CUDA Wishlist: Features for True Persistent GPU Actors

A Technical Proposal Based on RingKernel and RustGraph Implementation Experience

Michael Ivertowski

February 6, 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 and RustGraph's GPU-native living graph database. RingKernel implements persistent GPU actors today using workarounds like cooperative groups, mapped memory, and software barriers. RustGraph extends this model to graph analytics, where nodes and edges are persistent GPU actors that maintain analytics state (PageRank, BFS, Connected Components) via message passing across multi-layer hypergraph structures. 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 for stencil computations, while RustGraph extends the model to graph analytics—nodes and edges are persistent GPU actors that maintain PageRank, BFS distances, component IDs, and 64+ other analytics via K2K message passing. Both systems use 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 with irregular graph topologies
3. **Dynamic Block Scheduling** — Work stealing and load balancing for power-law graphs
4. **Persistent Kernel Preemption** — Cooperative preemption for multi-tenant scenarios
5. **Checkpointing/Migration** — Fault tolerance, GPU failover, and live graph snapshots
6. **GPU-Side Convergence Detection** — Hardware-accelerated quiescence for iterative algorithms
7. **Selective Grid Sync** — Predicate-based partial synchronization for layer-scoped analytics
8. **Variable-Fan-Out K2K Messaging** — Irregular neighbor messaging for graph topologies
9. **Multi-Algorithm BSP Execution** — Concurrent actor behaviors via bitmask selection
10. **Warp-Level Actor Granularity** — Sub-block actor assignment for hub nodes

2 Current State: How We Work Around CUDA Limitations

2.1 RingKernel Architecture

```
+---------------------------------------------------------------------+
|                         HOST (CPU)                                  |
|  +--------------+  +--------------+  +--------------+               |
|  | Controller   |  | Visualizer   |  |   Audio      |               |
|  +------+-------+  +------+-------+  +------+-------+               |
|         +------------------+-----------------+                       |
|                            |                                         |
|  +-------------------------v------------------------------------+   |
|  |          MAPPED MEMORY (CPU + GPU visible)                   |   |
|  |  [ControlBlock] [H2K Queue] [K2H Queue] [Progress]           |   |
|  +----------------------------------------------------------+   |   |
+---------------------------------------------------------------------+
                              | PCIe (commands only)
+---------------------------------------------------------------------+
|                    GPU (PERSISTENT KERNEL)                          |
|  +----------------------------------------------------------+      |
|  |                 COORDINATOR (Block 0)                        |        |
|  |  - Process H2K commands - Send K2H responses                 |        |
|  +-----------------------------------------------------------+        |
|                          | grid.sync()                                 |
|  +---------+ +---------+ +---------+ +---------+ +---------+           |
|  |Block 1 | |Block 2 | |Block 3 | |  ...    | |Block N |              |
|  | Actor  | | Actor  | | Actor  | |         | | Actor  |               |
|  +---+----+ +---+----+ +---+----+ +---------+ +---+----+               |
|      +----------+----------+----------------+                          |
|                    K2K HALO EXCHANGE                                   |
|  +-----------------------------------------------------------+        |
|  |  [Halo Buffers] - 6 faces x N blocks x 2 (ping-pong)    |        |
|  +-----------------------------------------------------------+        |
+----------------------------------------------------------------------+
```

2.2 RustGraph Architecture

RustGraph extends the persistent actor model to graph analytics. Each node is a 256-byte GPU-resident actor; edges are 64-byte actors. Analytics emerge from message passing in a BSP (Bulk Synchronous Parallel) tick loop:

```
+----------------------------------------------------------------------+
|                    GPU (PERSISTENT GRAPH RUNTIME)                     |
|                                                                       |
|  BSP Tick Loop:                                                       |
|  [1. Propagate] -> [2. Receive] -> [3. Apply] -> [4. Converge]       |
|                                                                       |
|  +---------+ +---------+ +---------+ +---------+ +---------+          |
|  |Node 0  | |Node 1  | |Node 2  | |  ...    | |Node N  |            |
|  |256B    | |256B    | |256B    | |         | |256B    |            |
|  |PR,BFS  | |PR,BFS  | |PR,BFS  | |         | |PR,BFS  |            |
|  |CC,EV   | |CC,EV   | |CC,EV   | |         | |CC,EV   |            |
|  +---+----+ +---+----+ +---+----+ +---------+ +---+----+              |
|      |          |          |                    |                   |
|      +----------+----------+--------------------+                   |
|          K2K RING BUFFER INBOXES (512 msg/node)                      |
|                                                                       |
|  Multi-Algorithm: active_algos bitmask (PR|CC|EV|BFS)                 |
|  Layer Scoping:   node.layer field -> flag-based filtering            |
|  Kernel Modes:    NodeCentric | SoA | EdgeCentric | Tiled | Auto    |
+----------------------------------------------------------------------+
```

2.3 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
Additional workarounds from RustGraph graph analytics:		
Graph K2K	Ring buffer inboxes (512 msg)	Fixed fan-out; no topology awareness
Convergence	Host-side polling per tick	Round-trip per convergence check
Layer Scoping	Flag-based filtering (CONVERGED)	All nodes visited even when inactive
Multi-Algorithm	Bitmask + branching	Warp divergence on mixed algorithms
Hub Nodes	Warp-cooperative (32 threads)	Manual warp-level programming

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. CUDA provides `__nanosleep()` (since CUDA 10) for power-efficient polling, which reduces SM power consumption during spin-waits but does not eliminate polling entirely—there is still no interrupt-driven host→kernel notification mechanism.

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 ~10μs 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 Current State: Hopper DSMEM and Thread Block Clusters

Hopper (SM 9.0+) introduces **Distributed Shared Memory (DSMEM)** and **Thread Block Clusters**, which provide a partial solution: blocks within a cluster (up to 8–16 blocks) can directly access each other's shared memory via `cluster.map_shared_rank()`. This enables mailbox-like semantics within a cluster. However, clusters are limited to a single SM group (typically 8–16 blocks) and do not support grid-wide or cross-kernel messaging. The gap remains for:

- Grid-wide K2K messaging beyond cluster boundaries
- Addressable actor mailboxes with routing
- Variable-topology messaging (graph adjacency, not just spatial neighbors)

4.3 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)

4.4 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

4.5 Graph-Irregular Messaging Extension

Beyond structured halo exchange, graph analytics require **irregular** messaging where each node sends to a variable number of neighbors determined by the adjacency structure. RustGraph currently implements this with per-node ring buffer inboxes (512 messages, lock-free atomics), but native support would be far more efficient:

```
// Send to all graph neighbors (variable fan-out)
__device__ void propagate_to_neighbors(
    uint64_t node_id,
    const uint32_t* csr_offsets,   // CSR row pointers
    const uint32_t* csr_neighbors, // CSR column indices
    void* message, size_t size
) {
    uint32_t start = csr_offsets[node_id];
    uint32_t end   = csr_offsets[node_id + 1];

    // Hardware-accelerated scatter to variable neighbors
    for (uint32_t i = start; i < end; i++) {
        __mailbox_send_csr(csr_neighbors[i], message, size,
                           MAILBOX_ASYNC | MAILBOX_COALESCE);
    }
}

// Receive with aggregation (e.g., sum for PageRank)
__device__ float __mailbox_recv_reduce(
    cudaMailbox_t* mailbox,
    cudaReduceOp_t op  // SUM, MAX, MIN, OR
);
```

CSR-driven irregular messaging

This pattern occurs in every graph analytics algorithm: PageRank (scatter rank/degree), BFS (scatter distance+1), Connected Components (scatter min component ID). Hardware support for CSR-driven scatter with built-in reduction would dramatically reduce the software complexity.

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 Current State: Device Graph Launch and Green Contexts

CUDA 12.0+ provides **Device Graph Launch**, allowing kernels to launch CUDA graphs from the device side, enabling some dynamic work generation. CUDA 13.1 introduces **Green Contexts**, which partition GPU resources (SMs, memory bandwidth) into isolated contexts. These provide building blocks for scheduling but do not offer work-stealing queues, dynamic load balancing within a persistent kernel, or priority-based scheduling across blocks.

5.3 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)

5.4 Graph-Specific Motivation: Power-Law Degree Distributions

Real-world graphs follow power-law degree distributions where a small number of "hub" nodes have 10,000+ neighbors while most nodes have < 10. This creates extreme load imbalance:

```
Node degree distribution (log-scale):
  1-10 neighbors:     85% of nodes  (fast to process)
  10-100 neighbors:   12% of nodes
  100-1000 neighbors:  2.5% of nodes
  1000+ neighbors:     0.5% of nodes (bottleneck: 50x slower)
```

RustGraph addresses this with multiple kernel modes (NodeCentric, EdgeCentric, Tiled, Warp-Cooperative) selected at launch time based on graph characteristics. But the optimal mode varies *within* a single graph—hub nodes need edge-centric processing while leaf nodes need node-centric. Hardware work-stealing queues with heterogeneous work items would allow mixing strategies dynamically within a single kernel launch.

5.5 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

CUDA does support hardware-level compute preemption (since Pascal/SM 6.0), which the driver uses for context switching and watchdog timers. However, this preemption is **not user-controllable**—there are no APIs to request preemption, set priorities, or define safe preemption points. The hardware preemption also does not support saving/restoring application-level state.

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

CUDA provides `cuCheckpointProcessSave/Restore` (driver API) for process-level GPU memory checkpointing, and CUPTI provides checkpoint/replay for profiler instrumentation. However, these save **GPU memory allocations only**—they do not capture kernel execution state (registers, program counter, shared memory) and cannot checkpoint a running persistent kernel at an application-defined safe point.

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

7.4 Live Graph Snapshots (Snapshot-Without-Stop)

RustGraph's temporal time machine takes named snapshots of the full graph state (all node analytics, edge weights, hyperedge aggregates) for period comparison. Currently, this requires a host-side copy of all GPU state vectors—$N \times 256\text{B} + E \times 64\text{B}$ per snapshot:

```
// Take a consistent snapshot without stopping the persistent kernel
cudaCheckpoint_t snapshot;
cudaKernelCheckpoint(kernel, &snapshot,
    CUDA_CHECKPOINT_NONBLOCKING |    // Don't stop the kernel
    CUDA_CHECKPOINT_CONSISTENT  |    // Wait for BSP phase boundary
    CUDA_CHECKPOINT_GPU_LOCAL);      // Keep snapshot in GPU memory

// Compare two GPU-resident snapshots (no host transfer)
cudaCheckpointDiff_t diff;
cudaCheckpointCompare(snapshot_a, snapshot_b, &diff,
    CUDA_DIFF_STRUCTURAL | CUDA_DIFF_ANALYTICS);
```

Proposed: GPU-side snapshot without stopping the kernel

For 100K-node graphs, each snapshot is ∼26 MB. With 50 max snapshots, the entire time machine fits in GPU memory. Hardware support for consistent-at-BSP-boundary snapshots would eliminate the current host round-trip overhead.

Benefits:

- Fault tolerance for long-running simulations
- GPU maintenance without data loss
- Multi-GPU load balancing via migration
- Zero-downtime graph snapshots for temporal analytics
- GPU-resident snapshot comparison without host transfer

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 Current State: Hopper Thread Block Clusters

Hopper (SM 9.0+) introduces `cluster_group` with `cluster.sync()`, providing synchronization across a cluster of blocks (up to 16). This is a significant step toward hierarchical sync, but clusters are fixed at launch time, limited in size, and do not support the dynamic, topology-aware, or named barrier patterns proposed below. Note: `cudaLaunchCooperativeKernelMultiDevice` was removed in CUDA 13.0, so multi-GPU grid sync is no longer available through cooperative groups.

8.3 Proposed Extensions

8.3.1 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)
```

8.3.2 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
```

8.3.3 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: Mapped Memory Coherence Control

10.1 Current State: libcu++/CCCL Already Provides Ordered Atomics

NVIDIA's libcu++ (part of CCCL) already provides C++ standard memory ordering for atomics and fences. The original motivation for this feature—sequentially consistent atomics and ordered fences—is **already available**:

```
#include <cuda/atomic>

// Sequentially consistent atomic (already works)
cuda::atomic<int, cuda::thread_scope_device> counter;
counter.fetch_add(1, cuda::memory_order_seq_cst);

// Acquire-release semantics (already works)
counter.fetch_add(1, cuda::memory_order_acq_rel);

// Scoped fences (already works)
cuda::atomic_thread_fence(cuda::memory_order_acquire,
                          cuda::thread_scope_system);
cuda::atomic_thread_fence(cuda::memory_order_release,
                          cuda::thread_scope_system);

// Memory Synchronization Domains (Hopper+, already available)
// Fine-grained coherence control via domain-scoped fences
```

Existing: libcu++ ordered atomics (CCCL)

RingKernel currently uses legacy CUDA intrinsics (`atomicAdd`, `__threadfence_system`) which lack explicit ordering. Migrating to libcu++ would address the atomics and fence requirements. **The remaining gap is mapped memory coherence control**, which libcu++ does not address.

10.2 Remaining Request: Explicit Mapped Memory Coherence Domains

Persistent actors continuously poll mapped memory for host commands. Current mapped memory provides implicit coherence that is either always-on (expensive) or relies on volatile semantics with no formal guarantees. Explicit coherence domains would allow actors to request visibility only when needed:

```
// Current: Implicit coherence (expensive, always-on)
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
}
```

This is distinct from Memory Synchronization Domains (Hopper+), which control L2 cache partitioning between domains but do not provide explicit acquire/release semantics for mapped CPU↔GPU memory regions.

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 Current State: Multi-GPU Cooperative Launch Was Removed

`cudaLaunchCooperativeKernelMultiDevice` was deprecated in CUDA 11.3 and **removed entirely in CUDA 13.0**. NVIDIA's current recommendation is **NVSHMEM** for multi-GPU communication, which provides one-sided put/get operations and collective synchronization across GPUs. However, NVSHMEM is designed for bulk-synchronous SPMD programs, not persistent actor systems with irregular messaging patterns.

The gap: there is no CUDA-native way to launch a single persistent kernel that spans multiple GPUs with unified block addressing and cross-GPU mailbox routing.

11.3 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

cudaLaunchMultiGpuPersistentKernel(&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

11.4 Graph Partitioning Considerations

RustGraph's multi-GPU module implements graph partitioning strategies (vertex-cut, edge-cut, 2D partitioning) to distribute graphs across GPUs. The challenge is that cross-partition edges require K2K messages across GPU boundaries. With NVLink, this is ∼10× slower than intra-GPU messaging; with PCIe, ∼100× slower.

Hardware-aware partitioning hints would significantly improve performance:

```
// Query NVLink topology for partition planning
cudaMultiGpuTopology_t topology;
cudaGetMultiGpuTopology(&topology);

// Get bandwidth between GPU pairs (for edge-cut cost model)
float bw = topology.bandwidth[gpu_a][gpu_b];  // GB/s
float lat = topology.latency[gpu_a][gpu_b];   // ns

// Hint: these node ranges communicate frequently
cudaMultiGpuAffinityHint_t hint;
hint.nodeRange = {start, end};
hint.preferDevice = gpu_id;
hint.communicatesWith = {neighbor_ranges};
cudaSetMultiGpuAffinity(&hint);
```

Topology-aware graph partitioning

Benefits:

- Scale simulations beyond single GPU memory
- Transparent NVLink utilization
- Single code path for any GPU count
- Topology-aware graph partitioning for minimal cross-GPU traffic

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 Feature Request 11: GPU-Side Convergence Detection

13.1 The Problem

Iterative graph algorithms (PageRank, Label Propagation, Connected Components) run in a BSP tick loop until convergence—when no node changes state beyond a threshold. Currently, convergence detection requires either:

- A host-side reduction after each tick (PCIe round-trip per iteration)
- A device-wide atomic counter with `grid.sync()` (high overhead)

RustGraph uses `grid.sync()` plus a shared atomic counter to detect quiescence. For PageRank on 100K nodes, convergence typically takes 15–30 iterations, meaning 15–30 full grid synchronizations just for the convergence check.

13.2 Current State: Graph Conditional Nodes and Block-Level Reduction

CUDA 12.4+ introduces **Graph conditional WHILE nodes**, which enable GPU-resident loops that repeat a subgraph until a device-side condition is met—without host involvement. Combined with block-level `cg::reduce()` for cooperative groups, this provides a partial solution for convergence loops in graph-launched workloads. However, this requires structuring work as CUDA Graphs rather than persistent kernels, and the reduction is block-scoped, not grid-wide. For persistent actor systems that run a continuous BSP tick loop, there is no hardware-accelerated grid-wide convergence primitive.

13.3 Proposed Solution: Hardware Convergence Counter

```
// Device-wide convergence counter (hardware-managed)
__device__ cudaConvergence_t convergence;

__device__ void pagerank_apply(uint64_t node_id) {
    float new_rank = compute_new_rank(node_id);
    float old_rank = nodes[node_id].pagerank;
    float diff = fabsf(new_rank - old_rank);

    nodes[node_id].pagerank = new_rank;

    // Hardware accumulates max diff across all threads
    __convergence_report(&convergence, diff);
}

// After grid.sync(), check convergence without host round-trip
__device__ bool check_convergence() {
    float max_diff = __convergence_result(&convergence);
    return max_diff < EPSILON;
}
```

Hardware convergence detection

```
// Query convergence status without stopping the kernel
cudaError_t cudaKernelConvergenceStatus(
    cudaKernel_t kernel,
    float* maxDiff,          // OUT: maximum change in last tick
    uint32_t* activeNodes,   // OUT: nodes that changed
    uint32_t* totalIterations // OUT: ticks since init
);
```
Host-side API

Benefits:

- Eliminates PCIe round-trip per iteration for convergence checking
- Hardware-accelerated reduction (faster than software atomics)
- Enables fully autonomous kernel convergence without host involvement
- Applicable to any iterative algorithm: PageRank, Jacobi, GMRES, etc.

14 Feature Request 12: Selective Grid Sync (Sync-by-Predicate)

14.1 The Problem

RustGraph's multi-layer hypergraph organizes nodes into layers (Governance, Process, Accounting). Layer-scoped analytics run PageRank/BFS/CC within a single layer by marking non-layer nodes as `CONVERGED` and clearing their propagation flags. However, `grid.sync()` still synchronizes *all* blocks—even those processing only inactive nodes:

```
Layer-scoped PageRank (Layer 1 = 500 nodes, Total = 10,000 nodes):

Block 0:  [Layer 1 nodes] -> ACTIVE, propagating
Block 1:  [Layer 1 nodes] -> ACTIVE, propagating
Block 2:  [Layer 2 nodes] -> CONVERGED, skip  (but still syncs!)
Block 3:  [Layer 3 nodes] -> CONVERGED, skip  (but still syncs!)
...
Block 99: [Layer 3 nodes] -> CONVERGED, skip  (but still syncs!)

grid.sync() waits for ALL 100 blocks, even though only 2 are doing work.
```

14.2 Current State: Cluster-Scoped Partial Sync on Hopper

Hopper's thread block clusters support partial synchronization within a cluster using bitmasked `mbarrier` operations—a block can wait on a subset of cluster members (up to 16 blocks). This enables partial sync at the cluster level but does not extend to grid-wide predicate-based synchronization. For graphs with 100+ blocks where only a subset is active, cluster-level partial sync is insufficient.

14.3 Proposed Solution: Predicate-Based Partial Sync

```
// Each block declares its activity status
__device__ bool my_block_active = has_active_nodes_in_layer(blockIdx.x,
    target_layer);

// Sync only active blocks (inactive blocks skip the barrier)
cg::grid_group grid = cg::this_grid();
grid.sync_if(my_block_active);

// Or: sync by layer membership
__device__ uint8_t my_layer = determine_layer(blockIdx.x);
cg::predicate_sync(grid, my_layer == target_layer);
```

Sync only blocks that match a predicate

```
// Create a sync group from active blocks only
cg::dynamic_group active_blocks = cg::create_group_if(
    grid, has_active_nodes(blockIdx.x)
);

// Sync only the active subset (O(active_blocks) instead of O(all_blocks))
active_blocks.sync();
```

Alternative: dynamic sync groups

Benefits:

- Layer-scoped analytics sync only relevant blocks (5–20× fewer blocks)
- Inactive blocks can be yielded to other kernels
- Generalizes to any partitioned workload (multi-physics, domain decomposition)
- Reduces grid sync overhead proportional to active fraction

15 Feature Request 13: Multi-Algorithm BSP Execution

15.1 The Problem

RustGraph runs multiple graph algorithms concurrently on the same graph using a bitmask (`active_algos: ALGO_PAGERANK|ALGO_CC|ALGO_BFS`). Each node processes all active algorithms in the same BSP tick. This creates warp divergence:

```
__device__ void apply_phase(uint64_t node_id) {
    if (active_algos & ALGO_PAGERANK) {
        apply_pagerank(node_id);  // Branch A
    }
    if (active_algos & ALGO_CC) {
        apply_cc(node_id);        // Branch B
    }
    if (active_algos & ALGO_BFS) {
        apply_bfs(node_id);       // Branch C
    }
    // Warp divergence: threads in same warp may take different branches
    // depending on node flags (e.g., BFS only propagates from frontier)
}
```

Current: branching on algorithm bitmask

15.2 Proposed Solution: Algorithm-Parallel Execution Lanes

```
// Define algorithm behaviors as independent execution lanes
cudaAlgoLane_t lanes[3];
lanes[0] = {.kernel = pagerank_kernel, .mask = ALGO_PAGERANK};
lanes[1] = {.kernel = cc_kernel,       .mask = ALGO_CC};
lanes[2] = {.kernel = bfs_kernel,      .mask = ALGO_BFS};

// Launch all algorithms in parallel lanes (same data, different compute)
cudaLaunchAlgorithmParallel(lanes, 3, node_data, edge_data,
    CUDA_ALGO_SYNC_BSP);  // BSP: all lanes sync at tick boundary

// Each lane runs without warp divergence from other algorithms
```

Hardware algorithm lanes

```
__device__ void pagerank_lane(uint64_t node_id) {
    // Pure PageRank - no branching on other algorithms
    float sum = receive_pagerank_messages(node_id);
    float new_rank = 0.15/N + 0.85 * sum;
    nodes[node_id].pagerank = new_rank;
    propagate_rank(node_id, new_rank);
}

// Lanes share node state but execute independently
// Hardware ensures visibility at BSP tick boundaries
```

Device-side: per-lane tick loop

Benefits:

- Eliminates warp divergence from multi-algorithm branching
- Each algorithm runs at full warp efficiency
- BSP synchronization preserved across lanes

- Shared node state with hardware-enforced consistency at tick boundaries

16 Feature Request 14: GPU-Resident Graph Data Structures

16.1 The Problem

Graph analytics kernels spend significant time traversing CSR (Compressed Sparse Row) structures to find neighbors. This involves indirect memory access patterns with poor cache locality:

```
__device__ void scatter_to_neighbors(uint64_t node_id) {
    uint32_t start = csr_offsets[node_id];      // Random read 1
    uint32_t end   = csr_offsets[node_id + 1]; // Sequential (cached)

    for (uint32_t i = start; i < end; i++) {
        uint32_t neighbor = csr_neighbors[i];  // Random read 2
        float* inbox = &inboxes[neighbor];     // Random read 3
        atomicAdd(inbox, message);             // Random atomic
    }
    // 3 levels of indirection per neighbor, poor L2 utilization
}
```

Current: software CSR traversal

RustGraph mitigates this with `__ldg()` L1 caching in tiled kernels, but the fundamental indirection overhead remains.

16.2 Proposed Solution: Hardware CSR Iterator

```
// Declare graph topology to hardware
cudaGraphTopology_t topo;
cudaCreateGraphTopology(&topo, csr_offsets, csr_neighbors,
    num_nodes, num_edges, CUDA_GRAPH_CSR);

// Hardware-assisted neighbor iteration
__device__ void scatter_to_neighbors(uint64_t node_id) {
    cudaNeighborIterator_t iter;
    __graph_neighbors_begin(&iter, &topo, node_id);

    while (__graph_neighbors_next(&iter)) {
        uint32_t neighbor = __graph_neighbor_id(&iter);
        // Hardware prefetches next neighbor while processing current
        atomicAdd(&inboxes[neighbor], message);
    }
}

// Hardware-level optimizations:
// - Prefetch next CSR segment into L1 while processing current
// - Coalesce neighbor accesses across warps
// - Vectorized CSR reads (128-bit loads for 4 neighbors at once)
```

Hardware-accelerated neighbor iteration

Benefits:

- Hardware prefetching of CSR segments eliminates stalls
- Coalesced neighbor access across warps
- Reduced register pressure (hardware manages iteration state)
- Applicable to any sparse graph algorithm (GNN, SSSP, triangle counting)

17 Feature Request 15: Warp-Level Actor Granularity

17.1 The Problem

In the standard actor model, one thread block = one actor. But graph analytics have extreme degree heterogeneity: hub nodes with 10,000+ neighbors need far more compute than leaf nodes with 1–2 neighbors. RustGraph implements warp-cooperative kernels where 32 threads collaborate on a single hub node:

```
__global__ void warp_coop_pagerank(/* ... */) {
    int warp_id = threadIdx.x / 32;
    int lane_id = threadIdx.x % 32;
    int node_id = blockIdx.x * warps_per_block + warp_id;

    // Each warp processes one node cooperatively
    uint32_t start = csr_offsets[node_id];
    uint32_t end   = csr_offsets[node_id + 1];
    uint32_t degree = end - start;

    // Warp-strided neighbor iteration
    float sum = 0.0f;
    for (uint32_t i = start + lane_id; i < end; i += 32) {
        uint32_t neighbor = csr_neighbors[i];
        sum += nodes[neighbor].pagerank / nodes[neighbor].out_degree;
    }

    // Warp-level reduction
    sum = __shfl_down_sync(0xFFFFFFFF, sum, 16);
    sum = __shfl_down_sync(0xFFFFFFFF, sum, 8);
    // ... full butterfly reduction
}
```

Current: manual warp-cooperative processing

This is error-prone and requires manual warp-level programming for every algorithm.

17.2 Current State: Cooperative Groups Warp Partitions

CUDA cooperative groups provide `tiled_partition<N>` for static warp subdivision and `labeled_partition` for dynamic grouping based on a label value. `coalesced_threads()` captures the active thread mask. These provide the building blocks for sub-warp computation but require manual orchestration—there is no concept of assigning actors to variable-sized thread groups based on workload, no `__actor_context()`, and no automatic load-balanced mapping of actors to hardware threads.

17.3 Proposed Solution: Sub-Block Actor Assignment

```
// Declare actors with variable compute requirements
cudaActorConfig_t config;
config.granularity = CUDA_ACTOR_ADAPTIVE;  // Not fixed to block
config.minThreads = 1;      // Leaf nodes: 1 thread
config.maxThreads = 32;     // Hub nodes: full warp
config.assignmentHint = degree_array;  // Per-actor work estimate

// Hardware assigns threads to actors based on degree
cudaLaunchAdaptiveActors(kernel, config, num_actors);
```

```
// In kernel: hardware provides actor context
__device__ void actor_compute() {
    cudaActorContext_t ctx = __actor_context();
    int my_threads = ctx.num_threads;  // 1 for leaves, 32 for hubs
    int my_lane    = ctx.thread_rank;  // 0..num_threads-1

    // Same code works for any thread count
    uint32_t start = csr_offsets[ctx.actor_id];
    uint32_t end   = csr_offsets[ctx.actor_id + 1];

    float sum = 0.0f;
    for (uint32_t i = start + my_lane; i < end; i += my_threads) {
        sum += neighbor_value(csr_neighbors[i]);
    }

    // Hardware-managed reduction across actor's threads
    sum = __actor_reduce(ctx, sum, CUDA_REDUCE_SUM);
}
```

Hardware sub-block actor assignment

Benefits:

- Eliminates manual warp-level programming
- Automatic load balancing: hubs get more threads, leaves fewer
- Same kernel code handles all degree ranges
- Hardware-optimized reduction within actor thread groups

18 Implementation Priority Matrix

Feature	Impact	Complexity	Priority
Core persistent actor infrastructure:			
Host↔Kernel Signaling	Very High	Medium	**P0**
K2K Mailboxes + Graph Messaging	Very High	High	**P0**
GPU-Side Convergence Detection	Very High	Low	**P0**
Dynamic Scheduling (Power-Law)	High	High	**P1**
Preemption	High	Very High	**P1**
Selective Grid Sync	High	Medium	**P1**
Graph analytics acceleration:			
Warp-Level Actor Granularity	High	High	**P1**
Multi-Algorithm BSP Execution	High	High	**P2**
GPU-Resident Graph Data Structures	Very High	Very High	**P2**
System-level capabilities:			
Checkpointing + Live Snapshots	High	Very High	**P2**
Extended Cooperative Groups	Medium	Medium	**P2**
Debugging Tools	Medium	Low	**P2**
Mapped Memory Coherence Control	Medium	Medium	**P3**
Multi-GPU Kernels	Very High	Very High	**P3**
Dynamic Actor Registry	Medium	High	**P3**

19 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
- **GPU-native graph analytics** with always-current PageRank, BFS, CC, and 64+ algorithms
- **Distributed GPU computing** with seamless multi-GPU scaling
- **Fault-tolerant HPC** with checkpoint/restart and live graph snapshots
- **Interactive scientific visualization** with responsive compute
- **Multi-layer hypergraph analytics** with layer-scoped computation and cross-layer traversal

RingKernel and RustGraph demonstrate that persistent GPU actors are viable today for both structured stencil computations and irregular graph analytics, but with significant engineering complexity. The five new feature requests (convergence detection, selective sync, multi-algorithm BSP, GPU-resident graph structures, warp-level actors) address the specific pain points of graph workloads that complement the original structured-grid requirements. 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 Current RustGraph Performance

Benchmark results on RTX 2000 Ada (AD106), persistent graph actor model:

Algorithm	100K Nodes	125K Nodes	150K Nodes
PageRank (ME/s)	176–189	**258**	**241**
Connected Components (ME/s)	8–13	9–12	8–12
BFS (ME/s)	19–32	21–30	18–30

Metric	Value
PageRank scaling exponent	**1.18** (superlinear)
Node state size	256 bytes (`#[repr(C, align(256))]`)
Edge state size	64 bytes (`#[repr(C, align(64))]`)
K2K inbox capacity	512 messages/node (lock-free ring buffer)
Concurrent algorithms	Up to 4 (PR + CC + BFS + EV via bitmask)
Temporal snapshot size	∼26 MB per snapshot (100K nodes)
Layer-scoped analytics	3 layers, flag-based filtering
Kernel modes (auto-selected)	NodeCentric, SoA, EdgeCentric, Tiled, Warp-Coop

Key observations relevant to CUDA feature requests:

- **Convergence overhead**: Host-side convergence check adds ∼5% overhead per tick. GPU-side convergence detection (Feature 11) would eliminate this entirely.
- **Layer-scoped waste**: When running analytics on a single layer (e.g., 500 of 10,000 nodes), 95% of blocks sync unnecessarily. Selective grid sync (Feature 12) would reduce sync cost by 20×.
- **Hub node bottleneck**: Warp-cooperative kernels achieve ∼258 ME/s on PageRank vs. ∼189 ME/s for node-centric. Hardware sub-block actors (Feature 15) would make this automatic.
- **Multi-algorithm divergence**: Running PR+CC+BFS simultaneously causes ∼15% warp divergence. Algorithm-parallel lanes (Feature 13) would eliminate this.

C Related Work

- **NVIDIA Cooperative Groups** (CUDA 9+): Foundation for grid-wide sync
- **Thread Block Clusters / DSMEM** (Hopper, CUDA 12+): Cluster-scoped shared memory and partial sync (up to 16 blocks)
- **CUDA Graphs + Conditional Nodes** (CUDA 12.4+): Device-side control flow, GPU-resident loops
- **Green Contexts** (CUDA 13.1): GPU resource partitioning across workloads
- **libcu++/CCCL**: C++ standard atomics and memory ordering for CUDA
- **NVSHMEM**: Multi-GPU communication (replacement for removed multi-device cooperative launch)
- **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)
- **GPU Graph Frameworks**: Gunrock, cuGraph — kernel-per-iteration model; RustGraph — persistent actor model with always-current analytics

D 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 and RustGraph project experience.

RingKernel: `https://github.com/mivertowski/ringkernel`

RustGraph: `https://github.com/mivertowski/rustgraph`