VibeTensor: System Software for Deep Learning, Fully Generated by AI Agents

Bing Xu, Terry Chen, Fengzhe Zhou, Tianqi Chen, Yangqing Jia, Vinod Grover, Haicheng Wu, Wei Liu, Craig Wittenbrink, Wen-mei Hwu, Roger Bringmann, Ming-Yu Liu, Luis Ceze, Michael Lightstone, Humphrey Shi
NVIDIA

Abstract

VIBETENSOR is an open-source research system software stack for deep learning, generated by LLM-powered coding agents under high-level human guidance. In this paper, "fully generated" refers to code provenance: implementation changes were produced and applied as agent-proposed diffs; validation relied on builds, tests, and differential checks executed by the agent workflow, without per-change manual diff review. It implements a PyTorch-style eager tensor library with a C++20 core (CPU+CUDA), a torch-like Python overlay via nanobind [1], and an experimental Node.js/TypeScript interface. Unlike thin bindings, VIBETENSOR includes its own tensor/storage system, schema-lite dispatcher, reverse-mode autograd engine, CUDA runtime (streams/events/graphs [2]), a stream-ordered caching allocator with diagnostics, and a stable C ABI for dynamically loaded operator plugins. We view this open-source release as a milestone for AI-assisted software engineering: it demonstrates that coding agents can generate a coherent deep learning runtime spanning language bindings down to CUDA memory management, with validation constrained by builds and tests. We describe the architecture, summarize the workflow used to produce and validate the system, and evaluate the resulting artifact. We report repository scale and test-suite composition, and summarize reproducible microbenchmarks from an accompanying AI-generated kernel suite, including fused attention measured against PyTorch SDPA/FlashAttention [3]. We also report end-to-end training sanity checks on three small workloads (sequence reversal, CIFAR-10 ViT, and a miniGPT-style model) on NVIDIA H100 (Hopper, SM90) and Blackwell-class GPUs; multi-GPU results are Blackwell-only and rely on an optional CUTLASS-based ring-allreduce plugin gated on CUDA 13+ and `sm103a` toolchain support. Finally, we discuss failure modes that arise in generated system software—in particular a "Frankenstein" composition effect where locally correct subsystems can interact to produce globally suboptimal performance. We open-source the resulting system software and evaluation artifacts at `https://github.com/NVLabs/vibetensor`.

1 Introduction

Deep learning systems such as TensorFlow [4] and PyTorch [5] have matured into large software stacks spanning language frontends, runtime systems, device libraries, compilers, and kernels. Building and evolving such systems has historically required sizable teams and long time horizons. At the same time, recent progress in large language models (LLMs) trained on code [6] and tool-using agents [7] has enabled workflows where models propose code changes and validate them using external tools (search, builds, and tests).

Preprint. Under review.

We ask a concrete systems question: *can coding agents generate a coherent deep learning system software stack that crosses the usual abstraction boundaries—from Python and JavaScript APIs down to C++ runtime components and CUDA memory management—and validate it through builds and tests?* We open-source VIBETENSOR under the Apache 2.0 license as a technical artifact for answering this question empirically. We do not introduce a new agent framework here; instead, we treat agents as black boxes and focus on the released system software and the validation scaffolding that constrains system-scale generation.

Platforms. The release targets Linux x86_64 and NVIDIA GPUs via CUDA; builds without CUDA are intentionally disabled. In this paper we report runs on NVIDIA H100 (Hopper, SM90) and Blackwell-class GPUs. Multi-GPU results are Blackwell-only and rely on the experimental Fabric subsystem together with an optional CUTLASS-based ring-allreduce plugin; the plugin is best-effort and gated on CUDA 13+ / `sm103a` toolchain support.

Scope. The goal of this paper is *not* to introduce a new training framework or to claim state-of-the-art end-to-end performance. Instead, we focus on (1) the architecture of the generated system, (2) the methodology used to produce and validate it, and (3) lessons for future "vibe-coded" system software.

Contributions.

- **End-to-end system software.** We describe a deep learning runtime with multi-language frontends (Python and Node.js) backed by a shared C++ dispatcher, tensor/storage implementation, autograd engine, and CUDA subsystem.
- **Interoperability and extension points.** We document DLPack interop [8], a stable C plugin ABI, and hooks for custom kernels authored in Triton [9] and CUDA template libraries such as CUTLASS [10].
- **AI-assisted development methodology.** We summarize a practical workflow for generating and validating system-scale software with agents, using builds, tests, differential checks, and multi-agent code review as guardrails.
- **Artifact evaluation.** We report repository scale and test composition, summarize reproducible microbenchmarks for an accompanying AI-generated kernel suite (including attention compared against PyTorch SDPA/FlashAttention), and report end-to-end training sanity checks (sequence reversal, CIFAR-10 ViT, and miniGPT) plus a Blackwell-only multi-GPU scaling benchmark.
- **Lessons and limitations.** We describe failure modes observed in generated system software, including a "Frankenstein" composition effect where correctness-first subsystems can structurally inhibit performance.

2 Background and related work

Eager execution and define-by-run. Define-by-run systems such as Chainer [11] and DyNet [12] established an imperative approach to building dynamic computation graphs. PyTorch [5] popularized this model with a Python-first interface and a performance-oriented C++ core. VIBETENSOR follows the same eager execution philosophy, but our emphasis is on the feasibility of generating a multi-layer stack (frontends, runtime, CUDA allocator/graphs) with coding agents.

Kernel authoring systems. Recent systems such as Triton [9] make custom kernel authoring accessible in a Python-native workflow. CUDA template libraries such as CUTLASS [10] provide a complementary approach for hand-optimized custom kernels. Throughout the paper, CUTLASS refers to the C++ template library; CuTeDSL refers to a separate DSL backend used in our accompanying kernel suite. VIBETENSOR includes integration hooks for both styles and ships a kernel suite that exercises these pathways.

AI-assisted software engineering. Code-focused LLMs [6] and tool-using agents [7] have enabled workflows where software artifacts are constructed by iteratively proposing edits and validating them via tools. Benchmarks such as SWE-bench [13] evaluate issue-resolution capability in real repositories. VIBETENSOR differs in that the artifact spans system-software layers down to CUDA

memory management, and the open-source release is intended as a substrate for studying how AI-assisted workflows behave at system scale.

3 Design principles

VIBETENSOR targets a pragmatic subset of the design space. The principles below shaped both the architecture and the evaluation criteria used during generation.

Eager execution with explicit control. VIBETENSOR follows the define-by-run model popularized by PyTorch [5]: operator calls execute immediately, and autograd traces a dynamic graph. This model is well matched to debugging, rapid iteration, and dynamic control flow.

End-to-end coherence. The system was generated to span the full path from user-facing APIs to GPU execution. We prioritize having *working* implementations of tensors, dispatch, autograd, and CUDA runtime mechanisms, even when incomplete or unoptimized.

Testability as a first-class constraint. In a generated codebase, tests serve as executable specifications and regression guards. VIBETENSOR includes C++ (CTest) and Python (pytest) test suites, as well as an import-only API-parity checker against PyTorch for a scoped manifest. The Python CUDA surface includes contract-tested runtime and allocator APIs (e.g., `memory_stats`, `memory_snapshot`, and per-process memory-fraction caps). We do not claim fully automated test synthesis; rather, the tests are included in the released artifact and can be inspected and rerun. In this project, tests were written in the same agent loop as the implementation: when failures were discovered, the workflow typically responded by adding a minimal regression test or contract check and rerunning the suite. This is still hand-written test code (not formal synthesis), and test authoring/maintenance remains a scalability cost as the API surface grows.

Interoperability and extensibility. We treat the runtime as a component in a larger ecosystem. VIBETENSOR supports zero-copy exchange via DLPack [8], provides a C ABI for plugins, and exposes Python-level override mechanisms for rapid experimentation.

Inspectability. Generated systems benefit from being observable. VIBETENSOR therefore includes allocator diagnostics implemented directly in its CUDA caching allocator—per-device statistics and segment/pool snapshot APIs exposed through the Python surface (e.g., `memory_stats` and `memory_snapshot`)—along with CUDA graph instrumentation and optional multi-GPU observability hooks (Section 4.7). These diagnostics are native to VIBETENSOR: the API is PyTorch-inspired, but the counters and snapshots reflect VIBETENSOR's allocator state rather than calling into PyTorch.

4 System overview

Figure 1 shows the macro architecture. VIBETENSOR is organized into frontends, a shared core runtime, a CUDA subsystem, and optional kernel and plugin layers.

4.1 Frontends: Python and Node.js

VIBETENSOR exposes a torch-like Python overlay (`vibetensor.torch`) implemented on top of a nanobind [1] extension module. The overlay provides tensor factories, an `ops` namespace that routes to the dispatcher, and CUDA utilities for streams, events, allocator stats, and CUDA graph capture.

VIBETENSOR also includes an experimental Node.js addon based on Node-API [14]. The JavaScript/TypeScript (JS/TS) overlay is async-first: heavy work is scheduled via `napi_async_work` to avoid blocking the Node event loop, and a process-wide inflight cap (`VBT_NODE_MAX_INFLIGHT_OPS`) bounds queued work. In the current prototype, JavaScript tensor objects are CPU-only; CUDA is exposed through explicit H2D/D2H and DLPack import/export.

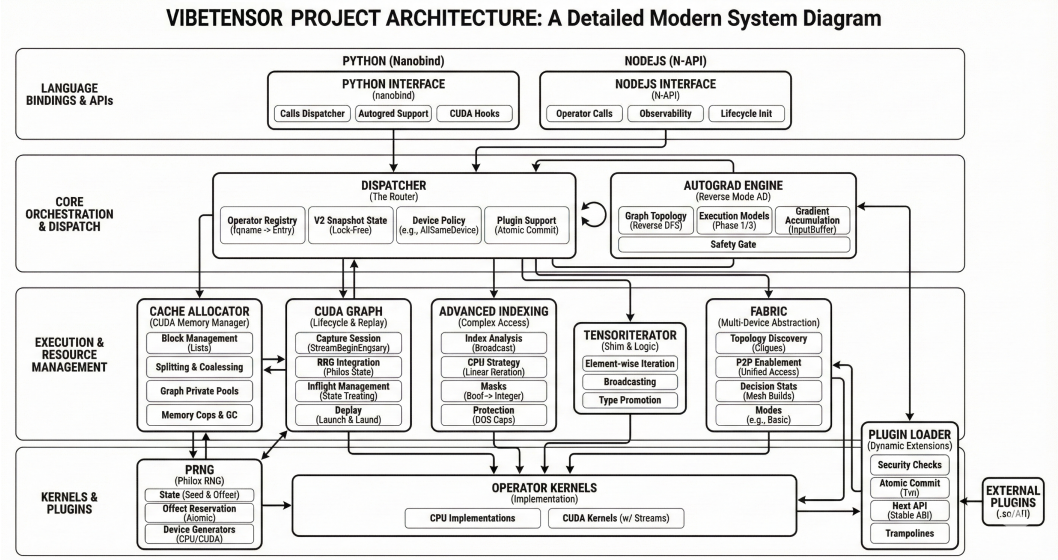


Figure 1: High-level architecture of VIBETENSOR: Python (nanobind) and Node.js (N-API) frontends dispatch into a shared C++ core implementing tensors/storage, dispatch, autograd, indexing, RNG, and CUDA runtime components (streams/events/graphs and caching allocator). Optional kernel libraries and dynamically loaded plugins extend the operator set.

4.2 Core runtime: tensors, storage, and views

The C++ core defines a `TensorImpl` as a view over reference-counted `Storage`, with sizes/strides, storage offset, dtype, and device metadata. This design supports non-contiguous views and as_strided semantics, and maintains an atomic version counter (shared by views) to support in-place safety checks.

For elementwise and reduction operators, VIBETENSOR provides a TensorIterator subsystem that computes iteration shapes and per-operand stride metadata. TensorIterator is also exposed through the plugin ABI so external kernels can reuse iteration logic safely.

4.3 Dispatcher: schema-lite operator registry

VIBETENSOR implements a schema-lite dispatcher that maps fully qualified operator names (e.g., `vt::add`) to kernel implementations. The dispatcher supports CPU and CUDA dispatch keys, boxed and unboxed call paths, and wrapper layers (e.g., autograd and Python overrides). Registration publishes immutable per-operator snapshot states that encode base kernels and wrapper presence, enabling lock-free call paths in steady state. Device-policy rules enforce common invariants (such as "all tensor inputs on the same device") while allowing specialized policies for experimental multi-device paths.

4.4 Reverse-mode autograd

Autograd is implemented as a reverse-mode engine with `Node`/`Edge` graph objects and per-tensor `AutogradMeta`. During backward, the engine maintains dependency counts, input buffers, and a ready queue. For CUDA tensors, the engine records and waits on CUDA events to synchronize cross-stream gradient flows. VIBETENSOR also contains an experimental multi-device mode intended for research on cross-device graph execution.

4.5 CUDA subsystem: streams, allocator, and graphs

VIBETENSOR provides C++ wrappers for CUDA streams and events, a caching allocator with stream-ordered semantics, and CUDA graph capture/replay support [2]. The allocator includes diagnostics (snapshots, statistics, fraction caps, and GC ladders) to make memory behavior observable during

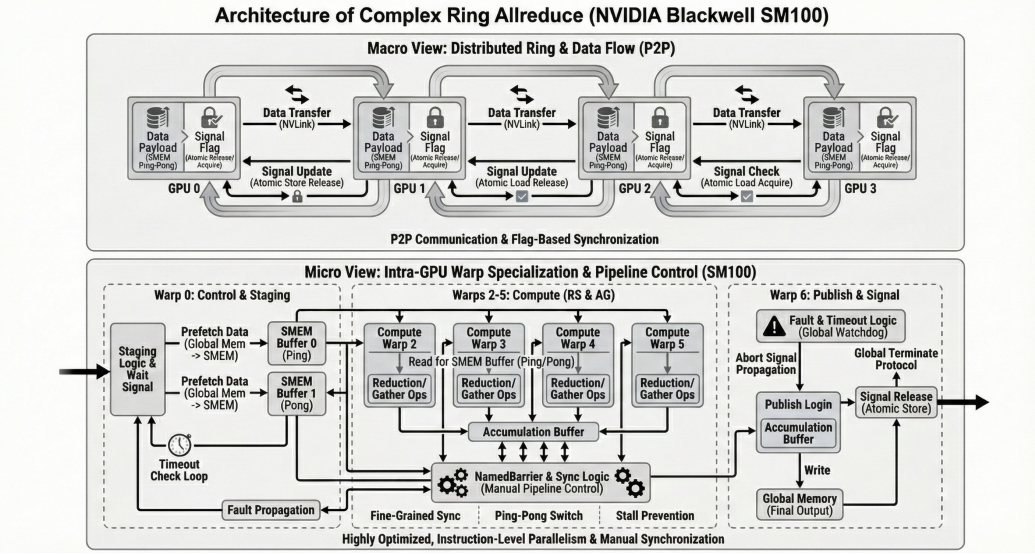


Figure 2: Macro and micro view of a warp-specialized ring allreduce kernel (Blackwell SM100/SM103) shipped as an example plugin. The plugin is best-effort, gated on toolchain and hardware availability, and is not required for using VIBETENSOR's core runtime; it is intended as a reference implementation rather than a replacement for NCCL.

testing and debugging. CUDA graphs integrate with allocator "graph pools" to manage memory lifetime across capture and replay.

4.6 Interop and extensions

VIBETENSOR supports interoperability via DLPack import/export [8] for both CPU and CUDA tensors. It also includes a C++20 safetensors loader/saver interface for efficient serialization [15].

For extensibility, VIBETENSOR offers: (i) Python-level overrides inspired by `torch.library`, (ii) a stable C ABI for dynamically loaded plugins that register new operators and kernels, and (iii) hooks for integrating custom GPU kernels authored in domain-specific systems such as Triton [9] or CUDA template libraries such as CUTLASS [10]. The C plugin ABI is versioned and exposes DLPack-based dtype/device metadata and TensorIterator helpers, allowing external kernels to reuse iteration logic and obey aliasing rules.

4.7 Fabric and experimental multi-GPU components

VIBETENSOR includes an experimental *Fabric* subsystem that exposes explicit peer-to-peer GPU access paths (CUDA P2P and UVA when supported by the device topology) and observability (stats and event snapshots). The intent is to enable research on single-process multi-GPU execution where the runtime can reason about topology, synchronization, and memory placement. In the current release, Fabric focuses on explicit elementwise operations and observability primitives rather than a full distributed training runtime.

As a reference example of hardware-specialized extensions, the release also includes a best-effort CUTLASS-based ring allreduce plugin targeting NVIDIA Blackwell-class GPUs; this is an illustrative plugin (not an NCCL-based drop-in replacement). The plugin directly binds CUTLASS experimental ring-allreduce kernels and does not call NCCL; for comparison, the plugin directory includes a standalone PyTorch `torch.distributed` (NCCL) bandwidth benchmark. Figure 2 illustrates the macro (inter-GPU ring) and micro (warp-specialized pipeline) view.

Table 1: Representative debugging episodes observed during agent-assisted development (selected from development notes).

Symptom	Root cause	Fix / regression guard
Kernel crash or segfault	Exceeded kernel parameter limits (e.g., too many explicit stride arguments)	Simplify call interface; add launch-configuration tests
Shared-memory compile failure	Requested > 48KB of statically allocated shared memory (toolchain cap)	Adaptive tiling / autotune; compile-time guards
Large numerical error vs reference	Incorrect stabilization math (e.g., log2 vs ln scaling in attention)	Use flash-attention-style formulation; differential tests vs PyTorch
Training loss divergence after many steps	Uninitialized GPU buffers reused during gradient accumulation	Initialize gradient buffers; add multi-step training regressions

5 AI-assisted development methodology

VIBETENSOR was developed with LLM-powered coding agents under high-level human guidance over approximately two months. Because the focus of this paper is system software rather than the agent implementation, we treat the agents as black boxes that propose code changes and use tools to validate them. Humans provided high-level requirements and priorities, but did not perform diff-level review or run validation commands; instead, agents executed builds, tests, and differential checks and retained changes when these checks passed.

5.1 Workflow and guardrails

The development workflow repeatedly applied a simple loop: (1) specify a scoped goal and invariants, (2) generate and apply code changes, (3) compile and run focused tests, and (4) broaden validation as subsystems composed.

Two guardrails were especially important: **tests as specifications** (CTest/pytest, plus targeted multi-step regression tests), and **differential checks** against reference implementations (e.g., PyTorch for selected operators, DLPack roundtrips, and kernel-vs-kernel comparisons). We also used multi-agent code review to catch missing cases, redundant abstractions, and unsafe patterns that a single agent might overlook.

5.2 Case study: debugging across layers

A recurring pattern in agent-generated system software is that "single-shot" correctness (an operator matches a reference once) does not imply stability when the subsystem is repeatedly composed inside a training loop. Development logs from one attention-kernel porting episode illustrate the range of issues that arose: launch-time limits (kernel parameter counts, shared-memory usage), numerical subtleties (log base and scaling in stabilized attention), and runtime hazards such as uninitialized GPU buffers whose reuse only manifests after several iterations.

Table 1 summarizes representative issues and the guardrails that prevented recurrence.

6 Evaluation

We evaluate VIBETENSOR along four axes: repository scale, correctness infrastructure, kernel-level performance of selected components, and end-to-end training sanity checks.

6.1 Repository scale

Table 2 summarizes the scale of the codebase in this release. Counts exclude vendor-provided third-party dependencies.

6.2 Correctness and reproducibility

VIBETENSOR includes C++ unit tests (run via CTest) and Python tests (run via pytest). The reference build targets Linux x86_64 and requires a CUDA toolchain; builds without CUDA are intentionally

Table 2: VIBETENSOR repository scale (excluding vendor-provided third-party code). "Core" refers to `include/` and `src/` in the C++ runtime.

Component	Source files	Non-blank LOC
Core C++/CUDA runtime	218	63,543
Plugins (C/CUDA)	50	17,500
Python overlay	33	9,016
Node.js/TS overlay	17	2,010
Tests (C++)	194	32,042
Tests (Python)	225	21,913
AI kernel suite (Python, Triton/CuTeDSL)	203	55,882

disabled in the current release. Tests cover core tensor semantics, dispatcher behavior, autograd, CUDA runtime utilities, allocator invariants, DLPack interop, CUDA graphs, and selected kernel paths. Allocator diagnostics are surfaced via the Python CUDA module (`vibetensor.torch.cuda`), including `memory_stats`, `memory_snapshot`, and per-process memory-fraction caps; the Python test suite includes contract checks for these APIs.

To reduce "it imports but it is broken" failures, VIBETENSOR also provides an import-only API-parity gate: a script that checks a scoped manifest of importable symbols against PyTorch. The manifest is intentionally limited and should be viewed as a regression guard rather than a claim of full API compatibility.

Reproducing checks. A typical validation sequence is:

- `python -m pip install -U pip build pytest numpy`
- `CMAKE_BUILD_TYPE=Debug python -m pip install -v -e .[test]`
- `ctest -test-dir build-py -j$(nproc) -output-on-failure`
- `pytest -q`
- `python tools/check_api_parity.py -manifest api/manifest_import_only.json`
- (optional) Node overlay: `cd js/vibetensor && npm ci && npm test`

6.3 Kernel suite and microbenchmarks

The accompanying AI-generated kernel suite provides Triton and CuTeDSL implementations for selected operators (e.g., norms, rotary embeddings, optimizers, and softmax/loss), along with benchmark harnesses that record performance and numerical diffs against PyTorch baselines. Most Triton and CuTeDSL kernels in this suite are exercised via a PyTorch-facing harness (PyTorch tensors and CUDA device discovery), which also serves as the reference for differential checks. Where available, we additionally provide VIBETENSOR-native wrappers (typically via DLPack) that can run without importing PyTorch; this pathway currently covers only selected kernels. We additionally include a dedicated fused-attention benchmark that compares a Triton kernel against PyTorch scaled dot-product attention (SDPA), which dispatches to FlashAttention when available. Figure 3 shows the macro structure of the kernel suite.

Table 3 reports selected microbenchmarks extracted from kernel-suite reports and the fused-attention benchmark. Because VIBETENSOR is async-first, all timings synchronize around the measured region to reflect device time rather than host dispatch latency.

Benchmark environment. Unless otherwise specified, the reported timings were collected on NVIDIA H100 PCIe (SM90) systems under CUDA 12–13 with PyTorch 2.9 nightly (to match the SDPA/FlashAttention baseline in this environment) and Triton 3.5.0. PyTorch is used here as a baseline and as a harness dependency for most Triton/CuTeDSL kernels; the VIBETENSOR core runtime does not depend on PyTorch. Each benchmark seeds RNGs, performs equal warmup for baseline and candidate, synchronizes around timed sections, and reports mean latency and numerical diffs (max absolute error and an allclose check).

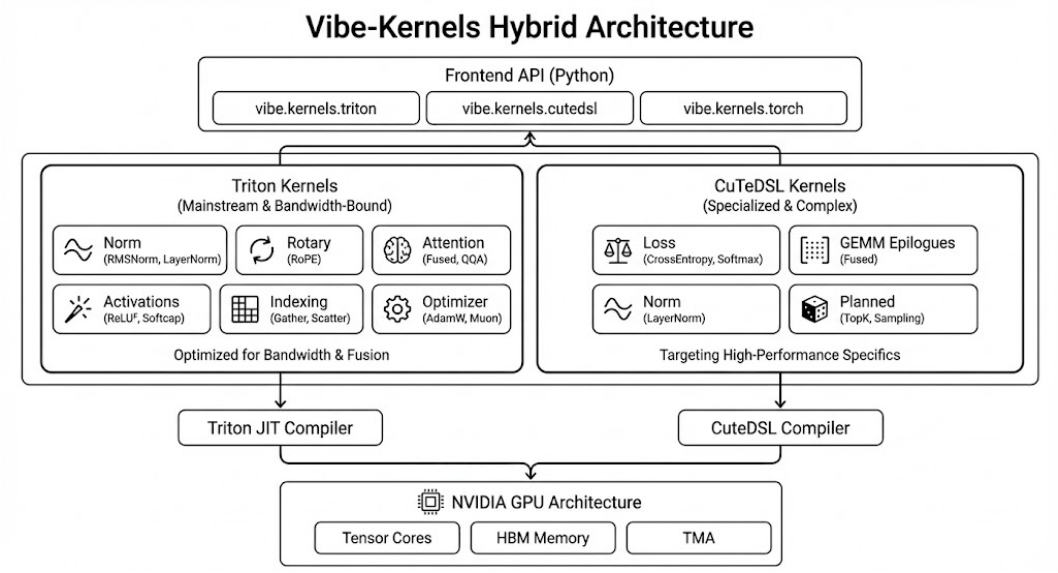


Figure 3: Macro view of the accompanying AI-generated kernel suite. Multiple backends (Triton, CuTeDSL, and PyTorch reference paths) share a common Python-facing interface and benchmark harnesses.

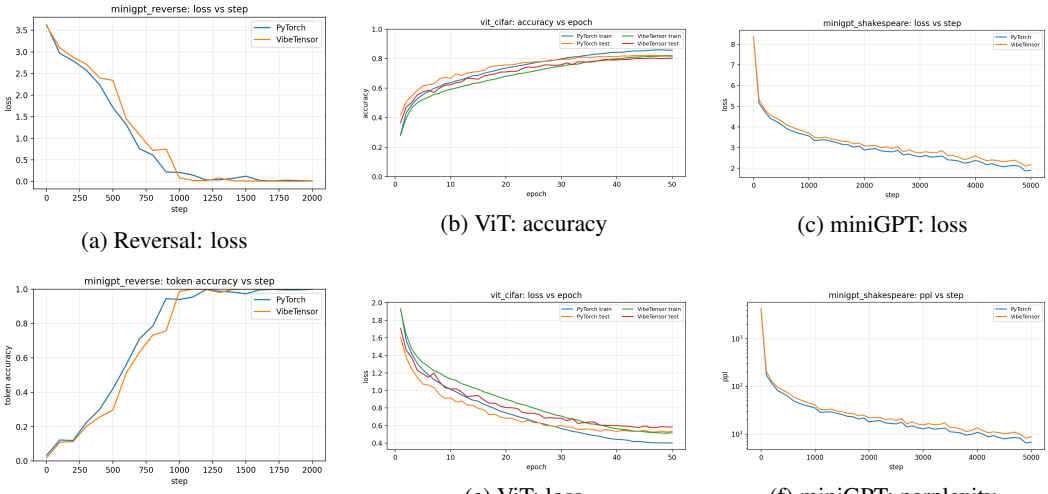
Table 3: Selected microbenchmarks from the accompanying kernel suite and the fused-attention benchmark on H100 PCIe (recorded in release artifacts). `Torch` refers to the PyTorch baselines used by the harnesses (e.g., `torch.nn.functional.layer_norm`, `torch.nn.functional.rms_norm`, and `torch.nn.functional.scaled_dot_product_attention` for attention/SDPA). `Best` reports the lowest measured latency among non-Torch backends (e.g., Triton, CuTeDSL) for that row. These results characterize isolated kernels, not end-to-end model performance.

Kernel	Shape/Dtype	Torch (ms)	Best (ms)	Speedup
LayerNorm (fwd+bwd)	(4096, 8192) (BF16)	0.47	0.45	1.06×
RMSNorm (fwd)	(4096, 8192) (BF16)	0.82	0.13	6.3×
Rotary (fwd)	(4, 8, 2048, 128) (BF16)	0.73	0.14	5.33×
Attention (fwd)	(32, 10, 10, 2048, 128) (BF16, causal)	2.24	1.46	1.54×
Attention (bwd)	(32, 10, 10, 2048, 128) (BF16, causal)	8.78	6.97	1.26×

Attention discussion. The attention benchmarks show a mixed picture. For a NanoChat-style training configuration (batch 32, sequence length 2048), the Triton kernel outperforms the PyTorch SDPA/FlashAttention baseline (1.54× forward, 1.26× backward). However, for small-batch GQA prefill workloads, the same pointer-based kernel can trail FlashAttention (0.67× forward, 0.66× backward). We view this as a representative example of how performance portability depends on kernel specialization and toolchain maturity (e.g., availability of Hopper warp-specialization knobs in Triton).

6.4 End-to-end training runs

Beyond unit tests and isolated microbenchmarks, we validated that VIBETENSOR composes into full training loops by running three small workloads: (i) a synthetic sequence-reversal task with a 2-layer decoder-only Transformer, (ii) a CIFAR-10 Vision Transformer based on Compact Vision Transformers [16], and (iii) a miniGPT-style language model [17] on Shakespeare. We ran these workloads on both NVIDIA H100 (Hopper) and Blackwell GPUs to sanity-check cross-architecture behavior. Figure 4 plots representative training curves from the Hopper H100 runs; Table 4 summarizes mean timing across both architectures. Across these workloads, VIBETENSOR is currently slower than PyTorch (1.7–6.2×), consistent with its prototype status and with known serialization points discussed in Section 7. These results are intended as end-to-end functional checks rather than performance claims.



(a) Reversal: loss (b) ViT: accuracy (c) miniGPT: loss

(d) Reversal: token accuracy (e) ViT: loss (f) miniGPT: perplexity

Figure 4: End-to-end training curves on Hopper H100 comparing VIBETENSOR and PyTorch across three workloads. In each case, VIBETENSOR matches the qualitative convergence behavior of the PyTorch baseline (loss decreases; accuracy/perplexity improve), indicating that core tensor semantics, autograd, and optimizer updates compose correctly at training-loop scale.

Table 4: End-to-end training workloads and mean timing (recorded in release artifacts). Iteration times exclude the first logged step where noted; epoch times are averaged over the compared span.

Workload	Span (GPU)	PyTorch	VIBETENSOR	Slowdown
Sequence reversal (2-layer Transformer)	2000 steps (Hopper H100)	3.958 ms/iter	12.02 ms/iter	3.04×
Sequence reversal (2-layer Transformer)	2000 steps (Blackwell)	7.25 ms/iter	12.48 ms/iter	1.72×
CIFAR-10 (ViT, depth 6)	50 epochs (Hopper H100)	6.544 s/epoch	37.67 s/epoch	5.76×
CIFAR-10 (ViT, depth 6)	14 epochs (Blackwell)	5.585 s/epoch	34.36 s/epoch	6.15×
Shakespeare (miniGPT, 6 layers)	5000 steps (Hopper H100)	13.94 ms/iter	80.62 ms/iter	5.79×
Shakespeare (miniGPT, 6 layers)	5000 steps (Blackwell)	18.63 ms/iter	74.81 ms/iter	4.01×

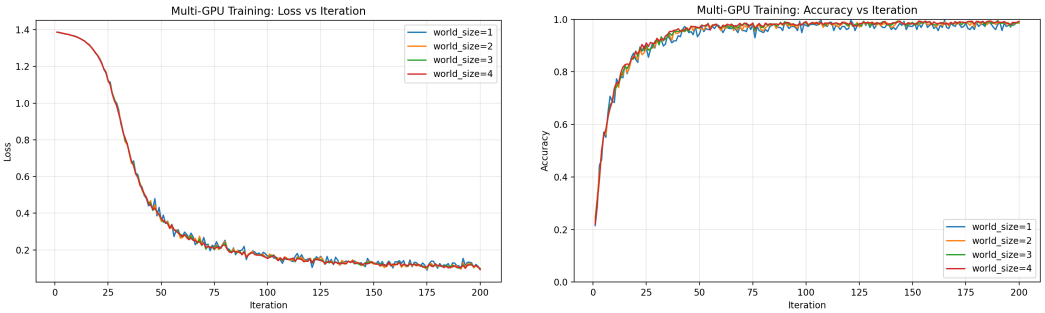
6.5 Multi-GPU scaling (Fabric + ring allreduce backend)

We also report a small multi-GPU benchmark on Blackwell GPUs using the experimental Fabric subsystem (Section 4.7) with a ring-allreduce backend (`fabric_ring`). Figure 5 shows representative training curves, and Table 5 shows throughput scaling to four GPUs under weak scaling (fixed per-GPU batch). Because this pathway depends on a Blackwell-only CUTLASS plugin, it was not run on the Hopper system. Fabric is not a full distributed training runtime; these results primarily indicate that cross-device synchronization and observability pathways execute end to end.

7 Limitations and lessons

VIBETENSOR is a research prototype and carries substantial limitations.

The "Frankenstein" composition effect. A recurring failure mode in generated systems is that individually reasonable components can compose into a globally suboptimal design. In VIBETENSOR, a correctness-first autograd and dispatch design can introduce serialization points that starve otherwise efficient backend kernels. One example is a non-reentrant global backward gate used to reject concurrent multithreaded backward runs (implemented as a process-wide try-locked mutex). It



(a) Loss (b) Accuracy

Figure 5: Multi-GPU training curves for the Blackwell scaling benchmark (world size 1–4).

Table 5: Multi-GPU training benchmark using VIBETENSOR Fabric on Blackwell GPUs (weak scaling).

world_size	per-GPU batch	avg iter (ms)	throughput (samples/s)
1	65536	29.88	2.19×10^6
2	65536	43.65	3.00×10^6
3	65536	60.36	3.26×10^6
4	65536	70.98	3.69×10^6

simplifies safety properties (e.g., avoids deadlock patterns from nested backward) but can serialize independent backward work. Figure 6 walks through this bottleneck: first a global gate in the engine funnels concurrent backward calls; then host-side serialization and synchronization propagate to the device; finally, otherwise efficient backend kernels observe reduced utilization due to starvation.

Incomplete API surface and performance. VIBETENSOR intentionally does not aim for full PyTorch compatibility. Many operators, datatypes, and distributed features are missing or incomplete, and performance has not been tuned to match production frameworks.

Validation gaps unique to generated code. Agent-generated code can pass local unit tests while failing under repeated composition (e.g., multi-step training loops) due to stateful interactions, uninitialized buffers, or accidental global synchronization. This motivates regression tests that exercise repeated execution, cross-stream behavior, and long-running workloads.

Maintenance, safety, and security. Machine-generated code can include inconsistent conventions, redundant abstractions, and subtle correctness or security issues. We therefore caution against production use and position VIBETENSOR primarily as a research and educational artifact.

8 Conclusion

VIBETENSOR demonstrates that AI-assisted workflows can generate a non-trivial, GPU-enabled deep learning system software stack spanning Python and Node.js frontends and a C++20/CUDA core, validated by builds and tests. The open-source release is intended to support reproducible study of system-scale software generation: what can be generated quickly, what kinds of bugs and bottlenecks arise, and what testing and architectural scaffolding most effectively constrain the search space.

From an ecosystem perspective, releasing such artifacts can benefit the broader GPU software community by providing concrete, hackable implementations of runtime and kernel components and by enabling rigorous discussion of AI-assisted engineering methods grounded in source code and tests.

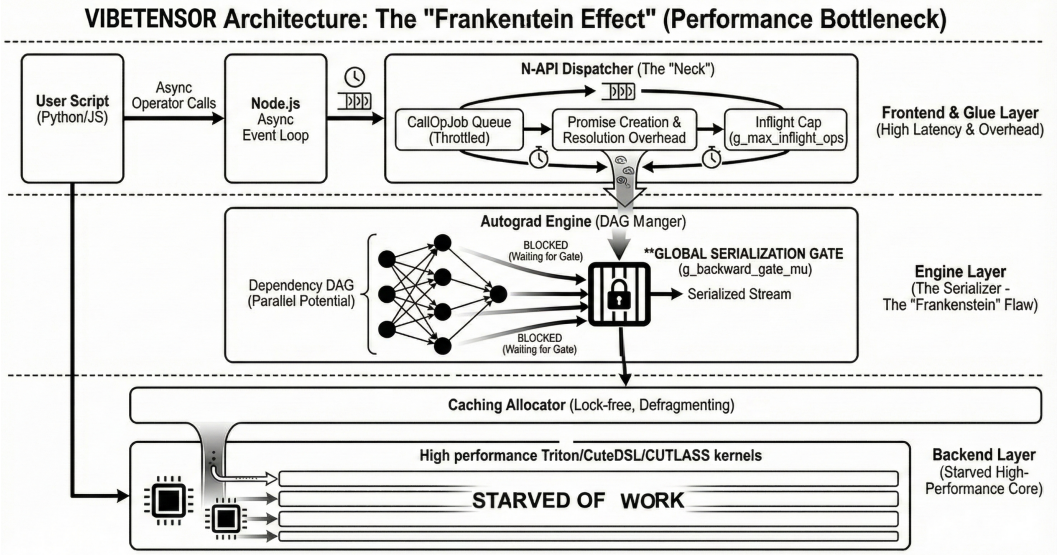


Figure 6: An example "Frankenstein" effect: a high-overhead frontend/engine layer can serialize execution and starve a high-performance backend. This reflects a limitation of correctness-first generation when global performance objectives are not encoded early.

Acknowledgments

We thank the maintainers of foundational open-source projects referenced or used by VIBETENSOR, including PyTorch, DLPack, nanobind, Triton, and CUTLASS. We also thank internal reviewers who provided feedback on the generated system and draft.

References

[1] Wenzel Jakob and nanobind contributors. nanobind: Small and efficient bindings for python. `https://github.com/wjakob/nanobind`, 2022–. Accessed 2026-01-08.

[2] NVIDIA. CUDA Graphs. `https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html`, 2020–. Accessed 2026-01-08.

[3] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems*, 2022. URL `https://arxiv.org/abs/2205.14135`.

[4] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems. *arXiv preprint arXiv:1605.08695*, 2016. URL `https://arxiv.org/abs/1605.08695`.

[5] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas K"opf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019. URL `https://arxiv.org/abs/1912.01703`.

[6] Mark Chen et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL `https://arxiv.org/abs/2107.03374`.

[7] Shunyu Yao et al. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022. URL `https://arxiv.org/abs/2210.03629`.

[8] DLPack contributors. DLPack: Open in-memory tensor structure for sharing tensors. `https://github.com/dmlc/dlpack`, 2017–. Accessed 2026-01-08.

[9] Triton contributors. Triton: A language and compiler for custom deep learning primitives. `https://github.com/triton-lang/triton`, 2019–. Accessed 2026-01-08.

[10] NVIDIA. CUTLASS: Cuda templates for linear algebra subroutines. https://github.com/NVIDIA/cutlass, 2017–. Accessed 2026-01-08.

[11] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015. URL http://learningsys.org/papers/LearningSys_2015_paper_33.pdf.

[12] Graham Neubig et al. Dynet: The dynamic neural network toolkit. *arXiv e-prints*, 2017. URL https://arxiv.org/abs/1701.03980.

[13] Carlos E. Jimenez et al. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023. URL https://arxiv.org/abs/2310.06770.

[14] Node.js contributors. Node-api. https://nodejs.org/api/n-api.html, 2018–. Accessed 2026-01-08.

[15] Hugging Face and safetensors contributors. Safetensors: Safe and fast tensor serialization. https://github.com/huggingface/safetensors, 2022–. Accessed 2026-01-08.

[16] Ali Hassani, Steven Walton, Nikhil Shah, Abulikemu Abuduweili, Jiachen Li, and Humphrey Shi. Escaping the big data paradigm with compact transformers. *arXiv preprint arXiv:2104.05704*, 2021. URL https://arxiv.org/abs/2104.05704.

[17] Andrej Karpathy. minGPT: A minimal implementation of a gpt. https://github.com/karpathy/minGPT, 2020–. Accessed 2026-01-13.