AGENDA

Turing Mesh Shaders
Turing Barycentrics
Buffer Reference
Turing Cooperative Matrix
Partitioned Subgroup
Turing Texture Access Footprint
Turing Derivatives in Compute Shader
Turing Corner Sampled Image
Turing Representative Fragment Test
Turing Exclusive Scissor Test
Cross API Interop
S9833 - NVIDIA VKRay - Ray Tracing in Vulkan
Hardware-Accelerated Real-time Raytracing
VK_NV_ray_tracing

S9891 - Updates on Professional VR and Turing VRWorks
Variable rate shading, multi-view, multi-GPU
VK_NV_shading_rate_image,
KHR_multiview and KHR_device_group
(promoted in VK 1.1)

S9661 - NVIDIA Nsight Graphics: Getting The Most From Your Vulkan Applications
Profiling and Debugging
MESH SHADERS
MOTIVATION

Vegetation, undergrowth, greebles

Fine geometric detail at massive scale

Pre-computed topologies for LODs

Efficient submission of small objects

Flexible instancing

Custom precision for vertices
MOTIVATION

Auxiliary Meshes

- Proxy hull objects
- Iso-surface extraction
- Particles
- Text glyphs
- Lines/Stippling etc.
- Instancing of procedural shapes

Bartz et al. [4]
MOTIVATION

CAD Models

➢ High geometric complexity (treat as many simple triangle clusters)

➢ Large assemblies can easily reach multiple 100 million triangles

➢ VR demands high framerates and detail

➢ Cannot always rely on static solutions (animations, clipping etc.)

➢ Allow compressed representations
MESH SHADING
New programming model for geometry processing

Evolution from singleton shaders to cooperative groups

- Pixel lighting ➔ Tile-based lighting via compute
- Vertex processing ➔ Meshlet processing

Essential components

- Compute-like execution model - data sharing and sync
- No fixed-function fetch for index processing or vertices
- One level of expansion, flexible work creation/tessellation

Cooperative thread groups operate on meshlets
EXECUTION
Compute Shader Model

- **SHADER INVOCATIONS**
- **Dispatched as 1D grid**
- **Input**
  - `uint WorkGroupID`
- **Thread group**
- **Output memory**
  - `<=16 KB`
- **Compile-time allocation size**
  - Generic Output or Vertices/Indices
  - Shared Memory

- Cooperative access to per-workgroup memory
- Manual synchronization required (barrier(...))
<table>
<thead>
<tr>
<th>Shader</th>
<th>Thread Mapping</th>
<th>Topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Shader</td>
<td>No access to connectivity</td>
<td>1 Vertex</td>
</tr>
<tr>
<td>Geometry Shader</td>
<td>Variable output doesn’t fit HW well</td>
<td>1 Primitive / 1 Output Strip</td>
</tr>
<tr>
<td>Tessellation Shader</td>
<td>Fixed-function topology</td>
<td>1 Patch / 1 Evaluated Vertex</td>
</tr>
<tr>
<td>Mesh Shader</td>
<td>Compute shader features</td>
<td>Flexible</td>
</tr>
</tbody>
</table>
MESH SHADING
New Geometric Pipeline

TRADITIONAL Vertex/Tessellation/Geometry (VTG) PIPELINE

_TASK SHADER_  _WORK GENERATION_  _MESH SHADER_  _RASTER_  _PIXEL SHADER_

Pipelined memory, keeping interstage data on chip

TASK/MESH PIPELINE

_VERTEX ATTRIBUTE FETCH_  _VERTEX SHADE_  _TESS. CONTROL SHADEL_  _TESS. TOPOLOGY GENERATION_  _TESS. EVALUATION SHADEL_  _GEOMETRY SHADEL_  _RASTER_  _PIXEL SHADEL_

Optional Expansion

Pipelined memory...
Task shader allows culling (subpixel, frustum, back-face, occlusion...) or lod picking to minimize mesh workgroups

For generic use we recommend meshlets with 64 vertices, 84 or 124 triangles

Use your own encodings for geometry, all data fetched by shader (compression etc.)

Provides more efficient procedural geometry creation (points, lines, triangles)

With disabled rasterizer implement basic compute trees
TREE EXPANSION

- **LAUNCH**
- **TASK SHADER**
- **MESH SHADER**

**WorkGroupID 0**

**e.g. 3 Tasks**

- **WorkGroupID 0**
  - Primitive 0,1,2,...

- **1**
- **2**

- **Spawn up to 8M-1 children per workgroup**

- **ID is relative to parent**

- **Workgroups are launched ordered within level (execution can be out of order)**

- **Primitive ordering is depth-first**
API

GL & VK & SPIR-V EXTENSIONS

Introduces new graphics stages (TASK, MESH) that cannot be combined with VTG stages

New drawcalls operate only with appropriate pipeline (similar calls in GL)

```c
void vkCmdDrawMeshTasksNV(VkCommandBuffer buffer, uint32_t taskCount, uint32_t taskFirst);

vkCmdDrawMeshTasksIndirectNV

vkCmdDrawMeshTasksIndirectCountNV
```
// same as compute
layout(local_size_x=32) in;
in ivec3  g1_WorkGroupID;
in ivec3  g1_LocalInvocationID;
...
shared MyStruct  s_shared;

// new for task shader
out uint  g1_TaskCountNV;

// new for mesh shader
layout(max_vertices=64) out;
layout(max_primitives=84) out;
layout(triangles/lines/points) out;
out uint  g1_PrimitivesCountNV;
out uint  g1_PrimitiveIndicesNV[];
out gl_MeshPerVertex {  
vec4  gl_Position;
float  gl_PointSize;
float  gl_ClipDistance[];
float  gl_CullDistance[];
}  g1_MeshVerticesNV[];  // [max_vertices]

perprimitiveNV out  g1_MeshPerPrimitive {  
int  gl_PrimitiveID;
int  gl_Layer;
int  gl_ViewportIndex
int  gl_ViewportMask;
}  g1_MeshPrimitivesNV[];  // [max_primitives]

taskNV in/out  MyCustomTaskData {  
...
}  blah;
layout(local_size_x=32) in;
layout(max_vertices=32, max_primitives=32, triangles) out;
out MyVertex { // define custom per-vertex as usual
    vec3 normal; // interfaces with fragment shader
} myout[];

void main() {
    uint invocation = gl_LocalInvocationID.x;
    uvec4 info = meshinfos[gl_WorkGroupID.x]; // #verts, vertoffset, #prims, primoffset

    uint vertex = min(invocation, info.x - 1);
    gl_MeshVerticesNV[invocation].gl_Position = texelFetch(texVbo, info.y + vertex);
    myout[invocation].normal = texelFetch(texNormal, info.y + vertex).xyz;

    uint prim = min(invocation, info.z - 1);
    uint topology = texelFetch(texTopology, info.w + prim);
    // alternative utility function exists to write packed 4x8
    gl_PrimitiveIndicesNV[invocation * 3 + 0] = (topology<<0) & 0xFF;
    gl_PrimitiveIndicesNV[invocation * 3 + 1] = (topology<<8) & 0xFF;
    gl_PrimitiveIndicesNV[invocation * 3 + 2] = (topology<<16) & 0xFF;
    gl_PrimitiveCountNV = info.z; // (actually one thread enough)
}
Replace traditional indexbuffer with pre-computed custom packing

Pack meshlets against a fixed vertex/primitive limit

**Meshlet Example**

**Data Structure**

- Meshlet Desc Buffer
  - Begin & count of unique vertices
  - Begin & count of primitive indices
  - Cluster culling information

- Vertex Index Buffer
  - Primitive indices are relative to vertex begin

- Primitive Index Buffer
  - Indices only need a few bits

Indices: 0, 1, 2, 3, ...
MESHLET EXAMPLE
Cluster Culling

Task shader handles cluster culling:

• Outside frustum
• User clipping plane
• Back-face cluster
• Below custom pixel size
Sample that replaces indexbuffer with meshlet data structure and uses task shader to perform cluster culling. It also saves 25-50% of memory compared to indexbuffer.

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

model courtesy of PTC
Sample that replaces indexbuffer with meshlet data structure and uses task shader to perform cluster culling. It also saves 25-50% of memory compared to indexbuffer.

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

model courtesy of Georgia Institute of Technology
TINY DRAW CALLS

Some scenes suffer from low-complexity drawcalls (< 512 triangles)

Task shaders can serve as faster alternative to Multi Draw Indirect (MDI)

• MDI or instanced drawing can still be bottlenecked by GPU
• Task shaders provide distributed draw call generation across chip
• Also more flexible than classic instancing (change LOD etc.)
Task shader can compute how much work needs to be generated per input primitive (line strips [4], grids, shapes etc.).

Can also skip invisible portions entirely.
BARYCENTRIC COORDINATES
VK/SPV_NV_fragment_shader_barycentric

Custom interpolation of fragment shader inputs

\[ P = A \cdot bx + B \cdot by + C \cdot bz \]

// new built-ins
in vec3 gl_BaryCoordNV;
in vec3 gl_BaryCoordNoPerspNV;

// new keyword to get un-interpolated inputs
pervertexNV in Inputs {
  uint packed;
} inputs[];

// manual interpolation, also allows using smaller datatypes
vec2 tc = unpackHalf2x16(inputs[0].packed) * gl_BaryCoordNV.x +
          unpackHalf2x16(inputs[1].packed) * gl_BaryCoordNV.y +
          unpackHalf2x16(inputs[2].packed) * gl_BaryCoordNV.z;
BUFFER REFERENCE
Greater flexibility in custom data structures stored within SSBOs

„pointer“-like workflow

Developer responsible to manage alignment

// declare a reference data type
layout(buffer_reference, buffer_reference_align=16) buffer MyType {
  uvec2 blah;
  vec2 blubb;
};
uniform Ubo {
  MyType ref; // buffer references are 64-bit sized, address via API
};

// behaves similar to struct, can also be passed to functions
... ref.blah ... or ... doSomething(ref);

// flexible casting, and constructing from other references/uint64
... MyType(uint64_t(ref) + 128).foo ... MyOtherType(ref).foo

// UPCOMING EXTENSION: array/arithmetic usage
... (ref+1).blah ... or ... doSomething(ref + idx);
... ref[1].blah ... or ... doSomething(ref[idx]);
Ability to get the physical address of buffers

The extension was also designed to be debug tool friendly (nsight, renderdoc etc.) to allow trace replay with old address values

// supported on all NVIDIA Vulkan devices

// at creation time enable the new usage
VkBufferCreateInfo info = {...};
info.usage |= VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_EXT;

// later query the address and use it as value
// within buffers or pushconstants

VkDeviceAddress addr = vkGetBufferDeviceAddressEXT(device, {... buffer ...});

// put addr into buffer/image etc. as seen in UBO variable before
SUBGROUP REFRESHER
Invocations within a subgroup can synchronize and share data with each other efficiently. For NVIDIA 32 invocations form one subgroup ("warp").

```
// Single Invocation : „Shader Thread“
gl_LocalInvocationID (1D) == gl_SubGroupInvocationID + gl_SubgroupID * gl_SubgroupSize;
```
A task shader culls 32 meshlets within a subgroup and outputs surviving meshletIDs

<table>
<thead>
<tr>
<th>Variable</th>
<th>Invoc. 0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>render</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>vote</td>
<td>101 (binary)</td>
<td>101</td>
<td>101</td>
</tr>
</tbody>
</table>

```cpp
// meshletID is different per invocation
bool render = valid && !(earlyCull(meshletID, object));

// The ballot functions can be used to easily count across
// a subgroup and create prefixsums
uvec4 vote = subgroupBallot(render);
uint tasks = subgroupBallotBitCount(vote);
// exclusive means the value of current invocation is excluded
uint outIndex = subgroupBallotExclusiveBitCount(vote);

if (render) {
    OUT.meshletIDs[outIndex] = meshletID;
}
if (gl_SubgroupInvocationID == 0) {
    gl_TaskCountNV = tasks;
}
COOPERATIVE MATRIX
VK_NV_cooperative_matrix brings very fast large matrix multiply-add to Vulkan

Supported for Turing RTX (NOT Volta)
// Classic datatype variables exist per invocation (thread) or are in shared memory.
// New datatype introduced that exists within a pre-defined scope.

fcoopmatNV<PRECISION_BITS, gl_ScopeSubgroup, ROWS, COLS> variable;

// new functions handle load/store (one example shown)
void coopMatLoadNV(out fcoopmatNV m,
    volatile coherent float16_t[] buf,       // ssbo or shared memory array variable
    uint element,   // starting index into buf to load from
    uint stride,    // element stride for one column or row
    bool colMajor) // compile-time constant
// if colMajor == true, load COLS many values from buf[element + column_idx * stride];

// perform the actual multiply within the scope (here subgroup)
fcoopmatNV coopMatMulAddNV(fcoopmatNV A, fcoopmatNV B, fcoopmatNV C)
Query support from device

 Optionally use specialization constants to quickly build multiple kernels

 Example here
 https://github.com/jeffbolznv/vk_cooperative_matrix_perf

 71 TFLOPS on Titan RTX

typedef struct VkCooperativeMatrixPropertiesNV {
    VkStructureType sType;
    void* pNext;
    uint32_t MSize;
    uint32_t NSize;
    uint32_t KSize;
    VkComponentTypeNV AType;
    VkComponentTypeNV BType;
    VkComponentTypeNV CType;
    VkComponentTypeNV DType;
    VkScopeNV scope;
} VkCooperativeMatrixPropertiesNV;

// Multiple configurations may be supported

vkGetPhysicalDeviceCooperativeMatrixPropertiesNV (
    VkPhysicalDevice, uint32_t* propCount, ...props)
PARTITIONED SUBGROUP
PARTITIONED SUBGROUP
VK_NV_shader_subgroup_partitioned

Identify invocations with the same variable value

Use bitfield masks to operate across subset of threads
// Find invocations with identical key values within a subgroup
uvec4 identicalBitMask = subgroupPartitionNV(key);

<table>
<thead>
<tr>
<th>Value</th>
<th>Invocation 0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>17</td>
<td>35</td>
<td>17</td>
<td>9</td>
<td>35</td>
</tr>
<tr>
<td>identicalBitMask</td>
<td>00101 (binary)</td>
<td>10010</td>
<td>00101</td>
<td>01000</td>
<td>10010</td>
</tr>
</tbody>
</table>

bool isFirstUnique = gl_SubgroupInvocationID == subgroupBallotFindLSB(identicalBitMask);

isFirstUnique    | true    | true | false | true | false |

// use mask for aggregate operations, for example
uint sum = subgroupPartitionedAddNV(value, identicalBitMask);

<table>
<thead>
<tr>
<th>value</th>
<th>7</th>
<th>3</th>
<th>13</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>20</td>
<td>5</td>
<td>20</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
TEXTURE ACCESS FOOTPRINT
TEXTURE SPACE SHADING
Aka Decoupled Shading

Geometry

Visibility sampling
(find texels that are needed)

Shading
(shade texels in uv space)

Resample to visibility
(regular texture fetch, anti-
aliasing via texture filtering)
TEXTURE SPACE SHADING

https://devblogs.nvidia.com/texture-space-shading/

https://www.youtube.com/watch?v=Rpy0-q0TyB0

Visit these links for more details
MOTIVATION

Find what texels contribute to a pixel
MOTIVATION
Find what texels contribute to a pixel
TEXTURE ACCESS FOOTPRINT

VK_NV_shader_image_footprint / GLSL_NV_shader_texture_footprint

New query functions in GLSL/SPIR-V

Returned footprint helps to identify which mips and which texel tiles within them would be touched

```c
void gl_TextureFootprint2DNV {
  ivec2 anchor;
  ivec2 offset;
  ivec2 mask;
  uint lod;
  uint granularity;
} footprint;

bool singleMipOnly = textureFootprintNV(
  tex, uv,
  granularity,
  bCoarseMipLevel,
  footprint);
```

Each bit in `mask` represents tiles: e.g. 2x2 texels

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

DERIVATIVES IN COMPUTE SHADERS
DERIVATIVES IN COMPUTE

VK_NV_compute_shader_derivatives

Previously only fragment shader texture lookups allowed the use of derivatives in texture lookups (implicit mip-mapping etc.)

Now compute shaders supports:

- All texture functions
- Derivative functions
- subgroup_quads functions

Local invocations as 2x2x1 (quads) as linear threads

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2x+0, 2y+0, z</td>
<td>2x+1, 2y+0, z</td>
<td>4n+0</td>
</tr>
<tr>
<td>2x+0, 2y+1, z</td>
<td>2x+1, 2y+1, z</td>
<td>4n+2</td>
</tr>
</tbody>
</table>

// enable the layout
layout(derivative_group_quadsNV) in;
// or
layout(derivative_group_linearNV) in;

// you can use all texture functions now
... texture(tex, uv);
// or derivatives
... dFdx(variable);
CORNER SAMPLED IMAGES

VK_NV_corner_sampled_image

A new extension that eases hardware-accelerated PTEX

No seams at borders

VkImageCreateInfo info = {...};
info.flags |= VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV;

Visible seams due to interpolation

All samples interpolated equally
REPRESENTATIVE FRAGMENT TEST
FASTER OCCLUSION TESTS
VK_NV_representative_fragment_test

This extension can help shader-based occlusion queries that draw many object proxies at once.

Enabling can reduce fragment-shader invocations when proxy primitives take up larger portions of the screen.

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

```cpp
// depth-test passing
// fragments tag objects as visible
layout(early_fragment_tests) in;
...
visibility[objectID] = 1;
```

Representative test OFF: primitives are rastered completely

Representative test ON: primitives can be rastered partially
FASTER OCCLUSION TESTS

VR-like scenario, occlusion test for ~9k bboxes at 2048 x 2048 x 2x msaa

Representative test OFF: 0.5 ms

Representative test ON: 0.15 ms
EXCLUSIVE SCISSOR TEST
EXCLUSIVE SCISSOR

VK_NV_scissor_exclusive

Can reverse the scissor-test to „stamp out“ areas

Traditional Inclusive  

New Exclusive

// specify at PSO create-time
VkPipelineViewportExclusiveScissorStateCreateInfoNV info;
info.pExclusiveScissors = {{offset,extent},..};
..
// or use dynamic state
vkCmdSetExclusiveScissorNV(cmd, first, count, rectangles);
CROSS API INTEROP

Vulkan or DX12 as exporters

Allocated Memory
- Buffer
- Image

Export Allocation
- VK 1.1 or VK_KHR_external_memory
- ID3D12Heap, ID3D12Resource

Imported Memory
- device memory
- cuArray

Directly Reference Allocation
- GL_EXT_memory_object
- GL_EXT_memory_attachment

Synchronization Object
- Import Semaphore
  - GL_EXT_semaphore
- Export Semaphore
  - VK_KHR_external_semaphore
  - ID3D12Fence
THANK YOU

nv_spec_contributors
{
  Jeff Bolz
  Pierre Boudier
  Pat Brown
  Chao Chen
  Piers Daniell
  Mark Kilgowski
  Pyarelal Knowles
  Daniel Koch
  Christoph Kubisch
  Chris Lentini
  Sahil Parmar
  Tyson Smith
  Markus Tavenrath
  Kedarnath Thangudu
  Yury Uralsky
  Eric Werness
}

cubisch@nvidia.com (professional vis, GL/VK) @pixeljetstream

[2] https://www.flickr.com/photos/14136614@N03/6209344182
[3] k-DOPs as Tighter Bounding Volumes for Better Occlusion Performance - Bartz, Klosowski & Staneker
https://pdfs.semanticscholar.org/bf4e/7c405d0f2a259f78e91ce1eb68a5d794c99b.pdf
[4] GTC 2016 - OpenGL Blueprint Rendering - Christoph Kubisch