# Enhancing LLM Training Efficiency with Clad for Automatic Differentiation in C++

**Rohan Timmaraju**[1]

[1]Columbia Unversity

## Contents

## §1. Project Details

- Organization: CERN-HSF, Compiler Research Group.
- Mentors: Vassil Vassilev, David Lange
- Duration: 350 hours

### §1.1. Contact Details

- Github: https://github.com/Rohan-T144
- Email: rohan.timmaraju@gmail.com

## §2. Abstract

Large Language Models (LLMs) have revolutionized AI, but their training is computationally intensive, often bottlenecked by Python frameworks. This project proposes to enhance LLM training efficiency by integrating Clad, a Clang-based automatic differentiation (AD) plugin, within a C++ environment. We will develop and utilize `cladtorch`, a custom C++ tensor library inspired by `llm.c` and `pytorch` but optimized for Clad integration. The core goal is to replace manual or internally managed gradient computations in `cladtorch` with Clad's compiler-level reverse-mode AD for key LLM operations (e.g., GPT-2 layers and loss functions). This involves adapting the tensor library's architecture and operations to interface effectively with Clad, potentially enhancing Clad itself if necessary. We will benchmark the performance (e.g. speed, memory) gains against the non-Clad baseline, quantifying the resulting efficiency and performance improvements. The project includes implementing GPT-2 training, integrating Clad, extensive testing, benchmarking, parallelization via OpenMP, and exploring further LLMs such as the Llama architecture. This work contributes to high-performance scientific computing and provides a basis for a more efficient, compiler-driven approach to automatic differentiation in computationally demanding ML tasks.

## §3. Benefits to Community

- **To CERN-HSF & Scientific Computing:** Provides a potential pathway for more efficient training of ML models within C++ ecosystems common in High-Performance Computing (HPC) and scientific research. Optimizing differentiation, a core ML bottleneck, aligns with HSF's focus on performant software for scientific discovery.
- **To the Clad/LLVM Community:** Offers a significant, real-world use case for Clad in the complex domain of deep learning. The project will stress-test Clad, potentially uncovering bugs and identifying areas for future development to better support ML workloads. Provides a reference implementation and potentially a useful framework for using Clad in ML training.
- **To the Open Source ML Community:** Introduces `cladtorch` as a potentially efficient, Clad-optimized C++ tensor library. Demonstrates the feasibility and performance characteristics of compiler-based AD for LLM training, potentially inspiring alternative approaches to Python-centric frameworks, especially for resource-constrained environments or performance-critical applications.

## §4. Introduction & Motivation

The training of state-of-the-art LLMs like GPT-2 and Llama demands vast computational resources. While Python frameworks (PyTorch, TensorFlow) dominate the landscape due to their flexibility and rich ecosystems (e.g., Hugging Face), their reliance on interpreted execution and dynamic computation graphs often leads to performance bottlenecks and high memory consumption. This limits the feasibility of training and deploying large models, especially in resource-constrained settings like embedded systems or HPC environments.

C++ offers a path to higher performance through closer control over hardware and memory, as demonstrated by efficient inference projects like `llama.cpp`. However, a crucial component for training deep learning models is efficient Automatic Differentiation (AD), particularly reverse-mode AD to compute gradients for backpropagation.

This project tackles this challenge by integrating Clad into a C++ LLM training pipeline. Clad performs differentiation at compile time, analyzing C++ code (specifically, the Clang AST) and generating optimized derivative code directly. This approach has the potential to significantly reduce the runtime overhead and memory footprint associated with AD compared to runtime graph-based or operator-overloading techniques common in Python frameworks or some C++ libraries.

The project will build **`cladtorch`**, a custom C++ tensor library that I am currently developing as part of the evaluation task. Inspired by the minimalistic approaches of `llm.c` and `pytorch`, `cladtorch` will be designed to facilitate integration with Clad. We aim to use Clad to compute gradients for critical components of LLM training (e.g., matrix multiplications, activation functions, normalization layers, loss functions) within `cladtorch`.

## §5. Background

Clad is an Automatic Differentiation plugin for the Clang compiler, built upon the LLVM infrastructure. Unlike many AD libraries that rely on runtime techniques like operator overloading or dynamic computation graphs, Clad employs a source-to-source transformation approach. It analyzes the Abstract Syntax Tree (AST) of C++ code and generates optimized C++ code for derivative computations. Clad supports both forward (`clad::differentiate`) and reverse (`clad::gradient`, `clad::jacobian`) modes. For the purposes of efficient neural network training, reverse-mode AD through `clad::gradient` is particularly relevant.

The interface of `clad::gradient` requires a single function and adheres to specific signature patterns. When differentiating a C++ function that takes pointer or array arguments (which in our case would represent tensor data in low-level implementations), Clad typically generates a derivative function that also takes pointer arguments to store the computed gradients. Furthermore, it is only usable on a function returning a scalar value (e.g., a loss function). This interface necessitates careful consideration when integrating Clad with tensor library operations.

Several projects have explored building machine learning models (LLMs in particular) in C and C++ to achieve greater control over performance and resource utilization, which we can take inspiration from:

- `llm.c` (by Andrej Karpathy): This project provides a minimalist, educational implementation of GPT-2 training in pure C. It implements forward and backward propagation for key components from scratch and using manually derived gradients. This provides a foundational structure around which we can base our LLM training infrastructure.
- `llm.cpp (tinytorch)`: A C++ port of `llm.c`, providing a single-header tensor library with its own internal AD mechanism based on building a computation graph during the forward pass and traversing it backward for reverse mode AD. For `cladtorch`, we want to build a minimal tensor library with efficient tensor operations based on a computational graph, so this is a great reference implementation.
- `ggml`: A tensor library focusing on C, used in `llama.cpp`, employing techniques like quantization and optimized CPU kernels. Widely adopted in the ML community for its efficiency and flexibility in inference for local models.

### §5.1. Target LLM Architectures

This project focuses predominantly on GPT-2. Developed by OpenAI, it is a decoder-only transformer model. Its architecture consists of token and positional embeddings, followed by a stack of transformer blocks. Each block incorporates multi-head self-attention mechanisms, layer normalization, and feed-forward networks with residual connections. GPT-2 operates autoregressively, predicting the next token based on the preceding context, and utilizes masked attention during training to enforce this autoregressive behavior.

Once the infrastructure for training GPT-2 utilizing clad is completed, modifying it to support additional LLM architectures such as Llama will leverage the established infrastructure, making adaptations for Llama's specific architectural elements (like RMSNorm, SwiGLU, RoPE) more manageable. The Llama family, introduced by Meta AI, is a more recent advancement in open source LLMs, with strong performance for low compute costs. Llama incorporates improvements such as the SwiGLU activation function, RMSNorm for normalization, and Rotary Positional Embeddings, contributing to efficiency and performance.

## §6. Implementation Plan

This plan outlines the key phases for integrating Clad into the `cladtorch`–based LLM training workflow, focusing on incremental development and explicitly addressing the core challenges of applying compiler-based AD in this domain.

### §6.1. Phase 1: Foundation

- **Develop/Refine `cladtorch`:** Solidify the `cladtorch` C++ tensor library, ensuring core operations (Tensor class, memory management, basic math/NN ops like MatMul, activations) are implemented with Clad compatibility in mind. This includes designing pointer-based interfaces for core computations to align with Clad's requirements.

- **Implement Baseline GPT-2:** Build and verify a complete GPT-2 (small model) training loop using `cladtorch`. This will include both forward and backward passes, initially implemented **without** Clad, using manual differentiation or techniques similar to `tinytorch`/`llm.c`.
- **Establish Benchmarks:** Measure and document the performance (speed, memory usage) and numerical correctness of this non-Clad baseline implementation. This baseline will serve as the crucial reference point for quantifying the impact of Clad integration.

## §6.2. Phase 2: Core Clad Integration Strategy Investigation

This phase focuses on investigating and developing the optimal strategy for integrating Clad's static, compile-time automatic differentiation with the dynamic computations inherent in the computational tensor graphs of neural network training. The ultimate goal is to develop a robust and potentially generalizable approach (represented by Strategy C), while using simpler strategies (A and B) as foundational experiments to understand Clad's behavior, limitations, and requirements in the context of our specific codebase and LLM workloads. This investigation will directly inform the design of the `cladtorch`–Clad interface and highlight potential areas where Clad itself might be enhanced for deep learning use cases.

- ***Strategy A*** *(Whole-Function Differentiation – Initial Feasibility Test):* We will first attempt to apply `clad::gradient` directly to the top-level function computing the scalar loss (`float compute_loss(parameters, data)`). This serves as an experiment to assess Clad's current ability to differentiate through the entire network's forward pass, involving deep function calls and complex data dependencies within `cladtorch`. Success or failure here, along with performance characteristics, will provide valuable insights into Clad's handling of large-scale computations and guide subsequent efforts. This is not the ideal end state as it requires inconvenient rearchitecting of the neural network implementation itself. This may require significant refactoring and its feasibility is a key point of investigation.

- ***Strategy B*** *(Layer-wise Differentiation – Modular Analysis & Fallback):* Concurrently or subsequently, we will investigate applying Clad to compute local derivatives for individual layers or core operations (e.g., MatMul, activation functions) using wrapper functions. While requiring manual implementation of the chain rule within `cladtorch`'s backward pass, this strategy allows us to:
  - Validate Clad's correctness and performance on specific, isolated computational units.
  - Gain practical experience integrating Clad-generated code into the backward pass.
  - Establish a potential fallback mechanism if whole-function differentiation proves impractical with current Clad capabilities.

- ***Strategy C*** *(Bridging Static Analysis and Runtime Graphs):* This is the primary strategy, aiming for a more robust and potentially generalizable solution. Informed directly by the findings from Strategies A and B (e.g., understanding which patterns Clad handles well/poorly, performance on key ops), we will actively develop and evaluate methods to bridge the gap between Clad's static analysis and `cladtorch`'s runtime computational graph (which is stored via pointers). This potentially involves exploring and implementing adaptations like:
  - Creating "static subgraphs": Bundling sequences of operations (e.g. layers or attention mechanisms) into larger, self-contained C++ functions or function objects that present a more analyzable unit to Clad at compile time.
  - Designing `cladtorch` operations as function objects with Clad-differentiable `forward()` methods.
  - This investigation will explicitly aim to identify potential features, limitations, or necessary extensions within Clad itself that would better support differentiation of tensor operations and potentially dynamic computational structures common in deep learning frameworks.

- **Initial Clad Application:** We will begin implementing Clad integration using simpler operations (e.g., activations, element-wise ops) likely applying Strategy A or B first to establish a working integration and validation framework.

- **Validate Correctness:** Develop rigorous unit tests throughout this phase to compare Clad-generated gradients against the non-Clad baseline, ensuring numerical accuracy and stability for each tested strategy and operation.
- The goal is to use this investigative phase to converge on the most practical and performant integration method achievable within the project scope, while documenting the rationale and trade-offs, and identifying areas for future work.

### §6.3. Phase 3: Expanding Clad Integration to Core Layers

- **Apply Strategy to Key Layers:** Systematically extend Clad integration (using the validated strategy from Phase 2) to the core, computationally significant components of the GPT-2 architecture. This includes Matrix Multiplications (within MLP and Attention), Softmax, and Layer Normalization.
- **Refine `cladtorch` and Wrappers:** Adapt `cladtorch` operation implementations, Clad wrapper functions, or the chosen integration strategy based on the experience gained integrating more complex layers. Address any performance bottlenecks or compatibility issues that arise during this phase.
- **Layer-Level Benchmarking:** Measure the performance impact of Clad integration at the layer or block level (e.g., benchmarking MLP forward and backward pass times with and without Clad).

### §6.4. Phase 4: Full Model Integration, Benchmarking, and Optimization

- **End-to-End Integration and Validation:** Integrate the chosen Clad strategy across the entire GPT-2 model, ensuring seamless functionality and numerical correctness throughout the complete training loop, including the loss function computation and gradient propagation.
- **Benchmarking:** Perform detailed performance comparisons of the fully Clad-integrated GPT-2 training against the non-Clad baseline. Metrics will include training time per epoch, throughput (tokens per second), and peak memory usage.
- **Profiling and Optimization:** Utilize profiling tools to identify performance bottlenecks within the Clad-integrated training process. Explore potential optimizations within `cladtorch`'s code structure, memory management, or the Clad integration approach to maximize performance.
- **Parallelization with OpenMP:** Integrate OpenMP pragmas to parallelize computationally intensive sections of both the forward and backward passes (including Clad-generated derivative code where applicable). Evaluate the performance improvements achieved through OpenMP parallelization.

### §6.5. Phase 5: Documentation and Extension

- **Document Findings and Analysis:** Thoroughly document the chosen Clad integration strategy and its rationale, the specific modifications made to `cladtorch`, detailed performance benchmark results, challenges encountered (particularly those related to bridging static AD with dynamic graph concepts), and any identified limitations of Clad or potential areas for future Clad enhancements.
- **Explore Further (Stretch Goals):** If time permits, explore stretch goals such as extending Clad integration to the Llama architecture, investigating integration with other tensor libraries like Eigen, or further generalizing `cladtorch` to be a more broadly applicable Clad-optimized tensor library.

## §7. Performance Evaluation

- **Baseline:** The `cladtorch`–based GPT-2 training implementation without Clad AD (using its own internal/manual gradient calculation, similar to `tinytorch`/`llm.c`).
- **Metrics:**
  - **Wall Clock Time:** Time per training step/epoch.
  - **Throughput:** Tokens processed per second.
  - **Memory Usage:** Peak resident memory during training.
  - **Numerical Stability:** Compare gradient values with baseline/numerical differentiation for correctness.

- **Method:** Compare the Clad-integrated version against the baseline on standardized hardware and datasets (e.g., TinyShakespeare) for the GPT-2 model. Analyze profiling results to pinpoint where Clad provides benefits or introduces overhead.
- Use OpenMP pragmas (`#pragma omp parallel for`) within the core computational kernels (both forward and Clad-generated backward passes where applicable). This is expected to provide significant speedups, building on the approach seen in `llm.c`.

## §8. Detailed Timeline

- Community Bonding Period (Before Coding Starts):
  - Deepen understanding of Clad's internals, API, and limitations for AD in ML.
  - Refine cladtorch structure based on Clad compatibility requirements.
  - Finalize detailed plan for Phase 1 & 2. Set up development environment thoroughly (Clang, Clad, CMake).
  - Engage with mentors, clarify expectations.
- Week 1-2: Foundation & Baseline
  - Solidify cladtorch implementation (Tensor class, core ops like MatMul, GELU).
  - Implement GPT-2 forward pass in cladtorch.
  - Implement backward pass *without* Clad (manual/internal AD).
  - Establish baseline benchmarks (speed, memory, correctness).
  - **Deliverable:** Working non-Clad GPT-2 training baseline; initial cladtorch library.
- Week 3-4: Core Clad Integration Strategy & Validation
  - Implement and evaluate potential Clad integration strategies as discussed above.
  - Choose the most promising strategy.
  - Apply chosen strategy to 1-2 simple cladtorch ops (e.g., activation, element-wise ops).
  - Develop unit tests comparing Clad vs. baseline gradients.
  - **Deliverable:** Decision on Clad integration strategy; initial Clad integration for simple ops validated.
- Week 5-6: Integrating Clad for Core Layers (MLP, Attention)
  - Integrate Clad (using chosen strategy) for MatMul within MLP and Attention.
  - Integrate Clad for Softmax.
  - Refine cladtorch/wrappers as needed. Debug integration issues.
  - Benchmark performance impact at the layer level.
  - **Deliverable:** Clad integrated for MLP & Attention computations; layer-level benchmarks if possible.
- Week 7: Integrating Clad for Remaining Layers & Loss
  - Integrate Clad for Layer Normalization and Cross-Entropy Loss.
  - Ensure end-to-end numerical correctness of the full backward pass with Clad.
  - **Deliverable:** Clad integrated for all key GPT-2 components.
- Week 8: Mid-term Evaluation & Initial Full Benchmarking
  - Prepare and complete Mid-term GSoC evaluation.
  - Conduct initial end-to-end benchmarks (Clad vs. baseline).
  - Profile the Clad-integrated version to identify initial bottlenecks.
  - Begin writing documentation.
  - **Deliverable:** Mid-term report/presentation; initial model benchmark results.
- Week 9: Optimization & Parallelization
  - Optimize cladtorch code and Clad integration based on profiling.
  - Integrate OpenMP pragmas for parallelization in key forward/backward kernels.
  - Benchmark the impact of OpenMP.
  - **Deliverable:** OpenMP integration complete; optimized performance benchmarks.
- Week 10: Stretch Goals & Documentation
  - Begin exploring stretch goals (e.g., Llama architecture elements, cladtorch generalization).

  - Focus heavily on writing detailed documentation (library, integration strategy, results).
  - Refine unit tests and examples.
- Week 11: Finalizing Code & Documentation
  - Complete any remaining implementation/debugging for core goals.
  - Finalize all documentation, ensuring clarity and completeness.
  - Clean up code, add comments, ensure the repository is well-organized.
  - Prepare final presentation materials.
- Week 12: Final Submission & Wrap-up
  - Submit final code and documentation.
  - Prepare and deliver final presentation/report for GSoC and CERN-HSF.
  - Ensure all project artifacts are accessible.

## §9. Expected Final Results

- A functional C++ tensor library (`cladtorch`) designed for and integrated with Clad.
- A C++ implementation for training a small GPT-2 model using `cladtorch`.
- Successful application of Clad (`clad::gradient`) to compute gradients for key layers (MLP, Attention, LayerNorm) and the loss function within the GPT-2 implementation.
- Identification of any necessary adaptations to `cladtorch` or potential enhancements/workarounds needed for Clad to handle the LLM training workload effectively.
- Comprehensive performance benchmarks comparing the Clad-integrated version against a non-Clad baseline, quantifying improvements in training speed and memory efficiency.
- Integration of OpenMP for multi-core parallelization and corresponding performance evaluation.
- (Stretch) Extension of the framework and Clad integration to handle elements of the Llama architecture.
- Thorough unit tests for numerical correctness and stability.
- Detailed documentation for the methodology, implementation details, `cladtorch`–Clad integration specifics, benchmark results, and findings.
- Presentations of progress and final results as required by GSoC and CERN-HSF.

### §9.1. Stretch Goals

- **Extend to Llama and beyond:** Fully adapt `cladtorch` and the Clad integration strategy to support training Llama architecture models and other local architectures (e.g. Gemma 2/3).
- **Generalize `cladtorch`:** Extend the tensor library to support a wider range of neural network operations and use cases beyond GPT/Llama to be a general-purpose solution specialized for Clad.
- **Explore Integration with Existing Libraries:** Investigate the feasibility and required Clad modifications (if any) to interface with established C++ tensor libraries like Eigen or explore bindings for PyTorch tensors.

## §10. About Me

I am currently a first-year undergraduate student pursuing a B.S. in Computer Science at Columbia University. I possess a strong academic foundation (GPA: 4.11) and an interest in high-performance computing, compiler design, and machine learning.

- **Relevant Experience:** My most significant experience comes from my Software Engineering Internship at Silicon Quantum Computing (SQC). There, I led the optimization and rewrite of their quantum compiler from Python to Rust, achieving over 80x speedup and simplifying the architecture. This involved working directly with low-level code, performance optimization, understanding complex hardware interactions, and becoming proficient in a large, complex codebase with minimal supervision – skills directly applicable to this project. I also have experience with C++ from competitive programming and embedded programming.

- **Machine Learning & LLMs:** I have hands-on experience with LLMs through projects involving fine-tuning for SQL query generation and developing document-based Retrieval-Augmented Generation (RAG) systems using models like Llama 3.1 and vector databases.
- **Programming & Technical Skills:** I am proficient in C++, Python, and Rust, along with C, Java, and several other languages. I am comfortable with Linux environments, Git, and algorithm design, further supported by my strong performance in competitive programming (e.g., 1st place in the Australian Invitational Informatics Olympiad).

**Motivation:** I am interested in the challenge of pushing performance boundaries in computationally intensive domains. Applying compiler-level tools like Clad to accelerate LLM training combines my interests perfectly. I am excited by the prospect of contributing to a cutting-edge tool like Clad, developing an efficient C++ ML library, and tackling the complex challenges involved. This project offers a unique opportunity to apply my skills in C++, optimization, and ML within the high-impact environment of CERN-HSF and the compiler research community.

**Availability:** I can dedicate the required 30-35 hours per week to this GSoC project throughout the entire coding period. I have no known conflicts or pre-planned absences during the summer and will be fully available to focus on this project. I am flexible with my schedule and will ensure I am readily available for communication and meetings with the mentors.