Framebuffer Modifiers
Supporting end-to-end graphics compression

Ben Widawsky
About Me

- Worked on all parts of graphics and other driver stacks
- Avid Buffer Modifier
- Motivated by disparity with closed implementations
  - If they can do it; damn it, we can too.
### Summarizing the Work

<table>
<thead>
<tr>
<th>Linux DRM</th>
<th>Linux i915</th>
<th>Mesa</th>
<th>Protocol</th>
<th>Khronos</th>
</tr>
</thead>
<tbody>
<tr>
<td>blobifier</td>
<td>blobifier</td>
<td>DRI</td>
<td>DRI3.1</td>
<td>image_dma_buf_import_modifiers</td>
</tr>
<tr>
<td>AddFB2</td>
<td>AddFB2</td>
<td>EGL</td>
<td>0</td>
<td>VK_EXT_external_*</td>
</tr>
<tr>
<td>multi-plane</td>
<td></td>
<td>ANV</td>
<td>0 Mutter</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RADV</td>
<td>0 Weston</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>other</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Intel**: ✔ ✔ ✔ ✔ ✔ ✔
- **Collabora**: ✔ ✔ ✔ ✔ ✔ ✔
- **Google**: ✔ ✔ ✔ ✔
- **Others**: ✔ ✔ ✔
Status

- Many years in the making: almost there!
- Early modifier support already released: Mesa 17.2, Kernel 4.14
- Full compression support soon:
  - Modesetting
  - Wayland/Mutter?
  - X.org/DRI3
    - Mesa DRI3 support
    - Mesa Wayland support
Mountains out of Molehills?

- Each EU needs 1GB/s bandwidth
  - Texturing (trilinear, anisotropic)
  - Transparency/Blending
  - Antialiasing
- Display
  - $3840 \text{ px} \times 2160 \text{ rows} \times 4 \text{ Bbpp} \times 60 \text{ Hz} = 1.85 \text{GB/s}$
  - It keeps getting worse
    - Increasing resolutions (5K, 8K)
    - Increasing refresh rates (120Hz, 240 Hz)
- Workloads are already memory bandwidth limited
  - Can’t scale up compute without more bandwidth
  - Reduce visual effects
  - Decrease resolution
Admiring the Problem
Texture Upload

The application needs to get its assets (geometry data, texture data, precompiled shaders, etc.) into memory from storage.
Texturing Fetch/Filtering

- Anisotropic 16 trilinear probes
- 128x
- Trilinear 8x
- Bilinear 4x
- Nearest 1x-2x

Bandwidth

Quality
#version 330

uniform sampler2D tex;
in vec2 texCoord;
out vec4 fragColor;

void main() {
    vec4 temp = texelFetch(tex, ivec2(texCoord));
    fragColor = temp;
}
Compositing

Compositor is responsible for taking client application’s window contents and amalgamates into a single image for display.

Like a window manager, but with offscreen buffers

- Needs to read from application’s rendered data, and write to the screen
Display Engine

Specialized, fixed function hardware which sources pixel data and pushes it out over some display protocol; possibly blending, and scaling the pixels along the way.
# Bandwidth Costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Color Depth</th>
<th>Desc.</th>
<th>Bandwidth</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Texture Upload</td>
<td>1Bpc (RGBX8)</td>
<td>File to DRAM</td>
<td>16KB (64 * 64 * 4)</td>
<td>W</td>
</tr>
<tr>
<td>Texel Fetch</td>
<td>1Bpc (RGBX8)</td>
<td>DRAM to Sampler</td>
<td>16KB (64 * 64 * 4)</td>
<td>R</td>
</tr>
<tr>
<td>FB Write</td>
<td>1Bpc (RGBX8)</td>
<td>GPU to DRAM</td>
<td>16KB (64 * 64 * 4)</td>
<td>W</td>
</tr>
<tr>
<td>Compositing</td>
<td>1Bpc (RGBX8)</td>
<td>DRAM to DRAM</td>
<td>32KB (64 * 64 * 4 * 2)</td>
<td>R+W</td>
</tr>
<tr>
<td>Display Scanout</td>
<td>1Bpc (RGBX8)</td>
<td>DRAM to PHY</td>
<td>16KB (64 * 64 * 4)</td>
<td>R</td>
</tr>
</tbody>
</table>

Total Bandwidth = (16 + 16 + 16 + 32 + 16) * 60Hz = **5.625 MB/s**
At Least it Looks Better

<table>
<thead>
<tr>
<th>Filter Mode</th>
<th>Multiplier (texel fetch stage)</th>
<th>Total Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nearest</td>
<td>1x</td>
<td>5.625 MB/s</td>
</tr>
<tr>
<td>Bilinear</td>
<td>4x</td>
<td>11.25 MB/s</td>
</tr>
<tr>
<td>Trilinear</td>
<td>8x</td>
<td>18.75 MB/s</td>
</tr>
<tr>
<td>Aniso 4x</td>
<td>32x</td>
<td>63.75 MB/s*</td>
</tr>
<tr>
<td>Aniso 16x</td>
<td>128x</td>
<td>243.75 MB/s*</td>
</tr>
</tbody>
</table>

* Oblique angle + implementation details would reduce further
Proposed Solution: Increase Headroom

<table>
<thead>
<tr>
<th>Color Depth</th>
<th>Operation</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1Bpc</td>
<td>Texture Upload</td>
<td>16KB (64 * 64 * 4)</td>
</tr>
<tr>
<td>1Bpc</td>
<td>Texel Fetch</td>
<td>16KB (64 * 64 * 4)</td>
</tr>
<tr>
<td>1Bpc</td>
<td>FB Write</td>
<td>16KB (64 * 64 * 4)</td>
</tr>
<tr>
<td>1Bpc</td>
<td>Composite</td>
<td>32KB (64 * 64 * 4 * 2)</td>
</tr>
<tr>
<td>1Bpc</td>
<td>Scanout</td>
<td>16KB (64 * 64 * 4)</td>
</tr>
</tbody>
</table>

Total Bandwidth = 5.625 MB/s

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DDR3-2133 34GB/s (dual channel)</td>
<td>DDR4-3200 51.2 GB/s (dual channel)</td>
<td>50%</td>
</tr>
<tr>
<td>GTX780 (Kepler) GDDR5 288 GB/s</td>
<td>GTX1080 (Pascal) GDDR5X 352 GB/s</td>
<td>22%</td>
</tr>
<tr>
<td>Radeon R9 290X (Hawaii) GDDR5 (320GB/s)</td>
<td>Radeon R9 Fury (Fiji) HBM1 512GB/s</td>
<td>60%</td>
</tr>
<tr>
<td>LPDDR3-1600 12.8 GB/s (single channel)</td>
<td>LPDDR4-3200 25.6 GB/s (single channel)</td>
<td>100%</td>
</tr>
</tbody>
</table>
Proposed Solution: Hardware Composition

Hardware is capable of having multiple hardware planes. Use them.

<table>
<thead>
<tr>
<th>Color Depth</th>
<th>Operation</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1Bpc (RGBA8)</td>
<td>Texture Upload</td>
<td>16KB (64 * 64 * 4)</td>
</tr>
<tr>
<td>1Bpc (RGBA8)</td>
<td>Texel Fetch</td>
<td>16KB (64 * 64 * 4)</td>
</tr>
<tr>
<td>1Bpc (RGBA8)</td>
<td>FB Write</td>
<td>16KB (64 * 64 * 4)</td>
</tr>
<tr>
<td>1Bpc (RGBA8)</td>
<td>Composite</td>
<td>32KB (64 * 64 * 4 * 2)</td>
</tr>
<tr>
<td>1Bpc (RGBA8)</td>
<td>Scanout</td>
<td>16KB (64 * 64 * 4)</td>
</tr>
</tbody>
</table>

Total Bandwidth = 3.75 MB/s (33% savings)
Proposed Solution: Texture Compression

- DXT1 (8:1)
- ETC1/2 (4:1)
- ASTC (variable, 6:1)

<table>
<thead>
<tr>
<th>Color Depth</th>
<th>Operation</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXT1</td>
<td>Texture Upload</td>
<td>16KB / 8</td>
</tr>
<tr>
<td>DXT1</td>
<td>Texel Fetch</td>
<td>16KB / 8</td>
</tr>
<tr>
<td>1Bpc</td>
<td>FB Write</td>
<td>16KB (64 * 64 * 4)</td>
</tr>
<tr>
<td>1Bpc</td>
<td>Composite</td>
<td>32KB (64 * 64 * 4 * 2)</td>
</tr>
<tr>
<td>1Bpc</td>
<td>Scanout</td>
<td>16KB (64 * 64 * 4)</td>
</tr>
</tbody>
</table>

Total Bandwidth = 3.925 MB/s (30%)
Problems (Increase Bandwidth)

1. Limited by process and design
2. Costly for manufacturing
   a. New memory modules
   b. New boards
   c. Utilizes new fabrication process
3. May be power hungry

Rating: Sure. Won’t hold my breath
Problems (More Planes)

1. Hardware specific
   a. Not all hardware can composite the same number of planes
2. Max planes is small
   a. Increasing this significantly isn’t feasible (die size)
3. Only helps compositing step

Rating: Great, doesn’t scale
Problems (Texture Compression)

1. May be lossy
2. Hardware compatibility
   a. Better formats require new hardware
   b. Increased gate counts
3. Patents or proprietary
4. Misses display improvement
5. Doesn’t play nicely with all filtering methods (aniso)

Rating: Great, but lacking
# Introducing E2E Lossless Compression

## Pros
- Lossless
- Transparent to applications/tools
  - Easier development
  - Hardware improvements automatically help
- No offline compression necessary
- Compression benefits through display
- Can be huge when texturing is small amount of bandwidth consumption

<table>
<thead>
<tr>
<th>Pros</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lossless</td>
<td></td>
</tr>
<tr>
<td>Transparent to applications/tools</td>
<td></td>
</tr>
<tr>
<td>Easier development</td>
<td></td>
</tr>
<tr>
<td>Hardware improvements automatically help</td>
<td></td>
</tr>
<tr>
<td>No offline compression necessary</td>
<td></td>
</tr>
<tr>
<td>Compression benefits through display</td>
<td></td>
</tr>
<tr>
<td>Can be huge when texturing is small amount of bandwidth consumption</td>
<td></td>
</tr>
</tbody>
</table>

## Cons
- Relatively low compression
  - 2:1 max on current Intel
  - 4:1 max seems to be industry standard
  - Not everything will be compressible
    - Will never get max.
- Limited by hardware
  - However, many GPUs getting this
    - UBWC (Qualcomm), AFBC (ARM), DCC (AMD), DCC (Nvidia), PVRIC (Imagination)

<table>
<thead>
<tr>
<th>Cons</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Relatively low compression</td>
<td></td>
</tr>
<tr>
<td>2:1 max on current Intel</td>
<td></td>
</tr>
<tr>
<td>4:1 max seems to be industry standard</td>
<td></td>
</tr>
<tr>
<td>Not everything will be compressible</td>
<td></td>
</tr>
<tr>
<td>Will never get max.</td>
<td></td>
</tr>
<tr>
<td>Limited by hardware</td>
<td></td>
</tr>
<tr>
<td>However, many GPUs getting this</td>
<td></td>
</tr>
<tr>
<td>UBWC (Qualcomm), AFBC (ARM), DCC (AMD), DCC (Nvidia), PVRIC (Imagination)</td>
<td></td>
</tr>
</tbody>
</table>
Several Opportunities for Savings

Reduced Bandwidth

BLORP Upload

Reduced Bandwidth

Reduced Bandwidth

Display Scanout
Dumb Example

Naive implementation will get 2:1 compression when a pair of cachelines has 12 or less colors

CCS cacheline = 32K main frame
## E2E Bandwidth Savings (2:1 compression)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Color Depth</th>
<th>Desc.</th>
<th>Bandwidth</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Texture Upload</td>
<td>1Bpc (RGBX8)</td>
<td>File to DRAM</td>
<td>16KB (64 * 64 * 4)</td>
<td>W</td>
</tr>
<tr>
<td>Texel Fetch</td>
<td>1Bpc (RGBX8)</td>
<td>DRAM to Sampler</td>
<td>16KB (64 * 64 * 4)</td>
<td>R</td>
</tr>
<tr>
<td>FB Write</td>
<td>Compressed (2:1)</td>
<td>GPU to DRAM</td>
<td>16KB (64 * 64 * 4) / 2</td>
<td>W</td>
</tr>
<tr>
<td>Compositing</td>
<td>Compressed (2:1)</td>
<td>DRAM to DRAM</td>
<td>32KB (64 * 64 * 4 * 2) / 2</td>
<td>R+W</td>
</tr>
<tr>
<td>Display Scanout</td>
<td>Compressed (2:1)</td>
<td>DRAM to PHY</td>
<td>16KB (64 * 64 * 4) / 2</td>
<td>R</td>
</tr>
</tbody>
</table>

*Total Bandwidth* = (16 + 16 + 8 + 16 + 8) * 60Hz = **3.75 MB/s (33%)**
Molehills out of Mountains!

<table>
<thead>
<tr>
<th>Technique</th>
<th>Bandwidth</th>
<th>BW Savings</th>
<th>Disk Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>5.625 MB/s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>+ HW compositing</td>
<td>3.75 MB/s</td>
<td>33%</td>
<td>0%</td>
</tr>
<tr>
<td>+ DXT1 Compression</td>
<td>2.11 MB/s</td>
<td>62%</td>
<td>30%</td>
</tr>
<tr>
<td>+ E2E Compression</td>
<td>1.17 MB/s</td>
<td>79%</td>
<td>30%</td>
</tr>
</tbody>
</table>
Intermission
Implementation Challenges

1. Currently, everything treats a framebuffer as a buffer of pixels.
   a. The main buffer is no longer just pixel data.
   b. There’s another buffer! (similar to planar formats)
2. Buffer allocation, buffer import/export, and display server protocol need to be made aware of this.
3. Applications and compositors cannot rely on compression working everywhere.
   a. Ex. Skylake doesn’t allow compression on pipe C
Several Solutions

1. Encode “modifiers” in fourcc format
   a. V4L does this (include/uapi/linux/videodev2.h)
   b. Works well for entirely proprietary formats
   c. Concern about amount of bits for modifiers in DRM
      i. Graphics formats combinatorially explode faster [apparently]
      ii. Even 64b modifier was questioned
   d. Never really considered (not sure why)

2. Intel specific plane property *(original proposal)*
   a. Many other drivers shared similar problem.
   b. KMS clients wanted a hardware agnostic mechanism
   c. Protocol still required anyway

3. dma-buf metadata
   a. Just a get/set IOCTL for adding modifiers to a dma-buf
The Result - Modifiers

- *Some* support already landed
- Describes **modifications** to a buffer's layout
- Easy to add new modifiers to support different tiling formats
- Missing some key pieces
  - Query interface
  - Protocol
  - Driver implementation
- Compression somewhat muddies the definition

commit e3eb3250d84ef97b766312345774367b6a310db8
Author: Rob Clark <robdclark@gmail.com>
Date: Thu Feb 5 14:41:52 2015 +0000

drm: add support for tiled/compressed/etc modifier in addfb2
Step 1: Compositor Negotiation

Query all “sink” APIs to find out what modifiers are supported for the given format, and hardware.
Queries

- Blobifier (KMS blob property for drm_plane)
  - What modifiers does the plane support?
- EGL extensions
  - EXT_image_dma_buf_import_modifiers
    - eglQueryDmaBufModifiersEXT
      - What modifiers does my format support?
      - "is used to query the dma_buf format modifiers supported by <dpy> for the given format."
- Vulkan/WSI (WIP)
  - VK_MESAX_external_image_dma_buf.

Plumbers: Collabora (funded by Intel and Google), Google, Intel
Step 2: Take That and Shove It
down your protocol pipe

With the optimal modifiers in hand, some protocol will tell the client which modifiers it might want to use.

Modifiers which can be used for direct scanout, and used to sample from EGLImages

Send modifiers over protocol (e.g., DRM 1.1, Wayland, kmscube)

DRI client
Protocols

- Wayland
  - “zwp_linux_buffer_params_v1" version="3"
- DRI3.1
  - Multi-plane support
  - xDRI3GetSupportedModifiers

Plumbers: Collabora (paid for by Intel)
Step 3: Making BOs

Next, the client creates the buffer either directly, or indirectly with the formats and modifiers it desires.
Buffer Creation

- **EGL**
  - eglCreateWindowSurface
    - Wayland
    - X11
      - (Mesa) Ask over DRI3.1 what's supported
      - (Mesa) Call into DRI driver to create an image

- **GBM**
  - gbm_*_create_with_modifiers

- **DRIImage**
  - createImageWithModifiers (made for GBM)
  - createImageFromDmaBufs2 (made for DRI 3.1)

- **Vulkan/WSI (WIP)**

Plumbers: Collabora, Google, Intel
The Whole Story Thus Far

1. The KMS client (AKA the compositor) will query the kernel via KMS APIs to find out what modifiers are supported for the formats it wishes to use on the sinks it wishes to use. To do this, first the KMS client will use the connector found via drmModeGetConnector(), and get a random CRTC from the connector. Next, the primary plane can be found with drmModeGetProperty, and finally, the modifiers come in as a blob drmModeGetPropertyBlob().

1b. The compositor also queries other sink APIs, eg EGL for texturing from buffer or encoder for on-screen capture.

2. The final list of supported format/modifier combinations is sent to the DRI client over the display server protocol or some other pre-defined protocol when not using normal client/server display server protocols.

3. The DRI client may query the supported modifiers it wishes to use based on the format it wants to render to and creates the buffer. With EGL, this will normally be handled by the EGL winsys layer. It could also be done explicitly via GBM which a compositor or very low level application may do. If it doesn't do this, no modifiers will be used.
Display it

1. Software Compositing (Option)
   a. EGL_EXT_image_dma_buf_import_modifiers
   b. Much work required

2. Hardware compositing (Option)
   a. drmModeAddFB2WithModifiers
   b. Relatively minor changes required. AddFB2 already supported modifiers
      i. Add new modifiers to drm_fourcc.h
      ii. Added error checking when modifiers change plane count.
      iii. Driver specific handling of modifiers.
He’ll Flip You (for real)

Option 1:
Create EGLImage then composites via texturing

dma-buf -> EGLImage

Some other BO HW Plane

BO Format Modifier

km client/compositor

Option 2:
Create drm_framebuffer from client’s BO then flip directly

HW Planes

dma-buf -> drm_framebuffer

BO Format Modifier

km client/compositor
**Preliminary Results**

<table>
<thead>
<tr>
<th>“Benchmark”</th>
<th>Original</th>
<th>CCS</th>
<th>%improved</th>
</tr>
</thead>
<tbody>
<tr>
<td>kmscube</td>
<td>1.22 GB/s</td>
<td>600 MB/s</td>
<td>51 (2x)</td>
</tr>
<tr>
<td>glxgears</td>
<td>1775 FPS</td>
<td>3900 FPS</td>
<td>54 (2.2x)</td>
</tr>
<tr>
<td>TRex</td>
<td></td>
<td></td>
<td>2.3 (.02x)</td>
</tr>
</tbody>
</table>
Takeaways

- Memory bandwidth requirements for graphics workloads can be astronomical.
- Don’t assume texture compression is the end of the bandwidth story.
- Modifiers “modify” the framebuffer’s pixel layout.
- Lossless compression reduces bandwidth, not size
  - Many GPUs support this transparently
- Hardware compositing is great.
- Getting features like this plumbed through can easily be a multi-year effort.
- Haiku isn’t supported :/
Thank Yous

Platinum Level
Kristian Høgsberg, Google
Daniel Stone, Collabora

Gold Level
Rob Clark, Red Hat
Jason Ekstrand, Intel
Ville Syrjälä, Intel
Daniel Vetter, Intel

Liviu Dudau, Arm Ltd
Eric Engestrom, Imagination Technologies
Varad Gautam, Collabora
Topi Pohjolainen, Intel
Lucas Stach, Pengutronix
Emil Velikov, Collabora
Chad Versace, Google
Tomeu Vizoso, Collabora
Q&A
(not about EGLStreams)