20180316 - Good Vulkan Binding

Simple recommendation for good Vulkan performance on both AMD and NVIDIA. I'm writing this to help people avoid stepping on the land mines that are disguised in the API as edible cakes ...

Push Descriptor = Getting Kicked In The Shorts
Classic GCN hardware has 16 special SGPRs (scalar registers) called USER-DATA-SGPRs which can get preloaded with "stuff" before a wave starts to execute. This is "push" data which doesn't require a possible long latency load at run-time. The driver uses a variable amount for itself depending on graphics state, etc. *So I suggest assuming conservatively best case the app might get 12 USER-DATA-SGPRs.*

> 8 SGPRs for an image descriptor.
> 4 SGPRs for a sampler descriptor.
> 4 SGPRs for a buffer descriptor (SSBO, UBO, CBV, etc).
> 2 SGPRs for a DYANMIC descriptor (they are passed as 64-bit pointers, with no range, same as root-table CBV).
> 1 SGPR for a set pointer (they are locked to a fixed 32-bit virtual address region).
> 1 SGPR for a push constant.

Hopefully this data makes it clear that there is no point in attempting to stuff say image descriptors in USER-DATA-SGPRs, as just one image paired with one sampler and USER-DATA-SGPRs are spent. Anything not fitting in USER-DATA-SGPRs gets "spilled" and thus loaded via SMEM at some point during execution.

Push descriptors are NOT like DX12 root table entries.

Rather push descriptors are "slow as dirt" GL/GLES style binding points inside Vulkan. With push descriptors you get a single hidden driver managed and versioned set with a GLES-like limit of binding points. Since this 'hidden set' is setup to be larger than what could be expected to fit in USER-DATA and is changed on each draw, this introduces a draw to draw dependency ... the exact kind of serialization which limits performance scaling!

Not Getting Kicked In The Shorts

On GCN it is easy to go fast. Use one DYNAMIC SSBO to point to the base (offset=0) of your giant vkDeviceMemory allocation for buffer data, ie the allocation you sub-allocate your buffer data from. Then use "push constants" for the offsets of sub-allocated buffers that a shader needs to access. I use immediate-indexed offsets for anything "per-frame" or "per-view", and would only resort to "push constants" for "per-draw". Simple, elegant. The shader will pass in the DYNAMIC SSBO as a 64-bit pointer, and the push constants as 32-bit offsets. Plenty of USER-DATA-SGPRs to spare.

This is the "bind-everything" once per command buffer model that I use.

Avoiding Getting Kicked In The Shorts on NVIDIA As Well
If the "offset" is to data larger than 64 KiB then an SSBO is needed on NVIDIA as well. If the data is less than 64 KiB, then NVIDIA asks for a sparate UBO binding (instead of a push constant) to workaround their hardware missing support for fast access to post-DX11-sized constant buffers. *So the portable compromise is to pass in a DYNAMIC UBO for data smaller than 64 KiB.* Clearly to avoid the kick in the shorts, be smart and only use as few DYNAMIC UBOs as possible.

Binding Aliasing

GLSL has a very strange way of implmenting resource typecasting: via aliasing of bindings. This is not just for 32/64/128-bit memory accesses, but also for memory qualifiers. Accessing a 256 MiB SSBO as anything important is as simple as doing this,

```
#define SIZ_ (256*1024*1024)
layout(set=0,binding=0,std430)                    buffer ssbo0_ {uint  bufA32[SIZ_/4];};
layout(set=0,binding=0,std430)                    buffer ssbo1_ {uvec2 bufA64[SIZ_/8];};
layout(set=0,binding=0,std430)           readonly buffer ssbo2_ {uint  bufR32[SIZ_/4];};
layout(set=0,binding=0,std430)           readonly buffer ssbo3_ {uvec2 bufR64[SIZ_/8];};
layout(set=0,binding=0,std430)           readonly buffer ssbo4_ {uvec4 bufR128[SIZ_/16];};
layout(set=0,binding=0,std430) coherent writeonly buffer ssbo5_ {uint  bufW32[SIZ_/4];};
layout(set=0,binding=0,std430) coherent writeonly buffer ssbo6_ {uvec2 bufW64[SIZ_/8];};
layout(set=0,binding=0,std430) coherent writeonly buffer ssbo7_ {uvec4 bufW128[SIZ_/16];};
layout(set=0,binding=0,std430) volatile writeonly buffer ssbo8_ {uint  bufV32[SIZ_/4];};
layout(set=0,binding=0,std430) volatile writeonly buffer ssbo9_ {uvec2 bufV64[SIZ_/8];};
layout(set=0,binding=0,std430) volatile writeonly buffer ssboA_ {uvec4 bufV128[SIZ_/16];};
```

Notice the same "binding=0" on everything, aka explicit binding aliasing.

Now some simple usage examples,

```
// Write through to a coherent L2$, to avoid needing cache flush in later pass to do a cached read.
bufW32[pointer>>2] = x;

// Read a 128-bit chunk of data.
// If 'pointer' is known by the compiler to be dynamically uniform this can go through SMEM.
// The 'readonly' memory qualifier is what enables that behavior in the current driver.
uvec4 b = bufR128[pointer>>4];

// Do an atomic on a 64-bit value.
z = atomicMin(bufA64[pointer>>3], y);

// Store out something for indirect dispatch, without needing a later cache flush.
bufV128[pointer>>4] = indirectArgs;
```

So the binding aliasing name becomes the thing which "typecasts" between different memory access granularity and cache controls. It would also be possible to setup macros to avoid the visible bit shift '>>' if so desired: `#define BUF_V128(x) bufV128[(x)>>4]` then `BUF_V128(x) = indirectArgs`. Also a compiler properly optimized could trivially remove the bit shift since buffer loads naturally take byte offsets in GCN. Lastly one can freely typecast without conversion between int and float via `floatBitsToUint()` and `uintBitsToFloat()`.