SHADER EXECUTION REORDERING

V1.0

Introduction

NVIDIA's introduction of RTX in 2018 was a milestone in computer graphics that enabled hardware-accelerated raytracing in games and professional rendering packages. Since then, raytracing has been used for increasingly complex and computationally costly effects. Applications simulate rich materials, employ sophisticated sampling strategies, and rely on advanced light transport algorithms such as path tracing. Because such computations happen in shader code, which runs on the SM rather than on the RT Core hardware, the speed of shader execution (as opposed to the speed of casting rays) plays an increasing role in overall performance. This makes efficient shader execution crucial.

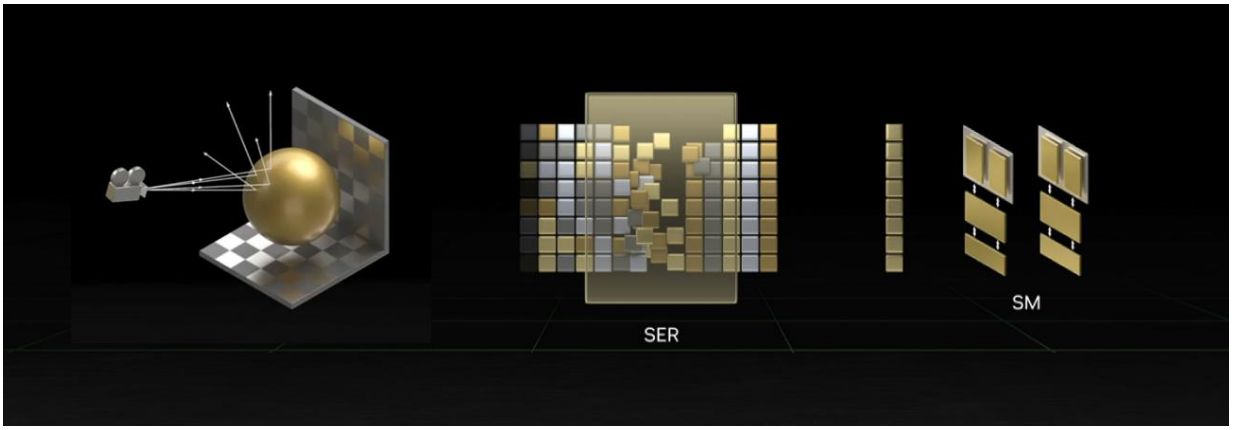

Figure 1: Left: rays bounce off an object in different directions, hitting different materials. Center: SER reorders threads, grouping similar work together. Right: SMs execute shaders with increased coherence.

Many raytracing workloads by nature exhibit a high amount of *divergence.* The GPU must execute different code paths at once (*execution divergence)* and access data in patterns that are difficult to coalesce or cache (*data divergence)*. To see why this might happen, picture a group of rays bouncing off some surface in random directions. Even rays that originate closely together will hit different objects made of different materials and surface characteristics. Evaluating these different materials requires running different shaders and accessing different textures, vertex attributes, and other per-object information. The divergence that results from this is undesirable because modern GPUs perform best when the workload is *coherent,* that is, when groups of threads execute similar work and access similar data.

Shader Execution Reordering (SER) is a new scheduling technology introduced with the Ada Lovelace generation of NVIDIA GPUs. It is highly effective at simultaneously reducing both execution divergence and data divergence. SER achieves this by on-the-fly reordering threads across the GPU such that groups of threads perform similar work and therefore use GPU resources more efficiently. This happens with minimal overhead: the Ada hardware architecture was designed with SER in mind and includes optimizations to the SM and memory system specifically targeted at efficient thread reordering. Using SER, we observe speedups of up to 2x in raytracing regimes of real-world applications, achieved with only a small amount of developer effort. To applications, SER is exposed through a small API that gives developers new flexibility and full control over where in their shaders reordering will happen. This API is detailed in the following sections.

API overview

ReorderThread

The main functionality of SER is encapsulated in a single function:

```
void ReorderThread( key )
```

This function is available in shaders of type raygeneration, and it does what its name suggests: it asks the system to reorder the calling thread, along with other threads that also call ReorderThread, across the physical execution units of the GPU. After the function returns, threads that execute together (e.g., in the same warp or on the same SM) will be more coherent than before ReorderThread. Coherence is measured with respect to the argument passed to ReorderThread, which can be thought of as a sort key. In most cases, that key will be a HitObject (described below), which represents the hit location of a ray that was traced into the scene. However, there are also variants of ReorderThread where 'key' is simply a number, allowing it to be used for generic reordering independent of raytracing.

ReorderThread and raytracing

To see how ReorderThread fits into the raytracing programming model, let's recall that the standard control flow for a ray tracing operation looks as depicted in Figure 2. The most common and significant source of shading divergence is closesthit shading, or more generally, any code

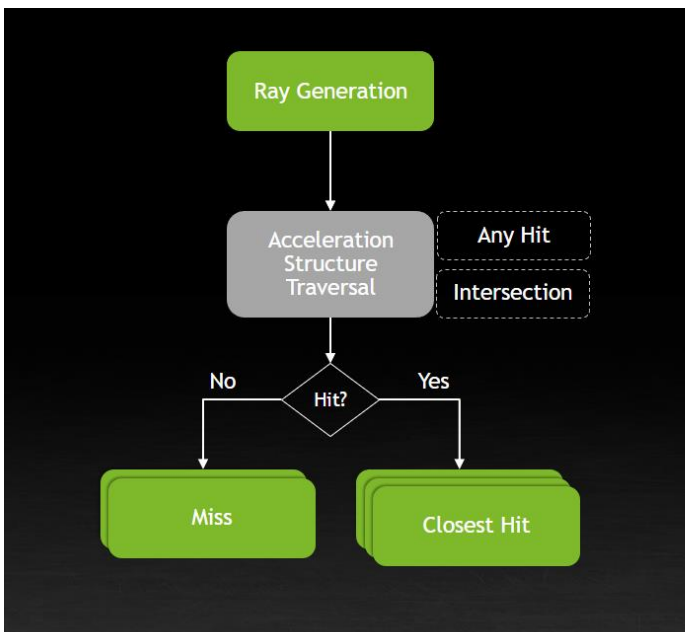

Figure 2: Control flow of TraceRay.

that performs material or scene-object specific computations. Therefore, we would like reordering to happen after a ray has been cast (i.e., when we know the hit location in the scene), but before further shading takes place. In other words, we'd usually like to reorder at the position indicated in Figure 3. It's worth noting that the standard APIs do allow systems to perform such reordering "under the hood", but this has a downside: it does not allow the application to influence the decision making on whether and how to reorder. That is, application-side knowledge, which is often key to performance, cannot be taken into account. This is where HitObject, combined with ReorderThread, comes into play.

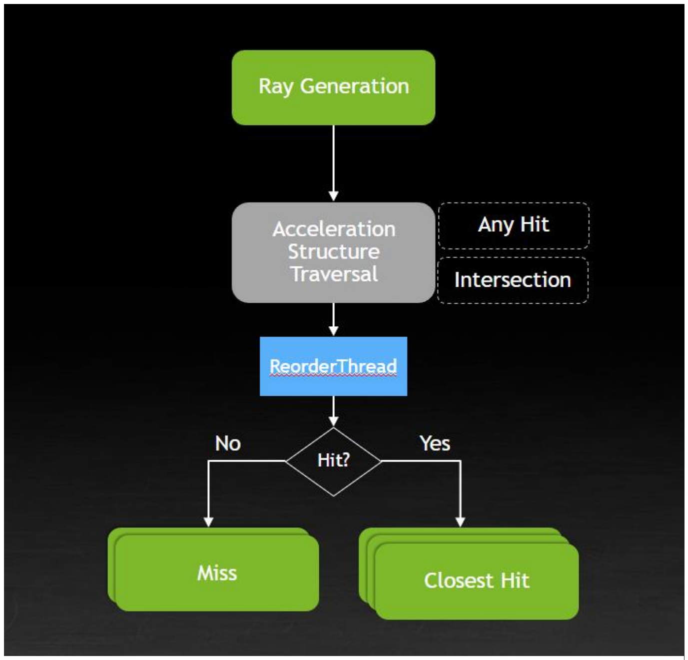

Figure 3: A good position for reordering is after acceleration structure traversal.

HitObject

In order to give the application explicit control over reordering, we introduce new API concepts that split the TraceRay operation into two phases:

1. Tracing the ray through the BVH, including executing potential anyhit and intersection shaders, and determining the hit point

2. Shading the hit point by executing a closesthit or miss shader.

The application can execute arbitrary code between those two phases, including a call to ReorderThread. The splitting up of tracing and shading is supported via a new API object called HitObject. A HitObject simply stores the result of the trace operation, i.e., which primitive in the scene was hit by the ray (if any). In addition, it provides functionality to explicitly *invoke* the hit object, which will trigger closesthit or miss shading.

Crucially, a HitObject can be used directly as an argument to ReorderThread. This causes the system to interpret the HitObject as a sort key that determines the coherence between different threads. During reordering, the system considers several pieces of information contained in the HitObject, such as shader ID and the 3D location of the hit. This is what allows SER to simultaneously tackle both execution divergence (by grouping the same shaders together) and data divergence (by grouping hits on the same object in the scene). By combining HitObject and ReorderThread, a traditional TraceRay call can be turned into this (pseudocode) sequence:

```
// New variant of tracing a ray that does not execute closesthit/miss shading,
// but instead stores the resulting hit data in a HitObject
HitObject hit = TraceRayHitObject( ray, payload );

// Reorder threads for coherence, based on what our ray has hit
ReorderThread( hit );

// Trigger closesthit/miss shading
InvokeHitObject( hit, payload );
```

This is the basic pattern that should serve as a starting point for developers working with SER. Performing a simple substitution of original TraceRay calls with this sequence can already result in significant performance gains. But there is more on the table.

Coherence hints

Using the pattern described in the previous section as a basis, the application can further inform reordering with *coherence hints*. Coherence hints are represented as an integer from which some number of bits are incorporated in the reordering key that the system computes from a HitObject. The full ReorderThread signature looks like this:

```
void ReorderThread( HitObject  HitOrMiss,
                    uint       CoherenceHint,
                    uint       NumCoherenceHintBitsFromLSB );
```

Where NumCoherenceBitsFromLSB specifies the number of bits that ReorderThread should take into account, starting from the least significant bit.

The final key used for reordering is then composed from the following components, in descending order of priority:

1. Shader ID stored in the HitObject
2. Coherence hint, with the most significant hint bit having highest priority
3. Spatial information stored in the HitObject

With this key arrangement, the application can use coherence hints to incorporate knowledge about execution divergence into the reordering that isn't already represented in the HitObject.

One example of this is branches within hit shader code, which are often based on material parameters (e.g., is the material emissive, is there a clear coat layer, does it cast a shadow ray, etc.). The application may peek at these parameters and include them in the coherence hint before reordering and executing the shader. One way to store material parameter information is as root constants in the Shader Table. HitObject provides a convenient method to read data from the Shader Table entry that the HitObject represents. Here is an example pattern:

```
HitObject hit = TraceRayHitObject( ray, payload );

// Peek at material flags encoded as a root constant at offset 0 in the
// Shader Table entry of our hit
uint materialFlags = hit.LoadLocalRootTableConstant( 0 );

// Reorder by hit point, using material flags as additional coherence hints.
// Assumes we have 4 bits worth of flags.
ReorderThread( hit, materialFlags, 4 );

// Invoke shading. Thanks to reordering with coherence hints, this will be
// coherent both w.r.t. which shader is executed, _and_ which path is taken
// through the shaders.
InvokeHitObject( hit, payload );
```

Coherence hints can also be used to reorder threads based on control flow in the raygeneration shader itself. In the simple loop depicted below, threads in a warp can be expected to stay on the same iteration without ReorderThread (e.g., first i=0, then i=1). With ReorderThread, after executing the call, the value of i can be different among threads (depending on the result of the reordering, which in turn depends on the contents of the HitObject). If this leads to divergence, it is suitable to add a coherence hint based on i. It is often beneficial to make sure that the entire warp finishes the loop at the same time.

```
for( uint i = 0; i < 4; ++i )
{
    ...

    // Reorder based on a HitObject, while also grouping threads that
    // will exit the loop
    uint coherenceHints = i == 3 ? 1 : 0;
    ReorderThread( hit, coherenceHints, 1 );

    ...
}
```

A related situation can be found in path tracers or multi-bounce reflections. There, it is often highly effective to include in the coherence hint some additional information about whether the main loop will terminate. On top of executing hit shaders coherently, this has the effect of compacting threads into warps for the next iteration of the loop. Hence, the number of inactive lanes from threads that exit the loop early is reduced. This benefits both the execution of the loop itself, as well as the utilization of the RT Core hardware, which operates more effectively the more threads per warp are active at the time TraceRayHitObject is called.

A pseudocode example might look like this:

```
while(..)  // loop over light bounces
{
    HitObject hit = TraceRayHitObject( ray, payload );

    // Before we reorder, figure out if this will be the last loop iteration,
    // and encode that information in a coherence hint bit.
    float albedo = hit.LoadLocalRootTableConstant( 0 );
    bool done = russianRoulette( payload, albedo ) || bounceCount >= maxBounces;
    uint coherenceHints = done ? 1 : 0;

    // Reorder and shade
    ReorderThread( hit, coherenceHints, 1 );
    InvokeHitObject( hit, payload );

    // Because we included the 'done' flag in the coherence hints, chances
    // are good that all threads in the warp will make the same decision about
    // whether or not to break out of the loop.
    if( done )
        break;
}
```

Sometimes, predicting the exact control flow is impossible at the point where a coherence hint is computed. In such situations, it is important to remember that it may be sufficient to have a confident estimate rather than an exact value for the coherence hint. This is because reordering only affects performance, not correctness. In other words, an approximate coherence hint is often better than none at all.

More ways to use HitObject

As we have seen, HitObject is the mechanism that enables the splitting of TraceRay into two phases and by extension the combination of application-side and system-side knowledge to inform reordering. It is worth noting, however, that HitObject is a versatile tool in and of itself. It allows the application to implement concepts that were difficult or impossible to achieve with the traditional raytracing programming model. Some examples that have proven useful in practice include:

Executing closesthit shading without casting a ray

A HitObject may be created from existing hit information, without tracing a ray, by calling MakeHit or MakeMiss. The resulting HitObject may be reordered or invoked just like a HitObject that resulted from TraceRayHitObject. This allows applications to use closesthit shaders in a raytracing pipeline to shade primary hits that are stored in a g-buffer, without the need to trace “dummy rays”.

Executing closesthit shading for rays cast via RayQuery

Similar to the previous example, MakeHit and MakeMiss may be used to create a HitObject based on information obtained from TraceRayInline. This provides an easy path to integrate reordering into existing applications that use RayQuery. In the process, the application might choose to set up a raytracing pipeline with hit shaders and invoke those using a HitObject. Or it might choose to implement its material shading within the raygeneration shader, and benefit from reordering by improved data coherence and optionally add coherence hints to reduce execution divergence.

Not executing closesthit shading after casting a ray
In some situations, it is useful to cast a ray without triggering shading, but while still obtaining basic information about the hit. Shadow or ambient occlusion rays are common examples, where material shading typically isn't needed, but knowing the closest intersection distance can be useful for filtering. The existing API provides the SKIP_CLOSESTHIT_SHADER ray flag, but using it offers no way to obtain information about the closest hit. Using RayQuery is sometimes a viable option, but it does not trigger anyhit or intersection shaders, which may be a requirement. The typical solution is to write a trivial closesthit shader that only fills the payload with the desired hit information. HitObject provides a simpler and more efficient path: one can simply trace the ray using TraceRayHitObject and then inspect the fields of the resulting HitObject without ever calling InvokeHitObject.

Executing multiple closesthit shaders for a single ray
There are scenarios in which an application may want to execute more than one closesthit shader for a single ray. For example, the implementation might be structured such that a first hit shader does some initial material processing, followed by code in the raygeneration shader that's common to all materials, followed by another hit shader that finalizes the material/lighting evaluation. HitObjects make this easy, because there are no restrictions on how HitObjects can be created and invoked with respect to rays that are cast. Of course, this can be freely combined with the use of ReorderThread; potentially even more than once per ray.

Custom indexing into the shader table
The shader table record that TraceRay invokes is found using a formula that takes into account a number of factors from various sources. This provides some amount of flexibility, but the formula itself is fixed and imposes certain constraints. A HitObject may be constructed referencing an arbitrary shader record, which opens new possibilities when it comes to shader table organization. This can be interesting particularly in applications that only use a single anyhit/intersection shader combination across all objects in the scene. In such cases, all instances can refer to the same shader table range used only to trigger anyhit and intersection shaders, while the closesthit or miss shader is selected using a manually computed index.

Best practices

When to use (and when not to use) reordering

The implementation of ReorderThread is highly optimized, and we continuously work on reducing overhead even further. But because it isn't free, and because the benefits of reordering depend on the work that is executed after reordering, there are situations where reordering is not advisable.

For example, hit shaders for shadow or ambient occlusion rays are typically trivial, which means that spending additional cycles to extract coherence for such shaders is usually not worth the cost. The same is true for cases where rays are very coherent to begin with, like primary rays.

In addition, work following ReorderThread may not actually benefit from hit coherence. Consider that reordering threads with respect to their ray hit location means giving up on the 2D screen-space locality that physical threads have by default. This can negatively impact the performance of reading gbuffer data or writing output buffers, which can turn reordering into a net loss when the rest of the workload is particularly cheap.

Where reordering shines is in situations that exhibit non-trivial hit shading (irrespective of whether the shading code lives in the raygeneration shader or in closesthit), paired with at least moderate divergence in the ray distribution. Reflection regimes are a typical example, especially when the reflections are glossy or diffuse. In general, the more numerous and the more complex the shaders, and the more object-space or world-space data (textures, vertex attributes, environment probes, etc.) they access, the more potential there is for reordering to improve performance. Note, however, that a high shader count isn't necessarily required. Increasing data coherence and using coherence hints to mitigate execution divergence can yield performance gains even in applications that use only a single shader across the entire scene.

Scenarios like multi-bounce reflections or full-fledged path tracers increase the potential even further, because additional execution divergence comes from the main loop itself. As indicated in the examples earlier, incorporating information about the loop state into coherence hints can yield additional performance improvements.

Furthermore, when multiple rays are traced in a loop, it can be important to reorder in a way that is optimized towards subsequent rays. E.g., if multiple rays are traced from the same origin with cheap shading, such as in an ambient occlusion shader, reordering after each ray may increase divergence in the ray origin for the next ray. Instead, it can be better to reorder once before the loop based on the common origin. In a path tracing loop, it can make sense to reorder for each radiance ray, but not for the one-off shadow ray from which we won't continue the light path.

Generally speaking, when and where SER is beneficial varies by application and depends on a number of factors. The API can be integrated into most engines with low engineering effort, which makes it easy to experiment with the feature. A good starting point is to replace traditional TraceRay calls with the HitObject/ReorderThread equivalent, which is often enough to show some initial performance gains. From there, further optimizations like coherence hints, live state reduction, etc., can be explored.

Optimizing warp coherence

Most developers familiar with GPU programming are accustomed to the concept of warp coherence, and it is one of the most effective metrics to optimize for when using SER. Warp coherence, that is, the number of threads active in a warp on average, is a good proxy for the

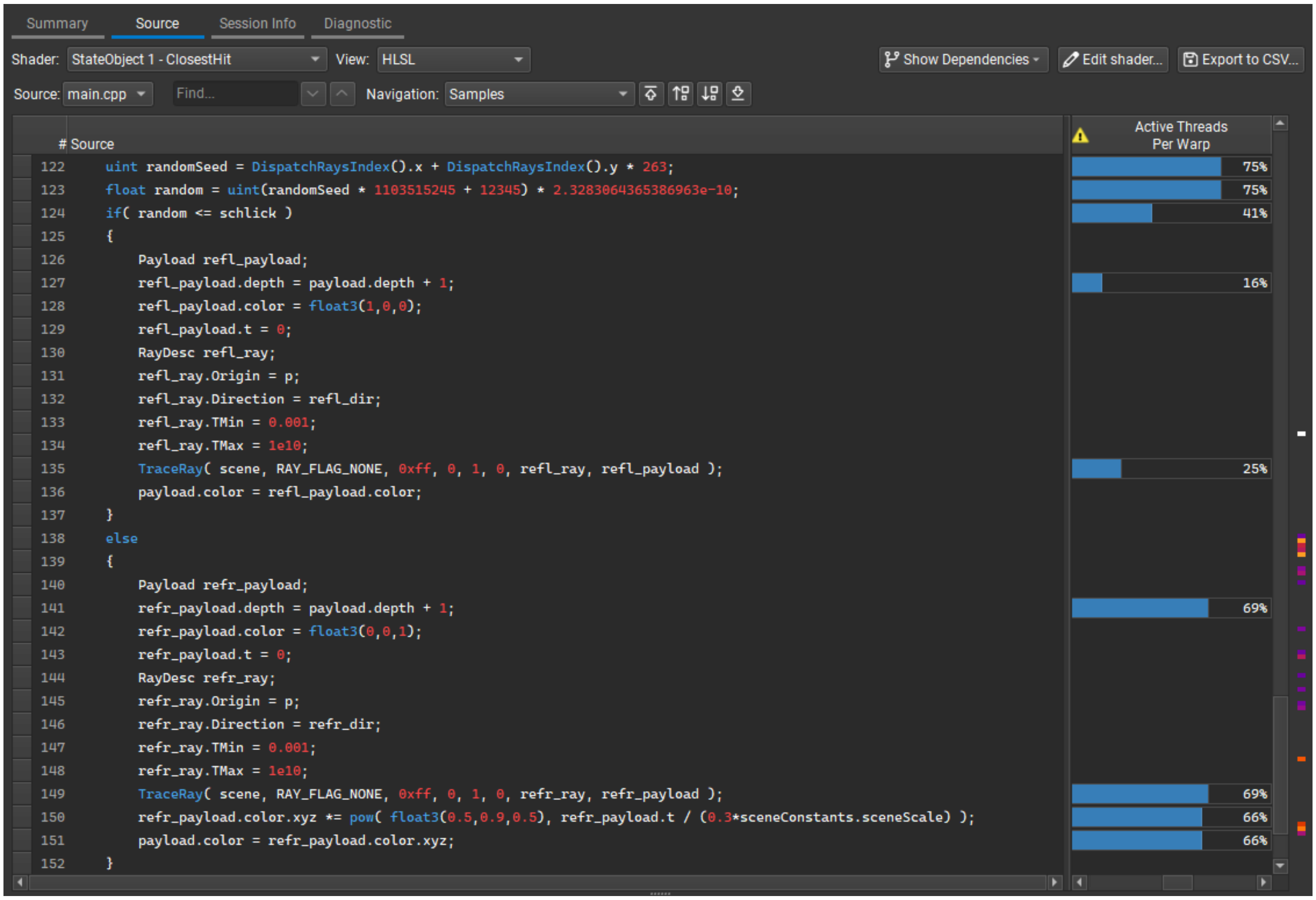

Figure 4: Active threads per warp visualization in Nsight Graphics, showing execution coherence in code with control flow. Unconditional code can be seen to run with less than 100% of threads per warp, as not all threads in the warp execute the closesthit shader depicted here. Using ReorderThread can improve coherence at the closesthit entry, and with a suitable coherence hint the branches can also be made to execute more coherently.

general execution coherence improvements that an application will achieve by using ReorderThread. Recent versions of the *NVIDIA Nsight Graphics* shader profiler can directly visualize warp coherence at individual lines of code with a new “Active Threads Per Warp” counter, as shown in the example in Figure 4.

This feature is extremely useful to quickly spot code paths that suffer from high execution divergence, both in hit shaders and in raygeneration code. This information can then be used to tune the use of reordering. For example, one might try to encode branch decisions into coherence hints, or eliminate the heaviest branches from hit shaders altogether by specializing them (reducing the number of cases handled by “ubershaders”).

Optimizing live state

Reordering threads across execution units on the GPU also requires migrating their *live state* through the memory system. Live state consists of any variables that are defined before reordering and used after reordering; that is, any variables that must persist across a

ReorderThread call. The smaller that state is, the lower the overhead of the ReorderThread call. Applications should therefore strive to reduce live state as much as possible.

Manually optimizing for live state can be challenging, mainly for two reasons. First, unless the shader code is very simple, it is hard to spot variables that are live across a ReorderThread call just by inspecting the shader code. Second, the compiler can often eliminate much of the live state memory traffic, which might lead to the developer optimizing for variables that don't affect performance to begin with.

This is addressed by another new feature in *NVIDIA NSight Graphics.* Live state visualization in the shader profiler shows the developer exactly which variables are both live across a ReorderThread call *and* were not already optimized away by the compiler, i.e., the ones that had to be spilled to memory. Optimizing the application to reduce such live state, e.g. by packing values, using lossy compression, or restructuring control flow, is likely to help performance.

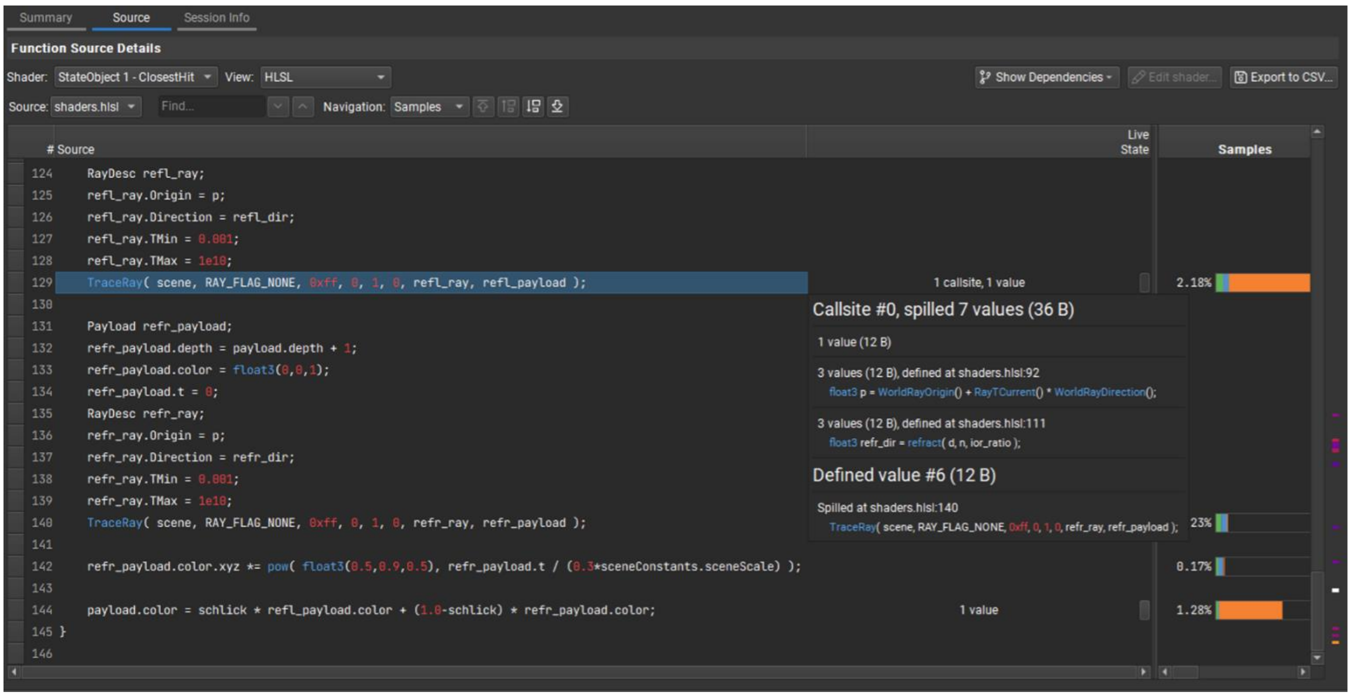

Figure 5: Live state visualization in NSight Graphics showing the amount of data spilled and where it was defined.

Figure 5 shows the feature in an example. In this case, the live state across a TraceRay call rather than a ReorderThread call is shown (reducing live state across TraceRay can improve performance as well).

Note: when SER is used via NVAPI, NSight Graphics will recognize a marker call inside the implementation of NvReorderThread as the relevant call site for reporting live state. This is a side effect of the way HLSL extensions work with NVAPI.

Using coherence hint bits judiciously

As discussed in previous sections, coherence hints are an important mechanism that the developer can use to insert application-side knowledge into the reordering. The more coherence hint bits are used, the higher the granularity at which the application can control reordering. However, using more coherence hint bits also reduces the resolution at which the hit object can be taken into account. Developers should keep this tradeoff in mind and not use more coherence hint bits than necessary. In particular, NumCoherenceHintBitsFromLSB should only be set to the number of coherence hint bits that are actually relevant, and no higher.

Tailoring payload types to invoked shaders

When replacing a traditional TraceRay call with its HitObject equivalents, it is most straightforward to use the same payload type in both TraceRayHitObject and InvokeHitObject. However, it is not mandatory that the payload types used in these functions match. The only requirement is that the payload type used at each call site matches the expectations in the shaders that are actually executed. It is not unusual that the payload requirements between shaders invoked from TraceRayHitObject (anyhit and intersection) and those invoked from InvokeHitObject (closesthit and miss) can differ. A frequent example is that anyhit requires a smaller payload than closesthit, because anyhit is only used for simple alpha testing. In cases like these, the application should use different actual payload types – ideally combined with payload access qualifiers (PAQs) as described in the API reference. This guarantees that register pressure from unnecessary payload fields is avoided as much as possible, which can help increase overall performance.

API Reference

Availability

The API concepts described above will be available for Microsoft DirectX 12 (via NVAPI), Vulkan (via vendor extension), and OptiX. The following reference covers NVAPI. For Vulkan and OptiX, please refer to the respective API documentation once it becomes available.

The SER NVAPI is supported on all raytracing-capable NVIDIA GPUs starting with R520 drivers. Vulkan and OptiX support will be added with later releases.

While the API itself is supported on all raytracing-capable GPUs, the behavior of ReorderThread depends on the GPU architecture: on Ada Lovelace GPUs, ReorderThread will perform reordering, while on previous generation GPUs, it will behave as a no-op. The idea behind this is that once support for the API itself is confirmed, the developer can unconditionally use the HitObject and ReorderThread APIs without adding further shader logic that switches behavior based on whether or not the GPU can actually do reordering. (Note that calling ReorderThread on GPUs that cannot reorder is zero-overhead, because in that case the call is removed by the shader compiler).

The developer can query both whether the SER API is supported and what the behavior of ReorderThread is for the current GPU/driver combination, as described in the following.

Querying API support

The application should use **NvAPI_D3D12_IsNvShaderExtnOpCodeSupported** to check driver support for the SER API. If the query indicates that support exists, then all the functionality covered in this reference (i.e., both ReorderThread and HitObject) is available.

The function is used as follows:

```
bool SERsupported = false;
NvAPI_Status ret = NvAPI_D3D12_IsNvShaderExtnOpCodeSupported(
    pDevice, NV_EXTN_OP_HIT_OBJECT_REORDER_THREAD, &SERsupported );
if( ret == NVAPI_OK && SERsupported )
{
    // OK to use SER NVAPI extensions in shaders
}
else
{
    // Driver does not support the SER NVAPI
}
```

Querying ReorderThread behavior

As mentioned above, the ReorderThread API is available on all raytracing-capable GPUs, but its behavior depends on the GPU architecture. In most cases, the app can ignore this fact, and can simply use ReorderThread as if reordering was always supported. This works because thread reordering affects performance, but not correctness. ReorderThread acting as a no-op on previous generation GPUs can therefore be viewed as an implementation of reordering that just happens to reproduce its input order (producing no benefit, but also not incurring any cost).

However, there are cases where it can make sense to make higher-level decisions based on the behavior of ReorderThread. For example, an application might have a legacy path that performs manual shader binning, which it would like to enable for GPUs that do not support SER reordering.

The behavior of ReorderThread can be queried using **NvAPI_D3D12_GetRaytracingCaps**. Please see the NVAPI header for documentation.

NvHitObject

```
NvHitObject
```

Immutable data type representing a ray hit or a miss. Can be used to invoke hit or miss shading, or as a key in NvReorderThread. Created by one of several methods described below. NvHitObject and its related functions are available in raytracing shader types only.

NvTraceRayHitObject

```
template<typename T>
NvHitObject NvTraceRayHitObject(
    RaytracingAccelerationStructure AccelerationStructure,
    uint RayFlags,
    uint InstanceInclusionMask,
    uint RayContributionToHitGroupIndex,
    uint MultiplierForGeometryContributionToHitGroupIndex,
    uint MissShaderIndex,
    RayDesc Ray,
    inout T Payload )
```

Executes ray traversal (including anyhit and intersection shaders) like TraceRay, but returns the resulting hit information as a HitObject and does not trigger closesthit or miss shaders.

NvMakeHit

```
template <typename T>
NvHitObject NvMakeHit(
    RaytracingAccelerationStructure AccelerationStructure,
    uint InstanceIndex,
    uint GeometryIndex,
    uint PrimitiveIndex,
    uint HitKind,
    uint RayContributionToHitGroupIndex,
    uint MultiplierForGeometryContributionToHitGroupIndex,
    RayDesc Ray,
    T Attributes )
```

Creates a HitObject representing a hit based on values explicitly passed as arguments, without tracing a ray. The primitive specified by AccelerationStructure, InstanceIndex, GeometryIndex, and PrimitiveIndex must exist. The shader table index is computed using the formula used with TraceRay. The computed index must reference a valid hit group record in the shader table. The Attributes parameter must either be an attribute struct, such as BuiltInTriangleIntersectionAttributes, or another HitObject to copy the attributes from.

NvMakeHitWithRecordIndex

```
template <typename T>
NvHitObject NvMakeHitWithRecordIndex(
    uint HitGroupRecordIndex,
    RaytracingAccelerationStructure AccelerationStructure,
    uint InstanceIndex,
    uint GeometryIndex,
    uint PrimitiveIndex,
    uint HitKind,
    RayDesc Ray,
    T Attributes )
```

Creates a HitObject representing a hit based on values explicitly passed as arguments, without tracing a ray. The primitive specified by AccelerationStructure, InstanceIndex, GeometryIndex, and PrimitiveIndex must exist. The shader table index is explicitly provided as an argument instead of being computed from the indexing formula used in TraceRay. The provided index must reference a valid hit group record in the shader table. The Attributes parameter must either be an attribute struct, such as BuiltInTriangleIntersectionAttributes, or another HitObject to copy the attributes from.

NvMakeMiss

```
NvHitObject NvMakeMiss(
    uint MissShaderIndex,
    RayDesc Ray )
```

Creates a HitObject representing a miss based on values explicitly passed as arguments, without tracing a ray. The provided shader table index must reference a valid miss record in the shader table.

NvMakeNop

```
NvHitObject NvMakeNop()
```

Creates a HitObject representing "NOP" (no operation) which is neither a hit nor a miss. Invoking a NOP hit object using NvInvokeHitObject has no effect. Reordering by hit objects using ReorderThread will group NOP hit objects together. This can be useful in some reordering scenarios where future control flow for some threads is known to process neither a hit nor a miss.

NvInvokeHitObject

```
template<typename T>
void NvInvokeHitObject(
    RaytracingAccelerationStructure AccelerationStructure,
    NvHitObject HitObj,
    inout T Payload )
```

Invokes closesthit or miss shading for the specified hit object. In case of a NOP HitObject, no shader is invoked.

HitObject status

```
bool NvHitObject::IsMiss()
bool NvHitObject::IsHit()
bool NvHitObject::IsNop()
```

Queries whether the hit object encodes a miss, a hit, or neither. One and only one of these functions will return true for a given hit object.

HitObject getters

```
RayDesc NvHitObject::GetRayDesc()
uint    NvHitObject::GetShaderTableIndex()
```

Queries properties from HitObject. Valid if the hit object represents a hit or a miss.

```
uint NvHitObject::GetInstanceID()
uint NvHitObject::GetInstanceIndex()
uint NvHitObject::GetPrimitiveIndex()
uint NvHitObject::GetGeometryIndex()
uint NvHitObject::GetHitKind()
template<typename T> T NvHitObject::GetAttributes()
```

Queries properties from HitObject. Valid if the hit object represents a hit.

LoadLocalRootTableConstant

```
uint NvHitObject::LoadLocalRootTableConstant( uint RootConstantOffsetInBytes )
```

Loads a root constant from the local root table referenced by the hit object. Valid if the hit object represents a hit or a miss. RootConstantOffsetInBytes must be a multiple of 4.

NvReorderThread

NvReorderThread causes the calling thread to be eligible for migration to another physical execution unit by the system at the location of the call to NvReorderThread. Reordering is done at the best of the system's capabilities to maximize the resulting thread coherence (as measured by the specified key arguments). However, no guarantees are made about the resulting thread order.

In particular, the resulting order is usually non-deterministic, may only approximate the order of the specified keys, or may be identical to the original order. The latter behavior is what is observed on older generation GPUs, where ReorderThread acts as a no-op.

NvReorderThread is only available in shaders of type raygeneration.

NvReorderThread (generic)

```
void NvReorderThread(
    uint CoherenceHint,
    uint NumCoherenceHintBits )
```

Reorders threads based on a coherence hint value. NumCoherenceHintBits indicates how many of the least significant bits of CoherenceHint should be considered during reordering (max: 16). Applications should set this to the lowest value required to represent all possible values in CoherenceHint. For best performance, all threads should provide the same value for NumCoherenceHintBits.

Where possible, reordering will also attempt to retain locality in the thread's launch indices (DispatchRaysIndex in DXR).

NvReorderThread (with HitObject)

```
void NvReorderThread(
    NvHitObject HitObj,
    Uint        CoherenceHint,
    uint        NumCoherenceHintBits )
```

Reorders threads based on a hit object, optionally extended by a coherence hint value. Coherence hints behave as described in the generic variant of NvReorderThread. The maximum number of coherence hint bits in this variant of NvReorderThread is 8. If no coherence hint is desired, set NumCoherenceHitBits to zero.

Reordering will consider information in the HitObject and coherence hint with the following priority:

1. Shader ID stored in the HitObject
2. Coherence hint, with the most significant hint bit having highest priority
3. Spatial information stored in the HitObject

That is, NvReorderThread will first attempt to group threads whose HitObject references the same shader ID. (Miss shaders and NOP HitObjects are grouped separately). Within each of these groups, it will attempt to order threads by the value of their coherence hints. And within ranges of equal coherence hints, it will attempt to maximize locality in 3D space of the ray hit (if any).

Interaction with Payload Access Qualifiers

Payload Access Qualifiers (PAQs) are based on how information flows between shader stages in TraceRay:

```
caller -> anyhit -> (closesthit|miss) -> caller
           ^  |
           |__|
```

With HitObject, control is returned to the caller after TraceRayHitObject completed. Therefore, caller is inserted between anyhit and (closesthit|miss):

```
caller -> anyhit -> caller
           ^  |
           |__|

caller -> (closesthit|miss) -> caller
```

To allow interchangeability of payload types between TraceRay and the new execution model introduced with TraceRayHitObject/ InvokeHitObject, the following additional PAQ rules apply:

- At the call to TraceRayHitObject, any field declared as read(miss) or read(closesthit) is treated as read(caller)

- At the call to InvokeHitObject, any field declared as write(anyhit) is treated as write(caller)

Integrating NVAPI HLSL extensions

The shader-side NVAPI that covers SER is accessed and enabled just like other NVAPI HLSL extensions. See the following page for an example:

https://developer.nvidia.com/blog/profiling-dxr-shaders-with-timer-instrumentation/

In addition, when compiling HLSL to DXIL, either

a) make sure that templates (along with other new features) are enabled in DXC. This is done by specifying the following command line argument: **-HV 2021**

b) use the "macro" version of the API which does not require templates. It can be enabled by #defining NV_HITOBJECT_USE_MACRO_API before #including nvHLSLExtns.h. The macro version has limitations on attribute types and is mainly intended for legacy codebases that have difficulties switching to HLSL 2021. The recommended path is a) if the codebase can support it.