

BurnMark: A Rust-based Benchmarking and Fuzzing Tool for LLM Systems

Kaleigh Ross Edusada, Harish Vasanth, Shaurya Raswan,
Samanyu Satheesh, Oscar Khaing
[BurnMark Repository](#)

March 20, 2026

1 Abstract

As Large Language Model (LLM) agents are increasingly integrated into autonomous decision-making pipelines, the demand for memory-safe, low-latency, and numerically reliable inference engines has grown. While PyTorch remains the industry standard, the emerging Burn framework offers a Rust-native alternative promising superior performance and safety. However, the lack of rigorous verification for its underlying tensor operations remains a barrier to adoption for high-stakes agentic tasks. In this paper, we present BurnMark, a dual-purpose framework for the systematic evaluation of Burn-based LLM agents. First, we implement a differential fuzzer that validates the correctness of Burn’s tensor arithmetic and autograd engine against PyTorch to detect numerical divergences and computational graph bugs. Second, we provide a performance benchmark focusing on the end-to-end latency of agentic loops, measuring the overhead of Burn with different backends compared to traditional Python environments. Our results demonstrate that while Burn offers major advantages in memory and latency, differential fuzzing is essential to ensuring the logical consistency required for truly reliable agent reasoning.

2 Introduction

While Python’s PyTorch is currently dominating research and production workflows, Rust is emerging as a new, powerful systems language geared towards maximizing performance and safety. It has been gaining increasing popularity in edge or embedded computing environments, both of which are current frontiers for LLM/AI development as a whole. Developing mechanisms to compare the correctness and performance of PyTorch to Rust-based alternatives can prove useful when developing LLM agent based systems within these kinds of environments where Python is not conventionally used.

Our project is a benchmarking and fuzzing tool that is designed to evaluate LLM performance across multiple different executable environments, utilizing the Burn library to compare the performance of LLM models on different backends versus Python’s PyTorch. It has a controlled comparison system, in which we have identical model configurations, input shapes, and evaluation procedures. The tool measures key performance metrics including throughput, time to first token (TTFT), per token latency, generation speed, and more.

This tool useful and relevant to LLM systems as it allows developers and researchers to understand possible tradeoffs between different frameworks, languages, and hardware backends particularly with respect to cases when a Rust-based framework may prove desirable over conventional PyTorch systems. Even in general, this tool can be used to evaluate model architectures and their overall effectiveness. Additionally, it highlights performance bottlenecks and possible implementation differences that can impact the observed results. This tool also includes capabilities to fuzz the functionality of Burn to determine the correctness of the different backends to PyTorch’s ground truth, as the latter already has been battle-tested with it being so commonly used.

3 Background/Preliminaries

Current LLMs are based on transformer architectures that rely on self-attention mechanisms to model long-range dependencies in data. During inference, token generation occurs in two primary stages: the prefill phase, where the initial input prompt is processed in parallel, and the decode stage, where subsequent tokens are generated auto-regressively. Important performance metrics for evaluating these phases include time-to-first-token (TTFT), per-token latency, and overall throughput (tokens per second).

Currently, Python dominates the landscape of LLM deployment, with researchers and practitioners heavily relying on PyTorch [PGM⁺19]. At its core, PyTorch provides an intuitive programming model with robust support for dynamic automatic differentiation [PGC⁺17]. This foundation is heavily augmented by ecosystem libraries like Hugging Face Transformers, which streamline model definition and deployment. While PyTorch is highly flexible and supports robust hardware acceleration through CUDA and Apple’s Metal Performance Shaders (MPS), this flexibility comes with inherent system-level overhead. Most notably, Python’s Global Interpreter Lock (GIL) prevents true multi-threading execution. This can create significant bottlenecks and latency overhead in highly concurrent serving environments, particularly when low latency and high throughput are strict requirements.

To address these system-level constraints, Rust-based frameworks like Burn and Candle have recently emerged. These frameworks aim to improve execution performance, memory safety, and concurrency guarantees without the overhead of a garbage collector or a GIL [LCG⁺17, APZ⁺24]. Recent efforts exploring Rust for high-performance systems and LLM workloads indicate that Burn can be highly reliable and efficient for these tasks [Mis24, HSMB25]. While Burn offers a comprehensive deep learning framework with a modular, multi-backend architecture, Candle provides a minimalist, efficient inference engine heavily optimized for quantized models. Overall, both Burn and Candle aim to offer developers lower-level system control while strictly maintaining memory safety.

Comparing different system-level frameworks for LLM workloads is often challenging due to variances in model configurations, backend behaviors, and measurement methodologies. BurnMark addresses this challenge by providing a benchmarking and fuzzing suite that evaluates both PyTorch and Rust-based frameworks under strictly controlled conditions, ensuring accurate and equivalent performance comparisons.

4 Scope and Use Cases

Its primary scope is to provide a controlled comparison between Rust-based LLM implementations and Python’s PyTorch. We are trying to assess the system’s level performance characteristics.

This tool can be applied in multiple practical situations:

- **Research Use:** The tool can be used to collect data on model performance as a whole, comparing how different LLM architectures perform on provided hardware across a variety of metrics.
- **Backend Evaluation:** Due to how the direct one-to-one model performance comparisons between backends, this tool is useful for understanding different system implementations and how it can affect the performance of the LLM. Depending on the model architecture and computing environment, this tool can help a developer decide which backend is best suited for a given task.
- **Decision Making:** The tool works with CPU, GPU, and Metal, allowing to determine how well common LLM tasks like training, inference, etc. will perform with different models or backends and when to prefer one over another.
- **Correctness Evaluation:** The fuzzing tool allows for developers to verify that certain commonly used functions in Burn are equivalent to their PyTorch counterparts. In cases where there appears to be a mismatch, changes can be made to improve Burn to enable it to compare better to the more robust PyTorch library.

In short, our tool provides engineers and researchers a practical way to study how different large language model implementations, system-level backend tools, and hardware configurations can influence system performance.

5 System Design and Key Components

Our model consists of two key parts: a benchmarking tool and a fuzzer.

5.1 Benchmarking Tool

The benchmarking tool is comprised of three separate sections. The first to measure the backend throughput of Burn, next to benchmark LLM inference, and finally benchmarking LLM training.

5.1.1 Burn Backend Throughput

This section is focused primarily on evaluating backend throughput. For our testing, we created two small toy GPT models, but this can be substituted out for other models. The same custom GPT model is run using different backends on the same inputs, with the goal being to exclusively isolate how a given backend performs.

Metrics evaluated include tokens per second, average time per iteration, 90/95th percentile latency, and finally peak memory. Tokens per second directly evaluates how quickly data can be pushed through a given model, being the critical metric in this section. Average time per iteration determines the average cost of a single forward pass through the model. 90/95th percentile iteration times judge how consistent the latency is between different passes on the backend, providing additional context to the average time.

Overall, this section offers a robust suite of benchmarks geared towards evaluating many different backends at once through Burn. A purely Python-based version is also provided. As Burn matures and offers more backends, this tool will still easily be able to compare against those as well.

5.1.2 LLM Inference Benchmarking

Within this section, we benchmark inference performance with the Candle backend using a real pre-trained model that can be selected beforehand. Provided a HuggingFace token, a model will be fetched from the site and inference will then be performed with several real prompts. By default, TinyLlama is used (in GGUF format) but different models can be incorporated.

Metrics we track in this section include time to first token (TTFT), decode throughput, 90/95th percentile per-token latency, total end-to-end time, and peak memory. TTFT is the time from when the model receives the input prompt to when the first output token is received, a valuable metric to judge how long model response takes as prompt size scales up. Decode throughput measures how quickly a model generates tokens after it produces its first output token, useful for comparing models to one another. 90/95th percentile per-token latency measures generally how consistently the model generates tokens. End-to-end time is just the total time from receiving the prompt to when the output generated by the model is completed.

In short, this section's metrics provide context to how Candle's inference tools compare to PyTorch-based systems, allowing for a good frame of reference for deciding when to use one over the other or to determine how capable the Candle library is overall.

5.1.3 LLM Training Benchmarking

This section takes advantage of Burn's features to provide a visual on how well a model is training, showing live loss curves for training and validation during each epoch. For the sake of our testing, we again used some basic toy GPT models and trained them using random data. Memory use is also logged during this full process. Different backends can be used, allowing for effective comparison between the Rust options and PyTorch.

Metrics collected in this section include items per second, 90th/95h percentile step latency, and peak memory use much like section 1. Items/sec uses Burn's built-in throughput metric, displaying the sequences processed per second through the entire training process (forward pass, backward pass, optimize update). The remaining metrics are much like before.

Ultimately, this rounds out our set of benchmarking tools so that we cover the three main useful areas in which we can evaluate an LLM system: backend choice, inference, and training. Comparisons to Python allow for a complete picture for an end user to make choices based on their use case. This

is extremely valuable for LLM systems as having all these metrics within one tool allows for ease of use when studying LLMs/agents within different environments.

5.2 Fuzzer

To ensure the reliability and correctness of the Rust-based backends evaluated in our benchmarking suite, BurnMark incorporates a dedicated fuzzing module. Because performance metrics are only valuable if the underlying calculations are mathematically sound, this fuzzer is designed to rigorously test the computational graph construction, automatic differentiation (autodiff) engine, and core tensor operations of the Burn library. It achieves this by performing continuous differential testing, utilizing PyTorch as the ground truth oracle.

With differential fuzzing, we compare multiple implementations of one specification to check against each other. For example, we compare Burn’s `NdArray` against Burn’s binding to `LibTorch`. `LibTorch` (PyTorch) acts as the ground truth. Our inputs are the randomly generated program, represented as our custom Abstract Syntax Tree (AST), and the random seeded values. Our outputs are the tensor values and gradient.

5.3 AST Generation

A *leaf* is an input tensor to our program. We can generate multiple leaves that are strung together with *operations*, which are nodes that actually do the computation in our program. Each operation creates a *register*, which can be an intermediate computation that has a probability of being borrowed again in the future, especially if it is a “sink” register. A sink register is one that is not used by other registers. The more recent the sink register, this means the “deeper” within the tree we are, and the more likely it is to be reused so we create dense trees with less dead ends. Finally, we do a **backward** pass on the last register, which propagates to at least one leaf.

We can support the following operations:

- **Linear Operations:** Matrix multiplications (`matmul`) and standard linear transformations.
- **Element-wise Operations:** Additions, subtractions, multiplications, and broadcasting.
- **Non-linear Activations:** Common activation functions such as ReLU, Sigmoid, and Softmax.
- **Transformations:** Reshaping, transposing, and slicing.

The random bytes are provided by `arbitrary`, and our `enum` for node operations is semi-randomly constructed. Due to shape and rank transformations that are possible, we need type inference within our AST generation so we do not keep generating invalid programs. As we pattern match throughout the AST/program, we view the previous registers, from which we can borrow shapes and valid tensors that allow us to use the current operation.

```

1 r0 [8x4] = leaf(8x4)
2 r1 [4x4] = leaf(4x4)
3 r2 [8x4] = r0.matmul(r1)
4 r3 [8x4] = r2.powf(r2)
5 r4 [1x1] = meanA11(r3)
6 backward(r4)

```

Listing 1: An example register file doing autograd.

For example, to generate a valid `matmul` operation, our generation:

- Picks a register (higher probability of choosing sink register)
- Filters all registers that are `matmul` compatible with the source register
- If empty, generates a unary operator instead and returns
- Else, picks register from the filtered list (higher probability of choosing sink register)
- Pushes the shape valid instruction to our register file vector

In this way, we are shape borrowing throughout our program, allowing for more valid AST generations and rapid fuzzing. Some improvements to our AST generation is that instead of picking a random register, we can put “compatible” registers in groups, and randomly pick a group. This is important for higher rank tensor support (3D, 4D).

6 Usage and Examples

All three sections can be run in the following ways. By default,

```
cargo run --release -p llm_benchmark --features candle,train -- --model tinyllama
```

will run all three sections. The metal feature is optional for GPU acceleration when running on a compatible mac. Model can be selected with the appropriate model flag. For certain models, a HuggingFace token will need to be set in the environment as such:

```
HF_TOKEN=hugging_face_token OR HUGGING_FACE_HUB_TOKEN=hugging_face_token
```

Currently, we have four models supported by default in our tool: TinyLlama, Phi3, Llama3-1b, and Llama3-3b. The latter two require the token while the former two do not. Other models can be added to the model registry in the code if a user desires for further selection.

Taking out the candle flag will prevent Section 2 from running and taking out the train flag will prevent Section 3 from running. This can be done depending on the use case to only run desired sections.

The Python equivalents of sections 2 and 3 can be run with the following command:

```
python3 llm_benchmark.py --model llama3-1b --device cpu
```

For any runs using the LibTorch backend in Burn, the tch feature needs to be included as well, with LibTorch also being installed and findable on the system.

For the fuzzer, to run tensor single operation fuzzing, perform the command:

```
cargo +nightly fuzz run fuzz_single_op
```

For operation fuzzing in general:

```
cargo +nightly fuzz run fuzz_tensor_ops
```

Finally, to run the fuzzer on one particular example provided, or a user provided fuzzing case:

```
cargo +nightly fuzz run example
```

6.1 PyTorch Benchmark

Prompt	Prompt Tokens	Output Tokens	TTFT (ms)	Decode Tok/s	P50 Latency (μ s)
short-creative	10	0	94.19	0.00	0
short-factual	13	0	124.76	0.00	0
medium-code	32	55	325.05	30.21	33102
medium-reasoning	50	0	591.32	0.00	0
long-summaries	181	150	2312.20	29.11	34641

Table 1: PyTorch inference benchmark results for TinyLlama-1.1B across prompt types.

PyTorch’s benchmarking results have variability and stability depending on the model and the prompt characteristics. For TinyLlama-1.1B from Table 2, it can be seen that there are several prompts that resulted in zero output tokens, which might be due to having an earlier termination in token generation, thus leading to having a zero decode throughput. However, there were some prompts

Prompt	Prompt Tokens	Output Tokens	TTFT (ms)	Total Time (s)	Decode Tok/s	P50 Latency (μ s)
short-creative	9	64	103.62	2.47	27.05	37505
short-factual	12	128	138.34	4.93	26.71	37880
medium-code	25	256	282.42	10.02	26.30	38248
medium-reasoning	44	200	607.17	8.18	26.41	38188
long-summaries	158	107	2372.60	6.49	26.01	38468

Table 2: PyTorch implementation (MPS backend) inference benchmark results for Llama-3.2-1B across prompt types.

that generated output tokens and achieved a throughput of approximately 29 to 30 tokens per second, while taking per-token latency between 33-35 ms.

For Llama-3.2-1B, the metrics and behavior is more consistent across all of the prompts. It got a decoding throughput that was pretty constant between 26 to 27 tokens per second, with a per-token latency that stayed around 26-27 ms. This suggests that the PyTorch implementation has a predictable decoding performance, with very small variance the second that token generation begins.

Across both of the models ran on PyTorch, the time-to-first-token (TTFT) increases significantly with the prompt length. Having this trend is to be expected because naturally as prompt length increases, the model will require additional time to process the input tokens before generating an output response. However, this is a valuable metric to see how compare how different models scale with increased prompt size, allowing for better evaluation of different model architectures.

6.2 Burn with Candle Benchmark

The results of Table 4 and Table 3 shows a significant difference between TinyLlama-1.1B and Llama-3.2-1B for Candle, Rust’s implementation.

First, in Table 4, it demonstrates that it is more consistent across all of the prompts. For Candle’s TinyLlama, it has zero output tokens for short creative and medium reasoning prompts, but in contrast, in Llama-3.2, it consistently generates outputs for every prompt. This means that there is no early termination for Candle in Llama-3.2.

Now looking at the time-to-first-token (TTFT), it has similar trends in both of the models, where it increases with the length of the prompt.

For latency, TinyLlama has a more consistent per-token latency when generation occurs, having approximately 17-19 ms. But Llama-3.2 maintains a bigger range of 16-20 ms, but also has a consistent output generation across all of the prompts.

Also note that TinyLlama occasionally produces no output or oddly short output. This is likely due to the model being particularly small and using greedy decoding which can lead to EOS tokens being output much earlier than it should. In comparison, we can see that the bigger, more robust Llama models consistently produce reasonable output.

Prompt	Prompt Tokens	Output Tokens	TTFT (ms)	Total Time (s)	Throughput (output tok/s)	P50 Latency (μ s)
short-creative	10	0	128.43	0.13	0.00	0
short-factual	13	106	163.70	2.03	56.81	16998
medium-code	32	57	448.95	1.44	57.60	17091
medium-reasoning	50	0	628.48	0.63	0.00	0
long-summaries	181	29	2356.40	2.92	51.13	19564

Table 3: Candle (Rust) implementation inference benchmark results fo TinyLlama-1.1B.

Prompt	Prompt Tokens	Output Tokens	TTFT (ms)	Total Time (s)	Decode Tok/s	P50 Latency (μ s)
short-creative	9	64	103.94	1.11	63.85	15949
short-factual	12	128	136.68	2.51	54.03	17293
medium-code	25	256	278.82	5.07	53.48	18519
medium-reasoning	44	200	490.62	4.16	54.47	18327
long-summaries	158	150	1816.52	4.84	49.65	20161

Table 4: Candle (Rust) implementation inference benchmark results for Llama-3.2-1B Q4_K_M.

6.3 PyTorch vs Candle

Model	Backend	Prompt	TTFT (ms)	Throughput (tok/s)	P50 Latency (μ s)
Llama-3.2	PyTorch	short-creative	103.62	27.05	37505
Llama-3.2	Candle	short-creative	103.94	63.85	15949
Llama-3.2	PyTorch	medium-code	282.42	26.30	38248
Llama-3.2	Candle	medium-code	278.82	53.48	18519
Llama-3.2	PyTorch	long-summaries	2372.20	26.01	38468
Llama-3.2	Candle	long-summaries	1816.52	49.65	20161

Table 5: Comparison of PyTorch and Candle inference performance for Llama-3.2-1B.

When comparing the differences between PyTorch and Candle implementations, there is a clear performance difference in its decoding throughput, as can be seen in Table 5. For both the TinyLlama and Llama models, Candles implementation has a significantly higher decoding throughput than PyTorch by approximately $2\times$ to $2.5\times$. For example, looking at the medium-code prompt, Candle gets 57.6 tokens per second for TinyLlama compared to PyTorch’s 30.21 tokens per second.

Additionally, Candle also has lower per-token latency. While PyTorch achieved a latency range of 33-38 ms per token, Candle got approximately 16-20 ms per token. This suggests that in terms of latency, Candle is more efficient in token generation.

TTFT is also another important comparison between PyTorch and Candle as both achieved an increase in latency as the length of the prompt increases. However, Candle received lower TTFT in medium-code and long-summaries, which means that it may be more efficient in some cases.

Overall, it can be concluded that Candle can be a viable backend for inference due to its excellent throughput and latency, but PyTorch is more stable and reliable considering its battle-tested nature.

6.4 Fuzz Bugs Uncovered

There were 17 different bugs we uncovered with our current BurnMark module. Many more can be uncovered in ‘continuous’ mode, higher rank tensor support, and more operations. Some issues include incorrect gradients when dividing tensor by itself and floating point precision issues with powf() and log() (possibly caused by SIMD vectorization).

```

1 === AutogradProgram [2x152] (max_leaves=4) ===
2 x_0 = input(2x152, 8 seed bytes, fallback) [requires_grad, seed]
3 t0 = x_0 # accumulator seed
4 t1 = log(t0)
5 grads = backward(t1)
6
7 oracle mismatch [0]: NdArray=-3.375, LibTorch=-3.371348 (grad x_0)
8 oracle mismatch [1]: NdArray=-1.3789063, LibTorch=-1.3772159 (grad x_0)
9 oracle mismatch [3]: NdArray=-3.7578125, LibTorch=-3.7527833 (grad x_0)
10 ...

```

Listing 2: Autograd mathematical bug.

Most importantly, our fuzzer successfully identified a critical memory management bug within Burn’s LibTorch backend binding. The issue was isolated using an auto-generated tensor operation sequence. Below is a realistic transformer example that computes a decorrelation/energy regularizer on a set of flat logits:

```

1 fn transpose_repro_loss<B: AutodiffBackend>(logits_flat: Tensor<B, 2>) -> Tensor<B, 1>
2 {
3   // Decorrelation/energy regularizer on logits (legit auxiliary signal).
4   // Build a square Gram matrix so transpose/add matches shapes.
5   let t0 = logits_flat.clone().transpose().matmul(logits_flat);
6   let t1 = t0.clone().transpose();
7   let t2 = t1.clone() + t0.clone();
8
9   // Penalize large correlations (L2 on Gram matrix).
10  (t2.clone() * t2).mean()
11 }

```

Listing 3: LLM tensor operation sequence triggering the FFI memory bug.

```
1 SECTION 3: Training Throughput + Burn TUI Dashboard
2 =====
3
4 Model: TINY-TRANSPPOSE (training)
5 [ Training: LibTorch-Autodiff (CPU) | tiny-transpose model | 3 epochs | batch=4 | seq
  =64 ]
6
7 === PANIC ===
8 A fatal error happened, you can check the experiment logs here => '/tmp/
  llm_benchmark_train/LibTorch-Autodiff_(CPU)/experiment.log'
9 thread 'main' (71928) panicked at .../burn-tch-0.20.1/src/ops/base.rs:361:40:
  called 'Result::unwrap()' on an 'Err' value: Torch("unsupported operation: some
  elements of the input tensor and the written-to tensor refer to a single memory
  location. Please clone() the tensor before performing the operation. \nException
  raised from assert_no_partial_overlap...")
```

Listing 4: Bug output with attempted LLM training.

The bug stems from how the Foreign Function Interface (FFI) handles tensor cloning and memory views. In tensor frameworks, operations that modify a tensor’s view (such as a `transpose`) require a proper clone if the tensor is being modified in-place without altering the original underlying data.

Through our fuzzer’s automated forward and backward pass evaluations, we discovered that Burn’s LibTorch binding was incorrectly performing a shallow clone rather than a deep copy, despite the explicit `clone()` calls. When the transposed view was modified or used in subsequent graph operations, the shallow clone caused unintended memory sharing across the FFI boundary, leading to incorrect gradient calculations and memory corruption.

The resolution to this bug requires maintaining an explicit parent/child relationship within the tensor metadata. By tracking these relationships, the backend can safely detect when memory sharing exists and force a deep copy during in-place view modifications.

```
1 // Before (Buggy): The new tensor loses its connection to the original storage
2 pub fn swap_dims(tensor: TchTensor, dim1: usize, dim2: usize) -> TchTensor {
3     let tensor = tensor.tensor.transpose(dim1 as i64, dim2 as i64);
4     TchTensor::new(tensor)
5 }
6
7 // After (Fixed): The new tensor explicitly shares the parent's storage
8 pub fn swap_dims(tensor: TchTensor, dim1: usize, dim2: usize) -> TchTensor {
9     let storage = tensor.storage;
10    let tensor = tensor.tensor.transpose(dim1 as i64, dim2 as i64);
11    TchTensor::partial(tensor, storage)
12 }
```

Listing 5: The required fix in Burn’s LibTorch backend. By using `TchTensor::partial` and passing the original storage, the parent/child memory relationship is explicitly maintained.

As shown in Listing 5, the original implementation of `swap_dims` instantiated a new `TchTensor` wrapper around the transposed LibTorch tensor. Because it used `TchTensor::new`, Burn’s higher-level graph lost the reference to the original tensor’s memory allocation, incorrectly assuming this new view was an independent, deeply-copied tensor.

The fix ensures the Rust-side memory metadata accurately reflects that the transposed tensor is merely a view sharing the same underlying data. With this explicit parent/child relationship maintained, Burn’s graph can correctly track memory sharing and force deep copies only when necessary.

This discovery highlights the necessity of BurnMark in real LLM frameworks. Similar edge-case oversights can be systematically caught and fixed, ultimately ensuring consistent, mathematically sound behavior across all of Burn’s diverse hardware backends, and consequently the LLMs that depend on it.

7 Limitations and Future Work

One limitation is that our tool is currently limited to only a subset of languages and model architectures. Even though our tool supports Rust frameworks and references to Python’s PyTorch, it is only limited

to Candle and PyTorch for inference benchmarking. We also only support 4 models by default, though it is easily extensible from a user side by adding to our model registry. A better front-end interface to enable easy model selection through providing links to HuggingFace would be ideal.

Additionally, we only ran our benchmarking on CPU rather than an external GPU due to hardware limitations faced by our team. It does not example provide a real-world deployment as GPUs are commonly used for LLM references. Thus, the performance metrics gathered would not be as efficient compared to GPU efficiency. That being said however, our data is still useful in the context of edge computing environments which may be more reliant on CPU in certain contexts.

Finally, we currently only benchmark performance on a limited set of prompts. While this is still useful data and can be again easily extended as the user desires, once again having a more easy to use front-end to the tool would allow for users to more easily select different prompts.

Some potential avenues for future work include:

- Adding additional comparisons to PyTorch for other alternatives like JAX, Fastai, Tensor-Flow/Keras, and more could be useful for greater coverage of different system backends.
- More workloads: Instead of just a limited set of simple prompts for inference, we could incorporate larger data sets and longer sequences. The same goes for training in which we can use more meaningful data and non-toy GPT models.
- Additional Benchmarks: We can incorporate other metrics such as energy consumption, possible accuracy for training, etc.
- Easy way to select backends for Section 2: Currently, for the inference benchmarking, we have only configured it with the Candle backend. While it is fairly trivial to adjust to other backends, allowing for this to be more easily configurable outside of code adjustments from the end user side would be ideal.
- More Flexibility through a UI: Having a more robust UI rather than a command line interface could prove useful both for visibility and for more modularity. Currently, adding models to benchmark will need to be done manually through the code with minor adjustments, but ideally would be able to provide a link of some sort to a model on HuggingFace to streamline the process for an end user. The same applies for selecting different prompts as discussed before.
- More Fuzzing Targets: Being able to fuzz more parts of the Burn library is also a great direction of future work and fuzzing higher dimensional tensor operations.

8 Acknowledgements

As part of this project, we made use of LLM models like Claude and ChatGPT during our code development process. These were used to help us refactor and debug code for our project as well as new functionality.

References

- [APZ⁺24] Javad Abdi, Gilead Posluns, Guozheng Zhang, Boxuan Wang, and Mark C. Jeffrey. When is parallelism fearless and zero-cost with rust? In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2024.
- [Bas25] Samit Basu. Rhdl: Rust as a hardware description language. In *Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE '25)*, 2025.
- [BBC⁺19] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: Fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 975–985. ACM, 2019.

- [FFG⁺25] Nick Fitzgerald, Corey Farwell, Manish Goregaokar, Simonas Kazlauskas, et al. rust-fuzz/libfuzzer. <https://github.com/rust-fuzz/libfuzzer>, Jul 2025. GitHub repository.
- [FSGW23] Andreas Fried, Maximilian Stemmer-Grabow, and Julian Wachter. Register allocation for compressed isas in llvm. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC '23)*, New York, NY, USA, 2023. ACM.
- [G⁺25] Roberto Gioiosa et al. Hardware-software co-development for emerging cxl architectures. In *Proceedings of the International Symposium on Memory Systems (MemSys '25)*, 2025.
- [Gre93] George D. Greenwade. The Comprehensive Tex Archive Network (CTAN). *TUGBoat*, 14(3):342–351, 1993.
- [HPC⁺13] Eric Holk, Milinda Pathirage, Arun Avinash Chauhan, Andrew Lumsdaine, and Nicholas D. Matsakis. Gpu programming in rust: Implementing high-level abstractions in a systems-level language. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 2013.
- [HSMB25] Zhaolan Huang, Kaspar Schleiser, Gyungmin Myung, and Emmanuel Baccelli. Ariel-ml: Computing parallelization with embedded rust for neural networks on heterogeneous multi-core microcontrollers. *arXiv preprint arXiv:2512.09800*, 2025.
- [IB25] Nafees Iqbal and Jed Brown. Enhancing type safety in mpi with rust: A statically verified approach for rsmapi. *arXiv preprint arXiv:2509.10803*, 2025.
- [LCG⁺17] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. The case for writing a kernel in rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys '17)*, 2017.
- [Lea18] Allan Leal. autodiff, 2018. Accessed 2025-12-10.
- [Maz25] Laurent Mazare. Laurentmazare/tch-rs. <https://github.com/LaurentMazare/tch-rs>, Nov 2025.
- [MC20] William Moses and Valentin Churavy. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 12472–12485. Curran Associates, Inc., 2020.
- [McK98] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10:100–107, 1998.
- [MFH⁺25] Eduardo M. Martins, Leonardo G. Faé, Renato B. Hoffmann, Lucas S. Bianchessi, and Dalvan Griebler. Npb-rust: Nas parallel benchmarks in rust. *arXiv preprint arXiv:2502.15536*, 2025.
- [Mis24] Rajavi Mishra. High-performance machine learning inference systems with rust: A review of techniques, tools, and optimizations. *International Research Journal of Engineering and Technology (IRJET)*, 11(11):639–644, Nov 2024.
- [NAT⁺20] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. Predictable accelerator design with time-sensitive affine types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.
- [PGC⁺17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS 2017 Autodiff Workshop*, 2017.
- [PGM⁺19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

- [The25] The Rust Project Developers. *The Rust Reference*. The Rust Foundation, 2025. Accessed: 2025-12-06.
- [TWS⁺25] Ruize Tang, Minghua Wang, Xudong Sun, Lin Huang, Yu Huang, and Xiaoxing Ma. Converos: Practical model checking for verifying rust os kernel concurrency. In *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC '25)*, 2025.
- [YSZ19] Zeming Yu, Linhai Song, and Yiying Zhang. Fearless concurrency? understanding concurrent programming safety in real-world rust software. *arXiv preprint arXiv:1902.01906*, 2019.
- [YTD⁺23] Chenyuan Yang, Yuxing Tu, Yinlin Deng, Hanchi Li, Jiayi Yao, and Lingming Zhang. Fuzzing automatic differentiation in deep-learning libraries, Feb 2023.