Enhancing LLM Training Efficiency with Clad

Midterm Presentation

Rohan Timmaraju, July 2025
Compiler Research Group

Recap: The Goal

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.

Midterm Progress & Achievements

1. Functional C++ Tensor Operations/Inference

- `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 (using optimized BLAS kernels).

2. End-to-End Differentiation

- This is the project’s core technical success.
- 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.

From C++ to Gradients

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

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

Clad-Generated Backward Pass

```cpp
void forward_pullback(
  const FTensor& input, FTensor _d_y,
  gpt2::LayerNorm* _d_this, FTensor* _d_input
) const {
  op_plus_pullback(&tmp, this->bias,
_d_result, &_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);
}
```

3. Initial Performance Benchmarks

- Our training loop is fully functional and already incorporates highly optimized BLAS kernels (Apple Accelerate) for matrix multiplications.
- Results:
  - With these optimizations, our `cladtorch` implementation is roughly on par with Andrej Karpathy's `llm.c`.
    - This confirms the baseline efficiency is strong.
  - It is currently 3-4x slower than a comparable PyTorch implementation.
    - This suggests the remaining overhead is not in matmul, but likely in other operations, memory access patterns, or parallelism.

Next Steps & Optimization

Next Steps: Functional to Fast

The focus for the remainder of GSoC is clear: find and reduce overhead.

- 1. Deep Profiling to Find Bottlenecks:
  - Since `matmul` is optimized, use profiling tools to find the next hotspots.
  - Candidates: custom derivatives, memory allocation/copying, or temporary object creation in the generated code.
- 2. Parallelize with OpenMP:
  - PyTorch’s speed heavily relies on parallelism, so we should do the same for a fair comparison.

Next Steps (pt. 2)

- 3. Optimize Memory Patterns:
  - Investigate and reduce intermediate tensor allocations and copies. Minimizing temporary objects in tight loops is critical for C++ performance.
- 4. Benchmarking & Documentation:
  - Conduct final, formal benchmarks after optimization.
  - Finalize the `cladtorch` library, documentation, and project report.

Summary

Midterm Summary & Impact

Progress:

- We have successfully built a working C++ ML setup (`cladtorch` + `clad`).
- We have demonstrated the core feasibility of using Clad for end-to-end differentiation of a complex LLM.
- We have a working, BLAS-optimized training loop.

Impact:

- This work is a step towards enabling efficient LLM training in C++ & HPC environments.
- It provides a powerful, real-world use case for Clad, highlighting its strengths.

Thank You & Questions