 2014. [Online]. Available: <http://it-ebooks.info/book/3674>
- [4] S. Marschner and P. Shirley, "he," in *Fundamentals of Computer Graphics*, Boca Raton: A K Peters/CRC Press, 2021, pp. 177–204. doi: 10.1201/9781003050339-9.
- [5] D. A. Hennessy, J. L., & Patterson, the Architecture of Computer hardware, System Software, And networking, vol. 53, no. 9. 2013.
- [6] "The Definitive Guide to Programmable Real- Time Graphics", [Online]. Available: <https://www.amazon.in/Cg-Tutorial-Definitive-Programmable-Real-Time/dp/0321194969>
- [7] "7.D. Shreiner, G. Sellers, and J. Kessenich, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*, 9th ed. Addison-Wesley, 2016."
- [8] "8.M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*, 3rd ed. Morgan Kaufmann, 2016."
- [9] C. Dachsbacher and J. Kautz, "Real-time global illumination for dynamic scenes," in *ACM SIGGRAPH 2009 Courses*, New York, NY, USA: ACM, Aug. 2009, pp. 1–217. doi: 10.1145/1667239.1667258.
- [10] "M. Garland and P. S. Heckbert, 'Surface Simplification Using Quadric Error Metrics,' in *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, 1997.