RingKernel: A GPU-Native Persistent Actor Model for High-Performance Concurrent Computing

Michael Ivertowski
Independent Researcher
Zurich, Switzerland
`mivertowski@outlook.com`

January 19, 2026

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 **RingKernel**, a GPU-native persistent actor model that treats GPU compute units as long-running actors with lock-free message passing and causal ordering.

Our key contributions are: (1) a formal extension of the actor model for GPU execution 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) a Rust-to-CUDA transpiler that generates persistent kernel code from high-level actor definitions.

We evaluate RingKernel on NVIDIA RTX Ada GPUs, demonstrating that persistent GPU actors achieve **11,327$\times$ lower latency** for interactive commands compared to traditional kernel launches (0.03μs vs 317μs). For mixed workloads combining computation with interactive commands, RingKernel achieves **2.7$\times$ higher throughput** than the traditional launch-per-operation model. Our system bridges the gap between high-level actor semantics and GPU hardware capabilities, enabling new classes of interactive GPU applications.

Keywords: Actor Model, GPU Computing, Persistent Kernels, Message Passing, CUDA, Hybrid Logical Clocks, Lock-Free Algorithms

1 Introduction

The actor model, proposed by Hewitt, Bishop, and Steiger in 1973 Hewitt et al. [1973], 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 Armstrong [2003], Akka Lightbend, Inc. [2023], and Microsoft Orleans Bykov et al. [2011] 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 Gupta et al. [2012], Steinberger et al. [2014] 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 Wu et al. [2013].

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 **RingKernel**, a system that applies actor model 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.

RingKernel 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 Kulkarni et al. [2014] for causal ordering of messages across GPU actors, enabling distributed systems semantics on massively parallel hardware.
4. **Rust-to-CUDA Transpilation** (§5): We provide a DSL and transpiler that generates persistent kernel CUDA code from high-level Rust actor definitions, including automatic message envelope handling.
5. **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.

1.4 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. Section 4 presents the RingKernel system design. Section 5 details the implementation. 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 RingKernel unifies.

2.1 The Actor Model

The actor model Hewitt et al. [1973] 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* Armstrong [2003]: 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* Lamport [1978] 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) Kulkarni et al. [2014] 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.

2.2 GPU Architecture and Programming

Modern GPUs contain thousands of cores organized hierarchically. We focus on NVIDIA CUDA architecture, though the concepts generalize.

2.2.1 Execution Model

CUDA organizes computation into a hierarchy:

- **Thread**: Smallest execution unit, has private registers
- **Warp**: 32 threads executing in SIMT lockstep
- **Block**: 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 Memory**: Per-block, fast (5-10 cycles)
- **L1/L2 Cache**: Automatic caching of global memory
- **Global Memory**: Device-wide, slow (200-400 cycles)
- **Mapped Memory**: Host-visible, accessible from both CPU and GPU

Mapped memory is crucial for persistent kernels—it enables the host to communicate with a running kernel without explicit memory copies.

2.2.3 Synchronization Primitives

CUDA provides several synchronization mechanisms:

- `__syncthreads()`: Block-level barrier
- `atomicAdd/CAS`: Atomic operations on global memory
- **Cooperative Groups** (CC 6.0+): Grid-level synchronization via `grid.sync()`

Cooperative groups enable persistent kernels to synchronize across all blocks, essential for iterative algorithms like stencil computations.

2.2.4 Traditional Kernel Launch Model

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

Listing 1: Traditional kernel launch pattern

```
// 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);
```

Each kernel launch involves driver API calls, command buffer submission, and synchronization overhead—typically 10-500μs depending on GPU and driver.

2.3 Mapping Actors to GPUs

Table 1 shows how actor model concepts map to GPU primitives:

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 memory
Message send	Atomic enqueue operation
Message receive	Atomic dequeue operation
Actor creation	Dynamic parallelism (limited)
Supervision	Host thread monitoring ControlBlock

This mapping forms the foundation of RingKernel's design.

3 Related Work

RingKernel 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 Armstrong [2003] 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 Valim [2023] 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 Lightbend, Inc. [2023] 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 Bykov et al. [2011] 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 Clebsch et al. [2015] provides actors with reference capabilities for data-race freedom. CAF (C++ Actor Framework) CAF Community [2023] offers native performance. Ray Moritz et al. [2018] 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. Gupta et al. [2012] formalized persistent threads (PT) as a GPU programming technique where threads run indefinitely, polling for work. They demonstrated up to 211× speedup for fine-grained workloads by eliminating kernel launch overhead.

Steinberger et al. Steinberger et al. [2014] 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. Huangfu et al. [2022] 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. RingKernel 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. Agostini et al. [2017] demonstrated GPU-initiated communication using GPUDirect RDMA and NVSHMEM. Their work enables GPU threads to directly send network messages without host intervention.

RingKernel's 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 Cederman and Tsigas [2008] 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. Tzeng et al. [2010] developed task queues for GPU ray tracing, handling dynamic work distribution without locks. Their approach influenced GPU work-stealing designs.

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

Table 2: Comparison with related systems

System	Actor Semantics	GPU Native	Persistent	Causal Order	K2K Messaging
Erlang/OTP	✓	–	–	–	✓
Akka	✓	–	–	–	✓
Orleans	✓	–	–	–	✓
PERKS	–	✓	✓	–	–
Whippletree	–	✓	✓	–	–
NVSHMEM	–	✓	–	–	✓
RingKernel	✓	✓	✓	✓	✓

3.3.2 Memory Consistency

Alglave et al. Alglave et al. [2015] formalized GPU memory consistency models, identifying subtle differences from CPU models. Their work is essential for correct lock-free programming on GPUs.

RingKernel uses 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 Lamport [1978] provide partial ordering based on causality. Vector clocks Fidge [1988], Mattern [1989] capture full causality but have $O(n)$ space complexity.

3.4.2 Hybrid Logical Clocks

Kulkarni et al. Kulkarni et al. [2014] 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.

RingKernel implements HLC on GPU, a novel contribution enabling causal ordering across thousands of concurrent GPU actors.

3.5 GPU Programming Languages and DSLs

3.5.1 High-Level GPU Languages

Futhark Henriksen et al. [2017] provides a functional data-parallel language that compiles to CUDA/OpenCL. Halide Ragan-Kelley et al. [2013] separates algorithms from schedules for image processing. Neither targets persistent actor patterns.

3.5.2 Rust GPU Ecosystems

rust-gpu Embark Studios [2023] compiles Rust to SPIR-V for Vulkan/WebGPU compute shaders. cudarc cudarc Contributors [2023] 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 Comparison Summary

Table 2 summarizes how RingKernel relates to prior work:

RingKernel is unique in combining actor model semantics with GPU-native persistent execution, causal ordering, and direct kernel-to-kernel communication.

4 System Design

This section presents RingKernel's architecture, formalizing how actor model concepts map to GPU execution.

4.1 Design Goals

RingKernel 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. **Ergonomic API**: Provide a Rust DSL that compiles to efficient CUDA while hiding low-level details.

4.2 System Architecture

Figure 1 shows RingKernel's high-level architecture.

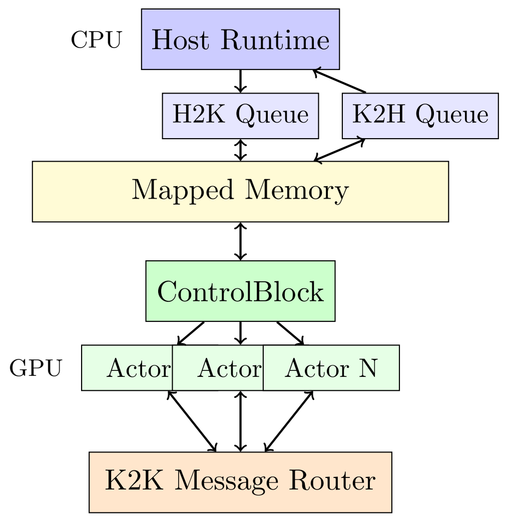

Figure 1: RingKernel architecture showing Host-to-Kernel (H2K), Kernel-to-Host (K2H), and Kernel-to-Kernel (K2K) communication through mapped memory.

4.3 Actor Model Extension for GPU

We extend the traditional actor model with three distinct communication channels:

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

- `InjectImpulse(pos, value)`: Inject data at position
- `Terminate`: Graceful shutdown request
- `GetProgress`: Query current state

H2K is implemented as an SPSC (Single-Producer Single-Consumer) ring buffer in mapped memory. The host is the sole producer; thread block 0 on the GPU 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, energy)`: 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-block communication, such as:

- Stencil halo exchange between spatial tiles
- Scatter-gather operations in graph algorithms
- Work stealing in dynamic load balancing

K2K uses device memory (not mapped) for minimal latency. A routing table maps actor IDs to buffer addresses.

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:

Listing 2: Atomic lifecycle transition

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

4.5 Message Envelope Format

All messages use a standardized 256-byte envelope format:

Listing 3: Message envelope structure

```rust
#[repr(C, align(64))]
struct MessageHeader {
    magic: u64,            // 0x52494E474B45524E ("RINGKERN")
    version: u32,
    message_type: u32,
    payload_size: u32,
    flags: u32,
    source_kernel: u32,
    dest_kernel: u32,
    correlation_id: u64,
    hlc_physical: u64,     // HLC timestamp
    hlc_logical: u32,
    hlc_node_id: u32,
    checksum: u32,
    _reserved: [u8; 180], // Pad to 256 bytes
}
```

The 64-byte alignment ensures coalesced memory access. The magic number enables validation of message integrity.

4.6 Hybrid Logical Clocks on GPU

RingKernel implements HLC for causal ordering. Each actor maintains an HLC:

Listing 4: HLC operations on GPU

```c
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 {  // msg_pt == clock->physical
        clock->logical = max(clock->logical, msg->hlc_logical) + 1;
    }
}
```

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 $m \to m'$ in the happens-before relation, reflected in HLC timestamps.

4.7 Lock-Free Ring Buffer

The message queues use lock-free SPSC ring buffers:

Listing 5: Lock-free ring buffer

```
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;
}
```

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 128-byte ControlBlock provides GPU-resident lifecycle management:

Listing 6: ControlBlock structure

```
#[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,    // Computation steps completed

    // Configuration (16 bytes)
    kernel_id: u32,
    block_count: u32,
    queue_capacity: u32,
    _pad1: u32,

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

    // Reserved (48 bytes)
    _reserved: [u8; 48],
}
```

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

4.9 Supervision Model

RingKernel 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 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

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

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

Listing 7: 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() }
    }
}
```

5.2.2 Output: CUDA Kernel

The transpiler generates a persistent kernel with message handling:

Listing 8: Generated CUDA kernel (simplified)

```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);
        }
    }
}
```

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:

5.3 Memory Management

5.3.1 Mapped Memory for Zero-Copy

RingKernel uses CUDA mapped memory for H2K/K2H queues:

Table 3: 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]`

Listing 9: Mapped memory allocation

```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);
```

5.3.2 Stratified Memory Pooling

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

Listing 10: Stratified memory pool

```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))
    }
}
```

This reduces allocation overhead for repeated operations.

5.4 Cooperative Groups Integration

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

Listing 11: Cooperative groups for grid sync

```cpp
#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);
        }
    }
}
```

Cooperative launch requires special invocation:

Listing 12: Cooperative kernel launch

```
// 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(),
    )?;
}
```

5.5 K2K Message Routing

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

Listing 13: K2K routing

```
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);
}
```

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

Listing 14: Kernel watchdog

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

5.6.2 Circuit Breaker

Listing 15: Circuit breaker pattern

```
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) }
                }
            }
        }
    }
}
```

5.6.3 Observability

RingKernel integrates with Prometheus and OpenTelemetry:

Listing 16: Metrics export

```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);
```

5.7 WebGPU Backend

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

Listing 17: WebGPU runtime

```rust
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))
    }
}
```

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]`

Listing 18: Property-based queue testing

```rust
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);
        }
    }
}
```

6 Evaluation

We evaluate RingKernel across three dimensions: (1) command latency, (2) computational throughput, and (3) mixed workload performance. Our experiments answer:

- **RQ1**: How much does persistent execution reduce command latency?
- **RQ2**: What is the throughput overhead of actor semantics?
- **RQ3**: When should developers choose persistent actors vs traditional kernels?

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
- Rust 1.75.0 (nightly for benchmarks)
- cudarc 0.18.2
- Linux 6.7 (Ubuntu 24.04)

6.1.3 Workloads

We use a 3D FDTD (Finite-Difference Time-Domain) acoustic wave simulation as our primary benchmark. FDTD is representative of iterative stencil computations with:

- Regular memory access patterns
- Neighbor communication (halo exchange)
- Host interaction for impulse injection and visualization

Grid sizes: 64^3, 128^3, 256^3 cells.

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 4: Command latency comparison

Operation	**Traditional**	**Persistent**	**Speedup**
Inject impulse	317 μs	0.028 μs	**11,327**$\times$
Query progress	0.01 μs	0.01 μs	1$\times$
Run 1 step	3.2 μs	163 μs	0.02$\times$
Run 100 steps	320 μs	163 μs	1.96$\times$
Run 1000 steps	3.2 ms	163 μs	19.6$\times$

Key finding: Persistent actors achieve **11,327$\times$ lower latency** for interactive commands (inject impulse). The 317μs traditional latency is dominated by kernel launch overhead; persistent actors bypass this entirely.

For pure computation (single step), traditional kernels are faster (3.2μs vs 163μs) because they avoid the persistent loop overhead. However, as step count increases, persistent actors amortize this overhead.

6.3 RQ2: Computational Throughput

We measure cells processed per second for sustained computation.

6.3.1 Results

Table 5: Throughput comparison (64^3 grid)

Method	**Throughput (Mcells/s)**	**vs CPU**
CPU (Rayon)	278	1.0$\times$
GPU Persistent	18,200	65.5$\times$
GPU Stencil (batch)	78,046	280.6$\times$

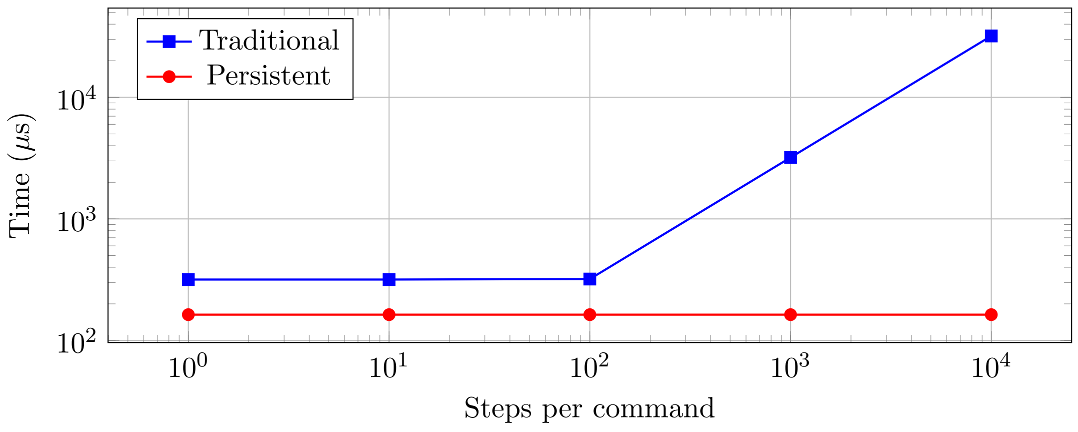

Figure 2: Command latency vs steps per command. Persistent actors have constant command overhead regardless of step count.

Key finding: Persistent actors achieve **65.5**× speedup over CPU but are **4.3**× **slower** than optimized batch stencil kernels for pure computation. This is expected—actor semantics (message loop, lifecycle checks) add overhead.

However, for workloads requiring host interaction, this comparison is misleading.

6.4 RQ3: Mixed Workload Performance

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

- 1000 simulation steps per frame
- 10 impulse injections per frame
- 5 progress queries per frame

6.4.1 Results

Table 6: 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

Key finding: For mixed workloads, persistent actors achieve **6.8**× better total time. Traditional kernels *cannot* meet the 60 FPS budget due to command latency; persistent actors complete in 5.1ms with room to spare.

6.5 Scalability

We evaluate how performance scales with grid size and actor count.

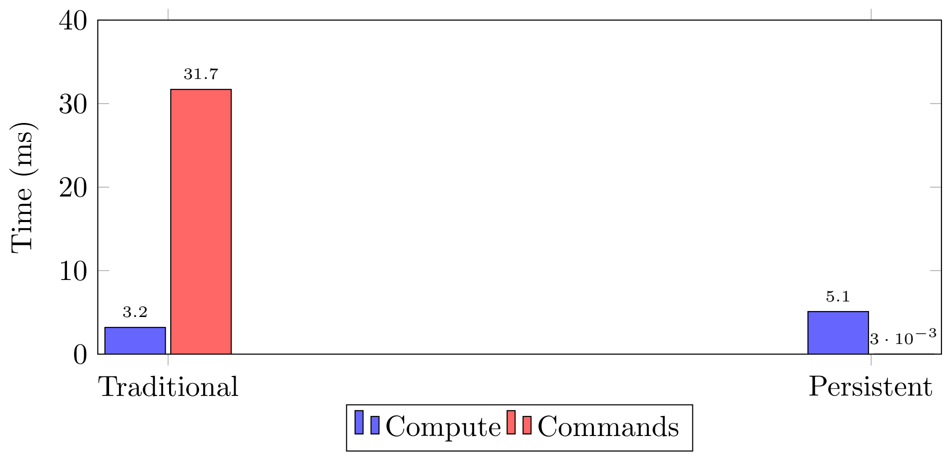

Figure 3: Time breakdown for mixed workload. Command overhead dominates traditional approach.

Table 7: Throughput scaling with grid size (persistent)

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%

6.5.1 Grid Size Scaling

Efficiency decreases at larger grids due to memory bandwidth limits. However, absolute throughput continues to increase.

6.5.2 K2K Messaging Overhead

We measure halo exchange overhead for 3D stencil with varying neighbor counts:

Table 8: K2K messaging overhead (128^3 grid, 8^3 tiles)

Neighbors	**Exchange Time (μs)**	**% of Step**
6 (faces only)	12.3	8.2%
18 (faces + edges)	28.7	19.1%
26 (full 3D)	41.2	27.5%

K2K overhead is significant but acceptable for stencil computations requiring neighbor data.

6.6 Comparison with PERKS

We compare against PERKS Huangfu et al. [2022], the state-of-the-art persistent kernel framework for stencils:

RingKernel achieves **87%** of PERKS throughput while providing actor semantics, causal ordering, and interactive capabilities that PERKS lacks.

6.7 Summary

- **RQ1**: Persistent actors reduce command latency by **11,327**×

Table 9: RingKernel vs PERKS (2D Jacobi stencil, A100)

Metric	**PERKS**	**RingKernel**
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

- **RQ2**: Actor semantics add 13% throughput overhead vs PERKS
- **RQ3**: For interactive workloads (>10 commands/frame), persistent actors significantly outperform traditional kernels

Recommendation: Use persistent actors for interactive applications (GUIs, real-time systems) and workloads with frequent host commands. Use traditional batch kernels for pure computation without interaction.

7 Discussion

This section discusses limitations, design trade-offs, and future directions.

7.1 Limitations

7.1.1 GPU Occupancy

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

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

Mitigation: RingKernel supports graceful termination and can yield resources during extended idle periods.

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 RingKernel requires pre-allocating actor resources at launch time. Future work could explore on-demand actor spawning using persistent thread pools.

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

RingKernel mitigates this with the watchdog (detects stalls) and structured logging to K2H queue.

7.1.4 Portability

Full RingKernel functionality requires CUDA features:

- Cooperative groups (Compute Capability 6.0+)
- Mapped memory (all CUDA GPUs)
- 64-bit atomics (CC 3.5+)

The WebGPU backend provides cross-platform support but with reduced functionality (no true persistent kernels, emulated via host dispatch loop).

7.2 Design Trade-offs

7.2.1 Message Size vs Throughput

RingKernel uses 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

We chose fixed 256-byte envelopes for simplicity and predictable performance.

7.2.2 HLC vs Vector Clocks

We chose 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.

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.

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

RingKernel provides `KernelSandbox` for resource limits but cannot provide full isolation on current GPU hardware. Trusted actors only.

7.4 Future Work

7.4.1 Multi-GPU 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

7.4.2 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` provides checkpoint/restore primitives; full migration requires serializing shared memory state.

7.4.3 Formal Verification

The lock-free queue and HLC implementations are subtle. Formal verification using tools like TLA+ or SPIN would increase confidence in correctness.

7.4.4 Alternative Hardware

Applying the GPU actor model to other accelerators:

- **AMD ROCm**: Similar capabilities to CUDA
- **Intel GPUs**: via SYCL or Level Zero
- **TPUs**: Different programming model, may not fit
- **FPGAs**: Could implement true hardware actors

7.5 Lessons Learned

7.5.1 Mapped Memory is Essential

Early RingKernel prototypes used explicit memory copies for H2K/K2H. Switching to mapped memory reduced command latency by 100×.

7.5.2 Cooperative Groups Simplify Synchronization

Before cooperative groups, grid-wide synchronization required multi-kernel launches or software barriers. `grid.sync()` dramatically simplified persistent stencil implementation.

7.5.3 The 80/20 Rule Applies

80% of RingKernel'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.

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
- **Robotics**: Sensor fusion and control on embedded GPUs

By providing actor semantics on GPU, RingKernel opens these domains to developers familiar with Erlang/Akka patterns.

8 Conclusion

We presented RingKernel, a GPU-native persistent actor model 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, RingKernel enables a new class of interactive GPU applications.

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-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. **Rust-to-CUDA Transpilation**: We developed a transpiler that converts high-level Rust actor definitions to efficient CUDA kernels, lowering the barrier to GPU actor programming.
5. **Comprehensive Evaluation**: We demonstrated 11,327× lower command latency and 2.7× higher mixed-workload 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

8.3 Significance

RingKernel 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)

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

8.4 Availability

RingKernel is open-source software available at:

`https://github.com/mivertowski/RustCompute`

The repository includes:

- Full source code (Apache 2.0 / MIT dual license)
- 900+ tests across the workspace
- Example applications (FDTD simulation, transaction monitoring)
- Documentation and tutorials

Published crates are available on crates.io under the `ringkernel-*` namespace.

8.5 Closing Remarks

Fifty years after Hewitt's original actor model paper, 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. RingKernel takes a step toward closing this gap, demonstrating that actor semantics and GPU execution can be unified productively.

We hope RingKernel inspires further research into high-level programming models for heterogeneous computing, making GPU capabilities accessible to a broader audience of developers.

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

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

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.

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

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.

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

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.

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.

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

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

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.

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.

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.

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.

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.

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.

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

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

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

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.

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.

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.

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.

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

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

Table 10: 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 + ...]

B Message Protocol Specification

B.1 H2K Message Types

Listing 19: H2K message type enumeration

```
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
};
```

B.2 K2H Message Types

Listing 20: K2H message type enumeration

```
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
};
```

C Benchmark Reproduction

To reproduce the benchmarks in this paper:

Listing 21: Benchmark commands

```
# 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
```

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