OVERVIEW

Ray tracing vs. rasterization
How did we get here?
Real-time ray tracing applications
Exposing RTX through Vulkan
Ray Tracing vs. Rasterization

- Rasterization: evaluate one triangle at a time
  - Store and accumulate some of the output data, but discard most of it

- Computationally cheap, but “local”
- Amenable to hardware implementation
Ray Tracing vs. Rasterization

- Ray tracing: sampling the entire scene one ray at a time
  - Entire scene is evaluated
  - Can visit same primitive many times

- Extremely flexible
- Can be computationally intensive
RAY TRACING AT NVIDIA
Decade+ of R&D

Real-time graphics: rasterization is king
   GPUs evolve to enable powerful rasterization-based algorithms

2007: CUDA brings general purpose computing to GPUs
   Ray tracing research focused on acceleration via GPGPU
OPTIX: A GENERAL PURPOSE RAY TRACING ENGINE

Parker et al., OptiX: A General Purpose Ray Tracing Engine, SIGGRAPH 2010

RT_PROGRAM void pinhole_camera() {
    Ray ray = PinholeCamera::makeRay( launchIndex );
    UserPayload payload;
    rtTrace( topObject, ray, payload );
    outputBuffer[launchIndex] = payload.result;
}

Figure 3: Example ray generation program (in CUDA C) for a single sample per pixel. The 2-dimensional grid location of the program invocation is given by the semantic variable launchIndex, which is used to create a primary ray using a pinhole camera model. Upon tracing a ray, the invoked material hit programs fill the result field of the user-defined payload structure. The variable topObject refers to the location in the scene hierarchy where ray traversal should start, typically the root of the node graph. At the location specified by launchIndex, the result is written to the output buffer to be displayed by the application.
TURING: HARDWARE ACCELERATED RAY TRACING
Physically correct shadows
Sharp and Contact Hardening

Ray Traced Shadows
Ray Traced Ambient Occlusion

Improved ambient occlusion detail with fewer artifacts
Physically-correct reflections on arbitrary surfaces
Physically-correct light transport in a complex game
Ray Tracing: Not Just Graphics

- Physics, particle simulations
- Audio path tracing / simulation
- AI visibility queries
EXPOSING RTX THROUGH VULKAN

VK_NV_raytracing
API design principles

- Proven ray tracing API concepts
- Hardware-agnostic
- Good fit with Vulkan API
- Implementable on compute functionality
GRAPHICS PIPELINES

Rasterization

Draw Call → Scheduling → Vertex Shading → Rasterization → Fragment Shading → ROP

Ray Tracing

Ray Generation → Scheduling → Traversal & Intersection → Scheduling → Shading
BUILDING THE PIPELINE
Ray tracing building blocks

Acceleration Structure
Abstraction for scene geometry, enables efficient scene traversal

Ray tracing shader domains
Handle ray generation and termination, intersections, material shading

Shader binding table
Links shaders and resources to be used during traversal

Ray tracing Pipeline object
Acceleration Structure Concepts

- Ray tracing: testing one ray against all scene primitives
- Acceleration structures required for efficient operation
Acceleration Structure Concepts

- Dual-level acceleration structure
- Opaque (implementation-defined) data structures
- Efficient build and update
Building Acceleration Structures

Vertex & Index Buffers
Build/Update
Bottom Level AS

Vertex & Index Buffers
Build/Update
Bottom Level AS

Vertex & Index Buffers
Build/Update
Bottom Level AS
Building Acceleration Structures

Bottom Level AS

Xform / Shading

Bottom Level AS

Xform / Shading

Bottom Level AS

Xform / Shading

Build/Update

Top Level AS
Creating Acceleration Structures

- New object type VkAccelerationStructureNV
- New descriptor type for binding AS to shaders

```
struct VkAccelerationStructureInfoNV {
    VkStructureType sType;
    void *pNext;
    // top-level or bottom-level
    VkAccelerationStructureTypeNV type;
    VkAccelerationStructureCreateFlagsNV flags;
    uint32_t instanceCount;
    uint32_t geometryCount;
    const VkGeometryNV* pGeometries;
};

vkCreateAccelerationStructureNV(
   VkDevice device,
    const VkAccelerationStructureCreateInfoNV *info,
    const VkAllocationCallbacks *pAllocator,
    VkAccelerationStructureNV *pAccelerationStruct );
```
Acceleration Structure Memory Management

- Backing store for AS is managed by application
- Memory requirements will vary

```c
vkGetAccelerationStructureMemoryRequirementsNV(
    VkDevice device,
    const
    VkAccelerationStructureMemoryRequirementsInfoNV
    *pInfo,
    VkMemoryRequirements2 *pMemoryRequirements
);

vkBindAccelerationStructureMemoryNV(...);
```
Acceleration Structure Build/Update

- Single command to build or update an AS
- AS updates have restrictions for efficiency

```c
vkCmdBuildAccelerationStructureNV(
    VkCommandBuffer cmdBuf,
    const VKAccelerationStructutreInfoNV*,
    VkBuffer instanceData,
    VkDeviceSize instanceOffset,
    VkBool32 update,
    VkAccelerationStructureNV dst,
    VkAccelerationStructureNV src,
    VkBuffer scratch,
    VkDeviceSize scratchOffset
);
```
Acceleration Structure Copy

- Acceleration structures can be copied, optionally compacting during the copy

```c
vkCmdCopyAccelerationStructure(
    VkCommandBuffer cmdBuf,
    VkAccelerationStructure dst,
    VkAccelerationStructure src,
    VkCopyAccelerationStructureModeNV mode
);```
Ray tracing shader domains

- Ray Generation
- Intersection
  - Any-Hit
  - Closest-Hit

- traceNV()
- Ray Generation
- Acceleration Structure Traversal
- Hit?
  - no: Miss
  - yes: Closest Hit
- Any Hit
- Intersection
Inter-shader communication

- **Ray payload**
  - Application-defined structure to pass data between hit stages and shader stage that spawned a ray
  - Used to return final intersection information to ray generation shader
Inter-shader communication

▪ Ray Attributes
  ▪ Application-defined structure to pass intersection information from intersection shader to hit shaders
Ray Generation Shaders

- Starting point for all ray tracing work
- Simple grid of threads launched from host
- Traces rays, writes final output
Intersection Shaders

- Compute ray intersections with app-defined primitives
- Ray-triangle intersections built-in
Any-hit shader

- Invoked after an intersection is found
- Multiple intersections invoked in arbitrary order
Closest-hit shader

- Invoked on the closest intersection of the ray
- Can read attributes and trace rays to modify the payload
Miss shader

- Invoked if no hit is found and accepted
- Can trace rays and modify ray payload
Inter-shader Interface Mapping to GLSL

- Ray payload and intersection attributes: new block decorations

- `traceNV()` builtin may be called with different payload structures in a single shader

```glsl
rayPayloadNV { ... }  
intersectionAttributesNV { ... }

layout (location= 1)  
rayPayloadNV firstInterface {...}  

layout (location= 2)  
rayPayloadNV secondInterface {...}  

traceNV(..., 1) // firstInterface  
traceNV(..., 2) // secondInterface
```
GLSL Extensions

- **Built-in variable decorations:**
  - ID/index information
  - ray parameters, hit parameters
  - instance transforms

- **New functions:**
  - `traceNV()` traces a ray into the scene
  - `reportIntersectionNV()` outputs intersection information
  - `ignoreIntersectionNV()` rejects an intersection
  - `terminateRayNV()` terminates the current ray

```c
traceNV(
    accelerationStructure topLevel,
    uint rayFlags,
    uint cullMask,
    uint sbtRecordOffset,
    uint sbtRecordStride,
    uint missIndex,
    vec3 origin,
    float tmin,
    vec3 direction,
    float tmax,
    int payload
);

reportIntersectionNV(
    float hit,
    uint hitKind
);

ignoreIntersectionNV();

terminateRayNV();
```
A given ray query into the scene can hit any object
- ... with different kinds of material shaders
- ... requiring different kinds of resources (e.g., textures)
- ... and different parameters (e.g., transparency value)
- ... per object instance!
Shader Binding Table

- Array of opaque shader handles + application storage inside VkBuffer
- Indexing used to determine shaders / data used
- App-defined stride for SBT records to accommodate different resource sets
Shader Binding Table

ShaderBindingTable Buffer

<table>
<thead>
<tr>
<th>Shader Reference</th>
<th>Custom Data &amp; Indices</th>
<th>Shader Reference</th>
<th>Custom Data &amp; Indices</th>
</tr>
</thead>
</table>

Traversal yields instanceOffset

RT Pipeline with RayGeneration

Descriptor Sets

Acceleration Structures

RT Pipeline with ClosestHit

Descriptor Sets

Textures etc.

Shader traces rays

Shader reacts on hit (can trace)
Shader Binding Table and vkCmdTraceRays

vkCmdTraceRaysNV(VkCommandBuffer, VkBuffer shaderBindingTable, raygenIndex, …)

1: API invokes RayGeneration Shader

2: Shader traces rays

3: Traversal yields instanceOffset

4: Shader reacts on hit (can trace)
Ray Tracing Pipeline Creation

- Ray tracing pipeline contains all shaders used together
- Can have multiple shaders of the same stage

```c
struct VkRaytracingPipelineCreateInfoNV {
    VkStructureType sType;
    void *pNext;
    VkPipelineCreateFlags flags;
    uint32_t stageCount;
    const VkPipelineShaderStageCreateInfo *stages;
    uint32_t groupCount;
    const VkRayTracingShaderGroupCreateInfoNV *pGroups;
    uint32_t maxRecursionDepth;
    VkPipelineLayout layout;
    VkPipeline basePipelineHandle;
    int32_t basePipelineIndex;
};

vkCreateRaytracingPipelinesNV(...);
```
Ray Tracing Shader Handles

- Exposes opaque shader handles to be used inside Shader Binding Table

vkGetRaytracingShaderGroupHandlesNV(
    VkDevice device,
    VkPipeline pipeline,
    uint32_t firstGroup,
    uint32_t groupCount,
    size_t dataSize,
    void *pData
);
vkCmdTraceRaysNV

- Entry point to invoke ray tracing work

- Uses existing vkCmdBindPipeline and vkCmdBindDescriptorSets with new bind point
// Binding for the acceleration structure
layout(set = 0, binding = 0) accelerationStructureNV scene;
layout(set = 1, binding = 0) image2D framebufferImage;

layout(set = 2, binding = 0, std140) uniform traceParameters {
    uint sbtMiss;
    uint sbtOffset;
    uint sbtStride;
    vec3 origin;
    vec3 dir;
}

layout(location = 1) rayPayloadNV primaryPayload {
    vec4 color;
}

void main() {
    primaryPayload.color = vec4(0.0, 0.0, 0.0, 0.0);

    traceNV(scene, 0, 0, sbtMiss, sbtOffset, sbtStride, traceParameters.origin,
            0.0, computeDir(gl_GlobalInvocationIDNV.xy, traceParameters.dir),
            1.e9, 1);

    imageStore(framebufferImage, gl_GlobalInvocationIDNV.xy, primaryPayload.color);
}
Example Closest Hit Shader

// Closest hit shader
//
// Closest hit shaders only have one payload -
// for the incoming ray
rayPayloadInNV primaryPayload {
    vec4 color;
}

layout(set = 3, binding = 0) sampler3D solidMaterialSampler;

void main() {
    vec3 pos = gl_WorldRayOriginNV + gl_hitTNV * gl_WorldRayDirectionNV;
    primaryPayload.color = texture( solidMaterialSampler, pos);
}
Iterative Loop Path Tracing in Ray Generation Shader

```glsl
vec3 color = 0.0f;
Float sample_pdf = 1.0f;
vec3 path_throughput = 1.0f;

// GBufferEntry contains all necessary info to reconstruct the hit and local brdf
GBufferEntry gbuffer_entry = imageLoad(gbuffer, gl_GlobalInvocationID.xy);

Ray ray = reconstruct_ray(gbuffer_entry, pixel); // reconstruct the first ray

for (uint32 bounce = 0; bounce < max_bounces; ++bounce) {
    // unpack the brdf
    Brdf brdf = get_brdf(gbuffer_entry);

    // accumulate emission of the local surface
    color += path_throughput * local_emission(gbuffer_entry, brdf, sample_pdf);

    // perform next-event estimation (casts one or more shadow rays, getting visibility as a return value)
    color += path_throughput * next_event_estimation(gbuffer_entry, brdf, sample_pdf);

    // sample the brdf
    vec3 sample_throughput;
    ray = sample_brdf(gbuffer_entry, brdf, sample_throughput, sample_pdf);

    // perform Russian-Roulette to terminate path
    if (random() >= max(sample_throughput))
        break;

    // accumulate the path throughput
    path_throughput *= sample_throughput / max(sample_throughput);

    // trace a ray and get a brdf as a return value
    gbuffer_entry = trace(…, ray.origin, ray.direction, …);
}
```
HLSSL Support

Mainline DXC now has initial support for compiling DXR shaders into SPIR-V!

- NVIDIA working with Google on enhancing Spiregg to support ray tracing shaders
- Support for RT shader stages and built-ins available in mainline today
- SBT not yet exposed, but **coming soon**
HLSL Support

- Resource binding model is different between DXR and Vulkan
  - DXR: SBT contains *local root signature*, used by the bytecode compiler to map resources
  - Vulkan: SBT is *just memory*, used by the shader directly
    - Intent is for app to use EXT_descriptor_indexing to make resources available, then store indices in the SBT record
    - EXT_device_buffer_address allows buffer pointers in SBT

- Design work underway to expose SBT with source-level compatibility for DXR shaders
Check out Quake 2 RTX at our booth!
Conclusion

- RTX is available in Vulkan!
- NVIDIA fully committed to working with Khronos on multi-vendor standardization

https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/
https://developer.nvidia.com/rtx/raytracing/vkray
https://devblogs.nvidia.com/rtx-best-practices (soon!)
QUESTIONS
Nuno Subtil | nsubtil@nvidia.com