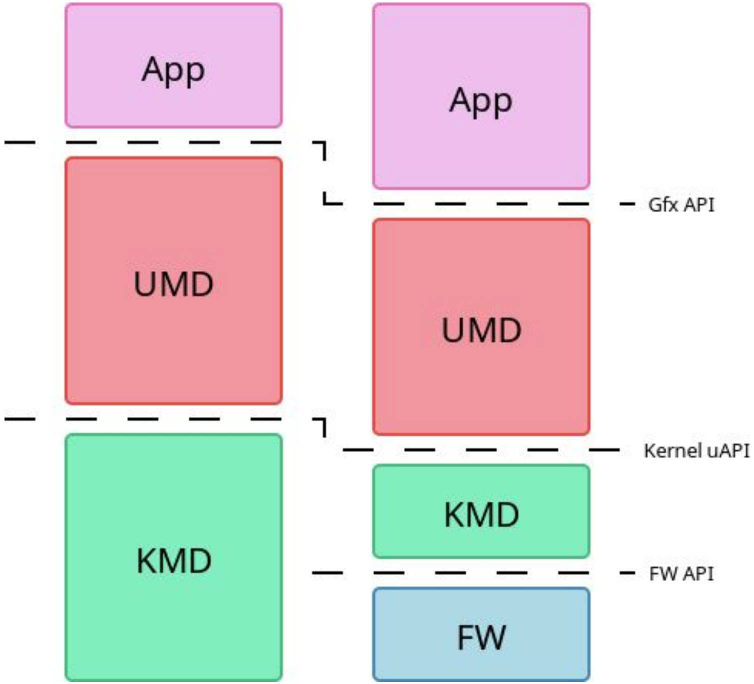
With great power comes less responsibility

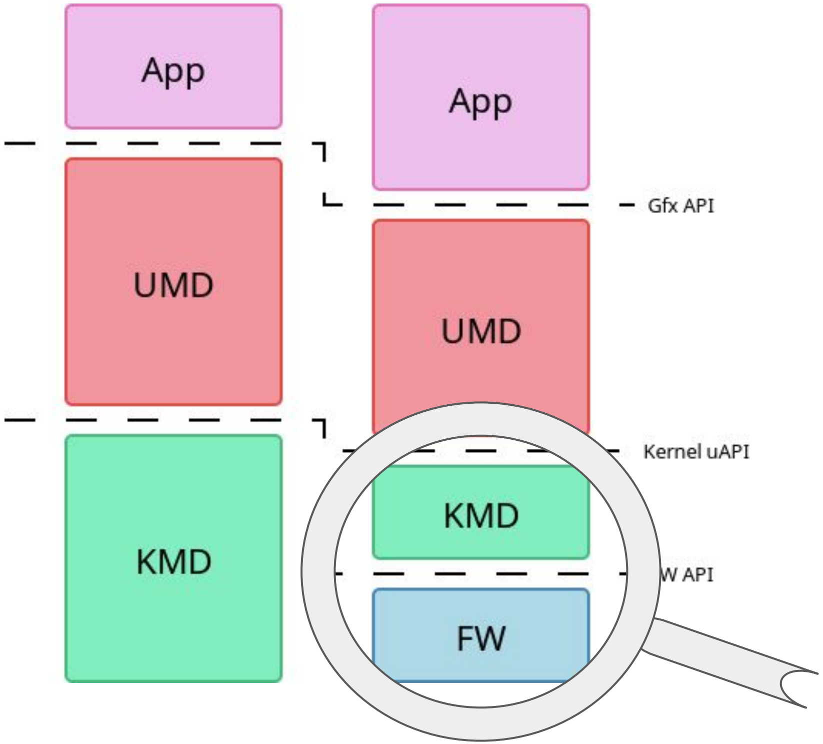
Boris Brezillon (Collabora)
Danilo Krummrich (Red Hat)

Kernel mode driver, can you move away please?

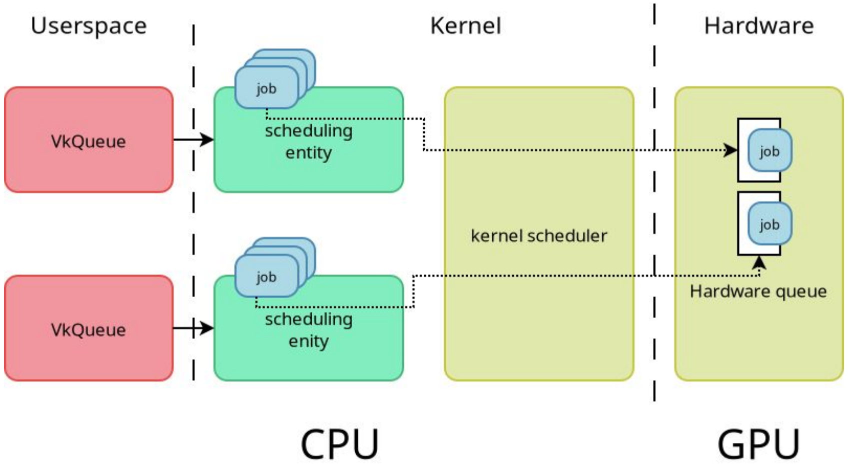
- Modern userspace want more control
- GPU vendors want things to be fast and consume less power
- Less work to do. Should be easy-peasy, but…
  - We need to make sure UMD can't break the system (some amount of checking is needed)
  - We need to interact with a new piece of HW (the MCU executing the FW)
  - Some frameworks no longer fit the bill
  - We have new features to support



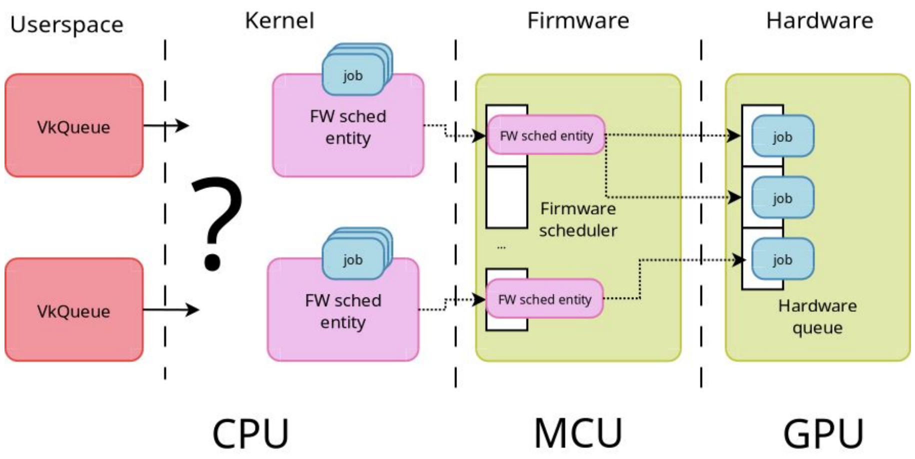
Kernel mode driver, on the hardware front



Kernel-based vs firmware-based scheduling

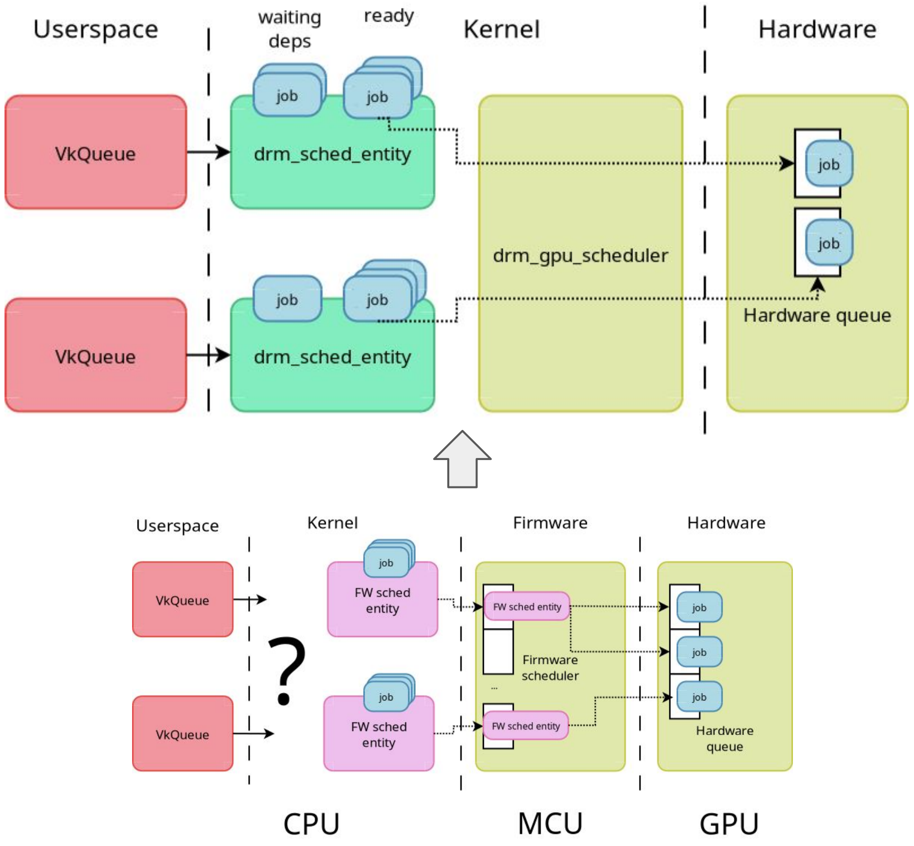


VS



drm_sched original design

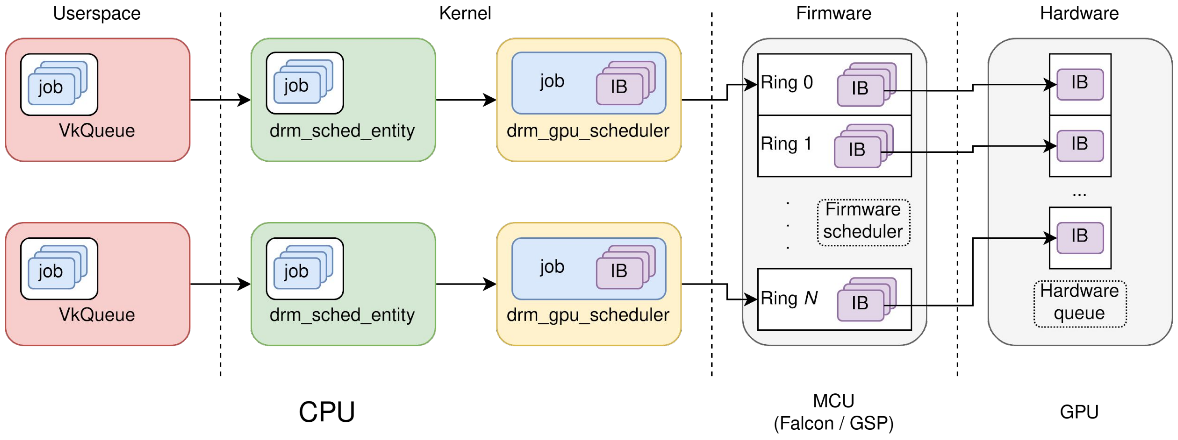
- Designed for kernel-based scheduling
- Deals with job dependencies
- Priority-based entity selection with RR or FIFO policy
- It's of great use for KMD drivers, but…
- … it's doing too much for FW-based scheduling



Red Hat | COLLABORA

Solution: Teach drm_sched to be dumb

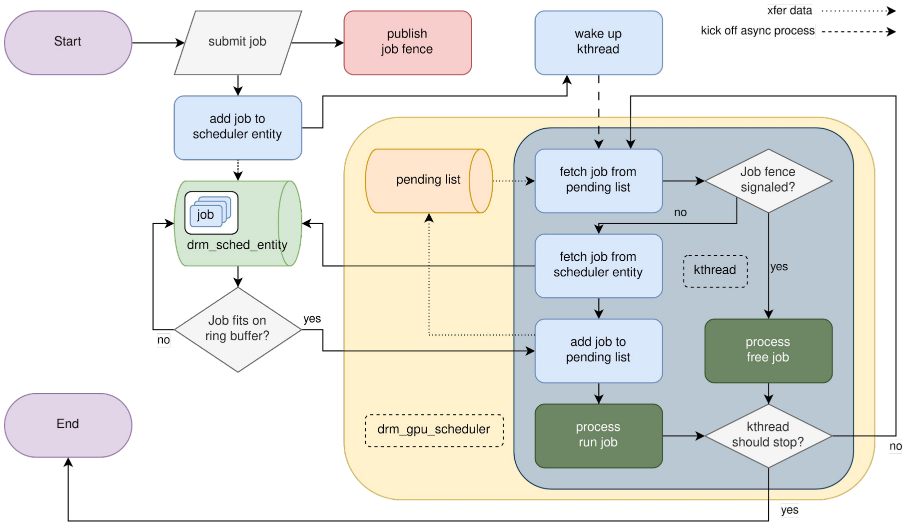
- Work conducted by **Matthew Brost** from Intel
- Single-entity scheduling policy
- drm_sched still deals with job dependencies
- The rest is left to the FW



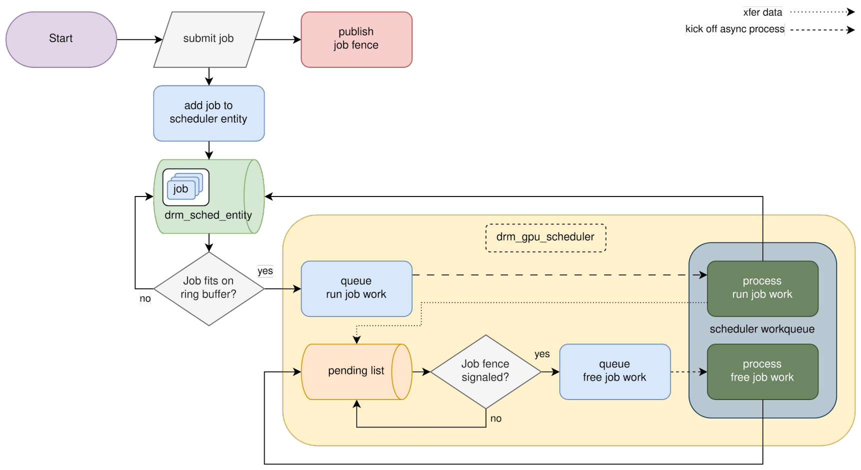
Drm_sched single-entity implementation details

- Multi entity scheduler:
  - One scheduler per execution engine
  - One thread per scheduler
  - => number of threads is acceptable
- Single entity scheduler
  - One scheduler per entity
  - Still one thread per scheduler
  - => number of threads explodes
- Solution => use a workqueue instead of a thread and let drivers pass their own workqueue
- Fast path for single-entity scheduling (no complex entity selection needed, the FW takes care of that)

drm_sched single-entity implementation details



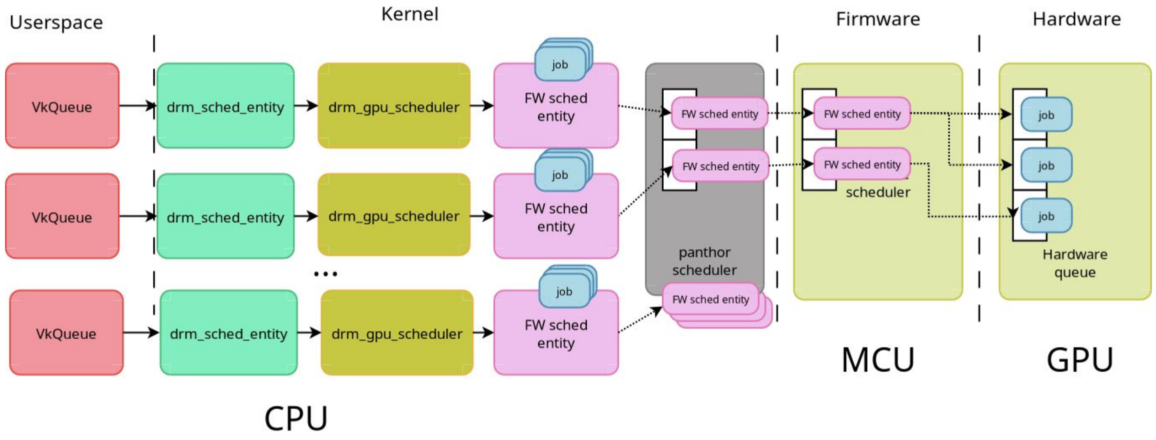
kthread



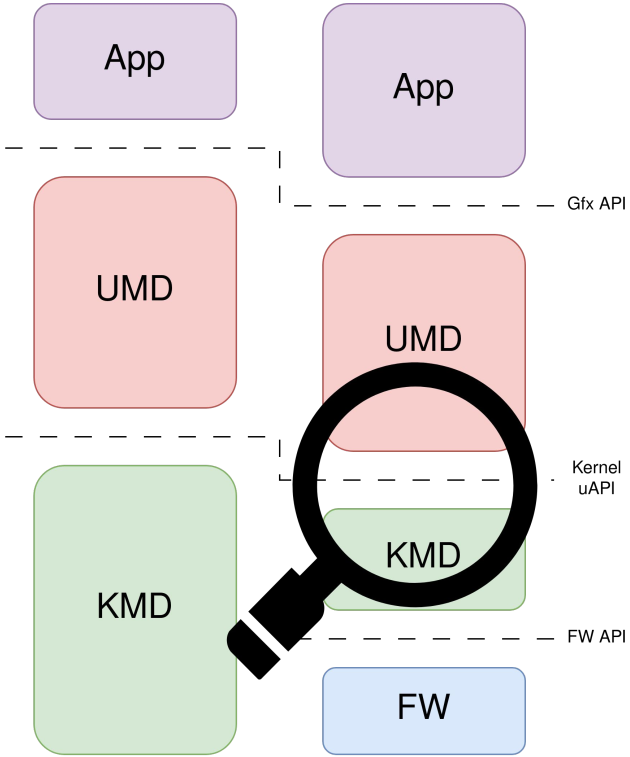
workqueue

FW-based scheduling, the Mali way

- Small number of FW scheduling slots available
- The kernel has to take part in the scheduling process
- Adds another level of scheduling kernel side
- Should work with the usermode queue model ;-)

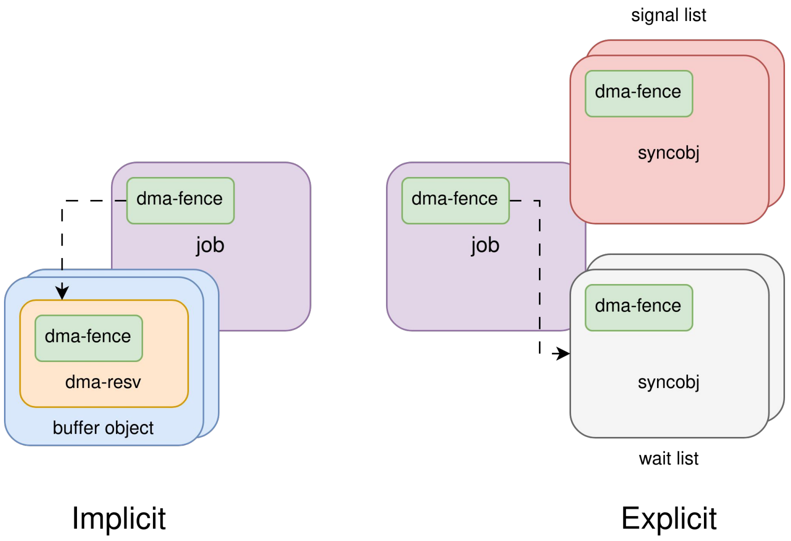


Kernel mode driver, on the user mode driver front



What Vulkan wants

- Vulkan has some new requirements not working with existing UAPIs
  - e.g. explicit synchronization and advanced VM management
- lead to new “VM_BIND style” UAPIs giving userspace control of the GPU’s virtual address space

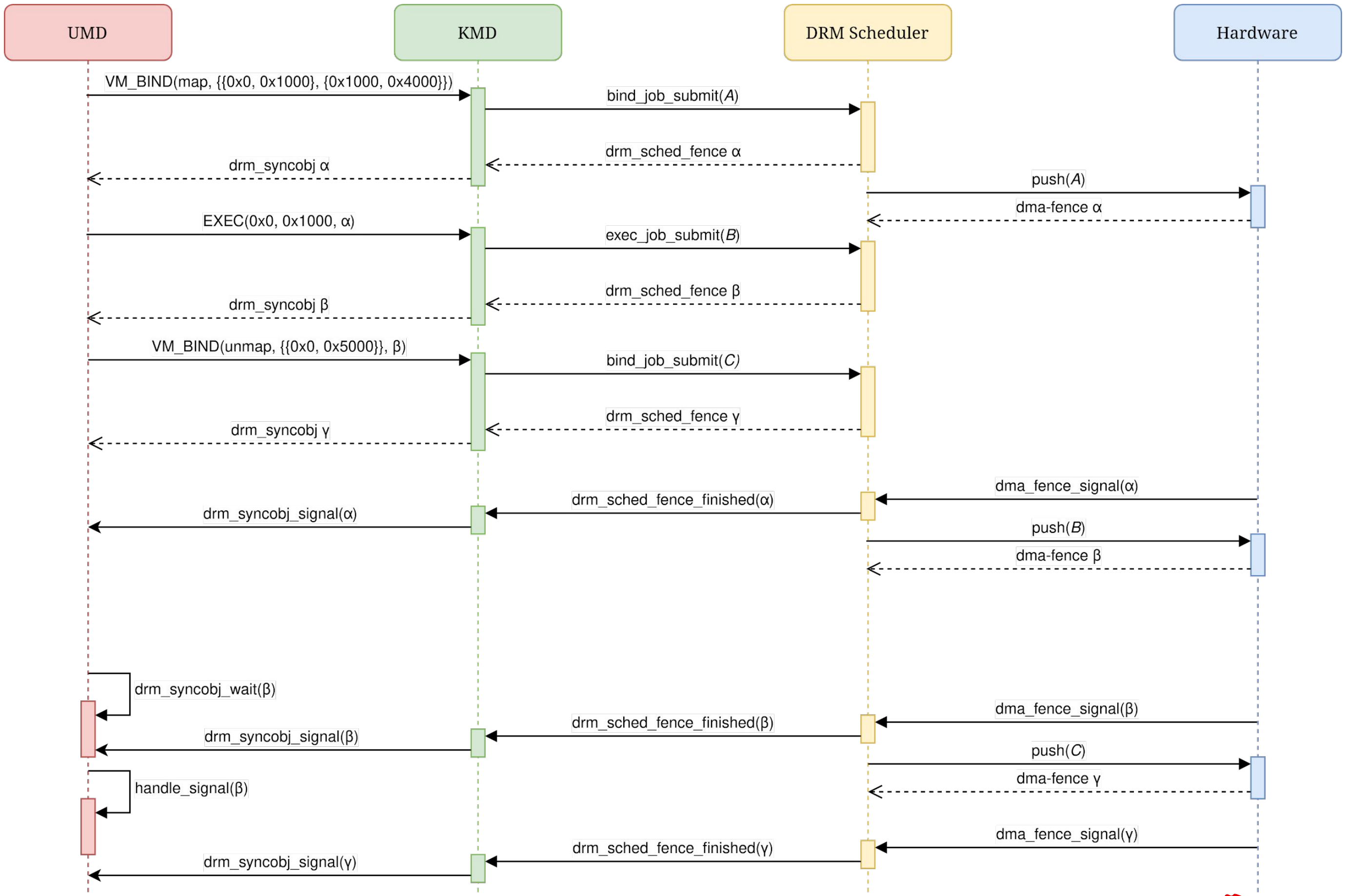


Red Hat | COLLABORA

VM_BIND style UAPIs

- VM_INIT – create a new GPU virtual address space
- VM_BIND – bind actual memory to a virtual address (create a mapping)
  - parameters: operation type (map/unmap), (virtual) address, size, BO (handle), offset within the BO, synchronization objects (syncobj; wait list, signal list)
  - legal for map/unmap operations to arbitrarily span across existing mappings
  - synchronous and asynchronous variants
- EXEC – execute a GPU command buffer
  - parameters: virtual base address, size, syncobjs (in / out)
  - command buffers / shaders can operate on the whole VA space
    - hence requires validation underlying BOs of the VA space

Red Hat | COLLABORA


UMD
KMD
DRM Scheduler
Hardware
VM_BIND(map, {{0x0, 0x1000}, {0x1000, 0x4000}})
bind_job_submit(A)
drm_sched_fence α
drm_syncobj α
push(A)
dma-fence α
EXEC(0x0, 0x1000, α)
exec_job_submit(B)
drm_sched_fence β
drm_syncobj β
VM_BIND(unmap, {{0x0, 0x5000}}, β)
bind_job_submit(C)
drm_sched_fence γ
drm_syncobj γ
dma_fence_signal(α)
drm_sched_fence_finished(α)
drm_syncobj_signal(α)
push(B)
dma-fence β
drm_syncobj_wait(β)
dma_fence_signal(β)
drm_sched_fence_finished(β)
drm_syncobj_signal(β)
push(C)
handle_signal(β)
dma-fence γ
dma_fence_signal(γ)
drm_sched_fence_finished(γ)
drm_syncobj_signal(γ)
Red Hat
COLLABORA

DRM GPUVM

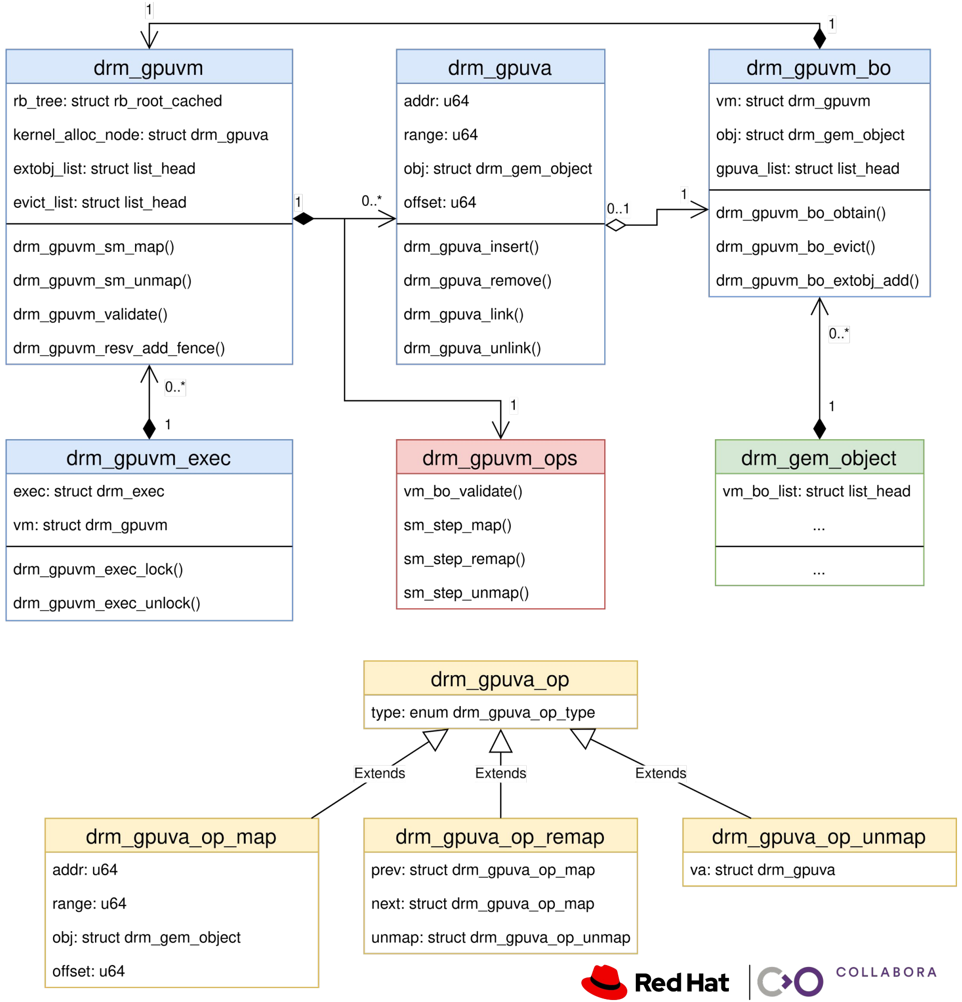
- common component to manage a GPU virtual address space
  - motivated by (but not limited to) Vulkan motivated UAPIs (VM_BIND)
- GPU Virtual Memory (Address Space)
- GPUVM was originally called DRM GPUVA Manager (in v6.6)
  - DRM GPU Virtual Address Manager
  - drivers typically call their structure VM
  - kernel documentation for *Asynchronous VM_BIND* and *VM_BIND locking* calls it “gpu_vm”
- Shout-out to **Dave Airlie** (Red Hat), who suggested having such a component in the first place

DRM GPUVM – What does it do?

- Merged upstream, comes with v6.6
  - infrastructure to track GPU VA allocations and mappings
  - connect GPU VA mappings to their backing buffers (DRM GEM objects)
  - break down complex map / unmap requests
    - into a set operations which drivers can perform directly, e.g.
      - mapping requests which intersect existent mappings
      - partial unmap requests
- Upcoming (targets v6.7)
  - common dma-resv for GEM objects local to the GPUVM; tracks external GEM objects
  - helper functions lock all backing GEM objects; based on drm_exec (**Christian König**, AMD)
  - track evicted GEM objects
    - accelerate validation of backing GEM objects

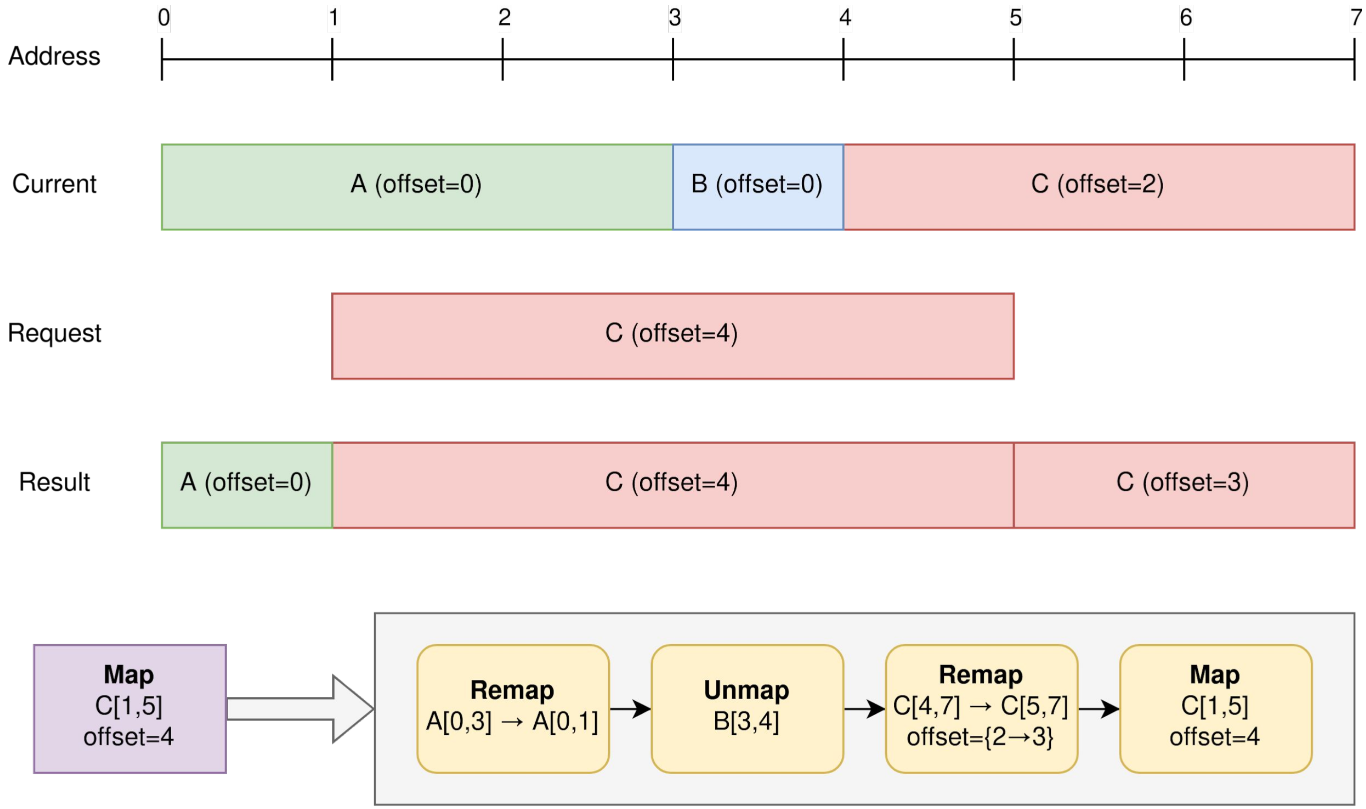
DRM GPUVM - Structure

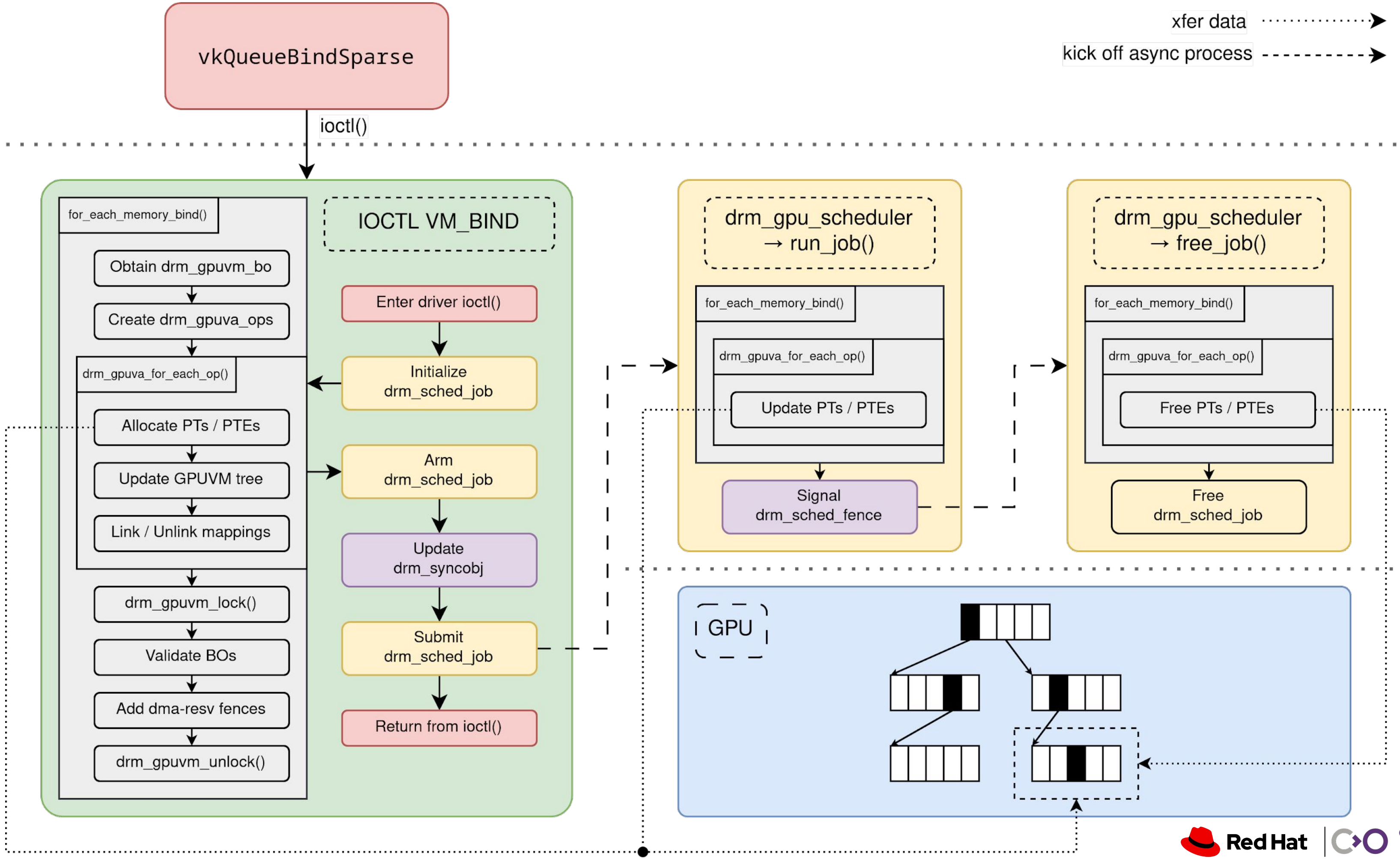
- *drm_gpuvm* - represents the GPU VA space
- *drm_gpuva* - represents a mapping
- *drm_gpuvm_bo* - represents a combination of a VM and a GEM object
- *drm_gpuvm_exec* - drm_exec (**Christian König**, AMD) abstraction to lock / unlock the VA space' mappings backing GEM objects
- *drm_gpuva_op* - base structure for map, remap and unmap operations
- *drm_gpuvm_ops* - driver callbacks of a drm_gpuvm



Red Hat | COLLABORA

DRM GPUVM - Map / Unmap Operations




xfer data
kick off async process
vkQueueBindSparse
ioctl()
IOCTL VM_BIND
for_each_memory_bind()
Obtain drm_gpuvm_bo
Create drm_gpuva_ops
drm_gpuva_for_each_op()
Allocate PTs / PTEs
Update GPUVM tree
Link / Unlink mappings
drm_gpuvm_lock()
Validate BOs
Add dma-resv fences
drm_gpuvm_unlock()
Enter driver ioctl()
Initialize drm_sched_job
Arm drm_sched_job
Update drm_syncobj
Submit drm_sched_job
Return from ioctl()
drm_gpu_scheduler → run_job()
for_each_memory_bind()
drm_gpuva_for_each_op()
Update PTs / PTEs
Signal drm_sched_fence
drm_gpu_scheduler → free_job()
for_each_memory_bind()
drm_gpuva_for_each_op()
Free PTs / PTEs
Free drm_sched_job
GPU
Red Hat
COLLABORA

Driver status update: Nouveau

- New uAPI implementing VM_BIND was merged upstream (released with v6.6)
  - sufficient for NVK to implement a fully functional Vulkan UMD
- Upcoming (targets v6.7):
  - making use of the drm_sched single-entity model
    - waiting for drm_sched patches (**Matthew Brost**, Intel)
  - performance improvements due to the tricks implemented in upcoming *drm_gpuvm* patches (should land in drm-misc-next soon)
- What's missing:
  - userptr support (might be postponed in favor of landing the GSP patches)
  - utilize the DMA engine for page table updates (currently page tables are updated from the CPU)

Driver status update: Panthor (Mali)

- Panthor uAPI should have enough to implement a functional Vulkan driver with all sort of fancy extensions
- Panthor is using the drm_sched single-entity model
- Panthor has a VM_BIND ioctl and is using drm_gpuvm under the hood
- What's missing:
  - More testing
  - Transparent buffer object eviction
    - drm_gem_shmem patches from **Dmitry Osipenko** (Collabora) should help
  - An actual UMD driver making use of all these fancy features (panvk2, we're waiting for you :-))
  - Waiting for drm_sched and drm_gpuvm to be merged

Driver status update: PowerVR

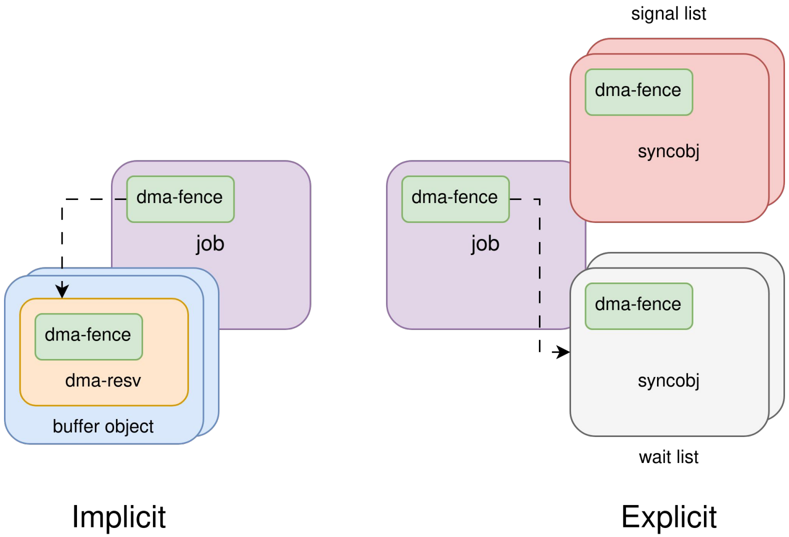
- PowerVR is using drm_sched single-entity
- PowerVR is using drm_gpuvm
- PowerVR has a vulkan driver that makes use of these new ioctls and it’s passing the 1.0 CTS \o/
- What’s missing:
  - VM_BIND is not supported yet, just synchronous VM_MAP/UNMAP
  - Transparent buffer object eviction

Questions?

Additional Slides (not part of the talk)

What Vulkan wants: Explicit synchronization

- Avoiding over-synchronization is the key
- Vulkan forces the user to express synchronization explicitly through various primitives
- Figuring out buffers needed for a specific job might be tricky (bindless)



What Vulkan wants: Advanced VM management

- Lifetime of GPU buffers and their mappings in GPU VA space is well defined in Vulkan
- Sparse bindings (and sparse residency)
  - Image / buffer objects can be partially bound, and take their memory from different VkDeviceMemory objects
- Aliasing: memory can be bound to several objects at the same time (there are restrictions though)
- Some extensions (VK_KHR_buffer_device_address) require fine grained control on the GPU VA space
- → UMDs require control of the GPU's virtual address space

DRM GPUVM - two state tracking modes

- Living in the present moment:
  - VM state is updated right in time, along with the MMU page table update (slight delay if the page table update is GPU-based)
- Planning for the future:
  - VM state is updated when VM_BIND jobs are submitted
  - VM state is ahead until all VM_BIND jobs have been flushed

DRM GPUVM - two state tracking modes

- Living in the present moment:
  - Pros:
    - We can easily query the buffer object mapped at a GPU address without having to revert diffs of pending jobs
    - Fast path for synchronous updates is easier to implement
    - We don’t need complicated unwind logic in the ioctl() to revert the VA space on failure
  - Cons:
    - We have to over-provision page table allocations for async VM_BIND jobs (we don’t know what the VM will look like when we get to execute the job)
    - We can’t easily query the future VM state

DRM GPUVM - two state tracking modes

- Planning for the future:
  - Pros:
    - We can easily query the future VM state
    - We don’t have to over-provision for page table allocation
  - Cons:
    - VM_BIND (sync) are queued as async jobs which are waited upon in the ioctl path. Fast-tracking of such operation is possible, but requires extra infrastructure to track fences per VM range, plus a dedicated VM bind queue for sync operation.
    - Querying the current VM state is more complicated (might be a problem if the kernel driver needs to get a BO from a GPU address)