CUDA Wishlist: Features for True Persistent GPU Actors

A Technical Proposal Based on RingKernel Implementation Experience

Michael Ivertowski

February 4, 2026

Abstract

This document outlines CUDA features that would significantly improve the implementation of persistent GPU actors. These recommendations are based on practical experience building RingKernel's persistent actor infrastructure. RingKernel implements persistent GPU actors today using workarounds like cooperative groups, mapped memory, and software barriers. While functional, these approaches have significant limitations. Native CUDA support for actor-model primitives would unlock order-of-magnitude improvements in capability and performance.

Contents

1 Executive Summary

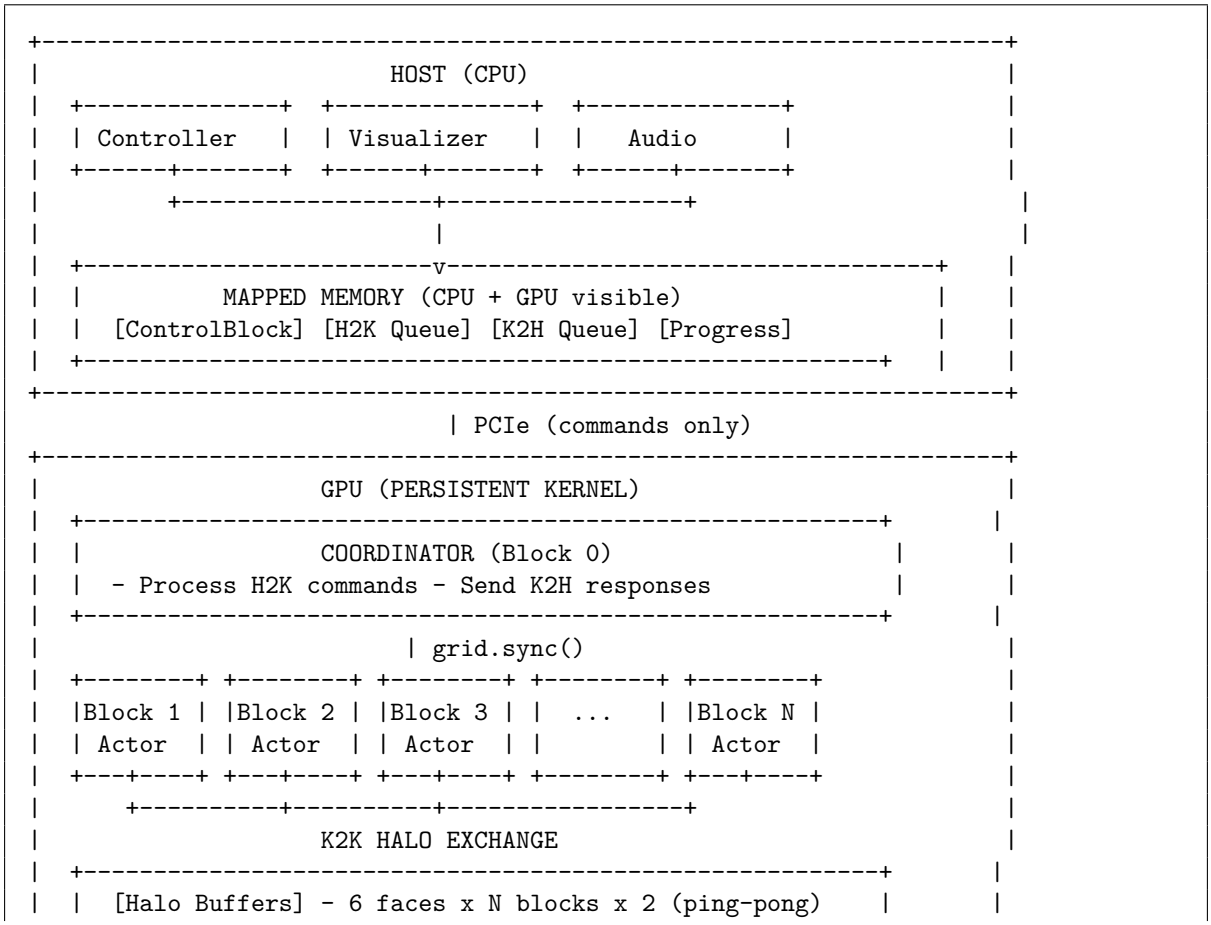
RingKernel implements persistent GPU actors today using workarounds like cooperative groups, mapped memory, and software barriers. While functional, these approaches have significant limitations. Native CUDA support for actor-model primitives would unlock order-of-magnitude improvements in capability and performance.

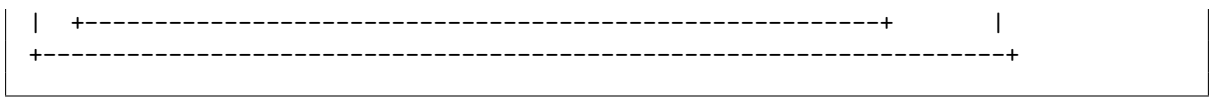
1.1 Key Feature Requests (Priority Order)

1. **Native Host↔Kernel Signaling** — Replace polling with interrupt-driven communication
2. **Kernel-to-Kernel Mailboxes** — First-class inter-block messaging
3. **Dynamic Block Scheduling** — Work stealing and load balancing
4. **Persistent Kernel Preemption** — Cooperative preemption for multi-tenant scenarios
5. **Checkpointing/Migration** — Fault tolerance and GPU failover

2 Current State: How RingKernel Works Around CUDA Limitations

2.1 Architecture Overview





2.2 Current Workarounds and Their Costs

Feature	Current Workaround	Cost
H2K Messaging	Mapped memory + polling	1000+ spin iterations when idle
K2H Responses	Mapped memory + volatile	No interrupt; host must poll
Grid Sync	Cooperative groups	Grid size limited to ∼1024 blocks
K2K Messaging	Device memory + barriers	Extra sync points; no addressing
Fault Tolerance	None	Single GPU failure = total loss
Load Balancing	Static block assignment	Hot spots cause tail latency

3 Feature Request 1: Native Host↔Kernel Signaling

3.1 The Problem

Currently, persistent kernels detect host commands by polling mapped memory:

```
while (true) {
    if (h2k_try_recv(header, slots, &cmd)) {
        process_command(&cmd);
    } else {
        // No work - spin uselessly
        volatile int spin_count = 0;
        for (int i = 0; i < 1000; i++) {
            spin_count++;
        }
    }
    grid.sync();  // Synchronize after each check
}
```

Current: Wasteful spin-wait

This wastes SM cycles and increases power consumption during idle periods.

3.2 Proposed Solution: `cudaKernelNotify()` / `__kernel_wait()`

```
cudaKernelNotify(kernelHandle, KERNEL_NOTIFY_MESSAGE);
```

Host side - signal the kernel

```
__device__ void actor_loop() {
    while (!should_terminate) {
        // Sleep until host signals (zero power consumption)
        __kernel_wait(KERNEL_WAIT_HOST_SIGNAL | KERNEL_WAIT_TIMEOUT,
                      timeout_cycles);

        // Process all available messages
        while (try_recv_message(&msg)) {
            process_message(&msg);
        }

        grid.sync();
    }
}
```

Kernel side - efficient wait

Benefits:

- Zero power consumption during idle periods
- Sub-microsecond wake-up latency (currently $\sim 10\mu$s with polling)
- Reduced PCIe traffic (no constant memory polling)

3.3 Extended API Concept

```
cudaError_t cudaKernelNotify(cudaKernel_t kernel, unsigned int flags);
cudaError_t cudaKernelWaitIdle(cudaKernel_t kernel, unsigned int timeoutMs);
```

Host API

```
__device__ unsigned int __kernel_wait(unsigned int flags, unsigned long timeout
    );
__device__ void __kernel_yield();  // Yield SM cycles to other kernels

// Flags
#define KERNEL_WAIT_HOST_SIGNAL     0x1
#define KERNEL_WAIT_PEER_SIGNAL     0x2
#define KERNEL_WAIT_TIMEOUT         0x4
#define KERNEL_WAIT_ANY            (KERNEL_WAIT_HOST_SIGNAL |
    KERNEL_WAIT_PEER_SIGNAL)
```

Device intrinsics

4 Feature Request 2: Kernel-to-Kernel Mailboxes

4.1 The Problem

Inter-block communication currently requires manual buffer management:

```
void pack_halo_faces(float tile[Z+2][Y+2][X+2], float* halo_buffers,
                     int block_id, int pingpong) {
    // Manually compute offsets
    int block_stride = 6 * face_size * 2;  // 6 faces, 2 ping-pong
    float* my_halo = halo_buffers + block_id * block_stride;

    // Pack each face manually
    if (tid < FACE_SIZE) {
        my_halo[FACE_POS_X * face_stride + tid] = tile[...][...][TILE_X];
        // ... repeat for 5 more faces
    }
}

// Then manually unpack from neighbors
void unpack_halo_faces(float tile[...], const float* halo_buffers,
                       const K2KRouteEntry* route, int pingpong) {
    // Manually read from neighbor's halo buffer
    if (route->neighbors.pos_x >= 0) {
        const float* n_halo = halo_buffers + route->neighbors.pos_x *
   block_stride;
        tile[...][...][TILE_X+1] = n_halo[FACE_NEG_X * face_stride + tid];
    }
    // ... repeat for 5 more directions
}
```

Current: Manual halo exchange

This is error-prone and requires explicit synchronization.

4.2 Proposed Solution: Native Block Mailboxes

```
__shared__ cudaMailbox_t mailbox;

__device__ void init_mailbox() {
    // Register block as actor with neighbors
    cudaMailboxConfig_t config;
    config.maxMessageSize = 64;
    config.queueDepth = 16;
    config.neighbors = compute_neighbors(blockIdx);  // 3D grid topology

    __mailbox_init(&mailbox, &config);
}

// Send to neighbor (non-blocking)
__device__ void send_halo_to_neighbor(int direction, void* data, size_t size) {
    __mailbox_send(&mailbox, direction, data, size, MAILBOX_ASYNC);
}

// Receive from neighbor (blocking with timeout)
__device__ bool receive_halo_from_neighbor(int direction, void* data,
                                           size_t* size, int timeout) {
    return __mailbox_recv(&mailbox, direction, data, size, timeout);
}

// Collective: exchange all halos with sync
```

```
__device__ void exchange_all_halos(void* send_bufs[6], void* recv_bufs[6]) {
    __mailbox_exchange(&mailbox, send_bufs, recv_bufs, MAILBOX_ALL_NEIGHBORS);
}
```

Kernel setup - declare mailbox topology

Benefits:

- Type-safe, bounded mailboxes
- Hardware-accelerated routing (leverage NVLink topology)
- Automatic flow control (backpressure)
- Reduced synchronization (send implies visibility)

4.3 Advanced: Addressable Actor Model

```
typedef struct {
    uint32_t device_id;
    uint32_t kernel_id;
    uint32_t block_id;
} cudaActorAddr_t;

// Send to any actor (multi-GPU aware)
__device__ cudaError_t __actor_send(
    cudaActorAddr_t dest,
    void* message,
    size_t size,
    unsigned int flags
);

// Receive with sender info
__device__ cudaError_t __actor_recv(
    cudaActorAddr_t* sender,  // OUT: who sent this
    void* message,
    size_t* size,
    unsigned int flags
);
```

Globally unique actor addresses

5 Feature Request 3: Dynamic Block Scheduling

5.1 The Problem

Current cooperative kernels have static block assignment. If one region requires more compute (e.g., wavefront in simulation), those blocks become hot spots:

```
Time ->
Block 0: ============================  (done, waiting)
Block 1: ============================  (done, waiting)
Block 2: ========================================  (heavy load)
Block 3: ============================  (done, waiting)
         ^                            ^
         All blocks must wait for slowest block to finish
```

5.2 Proposed Solution: Work Stealing Queues

```
__device__ cudaWorkQueue_t global_work_queue;

// Block claims work dynamically
__device__ void persistent_worker() {
    while (!should_terminate) {
        WorkItem item;

        // Try to get work from global queue
        if (__work_queue_pop(&global_work_queue, &item)) {
            process_work_item(&item);

            // May generate more work
            if (item.spawns_children) {
                for (auto& child : item.children) {
                    __work_queue_push(&global_work_queue, &child);
                }
            }
        } else {
            // Try stealing from other blocks
            if (!__work_queue_steal(&global_work_queue, &item, STEAL_RANDOM)) {
                // No work anywhere - yield
                __kernel_yield();
            }
        }
    }
}
```

Device-wide work queue

Benefits:

- Automatic load balancing
- Better SM utilization
- Adaptive to irregular workloads (graph algorithms, sparse matrices)

5.3 Configuration API

```
cudaWorkQueueConfig_t config;
config.maxItems = 1000000;
config.itemSize = 64;
config.policy = CUDA_WORK_QUEUE_POLICY_LIFO;  // or FIFO, PRIORITY
config.stealingEnabled = true;

cudaError_t cudaWorkQueueCreate(cudaWorkQueue_t* queue, cudaWorkQueueConfig_t*
    config);
cudaError_t cudaWorkQueueSeed(cudaWorkQueue_t queue, void* initialWork, size_t
    count);
```

Host side - configure work queue

6 Feature Request 4: Persistent Kernel Preemption

6.1 The Problem

Persistent kernels monopolize the GPU. In multi-tenant or interactive scenarios, this is problematic:

- Real-time visualization needs occasional GPU access
- Multiple simulations competing for resources
- System needs to respond to user input

Currently, we must either:

1. Design kernels with frequent exit points (loses state)
2. Accept high latency for other GPU work

6.2 Proposed Solution: Cooperative Preemption

```
__device__ void persistent_actor() {
    while (!should_terminate) {
        // Phase 1: Compute (non-preemptible)
        compute_intensive_work();

        // Phase 2: Preemption checkpoint
        __preemption_point();  // GPU can preempt here safely

        // Phase 3: Communication
        exchange_halos();

        __preemption_point();
    }
}
```

Mark preemption-safe points in kernel

```
// Request preemption with priority
cudaError_t cudaKernelPreempt(cudaKernel_t kernel, cudaPreemptConfig_t* config)
    ;

typedef struct {
    unsigned int urgency;        // 0=polite, 100=immediate
    unsigned int timeoutMs;      // How long to wait for checkpoint
    cudaStream_t resumeStream;   // Where to resume after
    void* checkpointBuffer;      // Save kernel state here
} cudaPreemptConfig_t;
```

Host-side control

Benefits:

- GPU time-slicing without kernel redesign
- Interactive responsiveness
- Fair scheduling in multi-tenant environments

7 Feature Request 5: Kernel Checkpointing and Migration

7.1 The Problem

GPU failures are catastrophic for long-running simulations:

- No way to save kernel state
- No way to migrate to another GPU
- Hours of simulation lost on hardware fault

7.2 Proposed Solution: Native Checkpointing

```
__device__ __checkpoint__ float simulation_state[N];
__device__ __checkpoint__ uint64_t step_counter;
__shared__ __checkpoint__ float shared_tile[16][16][16];

// Checkpoint intrinsic
__device__ void checkpoint_if_needed() {
    if (__checkpoint_requested()) {
        // Hardware captures all __checkpoint__ variables
        __checkpoint_save();

        // Optional: kernel continues or terminates
        if (__checkpoint_mode() == CHECKPOINT_AND_EXIT) {
            return;
        }
    }
}
```

Device side - declare checkpointable state

```
// Trigger checkpoint
cudaError_t cudaKernelCheckpoint(
    cudaKernel_t kernel,
    cudaCheckpoint_t* checkpoint,
    unsigned int flags
);

// Restore kernel on same or different GPU
cudaError_t cudaKernelRestore(
    cudaKernel_t* kernel,        // OUT: new kernel handle
    cudaCheckpoint_t checkpoint,
    int deviceId                 // Target GPU (can be different)
);

// Serialize checkpoint for storage
cudaError_t cudaCheckpointSerialize(
    cudaCheckpoint_t checkpoint,
    void** buffer,
    size_t* size
);
```

Host-side API

7.3 Migration Scenario

```
cudaCheckpoint_t checkpoint;
cudaKernelCheckpoint(kernel, &checkpoint, CUDA_CHECKPOINT_URGENT);

// Migrate to GPU 1
cudaSetDevice(1);
cudaKernel_t new_kernel;
cudaKernelRestore(&new_kernel, checkpoint, 1);

// Continue simulation on new GPU
cudaKernelResume(new_kernel);
```

GPU 0 has failing VRAM

Benefits:

- Fault tolerance for long-running simulations
- GPU maintenance without data loss
- Multi-GPU load balancing via migration

8 Feature Request 6: Extended Cooperative Groups

8.1 Current Limitations

Cooperative groups are powerful but limited:

- Grid size capped at ~1024 blocks (device-dependent)
- `grid.sync()` has high overhead (~50μs on large grids)
- No partial sync (e.g., sync only my neighbors)

8.2 Proposed Extensions

8.2.1 Hierarchical Sync Groups

```
// Create groups at different granularities
cg::grid_group grid = cg::this_grid();
cg::super_block_group<4, 4, 4> neighborhood = cg::this_neighborhood();
cg::warp_group warp = cg::this_warp();

// Sync at appropriate level
neighborhood.sync();  // Only sync 64 nearby blocks (faster)
```

8.2.2 Named Barriers

```
// Multiple independent sync phases
__device__ cg::named_barrier_t halo_barrier;
__device__ cg::named_barrier_t compute_barrier;

// Sync only participants in this phase
cg::named_sync(&halo_barrier);     // Only blocks doing halo exchange
cg::named_sync(&compute_barrier);  // Only blocks doing compute
```

8.2.3 Topology-Aware Groups

```
// Group blocks by NVLink connectivity
cg::nvlink_group linked_blocks = cg::this_nvlink_domain();

// Fast sync within NVLink domain (no PCIe)
linked_blocks.sync();

// Get topology info
if (linked_blocks.is_nvlink_connected(neighbor_block)) {
    // Use direct NVLink path
} else {
    // Route through host memory
}
```

9 Feature Request 7: Persistent Kernel Debugging

9.1 The Problem

Debugging persistent kernels is extremely difficult:

- `cuda-gdb` breaks the persistent loop
- `printf` causes synchronization issues
- No way to inspect state without terminating

9.2 Proposed Solution: Non-Intrusive Inspection

```
__device__ __debuggable__ float pressure[N];
__device__ __debuggable__ uint64_t step_counter;
```

Mark variables as debuggable

```
// Read kernel state without stopping it
cudaError_t cudaKernelInspect(
    cudaKernel_t kernel,
    const char* symbolName,
    void* buffer,
    size_t size
);

// Example usage
float pressure_snapshot[N];
cudaKernelInspect(kernel, "pressure", pressure_snapshot, sizeof(
    pressure_snapshot));
uint64_t step;
cudaKernelInspect(kernel, "step_counter", &step, sizeof(step));
```

Host-side inspection

Benefits:

- Debug without disrupting simulation
- Profile memory access patterns
- Monitor convergence in real-time

10 Feature Request 8: Memory Model Enhancements

10.1 Sequentially Consistent Atomics Option

Current CUDA atomics are relaxed by default. For actor mailboxes, we need SC:

```
// Current: Relaxed by default (can reorder)
atomicAdd(&counter, 1);

// Proposed: Explicit memory order
atomicAdd_sc(&counter, 1);       // Sequentially consistent
atomicAdd_acq_rel(&counter, 1);  // Acquire-release
atomicAdd_relaxed(&counter, 1);  // Explicit relaxed
```

10.2 System-Scope Fences with Ordering

```
// Current: Single fence type
__threadfence_system();

// Proposed: Explicit ordering
__threadfence_system_acquire();  // Acquire semantics
__threadfence_system_release();  // Release semantics
__threadfence_system_seq_cst();  // Full sequential consistency
```

10.3 Mapped Memory Coherence Control

```
// Current: Implicit coherence (expensive)
float* mapped = cudaHostAlloc(..., cudaHostAllocMapped);

// Proposed: Explicit coherence domains
cudaCoherentRegion_t region;
cudaCreateCoherentRegion(&region, mapped, size, CUDA_COHERENCE_ON_DEMAND);

// Kernel explicitly requests coherence when needed
__device__ void check_commands() {
    __coherence_acquire(&region);  // Ensure we see host writes
    if (command_pending) {
        process_command();
    }
    __coherence_release(&region);  // Make our writes visible to host
}
```

11 Feature Request 9: Multi-GPU Persistent Kernels

11.1 The Problem

RingKernel currently runs on single GPU. Multi-GPU requires:

- Separate kernel launches per GPU
- Manual NVLink/PCIe routing
- Complex synchronization

11.2 Proposed Solution: Unified Multi-GPU Kernel

```
cudaMultiGpuLaunchConfig_t config;
config.numDevices = 4;
config.devices = {0, 1, 2, 3};
config.blocksPerDevice = 512;
config.topology = CUDA_TOPOLOGY_RING;  // or MESH, TREE, CUSTOM

cudaLaunchCooperativeKernelMultiDevice(&kernel, config);
```

Launch spans multiple GPUs

```
__device__ void distributed_actor() {
    int global_block_id = __multi_gpu_block_id();
    int device_id = __multi_gpu_device_id();

    // Send to block on any GPU - runtime handles routing
    __mailbox_send(dest_global_block_id, data, size,
                   MAILBOX_CROSS_GPU | MAILBOX_ASYNC);

    // Grid sync spans all GPUs
    cg::multi_device_grid grid = cg::this_multi_device_grid();
    grid.sync();  // Sync all 2048 blocks across 4 GPUs
}
```

In kernel - seamless multi-GPU

Benefits:

- Scale simulations beyond single GPU memory
- Transparent NVLink utilization
- Single code path for any GPU count

12 Feature Request 10: Actor Lifecycle Management

12.1 The Problem

Currently, all blocks must participate from start to finish. We cannot:

- Spawn new actors dynamically
- Terminate individual actors
- Change the number of actors during execution

12.2 Proposed Solution: Dynamic Actor Registry

```
__device__ cudaActor_t spawn_actor(void (*handler)(void*), void* state) {
    return __actor_spawn(handler, state, ACTOR_FLAGS_DEFAULT);
}
```

Device-side actor creation

```
cudaError_t cudaActorSpawn(
    cudaKernel_t kernel,
    cudaActor_t* actor,
    cudaActorConfig_t* config
);

cudaError_t cudaActorTerminate(cudaActor_t actor, int exitCode);

cudaError_t cudaActorQuery(
    cudaActor_t actor,
    cudaActorInfo_t* info  // State, message count, etc.
);
```

Host-side actor management

Benefits:

- Elastic actor systems (scale up/down)
- Resource efficiency (terminate idle actors)
- Dynamic task graphs

13 Implementation Priority Matrix

Feature	Impact	Complexity	Priority
Host↔Kernel Signaling	Very High	Medium	**P0**
K2K Mailboxes	Very High	High	**P0**
Dynamic Scheduling	High	High	**P1**
Preemption	High	Very High	**P1**
Checkpointing	High	Very High	**P2**
Extended Cooperative Groups	Medium	Medium	**P2**
Debugging Tools	Medium	Low	**P2**
Memory Model Enhancements	Medium	Medium	**P3**
Multi-GPU Kernels	Very High	Very High	**P3**
Dynamic Actor Registry	Medium	High	**P3**

14 Conclusion

The persistent GPU actor model represents a paradigm shift from the traditional "launch, compute, exit" GPU programming model. NVIDIA has made significant progress with cooperative groups and persistent threads, but native support for actor-model primitives would unlock the next generation of GPU applications:

- **Real-time simulations** with sub-millisecond host interaction
- **Distributed GPU computing** with seamless multi-GPU scaling
- **Fault-tolerant HPC** with checkpoint/restart
- **Interactive scientific visualization** with responsive compute

RingKernel demonstrates that persistent GPU actors are viable today, but with significant engineering complexity. Native CUDA support would make these patterns accessible to the broader GPU programming community.

A Current RingKernel Performance

Benchmark results on RTX Ada (AD102):

Metric	Traditional Kernel	Persistent Actor	Delta
Command Injection	317 μs	0.03 μs	**11,327× faster**
Query Latency	0.01 μs	0.01 μs	Same
Single Step	3.2 μs	163 μs	51× slower
Mixed Workload (60 FPS)	40.5 ms	15.3 ms	**2.7× faster**

The persistent model excels at **interactive latency** while traditional excels at **batch throughput**. Native CUDA support could eliminate this tradeoff.

B Related Work

- **NVIDIA Cooperative Groups** (CUDA 9+): Foundation for grid-wide sync
- **AMD HIP Persistent Kernels**: Similar exploration in ROCm ecosystem
- **Vulkan/SPIR-V**: Different approach via command buffers
- **SYCL**: Exploring persistent execution model
- **Academic**: "GPUfs" (ASPLOS '13), "GPUnet" (OSDI '14), "Tango" (ISCA '21)

C Feedback Channel

We welcome discussion on these proposals:

- GitHub: `https://github.com/mivertowski/ringkernel`
- GTC: Annual Birds-of-a-Feather session on GPU actors

Contact Information

Michael Ivertowski

michael.ivertowski@ch.ey.com

Document generated from RingKernel project documentation.

https://github.com/mivertowski/ringkernel