Performance of Enzyme AD in Real World High-Complexity Deep Learning Models

Manuel Drehwald[1], Li Zhang[2], Mark Zhang[2]

[1]The Matter Lab, University of Toronto [2]University of Toronto

Abstract

Machine learning research has relied on Python, with libraries like PyTorch, JAX, and TensorFlow offering features such as automatic differentiation, vectorization, and parallelization, essential for fast prototyping. However, Python's lack of performance compared to Fortran, C++, or Rust creates tension between usability and efficiency. Performance bottlenecks are often mitigated by implementing critical operations in more performant languages. LLVM, a low-level compiler framework used in Rust, Julia, and C++, introduced automatic differentiation, potentially enabling both fast prototyping and efficient execution within the same language. We analyze how this new LLVM feature compares to a handwritten, well-optimized LLM implementation.

Introduction

Frameworks like PyTorch and JAX provide automatic differentiation (AD) for efficient prototyping but face Python's performance limitations and require extensive rewrites. Enzyme [1], an LLVM-based tool, addresses these issues by enabling "real AD" at the compiler level, supporting languages like C, C++, and Rust without major rewrites. We will evaluate Enzyme's capabilities on the transformer-based GPT-2 architecture, comparing its efficiency and scalability against manual implementations in C and Rust.

Goal: Minimize code alterations, rely on Enzyme to seamlessly replace manually written backward functions.

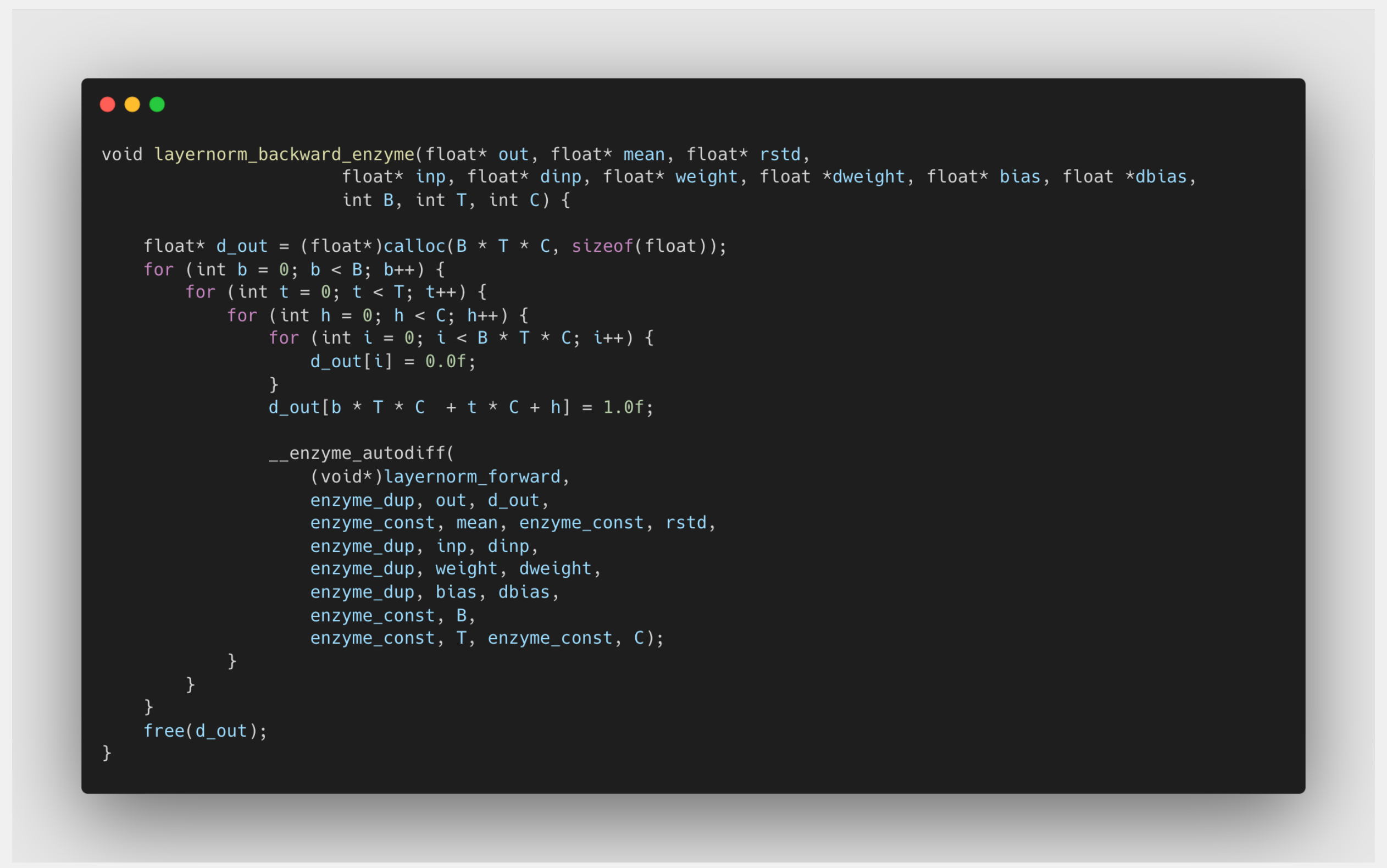
Differentiation with Enzyme

Differentiation Modes: Enzyme supports both forward-mode and reverse-mode differentiation:

- **Forward Mode:** For a function $f : \mathbb{R}^n \to \mathbb{R}^m$, it computes the Jacobian-vector product $df = J \cdot dx$ by seeding $dx \in \mathbb{R}^n$.
- **Reverse Mode:** It computes the vector-Jacobian product $\mu = J^\top \lambda$ by seeding $\lambda \in \mathbb{R}^m$. Reverse mode is often used for neural networks due to their scalar output after the loss function ($m \ll n$).

Shadow Variables: Enzyme requires shadow variables ($\hat{x}$) for each active variable x. These accumulate $\frac{\partial \text{output}}{\partial x}$ and are passed in pairs $(x, \hat{x})$ to enable efficient gradient computation.

Example: LayerNorm Enzyme Backward (Layer Differentiation)



```c
void layernorm_backward_enzyme(float* out, float* mean, float* rstd,
                       float* inp, float* dinp, float* weight, float *dweight, float* bias, float *dbias,
                       int B, int T, int C) {

    float* d_out = (float*)calloc(B * T * C, sizeof(float));
    for (int b = 0; b < B; b++) {
        for (int t = 0; t < T; t++) {
            for (int h = 0; h < C; h++) {
                for (int i = 0; i < B * T * C; i++) {
                    d_out[i] = 0.0f;
                }
                d_out[b * T * C  + t * C + h] = 1.0f;

                __enzyme_autodiff(
                    (void*)layernorm_forward,
                    enzyme_dup, out, d_out,
                    enzyme_const, mean, enzyme_const, rstd,
                    enzyme_dup, inp, dinp,
                    enzyme_dup, weight, dweight,
                    enzyme_dup, bias, dbias,
                    enzyme_const, B,
                    enzyme_const, T, enzyme_const, C);
            }
        }
    }
    free(d_out);
}
```

Figure 1. LayerNorm Enzyme Backward

Methodology

Model: Andrej Karpathy's GPT-2 manual C implementation [2].

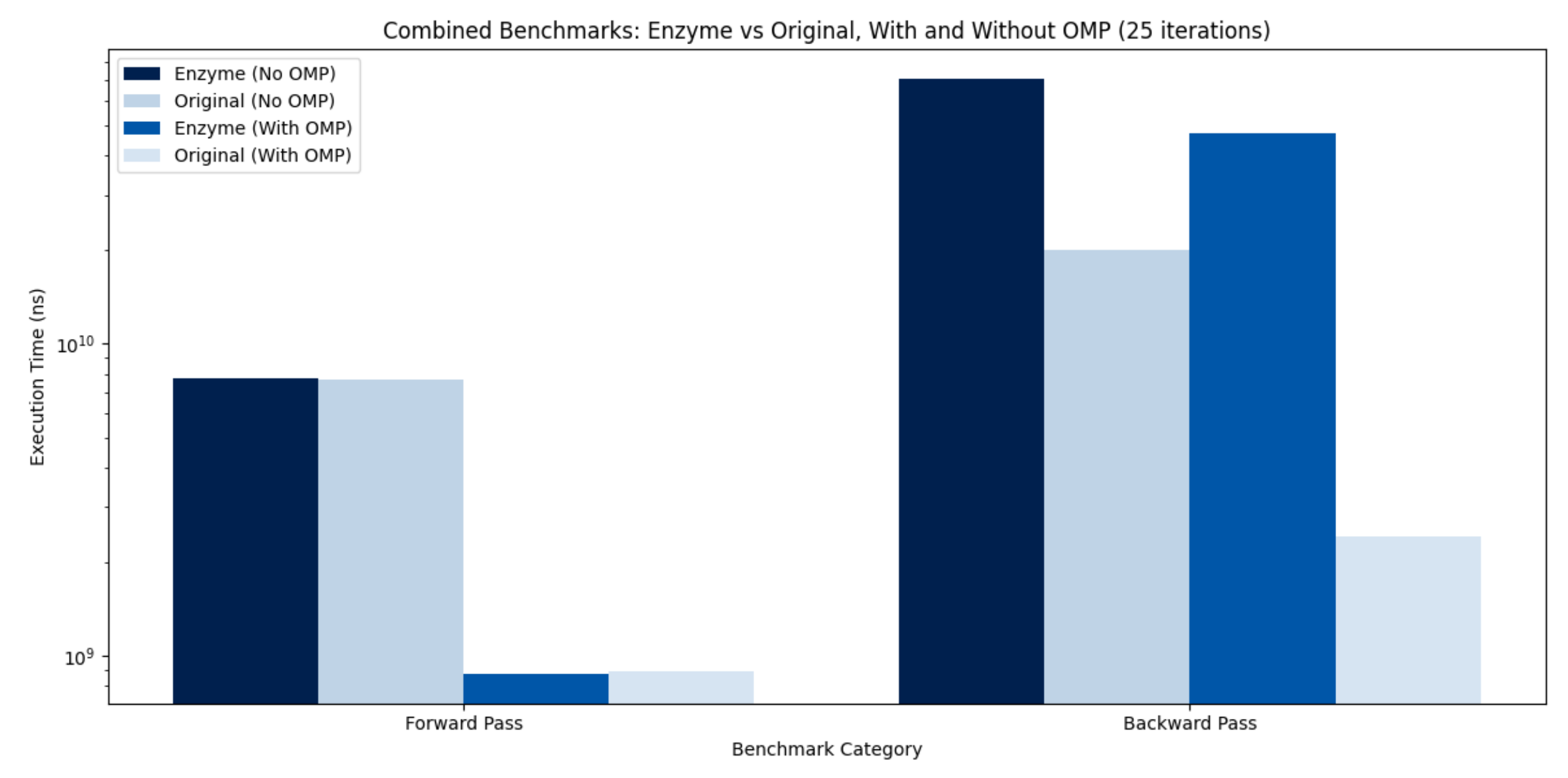
Benchmarking: For C, we compared the following differentiation strategies: `Enzyme AD, normal manual differentiation, openMP-parallelized Enzyme AD, and openMP parallelized manual differentiation`. For Rust, we explored only single-threaded manual backward compared to the Enzyme-based backward.

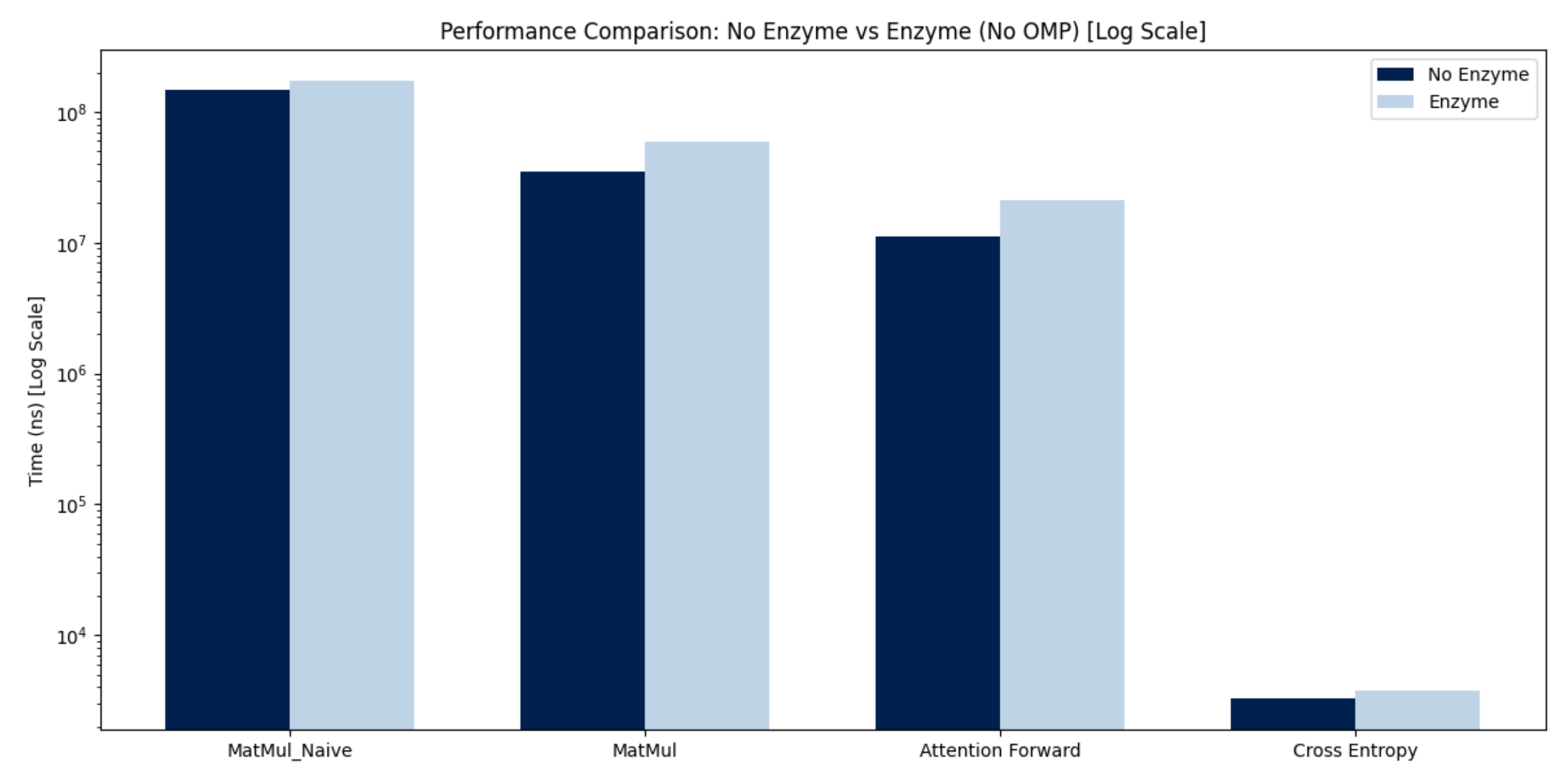
Hardware: Benchmarks were performed on a dual Intel Xeon 6248R CPU node using up to 48 cores and 378 GB RAM. Single-threaded runs were pinned to core one, while multi-core experiments utilized cores 0–23.

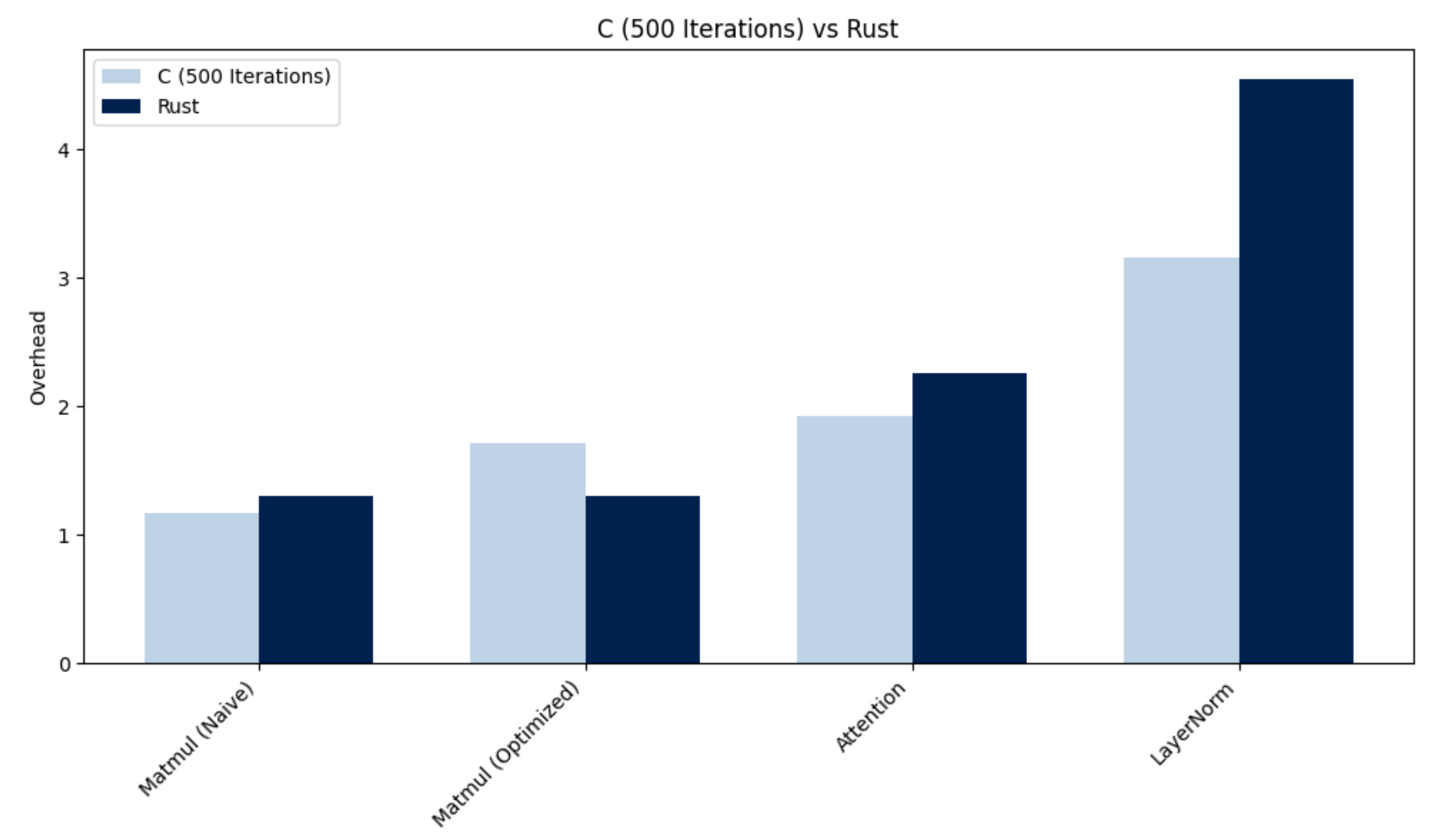
Software: Google Benchmark for C and Criterion for Rust, with high iteration counts for stability. Benchmarked components included attention, matrix multiplication (naive and efficient), layer normalization, cross-entropy, and the full GPT-2 model [3].

Dataset and Tokenizer: Benchmarks used the *tinyshakespeare* dataset with the default tokenizer in Karpathy's GPT-2 implementation.

Results







Discussion

- Enzyme shows promise [1, 7, 8] but currently lags behind well-optimized high-level tools like PyTorch and JAX for stacked layer models (e.g., GPT-2).
- High-level tools optimize by leveraging matrix properties, minimizing caching/recomputations. Low-level tools like Enzyme default to caching/recomputing due to technical uncertainties.
- Optimizations applied to Enzyme:
  - Splitting input arguments
  - Annotating pointers with the `__restrict` attribute
- Memory usage remains higher compared to manual implementations due to caching limitations.
- Further improvements to Enzyme and LLVM could narrow the gap, but achieving a 20× gain to compete with state-of-the-art tools is unlikely.
- High-level tools outperform low-level approaches in ML workloads, whereas the reverse often holds for HPC applications.

Limitations & Future Work

- Constraints in work:
  - Experimental state of C and Rust frontends for Enzyme.
  - Current limitations in Enzyme's implementation.
- Performance insights:
  - Both C-Enzyme and Rust-Enzyme show competitive performance for differentiating individual layers.
  - Future work could explore custom-derivative support for compute-intensive layers to better understand their impact.
- Collaborative efforts:
  - Collaboration with compiler developers to address current challenges.
  - Aim to present enhanced benchmarks in the future.
- Broader research directions:
  - Potential of high-level compiler optimizations, particularly the experimental C++ compiler pipeline enabled by C-IR and MLIR [4].
  - Possibilities of integrating Rust's language guarantees with frameworks like MLIR.
  - Exploring polyhedral optimizations, such as those demonstrated by Pollygeist [5].
  - Careful consideration of complexity trade-offs in polyhedral optimizations.
  - Focus on advancing these areas to enable more efficient compiler-driven optimizations.

References

[1] NeurIPS 2020 : Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients, March 2021.
[Online; accessed 14. Dec. 2024].

[2] llm.c, December 2024.
[Online; accessed 14. Dec. 2024].

[3] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei.
Scaling laws for neural language models.
arXiv, 2020.

[4] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko.
MLIR: Scaling compiler infrastructure for domain specific computation.
In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[5] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko.
Polygeist: Raising c to polyhedral mlir.
In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '21, New York, NY, USA, 2021. Association for Computing Machinery.

[6] William S. Moses and Valentin Churavy.
High-performance automatic differentiation of llvm.
LLVM Oct Dev Meeting, 2020.

[7] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert.
Reverse-mode automatic differentiation and optimization of gpu kernels via enzyme.
In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[8] William S. Moses, Sri Hari Krishna Narayanan, Ludger Paehler, Valentin Churavy, Michel Schanen, Jan Hückelheim, Johannes Doerfert, and Paul Hovland.
Scalable automatic differentiation of multiple parallel paradigms through compiler augmentation.
In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.