BITWISE-IDENTICAL MACHINE LEARNING ACROSS GPUS AND CPUS

Andrew Hendel [1]

ABSTRACT

The same machine-learning workload can produce different bits on different processors, changing predictions, learned models and subsequent training updates. Mojolearn is a machine-learning library whose default configuration produces bitwise-identical results across verified Apple, NVIDIA and AMD GPUs, and Arm and x86-64 CPUs. Results agree bit for bit, not merely within a numerical tolerance. It supports training and inference for over 100 machine-learning algorithms in 13 families, from decision trees to neural networks. Users can also run and train custom models built from supported operations under the same cross-vendor identity guarantee. Neural models trained in other frameworks can be served with identical outputs across vendors, and training workloads can be handed off between GPU vendors mid-run, or shared by GPUs of different vendors at once, while maintaining bitwise identity. Using Mojolearn, I trained two instances of a 162-million-parameter GPT-3 Small-shaped model along different hardware routes collectively spanning Apple, NVIDIA and AMD GPUs, and maintained bitwise identity at all 5,000 optimizer steps. Neural training and inference run in full FP32 floating-point arithmetic, without reducing the computation to integers. To my knowledge, I am the first to enable these capabilities.

1 INTRODUCTION

A model may serve predictions on NVIDIA GPUs, run locally on an Apple laptop, and later be evaluated on an AMD server. Even with identical weights and inputs, different GPU implementations can produce different output bits. For deployments that require exact replay, hardware becomes another variable in serving, regression testing and auditing.

Mojolearn is a machine-learning library whose default configuration produces bitwise-identical outputs across verified devices and extends that guarantee to supported training computations. It covers over 100 algorithms in 13 families, from decision trees to neural networks, with Apple M2, M3 and M4, NVIDIA H100, A100, RTX 4090, L40S and RTX A6000, and AMD MI325X and MI300X GPUs and x86-64 and Arm CPUs in its verification scope. Unsupported dtypes, shapes or options raise an error rather than silently weakening the numerical guarantee. Section 3 states the conditions and limits of that guarantee.

1.1 Portable inference across vendors

A model does not need to be trained in Mojolearn to be served by it. Supported PyTorch exports, ONNX graphs and Hugging Face checkpoints can be imported for cross-vendor bitwise-identical inference in Mojolearn. Users can bring supported models and obtain cross-vendor identity without implementing the numerical rules themselves. A team can test a deployed model on different hardware, replay a recorded decision during an audit, and replace serving hardware without numerical change. The guarantee applies to execution in Mojolearn; it does not promise bitwise agreement with the original framework.

1.2 Training: the same model on every vendor

Training in Mojolearn is also bitwise identical across vendors. Given the same code revision, data, hyperparameters and seed, Apple, NVIDIA and AMD GPUs produce the same model state, gradients, optimizer state and checkpoint bytes. A checkpoint written on one vendor can therefore resume on another and follow the same trajectory as uninterrupted training. To maintain cross-vendor bitwise identity, inference must still use Mojolearn. If weights trained by Mojolearn are served by another framework, that framework's arithmetic applies and outputs can again differ from GPU to GPU.

1.3 Portability and the numerical contract

Mojo and MAX provide compilation and execution across Metal, CUDA and HIP (Modular, 2026b; Lattner et al., 2021); they do not make the arithmetic agree across those backends. Mojolearn defines the shared numerical rules and

[1]Latent Archon LLC, United States; ORCID 0009-0000-9877-3623. Correspondence to: Andrew Hendel <ajhendel@latentarchon.com>.

Preprint.

implements the algorithms that obey them. Its own numerical kernels control arithmetic and reduction order, replacing vendor math routines where their results would depend on the backend. Mojolearn optimizes scheduling and reorders arithmetic only where the numerical contract permits, preserving bitwise identity across vendors. Section 5 explains how the contract is enforced.

Bitwise identity matters throughout the library, beyond neural networks. In a tree learner, rounding can change the winning split and every subsequent node; in k-means, it can move a point to another cluster and change every later centroid. These are applications of one numerical contract across many algorithms.

1.4 Contributions

(a) *Cross-vendor bitwise-identical machine-learning algorithms*: a single Mojo codebase supports training and inference for over 100 algorithms in 13 families, from decision trees to neural networks.

(b) *Inheritable identity across GPU and CPU vendors*: users can run and train their own models built from supported operations, or import supported PyTorch, ONNX and Hugging Face models for bitwise-identical inference across vendors. The same saved model produces identical output bits across verified devices.

(c) *Cross-vendor identical handoff and mixed-vendor training*: a training run can move from one GPU vendor to another mid-run, or be shared by Apple, NVIDIA and AMD GPUs working on one model at once. Either way, the training state after every step is bitwise identical to a single GPU's (Sections 7.2 and 7.4).

(d) *A numerical contract that separates arithmetic from scheduling*: the contract fixes the operations that determine model bits, including reduction order, rounding and elementary functions. Tiling, thread layout, memory placement and kernel scheduling remain free, so each backend can be tuned for its own hardware without changing a bit (Section 5). This vendor-specific tuning has only begun.

To my knowledge, these are the first cross-vendor bitwise-identical GPU training and inference implementations. I achieve this in full FP32 floating-point arithmetic for neural networks, without reducing the computation to integers. Prior and concurrent bitwise ML results include single-vendor execution, emulation of one vendor's arithmetic, and cross-vendor forward-pass or step-replay demonstrations without AMD GPUs (Section 8).

Building a cross-vendor numerical profile required an extensive implementation effort. Mojo 1.0 was released on August 11, 2026; development of Mojolearn began on August 19, 2026, and the original preprint was submitted to arXiv on September 8, 2026. Release 0.7.0, archived on September 9, 2026 (Hendel, 2026c), documents three-vendor training identity for a byte-level language model. In the release 0.8.55 source snapshot (October 20, 2026), Mojolearn comprises 792,375 lines of original Mojo code: 476,325 lines of library implementation and 316,050 lines of verification fixtures, test harnesses and benchmarks, with a further 420,942 lines of Python, shell and other code (Appendix B). Its public repository contains 17,382 commits through that snapshot, including merges. I built the system with extensive assistance from AI coding tools, as described in the AI Usage Disclosure. This scale illustrates the implementation effort required to cover the library's algorithm families.

2 The System

Mojolearn is designed to ease migration for machine-learning researchers and developers. Its Python API mirrors scikit-learn's module layout and estimator conventions (Pedregosa et al., 2011), allowing users to replace supported estimator imports while retaining familiar *fit*/*predict* calls, pipelines, grid search and cross-validation. Migration requires checking supported input types, parameter settings and the numerical configuration.

One Mojo source compiles for Metal, CUDA and HIP and ships as an Apple silicon wheel and a Linux x86-64 wheel. In the default identical configuration, the library's portable elementary functions run in its own compiled Mojo kernels (Modular, 2026b), without calling MAX's matrix-multiplication kernels, vendor BLAS and solver libraries, or external ML frameworks.

2.1 The user-facing contract

One flag, *numeric_mode="identical"*, selects the default numerical contract on every supported interface. For example, a caller passes weights to a transformer block and runs its forward pass. A fresh process using the same input and weight bytes, hyperparameters, seed and numerical profile returns the same bits on a verified device.

2.2 Algorithms and interfaces

Table 1 summarizes the neural, tree and classical interfaces.

The implementations build on published designs: CatBoost's oblivious, depth-wise and loss-guided trees (Prokhorenkova et al., 2018; CatBoost Developers, 2026b); XGBoost and LightGBM designs for tree boosting and feature sampling (Chen & Guestrin, 2016; Ke et al., 2017), including fixed-point gradient histograms (Shi et al., 2022; XGBoost Developers, 2026b); and cuML, RAFT and cuVS designs for forests and classical methods (RAPIDS Development Team, 2026). The contribution is the numerical contract they share and the kernels that enforce it, not new learning algorithms.

Figure 1 shows one computation that repeats on each device

family	algorithms and components
boosting	symmetric-tree, depth-wise, loss-guided and ordered boosting; AdaBoost, DART
trees and ensembles	decision trees, random forests (Breiman, 2001), extremely randomized trees (Geurts et al., 2006), isolation forests, bagging, voting, stacking
clustering	k-means, mini-batch and bisecting k-means, DBSCAN, HDBSCAN, agglomerative, spectral, mean shift, OPTICS, affinity propagation, Gaussian and Bayesian Gaussian mixtures
neighbors, density and search	nearest neighbors, k-NN, radius neighbors, random ball cover, kernel density, local outlier factor, label propagation, IVF-Flat, IVF-PQ, CAGRA
linear models	least squares, ridge, lasso, elastic net, logistic regression, SGD, Huber, Poisson, Gamma, Tweedie, Bayesian ridge, ARD, Lars, quantile and isotonic regression
kernel methods and Gaussian processes	SVMs, kernel ridge, Gaussian process regression and classification, sparse variational GP, Nyström, random Fourier and chi-square features
decomposition and manifold	PCA, incremental and kernel PCA, truncated SVD, NMF, ICA, factor analysis, PLS, CCA, dictionary learning, sparse PCA, latent Dirichlet allocation, UMAP, t-SNE, Isomap, MDS, LLE
time series	ARIMA, AutoARIMA, Holt-Winters, ETS, STL, VAR, Theta, Croston, GARCH, Prophet-style forecasting, KPSS
preprocessing, probabilistic models and resampling	scalers, encoders, imputers, discretization, feature selection, naive Bayes, discriminant analysis, bootstrap, permutation tests, Monte Carlo integration
neural blocks and optimizers	transformers, Mamba-1/2/3, Samba, MLPs, RNNs, LSTMs, GRUs, mixture of experts, Adam, AdamW, SGD, RMSprop, Adagrad, Lion, Adafactor, LAMB
convolutional and graph networks	1-D and 2-D convolutions, pooling, batch normalization, residual blocks, CNN classifiers, GCN and GraphSAGE convolutions
language models and tokenization	decoder language models, byte-level BPE tokenization
linear algebra	matrix products, Cholesky, QR, LU, eigendecomposition, SVD, least squares
also provided, not counted	evaluation metrics; cross-validation; CPU inference for saved models; stepwise decoding with explicit state; checkpoint-loaded causal language models; BF16 and INT8 weight storage; tokenized corpora; multi-GPU execution plans; the verifier

Table 1. Over 100 algorithms across 13 families. The table lists representative methods and model components from the current public API, grouping related estimator variants. A classifier and a regressor of the same method count once; losses, kernels, distance metrics, penalties, starting rules and multi-GPU execution are configurations, not separate algorithms. The fixed verification record and its neural configurations are reported separately in Section 7 and Table 5.

yet differs across vendors.

3 Scope and Limitations

The guarantees below require the same code revision, numerical profile, input bytes, initial state, hyperparameters and seed, using the default numerical contract and a configuration covered by the verification record.

3.1 Scope of inference

Every verified device returns the same output bits for the supplied model weights, whether those weights were trained in Mojolearn or elsewhere. Inference is also batch invariant (He & Thinking Machines Lab, 2025): a sequence's output bits do not depend on how many other sequences share the batch.

3.2 Scope of training

Every verified device produces the same learned model and supported training state, including gradients, optimizer state and checkpoint bytes. A refused configuration never counts as a pass.

3.3 Limitations

These limits apply throughout; the rest of the paper does not repeat them.

- *Devices.* The record and additional checks span Apple M2 Pro, M3 Ultra and M4 GPUs and CPUs; NVIDIA H100, A100, RTX 4090, L40S, RTX A6000 and RTX PRO 6000 Blackwell GPUs; AMD MI325X and MI300X GPUs; and AMD EPYC CPUs. Appendix A identifies which devices contributed to the fixed verification record. Mojo supports GPU programming on Apple M-series GPUs, supported NVIDIA GPUs via PTX and select AMD GPUs (Modular, 2026a). A GPU is not covered until it passes the same checks (Table 3).
- *Precision.* The guarantee is for FP32 arithmetic. FP64 input is converted or refused.
- *What was tested.* The record covers the configurations, inputs, model shapes and release it names. A call that succeeds on an untested configuration is not a verified result.
- *Bitwise-identical inference.* Inference must use the

family	what can vary	effect on the model	implementation response
neural building blocks	matrix products, reductions and state updates	different activations, gradients or training state	portable primitives and explicit rounding barriers
trees and forests	histogram order, split-score rounding and ties	different split, partition or tree structure	fixed accumulation, frozen geometry and total tie rules
clustering and neighbors	distance reductions, square root, traversal and output order	different neighbors, memberships, clusters or labels	portable distance arithmetic and canonical ordering
linear and decomposition	matrix-product order, solver folds and closed vendor libraries	different coefficients, margins or components	portable matrix operations and fixed fold order
time series	recurrences, elementary functions and pivot ties	different likelihoods, parameters or forecasts	pinned functions, pivots and decision rules

Table 2. Algorithmic breadth exercises distinct numerical failure modes. The claim is one contract across these mechanisms, not novelty in the learning algorithms themselves.

numerical contract above. Execution in another framework is not covered.

- *Option coverage.* Some interfaces accept more options than the record covers (Appendix A.3).
- *Input boundary.* The guarantee begins at the estimator input; preprocessing performed outside the library is not covered.
- *Quality.* Identical bytes can encode a poor model, so quality validation remains a separate obligation. The measurements in Appendix C show comparable quality on several workloads and report both implementations' quality alongside runtime.
- *Speed and cost.* Performance is evaluated in Section 5.8. Mojolearn has been in development only since August 2026 and has not received the years of tuning that established libraries have, so its current speed reflects the maturity of the implementation as well as the cost of identity.

Table 3 also lists prospective Qualcomm Adreno and Intel Xe hardware against the contract's requirements. Each new backend must pass verification; hardware that violates a pinned assumption is refused. Changes to the numerical rules require a new profile version.

4 Why GPUs Produce Different Models

GPU backends can differ in intermediate precision, reduction grouping and elementary functions even with identical models, inputs and seeds. Neural networks propagate these differences through outputs and training state; tree learners can turn them into structural changes.

4.1 Neural inference

Transformer matrix products depend on product precision and summation order; attention and normalization add division, exponentials and square roots. Every operation must obey shared numerical rules. An activation difference can reach the logits, change the ordering of nearly tied tokens, and alter even greedy decoding and its subsequent inputs.

Mamba blocks also carry recurrent state between decoding calls. Matching weights and current inputs cannot restore agreement if that state already differs, so identity must cover both outputs and retained state.

4.2 Neural training

Backward matrix products, gradient reductions, clipping and optimizer updates can change parameters and AdamW moments that feed the next step. A common numerical profile must therefore cover forward computation, loss, backward computation and updates. Checkpoint continuation also needs the same parameters, moments, counters and subsequent inputs. These dependencies motivate comparing intermediate outputs and training state alongside final checkpoint bytes (Section 7.2). Even a difference that leaves model quality unchanged breaks exact replay.

4.3 Tree training

CatBoost's GPU histogram implementation uses floating-point atomics, so thread arrival determines summation order. Its partition-statistics kernels derive chunk counts from the number of streaming multiprocessors, making device geometry a summation choice (CatBoost Developers, 2026a). Either can change split scores and, near a tie, select another partition and alter subsequent trees. Border selection, leaf updates and tie-breaking must obey the same numerical rules as histogram accumulation.

Table 2 identifies the corresponding hazards in the other algorithm families. Section 5 explains how the shared numerical contract controls them.

	verified, measured			buildable, unvalidated	prospective, not buildable	
	Apple M4	NVIDIA H100	AMD MI325X (CDNA)	AMD RDNA	Qualcomm Adreno	Intel Xe
square root	correctly rounded	**approximate**	correctly rounded	—	—	—
denormals	flushed	kept	kept	—	—	—
$a \cdot b + c$	fused	fused	fused	—	—	—
$\max(+0, -0)$	-0.0	$+0.0$	$+0.0$	—	—	—
NaN payload	0x7fc00000	0x7fffffff	0xffc00000	—	—	—
lockstep lane width	32	32	**64**	32	8, compiler-chosen	8, compiler-chosen
vendor FP32 matrix product	FP32	**TF32 by default**	FP32	FP32	FP32	FP32
float atomics	global only	yes	yes	yes	**no**	yes
threadgroup memory per block	32 KB	48 KB	64 KB	64 KB	32 KB	64 KB
meets minimum hardware requirements	yes	yes	yes	yes	yes	yes

Table 3. GPU arithmetic behavior and backend requirements. Adreno and Xe specifications are from vendor documentation. Minimum hardware requirements are 32 KB of threadgroup memory, threadgroup integer atomics and 1,024-thread blocks.

5 Enforcing Bitwise Identity

5.1 Numerical rules and hardware differences

Mojolearn enforces bitwise identity by *pinning* settings the hardware would otherwise choose, such as the number of blocks a sum is split into, and *replacing* functions that GPUs compute differently, such as square root, with its own portable implementations. It refuses configurations it cannot make identical, such as an unsupported dtype, shape or option, with an explanatory error. Operations that do not change arithmetic order or precision remain free to use each backend's normal scheduling. Table 4 summarizes the numerical issues addressed by pinning and replacement.

Floating-point contraction must be controlled per expression: the Mamba-3 state-rotation fixture requires explicit intermediate rounding barriers, not simply a global policy enabling FMA.

Users can build their own algorithms with Mojolearn's operations and tape, inheriting cross-vendor bitwise identity when the whole computation uses supported Mojolearn operations. Gradient kernels follow the numerical contract, and contributions to shared gradients are accumulated in a fixed order. Captured inference programs preserve the operation sequence, arguments, tensor shapes and weights. Reloading executes that computation through the destination backend's implementations of the same numerical rules. Within the same library version and numerical profile, the same input bytes therefore produce identical output bits across verified devices.

5.2 Machine parameters can change the result

A block count, a replication factor and a reduction width all determine which partial sums are combined. They are therefore numerical choices, even when an implementation labels them as scheduling or tuning parameters. This is established in the reproducible-summation literature (Demmel & Nguyen, 2013; Demmel et al., 2016; Collange et al., 2015), appears as a product constraint in vendor math libraries (Lubin & Mueller-Albrecht, 2024), and is restated for inference as batch invariance (He & Thinking Machines Lab, 2025). The implementation pins every machine-derived parameter that reaches floating-point arithmetic. Host-side arithmetic that selects device geometry must also obey the same numerical profile; device arithmetic tests alone do not establish this.

5.3 Floating-point and integer matrix units differ

Neural blocks expand BF16 and INT8 weight storage to FP32 for computation. The FP32 profile avoids vendor floating-point matrix units: their fragment shapes and internal accumulation do not implement the library's specified rounding order, and TF32 additionally rounds operands before multiplication. RepDL identifies hardware-specific low-precision matrix arithmetic as a similar obstacle (Xie et al., 2025). My measured FP32 workloads therefore retain the cost of the general-purpose arithmetic path. The separate INT8 arithmetic profile uses integer accumulation and admits integer matrix units because its bounded Int32 products and sums are exact before the pinned conversion to floating point. Changing the grouping of an overflow-free integer sum cannot change its value. The FP32 workloads do not use those units.

5.4 Ordinary-looking choices change bits

DBSCAN normally sizes its batches from free device memory, so unrelated memory use could change the graph decomposition; my implementation pins the batch size. A GPU's default square root may be an approximate instruction, so Mojolearn uses its own implementation on every backend. Table 3 summarizes the measured arithmetic differences that portable replacements must cover.

5.5 Neural matrix products

The FP32 matrix-product profile used by transformer projections separates the logical arithmetic from its physical execution. Each output element is a reduction over K products. The profile partitions that reduction into contiguous leaves whose length depends on K alone: K itself up to 128, then 128 until the leaf count would exceed 1,024, then

Numerical issue	Action
order-dependent accumulation	REPLACE
a machine query reaching a float reduction	PIN
vendor-divergent elementary functions	REPLACE
code generation	PIN
subnormals, signed zero and NaN payloads	PIN
ties resolved by arrival order	PIN
closed vendor libraries	REPLACE
host-side arithmetic that selects a device configuration	PIN
iteration counts and caps	PIN
random draws	PIN
unsynchronized buffer reuse	REPLACE
vendor-dependent output order	REPLACE
fixed-order floating-point folds	REPLACE
sign and extrema conventions	PIN

Table 4. Mechanisms used to enforce bitwise identity across backends.

$\lceil K/1024 \rceil$. Neither batch size nor GPU occupancy can change these constants. Within a leaf, the products are accumulated in ascending order from positive zero by a fused multiply-add that rounds once to FP32, and the accumulator is flushed to zero of the same sign after every step. Flushing only the final partial would allow a backend that retains subnormals to carry a different intermediate into the next update. The leaf partials are then combined in a fixed binary tree over adjacent pairs; an unpaired final partial is carried unchanged to the next level rather than padded with zero, and an empty reduction returns positive zero. GPU threads may compute leaves or output tiles concurrently, and intermediate values may live in registers, shared memory or a separate buffer, but each output must follow this same dependency tree. A GPU whose hardware runs 64 threads in lockstep, where the others run 32, cannot substitute its native reduction for the prescribed pairing. Thus changing tile size or batching changes the schedule without changing the arithmetic; changing a leaf boundary, pairing or rounding step requires a new numerical profile. The repository's contract document for this profile states each rule together with the fixture that checks it.

5.6 Tree histograms and splits

The fixed-point histogram path must first turn each floating-point statistic into the same integer on every device. It scales the gradients and Hessians by a power of two chosen from the row count and from fixed-order sums of their magnitudes, so that every subset sum, with an allowance of one quantization unit per row, fits in a signed Int32. Bounding magnitudes rather than the signed total also bounds intermediate sums when positive and negative gradients cancel. Each scaled statistic is quantized with deterministic dither: a hash of the row key, independent of thread arrival order, decides whether the fractional part rounds up, which reduces the systematic bias that truncating equal statistics would introduce. Integer atomics then accumulate the bounded values in any arrival order, because an overflow-free integer sum is exact, and the total is converted back to floating point once, at the histogram boundary; converting partials early and summing them as floats would forfeit that order independence.

Identical histograms are necessary but insufficient for identical trees. Partition totals and split-score expressions retain their pinned folds and rounding rules, and the final comparison orders candidates by gain, then feature index, then bin index, in a fixed traversal order; a full tie keeps the earlier candidate. The discrete split that determines the next partition is therefore the same on every device. Fixed-point quantization changes the numerical computation; its resolution and the resulting model quality remain separate from the identity guarantee. Together, the examples illustrate the scheduling rule: prescribe the order where rounding matters, and permit order changes where bounded integer arithmetic is exact.

5.7 Training handoff

The supported language-model trainer checkpoints parameters, optimizer moments, initialization flags, the completed-step counter and optimizer settings as raw state. Companion metadata binds that checkpoint to the numerical profile, code revision, corpus and token schedule, including the position at which training resumes. The receiving device restores the same state and consumes the same next batch with the same logical microbatch partition. Forward computation, loss, backward computation and the optimizer update follow the shared numerical rules, so the next state matches the uninterrupted run. Repeating this argument gives the same subsequent trajectory for supported computations. Transferring weights alone would not suffice: different moments or a different step counter can change the next update even when the forward outputs agree.

5.8 What Identity Costs

Cross-vendor bitwise identity carries implementation and execution costs: my implementation gives up vendor matrix units, approximate elementary functions and device-chosen reduction geometry. These constraints require additional implementation work and can increase execution time. Appendix C reports runtime and quality comparisons against GPU libraries on NVIDIA L40S and AMD MI300X, using the published release 0.8.55 wheels and paired opponent runs. The measurement appendix and full record report neural, classical and tree workloads, including workload settings, measured quality and timing boundaries. These

#	stage	type	n	Apple M4	NVIDIA H100	AMD MI325X
0	knn.index_norm	f32	20000	74a472e4925ec44e	74a472e4925ec44e	74a472e4925ec44e
1	knn.query_norm	f32	2000	18da124518cd0e52	18da124518cd0e52	18da124518cd0e52
2	knn.out_dist	f32	20000	18499d126dfd6ffe	**563af8012606fe61**	18499d126dfd6ffe ◀
3	knn.out_idx	u32	20000	379df04034c318b9	379df04034c318b9	379df04034c318b9
4	knn.sorted_dist	f32	20000	18499d126dfd6ffe	**563af8012606fe61**	18499d126dfd6ffe ◀
5	knn.sorted_idx	u32	20000	379df04034c318b9	379df04034c318b9	379df04034c318b9

Figure 1. Illustrative debugging example: optional trace hashing identifies the first divergent intermediate in an unpinned k-NN computation. The standard-library path first differs on NVIDIA at the distance buffer (◀), and that difference survives in its sorted copy; the preceding norms and resulting neighbor indices agree. This is CUDA's approximate PTX square root. Mojolearn uses its pinned square root and produces matching bits without requiring trace hashing during normal training.

comparisons reflect both implementation differences and the cost of enforcing bitwise identity.

6 Reproducibility and Verification

6.1 Verification checks

The installed verifier provides three complementary checks:

- *Against recorded references.* The verifier trains each covered algorithm on fixed fixtures and compares training-state, held-out-prediction and saved-model hashes with Apple, NVIDIA, AMD and CPU references. Matching input hashes are required before numerical comparison; an agreeing machine joins the recorded columns.
- *GPU against CPU on the user's own machine.* Each model is fitted on the user's GPU, then held-out predictions are compared with those from the saved model reloaded on the CPU. This checks inference without trusting a recorded reference and runs on any supported GPU, including a laptop's. It does not independently repeat training on the CPU.
- *Two independent parties.* Two people run the verifier on their own machines and compare the resulting evidence documents directly, without consulting the library's reference table. The comparison rejects mismatches, self-contradictions, uncomputed cells and a document compared with itself.

The checks share the library's implementation and establish cross-device consistency; algorithmic correctness and model quality require independent references and quality criteria. Verdicts distinguish agreement, divergence, missing references, refusal and non-applicability; only agreement counts as verified. A self-test accepts an unchanged fixture and detects an input differing in its last bit.

Where applicable, saved models must reproduce in-memory predictions. Rows and per-row gradients must remain batch invariant, and padded or ragged sequences must match isolated execution. Token-by-token decoding must equal a full forward pass; gradient accumulation must agree for splits preserving the reduction tree; and sampler log-probabilities must match trainer recomputation. Transformer and Mamba checks span batch sizes 1–1,024 and sequence lengths on either side of every kernel's chunk boundary. Numerical parts unsupported by an interface are explicitly non-applicable. Training state is compared at every recorded stage because different models can agree on predictions.

6.2 Optional trace hashing

Verification and debugging builds can record a *trace card*: hashes of the raw buffer bits at each named stage, so that comparing two cards locates the first recorded stage that differs. Figure 1 uses one to isolate CUDA's approximate square root in an unpinned k-NN computation; normal training omits this instrumentation.

7 Results: The Same Model Across Three Vendors

7.1 Published-release cross-device verification

I evaluated the published release 0.8.55 wheels on Apple M4, NVIDIA L40S, NVIDIA RTX A6000 and AMD MI300X GPUs, and on Apple M4 Arm and AMD EPYC x86-64 CPUs. Across the GPU targets, all 15,362 routine and 2,349 neural-training comparison parts agree bitwise, with zero differences. The Arm and x86 CPUs also agree with the GPUs and each other on their shared results. The NVIDIA tests exercise native CUDA on the L40S and the release's PTX route on the RTX A6000, without establishing coverage of every NVIDIA architecture.

Appendix A.1 details the comparisons and CPU coverage, reporting direct cross-device agreement separately from shipped-reference conformance. Negative controls test detection of deliberate arithmetic changes (Appendix A.2). The older two-GPU experiments in Section 7.4 are separate from these release totals.

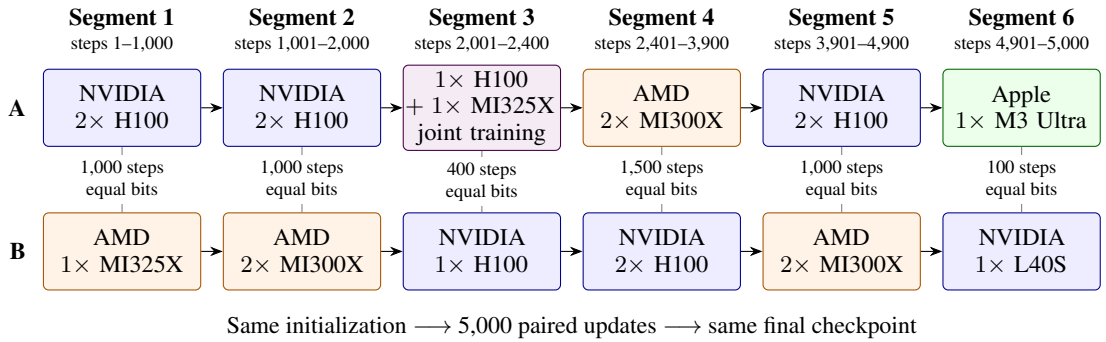


Figure 2. Two hardware routes through the same GPT-3 Small-shaped training workload: 162,147,840 parameters and 2,621,440,000 tokens per route. Boxes name the GPU model and number of participating devices; arrows denote checkpoint continuation, and vertical comparisons cover every step in each segment. A/3 trains jointly on an H100 and an MI325X in separate machines. All H100s are the 80 GB HBM3 model. The two routes agree on state and summed gradient digests, all 64 per-shard loss bit patterns, and learning-rate bits at all 5,000 steps; their final checkpoint file digests also match. Segment widths are schematic, not proportional to steps or elapsed time.

7.2 A GPT-3 Small-shaped model across six segments

I trained two instances of the same 162-million-parameter decoder for 5,000 optimizer steps each on the same sequence of 2.6 billion FineWeb-Edu tokens, following different hardware routes spanning NVIDIA, AMD and Apple GPUs (Figure 2). Both instances started from identical weights, computed their own updates and continued from their own checkpoints. Route B compared its results with route A's recorded values; those records supplied comparison values, not gradients or rounding corrections.

All 5,000 paired steps agree on recorded training-state and summed-gradient digests, per-shard loss bits and learning-rate bits, with zero divergent steps. The final checkpoint file digests also match. One segment additionally demonstrates joint NVIDIA–AMD training with matching worker state throughout. This scale experiment is separate from the release 0.8.55 fixture totals; Appendix A.5 provides the step records, checkpoint comparisons, handoff checks and negative controls.

7.3 Custom models and portable inference

Separate tests demonstrate matching training and prediction digests for custom models across the three GPU vendors, with additional Arm CPU comparisons. Six pretrained models imported through PyTorch or ONNX produce matching output digests across the three GPU vendors on fixed inputs. A program saved on NVIDIA also reloads on Apple with the same output digest. Appendix A.4 records the tested models, operations and provenance separately from the release campaign.

7.4 Two-GPU execution and mixed-vendor training

Changing the device count can change floating-point results by changing how sums are grouped. Mojolearn assigns complete units of work, such as trees or microbatches, to devices and combines their results in a fixed order.

Physical two-GPU experiments on NVIDIA and AMD hardware demonstrate bitwise agreement with single-GPU execution for the tested configurations (Appendix A.3). Two AMD MI300X GPUs also trained three segments of the scale experiment (Figure 2; Appendix A.5). In a separate experimental demonstration, an Apple M4, an NVIDIA H100 and an AMD MI300X jointly trained a small decoder for six steps, reporting identical state hashes across the three workers after every step. Offline replay checks additionally tested fixed-order combination of recorded microbatch gradients (Appendix A.5). These experiments establish identity for the tested workloads; multi-GPU throughput and execution on more than two GPUs within one machine were not measured.

8 Related Work

8.1 Reproducible arithmetic

Device geometry and partitioning determine summation order, as established (Demmel & Nguyen, 2013; Demmel et al., 2016; Lubin & Mueller-Albrecht, 2024; He & Thinking Machines Lab, 2025), and integer gradient histograms predate this work (Shi et al., 2022). ReproBLAS and Intel MKL's conditional reproducibility mode provide reproducible summation and linear algebra (Demmel et al., 2016; Intel, 2024). Climate models demonstrate identity across compiler versions and flags (Li et al., 2016); molecular-dynamics codes document processor-count sensitivity (OpenMD, 2013). My arithmetic probes confirm documented differences in denormals, FMA contraction and elementary functions (Dawson, 2013; Whitehead & Fit-Florea, 2011).

8.2 Deterministic modes

Framework deterministic modes switch known-nondeterministic operations to deterministic ones or refuse them (PyTorch, 2026; TensorFlow, 2024). NVIDIA's primitive library grades reductions as nondeterministic, run-to-run deterministic or GPU-to-GPU deterministic (Al Awar & Singanaboina, 2026). Environment capture and replay also support repeatable training (Chen et al., 2022; Heumos et al., 2023). Within-platform repeatability does not establish cross-vendor identity (Figure 1).

8.3 Bitwise training

RepDL (Xie et al., 2025) pursues cross-system reproducibility with correctly rounded FP32 operations, fixed summation order and deliberate FMA use. It provides CPU and CUDA implementations and a reproducible training example; the report has no evaluation section and names no tested hardware models. Optimistic Verifiable Training (Srivastava et al., 2024) reproduces training across three NVIDIA GPUs by recording rounding decisions for an auditor to replay. Verde's RepOps (Arun et al., 2025) fixes CUDA operation order across the NVIDIA configurations it tests; Hawkeye (Badash et al., 2026) emulates NVIDIA Tensor Core arithmetic on CPUs.

Concurrently, OPEN-1B (Donaghy et al., 2026) trains a 1.6-billion-parameter model on 48 NVIDIA H100 GPUs. Auditors replay individual steps bit for bit on a CPU, NVIDIA GPU or Apple GPU against published per-step hashes, with many auditors intended to cover the run collectively. Its large matrix products use INT8 with Int32 accumulation; it also controls floating-point reductions, fusion, subnormals and optimizer arithmetic. It covers one architecture and no AMD GPUs, with training on NVIDIA and replay on other devices. Mojolearn keeps neural matrix products in FP32 and compares complete independent training runs on each vendor.

8.4 Bitwise inference

EigenAI (Alves et al., 2026) fixes GPU architecture and reports different A100 and H100 outputs; Cankaya (2026) emulates NVIDIA inference on CPUs. Batch- and parallelism-invariant kernels support deterministic serving (He & Thinking Machines Lab, 2025; Zhang et al., 2025; vLLM Project, 2026); LLM-42 instead uses existing kernels with scheduling and verify-and-rollback (Gond et al., 2026). Gensyn's REE packages reproducible operators and checkable receipts on NVIDIA GPUs and CPUs (Gensyn, 2026; 2025). Power-of-two INT8 scales make two quantized kernels agree on one GPU (Chen, 2026a;b). One Digest's self-published report (Charlot, 2026) describes bitwise-identical FP32 transformer inference across Apple and NVIDIA GPUs, WebGPU and CPU targets. Its reported experiment covers a three-layer forward pass; it does not establish cross-vendor training or AMD GPU support. HEAL (Zhu et al., 2026) reduces numerical divergence and answer flips on NVIDIA and AMD without claiming bitwise identity.

8.5 Portable GPU programming

Kokkos and Triton provide portable GPU programming (Edwards et al., 2014; Tillet et al., 2019); Mojo and MAX supply this work's substrate (Lattner et al., 2021; Modular, 2026b; Johnson et al., 2024), with performance portability shown on H100 and MI300A (Godoy et al., 2025). Meganeura (Malyshau, 2026) trains through Vulkan and Metal on NVIDIA, AMD, Apple and Intel GPUs, validating against error thresholds. Mojmelo (yetalit, 2024) implements classical ML in Mojo for CPUs, with no GPU backend or identity contract. GPU boosting libraries and research (Chen & Guestrin, 2016; Ke et al., 2017; Prokhorenkova et al., 2018; Wen et al., 2020; 2019; Zhang et al., 2017) and RAPIDS (Raschka et al., 2020) lack Apple-silicon GPU support (XGBoost Developers, 2026a; LightGBM Developers, 2026; CatBoost Developers, 2026c; NVIDIA, 2026). MacBoost (Rowley, 2026) trains boosted trees through Apple Metal kernels, targeting single-vendor speed without a training-identity claim.

9 Conclusion

The same machine-learning workload can produce different bits on different processors. Mojolearn removes that variable. In its default configuration, one numerical contract, implemented once in Mojo, gives the same model and output bits across verified Apple, NVIDIA and AMD GPUs, and Arm and x86-64 CPUs, for over 100 machine-learning algorithms in 13 families, from decision trees to neural networks, in both inference and training. A supported imported model, executed through covered operations and configurations, returns the same answers on every verified device. A training run can move between GPU vendors, or be shared among them, without leaving its trajectory, and anyone can check these claims on their own hardware, including against their own CPU. Identity has implementation and execution costs.

Future work. The next steps are improving runtime performance, extending algorithm and option coverage, verifying more GPUs, and stabilizing the library for a 1.0 release. Bounded integer arithmetic offers further optimization opportunities under the conditions described in Section 5.

References

Al Awar, N. and Singanaboina, S. Y. Controlling floating-point determinism in NVIDIA CCCL. NVIDIA Technical Blog, March 2026. URL https://developer.nvidia.com/blog/controlling-floating-point-determinism-in-nvidia-cccl.

Alves, D. R., Patankar, V., Pereira, M., Stephens, J., Vaziri, N., and Kannan, S. EigenAI: Deterministic inference, verifiable results, 2026. URL https://arxiv.org/abs/2602.00182. arXiv:2602.00182 [cs.CR].

Arun, A., St. Arnaud, A., Titov, A., Wilcox, B., Kolobaric, V., Brinkmann, M., Ersoy, O., Fielding, B., and Bonneau, J. Verde: Verification via refereed delegation for machine learning programs, 2025. URL https://arxiv.org/abs/2502.19405. arXiv:2502.19405 [cs.LG].

Badash, E., Boneh, D., Komargodski, I., and Srivastava, M. Hawkeye: Reproducing GPU-level non-determinism. In *MLSys*, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/hash/e217c271a57c365a246b0ad39e668ba8-Abstract-Conference.html.

Breiman, L. Random forests. *Machine Learning*, 45(1): 5–32, 2001. doi: 10.1023/A:1010933404324.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

Cankaya, N. Bit-exact AI inference verification without performance tradeoffs, 2026. URL https://arxiv.org/abs/2606.00279. arXiv:2606.00279 [cs.CR].

CatBoost Developers. CatBoost gpu histogram and partition-statistics kernels, 2026a. URL https://github.com/catboost/catboost/tree/7055d33dd10bc5960bc5a32e4d884dab1a1356e8/catboost/cuda. Source code, revision 7055d33.

CatBoost Developers. CatBoost parameter tuning: Tree growing policy. CatBoost documentation, 2026b. URL https://catboost.ai/docs/en/concepts/parameter-tuning#tree-growing-policy. SymmetricTree, Depthwise and Lossguide policies; accessed October 30, 2026.

CatBoost Developers. pip install. CatBoost documentation, Python package installation, 2026c. URL https://catboost.ai/docs/en/installation/python-installation-method-pip-install. Supported-platforms table lists no GPU support on macOS; accessed September 20, 2026.

Charlot, D. J. One digest: Bit-identical neural computation across heterogeneous compute fabrics. Technical Report TR-2026-23, Institute for Physical AI @ JBI, July 2026. URL https://physicalai-bmi.org/assets/papers/one-digest. Self-published technical report, revised July 24, 2026.

Chen, B., Wen, M., Shi, Y., Lin, D., Rajbahadur, G. K., and Jiang, Z. M. Towards training reproducible deep learning models. In *ICSE*, pp. 2202–2214, 2022. URL https://arxiv.org/abs/2202.02326.

Chen, T. and Guestrin, C. XGBoost: A scalable tree boosting system. In *KDD*, pp. 785–794, 2016. doi: 10.1145/2939672.2939785.

Chen, T.-R. The integer alibi: Localizing cross-kernel divergence in INT8-quantized LLM inference, 2026a. URL https://arxiv.org/abs/2608.13756. arXiv:2608.13756 [cs.LG].

Chen, T.-R. Deterministic LLM inference across GPU kernels: Power-of-two INT8 quantization scales and the limits of tolerance-based conformance, 2026b. URL https://arxiv.org/abs/2609.00363. arXiv:2609.00363 [cs.LG].

Collange, C., Defour, D., Graillat, S., and Iakymchuk, R. Numerical reproducibility for the parallel reduction on multi- and many-core architectures. *Parallel Computing*, 49:83–97, 2015. doi: 10.1016/j.parco.2015.09.001.

Dawson, B. Floating-point determinism, 2013. URL https://randomascii.wordpress.com/2013/07/16/floating-point-determinism/.

Demmel, J. and Nguyen, H. D. Fast reproducible floating-point summation. In *2013 IEEE 21st Symposium on Computer Arithmetic (ARITH)*, pp. 163–172, 2013. doi: 10.1109/ARITH.2013.9.

Demmel, J., Ahrens, W., and Nguyen, H. D. Efficient reproducible floating point summation and BLAS. Technical Report UCB/EECS-2016-121, EECS Department, University of California, Berkeley, June 2016. URL https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-121.html.

Donaghy, J., Wilcox, B., Ersoy, O., Rastogi, S., St Arnaud, A., Titov, A., Greenberg, J., Fielding, B., and Grieve, H. OPEN-1B: A fully auditable training run, 2026. URL https://arxiv.org/abs/2609.17380. arXiv:2609.17380 [cs.LG], submitted September 15, 2026.

Edwards, H. C., Trott, C. R., and Sunderland, D. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014. doi: 10.1016/j.jpdc.2014.07.003.

Gensyn. RepOps, 2025. URL https://github.com/gensyn-ai/repops-demo. GitHub repository: bitwise reproducibility of ML operations across hardware targets.

Gensyn. Introducing REE: Reproducible execution environment, March 2026. URL https://www.gensyn.ai/news/ree.

Geurts, P., Ernst, D., and Wehenkel, L. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 2006. doi: 10.1007/s10994-006-6226-1.

Godoy, W. F., Melnichenko, T., Valero-Lara, P., Elwasif, W., Fackler, P., Ferreira Da Silva, R., Teranishi, K., and Vetter, J. S. Mojo: MLIR-based performance-portable HPC science kernels on GPUs for the Python ecosystem, 2025. URL https://arxiv.org/abs/2509.21039. arXiv:2509.21039 [cs.DC].

Gond, R., Kamath, A. K., Ramjee, R., and Panwar, A. LLM-42: Enabling determinism in LLM inference with verified speculation, 2026. URL https://arxiv.org/abs/2601.17768. arXiv:2601.17768 [cs.LG].

He, H. and Thinking Machines Lab. Defeating nondeterminism in LLM inference, 2025. URL https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/.

Hendel, A. Mojolearn 162M: One model, two hardware routes. Hugging Face, 2026a. URL https://huggingface.co/mojolearn-ai/mojolearn-162m. Final checkpoint, tokenizer, training evidence and evaluations of the six-segment run.

Hendel, A. mojolearn: Bitwise-identical GPU machine learning in Mojo on Apple Metal, NVIDIA CUDA and AMD HIP. Zenodo, 2026b. URL https://doi.org/10.5281/zenodo.22864165. Fixed evidence archive, evidence-0.8.9-cpu-gpu-20260920-r2.

Hendel, A. mojolearn: Bitwise-identical GPU machine learning in Mojo on Apple Metal, NVIDIA CUDA and AMD HIP. Zenodo, 2026c. URL https://doi.org/10.5281/zenodo.22675019. Version 0.7.0 (alpha-api-0.7.0-20260909), software release archived September 9, 2026.

Hendel, A. mojolearn: Bitwise-identical GPU machine learning in Mojo on Apple Metal, NVIDIA CUDA and AMD HIP. Zenodo, 2026d. URL https://doi.org/10.5281/zenodo.22864816. Version 0.8.10 (alpha-api-0.8.10-20260920), software release archived September 20, 2026.

Hendel, A. mojolearn: bitwise-identical GPU machine learning in Mojo on Apple Metal, NVIDIA CUDA and AMD HIP. Zenodo, 2026e. URL https://doi.org/10.5281/zenodo.23498691. Version 0.8.55 (alpha-api-0.8.55-macos-20261020), archived October 20, 2026.

Heumos, L., Ehmele, P., Kuhn Cuellar, L., Menden, K., Miller, E., Lemke, S., Gabernet, G., and Nahnsen, S. mlf-core: A framework for deterministic machine learning. *Bioinformatics*, 39(4):btad164, 2023. doi: 10.1093/bioinformatics/btad164.

Intel. Reproducibility conditions, 2024. URL https://www.intel.com/content/www/us/en/docs/onemkl/developer-guide-windows/2024-2/reproducibility-conditions.html. Developer Guide for Intel oneAPI Math Kernel Library, version 2024.2.

Johnson, E., Lattner, C., Davis, T., Hoge, C., and Clayton, J. MAX is here! what does that mean for Mojo? Modular, February 2024. URL https://www.modular.com/blog/max-is-here-what-does-that-mean-for-mojo.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. LightGBM: A highly efficient gradient boosting decision tree. In *NeurIPS*, 2017.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14, 2021. doi: 10.1109/CGO51591.2021.9370308. URL https://mlir.llvm.org/pubs/.

Li, R., Liu, L., Yang, G., Zhang, C., and Wang, B. Bitwise identical compiling setup: prospective for reproducibility and reliability of earth system modeling. *Geoscientific Model Development*, 9:731–748, 2016. doi: 10.5194/gmd-9-731-2016. URL https://gmd.copernicus.org/articles/9/731/2016/.

LightGBM Developers. LightGBM installation guide, 2026. URL https://lightgbm.readthedocs.io/en/stable/Installation-Guide.html.

Lubin, M. and Mueller-Albrecht, R. Floating point conditional numerical reproducibility on CPU and GPU with Intel oneAPI math kernel library (oneMKL). Intel developer article (archived), 2024. URL https://www.intel.com/content/www/us/en/developer/archive/training/conditional-numerical-reproducibility-cnr.html. Updated March 25, 2024.

Malyshau, D. Meganeura: Portable GPU training and inference through Vulkan and Metal, 2026. URL https://arxiv.org/abs/2608.01563. arXiv:2608.01563 [cs.LG].

Modular. Hardware compatibility. MAX GPU developer guide, 2026a. URL https://max.modular.com/gpu/hardware-compatibility/. GPU programming with Mojo and MAX; accessed October 30, 2026.

Modular. Mojo manual, 2026b. URL https://mojolang.org/docs/manual/.

NVIDIA. RAPIDS installation guide. NVIDIA RAPIDS documentation, 2026. URL https://docs.nvidia.com/datascience/install/. Requires an NVIDIA GPU with compute capability 7.0 or higher on Linux or WSL2; accessed September 20, 2026.

OpenMD. Reproducibility in parallel OpenMD simulations, February 2013. URL https://openmd.org/reproducibility-in-parallel-openmd-simulations/.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, É. Scikit-learn: Machine learning in Python. *JMLR*, 12: 2825–2830, 2011.

Penedo, G., Kydlíček, H., Ben Allal, L., Lozhkov, A., Mitchell, M., Raffel, C., Von Werra, L., and Wolf, T. The FineWeb datasets: Decanting the web for the finest text data at scale. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2024.

Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A. V., and Gulin, A. CatBoost: Unbiased boosting with categorical features. In *NeurIPS*, 2018.

PyTorch. torch.use_deterministic_algorithms, 2026. URL https://pytorch.org/docs/stable/generated/torch.use_deterministic_algorithms.html. PyTorch stable documentation; accessed September 20, 2026.

RAPIDS Development Team. cuML: GPU machine learning algorithms, 2026. URL https://github.com/NVIDIA/cuml/tree/265b9da6a0e75dbef071a3168398b993a5ff6f0e. version 26.08 source and API documentation.

Raschka, S., Patterson, J., and Nolet, C. Machine learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information*, 11(4):193, 2020. doi: 10.3390/info11040193.

Rowley, A. MacBoost: Gradient boosted trees on Apple-silicon GPUs, 2026. URL https://github.com/alex-rowley/macboost. Software, version 1.2.0.

Shi, Y., Ke, G., Chen, Z., Zheng, S., and Liu, T.-Y. Quantized training of gradient boosting decision trees. In *NeurIPS*, 2022.

Srivastava, M., Arora, S., and Boneh, D. Optimistic verifiable training by controlling hardware nondeterminism. In *NeurIPS*, 2024. URL https://arxiv.org/abs/2403.09603. arXiv:2403.09603 [cs.CR].

TensorFlow. tf.config.experimental.enable_op_determinism, 2024. URL https://www.tensorflow.org/api_docs/python/tf/config/experimental/enable_op_determinism. Page dated April 2024; accessed September 20, 2026.

Tillet, P., Kung, H., and Cox, D. Triton: An intermediate language and compiler for tiled neural network computations. In *MAPL*, pp. 10–19, 2019. doi: 10.1145/3315508.3329973.

vLLM Project. Batch invariance. vLLM documentation, 2026. URL https://docs.vllm.ai/en/latest/features/batch_invariance/.

Wen, Z., Shi, J., He, B., Chen, J., Ramamohanarao, K., and Li, Q. Exploiting GPUs for efficient gradient boosting decision tree training. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2706–2717, 2019. doi: 10.1109/TPDS.2019.2920131.

Wen, Z., Liu, H., Shi, J., Li, Q., He, B., and Chen, J. ThunderGBM: Fast GBDTs and random forests on GPUs. *JMLR*, 21(108):1–5, 2020.

Whitehead, N. and Fit-Florea, A. Floating point and IEEE 754 compliance for NVIDIA GPUs. NVIDIA white paper, 2011. URL https://docs.nvidia.com/cuda/floating-point/. Continually updated documentation; accessed September 20, 2026.

XGBoost Developers. XGBoost GPU support, 2026a. URL https://xgboost.readthedocs.io/en/stable/gpu/.

XGBoost Developers. XGBoost GPU gradient quantization. Source code, 2026b. URL https://github.com/dmlc/xgboost/blob/fc73d8a10b0055ad1d96b31bb575477ac80f8133/src/tree/gpu_hist/quantiser.cuh. Fixed-point gradient conversion, revision fc73d8a10b0055ad1d96b31bb575477ac80f8133; accessed October 30, 2026.

Xie, P., Zhang, X., and Chen, S. RepDL: Bit-level reproducible deep learning training and inference, 2025.

URL https://arxiv.org/abs/2510.09180. arXiv:2510.09180 [cs.LG].

yetalit. Mojmelo: Machine learning algorithms in pure Mojo. GitHub repository, 2024. URL https://github.com/yetalit/Mojmelo. accessed 2026-08-28.

Zellers, R., Holtzman, A., Bisk, Y., Farhadi, A., and Choi, Y. HellaSwag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

Zhang, H., Si, S., and Hsieh, C.-J. GPU-acceleration for large-scale tree boosting. *arXiv:1706.08359*, 2017.

Zhang, Z., Ding, X., Yuan, J., Liu, R., Mao, H., Xing, J., and Liu, Z. Deterministic inference across tensor parallel sizes that eliminates training-inference mismatch, 2025. URL https://arxiv.org/abs/2511.17826v2. arXiv:2511.17826, first posted November 21, 2025; version 2 revised May 29, 2026 [cs.LG].

Zhu, Z., Thai, L., Yu, S., Liu, Y., Qiao, Y., Wang, C., Xu, H., and Shu, J. Demystifying numerical instability in LLM inference: Achieving reproducible inference for mission-critical tasks with HEAL, 2026. URL https://arxiv.org/abs/2606.21023. arXiv:2606.21023 [cs.LG].

AI Usage Disclosure

I used Anthropic Claude and OpenAI Codex extensively and Google Gemini sparingly, in developing this work. These tools generated and revised substantial portions of the library implementation, build and packaging infrastructure, documentation, and manuscript text. They also assisted with debugging, evaluation design and execution, and analysis of retained results. AI tools located related work and helped describe and compare it, downloaded datasets used in this work, and created tests and test fixtures.

The verification feature, also AI-generated, is substantially built out with test code, fault injections that check whether the tests detect deliberately introduced errors, and cross-vendor comparisons. These checks provide evidence that the AI-generated code does what it is intended to do. Building a library of this scope and sophistication would not have been feasible for me without AI tooling.

I directed the work, selected the research objectives and acceptance criteria, and made the final decisions about implementation, scope, interpretation, and claims. Reported numerical results are supported by retained execution records; generated explanations are not evidence. I take responsibility for the final manuscript, software, references, and reported results.

A THE VERIFICATION RECORD

A.1 Published-release cross-device verification

The release 0.8.55 evidence archive, `evidence-0.8.55.zip`, retains GPU and CPU evidence documents, per-profile and per-shard direct comparison reports in its `release-identity/` folder; its `MANIFEST.sha256` lists every file's SHA-256.

A *configuration* is one algorithm with one set of options. A *cell* is one configuration on one fixture; its *parts* compare training state, saved-model bytes, held-out predictions and applicable batch or sequence properties. On the GPU targets, the routine profile selects 449 configurations, of which 394 are verified and 55 are not applicable, covering 3,546 cells on nine fixtures. The separate neural-training profile selects 68 configurations.

Reference conformance. The routine profile reports 11,510 identical reference-checked parts and zero divergent parts on each GPU target. The Arm routine run reports 11,375 reference-identical parts, 81 refused parts and zero divergent parts. Direct comparison and reference checking have different counting scopes; the 15,362 comparison parts are not 15,362 reference passes. The evidence archive includes the corrected reference table, which refreshes 36 entries in three optimizer lanes that were outdated in 0.8.55.

CPU coverage. The Arm run explicitly selects the CPU route on Apple M4. The x86 run uses an AMD EPYC cloud CPU and seven routine lane groups, plus the neural-training profile. Each group repeats the same 116 portable parts, so the groups' 11,999 recorded matches are 11,303 unique routine parts, which match NVIDIA and, separately, Arm with zero differences; the neural profiles agree on 2,349 parts. The Arm–NVIDIA routine comparison has 11,375 shared matching parts, 4,212 parts present in only one document, and 81 parts computed by neither side. Of the 11,510 reference-checked parts each GPU target verifies, 72 belong to two ranking lanes with no CPU route (the public interface refuses them on a CPU-only install), so 11,438 have a CPU route; the Arm run verifies 11,375 of them (99.4%). The other 63 are two weighted extra-trees lanes whose shipped table has no CPU entry (54 parts, owed, and equal across devices where compared) and the splitter below (9 parts). The parts present only in the GPU documents are GPU-specific checks, such as GPU-against-CPU inference, that have no CPU counterpart. Both CPU routes refuse 81 parts of a stratified group splitter whose native helper is absent from this release. The seven completed x86 groups report 11,303 unique reference-identical parts and zero divergent parts. Two x86 approximate-neighbor lanes (IVF-PQ and filtered ANN) terminate their workers; the affected groups were rerun without those lanes.

Scope and limitations. The nine fixtures cover ordinary dense input, ties, hash-derived inputs, wide column magnitudes, subnormal values and their flushed diagnostic twin, duplicate rows, odd dimensions, and negative values. Documents retain input hashes and explicit non-applicability. These runs used neither challenge responses nor exchanged commitments, so document agreement does not itself prove fresh computation rather than copied answers, and this campaign does not establish repeated-run stability.

A.2 Deliberate arithmetic changes (negative controls)

A negative control counts for a configuration when paired clean and deliberately altered runs on the same device class differ in at least one recorded part. Arithmetic changes test implementation sensitivity; harness perturbations test the comparison itself.

Three deliberate changes illustrate what the comparison can detect. Reversing the fold order of the language model's output-head product changes 9 of 33 recorded losses on every CPU tested. Transposing one operand of the backward matrix product changes 8 of 10 recorded arrays and leaves all two losses equal, which is what corrupting the backward pass alone should produce, and the comparison names the first differing tensor with both bit patterns. Deliberately zeroing the optimizer moments at a transferred checkpoint changes the following parameters and both moments while preserving the loss, gradients and token inputs. The property parts have controls of their own, which perturb one row of a whole-batch evaluation and must turn every batch-invariance cell from identical to divergent.

A.3 Two-GPU coverage and uncovered options

The separately retained 59 two-GPU configurations are also complete across the three GPU vendors (Table 6): every numerical part has a value in the Apple, NVIDIA and AMD columns, and no part disagrees. These values were completed after the 0.8.10 release archive was frozen; they are the reference table at repository revision 47372db76, SHA-256 e95fbbb8…d0b7c9. The release archive itself holds 1,494 of the 1,935 parts complete across the three vendors.

two-GPU numerical-part coverage	parts
CPU and all three GPU columns agree	771
all three GPU columns agree; no numerical CPU role	1,164
incomplete GPU agreement	0
total numerical parts	1,935

Table 6. Coverage of the 59 two-GPU configurations on nine fixtures. Each vendor's column runs the partitioned plan on one device. These parts are excluded from the single-device totals, and non-applicable roles are not counted.

Target compared with L40S	Execution route	Routine agreement	Neural agreement	Differences
Apple M4	Metal	15,362	2,349	0
AMD MI300X	HIP (gfx942)	15,362	2,349	0
Arm CPU	Apple M4, CPU route	11,375	2,349	0
x86-64 CPU	AMD EPYC, CPU route	11,303	2,349	0
NVIDIA RTX A6000	PTX build (sm_80), device sm_86	15,362	2,349	0

Table 5. Release 0.8.55 direct evidence-document comparisons. The anchor is an NVIDIA L40S using native CUDA (sm_89). Counts are comparison-report parts, excluding non-applicability markers; profiles are reported separately and are not summed into a unique-coverage total. CPU counts report shared numerical results; coverage details and reference conformance appear in Appendix A.1.

Physical two-GPU execution. With release 0.8.10 installed from PyPI on two NVIDIA RTX 4090 GPUs, the verifier's self-test passed and its quick check verified 51 numerical parts across 14 configurations, each with two distinct GPU identifiers. Across nine fixtures, the full sweep recorded 1,935 matching numerical parts and no numerical divergence. Of these, 1,908 parts across 56 configurations had accepted two-device witnesses; the remaining 27 parts, in three byte-LM configurations, matched numerically but lacked accepted device witnesses, so the affected commands were incomplete rather than verified. A subsequent verifier-only overlay on the same wheel checked the base fixture of the three configurations: two numerical parts matched with accepted two-device witnesses, while the offload configuration was classified as single-device and excluded from the two-device comparison. This follow-up was not a repeat of the full nine-fixture sweep. Separately, the recorded two-GPU configurations of release 0.8.5 on two AMD MI300X GPUs match the reference values; these are historical records, not a completed AMD qualification of the 0.8.10 wheel.

The source inventory separately enumerates 255 public API entries, including aliases, wrappers and helpers; every entry maps to a verifier configuration. That inventory is not a count of distinct algorithms, and mapping an entry does not establish complete coverage of its options. An interface can accept an option value that no recorded configuration exercises; such a call runs under the same contract but is not a verified result until a configuration records it.

What is counted as an algorithm. Table 1 groups the current algorithms into 13 families. An algorithm is a method a practitioner would name. A classifier and a regressor of the same method count once, as do its losses, kernels, distance metrics, penalties, starting rules and sharded two-GPU execution, all of which are configurations. The release 0.8.55 routine and neural-training profiles are described in Appendix A; Table 6 reports the separate older two-GPU coverage. The 24 evaluation metrics, such as accuracy, ROC AUC, silhouette and trustworthiness, are verified the same way and are not counted as algorithms.

A.4 Custom-model and imported-model evidence

The October 20 record tests a layered classifier and a mixture classifier defined through the public layer and tensor-operation interfaces. Their trained-parameter and prediction digests agree across Apple, NVIDIA and AMD; the record also retains matching session-training and exported-prediction digests. The six imported models are MNIST-12, MobileNet V3 Small, ResNet-18, ResNet-50, SmolLM2-135M and SqueezeNet 1.1. All use batch one and fixed seeded inputs: $1 \times 3 \times 224 \times 224$ for the four ImageNet-shaped networks, $1 \times 1 \times 28 \times 28$ for MNIST, and 32 token IDs for SmolLM2. The eighteen vendor/model records agree across vendors and repeat within each vendor. These checks establish agreement on the stated inputs; source-framework numerical comparisons are recorded separately and are not task-accuracy measurements.

This is a separate, composite source-built record. NVIDIA and AMD used revision 457c727d4; the Apple custom-model log comes from a separate build whose exact source revision was not retained. Imported-model receipts report version 0.8.54 and are carried forward explicitly in the 0.8.55 catalog. A retained ResNet-18 program and NVIDIA RTX 3090 receipt were checked on an Apple M4 with a 0.8.55 binding: the model-file hash, input digest and output digest agree. The `custom-models/` folder of `evidence-0.8.55.zip` retains the raw records, source recipes, provenance, an evidence checker and the location and hash of the saved model bytes. This record does not extend the release-verifier fixture totals or the performance tables.

A.5 Neural configurations and scale demonstrations

Verifier fixtures are small so that anyone can run them. The following training demonstrations provide separate checks beyond those fixtures.

A GPT-3 Small-shaped model. The decoder has 12 layers, width 768, 12 attention heads, context 2,048, a 50,257-token vocabulary and 162,147,840 parameters. Its blocks follow the Llama layer order (rotary positions, RMSNorm and a SwiGLU feed-forward of width 2,048), with untied input and output embeddings and no final norm; batch,

peak learning rate and AdamW settings follow GPT-3 Small (Brown et al., 2020). The released final checkpoint has a held-out FineWeb-Edu loss of 3.27 nats per token (perplexity 26.2) and 29.6% zero-shot HellaSwag accuracy (Zellers et al., 2019) (length-normalized). GPT-3 Small reports 33.7% after 300 billion training tokens, about 114 times this run's budget. Each optimizer step processes 64 logical shards of four sequences, or 524,288 tokens, using the same pinned FineWeb-Edu (Penedo et al., 2024) token stream and FP32 learning-rate table. Both routes start from the same step-0 checkpoint and continue from their own saved state. The state digest covers parameters, both AdamW moments and initialization flags; separate records bind the step counter, shard indices, recipe and token stream. Every step records the summed-gradient digest, 64 loss bit patterns and learning-rate bits as well.

segment	equal steps	divergent	files	endpoint SHA-256
1	1,000	0	11	ce8c6b98e45b
2	1,000	0	11	f03471f9afa9
3	400	0	5	82249517f68c
4	1,500	0	16	d935aba7cb29
5	1,000	0	11	8142c3eaec17
6	100	0	2	4129921e99ed
total	5,000	0	56	

Table 7. Six-segment training evidence. Each equal step has matching state and summed-gradient digests, all 64 loss bit patterns, learning-rate bits and shard indices across routes. The files column counts matching checkpoint digests at common saved steps, excluding initialization. Endpoint digests are abbreviated for display only; all comparisons use full digests.

Handoff and simultaneous training. For the NVIDIA-to-Apple handoff into A/6, the M3 Ultra loads the H100's step-4,898 checkpoint and independently recomputes steps 4,899 and 4,900. Both records match the sender's state, gradient, loss and learning-rate fields. It then trains steps 4,901–5,000; the L40S independently computes those same 100 updates in B/6 and writes the same final checkpoint digest. The retained raw arrival records permit the same comparison for eight handoffs (16 replayed steps); the ledger additionally records PASS for the arrivals into A/4 and B/5. In the simultaneous A/3 segment, the H100 computes logical shards 0–43 and the MI325X computes shards 44–63. Both workers record equal state, gradient, loss and learning-rate fields at every one of its 400 steps; B/3 independently obtains those values with one H100. Recovery overlaps within A/4 and B/5 are checked for equality and counted only once in Table 7.

Faithful replays and deliberate alterations. Faithful replays reproduce the reference computation (Table 8). In separate negative-control runs, optimizer state, shard count, shard order or a gradient value is *deliberately* changed to test whether verification detects the alteration (Table 9). Every replay matches and every deliberate alteration is detected. These controls test the checker's sensitivity; the unmodified training routes have zero divergent steps (Table 7).

replay from step 100	result
ordinary replay (H100, MI325X)	matches
split-fold replay (H100)	matches

Table 8. Successful replay controls at the scale experiment's full model shape. Every replay reproduces steps 101 and 102 with no changed fields.

deliberate change after step 100	result	changed fields
zero optimizer moments	detected	state
63 rather than 64 shards	detected	state, grad., loss
swap shards 5 and 40	detected	state, grad., loss
swap shards 0 and 1	detected	loss only
one-ULP gradient edit (H100, MI325X)	detected	state, grad.

Table 9. **Deliberately altered computations: every change detected.** These negative controls use the scale experiment's full model shape. Each alteration is detected at step 101; its archived FAIL verdict means the identity check successfully rejected the deliberate change. Loss denotes the ordered per-shard bit patterns.

A detected one-ULP change. In the H100 one-ULP control, element 123,585,811 of shard 63's gradient is deliberately changed from FP32 bits 3e4e221f to 3e4e2220. The summed element changes from 4139eddb to 4139eddc; at step 101 the state digest changes from abc8b816b5c3 to d3eefb6dba87 and the gradient digest from 25830bfc2016 to f386db390c34, while all 64 losses remain equal. The positive split-fold control exercises that same path without the edit and passes. Swapping only shards 0 and 1 is detected through loss ordering even though their commutative first addition leaves the summed gradient and updated state unchanged.

The evidence uses the recorded sliced-sha256-8.v2 digest scheme, which hashes all array bytes in eight contiguous ranges and combines their digests, and full SHA-256 for checkpoint files. The separate scale-experiment package (Appendix B) retains the step chains, projected numerical and environment receipts, checkpoint digests, checkpoint-source records and control records with a file manifest and an evidence-file checker. The checker verifies record completeness, agreement and checkpoint continuity without a GPU; it does not recompute training or independently validate the recorded GPU arithmetic. The `gpt3-six-segment/` folder of `evidence-0.8.55.zip` holds anonymized copies of these records and the checker's report on the originals. The run spans published wheels 0.8.15–0.8.24; the Apple segment uses 0.8.24. It is a separate scale record, not an extension of the release 0.8.55 fixture campaign.

A byte-level language model, step by step. A model with two decoder blocks and 34,944 parameters trains for 128 steps on pinned real text on all three vendors, and every recorded array of every step matches: parameters, gradients, AdamW moments, losses, held-out outputs and checkpoint bytes. The step-64 checkpoint resumes on the other vendor in both directions and rejoins the uninterrupted run. On seven CPUs with no GPU present, each given one step's recorded state, all 640 recomputed gradients, losses, parameters and moments equal the GPU record. Its records are in the `byte-lm/` folder of `evidence-0.8.55.zip`.

One model trained on three vendors at once. The fixed fold that makes two-GPU training match one GPU also lets GPUs of different vendors, in different machines, train one model together. In the experimental mojolearn.cross_vendor interface, which is not part of release 0.8.10, each worker computes the gradients of the logical microbatches it owns, a coordinator sums all of them in microbatch order with the fold the GPU reduction uses, and every worker applies that sum with the same AdamW update. An Apple M4, an NVIDIA H100 and an AMD MI300X, each in its own machine, trained one decoder with 2 blocks, width 32, a 512-token vocabulary and 51,328 parameters for 6 steps, with four microbatches per step (two on the M4 and one on each of the other GPUs). After every step the three workers reported the same SHA-256 of the complete training state (parameters, both AdamW moments and flags), and the first 16 hexadecimal digits of each state hash match the recorded single-GPU baseline for processing all four microbatches itself.

Runs of the same model in one process reach the same six states on one M4, one H100, two H100 GPUs and one MI300X. A checkpoint written after step 3 by the two H100 GPUs, and one written by the MI300X, each resume on the M4 and rejoin that trajectory. Offline replay uses four gradient-capture records: one M4, one H100, two H100 GPUs and one MI300X. Assigning each of four microbatches to one of these records, excluding assignments drawn entirely from one record, gives $6(4^4 - 4) = 1{,}512$ mixed-source gradient-fold checks across six steps. Every check reproduces the reference sum; these are offline replays, not separate live distributed training runs. The coordinator stops the run when any two workers' state hashes differ. In a negative-control test, one worker's summed gradient was deliberately changed by about 10^{-7} in a single element; the coordinator detected the change and rejected that worker as intended. Every step exchanges full gradients, so the result concerns identity, not throughput.

B Artifact

The primary release 0.8.55 verifier evidence is retained in `evidence-0.8.55.zip`, published with this preprint, with anonymized documents, direct comparison reports and content hashes. The name "anonlib" in these evidence files refers to Mojolearn. The older archives below preserve historical evidence and the separate two-GPU record.

Mojolearn is open source under Apache-2.0 at https://github.com/mojolearn/mojolearn. The primary software release evaluated in this paper is Mojolearn 0.8.55 (Hendel, 2026e), archived at DOI 10.5281/zenodo.23498691. Release 0.8.10, used for the NVIDIA two-GPU results, is published on PyPI for macOS arm64 and Linux x86-64 and archived as (Hendel, 2026d), DOI 10.5281/zenodo.22864816; its reference table preserves the historical single-device record. That older numerical record is preserved in the fixed evidence archive (Hendel, 2026b), DOI 10.5281/zenodo.22864165. It includes the canonical table, the machine-readable audit, the raw-record manifest and all 548 referenced record entries, plus release receipts. The paper audit verifies the manifest hashes and the 29,862 recorded numerical column values underlying that historical result. The project archive uses concept DOI 10.5281/zenodo.22068632.

The six-segment scale experiment has a separate supplement containing the complete per-step comparison records and an evidence-file checker (Appendix A.5). The final checkpoint is published with that supplement as a model repository, together with the tokenizer, a weights-only export, an offline integrity verifier and held-out evaluations. The model repository is https://huggingface.co/mojolearn-ai/mojolearn-162m (Hendel, 2026a), DOI 10.57967/hf/10832; its source, with the supplement, is in the Mojolearn repository at https://github.com/mojolearn/mojolearn/tree/main/release/gpt3-small-162m.

Implementation size. Table 10 counts the release 0.8.55 source snapshot at revision 4e927c92 with cloc 2.06, code lines only, over the committed Git archive, including the language and configuration categories listed below; comments, blank lines, generated reference data and prose are excluded. The Mojo is divided by role. The library row holds the estimators, kernels and bindings. The verification row holds the host reference implementations the GPU kernels are compared against, the per-family checks, the fixtures and the deliberately altered test implementations (sabotage arms), whose changes must be detected before a control counts; 930 source files outside the result directories name a sabotage arm. Verification is 0.6 times the size of the library it checks. Python holds the public package, its tests and the gate, tooling and packaging scripts. The result directories retain another 10,190 lines of copied scripts beside the evidence they produced, and the table excludes them. Code size describes implementation scope, not numerical coverage or correctness.

language and role	files	code lines
Mojo library (estimators, kernels, bindings)	1,727	476,325
Mojo verification (oracles, checks, fixtures, sabotage arms)	933	299,029
Mojo benchmarks	61	17,021
Python package	193	76,938
Python tests	294	42,783
Python gates, tooling and packaging	1,058	204,744
Python benchmarks	92	11,962
Python reference scripts	174	18,195
shell	693	59,163
C/C++ and headers	23	3,237
YAML and TOML configuration	24	3,920
total	5,272	1,213,317

Table 10. Release 0.8.55 source by language and role at revision 4e927c92. Code lines exclude comments and blanks. The Mojo rows sum to 792,375 and the Python rows to 354,622.

C MEASUREMENTS BY DEVICE

Performance comparisons use the published release 0.8.55 wheels on NVIDIA L40S and AMD MI300X GPUs, measured on October 30, 2026. Each row compares a complete algorithm against the fastest admitted GPU opponent on the same hardware and dataset, with matched work-changing settings. The tables show ties and slower runtimes; workloads without a completed GPU opponent are omitted. The full measurement record in `evidence-0.8.55.zip` also includes building blocks and inference results.

Times are reported in milliseconds, with measured quality alongside them. Whole-operation timings include host-to-device transfer; † marks comparisons using kernel-only timings on both sides, and § marks asymmetric transfer accounting, as defined in the table captions. Ratios divide my runtime by the opponent's. The retained board cells identify their scored rounds, effective settings, workload dimensions and timing provenance for both arms.

The campaign includes neural forward and training operations, classical estimators and tree methods. Neural comparisons include compiled and eager PyTorch, CUDA-graph arms where completed, and library Mamba implementations; each row selects among the opponents admitted by its board audit. The headline tables contain complete algorithms; building blocks and inference comparisons remain in the full record. Coverage depends on completed, admitted opponent runs on each device and is not a full performance census.

The pinned boards retain each harness revision, paired observations, quality results and comparison-clock audit. The source manifest preserves the previous measurement snapshot separately; its older functional review is not evidence for these release timings. Benchmark output hashes are retained separately from release-verifier results; performance admission does not establish cross-device identity.

C.1 NVIDIA L40S

Table 11: NVIDIA L40S: runtime comparisons. Times in milliseconds; ratios divide my runtime by the opponent's. Quality is shown alongside runtime. Snapshot: October 30, 2026.

family	algorithm	dataset	my ms	opponent	their ms	my ms ÷ their ms	my quality	their quality
neural	transformer-forward	gaussian	5.57	torch-graph-fp32 2.13.0+cu129	2.36	**2.36**×	rel_fro_vs_torch_eager_fp32 6.06e−8	0
neural	transformer-decode	gaussian	1,100	torch-graph-fp32 2.13.0+cu129	326	**3.38**×	max_abs_diff_vs_host 0	–
neural	lm-forward	bytes	44.9	torch-compile-bf16 2.13.0+cu129	31.1	**1.45**×	mean_nll 9.02	9.02
neural	lm-train-step	bytes	35.5	torch-compile-bf16 2.13.0+cu129	10.8	**3.28**×	loss_last_step 8.41	8.40
neural	mamba1-forward	gaussian	3.43	mamba-ssm-tf32 2.3.2.post1+ge9594ce1c732	3.20	**1.07**×	rel_fro_vs_torch_eager_fp32 6.38e−9	6.4e−7
neural	mamba2-forward	gaussian	5.57	mamba-ssm-tf32 2.3.2.post1+ge9594ce1c732	2.81	**1.98**×	rel_fro_vs_torch_eager_fp32 1.5e−7	9.07e−5
neural	mamba3-forward	gaussian	3.95	mamba-ssm-tf32 2.3.2.post1+ge9594ce1c732	3.07	**1.29**×	rel_fro_vs_torch_eager_fp32 5.55e−8	1.99e−4
neural	samba-forward	bytes	4.60	torch-graph-tf32 2.13.0+cu129	1.38	**3.34**×	mean_nll 5.64	5.64
neural	samba-train-step	bytes	45.7	torch-eager-tf32 2.13.0+cu129	11.8	**3.88**×	loss_last_step 4.82	4.82
neural	mlp-train-step	gaussian	1.90	torch-eager-tf32 2.13.0+cu129	1.68	**1.13**×	loss_last_step 1.12	1.12
trees	gbdt-lossguide	taxi	63,000	xgboost-gpu 3.2.0	11,400	**5.52**×	auc 0.632	0.632
trees	gbdt-symmetric	istella	9,950	catboost-gpu 1.2.10	7,450	**1.34**×	auc 0.980	0.980
trees	gbdt-symmetric	taxi	6,150	catboost-gpu 1.2.10	5,480	**1.12**×	auc 0.630	0.630
trees	rf	istella	30,800	cuml-rf-gpu 26.08.00	23,700	**1.30**×	auc 0.945	0.945

NVIDIA L40S, continued

family	algorithm	dataset	my ms	opponent	their ms	my ms ÷ their ms	my quality	their quality
classical	dbscan	istella	6.03e5	cuml-gpu 26.08.00	66,000	**9.13**×	n_clusters 40,100	40,100
classical	dbscan	taxi	1.58e6	cuml-gpu 26.08.00	4.4e5	**3.60**×	n_clusters 36.0	36.0
classical	hdbscan	istella	1,190	cuml-gpu 26.08.00	905	**1.32**×	silhouette −0.00386	−0.0182
classical	hdbscan	taxi	423	cuml-gpu 26.08.00	365	**1.16**×	silhouette 0.00150	0.0120
classical	svc	taxi	761	cuml-gpu 26.08.00	383	**1.98**×	accuracy 0.767	0.767
classical2	svr	istella	63.9	cuml-gpu 26.08.00	58.8	**1.09**×	r2 0.318	0.318
classical2	svr	taxi	65.3	cuml-gpu 26.08.00	49.1	**1.33**×	r2 0.768	0.768
classical2	kernel-ridge	istella	533	cuml-gpu 26.08.00	341	**1.56**×	r2 0.385	0.385
classical2	kernel-ridge	taxi	533	cuml-gpu 26.08.00	337	**1.58**×	r2 0.727	0.727

C.2 AMD MI300X

Table 12: AMD MI300X: runtime comparisons. Times in milliseconds; ratios divide my runtime by the opponent's. Quality is shown alongside runtime. Snapshot: October 30, 2026.

family	algorithm	dataset	my ms	opponent	their ms	my ms ÷ their ms	my quality	their quality
neural	transformer-forward	gaussian	8.04	torch-graph-fp32 2.13.0+rocm7.1	0.930	**8.64**×	rel_fro_vs_torch_eager_fp32 5.72e−8	0
neural	transformer-decode	gaussian	2,440	torch-eager-fp32 2.13.0+rocm7.1	929	**2.63**×	max_abs_diff_vs_host 0	–
neural	lm-forward	bytes	58.5	torch-eager-fp32 2.13.0+rocm7.1	18.4	**3.18**×	mean_nll 9.02	9.02
neural	lm-train-step	bytes	43.9	torch-compile-bf16 2.13.0+rocm7.1	9.70	**4.53**×	loss_last_step 8.41	8.40
neural	mamba1-forward	gaussian	3.02	mamba-ssm-fp32 2.3.2.post1+ge9594ce1c732	1.40	**2.15**×	rel_fro_vs_torch_eager_fp32 7.44e−9	7.34e−9
neural	mamba2-forward	gaussian	5.28	mamba-ssm-fp32 2.3.2.post1+ge9594ce1c732	2.71	**1.95**×	rel_fro_vs_torch_eager_fp32 1.37e−7	7.52e−8
neural	mlp-train-step	gaussian	3.32	torch-compile-bf16 2.13.0+rocm7.1	1.80	**1.84**×	loss_last_step 1.12	1.12
trees	gbdt-depthwise	taxi	3,550	xgboost-gpu 3.2.0	3,260	**1.09**×	auc 0.632	0.632
trees	gbdt-lossguide	taxi	19,500	xgboost-gpu 3.2.0	17,700	**1.10**×	auc 0.632	0.632

The published fit and inference comparisons are in the measurement record of `evidence-0.8.55.zip`, `measurements/measurements_full.pdf`, generated from the two boards in `measurements/paper-tables/`.