Vulkanised 2025

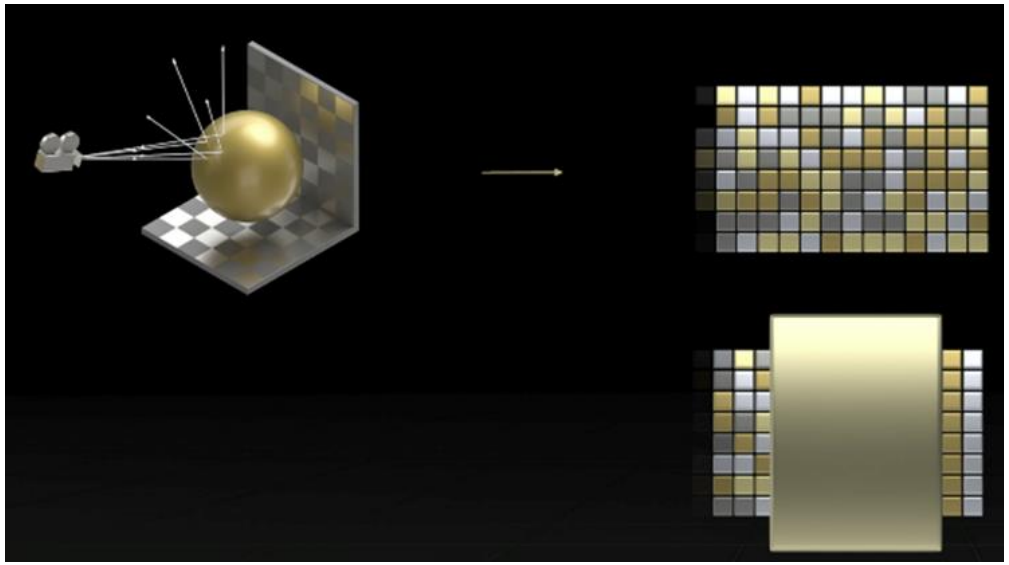
The 7th Vulkan Developer Conference
Cambridge, UK | February 11-13, 2025

Execution Reordering for Ray Tracing Performance.

Eric Werness, NVIDIA

Invocation Reordering for Ray Tracing Performance

- GPUs perform best with coherent work
  - Ray tracing can cause particularly significant divergence with rays starting nearby hitting different objects with different materials
- Recovering coherence is important
  - Execution divergence can reduce instruction throughput by the factor of the divergence
  - Data divergence can reduce the effectiveness of memory fetch paths and caches
- Invocation reordering provides data structures and functions to allow an application to provide more information to an implementation to efficiently regain coherence when tracing rays
- This presentation is about the upcoming VK_EXT_ray_tracing_invocation_reorder extension
- The extension is nearly complete but may change slightly before the final version



Coherent Work, Fast GPU

- Invocation reorder reorders threads so that more similar work happens together
- Convergent and coherent threads are better for performance
  - Convergent shader selection keeps subgroups full with the same instructions
  - Coherent memory accesses hit better in the cache
  - New techniques such as cooperative vectors can benefit even more
- Particularly beneficial for complex material systems on large scenes

New Functionality vs Existing Ray Pipeline Support

- Original ray pipeline API includes support for implicit reordering
  - Limited definition of ray generation launch grouping
  - Separate shader stages implicitly indexed by the implementation
  - Invocation repack instruction
- Why a new API in addition to existing ray pipeline reordering support?
  - Hit objects enable new techniques and more flexibility in shading
  - Explicit reorder points allow applications to be more aware of live state and choose which traces benefit from reordering
  - Ability to pass in application information as a hint for reordering
- Why ray generation and not general compute?
  - General compute has explicit thread grouping and workgroup memory
  - Ray generation already restricted those to enable reordering

API Overview

HitObject

- The HitObjectEXT type encapsulates intersection details or a miss state in an opaque GLSL object
  - Similar in declaration and semantics to a ray query object, but tracking slightly different state
- Declared in ray generation, closest-hit, or miss shaders without extra storage qualifiers.
- Built-ins like hitObjectIsHitEXT, hitObjectIsMissEXT, hitObjectIsEmptyEXT extract the state of the hit object
- Stores transform data, instance ID, geometry index, primitive index, etc.
- Allows reading and writing of shader attributes for intersection points.
- Replaces immediate pipeline calls with a flexible data-carrying handle.

API Overview

Trace and Create

- Hit object can be in one of three high-level states:
  - Empty: Represents not having been traced. Has no shader information.
  - Miss: Traced but didn't hit a primitive. Has shader information but no hit information
  - Hit: Traced and hit a primitive. Has shader information and hit information
- hitObjectRecordEmptyEXT(…) sets a hit object to empty
- hitObjectRecordMissEXT(…) records a miss including ray information
  - `void hitObjectRecordMissEXT(hitObjectEXT hitObject, uint rayFlags, uint sbtRecordIndex, vec3 origin, float Tmin, vec3 direction, float TMax)`
- hitObjectTraceRayEXT(...) issues a ray intersection and populates the hit object based on the results
  - Will execute any hit and intersection shaders as required but not closest hit or miss
  - `void hitObjectTraceRayEXT(hitObjectEXT hitobject,accelerationStructureEXT topLevel, uint rayFlags, uint cullMask, uint sbtRecordOffset, uint sbtRecordStride, uint missIndex, vec3 origin, float Tmin, vec3 direction, float Tmax, int payload);`
- HitObjectRecordFromQueryEXT(…) initializes a hit object from a ray query
  - Takes the ray information from the ray query and attribute information from a parameter
  - `void hitObjectRecordFromQueryEXT(hitObjectEXT hitobject, rayQuery rayQuery, int attributeLocation)`

API Overview

Get Functions

- Each hitObject accessor function returns a specific piece of ray state:
  - hitObjectGetWorldRayOriginEXT()
  - hitObjectGetObjectRayDirectionEXT()
  - hitObjectGetShaderRecordBufferHandleEXT()
- Offers flexible reading of TMin/TMax, instance info, and transformations.
- Lays the groundwork for custom shading logic based on intersection details.

API Overview

Reorder

- reorderThreadEXT(...) re-groups execution invocations based on a hint and/or hitObject state
  - `void reorderThreadEXT(uint hint, uint bits)`
  - `void reorderThreadEXT(hitObjectEXT hitObject)`
  - `void reorderThreadEXT(hitObjectEXT hitObject, uint hint, uint bits)`
- Only valid in ray-generation shaders.
- The spec defines this in decreasing order of priority for information used in the reorder
  - ShaderID in the HitObject
  - Coherence hint, from highest to lowest included
  - Other information in the HitObject
- Allows implementation flexibility on sorting, bucketing, etc

API Overview

Fused Functions

- Fused functions combine multiple of the primitive functions described above into a single function
- Behavior is exactly the same as a sequence of primitive functions but may be more efficient on some implementations
- hitObjectReorderExecuteEXT(…)
- hitObjectTraceReorderExecuteEXT(…)

Sample Use Cases

Basic Multi-Bounce

- Decouple intersection from shading:
  1. Trace a ray with hitObjectTraceRayEXT() to capture potential hit.
  2. reorderThreadEXT() if needed for coherence.
  3. hitObjectExecuteShaderEXT() to run the closest-hit logic.
- Repeat for multiple bounces: each bounce can do partial shading and reorder again.
- Ideal for path-tracing or multi-bounce effects where shading divergence is high.
- Reordering before checking the hit is most important for coherence when the threads continue after the loop

Sample Use Cases

Basic Multi-Bounce

```
#version 460
#extension GL_EXT_ray_tracing : enable
#extension GL_EXT_shader_invocation_reorder : enable

layout(binding = 0) uniform accelerationStructureEXT topLevelAS;
layout(location = 0) rayPayloadEXT vec3 accumulatedColor;

void main() {
    vec3 rayOrigin = ...; vec3 rayDir = ...;

    for(int i = 0; i < MAX_BOUNCES; i++) {
        hitObjectEXT hObj;
        hitObjectRecordEmptyEXT(hObj);

        // Trace for intersection
        hitObjectTraceRayEXT(hObj, topLevelAS,
          0, 0, 0, 0, // Ray flags & SBT offsets
          0, // Miss index
          rayOrigin, 0.001, rayDir, 1000.0, 0);

        // Reorder based on the intersection state
        reorderThreadEXT(hObj);

        // Invoke shading on a hit
        if(hitObjectIsHitEXT(hObj)) {
            hitObjectExecuteShaderEXT(hObj, 0);

            // Update bounce info: compute reflection, new ray dir, etc.
            rayOrigin = ... ; rayDir = ... ;
        } else {
            // Miss - accumulate environment color
            accumulatedColor += vec3(0.1, 0.1, 0.1);
            break;
        }
    }

    // Further code that processes the multi-bound result here
```

Sample Use Cases

Multi-Bounce with Russian Roulette

- Russian roulette sampling can be useful but frequently leads to poor occupancy
- Including the Russian roulette sampling decision in the hint for an existing sort is close to free
- Reordering to improve coherence on exit from a loop is an important technique in general
- Because the estimate of the sampling decision ahead of time is only used for reordering that can only effect performance from incorrect reordering, not correctness since the final decision is still computed after reorder

Sample Use Cases

Multi-Bounce with Russian Roulette

```
…
void main() {
    vec3 rayOrigin = ...; vec3 rayDir = ...;

    for(int i = 0; i < MAX_BOUNCES; i++) {
        hitObjectEXT hObj;
        hitObjectRecordEmptyEXT(hObj);

        // Trace for intersection
        hitObjectTraceRayEXT(hObj, topLevelAS,
          0, 0, 0, 0, // Ray flags & SBT offsets
          0, // Miss index
          rayOrigin, 0.001, rayDir, 1000.0, 0);

        float albedo = SRB(hitObjectGetShaderRecordBufferHandleEXT(hObj)).albedo;

        uint coherenceHint = russianRoulette(albedo) || i == MAX_BOUNCES-1;

        // Reorder based on the intersection state
        reorderThreadEXT(hObj, coherenceHint, 1);

        // Invoke shading on a hit
        if(hitObjectIsHitEXT(hObj) && payload.needsNextBounce) {
            hitObjectExecuteShaderEXT(hObj, 0);

            // Update bounce info: compute reflection, new ray dir, etc.
            rayOrigin = ... ; rayDir = ... ;
        } else {
            // Miss - accumulate environment color
            accumulatedColor += vec3(0.1, 0.1, 0.1);
            break;
        }
    }
}
```

Sample Use Cases

Near-Field/Far-Field

- Large scenes often split geometry into near-/far-field for efficiency.
- Using invocation reorder, you can do:
  1. Trace near-field, reorder and shade.
  2. For misses, trace far-field.
- Freed from artificially splitting passes, invocation reorder can reorder as one flow.
- This avoids storing intermediate results and launching separate passes.

Sample Use Cases

Near-Field/Far-Field

```
#version 460
#extension GL_EXT_ray_tracing : enable
#extension GL_EXT_shader_invocation_reorder : enable
layout(binding = 0) uniform accelerationStructureEXT nearFieldAS;
layout(binding = 0) uniform accelerationStructureEXT farFieldAS;
layout(location = 0) rayPayloadEXT vec4 shadingData;
void main() {
    vec3 rayOrigin = ...; vec3 rayDir = ...;
    hitObjectEXT hNear; hitObjectEXT hFar;

    hitObjectRecordEmptyEXT(hNear); hitObjectRecordEmptyEXT(hFar);

    hitObjectTraceRayEXT(hNear, nearFieldAS, 0, 0, 0, 0, 0, rayOrigin, 0.0, rayDir, 1000.0, 0);

    // Reorder and handle near hits
    reorderThreadEXT(hNear);

    // If no near hit, try far field in the same pass, using miss from far field if both miss
    if(! hitObjectIsHitEXT(hNear)) {
        // Transform ray origin and direction to far field TLAS as appropriate
        hitObjectTraceRayReorderExecuteEXT(hFar, farFieldAS, 0, 0, 0, 1, 0, rayOrigin, 0.0, rayDir, 100000.0, 0);
    } else {
        hitObjectExecuteShaderEXT(hNear, 0);
    }

}
```

Sample Use Cases

Common Computations with Hit Coherence

- Separating trace from execute allows extracting common computations, which may also benefit from coherence
- Can help remove variables from the payload as well

```
hitObjectTraceRayEXT(hit, topLevelAS, rayFlags, cullMask, sbtRecordOffset, sbtRecordStride, missIndex, origin, tMin, direction, tMax,
0);
reorderThreadEXT(hit);
payload.giData = GlobalIlluminationCacheLookup(hit);

// Finally, invoke the closest-hit or miss shader that 'hit' represents
hitObjectExecuteShaderEXT(hit, 0);
```

General Best Practices

Other Uses for Hit Objects

- Hit Objects enable other techniques with ray tracing other than the primary patterns
- Not executing closest hit after intersection
  - Ray pipelines provide the ability to skip closest hit execution, but that drops all hit information on a hit
  - Ray query gives hit information but doesn't integrate with any hit or intersection
  - Hit objects provide the ability to generate the intersection then inspect the results without invoking closest hit
- Executing multiple closest hit after intersection
  - Can split closest hit shading into multiple passes, for example splitting specific from shared
  - Can reorder between the multiple closest hit invocations to maximize coherence on each pass
- Custom shader table indexing and visibility
  - Return to the ray generation shader between intersection and execution allows code to modify the shader binding table index with hitObjectSetShaderBindingTableRecordIndexEXT(…)
  - Validation code can inspect the shader binding table index using hit object functionality

General Best Practices

When to Use or Not Use Reorder

- Reordering threads has overhead, so use it strategically.
- Great for highly divergent shaders (like path tracing with many materials).
- Less beneficial if your scene or materials are already quite coherent.
- Use profiling to decide whether reorder overhead is outweighed by the gains.

General Best Practices

Optimizing Live State

- Keep only essential data live in the pipeline when reordering threads.
- Extra global variables or unneeded resources can degrade performance.
- Minimize hidden data dependencies that hamper invocation reorder’s scheduling freedom.
- If you must keep a pointer or handle, ensure it’s easily broadcast or updated.
- Keep shading code well-structured to let the compiler optimize reordering paths.
- Validate that your memory usage patterns remain coherent after reordering.
- Update shared data carefully: reordering can reorder your writes as well.

General Best Practices

Use Coherence Hint Bits Carefully

- Don’t duplicate information already in the sort in hint bits
  - The spec defines this in decreasing order of priority for information used in the reorder
  - If a coherence hint bit is implied by the ShaderID, don’t include it in the hint
- Use minimum number of hint bits
  - Implementations may have a limited number of bits to use for the reorder
  - Reserving application hint bits that aren’t useful in the reorder limits the amount of hit object information that can be used