A RISC-V Accelerator for Triangle Intersection in Ray Tracing

Yugal Kithany, David Phillips, Jin Jiang
University of Illinois Urbana-Champaign
{kithany2, davidp2, jinj2}@illinois.edu

***Abstract*—Ray tracing simulates light by casting rays through pixels and determining their interactions with 3D triangle meshes. While realistic, ray tracing has a significant computational cost, especially in complex scenes with millions of triangles. This project presents a hardware accelerator for the Ray-Triangle intersection test, built on a RISC-V core. Using a Finite State Machine and floating-point units for parallel computation, the design improves performance by over 70% while significantly reducing energy consumption.**

I. Background

Ray tracing, unlike rasterization, simulates the physical behavior of light by casting rays from the camera into the scene. Each ray travels through a pixel on the screen and determines intersections with objects. If it hits an object, the ray gathers lighting information—such as diffuse reflection, specular reflection, refraction, and global illumination—and computes the final color for that pixel [1]. In modern computer graphics, an object is made up of multiple triangle meshes, and assembling a seemingly solid figure from triangle "building blocks" is the most fundamental process in 3D computer graphics [2].

While Ray Tracing produces realistic renderings compared with its old rasterization counterpart, the computational cost associated with this realism is significantly higher. In the most naive approach, each ray generated must be tested against all objects (in this case, triangle meshes) in the scene, unlike for rasterization [3]. For many scenes, the amount of meshes is astronomical. For example, the Moana Island scene in the reference contains over 146 million unique triangles [4]. Bounding Volume Hierarchy, which cuts down triangle searching and traversal cost from O(n) to O(log(n)), is one of many techniques that could reduce compute cost[5]. This project aims to build an accelerator that enhances the compute speed of this ray-triangle intersection. An open-source RI5CY: RISC-V core is utilized as the base upon which the accelerator is integrated [6], and validation and modeling of the module uses the VCS toolchain. Since most calculations are floating-point operations, the selected core needed to have fpNew integration, an open-source project, for floating-point operation calculation [7].

II. Methodology

The Ray-Triangle Intersection algorithm, shown in Figure 1, is described as follows. The input is 3 points representing the triangle in 3D space, and a point and vector representing the light ray. This process is divided into two stages: the ray-plane intersection phase and the barycentric coordinate phase. The split is necessary because if the ray does not intersect the plane containing the triangle, it cannot intersect the triangle itself.

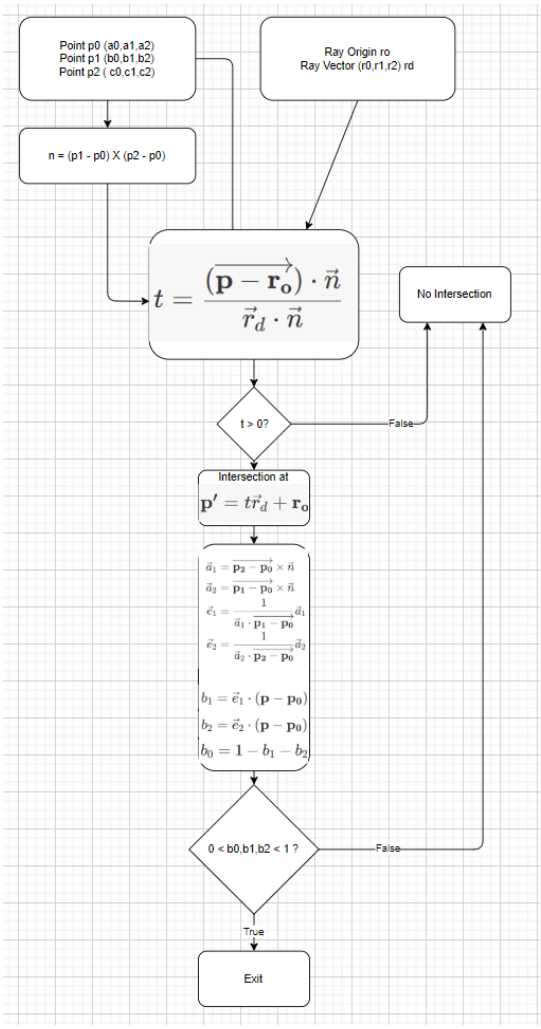

Fig. 1: Control Flow of Enhanced Architecture.

In the second phase, the intersection point is also required when checking if the point is within the triangle by calculating the barycentric coordinates. Once the normal vector of the triangular plane is computed, the distance of intersection from the ray origin to the plane is calculated. If the intersection

distance is negative, there are no intersections with the plane, let alone the triangle, so an early exit is allowed. In the case of an intersection, the point is calculated. The next step is inverse triangle mapping, which derives the barycentric coordinates of each vertex of the triangle, which states how close to each of the three vertices the point in question is. The key point of the number is that they sum to 1, and any negative number indicates that the point lies on the opposite side of the triangle, signifying that it is not contained within it. [8] These two properties can determine whether the point is within the triangle and, thus, if the ray intersects with the triangle [9].

The hardware acceleration is accomplished by creating a module with an FSM closely following the above algorithm with minimum floating point computational units required to enhance parallelized derivations. There are 26 states, each corresponding to a calculation in Figure 1. The optimal floating-point hardware configuration was determined to be 6 FP multipliers, 3 FP dividers, and 1FP comparer. Since all instructions are vector operations in a 3D environment, most stages are highly parallelized with high throughput.

Two custom instructions are required to enable communication between the processor and accelerator. The first, RTI.S, starts the computation on the accelerator by using an unused opcode from the original ISA, with funct7, rs1, and rs2 left empty and rd holding the result. The second instruction, RTILD.S, loads the first 15 floating-point numbers from a specific memory address into the accelerator with the same opcode as RTI.S. These two instructions form an interface for the processor to interact with the accelerator.

III. Results

The results are shown in the three graphs on the left. For the clock cycle comparison, a built-in Verilator testbench running a custom program with one iteration of the algorithm was used to compute the baseline cycles. The accelerated clock cycle count was derived using a program counter in Verdi running a similar program using our custom extension instead. For the Area comparison, the baseline was computed using 90 kGE from the core documentation [6] and a FreePDK area estimate of 0.798 µm2 per kGE [10]. The accelerated area was the result of Synopsys DC. For the Power comparison, the baseline was computed from a research paper that benchmarked RI5CY [11]. The accelerated power was computed by adding that value to the power estimate of our module from Synopsys DC. The total energy was computed by multiplying the average power of both configs and the clock cycles required to compute.

- **Performance**: 70.8% reduction in clock cycles (Verilator/Verdi).
- **Area**: Increase due to floating point units (Synopsys DC).
- **Energy**: 69.2% energy improvement despite higher area.

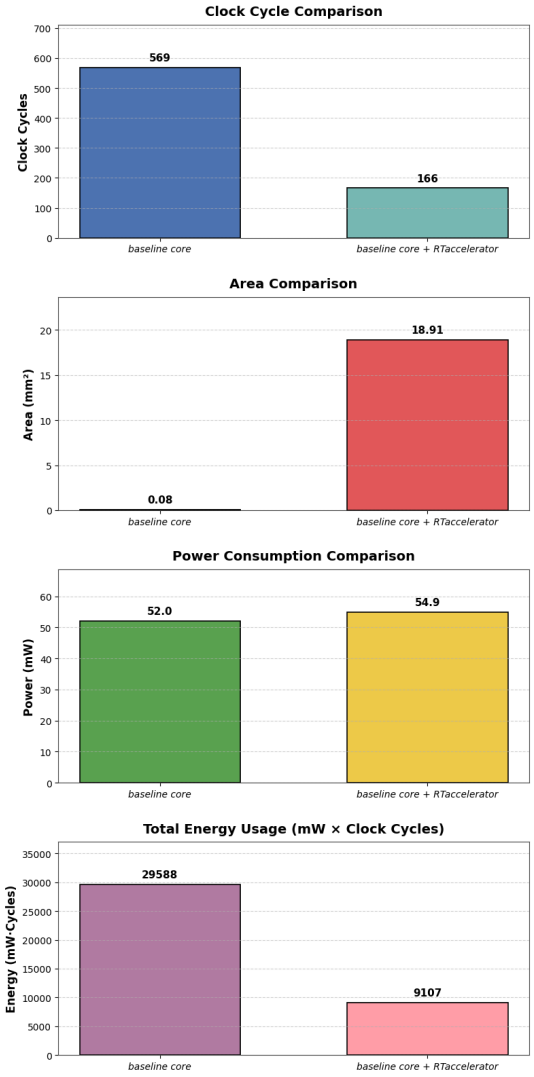

Fig. 2: Performance and power comparison of baseline vs. accelerated core.

IV. Conclusion

In conclusion, the Ray-Tracing accelerator successfully accelerates the ray-triangle intersection part of the rendering process. By carefully partitioning the intersection algorithm, it achieves a performance improvement of 70.8% and an energy usage improvement of 69.2% with constant power consumption. However, our area was notably impacted, increasing several orders of magnitude. Because the baseline core was not natively synthesizable with Synopsys DC due to its open-source background and age, this area comparison may distort the actual area impact.

Potential improvements to this design mostly involve the input registers; enabling the user to use the built-in FP registers and dynamically choosing which registers are used as inputs could avoid some redundant memory operations in actual use. Processing in memory via some special address space and I/O could also greatly improve the accelerator's utility.

Some area-increasing flaws to be addressed in future it-

erations of the design are replacing internal flip-flops with SRAM, reducing the number of scratchpad registers to the mathematical minimum required to execute the algorithm, and reducing the number of FP-Division units to 1 as they are particularly area-intensive.

REFERENCES

[1] E. Haines and T. Akenine-Möller, *Ray Tracing Gems*, Apress, 2019.
[2] C. Petzold, "DirectX Factor: Triangles and Tessellation," *MSDN Magazine*, vol. 29, no. 3, Mar. 2014.
[3] G. S. Owen, "Accelerating Ray Tracing," *SIGGRAPH Education*, Apr. 1998.
[4] M. Pharr et al., *Physically Based Rendering*, 4th ed., Morgan Kaufmann, 2020.
[5] "Bounding volume hierarchy," Wikipedia, May 2025.
[6] recogni, "RISCY: RISC-V Core," GitHub, 2025.
[7] S. Mach et al., "Fpnew: An open-source multiformat floating-point unit," *IEEE Trans. VLSI Syst.*, 2020.
[8] UIUC, "CS 418 – Raytracing."
[9] T. Möller and B. Trumbore, "Fast Ray-Triangle Intersection," *J. Graphics Tools*, 1997.
[10] lowRISC, "Memory Safety Features," 2025.
[11] R. Núñez-Prieto et al., "RisCO2," *Micromachines*, vol. 14, 2023.
[12] Y. Kithany, "RTAccelerator," GitHub. https://github.com/YugalKithany/RTAccelerator