Strategic Architecture Pivot: From CPU Assembly to GPU Graph Compilation for Volumetric Neural Networks

Executive Summary

The optimization of volumetric neural networks presents a unique challenge at the intersection of computational physics, compiler architecture, and machine learning engineering. Traditionally, performance gains in deep learning runtimes have been secured by dropping into low-level, hardware-specific instructions—such as Plan 9 assembly (ASM) for central processing units (CPUs) or Compute Unified Device Architecture (CUDA) for graphics processing units (GPUs). Recent empirical data gathered from the M-POLY-VTD (Multi-numerical POLYmorphic Volumetric Tiled-tensor Dispatcher) engine reveals a critical divergence between expected instruction-level optimizations and actual execution bottlenecks.[1] The data unequivocally demonstrates that while hand-written CPU ASM yields significant speedups for large, monolithic layer matrices, it suffers from severe performance degradation when applied to the spatial, cellular architectures inherent to volumetric grids. Simultaneously, the implementation of a rudimentary CPU "Executor v1" successfully proved that the abstraction tax of an interpreter loop exists, yet merely stripping away conditional switch statements yielded a negligible performance improvement.[1] The fundamental bottleneck in the architecture is not the speed of the arithmetic logic unit (ALU) processing matrix multiplications, but rather the exorbitant cost of operation dispatch. Recent academic characterizations of WebGPU overhead indicate that per-dispatch application programming interface (API) validation and command encoding cost approximately $95\mu s$ per operation, an overhead that completely dwarfs the execution time of small tensor operations in a subdivided grid matrix.[2]

To scale volumetric networks to production-level language model throughput, the architecture must undergo a fundamental paradigm shift. The strategy must pivot away from local, per-hop CPU assembly optimizations and toward global, ahead-of-time GPU graph compilation.[1] By introducing a load-time compilation step, the engine can segment contiguous spatial layers, effectively fusing operations across the volumetric grid. This report provides an exhaustive analysis of the empirical findings, the mathematical realities of dispatch overhead, the architectural blueprint for a GPU-native execution compiler, and the strategic realignment required to partition the engine into a distinct Research Runtime and a high-performance Inference Runtime.

1. Empirical Analysis of the Tiled Assembly Fallacy

The M-POLY-VTD architecture is fundamentally designed around the premise that a neural network is not merely a sequential stack, but a spatial 3D grid where each cell holds multiple layer types, dynamically morphing across 21 numerical data types (DTypes).[1] To accelerate this, the engine integrated a Plan 9 ASM path intended to bypass Go's execution overhead. However, rigorous benchmarking across varying grid dimensions exposed a profound arithmetic density paradox.

1.1 The Arithmetic Density Paradox in Volumetric Grids

The core assumption of the ASM strategy was that native assembly routines—utilizing IMUL for integer dots and avoiding floating-point fallback—would strictly outperform Go's tiled loops across all dimensionalities.[1] The logs from the seven-layer CPU suite dictate otherwise, proving that the overhead of transitioning into assembly context outweighs the computational benefits for smaller tensor shapes.[1]

In a dense, single-cell grid ($1 \times 1 \times 1$), the matrices are relatively fat, processing a standard pyramid scaling up to 64 features.[1] In this regime, the ASM paths demonstrate the intended theoretical gains. Evaluating the Single-Core (SC) forward timing on Float64 reveals that the Go tiled execution requires $87.1\mu s$, whereas the native ASM execution requires $54.4\mu s$, yielding an expected speedup of $1.60\times$.[1] The performance peak in this single-cell topology is achieved by the FP8-E5M2 DType, which reaches a remarkable $2.47\times$ speedup, executing in $16.3\mu s$ via ASM compared to $40.3\mu s$ in the Go reference.[1]

However, as the grid expands volumetrically into a $2 \times 2 \times 2$ (56-layer stack) and subsequently a $3 \times 3 \times 3$ (189-layer stack) configuration, the matrices within each discrete cell become proportionally thinner to maintain identical parameter bounds across the entire volume.[1] The performance inversion at these scales is stark and mathematically undeniable.

Grid Topology	Layer Count	Best Float32 Speedup (SC)	Best Float64 Speedup (SC)	Best Quantized Speedup (SC)
$1 \times 1 \times 1$	7	$1.64\times$	$1.60\times$	$2.47\times$ (FP8-E5M2)
$2 \times 2 \times 2$	56	$1.00\times$	$0.77\times$	$1.11\times$ (Uint8)

$3 \times 3 \times 3$	189	$0.88\times$	$0.86\times$	$1.00\times$ (Ternary)

As observed in the $3 \times 3 \times 3$ matrix, the ASM implementation is actively slower than the Go tiled reference for nearly all DTypes, operating at a fractional multiplier of $0.75\times$ to $0.88\times$ across the floating-point and integer spectrums.[1] For example, the Go tiled loop processes the Float32 network in $31.2\mu s$, while the ASM kernel takes $35.3\mu s$.[1]
This phenomenon is governed by instruction cache locality, context-switch overhead, and memory bandwidth. A $3 \times 3 \times 3$ grid requires 189 distinct transitions from the Go runtime into the ASM execution environment per forward pass. The setup overhead for the ASM kernel—including register saving, pointer passing, and boundary validation—exceeds the execution time of the micro-matrix multiplication inside the cell. The CPU's instruction cache continuously thrashes as it jumps between the polymorphic dispatcher, the layer logic, and the isolated assembly instructions. Therefore, investing engineering resources into expanding the ASM queue to SwiGLU, Multi-Head Attention (MHA), and Convolutional Neural Networks (CNNs) for volumetric grids will yield diminishing or negative returns.[1]

1.2 The Mechanics of DType Morphing and Overhead

The performance delta is further exacerbated by the engine's polymorphic nature. The WeightStore system allows any layer to switch its numerical precision on the fly via the Morph function, maintaining an FP32 master copy while generating optimized arrays of int8, uint4, or binary representations.[1] For a layer executing in DTypeInt4, the engine packs two 4-bit signed values per byte to achieve an $8\times$ memory reduction.[1]
While this creates highly efficient memory footprints, moving these bit-packed arrays into an ASM kernel requires either unpacking them in Go before dispatch (which destroys the cache benefits) or writing highly specialized assembly routines that handle byte-alignment and nibble extraction natively on the CPU register.[1] The complexity of maintaining these specialized assembly routines for 21 different DTypes across N layer types is unsustainable, particularly when the end result on a $3 \times 3 \times 3$ grid is a net performance loss compared to the generic Go implementation.[1]

2. The Executor v1 Experiment and the Interpreter Tax

Recognizing the overhead introduced by the ForwardPolymorphic interpreter and the DispatchLayer jump table, an experimental "Fused Executor v1" was constructed to measure

the pure interpreter tax.[1] This executor utilized a pre-planned DenseExecPlan designed to skip the dynamic type assertions and conditional switch statements that plague the generic routing pathway.

2.1 Quantifying the Interpreter Abstraction Tax

The results of the Executor v1 experiment confirmed the hypothesis that the interpreter imposes a measurable tax, but simultaneously proved that this tax is too insignificant to serve as a primary optimization vector for high-performance computing.

In the $1 \times 1 \times 1$ topology, Executor v1 achieved a $1.26\times$ speedup on Float64 Single-Core ($87.1\mu s$ vs. $69.2\mu s$).[1] As the grid scaled to the $3 \times 3 \times 3$ architecture, the peak advantage hovered around $1.38\times$ on Float32 ($31.2\mu s$ vs. $22.6\mu s$), with the Int32 variant reaching $1.26\times$.[1]

Architecture	DType	Interpret SC	Exec v1 SC	Speedup Multiplier
$1 \times 1 \times 1$	Float64	$87.1\mu s$	$69.2\mu s$	$1.26\times$
$2 \times 2 \times 2$	Int32	$21.6\mu s$	$16.9\mu s$	$1.27\times$
$3 \times 3 \times 3$	Float32	$31.2\mu s$	$22.6\mu s$	$1.38\times$

While stripping the DispatchLayer switch statement granted a $\sim 20\%$ to $\sim 40\%$ performance increase, it completely failed to address the architectural reality of modern hardware.[1] Executor v1 planned the execution order once at load time, but it still invoked 189 discrete forward micro-dispatches in a serial loop. It did not fundamentally alter the memory access pattern, nor did it batch the underlying mathematical operations. The CPU remained bottlenecked by the serialization of hundreds of tiny tensor operations.

2.2 The Disconnect Between Sequential and Cellular Execution

The limitation of Executor v1 highlights a broader issue in how volumetric execution differs from standard deep learning models. In a traditional sequential model, a layer processes a batch of tokens or an entire high-resolution image in one massive matrix multiplication. In the M-POLY-VTD framework, a $3 \times 3 \times 3$ grid evaluating a single token forces the system to execute 189 distinct operations on exceedingly small data slices.[1]

The step mesh engine (StepState), which evaluates all layers simultaneously in a discrete-time

clock cycle using double buffering, further complicates this.[1] In the step mesh, layers can feature IsRemoteLink properties, hopping across the spatial volume to fetch activations from previous ticks.[1] Executor v1, focused merely on stripping Go-level switch statements, was structurally incapable of capturing these spatial relationships to batch them effectively. The true path to massive throughput requires collapsing those 189 micro-dispatches into a single, cohesive hardware submission that understands the geometry of the entire grid.

3. The Mathematical Realities of WebGPU Dispatch Overhead

To understand why per-layer dispatch dooms volumetric topologies, one must examine the fundamental behavior of hardware accelerators and the Application Programming Interfaces (APIs) used to interface with them, specifically WebGPU. The engine currently leverages openfluke/webgpu v1.0.4 to execute WebGPU Shading Language (WGSL) compute shaders across diverse backends including Metal, Vulkan, and DirectX 12.[1]

The documentation notes immense speedups for monolithic layers on the GPU, such as $7600\times$ for 3D Convolutional Neural Networks and $13\times$ for standard Dense layers.[1]

However, transferring the $3 \times 3 \times 3$ volumetric paradigm to the GPU using naive, loop-based dispatch exposes severe latency traps that prohibit achieving necessary tokens-per-second (tok/s) targets.

3.1 Profiling API Validation and Encoding Costs

The design of WebGPU is inherently security-focused, imposing stringent validation, state tracking, and boundary checking on every operation submitted from the host to the device. A systematic characterization of WebGPU dispatch overhead for Large Language Model (LLM) inference reveals that naive, single-operation dispatches suffer from catastrophic latency scaling.[2]

Research indicates that the pure API overhead for a single WebGPU dispatch ranges between $24\mu s$ and $36\mu s$ on Vulkan backends, escalating to $32\mu s$ to $71\mu s$ on Apple's Metal backend.[2] When host-side framework abstractions—such as data structure updates, memory allocations, and tensor state tracking—are factored into the equation, the total per-operation overhead approaches $95\mu s$ to $99\mu s$.[2]

Let N represent the total number of cells in the volumetric grid ($3^3 = 27$) and L represent the number of layers per cell (7). In a naive execution model where the interpreter iterates over the grid and dispatches a WGSL compute shader for each layer, the total dispatch time overhead $T_{overhead}$ for a single forward pass is:

$$T_{overhead} = N \times L \times t_{dispatch}$$

$$T_{overhead} = 27 \times 7 \times 95\mu s = 17,955\mu s \approx 18 \text{ ms}$$

An absolute floor of 18 milliseconds per forward pass—solely dedicated to driver communication and validation, before a single floating-point operation is executed on the GPU ALU—mathematically limits the engine to a maximum theoretical throughput of roughly 55 tokens per second. This rigid upper bound directly contradicts the project's goal of achieving high-throughput inference (e.g., 70+ tok/s for advanced models) and makes scaling to larger $4 \times 4 \times 4$ or $8 \times 8 \times 8$ grids computationally unviable.[1]

3.2 The Insufficiency of Basic Command Batching

Initial attempts to mitigate WebGPU latency in traditional engines often rely on command batching and buffer pooling. The Loom engine already utilizes a WGPUContext that manages a BeginFrame and FlushFrame lifecycle.[1] This mechanism records all operations into a single shared wgpu.CommandEncoder and submits them to the wgpu.Queue in one driver call, effectively eliminating the Go-to-GPU driver overhead that would otherwise exceed 150+ round trips per batch.[1]

However, empirical evaluations of LLM inference runtimes demonstrate that command batching, buffer pooling, and bind group caching alone provide negligible benefits because autoregressive generation forces per-token synchronization.[2] If the engine continues to rely on DispatchForwardLayer inside a for loop spanning 189 layers, the host must still encode 189 distinct pipeline states and dispatch configurations. The validation layer within the browser or the native WGPU driver still processes 189 distinct events. Therefore, the implementation of a comprehensive graph compilation step—an advanced, GPU-native evolution of the "Fused Executor"—is mandatory to collapse these operations structurally.

4. The Paradigm Shift: Graph Compilation and Operator Fusion

To reconcile the expressive, non-linear power of volumetric grids with the rigid throughput requirements of hardware accelerators, the engine must pivot from a purely polymorphic interpreter to an ahead-of-time compilation architecture.

The literature surrounding deep learning compilers (such as Apache TVM, XLA, and TensorRT) confirms that the definitive solution to dispatch overhead is operator fusion.[3] These compilers ingest a computational graph, recognize adjacent compatible operations, and emit a single, highly optimized hardware kernel that executes the entire subgraph without returning control to the CPU host.[3]

4.1 Architecting CompileExecutionPlan

The current ForwardPolymorphic method traverses the grid in strict sequential reading order: traversing the Z (depth), Y (rows), X (columns), and L (layer index) axes sequentially.[1] The proposed graph compiler, invoked via a new CompileExecutionPlan(net) API, will ingest the VolumetricNetwork topology during the model load phase and perform a topological sort to generate an execution Directed Acyclic Graph (DAG).
Instead of mapping each layer to an independent kernel, the compiler will search for **Segments**. A segment is defined as a contiguous run of layers across the volumetric grid that share an identical LayerType, compatible tensor shapes, and equivalent numerical precision (DTypes).

For a homogenous $3 \times 3 \times 3$ grid where every cell contains a sequence of layers—such as ``—the compiler will rotate the execution axis. Rather than iterating sequentially through cells:

$$\text{Cell } 0 \ (\text{Dense} \rightarrow \ \text{SwiGLU}) \rightarrow \ \text{Cell } 1 \ (\text{Dense} \rightarrow \ \text{SwiGLU}) \dots$$

The execution plan will batch the spatial dimensions:

$$\text{Segment 1: Dense (Cells 0 through 26)} \rightarrow \ \ \text{Segment 2: SwiGLU (Cells 0 through 26)} \dots$$

By grouping operations by the L coordinate across the entire Z, Y, X volume, the compiler effectively flattens 27 separate Dense matrix multiplications into a single Batched General Matrix Multiply (Batched GEMM) operation. The 189 distinct network hops are reduced to exactly 7 batched shader launches. This reduces the theoretical $T_{overhead}$ from 18 ms down to less than 1 ms, entirely neutralizing the WebGPU API tax.

4.2 Resolving IsRemoteLink Spatial Dependencies

The primary complication in compiling a volumetric execution graph is the presence of the IsRemoteLink property. This feature allows a layer at coordinate (Z_1, Y_1, X_1, L_1) to bypass its immediate predecessor and read the output of a distant, arbitrary coordinate (Z_2, Y_2, X_2, L_2).[1] These remote links create explicit data dependencies that violate the assumptions of homogenous spatial batching.
During the CompileExecutionPlan phase, the compiler must perform rigorous data-flow analysis to maintain mathematical fidelity:

1. **Dependency Graphing:** The compiler maps all input and output dependencies. While the standard flow dictates that the output of L_n feeds L_{n+1}, any IsRemoteLink

creates cross-cell edges that must be evaluated.
2. **Segment Splitting:** If Layer 2 in Cell 15 requires the output of Layer 6 in Cell 2, the compiler cannot safely batch Layer 2 across Cell 2 and Cell 15 concurrently unless the mathematical operation guarantees no race conditions. If cross-cell temporal state is required, the segment must be split into distinct execution waves.
3. **Barrier Insertion:** The execution plan will automatically insert GPU memory synchronization barriers between split segments to ensure that the necessary data has been successfully written to the VRAM before the dependent segment initiates its read sequence.

4.3 Buffer Arenas and Static Memory Planning

An efficient graph compiler also eliminates dynamic memory allocation and pointer thrashing. The M-POLY-VTD engine currently utilizes an ActivationPool (map[string]*wgpu.Buffer) to persist named tensors across frames, avoiding per-step allocations during inference.[1]
The compilation step will expand upon this by analyzing the peak memory requirement of all intermediate tensors within the generated DAG. It will allocate a single, massive contiguous chunk of VRAM—an **Arena**—and statically assign byte offsets to each layer's input and output tensors.
When Segment 1 executes, the shader reads from Arena + Offset_A and writes to Arena + Offset_B. Segment 2 subsequently reads from Arena + Offset_B. This static memory planning removes the overhead of passing multiple disparate buffer pointers to the wgpu.CommandEncoder and ensures maximal cache locality on the GPU die, as the hardware is no longer forced to fetch memory from highly fragmented VRAM allocations.

5. Re-Architecting for Inference: WebGPU Batched Execution

With the execution plan successfully compiled, the runtime relies on a new ExecutePlan(plan, input) function to interface with the hardware. The existing WebGPU implementation already utilizes BeginFrame and FlushFrame semantics, but it currently lacks the capability to interpret batched volumetric segments generated by the compiler.[1]

5.1 Native Hardware Encoding

The execution of the compiled graph must be strictly asynchronous relative to the CPU. Under the new architecture, the execution flow is entirely deferred to the driver:

1. The CPU initializes a singular wgpu.CommandEncoder at the start of the token generation step.[1]
2. The engine iterates over the condensed list of Segments defined in the execution plan (e.g., the 7 batched operations instead of 189 discrete layers).
3. For each segment, the CPU binds the pre-compiled Pipeline State Object (PSO), binds the pre-computed wgpu.BindGroup (which points directly to the Arena offsets and the concatenated weight matrices), and issues a DispatchWorkgroups command.[7]

4. The CommandEncoder is finalized, and the entire sequence is submitted to the wgpu.Queue in one driver call.[1]

Because the GPU driver only validates the single command buffer submission, the validation overhead is incurred once per frame, rather than once per layer.[8]

5.2 Segmented WGSL Compute Shaders

To process a batched segment, the underlying WGSL (WebGPU Shading Language) compute shaders must be rewritten to natively understand the 3D grid layout. A standard Dense WGSL shader computes $Y = XW^T + b$ for a single matrix. A Volumetric Batched Dense shader must compute this across the spatial grid efficiently.

The grid coordinates (Z, Y, X) map elegantly to the WebGPU compute shader's 3D workgroup invocation ID GlobalInvocationID.xyz. If the execution plan dictates that a specific layer type should be executed across a $3 \times 3 \times 3$ grid, the CPU issues a dispatch of $3 \times 3 \times 3$ workgroups. The WGSL shader determines its spatial position by computing a linear cell index:

Code snippet

```
let cell_idx = GlobalInvocationID.z * (Rows * Cols) +
               GlobalInvocationID.y * Cols +
               GlobalInvocationID.x;
```

The shader then offsets its memory reads into the global Arena and the WeightStore based on the computed cell_idx. This requires the WeightStore of all 27 instances of that specific layer to be concatenated into a single large VRAM buffer during the CompileExecutionPlan phase, enabling the GPU to fetch the correct localized weights for each spatial cell without requiring the CPU to bind 27 different weight buffers dynamically.

5.3 Operator Fusion for Complex Sub-Layers

Beyond simple spatial batching, the execution compiler must handle complex topological structures such as the LayerParallel and LayerSequential containers.[1]

In LayerParallel, the input is fanned out to multiple sub-layers, and the outputs are combined using modes such as "add", "avg", "concat", or "filter" (a Soft Mixture of Experts gate).[1] A naive implementation dispatches separate shaders for each branch and a final shader for the combination. The execution compiler will map these branches into a single fused WGSL shader where possible. For instance, if a LayerParallel contains three LayerDense branches combined via "add", the compiler will emit a fused kernel that computes all three matrix multiplications within the same workgroup, accumulating the results in shared memory before writing the final

output to the Arena. This fusion eliminates intermediate VRAM writes, drastically improving memory bandwidth utilization.

5.4 DType Metamorphosis and Quantized Integration

The M-POLY-VTD engine is defined by its capability to execute networks across 21 numerical DTypes, accommodating high-precision Float64 alongside extreme low-bit formats like Q4_0, FP8-E4M3, and Binary (XNOR-Net).[1]

In the current interpreter model, the MorphToFloat32ForGPU function performs post-training quantization (PTQ) simulation by dequantizing weights back to Float32 on the CPU before uploading them to the GPU.[1] This guarantees mathematical correctness but entirely defeats the memory bandwidth savings of low-bit formats during GPU execution.

The inference compiler will enforce strict native WGSL parsing for quantized types. The engine already possesses specialized pathways like DispatchDenseQ4 for INT4 tensors.[1] The execution plan will analyze the DType of the segment during compilation. If the segment is mapped to DTypeInt4, the compiler assigns the ShaderDenseBatchedQ4 WGSL pipeline, which reads the 4-bit packed nibbles directly from VRAM (Q4_0Block format) and decompresses them locally within the GPU registers using the block's specific Float32 scale factor.[1]

This directly addresses the core issue with the Plan 9 ASM engine. While immense engineering effort was expended to build native Ternary and Uint8 dot products in assembly [1], those optimizations effectively strand the execution on the CPU. The high-performance implementations of the 21 DTypes must reside strictly within WGSL kernels orchestrated by the execution plan.

6. Mitigating Fusion Instability: Differential Testing and the Bedrock Oracle

Transitioning from a deterministic, layer-by-layer interpreter to an aggressively optimizing graph compiler introduces severe software engineering risks. The academic literature highlights that deep learning compilers (such as Apache TVM, Google XLA, and PyTorch 2.0) are notoriously prone to silent bugs that do not crash the system but instead subtly corrupt mathematical outputs.[9]

6.1 Vulnerabilities in Optimization Logic

A comprehensive study of deep learning compiler bugs conducted by researchers at Monash University and Tianjin University analyzed over 500 bugs in real-world systems.[11] The study revealed that 61.9% of confirmed DL compiler bugs originate from "Incorrect Code Logic," specifically within optimization passes such as operator fusion and dead-code elimination.[9] When compilers attempt to fuse layers or batch operations across spatial boundaries, they must meticulously manage tensor shapes, data type coercions, stride offsets, and memory alignments.[13] If the volumetric execution compiler merges a ReLU activation into a Batched Dense kernel, but miscalculates the memory stride of the Y coordinate within the shared

Arena, the network will silently generate incorrect activations. Because the weights are dynamically morphed across 21 DTypes with varying packing densities, the matrix of potential fusion errors is immense.

6.2 The Interpreter as a Differential Fuzzing Oracle

This risk profile provides the exact strategic justification for preserving the original CPU ForwardPolymorphic interpreter pathway. The interpreter is proven, battle-tested, and deterministic across all grid shapes and all 21 DTypes, successfully passing rigorous correctness checks in the seven_layer.txt test suite.[1]

To ensure the integrity of the graph compiler, the architecture will employ **Optimization-Aware Differential Fuzzing**.[15] During the compilation of a new model or upon the initialization of the runtime, the engine will execute an automated diagnostic verification pass:

1. Generate a randomized input tensor utilizing various distributional bounds.
2. Execute the input through the unoptimized, sequential ForwardPolymorphic CPU interpreter to establish the baseline truth.
3. Execute the same input through the compiled WebGPU Execution Plan.
4. Measure the maximum absolute divergence (|Δ|) across the output tensors.

As demonstrated in the empirical logs, the engine already tracks SC $\leftrightarrow$ MC and Go $\leftrightarrow$ ASM divergence, routinely achieving exact $0.00e+00$ variance on integer formats like Int8 and FP4, and sub-$1e^{-10}$ variance on floating-point types.[1] This existing parity-checking infrastructure will be repurposed to validate the ExecutePlan outputs against the interpreter oracle.

Furthermore, any structural alterations introduced by the NEAT (NeuroEvolution of Augmenting Topologies) engine—which splices DNA and dynamically modifies layer types, activation functions, and remote links [1]—must be subjected to this differential check. Before a newly evolved, mutated network is committed to a compiled GPU plan, its topological shifts (LogicShift) must be evaluated by the DNA Engine (ExtractDNA, CosineSimilarity) and validated by the CPU interpreter.[1] This guarantees that the aggressive graph fusion required to overcome WebGPU dispatch latency never compromises the mathematical fidelity of the inference engine.

7. Strategic Reallocation and Future Roadmap

Based on the empirical evidence, the rigid hardware limitations of WebGPU, and the theoretical necessities of graph compilation, the project roadmap must be aggressively pruned and redirected. The focus must shift from making volumetric execution fast on the CPU to compiling volumetric architectures into high-throughput GPU frames.

7.1 Redefining the Product Tiers

To manage the complexity of this transition, the mental model of the M-POLY-VTD engine must

be bifurcated into two distinct but interconnected runtimes operating on the same underlying core:

A. The Research Runtime (The Laboratory)

- **Components:** ForwardPolymorphic, DispatchLayer, the StepState double-buffered mesh engine, WeightStore morphing, and the NEAT topology evolution engine.[1]
- **Purpose:** Model training, topological discovery, loss calculation, neural target propagation (Tween), and DNA analysis.[1]
- **Performance Posture:** Correctness, flexibility, and spatial awareness are paramount. Wall-clock performance on the CPU is secondary. The 189 micro-dispatches are acceptable in this regime because this runtime executes on single-batch research datasets, offline evolutionary loops, or step-wise mesh evaluations, not token-streaming production applications.

B. The Inference Runtime (The Product)

- **Components:** CompileExecutionPlan, ExecutePlan, WebGPU CommandEncoder, WGSL Batched Shaders, and ENTITY (.entity) binary checkpoint loading.[1]
- **Purpose:** Production deployment of LLMs and complex volumetric agents (e.g., Lucy , SoulGlitch) to browsers, edge devices, and server infrastructure.[1]
- **Performance Posture:** Relentless throughput. It relies entirely on operator fusion, ahead-of-time compilation, and static VRAM buffer pooling. In this runtime, the CPU acts merely as an asynchronous dispatcher of pre-recorded command graphs.

7.2 Deprecating the ASM Rollout Queue

The poly/asm rollout queue, which slated complex layers like SwiGLU, Multi-Head Attention, and Convolutional Neural Networks for hand-written assembly optimization, must be heavily de-emphasized or frozen entirely.[1]

The empirical data from the $3 \times 3 \times 3$ grid categorically proves that ASM has a negative return on investment for thin volumetric slices due to the inescapable Go-to-ASM context transition overhead.[1] Furthermore, even for fat $1 \times 1 \times 1$ dense layers where ASM successfully yields a $2\times$ to $3.5\times$ speedup on the CPU [1], this optimization is strategically irrelevant when the WebGPU pipeline achieves hardware-accelerated speedups ranging from $13\times$ (Dense) to $7600\times$ (CNN 3D).[1]

The Plan 9 ASM engine should be retained purely as a niche fallback mechanism for deployment targets that utterly lack GPU support (such as legacy embedded systems), or for instances where highly specific quantized formats (like BitNet Ternary) must be executed in a headless CPU environment without a WebGPU adapter.[1] It is no longer the primary vehicle for performance scaling.

7.3 Advancing the ENTITY Checkpoint Integration

The recent shipment of the .entity native checkpoint format successfully localized full

volumetric topologies alongside native-packed weights, bypassing the inflation of Base64 encoding and reducing disk footprint by roughly 28% compared to legacy JSON formats.[1] For the graph compiler to function optimally, the LoadEntity pipeline must be tightly coupled with CompileExecutionPlan.
Currently, loading a Transformer imports HuggingFace safetensors block-by-block, morphing them into the internal WeightStore structure while releasing host memory to avoid RAM spikes.[1] In the compiled architecture, the compilation phase will occur concurrently with or immediately after LoadEntity. The compiled Directed Acyclic Graph, static VRAM memory offsets, and WebGPU pipeline caches will be generated simultaneously. Consequently, a loaded .entity file transforms directly into an executable hardware graph, allowing the Inference Runtime to bypass the generic VolumetricNetwork Go structs entirely during the hot execution loop.

Conclusion

The pursuit of optimizing volumetric neural networks through granular CPU assembly instructions has reached a definitive mathematical and architectural limit. The inherent fragmentation of computation across 3D cells creates an insurmountable dispatch overhead that raw arithmetic speed inside the ALU cannot conquer. As the network scales volumetrically, the minimum $95\mu s$ per-operation API penalty required by modern hardware accelerators throttles any architecture that relies on sequential, layer-by-layer interpretation.
The M-POLY-VTD engine must evolve from an executing interpreter into an ahead-of-time graph compiler. By implementing a CompileExecutionPlan that maps topological segments to batched WebGPU shaders, the runtime can collapse hundreds of spatial hops into a handful of unified CommandEncoder submissions. This fusion eliminates framework abstraction taxes, bypasses CPU-to-GPU memory bandwidth bottlenecks, and aligns the software architecture with the realities of highly parallel GPU execution.
Simultaneously, recognizing the severe stability risks inherent to deep learning compilers, the existing Go interpreter must be preserved as a bedrock oracle for automated differential fuzzing. By cleanly separating the Research Runtime (focused on polymorphic correctness and topology evolution) from the Inference Runtime (focused on compiled graph execution), the project secures a stable, scalable path toward deploying volumetric, 21-dtype neural architectures at production speeds.

Works cited

1. seven_layer.txt
2. Characterizing WebGPU Dispatch Overhead for LLM Inference Across Four GPU Vendors, Three Backends, and Three Browsers - arXiv, accessed June 25, 2026, https://arxiv.org/html/2604.02344v1
3. Characterizing WebGPU Dispatch Overhead for LLM Inference Across Four GPU Vendors, Three Backends, and Three Browsers - arXiv, accessed June 25, 2026, https://arxiv.org/pdf/2604.02344
4. Democratizing On-Device LLM Inference with Machine Learning Compilers and

Web Technologies - Carnegie Mellon University, accessed June 25, 2026, http://reports-archive.adm.cs.cmu.edu/anon/2025/CMU-CS-25-112.pdf
5. Leveraging Neural Graph Compilers in Machine Learning Research for Edge-Cloud Systems - arXiv, accessed June 25, 2026, https://arxiv.org/html/2504.20198v1
6. WebGPU Explained - WebLLM - Browser-Native AI Protocol, accessed June 25, 2026, https://www.webllm.org/blog/webgpu-explained
7. Web Neural Network API - W3C, accessed June 25, 2026, https://www.w3.org/TR/2023/CRD-webnn-20230414/
8. Web Neural Network API - W3C, accessed June 25, 2026, https://www.w3.org/TR/2023/CRD-webnn-20230830/
9. Optimization-Aware Test Generation for Deep Learning Compilers - arXiv, accessed June 25, 2026, https://arxiv.org/html/2511.18918v1
10. PartiNet is a dynamic adaptive neural network for high-performance particle picking in cryo-electron microscopy | bioRxiv, accessed June 25, 2026, https://www.biorxiv.org/content/10.64898/2026.01.23.700950v1.full-text
11. Yongqiang Tian, accessed June 25, 2026, https://yqtian.com/
12. A comprehensive study of deep learning compiler bugs - Monash, accessed June 25, 2026, https://research.monash.edu/en/publications/a-comprehensive-study-of-deep-learning-compiler-bugs/
13. Optimization-Aware Test Generation for Deep Learning Compilers - arXiv, accessed June 25, 2026, https://arxiv.org/html/2511.18918
14. Julia for Biologists - PMC - NIH, accessed June 25, 2026, https://pmc.ncbi.nlm.nih.gov/articles/PMC10216852/
15. Optimization-Aware Test Generation for Deep Learning Compilers (ICSE 2026 - Research Track) - conf.researchr.org, accessed June 25, 2026, https://conf.researchr.org/details/icse-2026/icse-2026-research-track/174/Optimization-Aware-Test-Generation-for-Deep-Learning-Compilers
16. (PDF) Optimization-Aware Test Generation for Deep Learning Compilers - ResearchGate, accessed June 25, 2026, https://www.researchgate.net/publication/397933953_Optimization-Aware_Test_Generation_for_Deep_Learning_Compilers