OpenGL Bindless Extensions

Jeff Bolz

Overview

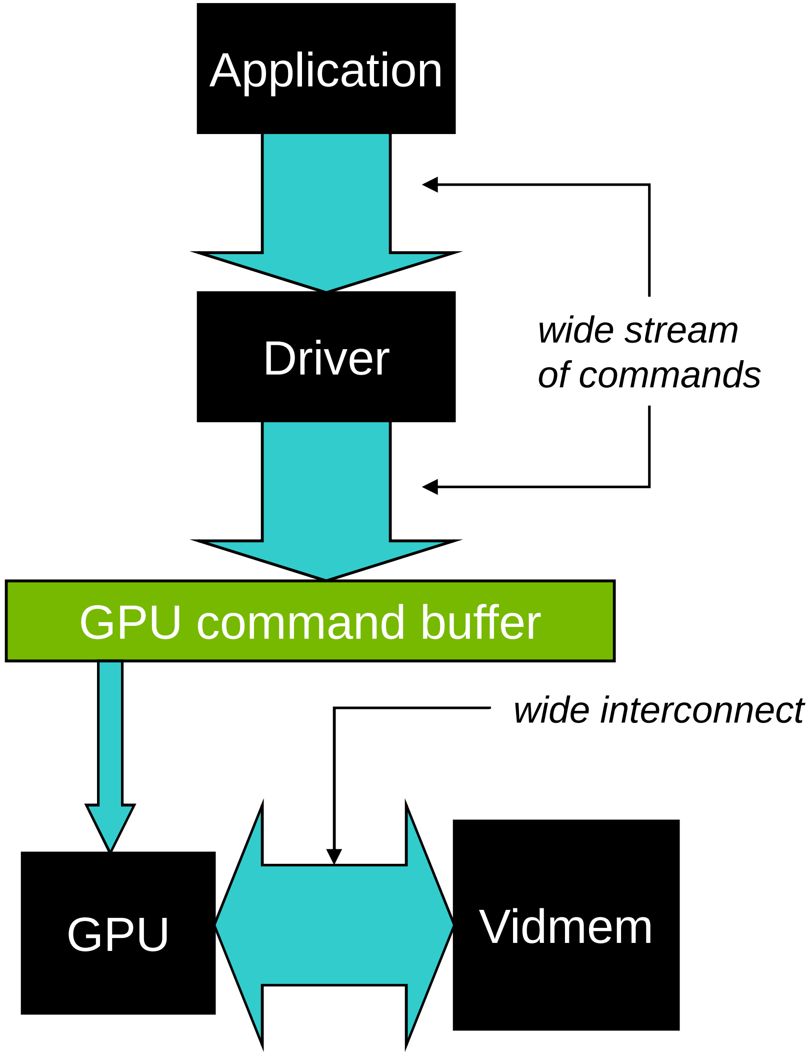
- Explain the source of CPU bottlenecks, past and present
- Show how new extensions alleviate these bottlenecks
  - GL_NV_shader_buffer_load
  - GL_NV_vertex_buffer_unified_memory

- Goal: Reduce the CPU overhead of launching a batch of geometry
- Allow more interesting and varied content by increasing the number of draw calls per frame
  - Imagine “Instancing” but with significant additional flexibility
- Akin to texture techniques that pack independent textures into a single object
  - Texture array – pack separate images as slices of an array. Choose between images with a single vertex attrib coordinate
  - Megatexture – pack tiles into a large virtual texture. Choose between images with clever page table techniques
  - But more flexible by still allowing separate objects

- Remove limitations on number/size of constant buffers

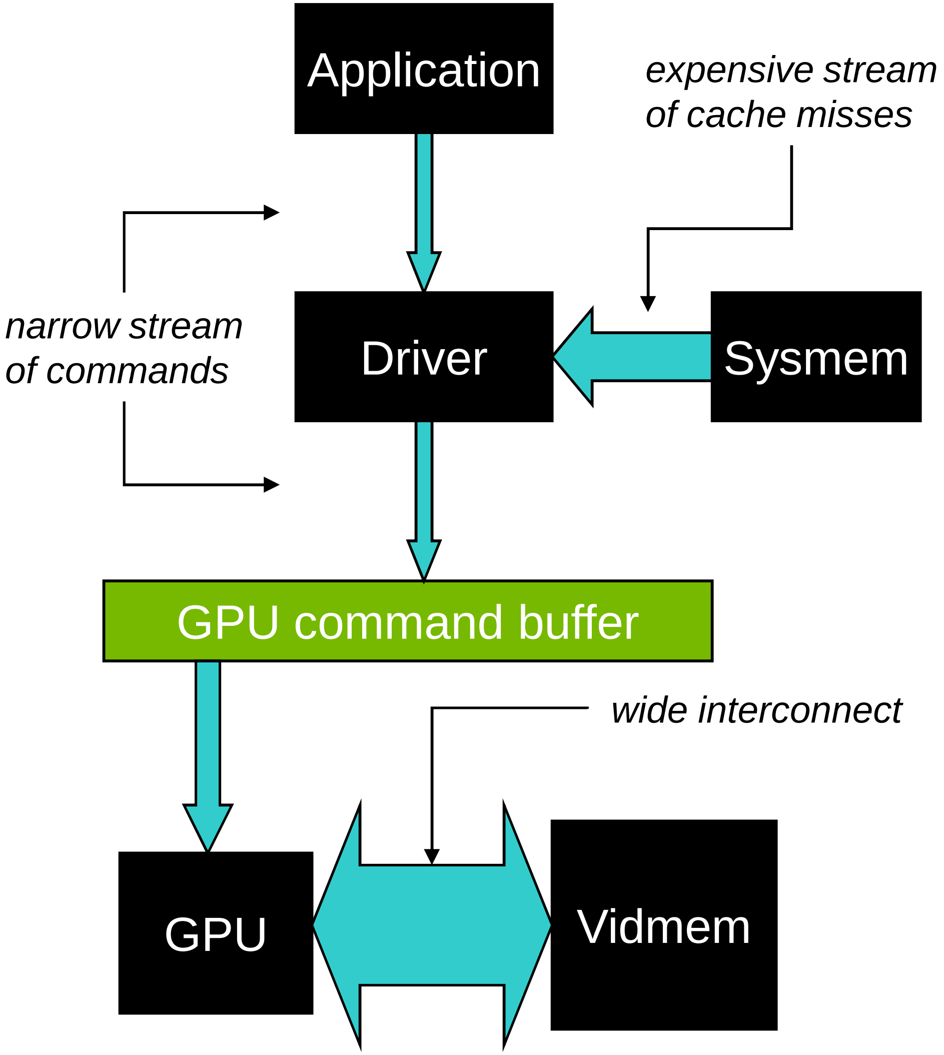
GL1.x Performance Characteristics

- A configurable state machine exposing low-level hardware state
- Lots of commands to set GL state
  - Transform and lighting: N lights, matrices, etc.
  - Per-pixel shading: N textures, texture environments
- LOTS of commands to specify vertex data
  - Immediate mode: Set each attribute individually, launch one vertex at a time
  - Classic vertex array: driver copies all vertex data each Draw
- Bottleneck: the API stream is too large



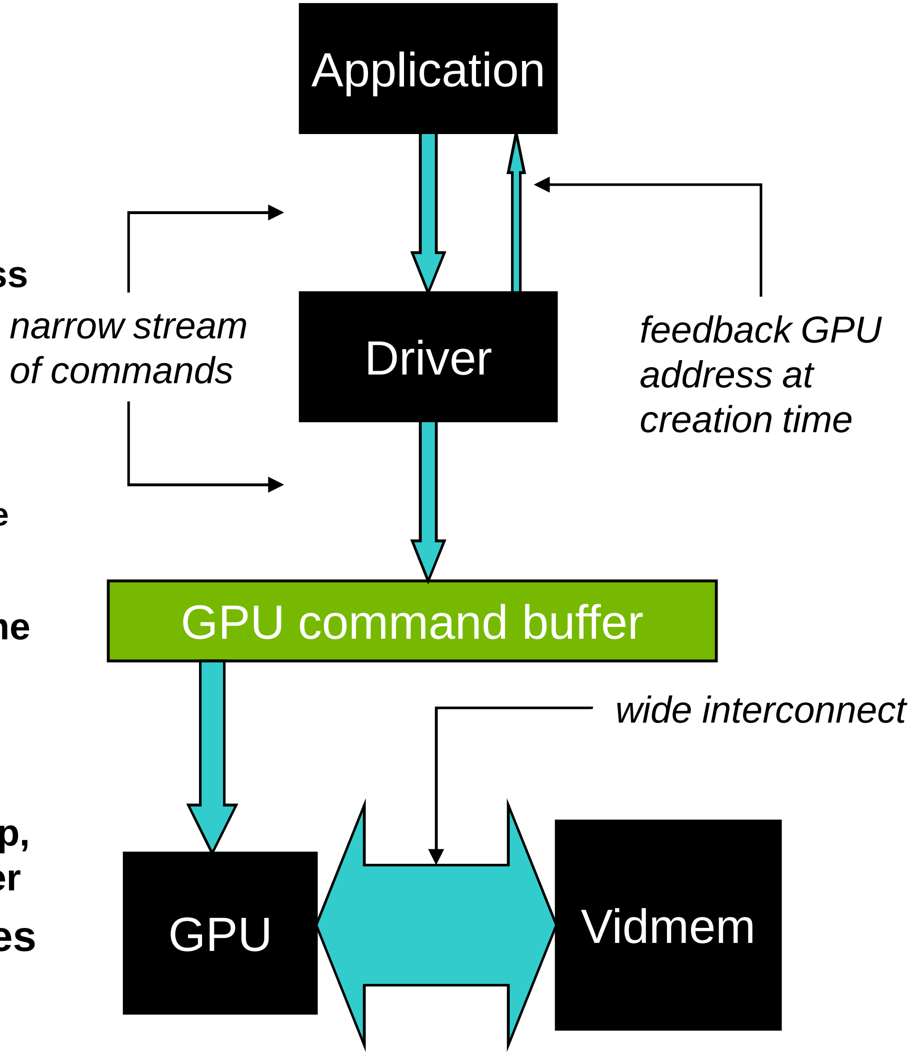
GL3.x Performance Characteristics

- Configurable state replaced with programmability and objects
  - Lighting, texenv -> shaders
  - Matrices, light values -> constant buffers
  - Immediate mode -> VBO
- Few commands to setup a rendering batch
  - Bind shaders, textures, constants, vertex buffers
- The API stream is now narrow, no longer the bottleneck
- Most commands (Binds) make the driver fetch object state from sysmem
  - The new bottleneck!
  - Hundreds of clocks per cache miss
  - Several Binds per Draw



Removing the Binds

- Still want to use objects, but more directly (by GPU address)
- Object creation time:
  - Application queries the GPU address
    - 64bit, static for object lifetime
  - Application informs driver to lock down the memory
    - MakeBufferResident
    - Amortized cost, rather than per-use
- Object use:
  - By GPU address rather than by name
  - As few commands as Binding
  - Driver no longer has to fetch GPU address from sysmem
  - Memory residency controlled by app, not handled worst-case by the driver
- The GL3.x bottleneck of cache misses on object use is gone!


Application
Driver
narrow stream of commands
feedback GPU address at creation time
GPU command buffer
wide interconnect
GPU
Vidmem

Vertex Buffer Unified Memory

- Goal: Reduce cache misses involved in setting vertex array state by directly specifying GPU addresses
- Set vertex attribute (and element array) GPU addresses directly
  - BufferAddressRangeNV(COLOR_ARRAY_ADDRESS_NV, 0, addr, length);
  - BufferAddressRangeNV(VERTEX_ATTRIB_ARRAY_ADDRESS_NV, i, addr, length);
  - BufferAddressRangeNV(ELEMENT_ARRAY_ADDRESS_NV, 0, addr, length);
- Decouple address from format
  - VertexFormatNV(size, type, stride);
  - ColorFormatNV(size, type, stride);
- Enable vertex/element GPU addresses explicitly
  - EnableClientState(VERTEX_ATTRIB_ARRAY_UNIFIED_NV);
  - EnableClientState(ELEMENT_ARRAY_UNIFIED_NV);
  - Unlike VBO where bound/latched buffers determine use

Example (Interleaved VBO)

```
for (i = 0; i < N; ++i) {
        BindBuffer(ARRAY_BUFFER, vboNames[i]);
        BufferData(ARRAY_BUFFER, size, ptr, STATIC_DRAW);
        GetBufferParameterui64vNV(ARRAY_BUFFER,
                                                   BUFFER_GPU_ADDRESS_NV,
                                                  &vboAddrs[i]);
        MakeBufferResidentNV(ARRAY_BUFFER, READ_ONLY);
}
```

Init (one time only)

```
EnableClientState(COLOR_ARRAY);
EnableClientState(VERTEX_ARRAY);
ColorFormatNV(4, UNSIGNED_BYTE, 20);
VertexFormatNV(4, FLOAT, 20);
EnableClientState(VERTEX_ATTRIB_ARRAY_UNIFIED_NV);
```

Format/Enables change (rare)

```
for (i = 0; i < N; ++i) {
    // point at buffer i
    BufferAddressRangeNV(COLOR_ARRAY_ADDRESS_NV,
                                          0, vboAddrs[i], size);
    BufferAddressRangeNV(VERTEX_ARRAY_ADDRESS_NV,
                                          0, vboAddrs[i]+4, size-4);
    DrawArrays(POINTS, 0, size/20);
}
```

Buffer change (frequent and efficient)

Easy to Port

- Old code:

```
foreach vertexattrib {
   BindBuffer(ARRAY_BUFFER, vbo name);
   VertexAttribPointer(attrib index, format, offset);
}
BindBuffer(ELEMENT_ARRAY, index buffer name);
DrawRangeElements(..., index offset);
```

- New code:

```
if (vertex format has changed) {   // rare
   // send VertexAttribFormat commands
}
foreach vertexattrib {
   BufferAddressRangeNV(VERTEX_ATTRIB_ARRAY_ADDRESS_NV,
                        attrib index, vbo gpu addr + offset, vbo size - offset);
}
BufferAddressRangeNV(ELEMENT_ARRAY_ADDRESS_NV,
                     0, index gpu addr, index size);
DrawRangeElements(..., index offset);
```

Perf Comparison

Old:

```
for (i = 0; i < N; ++i) {
   for (j = 0; j < 5; ++j) {
      BindBuffer(ARRAY, vboNames[x]);
      VertexAttribPointer(j, 4, FLOAT, 0, 4, 0);
   }
   BindBuffer(ELEMENT_ARRAY, vboNames[x]);
   DrawRangeElements(POINTS, ...);
}
```

N=100: 900K Draw/s

Cache Misses! → **N=10K: 400K Draw/s**

New:

```
for (i = 0; i < N; ++i) {
   for (j = 0; j < 5; ++j) {
     BufferAddressRangeNV(VERTEX_ATTRIB_ARRAY_ADDRESS_NV, j,
                                      vboAddrs[x], 100);
   }
   BufferAddressRangeNV(ELEMENT_ARRAY_ADDRESS_NV, 0,
                                      vboAddrs[x], 100);
   DrawRangeElements(POINTS, ...);
}
```

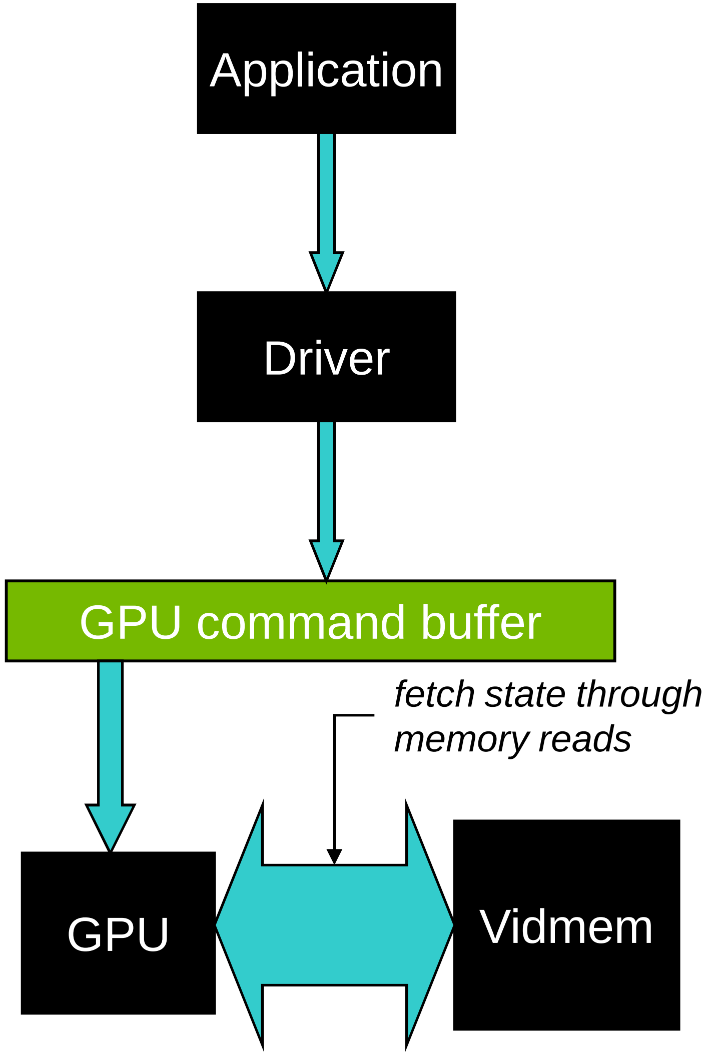
N=100: 3000K Draw/s

N=10K: 3000K Draw/s

7.5x speedup by removing cache misses!

Shader Buffer Load

- Allow shaders to fetch from buffer objects by GPU address
  - Exposed in the shading language as pointers
- No need to bind constant buffers between each draw
  - “Switch” dynamically, even at fine granularity
    - By immediate mode attrib (per batch)
    - By instance ID
    - By primitive ID
    - By vertex ID or vertex attributes
    - By varyings
- More flexible than indexable constant buffers
  - Can do dependent fetches, even across buffer objects
    - Can build complex data structures to be traversed in shaders
  - No limit on number of resident buffers
- Pull your state into shaders through cached memory reads rather than pushing through app/driver/commandbuffer



Easy to Port

- Old code:

```
(shader)
struct Material { vec4 color; ... };
bindable uniform Material mat;
void main() {
   gl_FrontColor = mat.color;
   ...
}

(app init)
loc = GetUniformLocation(pgm, “mat”);

(app render)
UniformBufferEXT(pgm, loc, buffer1);
Draw1();
UniformBufferEXT(pgm, loc, buffer2);
Draw2();
...
```

- New code:

```
(shader)
struct Material { vec4 color; ... };
in Material *mat;
void main() {
   gl_FrontColor = mat->color;
   ...
}

(app init)
loc = GetAttribLocation(pgm, “mat”);

(app render)
VertexAttribI2iEXT(loc, buf1Addr, buf1Addr>>32);
Draw1();
VertexAttribI2iEXT(loc, buf2Addr, buf2Addr>>32);
Draw2();
...
```

API Summary

- Query a GPU address and make a buffer resident
  - GetBufferParameterui64vNV(target, BUFFER_GPU_ADDRESS, &addr);
  - MakeBufferResident(target, READ_ONLY);
- Vertex Format functions, similar to existing VertexPointer functions
  - VertexAttribFormatNV(index, size, type, normalized, stride);
- Set GPU addresses for vertex attribs and element arrays
  - BufferAddressRangeNV(pname, index, address, length);
- Set pointer uniforms
  - Uniformui64NV(int location, uint64EXT value);
- Assembly LOAD instruction
  - LOAD.F32X4 result, address;
- Shader pointer types, enabling complex data structures:

```
struct LinkedListNode {
    vec4 color;
    LinkedListNode *next;
};
```