GENERIC ALLOCATOR UPDATE
James Jones, XDC 2017
OVERVIEW

Current Design
Prototype Status
Problems Encountered
Next Steps
CURRENT DESIGN
ALLOCATOR’S SPOT IN THE ECOSYSTEM

Application: Wayland Compositor
EGL/Vulkan Driver
Vendor1 Drv
DRM
Device 1

Application: Wayland EGL Client
EGL/Vulkan Driver
Vendor2 Drv
Device 2

EGL/Vulkan Driver
ION Drv
Device 3
ION

Allocators

Userspace
Kernel
ALLOCATOR OBJECTS

**ASSERTION**
The desired width, height, and format of a surface

**USAGE**
A single desired application of a surface, such as rendering, on a single device

**CONSTRAINT**
An imposed surface limitation for a given assertion and usage

**CAPABILITY**
A supported surface feature for a given assertion and usage

**CAPABILITY SET**
A valid combination of constraints and capabilities
CURRENT WORKFLOW
Based on USAGE.md on github project

1. Initialize an allocator device from a device file descriptor
2. Query capability sets from the device given an assertion and list of usages
3. [Optional] Query capability sets from additional devices with the same parameters
4. [Optional] Merge capability sets of desired devices to find common capabilities
5. Try allocating a surface on available devices until allocation succeeds
6. Import surfaces to graphics APIs, mode setting APIs, video APIs, etc.
PROTOTYPE STATUS
SUPPORTED/PLANNED FUNCTIONALITY

Goal is to Encourage and Substantiate Design Discussion

Creating Devices - IMPLEMENTED
Querying Capabilities and Constraints - IMPLEMENTED
Merging Capabilities and Constraints - IMPLEMENTED
Creating Allocations from Capabilities and Constraints - IMPLEMENTED
Exporting/Importing Allocations - TODO
Using Allocations in Vulkan/OpenGL - TODO
Using Allocations in DRM/Non-Graphics Devices - TODO
CAPABILITY SET MATH

Core of the Design

Current set derivation algorithm: merge/union constraints, intersect capabilities

Capabilities can be “required”. If operation removes a required capability, it fails

Needs more validation. Throw your worst usage/constraints/capabilities at it!
CAPABILITY SET MATH EXAMPLE

**DEV_1 SET 1 [A]**

Constraints:
1. Address aligned to 32B

Capabilities:
1. NVIDIA tiling/layout (*)
2. NVIDIA FB compression

**DEV_1 SET 2 [B]**

Constraints:
1. Address aligned to 64B

Capabilities:
1. Pitch-linear layout (*)

**DEV_2 SET 1 [C]**

Constraints:
1. Address aligned to 32B
2. Pitch aligned to 64B

Capabilities:
1. Pitch-linear layout (*)
2. Dev2 FB compression
CAPABILITY SET MATH EXAMPLE

DEV_1 SET 1 [A]

Constraints:
1. Address aligned to 32B

Capabilities:
1. NVIDIA tiling/layout (*)
2. NVIDIA FB compression

DEV_2 SET 1 [C]

Constraints:
1. Address aligned to 32B
2. Pitch aligned to 64B

Capabilities:
1. Pitch-linear layout (*)
2. Dev2 FB compression

FAIL!

Constraints:
1. Address aligned to 32B
2. Pitch aligned to 64B

Capabilities:
**CAPABILITY SET MATH EXAMPLE**

**DEV_1 SET 2 [B]**
- **Constraints:**
  1. Address aligned to 64B
- **Capabilities:**
  1. pitch-linear layout (*)

**DEV_2 SET 1 [C]**
- **Constraints:**
  1. Address aligned to 32B
  2. Pitch aligned to 64B
- **Capabilities:**
  1. Pitch-linear layout (*)
  2. Dev2-FB compression

**NEW VALID SET**
- **Constraints:**
  1. Address aligned to 64B
  2. Pitch aligned to 64B
- **Capabilities:**
  1. Pitch-linear layout (*)
  2. Dev2-FB compression
PROBLEMS ENCOUNTERED
DEVICE ENUMERATION/CREATION/IMPORT

Device file doesn’t necessarily uniquely identify a logical device object

Device creation from FD implies lack of need for additional /dev/file access

Alternative of exporting devices from APIs is problematic too

Enumeration/Correlation using UUID from Vulkan/GL APIs would provide consistency
NO DEVICE-LOCAL CAPABILITIES

Ex: local caching

GPU may have on-chip cache. When to use it? When capabilities say so of course!

Other devices don’t necessarily need to be aware of this cache usage

Intersecting capabilities from other devices will remove this “local cache” capability
FORMAT SPECIFICATION

Still an open issue that needs to be resolved. Prototype assumes RGBA8888

Khronos Data-format spec, FOURCC, ???

Needs to handle HDR formats

Should there be supported format enumeration?
IMPORT TO EXTERNAL APIS

Unlike Vulkan/OpenGL import APIs, additional meta-data is needed

How should that meta-data be packaged? DRM format modifiers not sufficient

Does the capability set suffice? If so, see issue with device-local capabilities

Is some level of in-kernel meta-data preferred? Limits future suballocation usage
RELATIONSHIP TO DMA-BUF

Unclear if it should be required that import/export consume/produce DMA-BUF FDs

Might bake Linux-specific assumptions into the API or usage

Even FDs can be non-portable

Any value in using DMA-BUF when usage is limited to a single device or driver stack?
NEXT STEPS

WHAT IS THAT?

THE FUTURE!!!
Vulkan introduced the idea of explicitly transitioning between various surface uses

Could be generalized across devices now that we can describe all usage explicitly

Apps could query usage transitions “meta-data” from allocator for usage pairs

That meta-data could then be passed into GPU APIs to perform transitions
MOTIVATION FOR USAGE TRANSITIONS

Alternative proposal
Re-allocate when usage changes

Justification
Simpler API
Steady-state is still optimal

Problems
Allocation can be expensive
Transitions have consistent cost
Usage may change at inconvenient times
// Some existing usage definitions
extern const usage_t samplingUsage;
extern const usage_t displayUsage;

// Usage lists
const usage_t sampling[] = { samplingUsage };  
const usage_t samplingAndDisp[] =  
  { samplingUsage, displayNVUsage };  
const usage_t dispOnly[] = { displayUsage };  
void *transitionData;
size_t transitionDataSize;

// Query a usage transition from an allocator library device
query_transition(dev,
  ARRAYLEN(sampling), sampling,
  ARRAYLEN(samplingAndDisp), samplingAndDisp,
  &transitionDataSize, &transitionData);
USAGE TRANSITIONS (EXAMPLE)

// Program the transition in Vulkan
VkImageMemoryCrossDeviceBarrierEXT crossDeviceBarrierData = {
    VK_STRUCTURE_TYPE_IMAGE_MEMORY_CROSS_DEVICE_BARRIER_EXT, // sType
    NULL, // pNext
    transitionDataSize, // dataSize
    transitionData // data
};

VkImageMemoryBarrier usageTransitionBarrier = {
    VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER, // sType
    &crossDeviceBarrierData, // pNext, takes precedence over oldLayout/newLayout members
};

vkCmdPipelineBarrier(..., 1, &usageTransitionBarrier);
WAYLAND INTEGRATION
Getting Back to our Original Goal...

Last year NVIDIA presented a vendor-agnostic EGL winsys client integration layer API

The sample implementation used EGLStream, but the API is mechanism-agnostic

Key functionality: Ability to build an EGLSurface from some lower-level primitive

How do we build an EGLSurface from allocator surfaces?
WHERE DOES THE ALLOCATOR CODE GO?

The prototype is a stand-alone library with runtime-loadable driver backends

However, the key mechanisms could live anywhere

Is it easier to move to this new library, merge functionality into GBM, or ???

If we keep the allocator library, does it need a better name than liballocator?
QUESTIONS I ASKED:

1. Any situations capability set math does not handle?
2. How should device-local capabilities be handled?
3. How should formats be defined?
4. How should surface meta-data be represented?
5. Is DMA-BUF a requirement? If so, why?
6. How should EGLSurface integration work?
7. Where does the allocator implementation live?

https://github.com/cubanismo/allocator
email: jajones ‘at’ nvidia.com
REFERENCES

https://github.com/cubanismo/allocator - Prototype Allocator Implementation & Documentation

https://github.com/cubanismo/allocator/blob/master/USAGE.md - Allocator Example Usage

https://www.khronos.org/registry/DataFormat/ - Khronos Data Format Spec

https://github.com/torvalds/linux/blob/master/include/uapi/drm/drm_fourcc.h - DRM FOURCC formats

https://www.khronos.org/registry/vulkan/specs/1.0-extensions/html/vkspec.html - Vulkan 1.0 spec