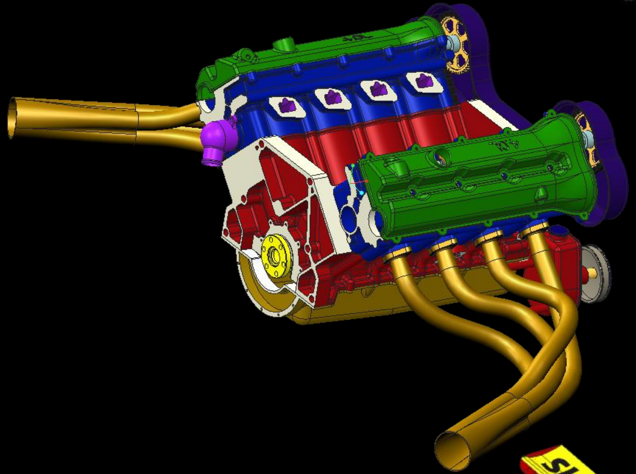
GPU TECHNOLOGY CONFERENCE

Advanced Scenegraph Rendering Pipeline

Markus Tavenrath - NVIDIA - matavenrath@nvidia.com
Christoph Kubisch - NVIDIA - ckubisch@nvidia.com

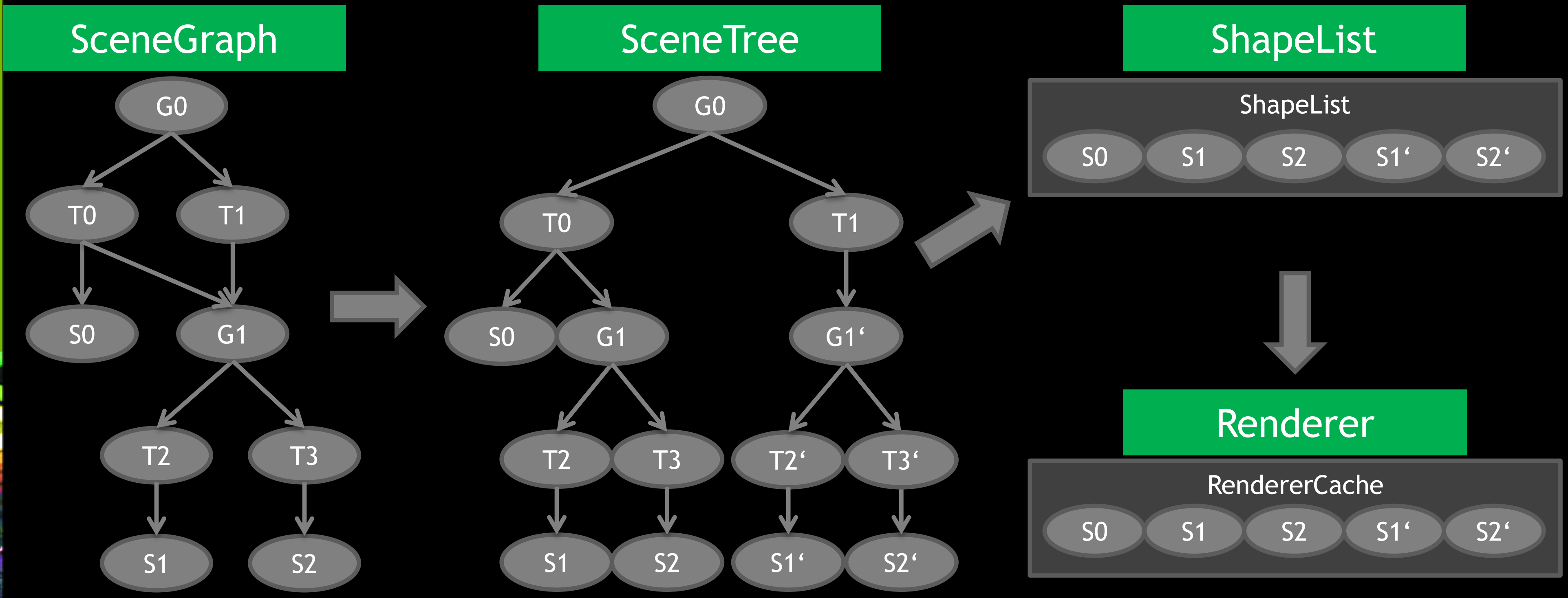
SceneGraph Rendering

- Traditional approach is render while traversing a SceneGraph
- Scene complexity increases
  - Deep hierarchies, traversal expensive
  - Large objects split up into a lot of little pieces, increased draw call count
  - Unsorted rendering, lot of state changes
- CPU becomes bottleneck when rendering those scenes



models courtesy of PTC

Overview



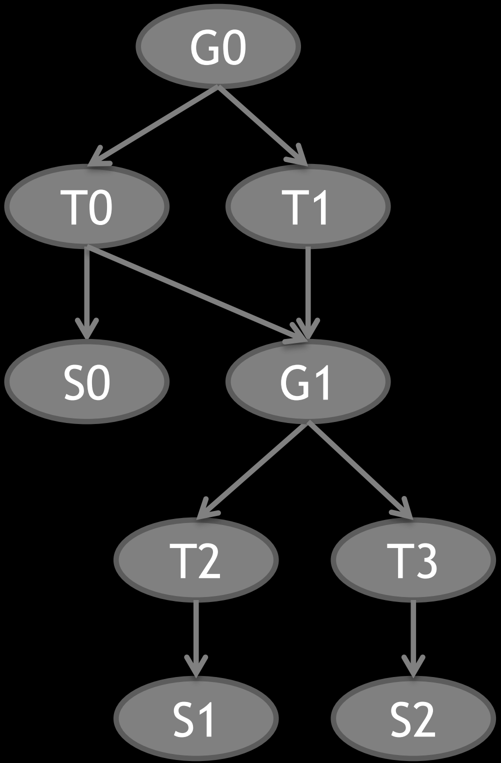
Gi Group

Ti Transform

Si Shape

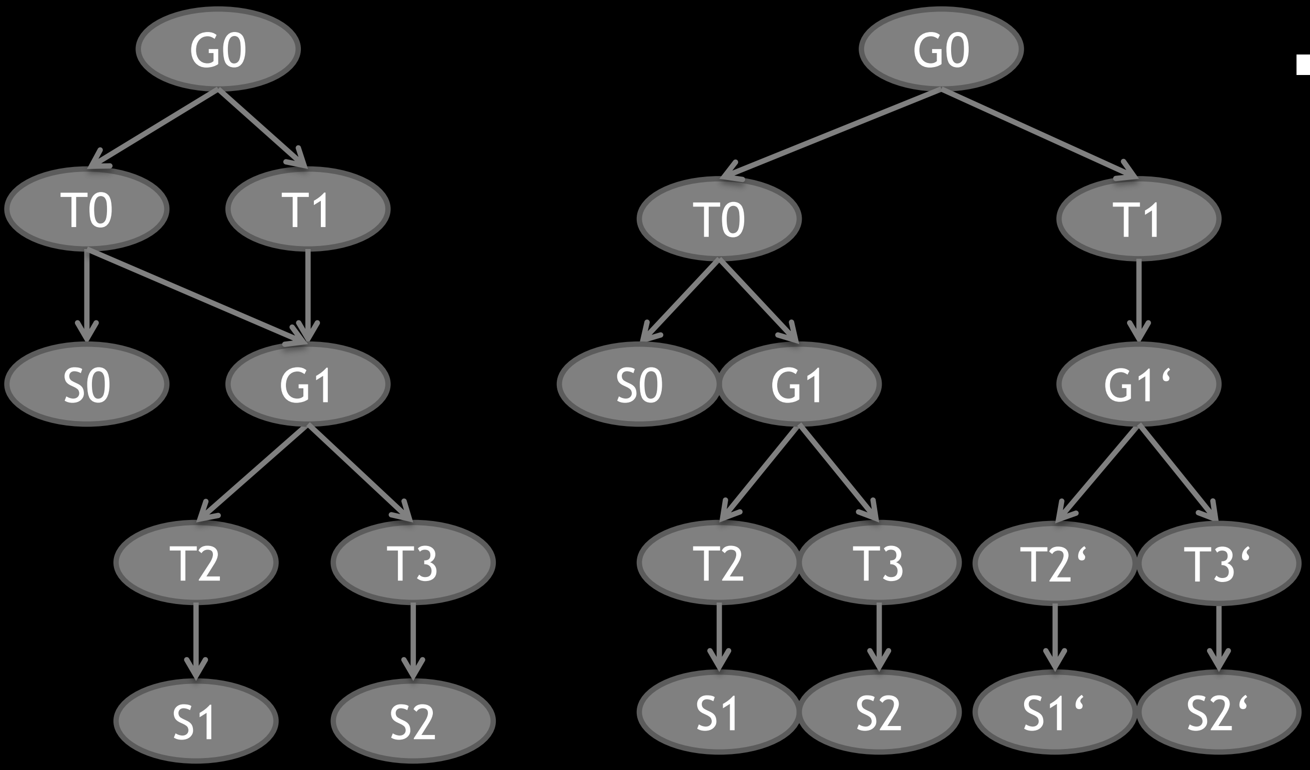
GPU TECHNOLOGY CONFERENCE

SceneGraph



- SceneGraph is DAG
- No unique path to a node
  - Cannot efficiently cache path-dependent data per node
- Traversal runs over 14 nodes for rendering.
- Processed 6 Transform Nodes
  - 6 matrix/matrix multiplications and inversions
- Nodes are usually ‚large‘ and not linear in memory
  - Each node access generates at least one, most likely cache misses

SceneTree construction

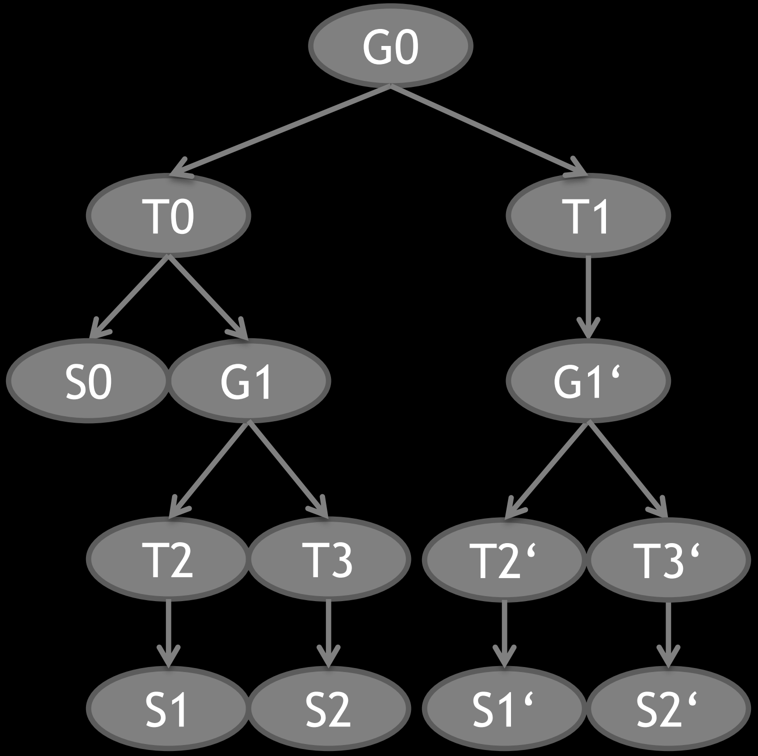


- Observer based synchronization

```
G0 -> (G0)
T0 -> (T0)
T1 -> (T1)
S0 -> (S0)
G1 -> (G1,G1')
T2 -> (T2,T2')
S1 -> (S1,S1')
T3 -> (T3,T3')
S2 -> (S2,S2')
```

Gi Group | Ti Transform | Si Shape

SceneTree

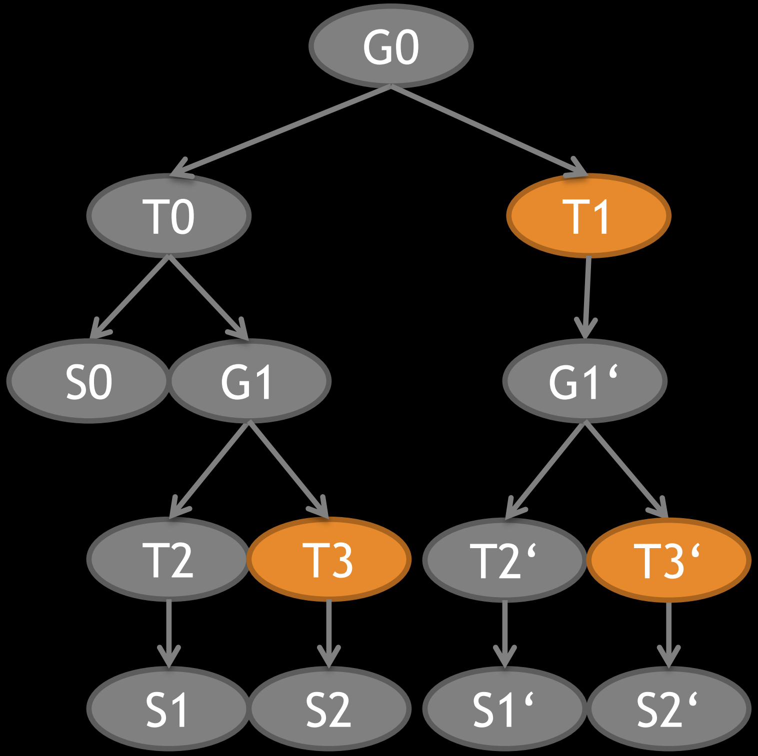


- SceneTree has unique path to each node
- Store accumulated attributes like transforms or visibility in each Node
- Trade memory for performance
  - 64-byte per node, 100k nodes ~6MB
  - Transforms stored separate vector
- Traversal still processes 14 nodes.

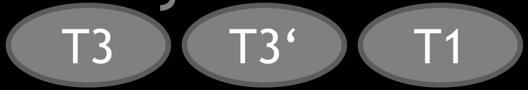
Gi Group Ti Transform Si Shape

GPU TECHNOLOGY CONFERENCE

SceneTree invalidate attributes cache



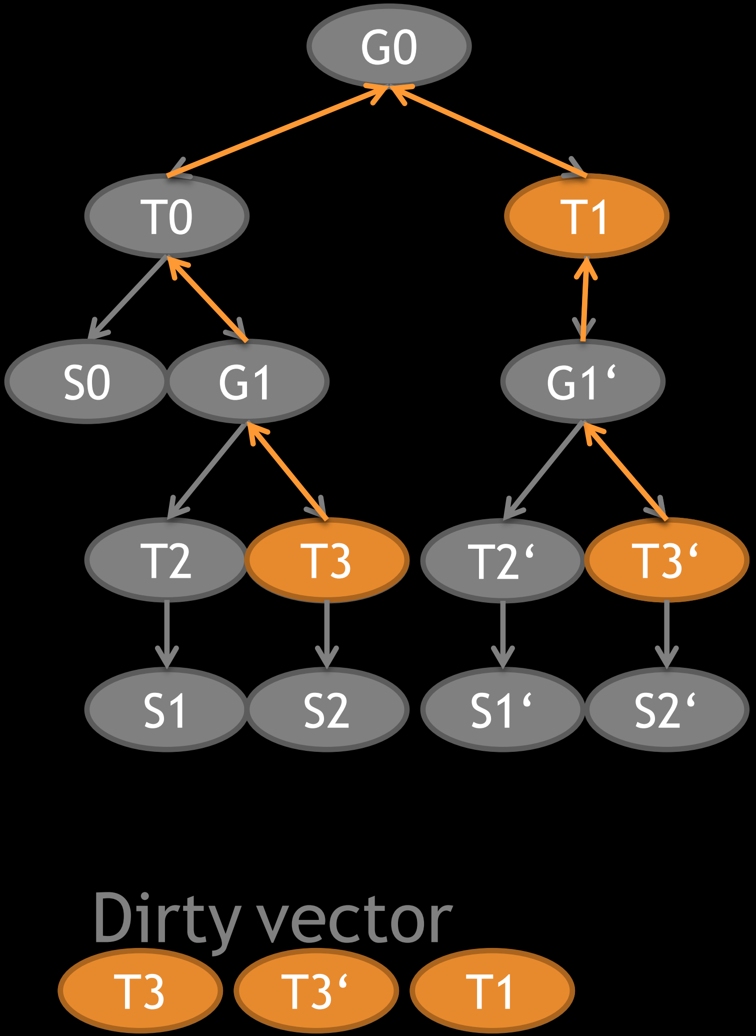
Dirty vector



- Keep dirty flags per node
- Keep dirty vector per flag
- SceneGraph change notifications invalidated nodes
  - If not dirty, mark dirty and add to dirty vector
  - O(1) operation, no sorting required upon changes
- Before rendering a frame process dirty vector

GPU TECHNOLOGY CONFERENCE

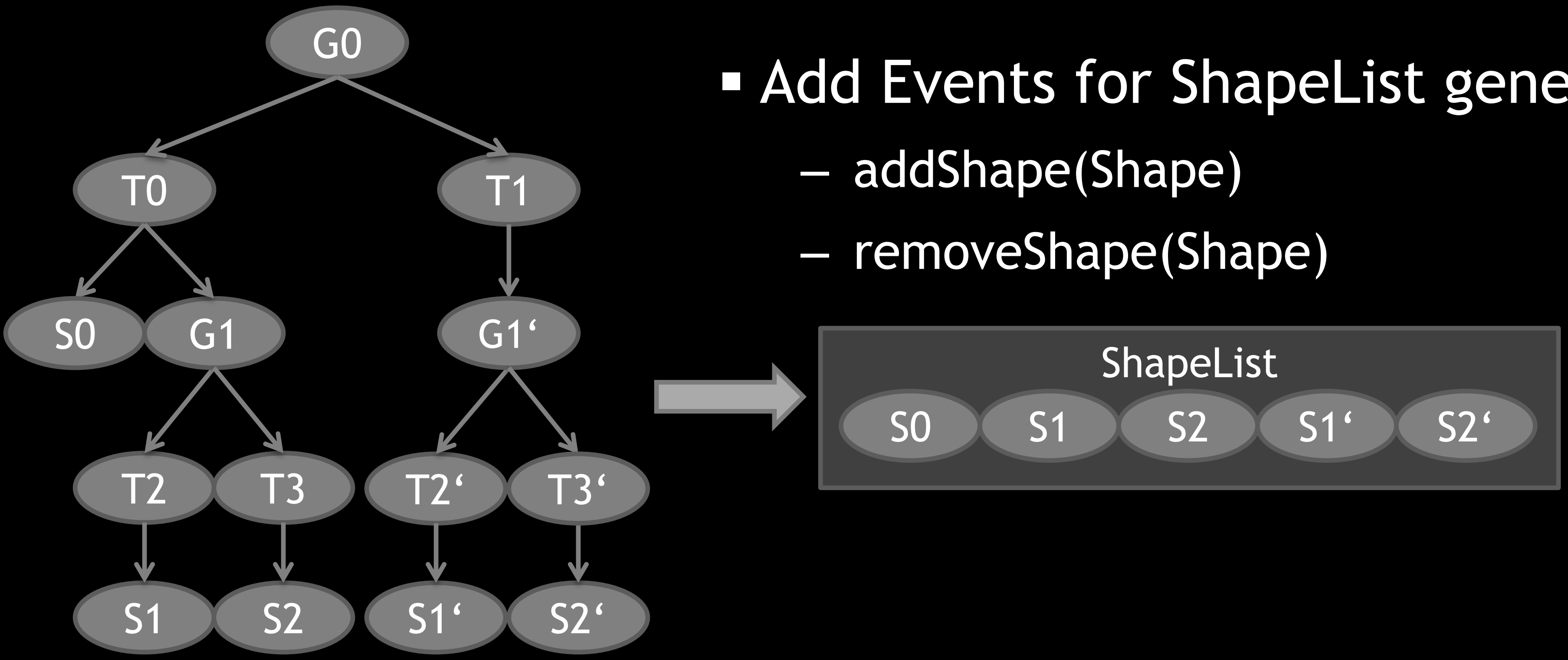
SceneTree validate attribute cache



- Walk through dirty vector
  - Node marked dirty -> search top dirty
  - Validate subtree from top dirty
- Validation example
  - T3 dirty, traverse up to root node
    - T3 top dirty node, validate T3 subtree
  - T3‘ dirty, traverse up to root node
    - T1 top dirty node, validate T1 subtree
  - T1 not dirty
    - No work to do

GPU TECHNOLOGY CONFERENCE

SceneTree to ShapeList



- Add Events for ShapeList generation
  - addShape(Shape)
  - removeShape(Shape)

Summary

- SceneGraph to SceneTree synchronization
  - Store accumulated data per node instance
- SceneTree to ShapeList synchronization
  - Avoid SceneTree traversal
- Next: Efficient data structure for renderer based on ShapeList

Renderer Data Structures

Shape
- Program ‚colored‘ | Geometry Shape1
- ParameterData Camera1
- ParameterData Lightset 1
- ParameterData Transform 1
- ParameterData red

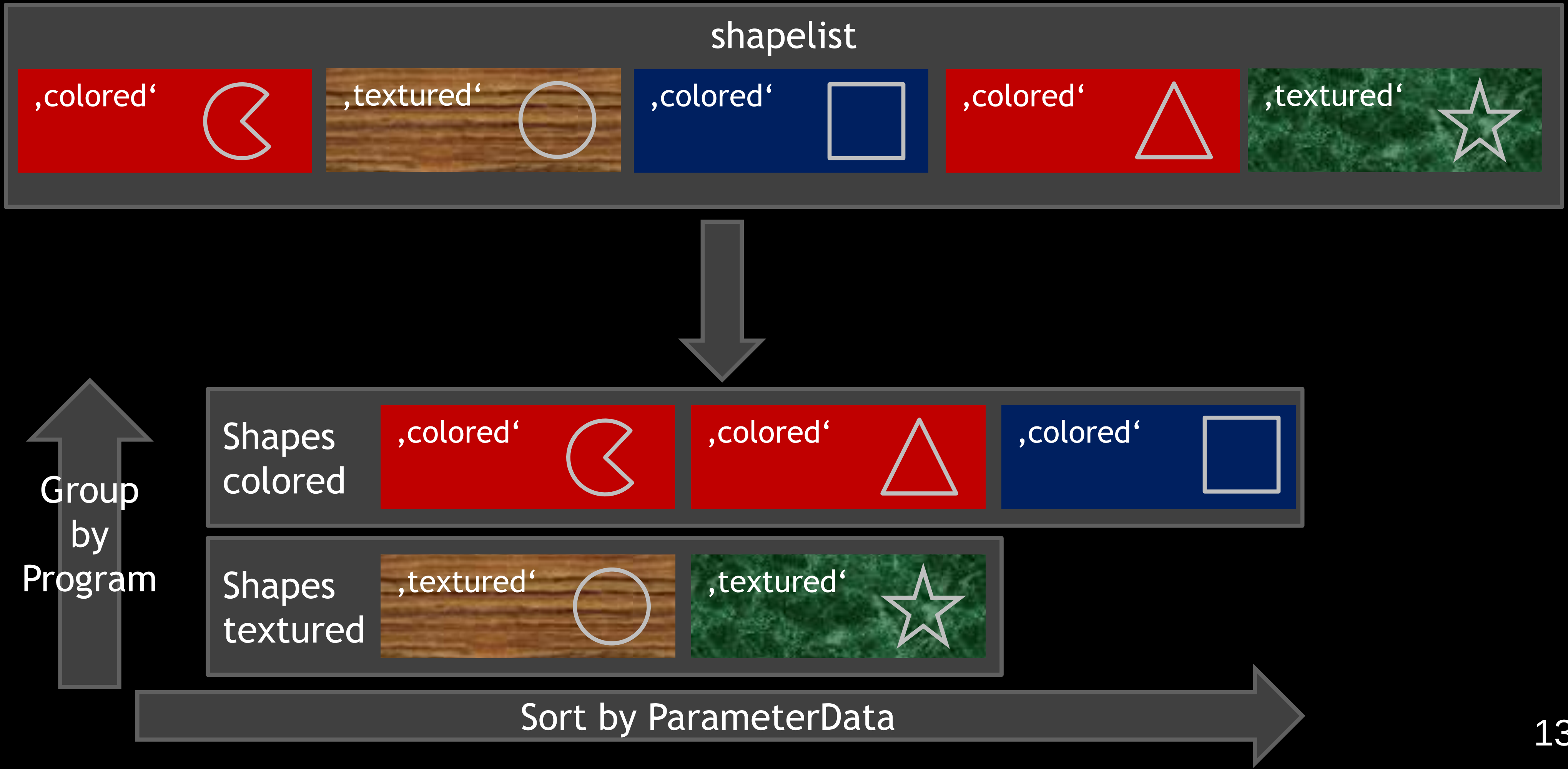
Program
- Shader Vertex | Shader Fragment
- ParameterDescription Camera
- ParameterDescription Light
- ParameterDescription Matrices
- ParameterDescription Material

ParamaterDescription		
Name	Type	Arraysize
ambient	vec3	2
diffuse	vec3	2
specular	vec3	2
texture	Sampler	0

Example Parameter Grouping

Parameters	Frequency
Shader independent globals, i.e. camera	constant
Shader dependent globals, i.e. environment map	
Light, i.e. light sources and shadow maps	rare
Material raw values, i.e. float, int and bool	frequent
Material handles, i.e. textures and buffers	
Object parameters, i.e. position/rotation/scaling	always

Rendering structures



ParameterData Cache

Parameters colored	red	blue

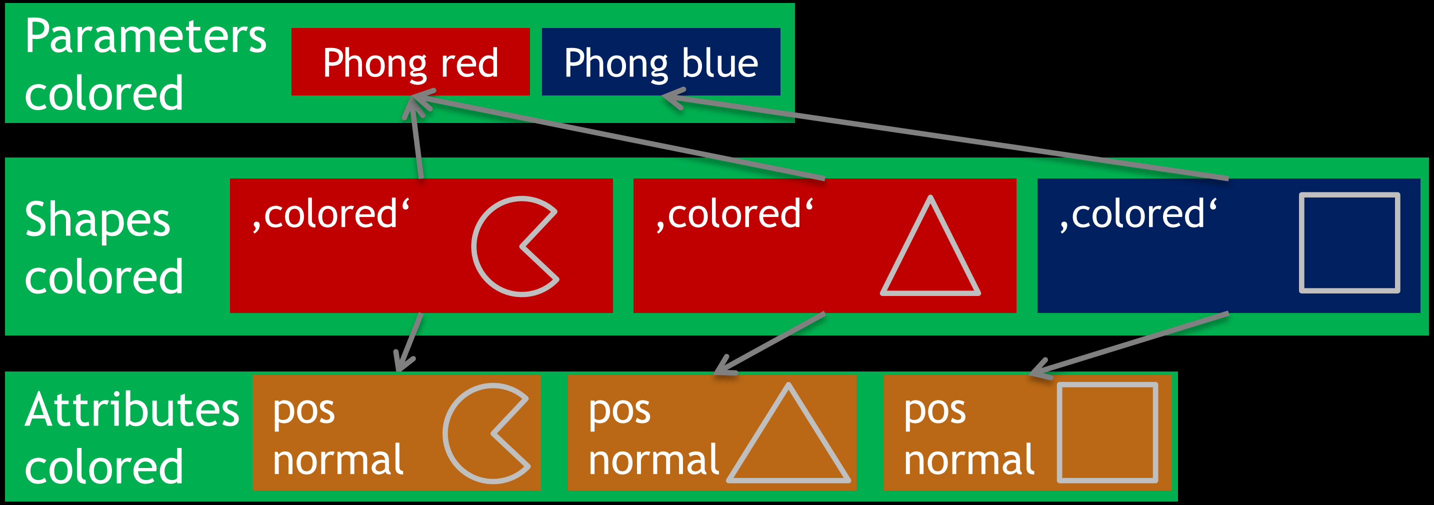
Parameters textured	wood	marble

- Cache is a big char[] with all ParameterData.
- ParameterData are sorted by first usage.
- Parameters are converted to Target-API datatype, i.e.
  - Int8 to int32, TextureHandle to bindless texture...
- Updating parameters is only playback of data in memory, no conditionals.
- Filter for used parameters to reduce cache size

Vertex Attribute Cache

- Big char[] with vertex attribute pointers
  - Bindless pointers, VBOs or VAB streams
- Each set of attributes stored only once
- Ordered by first usage
- Attributes required by program are known
  - Store only used attributes in Cache
  - Useful for special passes like depth pass where only pos is required

Renderer Cache complete




```
foreach(shape) {
  if (visible(shape)) {
    if (changed(parameters)) render(parameters);
    if (changed(attributes)) render(attributes);
    render(shape);
  }
}
```

Achievements

- CPU boundedness improved (application)
  - Recomputation of attributes (transforms)
  - Deep hierarchies: traversal expensive
  - Unsorted rendering, lot of state changes
- CPU boundedness remaining (OpenGL usage)
  - Large objects split up into a lot of little pieces, increased draw call count

SceneTree

ShapeList

Renderer

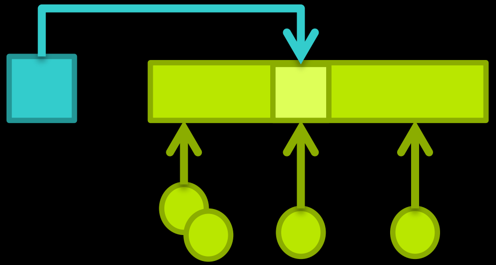
RendererCache

Renderer

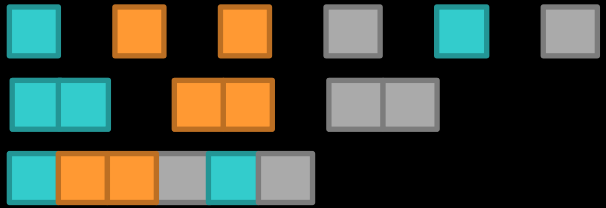
OpenGL implementation

Enabling Hardware Scalability

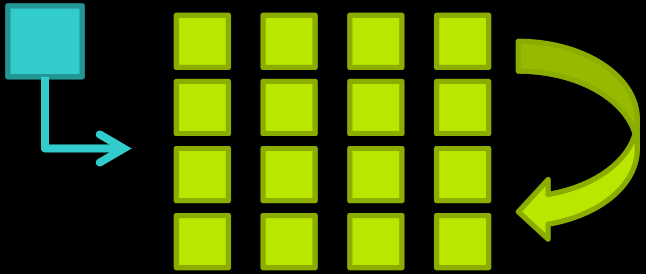
- Avoid data redundancy
  - Data stored once, referenced multiple times
  - Update only once (less host to gpu transfers)



- Increase batching potential
  - Further cuts api calls
  - Less driver CPU work



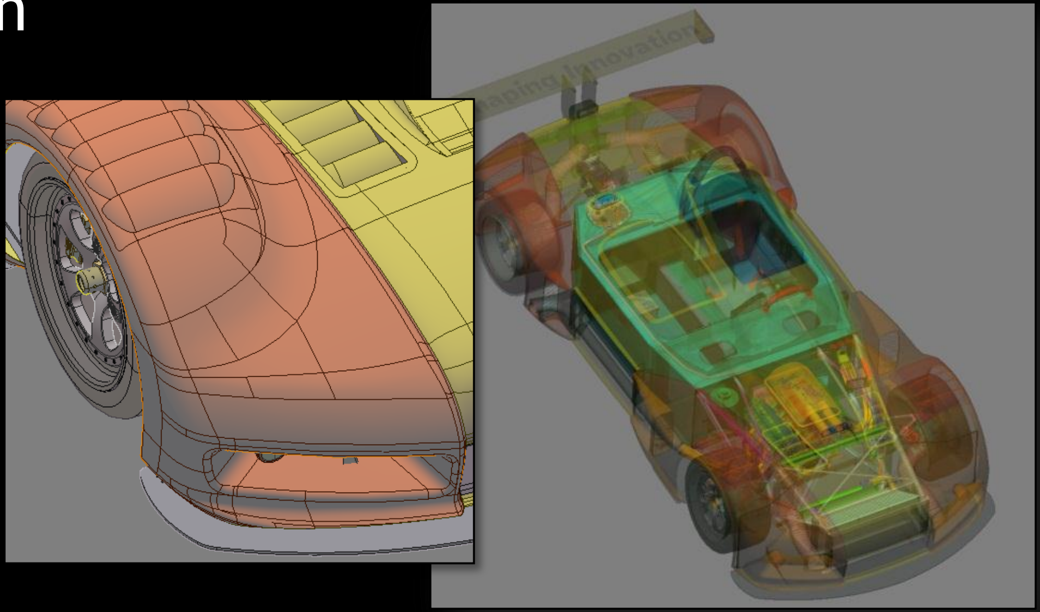
- Minimize CPU/GPU interaction
  - Allow GPU to update its own data
  - Lower api usage when scene is changed little
  - E.g. GPU-based culling



OpenGL Research Framework

- Avoids classic SceneGraph design
- Geometry
  - Vertex/IndexBuffer
  - BoundingBox
  - Divided into parts (CAD features)
- Material
- Node Hierarchy
- Object
  - Node and Geometry reference
  - For each Geometry part
    - Material reference
    - Enabled state

model courtesy of PTC



- 99000 total parts, 3.8 Mtris, 5.1 Mverts
- 700 geometries, 128 materials
- 2000 objects

Performance baseline

- Kepler Quadro K5000, i7
- vbo bind and drawcall per part, i.e. 99 000 drawcalls

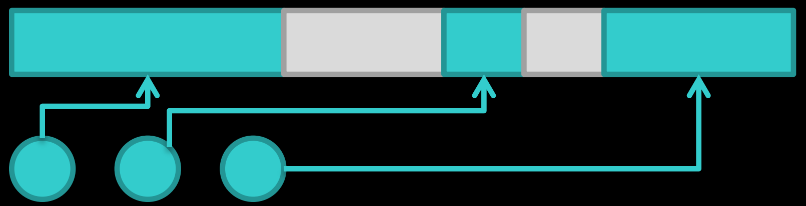
  scene draw time > 38 ms (CPU bound)
- vbo bind per geometry, drawcalls per part

  scene draw time > 14 ms (CPU bound)
- All subsequent techniques raise perf significantly

  scene draw time < 6 ms

  1.8 ms with occlusion culling

Drawcall Reduction



- MultiDraw (1.x)
  - Render ranges from current VBO/IBO
  - Single drawcall for many distinct objects
  - Reduces overhead for low complexity objects
- ARB_draw_indirect (4.x)
- ARB_multi_draw_indirect
  - Store drawcall information on GPU or HOST
  - Let GPU create/modify GPU buffers

```
DrawElementsIndirect
{
   GLuint        count;
   GLuint        instanceCount;
   GLuint        firstIndex;
   GLint         baseVertex;
   GLuint        baseInstance;
}
```

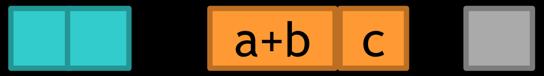
Drawing Techniques

- All use multidraw capabilites to render across gaps

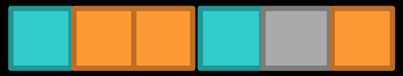
- BATCHED use CPU generated list of combined parts with same state
  - Object‘s part cache must be rebuilt based on material/enabled state

- INDIVIDUAL stay on per-part level
  - No caches, can update assignment or cmd buffers directly

Parts with different materials in geometry



Grouped and „grown“ drawcalls



Single call, encode material/matrix assignment via vertex attribute

Parameters

- Group parameters by frequency of change
- Generating shader strings allows different storage backend for „uniforms“

```
Effect "Phong {
  Group „material" (many) {
    vec4 "ambient"
    vec4 "diffuse"
    vec4 "specular"
  }
  Group „view" (few) {
    vec4 „viewProjTM„
  }
  Group „object" (many) {
    mat4 „worldTM„
  }
... Code ...
}
```

- OpenGL 2 uniforms
- OpenGL 3,4 buffers
- NVIDIA bindless technology...

Parameters

- GL2 approach:
  - Avoid many small uniforms
  - Arrays of uniforms, grouped by frequency of update, tightly-packed

```
uniform mat4 worldMatrices[2];

uniform vec4 materialData[8];

#define matrix_world   worldMatrices[0]
#define matrix_worldIT worldMatrices[1]

#define material_diffuse  materialData[0]
#define material_emissive materialData[1]
#define material_gloss    materialData[2].x

// GL3 can use floatBitsToInt and friends
// for free reinterpret casts within
// macros
...

  wPos = matrix_world * oPos;

...
// in fragment shader
  color = material_diffuse +
          material_emissive;
...
```

Parameters

- GL4 approach:
  – TextureBufferObject (TBO) for matrices
  – UniformBufferObject (UBO) with array data to save costly binds
  – Assignment indices passed as vertex attribute

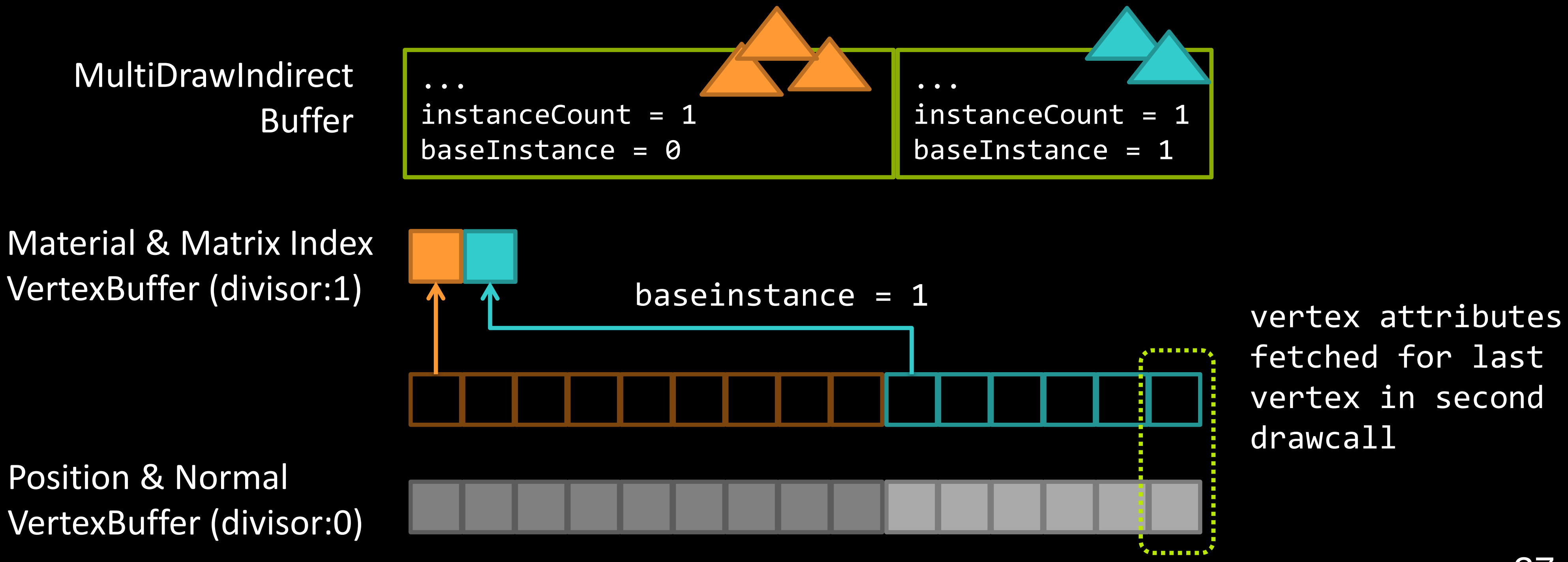
```
in vec4 oPos;

uniform  samplerBuffer matrixBuffer;

uniform  materialBuffer {
  Material materials[512];
};

in ivec2 vAssigns;
flat out ivec2 fAssigns;

// in vertex shader
  fAssigns = vAssigns;

  worldTM = getMatrix (matrixBuffer,
                       vAssigns.x);

  wPos = worldTM * oPos;
...
// in fragment shader
  color = materials[fAssigns.y].color;
...
```

OpenGL 4.x approach

```
setupSceneMatrixAndMaterialBuffer (scene);

foreach (obj in scene) {
  if ( isVisible(obj) ) {

    setupDrawGeometryVertexBuffer (obj);

    // iterate over different materials used
    foreach ( batch in obj.materialCaches) {

      glVertexAttribI2i (indexAttr, batch.materialIndex, matrixIndex);

      glMultiDrawElements (GL_TRIANGLES, batch.counts, GL_UNSIGNED_INT ,
                           batch.offsets,batched.numUsed);
    }
  }
}
```

Per drawcall vertex attribute

```
glVertexAttribDivisor == 0 : VArray[ gl_VertexID + baseVertex ]
glVertexAttribDivisor != 0 : VArray[ gl_InstanceID / VDivisor + baseInstance ]
                             VArray[             0 /        1 + baseInstance ]
```



OpenGL 4.2+ indirect approach

```
...

foreach ( obj in scene.objects ) {
  ...

  // instead of glVertexAttribI2i calls and a loop
  // we use the baseInstance for the attribute

  // bind special assignment buffer as vertex attribute
  glBindBuffer ( GL_ARRAY_BUFFER, obj->assignBuffer);
  glVertexAttribIPointer (indexAttr, 2, GL_INT, . . . );

  // draw everything in one go
  glMultiDrawElementsIndirect ( GL_TRIANGLES, GL_UNSIGNED_INT,
                                obj->indirectOffset, obj->numIndirects, 0 );
}
```

Vertex Setup

- ARB_vertex_attrib_binding (VAB)
  - Avoids many buffer changes
  - Separates format from data
  - Bind multiple vertex attributes to one buffer
- NV_vertex_buffer_unified_memory (VBUM)
  - Allows very fast switching through GPU pointers

```
/* setup once, similar to glVertexAttribPointer
but with relative offset last */
glVertexAttribFormat (ATTR_NORMAL, 3,
    GL_FLOAT, GL_TRUE,  offsetof(Vertex,normal));
glVertexAttribFormat (ATTR_POS, 3,
    GL_FLOAT, GL_FALSE, offsetof(Vertex,pos));
// bind to stream
glVertexAttribBinding (ATTR_NORMAL, 0);
glVertexAttribBinding (ATTR_POS, 0);
```

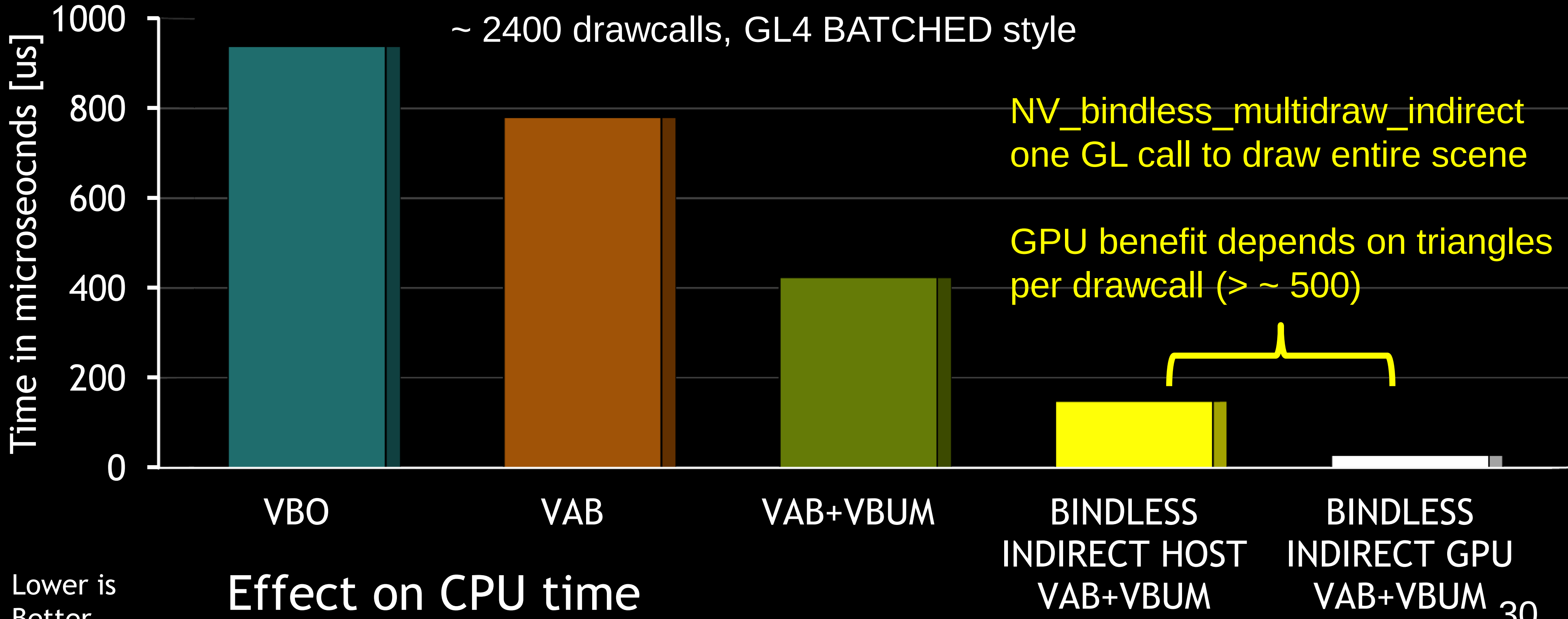
```
// switch single stream buffer
glBindVertexBuffer (0, bufID, 0, sizeof(Vertex));
```

```
// NV_vertex_buffer_unified_memory
// enable once and set stride
glEnableClientState (GL_VERTEX...NV);...
glBindVertexBuffer (0, 0, 0, sizeof(Vertex));
```

```
// switch single buffer via pointer
glBufferAddressRangeNV (GL_VERTEX...,0,bufADDR,
  bufSize);
```

NV_bindless_multidraw_indirect

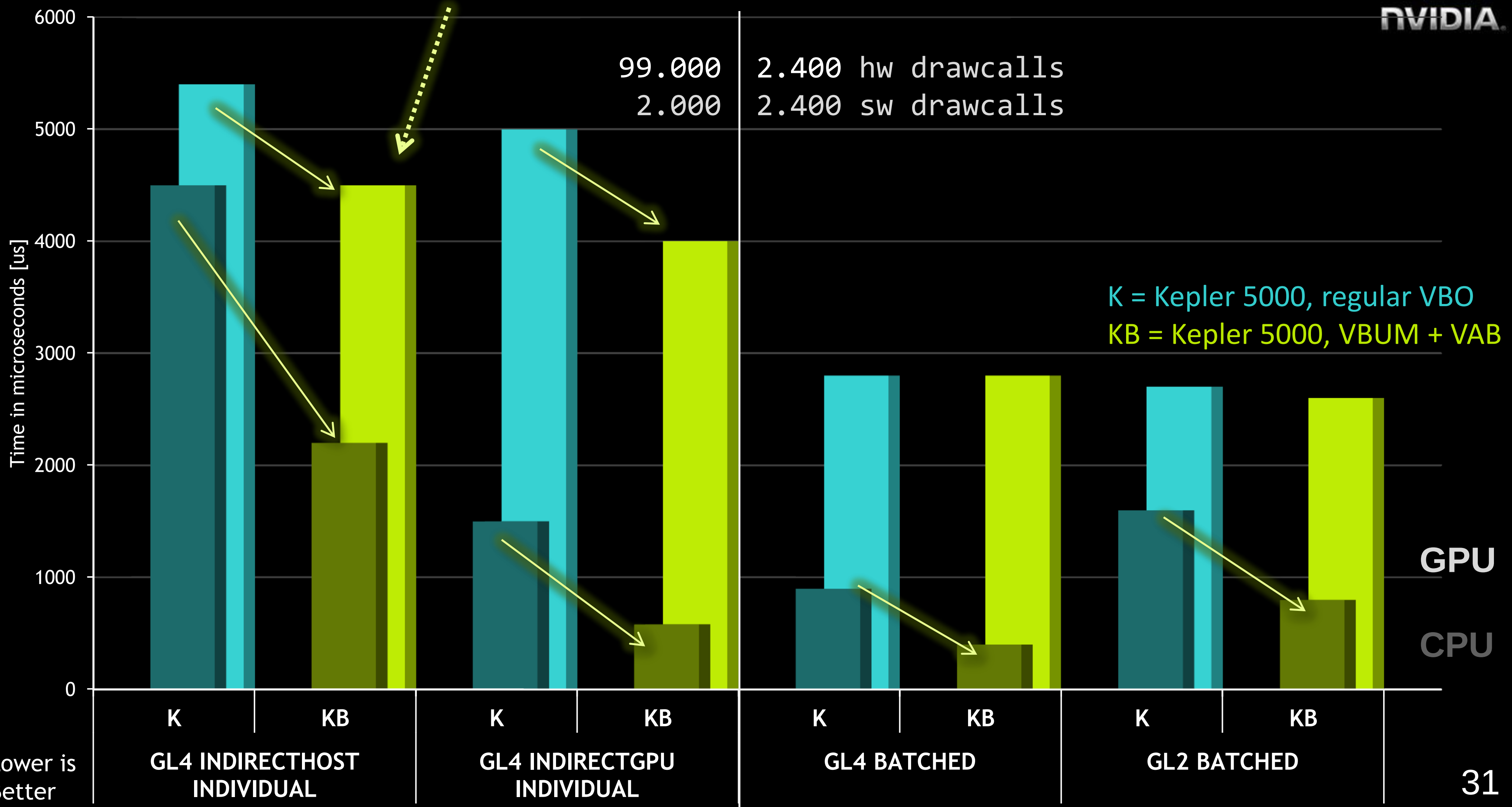
– Vertex/Index setup inside MultiDrawIndirect command



Lower is Better

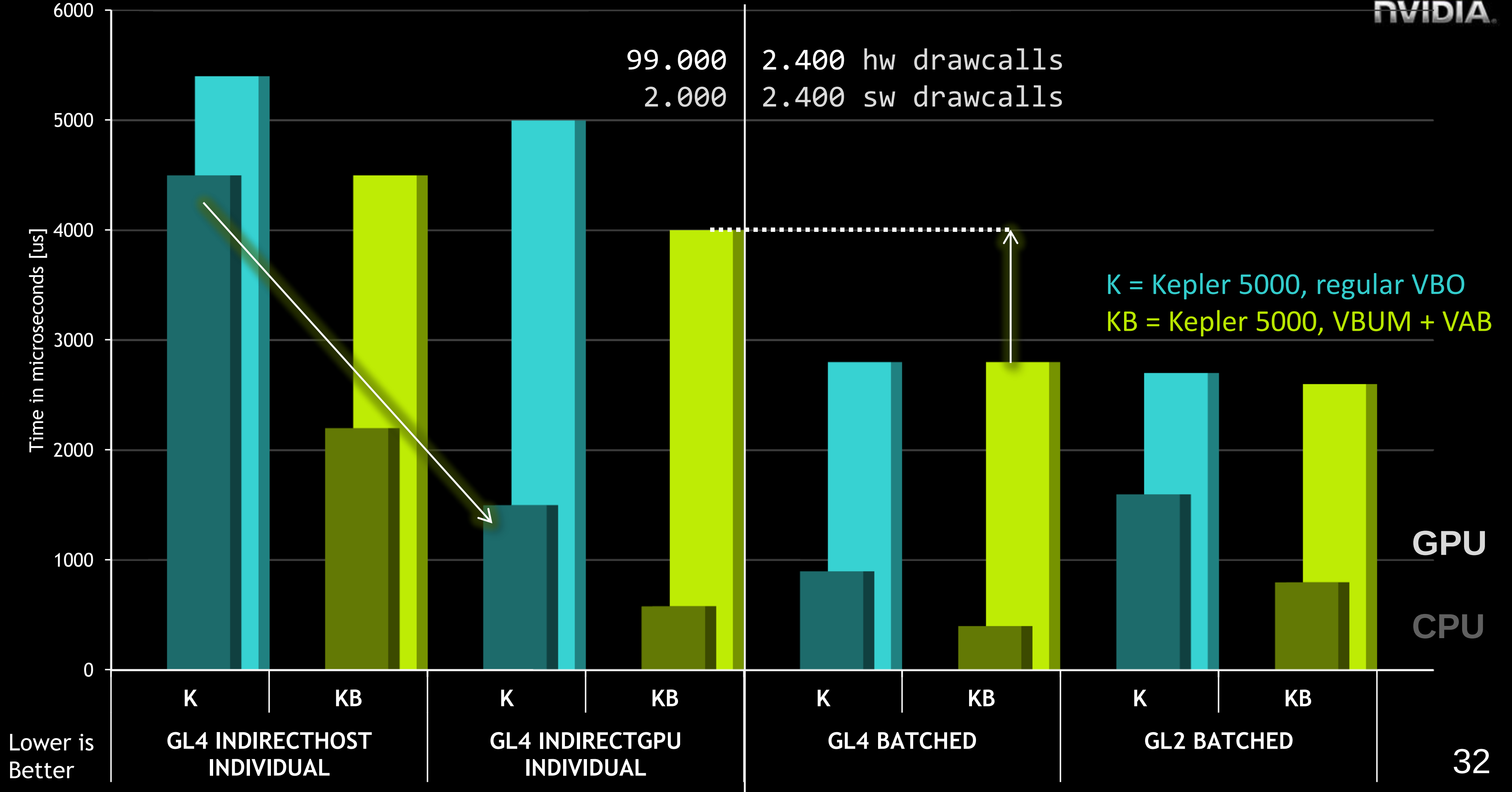
Effect on CPU time

Bindless (green) always reduces CPU, and may help framerate/GPU a bit



MultiDrawIndirect achieves almost 20 Mio drawcalls per second (2000 VBO changes, „only“ 1/3 perf lost). GPU-buffered commands save lots of CPU time

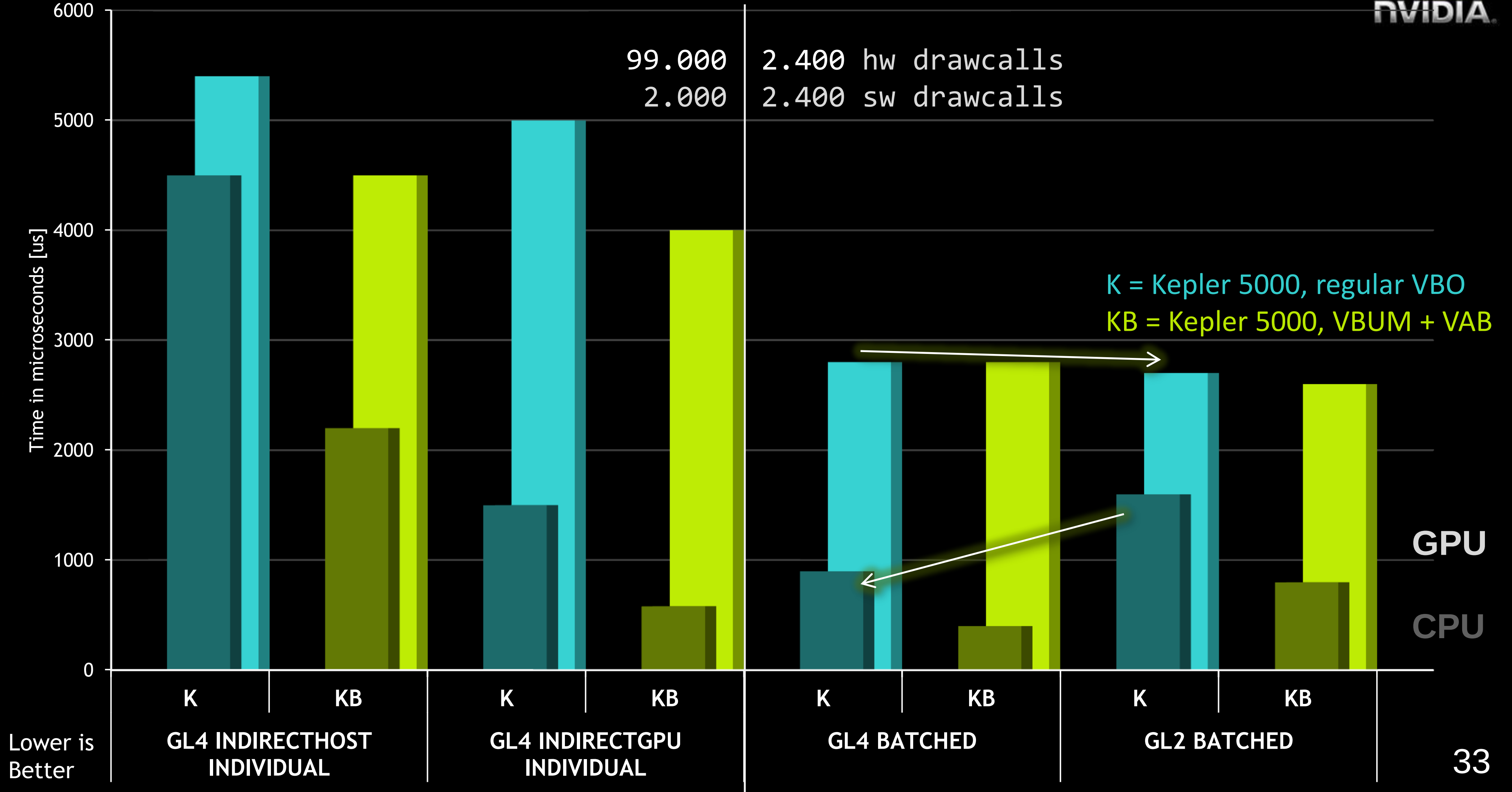
Scene-dependent! INDIVIDUAL could be as fast if enough work per drawcall



GPU TECHNOLOGY CONFERENCE

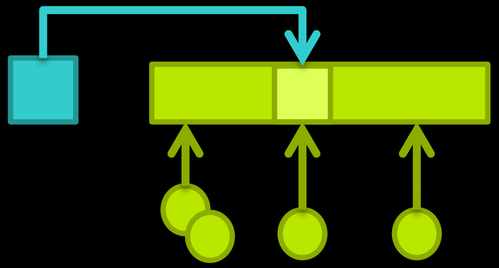
GL2 uniforms beat paletted UBO a bit in GPU, but are slower on CPU side. (1 glUniform call with 8x vec4, vs indexed UBO)

Scene-dependent!
GL4 better when more materials changed per object

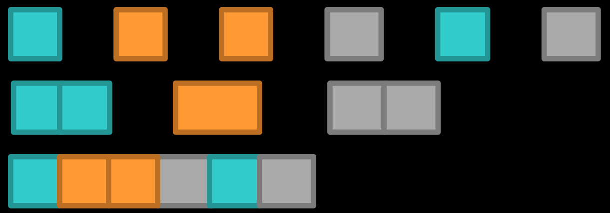


Recap

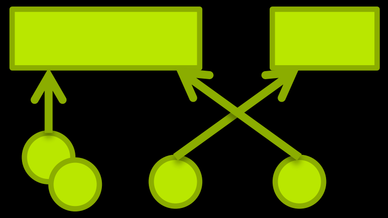
- Share geometry buffers for batching



- Group parameters for fast updating
- MultiDraw/Indirect for keeping objects independent or remove additional loops
  - baseInstance to provide unique index/assignments for drawcall

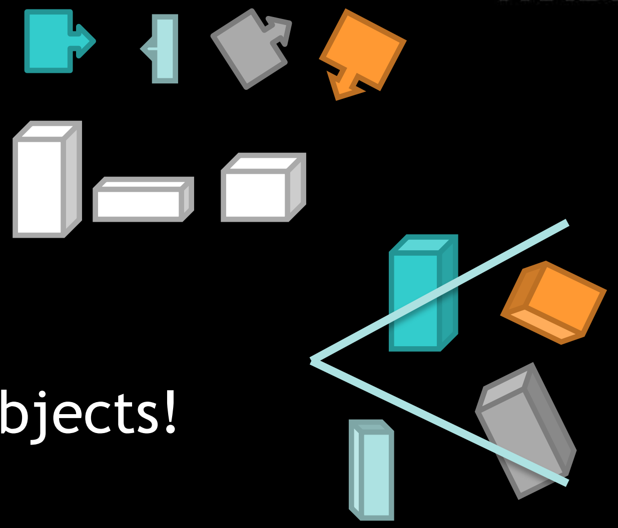


- Bindless to reduce validation overhead/add flexibility



GPU Culling Basics

- GPU friendly processing
  - Matrix and bbox buffer, object buffer
  - XFB/Compute or „invisible“ rendering
  - Vs. old techniques: Single GPU job for ALL objects!

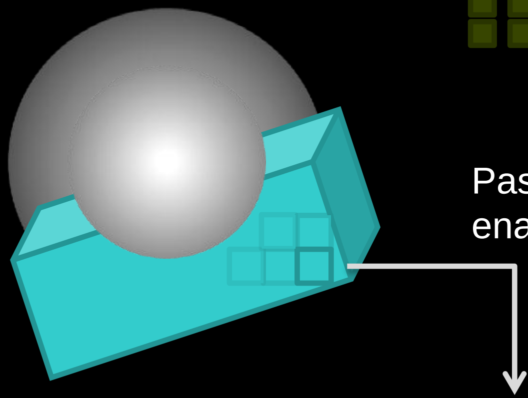


- Results
  - „Readback“ GPU to Host
    - Can use GPU to pack into bit stream
  - „Indirect“ GPU to GPU
    - Set DrawIndirect‘s instanceCount to 0 or 1

```
0,1,0,1,1,1,0,0,0

buffer cmdBuffer{
  Command cmds[];
};
...
cmds[obj].instanceCount = visible;
```

Occlusion Culling

Algorithm by
Evgeny Makarov, NVIDIA

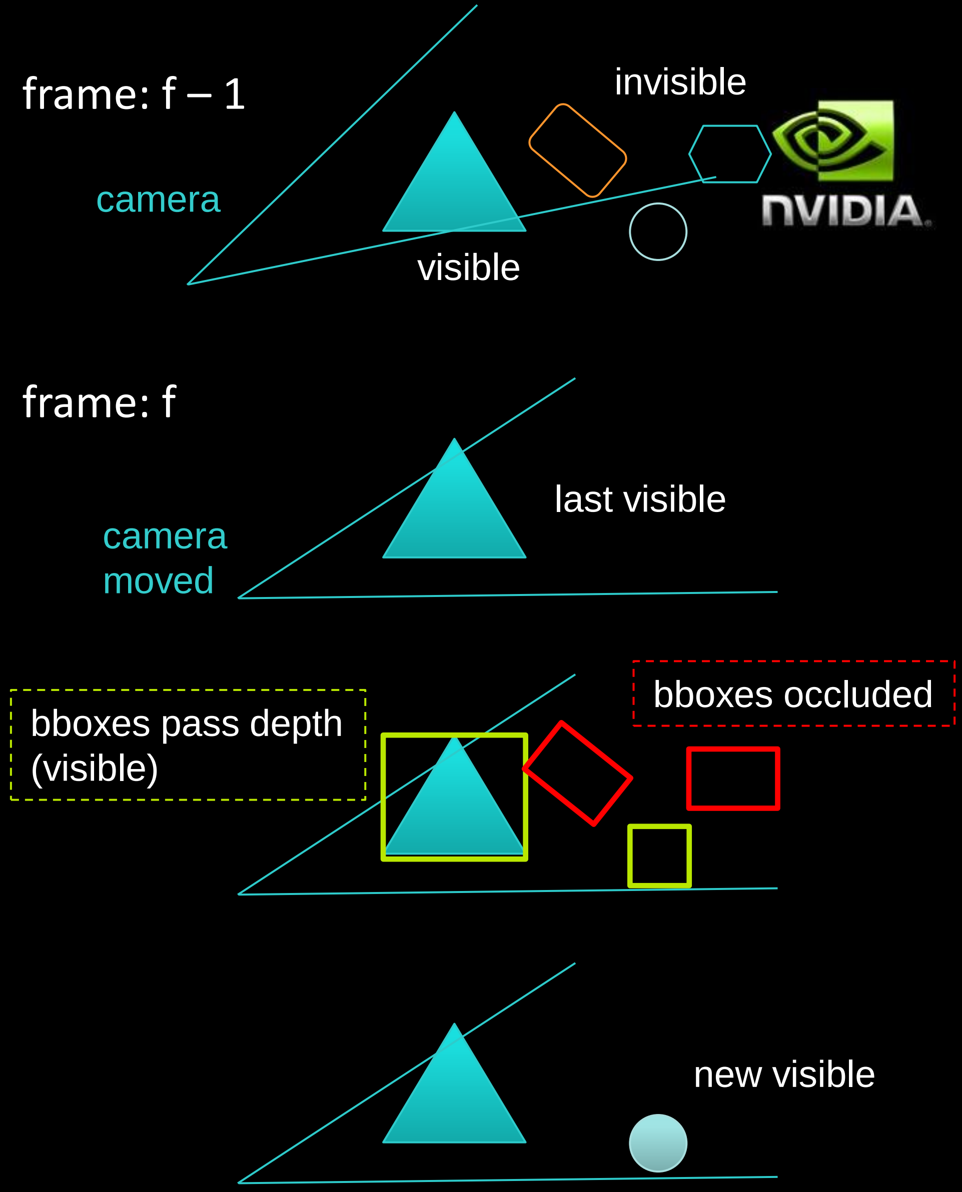


- OpenGL 4.2+
  - Depth-Pass
  - Raster „invisible“ bounding boxes
    - Disable Color/Depth writes
    - Geometry Shader to create the three visible box sides
    - Depth buffer discards occluded fragments (earlyZ...)
    - Fragment Shader writes output: visible[objindex] = 1

```
// GLSL fragment shader
// from ARB_shader_image_load_store
layout(early_fragment_tests) in;

buffer visibilityBuffer{
  int visibility[];
};

flat in int objID;
void main(){
 visibility[objID] = 1;
}
// buffer would have been cleared
// to 0 before
```

Temporal Coherence

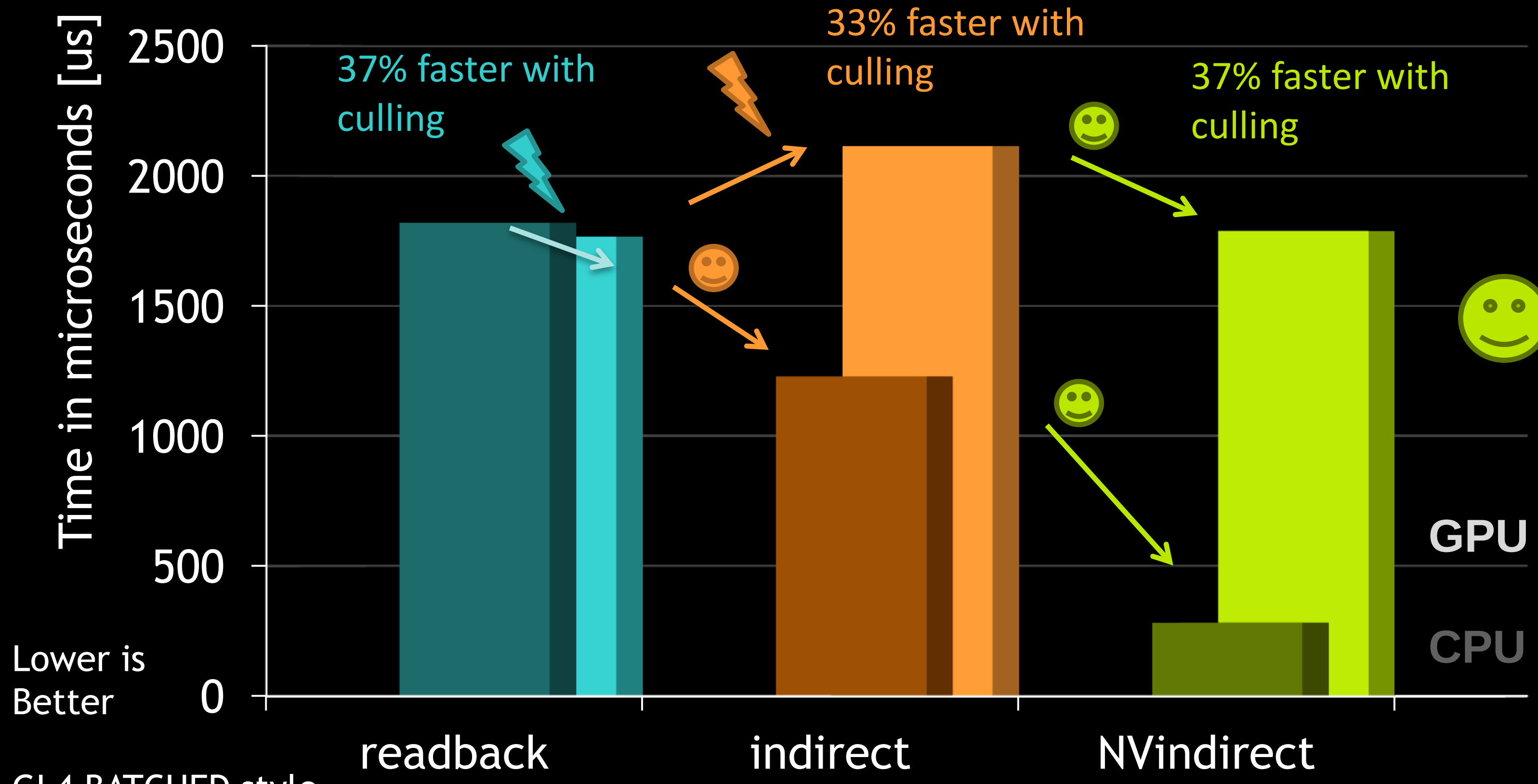
- Exploit that majority of objects don‘t change much relative to camera
- Draw each object only once (vertex/drawcall-bound)
  - Render last visible, fully shaded
    `(last)`
  - Test all against current depth:
    `(visible)`
  - Render newly added visible:
    none, if no spatial changes made
    `(~last) & (visible)`
  - `(last) = (visible)`



Culling Readback vs Indirect

For readback results, CPU has to wait for GPU idle

In the „draw new visible“ phase indirect cannot benefit of „nothing to setup/draw“ in advance, still processes „empty“ lists



GL4 BATCHED style

NV_bindless_multidraw_indirect saves CPU and bit of GPU time

Scene-dependent, i.e. triangles per drawcall and # of „invisible“

Culling Results

- Temporal culling very useful for object/vertex-boundedness
  - Can also apply for Z-pass...
- Readback vs Indirect
  - Readback variant „easier“ to be faster (no setups...), but syncs!
  - NV_bindless_multidraw benefit depends on scene (VBO changes and primitives per drawcall)
- Working towards GPU autonomous system
  - (NV_bindless)/ARB_multidraw_indirect as mechanism for GPU creating its own work, research and feature work in progresss

glFinish();

- Thank you!

  – Contact

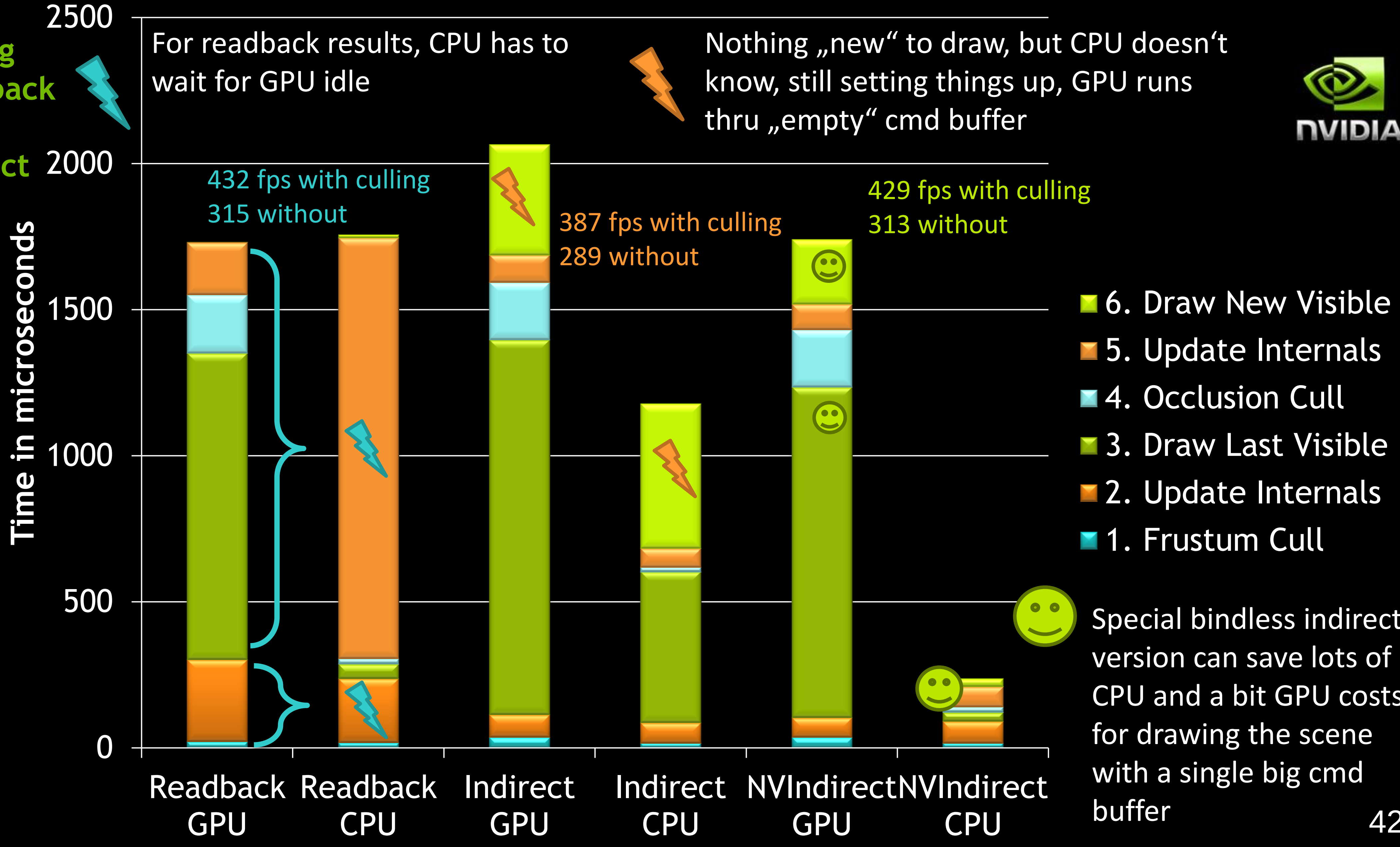
    - ckubisch@nvidia.com
    - matavenrath@nvidia.com

NVIDIA Bindless Technology

- Family of extensions to use native handles/addresses
  - NV_vertex_buffer_unified_memory
  - NV_bindless_multidraw_indirect
- NV_shader_buffer_load/store
  - – Pointers in GLSL
- NV_bindless_texture
  - – No more unit restrictions
  - – References inside buffers

```
// GLSL with true pointers
uniform MyStruct* mystructs;

// API
glUniformui64NV (bufferLocation,
                 bufferADDR);

texHDL = glGetTextureHandleNV (tex);
// later instead of glBindTexture
glUniformHandleui64NV (texLocation,
                       texHDL)

// GLSL
// can also store textures in resources
uniform materialBuffer {
  sampler2D manyTextures [LARGE];
}
```

Culling Readback vs Indirect



Special bindless indirect version can save lots of CPU and a bit GPU costs for drawing the scene with a single big cmd buffer