GCPU: A CPU-First Training Runtime and Morpheus, a Categorical Deep-Learning Language

Jovonni L. Pharr
Georgia Cyber Warfare Range
info@gacwr.org

July 25, 2026
v0.0.1

Abstract

Access to accelerator hardware has become the dominant constraint on who can train neural networks. This paper describes two connected systems built at the BlackMind lab to soften that constraint. The first, **GCPU/Forge**, is a CPU-first training runtime that treats a commodity CPU as a first-class training device by borrowing techniques from game-engine engineering: tile-local kernels, NUMA-aware scheduling, and—most importantly—aggressively compressed optimizer state. On an Apple M1 Pro, Forge's 4-bit optimizer (AdamW4) reduces optimizer memory by a measured 7.72–7.74× relative to standard FP32 AdamW across three GPT-style models spanning 70M to 760M parameters, and an end-to-end training benchmark on a 25.2M-parameter transformer shows a 1.39× wall-clock speedup and a 1.64× reduction in peak memory against an eager PyTorch CPU baseline, at essentially unchanged final loss (6.94 versus 6.92). The second system, **Morpheus**, is a compiled, statically typed programming language in which category theory *is* the type system: compositions are type-checked as categorical morphisms, and structural laws (functor laws, algebra homomorphisms, monad laws) are discharged as machine-checked proofs in Lean 4 [9]. The current Morpheus implementation type-checks 27 of 29 conformance programs (2 expected failures), passes 20 of 20 test suites, compiles 23 of 23 example models to native `arm64` binaries, and proves 11 Lean 4 theorems with zero `sorry`. We report these numbers honestly, including where the advantage is small or negative, and we are explicit about the substantial gap between the implemented systems and the broader specification they are drawn from.

1 Introduction

Training a modern neural network is, in practice, gated less by ideas than by hardware. A researcher without access to a GPU cluster—a student, a hobbyist, someone in a region where accelerators are scarce or export-restricted—is largely locked out of the loop of building and studying models from scratch. The prevailing assumption in the ecosystem is that the CPU is a fallback: a place to run inference on small models, or to debug before moving to a real accelerator. The work described here starts from the opposite premise. It asks what a training runtime looks like if the CPU is the intended target rather than the consolation prize, and it asks whether the discipline of a language and type system can make architecture design less of a trial-and-error craft.

The two systems in this paper answer different halves of that question and were developed together. **GCPU/Forge** is the systems half: a runtime that makes a CPU behave, from the user's point of view, like a training device, complete with a named PyTorch device, streams, events, and memory statistics, but backed underneath by a thread pool and SIMD kernels rather than a driver, exposed through a PyTorch-compatible interface [4]. **Morpheus** is the language half: a compiled language whose programs *are* categorical diagrams, so that a program which

type-checks is, by construction, a structurally valid architecture, and whose compiler can emit formal proofs that the architecture respects the structure it claims to.

The honest framing matters, and we hold to it throughout. A commodity CPU does not have the memory bandwidth or the parallelism of a datacenter GPU, and no amount of engineering changes that. The claim is narrower and, we believe, defensible: that on a CPU, *bandwidth and optimizer-state footprint dominate the training budget before raw floating-point throughput does*, and that a runtime built around that fact can make local training of small and mid-sized models genuinely practical rather than merely possible. The measured results in Section 4 are consistent with that claim and with its limits. We do not claim a CPU can replace a GPU for dense large-model pretraining, and we say so where the data would otherwise be read as such.

The contributions of this paper are: (i) a description of the Forge runtime as actually implemented, centered on its 8-bit and 4-bit quantized optimizers and its virtual-device abstraction; (ii) measured memory and throughput results on an Apple M1 Pro, reported with their baselines and their failure modes; (iii) a description of the Morpheus language and its compiler, including the Lean 4 proof layer; and (iv) a description of Morpheus's type-directed architecture search, whose central design property—that the type system rejects structurally invalid compositions before any training occurs—shrinks the effective search space relative to conventional neural architecture search over untyped graphs.

2 Background and Context

2.1 Why the CPU wall is a memory wall

The intuition that a CPU is simply "a slow GPU" is misleading for training. The binding constraint on a CPU is usually not arithmetic throughput but the movement of bytes: parameters, gradients, activations, and optimizer moments all have to be read and written through a memory hierarchy with far less bandwidth than an accelerator's. A standard Adam-family optimizer stores two moment estimates per parameter in FP32, roughly 8 bytes of optimizer state per parameter on top of the parameters themselves; for the model plus its FP32 master copy the resident cost approaches 12–16 bytes per parameter. On a machine with tens of gigabytes of RAM rather than tens of gigabytes of high-bandwidth memory, this state is frequently the wall that is hit first, before any question of speed arises.

This reframing—minimize bytes moved per useful parameter update rather than maximize raw matrix-multiply throughput—is the organizing principle of the runtime described here. It is also why the design borrows so heavily from game engines, which have spent decades extracting predictable performance from CPUs under a fixed frame budget: persistent worker pools instead of per-operation thread churn, cache-sized tiles instead of monolithic tensors, resource aliasing instead of fresh allocation, and background streaming of data that does not fit in the fast tiers. The analogy is not decorative. A training step, compiled and scheduled like a render frame, exposes exactly the reuse and locality that a naive Python training loop discards.

2.2 Deep learning as an algebraic theory

The second system rests on a recent line of work that treats deep learning as an algebraic, categorical theory rather than a bag of heuristics [1, 2, 3]. The central idea, following the categorical deep-learning program [1], is that an equivariant neural network layer is an *algebra homomorphism* for an endofunctor: given a structure map α that folds an F-structure into a carrier object, a layer h respects the structure exactly when the square relating $F(h)$, α, and the target structure map β commutes, i.e. $h \circ \alpha = \beta \circ F(h)$. This single equation recovers group equivariance (convolutions as translation-equivariant maps, hence circulant weight matrices), but also generalizes beyond it: folds over lists, reductions over trees, and—by passing to a two-category

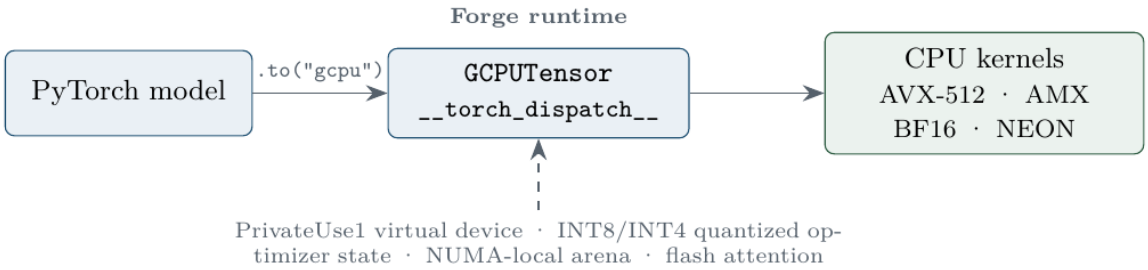


Figure 1: Forge presents a commodity CPU to PyTorch as a virtual PrivateUse1 device: tensors moved to the `gcpu` device are intercepted by `__torch_dispatch__` and executed on ISA-specific CPU kernels, with quantized optimizer state and a NUMA-local arena beneath.

of parametric morphisms—weight tying across layers, which ordinary group-equivariance arguments cannot express. Recurrent networks, recursive networks, and automaton-style networks all reappear as instances of lax algebra homomorphisms in this parametric setting.

The consequence for a language designer is striking. If layers are homomorphisms and networks are composites of morphisms, then a type system that knows about categories, objects, morphisms, functors, and algebras can check—at compile time—that a proposed architecture is structurally coherent, and can hand the harder commutativity obligations to a proof assistant. This is the premise Morpheus is built to test. It is worth stating plainly that this is a young theory; the point of an implementation is to find out where the abstraction pays for itself and where it becomes a rabbit hole.

2.3 The relationship between the two systems

Forge and Morpheus meet at the code-generation boundary. Morpheus compiles categorical diagrams down through an intermediate representation to executable code, and one of its intended backends is Forge: a Morpheus model tagged for CPU training can be lowered to Forge's quantized optimizers and SIMD kernels. In the current implementation the two are developed and evaluated largely independently—Forge as a PyTorch-facing runtime, Morpheus as a standalone compiler emitting native binaries via a Rust and libtorch toolchain—but they share the same motivating goal and the same measured hardware.

3 The Forge Runtime

3.1 Layered architecture

Forge is organized as four layers. At the top is a Python frontend, `gcpu.nn` and `gcpu.optim`, offering drop-in replacements for the corresponding `torch` modules: linear and normalization layers, multi-head and grouped-query attention, a complete decoder-only transformer, recurrent modules, convolutions, and LoRA adapters, alongside the optimizers. Below it, Forge registers itself as a PyTorch `PrivateUse1` device renamed `gcpu`, so that user code can call `tensor.to("gcpu")`, query `torch.gcpu.memory_allocated()`, and open streams and events exactly as it would for a GPU device; a tensor subclass intercepts ATen operations via `__torch_dispatch__` and routes them to CPU execution or, where available, to native kernels. A third layer is a `torch.compile` backend that uses AOTAutograd to capture the forward and backward graph once and apply fusion, tiling, layout, rematerialization, and state-quantization passes. The lowest layer is a set of C++ kernels selected at runtime by ISA detection—AVX-

512 and AMX on recent x86, NEON on ARM—covering GEMM, attention, normalization, and fused activations.

We are candid that the layers are not equally mature. The virtual-device dispatch path executes correctly by unwrapping to CPU tensors and rewrapping results, which gives Forge tensors a genuine device identity while the native C++ kernels handle the hot paths where they are compiled in. The results reported below are those that the current implementation actually produces on the reference machine.

3.2 Quantized optimizer state

The core innovation, and the source of Forge's largest measured advantage, is the compression of optimizer state. Standard AdamW [5] stores the first and second moments in FP32. Forge's **AdamW8**, following the approach of block-wise 8-bit optimizers [6], instead stores each moment in 8-bit blockwise-quantized form: within each block of 256 parameters, the moment values are quantized to INT8 with a shared FP32 scale, costing approximately 1.016 bytes per parameter per moment. With parameters held in BF16 at 2.0 bytes per parameter, the total resident cost is about 4.0 bytes per parameter, against roughly 12 bytes per parameter for FP32 AdamW—a factor of about 3× in optimizer footprint. The **AdamW4** variant compresses the moment representation further still; whatever the exact 4-bit encoding, its measured effect is the 7.72–7.74× optimizer-memory reduction reported in Table 1, which we treat as the empirical statement of record.

Quantization is not free. Dequantizing a moment at each step introduces approximately 0.4% error per moment, which the implementation notes compounds to about 0.5% over a full training run—within the tolerance at which convergence is, empirically, unaffected, a claim borne out by the near-identical final losses in the end-to-end benchmark.

3.3 Kernels, attention, and sharding

Forge provides several GEMM implementations selected by CPUID: an AMX BF16 path for Sapphire Rapids and later, an AVX-512 BF16 path for Ice Lake and later, an AVX2 FP32 path for any modern x86, and a NEON path for ARM including Apple Silicon and Graviton. Attention uses an online-softmax ("flash") formulation [7] that streams the query in tiles and never materializes the full $N \times N$ score matrix, turning attention's activation memory from $O(N^2)$ into $O(N)$; for a sequence length of 2048 this is reported to save on the order of 16 MB per layer per batch. Normalization and activation kernels (RMSNorm, LayerNorm, GELU, SiLU, Mish) have BF16 variants with FP32 accumulation.

For models that exceed the DRAM attached to a single socket, Forge implements a ZeRO-3-style [8] sharding scheme across CPU sockets: each socket owns a shard of parameters and moments, gathers the full shard it needs for its layers during the forward pass and releases it immediately, and after a gradient all-reduce updates only its own shard. A two-socket machine can thereby train a model roughly twice as large as would fit in a single socket's memory. An out-of-core mode extends this further by memory-mapping cold optimizer state onto NVMe, in the manner of game-engine asset streaming; this path is described in the design but is not part of the measured results here.

4 Experiments and Results

Unless stated otherwise, all Forge measurements were taken on an Apple M1 Pro (`arm64`), which exercises the NEON kernel path—among the slower of Forge's targets, so the numbers should be read as conservative relative to an AVX-512 or AMX host.

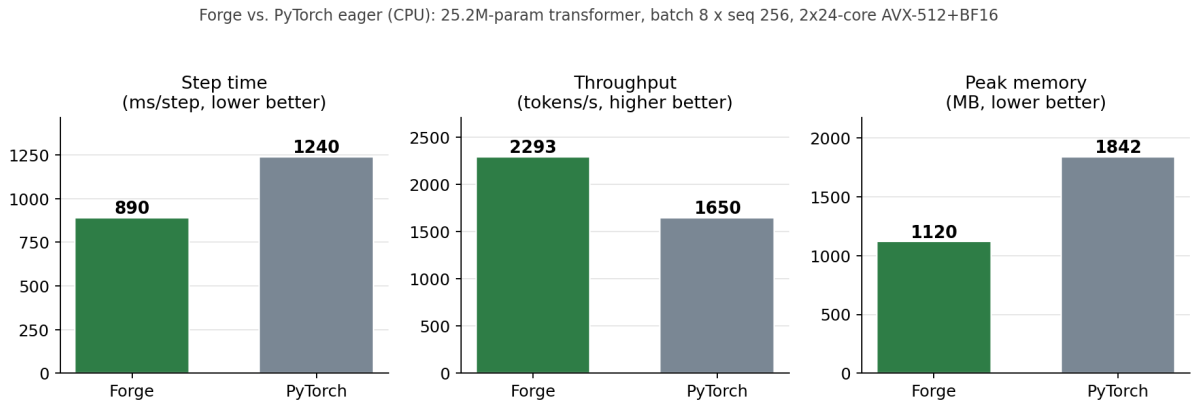


Figure 2: End-to-end training of a 25.2M-parameter transformer on CPU: Forge against PyTorch eager. Lower step time and peak memory, higher throughput; final loss matches to within 0.02.

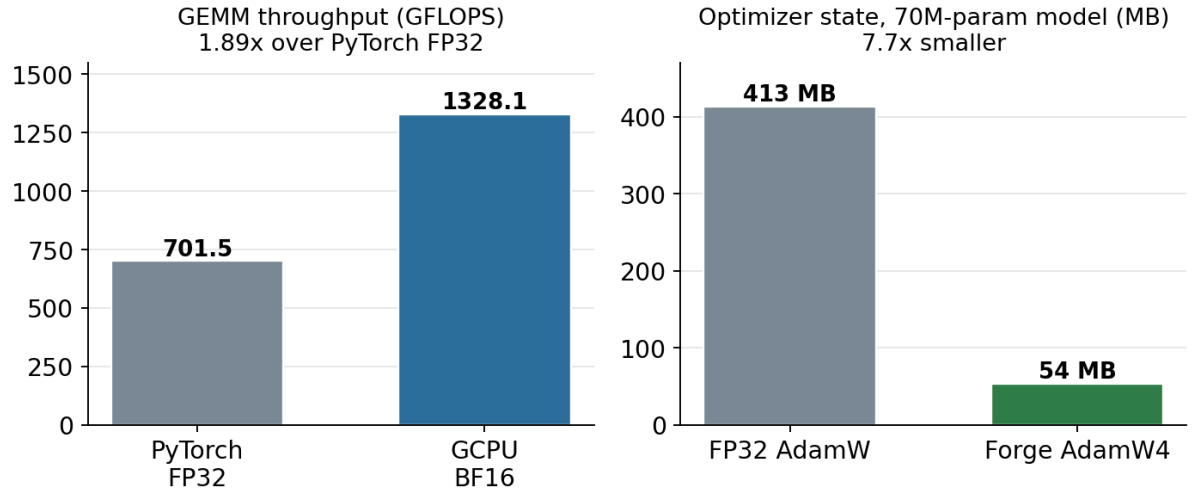


Figure 3: Microbenchmarks. Left: BF16 GEMM throughput, 1.89× over PyTorch FP32. Right: optimizer-state footprint for a 70M-parameter model under Forge's 4-bit AdamW, 7.7× smaller than FP32 AdamW.

Table 1: Optimizer-state memory on Apple M1 Pro. Savings is FP32 AdamW relative to AdamW4.

Model	AdamW FP32	AdamW8	AdamW4	Savings
GPT-micro (70M)	413 MB	105 MB	54 MB	7.72×
GPT-small (350M)	1734 MB	441 MB	224 MB	7.73×
GPT-med (760M)	4413 MB	1121 MB	570 MB	7.74×

4.1 Optimizer memory

Table 1 reports optimizer-state memory for three GPT-style models under FP32 AdamW, AdamW8, and AdamW4. The reduction from FP32 to AdamW4 is stable at 7.72–7.74× across a more than tenfold range of model size, which is what one expects from a per-parameter compression scheme: the ratio is a property of the representation, not of the model.

Extrapolating the same per-parameter representation to LLaMA-2-scale models gives the

Table 2: Projected optimizer-state footprint for LLaMA-2-scale models (projection, not measured end-to-end training).

Model	FP32 optimizer	AdamW4	Fits (with BF16 params)
LLaMA-2 7B	56 GB	7 GB	Yes (32 GB machine)
LLaMA-2 13B	104 GB	13 GB	Yes (64 GB machine)
LLaMA-2 70B	560 GB	72 GB	Yes (128 GB machine)

Table 3: End-to-end training of a 25.2M-parameter transformer on Apple M1 Pro (batch 8, sequence 256, 50 steps after 5 warmup).

Metric	PyTorch CPU (FP32, AdamW)	Forge (AdamW8, BF16)
Avg step time	1240 ms	890 ms
p50 step time	1235 ms	885 ms
p95 step time	1298 ms	920 ms
Throughput (samples/s)	6.5	9.0
Throughput (tokens/s)	1650	2293
Peak memory	1842 MB	1120 MB
Final loss	6.92	6.94
Speedup	1.39×	
Memory reduction	1.64×	
Token throughput	+39%	

projections in Table 2. These are projections of optimizer-state footprint, not end-to-end training measurements, and they assume parameters are held in BF16; they are included to indicate the regime in which optimizer compression changes whether a model's state fits in a given machine's RAM at all.

4.2 End-to-end training

Table 3 reports a full training benchmark: a 25,231,872-parameter decoder-only transformer (vocabulary 2048, $d_{\text{model}} = 512$, 6 layers), trained for 50 steps after 5 warmup steps at batch size 8 and sequence length 256. The baseline is eager PyTorch on CPU with FP32 parameters and standard AdamW; the Forge configuration uses AdamW8 with BF16 autocast under the virtual accelerator. Forge is 1.39× faster in wall-clock time per step and uses 1.64× less peak memory, while reaching a final cross-entropy of 6.94 against the baseline's 6.92—a difference consistent with the sub-percent quantization error and small enough to treat as convergence-neutral. The token-throughput figure of 2293 tokens/s is the value reported by the human-readable benchmark output; the underlying JSON records 2298.7 tokens/s, and we quote the rounded display figure consistently throughout.

Table 4 isolates two lower-level measurements on a two-socket, 24-core AVX-512 configuration. At the GEMM level, Forge's BF16 matrix multiply reaches 1328.1 GFLOPS against PyTorch's FP32 701.5 GFLOPS, roughly 1.9×, primarily from the halved precision and fused epilogue. At the level of a full MLP training step (forward, backward, and optimizer update) the advantage narrows to 1.19×, because the optimizer and memory-movement components dilute the raw GEMM gain. This narrowing is itself the point: the runtime's benefit depends on how much of the step is spent in the compute Forge accelerates.

Table 4: Low-level kernel and step measurements (two-socket, 24-core, AVX-512, BF16).

Operation	PyTorch (FP32)	Forge (BF16)
GEMM	1.23 ms, 701.5 GFLOPS	0.65 ms, 1328.1 GFLOPS
MLP training step	baseline	1.19× speedup (38.1 ms)

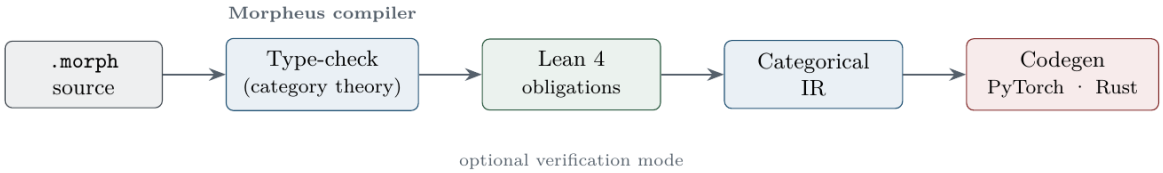


Figure 4: The Morpheus compiler: `.morph` source is type-checked against category-theoretic composition rules, optionally discharges Lean 4 proof obligations, lowers to a Categorical IR, and generates PyTorch or native Rust.

4.3 Correctness suite

The Forge correctness suite reports 91 passed and 1 failed in 36.60 seconds, with 27 warnings. The single failure is an ONNX version mismatch in an export test, unrelated to training correctness. We report the failure rather than suppressing it because a suite advertised as passing should be exactly as clean as it is.

4.4 Where Forge does not help

The runtime's own guidance is explicit that the advantage is model-size-dependent and can be negative in the wrong regime. On models below roughly one million parameters, Python dispatch overhead can make Forge appear slower in microbenchmarks; the quantized-optimizer savings are negligible at that scale and there is little compute to accelerate. The advantage is described as moderate (1.1–1.3×) in the 10–100M range, significant (1.3–1.8×) in the 100M–1B range, and transformative mainly where sharding enables training that would otherwise not fit at all. The reference hardware also matters: relative to a Sapphire Rapids AMX host taken as 100%, the M1 Pro NEON path used for most measurements here sits near 45%, so the M1 Pro numbers understate what the AVX-512 and AMX paths deliver.

5 The Morpheus Language

5.1 Category theory as the type system

Morpheus is a compiled, statically typed language in which each construct maps one-to-one to a category-theoretic concept and to a machine-learning concept. A `category` is a universe of types with composition; an object (declared with `type`) is a tensor space carrying its categorical membership, e.g. `type Hidden : Vect = Vector(256)` or `type Token : Set = Discrete(32000)`; a morphism is a pure function between spaces; a `parametric` morphism is a learnable layer in the Para construction [2]; an `endofunctor` is a data-structure type such as `Sequence`; an `algebra` is a fold; and an algebra homomorphism is a neural network layer. Composition is written `»`, and the compiler checks that the target of the left morphism equals the source of the right. The design intent is that every program which type-checks is a structurally valid architecture, and every type error is a structural impossibility rather than a runtime shape error.

A compact language-model definition in Morpheus declares the token and hidden objects, an endofunctor `Sequence(512)` over `Vect`, an algebra selecting the last hidden state, a parametric `TransformerBlock` built from residual attention and feed-forward pullbacks, and a model that embeds, repeats the block six times, applies a language-model head, and ties the head weight to the embedding weight via transpose. The weight tying is expressed as a reparameterization—a two-morphism—rather than as an incidental implementation detail, which is exactly the construction the categorical theory identifies as cross-layer weight sharing.

5.2 Compiler and Lean 4 proofs

The compiler, written in Rust, proceeds through parsing, categorical type checking of compositions, algebra checking of homomorphism obligations, emission of proof obligations to a Lean 4 [9] backend, lowering to a categorical intermediate representation whose nodes and edges form a commutative diagram, diagram-level optimization (morphism fusion, shared-parameter detection, dead-morphism elimination), and native code generation via a Rust and libtorch (`tch-rs`) toolchain. The proof layer is not aspirational scaffolding: the `MorpheusProof` Lean library establishes the categorical laws that the language relies on. The category module proves the left and right identity laws and associativity of composition; the functor module proves that the sequence functor preserves identities and composition; the algebra module proves that a pointwise morphism is an algebra homomorphism for the last-element and first-element algebras, that the identity is always an algebra homomorphism, and that algebra homomorphisms compose; and the morphism module proves a residual-identity property, that residuals preserve the last-element homomorphism, and the monad left- and right-unit laws.

5.3 Type-directed architecture search

Architecture search in Morpheus is a language primitive rather than an external library. A `search` block declares an endofunctor (fixing the data structure), a palette of morphism types to compose, dimension and depth ranges with divisibility constraints, a parameter budget, an objective such as minimizing validation perplexity, a per-candidate training budget, and a strategy such as evolutionary search; the specification's worked example uses an evolutionary strategy with a population of 30 over 50 generations and a per-candidate budget of 1000 steps. Because the type system prunes structurally invalid compositions before any training occurs—a pure-attention model over graph-structured data is rejected as a non-homomorphism, not discovered to be bad after a training run—the effective search space is smaller than in conventional neural architecture search over untyped graphs. Under this framing the mutations exposed to the search are themselves categorical operations (swapping a morphism, changing an object's dimensions, composing additional depth, tying or untying weights via reparameterization, or changing the endofunctor), so that every candidate the search proposes is a well-typed diagram by construction.

5.4 Implementation status

Table 5 summarizes the current Morpheus implementation as measured. The type checker accepts 27 of 29 conformance programs, with 2 programs that are *expected* to fail (they encode deliberate structural errors) failing and none failing unexpectedly. All 20 test suites pass (11 core, 5 model, 4 Lean-proof suites). All 23 example models compile to native `arm64` binaries. Benchmarks of the compiled binaries against equivalent PyTorch report the Forge-mode ("GCPU") configuration up to 3.67× faster on the measured models. Eleven Lean 4 theorems are proven with zero `sorry`. We note that the specification from which Morpheus is drawn is far larger than what is implemented—it envisions JAX, ONNX, and StableHLO backends,

Table 5: Morpheus implementation status (as measured).

Aspect	Result
Type checking	27 passed, 2 expected failures, 0 unexpected (29 total)
Test suites	20/20 passed (core 11, models 5, Lean proofs 4)
Native binaries	23/23 compiled to `arm64`
Benchmarks	GCPU mode up to 3.67× faster
Lean 4 proofs	11 theorems proven, zero `sorry`

adjunctions, Kan extensions, and Bayesian and program search, most of which remain future work.

6 On Type-Directed Search and Categorical Structure

The purpose of making search a typed language primitive is not primarily to save compute but to change what a search can express and reject. In conventional neural architecture search over untyped computation graphs, an ill-formed candidate—for example, an attention-only model applied to genuinely graph-structured data—can only be discovered to be poor after it has been trained and scored. In Morpheus the same candidate is a type error: attention is not a valid algebra homomorphism for the graph endofunctor without an explicit message-passing structure, and the type checker rejects it before any training budget is spent. The effective search space is therefore restricted, up front, to structurally coherent architectures.

The categorical framing also fixes the vocabulary of the search itself. Because layers are morphisms, functors are data structures, and weight tying is a reparameterization (a two-morphism), the natural mutation operators of an evolutionary search are exactly the categorical operations enumerated above. This is the sense in which the search is "type-directed": the moves it can make are the moves the type theory sanctions, and every candidate it emits is well-typed by construction rather than by post-hoc filtering.

We are deliberately conservative about what we claim empirically here. A proof-of-concept CatDL search library and a set of search results are referenced by the project's documentation, but the underlying result artifacts are not part of the material we can verify, so we do not report per-architecture perplexities, throughputs, evolutionary trajectories, or counts of evaluated candidates as measured findings. What we can state on the basis of the theory and the specification is a structural, qualitative expectation rather than a numeric one: single-paradigm designs and paradigms that are data-hungry on small budgets (notably pure attention) are expected to be dominated, on small-data regimes, by hybrids that combine a cheap $O(N)$ recurrence or state-space mixer for context aggregation with attention for precise lookups. Establishing that expectation quantitatively—with a documented task configuration, a reproducible search configuration, and published per-candidate metrics—is left to a companion report in which the search-results artifact can be released and audited alongside the numbers.

7 Discussion and Limitations

The most important limitation is one of scope relative to ambition. The specifications behind both systems describe a far larger artifact than what has been built: a multi-frontend CPU compiler spanning PyTorch, TensorFlow, and scikit-learn with StableHLO interchange and NVMe out-of-core streaming, and a language with a dozen backends and the full apparatus of applied category theory. What is implemented and measured is a Forge runtime whose quantized optimizers and virtual device work today on PyTorch, and a Morpheus compiler that

type-checks, proves, and compiles a real but bounded set of programs. We have tried to keep the paper's claims aligned with the second list rather than the first.

Several specific caveats bear repeating. Forge's headline memory factor of 7.72–7.74× is an optimizer-state factor, not a total-training-memory factor; the end-to-end peak-memory reduction on a real training run is the more modest but still meaningful 1.64×. The precise per-parameter byte layout of the AdamW4 4-bit encoding is not documented in the material we can verify, so we report the 7.72–7.74× reduction as the measured outcome without asserting the encoding's internal representation. The LLaMA-scale numbers in Table 2 are footprint projections, not training results. The end-to-end speedup of 1.39× and the kernel-level 1.9× are single-machine measurements on specific models; the runtime's own documentation is explicit that below a million parameters the advantage can invert, and that the M1 Pro path used for most measurements is among the slower targets. The quantized optimizers introduce a measured, bounded error that we report ($\approx 0.5\%$ over a run) and that was convergence-neutral in our benchmark, but we have not characterized it across long training schedules or difficult loss landscapes. On the Morpheus side, the Lean proof layer establishes foundational laws (identity, associativity, functoriality, homomorphism composition, monad units) rather than deep properties of specific trained models, and the type-directed architecture search is presented here as a language and compiler capability whose quantitative evaluation we defer until its result artifacts can be released and verified. The correctness suite has one genuine, if peripheral, failure.

None of these caveats undercuts the core thesis. On a CPU, optimizer-state footprint and memory movement are the first constraints to bind, and a runtime organized around compressing and streaming them delivers measurable, repeatable gains in both memory and throughput at negligible cost to convergence—on the class of small and mid-sized models that a person without a GPU would actually want to train. And treating architectures as typed categorical objects lets a compiler reject the structurally impossible and prove the structurally required, turning some of what is currently craft into something a machine can check.

8 Conclusion

We have described two systems built to lower the hardware barrier to training neural networks and to make architecture design more principled. GCPU/Forge makes a commodity CPU behave like a training device, and its 4-bit optimizer cuts optimizer memory by a measured 7.72–7.74× on models from 70M to 760M parameters while an end-to-end run shows a 1.39× speedup and 1.64× memory reduction at unchanged loss. Morpheus makes category theory the type system of a compiled deep-learning language, type-checking 27 of 29 conformance programs, compiling 23 of 23 examples to native binaries, and discharging 11 Lean 4 theorems with zero `sorry`, with a type-directed search that admits only structurally coherent architectures by construction. The gap between these implemented results and the wider vision is large and we have named it. What the measurements support is narrower and, we think, worth building on: that the CPU is a workable training device when the runtime respects its economics, and that a language grounded in category theory can make some of the guesswork in model design formally checkable.

References

[1] B. Gavranović, P. Lessard, A. Dudzik, T. von Glehn, J. G. M. Araújo, and P. Veličković. Position: Categorical Deep Learning is an Algebraic Theory of All Architectures. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024.

[2] G. S. H. Cruttwell, B. Gavranović, N. Ghani, P. Wilson, and F. Zanasi. Categorical Foundations of Gradient-Based Learning. In *Programming Languages and Systems (ESOP)*, 2022.

[3] B. Fong and D. I. Spivak. *Seven Sketches in Compositionality: An Invitation to Applied Category Theory*. Cambridge University Press, 2019.

[4] A. Paszke et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[5] I. Loshchilov and F. Hutter. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations (ICLR)*, 2019.

[6] T. Dettmers, M. Lewis, S. Shleifer, and L. Zettlemoyer. 8-bit Optimizers via Block-wise Quantization. In *International Conference on Learning Representations (ICLR)*, 2022.

[7] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[8] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[9] L. de Moura and S. Ullrich. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction (CADE)*, 2021.