Rethinking Deep Learning Autodiff for Rust & C

Manuel Drehwald 1010271983
Li Zhang 1009010026
Mark Zhang 1009007959

Abstract

Machine Learning research has traditionally been carried out in Python. Popular libraries like PyTorch [Pas+19], JAX [Bra+18], and TensorFlow [Mar+15] optimize tensor operations with features like automatic differentiation, vectorization, and parallelization. While these features could be implemented manually, their complexity makes it impractical for fast prototyping, making these tools valuable for scientific and high-performance computing. However, despite those valuable tools, Python was not designed to be a performant language on pair with Fortran, C++, or Rust, so scientist encounter a tension between good tooling in Python, and efficient programs in other languages. Python overhead can be amortized by implementing key operations in more efficient languages and extensive optimizations of few critical layers, common in Large Language Models (LLM). Lately however, the low level compiler framework "LLVM", used by languages like Rust, Julia, and C++, gained a new feature for automatic differentiation. This could enable scientists and developers to achieve both fast prototyping and efficient implementations in the same language. To understand how close such a new LLVM feature brings us to our goal, we analyze how well this new feature compares, when tested against a handwritten, well-optimized LLM implementation.

1 Introduction

Enzyme [21], a relatively new LLVM-based tool for automatic differentiation (AD), overcomes the limitations of popular frameworks like PyTorch [Pas+19] and JAX [Bra+18], which rely on predefined primitives or operator overloading (can be interpreted as "fake AD"). These frameworks often require extensive rewrites of manual implementations to match their abstractions, limiting flexibility and complicating adaptation to existing or unconventional implementations. Enzyme leverages LLVM's compiler-level gradient synthesis capabilities [Mos+21b; Mos+22] to enable "real AD" on optimized IR, which allows it to support a vast array of languages, including C/C++ and Rust, without extensive rewrites of existing code. However, current exploratory work with Enzyme predominantly focuses on small-scale networks or non-machine-learning HPC scientific applications [Ram+20; DJV24], leaving its performance and capabilities on large and complex architectures largely unexplored. In this work, we aim to explore Enzyme AD's CPU-based capabilities on a large and complex neural network, specifically the transformer-based architecture of GPT-2. By comparing Enzyme AD against high-quality manual implementations of GPT-2 in C and Rust, we evaluate its computational efficiency, scalability, and overall viability for use in real-world, demanding deep learning CPU training applications.

2 Related Works

Most approaches for automatic differentiation can be classified as either *operator overloading*, or *source transformation*. In the former camp, examples include Adept [Hog14] and JAX [Bra+18]. With this approach, however, a program has to be written using built-in differentiable operators and functions, preventing differentiation of arbitrary code or libraries. In the latter camp, examples include

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

Tapenade [HP13] and ADIC [NNW10]. With source rewriting, all source code to be differentiated must be provided to the tool ahead-of-time, making it difficult to use, e.g. pre-compiled libraries. We refer the reader to [Bay+18] for a comprehensive survey of AD techniques and implementations.

A second distinction between tools can be drawn by the abstraction level on which they operate. On the highest abstraction level we find operator overloading and template metaprogramming tools, which do not modify the underlying language or compiler. A slightly closer compiler integration happens for AD tools operating on the Abstract Syntax Tree of the code, usually by interfaces like macros, which are provided by the compiler. Both JAX [Bra+18] and PyTorch [Pas+19; LLV] are autodiff tools which enable compilation of python programs, but still tend to operate at the beginning of the compilation pipeline. JAX for example completely handles Automatic Differentiation on a high-level code representation, before "lowering" the differentiated code to a representation which can be understood by their XLA [Sab20; SL23] compiler backend, responsible for optimizations.

Enzyme [21] is a tool for automatic differentiation that works on LLVM Intermediate Representation (IR), offering a lower-level approach than previous methods. Unlike higher-level tools that analyze complex source languages like C++ or Rust, Enzyme leverages LLVM's simplified code representation to compute derivatives more efficiently, leading to significant performance gains in scientific and high-performance computing. Extensions of Enzyme have enabled gradient generation for GPU kernels [Mos+21b] and parallel frameworks [Mos+22].

Following the success of Enzyme, a new tool called Derivgrind [Aeh+22a; Aeh+22b] was developed, which is able to differentiate arbitrary binaries on assembly level, therefore being without question the most low-level AD tool under development. It has been shown to be even more versatile than Enzyme by differentiating languages like Python which do not offer an LLVM based compiler. However, this versatility comes at a cost, since most of the information needed for efficient AD was lost before assembly level, and can't efficiently be regained, leading to poor runtime performance.

3 Methodology

3.1 Selected Architecture and Benchmarking

GPT-2, a transformer decoder-based text-completion model, is ideal for evaluating Enzyme's capabilities due to its complexity and popularity. For a fair evaluation, we chose Andrej Karpathy's GPT-2 manual implementation in C [24e]. This well-known, optimized manual implementation ensures Enzyme is tested on realistic, production-quality code, not a simplified biased setup. We aim to compare the runtime in different configurations and the overall overhead of Enzyme backward compared to the forward pass. For C, we compared the following differentiation strategies: `Enzyme-based AD, normal manual differentiation, openMP-parallelized Enzyme-based AD, and openMP parallelized manual differentiation`. For Rust, we explored only single-threaded manual backward compared to the Enzyme-based backward.

3.2 Setup

Hardware: All benchmarking was done on a dual Intel Xeon 6248R CPU node with HyperThreading and CPU Frequency scaling disabled on up to 48 cores, combined with 378 GB of RAM. We pinned single-threaded benchmark runs to core one, to not interfere with background tasks running on core zero. Experiments using 24 cores or less were run on the first CPU, by disabling cores 24-47.
Software: Benchmarked using Google Benchmark (C) [24a] and Criterion (Rust) [24d]. In both cases, we experimented and settled with predefined high iteration counts for stable and accurate benchmarks.
Benchmarked Components: From findings on scaling laws discussed in [Kap+20], we benchmarked the following key components based on their computational complexity and role in model: Attention, Matrix Multiplication (Naive and Efficient), Layer Normalization, Cross Entropy, Entire GPT-2. See B for details.

Input Sizes: We will be using the original implementation's predefined input sizes for benchmarking.
Dataset and Tokenizer: we will be using the exact same dataset: *tinyshakespeare* and tokenizer as the one used by Karpathy's GPT-2 implementation.
See Appendix B for full setup details.

3.3 Enzyme Integration

Differentiation Modes: Enzyme provides intrinsic functions for forward and reverse-mode differentiation. For a function $f : \mathbb{R}^n \to \mathbb{R}^m$, forward mode computes the Jacobian-vector product $df = J \cdot dx$ by seeding $dx \in \mathbb{R}^n$, whereas reverse mode computes the vector-Jacobian product $\mu = J^\top \lambda$ by seeding $\lambda \in \mathbb{R}^m$.

For example, in reverse mode, for $f(x) \in \mathbb{R}^2$, seeding $\lambda = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ isolates $\frac{\partial f_1(x)}{\partial x}$. This property makes reverse mode efficient for neural networks as they generally have a scalar output after the loss function (scalar propagation into to a high-dimensional parameter space, $m \ll n$). Thus, all differentiation calls will be made in **reverse mode**.

Shadow Variables: Enzyme also expects shadow variables ($\hat{x}$). For each active variable x, $\hat{x}$ has to be allocated and initialized to accumulate $\frac{\partial output}{\partial x}$, where they are passed in pairs $(x, \hat{x})$.

Goal: The aim is to minimize changes, as the ideal scenario for Enzyme is to seamlessly replace manually written backward functions with automatic differentiated ones.

3.3.1 Implementation

We faced many significant challenges in adapting to the GPT-2 architecture. In C, the primary obstacle was the original implementation which incorporate a centralized struct for housing GPT-2 metadata and weighs which caused Enzyme to compute gradients for unnecessary components. This was resolved by splitting the struct into separate "ActivationTensors" and "ParameterTensors," which greatly reduced the amount of bugs and increased performance. This also required the alteration of all the functions that depended on the GPT2 struct and custom functions to initialize our secondary shadow model. Additionally, issues like memory overflow and zero-gradient outputs were fixed by enabling the '-enzyme-inline' flag which simplified compiler analysis at the cost of increasing compile time and binary size. These issues and solutions were experienced and applied similarly when benchmarking individual components. For Rust, integration faced even greater challenges due to the immaturity of Enzyme's Rust interface. It failed to deduce the type of some intermediate variables, in turn leading to a compilation failure, as closer described in [MC20].When computing derivatives for the full GPT-2-Rust version, Enzyme would further incorrectly return zeros as the gradient, a known bug for the Rust frontend. To mitigate these, we used a deterministic C-to-Rust translator to translate [24b] the entire GPT-2 for benchmarking GPT-2 as a whole to ensure consistency and accuracy across the implementations. On the other hand, for individual components benchmarking, we did high-quality manual rewrites of the corresponding C versions with the supervision GPT-4o [24c] as this allowed us to make small adjustments to adjust to Rust while not deviating from the C implementation.

4 Results & Discussion

4.1 Entire GPT-2 with Enzyme

Training: To ensure correctness, we initially trained GPT-2 with Enzyme on a single batch to verify over-fitting by observing if the training loss approached zero. Then, we proceeded to train the model with the same number of epochs as the intial implementation. See Appendix A for the plots.

Discussion / Interpretation: Our benchmarks show that Enzyme, despite the promising results in [21; Mos+21b; Mos+22] can not yet compete with well-written, high level LLM implementations in C, and therefore should also fall behind PyTorch and JAX implementations of the same model. Our benchmarking of the overhead of individual layers, as well as existing Enzyme papers shows that this is no inherit limitation of Enzyme, which is able to differentiate each layer on it's own reasonably efficiently. The challenge arises when stacking more layers, as it is common in an LLM like GPT-2. Manual Differentiation and high-level AD tools like JAX or PyTorch are able to leverage matrix properties to reason about the minimal set of variables to cache and recompute. A low-level AD tool like Enzyme has to analyze technical details like multiple pointer indirections and nested loops, and default to cache or recompute variables in case of doubt, even if these variables later turn out to be not needed for the gradient computations. We did reasonable effort to support Enzyme by splitting up input arguments, and annotating pointers with the `__restrict` attribute. The memory usage comparison after these adjustments clearly highlights that Enzyme still fails to find an equally effective set of variables to cache as the corresponding manual high-level implementation. One

could hope that further analysis improvements to Enzyme and LLVM narrow this gap. Still, we also recognize that for ML workloads the effect of high-level optimizations seems to outperform the effect of low-level optimization. In contrast, the opposite can be said for most applications in scientific computing and HPC. We cannot specify the performance impact of Enzyme bugs, which limited us in the number of configurations we were able to test. However, it seems highly unlikely that improving Enzyme to allow removing our aggressive inlining workaround would provide the 20 $\times$ performance improvement needed to compete with state-of-the-art implementations.

Combined Overhead Results

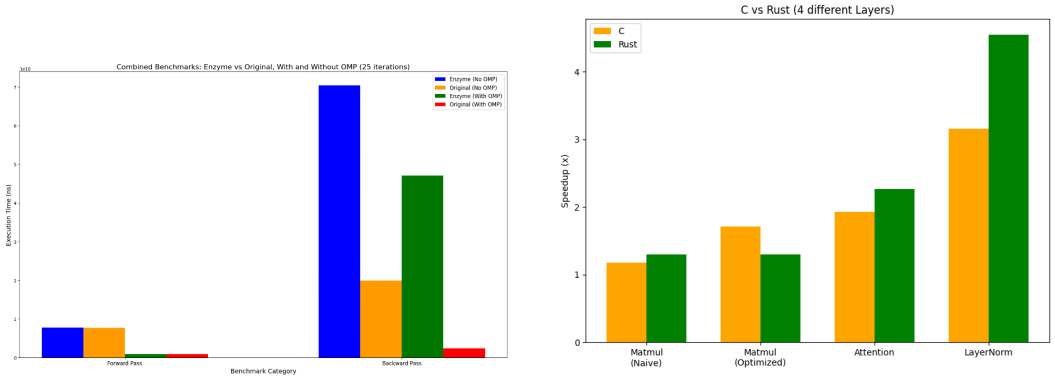


Figure 1: **Left:** Runtime of forward and backward times (ns) for C **Right:** Overhead (time to compute gradients divided by the original runtime) for C and Rust. See C.0.3 for more detailed breakdown.

5 Limitations & Future Work

Our experiments were limited by the experimental nature of C and Rust frontends for Enzyme, as well as bugs within the Enzyme implementation itself. To quantify the impact of high-level derivatives on specific layers of the neural network, it would be desirable to provide custom-derivatives for each of the compute intensive layers. However, the C frontend to Enzyme assumes that custom derivative rules will compute derivatives with respect to all input pointers. Given that we want to compute derivatives with respect to the weights, but not input or target vectors, such rules would be inefficient and would therefore not provide new insights. Manuel Drehwald worked on adding support for custom-derivatives to the Rust compiler by levering that he is the lead developer of the Rust frontend of Enzyme, however, Enzyme failed to differentiate the whole Rust implementation. Both C-Enzyme and Rust-Enzyme already offer competitive performance when only differentiating individual layers, so usage of custom-derivatives in such a limited context won't provide new insights either. We are in contact with multiple Rust and Enzyme compiler developers and hope to show these benchmarks in a future work.

On a more high-level outlook, we have seen the performance impact of high-level compiler optimizations and are curious to see the effect of the experimental high-level C++ compiler pipeline enabled by C-IR and MLIR[Lat+21]. We further are curious about whether the strong language guarantees of Rust could be combined with high-level compiler frameworks like MLIR to achieve even more efficient optimizations than currently enabled by LLVM. However, at the time of writing no such active efforts are known to the authors, and given the multi-year nature of such projects additional shorter experiments might be merited. One path which we want to highlight is through polyhedral optimizations, which try to "raise" low-level compiler languages to enable high-level optimizations. An LLVM based example of such an effort is Pollygeist [Mos+21a]. However, these efforts had inconsistent improvements in the past and might increase compile times by more than a magnitude, making them unsuitable for real-world applications. Further experiments should therefore carefully consider the complexity trade-offs for such ML-specific compiler extensions.

References

[NNW10] Sri Hari Krishna Narayanan, Boyana Norris, and Beata Winnicka. "ADIC2: Development of a component source transformation system for differentiating C and C++". In: *Procedia Computer Science* 1.1 (2010), pp. 1845–1853.

[HP13] Laurent Hascoet and Valérie Pascual. "The Tapenade automatic differentiation tool: principles, model, and specification". In: *ACM Transactions on Mathematical Software (TOMS)* 39.3 (2013), pp. 1–43.

[Hog14] Robin J Hogan. "Fast reverse-mode automatic differentiation using expression templates in C++". In: *ACM Transactions on Mathematical Software (TOMS)* 40.4 (2014), pp. 1–16.

[Mar+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[Bay+18] Atilim Gunes Baydin et al. "Automatic differentiation in machine learning: a survey". In: *Journal of Marchine Learning Research* 18 (2018), pp. 1–43.

[Bra+18] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: http://github.com/jax-ml/jax.

[Pas+19] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems* 32 (2019). URL: https://proceedings.neurips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[Kap+20] Jared Kaplan et al. "Scaling Laws for Neural Language Models". In: *arXiv* (2020). eprint: 2001.08361 (cs.LG). URL: https://arxiv.org/abs/2001.08361.

[MC20] William S. Moses and Valentin Churavy. "High-Performance Automatic Differentiation of LLVM". In: LLVM Oct Dev Meeting. 2020.

[Ram+20] Ali Ramadhan et al. "Oceananigans.jl: Fast and friendly geophysical fluid dynamics on GPUs". In: *Journal of Open Source Software* 5.53 (2020), p. 2018. DOI: 10.21105/joss.02018.

[Sab20] Amit Sabne. "Xla: Compiling machine learning for peak performance". In: *Google Res* (2020).

[Lat+21] Chris Lattner et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[Mos+21a] William S. Moses et al. "Polygeist: Raising C to Polyhedral MLIR". In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. PACT '21. Virtual Event: Association for Computing Machinery, 2021.

[Mos+21b] William S. Moses et al. "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. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476165. URL: https://doi.org/10.1145/3458817.3476165.

[21] *NeurIPS 2020 : Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients*. [Online; accessed 14. Dec. 2024]. Mar. 2021. URL: https://neurips.cc/virtual/2020/public/poster_9332c513ef44b682e9347822c2e457ac.html.

[Aeh+22a] Max Aehle et al. "Forward-Mode Automatic Differentiation of Compiled Programs". In: *arXiv* (Sept. 2022). DOI: 10.48550/arXiv.2209.01895. eprint: 2209.01895.

[Aeh+22b] Max Aehle et al. "Reverse-Mode Automatic Differentiation of Compiled Programs". In: *arXiv* (Dec. 2022). DOI: 10.48550/arXiv.2212.13760. eprint: 2212.13760.

[Mos+22] William S. Moses et al. "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. Dallas, Texas: IEEE Press, 2022. ISBN: 9784665454445.

[SL23] Daniel Snider and Ruofan Liang. "Operator Fusion in XLA: Analysis and Evaluation". In: *arXiv* (Jan. 2023). DOI: 10.48550/arXiv.2301.13062. eprint: 2301.13062.

[24a] *benchmark*. [Online; accessed 14. Dec. 2024]. Dec. 2024. URL: https://github.com/google/benchmark.

[24b] *c2rust*. [Online; accessed 14. Dec. 2024]. Dec. 2024. URL: https://github.com/immunant/c2rust.

[24c] *chatGPT-4o*. [Online; accessed 14. Dec. 2024]. Dec. 2024. URL: https://chatgpt.com/.

[24d] *criterion.rs*. [Online; accessed 14. Dec. 2024]. Dec. 2024. URL: https://github.com/bheisler/criterion.rs.

[DJV24] Manuel S. Drehwald, Asma Jamali, and Rodrigo A. Vargas-Hernández. "MOLPIPx: an end-to-end differentiable package for permutationally invariant polynomials in Python and Rust". In: *arXiv* (Nov. 2024). DOI: 10.48550/arXiv.2411.17011. eprint: 2411.17011.

[24e] *llm.c*. [Online; accessed 14. Dec. 2024]. Dec. 2024. URL: https://github.com/karpathy/llm.c.

[LLV] LLVM. *Torch-MLIR*. URL: https://github.com/llvm/torch-mlir.

A Additional Figures

Single Batch Overfitting

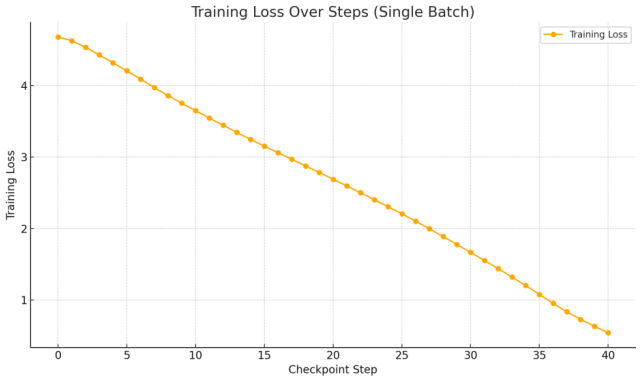
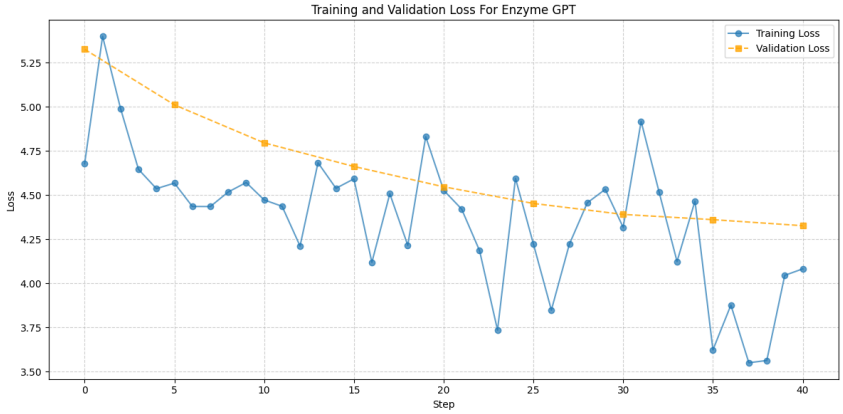
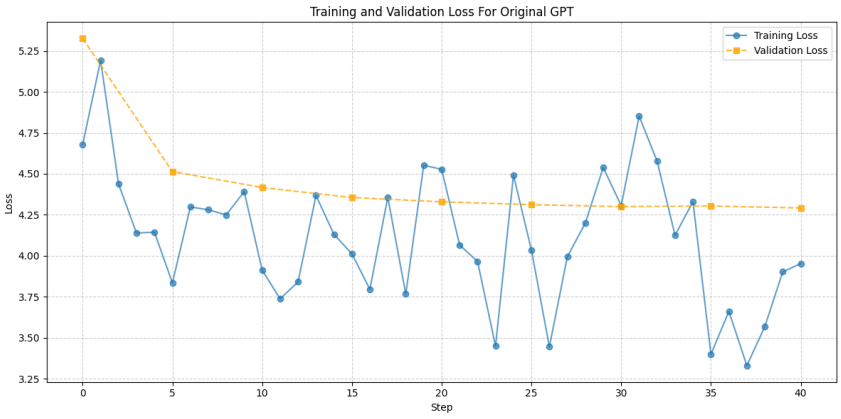


Figure 2: Overfitting on a singular batch (Enzyme)

Train / Validation Plots (Enzyme vs Original)



(a) Training and Validation Curve Enzyme (40 Epochs)



(b) Training and Validation Curve Original Implementation (40 Epochs)

B Full Setup Details

Components Bench-marked and Explanations:

Table 1: Complexity of Key Components in GPT-based Models

Component	**Explanation**
Attention	Nested matrix multiplications and softmax operations, highly sensitive to gradient accuracy.
Matrix Multiplication	Core to feedforward layers and attention, critical for performance.
Layer Normalization	Scaling/centering over input batches; adds computational overhead.
Cross Entropy	A loss function demanding stability in log and exponential operations.
Entire GPT-2	Combines all components, therefore the most challenging testcase.

Benchmarking Sizes: For the entire GPT-2 forward and backward benchmarking, we used:

$$B = 4, \quad T = 64 \text{ (Original Values)}$$

The table below lists the input sizes for the individual components benchmarked:

Layer Name	**Input Size**
MatMul Forward (Naive)	{4, 64, 768, 768}
MatMul Backward (Enzyme)	{4, 64, 768, 768}
MatMul Backward (Naive Enzyme)	{4, 64, 768, 768}
Attention Forward	{4, 64, 768, 12}
Attention Backward (Enzyme)	{4, 64, 768, 12}
LayerNorm Forward	{4, 64, 768}
LayerNorm Backward (Enzyme)	{4, 64, 768}
CrossEntropy Forward	{4, 64, 50304}
CrossEntropy Backward (Enzyme)	{4, 64, 50304}

Table 2: Input sizes used for benchmarking.

HPC Setup:

Table 3: Comprehensive Benchmark Setup Specifications (Idle)

Category	Details
Operating System	
OS	Ubuntu 20.04.6 LTS (focal)
Kernel Version	5.4.0-200-generic
Architecture	x86_64
CPU Mode	32-bit, 64-bit
Byte Order	Little Endian
CPU Details	
Vendor	GenuineIntel
Model	Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz
Cores per Socket	24
Total CPU Cores	48
Threads per Core	1
Sockets	2
NUMA Nodes	2
NUMA Node Distribution	Node 0: CPUs 0-46 (even), Node 1: CPUs 1-47 (odd)
CPU Max Frequency	4.0 GHz
CPU Min Frequency	1.2 GHz
L1 Cache	1.5 MiB
L2 Cache	48 MiB
L3 Cache	71.5 MiB
BogoMIPS	6000.00
Virtualization	VT-x
Memory Details	
Total Memory	376 GiB
Used Memory	1.5 GiB
Free Memory	314 GiB
Buffer/Cache	60 GiB
Available Memory	372 GiB
Swap Memory	975 MiB

C Appendix B: Code Examples

Listing 1: Enzyme Backward

```
void gelu_enzyme_backward(float *out, float *inp,
    float *dinp, int N) {
    float* d_out = (float*)calloc(N, sizeof(float));
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            d_out[j] = 0.0f;
        }
        d_out[i] = 1.0f;
        __enzyme_autodiff(
            (void*)gelu_forward,
            enzyme_dup, out, d_out,
            enzyme_dup, inp, dinp,
            enzyme_const, N);
    }
    free(d_out);
}
```

Listing 2: Manual Backward

```
void gelu_backward(float* dinp, float* inp, float* dout, int N) {
    for (int i = 0; i < N; i++) {
        float x = inp[i];
        float cube = 0.044715f * x * x * x;
        float tanh_arg = GELU_SCALING_FACTOR * (x + cube);
        float tanh_out = tanhf(tanh_arg);
        float coshf_out = coshf(tanh_arg);
        float sech_out = 1.0f / (coshf_out * coshf_out);
        float local_grad = 0.5f * (1.0f + tanh_out) +
            x * 0.5f * sech_out * GELU_SCALING_FACTOR *
            (1.0f + 3.0f * 0.044715f * x * x);
        dinp[i] += local_grad * dout[i];
    }
}
```

Figure 4: **Left:** Gelu backward implemented using Enzyme (C). **Right:** Original manual Gelu implementation (C).

Listing 3: Enzyme Structs

```c
typedef struct {
    GPT2Config config;
    // the weights (parameters) of the model, and their sizes
    // gradients of the weights
    ParameterTensors grads;
    float* grads_memory;
    // buffers for the AdamW optimizer
    float* m_memory;
    float* v_memory;
    // the activations of the model, and their sizes
    size_t num_activations;
    // gradients of the activations
    ActivationTensors grads_acts;
    float* grads_acts_memory;
    // other run state configuration
    int batch_size; // the batch size (B) of current forward pass
    int seq_len; // the sequence length (T) of current forward pass
    int* inputs; // the input tokens for the current forward pass
    int* targets; // the target tokens for the current forward pass
    float mean_loss; // after a forward pass with targets, will be populated with the mean loss
    size_t num_parameters;
} GPT2Const;

typedef struct {
    ParameterTensors params;
    size_t param_sizes[NUM_PARAMETER_TENSORS];
    float* params_memory;
    size_t act_sizes[NUM_ACTIVATION_TENSORS];
    ActivationTensors acts;
    float* acts_memory;
} GPT2;
```

Listing 4: Original GPT-2 Structs

```c
typedef struct {
    GPT2Config config;
    // the weights (parameters) of the model, and their sizes
    ParameterTensors params;
    size_t param_sizes[NUM_PARAMETER_TENSORS];
    float* params_memory;
    size_t num_parameters;
    // gradients of the weights
    ParameterTensors grads;
    float* grads_memory;
    // buffers for the AdamW optimizer
    float* m_memory;
    float* v_memory;
    // the activations of the model, and their sizes
    ActivationTensors acts;
    size_t act_sizes[NUM_ACTIVATION_TENSORS];
    float* acts_memory;
    size_t num_activations;
    // gradients of the activations
    ActivationTensors grads_acts;
    float* grads_acts_memory;
    // other run state configuration
    int batch_size; // the batch size (B) of current forward pass
    int seq_len; // the sequence length (T) of current forward pass
    int* inputs; // the input tokens for the current forward pass
    int* targets; // the target tokens for the current forward pass
    float mean_loss; // after a forward pass with targets, will be populated with the mean loss
} GPT2;
```

Figure 5: **Left:** New Struct GPT2Const (Shadow Model) Defined For Enzyme (C). **Right:** Original Struct For GPT2 (C).

C Detailed Benchmark Graphs

C.0.1 C and OMP

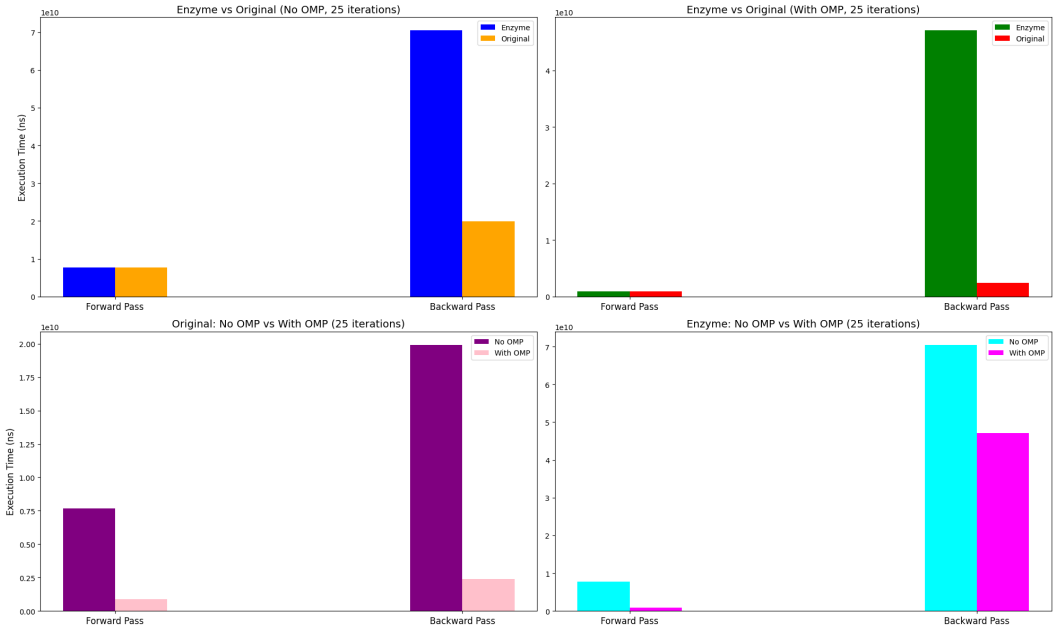


Figure 6: OMP vs No OMP (C)

C.0.2 C: Enzyme vs No Enzyme (ns) Timing (Log Scaled)

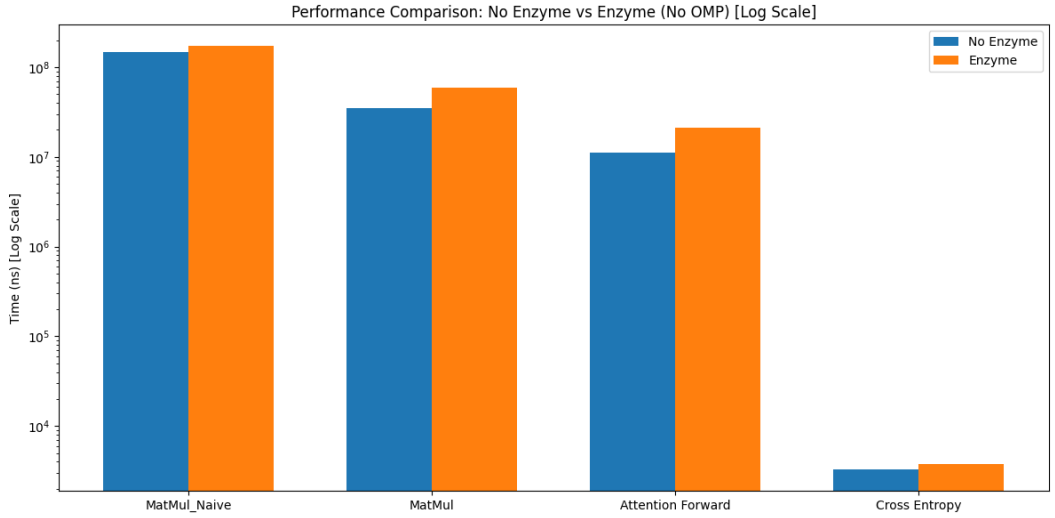


Figure 7: OMP vs No OMP (C)

C.0.3 Rust: Enzyme vs No Enzyme (ms) Timing (Log Scaled)

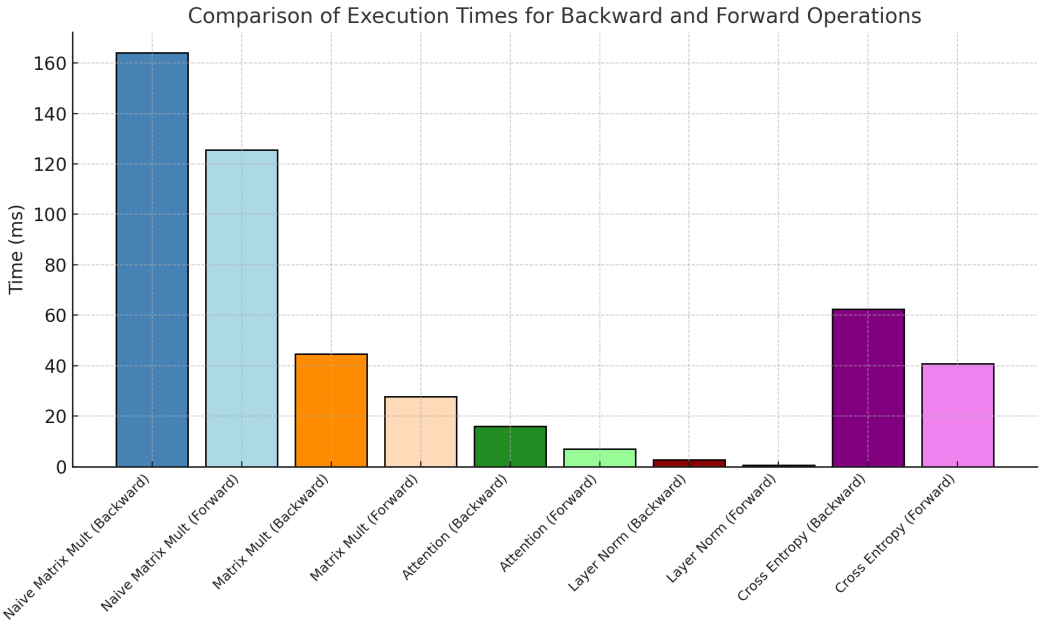


Figure 8: Rust: Enzyme Backward vs Forward (ms) Timing