DIRECTX ADVANCEMENTS IN THE MANY CORE ERA

Getting the most out of the PC Platform

System Architecture Trends

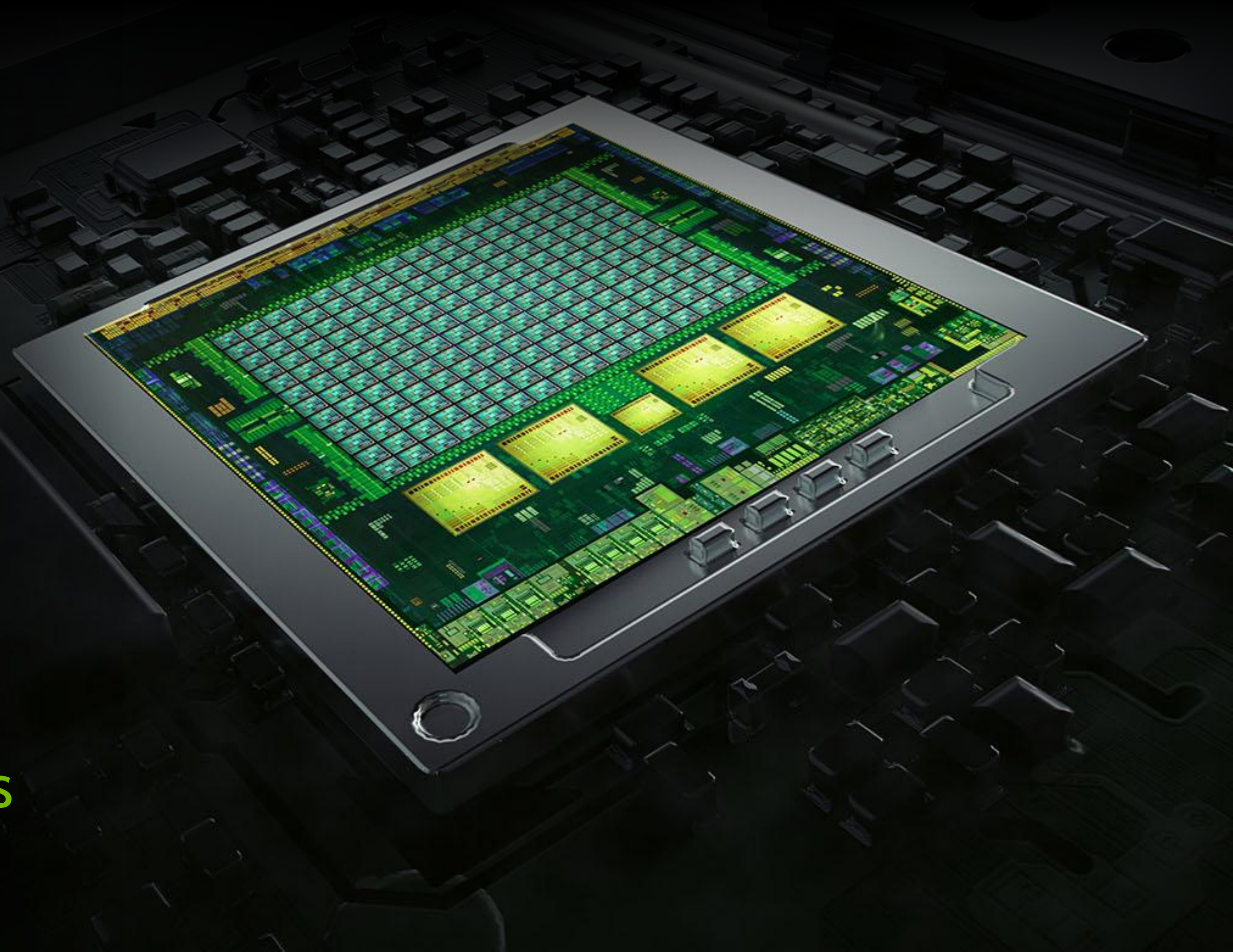
CPU Evolution

- From single core to multi-core
- Power wall limits frequency scaling

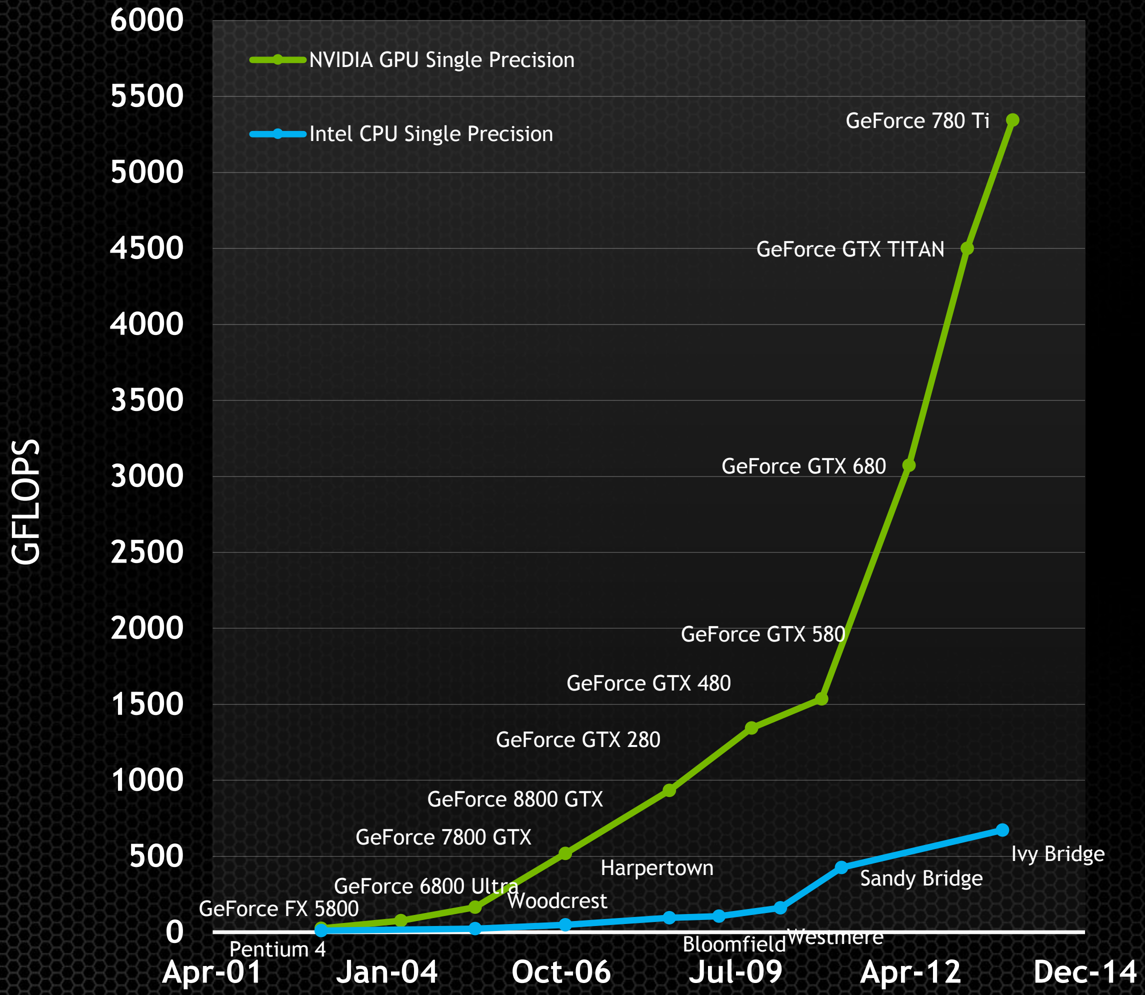
GPU Evolution

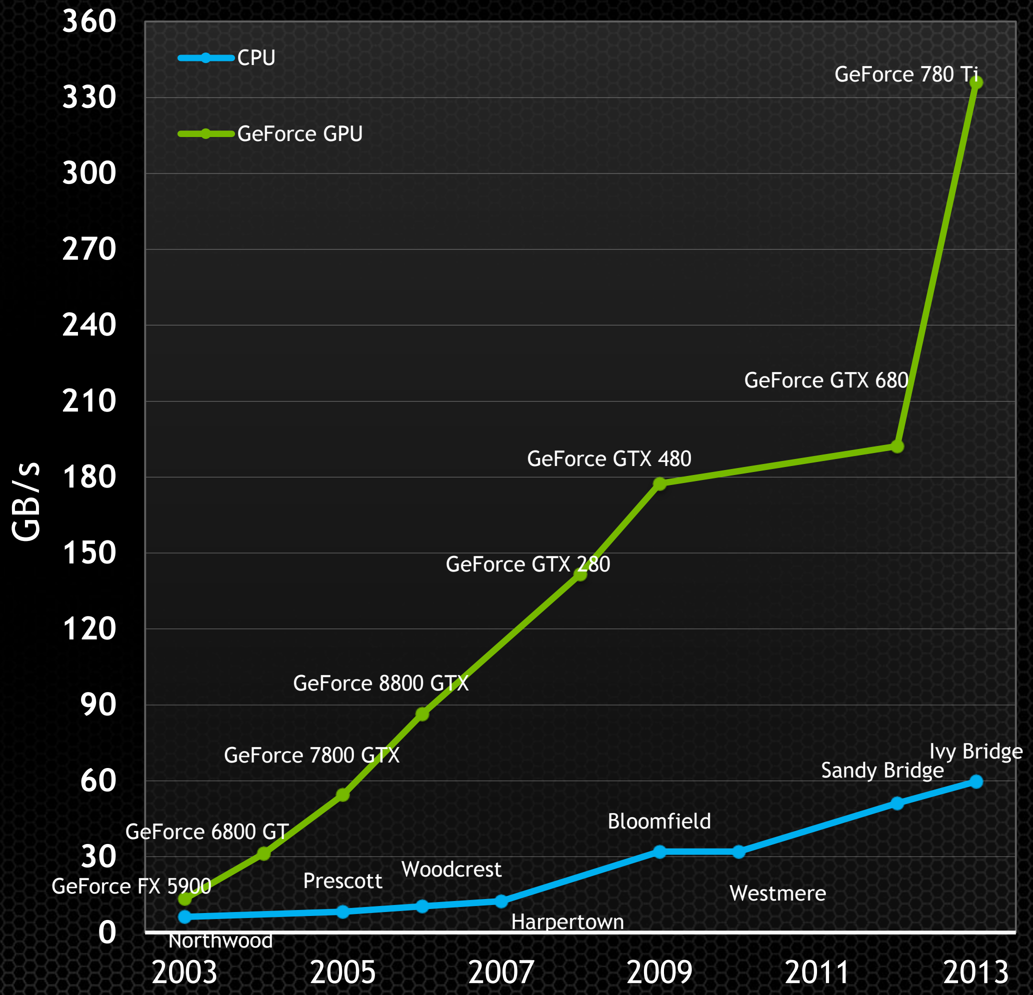
- Unified processor cores
- Multiple hardware engines
- MMU and paged memories
- More autonomous, fully programmable machines

Highly Integrated SoC Designs



GPU vs CPU Perf Scaling





Application Trends

From Functional Parallelism to Task-based Parallelism

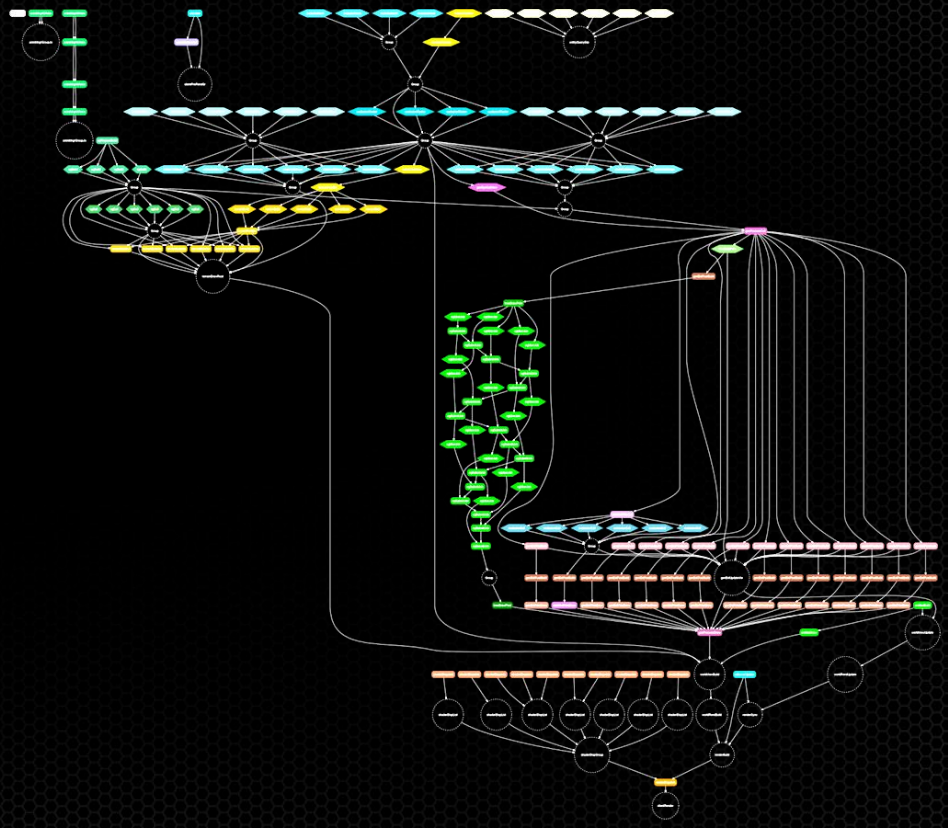
- Graphs of jobs, work stealing schedulers

GPU as a General Purpose Processor

- Physics & simulation, “Programmable Graphics”
- Requires more control over underlying hardware

Content is King

- Shifting balance between runtime costs vs production costs



From “Parallel Futures of a Game Engine”
© Johan Andersson

Current API Abstraction is Getting Old

Designed After >20 Years Old Machine Abstraction

Large State Machine Abstraction

- State transitions orchestrated by the CPU

Opaque Memory Model

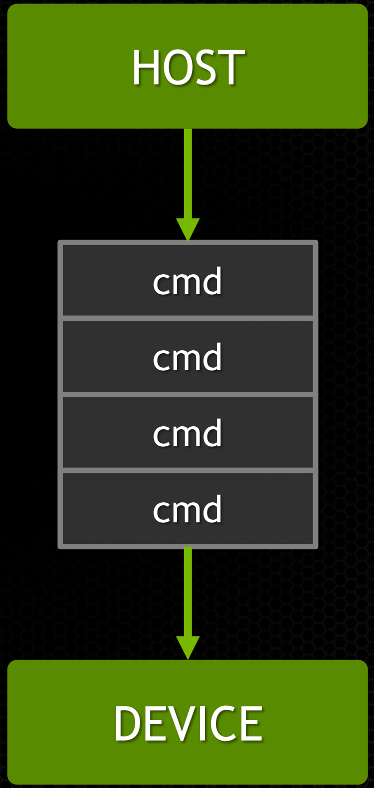
- Resource management hidden from the app

Limited Multi-core Scalability

- Serial submission model

Implicit Hazard Tracking and Synchronization

- High work submission costs


HOST

cmd

cmd

cmd

cmd

DEVICE

NVIDIA

DirectX 12

Radical Reductions in Submission Overhead

Embrace Latest GPU Architectural Improvements

Highly Scalable on Multi-core Systems

Provide Console-like Execution Environment

Designed in Close Collaboration Between Microsoft and NVIDIA

But…

Application Takes on More Responsibilities

- Resource hazard tracking
- CPU-GPU synchronization
- Memory management

Much Easier to Shoot Yourself in the Foot

- DX11 is still a great API for apps that want a simpler, more automatic programming model

Well Written App Will Get Great Benefits!

DX12 Nuts and Bolts

State Management Model

Command Lists

Resource Binding And Hazard Tracking

Residency Management And Memory Model

CPU/GPU Synchronization

State Management

Further State Grouping and Factorization

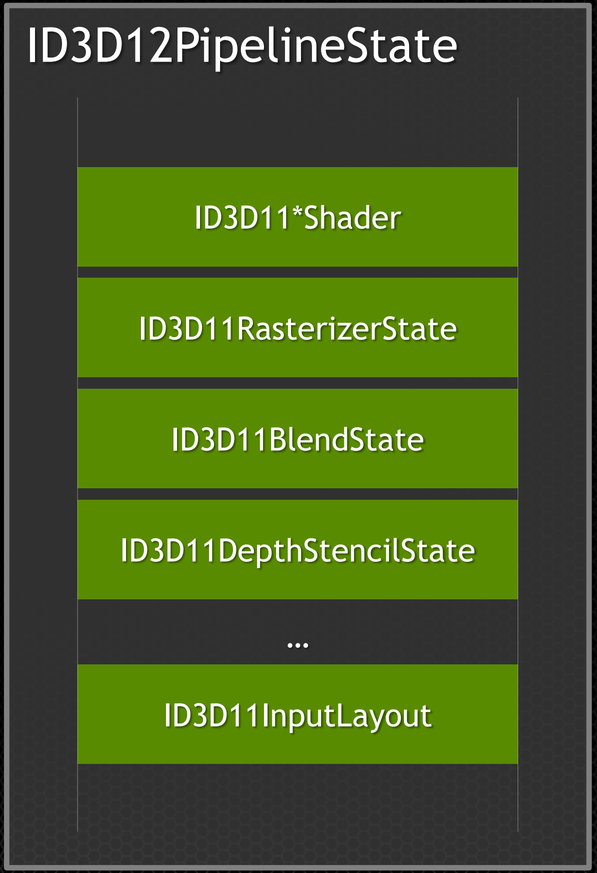
- Provide opportunities for aggressive pre-baking
- As much cost as possible is paid up-front

Pipeline State Encapsulates Heavy-weight State

- Bits of state specified and used together

More Frequently Changing State Still Lives Outside PSOs

- Lighter weight state changes



ID3D12PipelineState

ID3D11*Shader

ID3D11RasterizerState

ID3D11BlendState

ID3D11DepthStencilState

…

ID3D11InputLayout

PSO and Non-PSO State

PSO	Non-PSO
Shader Program at Every Stage	Resource Bindings
Blend State	Viewport Mappings
Rasterizer State	Scissor Rectangles
Depth-stencil State	Blend Factor
Input Layout	Depth Test
Render Target Properties	Stencil
Input Topology “Type”	Input Topology “Bucket” (List vs Strip)

New Binding Model

Current GPUs Can Reference a Large Namespace of Resources

- So called “bindless” model

Resource Binding State is “Pulled” Into the System

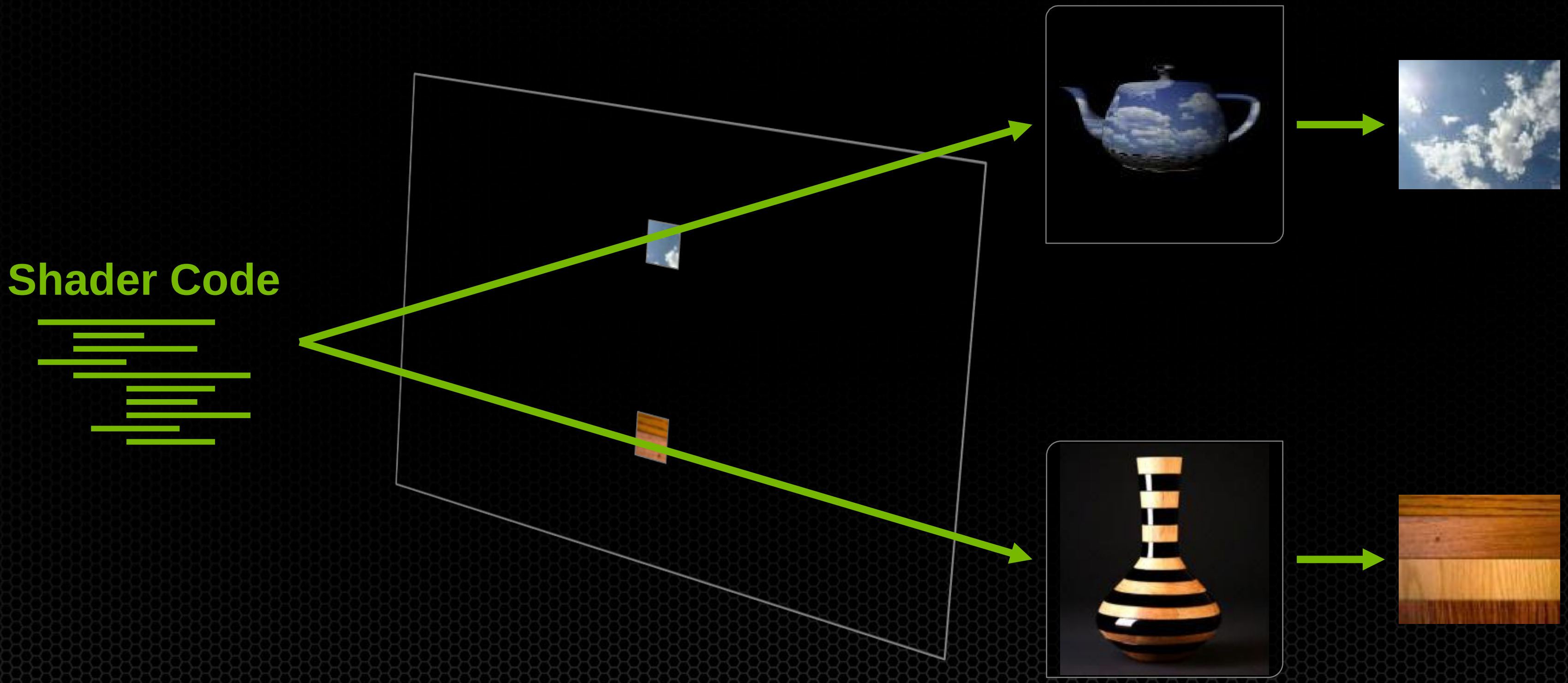
- Rather than CPU “pushing” it - more scalable and efficient

Can Reference a Very Large Working Set of Resources

- E.g. Kepler GPUs can use a “palette” of > 1M textures

Bindless Benefits

Essential for raytracing and next-gen rendering
where resource working set is not known in advance



Shader Code

Descriptors

Descriptors encapsulate handles to resources

- Semantically equivalent to ID3D11*ResourceView objects
- Opaque bag of bits, of implementation-specific format and size
- Expected to be on the order of 64B-128B

No driver-side allocations associated with descriptors

- Can be freely copied around and thrown away by the app

API functions to create descriptors given app-provided specification

Descriptor Tables and Heaps

Descriptor Tables Encapsulate Palettes of Resource References

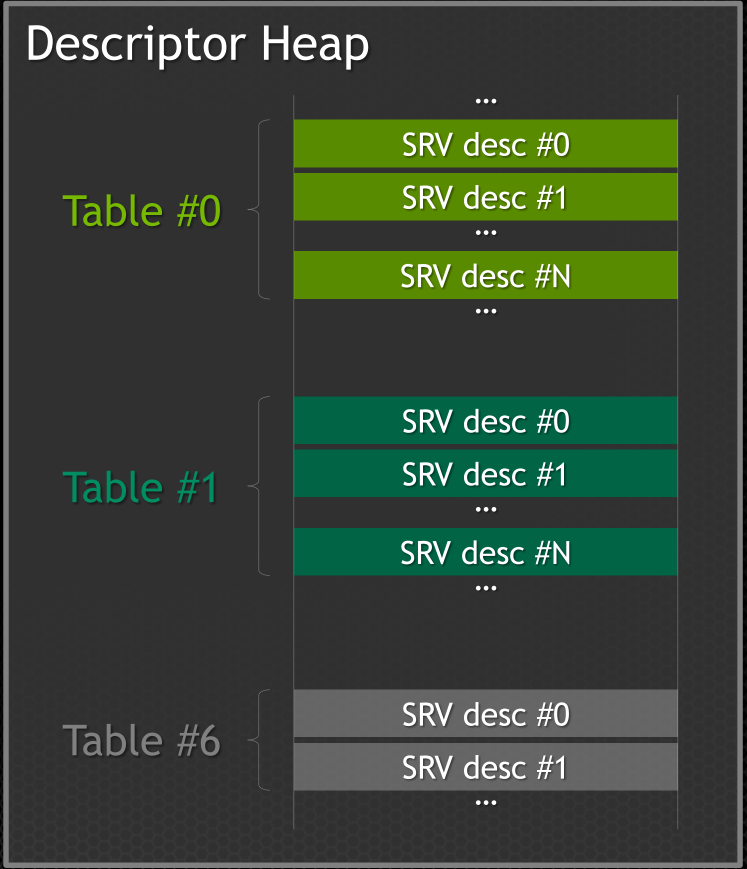
- Contiguous arrays of descriptor entries
- Individual descriptors indexable from the GPU

Tables Defined in Descriptor Heaps

- Driver provided memory
- Implementation-specific size restrictions

Represent Unit of Resource Binding State

- Allow for bulk binding of resources



Descriptors, Tables, and Heaps

Tables Very Cheap to Switch

- Ideally just a pointer change to the HW
- Efficent bulk binding state change

A Collection of Tables Can be Set at a Time

- Shaders can select a table to use for indexing
- Restrictions based on resource type

Applications Responsible for Managing Tables Within Heaps

- Various strategies, balancing heap space vs performance

Command Lists

Fundamental Unit of Work Submission

- No immediate contexts anymore

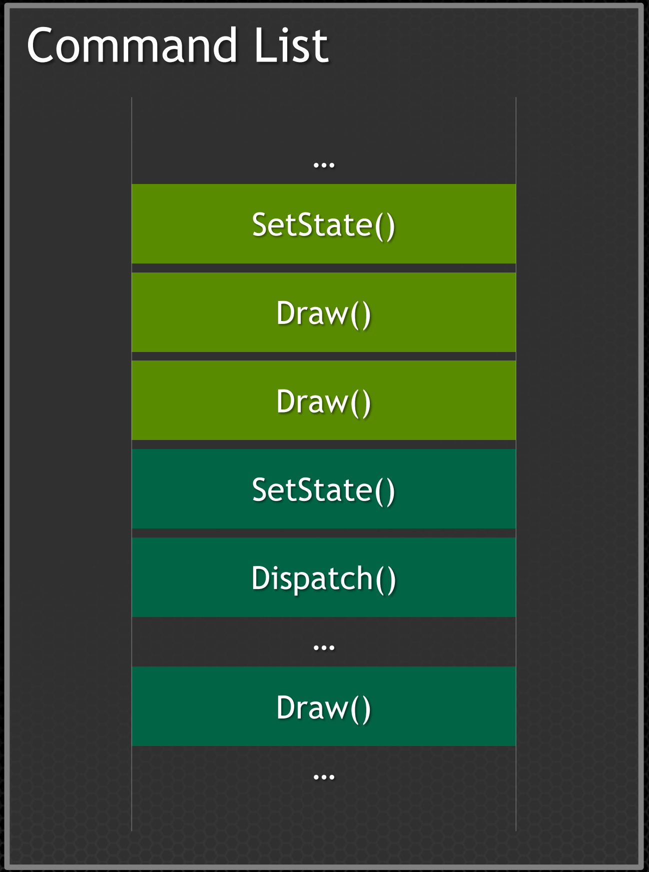
Designed to be Fully Bakable by the Driver

- Cannot be nested
- No further translation needed at submit time

Multiple Threads Can Record and Enqueue Concurrently

Represent Rendering Segments

- Re-recorded and played back every frame



Bundles

Highly Reusable Rendering Components

- A kind of command list

Can Only be Executed From “Direct” Command Lists

Non-pipeline State Inherited from the Calling Command List

- Provides for reuse flexibility

Represent Individual Objects in a Scene

- Collection of draws and state changes

Residency Management And Heaps

Heaps Represent Bulk Memory Allocation

- Unit of OS/driver memory management

Explicitly Separate From The Binding Model

- Individual resource bindings no longer tracked

Residency Managed At Coarse Grain Per Heap

- Applications responsible for managing memory within heaps

Hazard Tracking

Implicit Hazard Tracking Becoming Hard and Impractical

- Tiled resources allow for memory aliasing
- Very large palette of resources can be referenced

Applications Use Explicit Barriers to Signal Hazards

- E.g. resource transitioning from RTV to SRV
- Can exploit app-level knowledge and be less conservative

Robustness vs Efficiency Trade off

CPU/GPU Synchronization

Hazards Between Multiple Concurrent Processors Managed With Fences

- CPU sharing data with the GPU

Applications Responsible for Setting Fences and Tracking Them

- Can use standard OS synchronization APIs

Renaming and versioning optimizations are responsibility of the app

- Dynamic Resources are Effectively Permanently Mapped

Diving Into Nitrous

- Nitrous = Oxide’s custom engine
- Specifically designed for high throughput
- Core neutral. Main thread acts only as lightweight sequencer
- All work divided up into small jobs, which are in the microsecond range
- Can produce lots of jobs, 10,000+ range per frame

Multi-core CPU Basics

Be Wary, There Is A Lot Of Very Bad Advice In The Wild

- Spawning threads to handle tasks
- Relying OS preemptive scheduler, heavy weight OS synchronization primitives
- Functional threading in general

Your Survival Guide

- OK: Multi-thread read of same location
- OK: Multi-thread write to different locations
- OK: Multi-thread write to same location in ‘stamp’ mode
- CAUTION: Atomic instructions
- STOP: Multi-thread read/write to same location
- STOP: Multi-thread write to same CACHE line

Setting Up Our Frame

Concept of Asynchronous GPU Now Exposed Through API

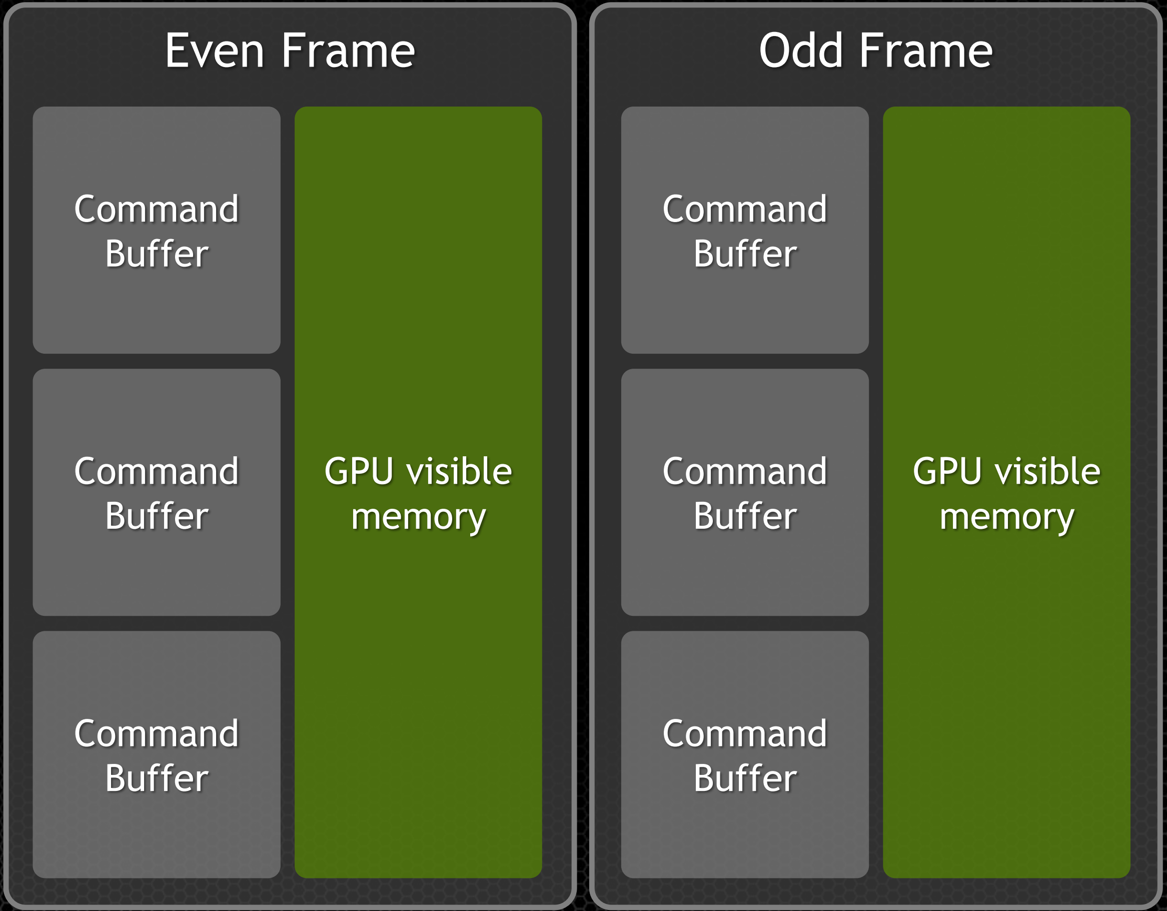
- Must buffer frames ourselves
- Will create 2+ copies of everything

CPU to GPU Data

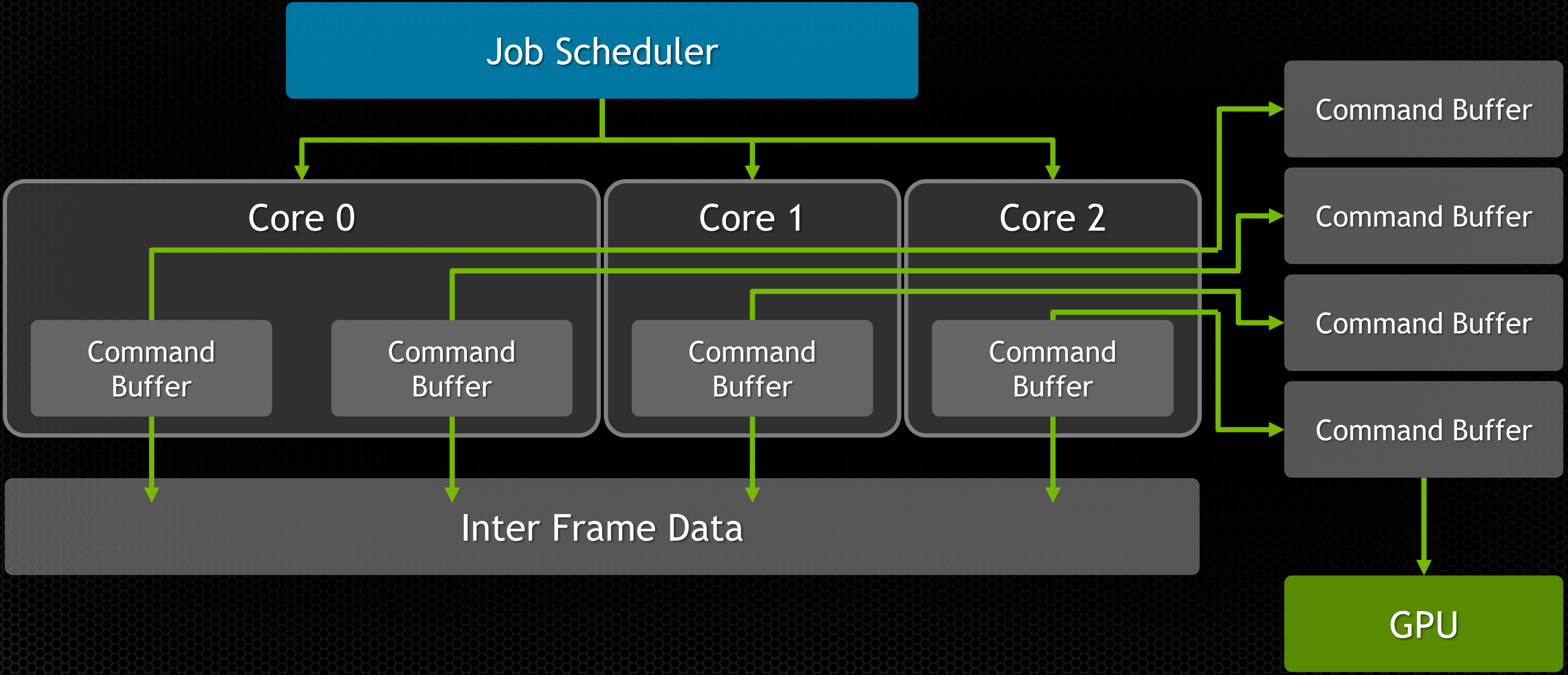
- Shader constants
- Texture updates

GPU Commands

- Command group = self contained, no cross thread hazards
- Means WAW, RAW hazards must be marked in command buffers
- Data nor command should not be changed while GPU could be reading



But … Commands May be Generated OoO



More details about our frame

- In reality, diagram is over simplified
- Nitrous has it’s own internal command format
  - Small, efficient commands
  - Stateless, each command contains references to all needed state
  - Inheritance unneeded
  - Seperates internal graphics system from any particular API
- Being Stateless, can be generated completely out of order
- Entire Frame is queued up in internal command format
- D3D12 is translated. Each internal command is translated, 1:1 mapping for each command buffer to a D3D12 command list
- Get’s more optimal use out of instruction cache and data cache
  - Allows us surrender the entire system to the driver

Shaders

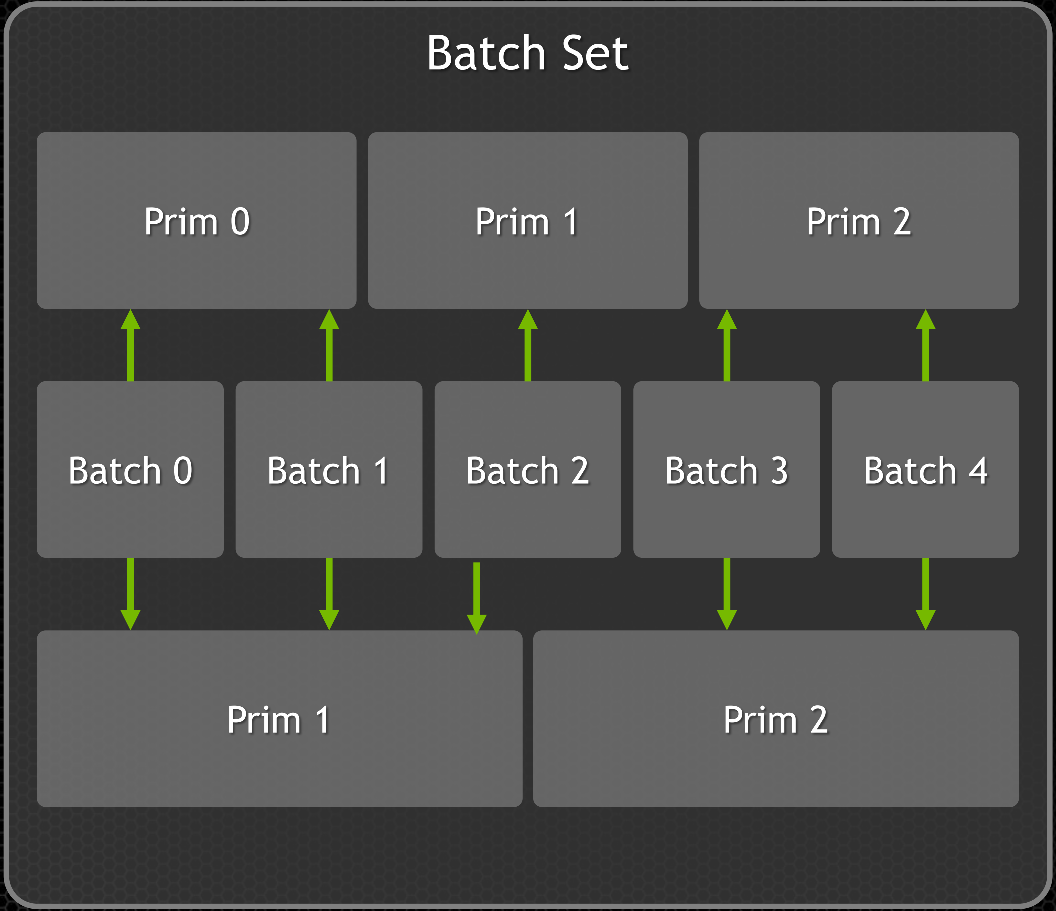
- Shader Blocks
  - Group of shaders with identicle resources
  - Key point: No changing of individual shaders, shader considered one monolithic block
  - All resources bound at same places across all stages
  - Maps well to a group of pipeline state objects in d3d12

```
ShaderGroup SimpleShader
{
    ResourceSetPrimitive   = VertexData;
    ConstantSetDynamic[0] = DynamicData;
    ResourceSetBatch[1]   = UserTS;
    ConstantSetShader[0] = Globals;

Methods
{
main:
     CodeBlocks = SimpleShaders;
     VertexShader = SimpleVSShader;
     PixelShader =  SimplePSShader;
zprime:
     CodeBlocks = SimpleShaders;
     VertexShader = SimpleVSShader;
     PixelShader =  BlankSimplePSShader;

}
}
```

Four Frequencies of Updates



Even Frame

PRIMITIVE

IB
Resources
Tri info

BATCH

Primitive
Shader
Resources (2)
Constants (2)

BATCH SET

Batches
Primitives
Shaders
RTs
Blend State

SHADER

Resources (2)
Constants (2)
Shader Block

Resource Sets

In Real World, Textures Are Grouped

Nitrous Has 5 Bind Points

- 2 for batch
- 2 for shader
- 1 for primitive

VB Is Just A Resource Set

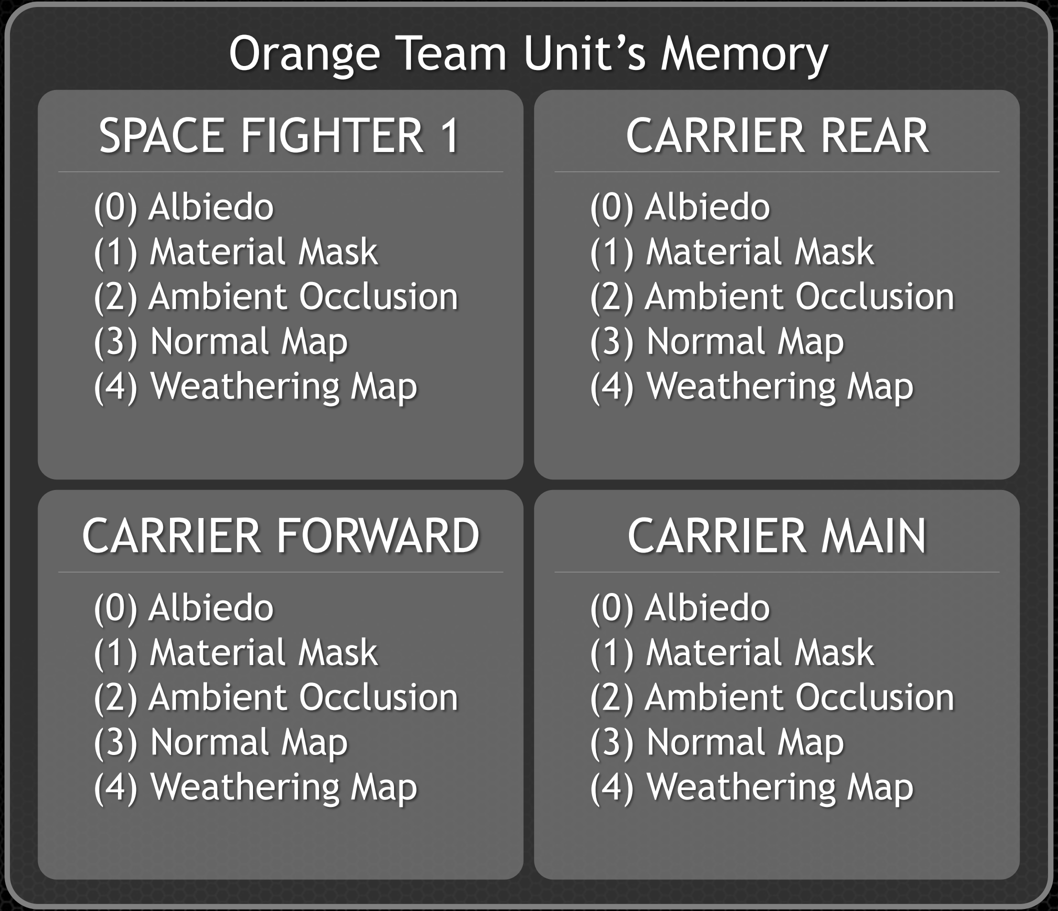
Maps Well To D3d12 Descriptor Tables

SPACE FIGHTER 1

(0) Albiedo
(1) Material Mask
(2) Ambient Occlusion
(3) Normal Map
(4) Weathering Map

Memory Pools

- Resources used together, created together
- Multiple resource sets are often pooled
- Simplifies memory management, less then 1000 total allocations
- Maps well to D3D12 memory heap



Orange Team Unit’s Memory

SPACE FIGHTER 1

(0) Albiedo
(1) Material Mask
(2) Ambient Occlusion
(3) Normal Map
(4) Weathering Map

CARRIER REAR

(0) Albiedo
(1) Material Mask
(2) Ambient Occlusion
(3) Normal Map
(4) Weathering Map

CARRIER FORWARD

(0) Albiedo
(1) Material Mask
(2) Ambient Occlusion
(3) Normal Map
(4) Weathering Map

CARRIER MAIN

(0) Albiedo
(1) Material Mask
(2) Ambient Occlusion
(3) Normal Map
(4) Weathering Map

Constants and Resource Updates

- Common data used per frame is dumped into a “Graphics Transfer Memory”
- Reset every frame, with a simple incrementing counter
- All constants are uploaded into this memory, and referenced by commands
- Per frame resource updates placed in this memory
  - No point in resizing and reallocating, need the max memory planned for in advance
- Non per frame updates, which are big, other memory is allocated, but caller must free memory after receiving notification that GPU command is complete
- D3d12 this ends up just being a buffer

Starting the frame

```
bool BeginFrame()
{
    bool bGPUbound = true
    int FrameData = g_uFrame % FramesQueued
    if g_Fences[FrameData] is blocking
        then g_Fences[FrameData]->WaitOnFence
        Else bGPUBound = false

    //we know the GPU is free with this now
    g_FrameData[FrameData]->LockAndMap()

    g_uFrame++;
    return bGPUBound;
}
```

This function should take place on sequencer thread, once ready to begin generating commands

Can place timers around both sides of the fence and get accurate CPU side time of our app

Median frames queed up should be FramesQueued + .5, so usually only 1.5 frames behind

Generating a command list

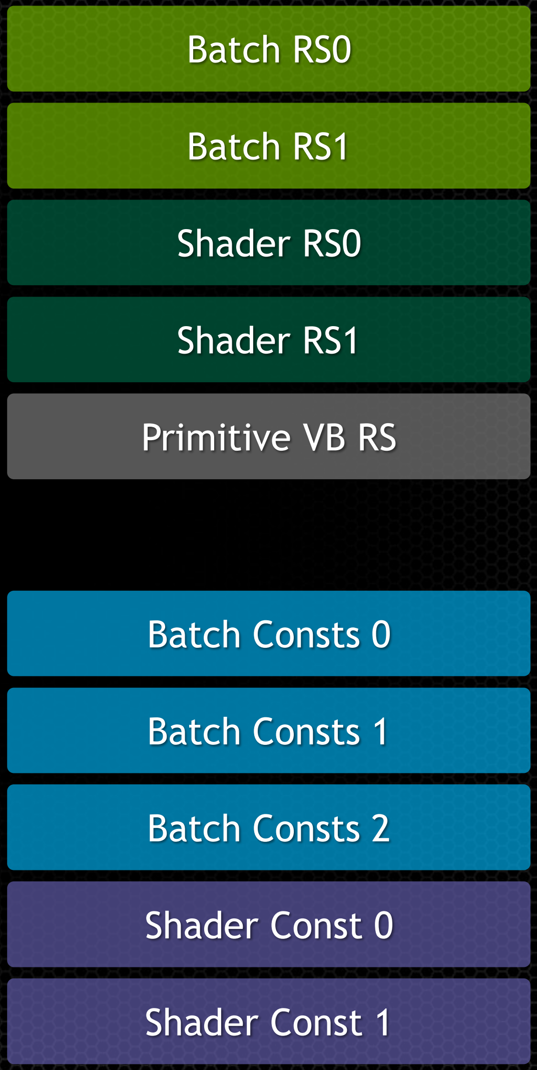
- Once frame has begun, can begin issuing command lists
- Easy strategy: Create 2 sets of command lists, 1 set for the frame being rendered, and 1 for the frame being generated (or n, if more then 2 frames are queued)
- What we do:
    - Create a series of tasks which dump rendering into command buffer
    - Don’t create a new command buffer for every task, however, each thread has a context which points to a command buffer
    - So, if we have 6 CPUs and 600 objects to process in a particular system, will end up generating 6 command buffers with 100 batches each
    - The draw order will fluctuate frame to frame, so must handle alpha differently

Resource Sets

Natural Allegory To Descriptor Tables

- Since Resource Sets are immutable, and can bind multiple, can pre create
- If hardware can support multiple descriptor tables, conceivable that we don’t have to update them per frame ever
- Otherwise, need to dynamically create a descritor table for each batch

What Resource Bindings Exist For Nitrous? Turns Out We Have A Small State Vector



Batch RS0

Batch RS1

Shader RS0

Shader RS1

Primitive VB RS

Batch Consts 0

Batch Consts 1

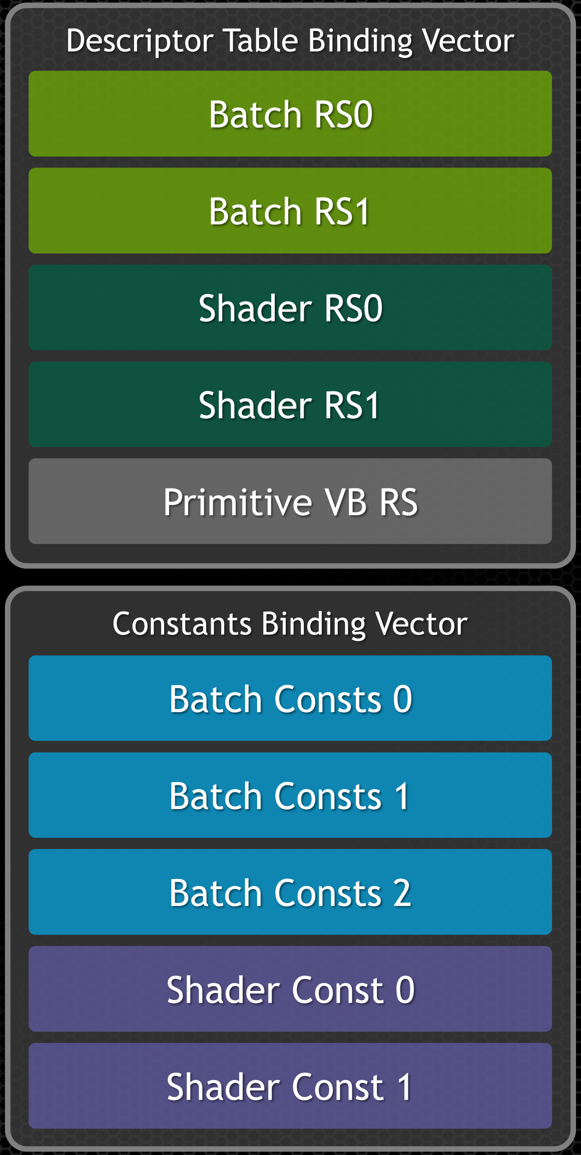
Batch Consts 2

Shader Const 0

Shader Const 1

Bindings

- If we have enough binding points, then we simply bind the descriptor table to that bind point
- If we have don’t have enough bind points (e.g. only 1), then we create a new table for parts of the bind vector that we need
- Can Cache them



Descriptor Table Binding Vector

Batch RS0

Batch RS1

Shader RS0

Shader RS1

Primitive VB RS

Constants Binding Vector

Batch Consts 0

Batch Consts 1

Batch Consts 2

Shader Const 0

Shader Const 1

Generating a Command List

```
For each Batch (inside a task)
    Create State vector from Batch, Shader, and Primitive
    if not enough bind points
        Lookup state vector in our cache (16 entry, unique cache per thread)
    if in cache
        Bind created descriptor table
    else
        Create New descriptor table (since same size, we have a simple pool of them)
        Bind descriptor table, evict last used thing in cache and add to cache
  …
```

Tracking Resource Usage

- Apps responsibility to track what resources get used
- Simple strategy: Stamp a frame number on each memory pool anytime it is bound
- Traverse the complete resource list, anything which matches current frame must be resident
- Quick as long as we keep # of heaps reasonable
- Important: Frame # should be padded into a cache line to avoid serialization

Heap Description	Last Frame Used
UI Textures intro	2401
UI Textures in Game	17204
Orange Faction Units	17204
Purple Faction Units	17204
Weapon effects	16392
Post Process RTs	17204
Terrain Heightmap	17204

Expected results

Expecting large increases in performance in terms of CPU time spent in driver/D3D

Expecting vast reduction in driver complexity, hopefully more robust drivers

Expecting less frames to be queued, generally more responsive games

Shouldn’t have frame hitches caused by driver at all.

WE WOULD LIKE YOUR FEEDBACK

Please take a moment to fill out this 2 minute survey on your own device for this talk

We appreciate your input