Enabling Differentiable Rendering via AD of Parallel C++ STL Primitives in Clad

Student: Abdelrhman Elrawy

Mentors: Vassil Vassilev, Alexander Penev

Organization: HEP Software Foundation (CompRes/Clad)

About Me

Abdelrhman Elrawy

Master in Applied Computing

Wilfrid Laurier University, Waterloo, Canada

Clad Contributor (GSoC 2025)

Successfully extended Clad's capabilities to differentiate through NVIDIA's Thrust parallel algorithms,

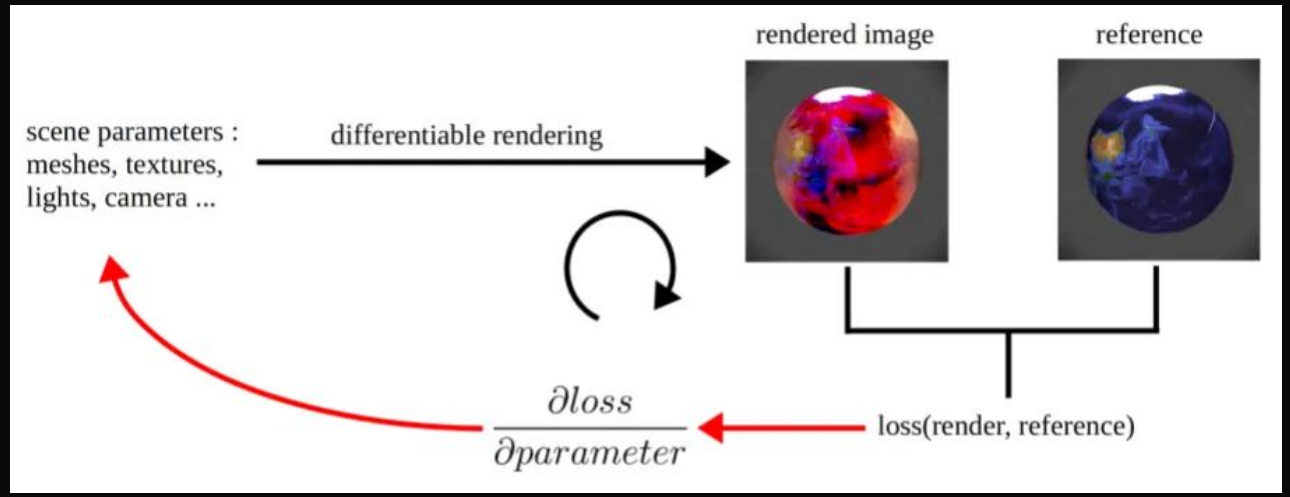
Professional Background

Former Machine Learning Engineer, specializing in creating high-fidelity animatable avatars.

What is Differentiable Rendering?

Differentiable rendering bridges the gap between 2D images and 3D scenes through **inverse graphics**.

- **Forward Pass:** Computes a 2D image from 3D scene parameters (geometry, materials, lighting).
- **Backward Pass:** Calculates the analytical derivatives (gradients) of the rendering process.
- **The Goal:** It allows us to propagate gradients from pixel colors directly back to 3D properties, enabling the optimization of complex scenes purely from 2D photos using gradient descent.



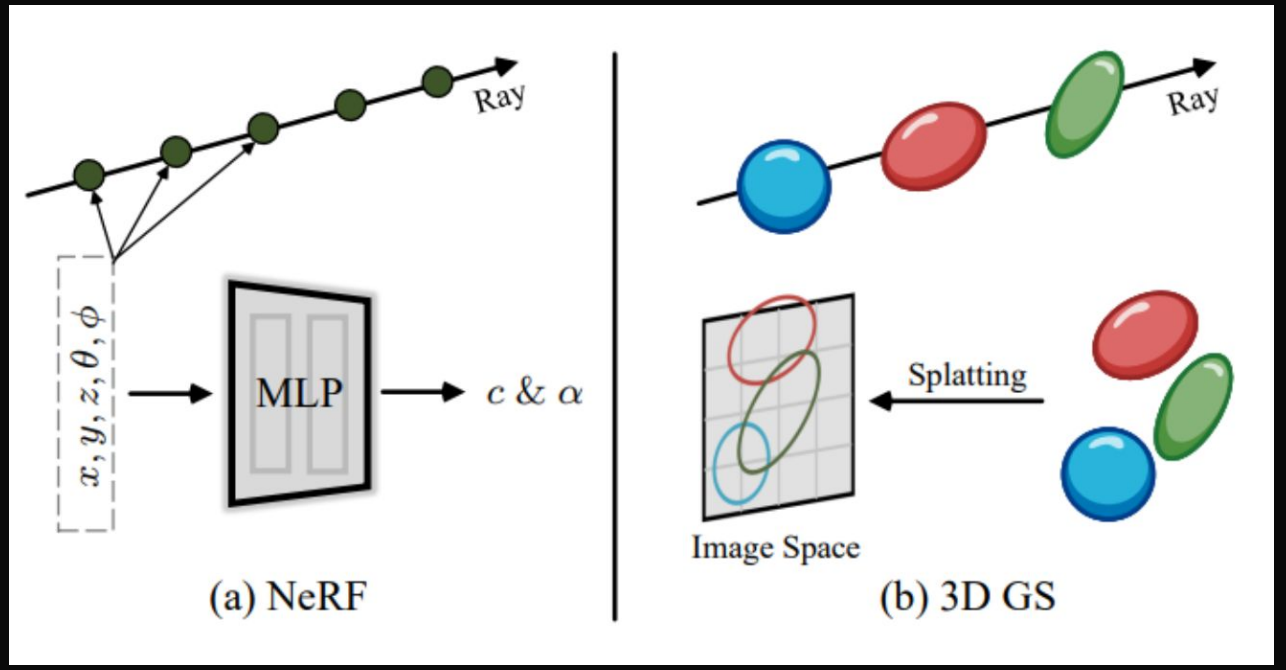
https://medium.com/qarnot/an-overview-of-differentiable-rendering-20ceb8d20cbe

3D Gaussian Splatting

Real-Time Photorealism

3DGS has revolutionized novel-view synthesis. Instead of heavy neural networks or traditional meshes, it represents scenes using millions of fuzzy, colorful 3D ellipsoids (Gaussians).

By dividing the screen into tiles and utilizing hardware-accelerated sorting and alpha-compositing, it achieves extremely fast, photorealistic real-time rendering, avoiding the computational expense of ray-marching.

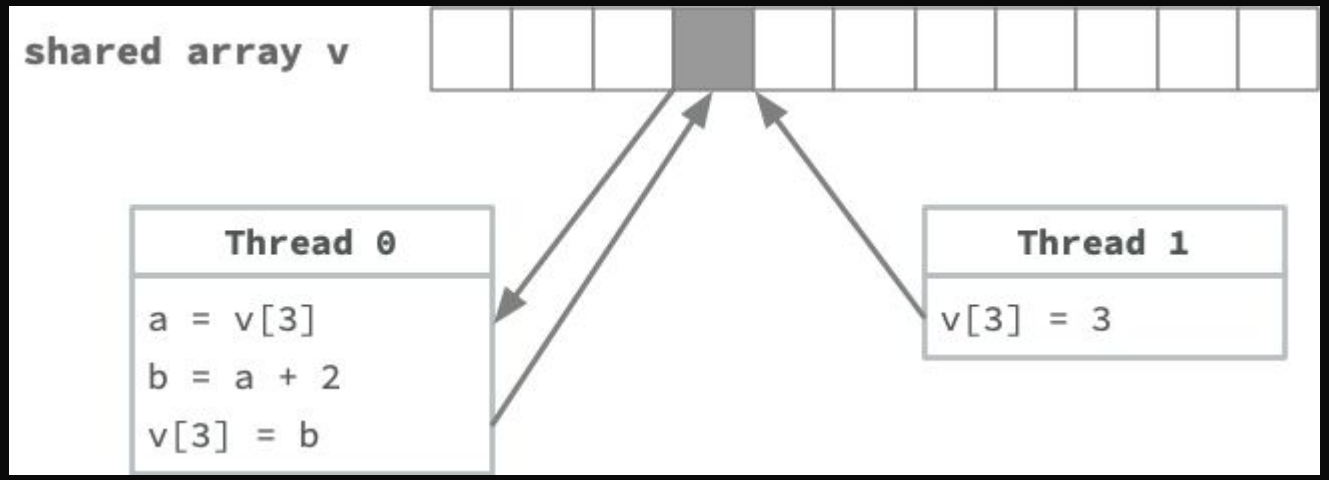


https://arxiv.org/abs/2308.04079

The Atomic Bottleneck

While the forward pass is incredibly fast, training these renderers hits a severe performance wall during backpropagation.

- Multiple pixels contribute gradients to the same 3D Gaussian simultaneously.
- To prevent data races, the backward pass relies heavily on atomicAdd instructions.
- This scatter-add pattern creates massive contention, overwhelming the GPU's L2 cache atomic units.
- Execution becomes serialized, significantly bottlenecking the entire training pipeline.



https://cvw.cac.cornell.edu/parallel/memory-access/multithreading

Shifting the Paradigm: Gather-Reduce

Current: Scatter-Add

The traditional graphics approach relies on hand-written CUDA loops scattering gradients directly to global memory.
Because multiple threads might hit the same memory address, the compiler (and hardware) is forced to utilize atomics, causing unpredictable serialization and degraded performance.

Proposed: Gather-Reduce

By restructuring the pipeline using explicit **Thrust API** primitives (like thrust::sort_by_key and thrust::reduce), we expose deterministic dataflow.
This allows Clad's static analysis to mathematically prove *single-owner memory access*, eliminating the need for atomics completely.

Extending Clad for Concurrency

- **Current Limitation:** Clad conservatively emits atomic operations in the reverse pass when encountering parallel constructs because it assumes non-deterministic memory access.
- **Static Analysis Solution:** Reframe concurrency-aware differentiation as a static analysis problem. We will extend Clad's liveness engine to analyze memory access patterns for injectivity.
- **Lock-Free Synthesis:** When Clad detects deterministic grouping and single-owner semantics mapped through Thrust, it will downgrade atomic operations to highly efficient, standard memory writes.
- **Broader Impact:** This establishes a foundation for compiler-driven automatic differentiation of *all* parallel C++ programs.

Core Implementation Phases

Phase 1: Isolation

Modify geometric abstractions in Clad's existing Path Tracer. Validate that Clad's liveness analysis correctly identifies Thrust-driven injective memory mappings before tackling full 3DGS.

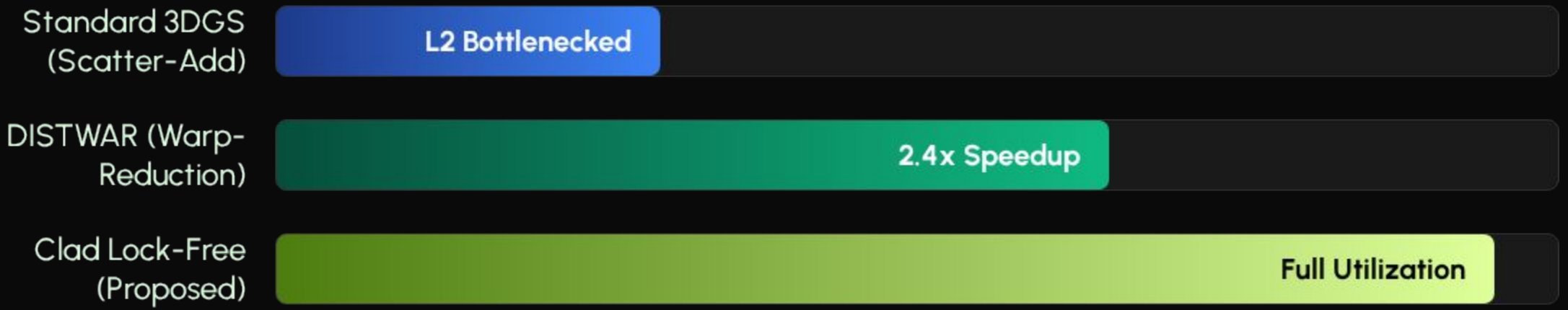
Phase 2: Redesign

Fork diff-gaussian-rasterization and replace scatter-add loops with a deterministic, tile-based memory ownership paradigm using high-level Thrust parallel algorithms.

Phase 3: Synthesis

Execute Clad on the refactored forward pass. The compiler will automatically synthesize a mathematically correct, lock-free backward pass leveraging standard load-add-store operations.

Performance Target (GPU Utilization)



By utilizing the compiler to completely eliminate L2 atomic unit saturation, our approach aims to significantly outperform even heavily optimized, hand-written warp-reduction baselines.

Execution Roadmap (24 Weeks)

Months 1-2

Clad Liveness Audit & Concurrency AD Support

Months 3-4

3DGS Forward Redesign & Backward Synthesis

Month 5

Nsight Profiling & Finite Difference Validation

Month 6

Documentation & Final Mainline Integration

Thank You