Efficient Rendering with DirectX 12 on Intel Graphics

Andrew Lauritzen
Michael Apodaca
INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS, NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A “Mission Critical Application” is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL’S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS’ FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice.

All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.

Intel processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Any code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel’s internal code names is at the sole risk of the user.

Intel product plans in this presentation do not constitute Intel plan of record product roadmaps. Please contact your Intel representative to obtain Intel’s current plan of record product roadmaps.

Performance claims: Software and workloads used in performance tests may have been optimized for performance only on Intel® microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.Intel.com/performance

Iris™ graphics is available on select systems. Consult your system manufacturer.

Intel, Intel Inside, the Intel logo, Intel Core and Iris are trademarks of Intel Corporation in the United States and other countries.
Decoder Cheat-sheet

✦ Intel 4\textsuperscript{th} Generation Core (i3/i5/i7 4xxx)
  – Code-named “Haswell”, Gen 7.5 GPU architecture
  – Intel HD Graphics 4400/4600/5000
  – Intel Iris Graphics 5100, Iris Pro Graphics 5200, ...

✦ Intel 5\textsuperscript{th} Generation Core (i3/i5/i7 5xxx, Core M 5xxx)
  – Code-named “Broadwell”, Gen 8 GPU architecture
  – Intel HD Graphics 5300/5500/6000
  – Intel Iris Graphics 6100, ...
Why DirectX 12?

✦ Performance
  – Improve CPU bound games
  – Improve multi-core scaling

✦ Power
  – Improve performance on power-constrained platforms
  – Improve heat and battery life

✦ How?
  – Reduce CPU overhead of rendering
Most GPU vendors have complex drivers
  – Do lots of fancy optimizations on the fly
  – Costs CPU, but makes the GPU run faster
    ✦ That’s ok, reviews compare GPUs using fast CPUs! 😊

Drivers spawn threads that conflict with application
  – Driver thread often consumes an entire core by itself
  – Plus another core for the game submission thread
  – Minimal multithreading beyond these two threads
CPU/GPU Optimization Tradeoff

- Has recently become far more serious with SoCs
  - Even if not “CPU bound”, CPU/GPU share power
  - More CPU load => less GPU power/performance
- Complex CPU optimizations are not a good idea...
  - Tax on all applications, even well-optimized ones
  - CPU work can take more power than it saves on the GPU!
  - Leads to lower overall performance
To address this, Intel wrote a much thinner DX11 driver
  – Introduced with Haswell

Big benefit to well-written applications
  – But does far less work to make poor ones run well
  – i.e. no redundant state elimination, minimal state-based shader recompiles, etc.

Still unavoidable CPU overhead due to API design
  – DirectX 12 addresses this
DirectX 12 on Intel

- Already significantly lower CPU overhead
- Large increase in power efficiency
  - Power saved on the CPU can be given to the GPU
  - Applications can both run faster and use less power
- Additional GPU optimization opportunities
  - i.e. stuff that we had to drop in the thinner driver
  - Pipeline state objects give the driver more context
DirectX 12 can significantly reduce CPU power or improve performance

less power @ same performance

higher performance @ same power
Agenda

✧ Commands and state
✧ Memory
✧ Resource binding
Commands and State
State in DirectX 11

- DirectX 11 context is “stateful”
  - State grouped into moderately sized chunks
  - Rasterizer, depth/stencil, blend, etc.

- Groupings do not always map perfectly to hardware
  - Ex. DirectX blend state != GPU blend state
  - Driver optimizations based on blend state + pixel shader
API functions cause one or more GPU commands to be added to the command buffer

Some GPU commands are deferred or conditional
  - Often lazily added at the next draw call

```c
deviceCtx->aaa();
deviceCtx->bbb();
deviceCtx->ccc();
deviceCtx->ddd();
```
At some point, the driver decides to commit the command buffer
– If the command buffer fills, max buffered frames, Flush(), etc.
It’s passed to kernel mode and GPU addresses are patched
Then, it’s submitted to the GPU
DirectX 11 Deferred Contexts

- Limited parallelism with a single context
- Deferred contexts do not address the problems
  - CPU performance and cache issues with transient objects
  - State mismatch and lazy state setting
  - Inherited internal states
  - MAP_DISCARD renaming, hazard tracking, etc.
  - Non-trivial patching happens at submission time
- Result: more overhead and limited parallelism
Each thread has its own command list and memory
- Fully independent
- Use ~1 command list/thread

Command lists are submitted to the GPU in arbitrary order
- Minimal driver work done at submission time
- Submit all command lists in a single API call where possible
While those commands are in flight, can record new commands
  – Can reuse command lists
  – Must use different memory

When GPU finishes with memory, it can also be reused
  – App handles synchronization
  – Typical to put fence at frame boundaries
  – Always reuse allocators!
State in DirectX 12

- Immutable, monolithic pipeline state objects (PSOs)
  - Single object captures as much state as possible
  - Much lower chance of missing driver context
  - Allows link-time optimizations on shaders

- No state inheritance between direct command lists
  - No API state or internal state inheritance (renaming, etc.)
  - Explicit barriers to handle hazards and resource transitions
Pipeline State Objects

- Create PSOs at initialization time
  - Multithread your initialization/PSO creation code!
  - Use PSO “libraries”

- PSO changes are usually fairly cheap
  - Minimal CPU cost, some GPU cost

- Some state sorting is still desirable
  - Turning shader stages on/off can cause pipeline stalls
Bundles

- Reusable command lists to further lower CPU overhead
- Some minimal state inheritance is allowed
  - Some patching may occur at submission time
  - If you don’t need to inherit something, set it (again) in the bundle
- Overhead is already very low in DirectX 12
  - Need ~10+ draws to make bundles a win on Haswell/Broadwell
  - Only consider bundles if you have lots of static draws that can't reasonably be combined (via instancing or similar)
  - Don’t add any GPU overhead/indirections to enable bundles!
DirectX 12 expands on DrawIndirect/DispatchIndirect

Command Signature
- Indirect Argument Buffer Format
- Draw/Dispatch calls
- Resource Bindings

Indirect Argument Buffer
- Dynamic parameters

Count Buffer
Execute Indirect on Haswell/Broadwell

- Internal Compute Shader Patches CommandList
  - Compiled at CreateCommandSignature
- If no resource bindings, then no compute shader (legacy)
 DirectX 12 exposes multiple “queues” to application
  – Graphics/compute, compute-only, copy, etc.

 Graphics and compute are not simultaneous on Intel
  – Using separate queues is not a performance benefit
  – Consider doing both on the main queue

 There is a simultaneous copy engine
  – ... but it has fairly low throughput
  – Driver may implement large copies using the 3D engine
Memory
Previous APIs (ex. DirectX 11) hide a lot of details
- GPU physical memory residency (if applicable)
- GPU memory addressing (virtual, physical)

OS/driver manage residency and addressing
- Ensures command buffers do not exceed hardware resources
- Track referenced allocations, ensure resident
- Allocate and patch GPU addresses
- Major source of CPU overhead!

Applications try not to over-commit “GPU memory”
 GPUs must be made “resident”
 – Memory referenced by the GPU must be made “resident”

 No dedicated video memory on Intel processors
 – “Resident” resources are allocated out of DRAM

 OS uses up to 45% of DRAM for graphics applications
 – Ex. 1.8GB on a 4GB system, 3.6GB on an 8GB system, ...
 – Global limit across the system, not per-process
 – Rest is reserved for regular CPU/OS use
Memory Residency Best Practices

- Allocations are initially made resident
  - Resource creation will fail if residency budget is exceeded
- OS will request that background apps trim residency
  - Misbehaved applications will be suspended from rendering
  - i.e. their GPU work will not be scheduled/make progress
- Be a good citizen; provide a good user experience
  - Handle allocation failures and trim requests gracefully
  - Evict idle resources, trim streaming pools, remove detailed mips, drop quality settings, etc.
GPU Virtual Addresses in WDDM 2.0

- Directly exposes per-process GPU virtual addresses
  - Can do pointer arithmetic, store in data structures, etc.
  - GPU virtual addresses allocated at resource allocation
  - Guaranteed to remain at the same address until release
  - Eliminates physical address patching overhead

- Haswell has a limited GPU virtual address space (~2GB)
  - Subtly different than residency
Applications typically optimize for this.
Haswell Memory

- Haswell is limited by this
- GPU virtual address
- GPU Page Table
- CPU virtual address
- CPU Page Table
- CPU virtual address
- GPU virtual address
- DRAM (DDR)

Copyright © 2015, Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.
Not quite the same as limited GPU physical memory
  – Limit on the amount of DRAM visible to the GPU at once
  – All GPU-visible memory counts (upload/read-back heaps, ...)
  – Even non-resident memory counts

In theory, managing only requires GPU page table edits
  – But GPU virtual addresses are visible in DirectX 12
  – Must reallocate/copy data

GPU VA exhaustion will fail at resource allocation
  – Again, please handle this gracefully! 😊
Good news: no longer an issue on Broadwell
  – Large GPU virtual address space (same as CPU)

Memory-related public service announcement:
  – Don’t make/ship 32-bit (CPU) D3D12 applications!
  – Even if it works today...
  – Thank me later 😊
Resource Binding
Resource Descriptors

- Resources views are effectively just a small structure
  - Metadata and a pointer to memory (usually ~32-64 bytes)
  - Stuff like texture dimensions, format, layout, etc.

- Direct3D 12 directly exposes these “descriptors”
  - Independent from the actual memory they reference
  - Can be created/copied/etc. freely
  - Application must ensure no dangling pointers
Resource Descriptors

✧ **Not** an API object – manipulated directly by application
  – Descriptor size query-able by application
  – Can be created at any time; free-threaded API call

✧ **Descriptors are put into** “heaps” (arrays)
  – CBVs, SRVs and UAVs can be mixed in one heap
  – Samplers in a separate heap
  – Can have one or more of each type, GPU visible or CPU only

✧ **Changing heaps is expensive** (pipeline flush)
  – Ideally use a single heap of each type (sampler, CBV/SRV/UAV)
  – Exception: changing heaps at command list boundary is “free”
Descriptors Example

D3D12_UNORDERED_ACCESS_VIEW_DESC uavDesc = { ... };
cmdList->CreateUnorderedAccessView(res, desc, [uavHandle])

D3D12_CONSTANT_BUFFER_VIEW_DESC cbvDesc = { ... };
cmdList->CreateConstantBufferView(res, cbvDesc, [cbvHandle]);
...

Descriptor Heap

UAV
CBV
SRV
CBV
SRV
SRV
Root Signature

- Think of it like a function signature for your shader(s)
- Defines *parameters* and how they map to shader inputs
  - Root constants (data: zero indirections)
  - Root descriptors (pointer to data: one indirection)
  - Descriptor tables (pointer to descriptors: two indirections)
- Each parameter can be visible to one or more shader stages
- Parameters are “versioned” by implementation/hardware
  - This is the single place the “stream” of versions are managed
  - Maximum size is very small to avoid abuse
Root Parameter Indirections

Root Signature

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Root Constants</td>
</tr>
<tr>
<td>1</td>
<td>Root Descriptor</td>
</tr>
<tr>
<td>2</td>
<td>Descriptor Table</td>
</tr>
</tbody>
</table>

Memory

Descriptor Heap

- ... 
- UAV 
- CBV 
- ...
Root Constants

- Pass a small number of constants directly to shaders
  - Bound to shader as a single constant buffer
- Useful for simple indirections; draw ID, material ID, etc.
  - Avoids creating versioned memory, descriptor, heap, etc
  - Shader can use to look up into arbitrary data structures
Root Descriptors

- Stores a single descriptor directly as a root parameter
  - No need to burn through descriptor heap space
  - Most useful for a descriptor that changes ~ every draw

- Can only reference “raw data”
  - Only buffer resources (CBVs, SRVs/UAVs of buffers)
  - No type conversions (i.e. only float/uint/sint components)
  - i.e. it’s just a pointer to memory
  - *No out of bounds checking!* Don’t do bad stuff 😊
Descriptor Tables

- Maps continuous range of descriptors to shader slots
  - Can mix SRVs, UAVs, and CBVs arbitrarily
- Multiple descriptor tables can point to disjoint ranges
  - Ex. Use separate parameters for different update frequencies
  - Per-scene, per-material, per-instance, per-draw, etc.
  - Similar to constant buffers, now also for the descriptors too
## Root Signature Example

### Root Signature

<table>
<thead>
<tr>
<th></th>
<th>Descriptor Table</th>
<th>t1</th>
<th>b1</th>
<th>t4</th>
<th>t5</th>
<th>u0</th>
<th>b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
D3D12_DESCRIPTOR_RANGE Param0Ranges[3];
Param0Ranges[0].Init(D3D12_DESCRIPTOR_RANGE_SRV, 1, 1); // t1
Param0Ranges[1].Init(D3D12_DESCRIPTOR_RANGE_CBV, 1, 1); // b1
Param0Ranges[2].Init(D3D12_DESCRIPTOR_RANGE_SRV, 2, 4); // t4-t5

D3D12_DESCRIPTOR_RANGE Param1Ranges[2];
Param1Ranges[0].Init(D3D12_DESCRIPTOR_RANGE_UAV, 1, 0); // u0
Param1Ranges[1].Init(D3D12_DESCRIPTOR_RANGE_CBV, 1, 2); // b2

// Visibility to all stages allows sharing binding tables
D3D12_ROOT_PARAMETER Param[2];
Param[0].InitAsDescriptorTable(3, Param0Ranges, D3D12_SHADER_VISIBILITY_ALL);
Param[1].InitAsDescriptorTable(2, Param1Ranges, D3D12_SHADER_VISIBILITY_ALL);
```
Root Signature Example

Root Signature

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Descriptor Table</td>
<td>t1</td>
<td>b1</td>
<td>t4</td>
</tr>
<tr>
<td>1</td>
<td>Descriptor Table</td>
<td>u0</td>
<td>b2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Shader Resource View</td>
<td>t0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>uint4 Constant</td>
<td>b0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... 

Param[2].InitAsShaderResourceView(1, 0); // t0

Param[3].InitAsConstants(4, 0); // b0 (4x32-bit constants)
Root Signature Example

Root Signature

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Descriptor Table</td>
<td>t1</td>
<td>b1</td>
<td>t4</td>
</tr>
<tr>
<td>1</td>
<td>Descriptor Table</td>
<td>u0</td>
<td>b2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Shader Resource View</td>
<td>t0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>uint4 Constant</td>
<td>b0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...  

Param[2].InitAsShaderResourceView(1, 0); // t0  

Param[3].InitAsConstants(4, 0); // b0 (4x32-bit constants)
## Root Signature Example (HLSL)

### Root Signature

<table>
<thead>
<tr>
<th></th>
<th>Descriptor Table</th>
<th>0</th>
<th>t1</th>
<th>b1</th>
<th>t4</th>
<th>t5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Descriptor Table</td>
<td>1</td>
<td>u0</td>
<td>b2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Shader Resource View</td>
<td>2</td>
<td>t0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>uint4 Constant</td>
<td>3</td>
<td>b0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```hlsl
DescriptorTable(SRV(t1), CBV(b1), SRV(t4, numDescriptors=2)),
DescriptorTable(UAV(u0), CBV(b2)),
SRV(t0),
RootConstants(b0, num32BitConstants=4)
```
Binding Example

Root Signature

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Descriptor Table</td>
<td>t1</td>
<td>b1</td>
<td>t4</td>
</tr>
<tr>
<td>1</td>
<td>Descriptor Table</td>
<td>u0</td>
<td>b2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Shader Resource View</td>
<td>t0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>uint4 Constant</td>
<td>b0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

cmdList->SetGraphicsRootDescriptorTable(0, [srvGPUHandle]);
cmdList->SetGraphicsRootDescriptorTable(1, [uavGPUHandle]);
### Root Signature

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Descriptor Table</td>
<td>t1, b1, t4, t5</td>
</tr>
<tr>
<td>1</td>
<td>Descriptor Table</td>
<td>u0, b2</td>
</tr>
<tr>
<td>2</td>
<td>Shader Resource View</td>
<td>t0, SRV</td>
</tr>
<tr>
<td>3</td>
<td>uint4 Constant</td>
<td>b0</td>
</tr>
</tbody>
</table>

```c
// Set descriptors
cmdList->SetGraphicsRootDescriptorTable(0, [srvGPUHandle]);
cmdList->SetGraphicsRootDescriptorTable(1, [uavGPUHandle]);

// Set constant buffer view
cmdList->SetGraphicsRootConstantBufferView(2, [srvCPUHandle]);
```
Binding Example

Root Signature

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Descriptor Table</td>
<td>t1</td>
<td>b1</td>
<td>t4</td>
</tr>
<tr>
<td>1</td>
<td>Descriptor Table</td>
<td>u0</td>
<td>b2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Shader Resource View</td>
<td>t0</td>
<td>SRV</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>uint4 Constant</td>
<td>b0</td>
<td>{1, 3, 3, 7}</td>
<td></td>
</tr>
</tbody>
</table>

cmdList->SetGraphicsRootDescriptorTable(0, [srvGPUHandle]);
cmdList->SetGraphicsRootDescriptorTable(1, [uavGPUHandle]);
cmdList->SetGraphicsRootConstantBufferView(2, [srvCPUHandle]);
cmdList->SetGraphicsRoot32BitConstants(3, {1, 3, 3, 7}, 0, 4);
Root constants implemented with “push constants”
- Buffer that hardware uses to prepopulate EU registers
- When EU thread launches, values are immediately available
- Can be a GPU performance win vs. loading buffer data

Root descriptors also use push constants
- Pointers passed as constants to the shader
- Data read through general memory path

Descriptor tables use “binding table” hardware
- Each descriptor binding requires one binding table slot
Haswell/Broadwell Descriptor Tables

Root Signature

<table>
<thead>
<tr>
<th></th>
<th>Descriptor Table</th>
<th>t1</th>
<th>b1</th>
<th>t4</th>
<th>t5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>u0</td>
<td>b2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

~2-12 reserved slots and render targets

<table>
<thead>
<tr>
<th>HLSL binding</th>
<th>u0</th>
<th>b2</th>
<th>t1</th>
<th>b1</th>
<th>t4</th>
<th>t5</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding table index (BTI)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Shader compiler
Emit proper BTIs

Driver runtime
Fill in binding tables
Haswell/Broadwell Descriptor Tables Example

Ring of Binding Tables

- UAV
- CBV
- SRV
- SRV
- ...

Surface state base address

- DWORD 0
- DWORD 1
- DWORD 2
- ...
- DWORD 7
- ...
- DWORD 16376
- DWORD 16377
- DWORD 16378
- ...
- DWORD 16384

User descriptors
Up to ~1 million, each
32 bytes (Gen7.5)
64 bytes (Gen8)

64KB
Haswell/Broadwell Descriptor Tables Example

- **Ring of Binding Tables**
  - 64KB
  - User descriptors
    - Up to ~1 million, each
    - 32 bytes (Gen7.5)
    - 64 bytes (Gen8)

- **Surface state base address**
- **Binding table pointer**

- **DWORD 0**
  - t1
- **DWORD 1**
  - b1
- **DWORD 2**
  - t4
- **DWORD 7**
  - ...
  - ...
- **DWORD 16376**
- **DWORD 16377**
- **DWORD 16378**
- **DWORD 16384**

- **64KB**
  - UAV
  - CBV
  - SRV
  - SRV
  - ...
### Haswell/Broadwell Descriptor Tables Example

#### Ring of Binding Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UAV</td>
<td>User Visible Access View (UAV) descriptors</td>
</tr>
<tr>
<td>CBV</td>
<td>Constant Buffer View (CBV) descriptors</td>
</tr>
<tr>
<td>SRV</td>
<td>Shader Resource View (SRV) descriptors</td>
</tr>
<tr>
<td>SRV</td>
<td>...</td>
</tr>
</tbody>
</table>

#### Surface State Base Address

<table>
<thead>
<tr>
<th>DWORD</th>
<th>Binding Table Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>t1</td>
</tr>
<tr>
<td>1</td>
<td>b1</td>
</tr>
<tr>
<td>2</td>
<td>t4</td>
</tr>
<tr>
<td>7</td>
<td>...</td>
</tr>
<tr>
<td>16376</td>
<td>...</td>
</tr>
<tr>
<td>16377</td>
<td>...</td>
</tr>
<tr>
<td>16378</td>
<td>...</td>
</tr>
<tr>
<td>16384</td>
<td>...</td>
</tr>
</tbody>
</table>

#### User Descriptors

- Up to ~1 million, each 32 bytes (Gen7.5)
- 64 bytes (Gen8)

---

*Other names and brands may be claimed as the property of others.*

Copyright © 2015, Intel Corporation. All rights reserved.
**Haswell/Broadwell Descriptor Tables Example**

<table>
<thead>
<tr>
<th>Ring of Binding Tables</th>
</tr>
</thead>
</table>
|                     ...
|                         UAV
|                         CBV
|                     ...
|                         SRV
|                         CBV
|                         SRV
|                     ...

- **User descriptors**
  - Up to ~1 million, each 32 bytes (Gen7.5)
  - 64 bytes (Gen8)

- **Surface state base address**

- **Binding table pointer**

- **Pipeline stall!**

---

Copyright © 2015, Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.*
Resource Binding Summary

- Minimize “types” of parameters changed in inner loop
  - Descriptor tables, samplers, root descriptors, root constants
  - Cost of changing 1 of type X ~ cost of changing all of type X

- Minimize # descriptors referenced by tables
  - Don’t leave dangling/unused descriptors in large ranges
  - Most important for root signatures used in inner loops
  - Future hardware will only cost # tables, not # descriptors
Define sampler parameters right in the root signature
– Or right in the shader with HLSL root signature language

No performance advantage on Haswell/Broadwell
– Driver places static samplers in the regular sampler heap
– Same as manually putting them there manually

Use them if they are convenient
– Performance should never be worse
DirectX 12 is a great fit for Intel hardware!
- Increased performance
- Increased power efficiency

Already supported today on Haswell and Broadwell
- Will get even better in the future
Questions?

✧ Follow @DirectX12 and @IntelSoftware
✧ Working on DirectX 12 on Intel?
  – andrew.t.lauritzen@intel.com, @AndrewLauritzen