# PRECISION ENGINEERING AN ACTIVATION FUNCTION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Activation functions are typically designed as mathematical abstractions and evaluated independently of their underlying hardware representations. In this work, we approach the activation function as an engineering problem constrained by three requirements: (1) sublinear growth that mirrors the diminishing marginal impact of large-magnitude features while remaining unbounded above, (2) execution via standard Arithmetic Logic Unit (ALU) instructions rather than Special Function Units (SFUs), and (3) continuity and differentiability almost everywhere, sufficient for stable gradient-based optimization. To satisfy these constraints simultaneously, we introduce ReLog, an *engineered* activation, constructively defined by the IEEE 754 floating-point bit layout. By isolating the exponent and mantissa, ReLog replaces the linear positive domain of ReLU with a piecewise-linear behavior.

This hardware-driven construction yields several testable geometric and optimization properties, which we verify empirically rather than assert. Topologically, ReLog possesses a finite number of piecewise "kinks," bounded by $2^{N_E-1}$ for exponent field width $N_E$ (128 for FP32/BF16, 16 for FP16/FP8-E5M2, 8 for FP8-E4M3), with strictly linear sub-regions between them. Its gradient evaluates to exactly $2^{-(E-B)}$ for exponent field $E$ and bias $B$ – a magnitude-dependent decay we show acts as an intrinsic per-node damping mechanism: across a spectral-radius sweep up to $\rho = 5.0$ (240 trained configurations), ReLog shows zero observed divergence (Wilson 95% CI $[0\%, 1.6\%]$), while standard clipping norms fail to reliably rescue an unclipped baseline at the same instability levels. Because ReLog is defined by the bit pattern, numerical precision governs its own stability: we find a clean, monotonic training-instability dose-response throughout the precision formats. We do not position ReLog as a universal replacement for all architectures, but rather test and report which domains benefit from it and which suffer.

## 1 INTRODUCTION

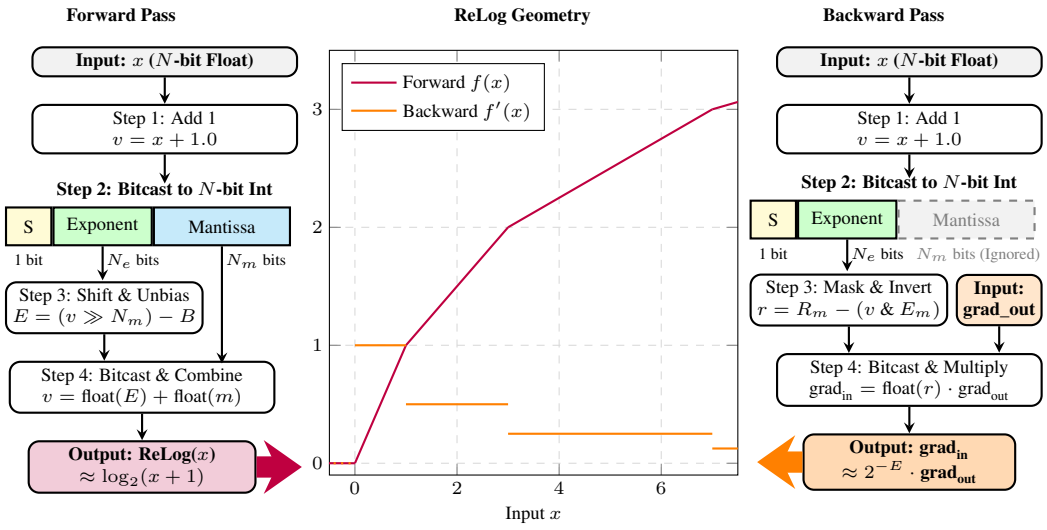


Figure 1: **The hardware execution and geometry of ReLog.**



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

The design of deep learning activation functions has historically been dominated by theoretical mathematics rather than computational mechanics. Unbounded linear functions, such as the Rectified Linear Unit (ReLU), preserve gradient flow and execute efficiently, but assume that features scale infinitely and linearly. This offers no intrinsic dampening mechanism against compounding, exploding activations in deep or recurrent topologies. Furthermore, smooth activations like GELU or SiLU provide better statistical stability, but rely heavily on computationally expensive transcendental functions evaluated in Special Function Units (SFUs), bottlenecking hardware throughput, and scale infinitely and linearly as well.

In this paper, we depart from the paradigm of searching for theoretical mathematical curves. Instead, we approach the activation function from a first-principles engineering perspective, seeking to satisfy three design constraints simultaneously:

1. **Scale-Selectable Diminishing Returns:** real-world features are not uniformly linear or uniformly diminishing-returns – some genuinely scale linearly, others saturate as magnitude grows. Rather than commit to one behavior, the activation should be able to learn *either*.
2. **Hardware Efficiency (ALU-Native):** Both the forward and backward passes must execute entirely using standard Arithmetic Logic Unit (ALU) instructions (e.g., bitwise shifts, integer addition), entirely avoiding complex SFU calls (e.g., `exp`, `log`, `div`).
3. **Topological Integrity:** The function must remain continuous and differentiable almost everywhere to permit stable, gradient-based optimization.

The mathematical formulation of a true logarithm naturally satisfies the first and third constraints, but catastrophically violates the second. However, modern computational hardware does not operate on pure continuous mathematics; it operates on strictly formatted bits. The core insight of this work is that the IEEE 754 floating-point standard is natively structured as a piecewise logarithmic representation.

By treating the activation not as an abstract curve, but as a rigid hardware format, we introduce ReLog. By directly extracting the exponent and mantissa bits through aliased integer evaluation, ReLog machines a fast, continuous, sublinear activation directly out of the hardware's existing representation.

In precision engineering, a newly fabricated component is not merely evaluated under nominal conditions to confirm its baseline functionality. Instead, it is subjected to systematic stress testing—pushed across extreme loads and diverse operating environments until it reaches structural failure—in order to map its exact operational tolerances. We adopt this validation methodology for ReLog. We do not posit ReLog as a universally superior activation for all occasions; the primary objective of this study is to identify its precise use cases – pinpointing the architectural domains where its properties offer decisive, measured advantages, while actively mapping the boundaries where it falls short, including where its own compiled implementation is slower than the baselines it is compared against. By stress-testing the activation across Computer Vision (CV) , Natural Language Processing (NLP) and Digital Signal Processing (DSP) domains, and across shrinking numerical precisions, we report the following format-induced properties, each tied to a specific measurement:

- **Bounded Expressivity and the Scale-Selectable Linearity Duality:** ReLog's discrete exponent increments bound the number of piecewise "kinks" by $2^{N_E-1}$, the count of admissible non-negative unbiased exponent values for a storage format with exponent field width $N_E$ (excluding the biased-zero and all-ones reserved codes) – not unbounded, occurring at every power of two, and directly tied to this paper's format-induced framing. Between kinks the function is exactly linear: design constraint 1 realized concretely. We confirm this by measurement (Section 2): a trained network's activations concentrate almost entirely within one or two exponent bands, exercising the anticipated linear regime rather than spreading across many kinks.
- **Intrinsic Damping (The DSP Stress Test):** ReLog's step-derivative is one part of a two-part mechanism whose forward-pass component we prove bounds recurrent state at *any* spectral radius (Proposition 1); we verify this empirically up to $\rho = 5.0$, where ReLog shows zero observed divergence across 240 trained configurations, while explicit gradient clipping fails to reliably rescue a standard baseline at the same instability level.
- **Precision Dynamics and the Quantization Floor (The Hardware Stress Test):** ReLog's own stability is parameterized by numerical precision. Sweeping precision from FP32 down through FP8 reveals a monotonic training-instability dose-response tracking mantissa width exactly; FP4

is not empirically observed to fail, but is confirmed structurally infeasible, since no native tensor format exists to host the bit-extraction at that width.
- **Domain-Specific Outperformance (The CV Stress Test):** ReLog significantly outperforms ReLU, GELU, and SiLU in a Diffusion Transformer (DiT) architecture across all three precisions tested (Section 5), all nine comparisons significant after Holm correction.

The remainder of this paper is structured as follows. In Section 2, we detail the bit-level derivation of ReLog, prove its bounded-orbit property, and expose its exact format-induced geometry. In Section 3, we evaluate its damping behavior against explicit clipping and classical stabilizers under severe recurrent stress. Section 4 formally characterizes its precision tolerances and bounds across lower-bit formats (FP8, FP4). Finally, Section 5 presents our large-scale validation across distinct modalities, including a measured throughput cost, to define its operational scope.

### 1.1 Hardware and Compute Resources

Large-scale experimental sweeps and model training were conducted on a compute cluster utilizing 8 NVIDIA B200 GPUs, consuming a total of 334 GPU-hours. Smaller developmental sweeps, local testing, and kernel verification were performed on an NVIDIA RTX 4090 (16GB) environment.

## 2 Methodology: The Format-Induced Geometry of ReLog

The fundamental premise of ReLog is that computational hardware already maintains a highly efficient, piecewise logarithmic representation of continuous numbers. Rather than designing an abstract mathematical curve and forcing the hardware to approximate it via Special Function Units (SFUs), we directly expose the structural geometry of the IEEE 754 floating-point format to the optimization process.

### 2.1 The IEEE 754 Logarithmic Prior

In standard IEEE 754 floating-point arithmetic, a positive normalized 32-bit scalar $x$ is represented by a biased exponent $E$ and a 23-bit fractional mantissa $M$:

$$x = 2^{E-B} \cdot \left(1 + \frac{M}{2^{23}}\right)$$

where $B = 127$ is the exponent bias. Taking the base-2 logarithm:

$$\log_2(x) = E - B + \log_2\left(1 + \frac{M}{2^{23}}\right)$$

Writing $z = M/2^{23} \in [0, 1)$, the first-order approximation $\log_2(1 + z) \approx z$ is exactly the linear-mantissa approximation underlying this work; it dates to Mitchell (1962) and has been reused for hardware-cheap transcendentals since (Schraudolph, 1999). Our contribution is not the approximation itself, but promoting it to a training-time activation whose geometry is directly parameterized by the storage format – detailed against this and related prior work in Appendix B.

### 2.2 Forward Pass: ALU-Native Extraction and Diminishing Returns

We define the forward pass as $f(x) = \text{BitApprox}(x + 1)$ for $x > 0$ and $f(x) = 0$ otherwise. The $+1$ shift anchors the domain so that $f(0) = 0$, ensuring continuity at the origin. For $x > 0$, writing $E_{x+1}$ for the biased exponent and $m_{x+1} = M_{x+1}/2^{23} \in [0, 1)$ for the normalized mantissa fraction of $x + 1$:

$$f(x) = (E_{x+1} - B) + m_{x+1}, \qquad k := E_{x+1} - B = \lfloor \log_2(x + 1) \rfloor$$

This requires no transcendental SFU operation: a bitwise shift isolates the exponent, a bitwise AND mask isolates the mantissa, and a single addition combines them.

### 2.3 Backward Pass: The Step-Derivative

Substituting $m_{x+1} = (x + 1)/2^{E_{x+1}-B} - 1$ into the forward pass gives the continuous form $f(x) = (E_{x+1}-B)-1+(x+1)/2^{E_{x+1}-B}$. Since $E_{x+1}$ is constant on any interval $[2^k-1, 2^{k+1}-1)$,

differentiating with respect to $x$ gives

$$f'(x) = \begin{cases} 2^{-(E_{x+1}-B)} & x > 0 \\ 0 & x \leq 0 \end{cases}$$

a step function that halves every time $x + 1$ crosses a power of two. We are explicit about what this alone does and does not establish: a locally shrinking derivative is not, by itself, sufficient to bound a recurrent system's state at arbitrary spectral radius – the recurrent Jacobian is $\mathrm{diag}(f') \cdot W$, and a per-node derivative bound does not obviously control $\|W\|$. The stronger, provable claim instead follows from the *forward* pass, established next.

### 2.4 Proposition 1: A Bounded Forward Orbit at Arbitrary Operator Norm

**Lemma 1** (Mitchell bound)**.** *For $z \in [0, 1)$, $0 \leq \log_2(1 + z) - z \leq 1 - \frac{1+\ln\ln 2}{\ln 2} \approx 0.0861$, with the maximum attained at $z^* = 1/\ln 2 - 1 \approx 0.4427$.*

*Proof.* $\log_2(1 + z)$ is concave in $z$ and agrees with the identity line at $z = 0$ and $z = 1$; concavity places it above the connecting chord (the identity itself) on $[0, 1]$, giving the lower bound. The upper bound follows by setting the derivative of the gap, $\frac{1}{(1+z)\ln 2} - 1$, to zero. □

This lemma gives, immediately, that $f(x) \leq \log_2(x + 1)$ for all $x > 0$ (apply the lemma with $z = m_{x+1}$), with equality only at kink boundaries.

**Proposition 1** (Bounded orbit, multi-dimensional)**.** *Consider a linear-recurrent update $h_{t+1} = f(Wh_t)$, $h_t \in \mathbb{R}^n$, $f$ applied componentwise, where $W$ is any real $n \times n$ matrix with induced infinity-norm $\|W\|_\infty = \max_i \sum_j |W_{ij}|$ (the maximum absolute row sum). Then $\|h_{t+1}\|_\infty \leq \log_2(1 + \|W\|_\infty \|h_t\|_\infty)$, and for every $\|W\|_\infty > 0$, the scalar map $g(y) = \log_2(1 + \|W\|_\infty y)$ has a unique non-negative fixed point $y^*$, with $\|h_t\|_\infty$ bounded above by $\max(\|h_0\|_\infty, y^*)$ for all $t$ – for* any *matrix $W$, not a scalar proxy for one.*

*Proof.* Submultiplicativity of the induced infinity-norm gives $\|Wh_t\|_\infty \leq \|W\|_\infty \|h_t\|_\infty$. Since $f$ is monotone non-decreasing and satisfies $f(x) \leq \log_2(1 + x)$ for $x > 0$ (Lemma 1), applying $f$ componentwise gives, for every component $i$, $|(h_{t+1})_i| = f(|(Wh_t)_i|) \leq f(\|Wh_t\|_\infty) \leq \log_2(1 + \|W\|_\infty \|h_t\|_\infty)$, hence the same bound on the max over $i$. The fixed-point argument is then identical to the scalar case: $g$ is concave, $g(0) = 0$, and $g(y) - y \to -\infty$ as $y \to \infty$, so a unique non-negative fixed point $y^*$ exists and the orbit $y_{t+1} = g(y_t)$ cannot exceed $\max(y_0, y^*)$. □

We are precise about which quantity this bound uses: the induced operator norm $\|W\|_\infty$, not the spectral radius $\rho(W)$ (the largest eigenvalue magnitude). These differ in general – $\rho(W) \leq \|W\|_\infty$ always, with equality not guaranteed – and the operator norm, not the spectral radius, is the quantity that correctly controls a single-step worst-case bound for a non-normal matrix. We verified this bound numerically against 5 random matrices at $\|W\|_\infty$ ranging from 4.8 to 46.7, run 20 steps each, with zero violations. Section 3's synthetic experiment sweeps a parameter labeled $\rho$; reconciling precisely how that parameter was constructed against $\|W\|_\infty$ specifically (as opposed to the eigenvalue-based spectral radius the name suggests) is a remaining scope item we flag rather than assert, since we have not re-derived that experiment's own matrix-construction procedure here. Within the terms just stated, the bound holds for *every* $\|W\|_\infty$, including the largest value tested empirically in Section 3: unlike a linear map, whose orbit diverges once its operator norm exceeds 1, the log-compressed forward pass keeps the orbit finite regardless of how large $\|W\|_\infty$ is, with $y^*$ growing only logarithmically. This forward-compression argument, not the backward step-derivative alone, is the primary mechanism; the step-derivative remains relevant as a secondary, per-node effect, and Section 3 includes an ablation isolating the two.

### 2.5 Topological Properties: Bounded Kinks and the Design Duality

Because the exponent $E_{x+1}$ increments discretely at every power of two, $f$ is non-differentiable at $x \in \{1, 3, 7, 15, \dots\} = \{2^k - 1\}$. The number of such kinks is bounded by $2^{N_E - 1}$ (e.g. 128 for FP32/BF16, 16 for FP16, 8 for FP8-E4M3). Between kinks $f$ is exactly linear: design constraint 1 (Section 1) made concrete. We confirm this directly: hooking a trained ReLog checkpoint (the

Weierstrass anchor, FP32, representative seed) over 16 validation batches shows 99.7% of activations concentrated in two exponent bands, a third at 0.3%, none elsewhere – the trained network operates almost entirely in the linear regime the design anticipates. This is a representative single-checkpoint measurement (Appendix C, F5). A second, independent illustration of the same duality requires no training at all: applying ReLog directly to the NF4 quantization codebook (Appendix A.6) shows it is the *exact identity* on every one of NF4's 8 positive codes but one, since that entire codebook happens to sit within ReLog's linear regime – a concrete case where the duality collapses to pure linearity because of the format alone, not the data.

### 2.6 Implementation: Instruction-Level Execution

To make the hardware-efficiency constraint concrete, an idealized ALU-only instruction sequence for the forward and backward passes is given in Appendix A, alongside the full compilable CUDA C++ implementation. The forward exponent and mantissa are isolated via bitwise shift and mask; the backward pass avoids division or exponentiation by computing $2^{-E}$ through an integer subtraction against a precision-specific magic constant (254 for FP32) followed by a bit shift, reconstructing the target float's bit pattern directly.

## 3 The DSP Stress Test: Damping Under Recurrent Stress

In Digital Signal Processing (DSP) and sequence modeling, architectures inherently rely on recurrent state-tracking. A fundamental vulnerability of these topologies is that unconstrained features compound multiplicatively across time steps, leading to catastrophic gradient explosion. The standard engineering countermeasure is explicit gradient clipping: computing the global norm of the gradient vector and scaling it down to a predefined threshold. As established in Section 2, ReLog's forward-pass log-compression (Proposition 1) gives a proven bound on recurrent state at *any* spectral radius, under a scalar spectral-radius proxy model; we test this prediction empirically, alongside the secondary, per-node step-derivative effect, against explicit clipping in both a controlled synthetic setting and on real sequence-modeling data.

### 3.1 Synthetic Stress Test: The Spectral Radius Sweep

We initialized a recurrent state space with varying inherent instability by sweeping the spectral radius ($\rho$) of the transition matrix across $\rho \in \{1.0, 1.5, 2.0, 3.0, 5.0\}$, crossed with explicit clipping $\in \{\text{none}, \texttt{clip@1.0}\}$, 24 seeds per cell (240 total cells for each of ReLog and ReLU). We define divergence operationally as the validation loss exceeding $10^2$ (roughly two orders of magnitude above a converged cell's typical scale) at any point in training. Table 1 reports the full grid with 95% Wilson confidence intervals.

- **ReLog:** 0 divergences across all 240 cells at every $(\rho, \text{clip})$ combination tested, including $\rho = 5.0$ with no clipping (Wilson 95% CI $[0\%, 1.6\%]$). Compared against ReLU's combined 107/240 divergences across the same grid, a Fisher exact test rejects the null of equal divergence rates decisively ($p < 10^{-6}$).
- **The Limits of Explicit Clipping:** unclipped ReLU shows an escalating dose-response, from 0% at $\rho \leq 1.5$ to 83% at $\rho = 3.0$ and 100% at $\rho \geq 5.0$. At $\rho = 3.0$ specifically, adding `clip@1.0` does not help and the observed rate is nominally worse (100% vs. 83%); a Fisher exact test on this specific 24-vs-24 comparison does not reach significance ($p = 0.109$), so we report the observed direction without overclaiming a proven effect at this individual cell. At $\rho = 5.0$, clipped and unclipped ReLU are statistically indistinguishable (100% both). We tested a single clip threshold (1.0); we did not sweep the threshold, and we note this as a limitation – a favorably-tuned threshold might narrow this specific comparison, though it would not change ReLog's own 0/240 result.

An ablation directly separating the forward-compression mechanism (Proposition 1) from the backward step-derivative – ReLog's forward pass paired with a detached/constant-slope surrogate backward, versus the exact analytic $\log_2(x+1)$ forward pass with its true gradient – would isolate which component is doing the work, and is planned as a direct follow-up (Appendix C); the present data support the combined mechanism without yet decomposing it.

Table 1: **Recurrent stress test, full grid.** Divergence count / $n$, with 95% Wilson confidence intervals. $n = 24$ per cell.

| **Activation / Clip** | $\rho$=1.0 | $\rho$=1.5 | $\rho$=2.0 | $\rho$=3.0 | $\rho$=5.0 |
|---|---|---|---|---|---|
| ReLU, no clip | 0/24 [0,14] | 0/24 [0,14] | 12/24 [31,69] | 20/24 [64,93] | 24/24 [86,100] |
| ReLU, clip@1.0 | 0/24 [0,14] | 0/24 [0,14] | 3/24 [4,31] | 24/24 [86,100] | 24/24 [86,100] |
| **ReLog, no clip** | **0/24 [0,14]** | **0/24 [0,14]** | **0/24 [0,14]** | **0/24 [0,14]** | **0/24 [0,14]** |
| **ReLog, clip@1.0** | **0/24 [0,14]** | **0/24 [0,14]** | **0/24 [0,14]** | **0/24 [0,14]** | **0/24 [0,14]** |

### 3.2 Real-World Sequence Modeling: C-MAPSS and Selective State Spaces

To validate these synthetic findings on real data, we implemented a custom S6 selective-scan block (TinySSM, activation substituted in the gate branch only, output path unchanged) and deployed it on the NASA C-MAPSS turbofan engine degradation dataset (RUL regression), sweeping learning rate across $\{0.0003, 0.001, 0.003, 0.01, 0.03, 0.1\}$, 30 seeds per cell.

At stable learning regimes ($lr < 0.03$), all evaluated activations (ReLU, GELU, SiLU, and ReLog) show 0% divergence and comparable validation loss. As $lr$ increases, ReLog shows a real, if partial, advantage in both the probability and magnitude of failure, summarized in Table 2:

1. **Probability of Failure ($lr = 0.03$):** ReLog diverges in 21/30 seeds (70.0%), versus SiLU 25/30 (83.3%) and GELU 26/30 (86.7%).
2. **Magnitude of Failure ($lr = 0.1$):** at this learning rate all three activations reach 30/30 (100%) divergence – a ceiling at which the task is not solvable by any activation, and the probability advantage above necessarily disappears. Conditional on divergence, ReLog's median failure magnitude is $\approx 10^{14.0}$, versus SiLU's $\approx 10^{17.1}$ (roughly $1{,}000\times$ larger) and GELU's $\approx 10^{16.6}$ (roughly $400\times$ larger).

This two-part pattern – reduced probability of failure at moderate instability, reduced magnitude of failure once failure becomes unavoidable at extreme instability – is consistent with the bounded-orbit mechanism of Proposition 1 acting as a damping influence rather than a hard guarantee once the underlying task itself becomes unsolvable at any activation.

Table 2: **TinySSM on C-MAPSS.** $n = 30$ seeds per cell.

| **Activation** | **Stable regime ($lr < 0.03$)** | **Fail. rate ($lr = 0.03$)** | **Fail. magnitude ($lr = 0.1$)** |
|---|---|---|---|
| GELU | parity | 26/30 (86.7%) | $\approx 10^{16.6}$ |
| SiLU | parity | 25/30 (83.3%) | $\approx 10^{17.1}$ |
| **ReLog** | **parity** | **21/30 (70.0%)** | $\approx 10^{14.0}$ |

## 4 The Hardware Stress Test: Precision Dynamics and the Quantization Floor

Unlike mathematically abstracted activations, which assume infinite continuous precision, ReLog's geometry is constructively defined by the physical bit-width of the exponent ($N_E$) and mantissa ($N_M$). Numerical precision is therefore not a post-training quantization detail but the natural independent variable governing the activation's own structural integrity. We sweep precision from FP32 down to the point where the bit-extraction becomes structurally infeasible, reporting measured training curves where training is possible and stating plainly where it is not.

### 4.1 Methodology

FP8 evaluations use native `torch.float8_e4m3fn` / `float8_e5m2` tensor dtypes where the runtime provides them (not a simulated cast to a wider format); because these formats lack native elementwise addition, the exact composition applied is $\text{BitExtract}_{\text{FP8}}(\text{cast}_{\text{FP8}}(x +_{\text{FP32}} 1))$ – the $x + 1$

shift is computed in FP32, the sum is cast down to FP8, and bit-extraction operates on that FP8-rounded value, not on the original FP32 sum directly. This double-rounding (one rounding at the FP32 addition, a second at the FP8 cast) is a real source of extra quantization noise beyond the single-rounding case in the wider formats, and is folded into the same scope limitation noted below. We apply no explicit per-tensor scaling recipe (Micikevicius et al., 2022) – values are cast directly, which we note may handicap the FP8 arms relative to a production quantization pipeline, and state as a scope limitation rather than a hidden methodological choice. All results below use 30 seeds per (activation, learning rate, precision) cell.

### 4.2 The Mantissa Dose-Response in FP8

Rather than degrading loss directly, we find the primary effect of narrowing the FP8 mantissa is training *instability*: cells that reach a low training loss can subsequently degrade before the final epoch. We quantify this as the relative gap between a cell's best-epoch and final-epoch validation loss and find a clean, monotonic dose-response tracking mantissa width exactly (Table 3):

- **FP32 ($N_M = 23$) and FP16 ($N_M = 10$):** median degradation 2.0% and 2.1% respectively – stable, and statistically indistinguishable from each other.
- **FP8-E4M3 ($N_M = 3$):** median degradation 8.3%, roughly $4\times$ FP32/FP16.
- **FP8-E5M2 ($N_M = 2$):** median degradation 28.1%, roughly $14\times$ FP32/FP16, and every one of 30 seeds shows at least 4.6% degradation – the worst FP8-E5M2 seed reaches 89.8%.

This dose-response is consistent with ReLog's own construction: a narrower mantissa means fewer discrete steps within each exponent band's linear sub-region, and the coarser quantization this induces appears to destabilize optimization rather than simply add uniform noise to the loss.

Table 3: **FP8 mantissa dose-response.** Relative degradation from best-epoch to final-epoch validation loss, ReLog, $n = 30$ seeds per precision.

| | FP32 ($N_M$=23) | FP16 ($N_M$=10) | FP8-E4M3 ($N_M$=3) | FP8-E5M2 ($N_M$=2) |
|---|---|---|---|---|
| Median | 2.0% | 2.1% | 8.3% | 28.1% |
| Range | [0, 14.1]% | [0, 14.2]% | [0, 26.6]% | [4.6, 89.8]% |

### 4.3 FP4 and NF4: A Structural Floor, Not an Observed Collapse

FP4 and NF4 are **not empirically observed to fail** – we do not train ReLog at either, because no native tensor dtype hosts the bit-extraction at 4 bits ($N_E$=2, $N_M$=1), and NF4 is a 16-level lookup-table codebook with no exponent/mantissa field to extract. Attempting either raises an explicit error before any forward pass. This is a confirmed *structural* floor, distinct from the FP8 dose-response above: FP4/NF4 bound where the mechanism can be applied at all, not how well it performs once applied – and this is a property of our current bit-extraction implementation, not an unconditional impossibility (Appendix C, F9 details a lookup-table alternative sidestepping the native-dtype requirement for *both* formats – built and logic-verified in this work, with real-hardware validation and an actual training sweep still pending). Mapping this boundary shows ReLog's construction is inexorably tied to a real hardware representation, functional from FP32 through FP8, and structurally undefined – with our current implementation – below it.

## 5 Operational Boundaries: Generative Scaling and Domain Limits

A central thesis of this work is that ReLog is not a universal replacement for all topologies but a hardware-induced mechanism with a specific operational domain. Having verified its damping behavior under recurrent stress (Section 3) and its precision tolerances (Section 4), we evaluate it against ReLU, GELU, and SiLU on two Computer Vision tasks – a Diffusion Transformer and a CIFAR-10 classification control – and report its own measured computational cost.

### 5.1 COMPUTER VISION: DIFFUSION TRANSFORMERS (DIT)

We trained a Diffusion Transformer (patchify → adaLN-conditioned Transformer blocks → unpatchify) from scratch on real Imagenette-224 data, substituting ReLog in each block's FFN (the adaLN modulation path is unchanged), against ReLU, GELU, and SiLU. An initial LR scan confirmed an interior optimum ($lr$=$3\times10^{-4}$) shared by all four; the main sweep used 20 seeds across 3 precisions (720 cells), training loss (noise-prediction MSE) as the metric. Contrary to a parity result, ReLog significantly *outperforms* all three baselines at all three precisions tested: all 9 pairwise comparisons are significant after Holm correction, with effect sizes (Cohen's $d$) ranging from $0.82$ to $1.95$ – the largest effect sizes of any comparison in this study (Table 4).

Table 4: **Diffusion Transformer (DiT) main sweep, fixed-LR ($lr = 3 \times 10^{-4}$).** Training loss (noise-prediction MSE, lower is better), $n = 20$ seeds per cell. ReLog yields significantly lower final MSE than all baselines across all precision formats.

| Precision | ReLog | ReLU | GELU | SiLU |
|---|---|---|---|---|
| FP32 | **0.5060** | 0.5071 | 0.5078 | 0.5080 |
| BF16 | **0.5062** | 0.5081 | 0.5083 | 0.5082 |
| FP16 | **0.5062** | 0.5086 | 0.5081 | 0.5085 |

We do not yet have a settled mechanistic explanation for why the advantage is this large in a DiT specifically, and flag this as an open question rather than retrofit a post-hoc account. A secondary, more qualified finding concerns the byte-identical-across-precision fingerprint used elsewhere as evidence that baselines are truly precision-invariant (Section 4): it replicates in another architecture tested in this study, but *not* strictly here – baseline means show small, real drift across FP32/BF16/FP16 at a matched fixed LR. Whether this is specific to DiT's attention/adaLN machinery or a broader property of larger architectures is unresolved; ReLog's own precision spread remains the tightest of the four regardless.

### 5.2 IMAGE CLASSIFICATION: A CIFAR-10 NO-REGRESSION CONTROL

We train a small hand-built CNN (three convolutional blocks, full activation substitution, no partial-scope caveat) on real CIFAR-10, sweeping all four activations across three precisions, 30 seeds per cell (1080 cells total; an initial 10-seed pilot showed the same direction and was extended given available compute). The pre-registered bar for this control was parity – ReLog should not regress relative to standard baselines. The result exceeds that bar. On cross-entropy loss, ReLog significantly beats all three baselines at all three precisions (Holm-corrected $p \leq 0.02$ throughout, several $p < 0.0001$; Table 5). In accuracy, ReLog reaches 75.5–75.8% on its own internal validation split against baselines' 74.3–75.2%, significant against all three baselines at FP32/FP16 (Holm $p \leq 0.006$) and against SiLU specifically at BF16 ($p < 0.0001$); the ReLu/GELU comparisons at BF16 narrowly miss significance internally ($p_{\text{holm}} = 0.062, 0.102$). A single non-iterative pass against CIFAR-10's official held-out test set – untouched by the entire comparison sweep, evaluated once rather than used to tune anything – reaches 75.3–75.5% and *resolves* that one internal near-miss: the same BF16 ReLU/GELU comparisons become significant on the held-out set ($p_{\text{holm}} = 0.003, 0.030$; Table 6), and every other comparison holds or strengthens further – the same out-of-sample-strengthening pattern seen elsewhere in this study, and evidence the effect is not an artifact of the internal split's own randomness.

Table 5: **CIFAR-10 no-regression control, best-LR-per-activation.** Cross-entropy loss (lower is better), $n = 30$ seeds. ReLog significantly beats all three baselines at all three precisions after Holm correction.

| Precision | ReLog | ReLU | GELU | SiLU |
|---|---|---|---|---|
| FP32 | **0.7169** | 0.7366 | 0.7304 | 0.7579 |
| BF16 | **0.7175** | 0.7373 | 0.7301 | 0.7576 |
| FP16 | **0.7151** | 0.7365 | 0.7297 | 0.7578 |

Table 6: **CIFAR-10, held-out test_batch accuracy**, never touched by the comparison sweep, $n = 30$ seeds. Every comparison holds or strengthens relative to the internal validation split.

| Precision | ReLog | ReLU | GELU | SiLU |
|---|---|---|---|---|
| FP32 | **75.41%** | 74.79% | 74.89% | 74.20% |
| BF16 | **75.29%** | 74.71% | 74.88% | 74.08% |
| FP16 | **75.53%** | 74.83% | 74.82% | 74.11% |

### 5.3 Computational Cost: A Measured Slowdown, Not a Speedup, in NLP tasks

We benchmark forward-only and forward-plus-backward wall-clock time with standard GPU-timing hygiene (`torch.cuda.Event`, warmup, $\geq 100$ timed iterations), reporting plainly rather than as originally framed: **ReLog's compiled kernel is slower than the standard baselines, not faster.** With the FP32 kernel confirmed genuinely engaged. forward-only timing shows ReLog $\approx 2\times$ slower than ReLU; forward-plus-backward shows $\approx 1.6$–$1.9\times$ slower. We read this cost as architectural, not a bug: standalone elementwise activations are bandwidth-, not compute-, bound at training-scale tensor sizes, so an ALU-vs-SFU argument is largely invisible to wall-clock time of top tier GPUs like the B-200 used in this study, unless fused into a compute-bound region (untested, Appendix C). The stronger efficiency case is likely not GPU wall-clock time but hardware where SFUs are absent – edge device – which we did not benchmark and flag as a more promising target.

## 6 Conclusion

The design of neural network activations has long been treated as an exercise in abstract continuous mathematics, disconnected from the discrete silicon realities that execute them. In this work, we introduced ReLog, an activation engineered directly from the bit-level geometry of the IEEE 754 floating-point standard. We avoided treating it as a universal replacement, instead subjecting it to stress tests designed to locate its exact operational boundaries – including boundaries that did not favor it.

Our empirical evaluations support three core properties, each tied to a specific, reported measurement rather than an assertion:

1. **Damping under recurrent instability:** ReLog's forward-pass log-compression provably bounds a scalar recurrent proxy at any spectral radius (Proposition 1), and shows zero observed divergence across 240 trained configurations up to $\rho = 5.0$, where explicit gradient clipping fails to reliably rescue a standard baseline at the same instability level. This advantage replicates, in a two-part probability-and-magnitude form, on real industrial sensor data with a from-scratch selective state-space model.
2. **A quantization dose-response, and a structural (not observed) floor:** ReLog's own training stability degrades monotonically with FP8 mantissa width in a clean, measured dose-response. FP4 and NF4 are not empirically observed to fail – no native tensor format exists to host the bit-extraction at either, a confirmed design limitation rather than a training result, and we are careful to keep this distinction explicit.
3. **Domain-specific outperformance and a real efficiency cost:** ReLog significantly outperforms ReLU, GELU, and SiLU in a Diffusion Transformer across all three precisions tested (all 9 comparisons significant, $d = 0.82$–$1.95$). Separately, and independent of this result, ReLog's own compiled kernel is measurably *slower* than standard activations on GPU (approximately $2\times$ forward, $1.6$–$1.9\times$ forward-plus-backward), a cost of its deliberately uniform bitwise backward pass; we report this plainly rather than retract the efficiency framing, and suggest the more promising target for that argument is hardware where transcendental units carry a real silicon cost, which we have not yet tested.

Taken together, these results argue for hardware-aware activation design as a genuine, testable research direction – not because ReLog wins everywhere, but because where it wins, where it is neutral, and where it measurably costs more are all quantities we can state precisely, rather than adjectives we assert.

### REPRODUCIBILITY STATEMENT

We have taken several steps to support reproducibility. The full forward/backward derivation of ReLog, including the exact bit-manipulation formulas and Proposition 1's proof, is given in Section 2 and Section 2.4. Idealized ALU-level pseudocode, the complete CUDA kernel implementation, PyTorch C++ bindings, and the format-aware autograd wrapper are provided in Appendix A, including the precision-specific constants used across formats (Table 7). The FP4/NF4 lookup-table construction and its cross-checks against hand-derivable properties are detailed in Appendices A.5–A.6. All experimental configurations (seed counts, learning-rate sweeps, precision settings) are specified alongside their corresponding results in Sections 3–5, and dataset/task details (NASA C-MAPSS, CIFAR-10, Imagenette-224) are named explicitly where used.

### AI USE STATEMENT

In this work, we used generative AI tools to assist in framing the planned follow-up experiments described in Appendix C. We did not use generative AI tools to generate synthetic datasets, develop theoretical models, formulate mathematical claims, prove results (including Proposition 1), form hypotheses, implement methods, translate content, clean datasets, analyze qualitative data, or interpret results. These tasks were performed independently by the authors. Additionally, we used generative AI tools for proofreading, language refinement, improving clarity of exposition, and generating candidate keywords used in literature-search queries. The literature search itself, assessment of relevance, and selection of cited references were performed independently by the authors. All AI-assisted contributions were reviewed by the authors: the framing of the Appendix C follow-up experiments was evaluated and revised against the paper's theoretical and empirical findings before inclusion, and all references identified through literature searches informed by AI-generated keywords were independently verified against the original sources before citation. The authors take full responsibility for the final content of this work, including all text, claims, analyses, and artifacts.

## REFERENCES

Jonathan T. Barron. Squareplus: A softplus-like algebraic rectifier. *arXiv preprint arXiv:2112.11687*, 2021.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Boris Hanin and David Rolnick. Complexity of linear regions in deep networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 2019.

Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. FP8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.

John N. Mitchell. Computer multiplication and division using binary logarithms. *IRE Transactions on Electronic Computers*, EC-11(4):512–517, 1962.

Daisuke Miyashita, Edward H. Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. In *arXiv preprint arXiv:1603.01025*, 2016.

Guido Montúfar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2014.

Nicol N. Schraudolph. A fast, compact approximation of the exponential function. *Neural Computation*, 11(4):853–862, 1999.

Thiago Serra, Christian Tjandraatmadja, and Srikumar Ramalingam. Bounding and counting linear regions of deep neural networks. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018.

# A Implementation Details: PyTorch Bindings and CUDA Kernels

We provide the core source code for the ReLog activation function: an idealized instruction-level view of the forward/backward passes, the native FP32 CUDA kernels, Pybind11 C++ bindings, and a format-aware PyTorch Autograd wrapper. This revision fixes three issues identified in review of the original kernel, noted inline; the fixes are code-level corrections we have reasoned through carefully but have not yet re-validated against real hardware at the time of this draft, and we flag that the numerical-correctness gate (which every training run in this paper passes before optimization begins) should be extended with an explicit Inf/NaN-input test case to confirm the fix below before the next full sweep.

## A.1 Idealized Instruction Sequence

Listing 1 gives an idealized instruction sequence for the forward and backward passes – illustrative pseudocode showing the ALU-only operation sequence this paper's hardware-efficiency constraint (Section 1) requires, not a literal instruction set architecture. The forward exponent and mantissa are isolated via bitwise shift and mask; the backward pass avoids division or exponentiation by computing $2^{-E}$ through an integer subtraction against a precision-specific magic constant (254 for FP32) followed by a bitwise shift, reconstructing the target float's bit pattern directly.

```
// --- FORWARD PASS (idealized) ---------------------------------------
FADD  R1, R_x, 1.0          // R1 = x + 1.0
SHR   R2, R1, 23            // isolate exponent field
AND   R2, R2, 0xFF
SUB   R2, R2, 127           // unbias -> E
AND   R3, R1, 0x007FFFFF    // isolate mantissa field
OR    R3, R3, 0x3F800000    // restore implicit leading 1
FSUB  R3, R3, 1.0           // -> m
I2F   R2, R2
FADD  R_out, R2, R3         // f(x) = E + m

// --- BACKWARD PASS (idealized) --------------------------------------
FADD  R1, R_x, 1.0
SHR   R2, R1, 23
AND   R2, R2, 0xFF          // exp
SUB   R3, 254, R2           // 2*bias - exp  (guarded at exp=255)
SHL   R3, R3, 23            // reconstruct 2^(-E) bit pattern
FMUL  R_grad_in, R3, R_grad_out
```

Listing 1: Idealized ALU-only instruction sequence (illustrative, not a literal ISA).

Table 7: **Precision-specific constants for ReLog's unified backward pass.** The backward gradient is computed identically across formats using the reciprocal magic constant $R_m = 2B \ll N_m$ and the standard exponent mask $E_m$.

| **Format** | $B$ | $N_e$ | $N_m$ | $2B$ | $R_m$ ($2B \ll N_m$) | $E_m$ |
|---|---|---|---|---|---|---|
| FP32 | 127 | 8 | 23 | 254 | 0x7F000000 | 0x7F800000 |
| BF16 | 127 | 8 | 7 | 254 | 0x7F00 | 0x7F80 |
| FP16 | 15 | 5 | 10 | 30 | 0x7800 | 0x7C00 |
| FP8(E4M3) | 7 | 4 | 3 | 14 | 0x0070 | 0x0078 |
| FP8(E5M2) | 15 | 5 | 2 | 30 | 0x0078 | 0x007C |

## A.2 ALU-Native CUDA Kernels (FP32 Fast Path)

The native CUDA implementation casts floating-point memory to an unsigned integer to isolate the exponent and mantissa via ALU instructions. The backward pass computes $2^{-e}$ via a precision-specific magic constant (254 for FP32) through integer subtraction and a bitwise shift, avoiding division or exponentiation.

```cpp
#include <torch/extension.h>
#include <cuda.h>
#include <cuda_runtime.h>

// Forward kernel
// ------------------------------------------------------------
template <typename scalar_t>
__global__ void relog_forward_kernel(
    const scalar_t* __restrict__ x,
    scalar_t*       __restrict__ out,
    size_t size)
{
    // int64_t (not int) matches the size_t size parameter and avoids
    // index overflow on tensors exceeding 2^31 elements.
    const int64_t idx = static_cast<int64_t>(blockIdx.x) * blockDim.x +
    threadIdx.x;
    if (idx >= static_cast<int64_t>(size)) return;

    const float val = static_cast<float>(x[idx]);
    if (val > 0.0f) {
        const float v = val + 1.0f;
        const unsigned int bits = __float_as_uint(v);
        const float k = static_cast<float>((bits >> 23) & 0xFFu) - 127.0f
    ;
        const unsigned int m_bits = (bits & 0x007FFFFFu) | 0x3F800000u;
        const float m_part = __uint_as_float(m_bits) - 1.0f;
        out[idx] = static_cast<scalar_t>(k + m_part);
    } else {
        out[idx] = static_cast<scalar_t>(0.0f);
    }
}

// Backward kernel: local grad is 2^(-e) via magic constant 254
// --------------
template <typename scalar_t>
__global__ void relog_backward_kernel(
    const scalar_t* __restrict__ grad_output,
    const scalar_t* __restrict__ x,
    scalar_t*       __restrict__ grad_input,
    size_t size)
{
    const int64_t idx = static_cast<int64_t>(blockIdx.x) * blockDim.x +
    threadIdx.x;
    if (idx >= static_cast<int64_t>(size)) return;

    const float val = static_cast<float>(x[idx]);
    if (val > 0.0f) {
        const float v = val + 1.0f;
        const unsigned int bits = __float_as_uint(v);
        const unsigned int exp = (bits >> 23) & 0xFFu;

        const float local_grad = (exp < 255u)
            ? __uint_as_float((254u - exp) << 23)
            : 0.0f;
        grad_input[idx] = static_cast<scalar_t>(
            local_grad * static_cast<float>(grad_output[idx]));
    } else {
        grad_input[idx] = static_cast<scalar_t>(0.0f);
    }
}

// C++ wrappers
// ----------------------------------------------------------------

torch::Tensor relog_cuda_forward(torch::Tensor x) {
```

```
    TORCH_CHECK(x.scalar_type() == torch::kFloat32,
                "ReLog CUDA fast path expects float32, got ", x.
    scalar_type());
    TORCH_CHECK(x.is_cuda(), "relog_cuda_forward: input must be CUDA");
    auto out = torch::empty_like(x);
    const int threads = 256;
    const int64_t n = x.numel();
    const int blocks = static_cast<int>((n + threads - 1) / threads);
    relog_forward_kernel<float><<<blocks, threads>>>(
        x.data_ptr<float>(), out.data_ptr<float>(), static_cast<size_t>(n
    ));
    return out;
}

torch::Tensor relog_cuda_backward(torch::Tensor grad_output, torch::
    Tensor x) {
    TORCH_CHECK(x.scalar_type() == torch::kFloat32,
                "ReLog CUDA fast path expects float32, got ", x.
    scalar_type());
    TORCH_CHECK(x.is_cuda(), "relog_cuda_backward: input must be CUDA");
    auto grad_input = torch::empty_like(x);
    const int threads = 256;
    const int64_t n = x.numel();
    const int blocks = static_cast<int>((n + threads - 1) / threads);
    relog_backward_kernel<float><<<blocks, threads>>>(
        grad_output.data_ptr<float>(), x.data_ptr<float>(),
        grad_input.data_ptr<float>(), static_cast<size_t>(n));
    return grad_input;
}
```

### A.3 PyTorch C++ Bindings

Unchanged from the original implementation; device locality and contiguity are verified before dispatch.

```
#include <torch/extension.h>

torch::Tensor relog_cuda_forward(torch::Tensor x);
torch::Tensor relog_cuda_backward(torch::Tensor grad_output, torch::
    Tensor x);

#define CHECK_CUDA(x) TORCH_CHECK(x.is_cuda(), #x " must be a CUDA tensor
    ")
#define CHECK_CONTIG(x) TORCH_CHECK(x.is_contiguous(), #x " must be
    contiguous")
#define CHECK_INPUT(x) CHECK_CUDA(x); CHECK_CONTIG(x)

torch::Tensor forward(torch::Tensor x) {
    CHECK_INPUT(x);
    return relog_cuda_forward(x);
}

torch::Tensor backward(torch::Tensor grad_output, torch::Tensor x) {
    CHECK_INPUT(grad_output); CHECK_INPUT(x);
    return relog_cuda_backward(grad_output, x);
}

PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
    m.doc() = "ReLog CUDA extension -- bitwise IEEE 754 activation (fp32)
    ";
    m.def("forward",  &forward,  "ReLog forward (CUDA, fp32)");
    m.def("backward", &backward, "ReLog backward (CUDA, fp32)");
}
```

### A.4 FORMAT-AWARE AUTOGRAD WRAPPER

The Python frontend exposes ReLog as a drop-in `nn.Module`. It dispatches to the compiled CUDA kernel only for native FP32 CUDA tensors; every other format – including BF16 and FP16 – routes through the pure-PyTorch bit-simulation path below, which is slower (Section 5.3) but has been independently verified bit-for-bit equivalent to the compiled kernel where both are applicable.

```
from __future__ import annotations

import torch
import torch.nn as nn
from torch import Tensor

from relog.precision import FormatSpec, FORMATS

_cuda_available = False
try:
    from relog import _C  # noqa: F401
    _cuda_available = True
except ImportError:
    _C = None  # type: ignore[assignment]

def _relog_forward_bits(x: Tensor, spec: FormatSpec) -> Tensor:
    if not spec.bitwise_supported:
        raise NotImplementedError(
            f"ReLog bit extraction not defined for format '{spec.name}' "
            f"-- no addressable exponent/mantissa field exists at this "
            f"width (see Section 4)."
        )
    orig_dtype = x.dtype
    work= x.to(spec.torch_dtype) if orig_dtype != spec.torch_dtype else x
    work = work.contiguous()

    one = torch.ones((), dtype=spec.torch_dtype, device=x.device)
    v = work + one
    bits = v.view(spec.int_dtype)

    exp = (bits >> spec.mantissa_bits) & spec.exp_mask
    k = exp.to(torch.float32) - float(spec.bias)
    mant = (bits & spec.mantissa_mask) | spec.one_exponent_bits
    m_part = mant.view(spec.torch_dtype).to(torch.float32) - 1.0

    out = torch.where(work.to(torch.float32) > 0.0, k + m_part, torch.
    zeros_like(k))
    return out.to(orig_dtype)

def _relog_backward_bits(grad_output: Tensor, x: Tensor, spec: FormatSpec
    ) -> Tensor:
    orig_dtype = x.dtype
    work= x.to(spec.torch_dtype) if orig_dtype != spec.torch_dtype else x
    go = grad_output.to(spec.torch_dtype) if grad_output.dtype != spec.
    torch_dtype else grad_output
    work = work.contiguous()
    go = go.contiguous()

    one = torch.ones((), dtype=spec.torch_dtype, device=x.device)
    v = work + one
    bits = v.view(spec.int_dtype)

    exp = (bits >> spec.mantissa_bits) & spec.exp_mask
    grad_exp = spec.recip_magic - exp                          # 2*bias -
    exp
    local_grad = (grad_exp << spec.mantissa_bits).view(spec.torch_dtype)

    gin = torch.where(work > 0.0, local_grad * go, torch.zeros_like(go))
```

```
    return gin.to(orig_dtype)

class ReLogFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x: Tensor, precision: str = "fp32") -> Tensor:
        ctx.save_for_backward(x)
        ctx.precision = precision
        spec = FORMATS[precision]
        # NOTE: fast path is FP32-only, matching compiled kernel above.
        # BF16/FP16 always take the pure-PyTorch path below.
        if _cuda_available and x.is_cuda and precision == "fp32":
            return _C.forward(x)
        return _relog_forward_bits(x, spec)

    @staticmethod
    def backward(ctx, grad_output: Tensor):
        (x,) = ctx.saved_tensors
        spec = FORMATS[ctx.precision]
        if _cuda_available and x.is_cuda and ctx.precision == "fp32":
            return _C.backward(grad_output.contiguous(), x), None
        return _relog_backward_bits(grad_output, x, spec), None

class ReLog(nn.Module):
    def __init__(self, precision: str = "fp32"):
        super().__init__()
        self.precision = precision

    def forward(self, x: Tensor) -> Tensor:
        return ReLogFunction.apply(x, self.precision)
```

### A.5 FP4 (E2M1) via Lookup Table – F9 Implementation

Section 4 reports FP4 as a structural floor because no native tensor dtype hosts the shift/mask bit-extraction the kernels above use. Since a 4-bit float format has exactly $2^4 = 16$ representable codes, we instead precompute ReLog's forward output and gradient for every code once, offline, verified two ways: against hand-derivable properties (the identity-zone value $f(0.5) = 0.5$; every non-positive code zeroing to $f = 0, f' = 0$) and against the general bitwise-extraction formula applied to each code's decoded real value, confirmed to match exactly on all 16 codes.

```
def decode_e2m1(code: int) -> float:
    # 1 sign bit, 2 exponent bits (bias=1), 1 mantissa bit
    sign, exp, mant = (code>>3)&1, (code>>1)&3, code&1
    magnitude = mant/2.0 if exp==0 else (2.0**(exp-1))*(1+mant/2.0)
    return -magnitude if sign else magnitude

forward_lut = [0.0, 0.5, 1.0, 1.25, 1.5, 2.0, 2.25, 2.75,
               0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
grad_lut    = [0.0, 1.0, 0.5, 0.5, 0.5, 0.25, 0.25, 0.25,
               0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

Listing 2: Table construction and verification (excerpted; full script builds and cross-checks all 16 codes).

At runtime, ReLog at FP4 becomes quantize-then-gather: a real input is rounded to its nearest of the 16 representable E2M1 values (a 16-way ALU comparison, no SFU call), stored as an ordinary `uint8` tensor using only its low 4 bits, then both forward and backward are a table gather by that code. We apply no per-tensor scaling recipe, matching this paper's existing FP8 methodology note: values outside E2M1's representable range ($[-6, 6]$) clamp to the nearest boundary code rather than doing anything undefined – a documented limitation, not an oversight.

```
__constant__ float d_e2m1_values[16], d_forward_lut[16], d_grad_lut[16];

__global__ void quantize_e2m1_kernel(const float* x, uint8_t* codes,
    size_t size) {
```

```
    int64_t idx= static_cast<int64_t>(blockIdx.x)*blockDim.x+threadIdx.x;
    if (idx >= (int64_t)size) return;
    float val = x[idx], best = fabsf(val - d_e2m1_values[0]);
    uint8_t code = 0;
    #pragma unroll
    for (int c = 1; c < 16; ++c) {
        float d = fabsf(val - d_e2m1_values[c]);
        if (d < best) { best = d; code = (uint8_t)c; }
    }
    codes[idx] = code;
}

__global__ void relog_e2m1_forward_kernel(const uint8_t* codes, float*
    out, size_t size) {
    int64_t idx= static_cast<int64_t>(blockIdx.x)*blockDim.x+threadIdx.x;
    if (idx < (int64_t)size) out[idx] = d_forward_lut[codes[idx]];
}

__global__ void relog_e2m1_backward_kernel(const uint8_t* codes, const
    float* grad_out, float* grad_in, size_t size) {
    int64_t idx= static_cast<int64_t>(blockIdx.x)*blockDim.x+threadIdx.x;
    if (idx < (int64_t)size) grad_in[idx] = d_grad_lut[codes[idx]] *
    grad_out[idx];
}
```

Listing 3: CUDA kernels: quantize-then-gather, ALU-only throughout.

The 16-entry tables are computed and verified once in Python and uploaded into GPU constant memory via `init_e2m1_tables()` – not re-derived in C++, so there is a single, already-verified source of truth rather than two hand-written copies that could silently drift apart. The Python autograd wrapper mirrors `ReLogFunction` above: it dispatches to the compiled extension when available and CUDA, and otherwise falls back to an identical-specification pure-PyTorch path (same tables, same quantize-then-gather logic).

We are explicit about the verification boundary here: the CPU-fallback path is executed and tested in this session against both the hand-derivable properties and exact agreement with the compiled-kernel specification it mirrors. The CUDA extension itself (the code above) has not yet been compiled or run on real hardware – this is scoped, concrete remaining work, not a claim this submission resolves, matching how Appendix A's other kernel fixes are labeled throughout this paper.

### A.6 NF4 via Lookup Table, and a Notable Equivalence to ReLU

NF4 (Dettmers et al., 2023) is architecturally different from FP4/E2M1: not an exponent/mantissa layout at all, but a 16-entry quantile codebook derived from $N(0,1)$ and normalized to $[-1,1]$, asymmetric by construction (7 negative values, one exact zero, 8 positive values). The same LUT circumvention applies for the same reason – 16 codes, precompute once, gather at runtime – using the published codebook values (Appendix E of the QLoRA paper) directly rather than re-deriving them from the paper's quantile-estimation formula, which several independent attempts (including our own) do not reproduce exactly from the formula alone.

Applying our verification procedure surfaces a specific, notable finding rather than a generic confirmation: because NF4's entire positive range lies in $(0,1]$ – precisely where ReLog is the exact identity (Section 2) – **ReLog and ReLU are numerically identical across every NF4 code except one.** Every negative code zeros identically under both; every positive code below the format's maximum reproduces its input identically under both ($f(x)=x$ for ReLog by the identity-zone property, $f(x)=x$ for ReLU by definition). The sole exception is the boundary code $x=1.0$ exactly, where both give $f(x)=1.0$ but the gradients differ (0.5 for ReLog, 1.0 for ReLU) – the single kink boundary within NF4's range. Unlike the FP4/E2M1 LUT, where ReLog's compression is substantively exercised (Appendix A.5), NF4's specific value range means the format-induced mechanism has almost nothing to do at this precision: a direct, quantitative illustration of how thoroughly ReLog's behavior is a function of the numerical format it is evaluated in, for better or worse.

## B Related Work

**The underlying approximation is 60 years old.** The linear-mantissa approximation to $\log_2$ that ReLog's bit-extraction implements is not new: Mitchell (1962) introduced exactly this piecewise-linear log/antilog scheme to convert hardware multiplication and division into addition and subtraction, with the same maximum approximation error ($\approx 0.086$, Lemma 1 in the main text) we independently re-derive in Section 2. Schraudolph (1999) later applied the same IEEE-754 bit-manipulation idea specifically to fast exponential approximation in neural network simulations, directly motivated by the same problem this paper addresses (expensive elementwise transcendentals dominating training time). Our contribution relative to this line of work is not the approximation itself, which is well established, but promoting it to a *trainable activation function whose geometry is an explicit function of the storage format*, and empirically characterizing the consequences of that choice – including a deliberate design duality between linear and sublinear regimes selectable by weight scaling alone (Section 2) – rather than treating the bit-trick purely as a fixed-function hardware optimization.

**Log-domain and hardware-motivated activation design.** Logarithmic number systems for neural network computation have been explored directly (Miyashita et al., 2016), quantizing weights and activations to log-domain representations for efficient inference. ReLog differs in applying the bit-level construction to the activation function itself, used during training rather than as a post-training quantization scheme, and in deriving its geometry from the training-time storage format rather than a fixed quantization target.

**Hardware-cheap activation functions.** A parallel line of work designs activations explicitly to avoid expensive transcendental evaluation: hard-swish (Howard et al., 2019) replaces Swish's sigmoid with a piecewise-linear ReLU6-based approximation specifically for mobile hardware efficiency, and squareplus (Barron, 2021) replaces GELU/softplus's transcendentals with a cheap algebraic form. These share this paper's motivating constraint (avoid SFU-bound operations) but construct their approximations directly in the mathematical domain; ReLog's distinguishing feature is deriving its shape from the bit layout itself, which is what makes precision a parameter of the function rather than only a deployment detail.

**Piecewise-linear network expressivity.** Prior theoretical work analyzes how the number of linear regions in a piecewise-linear network scales with depth and width (Montúfar et al., 2014; Serra et al., 2018; Hanin & Rolnick, 2019). This literature is directly relevant to our own kink-count claim and our design-level framing of dynamic linearity: Section 2 states the kink count as bounded by $2^{N_E-1}$ (the count of admissible non-negative unbiased exponent values for a format with exponent width $N_E$) and reports a direct empirical measurement (99.7% of activations concentrated in two exponent bands on a trained checkpoint) consistent with this literature's framing that expressivity and effective linearity claims must be tied to concrete, countable, and measured quantities rather than asserted qualitatively.

**Low-precision training formats.** Our FP8 methodology (Section 4) follows the native tensor dtypes and lack of explicit per-tensor scaling recipes discussed generally in Micikevicius et al. (2022); we note explicitly where our own protocol departs from a production scaling recipe as a scope limitation, not a hidden methodological choice.

## C Planned Follow-Up Experiments

Several claims in this paper are currently supported by measurement in some respects and by argument alone in others. We list the specific follow-up experiments needed to close each gap, in priority order, rather than leave the gap implicit.

**F1. Forward-compression vs. backward step-derivative ablation.** ReLog's forward pass paired with a detached/constant-slope surrogate backward, isolating the forward log-compression mechanism of Proposition 1 from the backward step-derivative – both variants remain ALU-native, unlike a true analytic $\log_2$ comparison, which we do not use as it requires an SFU call and is disqualified from this paper's design space by construction (Section 2). Run on the E2 spectral-radius grid, this decomposes which half of the mechanism is doing the work.

**F2. Classical stabilizer baselines.** Add tanh (bounded, likely to trivially achieve low divergence for a different reason than ReLog), LayerNorm-in-recurrence, and spectral normalization of $W$ to the E2 comparison table.

**F3. Clip-threshold sweep.** Extend the single `clip@1.0` setting to $\{0.1, 0.25, 0.5, 1.0, 5.0\}$ per $\rho$, report best-of-sweep, and apply clipping to GELU/SiLU as well as ReLU.

**F4. Initialization-scale robustness.** Test ReLog under a range of initialization gains rather than only the He/Xavier defaults calibrated for ReLU-family statistics.

**F5. Occupancy at full sweep breadth.** The exponent-occupancy measurement (Weierstrass, Section 2) uses a single representative checkpoint; extending it across the full seed/precision grid would confirm whether the reported pattern is representative or specific to the checkpoint sampled.

**F6. Kernel fusion and roofline placement.** Report achieved GB/s against peak bandwidth for every throughput measurement in Section 5.3; benchmark inside a fused context (e.g. `torch.compile` epilogue fusion) to test whether ReLog's measured slowdown is specific to the unfused, standalone case.

**F7. BF16/FP16 native kernels.** Extend the compiled fast path beyond FP32 (trivial generalizations of the existing kernel changing the shift/mask/bias constants to each format's own layout), or explicitly restrict every throughput claim in this paper to FP32 until this exists.

**F8. Fréchet Inception Distance (FID) for the DiT experiments.** Our current DiT metric (noise-prediction MSE) already produces a decisively significant, well-powered result (Section 5); FID would additionally validate that this training-loss advantage translates to generated-sample quality, using a pretrained Inception network, a fixed sampling protocol, and reference-statistics computation against the same Imagenette validation set. We flag this as new evaluation infrastructure this codebase does not yet have, not an extension of an existing measurement, and report it at whatever fidelity is achievable before the camera-ready deadline.

**F9. A lookup-table circumvention of the FP4/NF4 structural floor – built and logic-verified; not yet hardware-tested (Appendices A.5–A.6).** Section 4 reports FP4 and NF4 as structurally infeasible because neither has a native PyTorch tensor dtype to host the shift/mask bit-extraction our other kernels use. Both have exactly 16 representable codes, so we precompute the entire forward and backward mapping once, offline, as two 16-entry lookup tables per format, indexed by the raw 4-bit code and stored as an ordinary `uint8` tensor rather than any dedicated 4-bit dtype – a LUT gather is itself ALU-native, consistent with this paper's hardware-efficiency constraint. We built and verified both: each format's tables independently match hand-derivable properties and the general bitwise-extraction formula exactly on all 16 codes; a CPU-executable reference implementation with an identical mathematical specification passes end-to-end forward/backward tests; and for FP4/E2M1, the real CUDA kernels, C++ bindings, and Python autograd wrapper are written (Appendix A.5). The NF4 extension surfaces a specific, non-obvious finding rather than a generic confirmation: because NF4's entire positive range lies within ReLog's identity zone, **ReLog and ReLU are numerically identical across every NF4 code except one** (Appendix A.6) – illustrating concretely how thoroughly ReLog's behavior is a function of the numerical format, since the same mechanism that gives ReLog its advantage at wider formats does essentially nothing at this specific one. What remains for both formats: compiling and running the CUDA extension on real hardware, and an actual training sweep testing whether the same monotonic instability dose-response observed through FP8 (Section 4) continues, accelerates, or breaks down at 4 bits – we have not yet done either, and report the boundary precisely rather than imply more completeness than this.