article

Zero-Copy CPU-GPU Pipeline Architecture for Unified Memory Systems: Design, Implementation, and Performance Analysis

Abstract This paper presents a novel software architecture for efficient data processing pipelines between heterogeneous processors (CPU and GPU) on unified memory platforms. I introduce two complementary synchronization paradigms: event-based orchestration (V1) and persistent kernel with lock-free job queues (V2). Our approach eliminates explicit memory copies between host and device through careful exploitation of cache-coherent unified memory, atomic operations with system-wide memory ordering semantics, and a deterministic finite state machine for buffer ownership management. The architecture specifically targets the NVIDIA GB10 Grace-Blackwell platform featuring NVLink-C2C interconnect with 900 GB/s bidirectional bandwidth. I present detailed algorithmic specifications, comparative analysis of synchronization strategies, and architectural considerations for big.LITTLE ARM heterogeneous CPU configurations.

Keywords: Unified Memory, Zero-Copy, CPU-GPU Pipeline, Lock-Free Synchronization, Persistent Kernels, Heterogeneous Computing, Cache Coherence

Author: Emmanuel Forgues

---

1. Introduction

1.1 Motivation and Context

Modern heterogeneous computing systems combine general-purpose CPUs with massively parallel accelerators such as GPUs. Traditional programming models require explicit data transfers between host (CPU) and device (GPU) memory spaces, introducing significant latency and bandwidth overhead. The emergence of cache-coherent unified memory architectures, exemplified by the NVIDIA Grace-Blackwell GB10 System-on-Chip, enables a paradigm shift toward zero-copy data sharing between processors. The GB10 platform integrates a 20-core ARM Grace CPU (heterogeneous big.LITTLE configuration) with a Blackwell GPU, connected via NVLink-C2C providing 900 GB/s bidirectional bandwidth to a shared 128GB LPDDR5X memory pool. This architectural advancement necessitates new software design patterns that fully exploit cache coherence while maintaining correct memory ordering semantics.

1.2 Problem Statement

Efficient CPU-GPU collaboration requires solving three fundamental challenges:

1. **Ownership Management**: At any instant, a shared memory region must have a clearly defined owner (CPU or GPU) to prevent data races.
2. **Synchronization Overhead**: Traditional synchronization mechanisms (CUDA events, stream synchronization) introduce microsecond-scale latencies inappropriate for fine-grained pipelines.
3. **Cache Coherence Exploitation**: Unified memory systems require explicit memory barriers to ensure visibility of writes across processor boundaries.

1.3 Contributions

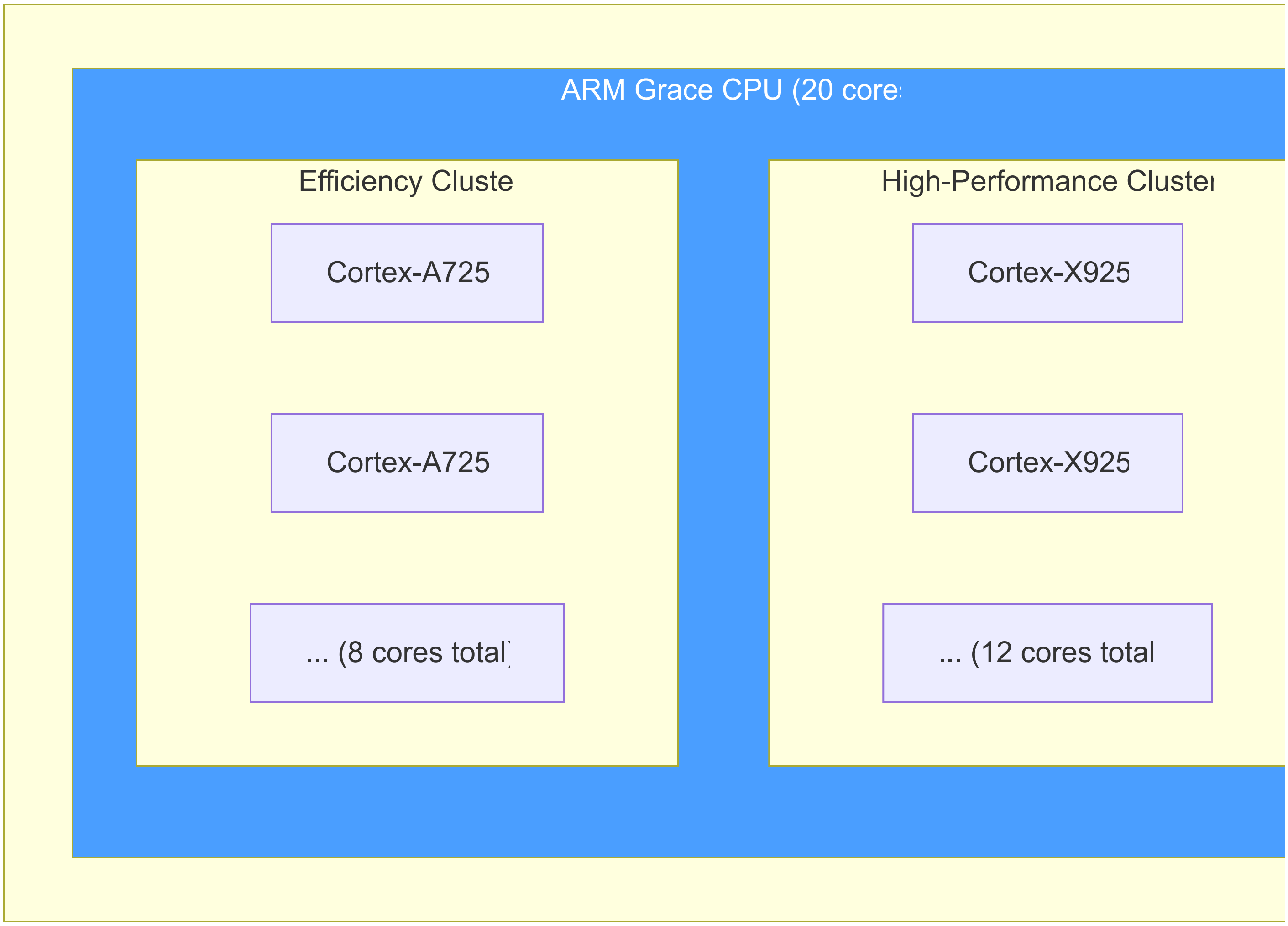
This work introduces:

- A five-state finite automaton for deterministic buffer ownership transitions
- Two synchronization policies: host-orchestrated (EventsSync) and device-autonomous (AtomicsSync)
- A persistent kernel architecture with lock-free SPMC (Single-Producer Multi-Consumer) job queue
- Memory layout optimizations eliminating false sharing between metadata and payload
- Algorithmic specifications for ARM big.LITTLE task scheduling

---

2. System Architecture Overview

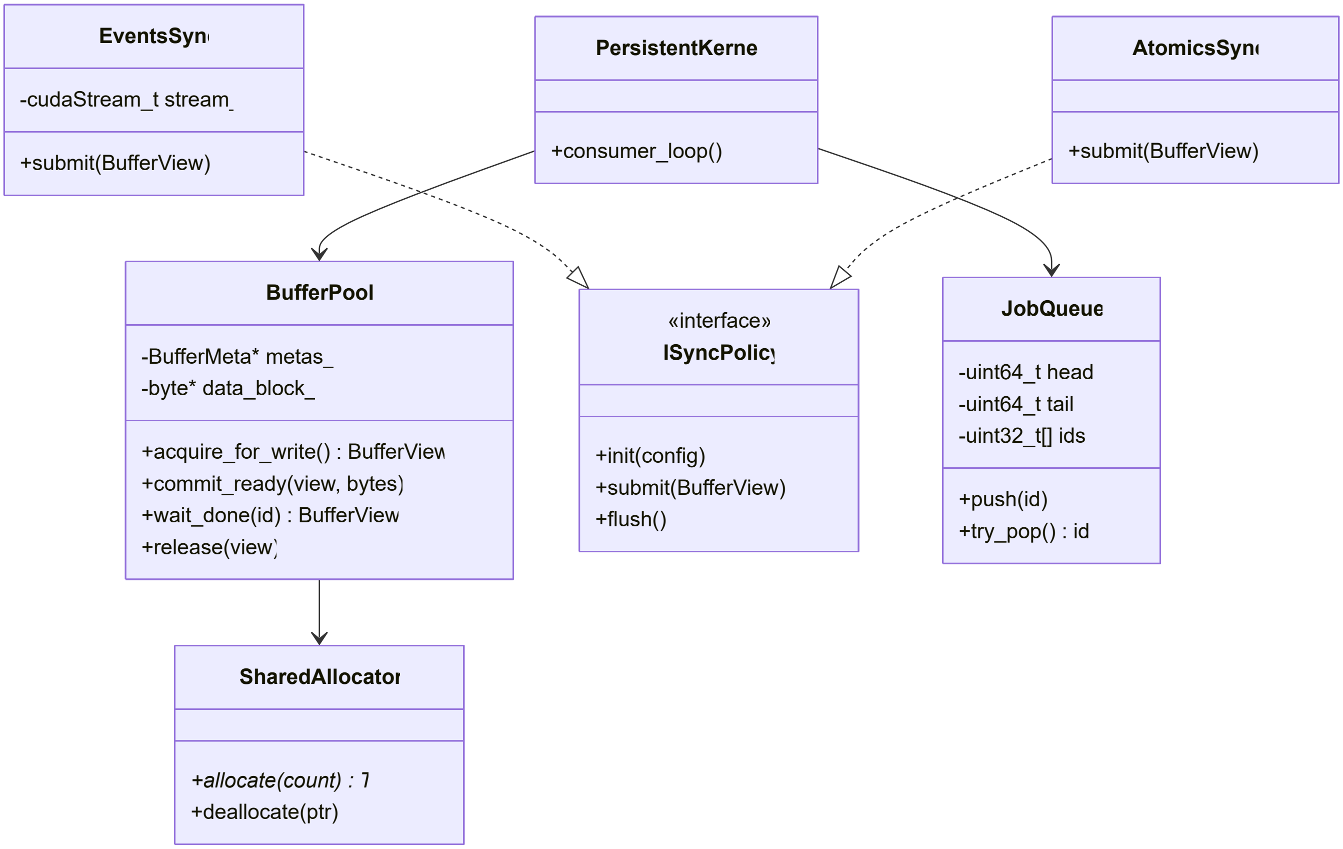
2.1 Target Platform: NVIDIA GB10 Grace-Blackwell



2.2 High-Level Component Architecture

The library comprises five principal abstractions:

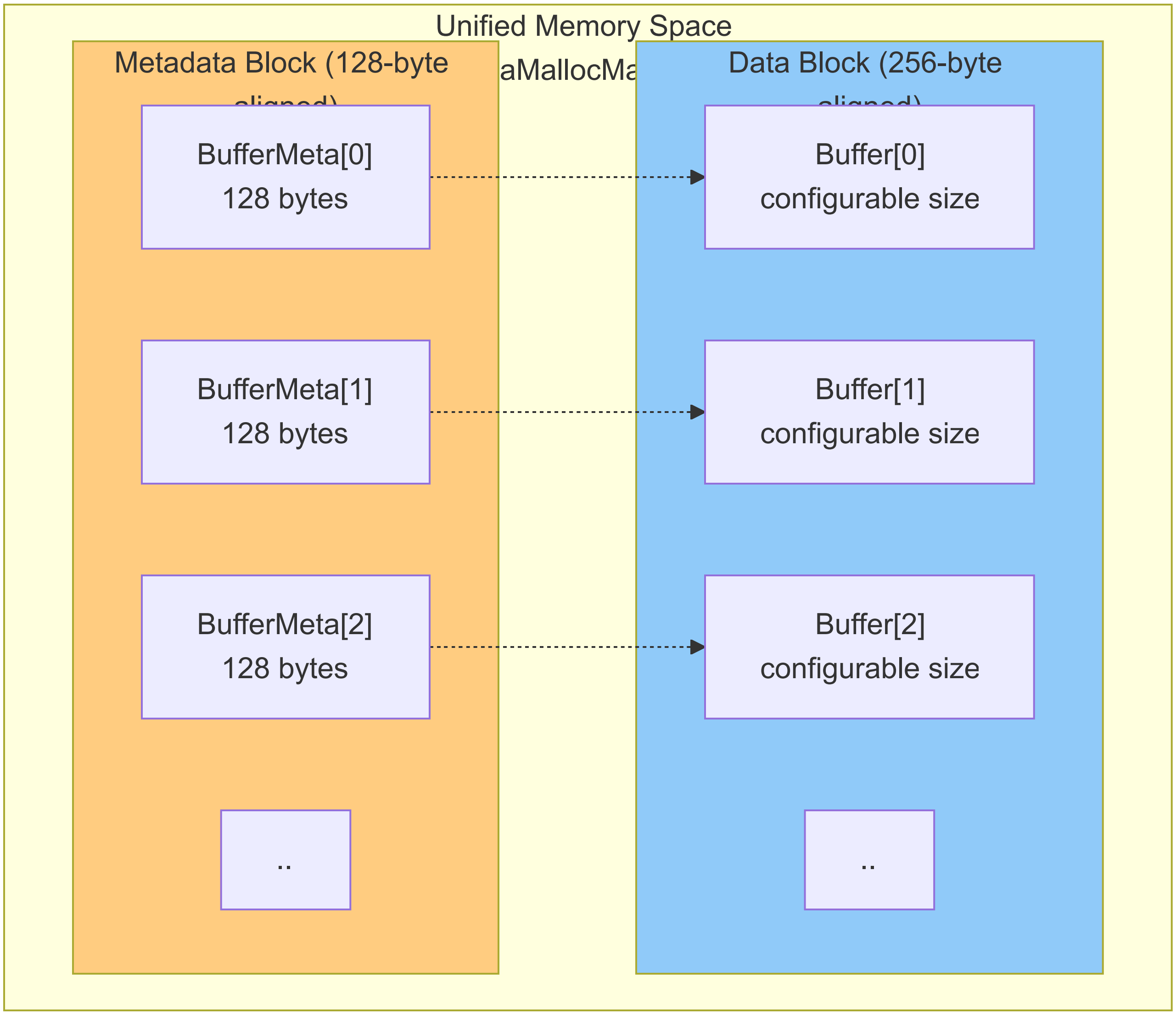
Component	Responsibility
SharedAllocator	Unified memory allocation (cudaMallocManaged)
BufferPool	Pre-allocated buffer management with state tracking
SyncPolicy	Pluggable synchronization strategy interface
JobQueue	Lock-free ring buffer for persistent kernel communication
Pipeline	Unified API orchestrating all components



3. Buffer Management and State Machine

3.1 Memory Layout Design

A critical design decision separates metadata from payload to eliminate false sharing. Cache line invalidations caused by metadata updates (state transitions, sequence counters) must not affect data cachelines being read or written.



Alignment Rationale:

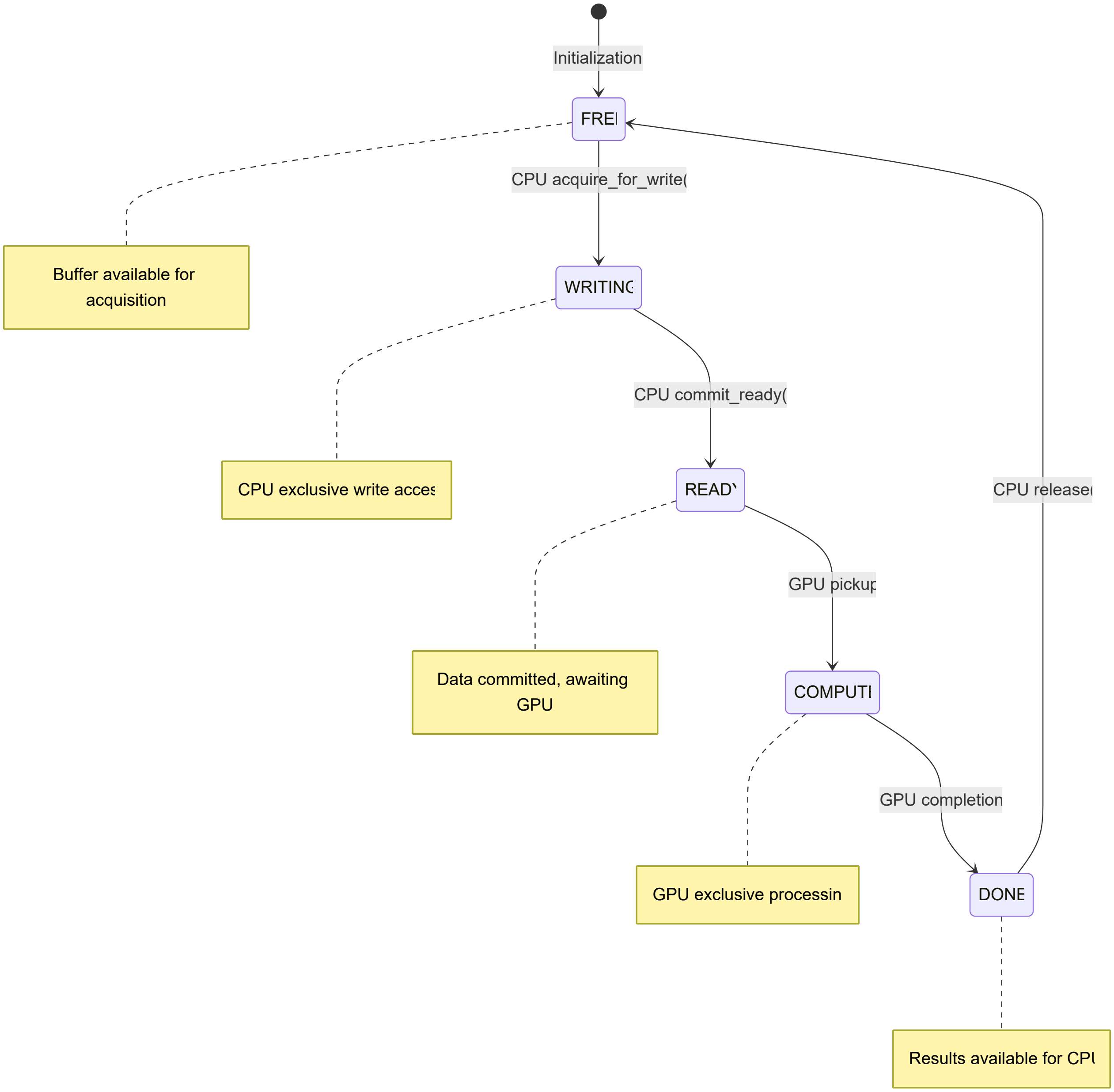
- 128-byte metadata alignment matches GPU L2 cache sector size
- 256-byte data alignment ensures coalesced GPU memory access
- Separation guarantees that `state` and `seq_*` fields reside in distinct cachelines from payload

3.2 Buffer Metadata Structure

Each buffer carries associated metadata for synchronization: | Field | Type | Purpose | |-------|------|---------| | `state` | atomic<uint32_t> | Current state in the finite automaton | | `seq_ready` | atomic<uint64_t> | Monotonic counter incremented when CPU commits | | `seq_done` | atomic<uint64_t> | Monotonic counter incremented when GPU completes | | `bytes_valid` | atomic<uint32_t> | Actual payload size for current job |

3.3 Five-State Finite Automaton

Buffer ownership transitions follow a deterministic finite state machine:

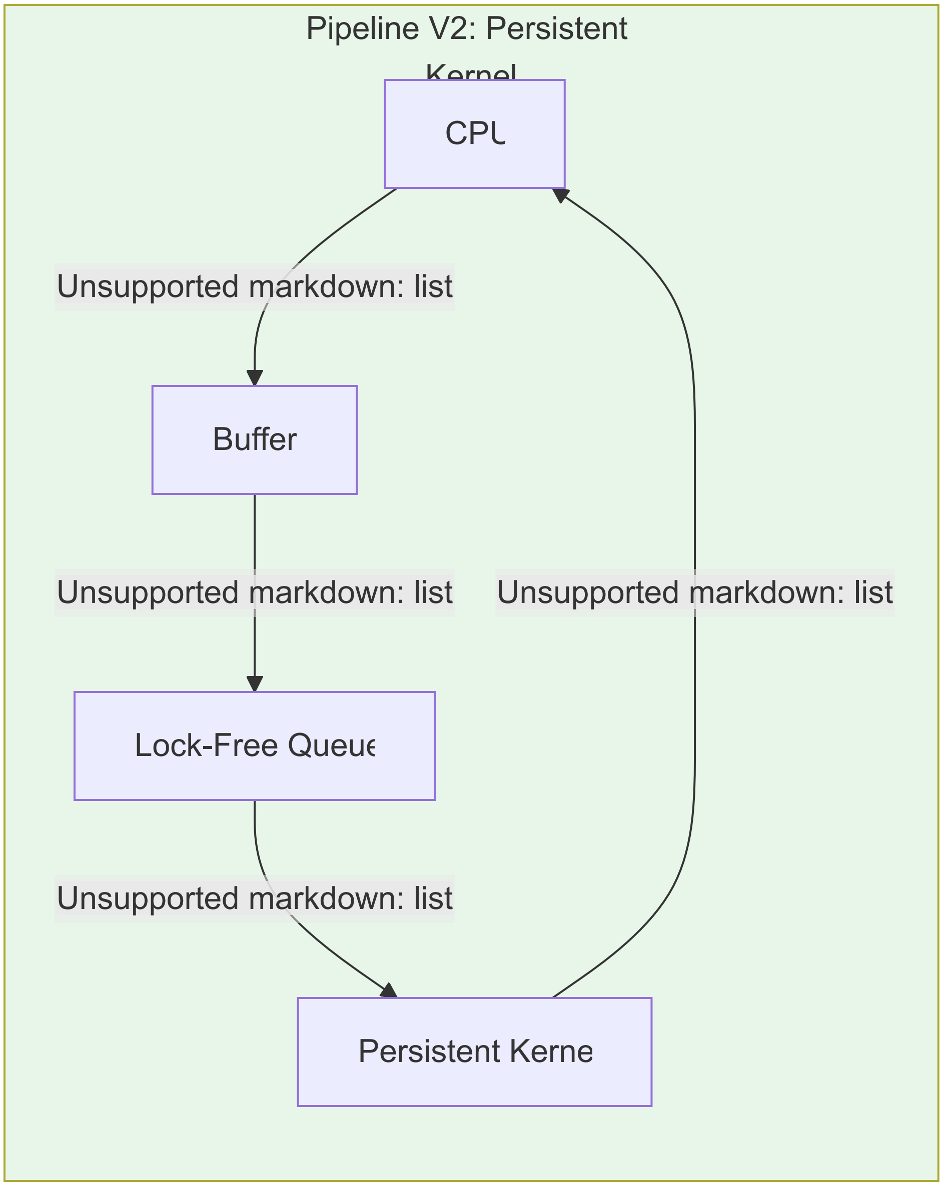


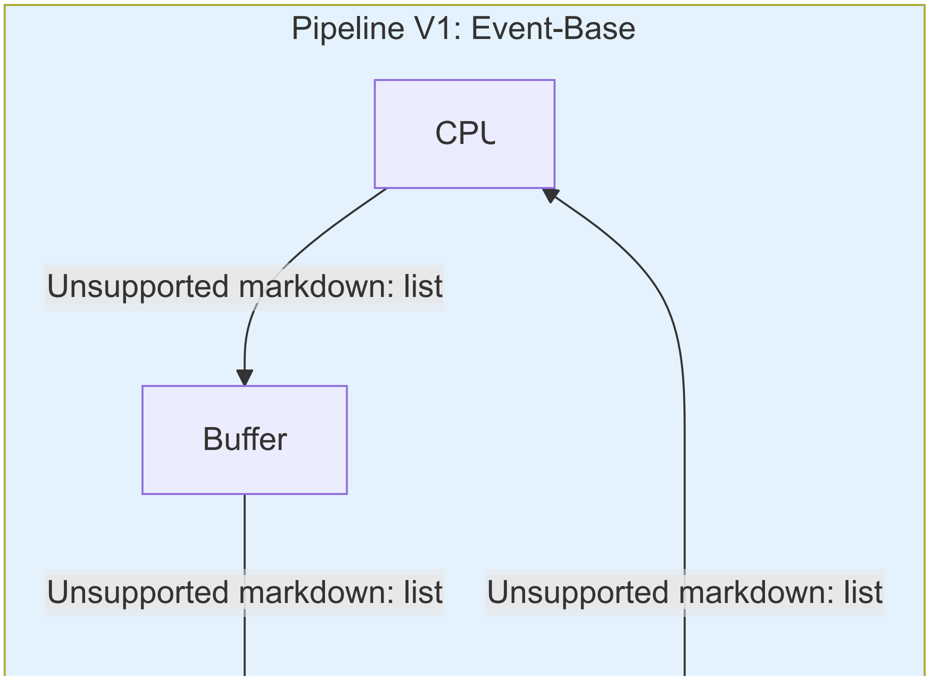
State Transition Rules: | Transition | Actor | Precondition | Memory Ordering | |------------|-------|--------------|-----------------| | FREE -> WRITING | CPU | CAS(state, FREE|DONE, WRITING) | acquire-release | | WRITING -> READY | CPU | After data write completion | release | | READY -> COMPUTE | GPU | seq_ready observed | acquire | | COMPUTE -> DONE | GPU | After kernel completion | release | | DONE -> FREE | CPU | After result consumption | release |

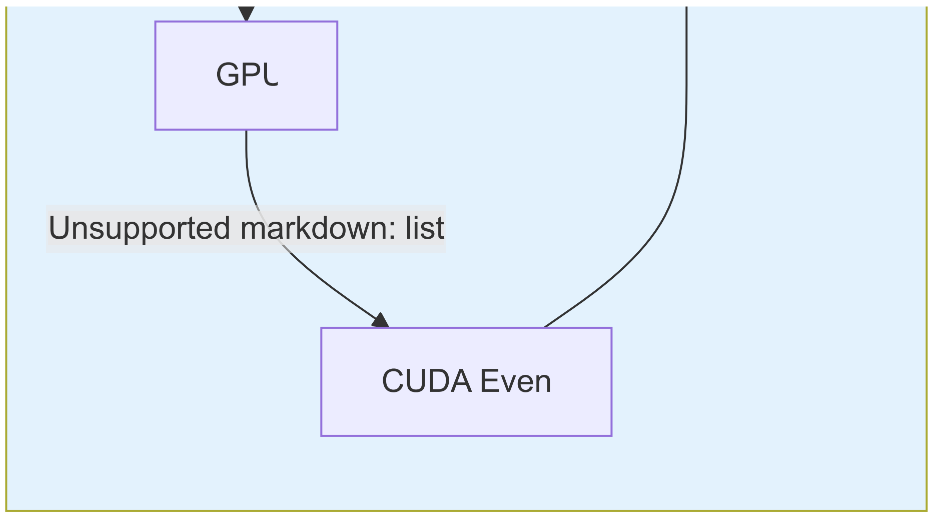
4. Synchronization Strategies

4.1 Strategy Comparison

We implement two distinct synchronization paradigms with complementary characteristics:


Pipeline V2: Persistent Kernel
CPU
Unsupported markdown: list
Buffer
Unsupported markdown: list
Unsupported markdown: list
Lock-Free Queue
Unsupported markdown: list
Persistent Kerne

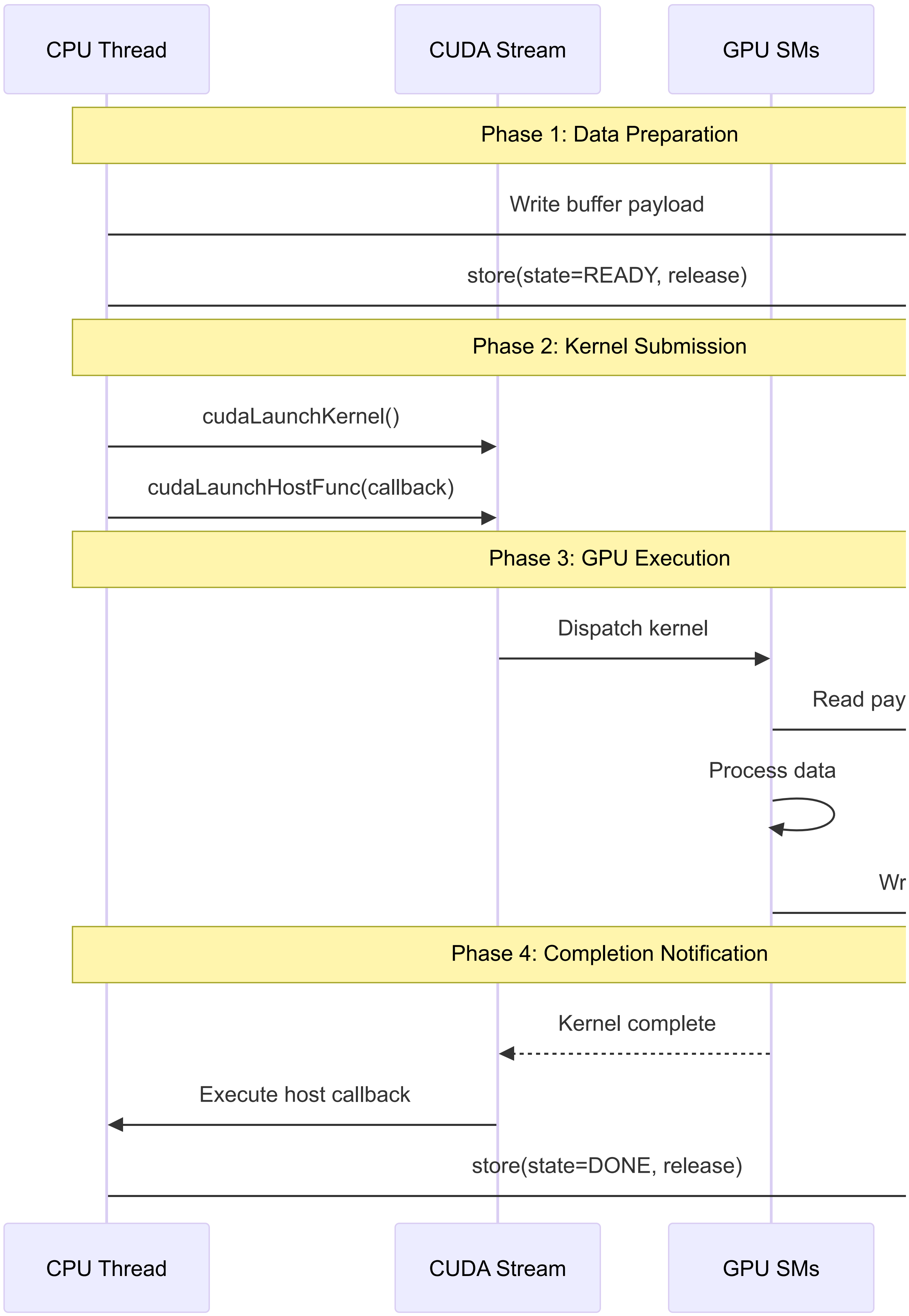

Pipeline V1: Event-Base
CPU
Unsupported markdown: list
Buffer
Unsupported markdown: list
Unsupported markdown: list



Characteristic	EventsSync (V1)	AtomicsSync/Persistent (V2)
Kernel launch overhead	Per-job (3-10 us)	Once at startup
Synchronization mechanism	CUDA Events + Host callbacks	Atomic flags + polling
CPU blocking	Yes (event wait)	Optional (polling)
Minimum latency	~5-10 microseconds	~0.5-2 microseconds
Implementation complexity	Low	High
Debugging difficulty	Low	High
Resource utilization	Variable SM occupancy	Dedicated SM reservation

4.2 Event-Based Synchronization (V1)

The event-based strategy uses CUDA's native synchronization primitives:

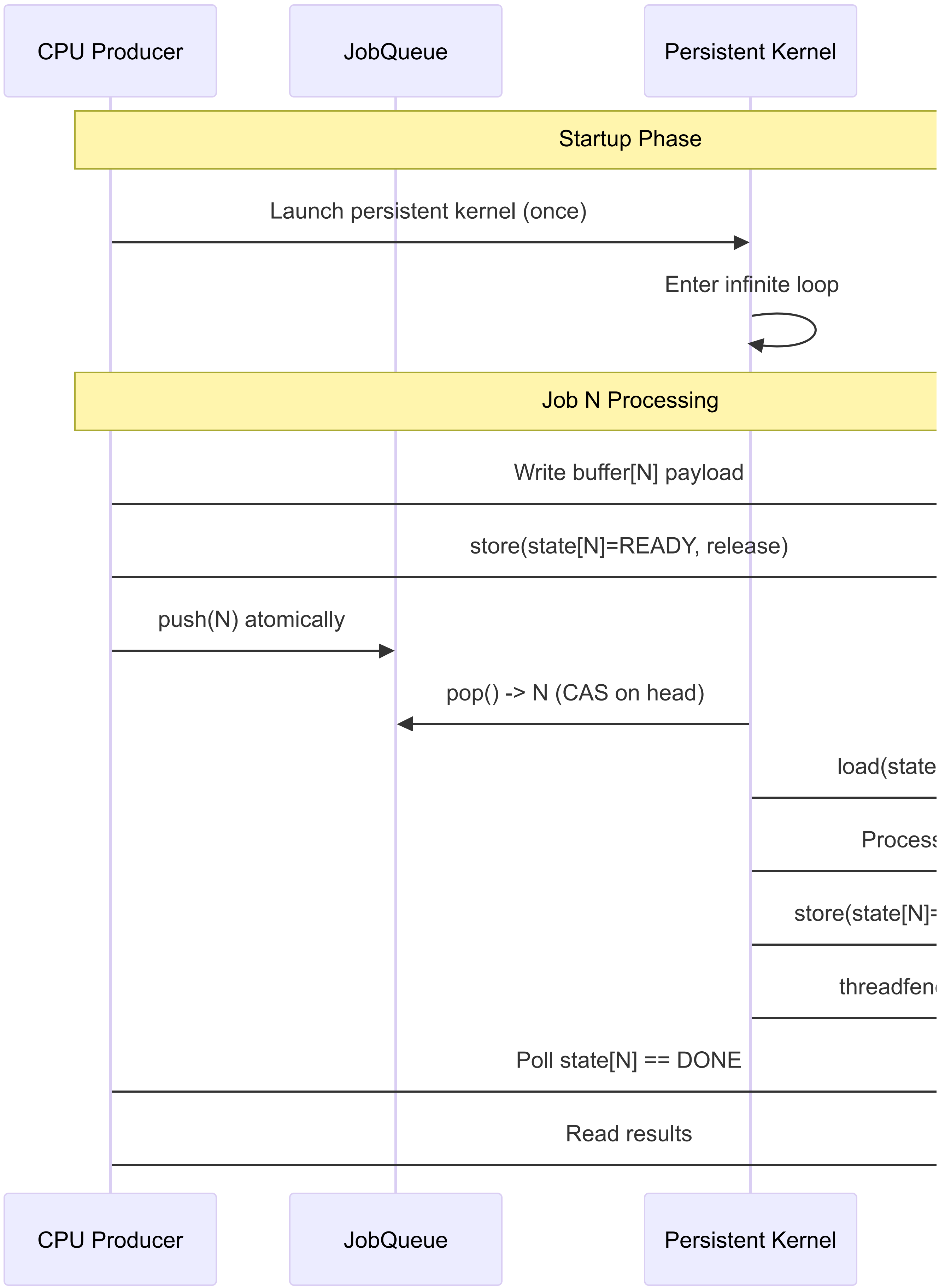

CPU Thread
CUDA Stream
GPU SMs
Phase 1: Data Preparation
Write buffer payload
store(state=READY, release)
Phase 2: Kernel Submission
cudaLaunchKernel()
cudaLaunchHostFunc(callback)
Phase 3: GPU Execution
Dispatch kernel
Read pay
Process data
Wr
Phase 4: Completion Notification
Kernel complete
Execute host callback
store(state=DONE, release)
CPU Thread
CUDA Stream
GPU SMs

Algorithm: EventsSync Submit

```
PROCEDURE EventsSync.submit(buffer)
INPUT: BufferView buffer with state=READY
OUTPUT: Asynchronous GPU processing initiated
1. Store state <- COMPUTE with release semantics
2. Read bytes_valid from buffer.meta
3. Calculate grid dimensions: blocks <- ceil(bytes_valid / threads_per_block)
4. Launch kernel on stream with (grid, block, stream) parameters
5. Enqueue host callback on stream:
   a. On callback execution:
      - Increment seq_done with release semantics
      - Store state <- DONE with release semantics
6. Return (non-blocking)
```

4.3 Atomic-Based Synchronization with Persistent Kernel (V2)

The persistent kernel strategy eliminates per-job launch overhead:



Algorithm: Lock-Free Queue Push (CPU Side)

```
PROCEDURE JobQueue.push(buffer_id)
INPUT: buffer_id - identifier of ready buffer
OUTPUT: buffer_id enqueued for GPU consumption
1. LOOP forever:
   a. t <- atomic_load(tail, relaxed)
   b. h <- atomic_load(head, acquire)
   c. IF t - h < capacity THEN:
      - ids[t mod capacity] <- buffer_id
      - atomic_store(tail, t + 1, release)
      - RETURN success
   d. ELSE:
      - Yield CPU (backpressure handling)
      - Continue loop
```

Algorithm: Lock-Free Queue Pop (GPU Side)

```
PROCEDURE JobQueue.try_pop()
OUTPUT: (success, buffer_id) or (failure, _)
1. LOOP forever:
   a. h <- atomic_load(head) via atomicAdd(&head, 0)
   b. Execute threadfence_system()
   c. t <- atomic_load(tail) via atomicAdd(&tail, 0)
   d. Execute threadfence_system()
   e. IF h >= t THEN:
      - RETURN (false, _)  // Queue empty
   f. old <- atomicCAS(&head, h, h + 1)
   g. IF old == h THEN:
      - buffer_id <- ids[h mod capacity]
      - RETURN (true, buffer_id)
   h. ELSE:
      - Continue loop  // Contention retry
```

Algorithm: Persistent Kernel Main Loop

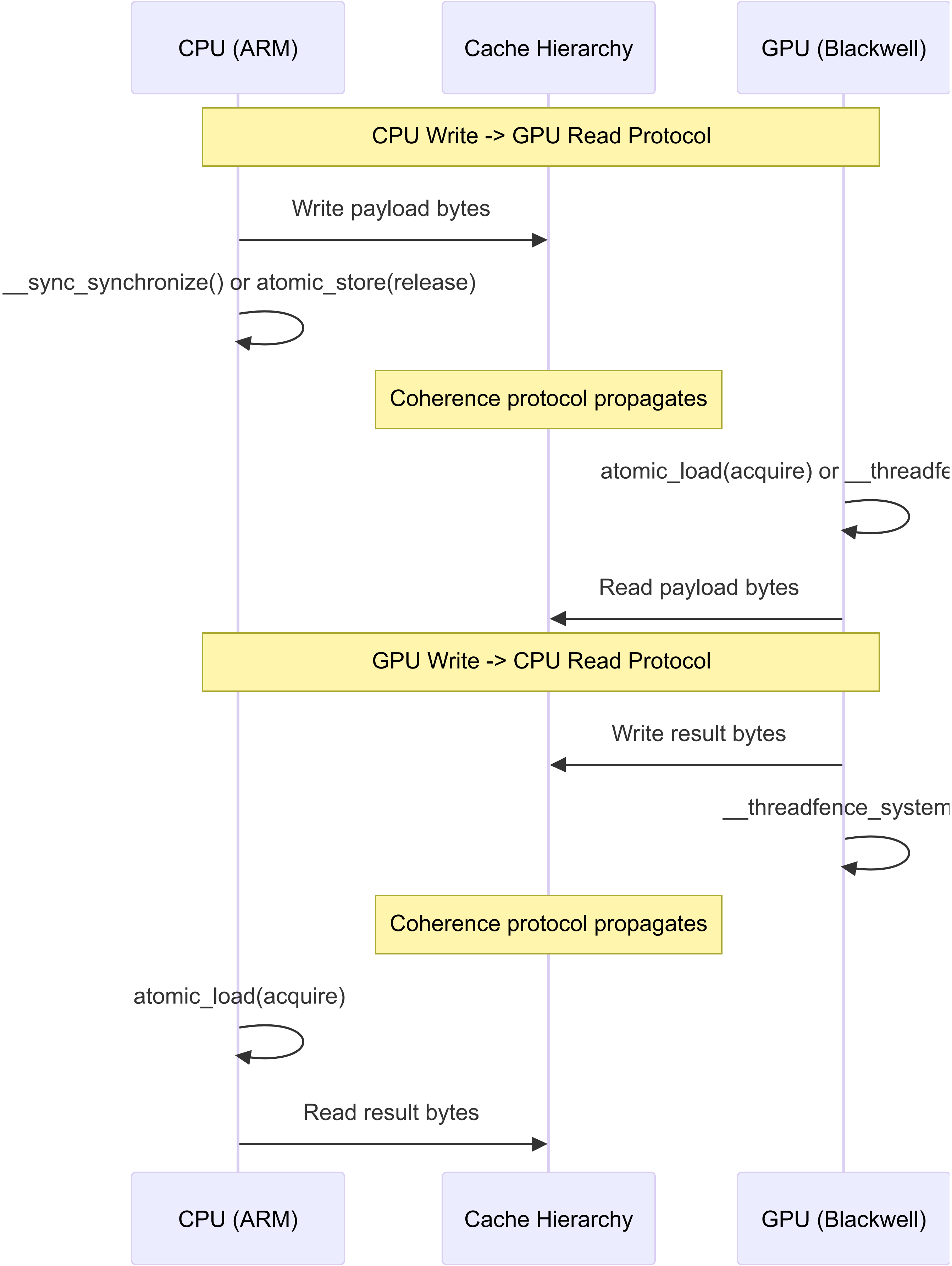
```
PROCEDURE PersistentKernel.consumer_loop(queue, metas, data_block)
INPUT: Shared queue, metadata array, data block pointer
EXECUTION: Runs until termination signal
1. backoff_iteration <- 0
2. LOOP forever:
   a. Execute threadfence_system()
   b. IF atomic_load(queue.stop) != 0 THEN:
      - BREAK
   c. IF thread_index == 0 THEN:
      - (ok, id) <- queue.try_pop()
```

```
       - Store to shared memory: sh_ok, sh_id
    d. Execute syncthreads()
    e. IF sh_ok AND sh_id == POISON_PILL THEN:
       - BREAK  // Graceful termination
    f. IF NOT sh_ok THEN:
       - Execute adaptive_backoff(backoff_iteration)
       - backoff_iteration <- backoff_iteration + 1
       - CONTINUE
    g. backoff_iteration <- 0
    h. buffer <- get_buffer(sh_id, metas, data_block)
    i. atomic_exchange(buffer.meta.state, COMPUTE)
    j. Execute parallel_process(buffer)
    k. Execute syncthreads()
    l. IF thread_index == 0 THEN:
       - atomic_exchange(buffer.meta.state, DONE)
       - Execute threadfence_system()
```

5. Memory Ordering and Coherence Protocol

5.1 Memory Barrier Requirements

Cache-coherent unified memory does not eliminate the need for explicit memory ordering. The following barriers ensure correct visibility:


CPU (ARM)
Cache Hierarchy
GPU (Blackwell)
CPU Write -> GPU Read Protocol
Write payload bytes
__sync_synchronize() or atomic_store(release)
Coherence protocol propagates
atomic_load(acquire) or __threadfe
Read payload bytes
GPU Write -> CPU Read Protocol
Write result bytes
__threadfence_system
Coherence protocol propagates
atomic_load(acquire)
Read result bytes
CPU (ARM)
Cache Hierarchy
GPU (Blackwell)

5.2 Memory Ordering Semantics

Operation	CPU (C++11/20)	GPU (CUDA)	Purpose
Producer commit	`store(seq_ready, release)`	N/A	Ensure payload visible before signaling
Consumer acquire	N/A	`atomicAdd(&seq, 0)` + `__threadfence_system()`	Ensure signal observed before reading
Consumer completion	N/A	`atomicExch(state, DONE)` + `__threadfence_system()`	Ensure results visible before signaling
Result read	`load(state, acquire)`	N/A	Ensure signal observed before reading results

5.3 Adaptive Backoff Strategy

Polling-based synchronization requires careful backoff to balance latency against power consumption: **Algorithm: Adaptive Backoff (GPU)**

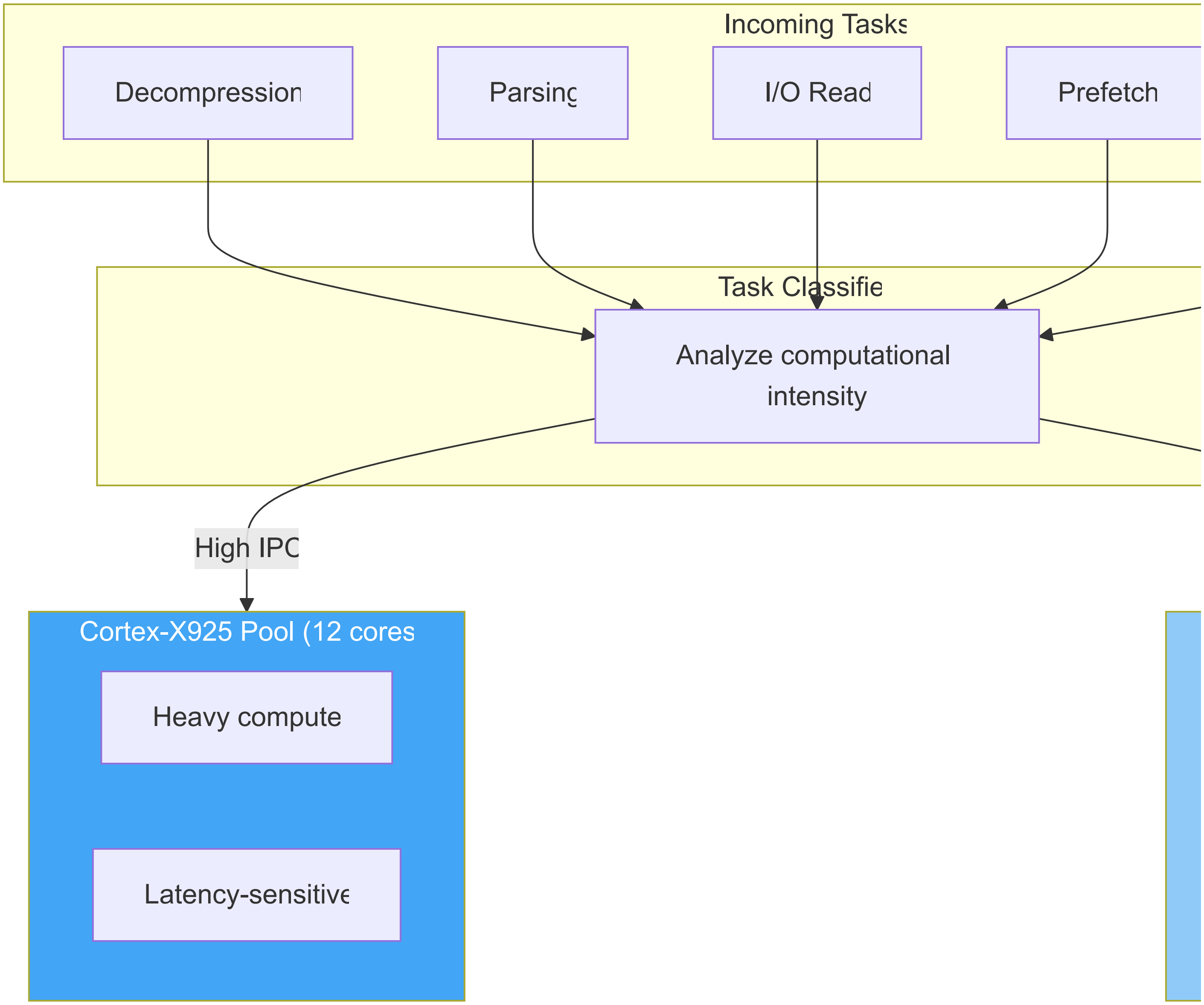
```
PROCEDURE adaptive_backoff(iteration)
INPUT: iteration counter
EFFECT: Delays execution to reduce power consumption
1. base_delay <- 100 nanoseconds
2. variable_delay <- (iteration mod 64) * 50 nanoseconds
3. total_delay <- base_delay + variable_delay
4. IF GPU_ARCHITECTURE >= SM_70 THEN:
   - Execute __nanosleep(total_delay)
5. ELSE:
   - Execute volatile spin loop for equivalent cycles
```

---

6. ARM big.LITTLE Task Scheduling

6.1 Heterogeneous Core Utilization

The GB10's ARM Grace CPU employs a big.LITTLE architecture requiring workload-aware scheduling:



6.2 Task Classification Criteria

Task Type	Target Cluster	Rationale
Decompression (zstd, lz4)	Cortex-X925 (Big)	High IPC, branch-heavy
Tokenization/Parsing	Cortex-X925 (Big)	Complex control flow
Buffer preparation	Cortex-X925 (Big)	Latency-sensitive
File I/O	Cortex-A725 (Little)	I/O-bound, low compute
Memory prefetch	Cortex-A725 (Little)	Memory-bound
Monitoring/Logging	Cortex-A725 (Little)	Background, low priority

Algorithm: Task Scheduling

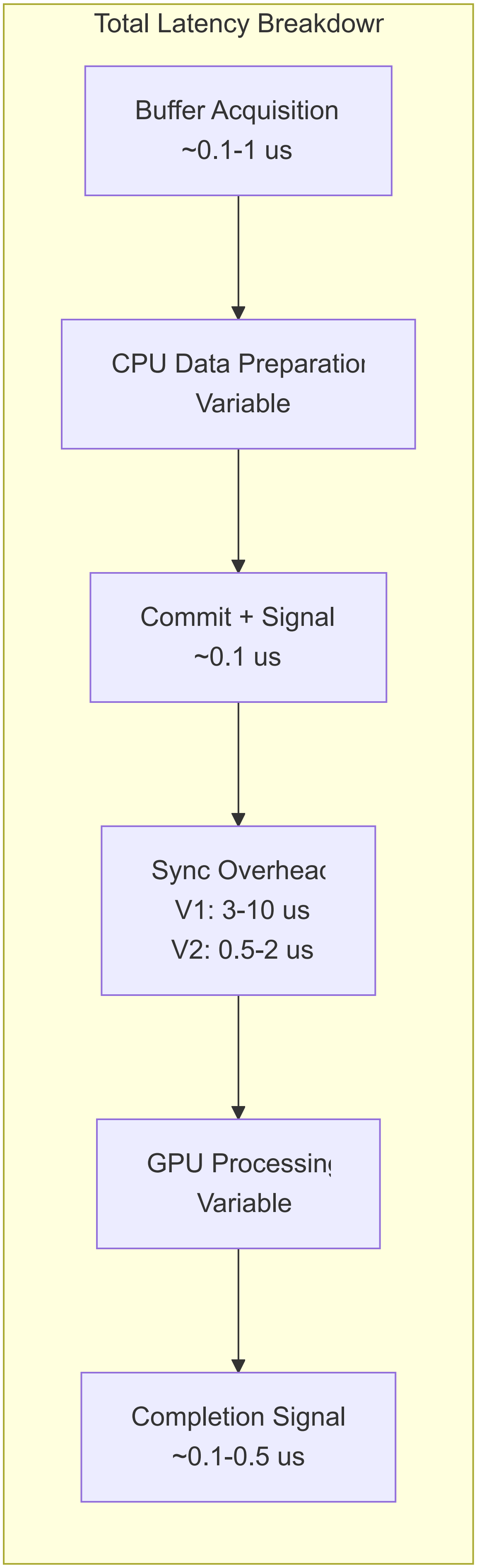
```
PROCEDURE schedule_task(task)
INPUT: Task with type and priority
OUTPUT: Task assigned to appropriate core pool
1. intensity <- analyze_computational_intensity(task)
2. is_latency_sensitive <- task.priority == HIGH
3. IF intensity > THRESHOLD_HIGH OR is_latency_sensitive THEN:
   - pool <- big_core_pool
4. ELSE:
   - pool <- little_core_pool
5. IF pool.available_cores == 0 THEN:
   - IF allow_work_stealing THEN:
      - pool <- alternate_pool
   - ELSE:
      - Wait for core availability
6. Assign task to pool with CPU affinity (pthread_setaffinity_np)
```

---

7. Performance Model and Analysis

7.1 Latency Components

The end-to-end latency for processing a single buffer comprises:


Total Latency Breakdowr
Buffer Acquisition ~0.1-1 us
CPU Data Preparatioı Variable
Commit + Signal ~0.1 us
Sync Overheac V1: 3-10 us V2: 0.5-2 us
GPU Processinç Variable
Completion Signal ~0.1-0.5 us

7.2 Throughput Model

For a pipeline with N buffers processing jobs of size S bytes: **Maximum Throughput (Event-Based V1)**:

$$T_{V1} = \frac{N \cdot S}{max(t_{CPU}, t_{GPU}) + t_{sync}}$$

Where:

- t_{CPU} = CPU preparation time per buffer
- t_{GPU} = GPU processing time per buffer
- t_{sync} = Synchronization overhead (3-10 us for events) **Maximum Throughput (Persistent V2)**:

  $$T_{V2} = \frac{N \cdot S}{max(t_{CPU}, t_{GPU}) + t_{atomic}}$$

  Where:
- t_{atomic} = Atomic polling overhead (0.5-2 us)

7.3 Optimal Buffer Count

The optimal number of buffers N *balances pipeline depth against memory consumption: $$N^* = \left\lceil \frac{max(t*{CPU}, t*{GPU})}{min(t*{CPU}, t*{GPU})} \right\rceil + 1$ $Forbalancedworkloadswhere\$t_{CPU} \approx t_{GPU}$, double-buffering (N=2) is optimal. For imbalanced workloads, additional buffers smooth throughput variability.

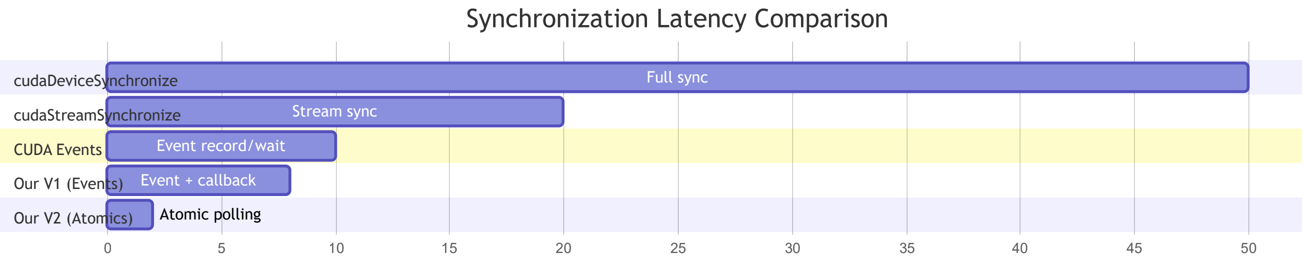
---

8. Comparative Analysis with Traditional Approaches

8.1 Memory Transfer Comparison

Approach	CPU->GPU Transfer	GPU->CPU Transfer	Total Overhead
Explicit cudaMemcpy	Full copy + sync	Full copy + sync	2x data size
Pinned Memory + DMA	DMA transfer	DMA transfer	PCIe latency
Unified Memory (migration)	Page faults	Page faults	Variable
Our Approach (zero-copy)	None	None	Barrier only

8.2 Synchronization Comparison



9. Implementation Considerations

9.1 Common Pitfalls

Pitfall	Symptom	Solution
False sharing on metadata	Performance degradation	128-byte alignment
Missing memory barriers	Corrupted data	Proper acquire/release semantics
Aggressive polling	CPU/GPU power waste	Adaptive backoff
Buffer size too small	Overhead domination	Increase buffer size
Buffer size too large	Cache thrashing	Tile-based processing
Insufficient buffers	Pipeline stalls	Increase N or implement backpressure

9.2 Graceful Shutdown Protocol

The persistent kernel requires a clean termination mechanism: **Algorithm: Pipeline Shutdown**

```
PROCEDURE shutdown_pipeline()
OUTPUT: All GPU threads terminated, resources released
1. FOR i <- 1 TO num_consumer_ctas DO:
   - queue.push(POISON_PILL)  // 0xFFFFFFFF
2. Wait for kernel completion (cudaStreamSynchronize)
3. Verify all buffers in FREE or DONE state
4. Deallocate unified memory resources
```

10. Conclusion and Future Work

10.1 Summary

I have presented a comprehensive software architecture for zero-copy CPU-GPU data pipelines on cache-coherent unified memory platforms. The dual-strategy approach (event-based V1 and persistent kernel V2) provides flexibility for different use cases:

- **V1 (EventsSync)**: Recommended for development, debugging, and workloads with variable kernel complexity
- **V2 (PersistentKernel)**: Recommended for production streaming workloads requiring minimal latency The architecture's key innovations include:
- Five-state ownership automaton ensuring race-free buffer access
- Physical separation of metadata and payload eliminating false sharing
- Lock-free SPMC queue enabling efficient multi-CTA consumption
- Adaptive backoff balancing latency against power consumption

10.2 Future Research Directions

1. **Multi-GPU Extension**: Extend the job queue to support multiple GPU consumers across NVLink fabrics
2. **Adaptive N-Buffering**: Runtime adjustment of buffer count based on workload characteristics
3. **CUDA Graph Integration**: Capture repetitive kernel sequences for reduced launch overhead in V1
4. **Quantum/Photonic Accelerator Generalization**: Abstract synchronization primitives for emerging accelerator paradigms
5. **Formal Verification**: Apply model checking to verify absence of deadlocks and data races

---

References

1. NVIDIA Corporation. "CUDA C++ Programming Guide." Version 12.x, 2024.
2. Harris, M. "Unified Memory for CUDA Beginners." NVIDIA Developer Blog, 2017.
3. Herlihy, M. and Shavit, N. "The Art of Multiprocessor Programming." Morgan Kaufmann, 2012.
4. ARM Limited. "ARM Cortex-X925 Technical Reference Manual." 2024.
5. NVIDIA Corporation. "NVIDIA Grace CPU Superchip Architecture." Technical Brief, 2024.
6. Boehm, H.J. and Adve, S.V. "Foundations of the C++ Concurrency Memory Model." PLDI 2008.

7. McKenney, P.E. "Memory Barriers: a Hardware View for Software Hackers." Linux Technology Center, 2010.

Appendix A: Complexity Analysis

A.1 Space Complexity

Component	Memory Footprint
BufferPool (N buffers, S bytes each)	$O(N \cdot S + N \cdot 128)$
JobQueue (capacity C)	$O(C \cdot 4 + 32)$
Per-kernel shared memory	$O(blockDim.x)$

A.2 Time Complexity

Operation	Best Case	Worst Case
Buffer acquisition	$O(1)$	$O(N)$ contention
Queue push	$O(1)$	$O(\infty)$ backpressure
Queue pop	$O(1)$	$O(C)$ contention
State transition	$O(1)$	$O(1)$

Appendix B: Synchronization Correctness Proof Sketch

Theorem: The five-state automaton with acquire-release semantics guarantees that:

1. No buffer is simultaneously written by CPU and read by GPU
2. All CPU writes are visible to GPU before processing begins
3. All GPU writes are visible to CPU before result consumption **Proof Sketch**: *Invariant 1 (Mutual Exclusion)*: The CAS operation on state transitions ensures atomic ownership changes. Only one actor can successfully transition from FREE/DONE to WRITING, and from READY to COMPUTE. *Invariant 2 (Write Visibility)*: The release semantics on `seq_ready` store and READY state store establish a happens-before relationship. The acquire semantics on GPU's observation of seq_ready ensures all prior CPU writes are visible. *Invariant 3 (Result Visibility)*: The `__threadfence_system()` followed by DONE

state store (release) ensures GPU writes complete before signaling. CPU's acquire load of state=DONE observes the DONE store only after GPU's fence, ensuring results are visible.

Manuscript prepared for submission to a peer-reviewed journal in high-performance computing.