



From KFD to DRM SVM: Toward a Unified Memory Model in DRM

Ray Huang / Honglei Huang / Junhua Shen

X.Org Developer's Conference 2026, Toronto, Canada

Team members



Honglei Huang

Kernel (KFD, AMDGPU, DRM) · ROCR



Junhua Shen

Kernel (AMDGPU) · Thunk · ROCR · IGT



Ray Huang

Kernel (KFD, AMDGPU, DRM)

Honglei - SVM core, locking rework, xnack-off restore Junhua - VRAM migration, drm_pagemap

Why Not Just Extend KFD SVM?

Functionally complete and in production - but structurally constrained

1. Synchronization built on the mmap lock

```
/* Remove from deferred_list must be inside
   mmap write lock, for two race cases:
   ... cause use after free bug. */
```

Driver behaviour is coupled to core MM - every change has to be re-reasoned against mmap lock ordering.

2. One MMU notifier per range

	KFD SVM	DRM GPUSVM
granularity	per range	512 MB
2 GB @ 2 MB	~1000	4

The more ranges a process has, the slower every CPU-side munmap gets.

3. Fixed per-GPU cost, fragile range splitting

```
#define MAX_GPU_INSTANCE 64
dma_addr_t *dma_addr[MAX_GPU_INSTANCE];
DECLARE_BITMAP(bitmap_access, ...);
DECLARE_BITMAP(bitmap_mapped, ...);
```

500+ bytes per range whatever the card count - and splitting spans seven interlocking functions.

4. Outside the DRM framework

```
/dev/kfd          -> KFD SVM   OK
/dev/dri/renderD* -> NO      (Vulkan / GL / user queues)
```

No shared code path with other drivers either - every vendor re-implements the same HMM plumbing.

KFD SVM is not a failed design - these are structural limits, not bugs. ROCm's hipMallocManaged runs on it today.

DRM SVM: One Abstraction for Every Driver

XDC 2025 introduced `drm_gpusvm` on Intel Xe - we evolve it and bring it to `amdgpu`

XDC 2025 - `drm_gpusvm` introduced on Intel Xe



This talk - unified DRM abstraction, demonstrated on `amdgpu`

Simplified synchronization

- No heavy `mmap` locking on the driver path
- One `rwsem` shared with the `amdgpu` VM core
- Notifier, eviction and PTE updates all serialize on it

Native large page support

- 2 MB / 64 KB / 4 KB chunk sizes built into the framework
- THP handled without invasive driver changes
- One 512 MB notifier covers many ranges

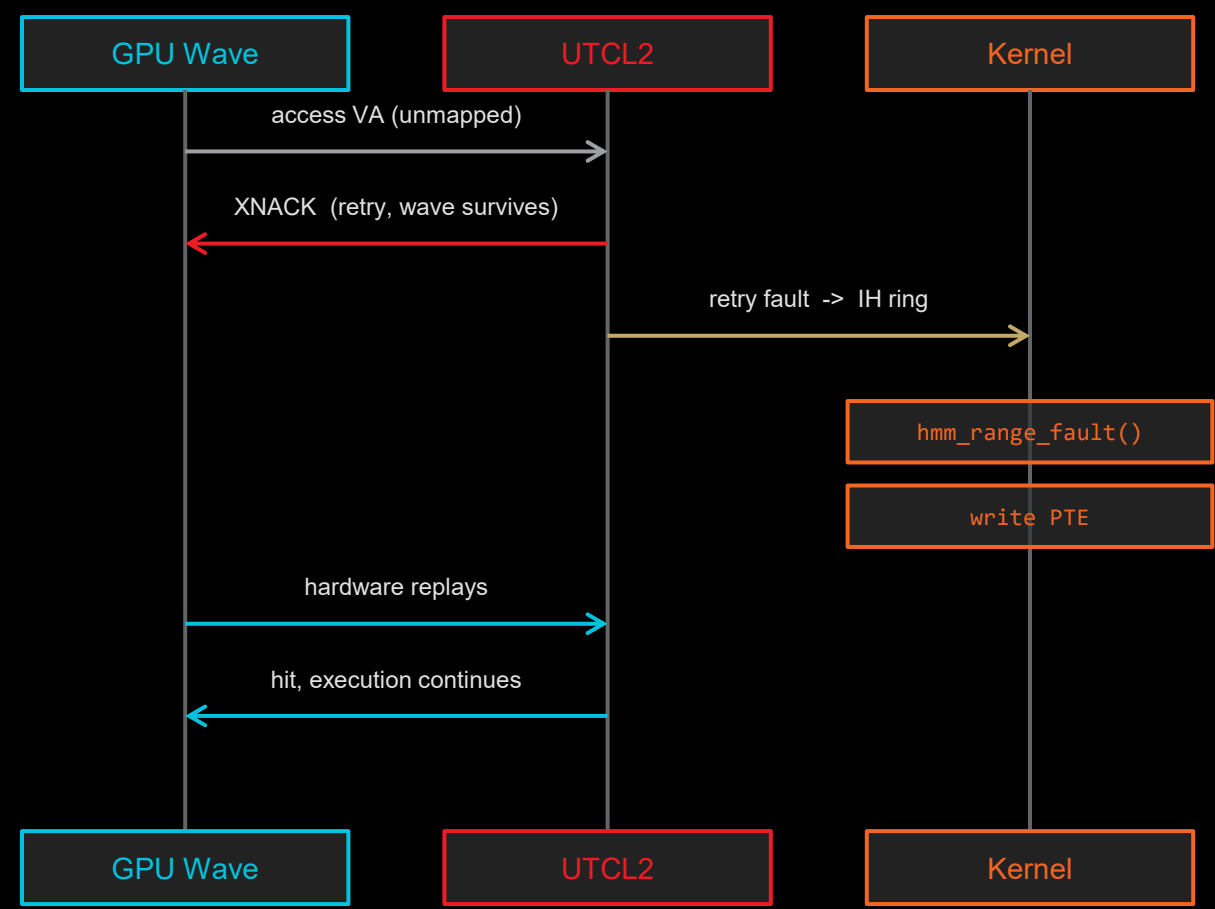
Cleaner attribute abstraction

- Policy lives in its own interval tree
- Location, granularity and access per range
- Change policy without touching mappings

Outcome	How
Every DRM client can use it	<code>DRM_IOCTL_AMDGPU_GEM_SVM</code> on the render node - no <code>/dev/kfd</code> dependency
Shared evolution with Intel Xe	Fixes and features flow both ways through <code>drm_gpusvm</code>
KFD SVM keeps working	An addition, not a replacement - the two coexist
Room to grow	Single-device today; multi-device support is already in flight

XNACK: The Hardware Basis

Without it, one page fault kills the wave



The wave never dies - it stalls, the kernel fills the PTE, the hardware replays the access.

Key point

With XNACK on, the wave is not killed - it just stalls and retries. That is the precondition for demand paging.

Both modes are now implemented

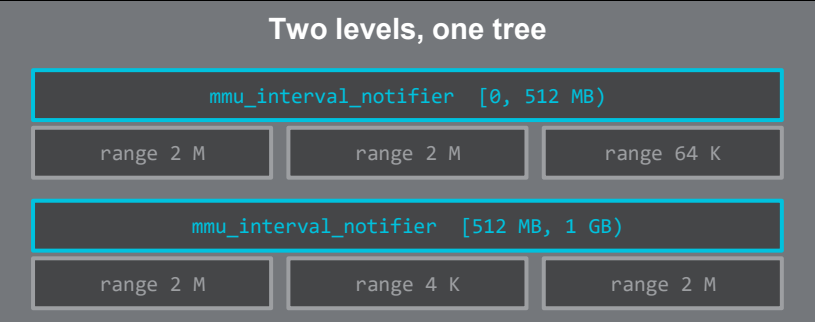
```
if (svm->xnack_enabled) {
    amdgpu_svm_init_with_ops(svm,
        amdgpu_svm_range_invalidate, ...,
        NULL, NULL, ...); /* no quiesce */
} else {
    amdgpu_svm_init_with_ops(svm,
        amdgpu_svm_restore_invalidate, ...,
        amdgpu_svm_quiesce_compute,
        amdgpu_svm_resume_compute, ...);
}
```

XNACK-off gets its own invalidate callback, its own GC, and a quiesce/resume pair. Covered in two slides later on.

Standing on the Framework

drivers/gpu/drm/drm_gpusvm.c - shared with Intel Xe

Framework provides	Driver implements
Notifier / range red-black trees	ops->invalidate
hmm_range_fault wrapper	PTE flag computation
DMA mapping + notifier_seq	Page table writes
notifier_lock	driver_svm_lock
512 MB notifier / 2M-64K-4K chunks	GC reclaim policy



One notifier covers many ranges. KFD SVM registers one per range.

```
drm_gpusvm_init(&svm->gpusvm, "AMDGPU SVM",
                adev_to_drm(adev), current->mm, 0,
                max_pfn << AMDGPU_GPU_PAGE_SHIFT,
                512MB, /* notifier size */
                &amdgpu_gpusvm_ops,
                amdgpu_svm_chunk_sizes, /* 2M, 64K, 4K */
                ARRAY_SIZE(amdgpu_svm_chunk_sizes));
```

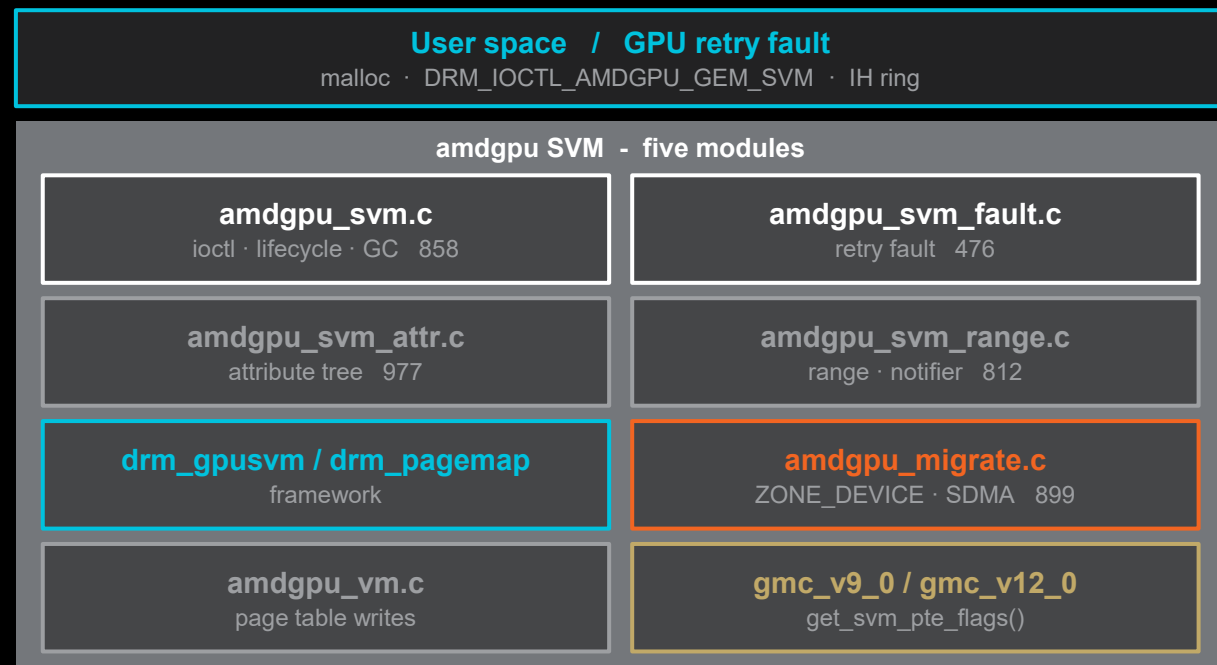
Cost

You must obey the framework's locking contract.

Three commits in this series exist purely to deal with that.

Architecture and Core Structures

Five modules; embedded extension follows the amdgpu convention



```
struct amdgpu_svm_range {
    struct drm_gpusvm_range base; /* embedded */
    struct list_head work_node;
    bool gpu_mapped;
    u8 queue_state;
    uint32_t attr_flags;
    ktime_t validate_timestamp;
};
```

What the private state buys

gpu_mapped - fast path, skip mapped ranges
validate_timestamp - 2 ms fault dedup
checkpoint_ts - drop faults older than unmap

```
struct amdgpu_svm {
    struct drm_gpusvm gpusvm; /* first member */
    struct kref refcount; /* _get / _put */
    struct amdgpu_vm *vm;
    struct rw_semaphore svm_lock; /* driver_svm_lock */
    bool xnack_enabled;
};
```

get_svm_pte_flags() keeps per-IP MTYPE logic out of the SVM core

```
#define DRM_AMDGPU_GEM_SVM 0x1a /* render node */
struct drm_amdgpu_gem_svm {
    __u64 start_addr; __u64 size;
    __u32 operation; /* SET / GET / RESET_ATTR */
    __u32 nattr; __u64 attrs_ptr;
};
```

Attribute intervals, not alloc / free - the kernel only tags memory that malloc already returned.

The Fault Path

The single most important slide

```
GPU retry fault -> IH ring
+- amdgpu_vm_handle_fault()
  +- unreserve root PD (SVM re-takes it)
  +- amdgpu_svm_handle_fault()
    +- checkpoint_ts filters stale faults
    +- [LOCK] svm_lock
    +- find / create attribute interval
    +- fault_map_range()          retry x 3
      +- (1) garbage_collector()
      +- (2) fault_check_allowed()
      +- (3) range_find_or_insert() 2M/64K/4K
      +- (4) 2ms dedup / gpu_mapped fast path
      +- (5) get_pages() -> hmm_range_fault()
      +- (6) update_mapping()
        +- [LOCK] root PD dma_resv
        +- [LOCK] notifier_lock (R)
        |   +- pages_valid() <== CRITICAL
        |   +- map_range()   write PTE
        |   +- update_pdes() write PDE
        +- [UNLOCK]
```

Why step 6 matters

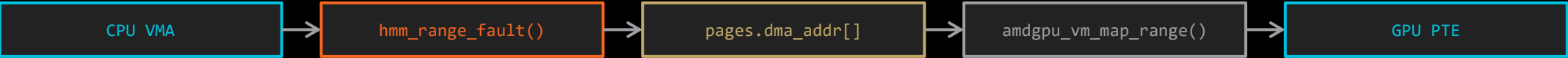
Between getting DMA addresses in step 5 and writing the PTE, the CPU may already have unmapped the pages.

pages_valid() and the PTE write must be atomic under notifier_lock read mode.

Three retry loops

Error	Meaning
-EPERM	read-only VMA, retry RO
-EAGAIN	CPU side changed
-EBUSY	HMM timed out

Data flow: CPU pages -> DMA addresses -> GPU page tables



MMU Notifier Path and Garbage Collection

The other half of SVM correctness

CPU: munmap / migrate



drm_gpusvm_notifier_invalidate()

down_write(notifier_lock)



Pass 1 · event_begin

zap PTEs, clear gpu_mapped, one flush_tlb at the end



Pass 2 · event_end

DMA unmap; UNMAP event -> enqueue to GC



GC workqueue

svm_lock -> drm_gpusvm_range_remove()

Why two passes?

Pass 1 only clears PTEs and accumulates, then flushes the TLB once - instead of flushing per range.

Why defer removal to GC?

The notifier context cannot take svm_lock (it would invert against the fault path), so it can only enqueue. GC exists to dodge lock ordering, not for speed.

What does checkpoint_ts do?

Record the IH ring timestamp on UNMAP; any later fault with an older timestamp is dropped - it targets a mapping that is already gone.

VRAM Migration (1/2) - Setup and Policy

Register VRAM as device-private memory, then decide when to move

1. Register VRAM as MEMORY_DEVICE_PRIVATE

```
/* amdgpu_svm_migration_init() */
pgmap->type = MEMORY_DEVICE_PRIVATE;
pgmap->owner = AMDGPU_PGMAP_OWNER(adev);
r = devm_memremap_pages(adev->dev, pgmap);

/* owner token lets HMM tell our device-private
   pages apart from another device's */
```

2. Two callback tables handed to the framework

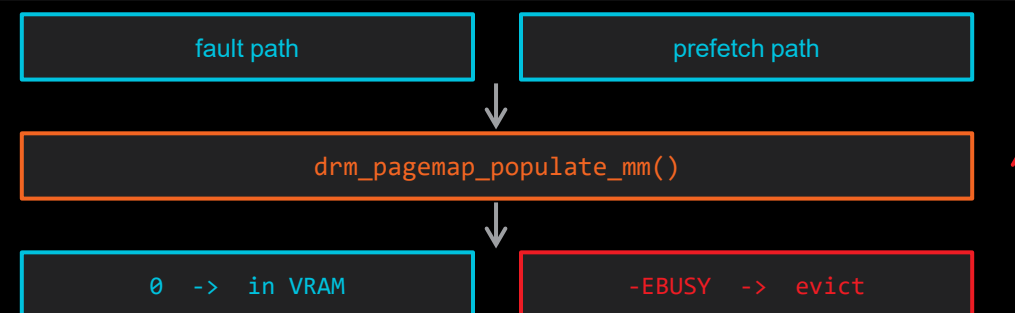
```
const struct drm_pagemap_ops amdgpu_svm_drm_pagemap_ops = {
    .destroy, .device_map, .populate_mm
};
static const struct drm_pagemap_devmem_ops amdgpu_pagemap_ops = {
    .devmem_release, .populate_devmem_pfn,
    .copy_to_devmem, .copy_to_ram
};
```

The framework drives migration; the driver only supplies device-specific primitives.

Nothing here is gated on the XNACK mode - migration hangs off `amdgpu_svm_range_map_attrs()`, which both modes call.

3. Fault, prefetch and restore all trigger migration

```
/* amdgpu_svm_range_migrate_to_vram() */
do {
    ret = drm_pagemap_populate_mm(dpagemap,
        start, end, svm->gpusvm.mm, 0);
    if (ret == -EBUSY && retries)
        drm_gpusvm_range_evict(&svm->gpusvm, range);
} while (ret == -EBUSY && retries--);
```



-EBUSY means another migration raced us - evict, then retry once.

Policy comes from the attribute tree

PREFERRED_LOC / PREFETCH_LOC decide whether a range wants VRAM; `amdgpu_svm_attr_prefer_vram()` is the single decision point.

VRAM Migration (2/2) - The Data Path

From buddy blocks to device PFNs, then SDMA through a GART window

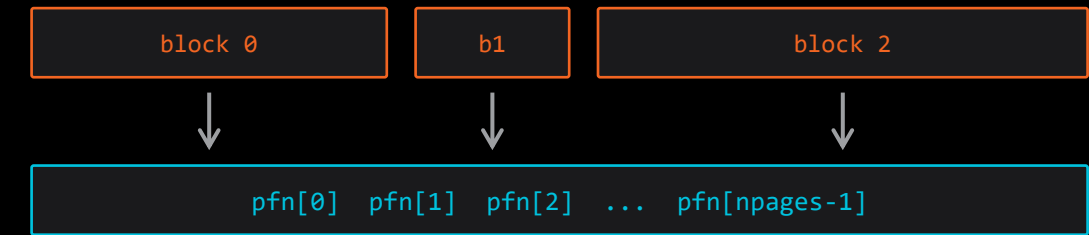
1. Allocate a VRAM BO that carries the devmem allocation

```
/* amdgpu_bo_svm_alloc() */
bp.bo_ptr_size = sizeof(struct amdgpu_bo_svm);
bp.domain      = AMDGPU_GEM_DOMAIN_VRAM;
bp.type        = ttm_bo_type_device;
```

2. Turn TTM buddy blocks into device PFNs

```
amdgpu_res_first(bo->tbo.resource, 0, ..., &cursor);
while (cursor.remaining && i < npages) {
    u64 pfn_base = PHYS_PFN(cursor.start +
                           svm_dm->hpa_base);
    ... pfn[i] = pfn_base + j;
}
```

VRAM is fragmented into buddy blocks; the cursor flattens them



pfn_base = PHYS_PFN(cursor.start + hpa_base) - a linear array is what HMM expects

3. The actual copy: SDMA through a GART window

```
/* amdgpu_svm_copy_memory_gart() */
const u64 max_pages =
    AMDGPU_GTT_MAX_TRANSFER_SIZE >> PAGE_SHIFT;
mutex_lock(&entity->lock); /* GART window */
... amdgpu_copy_buffer(ring, gart_s, gart_d, ...)
mutex_unlock(&entity->lock);
```

Direction	Callback	Trigger
RAM -> VRAM	copy_to_devmem	fault / prefetch
VRAM -> RAM	copy_to_ram	CPU access / eviction

The way back: TTM eviction

amdgpu_svm_bo_evict() migrates device pages back to RAM so VRAM can be reclaimed under pressure.

Mirrors the existing KFD svm_migrate_copy_memory_gart() pattern - proven code, new plumbing.

XNACK-off - A Different Model

No retry means mappings must be valid before the GPU runs

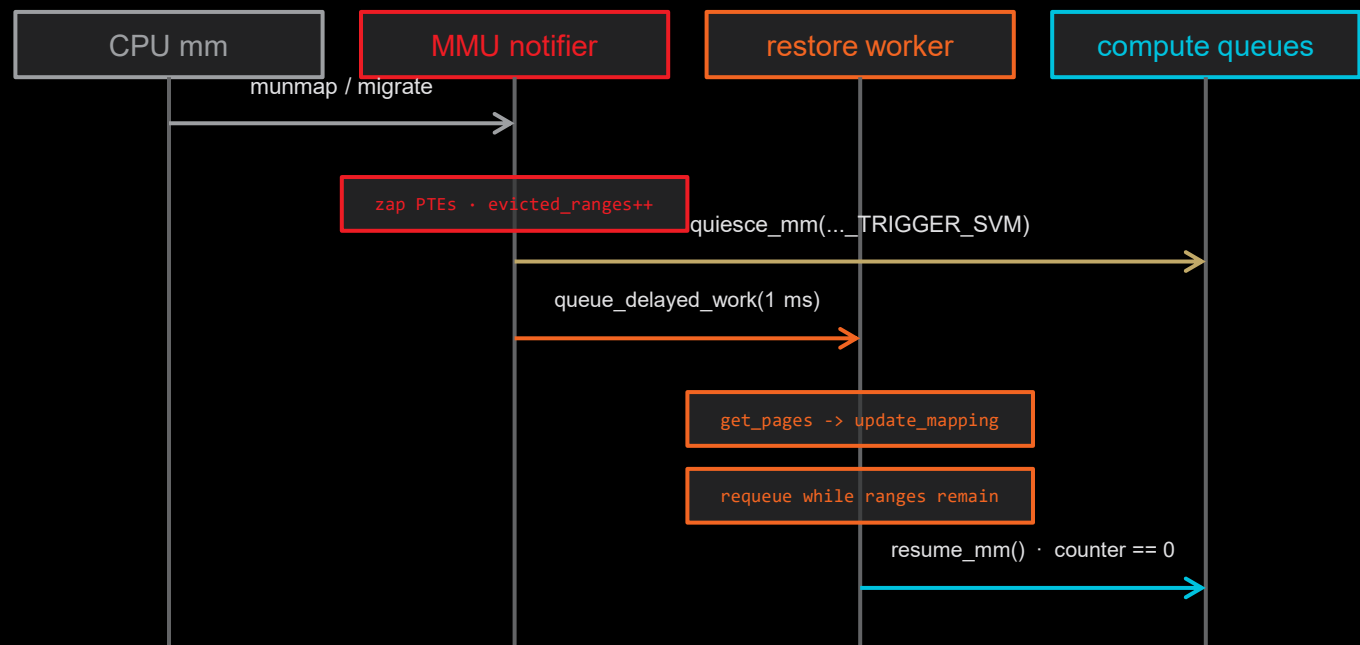
XNACK-on - lazy, fault-driven

GPU faults -> hardware retries -> driver restores on demand.

XNACK-off - eager, quiesce-driven

A fault kills the wave, so stop the queues, rebuild every mapping, resume.

The quiesce -> restore -> resume cycle



Only an empty restore list and a zero counter release the compute queues.

```
/* amdgpu_userptr.h */
#define AMDGPU_SVM_RANGE_RESTORE_DELAY_MS 1
struct amdgpu_svm_restore {
    struct workqueue_struct *wq;
    struct list_head list;
    struct delayed_work work;
    atomic_t quiesced;
    atomic_t evicted_ranges;
    void (*begin)(struct amdgpu_svm *);
    void (*end)(struct amdgpu_svm *);
};
```

Four constraints, one design

- No fault path -> map eagerly, even if attrs did not change
- Notifier zaps PTEs synchronously -> stop the queues first
- get_pages() can fail -> restore must be asynchronous
- Partial unmap -> GC must rebuild, not just delete



XNACK-off - The Quiesce Window

One counter opens it, one compare-and-swap closes it

evicted_ranges is the window - not a range count

counter == 0 · queues running, no window



0 -> 1 svm_restore_eviction_begin()

atomic_inc_return() returns 1 -> begin() = quiesce_compute()



1 -> N more invalidates arrive

Merged into the same window - no second quiesce, no second resume



worker: restore.list and gc.list both empty?

Checked three times - GC's partial rebuild feeds work back into restore



atomic_cmpxchg(&evicted_ranges, record, 0)



== record -> resume

!= record -> stay open

record is the snapshot the worker took when it started; the compare runs under notifier_lock.

Open and close, side by side

```
/* open - svm_restore_eviction_begin() */
if (atomic_inc_return(&svm->restore.evicted_ranges) != 1)
    return; /* already open */
svm->restore.begin(svm); /* quiesce_compute */

/* close - amdgpu_svm_restore_worker() */
drm_gpusvm_notifier_lock(&svm->gpusvm);
if (!has_pending &&
    atomic_cmpxchg(&evicted_ranges, record, 0) == record)
    svm->restore.end(svm); /* resume_compute */
```

When the quiesce is skipped

full-range unmap or !gpu_mapped -> skip
 partial unmap + gpu_mapped -> quiesce
 A whole-range munmap() no longer stops the queues.

Errors that must never be retried

-ENOENT -EFAULT -EPERM -EINVAL -EHWPOISON

Retry one of those forever and the counter never reaches zero - the queues stop for good.

Locking: One Lock, Not Three

Three commits, all Suggested-by: Christian König

The problem - one VM's page tables are updated under different lock domains

```
svm_lock (rwsem)      drm_gpusvm range tree      fault / attr / GC
notifier_lock (rwsem) DMA mappings + notifier_seq framework + bind
eviction_lock (mutex) PTE writes + TTM eviction  map / unmap_range

fault path:      svm_lock -> PD resv -> notifier_lock(R) -> eviction_lock
notifier path:   notifier_lock(W) -> eviction_lock          (no svm_lock)
classic B0:      PD resv -> eviction_lock                  (neither)
```

No single lock covers every path that writes page tables.

The fix - make them the same lock

1 eviction_lock becomes selectable

```
struct rw_semaphore default_eviction_lock;
struct rw_semaphore *eviction_lock;

if (p->unlocked) down_write(...);
else             down_read (...);
```

2 page-table alloc can reclaim

```
amdgpu_vm_update_unlock(p); /* drop */
r  = amdgpu_vm_pt_create(...); /* alloc */
r2 = amdgpu_vm_update_relock(p);

/* seqno moved? -> -EAGAIN */
```

3 point it at the notifier lock

```
vm->eviction_lock =
    &svm->gpusvm.notifier_lock;
smp_store_release(&vm->svm, svm);

/* alias visible before vm->svm */
```

On an SVM VM, eviction_lock IS notifier_lock - one serialization point, no inversion, no recursion.

Plain graphics VMs keep default_eviction_lock and are unchanged. svm_lock is registered as the framework's driver_svm_lock - the lock GPU SVM asks for, not an extra one.

Patch Series Status

21 + 8 commits · xnack-on series plus the xnack-off series

```
-- xnack-on core (12)
1 add SVM ioctl UAPI definitions
2 add SVM core header and VM integration
3 implement SVM attribute tree and helper functions
4 implement SVM attribute set/get/clear operations
5 add SVM range types and work queue interface
6 gmc: add get_svm_pte_flags callback
7 implement SVM range GPU mapping core
8 implement SVM range notifier and GC helpers
9 add SVM notifier invalidate callback and checkpoint
10 implement SVM initialization and lifecycle
11 add SVM ioctl entry and fault handler module
12 integrate SVM into build system and VM fault path
-- locking rework Suggested-by: Christian König (3)
13 make VM eviction lock read/write to match notifier
14 recheck range validity after page table allocation
15 use drm_gpusvm notifier lock for VM eviction
-- VRAM migration - serves both xnack modes (6)
16 add VRAM migration infrastructure for drm_pagemap
17 implement drm_pagemap SDMA migration callbacks
18 implement synchronous TTM eviction for SVM BOs
19 hook up ZONE_DEVICE registration in device init/reset
20 add SVM range migration helpers for drm_pagemap
21 integrate VRAM migration into SVM fault and prefetch
```

xnack-off series (8)

```
1 add xnack-off restore types header
2 integrate restore types into core headers
3 implement restore core and invalidate cb
4 implement xnack-off restore worker
5 implement xnack-off GC work function
6 add compute queue quiesce / resume helpers
7 add attr change boundary realign helper
8 wire restore into lifecycle and attr set
```

Giving back to the framework

A THP page-count fix for `drm_pagemap` came out of this work - now sent upstream on its own, no longer carried by the series.

Before posting

[] no systematic performance numbers yet

[] xnack-off still carries local TRACE/ERR shims

[] cover letter Testing section empty

Roadmap and Takeaways



today's topic

Phase 1 is complete and going upstream; Phase 2 is in progress. Both are covered today, and VRAM migration underpins both - it is not tied to either XNACK mode.

Three takeaways

- 1 Framework first**
Reuse DRM GPUSVM and share evolution with Xe - and a framework THP counting fix went back upstream on its own
- 2 One lock, not several**
A given page table must be serialized by a single lock; merging notifier_lock and eviction_lock was the key
- 3 No retry changes everything**
XNACK-off turns a lazy fault handler into an eager quiesce / restore / resume pipeline - same data structures, opposite control flow

References

- This talk
 - <https://indico.freedesktop.org/event/12/contributions/537/>
- Patch series
 - [1] <https://lore.kernel.org/dri-devel/20260612090928.29682-1-ray.huang@amd.com/>
 - [2] <https://lore.kernel.org/amd-gfx/20260519092313.3115848-1-honglei1.huang@amd.com/>
 - [3] <https://lore.kernel.org/amd-gfx/20260605075340.20199-1-Junhua.Shen@amd.com/>
 - [4] <https://lore.kernel.org/dri-devel/20260603065620.2555316-1-honglei1.huang@amd.com/>
- X.Org Developer's Conference 2025 - GPU SVM in DRM / Xe
 - <https://indico.freedesktop.org/event/10/contributions/437/>
- AMD ROCm Documentation
 - <https://rocm.docs.amd.com/en/latest/>

Thank You and Q&A



Copyright and Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED ‘AS IS.’ AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

© 2026 Advanced Micro Devices, Inc. All rights reserved.

AMD, the AMD Arrow logo, Radeon, Ryzen and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies. Linux is a trademark of Linus Torvalds and OpenCL is a trademark of Apple Inc. Windows and DirectX are the registered trademarks of Microsoft Corporation in the US and other jurisdictions.

