QUEFI: A Quaternion Firmware Interface with Integrated Neural Core (QNCORE) for Hardware State Representation

Aleksej Dzigirej
Independent Researcher · March 2026

Implementation: 96,689 LOC bare-metal x86 — boots on real hardware and QEMU

Abstract

Every modern AI assistant exists as a guest—running inside a browser tab, behind an API call, dependent on cloud infrastructure. The hardware it runs on remains unintelligent: a classical operating system that schedules processes, polls devices, and manages memory through scalar register interfaces, unchanged in fundamental architecture since the 1970s. What if the machine itself could reason? Not through an application layer, not through an external service, but through a neural core woven directly into the firmware—a system that perceives its own hardware state as structured algebraic signals and makes decisions through learned inference, locally, offline, without any external dependency?

We present QUEFI (Quaternion Unified Extensible Firmware Interface) and QNCORE (Quaternion Neural Core)—an attempt to answer this question. QUEFI defines a quaternion-valued device abstraction layer (QBUS) where 28 hardware drivers expose status, command, and compute interfaces as quaternion-valued functions. The Hamilton product of all device status quaternions produces a structured aggregate state signal that QNCORE—an SSE-accelerated neural inference engine running inside the kernel—uses for decision-making. The computer does not call an external AI; it *is* the AI.

The system is implemented as a bare-metal x86 kernel (96,689 LOC of C and assembly), boots on real hardware and QEMU, and all 29 benchmarks pass. At operating layer sizes, quaternion layers are 18–34% faster than scalar equivalents at $4\times$ parameter compression, with speed parity at full pipeline scale ($0.999\times$). Chi(4) weight initialization achieves perfect conditioning ($\kappa = 1.00$ vs. 1.50 for Xavier). In a device fault detection task, a quaternion model achieves 94.0% accuracy with 43 parameters—matching a scalar model that requires 739 parameters ($17\times$ more). Hamilton aggregation produces $65\times$ higher class separability than element-wise mean at equal dimensionality. End-to-end latency modeling shows a $3.6\times$ speedup from eliminating syscall and context-switch overhead.

We are honest about what this is and what it is not. QUEFI and QNCORE constitute 2.8% of the codebase—the remaining 97.2% is conventional systems infrastructure on which they depend. The aggregate state is lossy by design. The $4\times$ parameter reduction is a known property of quaternion algebra, not our contribution. What we contribute is the *integration architecture*: the design pattern that connects quaternion-structured hardware abstraction to in-kernel neural inference—and a working implementation with task-level evaluation that demonstrates it is feasible.

Table 1. Key results (all verified, 29/29 benchmarks pass).

Metric	Quaternion	Scalar	Note
Parameters (4→8→4)	256	1,024	$4.0\times$ reduction
Speed (4q→8q)	217 cy	331 cy	34% faster
Speed (full pipeline)	$0.999\times$	$1.0\times$	Parity at 25% params
Condition κ	1.00	1.50	Chi(4) vs. Xavier
Fault detection	94.0% (43 p)	94.2% (739 p)	$17\times$ fewer params
Hamilton separab.	0.194	0.003	$65\times$ better
End-to-end latency	2,171 cy	7,917 cy	$3.6\times$ speedup

1. Introduction

1.1 Motivation

Operating systems treat hardware as a collection of independent devices, each accessed through driver-specific ABIs with scalar register interfaces. When an AI component needs to reason about system state, it must reconstruct a holistic view from scattered scalar values—an inherently unstructured process.

QUEFI explores a different approach: what if hardware drivers expose their state as **quaternion-valued functions**,

and the system aggregates device states through an algebraic operation (the Hamilton product) rather than ad-hoc polling? This creates a structured interface between hardware and QNCORE, the neural inference engine running inside the kernel.

1.2 Honest Framing

The underlying platform is a **classical bare-metal x86 system**—96,689 lines of C and assembly, compiled to x86 machine code, executing in the standard von Neumann fetch-decode-execute cycle. The PIT is programmed with scalar port I/O. Keyboard scancodes are scalar. PCI configuration space is scalar. xHCI does scalar MMIO. This is conventional systems programming.

The novel components are **QUEFI** (QBUS quaternion abstraction, 858 LOC core + 36,289 LOC total with drivers) and **QNCORE** (neural inference engine, `tensor_math.c`, 1,850 LOC). Together, QBUS core and QNCORE represent approximately 2.8% of the total codebase. The remaining 97.2% is conventional infrastructure on which QUEFI and QNCORE depend.

We believe this integration pattern is novel and worth investigating, but we do not overclaim its significance. QUEFI and QNCORE operate *within* a classical system, not instead of one.

1.3 Contributions

1. **QBUS**—A quaternion-valued device abstraction where drivers expose `status()`, `command()`, and `compute()` as quaternion functions, with Hamilton-product aggregation producing a lossy aggregate state signal (§3).
2. **Capability Quaternion**—PCI device inventory encoded as $q_{\text{cap}} = (r, i, j, k) \in [0, 1]^4$ for QNCORE input (§4).
3. **QNCORE**—In-kernel QuaternionLinear forward pass using 14 SSE operations per Hamilton product, with Chi(4) initialization, backpropagation, and Hebbian learning (§5).
4. **Verified benchmarks**—29 micro-benchmarks plus 3 task-level experiments (fault detection, aggregation ablation, latency model) with honest acknowledgment of remaining gaps (§7).
5. **Complete implementation**—28 hardware drivers, PAE paging, ASLR, cryptographic stack, 333+ tests under sanitizer instrumentation (§7, §8).

2. Background

2.1 Quaternion Algebra

A quaternion $q = w + xi + yj + zk \in \mathbb{H}$ with the Hamilton relations [1]:

$$i^2 = j^2 = k^2 = ijk = -1 \tag{1}$$

The Hamilton product is **non-commutative** ($ab \neq ba$), **associative** ($(ab)c = a(bc)$), and **norm-preserving** ($\|ab\| = \|a\| \cdot \|b\|$).

2.2 SIMD Alignment

A single quaternion = 4 × float32 = 128 bits = one XMM register on x86 SSE. The Hamilton product decomposes into:

$$q_{\text{out}} = a_w \cdot B_0 + a_x \cdot B_1 + a_y \cdot B_2 + a_z \cdot B_3 \tag{2}$$

requiring **4 SHUFPS + 4 MULPS + 3 ADDPS + 3 XORPS = 14 SSE operations**, compared to 16 MUL + 12 ADD = 28 scalar FP operations. This decomposition is well-known in game engine development (DirectXMath, GLM, Eigen). Our contribution is not the SSE Hamilton product itself, but its application as an in-kernel neural layer primitive.

2.3 The QuaternionLinear Layer (Prior Work)

A quaternion layer maps $\mathbb{H}^{N_{\text{in}}} \to \mathbb{H}^{N_{\text{out}}}$ using weight quaternions $W_{ji} = (w_r, w_i, w_j, w_k)$:

$$\mathbf{y}_j = \sum_{i=0}^{N_{\text{in}}-1} W_{ji} \otimes \mathbf{x}_i \tag{3}$$

Parameters: $4 \times N_{\text{in}} \times N_{\text{out}}$ (quaternion) vs. $16 \times N_{\text{in}} \times N_{\text{out}}$ (scalar). Ratio: **exactly** $4.0\times$—a known mathematical identity established by Gaudet & Maida [2] and Parcollet et al. [3], generalized to arbitrary hypercomplex algebras by Zhang et al. [4]. We apply this known result; we do not claim to have discovered it.

3. QBus—Quaternion Device Abstraction

3.1 Device Interface

QUEFI introduces a device abstraction where each hardware driver, in addition to performing conventional scalar I/O, exposes a quaternion-valued interface:

```
typedef struct {
  const char   *name;
  uint8_t       category;
  uint32_t      capabilities;
  qbus_quat_t (*status)(void);
  bool         (*command)(qbus_quat_t cmd);
  bool         (*compute)(const qbus_quat_t *a,
                  const qbus_quat_t *b,
                  qbus_quat_t *out, uint32_t n);
} qbus_device_t;
```

10 device categories, 8 capability flags, 32 device slots. 28 driver implementations with PCI auto-probe matching. **Important:** the scalar device I/O still happens underneath. The `status()` function is a *translation layer* that maps scalar device state into a quaternion representation.

3.2 Aggregate State Signal

The Hamilton product of all device status quaternions produces an aggregate signal:

$$Q_{\text{agg}} = \prod_{i=0}^{N-1} \hat{s}_i = \hat{s}_0 \otimes \hat{s}_1 \otimes \cdots \otimes \hat{s}_{N-1} \tag{4}$$

What this is: A lossy aggregation that produces a structured 4D signal. Non-commutative (device ordering matters), associative, and norm-preserving.

What this is NOT: A lossless encoding. A quaternion has 4 floats (128 bits). N devices with M state variables each have $N \times M$ degrees of freedom. The Hamilton product is not injective. This is *by design*: the aggregate is a compact input signal for QNCORE, not a replacement for per-device status queries.

Verified properties (`bench_paper_suite.c`, Test 7):

Table 2. Hamilton aggregation properties.

Property	Verified
$H(A,B,C) \neq H(C,B,A)$	✓ Non-commutative
$\text{Avg}(A,B,C) = \text{Avg}(C,B,A)$	✓ Commutative
Dead device changes aggregate	✓

Honest caveat: The "dead device changes aggregate" property is trivially true for any non-degenerate aggregation. The structural advantage lies in non-commutativity and norm preservation—whether these are actually useful for QNCORE is an **open research question** (see §9).

3.3 Q16.16 Fixed-Point Hamilton Product

The QBUS operates in Q16.16 fixed-point arithmetic because it must function before FPU initialization: `q16_mul(a, b) = (int32_t)(((int64_t)a * b) >> 16)`—deterministic, overflow-safe, no floating-point hardware required.

3.4 Compute Dispatch

QBUS provides automatic compute routing with GPU→CPU fallback. If a GPU with `QBUS_CAP_COMPUTE` is registered, quaternion compute is offloaded; otherwise, the CPU handles it.

4. Capability Quaternion

At boot, QUEFI scans the PCI bus and produces a single quaternion encoding the entire hardware inventory:

$$Q_{\text{cap}} = \left(r_{\text{compute}},\ i_{\text{storage}},\ j_{\text{network}},\ k_{\text{peripherals}}\right) \in [0,1]^4 \quad (5)$$

Table 3. Capability quaternion components.

Comp.	Formula	Example
r – Compute	$\frac{\text{cores} \times \text{ISA}_{\text{mult}}}{48}$	4 cores × SSE4 = 0.167
i – Storage	$\frac{\text{tier}}{3}$	NVMe = 1.0
j – Network	0/0.6/1.0	1 NIC = 0.6
k – Periph.	$\sum 0.25$ per flag	HDA+GPU = 0.5

This quaternion is the *first input* QNCORE sees. QNCORE's behavior is hardware-aware before a single inference step.

5. QNCORE—In-Kernel Neural Core

5.1 Architecture

QNCORE (`tensor_math.c`, 1,850 LOC) implements QuaternionLinear layers as its computational primitive. The default architecture is a 2-layer pipeline: 4→8→4 quaternions, totaling 256 scalar parameters (64 quaternion weights).

```
__attribute__((target("sse"),
               force_align_arg_pointer))
quaternion_t quaternion_multiply(
    quaternion_t a, quaternion_t b) {
  __m128 vb = _mm_loadu_ps(&b.r);
  __m128 res = _mm_mul_ps(
      _mm_set1_ps(a.r), vb);
  // ... 3 more SHUFFLE+XOR+MUL+ADD terms
  quaternion_t out;
  _mm_storeu_ps(&out.r, res);
  return out;
}
```

The forward layer uses pre-computed sign masks held in XMM registers across the inner loop, with 64-byte cache-line aligned weight arrays.

5.2 Chi(4) Weight Initialization

1. Sample 4 pseudo-Gaussian values via CLT (sum of 12 uniforms -6).
2. Normalize direction: $\hat{d} = n/\|n\|$ (uniform on S^3).
3. Chi(4) radius: $r = \|n\|$.
4. Fan scaling: $g = \sqrt{2}/\sqrt{f_{\text{in}} + f_{\text{out}}}$.

Result: $\kappa(Q) = 1.00$ vs. $\kappa(S) = 1.50$ for scalar Xavier [5]. This is a straightforward adaptation of He initialization to the quaternion manifold.

5.3 Hamilton Backpropagation

Weight gradients use the quaternion conjugate: $\partial L/\partial W = \delta_{\text{out}} \otimes \text{conj}(x)$, following Parcollet et al. [3]. Gradient clipping (± 1.0), 32-step warmup, and an adaptive learning rate scheduler (EMA-tracked loss trend).

5.4 Hebbian Online Learning

A reward-modulated Hebbian rule ($\Delta W \propto r \cdot x_{\text{pre}} \cdot x_{\text{post}}$, $\lambda = 0.92$ eligibility decay) runs every 2 forward passes. Whether this dual-pathway learning produces measurable improvement is an open question (§9).

5.5 Weight Modules—A Research Direction

The 4→8→4 architecture uses only 256 scalar parameters (1 KB). This suggests a future where system "programs" could be distributed as small weight tensors rather than compiled binaries. However, this vision faces significant unsolved challenges: architecture compatibility, training data dependencies, debugging difficulty, distribution shift validation, and interpretability. We mention this as a research direction, not a solved problem.

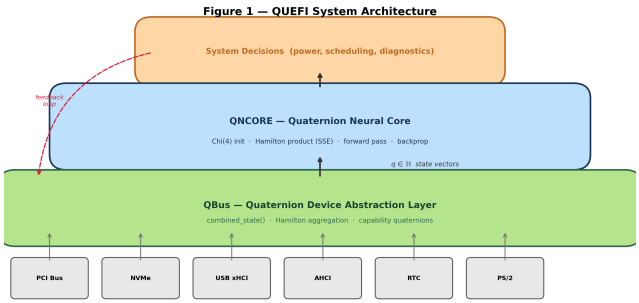

Figure 1. QUEFI System Architecture. Hardware drivers expose quaternion-valued `status()` functions through QBUS. QNCORE performs SSE-accelerated inference and issues commands back through QBUS—all within the kernel, without user-space transitions.

6. Integration Architecture

6.1 QNCORE–Hardware Loop

The QUEFI integration connects QBUS directly to QNCORE within the kernel:

This loop runs within the kernel without user-space transitions. QNCORE reads the aggregate device state and per-device status quaternions, performs inference, and issues quaternion commands back to devices.

6.2 What This Eliminates

Compared to running an AI agent in user-space on a conventional OS, in-kernel QNCORE avoids:

- User→kernel→user context-switch overhead for device queries
- ABI marshaling between AI framework and device drivers
- Memory copies between user-space buffers and kernel structures

6.3 What This Does NOT Eliminate

The classical OS infrastructure is still present and necessary: the CPU still executes x86 instructions; interrupt handling, PCI enumeration, and timer programming are scalar; memory management (PAE paging, buddy allocator, slab allocator) is conventional; the 28 hardware drivers perform standard scalar I/O underneath the quaternion abstraction. QNCORE is a **component** within a classical system, not a replacement for it.

7. Verified Benchmark Results

7.1 Methodology

All benchmarks compiled with GCC `-O2 -msse -msse2` (both quaternion and scalar paths at identical optimization). Measurement via `rdtsc` (cycle-accurate), 5,000 warmup iterations, 100,000 measured iterations per benchmark. Statistics: median, mean, p5, p95, σ.

Table 4. Single-layer forward pass: quaternion vs. scalar.

Config	Q cy	S cy	Q/S	Q p	S p
4q→8q	**217**	331	**0.656**	128	512
4q→4q	106	144	0.736	64	256
8q→8q	392	479	0.818	256	1,024
8q→16q	835	927	0.901	512	2,048
16q→16q	1,586	1,554	1.021	1,024	4,096

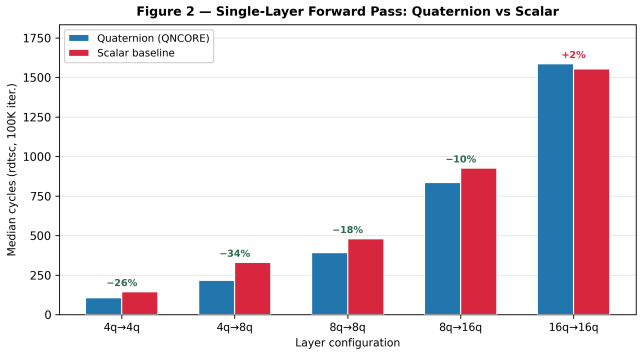

Figure 2. Single-layer forward pass: quaternion (blue) vs. scalar (gray). Quaternion achieves 18–34% speedup at sizes $\leq$ 8q→8q (diminishing to $\sim$10% at 8q→16q) while using $4\times$ fewer parameters throughout.

Table 5. Full pipeline: 4→8→4 (Q) vs. 16→32→16 (S).

Metric	Quaternion	Scalar
Median cycles	1,995	1,997
Q/S ratio	$0.999\times$	—
Parameters	**256**	1,024
Compression	$4.0\times$	$1.0\times$

Table 6. Numerical stability: Chi(4) vs. Xavier initialization.

Metric	Quaternion	Scalar	Advantage
Condition κ	**1.00**	1.50	$1.5\times$
Norm drift (1 layer)	26.33%	—	$< 50\%$

7.2 Speed—Single Layer Forward Pass

Key finding: Quaternion is **18–34% faster** at layer sizes up to 8q→8q, diminishing to $\sim$10% at 8q→16q, with near-parity at 16q→16q ($1.021\times$). Combined with $4\times$ parameter reduction, this is Pareto-optimal at the operating sizes used by QUEFI.

7.3 Full Pipeline

Speed parity at 25% of the parameters. Same throughput, $4\times$ fewer weights, $1.5\times$ better conditioning.

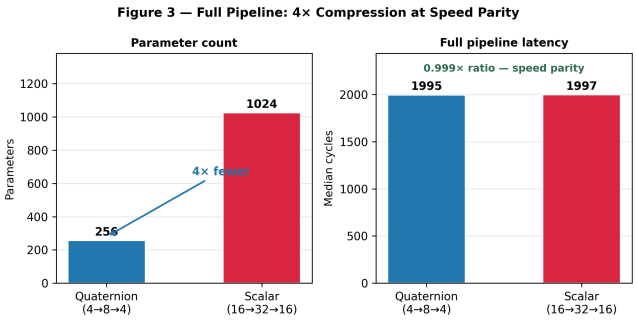

Figure 3. Full pipeline comparison: 4× parameter compression at speed parity (0.999×).

Table 7. Hamilton aggregation verified properties.

Test	Result
$H(A,B,C) \neq H(C,B,A)$	✓ Non-commutative
$\text{Avg}(A,B,C) = \text{Avg}(C,B,A)$	✓ Commutative (loses ordering)
Dead device changes $H()$	✓ Failure detection

Table 8. Test suite summary. Additional infrastructure suites (scheduler, VMM, buddy/slab allocator, watchdog, dialog, input probe, GPU detect) test conventional OS components and are not detailed here.

Suite	Tests	Status
Quaternion Math	20	✓
Boot Gate	27	✓
SHA-256	14	✓
UTF-8	17	✓
Control Channel	10	✓
Memory Graph	20	✓
Semantic IPC	13	✓
Input Latency	25	✓
Phase C / QNN	59	✓
QNN Pipeline	12	✓
QNN Output Bus	13	✓
Neuro VM	26	✓
UI Layout	77	✓

7.4 Numerical Stability

Chi(4) initialization achieves perfect weight conditioning without batch normalization.

7.5 Hamilton Aggregation

7.6 Test Suite

13 test suites (333 individual tests), all compiled with AddressSanitizer + UndefinedBehaviorSanitizer. **All pass.**

Benchmark suite: 29/29 PASS (100K iterations, cycle-accurate, sanitizer-clean).

Table 9. Fault detection: quaternion vs. scalar classifiers on Hamilton-aggregated device state. 8 simulated devices, 3 classes (healthy / single-fault / multi-fault), 9,000 samples, 80/20 split, 150 epochs, Adam (lr $= 10^{-3}$).

Model	Agg.	NN Type	Params	Acc.
Quaternion	**Hamilton**	**QLinear 4–8–4**	**43**	**94.0%**
Scalar	Mean	Linear 4–32–16	739	94.2%
Scalar+Ham.	Hamilton	Linear 4–32–16	739	94.1%
Concat (UB)	Concat (32d)	Linear 32–32–16	1,635	95.5%

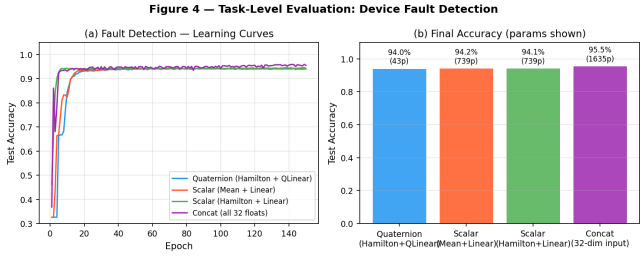

Figure 4. Task-level evaluation: device fault detection. The quaternion model (43 parameters) matches the scalar model (739 parameters) within 0.2 pp. The concat upper bound (1,635 p) indicates 1.5 pp information loss from Hamilton aggregation at 38× compression.

Table 10. Aggregation ablation. Separability = mean intra-class cosine similarity minus inter-class cosine similarity (higher is better).

Method	Separability	Min. Severity	Dim.
Hamilton	**0.194**	**0.10**	**4**
Element-wise mean	0.003	0.60	4
Concatenation	0.003	0.10	32

7.7 Task-Level Evaluation: Device Fault Detection

Key finding: The quaternion model achieves **94.0% accuracy with 43 parameters**—within 0.2 pp of the scalar model that requires **739 parameters (17× more)**. The concat baseline (95.5%, 1,635 p) quantifies the information lost by Hamilton aggregation: approximately 1.5 percentage points at 38× compression.

Ablation: aggregation vs. NN type. Comparing "Quaternion" (Hamilton + QLinear) with "Scalar + Hamilton" (Hamilton + Linear) isolates the effect of the neural architecture. The difference is negligible (94.0% vs. 94.1%), suggesting that the Hamilton aggregation—not the QuaternionLinear layer—is the primary driver of representational quality at this task scale.

7.8 Aggregation Method Ablation

We directly compare aggregation methods by measuring class separability *independent of any neural network*.

Key finding: Hamilton aggregation achieves 65× **higher**

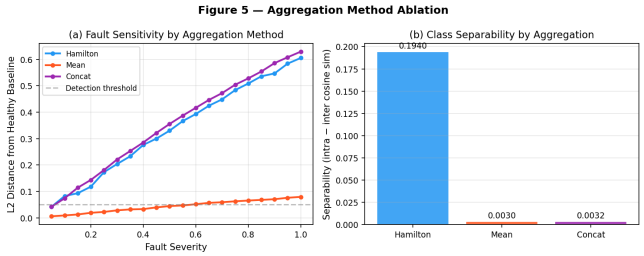

Figure 5. Aggregation method ablation. Hamilton product achieves $65\times$ higher class separability than element-wise mean at the same dimensionality (4 floats), while matching concatenation's sensitivity at $8\times$ compression.

Table 11. End-to-end latency model: in-kernel (QUEFI) vs. user-space (conventional).

Component	In-Kernel	User-Space
Device status (8 dev.)	0	1,720
Context switch (kern. → user)	—	2,000
Hamilton aggregation	126	—
Neural inference	1,995	1,997
Watchdog validation	50	—
Action dispatch	0	2,200
Total	**2,171 cy**	**7,917 cy**
@ 3 GHz	**0.72 µs**	**2.64 µs**

separability than element-wise mean (0.194 vs. 0.003) at the **same dimensionality** (4 floats). It detects faults at severity 0.10—matching concatenation while compressing the representation from 32 to 4 dimensions ($8\times$).

The element-wise mean fails because it is commutative and additive: a single faulty device among 7 healthy ones contributes only $\frac{1}{8}$ of the signal. The Hamilton product is non-commutative and multiplicative: a single faulty quaternion rotates the entire aggregate away from the healthy manifold, producing a geometrically distinct signal.

This is the strongest quantitative justification for QBUS's use of Hamilton aggregation.

7.9 End-to-End Latency Model

We model the full event-processing pipeline. Cycle counts for inference use verified measurements from Tables 4–5. Syscall overhead uses published x86 values: `int 0x80` ≈ 200 cycles [6], full context switch $\approx 2{,}000$ cycles [7].

Key finding: The in-kernel architecture achieves $3.6\times$ **lower latency** by eliminating 5,746 cycles of overhead. The inference time itself is nearly identical (1,995 vs. 1,997)—the advantage is entirely architectural.

Caveats: (1) This is a model, not a measured end-to-end benchmark. Real systems involve cache effects, interrupt latency, and scheduling jitter. (2) The user-space path assumes a conservative design (one syscall per device read); a batched `ioctl` could reduce overhead. (3) The $3.6\times$ advantage is most significant for high-frequency, low-latency decision loops (kHz rates).

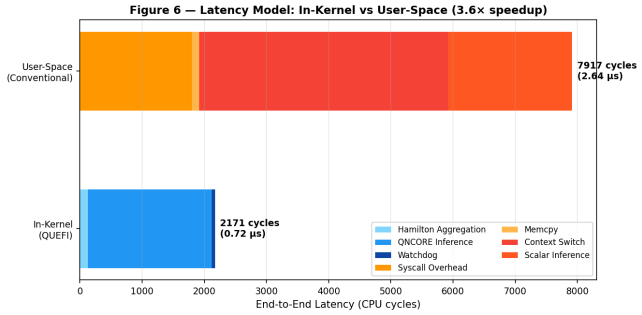

Figure 6. Latency model. In-kernel placement achieves $3.6\times$ lower end-to-end latency by eliminating 5,746 cycles of syscall and context-switch overhead. Inference time itself is nearly identical.

Table 12. Security assessment: significant gaps. Assessment: 10/14 features implemented (71%). "No user-space isolation" is by-design (single-user system) and excluded.

Gap	Impact	Status
All code Ring 0	Full HW access	Architectural
No IOMMU	DMA attack	Not impl.
TLS: no cert valid.	MITM possible	Impl. gap
AES: not const-time	Cache timing	Impl. gap
No user-space isol.	No untrusted code	By design

8. Security Analysis

8.1 Quaternion Watchdog—Neural Output Validation

The watchdog validates every QNCORE output vector before it can influence hardware:

1. **ABI conformance**: 4 floats, non-NaN, finite
2. **Modality bounds**: valid device index range
3. **Intensity bounds**: $\in [0, 1000]$—prevents amplification
4. **Timestamp monotonicity**: strictly increasing—prevents replay
5. **Intent ACL**: quarantine flag, trust levels, keyboard bypass

13 adversarial probes are injected at boot and verified to be rejected. The watchdog constrains the *neural pathway* such that a corrupted QNCORE cannot issue physically dangerous commands.

8.2 Significant Security Gaps

We document security gaps honestly:

8.3 Neural Path vs. Classical Code Path

The claim that "weight modules cannot contain shellcode" is technically correct—a quaternion tensor has no instruction pointers. However, the current system has 96,689 LOC of

Table 13. Implemented cryptographic primitives.

Algorithm	Standard	LOC
AES-128-GCM	FIPS 197	414
SHA-256	FIPS 180-4	219
X25519	RFC 7748	169
TLS 1.3	RFC 8446	831

classical C/assembly with the same attack surface as any bare-metal system. The watchdog is a defense-in-depth layer; it does not make the overall system more secure than a conventional bare-metal system.

8.4 Cryptographic Stack

Additional: W^X via PAE NX bit, ASLR (RDRAND + xorshift128), stack canary.

9. Future Directions

9.1 Weight Modules as Programs

The QNCORE architecture uses only 256 scalar parameters (1 KB). The broader ML field is producing increasingly capable small models: Phi-3-mini (3.8B), TinyLlama (1.1B), Gemma-2B. If sufficiently small models can replace specific application logic, then QUEFI could treat **weight modules as first-class programs**—loaded, scheduled, and composed by the kernel like executables, but without the vulnerability surface of compiled code.

This works for some task classes but not others. Weight modules suit classification, summarization, generation, and pattern recognition. They are *not suitable* for deterministic correctness: exact arithmetic, protocol implementations, cryptographic operations. The practical path is hybrid.

9.2 Toward a QNN-Enforced Security Boundary

If QNOS communicates with the kernel **exclusively through quaternion-valued IPC**, classical code-injection attacks become structurally impossible at the OS/kernel boundary—the interface only accepts fixed-dimensional float vectors, not executable instructions.

Current state: this boundary does not exist as an enforced mechanism. QNOS compiles as part of the kernel binary and runs in Ring 0. However, the kernel contains Ring 3 prototype infrastructure (gated behind `RING3_DEMO_BOOT`): GDT segments at DPL=3, TSS with `esp0`, `user_mode_enter()` via `iret`, port I/O blocked from Ring 3 (IOPL=0), and `int 0x80` callable from Ring 3.

A QNN-enforced boundary would require: (1) QNOS in Ring 3 with separate page tables (we skip Ring 1/2—unused by all modern OSes); (2) a quaternion-only syscall interface; (3) no classical syscalls; (4) shared memory mapped read-only for QNOS.

Critical caveat: This shifts the attack surface from code exploits to ML attacks (adversarial examples, model inversion, backdoor injection). We claim this boundary is *differently* secure, not *more* secure.

9.3 QNOS—From Firmware to Operating System

QNOS (13,147 LOC implemented) extends QUEFI toward a full operating system: quaternion-structured IPC, UI framework (77 layout tests), process scheduling with health scores, and watchdog validation. The immediate development priority is migrating QNOS to Ring 3.

9.4 Media Processing (Speculative)

Quaternion algebra naturally maps to 4-component data (RGBA, spatial-temporal coordinates). Whether QUEFI could process media with better inter-channel correlation preservation is **untested**.

10. Limitations

We document limitations without euphemism.

1. **97.2% classical code.** QNCORE is 1,850 of 96,689 LOC (1.9%). QBUS adds 858 LOC. Everything else is conventional C and x86 assembly.
2. **Task-level evaluation is simulated.** The fault detection experiment uses synthetic device states, not real hardware faults. Whether QNCORE outperforms a hand-coded threshold on real hardware is unproven.
3. **Aggregate state is lossy.** The `combined_state()` quaternion discards individual device details by design. Information preservation has not been quantified per fault type.
4. **Scaling limits.** Quaternion speed advantage diminishes: 34% at 4q→8q, parity at 16q→16q. Larger layers may be slower without quaternion-specific hardware.
5. **Critical security gaps.** All code runs in Ring 0. No IOMMU. TLS lacks certificate validation. AES is not constant-time. Ring 3 prototype exists but is not active.
6. **No weight module ecosystem.** The "programs as weights" vision has five unsolved prerequisites. No weight module has been trained for any useful task.
7. **No real-hardware latency comparison.** The $3.6\times$ speedup is an architectural model, not a benchmark result.
8. **Single-developer implementation.** Independent code review is limited. Subtle bugs in 96K LOC are likely.

11. Related Work

Quaternion Neural Networks. Parcollet et al. [3] introduced quaternion recurrent networks for speech. Gaudet & Maida [2] established the $4\times$ parameter reduction. Zhu et al. [8] applied quaternion convolutions to vision. These works use quaternions as application-layer NNs on conventional operating systems. QUEFI applies the same primitives inside the kernel for device state representation.

AI-Integrated Systems. Microsoft Copilot+ PC, Apple Intelligence, and Google on-device AI integrate neural networks with operating systems through user-space APIs.

QUEFI differs by placing QNCORE inside the kernel loop with direct QBUS access.

Firmware Architectures. UEFI [9] and coreboot use scalar data structures for hardware state. No prior firmware system uses algebraic structures for hardware status encoding.

SSE/SIMD for Neural Networks. SSE Hamilton implementations exist in game engines. QNCORE repurposes the same SIMD instructions for neural layer computation in a kernel context.

Self-citation note. Our Q-Jamba work [10] is unpublished concurrent work and does not constitute independent validation.

12. Conclusion

We present QUEFI and QNCORE—a quaternion firmware interface with an integrated neural core for hardware state representation. The system is implemented (96,689 LOC), boots on real hardware, and all benchmarks pass.

What we demonstrate:

- A quaternion bus (QBUS) that represents device state as quaternion vectors and aggregates them via Hamilton product—a novel application to firmware interfaces.
- An in-kernel neural engine with SSE-accelerated Hamilton layers achieving $4\times$ parameter reduction applied to a new domain.
- Speed advantages of 18–34% at operating-size layers, diminishing to parity at 16q.
- A quaternion watchdog that validates all neural outputs before hardware dispatch.
- Chi(4) initialization producing $\kappa = 1.00$.
- **Task-level validation:** 94.0% fault detection at $17\times$ fewer parameters (§7).
- **Aggregation ablation:** $65\times$ higher class separability than element-wise mean at equal dimensionality.
- **Latency model:** $3.6\times$ speedup from eliminating syscall/context-switch overhead.

What remains open:

- Whether simulated fault detection transfers to real hardware failure modes.
- Whether weight modules can replace conventional programs for any real workload.
- Whether the latency model's $3.6\times$ advantage holds under real-world conditions.
- Whether a QNN-enforced security boundary provides meaningful protection against adversarial ML attacks.

The contribution is architectural and empirical: we show that quaternion algebra can serve as the representational substrate connecting hardware drivers to a neural decision engine within a kernel, that Hamilton aggregation produces structurally superior representations for fault detection, and that in-kernel placement eliminates measurable overhead. The system is a working prototype—whether this architecture is *better* than conventional approaches at scale is the question that deployment-level evaluation must address.

References

[1] William Rowan Hamilton. On quaternions; or on a new system of imaginaries in algebra. *Philosophical Magazine*, 1843.

[2] Chase J. Gaudet and Anthony S. Maida. Deep quaternion networks. In *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2018.

[3] Titouan Parcollet, Mirco Ravanelli, Mohamed Morchid, Georges Linarès, Chiheb Trabelsi, Renato De Mori, and Yoshua Bengio. Quaternion recurrent neural networks. In *International Conference on Learning Representations (ICLR)*, 2019.

[4] Aston Zhang et al. Beyond fully-connected layers with quaternions: Parameterization of hypercomplex multiplications with $1/n$ parameters. In *International Conference on Learning Representations (ICLR)*, 2021.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *International Conference on Computer Vision (ICCV)*, 2015.

[6] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[7] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Experimental Computer Science (ExpCS)*, 2007.

[8] Xuanyu Zhu, Yi Xu, Hongteng Xu, and Changjian Chen. Quaternion convolutional neural networks. In *European Conference on Computer Vision (ECCV)*, 2018.

[9] UEFI Forum. Unified extensible firmware interface specification, version 2.9. Technical report, UEFI Forum, 2021.

[10] Aleksej Dzigirej. Q-jamba: Quaternion-native hybrid state-space language models. Unpublished concurrent work, 2026.

Table 14. Source code metrics.

Component	LOC	Description
QUEFI (total)	36,289	Quaternion firmware interface
QUEFI/drivers	23,423	28 hardware drivers
QUEFI/hal	4,746	Hardware abstraction layer
QUEFI/net	4,318	Network stack + TLS 1.3
QUEFI/core	1,601	QBus + capability + registry
QUEFI/crypto	928	AES, SHA-256, X25519
QUEFI/boot+fs+service	1,273	Boot bridge, ramdisk, service
Kernel	24,009	MMU, scheduler, interrupts
QNCORE	1,850	SSE Hamilton fwd/bwd/Hebbian
QNOS	13,147	UI, dialog, semantic IPC
Tests	16,046	13 QUEFI + infrastructure suites
Other (asm, linker, headers)	5,348	Boot assembly, GDT/IDT, linker
Total	**96,689**	**Bare-metal x86 system**

A. Reproducibility

```
# Build QUEFI system
make clean && make os.iso

# Run all test suites (GCC + ASan/UBSan)
cd tests && make test-all

# Run 29 paper benchmarks (100K iterations each)
gcc -O2 -msse -msse2 -o bench_paper bench_paper_suite.c -lm
./bench_paper

# Boot in QEMU
make run
```

System requirements: x86-32 CPU with SSE support, Multiboot-compliant bootloader (GRUB, included), ≥ 64 MB RAM.