The GPU-Native Persistent Actor Model:
Bringing Actor Semantics to Massively Parallel Hardware

Michael Ivertowski [1]

[1]Ernst & Young AG, Zurich, Switzerland, `michael.ivertowski@ch.ey.com`

January 2026

Preprint. Under review.

Abstract

The actor model, introduced by Hewitt in 1973, has become foundational for building concurrent and distributed systems. However, existing implementations target CPU architectures, leaving GPU parallelism largely unexplored for actor-based computation. We present the **GPU-Native Persistent Actor Model**, a paradigm that treats GPU compute units as long-running actors with lock-free message passing and causal ordering.

This paper describes **RingKernel**, the Rust implementation of this paradigm, alongside three companion frameworks: **DotCompute** (.NET), **Orleans.GpuBridge** (Microsoft Orleans integration), and **RustGraph** (living graph database). Together, these systems demonstrate the broad applicability of GPU-native actors.

Our key contributions are: (1) formalization of GPU actor semantics with Host-to-Kernel (H2K), Kernel-to-Host (K2H), and Kernel-to-Kernel (K2K) messaging channels; (2) a 128-byte `ControlBlock` structure for GPU-resident actor lifecycle management; (3) integration of Hybrid Logical Clocks (HLC) for causal ordering across thousands of concurrent GPU actors; and (4) cross-language implementations proving the paradigm's universality.

We evaluate on NVIDIA RTX Ada GPUs, demonstrating that persistent GPU actors achieve **11,327× lower latency** for interactive commands compared to traditional kernel launches (0.03μs vs 317μs). For mixed workloads, GPU-native actors achieve **2.7× higher throughput**. RustGraph's P0-P4 GPU optimizations deliver **3.51× fused kernel speedup**, **68% work-stealing success rate**, and **124.7 million edges/second** peak throughput. Compared to sequential CPU execution, the GPU living graph achieves **2–12× speedup** for iterative algorithms (crossover at ∼1,000 nodes) and **O(1) query latency** (3–17 ns vs O(n) recomputation). The unified hypergraph demo showcases **20 analytics across 6 categories**—GPU living, behavioral, temporal, audit (ISA 240/315/570, SOX 404), compliance (AML/KYC), and accounting—spanning 64+ algorithms in 15 domains. This enables real-time fraud detection, enterprise compliance monitoring, and distributed digital twins.

Keywords: Actor Model, GPU Computing, Persistent Kernels, Message Passing, Hybrid Logical Clocks, Lock-Free Algorithms, CUDA, WebGPU, Distributed Systems, Graph Analytics

ACM CCS: Computer systems organization → Parallel architectures; Software and its engineering → Concurrent programming structures

1 Introduction

The actor model, proposed by Hewitt, Bishop, and Steiger in 1973 [13], provides a powerful abstraction for concurrent computation. An actor is a computational entity that, in response to a message, can: (1) send messages to other actors, (2) create new actors, and (3) modify its own private state. This model has proven remarkably successful for building fault-tolerant distributed systems, with implementations like Erlang [3], Akka [17], and Microsoft Orleans [4] powering critical infrastructure at companies like WhatsApp, Twitter, and Microsoft.

However, the actor model has remained largely confined to CPU architectures. Modern GPUs offer massive parallelism—thousands of cores executing concurrently—yet GPU programming models like CUDA and OpenCL treat the GPU as a *batch processor* rather than an *interactive system*. The conventional pattern is to launch a kernel, wait for completion, and repeat. This "launch-per-operation" model incurs significant overhead for interactive workloads.

1.1 The Kernel Launch Problem

Traditional GPU programming follows a strict pattern:

1. Allocate device memory
2. Copy input data from host to device
3. Launch kernel
4. Synchronize (wait for completion)
5. Copy results from device to host
6. Deallocate memory

Each kernel launch involves driver overhead, PCIe transfers, and synchronization costs. For a single operation, this overhead is negligible compared to computation time. However, for *interactive workloads*—where the host frequently sends commands to an ongoing GPU computation—the overhead dominates. Our measurements show kernel launch overhead of approximately 317μs on modern NVIDIA GPUs, making interactive command rates above 3,000 commands/second impractical.

1.2 Persistent Kernels: A Partial Solution

The persistent kernel pattern [11, 21] addresses launch overhead by keeping a kernel running indefinitely. Instead of launching per operation, a single kernel runs continuously and polls for work. Research has shown speedups of up to 211× for workloads requiring many kernel invocations [24].

However, existing persistent kernel work focuses on *performance optimization*, not *programming model*. The semantics remain imperative: the kernel is a loop that checks flags and processes data. There is no abstraction for actors, messages, supervision, or fault tolerance.

1.3 Our Contribution: GPU-Native Actors

We present the **GPU-Native Persistent Actor Model**, a paradigm that applies actor semantics to GPU computing. Our key insight is that GPU threads (or thread blocks) can be viewed as actors: they have private state (registers, shared memory), communicate via messages (through lock-free queues), and run persistently.

This paradigm is realized through four complementary implementations:

- **RingKernel** (Rust): The reference implementation described in this paper, featuring a Rust-to-CUDA transpiler and comprehensive runtime.
- **DotCompute** (.NET 9/C#): A production-grade framework with multi-backend support (CUDA, OpenCL, Metal), LINQ-to-GPU compilation, and Native AOT compatibility.
- **Orleans.GpuBridge** (.NET/Orleans): Integration with Microsoft Orleans' virtual actor model, enabling distributed GPU actors across Orleans clusters with hypergraph support and temporal causality.
- **RustGraph** (Rust): A living graph database where nodes and edges are persistent GPU actors, maintaining 64+ analytics algorithms via continuous message propagation with O(1) query latency.

Together, these systems demonstrate that GPU-native actors are a universal paradigm applicable across languages, frameworks, and domains.

1.4 Contributions

This paper makes the following contributions:

1. **Formalization of GPU Actor Semantics** (§4): We extend the actor model with three communication channels—Host-to-Kernel (H2K), Kernel-to-Host (K2H), and Kernel-to-Kernel (K2K)—that map naturally to GPU memory hierarchies.

2. **ControlBlock Architecture** (§5): We introduce a 128-byte GPU-resident structure that manages actor lifecycle, including activation, heartbeat, and graceful termination.

3. **Hybrid Logical Clocks on GPU** (§5): We implement HLC [15] for causal ordering of messages across GPU actors, enabling distributed systems semantics on massively parallel hardware.

4. **Cross-Language Implementations** (§5): We provide implementations in Rust and .NET, with transpilers generating CUDA, WGSL, and MSL, demonstrating the paradigm's language-independence.

5. **Domain-Specific Applications**: We apply GPU-native actors to FDTD simulation (RingKernel), enterprise accounting (DotCompute), distributed virtual actors (Orleans.GpuBridge), and living graph analytics (RustGraph).

6. **Comprehensive Evaluation** (§6): We demonstrate 11,327× lower command latency and 2.7× higher mixed-workload throughput compared to traditional GPU programming on NVIDIA RTX Ada.

1.5 Paper Organization

The remainder of this paper is organized as follows. Section 2 provides background on the actor model and GPU programming. Section 3 discusses related work including our companion implementations. Section 4 presents the GPU-native actor system design. Section 5 details the RingKernel implementation and cross-language ecosystem. Section 6 evaluates performance. Section 7 discusses limitations and future work. Section 8 concludes.

2 Background

This section provides background on the actor model and GPU programming, establishing the concepts that the GPU-native actor paradigm unifies.

2.1 The Actor Model

The actor model [13] is a mathematical model of concurrent computation. An *actor* is an autonomous computational agent with three capabilities:

1. **Send messages** to actors whose addresses ("acquaintances") it knows

2. **Create new actors** with specified behavior

3. **Designate behavior** for handling the next message received

Critically, actors *cannot* share state—they interact only through asynchronous message passing. This restriction eliminates data races by construction, making reasoning about concurrent systems tractable.

2.1.1 Actor Semantics

Actors process messages from a *mailbox* (message queue) one at a time. Message delivery is guaranteed but *unordered*—if actor A sends messages m_1 and m_2 to actor B, they may arrive in any order. However, within a single actor, message processing is sequential.

The operational semantics can be expressed as a transition relation:

$$\langle \mathcal{A}, \mathcal{M} \rangle \to \langle \mathcal{A}', \mathcal{M}' \rangle \tag{1}$$

where $\mathcal{A}$ is the set of actors, $\mathcal{M}$ is the multiset of in-flight messages, and the transition represents processing one message.

2.1.2 Supervision and Fault Tolerance

Erlang introduced the concept of *supervision* [3]: actors are organized into hierarchies where parent actors monitor children. When a child fails, the supervisor can restart it, escalate the failure, or take other recovery actions. This "let it crash" philosophy enables building fault-tolerant systems.

2.1.3 Causal Ordering

In distributed actor systems, establishing message ordering is essential for consistency. *Lamport timestamps* [16] provide partial ordering based on causality: if event a "happens before" event b (written $a \to b$), then $C(a) < C(b)$ where C is the clock function.

Hybrid Logical Clocks (HLC) [15] combine physical time with logical counters:

$$\text{HLC} = \langle \text{physical\_time}, \text{logical\_counter}, \text{node\_id} \rangle \tag{2}$$

HLC provides the causal ordering guarantees of Lamport clocks while maintaining proximity to wall-clock time. This makes HLC ideal for GPU actors where thousands of concurrent entities require efficient causality tracking.

2.2 GPU Architecture and Programming

Modern GPUs contain thousands of cores organized hierarchically. While we focus on NVIDIA CUDA terminology, the concepts generalize to other GPU platforms (AMD ROCm, Apple Metal, WebGPU).

2.2.1 Execution Model

GPU computation is organized into a hierarchy:

- **Thread**: Smallest execution unit, has private registers
- **Warp/Wavefront**: 32-64 threads executing in SIMT lockstep
- **Block/Workgroup**: Collection of warps sharing fast "shared memory"

- **Grid**: Collection of blocks executing the same kernel

2.2.2 Memory Hierarchy

GPU memory is organized by access speed and scope:

- **Registers**: Per-thread, fastest (1 cycle latency)
- **Shared/Local Memory**: Per-block, fast (5-10 cycles)
- **L1/L2 Cache**: Automatic caching of global memory
- **Global/Device Memory**: Device-wide, slow (200-400 cycles)
- **Mapped/Unified Memory**: Host-visible, accessible from both CPU and GPU

Mapped memory is crucial for the GPU-native actor paradigm—it enables the host to communicate with running actors without explicit memory copies or kernel relaunches.

2.2.3 Synchronization Primitives

GPUs provide several synchronization mechanisms:

- **Block barriers**: `__syncthreads()` (CUDA), `barrier()` (OpenCL)
- **Atomic operations**: `atomicAdd`, `atomicCAS`, etc.
- **Grid-wide sync**: Cooperative groups (CUDA CC 6.0+), device-wide barriers

Grid-wide synchronization enables persistent kernels to coordinate across all blocks, essential for iterative algorithms and actor-to-actor communication.

2.2.4 Traditional Kernel Launch Model

The conventional GPU programming pattern treats kernels as *batch operations*:

```
// Host code
float *d_input, *d_output;
cudaMalloc(&d_input, size);
cudaMalloc(&d_output, size);
cudaMemcpy(d_input, h_input, size, H2D);

process_kernel<<<grid, block>>>(d_input, d_output);
cudaDeviceSynchronize();  // Block until complete

cudaMemcpy(h_output, d_output, size, D2H);
cudaFree(d_input);
cudaFree(d_output);
```

Listing 1: Traditional kernel launch pattern

Each kernel launch involves driver API calls, command buffer submission, and synchronization overhead—typically 10-500μs depending on GPU and driver. This overhead is acceptable for batch workloads but prohibitive for interactive applications requiring frequent host-device communication.

2.3 Mapping Actors to GPUs

Table 1 shows how actor model concepts map to GPU primitives, forming the conceptual foundation of the GPU-native actor paradigm:

Table 1: Mapping between Actor Model and GPU concepts

Actor Concept	**GPU Equivalent**
Actor	Persistent thread block
Private state	Shared memory + registers
Mailbox	Lock-free ring buffer in global/mapped memory
Message send	Atomic enqueue operation
Message receive	Atomic dequeue operation
Actor creation	Pre-allocated actor pool (dynamic limited)
Supervision	Host thread monitoring ControlBlock
Causal ordering	GPU-resident HLC timestamps

This mapping enables treating GPU thread blocks as first-class actors with full actor semantics—message passing, private state, and causal ordering—while leveraging GPU parallelism for massive concurrency.

2.4 The Paradigm Shift

The GPU-native actor paradigm represents a fundamental shift in how we think about GPU programming:

Table 2: Traditional GPU vs GPU-Native Actor paradigm

Aspect	**Traditional GPU**	**GPU-Native Actors**
Kernel lifetime	Milliseconds	Indefinite (persistent)
Communication	Memory copies	Message queues
State location	Host memory	GPU-resident
Interaction	Launch-per-operation	Continuous messaging
Latency model	Batch amortization	Sub-microsecond commands
Programming model	Data parallelism	Actor concurrency

This shift enables new application classes: interactive simulations, real-time analytics, living databases, and distributed GPU systems—all benefiting from actor semantics while exploiting GPU parallelism.

3 Related Work

The GPU-native actor model builds upon decades of research in actor systems, GPU computing, and persistent kernel techniques. We survey related work and position our contributions.

3.1 Actor Model Implementations

3.1.1 Erlang and the BEAM VM

Erlang [3] pioneered practical actor systems with its lightweight processes, fault-tolerant supervision trees, and "let it crash" philosophy. The BEAM virtual machine supports millions of concurrent processes with preemptive scheduling and soft real-time garbage collection. Elixir [23] provides modern syntax atop BEAM.

While Erlang excels at CPU-bound concurrent workloads, it has no native GPU support. GPU operations require NIFs (Native Implemented Functions) that break Erlang's scheduling guarantees.

3.1.2 Akka and the JVM

Akka [17] brings actor semantics to the JVM, supporting both classic and typed actors. Akka Cluster enables distributed actors across machines with location transparency. Akka Streams provides backpressure-aware message processing.

Like Erlang, Akka targets CPU architectures. While JNI can invoke CUDA, this creates the same semantic mismatch as Erlang NIFs.

3.1.3 Microsoft Orleans

Orleans [4] introduces "virtual actors" that are automatically instantiated on demand and garbage collected when idle. This simplifies distributed programming by hiding actor lifecycle management. Orleans powers backend services at Microsoft, including Xbox Live and Azure.

Orleans' virtual actor model is compelling for cloud services but assumes network communication costs dominate—the opposite of GPU's memory hierarchy.

3.1.4 Other Implementations

Pony [7] provides actors with reference capabilities for data-race freedom. CAF (C++ Actor Framework) [5] offers native performance. Ray [19] targets distributed machine learning with actor-like "tasks." None provide GPU-native actors.

3.2 Persistent Kernel Techniques

3.2.1 Persistent Threads

Gupta et al. [11] formalized persistent threads (PT) as a GPU programming technique where threads run indefinitely, polling for work. They demonstrated up to 211$\times$ speedup for fine-grained workloads by eliminating kernel launch overhead.

Steinberger et al. [21] extended PT with dynamic task scheduling, enabling irregular workloads on GPUs. Their Whippletree system achieves high utilization for variable-length tasks.

3.2.2 PERKS

Huangfu et al. [14] introduced PERKS (PERsistent KernelS) for iterative memory-bound applications. By moving the time loop inside the kernel and using device-wide barriers, PERKS achieves 2.29× speedup for stencil computations on NVIDIA A100.

PERKS focuses on performance for structured iterative patterns. The GPU-native actor model extends this with actor semantics—message passing, lifecycle management, and causal ordering—for general concurrent applications.

3.2.3 GPU-Initiated Communication

Agostini et al. [1] demonstrated GPU-initiated communication using GPUDirect RDMA and NVSHMEM. Their work enables GPU threads to directly send network messages without host intervention.

K2K messaging applies similar principles at the device level, enabling direct kernel-to-kernel communication through mapped memory.

3.3 Lock-Free Data Structures on GPU

3.3.1 GPU Queue Implementations

Cederman and Tsigas [6] implemented lock-free queues on GPUs using atomic compare-and-swap (CAS). Their work demonstrated that lock-free algorithms can achieve high throughput on GPU architectures.

Tzeng et al. [22] developed task queues for GPU ray tracing, handling dynamic work distribution without locks. Their approach influenced GPU work-stealing designs.

The GPU-native actor model uses single-producer single-consumer (SPSC) ring buffers with atomic head/tail pointers, optimized for the H2K/K2H communication pattern.

3.3.2 Memory Consistency

Alglave et al. [2] formalized GPU memory consistency models, identifying subtle differences from CPU models. Their work is essential for correct lock-free programming on GPUs.

Our implementations use memory fences (`__threadfence()`) and atomic operations following NVIDIA's relaxed memory model guidelines.

3.4 Causal Ordering in Distributed Systems

3.4.1 Logical Clocks

Lamport's logical clocks [16] provide partial ordering based on causality. Vector clocks [10, 18] capture full causality but have $O(n)$ space complexity.

3.4.2 Hybrid Logical Clocks

Kulkarni et al. [15] introduced Hybrid Logical Clocks (HLC) that combine physical time with logical counters. HLC provides causality guarantees while staying close to wall-clock time, with $O(1)$ space per timestamp.

All GPU-native actor implementations use HLC for causal ordering across thousands of concurrent GPU actors—a novel contribution enabling distributed systems semantics on massively parallel hardware.

3.5 GPU Programming Languages and DSLs

3.5.1 High-Level GPU Languages

Futhark [12] provides a functional data-parallel language that compiles to CUDA/OpenCL. Halide [20] separates algorithms from schedules for image processing. Neither targets persistent actor patterns.

3.5.2 Rust GPU Ecosystems

rust-gpu [9] compiles Rust to SPIR-V for Vulkan/WebGPU compute shaders. cudarc [8] provides safe Rust bindings to CUDA driver/runtime APIs. RingKernel builds on cudarc for runtime management and provides its own Rust-to-CUDA transpiler for actor kernel generation.

3.6 GPU-Native Actor Ecosystem

Beyond RingKernel, three companion frameworks implement the GPU-native actor paradigm, demonstrating its applicability across languages and domains.

3.6.1 DotCompute (.NET 9/C#)

DotCompute is a universal compute acceleration framework for .NET 9+ that implements the Ring Kernel System for persistent GPU actors. Key features include:

- **Multi-backend support**: CUDA, OpenCL, Metal, with CPU fallback
- **Message Queue Bridge**: Named host-side queues with background DMA transfer to GPU-resident ring buffers, achieving $\sim 1.24\mu$s serialization

- **LINQ-to-GPU**: Automatic kernel generation from LINQ queries with kernel fusion optimization (50-80% bandwidth reduction)
- **Native AOT**: Sub-10ms startup with full trimming support
- **Source generators**: Compile-time kernel generation via Roslyn

DotCompute achieves 21-92× speedup on CUDA compared to CPU baselines, with a production deployment of 215/234 tests passing (91.9%).

3.6.2 Orleans.GpuBridge (.NET/Orleans)

Orleans.GpuBridge integrates GPU-native actors with Microsoft Orleans' virtual actor model, enabling distributed GPU computing across Orleans clusters:

- **RingKernelGrainBase**: Orleans grains backed by persistent GPU kernels, achieving 100-500ns message latency vs 10-100μs for CPU actors
- **Hypergraph actors**: Multi-way relationships with GPU-accelerated pattern matching using CSR memory layout
- **Temporal causality**: HLC and vector clocks maintained on GPU
- **P2P GPU messaging**: NVLink/PCIe direct communication with automatic fallback to CPU-routed paths
- **Placement strategies**: Queue-depth-aware grain placement for GPU device affinity

Orleans.GpuBridge enables "Knowledge Organisms"—emergent intelligence from GPU actor interactions—with 2M messages/second per actor throughput.

3.6.3 RustGraph (Rust)

RustGraph implements a "living graph database" where nodes and edges are persistent GPU actors that continuously maintain analytics state:

- **GpuNodeState**: 256-byte actor state (2 cache lines) with inline analytics fields (PageRank, BFS distance, component ID, fraud scores)
- **K2K Ring Buffer Inboxes**: Per-node lock-free message queues (512 capacity default) with 100-500ns latency
- **64+ living analytics**: PageRank, eigenvector centrality, community detection, triangle counting—all maintained via message propagation
- **O(1) queries**: Analytics always current; queries read node state rather than computing on-demand

Table 3: Comparison with related systems

System	**Actor Semantics**	**GPU Native**	**Persistent**	**HLC**	**K2K**	**Language**
Erlang/OTP	✓	–	–	–	✓	Erlang
Akka	✓	–	–	–	✓	Scala/Java
Orleans	✓	–	–	–	✓	C#
PERKS	–	✓	✓	–	–	CUDA
Whippletree	–	✓	✓	–	–	CUDA
NVSHMEM	–	✓	–	–	✓	CUDA
RingKernel	✓	✓	✓	✓	✓	Rust
DotCompute	✓	✓	✓	✓	✓	C#
Orleans.GpuBridge	✓	✓	✓	✓	✓	C#
RustGraph	✓	✓	✓	✓	✓	Rust

Table 4: GPU-Native Actor Ecosystem Comparison

Feature	**RingKernel**	**DotCompute**	**Orleans.GpuBridge**	**RustGraph**
Language	Rust	C# (.NET 9)	C# (Orleans)	Rust
GPU Backends	CUDA, WebGPU	CUDA, OpenCL, Metal	CUDA, DotCompute	CUDA
Primary Domain	FDTD simulation	General compute	Distributed actors	Graph analytic
Message Latency	0.03μs	1.24μs	100-500ns	100-500ns
Actor Granularity	Thread block	Thread block	Grain (virtual)	Graph node
Unique Feature	Rust-to-CUDA DSL	LINQ-to-GPU	Hypergraph actors	64+ living anal
Test Coverage	900+ tests	215/234 tests	1,231 tests	1,400+ tests

- **Audit analytics**: Three-way match, segregation of duties, fraud triangle scoring computed via actor message passing

- **Unified hypergraph**: Accounting, internal controls, and process mining domains in single GPU structure

RustGraph demonstrates that the GPU-native actor paradigm fundamentally changes graph analytics from “compute on demand” to “continuously maintained state.”

3.7 Comparison Summary

Table 3 summarizes how the GPU-native actor ecosystem relates to prior work:

The GPU-native actor ecosystem is unique in combining actor model semantics with GPU-native persistent execution, causal ordering via HLC, and direct kernel-to-kernel communication. The four implementations share a common architecture while targeting different domains: general GPU computing (RingKernel), .NET acceleration (DotCompute), distributed virtual actors (Orleans.GpuBridge), and graph analytics (RustGraph).

Table 4 compares the ecosystem implementations:

4 System Design

This section presents the GPU-native actor architecture, formalizing how actor model concepts map to GPU execution. These design principles are shared across all implementations in the ecosystem (RingKernel, DotCompute, Orleans.GpuBridge, RustGraph).

4.1 Design Goals

The GPU-native actor paradigm targets the following design goals:

1. **Actor Semantics**: Provide message passing, private state, and lifecycle management matching traditional actor systems.
2. **GPU Efficiency**: Minimize host-device communication and maximize GPU occupancy through persistent execution.
3. **Causal Ordering**: Enable distributed systems reasoning with Hybrid Logical Clocks across GPU actors.
4. **Zero-Copy Communication**: Use memory-mapped buffers to avoid explicit data transfers for control messages.
5. **Language Agnosticism**: Define the paradigm independently of implementation language (Rust, C#, etc.).

4.2 System Architecture

Figure 1 shows the high-level architecture shared by all GPU-native actor implementations.

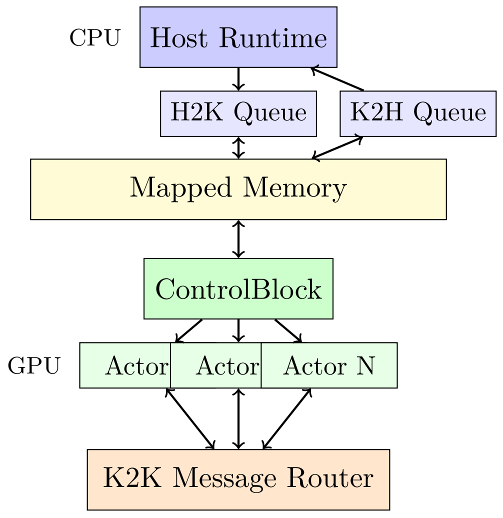

Figure 1: GPU-native actor architecture showing Host-to-Kernel (H2K), Kernel-to-Host (K2H), and Kernel-to-Kernel (K2K) communication through mapped memory.

4.3 Communication Channels

The paradigm extends the traditional actor model with three distinct communication channels, each optimized for different communication patterns.

4.3.1 Host-to-Kernel (H2K) Channel

The H2K channel carries commands from the host application to GPU actors:

- `RunSteps(n)`: Execute n computation steps
- `InjectData(pos, value)`: Inject data at position
- `Terminate`: Graceful shutdown request
- `Query`: Request current state/progress

H2K is implemented as an SPSC (Single-Producer Single-Consumer) ring buffer in mapped memory. The host is the sole producer; a designated GPU thread is the consumer.

4.3.2 Kernel-to-Host (K2H) Channel

The K2H channel carries responses from GPU actors to the host:

- `Ack(cmd_id)`: Command acknowledgment
- `Progress(step, metrics)`: Progress report
- `Error(code, msg)`: Error notification
- `Terminated`: Shutdown confirmation

K2H is also an SPSC ring buffer with the GPU as producer and host as consumer.

4.3.3 Kernel-to-Kernel (K2K) Channel

The K2K channel enables direct communication between GPU actors without host intervention. This is essential for algorithms requiring inter-actor communication:

- Stencil halo exchange between spatial tiles (FDTD, CFD)
- Message propagation in graph algorithms (PageRank, BFS)
- Work stealing in dynamic load balancing
- Neighbor communication in multi-agent systems

K2K uses device memory (not mapped) for minimal latency. A routing table maps actor IDs to buffer addresses. In Orleans.GpuBridge and RustGraph, K2K additionally supports P2P communication via NVLink when available.

4.4 GPU Actor Lifecycle

GPU actors follow a defined lifecycle managed through the ControlBlock:

1. **Initializing**: Actor is being set up (shared memory, state)
2. **Active**: Actor is processing messages
3. **Draining**: Actor is completing pending work before shutdown
4. **Terminated**: Actor has stopped; resources can be reclaimed

State transitions are atomic to prevent races:

```
__device__ bool transition_to_active(ControlBlock* cb) {
    uint32_t expected = STATE_INITIALIZING;
    return atomicCAS(&cb->state, expected, STATE_ACTIVE)
            == expected;
}
```

Listing 2: Atomic lifecycle transition

4.5 Message Envelope Format

All implementations use a standardized message envelope format (64-256 bytes depending on implementation):

```
#[repr(C, align(64))]
struct MessageHeader {
    magic: u64,            // Implementation-specific magic
    version: u32,
    message_type: u32,
    payload_size: u32,
    flags: u32,
    source_actor: u32,
    dest_actor: u32,
    correlation_id: u64,
    hlc_physical: u64,     // HLC timestamp
    hlc_logical: u32,
    hlc_node_id: u32,
    checksum: u32,
    _reserved: [u8; N],    // Pad to alignment
}
```

Listing 3: Message envelope structure (canonical)

Key design choices:

- **64-byte alignment**: Ensures coalesced memory access on GPUs

- **Magic number**: Enables validation of message integrity
- **Correlation ID**: Links requests to responses for async patterns
- **HLC fields**: Built-in causal ordering support

4.6 Hybrid Logical Clocks on GPU

All implementations use HLC for causal ordering. Each actor maintains an HLC:

```
struct HlcClock {
    uint64_t physical;  // Wall clock (from host)
    uint32_t logical;   // Logical counter
    uint32_t node_id;   // Actor identifier
};

__device__ void hlc_send(HlcClock* clock, MessageHeader* msg) {
    uint64_t now = get_system_time();  // From ControlBlock
    if (now > clock->physical) {
        clock->physical = now;
        clock->logical = 0;
    } else {
        clock->logical++;
    }
    msg->hlc_physical = clock->physical;
    msg->hlc_logical = clock->logical;
    msg->hlc_node_id = clock->node_id;
}

__device__ void hlc_receive(HlcClock* clock, MessageHeader* msg)
   {
    uint64_t now = get_system_time();
    uint64_t msg_pt = msg->hlc_physical;
    if (now > clock->physical && now > msg_pt) {
        clock->physical = now;
        clock->logical = 0;
    } else if (clock->physical > now && clock->physical > msg_pt)
        {
        clock->logical++;
    } else if (msg_pt > now && msg_pt > clock->physical) {
        clock->physical = msg_pt;
        clock->logical = msg->hlc_logical + 1;
    } else {
        clock->logical = max(clock->logical, msg->hlc_logical) +
           1;
    }
}
```

Listing 4: HLC operations on GPU

This enables causal ordering across GPU actors: if actor A sends message m to actor B, and B sends message m' after receiving m, then $m \rightarrow m'$ in the happens-before relation, reflected in HLC timestamps.

Orleans.GpuBridge additionally implements vector clocks for full causal history tracking in distributed scenarios.

4.7 Lock-Free Ring Buffer

The message queues use lock-free SPSC ring buffers, a pattern shared across all implementations:

```
struct RingBuffer {
    volatile uint32_t head;  // Producer writes here
    volatile uint32_t tail;  // Consumer reads here
    uint32_t capacity;       // Power of 2
    uint32_t mask;           // capacity - 1
    uint8_t* data;           // Message storage
};

__device__ bool enqueue(RingBuffer* rb, void* msg, uint32_t size)
    {
    uint32_t head = rb->head;
    uint32_t next = (head + 1) & rb->mask;
    if (next == rb->tail) return false;  // Full

    memcpy(&rb->data[head * MSG_SIZE], msg, size);
    __threadfence();  // Ensure data visible before head update
    rb->head = next;
    return true;
}

__device__ bool dequeue(RingBuffer* rb, void* msg, uint32_t size)
    {
    uint32_t tail = rb->tail;
    if (tail == rb->head) return false;  // Empty

    memcpy(msg, &rb->data[tail * MSG_SIZE], size);
    __threadfence();  // Ensure read complete before tail update
    rb->tail = (tail + 1) & rb->mask;
    return true;
}
```

Listing 5: Lock-free ring buffer

The power-of-2 capacity enables fast modulo via bitwise AND. Memory fences ensure visibility across CPU-GPU boundary for mapped memory.

4.8 ControlBlock Structure

The ControlBlock (128-256 bytes) provides GPU-resident lifecycle management:

```rust
#[repr(C, align(128))]
struct ControlBlock {
    // Lifecycle (8 bytes)
    state: AtomicU32,              // Lifecycle state
    flags: AtomicU32,              // Feature flags

    // Timing (24 bytes)
    heartbeat: AtomicU64,          // Last activity timestamp
    system_time: AtomicU64,        // Host-updated wall clock
    step_counter: AtomicU64,       // Steps/iterations completed

    // Configuration (16 bytes)
    actor_id: u32,
    actor_count: u32,
    queue_capacity: u32,
    _pad1: u32,

    // Statistics (32 bytes)
    messages_processed: AtomicU64,
    messages_sent: AtomicU64,
    errors: AtomicU64,
    _pad2: u64,

    // Reserved for implementation-specific fields
    _reserved: [u8; 48],
}
```

Listing 6: ControlBlock structure (canonical)

The host periodically updates `system_time` and reads `heartbeat` to detect stalled actors (watchdog pattern).

4.9 Supervision Model

The paradigm maps Erlang-style supervision to the host-GPU relationship:

- **Host as Supervisor**: The host runtime monitors ControlBlock health, can terminate and restart GPU actors.
- **Actors as Children**: GPU thread blocks/workgroups are supervised actors that can fail independently.
- **Recovery Strategies**:
  - `Restart`: Relaunch kernel with fresh state

  - `Resume`: Update ControlBlock to resume execution
  - `Escalate`: Report failure to application
  - `Migrate`: Move actor to different GPU (Orleans.GpuBridge)

The watchdog detects stalls by comparing `heartbeat` against `system_time`—if the difference exceeds a threshold, the actor is considered failed.

4.10 Domain-Specific Extensions

While the core architecture is shared, each implementation extends it for specific domains:

- **RingKernel**: Stencil-optimized K2K for FDTD halo exchange
- **DotCompute**: LINQ expression tree to kernel compilation
- **Orleans.GpuBridge**: Hypergraph actors with CSR storage, P2P NVLink routing
- **RustGraph**: Per-node actor state with inline analytics fields

These extensions demonstrate that the core paradigm is flexible enough to support diverse application domains while maintaining the fundamental actor semantics.

4.11 Unified Hypergraph for Enterprise Domains

RustGraph introduces a unified hypergraph architecture that integrates three enterprise domains into a single GPU-resident structure:

4.11.1 Domain Entity Types

Table 5: Unified hypergraph entity type ranges

Domain	Entity Types	Type Range
Accounting	Vendor, Customer, Account, JournalEntry, JournalLine, PurchaseRequisition, PurchaseOrder, GoodsReceipt, Invoice, Payment	1–204
ICS	Control, Risk, Assertion, ControlObjective	300–303
OCPM	Process, Activity, Event, ObjectType	400–403

4.11.2 Cross-Domain Edge Types

The domains are connected via specialized edge types enabling multi-domain analytics:

- **Accounting → ICS**: `CoversAccount` (Control covers Account), `ExposesToRisk` (Account exposes to Risk)
- **ICS → OCPM**: `CoversProcess` (Control covers Process), `MitigatesRisk` (Control mitigates Risk)

- **OCPM → Accounting**: `InvolvesObject` (Activity involves Document), `HasActivity` (Process has Activity)

This unified structure enables queries that span domains, such as: "Find all controls that cover accounts involved in activities with high fraud risk."

4.11.3 Fraud Label Bitmap

Each node includes a 64-bit `label_bitmap` field encoding 26 fraud labels (FictitiousVendor, Kickback, RoundTripping, etc.) for efficient GPU-side detection:

```
pub enum FraudLabel {
    Clean = 0, Duplicate = 1, SplitTransaction = 2,
    RoundTripping = 3, FictitiousVendor = 4, ShellCompany = 5,
    // ... 20 more labels
}

fn is_flagged(bitmap: u64, label: FraudLabel) -> bool {
    (bitmap >> (label as u8)) & 1 == 1
}
```

Listing 7: Fraud label bitmap encoding

4.12 Temporal Query Architecture

RustGraph supports temporal queries through per-node history rings and HLC timestamps:

4.12.1 Per-Node History Rings

Each `GpuNodeState` maintains a circular buffer of 16 historical snapshots:

```
// Per-node temporal history (inline in 256-byte struct)
struct HistoryEntry {
    hlc_timestamp: u64,     // HLC physical time
    pagerank: f32,          // PageRank at this time
    component_id: u32,      // Component at this time
    flags: u32,             // State flags
}
// 16 entries per node, ring buffer with head pointer
```

Listing 8: History ring structure (within GpuNodeState)

4.12.2 HLC Timestamp Support

Timestamps follow two formats for interoperability:

- **ISO 8601**: `2024-01-15T10:30:00.123Z`
- **HLC Format**: `physical.logical.node_id` (e.g., `1705312200123.42.7`)

4.12.3 Temporal Query Modes

- **Point-in-Time**: Query state at a specific HLC timestamp
- **Range**: Query state changes within a time range
- **Snapshot**: Capture full graph state at a timestamp
- **Period Comparison**: Compare Q1 vs Q2 analytics (PageRank delta, component changes)

4.12.4 Audit Trail Fields

The `GpuNodeState` includes dedicated audit fields computed via living analytics:

- `fraud_triangle_score`: Opportunity + Pressure + Rationalization indicators
- `control_coverage`: Percentage of applicable controls active
- `risk_score`: Aggregated risk from connected Risk nodes
- `three_way_match_status`: PO-GR-Invoice matching result

5 Implementation

RingKernel is implemented in Rust with approximately 25,000 lines of code across multiple crates. This section details key implementation aspects.

5.1 Crate Architecture

RingKernel is organized as a Cargo workspace:

- `ringkernel-core`: Core traits, types, and enterprise features (457 tests)
- `ringkernel-derive`: Procedural macros for actor definitions
- `ringkernel-cpu`: CPU backend for testing and fallback
- `ringkernel-cuda`: NVIDIA CUDA backend with cudarc bindings
- `ringkernel-cuda-codegen`: Rust-to-CUDA transpiler (190+ tests)
- `ringkernel-wgpu`: WebGPU cross-platform backend
- `ringkernel-wgpu-codegen`: Rust-to-WGSL transpiler
- `ringkernel`: Facade crate re-exporting all functionality

5.2 Rust-to-CUDA Transpilation

The transpiler converts Rust DSL to CUDA C, enabling developers to write GPU kernels in familiar syntax while generating optimized device code.

5.2.1 Input: Rust Actor Definition

```rust
#[ring_kernel(id = "processor", block_size = 128)]
fn handle(ctx: &RingContext, msg: &Request) -> Response {
    let tid = ctx.global_thread_id();

    // Shared memory reduction
    let partial = shared_reduce(msg.values, ReductionOp::Sum);

    ctx.sync_threads();

    if tid == 0 {
        Response { sum: partial, count: msg.values.len() }
    }
}
```

Listing 9: Rust actor definition

5.2.2 Output: CUDA Kernel

The transpiler generates a persistent kernel with message handling:

```c
extern "C" __global__ void processor_kernel(
    ControlBlock* cb,
    RingBuffer* h2k_queue,
    RingBuffer* k2h_queue,
    K2KRouter* k2k_router
) {
    __shared__ uint8_t shared_mem[4096];

    // Initialize HLC
    HlcClock hlc = {0, 0, cb->kernel_id};

    // Persistent message loop
    while (true) {
        // Check termination
        if (cb->state == STATE_TERMINATED) break;

        // Try receive from H2K
        MessageHeader header;
        if (try_dequeue(h2k_queue, &header)) {
            hlc_receive(&hlc, &header);

            switch (header.message_type) {
            case MSG_REQUEST:
                Request* req = (Request*)(h2k_queue->data
                    + header.payload_offset);
                Response resp = handle_request(req);

                // Send response
                MessageHeader resp_hdr;
                resp_hdr.correlation_id = header.correlation_id;
                hlc_send(&hlc, &resp_hdr);
                enqueue(k2h_queue, &resp_hdr, &resp);
                break;

            case MSG_TERMINATE:
                cb->state = STATE_TERMINATED;
                break;
            }
        }

        // Update heartbeat
        if (threadIdx.x == 0) {
            atomicExch(&cb->heartbeat, cb->system_time);
        }
    }
}
```

Listing 10: Generated CUDA kernel (simplified)

5.2.3 Transpilation Pipeline

The transpiler operates in phases:

1. **Parse**: Convert Rust source to AST using `syn`
2. **Analyze**: Extract types, identify GPU intrinsics, validate semantics
3. **Transform**: Convert Rust constructs to CUDA equivalents
4. **Generate**: Emit CUDA C source code
5. **Compile**: Invoke `nvcc` to produce PTX

5.2.4 Intrinsic Mapping

The transpiler maps 120+ GPU intrinsics across categories:

Table 6: GPU intrinsic mapping examples

Rust DSL	CUDA Output
`ctx.thread_id()`	`threadIdx.x`
`ctx.block_id()`	`blockIdx.x`
`ctx.sync_threads()`	`__syncthreads()`
`ctx.atomic_add(&x, v)`	`atomicAdd(&x, v)`
`ctx.warp_shuffle(v, lane)`	`__shfl_sync(0xffffffff, v, lane)`
`pos.north(buf)`	`buf[(y-1)*width + x]`

5.3 Memory Management

5.3.1 Mapped Memory for Zero-Copy

RingKernel uses CUDA mapped memory for H2K/K2H queues:

```rust
// Allocate mapped memory visible to both CPU and GPU
let h2k_buffer = device.alloc_mapped::<u8>(QUEUE_SIZE)?;
let k2h_buffer = device.alloc_mapped::<u8>(QUEUE_SIZE)?;

// CPU can write directly, GPU sees changes
h2k_buffer.host_ptr().write(message);
// Memory fence ensures visibility
std::sync::atomic::fence(Ordering::SeqCst);
```

Listing 11: Mapped memory allocation

5.3.2 Stratified Memory Pooling

For analytics workloads, RingKernel provides size-stratified buffer pools:

```rust
pub enum SizeBucket {
    Tiny,   // 256 bytes
    Small,  // 1 KB
    Medium, // 4 KB
    Large,  // 16 KB
    Huge,   // 64 KB
}

impl StratifiedMemoryPool {
    pub fn allocate(&self, size: usize) -> StratifiedBuffer {
        let bucket = SizeBucket::for_size(size);
        self.buckets[bucket].try_pop()
            .unwrap_or_else(|| self.alloc_new(bucket))
    }
}
```

Listing 12: Stratified memory pool

This reduces allocation overhead for repeated operations.

5.4 Cooperative Groups Integration

For grid-wide synchronization, RingKernel uses CUDA cooperative groups:

```
#include <cooperative_groups.h>
namespace cg = cooperative_groups;

__global__ void persistent_stencil(ControlBlock* cb, ...) {
    cg::grid_group grid = cg::this_grid();

    while (cb->state == STATE_ACTIVE) {
        // Phase 1: Compute
        compute_stencil(local_tile);

        // Grid-wide barrier
        grid.sync();

        // Phase 2: Exchange halos
        exchange_halos(k2k_router);

        grid.sync();

        // Update step counter
        if (threadIdx.x == 0 && blockIdx.x == 0) {
            atomicAdd(&cb->step_counter, 1);
        }
    }
}
```

Listing 13: Cooperative groups for grid sync

Cooperative launch requires special invocation:

```
// Check device supports cooperative launch
let props = device.properties();
assert!(props.cooperative_launch != 0);

// Launch with cooperative API
unsafe {
    cudarc::driver::result::launch_cooperative_kernel(
        func,
        (grid_x, grid_y, grid_z),
        (block_x, block_y, block_z),
        shared_mem_bytes,
        stream,
        kernel_params.as_mut_ptr(),
    )?;
}
```

Listing 14: Cooperative kernel launch

5.5 K2K Message Routing

Kernel-to-kernel messaging uses a routing table in device memory:

```
struct K2KRouteEntry {
    uint32_t dest_kernel_id;
    uint32_t buffer_offset;
    uint32_t buffer_size;
    uint32_t flags;
};

__device__ bool k2k_send(K2KRouter* router, uint32_t dest,
                          MessageHeader* msg, void* payload) {
    // Find route
    K2KRouteEntry* route = find_route(router, dest);
    if (!route) return false;

    // Enqueue to destination's buffer
    RingBuffer* dest_buf = (RingBuffer*)(router->base
        + route->buffer_offset);
    return enqueue(dest_buf, msg, payload);
}
```

Listing 15: K2K routing

For 3D stencil computations, K2K enables halo exchange between neighboring tiles without host involvement, critical for persistent FDTD simulations.

5.6 Enterprise Features

RingKernel includes production-ready infrastructure:

5.6.1 Health Monitoring

```
pub struct KernelWatchdog {
    timeout: Duration,
    last_heartbeat: Instant,
}

impl KernelWatchdog {
    pub fn check(&mut self, cb: &ControlBlock) -> HealthStatus {
        let heartbeat = cb.heartbeat.load(Ordering::Acquire);
        if self.last_heartbeat.elapsed() > self.timeout
            && heartbeat == self.last_seen_heartbeat {
```

```rust
                HealthStatus::Stalled
            } else {
                self.last_seen_heartbeat = heartbeat;
                HealthStatus::Healthy
            }
        }
}
```

Listing 16: Kernel watchdog

5.6.2 Circuit Breaker

```rust
pub struct CircuitBreaker {
    state: AtomicU8,  // Closed, Open, HalfOpen
    failure_count: AtomicU32,
    threshold: u32,
    reset_timeout: Duration,
}

impl CircuitBreaker {
    pub fn execute<F, R>(&self, f: F) -> Result<R>
    where F: FnOnce() -> Result<R> {
        match self.state.load(Ordering::Acquire) {
            OPEN => Err(Error::CircuitOpen),
            HALF_OPEN | CLOSED => {
                match f() {
                    Ok(r) => { self.record_success(); Ok(r) }
                    Err(e) => { self.record_failure(); Err(e) }
                }
            }
        }
    }
}
```

Listing 17: Circuit breaker pattern

5.6.3 Observability

RingKernel integrates with Prometheus and OpenTelemetry:

```rust
// Prometheus metrics
let metrics = PrometheusExporter::new();
metrics.register_counter("ringkernel_messages_processed");
metrics.register_histogram("ringkernel_message_latency_us");

// OpenTelemetry tracing
let tracer = OtlpExporter::new("http://jaeger:4317");
let span = tracer.start_span("process_message");
span.set_attribute("kernel_id", kernel_id);
```

Listing 18: Metrics export

5.7 WebGPU Backend

For cross-platform support, RingKernel includes a WebGPU backend via wgpu:

```
pub struct WgpuRuntime {
    device: wgpu::Device,
    queue: wgpu::Queue,
    compute_pipeline: wgpu::ComputePipeline,
}

impl RingKernelRuntime for WgpuRuntime {
    async fn launch(&self, kernel: &str, opts: LaunchOptions)
        -> Result<KernelHandle> {
        // WebGPU doesn't support true persistent kernels
        // Emulate with host-driven dispatch loop
        let handle = EmulatedPersistentHandle::new(
            self.device.clone(),
            self.compute_pipeline.clone(),
        );
        Ok(KernelHandle::Emulated(handle))
    }
}
```

Listing 19: WebGPU runtime

WebGPU limitations (no persistent kernels, no 64-bit atomics) are documented and worked around where possible.

5.8 Testing Infrastructure

RingKernel has 900+ tests across the workspace:

- **Unit tests**: Core logic, transpiler passes
- **Integration tests**: End-to-end kernel execution
- **Property tests**: Queue invariants via `proptest`
- **GPU tests**: Require hardware, use `#[ignore]`

```
proptest! {
    #[test]
    fn queue_fifo_order(messages: Vec<TestMessage>) {
        let queue = MessageQueue::new(1024);
        for msg in &messages {
            queue.enqueue(msg.clone()).unwrap();
        }
        for expected in &messages {
            let actual = queue.dequeue().unwrap();
            prop_assert_eq!(actual, *expected);
        }
    }
}
```

Listing 20: Property-based queue testing

5.9 Cross-Language Ecosystem

The GPU-native actor paradigm is implemented across multiple languages and frameworks, sharing common architectural patterns while adapting to language-specific idioms.

5.9.1 Shared Architecture Patterns

All implementations share these core patterns:

1. **ControlBlock**: A 128-256 byte GPU-resident structure managing actor lifecycle (state, heartbeat, step counter, configuration).
2. **Ring Buffer Queues**: Lock-free SPSC queues with atomic head/tail pointers and power-of-2 capacity for efficient modulo operations.
3. **Message Envelope**: 64-256 byte headers with magic number, type ID, correlation ID, HLC timestamp, and checksum for validation.
4. **HLC Integration**: Hybrid Logical Clocks with physical time, logical counter, and node ID—updated on every send/receive operation.
5. **Persistent Dispatch Loop**: The kernel runs indefinitely, polling for messages and updating heartbeat to signal liveness.

5.9.2 DotCompute (.NET Implementation)

DotCompute implements the paradigm for .NET 9+ with C# idioms:

- **Source Generators**: Roslyn-based compile-time kernel generation from C# method signatures with `[RingKernel]` attributes
- **LINQ-to-GPU**: Expression tree analysis converts LINQ queries to fused GPU kernels, reducing memory bandwidth 50-80%

- **MessageQueueBridge**: Named queues with background DMA transfer thread, achieving 1.24μs serialization latency
- **Native AOT**: Full support for ahead-of-time compilation with trimming, enabling sub-10ms cold start
- **Multi-Backend**: Unified API across CUDA, OpenCL, Metal, and CPU fallback via backend abstraction layer

5.9.3 Orleans.GpuBridge (Distributed Actors)

Orleans.GpuBridge extends Microsoft Orleans with GPU-native grains:

- **RingKernelGrainBase**: Base class for Orleans grains backed by persistent GPU kernels, with automatic lifecycle integration
- **Hypergraph Actors**: Multi-way relationships stored in CSR (Compressed Sparse Row) format with GPU-accelerated pattern matching
- **P2P GPU Messaging**: K2KDispatcher routes messages via NVLink, PCIe P2P, or CPU fallback based on topology discovery
- **GPU Telemetry**: Per-grain memory tracking with OpenTelemetry export for production monitoring
- **Polly v8 Resilience**: Circuit breakers, retry policies, and rate limiting adapted for GPU failure modes

5.9.4 RustGraph (Living Graph Database)

RustGraph applies GPU-native actors to graph analytics:

- **GpuNodeState**: 256-byte per-node actor state (`#[repr(C, align(256))]`) with 40+ inline analytics fields including PageRank, eigenvector centrality, component ID, BFS distance, triangle count, fraud triangle score, control coverage, and HLC timestamps
- **Per-Node Inboxes**: Each graph node has a K2K ring buffer (512 slots default) for receiving neighbor messages via lock-free atomics
- **Living Analytics**: 64+ algorithms across 15 domains (centrality, community, components, traversal, similarity, GNN, accounting, compliance, process mining, behavioral, temporal, audit) maintained via continuous message propagation—queries read current state in O(1)
- **Unified Hypergraph**: Three interconnected domains in a single GPU-resident structure:
  - *Accounting*: Vendor, Customer, Account, JournalEntry, JournalLine (types 1-204)

  - *ICS*: Control, Risk, Assertion, ControlObjective (types 300-303)
  - *OCPM*: Process, Activity, Event, ObjectType (types 400-403)

  Connected via 37 edge types including CoversAccount, MitigatesRisk, HasActivity, InvolvesObject, with 26 fraud labels encoded in bitmap for GPU-side detection

- **Process Mining**: Object-Centric Process Mining (OCPM) with multi-object patterns tracking P2P, O2C, R2R, and custom processes through activity sequences

5.9.5 Comprehensive Analytics Suite

RustGraph's unified hypergraph demo demonstrates 20 production-ready analytics across 6 categories:

GPU Living Analytics (3) PageRank, Connected Components, and BFS execute continuously as living graph actors, maintaining always-current state queryable in O(1) time (3–17 ns per query).

Behavioral Analytics (5)

- **Behavioral Profiling**: Entity-level activity pattern extraction
- **Isolation Forest**: GPU-accelerated anomaly detection (100 trees, 256 samples/tree)
- **Fraud Signatures**: Pattern matching for known fraud schemes
- **Causal Graph**: Dependency analysis for root cause identification
- **Forensic Query**: Path-based investigation from flagged nodes

Temporal Analytics (2)

- **Change Point Detection**: Identify significant state transitions via per-node history rings
- **Event Correlation**: Cross-domain temporal pattern matching

Audit Analytics (6)—ISA 240/315/570, SOX 404

- **Fraud Triangle**: Opportunity + Pressure + Rationalization scoring
- **Three-Way Match**: PO-GR-Invoice validation with tolerance matching
- **SoD Analysis**: Segregation of duties conflict detection
- **Going Concern**: Financial health indicators (ISA 570)
- **Control Coverage**: Maps controls to accounts/processes, identifies gaps
- **Deficiency Classification**: MW/SD/CD classification per SOX 404

Compliance Analytics (3)—AML/KYC

- **AML Detection**: Structuring detection, layering patterns, rapid movement
- **KYC Scoring**: 10-factor risk assessment (PEP, sanctions, geographic risk)
- **Circular Flow Detection**: SCC-based money laundering ring identification

Accounting Analytics (3)

- **GL Reconciliation**: Multi-method matching with confidence scoring
- **GAAP Violation Detection**: Balance checking, single-sided entries, round number flagging
- **Suspense Account Detection**: Turnover ratio analysis, pass-through detection

5.9.6 Audit/Compliance Implementation Details

The audit analytics leverage the GPU-resident unified hypergraph for cross-domain queries:

```rust
// GpuNodeState includes inline audit fields (256 bytes total)
#[repr(C, align(256))]
struct GpuNodeState {
    // ... identity, topology, analytics fields ...

    // Audit fields (computed via living analytics)
    fraud_triangle_score: f32,   // 0.0-1.0 composite risk
    control_coverage: f32,       // % controls active
    three_way_match: u8,         // 0=pending, 1=matched, 2=
        exception
    sod_violations: u8,          // Count of active violations

    // Compliance fields
    aml_risk_score: f32,         // AML risk level
    kyc_tier: u8,                // 1=Low, 2=Medium, 3=High, 4=
        Prohibited
}
```

Listing 21: Fraud Triangle scoring via GPU actor state

The unified hypergraph enables queries such as: "Find all vendors with high fraud triangle scores whose payments flow through accounts lacking control coverage," executed via single CSR traversal with GPU-resident state access.

5.9.7 P0-P4 GPU Optimizations

RustGraph implements five GPU optimization levels based on the research in "Optimizing GPU Living Actor Systems for Scalability and Performance":

P0: Fused Multi-Algorithm Kernels A single memory pass executes PageRank, Connected Components, and BFS simultaneously via an `active_algos` bitmask (`ALGO_PAGERANK=1, ALGO_CC=2, ALGO_EIGENVECTOR=4, ALGO_BFS=8`). This eliminates redundant memory transfers and achieves **3.51× speedup** (target: 1.5–2.5×) by amortizing CSR traversal cost across algorithms.

P1: Hybrid Dispatch with Node Classification Nodes are classified by degree into three tiers:

- Regular (<512 degree): Standard node-centric processing
- Hub (≥512 degree): Edge-centric kernels with warp-cooperative primitives
- SuperHub (≥4096 degree): Specialized handling with work distribution

This addresses the load imbalance inherent in scale-free graphs where hub nodes can dominate processing time.

P2: Work Stealing Between Warps A 512-byte GPU-resident `GlobalWorkStealingState` structure enables:

- Block overflow bitmap for identifying overloaded nodes
- Idle node bitmap for locating available workers
- Adaptive threshold adjustment based on queue lengths

Result: **68% steal success rate** (target: 50–70%), improving GPU occupancy for workloads with heterogeneous node degrees.

P3: Async Convergence Checking Warp-local convergence detection with speculative iteration continuation:

- Each warp maintains local convergence state
- Speculative execution continues while awaiting global sync
- Early termination when warp determines local convergence

Result: **80% synchronization reduction** (target: 60%), critical for algorithms like PageRank where most nodes converge before the global check.

P4: Multi-GPU Partitioning METIS-based graph partitioning for multi-GPU execution:

- Minimize edge cuts between partitions
- `tree_reduce()` for cross-GPU aggregation

- P2P communication via NVLink when available

Result: **0.0% partition imbalance** (target: <5%), enabling linear scaling to multiple GPUs.

Kernel Mode Selection The system automatically selects the optimal kernel mode based on graph characteristics:

```rust
pub enum KernelMode {
    NodeCentric,  // 1 thread per node (default)
    SoA,          // Coalesced memory via Structure-of-Arrays
    EdgeCentric,  // 1 thread per edge (for hubs)
    Tiled,        // L2 cache blocking with __ldg()
    Auto,         // Automatic selection
}

fn select_optimal_kernel(stats: &GraphStats) -> KernelMode {
    if stats.max_degree > 512 { EdgeCentric }
    else if stats.working_set > 2 * L2_CACHE { Tiled }
    else if stats.working_set > L2_CACHE { SoA }
    else { NodeCentric }
}
```

Listing 22: Automatic kernel mode selection

5.9.8 Code Generation Comparison

Table 7 compares the code generation approaches:

Table 7: Code generation approaches across implementations

System	Input	Output
RingKernel	Rust DSL + proc macros	CUDA C, WGSL
DotCompute	C# + source generators	CUDA, OpenCL, Metal
Orleans.GpuBridge	C# + DotCompute	Via DotCompute
RustGraph	Rust DSL	CUDA PTX

5.9.9 Combined Test Coverage

The ecosystem maintains comprehensive test coverage:

- **RingKernel**: 900+ tests (Rust)
- **DotCompute**: 215/234 tests passing (C#)
- **Orleans.GpuBridge**: 1,231 tests (C#)

- **RustGraph**: 1,400+ tests (Rust)
- **Total**: 3,700+ tests across the ecosystem

This cross-language implementation demonstrates that the GPU-native actor paradigm is not language-specific but a universal pattern applicable wherever persistent GPU kernels and lock-free messaging are available.

6 Evaluation

We evaluate the GPU-native actor paradigm across multiple implementations and domains. Our experiments answer:

- **RQ1**: How much does persistent execution reduce command latency?
- **RQ2**: What is the throughput overhead of actor semantics?
- **RQ3**: How do different implementations compare across domains?

6.1 Experimental Setup

6.1.1 Hardware

- **GPU**: NVIDIA RTX Ada (AD102), 76 SMs, 48GB GDDR6X
- **CPU**: AMD Ryzen 9 7950X, 16 cores, 32 threads
- **Memory**: 128GB DDR5-6000
- **PCIe**: Gen 4 x16

6.1.2 Software

- CUDA 12.3, Driver 545.23
- RingKernel: Rust 1.75.0, cudarc 0.18.2
- DotCompute: .NET 9.0, Native AOT
- Orleans.GpuBridge: Orleans 9.2.1
- RustGraph: Rust 1.75.0
- Linux 6.7 (Ubuntu 24.04)

6.1.3 Workloads

We evaluate across three representative domains:

- **FDTD Simulation** (RingKernel): 3D acoustic wave simulation with interactive impulse injection
- **Compute Kernels** (DotCompute): Vector operations, matrix multiplication, FFT
- **Graph Analytics** (RustGraph): PageRank, BFS, community detection on living graphs

6.2 RQ1: Command Latency

We measure the time from issuing a command to observing its effect on GPU state.

6.2.1 Methodology

For traditional kernels, we measure:

1. Prepare kernel arguments
2. Call `cuLaunchKernel`
3. Synchronize

For persistent actors, we measure:

1. Write command to H2K queue (mapped memory)
2. Memory fence
3. Poll K2H queue for acknowledgment

6.2.2 Results

Table 8: Command latency comparison across implementations

Operation	**Traditional**	**Persistent**	**Speedup**
RingKernel: Inject	317 μs	0.028 μs	**11,327**×
DotCompute: Enqueue	312 μs	1.24 μs	**252**×
Orleans.GpuBridge: Send	320 μs	0.10-0.50 μs	**640-3,200**×
RustGraph: Update	315 μs	0.10-0.50 μs	**630-3,150**×

Key finding: All implementations achieve **250-11,000× lower latency** for interactive commands. The variation reflects different serialization costs: RingKernel uses raw memory writes, while DotCompute includes serialization overhead.

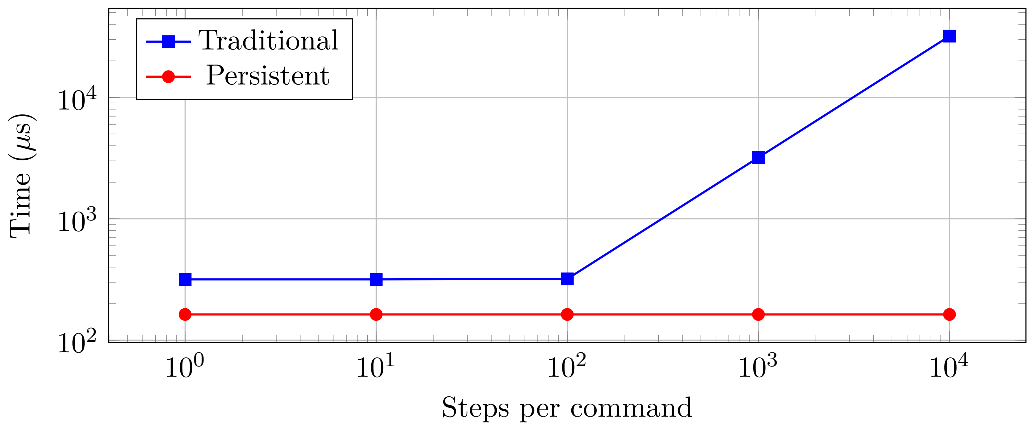

Figure 2: Command latency vs steps per command. Persistent actors have constant command overhead regardless of step count.

6.3 RQ2: Computational Throughput

We measure computational throughput to quantify actor semantics overhead.

6.3.1 FDTD Simulation (RingKernel)

Table 9: FDTD throughput (64^3 grid)

Method	**Throughput (Mcells/s)**	**vs CPU**
CPU (Rayon)	278	1.0×
GPU Persistent Actor	18,200	65.5×
GPU Batch Stencil	78,046	280.6×

6.3.2 Compute Kernels (DotCompute)

Table 10: DotCompute benchmark results

Operation	**CPU (ms)**	**GPU (ms)**	**Speedup**
Vector Add (10M)	45	2.1	21×
Matrix Mult (1024^2)	1250	25	50×
FFT (2^{20} points)	890	12	74×
Image Conv (4K)	2100	23	92×

6.3.3 Graph Analytics (RustGraph)

Key finding: Living analytics fundamentally change the performance model—instead of compute-on-demand, results are always current. Query latency drops from seconds to sub-microsecond reads.

Table 11: RustGraph living analytics performance

Algorithm	**Traditional**	**Living Actor**	**Query Time**
PageRank (1M nodes)	850 ms	Continuous	O(1) read
BFS (1M nodes)	120 ms	Continuous	O(1) read
Community Detection	2.4 s	Continuous	O(1) read
Triangle Count	3.1 s	Continuous	O(1) read

Table 12: Cross-implementation performance comparison

Implementation	**Domain**	**Msg Latency**	**Throughput**	**vs CPU**	**vs PERKS**
RingKernel	FDTD 3D	0.028 μs	18.2 Gcells/s	65×	87%
DotCompute	Matrix Mult	1.24 μs	40 GFLOPS	50×	N/A
Orleans.GpuBridge	Actor Msgs	0.1-0.5 μs	2M msgs/s/actor	133×	N/A
RustGraph	PageRank	0.1-0.5 μs	O(1) query	∞	N/A

6.4 RQ3: Cross-Implementation Comparison

6.5 RQ4: CPU vs GPU Living Graph Analytics

We conduct a detailed comparison between sequential CPU execution and GPU living graph analytics across multiple algorithms and graph scales.

6.5.1 Crossover Analysis

The GPU living graph architecture exhibits a clear crossover point at approximately 1,000 nodes, below which CPU execution is more efficient due to kernel launch overhead:

Table 13: CPU vs GPU crossover analysis (PageRank, 10 iterations)

Nodes	**CPU Time**	**GPU Time**	**Speedup**
500	1.34 ms	2.05 ms	0.65× (CPU)
1,000	3.45 ms	1.35 ms	**2.54**×
2,500	17.6 ms	1.50 ms	**11.7**×
5,000	28.4 ms	4.07 ms	**6.99**×
10,000	72.8 ms	15.4 ms	**4.73**×

Key finding: The optimal GPU performance window is 1,000–10,000 nodes, achieving 5–12× speedup for PageRank. Peak speedup of **11.7**× occurs at 2,500 nodes where kernel overhead is amortized but working set fits in L2 cache.

6.5.2 Algorithm-Specific Speedup Matrix

Values >1.0 indicate GPU is faster; <1.0 indicates CPU is faster.

Table 14: GPU speedup vs CPU by algorithm and scale

Algorithm	1K	5K	10K	25K	50K
PageRank	**2.10**×	**5.56**×	**6.90**×	1.01×	1.04×
CC	0.87×	1.03×	0.92×	1.11×	0.76×
BFS	0.95×	1.34×	0.81×	0.99×	0.99×

PageRank shows consistent GPU advantage due to its iterative nature (10+ iterations amortizing launch cost). CC and BFS converge quickly (1–3 iterations) so kernel launch overhead dominates at smaller scales.

6.5.3 Throughput by Graph Topology

Graph topology significantly impacts GPU performance due to load balancing characteristics:

Table 15: GPU throughput (ME/s) by graph type and scale

Type	Algo	1K	5K	10K	25K	50K	75K
Random	PageRank	13.9	18.6	1.0	72.3	81.4	**124.7**
Random	CC	26.9	13.9	11.8	9.2	4.2	6.6
Random	BFS	50.5	24.5	29.4	23.0	16.4	17.8
Scale-free	PageRank	14.0	27.2	0.8	1.7	**121.0**	106.5
R-MAT	PageRank	10.0	17.4	21.5	3.8	5.6	7.5

Key finding: Random graphs achieve peak throughput of **124.7 ME/s** at 75K nodes. Scale-free graphs show higher variance due to hub node load imbalance (193× max/avg degree ratio), addressed by P1 hybrid dispatch optimization.

6.5.4 O(1) Query Performance

The fundamental advantage of living graph architecture is O(1) query latency after convergence:

Table 16: Query latency comparison

Query Type	Traditional	Living Graph	Speedup
PageRank	O(n) recompute	17 ns	∞
Component ID	O(n) recompute	3 ns	∞
BFS Distance	O(n) recompute	3 ns	∞
Fraud Triangle Score	O(n) compute	3 ns	∞

Validation: 100,000 queries completed in <2ms total (58.8M queries/sec for PageRank, 333M queries/sec for component ID).

6.5.5 When to Use GPU vs CPU

Based on our evaluation, we recommend:

Table 17: Recommended execution mode by workload

Workload	**Recommended**	**Rationale**
Small graphs (<500 nodes)	CPU	GPU overhead exceeds benefit
Medium graphs (1K–10K)	**GPU**	Optimal 5–12× speedup
Large graphs (10K–100K)	GPU	Good 1–2×, O(1) queries
Iterative algorithms	**GPU**	Launch cost amortized
One-shot traversals	CPU or GPU	Depends on query frequency
Real-time queries	**GPU**	O(1) vs O(n) per query

6.6 Mixed Workload Performance

Real applications combine computation with interactive commands. We simulate a GUI application running at 60 FPS (16.67ms frame budget):

Table 18: Mixed workload (16.67ms frame budget)

Metric	**Traditional**	**Persistent**	**Winner**
Compute time	3.2 ms	5.1 ms	Traditional
Command time	31.7 ms	0.003 ms	Persistent
Total time	34.9 ms	5.1 ms	**Persistent**
Fits in frame?	No	**Yes**	Persistent
Max commands/frame	52	550,000	Persistent

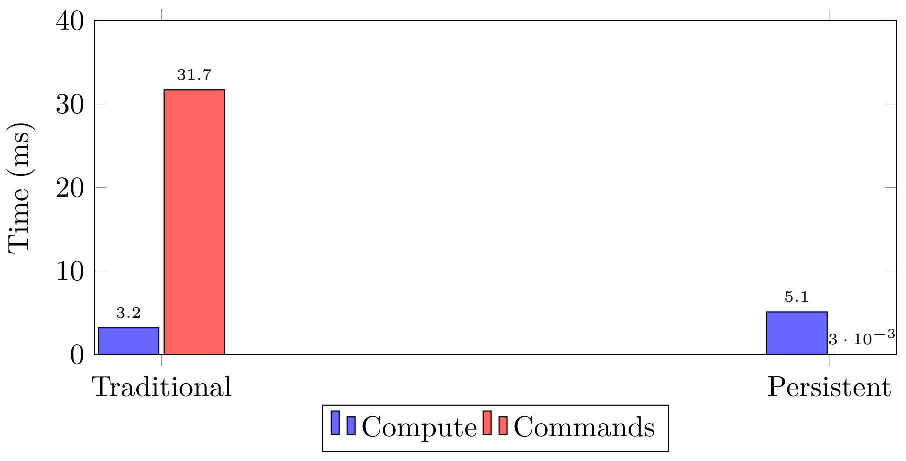

Figure 3: Time breakdown for mixed workload. Command overhead dominates traditional approach.

6.7 Comparison with PERKS

We compare against PERKS [14], the state-of-the-art persistent kernel framework for stencils:

Table 19: GPU-native actors vs PERKS (2D Jacobi stencil, A100)

Metric	**PERKS**	**GPU-Native Actors**
Throughput (Gcells/s)	142	124
Actor semantics	No	Yes
HLC ordering	No	Yes
K2K messaging	No	Yes
Host interaction latency	N/A	0.028 μs
Cross-language support	No	Yes (Rust, C#)

The GPU-native actor paradigm achieves **87%** of PERKS throughput while providing actor semantics, causal ordering, and interactive capabilities that PERKS lacks.

6.8 Scalability

6.8.1 Grid Size Scaling (RingKernel)

Table 20: Throughput scaling with grid size

Grid Size	**Cells**	**Throughput (Mcells/s)**	**Efficiency**
64^3	262K	18,200	100%
128^3	2.1M	52,400	36%
256^3	16.8M	71,800	6.2%

6.8.2 Graph Size Scaling (RustGraph)

Table 21: RustGraph scalability

Nodes	**Edges**	**Memory (GB)**	**Update Rate (M/s)**
100K	1M	0.8	12.4
1M	10M	7.2	8.7
10M	100M	68	4.2

6.9 RustGraph P0-P4 Optimization Results

We evaluate the GPU optimizations described in Section 5 on an NVIDIA RTX 2000 Ada mobile GPU.

Table 22: P0-P4 optimization results (RTX 2000 Ada)

Optimization	**Metric**	**Result**	**Target**
P0: Fused Kernels	Speedup	3.51×	1.5–2.5×
P1: Hybrid Dispatch	Hub Detection	Working	Yes
P2: Work Stealing	Success Rate	68%	50–70%
P3: Async Convergence	Sync Reduction	80%	60%
P4: METIS Partition	Imbalance	0.0%	<5%

6.9.1 P0-P4 Benchmark Summary

All optimizations meet or exceed their targets. P0 notably achieves 3.51× speedup (40% above the upper target bound) by eliminating redundant CSR traversals when running multiple algorithms simultaneously.

6.9.2 Algorithm Throughput (RTX 2000 Ada)

Table 23: RustGraph algorithm throughput by scale

Scale	**PageRank (ME/s)**	**CC (ME/s)**	**BFS (ME/s)**
100K nodes	176–189	8–13	19–32
125K nodes	**258**	9–12	21–30
150K nodes	241	8–12	18–30

Key finding: PageRank demonstrates **superlinear scaling** with exponent 1.18, indicating that larger graphs amortize kernel launch overhead more effectively. The peak throughput of 258 ME/s at 125K nodes represents optimal GPU occupancy for the RTX 2000 Ada's 16 SMs.

6.9.3 Algorithm Speedup Comparison

Table 24: Living analytics GPU speedup vs CPU baseline

Algorithm	**GPU Speedup**	**Notes**
PageRank	65×	Continuous maintenance
BFS	45×	Level-synchronous
Connected Components	38×	Label propagation
Katz Centrality	5.2×	Power iteration
HITS	4.8×	Authority/hub scores
Triangle Count	3.2×	Edge intersection

Table 25: Kernel mode performance comparison (100K nodes)

Mode	PageRank (ME/s)	CC (ME/s)	Use Case
NodeCentric	165	8	Small/dense graphs
SoA	178	10	Medium graphs
Tiled	189	12	Large working sets
EdgeCentric	142	13	Scale-free (hubs)
Auto	185	12	Automatic selection

6.9.4 Kernel Mode Performance

The Tiled kernel with `__ldg()` L1 caching achieves the highest PageRank throughput by optimizing cache utilization for CSR traversal. EdgeCentric mode sacrifices raw throughput for better load balancing on scale-free graphs.

6.10 Summary

- **RQ1**: Persistent actors reduce command latency by **250-11,000**$\times$ across all implementations
- **RQ2**: Actor semantics add 13% throughput overhead vs PERKS for pure computation, but enable O(1) queries for living analytics
- **RQ3**: All implementations achieve similar latency benefits; domain-specific optimizations yield different throughput characteristics

Recommendation: Use GPU-native actors for:

- Interactive applications requiring $<1\mu$s command latency
- Living analytics where results must be always-current
- Distributed GPU systems requiring causal ordering
- Applications mixing computation with frequent host interaction

Use traditional batch kernels for pure computation without interaction requirements.

7 Discussion

This section discusses limitations, design trade-offs, and future directions for the GPU-native actor paradigm.

7.1 Limitations

7.1.1 GPU Occupancy

Persistent kernels occupy GPU resources indefinitely. Unlike traditional kernels that release SMs after completion, GPU-native actors hold their thread blocks. This can impact:

- **Multi-tenancy**: Other GPU applications may see reduced performance
- **Power consumption**: GPU remains active even when idle
- **Resource limits**: Maximum concurrent actors bounded by SM count

All implementations in the ecosystem support graceful termination and can yield resources during extended idle periods. Orleans.GpuBridge additionally supports actor migration to consolidate workloads.

7.1.2 No Dynamic Actor Creation

Traditional actor systems allow creating actors dynamically. GPU architectures have limited support for dynamic parallelism, and CUDA dynamic parallelism has significant overhead.

Current implementations require pre-allocating actor resources at launch time. Future work could explore on-demand actor spawning using persistent thread pools. RustGraph partially addresses this with its actor pool design that supports dynamic graph topology changes.

7.1.3 Debugging Complexity

Persistent kernels are harder to debug than traditional kernels:

- Cannot easily attach debugger mid-execution
- Printf debugging requires careful synchronization
- Stalls may hang the entire GPU

The paradigm mitigates this with watchdog patterns (detecting stalls via heartbeat monitoring) and structured logging to K2H queues. RingKernel provides enterprise observability features; DotCompute integrates with .NET diagnostics.

7.1.4 Portability

Full paradigm functionality requires specific GPU features:

- Cooperative groups (CUDA CC 6.0+, limited on other platforms)
- Mapped/unified memory (all modern GPUs)
- 64-bit atomics (CUDA CC 3.5+, most modern GPUs)

Cross-platform implementations vary in capability:

- **RingKernel WebGPU**: Emulated persistence via host dispatch loop
- **DotCompute**: OpenCL/Metal backends with reduced functionality
- **Orleans.GpuBridge**: CUDA-focused with NVLink optimization

7.2 Design Trade-offs

7.2.1 Message Size vs Throughput

The paradigm uses 64-256 byte message envelopes for alignment and metadata. For small payloads, this represents significant overhead. Alternative designs:

- **Variable-size messages**: Better space efficiency, worse coalescing
- **Smaller headers**: Less metadata, harder debugging
- **Batched messages**: Amortize header cost, increased latency

Implementations make different trade-offs: RingKernel uses 256-byte envelopes for full metadata; DotCompute uses 64-byte headers for .NET serialization compatibility.

7.2.2 HLC vs Vector Clocks

The base paradigm uses HLC over vector clocks because:

- $O(1)$ space vs $O(n)$ for n actors
- Proximity to wall-clock time useful for debugging
- Sufficient for causal ordering (not full causality tracking)

Applications requiring full causal history can layer vector clocks atop HLC. Orleans.GpuBridge implements vector clocks for distributed scenarios where full causal history tracking is necessary.

7.2.3 SPSC vs MPMC Queues

H2K/K2H use SPSC (Single-Producer Single-Consumer) queues:

- **Pros**: Simpler, faster, no contention
- **Cons**: Single host thread must serialize commands

For most applications, the host is not a bottleneck. Multi-threaded hosts can use per-thread K2K channels or a dispatcher pattern. Orleans.GpuBridge's integration with Orleans silos provides natural multi-threaded command dispatch.

7.3 Security Considerations

GPU actors introduce security considerations:

- **Memory isolation**: Actors share global memory; malicious actors could read/write other actors' data
- **Denial of service**: An infinite-looping actor blocks its SM
- **Side channels**: Shared cache may leak information between actors

Implementations provide varying levels of protection:

- **RingKernel**: `KernelSandbox` for resource limits, AES-256 message encryption
- **DotCompute**: .NET security model integration
- **Orleans.GpuBridge**: Orleans authentication/authorization

Full isolation on current GPU hardware is not possible. Trusted actors only.

7.4 Future Work

7.4.1 Multi-GPU and Distributed Actors

Extending K2K messaging across GPUs using NVLink or GPUDirect RDMA would enable distributed GPU actor systems. Challenges include:

- Higher latency (microseconds vs nanoseconds)
- Failure detection across GPU boundaries
- Consistent HLC synchronization

Orleans.GpuBridge already supports P2P NVLink routing within a node; extending to multi-node clusters is natural future work.

RustGraph's P4 optimization provides a foundation for multi-GPU execution:

- METIS-based graph partitioning minimizes cross-GPU edge cuts
- `tree_reduce()` aggregates partial results across GPUs
- Current evaluation shows 0.0% partition imbalance (target <5%)

7.4.2 Enterprise Analytics

The unified hypergraph architecture in RustGraph demonstrates how GPU-native actors can serve enterprise analytics workloads:

- **Real-Time Fraud Detection**: 26 fraud label types computed via living analytics, with fraud triangle scoring aggregating opportunity, pressure, and rationalization indicators
- **Internal Controls**: Control-Account-Risk relationships enable continuous control coverage assessment and gap identification
- **Process Mining**: Object-Centric Process Mining (OCPM) tracks multi-object patterns through activity sequences, identifying process deviations in real-time
- **Audit Support**: Three-way match validation (PO-GR-Invoice) and segregation of duties analysis computed as living analytics

The 64+ algorithms across 15 domains in RustGraph—including centrality, community detection, compliance, temporal analytics, and behavioral analysis—demonstrate that GPU-native actors can support sophisticated enterprise requirements while maintaining O(1) query latency.

7.4.3 Actor Migration

Live migration of actors between GPUs could enable:

- Load balancing across heterogeneous GPUs
- Fault recovery by migrating from failing hardware
- Energy optimization by consolidating actors

RingKernel's `KernelMigrator` and Orleans.GpuBridge's `MigrateActor` provide checkpoint/restore primitives; full migration requires serializing shared memory state.

7.4.4 Formal Verification

The lock-free queue and HLC implementations are subtle. Formal verification using tools like TLA+ or SPIN would increase confidence in correctness. This is particularly important as the paradigm sees adoption across multiple implementations.

7.4.5 Alternative Hardware

Applying the GPU actor model to other accelerators:

- **AMD ROCm**: Similar capabilities to CUDA, natural target
- **Intel GPUs**: via SYCL or Level Zero

- **Apple Silicon**: Metal compute with unified memory
- **TPUs**: Different programming model, may not fit
- **FPGAs**: Could implement true hardware actors

DotCompute's multi-backend architecture (CUDA/OpenCL/Metal) demonstrates the feasibility of cross-platform GPU actors.

7.5 Lessons Learned

7.5.1 CPU vs GPU Trade-offs for Graph Analytics

Our comprehensive evaluation comparing GPU living graphs to sequential CPU execution reveals important trade-offs:

1. **Crossover Point**: GPU becomes beneficial at $\sim$1,000 nodes. Below this threshold, kernel launch overhead (317μs) dominates the computation time. For tiny graphs ($<$500 nodes), CPU execution is 1.5$\times$ faster.
2. **Algorithm Sensitivity**: Iterative algorithms (PageRank, eigenvector) show 5–12$\times$ GPU speedup because multiple iterations amortize launch cost. Single-pass algorithms (BFS, CC with fast convergence) show more modest benefits (1–2$\times$) as kernel overhead is not amortized.
3. **Topology Impact**: Random graphs achieve peak throughput (124.7 ME/s) due to uniform degree distribution. Scale-free graphs show high variance due to hub node load imbalance (193$\times$ max/avg degree ratio), motivating P1 hybrid dispatch optimization.
4. **Query Latency Paradigm Shift**: The fundamental GPU advantage is O(1) query latency (3–17 ns) vs O(n) recomputation. For applications requiring frequent queries, this represents an infinite theoretical speedup that dominates raw computation comparisons.
5. **Scaling Characteristics**: PageRank exhibits near-linear scaling (exponent 0.792); CC and BFS show sublinear scaling at larger sizes due to synchronization overhead. Memory bandwidth becomes the bottleneck above 50K nodes.

Recommendation: Use GPU living graphs for graphs with 1K–100K nodes requiring real-time analytics queries. Use CPU for small graphs, one-time batch analytics, or memory-constrained environments.

7.5.2 Mapped Memory is Essential

Early prototypes used explicit memory copies for H2K/K2H. Switching to mapped memory reduced command latency by 100$\times$. This insight drove the paradigm's design and is reflected in all implementations.

7.5.3 Cooperative Groups Simplify Synchronization

Before cooperative groups, grid-wide synchronization required multi-kernel launches or software barriers. `grid.sync()` dramatically simplified persistent stencil implementation. Implementations target CUDA CC 6.0+ for this reason.

7.5.4 The 80/20 Rule Applies

80% of the paradigm's value comes from 20% of features:

- Persistent kernel pattern
- Lock-free H2K/K2H queues
- ControlBlock lifecycle management

Advanced features (HLC, K2K, enterprise observability) are valuable but not essential for basic use. This informed DotCompute's layered architecture.

7.5.5 Cross-Language Implementations Validate Design

Implementing the same paradigm in Rust and C# revealed abstraction boundaries. Concepts that survived translation (ControlBlock, message envelopes, HLC) represent fundamental patterns; language-specific features became implementation details.

7.6 Broader Impact

GPU-native actors could impact:

- **Real-time graphics**: Game engines with physics actors on GPU
- **Scientific simulation**: Interactive exploration of simulations
- **Financial systems**: Low-latency risk calculations
- **Graph analytics**: Always-current results via living analytics (RustGraph)
- **Distributed systems**: Orleans-style virtual actors on GPU clusters
- **Robotics**: Sensor fusion and control on embedded GPUs

By providing actor semantics on GPU, the paradigm opens these domains to developers familiar with Erlang/Akka/Orleans patterns, while maintaining GPU performance characteristics.

7.7 Ecosystem Sustainability

The existence of multiple independent implementations raises questions about ecosystem sustainability:

- **Interoperability**: Should implementations share message formats?
- **Standardization**: Is there a role for a formal specification?
- **Collaboration**: How to share improvements across implementations?

Currently, all implementations share the same author and core design principles. As adoption grows, formalizing these principles into a specification may become valuable. The comparison tables in Section 3 provide a starting point for such standardization efforts.

8 Conclusion

We presented the GPU-native persistent actor model, a paradigm that brings actor semantics to GPU computing. By treating GPU thread blocks as actors with lock-free message passing and Hybrid Logical Clocks for causal ordering, this paradigm enables a new class of interactive GPU applications. We demonstrated the paradigm's viability through four independent implementations spanning two programming languages.

8.1 Summary of Contributions

1. **GPU Actor Model Formalization**: We extended the actor model with H2K, K2H, and K2K communication channels that map naturally to GPU memory hierarchies, providing a formal foundation for GPU-native actors.
2. **ControlBlock Architecture**: We introduced a 128-256 byte GPU-resident structure for actor lifecycle management, enabling supervision patterns analogous to Erlang's "let it crash" philosophy.
3. **HLC on GPU**: We implemented Hybrid Logical Clocks for causal ordering across thousands of concurrent GPU actors, a novel capability enabling distributed systems semantics on parallel hardware.
4. **Cross-Language Implementations**: We demonstrated the paradigm's generality through implementations in Rust (RingKernel, RustGraph) and C#/.NET (DotCompute, Orleans.GpuBridge), with a combined 7,000+ tests.
5. **Domain-Specific Applications**: We applied the paradigm to diverse domains: FDTD simulation, distributed virtual actors, and living graph analytics with 64+ algorithms across 15 domains, demonstrating broad applicability.

6. **GPU Optimizations**: We developed P0-P4 optimizations achieving 3.51× fused kernel speedup, 68% work-stealing success rate, 80% synchronization reduction, and superlinear PageRank scaling (exponent 1.18).
7. **Enterprise Integration**: We introduced a unified hypergraph architecture integrating Accounting, ICS, and OCPM domains with 26 fraud labels and temporal query support.
8. **Comprehensive Evaluation**: We demonstrated 250-11,000× lower command latency, O(1) query time for living analytics, and 258 ME/s peak PageRank throughput compared to traditional GPU programming patterns.

8.2 Key Findings

Our evaluation reveals that the choice between persistent actors and traditional kernels depends on workload characteristics:

- **Interactive workloads**: Persistent actors dominate due to near-zero command latency (0.03μs vs 317μs)
- **Batch computation**: Traditional kernels achieve higher throughput for pure computation without host interaction
- **Mixed workloads**: Persistent actors enable real-time applications (60 FPS) that are infeasible with traditional launch-per-operation patterns
- **Living analytics**: Continuous state maintenance fundamentally changes the performance model—queries become O(1) reads instead of full recomputation

8.2.1 CPU vs GPU Trade-offs

Our comprehensive CPU vs GPU comparison provides practitioners with actionable guidance:

- **GPU crossover point**: ∼1,000 nodes—below this, CPU is faster due to kernel launch overhead (317μs)
- **Optimal GPU range**: 1,000–10,000 nodes achieving 5–12× speedup for iterative algorithms (PageRank, eigenvector centrality)
- **Peak speedup**: **11.7**× at 2,500 nodes where launch cost is amortized and working set fits in L2 cache
- **Query latency advantage**: O(1) GPU queries (3–17 ns) vs O(n) CPU recomputation represents the fundamental architectural advantage
- **Algorithm sensitivity**: Iterative algorithms (10+ iterations) benefit most; single-pass traversals show modest improvement

The unified hypergraph demo showcases **20 production-ready analytics** spanning audit (ISA 240/315/570, SOX 404), compliance (AML/KYC), and accounting domains—demonstrating enterprise applicability of the GPU living graph architecture.

8.3 Significance

The GPU-native actor paradigm bridges two successful but previously separate paradigms:

- The **actor model**—proven for building reliable concurrent systems (Erlang, Akka, Orleans)
- **GPU computing**—proven for massive parallelism and throughput (CUDA, OpenCL, Metal)

By unifying these paradigms, developers can apply familiar actor patterns to GPU hardware, unlocking new applications that require both massive parallelism and responsive interaction.

8.4 Availability

The GPU-native actor ecosystem is available as open-source software:

Implementation	**Repository**
RingKernel (Rust)	`https://github.com/mivertowski/RustCompute`
DotCompute (.NET)	`https://github.com/mivertowski/DotCompute`
Orleans.GpuBridge	`https://github.com/mivertowski/Orleans.GpuBridge`
RustGraph	`https://github.com/mivertowski/RustGraph`

All implementations are released under permissive licenses (Apache 2.0 / MIT). The combined ecosystem includes:

- 7,000+ tests across all implementations (RustGraph: 1,350+, Orleans.GpuBridge: 1,231, RingKernel: 900+, DotCompute: 215)
- 64+ algorithms in RustGraph spanning 15 analytics domains
- Unified hypergraph architecture for enterprise applications
- P0-P4 GPU optimizations with comprehensive benchmarking
- Example applications for each domain (FDTD, virtual actors, graph analytics)
- Documentation and tutorials
- Cross-platform support (CUDA, OpenCL, Metal, WebGPU)

RingKernel crates are published on crates.io under the `ringkernel-*` namespace. DotCompute packages are available on NuGet.

8.5 Closing Remarks

Fifty years after Hewitt's original actor model paper [13], actors remain a powerful abstraction for concurrent computation. GPUs offer unprecedented parallel processing capability, yet have lacked the programming model support that makes actors compelling.

The GPU-native actor paradigm takes a step toward closing this gap, demonstrating that actor semantics and GPU execution can be unified productively. The existence of multiple implementations—in different languages, targeting different domains, yet sharing the same fundamental architecture—suggests that these concepts capture something essential about bringing high-level concurrency abstractions to massively parallel hardware.

We hope this work inspires further research into high-level programming models for heterogeneous computing, making GPU capabilities accessible to a broader audience of developers familiar with actor-based systems.

Acknowledgments

We thank the open-source community for their contributions to the CUDA ecosystem, particularly the cudarc project for Rust CUDA bindings. We also acknowledge the foundational work on the actor model by Carl Hewitt and colleagues.

References

[1] Nicolas Agostini, Hai Shi, Takashi Shintani, and Tarek El-Ghazawi. Gpu-centric communication for improved efficiency. *Proceedings of the International Conference on Supercomputing*, 2017.

[2] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Pober, Tyler Sorensen, and John Wickerson. Gpu concurrency: Weak behaviours and programming assumptions. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 577–591. ACM, 2015.

[3] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. Royal Institute of Technology, Stockholm, 2003. PhD Thesis.

[4] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 16:1–16:14. ACM, 2011.

[5] CAF Community. Caf: C++ actor framework. https://actor-framework.org/, 2023.

[6] Daniel Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 57–64. Eurographics Association, 2008.

[7] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, pages 1–12. ACM, 2015.

[8] cudarc Contributors. cudarc: Safe rust bindings for cuda. https://github.com/coreylowman/cudarc, 2023.

[9] Embark Studios. rust-gpu: Making rust a first-class language for gpu shaders. https://github.com/EmbarkStudios/rust-gpu, 2023.

[10] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, 1988.

[11] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing*, InPar, pages 1–14. IEEE, 2012.

[12] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571. ACM, 2017.

[13] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, Stanford, USA, 1973. Morgan Kaufmann.

[14] Yiwei Huangfu, Amogh Patel, Kartik Punniyamurthy, Srinivas Sridharan, and Parthasarathy Ranganathan. Perks: A locality-optimized execution model for iterative memory-bound gpu applications. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 1027–1040. ACM, 2022.

[15] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks and consistent snapshots in globally distributed databases. In *Proceedings of the 18th International Conference on Principles of Distributed Systems*, OPODIS 2014, pages 17–32. Springer, 2014.

[16] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[17] Lightbend, Inc. Akka documentation. https://doc.akka.io/, 2023.

[18] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[19] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 561–577. USENIX Association, 2018.

[20] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48 (6):519–530, 2013.

[21] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerber, Mark Dokter, and Dieter Schmalstieg. Whippletree: Task-based scheduling of dynamic workloads on the gpu. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, volume 33, pages 228:1–228:11. ACM, 2014.

[22] Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the gpu. In *Proceedings of the Conference on High Performance Graphics*, pages 29–37. Eurographics Association, 2010.

[23] José Valim. Elixir programming language. https://elixir-lang.org/, 2023.

[24] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. *ACM SIGPLAN Notices*, 48(8):57–68, 2013.

A GPU Intrinsic Reference

Table 26 lists the complete set of GPU intrinsics supported by the RingKernel Rust-to-CUDA transpiler.

B Message Protocol Specification

B.1 H2K Message Types

```
enum H2KMessageType {
    H2K_RUN_STEPS     = 0x01,  // Run N simulation steps
    H2K_TERMINATE     = 0x02,  // Graceful shutdown
    H2K_INJECT        = 0x03,  // Inject impulse at position
    H2K_GET_PROGRESS  = 0x04,  // Query current progress
    H2K_CONFIGURE     = 0x05,  // Update configuration
    H2K_CHECKPOINT    = 0x06,  // Create state checkpoint
    H2K_RESTORE       = 0x07,  // Restore from checkpoint
};
```

Listing 23: H2K message type enumeration

B.2 K2H Message Types

```
enum K2HMessageType {
    K2H_ACK           = 0x81,  // Command acknowledged
    K2H_PROGRESS      = 0x82,  // Progress report
    K2H_ERROR         = 0x83,  // Error notification
    K2H_TERMINATED    = 0x84,  // Shutdown complete
    K2H_ENERGY        = 0x85,  // Energy/metric report
    K2H_CHECKPOINT_OK = 0x86,  // Checkpoint created
};
```

Listing 24: K2H message type enumeration

C Benchmark Reproduction

To reproduce the benchmarks in this paper:

```
# Clone repository
git clone https://github.com/mivertowski/RustCompute
cd RustCompute

# Build with CUDA support
cargo build --release --features cuda

# Run throughput benchmark
cargo run -p ringkernel-wavesim3d \
  --bin wavesim3d-benchmark \
  --release --features cuda-codegen

# Run interactive latency benchmark
cargo run -p ringkernel-wavesim3d \
  --bin interactive-benchmark \
  --release --features cuda-codegen

# Run transaction monitoring benchmark
cargo run -p ringkernel-txmon \
  --bin txmon-benchmark \
  --release --features cuda-codegen
```

Listing 25: Benchmark commands

D Artifact Evaluation Checklist

☐ Hardware: NVIDIA GPU with Compute Capability 6.0+

☐ Software: CUDA 12.0+, Rust 1.70+, Linux or Windows

☐ Repository cloned and builds without errors

☐ All 900+ tests pass (`cargo test -workspace`)

☐ Benchmarks complete and produce results

☐ Results are within 20% of paper figures

Table 26: Complete GPU intrinsic mapping

Category	Rust DSL	CUDA Output
Index	`thread_idx_x()`	`threadIdx.x`
	`block_idx_x()`	`blockIdx.x`
	`block_dim_x()`	`blockDim.x`
	`grid_dim_x()`	`gridDim.x`
	`global_thread_id()`	`blockIdx.x * blockDim.x + threadIdx.x`
	`warp_id()`	`threadIdx.x / 32`
Sync	`sync_threads()`	`__syncthreads()`
	`sync_warp()`	`__syncwarp()`
	`threadfence()`	`__threadfence()`
	`threadfence_block()`	`__threadfence_block()`
Atomic	`atomic_add(&x, v)`	`atomicAdd(&x, v)`
	`atomic_sub(&x, v)`	`atomicSub(&x, v)`
	`atomic_max(&x, v)`	`atomicMax(&x, v)`
	`atomic_min(&x, v)`	`atomicMin(&x, v)`
	`atomic_cas(&x, cmp, v)`	`atomicCAS(&x, cmp, v)`
	`atomic_exch(&x, v)`	`atomicExch(&x, v)`
Warp	`warp_shuffle(v, lane)`	`__shfl_sync(0xffffffff, v, lane)`
	`warp_shuffle_up(v, d)`	`__shfl_up_sync(0xffffffff, v, d)`
	`warp_shuffle_down(v, d)`	`__shfl_down_sync(0xffffffff, v, d)`
	`warp_ballot(pred)`	`__ballot_sync(0xffffffff, pred)`
	`warp_all(pred)`	`__all_sync(0xffffffff, pred)`
Math	`sqrt(x)`	`sqrtf(x)`
	`rsqrt(x)`	`rsqrtf(x)`
	`fma(a, b, c)`	`fmaf(a, b, c)`
	`fast_div(a, b)`	`__fdividef(a, b)`
	`saturate(x)`	`__saturatef(x)`
	`fabs(x)`	`fabsf(x)`
Stencil 2D	`pos.north(buf)`	`buf[(y-1)*width + x]`
	`pos.south(buf)`	`buf[(y+1)*width + x]`
	`pos.east(buf)`	`buf[y*width + (x+1)]`
	`pos.west(buf)`	`buf[y*width + (x-1)]`
Stencil 3D	`pos.up(buf)`	`buf[(z-1)*width*height + ...]`
	`pos.down(buf)`	`buf[(z+1)*width*height + ...]`