Accelerating LLM Training in C++ with Clad

GSoC 2025 Final Presentation

Rohan Timmaraju, October 2025
Compiler Research Group

The Challenge of LLM Training

- Large Language Models (LLMs) are computationally expensive to train.
- Python frameworks (PyTorch, TensorFlow) dominate but can have performance overhead, especially in C++-centric HPC environments.
- Goal: Leverage C++ performance and compiler-level Automatic Differentiation (AD) for more efficient LLM training.

Our Approach: `clad` for Backpropagation

- Idea: Implement the LLM entirely in C++, then use Clad — a Clang plugin for source-to-source AD — to automatically generate the gradient code (backpropagation) at compile time.
- Hypothesis: A static, compile-time approach can enable deeper compiler optimizations across the entire computation graph.

The Journey

Two Paths to Training

Phase 1: `cladtorch` (Flexibility First)	Phase 2: C-Style Engine (Performance First)
• Design: PyTorch-style, Object-Oriented API • Data: `Tensor` class • State: Encapsulated in objects with RAII & cleanup • Result: Functional but high overhead	• Design: `llm.c`-inspired, procedural • Data: Raw `float*` arrays • State: Manually managed in a struct • Result: Minimalist and extremely fast

Cladtorch

Cladtorch: C++ Tensor Operations

- `cladtorch` Library:
  - Successfully developed a custom C++ tensor library from the ground up.
  - Provides core tensor operations, neural network layers (Linear, LayerNorm, Softmax), and loss functions.
  - Designed specifically for optimal compatibility with Clad.
- GPT-2 Forward Pass:
  - Implemented a full GPT-2 model (125M parameters) using `cladtorch`.
  - The forward pass is functional and validates the library's correctness.
  - Achieves ~12 tokens/second for inference on a single CPU core.

Cladtorch: End-to-End Differentiation

- We can apply `clad::gradient` to the entire model's loss function.

```
// The goal: Differentiate the whole loss function w.r.t model params
float gpt2_loss(const GPT2& model, const ITensor& input, const ITensor& targets) {
    FTensor probs = model.forward(input);
    return cross_entropy_loss(probs, targets);
}

// This now works!
auto grad_fn = clad::gradient(gpt2_loss, "model"); // Differentiate w.r.t. 'model'
```

- Clad successfully processes the entire, complex C++ codebase—including loops, custom classes, and nested function calls—to generate the complete backward pass.

Cladtorch: Backpropagation

Clad transforms human-written forward pass code into an efficient backward pass. This required writing custom derivatives for `cladtorch` operations to guide the process.

Human-Written C++ Forward Pass

```
// Inside gpt2::LayerNorm
FTensor forward(const FTensor&
input) const {
  auto norm = input.norm();
  auto tmp = norm * weight;
  return tmp + bias;
}
```

Clad-Generated Backward Pass

```
void forward_pullback(
  const FTensor& input, FTensor _d_y,
  gpt2::LayerNorm* _d_this, FTensor* _d_input
) const {
  op_plus_pullback(tmp, this->bias, _d_y,
&_d_tmp, &_d_this->bias);
  op_star_pullback(norm, this->weight, _d_tmp,
&_d_norm, &_d_this->weight);
  norm_pullback(input, _d_norm, _d_input);
}
```

Optimized Implementation

Anatomy

Core Principle: Avoid all sources of C++ abstraction overhead.

1. Pre-allocated Memory Arena:
   - A single, large `float*` buffer holds all model parameters, gradients, and activations.
     - Eliminates dynamic memory allocation during training and improves data locality/cache performance.
     - No freeing or reallocations of temporaries due to RAII, ensuring efficient memory use.
   - The `GPT2` struct simply holds pointers into the main memory arena.
2. Stateless C-Style Kernels:
   - All operations (`matmul`, `softmax`, `layernorm`) are pure functions acting on these pre-allocated buffers.
     - Simple, predictable, and easy for the clad and the compiler to optimize.

Clad Integration: A Perfect Match for C-Style Kernels

The procedural design simplified Clad integration significantly by mapping each forward kernel to its hand-optimized backward counterpart using `clad::custom_derivatives`. Clad can then generate the backpropagation code that orchestrates these kernels.

1. Forward Kernel

```
// Stateless function
void layernorm_forward(
  float* out, float* inp,
  float* weight, float* bias,
  int N, int C
);
```

3. Clad Pullbacks

```
void layernorm_forward_pullback(
  float* out, float* inp,
  float* weight, float* bias,
  int N, int C,
  float* dout, float* dinp,
  float* dweight, float* dbias
);
```

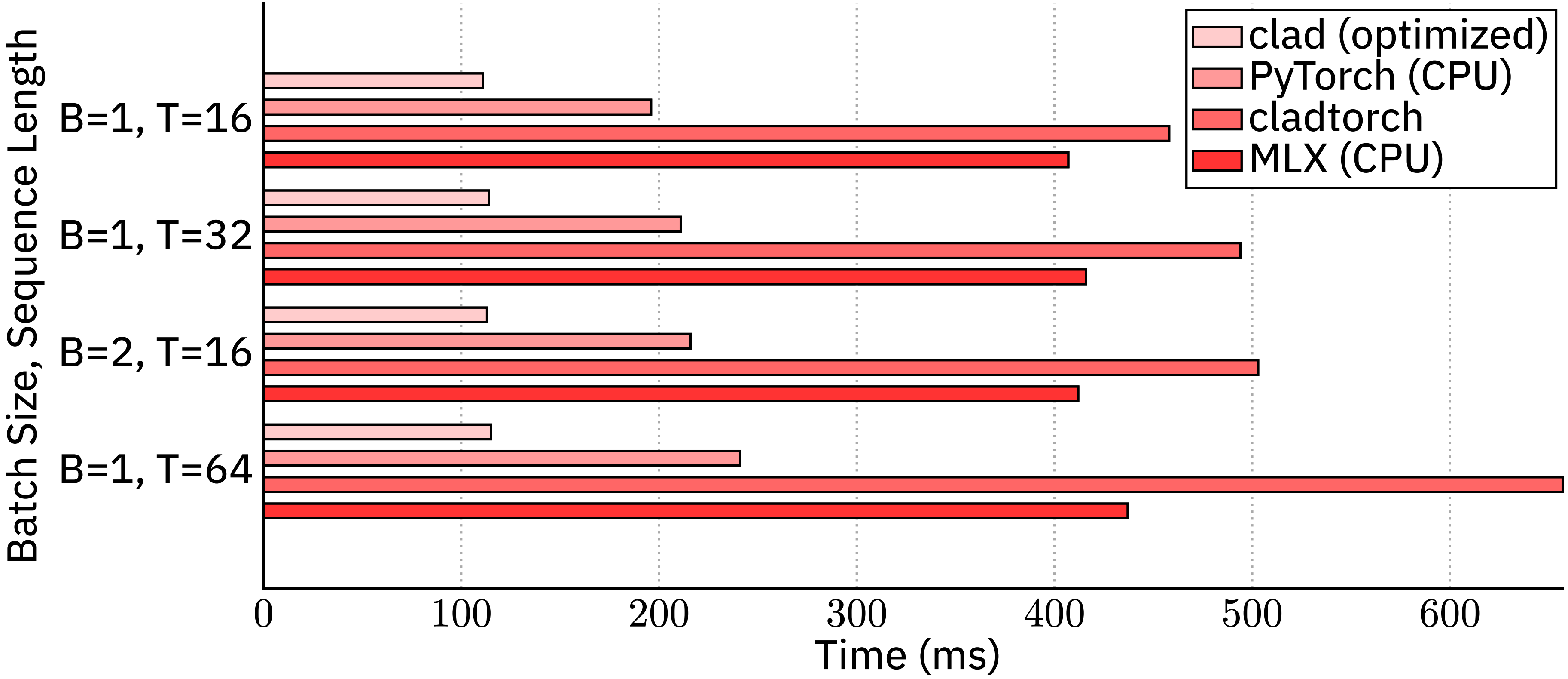
Results

Performance

- System: Apple M3 Max CPU
- Task: Full GPT-2 training iteration (forward + backward pass)
- Result: Our C++ implementation is consistently faster than PyTorch on CPU.

Config (Batch, SeqLen)	Clad (optimized) (ms)	PyTorch (ms)	Speedup
B=1, T=16	111	196	$1.77\times$
B=1, T=32	114	211	$1.85\times$
B=2, T=16	113	216	$1.91\times$
B=1, T=64	115	241	$2.1\times$

Performance Benchmarks



Performance Analysis

- 1. No Python Overhead:
  - The entire training loop is a compiled, monolithic binary. No calls between Python and C++, no GIL, no dynamic dispatch.
- 2. Cache-Friendly Memory Layout:
  - The single pre-allocated buffer leads to excellent data locality, and no freeing or reallocations of temporaries due to RAII (like in cladtorch), ensuring efficient memory use.
- 3. Direct BLAS & Kernel Fusion:
  - We call optimized libraries like Apple's Accelerate framework directly for `cblas_sgemm` without framework abstractions.
  - This design allows for manual kernel fusion.

Summary & Future Work

Project Summary & Key Achievements

- Two functional C++ implementations for LLM training: a flexible prototype and one for high-performance.
- Successfully demonstrated Clad's capability for end-to-end differentiation of a real-world, complex model like GPT-2.
- Achieved a significant performance milestone: The optimized C++ implementation outperforms PyTorch on CPU.
- Created a strong foundation for future research into C++-based ML and GPU acceleration.

Future Work/Promising Directions

1. GPU Acceleration (CUDA):
   - The C-style, optimized kernel design is an ideal foundation for porting to GPUs.
   - This would allow us to investigate the performance characteristics of this on hardware best suited for training.

2. Clad-Driven Kernel Fusion:
   - Leverage Clad's static analysis capabilities to automatically fuse sequential operations.
   - Example: Fusing softmax and cross_entropy_loss into a single, more efficient kernel.
   - Benefit: Reduces memory bandwidth bottlenecks and kernel launch overhead.

Thank You