Comprehensive Technical Analysis of Loom/Poly Version 0.78: Architectural Paradigm, Assembly Acceleration, and Cross-Platform Sovereignty

Introduction to the M-POLY-VTD Engine

The contemporary landscape of artificial intelligence frameworks is heavily stratified, dominated by massive, monolithic runtime environments that couple high-level Python wrappers to proprietary C++ and CUDA binaries. This paradigm naturally restricts edge deployment, hampers cross-platform reproducibility, and forces reliance on cloud-compute infrastructure. Loom/Poly version 0.78—formally classified as the Multi-numerical Polymorphic Volumetric Tiled-tensor Dispatcher (M-POLY-VTD)—represents a radical divergence from this industry standard.[1] Engineered entirely in pure Go, WebGPU, and Plan 9 Assembly, Loom is designed as a foundational, bare-metal physics engine for edge devices.[1]

Colloquially conceptualized as the "SQLite of AI," Loom/Poly allows neural networks to be trained once and deployed anywhere—from bare-metal server racks to mobile smartphones and embedded WebAssembly browser contexts—without any reliance on cloud architecture.[1] This exhaustive technical report provides a deep architectural breakdown of Loom version 0.78. It dissects its three-dimensional volumetric grid system, its 21-datatype numerical polymorphism, and the implementation of its novel Plan 9 assembly acceleration paths.[1] Furthermore, the analysis evaluates the empirical benchmark data extracted from engine logs, determining precisely why and how optimized CPU assembly outclasses modern GPU compute architectures, and contextualizes Loom within the broader Golang machine learning ecosystem.[1]

The Volumetric Tensor Dispatch (VTD) Architecture

Traditional neural network frameworks execute computational graphs as Directed Acyclic Graphs (DAGs) representing sequential, one-dimensional stacks of layers. Loom completely abandons this linear paradigm in favor of a three-dimensional Volumetric Coordinate System.[1] The primary engine context is instantiated via the VolumetricNetwork struct, which serves as a macroscopic spatial grid.[1] Layers are not simply appended to an array; they are strictly localized within this Cartesian topology using four definitive coordinates: Depth (Z), Row (Y), Column (X), and Layer index per cell (L).[1] This 3D mapping facilitates biological plausibility within the network architecture by enabling complex spatial routing mechanisms that transcend linear execution.[1]

Spatial Hopping and Remote Links

Because layers exist in a volumetric grid, Loom supports arbitrary signal routing through spatial hopping.[1] A layer can be configured with an IsRemoteLink boolean flag. When this flag is asserted, the layer bypasses standard sequential data flow and routes its activations directly to the coordinates specified by TargetZ, TargetY, TargetX, and TargetL.[1] This allows researchers to trivially implement highly complex recurrent feedback loops, skip connections, and cross-depth structural alignments without writing custom forward-pass boilerplate.

Container Layers and Recursive Execution

To manage topological complexity, Loom provides structural container layers, primarily LayerParallel and LayerSequential, which allow for infinite nesting.[1] A LayerSequential container encapsulates multiple VolumetricLayer sub-layers, executing them in strict order and passing the output of one sub-layer as the input to the next.[1]

More significantly, the LayerParallel construct enables complex branching. An input tensor is broadcast simultaneously to all layers held within its ParallelBranches array. The outputs of these parallel branches are then unified based on the layer's defined CombineMode.[1] The engine natively supports several combination mathematics:

- **Concat / Grid Scatter:** Concatenates the output vectors from all branches into a single flattened tensor.
- **Add / Avg:** Performs element-wise addition or averaging across the branch outputs.
- **Filter (Gated MoE):** Implements a Mixture-of-Experts routing paradigm. If a FilterGateConfig layer is provided, the input is first passed through the gate to produce routing logits. A Softmax function is applied to these logits to generate coefficients, which are then used to perform a weighted sum of the parallel branch outputs.[1]

This recursive, containerized architecture permits the construction of exceedingly complex blocks—such as a complete Transformer decoder block (RMSNorm $\rightarrow$ MultiHeadAttention $\rightarrow$ RMSNorm $\rightarrow$ SwiGLU) or a Residual Graft—to be mathematically represented as a single coordinate node within the macroscopic VolumetricNetwork grid.[1]

Polymorphic Layer-Morphing and the 21-DType System

The core innovation denoted by the "POLY" in M-POLY-VTD is the engine's dynamic precision capability. Loom 0.78 implements native support for 21 distinct numerical data types, enabling individual network layers to transition their mathematical representations on the fly without requiring the entire network to be recast.[1]

The Complete Numerical Matrix

The dispatcher natively supports the following matrix of precision types:

Precision Category	Supported Data Types (DType)	Primary Application
High-Precision	Float64, Int64, Uint64	Scientific modeling, gradient checking, financial prediction.
Standard-Precision	Float32, Int32, Uint32, Int16, Uint16	Standard deep learning baseline execution and legacy model ingestion.
Optimized-Precision	Float16, BFloat16, Int8, Uint8	High-speed mobile deployment and conventional GPU acceleration.
Sub-Byte / Low-Bit	FP8-E4M3, FP8-E5M2, Int4, Uint4, FP4	Advanced quantization, striking a balance between perplexity and VRAM reduction.
Extreme Quantization	Int2, Uint2, Ternary (-1, 0, 1), Binary (1-bit)	Ultra-low power edge devices, severe memory constraints, and binary neural networks (BNNs).

The WeightStore and Dynamic Projection

This polymorphism is governed by the WeightStore component attached to every weighted VolumetricLayer.[1] To ensure deterministic consistency and prevent precision degradation during repeated quantization cycles, the WeightStore maintains a continuous, high-precision Float32 array called Master.[1] This array acts as the absolute source of truth for the layer's weights. During execution or compilation, the engine invokes the Morph(dtype) function. This function analyzes the target data type and dynamically projects the Master weights into the target precision, caching the result within a Versions or CPUPacked map.[1] For standard integer types, this involves scaling the float by a dynamically computed quantization scale factor and rounding to the nearest integer.[1]

Pure Go Sub-Byte Bit-Packing

To achieve its targeted 75% to 80% reduction in weight size, Loom heavily utilizes sub-byte bit-packing for its lowest precision types.[1] The bit-packing algorithms are implemented entirely in pure Go to maintain the cross-platform compilation guarantee.[1]

For FP4 (E2M1) quantization, the engine does not rely on hardware-specific floating-point extensions. Instead, it utilizes a hardcoded static lookup array, fp4DecodeTable, mapping the 16 possible 4-bit codes to their exact floating-point representations (e.g., 0.0, 0.75, 1.0, 1.5, 2.0, 3.0 and their negative counterparts).[1] The nearestFP4Code function iterates through these finite values to find the absolute minimum difference between the unscaled input float and the table reference, assigning the corresponding 4-bit nibble.[1]

For FP8 conversions, the engine uses pure mathematical bit-shifting on the IEEE 754 representations of Float32 values. For FP8-E4M3 (1 sign bit, 4 exponent bits, 3 mantissa bits,

bias 7), the float32ToE4M3 function extracts the sign, exponent, and mantissa, subtracts the standard 127 bias, adds the new 7 bias, and handles subnormal truncation natively.[1] This allows Loom to execute complex mathematical quantization deterministically on any CPU architecture without linking to external C libraries.

For 1-bit and 2-bit networks (BitNet architectures), the WeightStore implements specific caching for Microsoft's bitnet-b1.58 offline packed layouts. Weights can be directly injected into the CPUPacked map as byte arrays, where Ternary weights are packed 16 values per 32-bit word, completely bypassing the instantiation of a Float32 master array to conserve RAM on heavily constrained edge devices.[1]

Plan 9 Assembly Acceleration (poly/asm)

While Go's standard library provides exceptional concurrency and cross-platform compilation, its compiler intentionally limits low-level memory unsafe optimizations and does not explicitly guarantee optimal SIMD (Single Instruction, Multiple Data) vectorization across all hardware.[1] To bypass the computational ceiling inherent in pure Go math loops, Loom version 0.78 introduces the poly/asm subsystem.[1]

The engine leverages Go's native support for Plan 9 Assembly, writing architecture-specific .s files for amd64 and arm64 targets.[1] These assembly routines are bridged directly into the Go runtime using //go:noescape directives.[1] Because these routines do not rely on CGO (C-to-Go), they incur zero context-switching overhead, allowing the neural dispatcher to invoke them directly within the hot path of the matrix multiplication loops.

Floating-Point Vectorization

For standard precision types, the poly/asm/dot package provides manually unrolled dot-product loops.[1] Analysis of the f32_amd64.s kernel reveals that it processes blocks of four floating-point numbers simultaneously to maintain pipeline saturation. It issues MOVSS instructions to load data from memory into XMM registers, followed by MULSS for parallel multiplication, and ADDSS to accumulate the partial sums into an accumulator register (X0).[1] The arm64 equivalent (f32_arm64.s) performs the exact same unrolling pattern utilizing ARM Neon FPU registers (FMOVS, FMULS, FADDS).[1]

Native Integer Accumulation

The most critical optimization provided by the poly/asm layer is applied to low-bit and integer data types. When executing a layer configured for Int8, Int4, Ternary, or Binary, the engine routes the matrix multiplication through denseForwardAsmNative.[1]

In traditional high-level wrappers, quantized integer weights are often decoded back to Float32 or Float16 right before multiplication inside the GPU shader or CPU loop. Loom entirely eradicates floating-point operations from the inner loop for these types.[1] The assembly kernels execute integer-by-integer multiplication natively. For instance, the dotI8AccF64 kernel utilizes MOVBQSX to perform sign-extended memory loads of byte-sized weights, executes integer multiplication using the IMULQ instruction, and only converts the final accumulated sum to a

float via CVTSQ2SD at the very end of the tile block.[1]

Bitstream Packed Execution

For extreme quantization levels, the assembly engine executes directly on bitstreams without unpacking the bytes in memory.[1]

- **NibblePackedRowNativeI64:** Decodes 4-bit weights (Int4, Uint4, FP4) packed 8-per-uint32 by dynamically shifting the target word shift := uint((idx % 8) * 4), masking with 0x0F, and accumulating.[1]
- **TwoBitPackedRowNativeI64:** Decodes 2-bit weights (Int2, Uint2, Ternary) packed 16-per-uint32.[1]
- **BinaryPackedRowNativeI64:** Decodes 1-bit weights packed 32-per-uint32 using a single bitwise AND operation (word>>shift)&1 to determine the sign of the accumulation.[1]

This capability allows the CPU to fetch massive vectors of weights using minimal memory bandwidth. Because the CPU L1 and L2 caches can house significantly more weights when they are packed 32-per-word, cache misses plummet to near zero, heavily accelerating inference speed on standard mobile processors.[1]

Exhaustive Benchmark Analysis: The CPU vs. GPU Phenomenon

A highly rigorous empirical analysis of Loom's execution speeds is documented within logs.txt, detailing the performance of generic layer suites across all 21 numerical types. The benchmarks compare pure Go single-core (SC) and multi-core (MC) routines, Plan 9 Assembly SC and MC routines, and WebGPU-accelerated Metal execution.[1] The host environment for the GPU analysis was an Apple Silicon Metal backend (Adapter: 0x0 [metal]) with shader-f16 features enabled.[1]

The data exposes a remarkable architectural phenomenon: under a wide array of precision configurations, highly optimized Multi-Core Assembly on the CPU vastly outclasses the compute throughput of the GPU.[1]

CNN1 GPU Forward Pass Latency

The table below synthesizes the Forward pass latencies for the CNN1 layer suite, highlighting the performance delta between CPU execution and GPU execution.

Data Type	Tile Size	CPU MC Latency	GPU MC Latency	Speedup (GPU vs CPU)	Base Discrepancy
Float64	8	113.667 µs	1.368917 ms	0.08x (CPU Faster)	$0.00e+00$
Float32	32	127.167 µs	1.346292 ms	0.09x (CPU Faster)	$0.00e+00$
Float16	32	136.167 µs	1.367084 ms	0.10x (CPU Faster)	$0.00e+00$

BFloat16	32	104.833 µs	1.347375 ms	0.08x (CPU Faster)	$0.00e+00$
FP8-E4M3	32	76.75 µs	1.350625 ms	0.06x (CPU Faster)	$0.00e+00$
FP8-E5M2	32	101.583 µs	1.353083 ms	0.08x (CPU Faster)	$0.00e+00$
Int64	32	92.25 µs	1.347709 ms	0.07x (CPU Faster)	$0.00e+00$
Uint64	32	106.375 µs	1.333458 ms	0.08x (CPU Faster)	$0.00e+00$
Int32	32	803.291 µs	1.341584 ms	0.60x (CPU Faster)	$0.00e+00$
Uint32	32	107.75 µs	1.342334 ms	0.08x (CPU Faster)	$0.00e+00$
Int16	32	93.792 µs	1.350916 ms	0.07x (CPU Faster)	$0.00e+00$
Uint16	32	88.375 µs	1.334542 ms	0.07x (CPU Faster)	$0.00e+00$
Int8	32	113.709 µs	1.345916 ms	0.08x (CPU Faster)	$0.00e+00$
Uint8	32	91.916 µs	1.385042 ms	0.07x (CPU Faster)	$0.00e+00$
Int4	32	112.209 µs	1.335875 ms	0.08x (CPU Faster)	$0.00e+00$
Uint4	32	117.666 µs	1.364375 ms	0.09x (CPU Faster)	$0.00e+00$
FP4	32	87.708 µs	1.345916 ms	0.07x (CPU Faster)	$0.00e+00$
Int2	32	98.584 µs	1.3725 ms	0.07x (CPU Faster)	$0.00e+00$
Uint2	32	103.167 µs	1.342917 ms	0.08x (CPU Faster)	$0.00e+00$
Ternary	32	96.458 µs	1.340292 ms	0.07x (CPU Faster)	$0.00e+00$
Binary	32	124.334 µs	1.354667 ms	0.09x (CPU Faster)	$0.00e+00$

As illustrated by the data, across all 21 numerical types, CPU multi-core execution was invariably faster than GPU execution for these specific layer dimensions, often achieving compute times an order of magnitude lower (e.g., FP8-E4M3 executing in 76.75 µs on CPU vs 1.35 ms on GPU).[1] The Int32 configuration presented the most competitive GPU timing, yet the

CPU remained dominant with a 0.60x speed ratio constraint.[1]

Plan 9 Assembly Dominance over GPU Multi-Core

The superiority of the CPU path becomes explicitly evident when transitioning to the Assembly (poly/asm) backend evaluated in the Dense Forward suite.[1] A direct comparison between Multi-Core Assembly (ASM MC) and Multi-Core GPU (GPU MC) illustrates profound speedups.

Precision Category	Data Type	Assembly MC Latency	GPU MC Latency	Assembly Speedup vs GPU
Low-Bit Quantized	Uint4	349.25 µs	1.340333 ms	**3.84x**
	FP4	350.375 µs	1.3155 ms	**3.75x**
	Int4	419.417 µs	1.433417 ms	**3.42x**
	Int2	421.292 µs	1.331125 ms	**3.16x**
	Uint2	453.5 µs	1.322583 ms	**~2.92x**
Mid-Bit Integer	Uint16	370.958 µs	1.325167 ms	**3.57x**
	Int16	394.25 µs	1.314083 ms	**3.33x**
	Uint8	411.083 µs	1.318041 ms	**3.21x**
	Int8	441.625 µs	1.347625 ms	**3.05x**
High-Bit Integer	Uint32	514.958 µs	1.336167 ms	**2.59x**
	Int32	539.458 µs	1.329291 ms	**2.46x**
	Int64	678.166 µs	1.332875 ms	**1.97x**
	Uint64	776.584 µs	1.314416 ms	**1.69x**
Floating-Point	Float64	1.059084 ms	1.694583 ms	**1.60x**
	Float32	911.583 µs	1.356333 ms	**1.49x**
	BFloat16	961.042 µs	1.349834 ms	**1.40x**

This data forces a re-evaluation of standard hardware assumptions. For heavily compressed, sub-byte models (Uint4, FP4, Int4), the Assembly engine operates nearly four times faster than the GPU.[1]

The Physics of Dispatch Latency and Cache Locality

The architectural reasoning behind this CPU dominance is grounded in the disparity between PCIe memory bus latency and L1/L2 cache locality. Submitting a compute payload to the GPU requires invoking the WebGPU graphics API, allocating buffers, copying memory across the bus, and submitting command encoders. This fixed dispatch latency typically consumes between 1.0ms to 1.5ms.[1]

When a neural network is compressed into 4-bit or 1-bit formats, the total computational volume plummets.[1] Because the memory footprint of the weight matrix is so drastically reduced, the CPU can fit the entire layer's parameters comfortably inside its ultra-fast L2 or L3 cache. The Plan 9 assembly routines iterate over this localized data using native integer multiplication

(IMULQ), bypassing main system RAM bottlenecks entirely.[1] The GPU, conversely, spends the vast majority of its execution time simply waiting for the API to schedule the kernel, failing to saturate its massive array of compute units.[1] Furthermore, GPUs are heavily biased toward floating-point math; executing arbitrary bit-shifts to decode Int2 or FP4 inside a WGSL shader induces register bloat and thread divergence, whereas the CPU executes these bit-shifts trivially.[1]

Backward Pass Parity: The INDUS and H-DRIFT Phenomenon

Training neural networks requires absolute deterministic stability during the backward pass to prevent gradients from exploding or vanishing across epochs. Loom evaluates this stability by calculating the discrepancy (D-DX for input gradients, D-DW for weight gradients) between the CPU reference and the GPU execution, categorizing the results into industrial parity states.[1]

Data Type	Backward Status	Input Grad Discrepancy (D-DX)	Weight Grad Discrepancy (D-DW)	Observation
Float64	✅ **INDUS**	2.98×10^{-8}	1.53×10^{-5}	Perfect industrial parity.
Float32	✅ **INDUS**	2.98×10^{-8}	1.53×10^{-5}	Perfect industrial parity.
FP8-E4M3	✅ **INDUS**	2.98×10^{-8}	1.34×10^{-5}	Perfect industrial parity.
FP8-E5M2	✅ **INDUS**	2.98×10^{-8}	1.53×10^{-5}	Perfect industrial parity.
Int64	✅ **INDUS**	4.47×10^{-8}	1.34×10^{-5}	Perfect industrial parity.
Float16	🟤 **H-DRIFT**	**0.283**	1.53×10^{-5}	Severe gradient input drift.
BFloat16	🟤 **H-DRIFT**	**0.282**	1.53×10^{-5}	Severe gradient input drift.

Standard precision types like Float32, Float64, and highly specific sub-byte formats like FP8-E4M3 achieved strict **INDUS** status, confirming that their gradient discrepancies were mathematically negligible.[1] However, Float16 and BFloat16 exhibited **H-DRIFT** (Heavy Drift). While their weight gradients remained stable, their input gradients spiked to discrepancies of roughly 0.28.[1]

This drift is caused by the non-associative nature of floating-point accumulation within the GPU's highly parallel thread architecture. When thousands of shader invocations sum gradients concurrently, the lack of strict atomic ordering in Float16 induces severe rounding errors. To mitigate this during training, Loom's architecture necessitates a host-side CPU fallback mechanism to calculate backwards passes for these volatile datatypes, ensuring that training maintains bit-perfect determinism across devices.[1]

WebGPU Shading Architecture and Execution

When model dimensions expand beyond the capacity of the CPU's cache, Loom offloads execution to the GPU via a sophisticated WebGPU implementation.[1]

Dynamic WGSL Compilation

Because WebGPU Shading Language (WGSL) does not natively support runtime-sized workgroup arrays or arbitrary precision decoders, Loom compiles its shaders dynamically in Go.[1] For instance, ShaderTiledDense(tileSize int) injects the required tiling constraints directly into the WGSL payload (e.g., @workgroup_size(64, 1, 1)), ensuring that the shader is structurally customized for the exact matrix topology required by the layer.[1]

For customized bit-depths, the engine dynamically binds mathematical decode functions directly into the shader.[1] If a layer uses DTypeFP4, the engine injects a WGSL function that maps bitwise codes directly to floating-point scalars: if (c == 1u) { return 0.75; }... if (c == 13u) { return -3.0; }.[1] This ensures that the GPU can stream compressed 4-bit payloads across the memory bus, expanding them into 32-bit floating-point registers locally within the compute core, vastly enhancing functional memory bandwidth.

Sophisticated Softmax Engineering

Loom provides highly engineered, hardware-accelerated variants of the Softmax function via the ShaderSoftmaxForward WGSL script.[1] Computing exponential sums across massive vectors is numerically unstable and prone to overflow. Loom mitigates this by executing a two-pass algorithm utilizing workgroupBarrier() synchronization and shared local memory (var<workgroup> shared_reduce: array<f32, 256>) to isolate local maximums prior to exponential summation.[1]

Crucially, the shader natively implements statistical modifications:

1. **Masked Softmax:** Automatically zeroes out padded sequence lengths by applying -1×10^{30} to masked coordinates.[1]
2. **Gumbel Softmax:** Introduces categorical stochasticity by synthesizing random noise natively inside the shader. It utilizes a Permuted Congruential Generator (PCG) hash (fn pcg_hash(input: u32)) seeded by the global thread invocation ID to generate uniform noise, then applies the Gumbel transformation (-log(-log(u))) before normalization.[1]

VRAM Data Residency

To circumvent the fixed PCIe dispatch latency discussed in the benchmarks, the Transformer struct implements persistent VRAM residency.[1] The ForwardTokenIDsWGPU function accepts raw token IDs and executes the initial Embedding lookup directly on the GPU using DispatchEmbedding.[1] Because the Embeddings, LMHead, and FinalNorm arrays are permanently cached as persistent wgpu.Buffer objects, an entire auto-regressive generation step can occur on the GPU without a single byte of activation data transferring back to the CPU

until the final probability logit is requested.[1]

The DNA Engine: Hierarchical Spatial Correlation

A deeply specialized facet of the Loom architecture is the DNA Engine—a topological reconstruction module designed to analyze, correlate, and index distinct neural networks irrespective of their active numerical precision or parametric scale.[1]

The engine operates by parsing a VolumetricNetwork through the ExtractDNA() pipeline.[1] This function iterates over the 3D grid and synthesizes a NetworkDNA array composed of LayerSignature structs. Each signature strips away high-level semantic labeling and focuses strictly on structural metadata: spatial coordinates (Z, Y, X, L), geometric Type (e.g., CNN, LSTM, Dense), active DType, and an L2-normalized vector derived from the layer's WeightStore.Master array.[1] For non-weighted, structural routing layers such as LayerSoftmax or LayerResidual, a neutral scalar [1.0] is registered as a topological presence marker.[1]

By converting massive neural networks into these streamlined topological signatures, the CompareNetworks() function can execute complex comparative analysis using CosineSimilarity.[1] This mathematical approach allows the engine to accurately calculate a Similarity Index (SI) between a high-fidelity FP64 scientific model and its severely compressed 1-bit Binary derivative.[1]

Furthermore, the DNA engine implements cross-depth spatial tracking to detect LogicShift phenomena.[1] If the topological signature of a layer at coordinates $(0, 0, 0, 1)$ in Network A strongly matches the signature of a layer at $(0, 0, 0, 5)$ in Network B, the engine formally registers a logic shift. This provides unparalleled transparency into how architectural behaviors migrate and adapt across deep learning training iterations, knowledge distillation pipelines, or NeuroEvolution of Augmenting Topologies (NEAT) mutations.[1]

Non-Differentiable Learning: Target Propagation (Tween)

Traditional backpropagation is entirely dependent on the calculus of the chain rule. It requires continuous, differentiable activation functions across the entire computational graph, and relies on back-propagating global error gradients layer by layer. This forces the entire network state to be cached in RAM, causing immense memory pressure.[1]

Loom 0.78 challenges this standard through its implementation of Neural Target Propagation, internally classified as "Tween".[1] In this biological learning paradigm, rather than computing gradients of a global loss function, the network estimates an "ideal target" for the activations of a specific layer. The divergence between the layer's actual output and this ideal target defines a localized gap.[1]

The layer then adjusts its weights directly using a localized, Hebbian-style logic update: delta = learningRate * input * gap.[1] Because this process updates weights using local information exclusively, it entirely bypasses the chain rule.[1] This mechanism is crucial for the extreme

quantization types supported by Loom. 1-bit Binary and 2-bit networks are fundamentally non-differentiable; their step functions have zero gradients almost everywhere. Tween allows Loom to physically train these ultra-low-bit models natively on edge devices without requiring VRAM-intensive automatic differentiation graphs.[1]

The Pure Golang Machine Learning Ecosystem

Loom's decision to architect a highly performant neural engine in pure Go without relying on C++ or CUDA is highly unconventional.[5] The Golang machine learning ecosystem has historically struggled to rival the comprehensive dominance of Python-centric frameworks.[5] Analyzing the existing competitors within this space highlights the specific engineering hurdles that Loom was designed to overcome.

The Legacy Graveyard: Gorgonia and GoLearn

For years, the most prominent deep learning library in Go was **Gorgonia**.[6] Gorgonia operated as a graph-based computational engine conceptually mirroring early iterations of Theano or TensorFlow, allowing developers to define complex multidimensional tensor equations.[7] However, the framework struggled to maintain momentum and is widely considered dormant or abandoned by the Go community.[10] Its reliance on early graph abstraction methodologies failed to scale with the rapid emergence of dynamic Transformer structures and complex MoE (Mixture of Experts) architectures.

Similarly, libraries such as **GoLearn** and **goml** provided capable tooling for classical machine learning algorithms—such as decision trees, K-means clustering, and online reactive Bayesian classifiers—but lacked the rigorous tensor calculus and GPU-acceleration pipelines mandatory for modern deep learning inference.[13]

The CGO Wrapper Paradigm: GoMLX

Currently, the most actively maintained alternative in the ecosystem is **GoMLX**.[11] GoMLX functions as a sophisticated Go binding layer interfacing directly with **OpenXLA** (Accelerated Linear Algebra) and PJRT—the identical C++ backend engine powering Google's JAX and TensorFlow.[16]

GoMLX offers tremendous performance on massive server clusters, supporting distributed multi-TPU and multi-GPU execution via XLA Shardy technology.[11] However, this power comes at the severe cost of portability. Because GoMLX acts as a wrapper requiring dense CGO (C-to-Go) interoperability with OpenXLA C++ binaries, it cannot easily be cross-compiled to diverse edge platforms.[5] It excels in high-performance cloud environments but fundamentally violates the requirements of sovereign, lightweight edge deployment.

The Edge Competitors: Born and Cactus Compute

Recent initiatives have attempted to address this edge deficiency. **Born** is a modern deep learning framework for Go directly inspired by Rust's *Burn*.[17] Like Loom, Born champions a pure Go architecture free of CGO dependencies, enabling single-binary deployment with WebGPU

acceleration.[18] It successfully implements advanced architectures like Transformers, GQA, and SwiGLU.[19] However, Born adheres to traditional PyTorch-style API interfaces and sequential DAG layers [20], lacking the 3D volumetric routing and the aggressive sub-byte native bit-packing optimizations that give Loom its edge on heavily constrained devices.[1]

Cactus Compute, backed by Y Combinator (S25), takes a different approach. Cactus is a low-latency inference engine optimized exclusively for mobile smartphones.[21] It achieves excellent throughput on modern devices (e.g., 50-70 tokens/sec on an iPhone 16) through zero-copy memory mapping and NPU-specific quantization kernels.[22] However, Cactus relies on custom C/C++ architecture rather than pure Go, and operates as a hybrid engine that automatically routes complex voice and reasoning tasks to centralized cloud models like Gemini.[21]

Loom's Sovereign Distinction

Loom separates itself from this entire ecosystem through a rigid commitment to absolute sovereignty and what the documentation describes as "Bit-Perfect Determinism".[2] By refusing to integrate CGO wrappers (like GoMLX) or cloud-fallback APIs (like Cactus), Loom guarantees that a complex model will yield mathematically identical matrices—down to the exact decimal—whether executed on a Windows desktop, an Android phone, or within a WebAssembly browser sandbox.[2] It incorporates everything from HuggingFace Safetensor decoders and BPE tokenizers to 21 distinct numeric kernels directly into its static binary [1], making it a singular, standalone entity in the Golang landscape.

Universal Teleportation: The Export and Deployment Matrix

The architectural decision to write Loom entirely in pure Go without external C dependencies unlocks universal cross-compilation capabilities.[1] The exact same codebase orchestrating the volumetric grid and the Plan 9 assembly routines can be exported natively to a highly diverse matrix of operating systems and hardware targets.[3]

1. Mobile Environments (android_arm64, android_x86_64, ios_arm64)

Loom compiles natively into dynamic shared objects (.so) for Android or static libraries (.a) for iOS architectures. By utilizing the github.com/openfluke/webgpu/wgpu package, the engine automatically hooks into the native mobile graphics APIs—Vulkan on Android and Metal on iOS.[24] This ensures that intensive tensor dispatches can operate securely within strict mobile sandboxes, circumventing battery-draining CPU loops, while the Plan 9 assembly handles non-GPU-friendly quantized matrices.[1]

2. Apple Ecosystem Integration (ios_xcframework, macos_amd64, macos_arm64, macos_universal)

For rigorous integration into Swift and Objective-C, standard raw binaries are inadequate.

Loom's architecture allows it to be packaged seamlessly as an ios_xcframework, which cleanly encapsulates all required headers and binaries for physical iPhone silicon (arm64) as well as local Xcode simulators. On desktop hardware, Loom targets both Intel (macos_amd64) and Apple Silicon (macos_arm64), utilizing the macOS lipo toolchain to synthesize a single macos_universal binary that intelligently routes execution based on the host chip.[1]

3. Server and Desktop Deployments (linux_amd64, linux_arm64, windows_amd64, windows_arm64)

The engine compiles into standard ELF executables for Linux environments and PE/COFF .exe files for Windows distributions.[1] The internal SystemAudit module provides deep hardware introspection tailored to these platforms.[1] On Linux, Loom parses /proc/cpuinfo and /sys/class/drm to deduce GPU topology and VRAM allocation dynamically. On Windows, it safely interfaces with system APIs via syscall.NewLazyDLL, utilizing kernel32.dll and dxgi.dll to interrogate the DXGI adapter architecture without requiring external C++ build chains.[1]

The Teleport and Wrap Bridging Layers

To extend Loom's reach beyond native Go developers, the OpenFluke ecosystem implements a suite of robust bridging technologies [25]:

- **Teleport:** A standardized C-ABI bridge that exposes the engine's internal memory structures to Python, C#, Rust, and C++. This architecture directly powers the @openfluke/welvet Python package distributed on PyPI, allowing data scientists to invoke Loom's 3D grid natively within Python scripts without managing complex C++ compilation environments.[3]
- **Wrap and Portal:** The Wrap module compiles the engine's logic down to highly optimized WebAssembly (js/wasm). The Portal package then encapsulates this WASM runtime into an NPM library (@openfluke/welvet). This allows developers to run massively compressed bit-packed matrices directly inside a user's browser tab using native WebGPU browser standards, ensuring data sovereignty without server transmission.[2]

Advanced Integrations: Tokenization and Model Ingestion

To function as a universal AI engine, Loom 0.78 features robust native handlers designed to ingest and parse contemporary AI formats seamlessly.[1]

The bpe.go package contains an entirely independent, highly sophisticated Byte-Pair Encoding (BPE) Tokenizer engineered to parse and decode the HuggingFace tokenizer.json specification directly into Go structures.[1] It dynamically constructs token ID mapping dictionaries, processes complex BPE MergePair rules, and executes sophisticated regex-based pre-tokenization logic to parse whitespace and punctuation accurately.[1] Crucially, it implements the exact GPT-2 byte-fallback mechanism necessary for LLM generation, ensuring that unknown UTF-8 characters are safely mapped to byte-level hex tokens (e.g., <0x4A>) rather than crashing the

generation pipeline with out-of-vocabulary exceptions.[1]
For structural ingestion, Loom implements a deeply analytical UniversalLoader. When presented with a raw safetensors file, the LoadUniversalDetailed routine parses the tensor geometries. It evaluates the spatial rank of each tensor—recognizing that a Rank 4 tensor implies a LayerCNN2, while a Rank 2 tensor implies LayerDense or LayerEmbedding.[1] The loader then executes a greedy bias heuristic, identifying associated 1-dimensional bias vectors by checking dimension matching and variance profiles.[1] It algorithmically reconstructs complex LayerSwiGLU, LayerLSTM, and LayerMultiHeadAttention topologies directly from disjointed arrays by mapping structural archetypes against the tensors.[1]
Once mapped, the PrefixWeightMapper routes the data. It uses regex and string suffix matching to automatically inject weights like model.layers.N.self_attn.q_proj into the precise coordinate slots (layerIdx, slot, subRole) of the pre-configured VolumetricNetwork grid.[1]

Strategic Trajectory and Future Outlook

Analyzing the architecture of version 0.78 reveals a highly specific vector for Loom's future development: the democratization of localized, evolving, sovereign intelligence.[3]

Autonomous On-Device Entities

The primary instantiation of Loom's architecture is **SoulGlitch**—an in-development AI digital pet that resides entirely on-device without cloud connectivity.[1] This application highlights the shift from stateless conversational APIs to stateful, persistent companions.[28] Because Loom leverages Target Propagation (Tween) rather than VRAM-intensive global backpropagation, edge applications like SoulGlitch can learn and evolve their emotional reactions to users locally, running active training loops on mobile silicon without requiring server synchronization.[4]

Evolutionary Topologies

The prominent integration of KMeansCluster heuristics, the DNA Engine signature parsing, and infrastructure designated for NEAT (NeuroEvolution of Augmenting Topologies) strongly indicates that future iterations of Loom will focus on dynamic network growth.[1] The neatAddConnection function already exists within the codebase, actively injecting random IsRemoteLink spatial connections into the 3D grid.[1] This implies a trajectory where networks are not statically compiled, but organically mutate, splice, and adapt their volumetric topology to optimize for the specific hardware constraints of the device they reside on.

Enterprise Sovereign Data Security

As centralized enterprise AI providers—such as Box—push strategies that demand corporate data be uploaded to cloud platforms for analysis [1], Loom offers the exact inverse paradigm. By utilizing 1-bit and 2-bit quantization to compress multi-gigabyte foundation models by 98.4%, and leveraging Plan 9 assembly to execute them natively on low-power CPUs without external C++ or CUDA dependencies, Loom allows organizations to execute powerful LLMs on air-gapped hardware.[1] This establishes Loom as a foundational technology for highly regulated

sectors—defense, healthcare, and finance—requiring localized intelligence with zero possibility of data exfiltration.

Conclusion

Loom/Poly version 0.78 establishes a formidable architectural milestone in the pursuit of deterministic, hardware-agnostic neural computation. By dismantling the sequential DAG paradigm in favor of a 3D volumetric grid, and decoupling mathematical precision from high-level topology via its 21-dtype polymorphism, it provides unparalleled developmental flexibility.

The integration of pure Go sub-byte packing algorithms and highly optimized Plan 9 assembly inner loops conclusively demonstrates that specialized, cache-localized CPU execution can vastly outclass GPU multi-core processing for heavily quantized networks. As the broader machine learning ecosystem continues to suffer from dependency bloat, CGO friction, and mandatory cloud centralization, Loom stands as a uniquely sovereign alternative. From massive linux_amd64 computing clusters down to battery-constrained ios_arm64 edge devices and WebAssembly browser tabs, its pure-Go architecture ensures that a model trained once behaves identically everywhere, cementing its position as the bedrock engine for the future of localized, evolving artificial intelligence.

Works cited

1. README.md
2. loom - Layered Omni-architecture Openfluke Machine - GitHub, accessed May 18, 2026, https://github.com/openfluke/loom
3. Loom — The Universal AI Engine - OpenFluke, accessed May 18, 2026, https://openfluke.com/loom
4. OpenFluke — Where AI Learns to Play, accessed May 18, 2026, https://openfluke.com/
5. Why Machine Learning in Go Struggles — and Why I'm Not Giving Up | by Davide Marro, accessed May 18, 2026, https://levelup.gitconnected.com/why-machine-learning-in-go-struggles-and-why-im-not-giving-up-46158452d0d0
6. Machine Learning - Awesome Go, accessed May 18, 2026, https://awesome-go.com/machine-learning/
7. Gorgonia: main, accessed May 18, 2026, https://gorgonia.org/
8. Gorgonia is a library that helps facilitate machine learning in Go. - GitHub, accessed May 18, 2026, https://github.com/gorgonia/gorgonia
9. Go in the AI/ML Landscape: A Practical Guide - Medium, accessed May 18, 2026, https://medium.com/@vladimirvivien/go-in-the-ai-ml-landscape-a-practical-guide-d36d44f360d2
10. Is Gorgonia dead? : r/golang - Reddit, accessed May 18, 2026, https://www.reddit.com/r/golang/comments/1bvy6at/is_gorgonia_dead/
11. GoMLX: Machine Learning in Go - Why This Matters for AI Agents, accessed May 18, 2026, https://muleai.io/blog/gomlx-machine-learning-go-wasm/

12. Are golang ML frameworks all dead - Reddit, accessed May 18, 2026, https://www.reddit.com/r/golang/comments/1gfi902/are_golang_ml_frameworks_all_dead/
13. cdipaolo/goml: On-line Machine Learning in Go (and so much more) - GitHub, accessed May 18, 2026, https://github.com/cdipaolo/goml
14. golearn package - github.com/sjwhitworth/golearn - Go Packages, accessed May 18, 2026, https://pkg.go.dev/github.com/sjwhitworth/golearn
15. How to start learning ML with golang - Reddit, accessed May 18, 2026, https://www.reddit.com/r/golang/comments/1hpt1op/how_to_start_learning_ml_with_golang/
16. GoMLX: An Accelerated Machine Learning Framework For Go - GitHub, accessed May 18, 2026, https://github.com/gomlx/gomlx
17. Born v0.2.0 Released - First Go ML Framework with Zero ... - GitHub, accessed May 18, 2026, https://github.com/born-ml/born/discussions/2
18. GitHub - born-ml/born: Production-ready ML framework for Go with zero dependencies. Train and deploy neural networks as single binaries. PyTorch-like API, type-safe tensors, automatic differentiation., accessed May 18, 2026, https://github.com/born-ml/born
19. I Skipped My Birthday to Give Go Its First Real ML Framework - DEV Community, accessed May 18, 2026, https://dev.to/kolkov/i-skipped-my-birthday-to-give-go-its-first-real-ml-framework-13gj
20. Born ML - GitHub, accessed May 18, 2026, https://github.com/born-ml
21. cactus-compute/voice-agents-hack: Get started with Cactus x Google DeepMind x Y Combinator - Gemma 4 Voice Agents - GitHub, accessed May 18, 2026, https://github.com/cactus-compute/voice-agents-hack
22. Cactus Compute, accessed May 18, 2026, https://cactuscompute.com/
23. Launch HN: Cactus (YC S25) – AI inference on smartphones | Hacker News, accessed May 18, 2026, https://news.ycombinator.com/item?id=45291024
24. openfluke - GitHub, accessed May 18, 2026, https://github.com/openfluke
25. Paragon & OpenFluke: Running AI Everywhere — From Desktop to Android - Medium, accessed May 18, 2026, https://medium.com/@planetbridging/paragon-openfluke-running-ai-everywhere-from-desktop-to-android-320c5d36de8a
26. welvet - PyPI, accessed May 18, 2026, https://pypi.org/project/welvet/
27. @openfluke/welvet CDN by jsDelivr - A free, fast, and reliable Open, accessed May 18, 2026, https://cdn.jsdelivr.net/npm/@openfluke/welvet/
28. SoulGlitch — AI Creature Evolution Game - OpenFluke, accessed May 18, 2026, https://openfluke.com/soulglitch