# Virtgpu DRM Native Contexts

Fast gfx in a vm with this one little trick!

Rob Clark
XDC2022

# Classes of GPU Virtualization

- Device Emulation
- API Remoting
- Fixed Passthrough
- Mediated Passthrough

# Classes of GPU Virtualization

- Device Emulation
  - Are you kidding?
- API Remoting
- Fixed Passthrough
- Mediated Passthrough

# Classes of GPU Virtualization

- Device Emulation
- API Remoting
  - Ie, virgl, gfxstream, venus
  - 12-86% of native
  - Isolation?
- Fixed Passthrough
- Mediated Passthrough

# Classes of GPU Virtualization

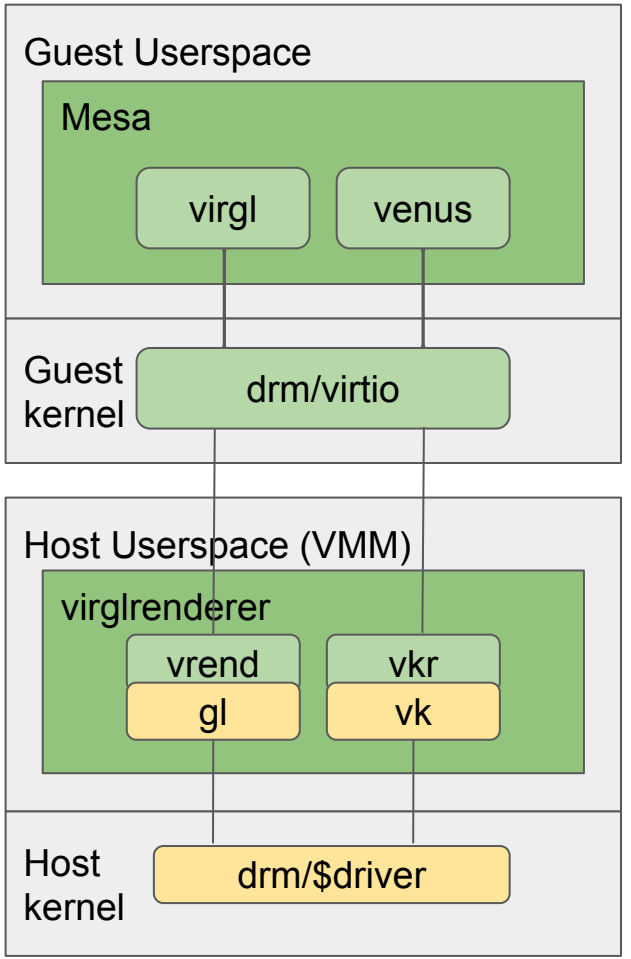
- Device Emulation
- API Remoting
- Fixed Passthrough
  - Ie. PCI passthrough
  - Near native perf
  - Not useful if host and other VMs also need same GPU
- Mediated Passthrough

# Classes of GPU Virtualization

- Device Emulation
- API Remoting
- Fixed Passthrough
- Mediated Passthrough
  - Great… if your hw supports it
  - Fence/buffer integration with host?

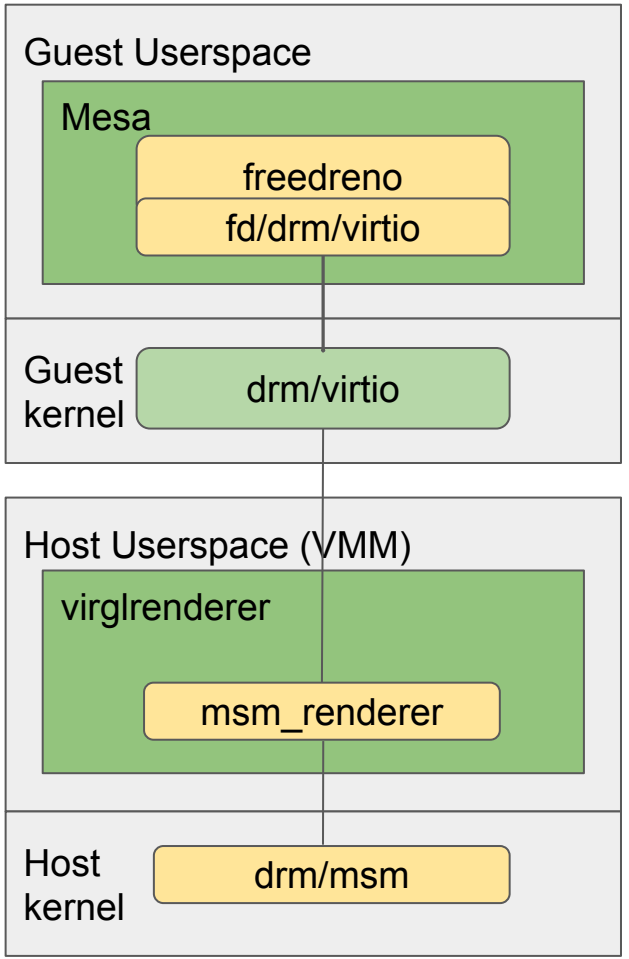
# About virtgpu

- Aka virtio_gpu aka drm/virtio
  - Upstream virtio based guest kernel vgpu driver
- Host/guest interop
- Recent(ish) additions
  - Blob resources
  - Context Type
  - Ring-idx



# DRM Native Context!

- Native usermode driver in guest!
- API remoting at kernel uabi layer
  - Level of least frequent calls
  - And least frequent change
  - Make it more async!
- Hooks in at src/freedreno/drm
  - Basically equiv of winsys layer
- Integrates with virtgpu's buffer/fence passing
- Not much code
  - < 2kloc guest userspace
  - ~ 1.3kloc in virglrenderer



# Virtgpu DRM Native Context: Structure

- Device fd's (virglrenderer):
  - 1 device fd (`struct drm_file *`) per guest process
  - Guest drm_file 1:1 with host virglrenderer context
  - Virglrenderer context 1:1 with host drm_file
  - GPU address spaces are 1:1 with drm_file
  - One GPU address space per guest process
- GEM:
  - 1:1 between host and guest (plus shmem buffer)
  - All host-storage blob's (`VIRTGPU_RESOURCE_CREATE_BLOB`)

# Virtgpu DRM Native Context: Fences/Sync

- `ring_idx == 0`: The CPU timeline
  - Used in cases where guest needs to wait for host CPU
- `ring_idx > 0`: Maps 1:1 to GPU priority levels
  - Which map 1:1 to host dma-fence contexts / timelines
- res_id handles passed to `VIRTGPU_EXECBUFFER` ioctl
- Synchronization in guest
  - Don’t block in host VMM
  - TODO virtgpu needs proto to pass host fences from guest

# Virtgpu DRM Native Context: Protocol

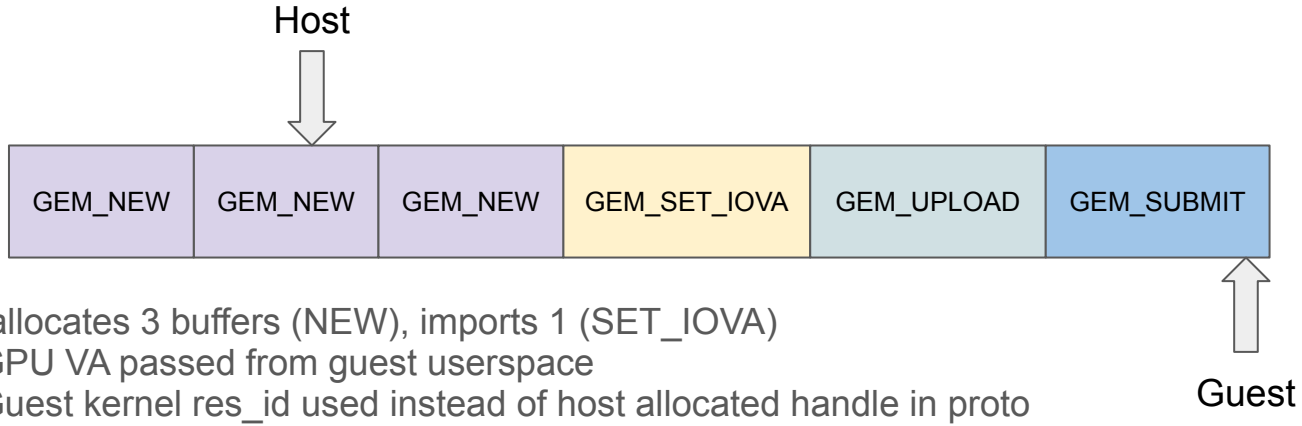
- Guest → Host: `VIRTGPU_EXECBUFFER`
  - Req messages tunneled over EB
  - Batching for async requests
- Host → Guest: shmem buffer
  - Rsp messages written into shmem rsp_mem buffer at offset guest asks for
  - Keeps the design/implementation of host VMM simple

# Virtgpu DRM Native Context: Protocol

- Keep it asynchronous!!
  - To mitigate host ⇔ guest latency, keep hot-paths async
- Host uapi additions to support this
  - Userspace allocated GPU virtual address – GEM create/import can be async
  - Seqno fence # assignment in userspace – GEM submit can be async
- Treat errors as context-lost

# Virtgpu DRM Native Context: Protocol

- Keep it asynchronous!!  Simple Example:




- Guest allocates 3 buffers (NEW), imports 1 (SET_IOVA)
  - GPU VA passed from guest userspace
  - Guest kernel res_id used instead of host allocated handle in proto
- UPLOAD can avoid mmap into host or immediate mmap
  - Guest mmap requires host to have actually allocated the BO

# Alternatives? rendernode fwding?

- Rendernode device exposed in guest
  - Fwd (modern subset) of ioctls to host
  - Use unmodified mesa in guest
- Too synchronous!
  - Ioctls return a value + _IOC_READ
  - Existing uapi designed around low syscall cost
  - Better to embrace the async!

# Show me the codez

- VirglRenderer:
  - src/drm/msm/msm_proto.h – the guest ⇔ host protocol
  - src/drm/msm/msm_renderer.c – one .c file, ~1.3kloc
  - src/drm – helper to deal with fences, simple driver loader
- mesa:
  - src/freedreno/drm/virtio

# Adding your driver in three easy steps (1/3)

- Step #1 – add context id and extend capset (drm_hw.h):

```
struct virgl_renderer_capset_drm {
   uint32_t wire_format_version;
   /* Underlying drm device version: */
   uint32_t version_major;
   uint32_t version_minor;
   uint32_t version_patchlevel;
#define VIRTGPU_DRM_CONTEXT_MSM   1
   uint32_t context_type;
   uint32_t pad;
   union {
      struct {
         uint32_t has_cached_coherent;
         ...
      } msm;  /* context_type == VIRTGPU_DRM_CONTEXT_MSM */
   } u;
};
```

# Adding your driver in three easy steps (2/3)

- Step #2 – ???
  - Define and implement your own protocol
  - Then add yourself to the loader table in drm_renderer.c:

```
static const struct backend {
   uint32_t context_type;
   const char *name;
   int (*probe)(int fd, struct virgl_renderer_capset_drm *capset);
   struct virgl_context *(*create)(int fd);
} backends[] = {
#ifdef ENABLE_DRM_MSM
   {
      .context_type = VIRTGPU_DRM_CONTEXT_MSM,
      .name = "msm",
      .probe = msm_renderer_probe,
      .create = msm_renderer_create,
   },
#endif
};
```

# Adding your driver in three easy steps (3/3)

- Step #3 – Profit!

# Status / Next Steps

- Possible optimizations:
  - Fencing improvements in virtgpu – pass host fences back to host
  - Reduce host → guest fence latency
- Virtgpu drm_syncobj support
- QEMU support

# Thank you

Questions?