Gaijin's HLSL to Spir-V Compiler Stack Evolution
Tiemo Jung, Gaijin Entertainment
Basic Compiler Pipeline

- Shader Script
  - Front End
    - HLSL
      - FXC
        - DirectX Shader Blob
      - HLSL2GLSL
        - glsl optimize
          - OpenGL Shader Blob
      - HLSL to Spir-V
        - Vulkan Shader Blob
Compiler: HLSL2GLSL

- HLSL2GLSL and glsl optimize
  - Taken from OpenGL shader compiler
  - Inherited required hacks and workarounds in shaders for HLSL2GLSL to work properly

- Interface Patching
  - Pattern matching to find interface variables in GLSL source
  - Generated global uniform block out of loose uniforms
  - Added necessary layout attributes for Vulkan compatibility

- Shader Header
  - Small header with SPIR-V shader hash and resource slot usage mask
  - Evolved over time to meet the needs of the renderer
Compiler: HLSLcc

- **FXC**
  - Still most commonly used to compile HLSL
  - No longer any hacks in shaders because of HLSL2GLS
  - Decent optimized DXBC
  - Stuck with shader model 5

- **HLSLcc**
  - Started with James Jones version
  - Moved over to Unity version
  - Both versions needed fixes to work properly with our shaders
  - Extended both to fit our needs
  - In DXBC instructions dictate how registers are interpreted, this lead to guessing and awkward shader code with lots of bit casts
Compiler: DXC

- **DXC**
  - Replacement for FXC targeting DX12
  - Open source
  - Includes SPIR-V output (spiregg - maintained by google)
  - Modern shader features like shader model 6 and ray tracing

- **Gaijin SPIR-V Tools**
  - More later

- **zlib**
  - Will be replaced with zstd or lz4
Atomic Counter: Embedded

- Embedded at the top of the storage buffer
- Easy to implement in HLSLcc compiler pipeline
- Much more complex in SPIR-V
  - Clone and modify structure type
  - Patch load, store, access chain and atomic operations with new target or new indices
  - Remove counter buffer and any left over types
- Easy to implement in renderer
  - Allocate more memory to store counter
  - Have two views, with and without counter
- Wastes memory because of padding requirements (see minStorageBufferOffsetAlignment)
- Limited to one counter per buffer

```cpp
buffer StorageBufferWithCounter {
  int counter; // embedded by shader compiler
  StorageBufferElement data[];
};
```
**Atomic Counter: Global Indexed**

- Arbitrary amount of counters per buffer
- Counters are tightly packed in one global buffer
- Reset to 0 is just handing out an unused counter index and reset old counters when it is convenient
- Requires 2 resource slots if used
- Not implemented yet
- Performance characteristics yet unknown

```
uniform CounterIndexStore {
    int indexForCounter;
};
buffer CounterStore {
    int counterStore[];
};
buffer DataStore {
    StorageBufferElement data[];
};
void main() {
    int index = atomicAdd(counterStore[indexForCounter], 1);
    data[index] = computeData();
}
```
Pipeline Layout: Fixed

- No need to manage layouts
- Easy to implement in renderer and compiler
- Waste of descriptor memory
- Does not scale with growing and changing requirements
- Easy to run into HW limits even if never actually used
- Not suitable for compute, as layouts vary a lot

<table>
<thead>
<tr>
<th>Set</th>
<th>Binding</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Uniform Buffer Vertex Shader</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Uniform Buffer Fragment Shader</td>
</tr>
<tr>
<td>1</td>
<td>0 - 3</td>
<td>Sampled Textures Vertex Shader</td>
</tr>
<tr>
<td></td>
<td>4 - 19</td>
<td>Sampled Textures Fragment Shader</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set</th>
<th>Binding</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Uniform Buffer Vertex Shader</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Uniform Buffer Fragment Shader</td>
</tr>
<tr>
<td>1</td>
<td>0 - 15</td>
<td>Sampled Textures Fragment Shader</td>
</tr>
<tr>
<td>2</td>
<td>0 - 3</td>
<td>Sampled Textures Vertex Shader</td>
</tr>
<tr>
<td>3</td>
<td>0 - 7</td>
<td>Storage Images Fragment Shader</td>
</tr>
</tbody>
</table>
Pipeline Layout: Compacted (per Stage)

- Allows mix and match of resources as needed
- HW limits not a problem except on very limited targets
- Sharing of SPIR-V when shader only differs in DX resource binding slot
- Each unique descriptor set has its own descriptor heaps to avoid fragmentation
- Used for all shader stages uniformly
- Easy to extent
- Needs management of descriptor layouts

<table>
<thead>
<tr>
<th>Table Index</th>
<th>DX Mapping</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 7</td>
<td>b0 - b7</td>
<td>Uniform Buffers</td>
</tr>
<tr>
<td>8 - 39</td>
<td>t0 - t31</td>
<td>Sampled Images</td>
</tr>
<tr>
<td>40 - 71</td>
<td>t0 - t31</td>
<td>Sampled Images With Comparison</td>
</tr>
<tr>
<td>72 - 103</td>
<td>t0 - t31</td>
<td>Buffer Images</td>
</tr>
<tr>
<td>104 - 135</td>
<td>t0 - t31</td>
<td>Read Only Storage Buffers</td>
</tr>
<tr>
<td>136 - 143</td>
<td>u0 - u7</td>
<td>Storage Images</td>
</tr>
<tr>
<td>144 - 151</td>
<td>u0 - u7</td>
<td>Storage Buffer Images</td>
</tr>
<tr>
<td>152 - 159</td>
<td>u0 - u7</td>
<td>Storage Buffers</td>
</tr>
<tr>
<td>160 - 167</td>
<td>u0 - u7</td>
<td>Storage Buffers With Counter</td>
</tr>
<tr>
<td>168 - 199</td>
<td>t0 - t31</td>
<td>Raytrace Acceleration Structure</td>
</tr>
</tbody>
</table>
Pipeline Layout: Compacted (per Stage)
Gaijin SPIR-V Tools: Rationale

• Why own SPIR-V Tools?
  - Most other SPIR-V tools and libraries are designed to be used for their special purpose and are closed off for other uses
  - We want a thin decoder for fast, easy and safe peeking at SPIR-V blobs
  - We also want a full suit to load, modify and store SPIR-V in our compiler chain

• Bulk is auto generated with a Python script from the grammar JSON spec files, inspired by the Khronos SPIR-V Tools.
  - Basic information to print and decode SPIR-V binary format
  - Pattern matching to generate AST node and properties for each SPIR-V instruction
    - Required some handcrafted overrides for a few instructions
Gaijin SPIR-V Tools: Components

- **Decoder**
  - Basic SPIR-V binary format decoder
  - Converts word stream in properly typed commands to the user
  - Does not allocate any memory

- **Module Builder**
  - Handwritten
  - Interfaces with auto generated AST nodes and properties
  - In memory representation of a SPIR-V module
  - Provides basic tools to build, inspect and modify the SPIR-V module

- **Loader**
  - Bridge from Decoder to Module Builder
  - Auto generated
Gaijin SPIR-V Tools: Components

• **Writer / Encoder**
  - Encodes Module Builder state into SPIR-V binary representation
  - Auto generated

• **Passes**
  - Map HLSL Semantics to location indices
  - Merge separate Samplers and Images to combined Sampled Images
  - Transform Buffer Counters
  - Descriptor packing and Shader Header compilation
  - Forking shaders with different extension implementations?
  - Target specific optimizations?