
# **Project Title**: Enabling Differentiable Rendering via AD of Parallel C++ STL Primitives in Clad

### **Project Parameters**:

- **Duration**: June-September (4 months)
- **Total Hours**: 350 hours (~22 hours/week)
- **Focus**: Differentiable Rendering, CUDA Atomics Elimination, Thrust API Integration, C++ concurrency

**Student:** Abdelrhman Elrawy
**Mentors:** Vassil Vassilev, David Lange
**Organization:** HEP Software Foundation (CompRes/Clad)

Contact Information:

Abdelrhman Elrawy
GitHub: @a-elrawy
E-mail: abdelrhman.elrawy1@gmail.com
Phone number: +1 416 357 8105

# 1. Project Summary

## Introduction

This project proposes to enhance Clad, a Clang-based automatic differentiation (AD) tool, by leveraging its liveness analysis to automatically generate lock-free backward passes for highly parallel differentiable rendering pipelines. Specifically, the project targets the atomic bottleneck inherent in 3D Gaussian Splatting (3DGS) rasterization.

Modern GPU-based differentiable renderers suffer from heavy use of atomic operations (e.g., atomicAdd) during the backward pass because many threads attempt to update the same pixel and gradient buffers simultaneously. This contention overwhelms L2 cache atomic units, stalling execution and causing the gradient computation to consume a significant portion of training time.

My approach is twofold. First, I will redesign the rendering pipeline to avoid write conflicts by construction using deterministic sorting, tile-based memory ownership, and local accumulation before a single global write. Crucially, I plan to build this new architecture on top of my previous GSoC work that integrated the Thrust API into Clad. By utilizing Thrust's highly optimized parallel primitives (such as thrust::sort_by_key and thrust::reduce) for the sorting and tile-based operations, we can provide a clean, high-level C++ structure to the compiler. Second, I will utilize Clad to automatically generate the backward pass. By relying on Clad's liveness analysis, the compiler will prove exclusive memory ownership and automatically eliminate the need for atomicAdd calls. To isolate the compiler complexity from the graphics complexity, the project will use a differentiable geometry path tracer as the foundational stepping stone. Beyond differentiable rendering, this work establishes a foundation for **compiler-driven automatic differentiation of parallel C++ programs**, enabling efficient gradient computation in a wide range of high-performance computing applications.

---

## Key Challenges

1. **Concurrency Semantics in AD:** Ensuring correctness when relaxing synchronization assumptions, avoiding silent data races in compiler-generated gradients.
2. **Balancing Graphics vs. Compiler Work:** Safely decoupling the refactoring of the complex 3DGS pipeline from the AST-level compiler analysis required in Clad.
3. **Liveness-Aware Differentiation**: Structuring C++ graphics code in a way so Clad's static analysis can prove injective memory access patterns and remove atomics.
4. **GPU Memory Management**: Managing local gradient accumulations entirely within SM Shared Memory and guaranteeing safe, single-owner global commits without relying on standard hand-written reverse CUDA kernels.

---

# Extending AD Support to C++ Concurrency Primitives

A core component of this project is extending Clad's automatic differentiation capabilities to correctly handle modern C++ concurrency primitives. While Clad currently supports differentiation through standard sequential C++ constructs and selected Thrust APIs, it lacks a systematic framework for reasoning about concurrency constructs such as:

- Atomic operations (e.g., `atomicAdd`, `std::atomic`)
- Parallel execution policies (e.g., `std::execution::par`, `thrust::device`)
- Parallel algorithms (e.g., `thrust::reduce_by_key`, `thrust::for_each`)

These primitives introduce non-trivial challenges for AD because they break the assumption of deterministic, single-writer memory access. As a result, Clad conservatively emits atomic operations in the reverse pass, even when they are not semantically required.

This project reframes concurrency-aware differentiation as a **static analysis problem**, where:

- Memory access patterns are analyzed for injectivity
- Parallel constructs are mapped to deterministic dataflow
- Atomic operations are eliminated when single-owner semantics can be proven

By extending Clad's liveness and dependency analysis to reason about these concurrency primitives, the project enables **lock-free automatic differentiation for parallel C++ programs**, not just differentiable rendering pipelines.

# 2. Technical Approach

## Background: Differentiable Rasterization in 3D Gaussian Splatting

3D Gaussian Splatting (3DGS) relies on a tile-based differentiable rasterization pipeline that transforms a set of 3D Gaussians into a rendered image while preserving gradients for optimization. Understanding this pipeline is essential to identifying the atomic bottleneck that this project targets.

### Forward Pipeline

The forward rendering process consists of several structured stages:

1. **Tile-Based Decomposition**
   The image is partitioned into fixed-size tiles (e.g., 16×16 pixels). Each tile is processed independently by a CUDA thread block, enabling high parallelism.

2. **Gaussian Projection and Culling**
   Each 3D Gaussian (defined by mean, covariance, opacity, and color) is projected into screen space and approximated as a 2D ellipse. Gaussians outside the view frustum or not overlapping a tile are discarded.

3. **Tile Instancing and Key Generation**
   Each Gaussian is duplicated for every tile it overlaps. A sorting key is constructed: key = (tile_id, depth). This ensures both correct visibility ordering (front-to-back) and grouping by tile.

4. **Global Sorting**
   All Gaussian instances are sorted using a GPU radix sort. This produces a globally ordered list where Gaussians belonging to the same tile occupy contiguous memory ranges.

5. **Per-Tile Rasterization (Forward Pass)**
   Each tile is processed by a thread block where:
   - each thread corresponds to a pixel,
   - Gaussians are traversed in front-to-back order,
   - color is accumulated using alpha compositing:
     i. $C\_out = C\_in + (1 - \alpha\_in) * \alpha\_i * c\_i$
     ii. $\alpha\_out = \alpha\_in + (1 - \alpha\_in) * \alpha\_i$

   This stage is inherently conflict-free since each thread writes exclusively to its own pixel.

## Backward Pass and Atomic Bottleneck

During differentiation, gradients must be propagated from pixels back to Gaussian parameters. This is achieved by traversing the same sorted lists in reverse order (back-to-front) and applying the chain rule.

However, unlike the forward pass, **multiple pixels contribute gradients to the same Gaussian**, resulting in a scatter-add pattern:

```
atomicAdd(&grad_gaussian[i], contribution);
```

This introduces:

- heavy contention in L2 cache atomic units,
- serialization of memory updates,
- significant slowdown in the backward pass.

In practice, this atomic accumulation becomes the dominant performance bottleneck in differentiable rendering pipelines such as 3DGS.

## Key Insight: From Scatter-Add to Deterministic Gather-Reduce

The core limitation is not the rendering equation itself, but the **memory access pattern** used during gradient accumulation.

This project proposes to reformulate rasterization as a **deterministic gather-reduce computation**, where:

- Contributions are grouped using sorting,
- Accumulation is performed locally without conflicts,

- Each memory location has a single writer.

This transformation eliminates the need for atomic operations by construction and exposes a structure that can be exploited by compiler-level analysis.

## Minimal Example: Differentiable Splat Accumulation

To illustrate the transformation, consider a simplified per-pixel compositing process:

### Forward Pass (Sequential Compositing)

```
struct Splat {
   float alpha;
   float color;
};

float render_pixel(const thrust::device_vector<Splat>& splats) {
   float T = 1.0f;
   float C = 0.0f;

   for (int i = 0; i < splats.size(); i++) {
       float w = T * splats[i].alpha;
       C += w * splats[i].color;
       T *= (1.0f - splats[i].alpha);
   }
   return C;
}
```

### Naive Backward (Requires Atomics)

```
// Multiple pixels updating same splat
atomicAdd(&grad_alpha[i], contribution);
Refactored Gather-Reduce Formulation
thrust::device_vector<float> contributions(splats.size());

thrust::transform(
   splats.begin(), splats.end(),
   contributions.begin(),
   [] __device__ (const Splat& s) {
       return compute_contribution(s);
   }
);

// Conflict-free accumulation
float total = thrust::reduce(contributions.begin(),
contributions.end());
```

Because the forward pass enforces deterministic ordering and single-writer accumulation, Clad's liveness analysis can prove that gradient updates are non-conflicting. As a result, the generated backward pass uses standard load-add-store operations instead of atomicAdd.

### Slang.D-Based Gaussian Rasterization

To further validate and guide the proposed approach, this project will integrate insights from slang-gaussian-rasterization by using it as a high-level differentiable rasterization reference for Gaussian splatting pipelines. Specifically, I will leverage its unified, differentiable rendering framework to prototype and analyze structured gather-reduce formulations, cross-check gradient correctness against Clad-generated backward passes, and study how its explicit dataflow (sorting, grouping, reduction) can be mapped into equivalent C++/Thrust abstractions. These insights will directly inform the design of Clad-compatible code patterns that expose deterministic memory access and enable liveness-based atomic elimination, effectively bridging high-level differentiable rendering design with efficient, lock-free execution in standard C++ pipelines.

### Transition to Proposed Approach

Based on this analysis, the central idea of this proposal is to restructure the 3DGS rasterization pipeline into a deterministic gather-reduce formulation using Thrust primitives, enabling Clad to automatically synthesize lock-free backward passes through liveness-aware differentiation.

## Community Bonding Period (May, pre-coding)

### Relationship Building and Environment Setup

During the three-week Community Bonding period, I'll focus on establishing relationships with mentors and finalizing the integration of my differentiable geometry path tracer branch into the Clad tree. I'll ensure my LLVM/Clang and CUDA development environments are perfectly aligned with project standards to allow for immediate, effective testing.

### Proof-of-Concept Implementation

Building on the deterministic gather-reduce formulation introduced above, this phase focuses on expressing the rasterization pipeline using high-level Thrust primitives that expose explicit dataflow to Clad's static analysis.

A critical goal during this period is to manually demonstrate how a shift from "scatter-add" to a high-level Thrust-driven "gather-reduce" allows a compiler to drop atomics. Rather than

jumping straight into 3DGS, I will build on my path tracer work to create a standalone proxy demonstrating single-owner memory.

An example of the architectural shift required for the forward computation looks like:

```
// Traditional scatter-add paradigm (Forces Clad to emit atomicAdd in
reverse mode)
// Hand-written CUDA loops or naive transforms cause memory contention
void standard_accumulation(const thrust::device_vector<float>& inputs,
float* global_gradients) {
    thrust::for_each(inputs.begin(), inputs.end(), [=] __device__ (float
input) {
        int target_pixel = compute_target(input);
        // Race condition: Multiple threads hitting the same
target_pixel forces atomics
        atomicAdd(&global_gradients[target_pixel], compute_loss(input));
    });
}

// Proposed gather-reduce paradigm leveraging High-Level Thrust
Primitives
// Completely eliminates atomics by using deterministic sorting and
reduction
void thrust_deterministic_accumulation(thrust::device_vector<Gaussian>&
gaussians, thrust::device_vector<float>& global_gradients) {
    thrust::device_vector<int> pixel_keys(gaussians.size());
    thrust::device_vector<float> pixel_losses(gaussians.size());

    // 1. Generate Keys: Map each Gaussian to its target pixel/tile and
compute initial loss
    thrust::transform(gaussians.begin(), gaussians.end(),

thrust::make_zip_iterator(thrust::make_tuple(pixel_keys.begin(),
pixel_losses.begin())),
                      map_to_pixel_and_loss());

    // 2. Deterministic Sorting: Group all operations hitting the same
pixel together
    thrust::sort_by_key(pixel_keys.begin(), pixel_keys.end(),
pixel_losses.begin());

    // 3. Conflict-Free Accumulation: Reduce by key guarantees
single-owner memory.
    thrust::device_vector<int> unique_pixels(gaussians.size());
    thrust::device_vector<float> reduced_gradients(gaussians.size());
```

```
    auto new_end = thrust::reduce_by_key(
        pixel_keys.begin(), pixel_keys.end(),
        pixel_losses.begin(),
        unique_pixels.begin(),
        reduced_gradients.begin()
    );

    // 4. Safe, non-atomic global commit using the unique keys
    thrust::scatter(reduced_gradients.begin(), new_end.second,
unique_pixels.begin(), global_gradients.begin());
}
```

By manually writing out the target C++ forward structure and leveraging high-level Thrust APIs like thrust::sort_by_key and thrust::reduce_by_key, I will isolate the variables needed to test Clad's ReverseModeVisitor and its atomic elimination logic before the official coding period begins.

### Benchmark Development Plan

To systematically drive the implementation and validate performance, I will develop benchmarks in order of increasing complexity:

1. **Differentiable Geometry Path Tracer**: Using the foundational path tracer already in the Clad tree. I will restructure its intersection aggregations to utilize deterministic Thrust arrays, ensuring Clad correctly drops atomics during the reverse mode pass.

2. **2D Toy Splatting**: A simplified tile-based 2D Gaussian rasterizer. This will test Clad's ability to handle the scope of Thrust reduction operations during AD generation.

3. **Full 3DGS Pipeline**: Integrating the compiler-generated backward pass into the diff-gaussian-rasterization codebase and comparing its speed and memory profile against heavily optimized, hand-written warp-reduction baselines like DISTWAR (which achieves ~2.4x speedups but still relies on warp-level atomics).

## Research and Analysis (June 1-20, ~80 hours)

### Clad Liveness Analysis Audit (35 hours)

This phase involves a thorough analysis of Clad's existing architecture for handling abstract syntax tree (AST) transformations and CUDA operations. My previous GSoC work gave me deep insight into ReverseModeVisitor.cpp and how Clad handles VisitCallExpr. However, for this project, the critical focus is on the liveness analysis engine. I will audit how Clad currently defaults to `BuildCallToCudaAtomicAdd` when differentiating parallel memory updates, and identify how this behavior generalizes to broader C++ concurrency primitives

such as `std::atomic`, parallel execution policies (`std::execution::par`), and Thrust/CUDA parallel algorithms.

In particular, I will analyze:

- How Clad models side effects and memory dependencies in parallel regions
- Whether current liveness analysis distinguishes between true write conflicts and statically provable single-writer patterns
- How concurrency constructs are represented in the Clang AST and how they propagate into Clad's ReverseModeVisitor

This will guide the extension of Clad's analysis to treat concurrency primitives not as inherently unsafe, but as **conditionally eliminable synchronization constructs** when stronger guarantees (e.g., injective indexing, deterministic grouping) are present.

I will map out the specific AST configurations required to trigger Clad's newly implemented atomic elimination features. Specifically, I need to understand the boundaries of Clad's static analyzer in detecting injective index computations (i.e., verifying a strict 1:1 mapping between a thread's identifier and its memory output target). I will analyze how Clad resolves the dependency graphs of Thrust's reduce_by_key to ensure we can consistently bypass global atomicAdd instructions.

### Graphics Pipeline Memory Analysis (25 hours)

Simultaneously, I will audit the diff-gaussian-rasterization forward pass. I will trace the exact data flow from 3D Gaussian projection, radix sorting, and tile assignment. The goal is to identify the exact raw CUDA loops and scatter-add mechanics that cause the L2 cache memory contention. I will then design a refactoring strategy to replace these loops with Thrust primitives that the compiler can definitively interpret as single-owner.

### Implementation Strategy Development (20 hours)

Based on the compiler audit and the graphics pipeline analysis, I will formalize the C++ architectural redesign. I will design the deterministic sorting algorithm (sorting intersections by both depth and Gaussian ID within a tile context using thrust::sort_by_key) and define the exact memory boundaries required so that Clad can safely emit load-add-store operations instead of atomics during the backward pass.

## Core Implementation (June 21-July 31, ~120 hours)

### Phase 0: Concurrency Primitive Differentiation (30 hours)

Before integrating with rendering pipelines, I will extend Clad to explicitly support differentiation through C++ concurrency primitives.

This includes:

- Defining derivative rules for atomic operations (e.g., modeling `atomicAdd` as an associative reduction over contributions)
- Introducing abstractions to represent parallel regions in Clad's internal IR
- Extending liveness analysis to detect when atomic operations can be safely downgraded to standard memory writes

I will construct minimal examples involving:

- `std::atomic` accumulations
- Thrust parallel algorithms
- CUDA atomic operations

These examples will serve as unit tests to validate that Clad:

- Preserves correctness under concurrency
- Eliminates unnecessary synchronization when safe

This phase ensures that concurrency-aware AD is validated independently before being applied to the full 3DGS pipeline, and it establishes the foundation for treating concurrency primitives as first-class differentiable constructs within Clad.

## Phase 1: Validating Abstractions in the Differentiable Geometry Path Tracer (30 hours)

To completely isolate the Clad work from the Graphics work, I will begin integration entirely within the differentiable geometry path tracer codebase.

1. I will modify the geometric intersection abstractions to enforce predictable, injective memory access using Thrust.

2. I will run Clad's reverse mode AD on these modified algorithms.

3. I will iteratively adjust the C++ abstractions and modify Clad's internal AST visitor logic until the compiler's liveness analysis successfully and consistently proves exclusive memory ownership, resulting in a lock-free, mathematically correct backward pass.

## Phase 2: 3DGS Forward Pass Redesign (30 hours)

This phase represents the core graphics engineering. I will fork the diff-gaussian-rasterization engine and implement the tile-based memory ownership paradigm. This replaces the standard parallel scatter-add logic with a strictly deterministic gather-reduce execution model. By utilizing Thrust's high-level reduction algorithms (thrust::reduce_by_key) to enforce single-owner memory at the tile level, I will create a

forward pass whose memory access patterns are fully transparent to Clad's static analysis constraints.

## Phase 3: Clad Backward Pass Synthesis (30 hours)

I will apply Clad to the refactored 3DGS forward pass. Because the C++ code explicitly defines injective mappings by design using Thrust, Clad's extended liveness analysis will recognize the safe access patterns. It will bypass the conservative BuildCallToCudaAtomicAdd fallbacks and automatically emit standard, high-performance load-add-store machine code for the backward pass.

To illustrate, the core objective of this phase is for Clad to automatically synthesize pullbacks for our Thrust operations that look conceptually like this internally:

```
// Example of the Clad-synthesized pullback (adjoint) for a Thrust
primitive.
// Because the forward pass used deterministic sorting, Clad's liveness
analysis
// proves 'd_input_begin' has single-owner access and emits standard
assignments instead of atomicAdd.
namespace clad {
namespace custom_derivatives {
namespace thrust {
    template <typename InputIt, typename OutputIt, typename UnaryOp>
    void transform_pullback(
        InputIt input_begin, InputIt input_end,
        OutputIt output_begin, UnaryOp op,
        OutputIt d_output_begin, InputIt d_input_begin) {

        // Clad leverages Thrust's internal parallelism for the backward
pass
        ::thrust::transform(
            d_output_begin, d_output_begin + (input_end - input_begin),
            input_begin, d_input_begin,
            [op] __device__ (const auto& d_out, const auto& x) {
                // Chain rule application without atomicAdd due to
guaranteed 1:1 mapping
                auto derivative = derivative_of(op, x);
                return d_out * derivative;
            });
    }
}}}
```

# Testing and Integration (August 1-31, ~80 hours)

## Unit Testing Framework (40 hours)

A robust testing framework is essential to ensure the mathematical correctness of a lock-free backward pass. Since we are removing safety nets (atomics), we must be certain no silent data races exist. I will implement numerical finite difference approximations to verify the analytical gradients generated by Clad.

## Performance Benchmarking (40 hours)

Once numerical correctness is guaranteed, I will profile the compiler-generated renderer. I will benchmark it against the standard 3DGS implementation and state-of-the-art software solutions like DISTWAR (which achieves a ~2.44x speedup by utilizing warp-level reduction but still relies heavily on L2 atomics). I will use NVIDIA Nsight Compute to verify L2 cache hit rates, track memory throughput, and definitively confirm the complete elimination of atomic unit saturation.

# Documentation and Finalization (September 1-30, ~70 hours)

## Comprehensive Documentation (30 hours)

Clear documentation is critical for adoption. I will write comprehensive guides detailing the required C++ programming patterns needed to trigger Clad's atomic elimination for parallel GPU code. I will finalize all pull requests to the main Clad repository and the diff-gaussian-rasterization fork, and prepare my final technical report and presentation highlighting the speedups achieved.

## Final Integration and Testing (25 hours)

This phase involves integrating all components into the main Clad codebase and performing comprehensive testing. I will integrate the liveness analysis triggers with Clad's main test suite, perform cross-platform testing on various NVIDIA GPU architectures (e.g., Ampere, Hopper) to verify hardware-agnostic atomic elimination, set up continuous integration for ongoing testing, and conduct final performance benchmarking to document rendering speedups.

Integration testing will ensure that the new atomic elimination logic works correctly alongside Clad's existing functionality, without regressions or conflicts. Cross-platform testing will verify that the lock-free generated kernels execute correctly across different compute capabilities without silent data races. The continuous integration setup will ensure that the functionality remains working as Clad evolves.

The final performance benchmarking will provide quantitative evidence of the benefits of our implementation, showing how compiler-driven atomic elimination can speed up 3D Gaussian

Splatting and other raster-based inverse rendering workloads. These benchmarks will be included in the documentation to help users understand the performance characteristics of the system compared to traditional scatter-add implementations.

### Final Report and Presentation (15 hours)

The project concludes with comprehensive documentation of the work completed, including a technical report detailing the implementation approach and results, presentation materials, future work recommendations for further development, and contribution to Clad documentation for user guidance.

The technical report will provide a complete account of the work done, the challenges encountered in aligning graphics pipelines with static analysis, and the solutions developed. It will serve as a reference for future developers working on Clad or similar differentiable rendering systems. The presentation materials will help communicate the value of the work to the broader visual computing and machine learning communities.

The future work recommendations will identify opportunities for further enhancement of Clad's liveness analysis capabilities, such as extending the atomic elimination support to other inverse rendering techniques, optimizing shared memory boundaries further, or applying these compiler-driven graphics optimizations to other GPU computing frameworks. These recommendations will help guide the ongoing development of Clad.

## 3. Timeline

| Dates | Tasks | Hours | Deliverables |
|---|---|---|---|
| **June 1–7** | Audit Clad's ReverseModeVisitor for CUDA atomic fallbacks. | 20h | Gradient pattern report. |
| **June 8–14** | Deep dive into liveness analysis and injective index detection. | 20h | Index constraints report. |

| | | | |
|---|---|---|---|
| **June 15–21** | Analyze memory patterns in diff-gaussian-rasterization. | 20h | Forward memory flow map. |
| **June 22–30** | Design Thrust refactoring strategy for 3DGS sorting. | 20h | Draft architecture for 3DGS. |
| **July 1–7** | **Phase 0:** Implement AD support for C++ concurrency primitives. | 30h | Updated Clad IR for atomics. |
| **July 8–14** | **Phase 1:** Path Tracer refactor and lock-free AD trigger. | 30h | PR: Lock-free AD for Path Tracer. |
| **July 15–21** | **Phase 2:** 3DGS Forward redesign (Tile-based ownership). | 30h | Refactored Forward Rasterizer. |
| **July 22–31** | **Phase 3:** Synthesize 3DGS lock-free Backward pass via Clad. | 30h | Auto-generated Backward kernel. |
| **Aug 1–7** | Unit testing and finite difference verification for stability. | 20h | Gradient accuracy report. |
| **Aug 8–14** | Silent data race detection and numerical stability checks. | 20h | Stability test suite results. |

| | | | |
|---|---|---|---|
| **Aug 15–21** | Performance profiling for L2 cache hit rates (Nsight Compute). | 20h | Cache throughput analysis. |
| **Aug 22–31** | Benchmarking against DISTWAR and baseline 3DGS. | 20h | Empirical speedup data. |
| **Sept 1–7** | Documentation: C++ and Thrust pattern guides for users. | 15h | Developer tutorials. |
| **Sept 8–14** | Integration with Clad's test suite and cross-platform CI. | 20h | Finalized PRs and bug fixes. |
| **Sept 15–21** | Draft final Technical Report and future work roadmap. | 15h | Technical documentation. |
| **Sept 22–30** | Final code cleanup, submission, and project presentation. | 20h | Final GSoC Submission. |
| **Total** | | 350h | |

---

## 4. Final Deliverables

1. **Code Contributions:**
    - Merged differentiable geometry path tracer branch demonstrating compiler-generated lock-free AD.

- Forked diff-gaussian-rasterization repository featuring a deterministically sorted forward pass built on Thrust and a Clad-synthesized backward pass.
- Extensions to Clad enabling differentiation through C++ concurrency primitives (atomics, parallel algorithms, execution policies)

2. **Testing Infrastructure:**
   - Finite difference validation suite ensuring numerical stability of lock-free gradients.
3. **Documentation:**
   - Developer guidelines on structuring C++ CUDA code to leverage Clad's liveness analysis and avoid atomic emission.

4. **Performance Benchmarks:**
   - Comparative analysis measuring training Frames Per Second (FPS) and memory utilization against standard 3DGS and DISTWAR implementations.

5. **Final Report & Presentation:**
   - Detailed technical report highlighting design decisions, challenges, and results.
   - Presentation with project highlights and future directions.

# 5. Qualifications

I am a graduate student pursuing a Master's in Applied Computing, specializing in Machine Learning and Parallel Programming. With a strong foundation in software development and optimization, I have 1.5 years of experience as a Machine Learning Engineer, where I focused on building and refining models for real-world applications.

- **Clad & Thrust API Contributions**:
  - I successfully completed my Google Summer of Code 2025 project on Supporting Thrust API in Clad. I merged 16 pull requests that extended Clad's capabilities to differentiate through NVIDIA's Thrust parallel algorithms library (including thrust::reduce, thrust::transform, etc.). This deep familiarity with Clad's internals makes me uniquely qualified to build a 3DGS pipeline leveraging these differentiated primitives.

- **Differentiable Rendering**:
  - I am currently contributing to Clad through an active pull request: PR #1789
  - This PR introduces differentiable rendering demos using a modified SmallPT path tracer with soft rasterization, enabling gradients to flow from image pixels to scene parameters via an MSE loss.
  - The demos cover:
    - Camera pose optimization
    - Object position recovery
    - Light source estimation
  - I also developed visualization tools to analyze optimization convergence.

    - This work provides a validated differentiable rendering pipeline in Clad and serves as a foundation for the proposed extension to parallel and concurrency-aware automatic differentiation.

- **Machine Learning Expertise:**
    - 1.5 years of experience as a Machine Learning Engineer, focusing on developing and optimizing models for real-world applications.
    - **Education:** Pursuing a Master's in Applied Computing (MAC), specializing in Machine Learning and Parallel Programming.

- **Parallel Programming Experience:**
    - Worked on a parallel programming project implementing Sobel edge detection using multiple paradigms (Github Repository).
    - Implemented and compared MPI, OpenMP, and hybrid approaches.
    - This experience with GPU programming patterns and memory optimization is directly applicable to implementing efficient custom derivatives for Thrust operations.

- **Availability:**
  I am fully committed to this project throughout the summer. I can dedicate at least 30 hours per week consistently from May through November. Additionally, I have no other commitments after April, so I am able to extend my availability if needed.