

GPU TECHNOLOGY
CONFERENCE

April 4-7, 2016 | Silicon Valley

GPU-DRIVEN RENDERING

Christoph Kubisch Sr. Developer Technology Engineer, NVIDIA

Pierre Boudier Quadro Software Architect, NVIDIA

4/4/2016

PRESENTED BY



WHAT IS THIS TALK ABOUT?

Or should I sneak out of the room now

Latest of a series of presentations

Rendering heterogenous scenes with many distinct objects

Improving the overall system, rather than tuning individual shader

Using knowledge of how the hardware implements the graphics pipeline

GPU TECHNOLOGY CONFERENCE

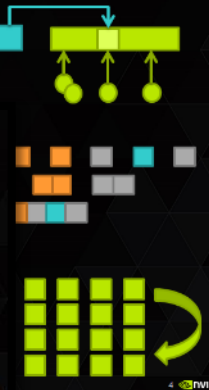
ENABLING GPU SCALABILITY

- Avoid data redundancy
 - Data stored once, referenced multiple times
 - Update only once (less back to gpu transfer)

GPU TECHNOLOGY CONFERENCE

CHALLENGE OF ISSUING COMMANDS

- Issuing drawcalls and state changes can be a real bottleneck



	650,000 Triangles	3,700,000 Triangles	14,338,275 Triangles/lines
Triangles	650,000	3,700,000	14,338,275
Parts	68,000	98,000	300,528
Drawcalls (parts)	-	-	300,528
Triangles per part	~ 10	~ 37	~ 48

3 NVIDIA

<http://on-demand.gputechconf.com/gtc/2013/presentations/S3032-Advanced-Scenegraph-Rendering-Pipeline.pdf>
<http://on-demand.gputechconf.com/siggraph/2014/presentation/SG4117-OpenGL-Scene-Rendering-Techniques.pdf>
<http://on-demand.gputechconf.com/gtc/2015/presentation/S5135-Christoph-Kubisch-Pierre-Boudier.pdf>

PAST, PRESENT, FUTURE

Dawn of GPU: CPU was fast enough to feed GPU

Past: Increase in scene complexity challenged CPU to feed GPU fast enough

Present: Modern APIs and data-driven design methods provide efficient ways to render scenes

- OpenGL's Multi Draw Indirect, NVIDIA's bindless and NV_command_list technology
- Vulkan's native support for command-buffers and general design that allows re-use and much better control over validation costs



```
glBegin (GL_TRIANGLES)  
glVertex3f  
...
```

```
glMultiDrawElementsIndirect  
glDrawCommandsStatesNV
```



```
VkSubmitInfo info = {...};  
info.pCommandBuffers = ...  
vkQueueSubmit (queue,1, &info..)
```

PAST, PRESENT, FUTURE

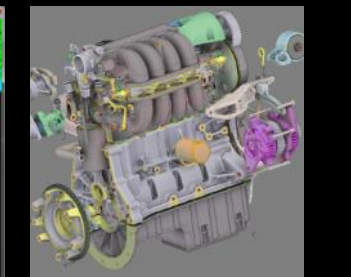
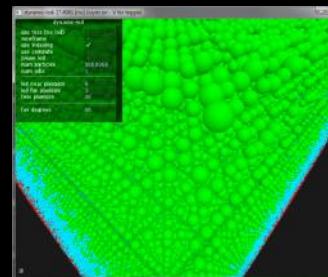
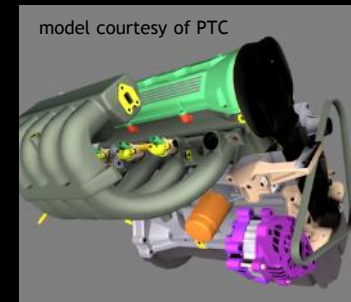
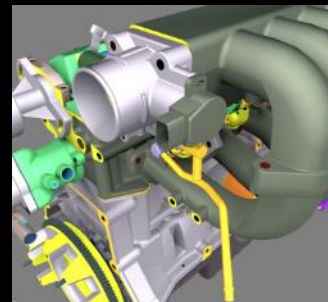
Future: Decouple CPU and GPU further as GPU becomes more capable to drive decision making

Occlusion Culling, Level of Detail (geometry and shading), Animation...

Easy access to practical information like depth-buffer, past frames...

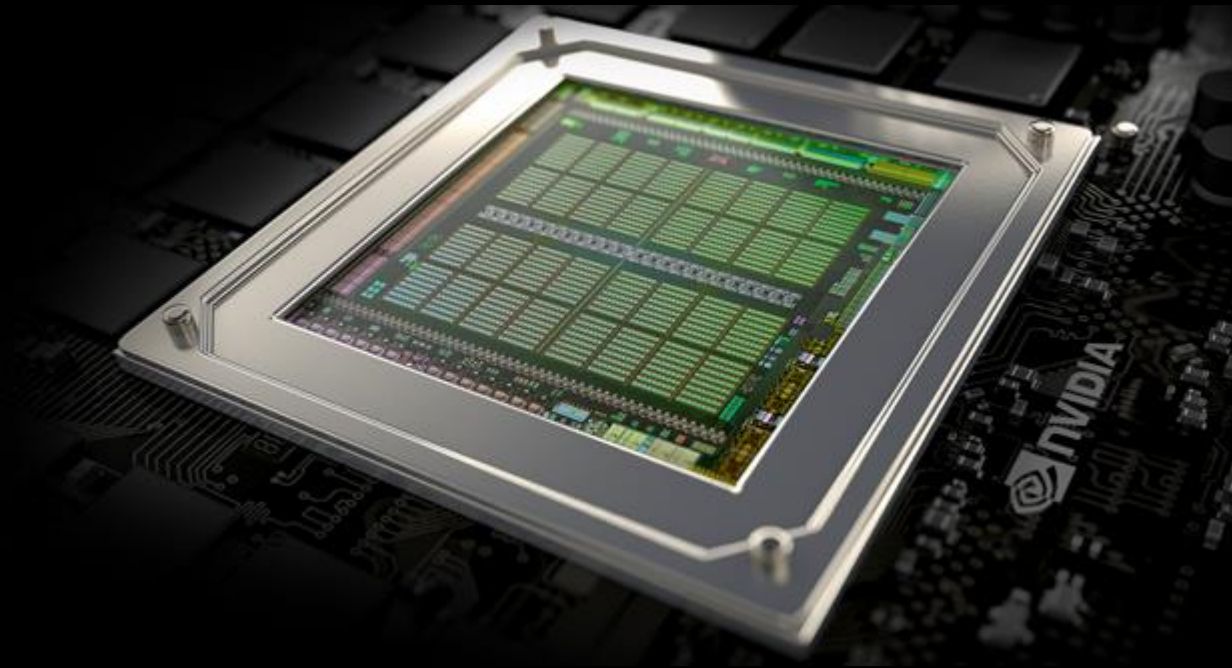
Minimizes latency, keeps data where it is used, produce and consume directly

Geometry and visual effect generation beyond basic tessellation



Distance Field Rendering - Iñigo Quilez, ShaderToy

PAST, PRESENT, FUTURE



How do we leverage the hardware well?

How does the hardware actually work?

AGENDA

Hardware design overview

Life of a triangle, a trip through the GPU
graphics pipeline

Practical Consequences

HARDWARE DESIGN

Wide Chips

Wider chips with more units favor large work:

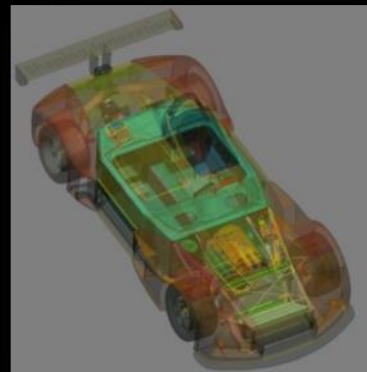
- Every cycle, if a unit does not have work to do, then efficiency is lost
- A typical applications goes over a series of tasks (shadows, object shading, post-fx...), each task is likely limited by a subset of units

Increased programmability allows for more data "generated" on chip

The trend is add more programmable units than fixed function ones



GM200 (launched in 2015)
6 Graphics Processing Clusters (GPC)
24 Streaming Multiprocessors (SM)
3072 CUDA cores
96 Raster Output Units (ROP)
192 Texture Units
384 bit memory interface



Changing bindings,
small object draw-
calls...

model
courtesy of
PTC

HARDWARE DESIGN

Fixed Function Units

Pipeline needs to scale with data expansion & reduction

➤ **drawcalls** (1..N primitives) → [**tessellation**] → **visible triangle** (0..M screen **pixels**)

Not efficient to do all in “generic units”, put code into metal

Any fixed function unit in the pipeline:

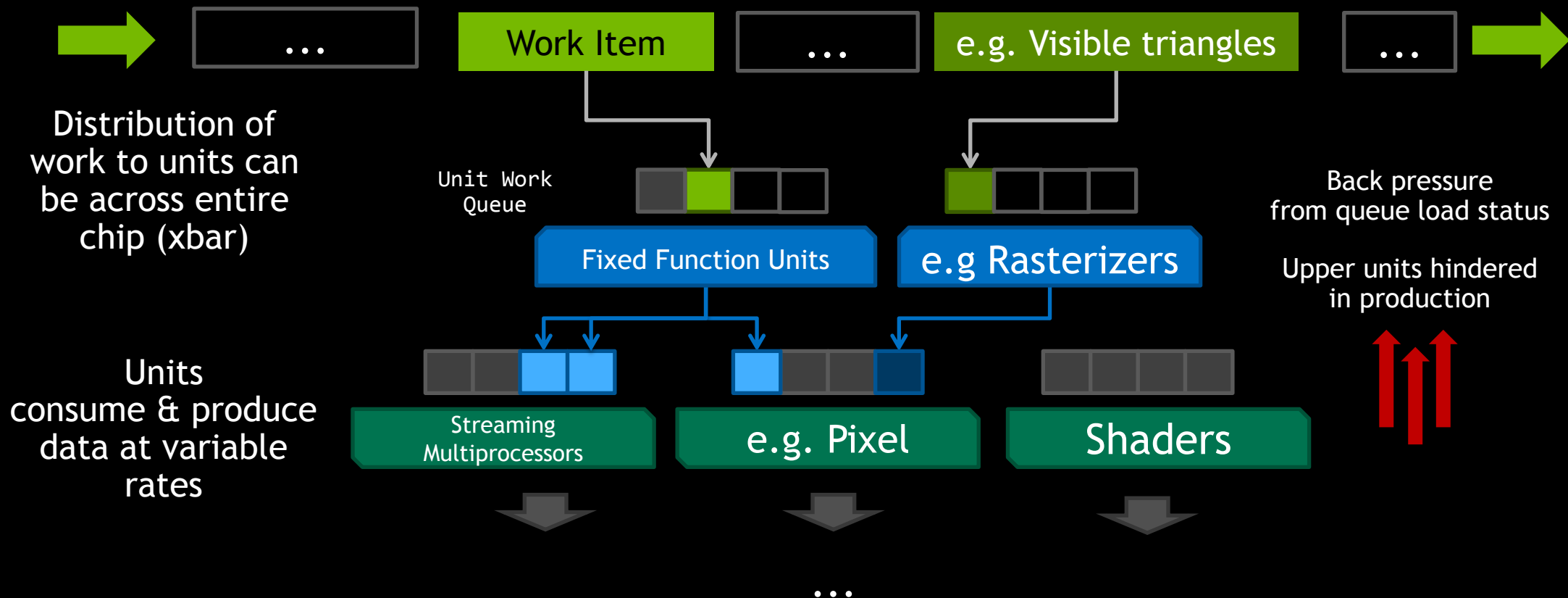
- Is a free resource to get useful work done
- But can potentially become a bottleneck if it is overloaded
- The number of units will vary for each GPU based on workload & cost

Rasterization Unit

Texture Unit

HARDWARE DESIGN

Distribution and Queueing of Work



HARDWARE DESIGN

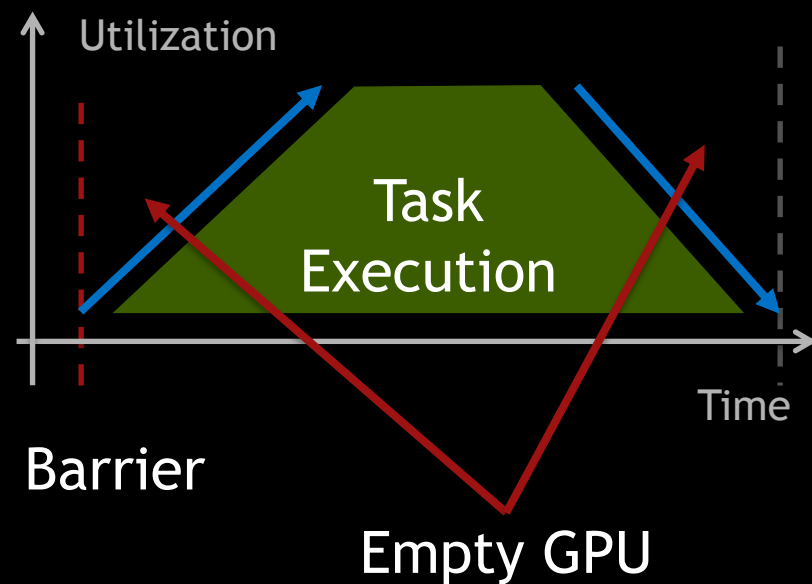
Ramp up/down time

Large number of units and system latencies cause **efficiency loss before & after** a GPU “wait for idle”

Ramp up/down time depends on the actual workload

Barriers may be caused by

- Hardware internal state
- Application (as in YOU)
glMemoryBarrier, RenderPasses...
ensure prior work has finished computation



HARDWARE DESIGN

Memory Hierarchy

The closer the memory to the execution units the faster the access

Off-Chip memory takes a loooong time to get to the execution unit (avoid cache miss)

Off-Chip
Memory

Chip
Global

Processor
Local

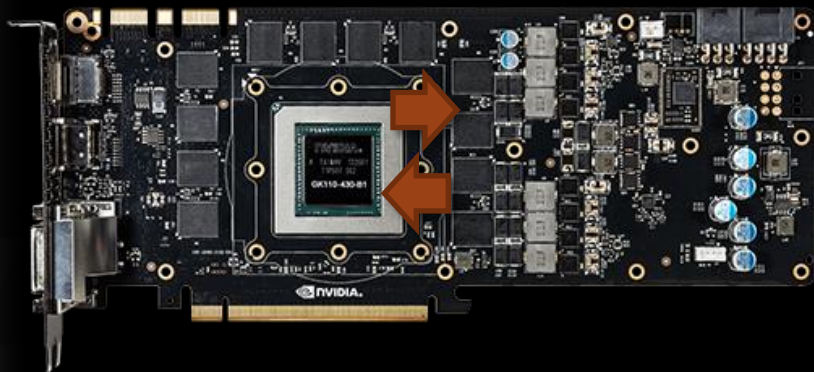
DRAM

L2 Cache

L1 Cache

Register Files

Execution Units



HARDWARE DESIGN

Latencies

FRONT-END

Fetching from pushbuffer:
~1000 cycles (PCIe latency)

Processing of one simple
state:
1 cycle

VERTEX SHADER

Fetching vertices:
~300 cycles

Simple transformation:
~200 cycles

PIXEL SHADER

Fetching texture from L1:
~100 cycles

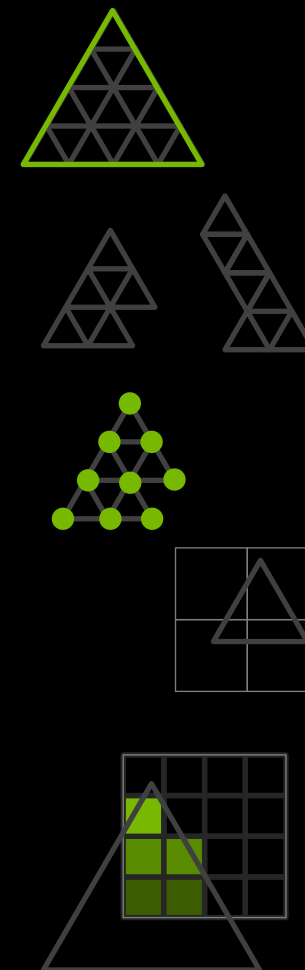
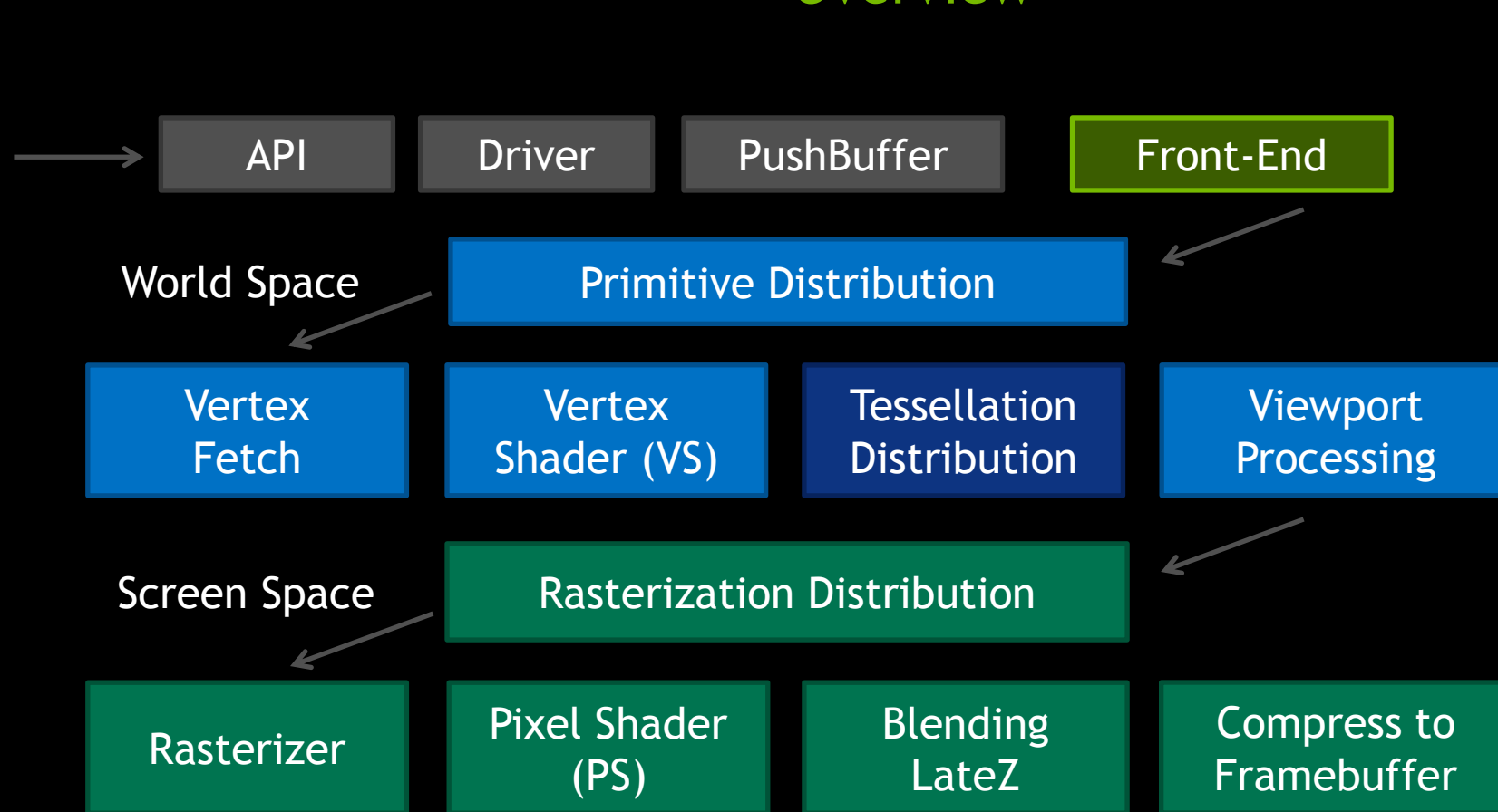
Fetching texture from L2:
~200 cycles

Fetching texture from DRAM:
~500 cycles

LIFE OF A TRIANGLE

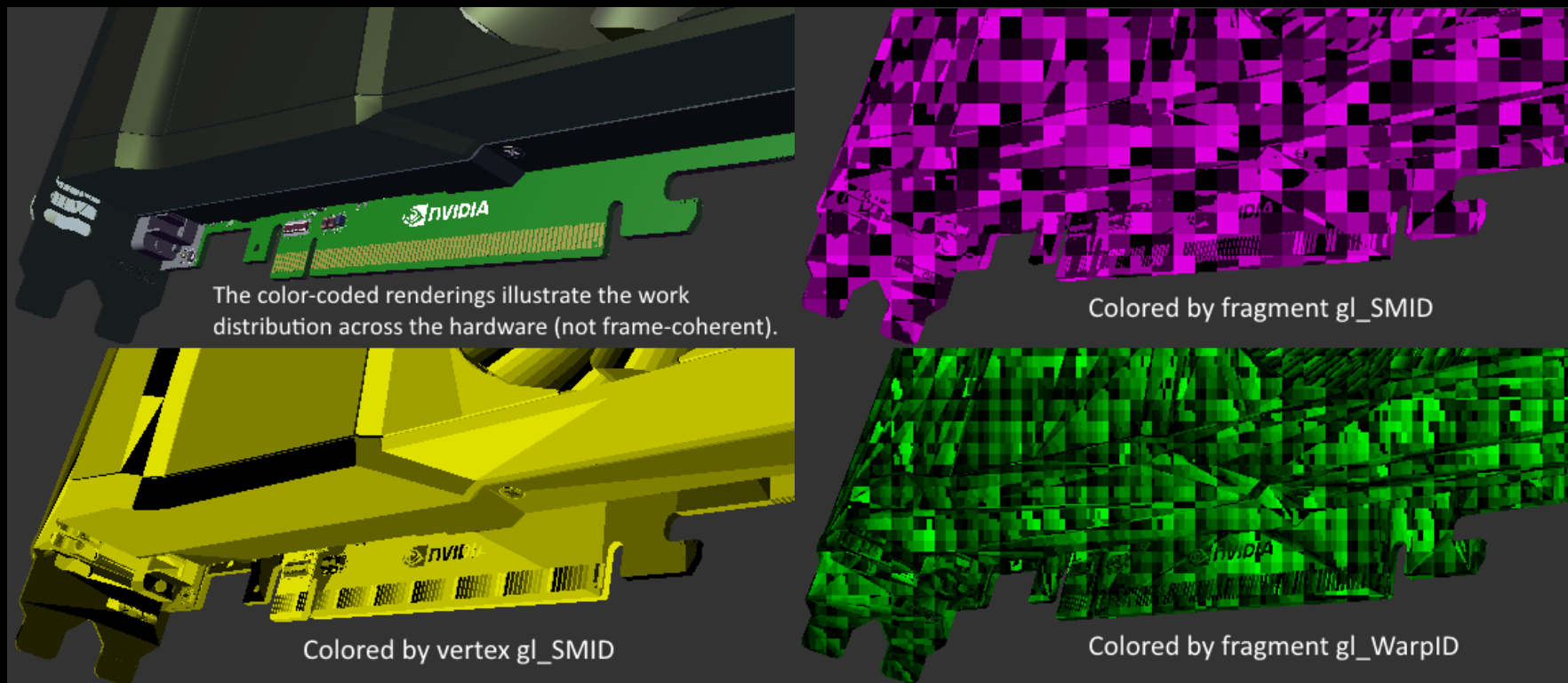
LIFE OF A TRIANGLE

Overview



LIFE OF A TRIANGLE

Work Distribution



Shaders executed as Warps: Group of 32 threads, sharing instruction state

LIFE OF A TRIANGLE

From Api to GPU



```
VKAPI_ATTR void VKAPI_CALL vkCmdDraw (  
    VkCommandBuffer commandBuffer,  
    uint32_t vertexCount, uint32_t instanceCount, uint32_t firstVertex, uint32_t  
    firstInstance)  
{  
    checkAvailableSpace(commandBuffer->pushBuffer, 5);  
  
    commandBuffer->pushBuffer[0] = DRAW_HEADER;  
    commandBuffer->pushBuffer[1] = vertexCount;  
    commandBuffer->pushBuffer[2] = instanceCount;  
    commandBuffer->pushBuffer[3] = firstVertex;  
    commandBuffer->pushBuffer[4] = firstInstance;  
    commandBuffer->pushBuffer+=5;  
}
```

LIFE OF A TRIANGLE

From Api to GPU



```
VKAPI_ATTR VkResult VKAPI_CALL vkQueueSubmit(  
    VkQueue queue, uint32_t submitCount, const VkSubmitInfo* pSubmits, VkFence fence)  
{  
    for (uint32_t i = 0; i < submitCount; ++i) {  
        // submit wait for pSubmitInfo->pWaitSemaphores  
        // submit pSubmitInfo->pCommandBuffers to the GPU, typically by the kernel mode driver  
        // submit signal for pSubmitInfo->pSignalSemaphores  
        // submit fence signal  
    }  
}
```

LIFE OF A TRIANGLE

Top of GPU Pipeline

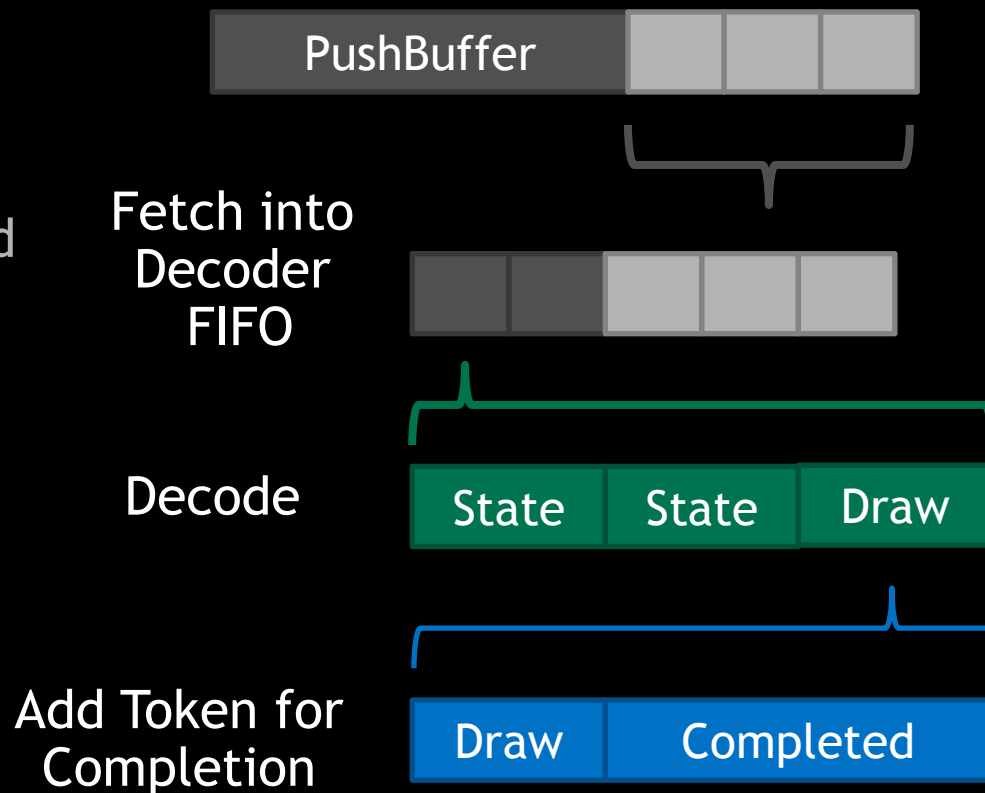


Fetch the pushbuffer over PCIe into a FIFO to hide latency

Decode resource bindings & state changes, and update the HW state

Decode initiator (draws)

Send token to track when all the work is finished (at the end of the pipeline)



LIFE OF A TRIANGLE

State Management

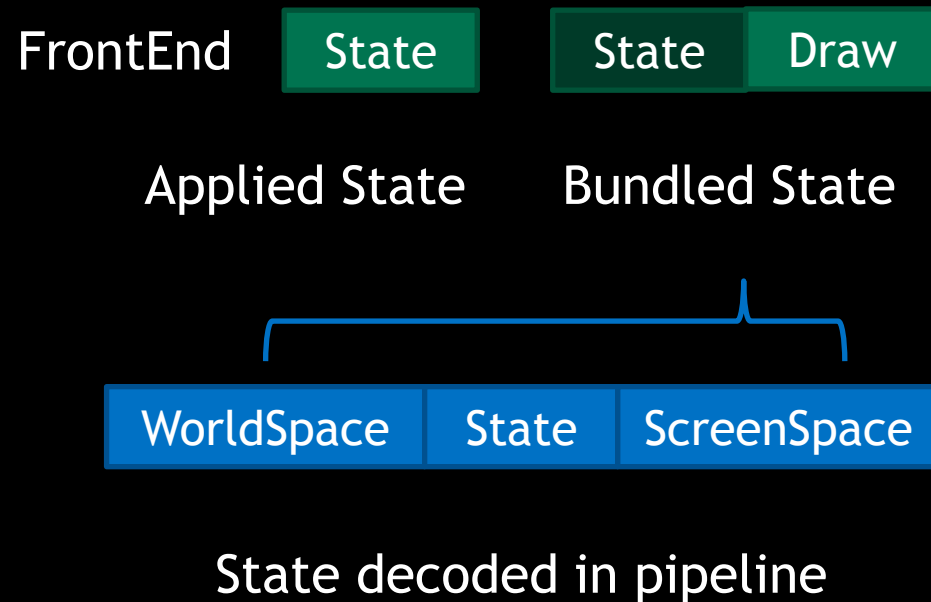


States can be **processed immediately**

Or can be **bundled to be decoded later** when the world space operation is finished

Redundant states are filtered when possible (but it may still take 1 clock to reject each one)

Some states are complex and can take many cycles (like a shader switch)



LIFE OF A TRIANGLE

Work Distributor

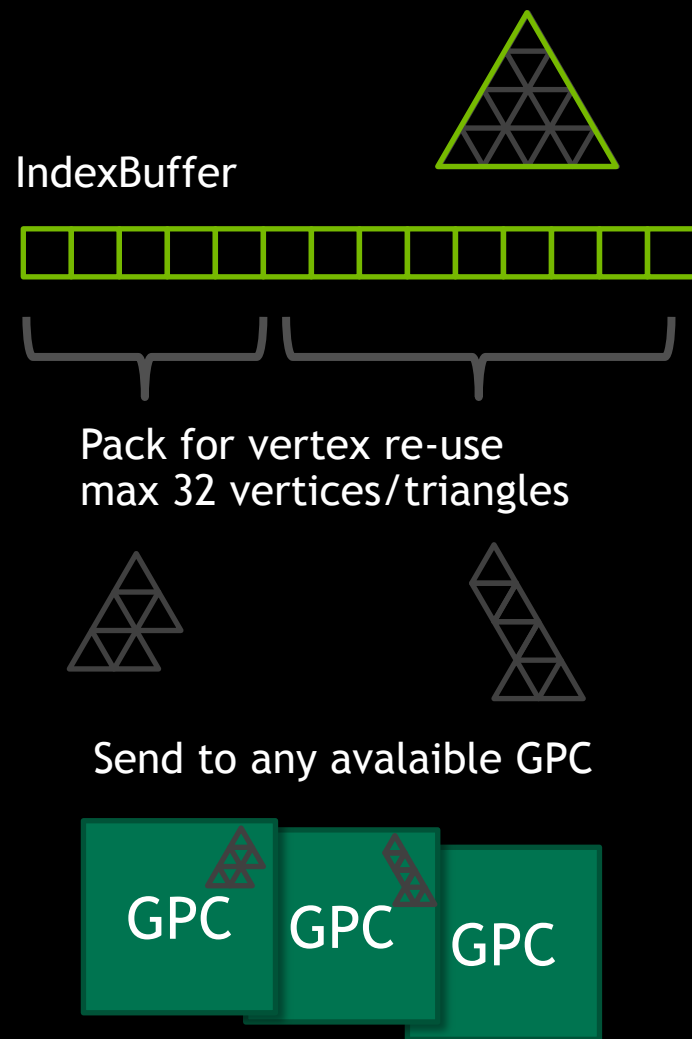
The main unit starting the “draw” operation

Collect indices (or auto generate them for non-indexed operation)

Manage vertex reuse to reduce the number of vertex threads

Create a batch of up to 32 triangles and 32 vertices (all vertices of a primitive always in same batch)

Select the GPC which will launch the next VS warp (1 warp = 32 threads)



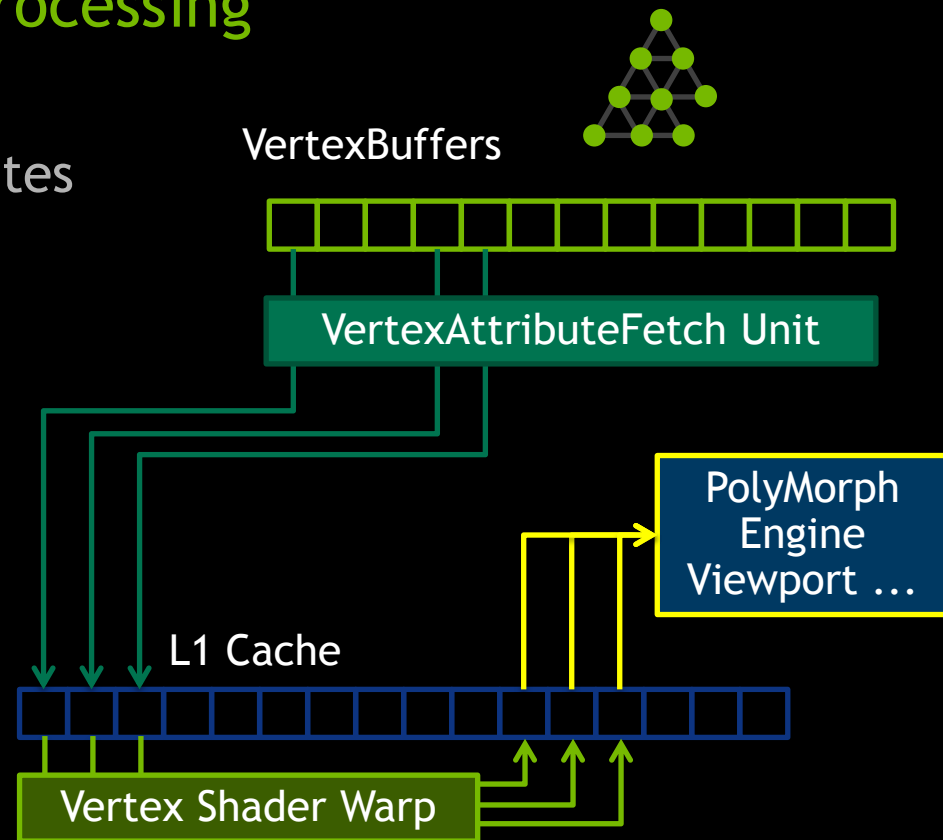
LIFE OF A TRIANGLE

World Space Processing

Dedicated HW unit pre-fetches the vertex attributes into L1 (hides latency for VS startup)

The vertex shader runs in the SM, and outputs attributes into L1

Viewport transform, Clipping, Culling in the PolyMorph Engines



LIFE OF A TRIANGLE

Screen Space Processing

A xbar will distribute the work to the raster units per GPC based on screen-space tiling

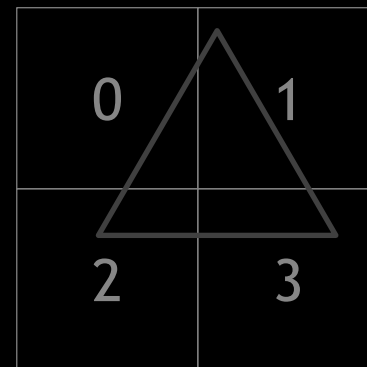
Hierarchical Z unit can reject tiles

Edge setup to evaluate sample coverage

2x2 pixel quads (4 threads) are generated for the PS for any non-null coverage

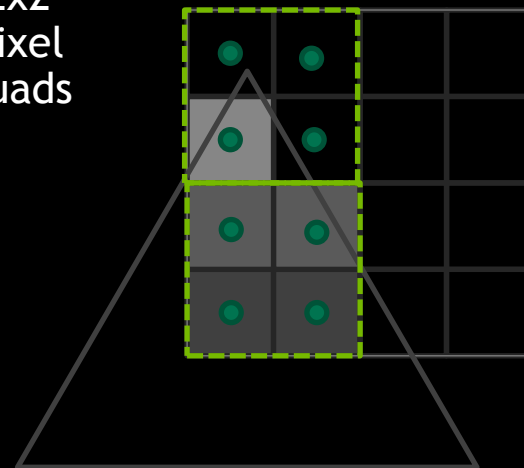
Optionally, early depth & stencil test can reject entire quad

Triangle sent to
Raster Engine(s)



Sampling Grid

2x2
pixel
quads



LIFE OF A TRIANGLE

Screen Space Processing

Pixel shader warp runs on multiple quads

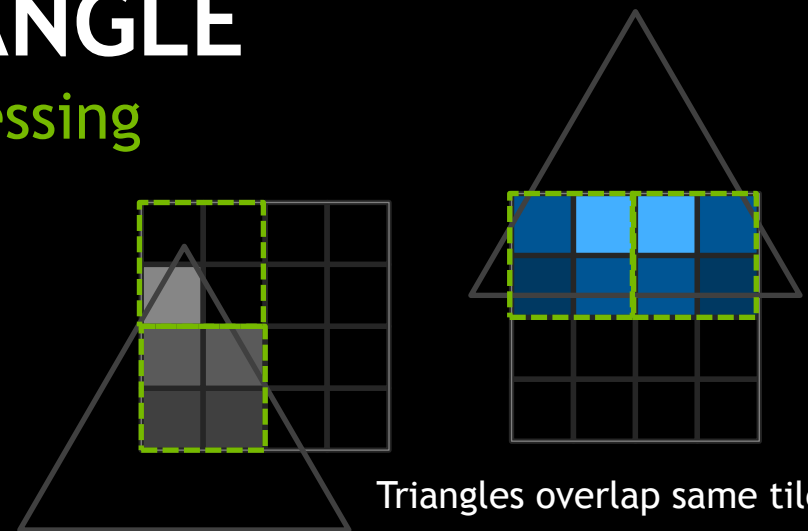
Fetches interpolated attributes

Masks read/writes from zero coverage samples

Texture filtering gradients(Mip-map levels, Aniso...) or ddx/ddy based on quad differencing

Writes out color

Can modify depth and coverage mask



Max 8 quads merged into warp (32 threads)



Mask out
„helper“ threads

```
bool gl_HelperInvocation  
// is true for helper threads
```

Easy access to
quad values for
gradients

LIFE OF A TRIANGLE

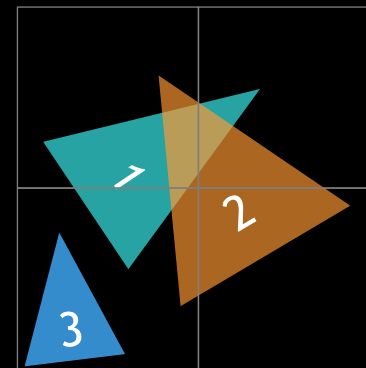
Color Blending / Depth Testing

Raster Output (ROP) units maintain API primitive order

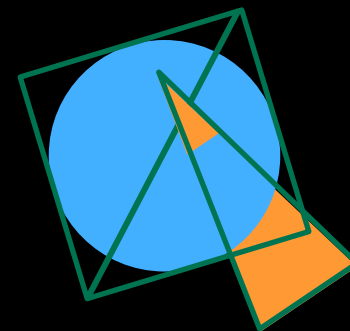
- Easy since the rasterization is screen mapped
- Each unit only needs to handle warp in order

Fixed function or programmable blending of the color

Optionally perform Late-Z rejection
(triggered by discard or `gl_FragDepth` writes in PS)



Only order within tile
important for ROP



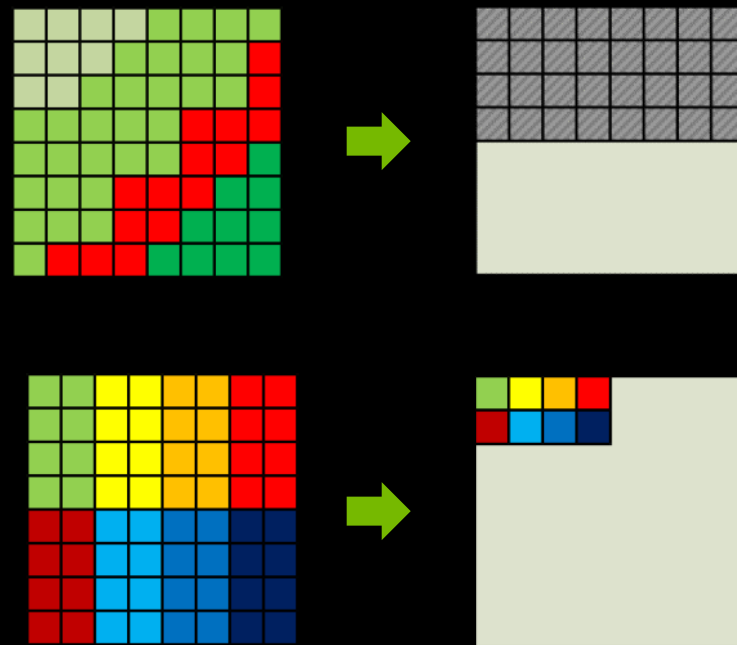
Discard needs Late-Z

LIFE OF A TRIANGLE

Compression

The hardware employs a number of techniques to **reduce memory traffic** for both depth and color render targets in a transparent scheme to all units

To benefit, **avoid scatter writes**, since algorithms prefer to look at a “tile” to find lossless compression opportunities



Maxwell
3rd Generation
Delta Color Compression

PRACTICAL CONSEQUENCES

UTILIZATION

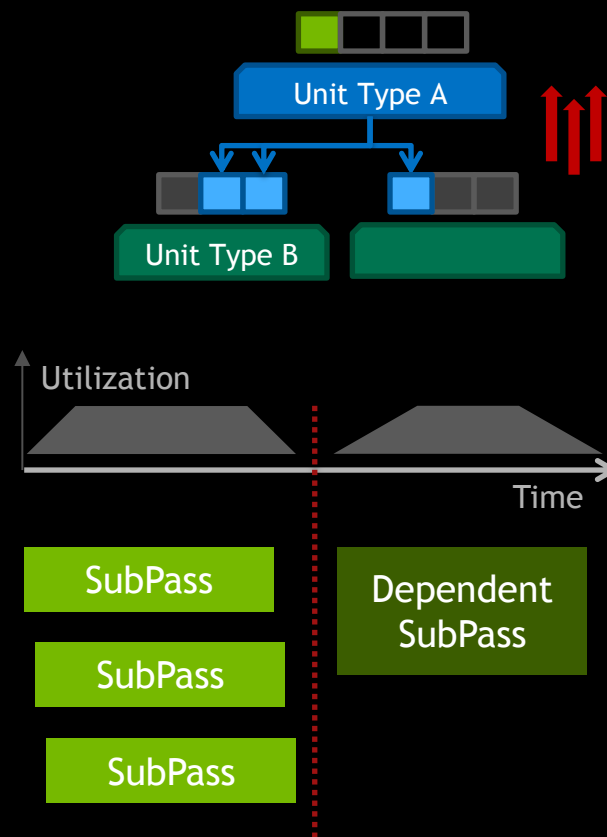
Front-End-limited: vkGraphicsPipeline switch can be expensive

Shader-limited: the switch could be free, allowing more specialized shaders

At high resolutions, the **pixel shader eventually dominates** all other bottlenecks.

Avoiding idle barriers in Vulkan

- **Subpass allows** the GPU to work on different render target in **parallel**
- Just ensure that you **don't create a false dependency** with input attachments

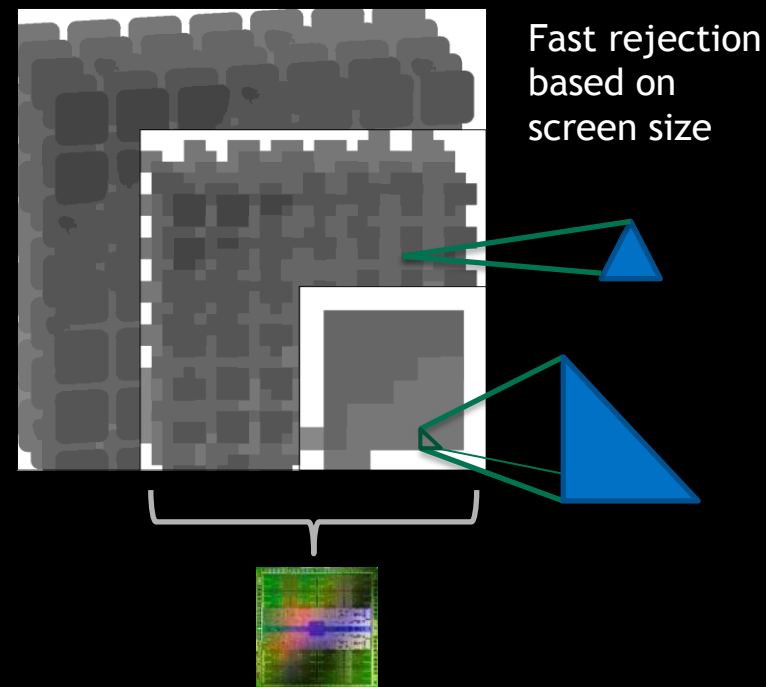
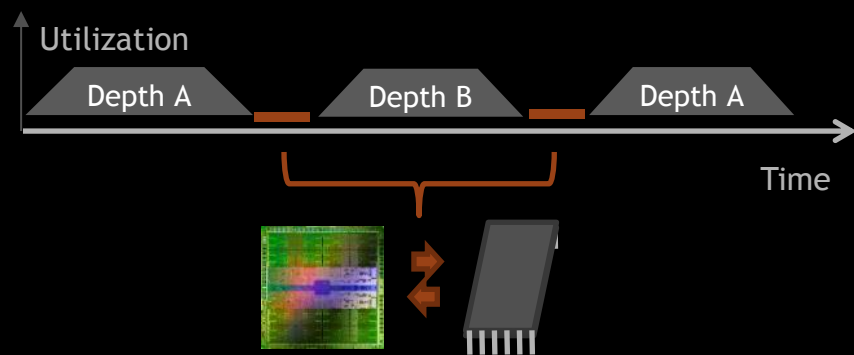


UTILIZATION

Depth buffer rendering

Hierarchical depth buffer is cached on-chip for Z-Cull

As a result, **switching between depth buffers** often will **need in & out copy**, increasing “idle” time



- Alternatively use „non-depth“ SubPasses inbetween to keep data on-chip

STATE MACHINE

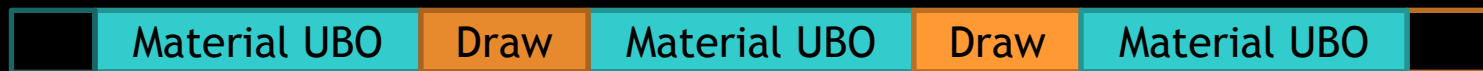
Redundant State Changes

Redundant state less of a problem on CPU-side in modern APIs with pre-validation

GPU still has to process the commands, hence state filtering and sorting remain beneficial

```
glDrawCommandsStatesNV  
(... stateobjects[] ...);
```

```
VkSubmitInfo info = {...};  
info.pCommandBuffers = cmdbuffers;  
vkQueueSubmit (queue,1, &info..)
```



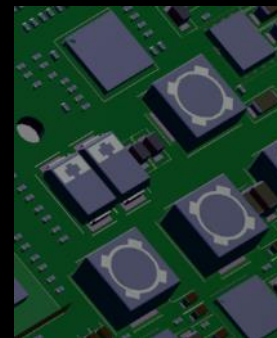
CPU time: 5 us

GPU time: 1.9 ms



CPU time: 5 us

GPU time: 0.9 ms



Graphicscard model
68k drawcalls

STATE MACHINE

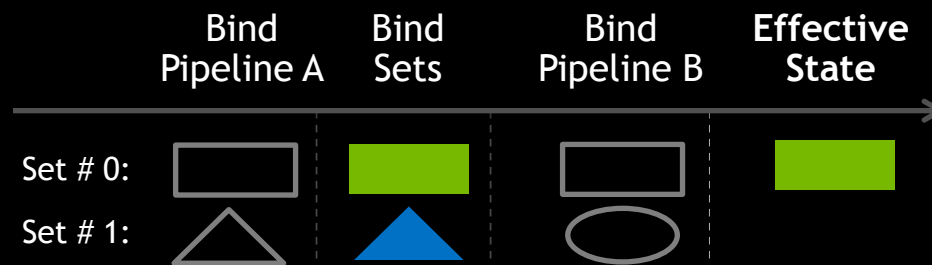
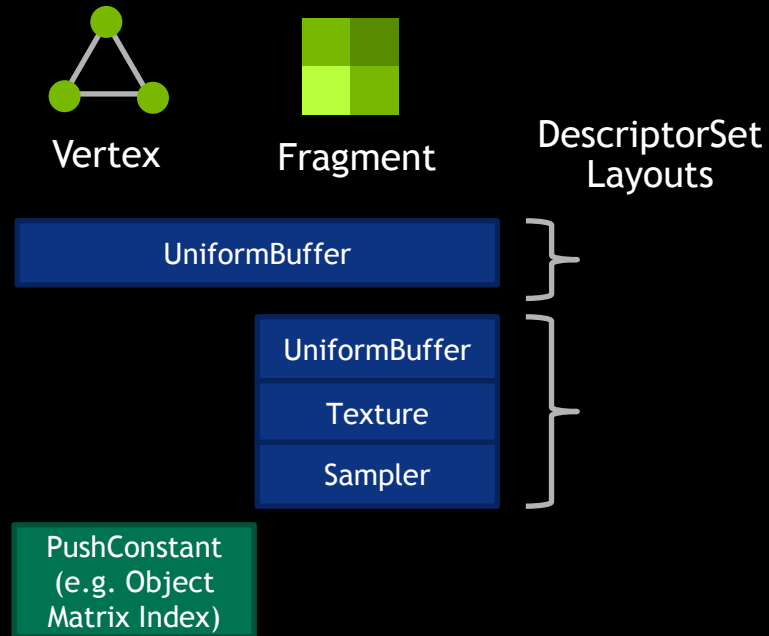
Vulkan Bindings

Be precise about stage resources, which buffers/samplers are used in which

Balance use of bindings/pushconstants to utilize different hardware mechanisms

Prefer single stage pushconstant

Organize DescriptorSetLayouts to keep them bound across pipeline changes



REDUCING CPU TRANSFERS

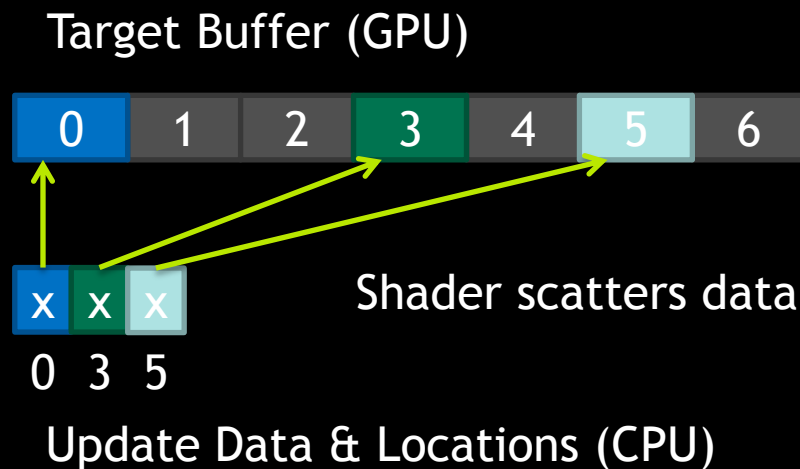
Compact Updates

Updating memory from the CPU is serial:

- Many small updates will not get much overlap, and suffer from PCIe latency

Instead

- **Collect updates via staging buffer** in host memory and scatter via shaders
- For one-shot use, data can be fetched from CPU resident buffer directly
- Vulkan: **Avoid VK_IMAGE_TILING_LINEAR** for GPU resident resources, large performance penalty



REDUCING CPU TRANSFERS

Incremental Changes

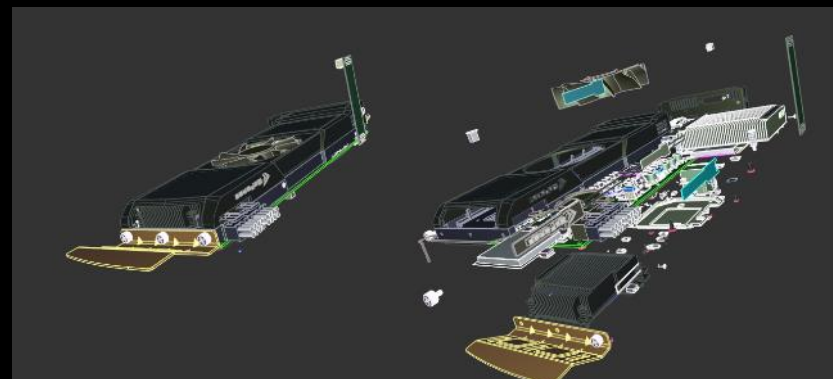
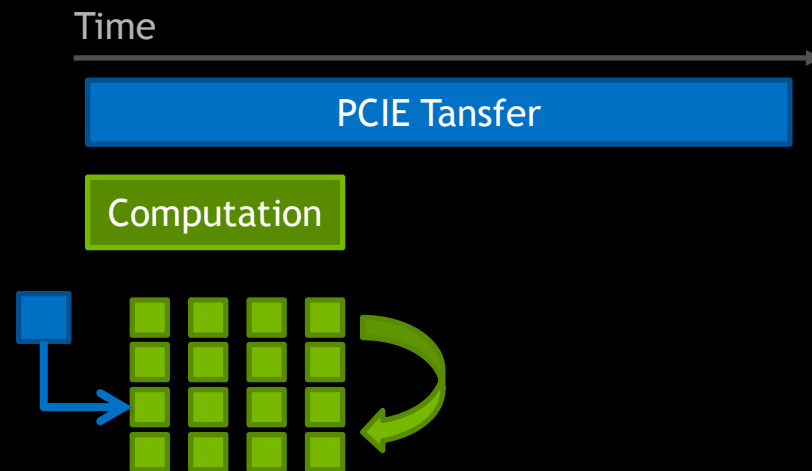
By computing data directly on the GPU, we need to send less data

Sufficient to send time values for animation or modified individual nodes

Hierarchical transform propagation on GPU

What if double is needed?

- Matrix updates in double
- Concat object & view matrix in double
- Store as float for vertex transforms



COMPUTING EFFICIENCY

Small Work

Launch overhead of compute dispatch not negligible for < 10 000 threads

Use Vertex-Shader to do compute work via GL_RASTERIZER_DISCARD and overlap with graphics (multi-invocation Geometry-Shaders are also fast)

For many threads keep using compute.

Use ARB_shader_ballot and NV_shader_thread_shuffle to pass data between threads, avoiding shared memory

```
vec3 posA = getPosition ( gl_ThreadInWarpNV + ...)
vec3 posB = shuffleUpNV (posA, 1, gl_WarpSizeNV);
```

```
... "Compute" alternative for few threads
if (numThreads < FEW_THREADS){
    glUseProgram( vs );
    glEnable      ( GL_RASTERIZER_DISCARD );
    glDrawArrays( GL_POINTS, 0, numThreads );
    glDisable     ( GL_RASTERIZER_DISCARD );
}
else {
    glUseProgram( cs );
    numGroups = (numThreads+GroupSize-1)/GroupSize;
    glUniformi1 (0, numThreads);
    glDispatchCompute ( numGroups, 1, 1 );
}
```

```
... Shader
#if USE_COMPUTE
    layout (local_size_x=GROUP_SIZE) in;
    layout (location=0) uniform int numThreads;
    int threadID = int( gl_GlobalInvocationID.x );
#else
    int threadID = int( gl_VertexID );
#endif
```

COMPUTING EFFICIENCY

Divergent Control Flow

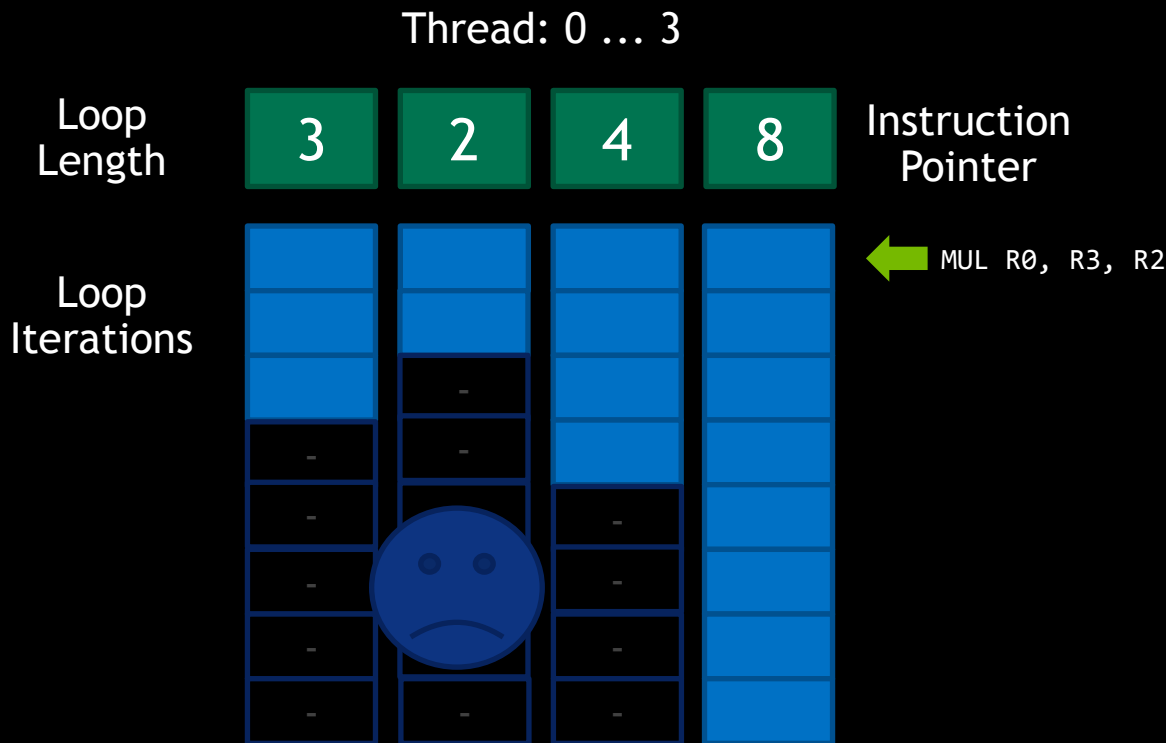
SIMT (Single Instruction Multiple Threads)

Processes threads in lock-step, common instruction pointer (masks out inactive)
NVIDIA: 1 warp = 32 threads

Longest for-loop will block progress on other threads in warp

If / else may execute both branches serially if condition was divergent

Positive: can communicate and access within warp data (e.g. ddx/ddy)



Check „Blueprint Rendering“ presentation
for alternative solution

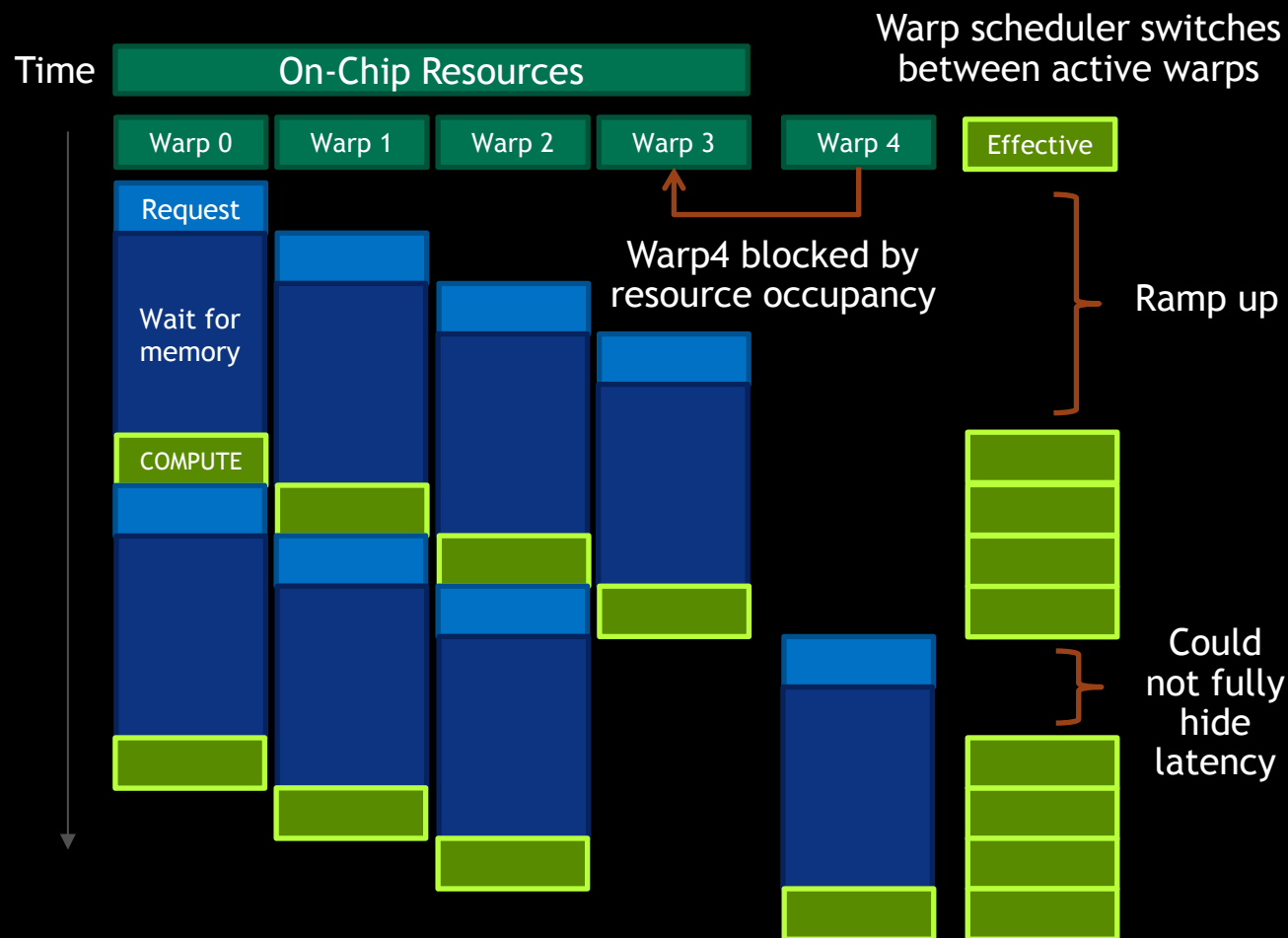
LATENCY HIDING

Throughput is what matters
 $\text{throughput} = \text{occupancy} / \text{latency}$

Memory accesses are typically the
highest latency operations

If shaders use too much on-chip
resources, then lower occupancy

- Registers
- Cross-Stage Data
- Vertex Input Attributes



LATENCY HIDING

Cross-Stage Data

Passing a lot of data across shader stages negatively impacts number of threads run (gl_ClipDistance also counts into this)

Prefer specialized shaders, over large übershaders, to minimize data

Re-computing in the fragment shader can be faster sometimes

Particularly high polygon scenes can suffer from this, due to sheer amount of vertex data

```
in Interpolants {  
    vec3    oPos;  
    vec3    oNormal;  
    vec3    oTangent;  
    vec3    wPos;      // o'rlly?  
    vec3    wNormal;   // o'rlly?  
    vec3    wTangent;  // o'rlly?  
    vec2    tex;  
    vec4    color;  
} IN;
```

```
...  
// bad  
if ( ubo.hasNormalmap) {  
    ... Using IN.oTangent ..  
}  
// better, let compiler remove unused data  
#if HAS_NORMALMAP  
    ... Using IN.oTangent ..  
#endif
```

```
// could be bottleneck  
wLightDir = wPos - ubo.wLightPos;  
// recalculate from other data  
wLightDir = ubo.oMatrix * vec4 (oPos,1) -  
...
```


LATENCY HIDING

Texturing

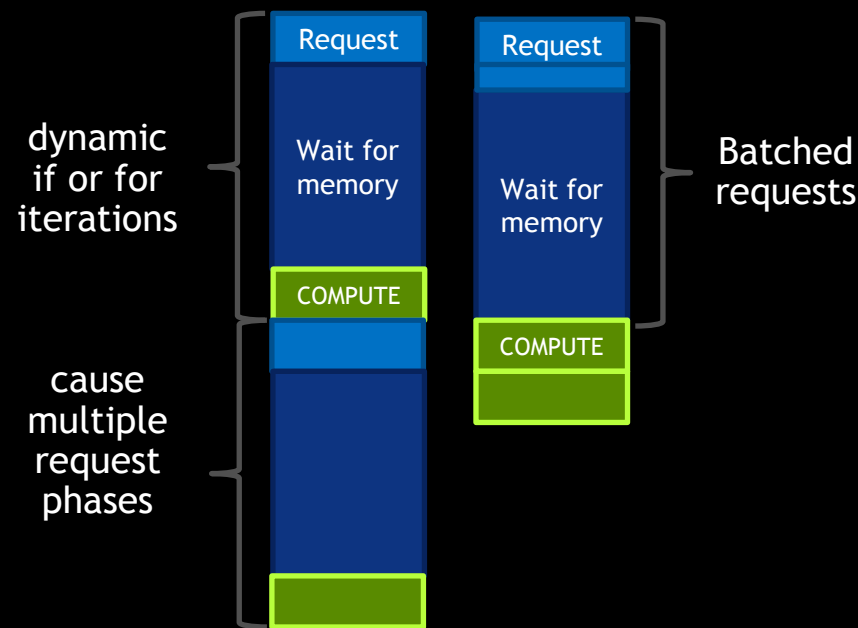
In the pixel shader, **favor batched memory access** (many textures fetched at once)

Lots of run-time branching & dynamic loops, hinders this compiler optimization

Aid compiler by grouping fetches & organizing dependent reads

Generate shaders with popular compile-time loop-counts by making use of pre-processor or Vulkan specialization constants

Use `textureGatherOffsets` if possible
(offset functions express locality, gather good for reading more data at once)



MEMORY TYPES

Fetching Buffer Data

Constant
Cache

UNIFORM BUFFER

Fastest for uniform, bad
for divergent access

```
uniform ubo {
    bool blah;
    vec4 data[128];
};

// okay, single load for warp
if (blah) ... = data[0];
else     ... = data[1];

// not good,
// serializes loads within warp
... = data[thread_specific];
```

L1/L2
Cache

TEXTURE BUFFER

Good for generic access

Address math handled by
texture unit

Also handles format
conversion

```
uniform samplerBuffer tbo;

// tbo could be FP16 or FP32
texelFetch(tbo,
    thread_specific);
```

L1/L2
Cache

SHADER STORAGE BUFFER

Good for generic access, can
be a tad slower than texture

Address math in shader and
64-bit registers

Format conversion in shader

Robust contexts add out-of-
bounds checking in shader

```
buffer ssbo {
    vec4 data[];
};
```

Can balance SSBO/TBO use to avoid hammering TextureUnits

WORK REDUCTION

Multi Draw Indirect, NV command list

When rendering lots of geometry, can become drawcall, overdraw and triangle limited again

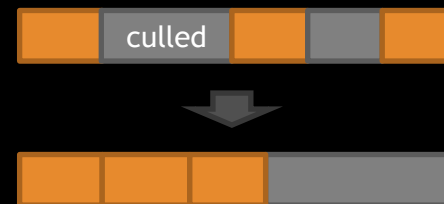
Remove unnecessary detail on the fly

Multi Draw Indirect (Vulkan & GL) and NV_command_list provide ways to implement level of detail or efficient culling

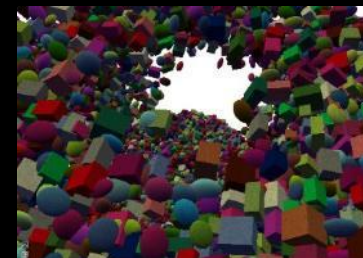
Prefer compact command buffers

No CPU interaction means **no frame-latency**, great for VR
(see demo in VR Village 52+ million triangle car model)

```
DrawElements {  
    Gluint    count;  
    Gluint    instanceCount;  
    Gluint    firstIndex;  
    Gluint    baseVertex;  
    Gluint    baseInstance;  
}  
  
buffer commands {  
    DrawElements cmds[];  
}
```



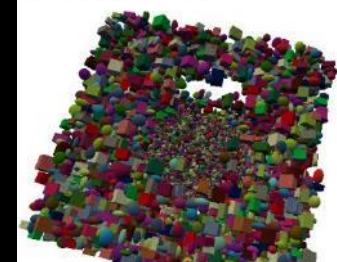
Compaction avoids
pipeline bubbles



"Frozen" culling result of above view



All objects rendered



https://github.com/nvpro-samples/gl_occlusion_culling

GEOMETRY CREATION

Tessellation, Geometry Shader, Draw Indirect

TESSELLATION

Hardware can allocate needed space on-demand and distribute Tess. Eval Shader work across SMs

All data produced & consumed on chip



GEOMETRY SHADER

Needs to pre-allocate worst-case output data in L1

Therefore can negatively reduce amount of parallel threads

```
layout(max_vertices=4) out;  
// avoid high counts  
// and low output ratios
```

FAST GEOMETRY SHADER

`NV_geometry_shader_passthrough`

Works on the Vertex-Shader data directly

No extra stage/storage

Send same primitive to multiple viewports, or discard it

```
gl_ViewportMask[0] = 0;  
// culls primitive
```

GEOMETRY CREATION

Tessellation, Geometry Shader, Draw Indirect

VERTEX SHADER

Generate geometry based purely on `gl_VertexID`

May use custom fetching or even generate index-buffer on the GPU

Use „shuffle“ to access other vertices of same primitive

```
gl_Position.x = gl_VertexID * scale;
```

DRAW INDIRECT

Use Draw Indirect to generate variable amount of data

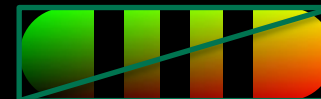
Preferably avoid low primitive counts (risk of being FrontEnd-limited)

```
DrawArrays {  
    Gluint    count;  
    ...  
}
```

PIXEL SHADER

Use discard and distance fields to clip geometry

Compute coverage analytically for MSAA



CONCLUSION

Make use of all the power 😊

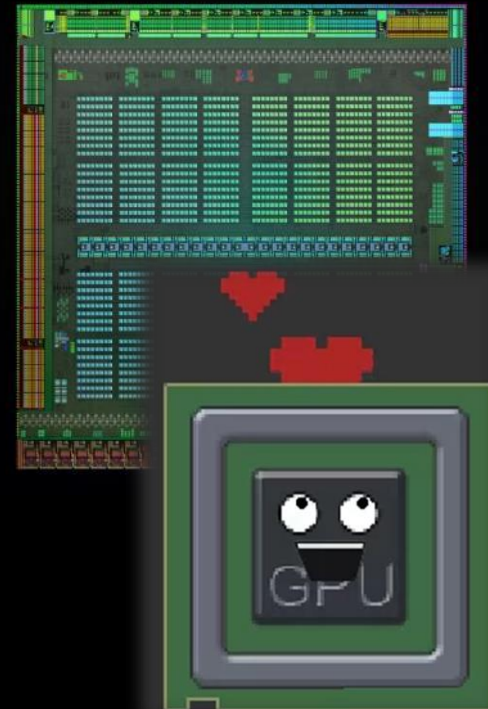
Investigate where to put optimization effort in

Balance different hardware units for optimal efficiency, avoiding pipeline „bubbles“

Generate & update data on the GPU itself

Use modern API mechanisms!

Makes a #HappyGPU and ideally a happy you



Courtesy of Simon Trümpler

REFERENCES

Life of a triangle article

<https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline>

Illustrated & animated run through the graphics pipeline

<https://simonschreibt.de/gat/renderhell-book2/>

In-depth look at SM processing

<http://on-demand.gputechconf.com/gtc/2013/presentations/S3466-Programming-Guidelines-GPU-Architecture.pdf>

Past GTC presentations

<http://on-demand.gputechconf.com/gtc/2013/presentations/S3032-Advanced-Scenegraph-Rendering-Pipeline.pdf>

<http://on-demand.gputechconf.com/siggraph/2014/presentation/SG4117-OpenGL-Scene-Rendering-Techniques.pdf>




<http://on-demand.gputechconf.com/gtc/2015/presentation/S5135-Christoph-Kubisch-Pierre-Boudier.pdf>

GPU TECHNOLOGY
CONFERENCE

April 4-7, 2016 | Silicon Valley

THANK YOU

JOIN THE CONVERSATION

#GTC16   

JOIN THE NVIDIA DEVELOPER PROGRAM AT developer.nvidia.com/join

ckubisch@nvidia.com @pixeljetstream
pboudier@nvidia.com @pboudier09

PRESENTED BY



LE CHIP

GM204

4 Graphics Processing Clusters (GPC)

16 Streaming Multiprocessors (SM)

2048 CUDA Cores

64 Raster Output (ROP) Units

128 Texture Units

16 Geometry Engines

4 Raster Engines

