# Chiplet Level Release/Acquire Fence on the AMD MI300X

David Yue, Leo Sciortino

## 1. Introduction

The AMD MI300X packages eight Accelerator Complex Dies (XCDs), each with a private L2 cache, shared memory side last level cache (LLC), and high bandwidth memory (HBM). All eight XCDs are connected by the AMD Infinity Fabric interconnect. To communicate between wavefronts, work groups or compute units (CUs), the MI300X instruction set supports scoped memory fences at different levels of the compute hierarchy. For instance, `__threadfence_block()` can be used to order memory accesses between threads in a block, `__atomic_thread_fence()` orders memory accesses between threads in a workgroup and `__threadfence()` orders memory accesses between threads device-wide. What is missing from this list is the ability to order memory operations between threads in an XCD. Currently, to establish an order on memory operations between CUs in one chiplet, programmers must rely on LLC or HBM probes. Since data must traverse the infinity fabric, this is incredibly expensive compared to using L2 cache as a point of coherence – which is currently not supported. Furthermore, not all computations – especially large tensor reductions – can be done efficiently on a single XCD. In these cases the programmer must rely on inter-chiplet data movement through AMD's Infinity Fabric interconnect, use device-wide memory such as HBM, and global memory fences to facilitate the communication. However, inter-chiplet data movement through Infinity Fabric or HBM probes are equally expensive. To resolve this, we implement a release/acquire chiplet level fence – with L2 cache as the point of coherence – that allows threads in an XCD to establish a memory ordering and synchronize with each other. A message-passing test confirms that the fence enforces the intended ordering and that removing it produces the expected violations. On MI300X we measure a chiplet release that is 26% faster than `__builtin_amdgcn_fence(...,"agent")` in isolation, and an 18% reduction in latency under a single-wavefront fence storm relative to the device-fence baseline. Our code can be found at https://github.com/leosciortino/chiplet-fence.

## 2. Background

Traditional GPUs have compute units that are distributed across one logical compute die. Figure 1 shows a unified L2 and High Bandwidth Memory (HBM).

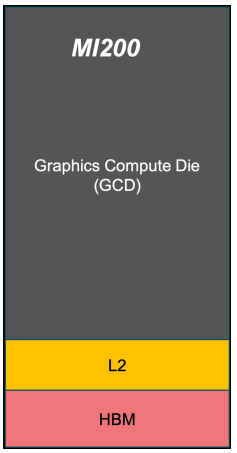


Figure 1: MI200 Die[1]

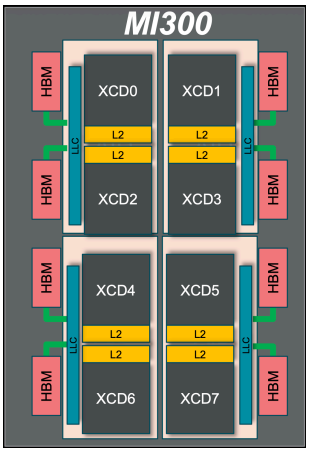


Figure 2: MI300 Die[2]

[1]This figure was taken from a presentation by Sangeeta Chowdhary
[2]This figure was taken from a presentation by Sangeeta Chowdhary

The MI300X L1 caches are split into a vector L1 cache (vL1) and scalar L1 cache (SL1). Whereas vL1 is a write-through cache, SL1 is a write-back cache. Any fence that hopes to support both scalar and vector data paths must drain both vL1 and sL1 into L2 and nowhere beyond that. Otherwise, we will have lost the benefits of a chiplet fence compared to a device-wide fence.

# 3. Chiplet Fence

In our implementation, the chiplet release pushes the issuing CU's stores into the shared L2 but not beyond. It compiles to three ISA instructions:

```
s_waitcnt vmcnt(0) ; drain vector stores to L2
s_dcache_wb ; write back dirty scalar L1 lines
s_waitcnt lgkmcnt(0) ; wait for the write-back
```

Each instruction is needed. Without `s_waitcnt vmcnt(0)` a vector store may still be in flight when the flag is published; without `s_dcache_wb`, a scalar store of the data stays in S_L1 and never reaches L2; and without the second `s_waitcnt`, the write-back itself is in flight when the next instruction executes. On the consumer side, the acquire invalidates the local L1 caches so that subsequent loads miss in L1 and are served by the shared L2 [1]:

```
buffer_inv sc1 ; invalidate vL1
s_dcache_inv ; invalidate SL1
s_waitcnt vmcnt(0) lgkmcnt(0)
```

Both fences are exposed as `__forceinline__` wrappers using `asm volatile(... ::: "memory")`, where the `"memory"` clobber keeps the compiler from reordering memory operations across the fence.

# 4. Microbenchmarks

We implemented a series of micro-benchmarks to measure the performance of our chiplet fence implementation under different conditions and compare against other built-in memory fences.. All measurements were taken on a single AMD Instinct MI300X (gfx942, 304 CUs, 2100 MHz) using ROCm and `hipcc`. Cycle counts come from the GPU's `clock64()` counter. We used the median result over 20 individual samples.

## 4.1. Message-Passing Test

To check that the chiplet fence enforces the intended ordering, we run a message-passing litmus test. The kernel is launched with 16 thread blocks of 64 threads each. Under CDNA3′s round-robin scheduling, two blocks land on each XCD, and within each block only thread 0 participates while the remaining 63 threads exit immediately. The two surviving threads on each XCD race for a per-XCD role lock implemented as a relaxed-atomic CAS; the block that wins the lock becomes the producer for that XCD, and the block that loses becomes the consumer. This avoids any dependency on the precise block-to-XCD scheduling map, since role assignment is established at runtime by the lock race rather than by block index.

The producer writes its own XCD ID into a shared buffer, executes the chiplet release, and increments the per-XCD semaphore. The consumer waits on the same semaphore, executes the chiplet acquire, reads the buffer, and records both the observed value and its own XCD ID alongside a cycle-counter delta:

```
require blockDim = 64
if threadIdx ≠ 0: return // one representative per block
XCD ← current XCD id

if try_lock(role[XCD]): // producer
    buffer[XCD] ← XCD
    Fence.release()
    sem_release(sems[XCD])

else:                   // consumer
    sem_acquire(sems[XCD])
    t_start  ← cycle_counter()
    Fence.acquire()
    observed ← buffer[XCD]
    t_end    ← cycle_counter()

    result[XCD] ← (
        producer_XCD  = observed,
        consumer_XCD  = XCD,
        cycle_delta   = t_end − t_start
        )
```

Listing 3: Message Passing Litmus Test

The host launches the kernel and, after device synchronization, copies the per-XCD results back and checks two invariants on each XCD: that the consumer's recorded XCD ID matches its index in the result array (confirming both blocks landed on the same XCD, as required for an intra-XCD test), and that the value the consumer observed in the buffer matches its own XCD ID. The second condition is the message-passing contract: since the producer writes its own XCD ID into the buffer and the producer and consumer are on the same XCD, a passing iteration is one in which `observed = consumer_XCD`. Any mismatch indicates that the consumer drained the semaphore but missed the preceding data write. Across all eight XCDs, the test passes on every run.

All atomic operations in the protocol — the role lock's CAS, the semaphore's increment and decrement, and the relaxed loads in the spin loop — use relaxed ordering at agent scope. They contribute no memory ordering beyond the atomicity of the locations they operate on directly. The release-acquire synchronization between the producer's buffer write and the consumer's buffer read is therefore entirely the responsibility of the chiplet fence.

As a sanity check that the chiplet fence is actually contributing to memory ordering, we re-ran the kernel with both `chiplet::release()` and `chiplet::acquire()` removed. In this configuration the consumer drains the semaphore but reads a stale value from the buffer: without the release, the producer's store has not necessarily reached L2, and without the acquire, the consumer's vL1 may still hold the previous value. This configuration produces numerous violations, as expected. We also tested removing only the release and only the acquire; both partial configurations produced violations as well, confirming that both halves of the fence pair are load-bearing.

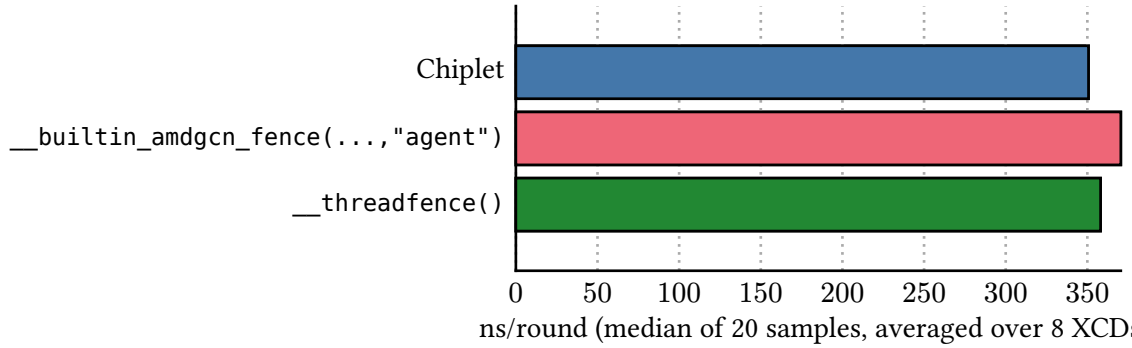


Figure 3: Producer-consumer message-passing latency, averaged across all 8 XCDs. The chiplet fence achieves the lowest average latency.

`__builtin_amdgcn_fence(..., "agent")` is HIP-Clang's compiler builtin for emitting an LLVM `fence` instruction at agent scope, where an agent is a single GPU device. An agent-scope release-acquire pair synchronizes all CUs on the GPU but does not extend to the host CPU or to peer GPUs. The release writes back dirty L2 cache lines so that they become visible to CUs on other XCDs, and the acquire invalidates non-local L2 cache lines so that subsequent loads refetch from the coherent point.

## 4.2. Fence-Only Latency

The fence-only benchmark isolates the raw per-call cost of each fence variant from any surrounding protocol overhead. A single thread issues `N` fences in a tight loop, with a `volatile` sink store between consecutive fences to prevent the compiler from eliding them; without it, consecutive fences with no intervening memory traffic can be collapsed by the optimizer. The benchmark is parameterized on the fence under test, and we report results for the chiplet release, the chiplet acquire, the chiplet release-acquire pair, and the corresponding agent-scope and device-wide variants.

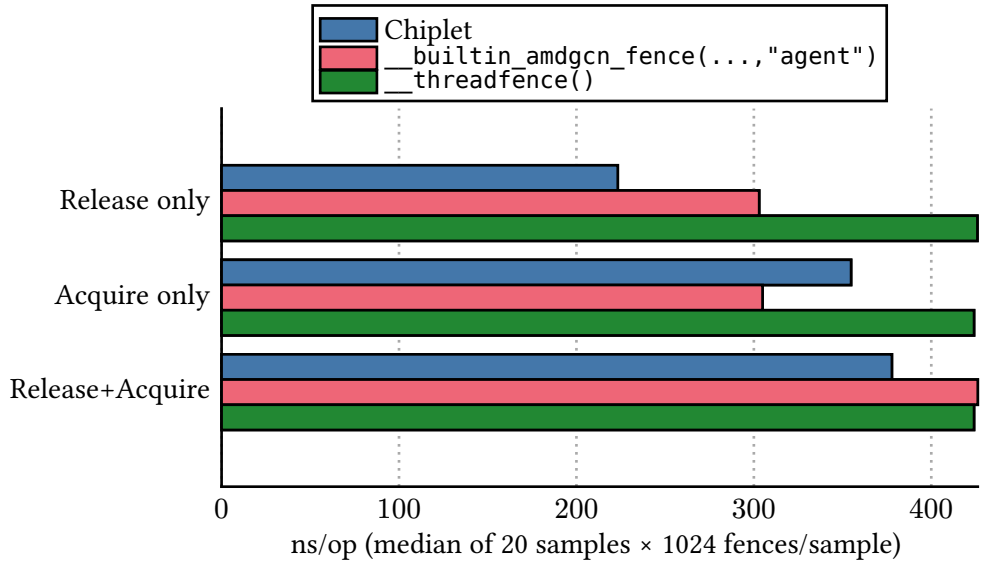


Figure 4: Fence-only latency: per-call cost of release, acquire, and the release-acquire pair, measured by issuing 1024 back-to-back fences from a single thread with a `volatile` sink store between fences to prevent compiler elision. The chiplet release is faster than both device-fence variants. The chiplet acquire is slower than the agent-scope acquire emitted by `__builtin_amdgcn_fence`. The release-acquire pair shows the chiplet sequence outperforming both device-fence baselines.

```
if threadIdx ≠ 0 or blockIdx ≠ 0: return
t_start ← cycle_counter()
for i in 0 .. N−1:
    *sink ← i              // volatile store, defeats fence elision
    Fence.do()             // release / acquire / pair / etc.
t_end ← cycle_counter()
cycles ← t_end − t_start
```

Listing 4: Fence-only microbenchmark, parameterized on the fence under test. A single thread (thread 0 of block 0) issues `N` fences in a tight loop, with a `volatile` sink store between them to prevent the compiler from eliding consecutive fences. Six variants are instantiated by substituting different bodies for `Fence.do()`: `chiplet_release()`, `chiplet_acquire()`, the chiplet release-acquire pair, `__threadfence()`, a `__threadfence()` pair, and a compiler-only barrier (`asm("" ::: "memory")`) as the no-fence baseline.

## 4.3. Ping Pong

The ping-pong benchmark involves two blocks running on the same XCD, each contributing one active thread. The kernel launches with 16 blocks of 64 threads; under round-robin scheduling, two blocks land on each XCD, and within every block only thread 0 participates while the rest exit immediately (`if (threadIdx.x != 0) return;`). To assign roles deterministically, the two surviving threads on each XCD race for a per-XCD role lock: the winner takes role 0 and the loser takes role 1. This handshake happens once, before the main loop, and avoids any dependency on the precise block-to-XCD scheduling map.

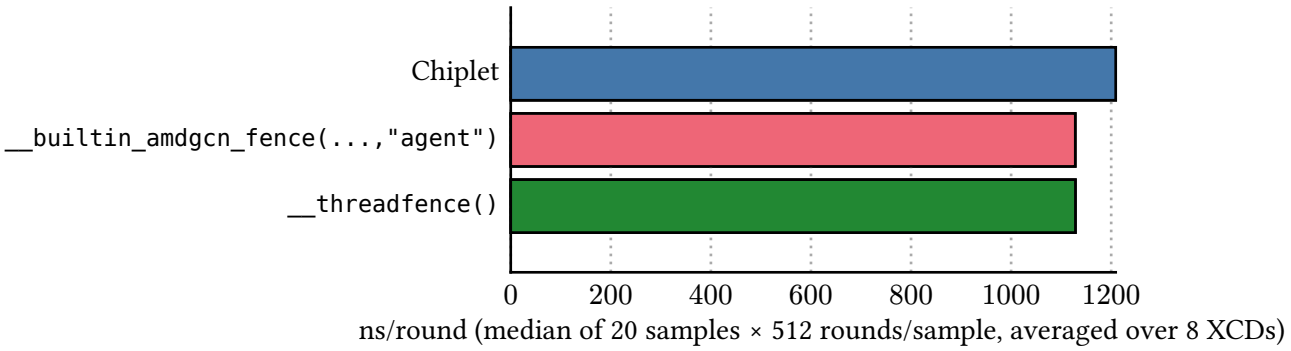


Figure 5: Ping-pong handshake latency, averaged across all 8 XCDs. The chiplet fence is approximately 7% slower than the device-scope fences in this high-frequency loop.

We then run the ping-pong for 1024 rounds, alternating producer and consumer by iteration parity: on even iterations role 0 produces and role 1 consumes, and on odd iterations the assignment reverses. On each round, the producer writes the current iteration index to a shared buffer, calls `Fence.release()` (which in this benchmark is `chiplet_release()`, propagating the write to the L2), and increments a per-XCD semaphore. The consumer spins on the same semaphore, and once the expected value is observed it calls `Fence.acquire()` (here `chiplet_acquire()`, which invalidates its stale cache lines), then reads the buffer from L2. Because the two threads live in different blocks they execute on different wavefronts, so the handshake is a real concurrency test rather than a sequential one masked by lockstep execution within a single wavefront. A single semaphore per XCD suffices because each release is consumed by the partner before either side advances, returning the count to zero between rounds. Each thread issues both `Fence.release()` and `Fence.acquire()` over the course of the loop, so the benchmark characterizes the fence's round-trip behavior rather than only one direction.

We run the same benchmark a second time with `Fence.release()` and `Fence.acquire()` instantiated as `__threadfence()` instead of the chiplet-scope variants, to compare the intra-XCD fence pair against the device-wide fence.

```
if threadIdx ≠ 0: return
XCD ← current XCD id
me  ← 0 if try_lock(roles[XCD]) else 1

for i in 0 .. N−1:
    if (i mod 2) = me:
        buffer[XCD] ← i
        Fence.release()
        sem_release(sems[XCD])

    else:
        sem_acquire(sems[XCD])
        t_start  ← cycle_counter()
        Fence.acquire()
        observed ← buffer[XCD]
        t_end    ← cycle_counter()

        results[XCD].expected      ← i
        results[XCD].observed      ← observed
        results[XCD].cycle_delta  += t_end − t_start
```

Listing 5: Ping Pong Microbenchmark

## 4.4. Fence Storm

The final experiment puts the fence under concurrent load. The benchmark is launched as a single workgroup of 64 threads (one wavefront), and every lane participates: each thread writes its own slot in a shared buffer, then issues a release/acquire fence pair, then waits at a `__syncthreads()` barrier so that every lane completes its fence pair before any lane begins the next iteration. The barrier is what makes this a "storm" — fences from 64 concurrent lanes, all forced into lockstep at every iteration — rather than a serialized stream of single-thread fences. Only thread 0 records the elapsed cycle count.

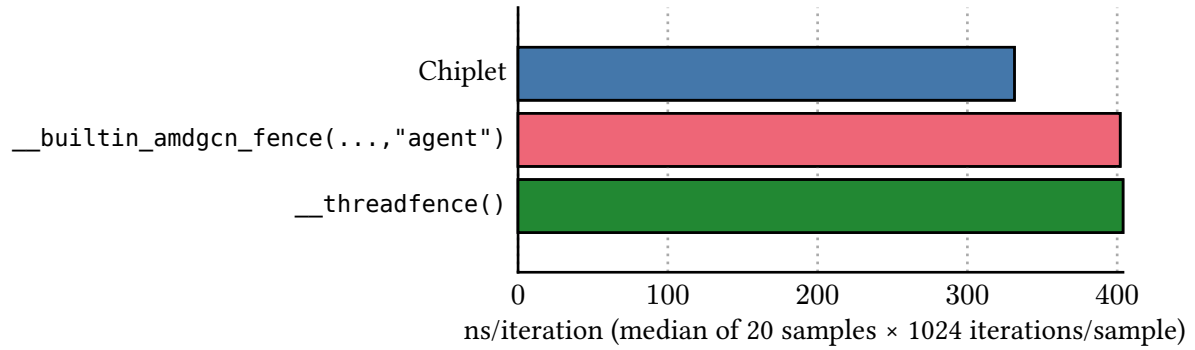


Figure 6: Fence storm: per-iteration latency under concurrent load, with all 64 lanes of one wavefront issuing a release-acquire pair every iteration and rendezvousing at a `__syncthreads()` barrier. The chiplet fence outperforms both device-scope variants by approximately 18%.

```
t_start ← cycle_counter()
for i in 0 .. N−1:
    buffer[threadIdx] ← threadIdx
    Fence.release()
    Fence.acquire()
    block_barrier()
t_end ← cycle_counter()

if threadIdx = 0:
    result.cycle_delta ← t_end − t_start
```

## 5. Conclusion

We presented a chiplet-level release/acquire fence for the AMD MI300X that uses the shared L2 cache as its point of coherence, filling the gap between workgroup-scoped and device-wide fences in HIP's existing scoped-fence hierarchy. The release compiles to a vector-store drain followed by a scalar L1 write-back, and the acquire to a pair of L1 invalidations, totalling three ISA instructions on each side. A message-passing litmus test confirmed that the fence enforces the intended ordering, and that removing either the release or the acquire produces exactly the violations predicted by the cache hierarchy: without the release, the producer's store has not yet drained to L2; without the acquire, the consumer's vL1 still holds the stale value. On performance, the chiplet fence achieves a 26% reduction in release latency over `__builtin_amdgcn_fence(...,"agent")` in isolation, an 18% reduction in latency under a single-wavefront fence storm, and a 5% reduction in producer-consumer message-passing latency relative to the device-fence baselines, by avoiding the LLC write-backs and HBM probes that the device-wide fence is forced to issue. Together, these results show that exposing L2 as a synchronization point on a chiplet GPU is both correct and cheaper than the device-fence baseline whenever inter-CU coordination can be confined to a single XCD, and they motivate the persistent-kernel and tensor-reduction extensions discussed next.

## 6. Future Work

There are several directions in which this work can be extended. A natural next step is to combine the chiplet fence with a persistent kernel runtime that pins workgroups to specific XCDs, allowing each chiplet to specialize in a distinct phase of a computation while using the chiplet fence as its primary synchronization primitive. The HipKittens [2] work reports that warp specialization underperforms on AMD GPUs, which raises the question of whether specialization at chiplet granularity fares better: an XCD is a much coarser unit than a warp, with its own L2 acting as a private staging area, so the producer/consumer imbalance and register pressure issues that hurt warp specialization may be far less pronounced when the specialized roles are assigned to entire chiplets. Building on this, we would like to apply the fence to efficient tensor reductions across the eight XCDs of the MI300X. Each chiplet would reduce its assigned portion of the tensor entirely within its own L2 cache, using the chiplet release and acquire to coordinate between CUs without escalating traffic to the LLC or HBM. Once every chiplet has completed its local reduction, the partial results would be forwarded over the Infinity Fabric interconnect to a single chiplet and combined there to produce the final value, paying the cost of inter-chiplet communication exactly once per reduction rather than at every synchronization point. Beyond reductions, we plan to evaluate the chiplet fence on larger and more representative workloads such as GEMM and split-K GEMM, where chiplet-local partial accumulation followed by a single inter-chiplet combine maps cleanly onto the same pattern and would allow us to quantify end-to-end speedups on kernels that real applications actually run.

## 7. Use of Generative AI

Anthropic's Claude was used during the preparation of this report. Its use spanned several categories. As an interactive technical reference, the model was consulted for AMD ROCm and CDNA3-specific information including HIP atomic builtins, AMDGPU memory model scopes, the syntax of `__builtin_amdgcn_fence`, and the architectural context for cross-XCD coherence; all such facts cited in the report were independently verified against the LLVM AMDGPU documentation, the AMD Instinct MI300 ISA Reference Guide, and disassembly of compiled kernels. As a methodology collaborator, the model contributed to the design of the benchmarking strategy, including the choice of microbenchmarks (fence-only latency, message-passing, ping-pong, and fence storm), the rationale for measuring each, and the experimental controls used to isolate fence cost from surrounding protocol overhead. We originally had it generate many of the benchmarks, but ended up re-imple-

menting them due to some hallucinations. As a code reviewer, the model identified concrete bugs in earlier drafts of the benchmark harness, including a producer-consumer placement issue under round-robin XCD scheduling, a self-handshake deadlock in the original single-semaphore ping-pong design, and several minor indexing and resource-cleanup errors. The chiplet fence implementation, the benchmark kernels, and the synchronization primitives (relaxed spinlock and semaphore) were written by the authors; the host-side harness around those kernels (sample-loop collection, median computation, warmup handling, and result reporting) was drafted by the model and reviewed by the authors. As a writing assistant, the model produced initial drafts of the pseudocode renderings of the benchmark kernels, the explanatory paragraphs accompanying them, and portions of the introduction and conclusion; these drafts were edited by the authors for accuracy against the implemented code and against the measured results. The authors have reviewed all generated content and accept full responsibility for its correctness.

# Bibliography


[1] Advanced Micro Devices, Inc., "AMD Instinct MI300: Instruction Set Architecture Reference Guide." 2024. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/amd-instinct-mi300-cdna3-instruction-set-architecture.pdf

[2] W. Hu *et al.*, "HipKittens: Fast and Furious AMD Kernels." [Online]. Available: https://arxiv.org/abs/2511.08083