Project Title: Support usage of Thrust API in Clad

Project Parameters:

- **Duration**: June-September (4 months)
- **Total Hours**: 350 hours (~22 hours/week)
- **Focus**: Support usage of Thrust API in Clad

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, with support for NVIDIA's Thrust library. By enabling differentiation of Thrust's GPU-parallel algorithms, Clad users will gain the ability to automatically generate gradients for CUDA-accelerated code in scientific computing and machine learning applications. The implementation will include extending Clad's source-to-source transformation engine to recognize Thrust primitives (e.g., `transform`, `reduce`), implement custom derivatives, and validate performance through real-world use cases. This work will bridge the gap between high-performance GPU computing and AD, potentially accelerating gradient-based optimization tasks by orders of magnitude.

Key Challenges

1. **Parallelism-Aware Differentiation**: Handling data dependencies in parallel primitives like `thrust::reduce`
2. **GPU Memory Management**: Propagating adjoints through Thrust's device vectors and iterators
3. **Performance Preservation**: Ensuring generated derivatives maintain Thrust's execution efficiency

---

2. Technical Approach

Community Bonding Period (May, pre-coding)

Relationship Building and Environment Setup

During the three-week Community Bonding period before the official coding begins, I'll focus on establishing relationships with mentors and the Clad community. This period will be crucial for aligning expectations, understanding team workflows, and planning the project in detail.

I'll set up a complete development environment with the LLVM/Clang toolchain, Clad source code, Thrust libraries, and appropriate GPU development tools. Working closely with mentors, I'll ensure my development environment matches the project standards and allows effective testing and contribution.

Proof-of-Concept Implementation

A critical goal during the Community Bonding period will be to manually implement and test both the forward computations and their corresponding hand-written derivatives for key Thrust primitives. Rather than immediately integrating with Clad, I'll first create standalone programs that:

1. Implement forward operations using Thrust functions
2. Manually code the corresponding gradient functions
3. Test that these hand-written derivatives produce correct results

An example program for `thrust::transform` with a square function can look like:

```
// Manual forward computation
thrust::device_vector<float> forward_square(const
thrust::device_vector<float>& input) {
    thrust::device_vector<float> output(input.size());
    thrust::transform(input.begin(), input.end(), output.begin(),
                     [](float x) { return x * x; });
    return output;
}

// Hand written derivative function
void backward_square(
    const thrust::device_vector<float>& input,
    const thrust::device_vector<float>& d_output,
    thrust::device_vector<float>& d_input) {

    // Apply chain rule: d_input = d_output * 2x
    thrust::transform(
        d_output.begin(), d_output.end(),
        input.begin(), d_input.begin(),
        [](float d_out, float x) { return d_out * 2.0f * x; });
}

// Numerical finite difference approximation for verification
thrust::device_vector<float> finite_diff_square(const
thrust::device_vector<float>& input, float epsilon = 1e-6) {
    thrust::device_vector<float> gradients(input.size());
    thrust::host_vector<float> h_input = input;

    for (size_t i = 0; i < input.size(); i++) {
        // Forward difference
        thrust::host_vector<float> input_plus = h_input;
        input_plus[i] += epsilon;

        // Convert back to device
        thrust::device_vector<float> d_input_plus = input_plus;

        // Forward evaluations
        float val_plus =
thrust::reduce(forward_square(d_input_plus).begin(),
forward_square(d_input_plus).end());

        float val = thrust::reduce(forward_square(input).begin(),
                                  forward_square(input).end());

        // Finite difference approximation
        gradients[i] = (val_plus - val) / epsilon;
    }

    return gradients;
}

// Test function
```

```
void test_square_differentiation() {
    // Create test data
    thrust::device_vector<float> input(5, 2.0f); // (all 2.0)

    // Forward pass
    auto output = forward_square(input);

    // Create artificial gradient (all 1.0)
    thrust::device_vector<float> d_output(5, 1.0f);
    thrust::device_vector<float> d_input(5);

    // Backward pass with our manual gradient
    backward_square(input, d_output, d_input);

    // Compute numerical gradients for comparison
    auto numerical_grads = finite_diff_square(input);

    // Verify gradients
    thrust::host_vector<float> h_d_input = d_input;
    thrust::host_vector<float> h_numerical = numerical_grads;

    std::cout << "Comparing analytical vs numerical gradients:" <<
std::endl;
    for (int i = 0; i < h_d_input.size(); i++) {
        std::cout << "  Element " << i << ": analytical = " <<
h_d_input[i]
                  << ", numerical = " << h_numerical[i] << std::endl;

        // Verify analytical gradient matches numerical approximation
        assert(fabs(h_d_input[i] - h_numerical[i]) < 1e-4);
    }
    std::cout << "Square gradient test passed" << std::endl;
}
```

This approach separates understanding mathematics from the Clad integration challenges, allowing me to focus on one aspect at a time. The validated manual implementations will serve as specifications for the custom derivatives in Clad, making the integration more straightforward.

Benchmark Development Plan

To drive implementation and demonstrate real-world value, I'll research existing open source projects that utilize the Thrust API for scientific and machine learning applications. This research will focus specifically on how real-world applications implement differentiable algorithms using Thrust primitives, helping to identify common patterns and operations that should be prioritized for differentiation support.

Based on this research, I'll develop these benchmarks in order of increasing complexity:

1. **Vector Norm Computation**: Implementing L2 norm calculation using `transform_reduce` to compute the square root of sum of squares. This tests differentiation through both reduction and element-wise operations.

2. **Logistic Regression Training**: A machine learning example using gradient descent to optimize weights, with a binary cross-entropy loss function. This demonstrates differentiation through a realistic ML training workflow.

3. **Neural Network Layer**: A dense layer implementation with matrix multiplication and activation functions, demonstrating Clad's ability to differentiate through user-defined types and complex compositions of Thrust operations.

These benchmarks will serve multiple purposes:
- Guide implementation priorities based on real-world needs
- Validate correctness of automatically generated derivatives
- Measure performance improvements from GPU acceleration
- Demonstrate the value of Thrust+Clad integration

By focusing on these practical examples throughout development, I'll ensure the implementation addresses real-world use cases in machine learning, scientific computing, and other computational domains.

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

Clad Differentiation Architecture Analysis (30 hours)

This phase involves a thorough analysis of Clad's existing architecture for handling external function calls and CUDA operations. Understanding how Clad currently processes function calls in its visitors is essential for extending support to Thrust functions.

The first key aspect to investigate is Clad's call expression handling through the `VisitCallExpr` method in `ReverseModeVisitor.cpp`. This method determines how derivatives are generated for function calls, with patterns for handling special cases:

```
// From ReverseModeVisitor.cpp
if (!FD) {
  // How Clad currently handles unknown functions
}

if (FD->getNameAsString() == "printf" || FD->getNameAsString() ==
"fprintf") {
  // Pattern for special function handling
}
```

This code illustrates how Clad identifies specific functions for special treatment. We'll need to extend this pattern to recognize Thrust API calls such as `thrust::transform` and `thrust::reduce`.

Next, we'll examine Clad's existing CUDA support to understand the interface between GPU operations and derivative computation:

```
if (shouldUseCudaAtomicOps(base)) {
  Expr* atomicCall = BuildCallToCudaAtomicAdd(it->second, dfdx());
  addToCurrentBlock(atomicCall, direction::reverse);
}
```

This snippet shows how Clad handles atomic operations on CUDA devices. Understanding this mechanism is crucial because Thrust operations often involve parallel computation on GPU, requiring similar synchronization considerations. We'll need to ensure our implementation properly handles race conditions during gradient accumulation in parallel environments.

Thrust API Analysis and Prioritization (30 hours)

In this phase, we'll perform a detailed analysis of the Thrust API to identify which functions are most valuable for automatic differentiation in scientific computing and machine learning contexts. Thrust provides numerous parallel algorithms grouped into several categories, including transformations, reductions, prefix sums, reordering operations, and searching algorithms.

Transformations like `thrust::transform` apply a function to each element of a sequence, making them fundamental building blocks for many numerical algorithms. Reductions like `thrust::reduce` combine elements using binary operations, often appearing at the end of loss function computations in machine learning models. Prefix sums through operations like `thrust::inclusive_scan` compute cumulative operations and are essential in many dynamic programming and parallel algorithms.

Our prioritization will consider differentiability, usage frequency, and implementation complexity. Some operations, like sorting, have discontinuities that complicate differentiation, while others like element-wise transformations follow clear differentiation patterns. We'll focus first on operations commonly used in differentiable algorithms, balancing implementation effort with value to users.

Implementation Strategy Development (20 hours)

Based on the analysis and the proof-of-concept developed during Community Bonding, this phase will refine the implementation strategy for the project. We'll design the architecture for Thrust function recognition in Clad's visitors and create a framework for implementing custom derivatives for Thrust operations.

Key tasks include:

- Defining the mechanism for recognizing Thrust function calls
- Creating a template for implementing custom derivatives for Thrust algorithms
- Designing the approach for differentiating user-defined functors passed to Thrust

Building on the lessons from the proof-of-concept, we'll create a consistent approach for implementing the derivatives of various Thrust algorithms, ensuring that the implementation is both mathematically correct and computationally efficient.

Example:

A simpler example to understand automatic differentiation with Thrust is the `thrust::reduce` operation, which is both fundamental and has a straightforward derivative calculation.

In the forward pass, `thrust::reduce` combines all elements in an array using a binary operation:

```
// Sum all elements in a vector
float total = thrust::reduce(input.begin(), input.end(), 0.0f,
thrust::plus<float>());
```

This is equivalent to:

```
total = input[0] + input[1] + ... + input[n-1]
```

Simple Derivative Rule

The derivative of a sum with respect to each of its inputs is exactly 1.0. This is a fundamental rule in calculus:

$$\partial(x_1 + x_2 + ... + x_n)/\partial x_i = 1 \text{ for all } i$$

This means that when we compute the backward pass, we simply need to distribute the output gradient equally to all input elements.

Implementation Example

For a concrete example, consider computing the gradient of a function that sums all elements of a vector:

```
float sum_all(thrust::device_vector<float>& x) {
    return thrust::reduce(x.begin(), x.end(), 0.0f,
thrust::plus<float>());
}
```

When we call `clad::gradient(sum_all, input)`, the derivative implementation needs to:

```
namespace clad {
namespace custom_derivatives {
  template <typename InputIt, typename T>
  void thrust_reduce_sum_pullback(
      InputIt input_begin, InputIt input_end,
      T init,
      T d_output,  // The gradient flowing back from the output
      InputIt d_input_begin) {
    size_t n = std::distance(input_begin, input_end);
    // For sum reduction, simply copy the output gradient to all inputs
    thrust::fill(d_input_begin, d_input_begin + n, d_output);
  }
}
}
```

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

Thrust Function Recognition in Clad (30 hours)

This phase focuses on extending Clad to recognize and process Thrust function calls during the differentiation process. For Thrust support, we would create a `ThrustBuiltins.h` header that users can include to enable Thrust differentiation.

For example, we'll need to extend the namespace with logic like:

```
#ifndef CLAD_DIFFERENTIATOR_THRUSTBUILTINS_H
#define CLAD_DIFFERENTIATOR_THRUSTBUILTINS_H

#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/transform.h>
#include <thrust/reduce.h>
// Other Thrust headers as needed
```

```
#include "clad/Differentiator/Differentiator.h"

namespace clad::custom_derivatives {
namespace thrust {
  // Custom derivative for thrust::reduce (with sum)
  template <typename InputIt, typename T>
  void reduce_pullback(
      InputIt input_begin, InputIt input_end,
      T init, T d_output,
      InputIt d_input_begin) {

    // For sum reduction, gradient flows equally to all inputs
    size_t n = std::distance(input_begin, input_end);
    ::thrust::fill(d_input_begin, d_input_begin + n, d_output);
  }
  // Additional specialized overloads for other reduction operations
  // ...
}
}

#endif // CLAD_DIFFERENTIATOR_THRUSTBUILTINS_H
```

This code illustrates how we'll identify Thrust functions by name and dispatch to specialized handlers for each supported function. The actual implementation will include mechanisms for template parameter extraction and handling various calling patterns.

Basic Thrust Algorithms Support (50 hours)

This phase will implement custom derivative handlers for the fundamental Thrust algorithms, starting with the most widely used parallel operations. For each operation, we need to:

1. **Define the mathematical model** of the forward and reverse operations
2. **Create custom derivative implementation** in the `clad::custom_derivatives` namespace
3. **Ensure efficient memory management** for intermediates and adjoints
4. **Test correctness** against finite difference approximations

The implementation of `thrust::transform` differentiation provides a good example:

```
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) {
    // Implementation using thrust::transform for the derivative
    thrust::transform(
        d_output_begin, d_output_begin + (input_end - input_begin),
        input_begin, d_input_begin,
        [op](const auto& d_out, const auto& x) {
          // Apply chain rule using derivative of the operation
          auto derivative = derivative_of(op, x);
          return d_out * derivative;
        });
  }
}}
```

This implementation demonstrates how we apply the chain rule in a parallel context. The key insight is that for elementwise operations like `transform`, the derivative is also elementwise, making it natural to express using the same parallel pattern.

Advanced Thrust Algorithm Support (40 hours)

After implementing basic operations, this phase will tackle more complex Thrust algorithms that require sophisticated differentiation approaches. These include reduction operations (`thrust::reduce`, `thrust::transform_reduce`), scanning operations (`thrust::inclusive_scan`, `thrust::exclusive_scan`), and reordering operations with continuous approximations where needed.

For example, differentiating `thrust::reduce` requires understanding that the backward pass must distribute the output gradient to all input elements according to the reduction operation. For a simple sum reduction:

```
// Forward pass
float result = thrust::reduce(input.begin(), input.end());
// Backward pass must distribute gradient to all inputs
thrust::fill(d_input.begin(), d_input.end(), d_output);
```

For more complex reductions, the gradient distribution depends on the reduction operation. This requires careful mathematical analysis and implementation. Other operations like scans present unique challenges because the output at each position depends on multiple input elements, creating more complex dependency patterns in the derivative computation.

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

Unit Testing Framework (30 hours)

A robust testing framework is essential to ensure the correctness and performance of Thrust differentiation in Clad. This phase will create correctness verification comparing

Clad-generated derivatives against finite difference approximations, performance benchmarks measuring speedup from GPU-accelerated differentiation, and edge case testing ensuring proper handling of corner cases like empty containers.

For example, a test for `thrust::transform` differentiation might look like:

```
TEST(ThrustDerivatives, TransformPullback) {
  // Setup input data
  thrust::device_vector<float> input(100, 1.0f);

  // Define function using thrust::transform
  auto f = [](thrust::device_vector<float>& x) {
    thrust::device_vector<float> y(x.size());
    thrust::transform(x.begin(), x.end(), y.begin(),
                     [](float val) { return val * val; });
    return thrust::reduce(y.begin(), y.end());
  };

  // Compute gradient using Clad
  auto result = clad::gradient(f, input);

  // For x=1, derivative of x² should be 2x = 2
  for (size_t i = 0; i < input.size(); ++i) {
    EXPECT_NEAR(result[i], 2.0f, 1e-5);
  }
}
```

This test illustrates a common pattern: applying a transformation followed by a reduction, which is fundamental in many machine learning loss functions. The test verifies that Clad correctly applies the chain rule across both operations. We'll develop a comprehensive suite of such tests covering all supported Thrust operations and their combinations, ensuring that the differentiation is correct across a wide range of use cases.

Real-world Integration Examples (50 hours)

To demonstrate the practical value of Thrust support in Clad, this phase will develop several real-world examples, including neural network training and optimization algorithms. These examples will not only serve as validation of the implementation but also as educational resources for users.

For each example, we'll explain the mathematical principles of the algorithm, show the implementation using Thrust and Clad, and demonstrate performance benefits compared to CPU-only or manually-derived approaches. For instance, a simple neural network implementation would demonstrate how Thrust operations can be composed to implement forward and backward passes through network layers, with Clad automatically generating the correct derivatives.

These examples will serve both as validation of our implementation and as documentation for users. By showing how Thrust and Clad can be used together in realistic scenarios, we'll help users understand how to apply these tools to their own problems. The performance comparisons will demonstrate the value proposition of GPU-accelerated automatic differentiation, showing how it can accelerate scientific computing and machine learning workloads.

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

Comprehensive Documentation (30 hours)

Clear, comprehensive documentation is essential for adoption of the Thrust support in Clad. This phase will create API documentation for all supported Thrust functions, mathematical background explaining the differentiation principles, usage examples showing how to leverage Thrust+Clad in real applications, and performance guidelines for optimal usage.

For example, the documentation for `thrust::transform` differentiation would explain: "The `thrust::transform` operation applies a unary function to each element of an input range. When differentiated, Clad applies the chain rule by computing the derivative of the unary function at each input point and multiplying by the corresponding output gradient. This operation efficiently parallelizes on GPU, providing substantial speedup for large datasets.

```
// Example: Computing gradient of a transform operation
thrust::device_vector<float> x(1000, 2.0f);  // Initialize with value
2.0
// Define a function using thrust::transform
auto square_sum = [](const thrust::device_vector<float>& input) {
  thrust::device_vector<float> squared(input.size());
  thrust::transform(input.begin(), input.end(), squared.begin(),
                   [](float x) { return x * x; });  // Square each
element
  return thrust::reduce(squared.begin(), squared.end());  // Sum the
squares
};

// Compute gradient using Clad
auto gradient = clad::gradient(square_sum, x);
// Result: each element of gradient will be 2*x[i] = 4.0
```

In this example, Clad automatically computes the derivative of the lambda function using chain rule, and the resulting gradient correctly reflects that the derivative of x^2 is 2x."

We'll create similar documentation for all supported functions, ensuring that users understand both the mathematical principles and the practical usage patterns. This documentation will be integrated with Clad's existing documentation system.

Final Integration and Testing (25 hours)

This phase involves integrating all components into the main Clad codebase and performing comprehensive testing. We'll integrate our implementation with Clad's main test suite, perform cross-platform testing on various GPU architectures, set up continuous integration for ongoing testing, and conduct final performance benchmarking to document speedups.

Integration testing will ensure that the Thrust support works correctly alongside Clad's existing functionality, without regressions or conflicts. Cross-platform testing will verify that the implementation works correctly across different GPU vendors and compute capabilities. 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 GPU-accelerated differentiation can speed up common numerical computing workloads. These benchmarks will be included in the documentation to help users understand the performance characteristics of the system.

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, and the solutions developed. It will serve as a reference for future developers working on Clad or similar systems. The presentation materials will help communicate the value of the work to the broader scientific computing and machine learning communities.

The future work recommendations will identify opportunities for further enhancement of Thrust support in Clad, such as supporting additional algorithms, optimizing performance further, or integrating with other GPU computing frameworks. These recommendations will help guide the ongoing development of Clad.

3. Timeline

Dates	**Tasks**	**Hours**	**Deliverables**
June 1–7	Analyze ReverseModeVisitor.cpp for CUDA/Thrust compatibility	15h	Report on Clad's gradient accumulation patterns

Dates	Tasks	Hours	Deliverables
June 8–14	Prioritize Thrust functions (e.g., transform, reduce)	15h	List of Thrust APIs to support
June 15–20	Design API for Thrust differentiation in Clad	20h	Draft implementation strategy
June 21–27	Extend VisitCallExpr to recognize Thrust calls	15h	PR: Thrust function detection in Clad's AST
June 28–July 4	Implement derivatives for thrust::transform	25h	PR: Pushforward/pullback for transform with CUDA support
July 5–11	Add support for thrust::reduce	25h	PR: Gradient rules for reductions
July 12–18	Support composite operations (transform_reduce, inner_product)	20h	PR: Nested Thrust call handling
July 19–25	Optimize memory management for GPU gradients	20h	PR: CUDA memory utilities
July 26–31	Validate derivatives for complex Thrust pipelines	15h	Test suite for advanced Thrust algorithms
Buffer 1	**Aug 1–7** (Catch-up for Phase 2 delays)	–	–

Dates	Tasks	Hours	Deliverables
Aug 8–14	Design unit tests for Thrust derivatives	15h	Test cases for edge cases
Aug 15–21	Develop ML mini-app (e.g., GPU-based gradient descent)	25h	Demo: Differentiated optimization workflow
Aug 22–31	Integrate Thrust with Clad's CI/CD pipeline	20h	CI workflow for Thrust tests
Buffer 2	**Sept 1–7** (Catch-up for Phase 3 delays)	–	–
Sept 8–14	Write API documentation and user guide	15h	Docs: Tutorials for Thrust+Clad
Sept 15–21	Finalize code integration and edge-case tests	15h	PR: Bug fixes and stability improvements
Sept 22–30	Prepare final report, submit code/docs, present findings	20h	Final submission: Code, benchmarks, and presentation

---

4. Final Deliverables

1. **Code Contributions:**
   - Thrust function support integrated into Clad.
2. **Testing Infrastructure:**
   - 50+ unit tests covering various Thrust algorithms.

    - CI/CD pipeline with GPU nodes.
3. **Documentation:**
    - API reference for Thrust support in Clad.
    - User guide with code examples and tutorials.
4. **Performance Benchmarks:**
    - Comparative analysis between Clad+Thrust and baseline approaches.
    - Case study on GPU-accelerated gradient-based tasks.
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 Contributions**:
    - PR #1237 : Add new custom derivatives for math functions
    - PR #1236 : Update Developer Installation Instruction on README.md.
- **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:**
    - Recently completed a parallel programming project implementing Sobel edge detection using multiple paradigms (Github Repository).
    - Gained hands-on experience with CUDA GPU programming, including thread grid optimization, and shared memory techniques.
    - 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 24 hours per week consistently from May through September. My summer semester has no course load specifically to accommodate this project.