NVIDIA OptiX 9.0

Programming Guide

30 January 2025
Version 1.17

NVIDIA OptiX 9.0 – Programming Guide

Document build number CL35466206

Contents

Preface

DirectX Raytracing (DXR),[1] Vulkan[2] (through the `VK_NV_ray_tracing` extension) and the NVIDIA OptiX™ API[3] employ a similar programming model to support ray tracing capabilities. DXR and Vulkan enable ray tracing effects in raster-based gaming and visualization applications. NVIDIA OptiX is intended for ray tracing applications that use NVIDIA® CUDA® technology, such as:

- Film and television visual effects
- Computer-aided design for engineering and manufacturing
- Light maps generated by path tracing
- High-performance computing
- LIDAR simulation

NVIDIA OptiX also includes support for motion blur and multi-level transforms, features required by ray-tracing applications designed for production-quality rendering.

Terms used in this document

This document and the OptiX API use abbreviations for the software components of OptiX.

The nine types of user-defined ray interactions, called *programs*, are abbreviated as follows:

Program type	*Abbreviation*
Ray generation	RG
Intersection	IS
Any hit	AH
Closest hit	CH
Miss	MS
Exception	EX
Direct callable	DC
Continuation callable	CC

The NVIDIA OptiX program types resemble shaders in traditional rendering systems; the term "shader" is sometimes used in the names of API elements.

[1] `https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html`

[2] `https://www.khronos.org/vulkan/`

[3] `https://developer.nvidia.com/optix/`

The geometry of the scene to be rendered is optimized for ray tracing through *acceleration structures*:

Acceleration structure type	*Abbreviation in this document*	*Abbreviation in the API*
Geometry acceleration structure	geometry-AS	GAS
Instance acceleration structure	instance-AS	IAS
Acceleration structures in general		AS
Bottom-level acceleration structure (DXR and Vulkan)	BLAS	
Top-level acceleration structure (DXR and Vulkan)	TLAS	

The relationship of NVIDIA OptiX programs and the elements of the acceleration structures with which they interact are defined in the *shader binding table*, abbreviated as "SBT". (Note that "shader" in this context refers to an NVIDIA OptiX "program.") No other terms associated with the shader binding table are abbreviated.

Other abbreviations in the document include:

Term	*Abbreviation*
application programming interface	API
axis-aligned bounding box	AABB
graphics processing unit	GPU
high dynamic range	HDR
just-in-time	JIT
low dynamic range	LDR
multiple instruction, multiple data	MIMD
parallel thread execution	PTX
scaling, rotation, translation	SRT
streaming assembly [language]	SASS
streaming multiprocessor	SM

In this document and in the names of API elements, the "host" is the processor that begins execution of an application. The "device" is the GPU with which the host interacts. A "build" is the creation of an acceleration structure on the device as initiated by the host.

1 Overview

The NVIDIA OptiX API is a CUDA-centric API that is invoked by a CUDA-based application. The API is designed to be stateless, multi-threaded and asynchronous, providing explicit control over performance-sensitive operations like memory management and shader compilation.

It supports a lightweight representation for scenes that can represent instancing, vertex- and transform-based motion blur, with built-in triangles, built-in swept curves, built-in spheres, and user-defined primitives. The API also includes highly-tuned kernels and neural networks for machine-learning-based denoising.

An NVIDIA OptiX context controls a single GPU. The context does not hold bulk CPU allocations, but like CUDA, may allocate resources on the device necessary to invoke the launch. It can hold a small number of handle objects that are used to manage expensive host-based state. These handle objects are automatically released when the context is destroyed. Handle objects, where they do exist, consume a small amount of host memory (typically less than 100 kilobytes) and are independent of the size of the GPU resources being used. For exceptions to this rule, see "Program pipeline creation" (page 61).

The application invokes the creation of acceleration structures (called *builds*), compilation, and host-device memory transfers. All API functions employ CUDA streams and invoke GPU functions asynchronously, where applicable. If more than one stream is used, the application must ensure that required dependencies are satisfied by using CUDA events to avoid race conditions on the GPU.

Applications can specify multi-GPU capabilities with a few different recipes. Multi-GPU features such as efficient load balancing or the sharing of GPU memory via NVLINK must be handled by the application developer.

For efficiency and coherence, the NVIDIA OptiX runtime—unlike CUDA kernels—allows the execution of one task, such as a single ray, to be moved at any point in time to a different lane, warp or streaming multiprocessor (SM). (See section "Kernel Focus"[1] in the CUDA Toolkit Documentation.[2]) Consequently, applications cannot use shared memory, synchronization, barriers, or other SM-thread-specific programming constructs in their programs supplied to OptiX.

The NVIDIA OptiX programming model provides an API that future-proofs applications: as new NVIDIA hardware features are released, existing programs can use them. For example, software-based ray tracing algorithms can be mapped to hardware when support is added or mapped to software when the underlying algorithms or hardware support such changes.

[1] `https://docs.nvidia.com/cuda/cuda-gdb/index.html#kernel-focus`

[2] `https://docs.nvidia.com/cuda/`

2 Basic concepts and definitions

2.1 Program

In NVIDIA OptiX, a *program* is a block of executable code on the GPU that represents a particular shading operation. This is called a *shader* in DXR and Vulkan. For consistency with prior versions of NVIDIA OptiX, the term *program* is used in the current documentation. This term also serves as a reminder that these blocks of executable code are programmable components in the system that can do more than shading. See "Program input" (page 62).

2.2 Program and data model

NVIDIA OptiX implements a single-ray programming model with ray-generation, any-hit, closest-hit, miss and intersection programs.

The ray tracing pipeline provided by NVIDIA OptiX is implemented by eight types of programs:

Ray generation

The entry point into the ray tracing pipeline, invoked by the system in parallel for each pixel, sample, or other user-defined work assignment. See "Ray generation launches" (page 117).

Intersection

Implements a ray-primitive intersection test, invoked during traversal. See "Traversing the scene graph" (page 8) and "Ray information" (page 138).

Any hit

Called when a traced ray finds a new, potentially closest, intersection point, such as for shadow computation. See "Ray information" (page 138).

Closest hit

Called when a traced ray finds the closest intersection point, such as for material shading. See "Constructing a path tracer" (page 79).

Miss

Called when a traced ray misses all scene geometry. See "Constructing a path tracer" (page 79).

Exception

Exception handler, invoked for conditions such as stack overflow and other errors. See "Exceptions" (page 145).

Direct callables

Similar to a regular CUDA function call, direct callables are called immediately. See "Callables" (page 153).

Continuation callables

Unlike direct callables, continuation callables are executed by the scheduler. See "Callables" (page 153).

The ray-tracing "pipeline" is based on the interconnected calling structure of the eight programs and their relationship to the search through the geometric data in the scene, called a *traversal*. Figure 2.1 is a diagram of these relationships:

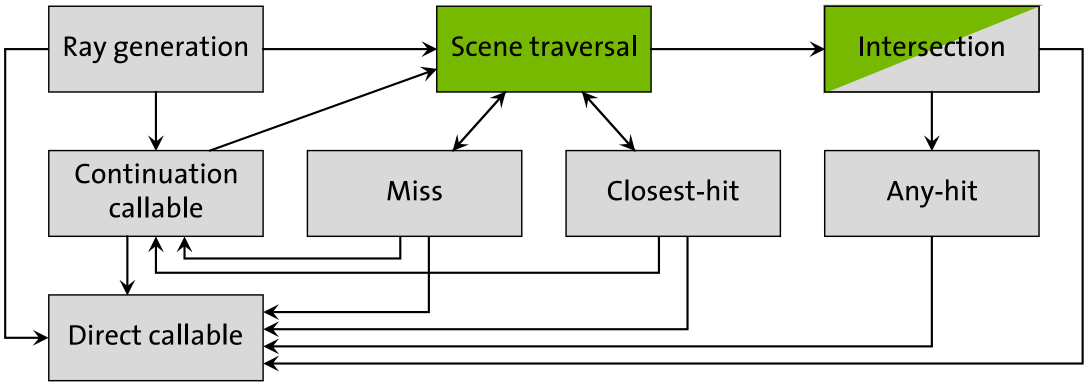

Fig. 2.1 - Relationship of NVIDIA OptiX programs. Green represents fixed functions; gray represents user programs.

In Figure 2.1, green represents fixed-function, hardware-accelerated operations, while gray represents user programs. The built-in or user-provided exception program may be called from any program or scene traversal in case of an exception if exceptions are enabled.

2.2.1 Shader binding table

The *shader binding table* connects geometric data to programs and their parameters. A *record* is a component of the shader binding table that is selected during execution by using offsets specified when acceleration structures are created and at runtime. A record contains two data regions, *header* and *data*.

Record header

- Opaque to the application, filled in by `optixSbtRecordPackHeader`
- Used by NVIDIA OptiX to identify programmatic behavior. For example, a primitive would identify the intersection, any-hit, and closest-hit behavior for that primitive in the header.

Record data

- Opaque to NVIDIA OptiX
- User data associated with the primitive or programs referenced in the headers can be stored here, for example, program parameter values.

2.2.2 Ray payload

The *ray payload* is used to pass data between `optixTrace` and the programs invoked during ray traversal. Payload values are passed to and returned from `optixTrace`, and follow a copy-in/copy-out semantic. There is a limited number of payload values, but one or more of these values can also be a pointer to stack-based local memory, or application-managed global memory. See "Payload" (page 149).

2.2.3 Primitive attributes

Attributes are used to pass data from intersection programs to the any-hit and closest-hit programs. Triangle intersection provides two predefined attributes for the barycentric coordinates (U,V). User-defined intersections can define a limited number of other attributes that are specific to those primitives.

2.2.4 Buffer

NVIDIA OptiX represents GPU information with a pointer to GPU memory. References to the term "buffer" in this document refer to this GPU memory pointer and the associated memory contents. Unlike NVIDIA OptiX 6, the allocation and transfer of buffers is explicitly controlled by user code.

2.3 Acceleration structures

NVIDIA OptiX acceleration structures are opaque data structures built on the device. Typically, they are based on the *bounding volume hierarchy*[1] model, but implementations and the data layout of these structures may vary from one GPU architecture to another.

NVIDIA OptiX provides two basic types of acceleration structures:

Geometry acceleration structures

- Built over primitives (triangles, curves, spheres, or user-defined primitives)

Instance acceleration structures

- Built over other objects such as acceleration structures (either type) or motion transform nodes
- Allow for instancing with a per-instance static transform

2.4 Opacity micromaps

NVIDIA OptiX opacity micromaps are opaque data structures built on the device. An opacity micromap specifies detailed opacity information for a triangle. See "Opacity micromaps" (page 43).

[1] `https://en.wikipedia.org/wiki/Bounding_volume_hierarchy`

2.5 Traversing the scene graph

To determine the intersection of geometric data by a ray, NVIDIA OptiX searches a graph of nodes composed of acceleration structures and transformations. This search is called a *traversal*; the nodes in the graph are called *traversable objects* or *traversables*.

The following types of traversable objects exist:

- An instance acceleration structure
- A geometry acceleration structure (as a root for graph with a single geometry acceleration structure (see Traversal of a single geometry acceleration structure (page 35))
- Static transform
- Matrix motion transform
- Scaling, rotation, translation (SRT) motion transform

For transformation traversables, the corresponding transformation applies to all descendant child traversables (the sub graph spanned by the child of the transformation traversable). The transformation traversables should only be used in case of motion as applying transformations to geometry is order dependent and motion transformations are time dependent. Static transformations are available as they cannot be merged with any motion transformation due to time-dependency, but should be merged with instance transformations (if desired as the child of an instance) or any other static transformation (i.e., there should be at most one static transformation following a motion transformation). For example, Figure 2.2 combines both types:

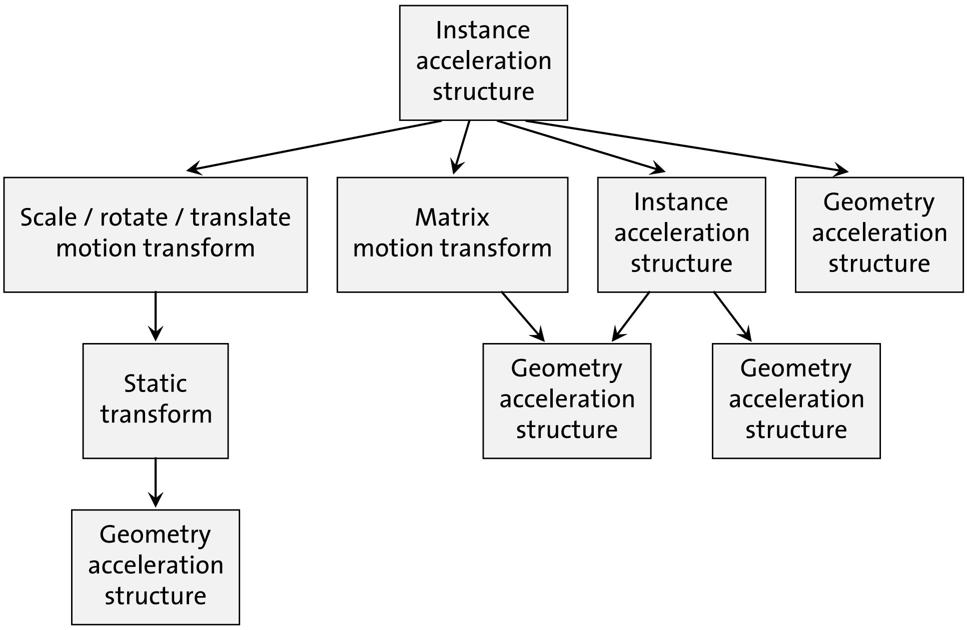
Instance acceleration structure
Scale / rotate / translate motion transform
Matrix motion transform
Instance acceleration structure
Geometry acceleration structure
Static transform
Geometry acceleration structure
Geometry acceleration structure
Geometry acceleration structure

Fig. 2.2 - Example graph of traversables for a scene containing static as well as dynamic motion-transform driven objects

OptiX uses handles as references to traversable objects. These *traversable handles* are 64-bit opaque values that are generated from device memory pointers for the graph nodes. The handles identify the connectivity of these objects. All calls to `optixTrace` begin at a traversable handle.

Note: DXR and VulkanRT use the terms *top-level acceleration structure* and *bottom-level acceleration structure*. A bottom-level acceleration structure is the same as a geometry acceleration structure; a top-level acceleration structure is similar to an instance acceleration structure. Traversing against a single geometry acceleration structure, motion transform nodes, or nested instance acceleration structures (multi-level instancing) are not supported in DXR or VulkanRT. In NVIDIA OptiX, the terms were changed due to the additional possible configurations of scene graphs beyond the strict two-level, top-bottom configurations supported by DXR and VulkanRT. (See DirectX Raytracing (DXR) Functional Spec.[2])

2.6 Ray tracing with NVIDIA OptiX

A functional ray tracing system is implemented by combining four components as described in the following steps:

1. Create one or more acceleration structures over one or many geometry meshes and instances of these meshes in the scene. See "Acceleration structures" (page 19).
2. Create a pipeline of programs that contains all programs that will be invoked during a ray tracing launch. See "Program pipeline creation" (page 61).
3. Create a shader binding table that includes references to these programs and their parameters and choose a data layout that matches the implicit shader binding table record selection of the instances and geometries in the acceleration structures. See "Shader binding table" (page 81).
4. Launch a device-side kernel that will invoke a ray generation program with a multitude of threads calling `optixTrace` to begin traversal and the execution of the other programs. See "Ray generation launches" (page 117). Device-side functionality is described in "Device-side functions" (page 129).

Ray tracing work can be interleaved with other CUDA work to generate data, move data to and from the device, and move data to other graphics APIs. It is the application's responsibility to coordinate all work on the GPU. NVIDIA OptiX does not synchronize with any other work.

[2] `https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html#geometry-and-acceleration-structures`

3 Implementation principles

3.1 Error handling

Errors are reported using enumerated return codes. An optional log callback can be registered with the device context to receive any additional logging information.

Functions that compile can optionally take a string character buffer to report additional messaging for errors, warnings and resource use.

3.2 Thread safety

Almost all host functions are thread-safe. Exceptions to this rule are identified in the API documentation. A general requirement for thread-safety is that output buffers and any temporary or state buffers are unique. For example, you can create more than one acceleration structure concurrently from the same input geometry, as long as the temporary and output device memory are disjoint. Temporary and state buffers are always part of the parameter list if they are needed to execute the method.

3.3 Stateless model

Given the same input, the same output should be generated. GPU state is not held by NVIDIA OptiX internally.

In NVIDIA OptiX functions, a CUstream is associated with the CUcontext used to create the `OptixDeviceContext`. Some API functions take a CUstream as an argument. These functions incur work on the device and require that the CUcontext associated with the `OptixDeviceContext` is the current context when they are called. Applications can expect the CUcontext to remain the same after invoking NVIDIA OptiX functions.

3.4 Asynchronous execution

Work performed on the device is issued on an application-supplied CUstream using asynchronous CUDA methods. The host function blocks execution until all work has been issued on the stream, but does not do any synchronization or blocking on the stream itself.

3.5 Opaque types

The API employs several opaque types, such as `OptixModule` and `OptixPipeline`. Such values should be treated like pointers, insofar as copying these does not create new objects.

3.6 Function table and entry function

The NVIDIA OptiX library uses a function table approach to assist in the introduction of new features in future releases while maintaining backward compatibility. To that end, it defines a struct `OptixFunctionTable` that holds pointers to all functions of the host API for a particular version. The current version is specified in the `OPTIX_ABI_VERSION` macro definition.

Listing 3.1

```
struct OptixFunctionTable
{
    OptixResult( *optixDeviceContextCreate )(
    ...             Parameter list details omitted
    );
    ...             Struct members for other host API functions omitted
};
```

The NVIDIA OptiX driver library exports the symbol `optixQueryFunctionTable`. This function is used to obtain pointers to the actual API functions:

Listing 3.2

```
OptixQueryFunctionTable_t* optixQueryFunctionTable;

...             OS-specific code to load the library and to assign the address of the function
                named "optixQueryFunctionTable" to optixQueryFunctionTable omitted

OptixFunctionTable optixFunctionTable = {};
OptixResult result = optixQueryFunctionTable(
    OPTIX_ABI_VERSION, 0, 0, 0, &optixFunctionTable,
    sizeof( OptixFunctionTable ) );
...             Error check omitted
```

Note that the three “0 ”arguments in the example above allow for future extensions of the entry function without changing its signature and are currently unused. A complete example implementation of this functionality including code specific to the operating system is provided as source code in `optixInit`() found in the header file `optix_stubs.h`.

After a successful call to `optixQueryFunctionTable`, the function table can be used as follows, for example, for context creation:

Listing 3.3

```
CUcontext fromContext = nullptr;
...             fromContext initialization omitted

OptixDeviceContextOptions options = {};
...             options initialization omitted

OptixResult result = optixFunctionTable.optixDeviceContextCreate(
   fromContext, &options, &context );

...             Error check omitted
```

Since the explicit call qualifications with the function table instance can be inconvenient, optional stubs that wrap the addresses in the function table into C functions are provided. These stubs are made available by including the header file `optix_stubs.h`. With these stubs the previous example can be simplified as follows:

Listing 3.4

```
CUcontext fromContext = nullptr;
...          fromContext initialization omitted

OptixDeviceContextOptions options = {};
...          options initialization omitted

OptixResult result = optixDeviceContextCreate(
    fromContext, &options, &context );

...          Error check omitted
```

Using these stubs is purely optional and applications are free to implement their own solution to make the addresses in the function table more easily available.

4 Context

The API functions described in this section are:

```
optixDeviceContextCreate
optixDeviceContextDestroy
optixDeviceContextGetProperty
optixDeviceContextSetLogCallback
optixDeviceContextSetCacheEnabled
optixDeviceContextSetCacheLocation
optixDeviceContextSetCacheDatabaseSizes
optixDeviceContextGetCacheEnabled
optixDeviceContextGetCacheLocation
optixDeviceContextGetCacheDatabaseSizes
```

A *context* is created by `optixDeviceContextCreate` and is used to manage a single GPU. The NVIDIA OptiX device context is created by specifying the CUDA context associated with the device. For convenience, zero can be passed and NVIDIA OptiX will use the current CUDA context.

Listing 4.1

```
OptixDeviceContext context = nullptr;
cudaFree( 0 );  Initialize CUDA for this device on this thread

CUcontext cuCtx = 0;  Zero means take the current context
optixDeviceContextCreate( cuCtx, 0, &context );
```

Additional creation time options can also be specified with `OptixDeviceContextOptions`, including parameters for specifying a callback function, log and data. (See "Sending messages with a callback function" (page 16).)

A small set of context properties exist for determining sizes and limits. These are queried using `optixDeviceContextGetProperty`. Such properties include maximum trace depth, maximum traversable graph depth, maximum primitives per build input, and maximum number of instances per acceleration structure.

The context may retain ownership of any GPU resources necessary to launch the ray tracing kernels. Some API objects will retain host memory. These are defined with create/destroy patterns in the API. The application must invoke `optixDeviceContextDestroy` to clean up any host or device resources associated with the context. If any other API objects associated with this context still exist when the context is destroyed, they are also destroyed.

A context can hold a decryption key. When specified, the context requires user code passed into the API to be encrypted using the appropriate session key. This minimizes exposure of the input code to security attacks.

Note: The context decryption feature is available upon request from NVIDIA.

An application may combine any mixture of supported GPUs as long as the data transfer and synchronization is handled appropriately. Some applications may choose to simplify multi-GPU handling by restricting the variety of these blends, for example, by mixing only GPUs of the same streaming multiprocessor version to simplify data sharing.

4.1 Sending messages with a callback function

A log callback and pointer to host memory can also be specified during context creation or later by using `optixDeviceContextSetLogCallback`. This callback will be used to communicate various messages. It must be thread-safe if multiple NVIDIA OptiX functions are called concurrently.

This callback must be a pointer to a function of the following type:

Listing 4.2

```
typedef void( *OptixLogCallback )(
    unsigned int level,
    const char* tag,
    const char* message,
    void* cbdata );
```

The log level indicates the severity of the message. The tag is a terse message category description (for example, `SCENE STAT`). The message is a null-terminated log message (without a newline character at the end) and the value of `cbdata`, the pointer provided when setting the callback function.

The following log levels are supported:

`disable`
Disables all messages. The callback function is not called in this case.

`fatal`
A non-recoverable error. The context, as well as NVIDIA OptiX itself, may no longer be in a usable state.

`error`
A recoverable error, for example, when passing invalid call parameters.

`warning`
Hints that the API might not behave exactly as expected by the application or that it may perform slower than expected.

`print`
Status and progress messages.

4.2 Compilation caching

Compilation of input programs will be cached to disk when creating `OptixModule`, `OptixProgramGroup`, and `OptixPipeline` objects if caching has been enabled. Subsequent compilation can reuse the cached data to improve the time to create these objects. The cache can be shared between multiple `OptixDeviceContext` objects, and NVIDIA OptiX will take care of ensuring correct multi-threaded access to the cache. If no sharing between `OptixDeviceContext` objects is desired, the path to the cache can be set differently for each `OptixDeviceContext`. Caching can be disabled entirely by setting the environment variable `OPTIX_CACHE_MAXSIZE` to 0. Disabling the cache via the environment variable will not affect existing cache files or their contents.

The disk cache can be controlled with:

`optixDeviceContextSetCacheEnabled(..., int enabled)`

When `enabled` has a value of 1, the disk cache is enabled; a value of 0 disables it. Note that no in-memory cache is used when caching is disabled.

The cache database is initialized when the device context is created and when enabled through this function call. If the database cannot be initialized when the device context is created, caching will be disabled; a message is reported to the log callback if caching is enabled. In this case, the call to `optixDeviceContextCreate` does not return an error. To ensure that cache initialization succeeded on context creation, the status can be queried using `optixDeviceContextGetCacheEnabled`. If caching is disabled, the cache can be reconfigured and then enabled using `optixDeviceContextSetCacheEnabled`. If the cache database cannot be initialized with `optixDeviceContextSetCacheEnabled`, an error is returned. Garbage collection is performed on the next write to the cache database, not when the cache is enabled.

`optixDeviceContextSetCacheLocation(..., const char* location)`

The disk cache is created in the directory specified by `location`. The value of `location` must be a NULL-terminated string. The directory is created if it does not exist.

The cache database is created immediately if the cache is currently enabled. Otherwise the cache database is created later when the cache is enabled. An error is returned if it is not possible to create the cache database file at the specified location for any reason (for example, if the path is invalid or if the directory is not writable) and caching will be disabled. If the disk cache is located on a network file share, behavior is undefined.

The location of the disk cache can be overridden with the environment variable `OPTIX_CACHE_PATH`. This environment variable takes precedence over the value passed to this function when the disk cache is enabled.

The default location of the cache depends on the operating system:

Operating system	*Pathname*
Windows	`%LOCALAPPDATA%\NVIDIA\OptixCache`
Linux	`/var/tmp/OptixCache_`*username*, or `/tmp/OptixCache_`*username* if the first choice is not usable. The underscore and username suffix are omitted if the username cannot be obtained.

```
optixDeviceContextSetCacheDatabaseSizes(
  ..., size_t lowWaterMark, size_t highWaterMark)
```

Parameters `lowWaterMark` and `highWaterMark` set the low and high water marks for disk cache garbage collection. Setting either limit to zero disables garbage collection. Garbage collection only happens when the cache database is written. It is triggered whenever the cache data size exceeds the high water mark and proceeding until the size reaches the low water mark. Garbage collection always frees enough space to allow the insertion of the new entry within the boundary of the low water mark. An error is returned if either limit is nonzero and the high water mark is lower than the low water mark. If more than one device context accesses the same cache database with different high and low water mark values, the device context uses its values when writing to the cache database.

The high water mark can be overridden with the environment variable `OPTIX_CACHE_MAXSIZE`. Setting `OPTIX_CACHE_MAXSIZE` to 0 will disable the cache. Negative and non-integer values will be ignored.

The value of `OPTIX_CACHE_MAXSIZE` takes precedence over the `highWaterMark` value passed to this function. The low water mark will be set to half the value of `OPTIX_CACHE_MAXSIZE`.

Corresponding `get*` functions are supplied to retrieve the current value of these cache properties.

4.3 Validation mode

The NVIDIA OptiX *validation mode* can help uncover errors which might otherwise go undetected or which occur only intermittently and are difficult to locate. Validation mode enables additional tests and settings during application execution. This additional processing can reduce performance, so it should only be used during debugging or in the final testing phase of a completed application.

Validation mode is enabled by setting a field in the `OptixDeviceContextOptions` struct:

Listing 4.3

```
OptixDeviceContextOptions options = {};
options.validationMode = OPTIX_DEVICE_CONTEXT_VALIDATION_MODE_ALL;
```

The error `OPTIX_ERROR_VALIDATION_FAILURE` is signaled if an error is caught when validation mode is enabled. Function `optixLaunch` will synchronize after the launch and report errors, if any.

Among other effects, validation mode implicitly enables all OptiX debug exceptions and provides an exception program if none is provided. The first non-user exception caught inside an exception program will therefore be reported and the launch terminated immediately. This will make exceptions more visible that otherwise might be overlooked.

5 Acceleration structures

The API functions described in this section are:

```
optixAccelComputeMemoryUsage
optixAccelBuild
optixAccelRelocate
optixConvertPointerToTraversableHandle
optixOpacityMicromapArrayComputeMemoryUsage
optixOpacityMicromapArrayBuild
optixOpacityMicromapArrayRelocate
optixClusterAccelComputeMemoryUsage
optixClusterAccelBuild
```

NVIDIA OptiX provides *acceleration structures* to optimize the search for the intersection of rays with the geometric data in the scene. Acceleration structures can contain two types of data: geometric primitives (a *geometry-AS*) or instances (an *instance-AS*). Acceleration structures are created on the device using a set of functions. These functions enable overlapping and pipelining of acceleration structure creation, called a *build*. The functions use one or more `OptixBuildInput` or `OptixClusterAccelBuildInput` structs to specify the geometry plus a set of parameters to control the build.

In OptiX 9.0, two specialized acceleration structures were introduced: the *Cluster Acceleration Structure (CLAS)* and *Cluster GAS*; these are built using `optixClusterAccelBuild` rather than `optixAccelBuild`, and are described in "Clusters" (page 49)

Acceleration structures have size limits, listed in "Limits" (page 127). For an instance acceleration structure, the number of instances has an upper limit. For a geometry acceleration structure, the number of geometric primitives is limited, specifically the total number of primitives in its build inputs, multiplied by the number of motion keys.

The following build input types are supported for an instance-AS and geometry-AS:

Instance acceleration structures

`OPTIX_BUILD_INPUT_TYPE_INSTANCES`
`OPTIX_BUILD_INPUT_TYPE_INSTANCE_POINTERS`

A geometry acceleration structure containing built-in triangles

`OPTIX_BUILD_INPUT_TYPE_TRIANGLES`

A geometry acceleration structure containing built-in curve primitives

`OPTIX_BUILD_INPUT_TYPE_CURVES`

A geometry acceleration structure containing built-in spheres
`OPTIX_BUILD_INPUT_TYPE_SPHERES`

A geometry acceleration structure containing custom primitives
`OPTIX_BUILD_INPUT_TYPE_CUSTOM_PRIMITIVES`

A geometry acceleration structure containing clusters of triangles
see the Clusters section for details

For geometry-AS builds using `optixAccelBuild`, each build input can specify a set of triangles, a set of curves, a set of spheres, or a set of user-defined primitives bounded by specified axis-aligned bounding boxes. Multiple build inputs can be passed as an array to `optixAccelBuild` to combine different meshes into a single acceleration structure. All build inputs for a single build must agree on the build input type.

Instance acceleration structures have a single build input and specify an array of instances. Each instance includes a ray transformation and an `OptixTraversableHandle` that refers to a geometry-AS, a transform node, or another instance acceleration structure.

To prepare for a build, the required memory sizes are queried by passing an initial set of build inputs and parameters to `optixAccelComputeMemoryUsage`. It returns three different sizes:

`outputSizeInBytes`
Size of the memory region where the resulting acceleration structure is placed. This size is an upper bound and may be substantially larger than the final acceleration structure. (See "Compacting acceleration structures" (page 32).)

`tempSizeInBytes`
Size of the memory region that is temporarily used during the build.

`tempUpdateSizeInBytes`
Size of the memory region that is temporarily required to update the acceleration structure.

Using these sizes, the application allocates memory for the output and temporary memory buffers on the device. The pointers to these buffers must be aligned to a 128-byte boundary. These buffers are actively used for the duration of the build. For this reason, they cannot be shared with other currently active build requests.

Note that `optixAccelComputeMemoryUsage` does not initiate any activity on the device; pointers to device memory or contents of input buffers are not required to point to allocated memory.

The application can store custom per-acceleration-structure data (e.g. geometric normals) by allocating a larger output buffer and storing the data before the acceleration structure. Device programs can retrieve the application data by calling `optixGetGASPointerFromHandle` and subtracting a fixed offset.

The function `optixAccelBuild` takes the same array of `OptixBuildInput` structs as `optixAccelComputeMemoryUsage` and builds a single acceleration structure from these inputs. This acceleration structure can contain either geometry or instances, depending on the inputs to the build.

The build operation is executed on the device in the specified CUDA stream and runs asynchronously on the device, similar to CUDA kernel launches. The application may choose to block the host-side thread or synchronize with other CUDA streams by using available CUDA synchronization functionality such as cudaStreamSynchronize or CUDA events. The traversable handle returned is computed on the host and is returned from the function immediately, without waiting for the build to finish. By producing handles at acceleration time, custom handles can also be generated based on input to the builder.

The acceleration structure constructed by `optixAccelBuild` does not reference any of the device buffers referenced in the build inputs. All relevant data is copied from these buffers into the acceleration output buffer, possibly in a different format.

The application is free to release this memory after the build without invalidating the acceleration structure. However, instance-AS builds will continue to refer to other instance-AS and geometry-AS instances and transform nodes.

The following example uses this sequence to build a single acceleration structure:

Listing 5.1

```
OptixAccelBuildOptions accelOptions = {};
OptixBuildInput buildInputs[2];

CUdeviceptr tempBuffer, outputBuffer;
size_t tempBufferSizeInBytes, outputBufferSizeInBytes;

memset( accelOptions, 0, sizeof( OptixAccelBuildOptions ) );
accelOptions.buildFlags = OPTIX_BUILD_FLAG_NONE;
accelOptions.operation = OPTIX_BUILD_OPERATION_BUILD;
accelOptions.motionOptions.numKeys = 0;
```

A `numKeys` *value of zero specifies no motion blur*

```
memset( buildInputs, 0, sizeof( OptixBuildInput ) * 2 );
```

Initialize `buildInputs` *memory to 0*

```
...
```

Setup build inputs; see below.

```
OptixAccelBufferSizes bufferSizes = {};
optixAccelComputeMemoryUsage( optixContext, &accelOptions,
   buildInputs, 2, &bufferSizes );

void* d_output;
void* d_temp;

cudaMalloc( &d_output, bufferSizes.outputSizeInBytes );
cudaMalloc( &d_temp, bufferSizes.tempSizeInBytes );

OptixTraversableHandle outputHandle = 0;
OptixResult results = optixAccelBuild( optixContext, cuStream,
    &accelOptions, buildInputs, 2, d_temp,
    bufferSizes.tempSizeInBytes, d_output,
    bufferSizes.outputSizeInBytes, &outputHandle, nullptr, 0 );
```

To ensure compatibility with future versions, the `OptixBuildInput` structure should be initialized with zeros before populating it with specific build inputs.

5.1 Primitive build inputs

A triangle build input references an array of triangle vertex buffers in device memory, one buffer per motion key (a single triangle vertex buffer if there is no motion). (See "Motion blur" (page 35).) Optionally, triangles can be indexed using an index buffer in device memory. Various vertex and index formats are supported as input, but may be transformed to an internal format (potentially of a different size than the input) that is more efficient.

For example:

Listing 5.2

```
OptixBuildInputTriangleArray& buildInput =
    buildInputs[0].triangleArray;
buildInput.type = OPTIX_BUILD_INPUT_TYPE_TRIANGLES;
buildInput.vertexBuffers = &d_vertexBuffer;
buildInput.numVertices = numVertices;
buildInput.vertexFormat = OPTIX_VERTEX_FORMAT_FLOAT3;
buildInput.vertexStrideInBytes = sizeof( float3 );
buildInput.indexBuffer = d_indexBuffer;
buildInput.numIndexTriplets = numTriangles;
buildInput.indexFormat = OPTIX_INDICES_FORMAT_UNSIGNED_INT3;
buildInput.indexStrideInBytes = sizeof( int3 );
buildInput.preTransform = 0;
```

The `preTransform` is an optional pointer to a 3x4 row-major transform matrix in device memory. The pointer needs to be aligned to 16 bytes; the matrix contains 12 floats. If specified, the transformation is applied to all vertices at build time with no runtime traversal overhead.

A curves build input or a spheres build input is similar to a triangle build input; see "Curve build inputs" (page 25), "Sphere build inputs" (page 26).

The acceleration structure build input for custom primitives uses the type `OptixBuildInputCustomPrimitiveArray`. Each custom primitive is represented by an *axis-aligned bounding box* (AABB), which is a rectangular solid defined by ranges of x, y, and z values, and which must completely enclose the primitive. The memory layout of an AABB is defined in the struct `OptixAabb`. The AABBs are organized in an array of buffers in device memory, with one buffer per motion key. The precise shape of each custom primitive will be depend on the intersection program in its SBT record.

Listing 5.3

```
OptixBuildInputCustomPrimitiveArray& buildInput =
    buildInputs[0].customPrimitiveArray;
buildInput.type = OPTIX_BUILD_INPUT_TYPE_CUSTOM_PRIMITIVES;
buildInput.aabbBuffers = d_aabbBuffer;
buildInput.numPrimitives = numPrimitives;
```

The `optixAccelBuild` function accepts multiple build inputs per call, but they must be all triangle inputs, all curve inputs, all sphere inputs, or all AABB inputs. Mixing build input types in a single geometry-AS is not allowed.

Each build input maps to one or more consecutive records in the shader binding table (SBT), which controls program dispatch. (See "Shader binding table" (page 81).) If multiple records in the SBT are required, the application needs to provide a device buffer with per-primitive SBT record indices for that build input. If only a single SBT record is requested, all primitives reference this same unique SBT record. Note that there is a limit to the number of referenced SBT records per geometry-AS. (Limits are discussed in "Limits" (page 127).)

For example:

Listing 5.4

```
buildInput.numSbtRecords = 2;
buildInput.sbtIndexOffsetBuffer =
    d_sbtIndexOffsetBuffer;
```

Values must be in range [0,1] for two SBT records

```
buildInput.sbtIndexOffsetSizeInBytes = sizeof( int );
```

1-4 byte unsigned integer offsets allowed

```
buildInput.sbtIndexOffsetStrideInBytes = sizeof( int );
```

Each build input also specifies an array of `OptixGeometryFlags`, one for each SBT record. The flags for one record apply to all primitives mapped to this SBT record.

For example:

Listing 5.5

```
unsigned int flagsPerSBTRecord[2];
flagsPerSBTRecord[0] = OPTIX_GEOMETRY_FLAG_NONE;
flagsPerSBTRecord[1] = OPTIX_GEOMETRY_FLAG_DISABLE_ANYHIT;
...
buildInput.flags = flagsPerSBTRecord;
```

The following flags are supported:

`OPTIX_GEOMETRY_FLAG_NONE`

Applies the default behavior when calling the any-hit program, possibly multiple times, allowing the acceleration-structure builder to apply all optimizations.

`OPTIX_GEOMETRY_FLAG_REQUIRE_SINGLE_ANYHIT_CALL`

Disables some optimizations specific to acceleration-structure builders. By default, traversal may call the any-hit program more than once for each intersected primitive. Setting the flag ensures that the any-hit program is called only once for a hit with a primitive. However, setting this flag may change traversal performance. The usage of this flag may be required for correctness of some rendering algorithms; for example, in cases where opacity or transparency information is accumulated in an any-hit program.

`OPTIX_GEOMETRY_FLAG_DISABLE_ANYHIT`

Indicates that traversal should not call the any-hit program for this primitive even if the corresponding SBT record contains an any-hit program. Setting this flag usually improves performance even if no any-hit program is present in the SBT.

Primitives inside a build input are indexed starting from zero. This primitive index is accessible inside the intersection, any-hit, and closest-hit programs. If the application chooses to offset this index for all primitives in a build input, there is no overhead at runtime. This can be particularly useful when data for consecutive build inputs is stored consecutively in device memory. The `primitiveIndexOffset` value is only used when reporting the intersection primitive.

For example:

Listing 5.6

```
buildInput[0].aabbBuffers = d_aabbBuffer;
buildInput[0].numPrimitives = ...;
buildInput[0].primitiveIndexOffset = 0;

buildInput[1].aabbBuffers = d_aabbBuffer +
    buildInput[0].numPrimitives * sizeof( float ) * 6;
buildInput[1].numPrimitives = ...;
buildInput[1].primitiveIndexOffset = buildInput[0].numPrimitives;
```

5.2 Curve build inputs

In addition to triangles and custom primitives, NVIDIA OptiX supports curves and spheres as geometric primitives. Curves are used to represent long thin strands, such as for hair, fur, and carpet fibers. Another variant of curves, ribbons (oriented curves), can be used for blades of grass and similar applications. Scenes may contain thousands or millions of curves, and they will often be no wider than a couple of pixels in the final image.

Each curve is a swept surface defined by a three-dimensional series of vertices, called *control points*, and a possibly varying radius. The NVIDIA OptiX API provides curves with these characteristics:

- Curve geometry is defined by a cubic uniform B-spline curve, a quadratic uniform B-spline curve, a Catmull-Rom spline curve, a Bézier curve, or a series of linear segments.
- The cross-section of the curve primitive is a circle.
- For quadratic B-spline curves the cross-section can also be a straight line segment which allows to represent flat oriented curves, called ribbons. These are ruled surfaces which are formed by sweeping a moving straight line along the curve axis.
- A radius is specified at each control point. The radius is interpolated along the curve using the same spline basis as position.
- For ribbons, normals can be specified but are not required.
- Linear curves have spherical end caps, with spherical "elbows" for smooth joints between segments. By default, the ends of cubic and quadratic splines are open and do not have end caps. Flat end caps for cubic and quadratic splines can be enabled by setting `OptixBuildInputCurveArray::endcapFlags` and `OptixBuiltinISOptions::curveEndcapFlags` to `OPTIX_CURVE_ENDCAP_ON`.

Spline curves are composed of a series of polynomial segments. Each segment is defined by two, three, or four control points, depending on the curve type:

Curve type	*Control points per segment*
Piecewise linear	2
Quadratic	3
Cubic	4
Catmull-Rom	4
Bézier	4

NVIDIA OptiX considers each polynomial segment to be a primitive, with its own primitive ID.

A curve build input (`OptixBuildInputCurveArray`) references an array of vertex buffers in device memory, one buffer per motion key (a single vertex buffer if there is no motion). (See "Motion blur" (page 35).) Parallel to this, there is an array of radius buffers in device memory, one buffer per motion key, providing a radius value at each control vertex at each motion key. There is also a (required) index buffer in device memory. Ribbons can also reference optional normal buffers that are parallel to the vertex buffers.

The B-spline control points of each curve strand will appear sequentially in the vertex buffer. The index array contains one index per segment, namely, the index of the segment's first control point. For example, a cubic curve with three segments will have six vertices. The index array might contain `{10, 11, 12}`, in which case the 3 segments will have control points: `{v[10], v[11], v[12], v[13]}`, `{v[11], v[12], v[13], v[14]}` and `{v[12], v[13], v[14], v[15]}`.

The vertex buffers for ribbons store quadratic B-spline control points whereas the normal buffers contain the normals at borders of the ribbon segments. A ribbon strand with three segments will store five control points. If normals are specified, four normals will be required for three segments. They are linearly interpolated along the curve segments. For example, the ribbon strand might have indices `{0, 1, 2}`. In this case, the control points of the segments would be `{v[0], v[1], v[2]}`, `{v[1], v[2], v[3]}` and `{v[2], v[3], v[4]}`, the normals `{n[0], n[1]}`, `{n[1], n[2]}` and `{n[2], n[3]}`. The segment with index `i` will use control points `{v[i], v[i+1], v[i+2]}` and normals `{n[i], n[i+1]}`. Note that there is one more control point than normals in the ribbon strand. Since the indices are used for addressing both vertices and normals, the normals need to be padded with an unused vector at the end of the strand.

End caps appear at the ends of strands. NVIDIA OptiX detects the strands by checking the overlap of segment control points. Within a B-spline strand, adjacent segments overlap all but one of their control points. In other words, unless `indexArray[N+1]` is equal to `indexArray[N]+1`, segment N is the end of one strand and segment N+1 is the beginning of another.

See also "Differences between curves, spheres, and triangles" (page 109) .

5.3 Sphere build inputs

Similar to curves, NVIDIA OptiX supports spheres as geometric primitives. Spheres can be used in different applications to represent, for example, molecules, spray, or smoke.

Each sphere is defined by a three-dimensional center point and a radius.

A sphere build input (OptixBuildInputSphereArray) references an array of vertex buffers in device memory storing the center points, one buffer per motion key (a single vertex buffer if there is no motion). (See "Motion blur" (page 35).) Parallel to this, there is an array of radius buffers in device memory, one buffer per motion key, providing a radius value at each vertex at each motion key. If all spheres have the same radius per motion key, a single radius per radius buffer is sufficient if the singleRadius flag is set.

See also "Differences between curves, spheres, and triangles" (page 109) .

5.4 Instance build inputs

An instance build input specifies a buffer of OptixInstance structs in device memory. These structs can be specified as an array of consecutive structs or an array of pointers to those structs. Each instance description references:

- A child traversable handle
- A static 3x4 row-major object-to-world matrix transform
- An instance ID (user defined)
- An SBT offset
- A visibility mask
- Instance flags

Unlike the triangle and AABB inputs, optixAccelBuild only accepts a single instance build input per build call. There are upper limits to the possible number of instances (the size of the buffer of the OptixInstance structs), the SBT offset, the visibility mask, as well as the user-supplied instance ID. (These limits are discussed in "Limits" (page 127).)

An example of this sequence:

Listing 5.7

```
OptixInstance instance = {};
float transform[12] = {1,0,0,3,0,1,0,0,0,0,1,0};
memcpy( instance.transform, transform, sizeof( float )*12 );
instance.instanceId = 0;
instance.visibilityMask = 255;
instance.sbtOffset = 0;
instance.flags = OPTIX_INSTANCE_FLAG_NONE;
instance.traversableHandle = gasTraversable;

void* d_instance;
cudaMalloc( &d_instance, sizeof( OptixInstance ) );
cudaMemcpy( d_instance, &instance,
    sizeof( OptixInstance ), cudaMemcpyHostToDevice );

OptixBuildInputInstanceArray* buildInput =
    &buildInputs[0].instanceArray;
```

```
buildInput->type = OPTIX_BUILD_INPUT_TYPE_INSTANCES;
buildInput->instances = d_instance;
buildInput->numInstances = 1;
```

The `OPTIX_BUILD_INPUT_TYPE_INSTANCE_POINTERS` build input is a variation on the `OPTIX_BUILD_INPUT_TYPE_INSTANCES` build input where `instanceDescs` references a device memory array of pointers to `OptixInstance` data structures in device memory.

Instance flags are applied to primitives encountered while traversing the geometry-AS connected to an instance. The flags override any instance flags set during the traversal of parent instance-ASs.

`OPTIX_INSTANCE_FLAG_DISABLE_TRIANGLE_FACE_CULLING`
: Disables face culling for triangles. Overrides any culling ray flag passed to `optixTrace`.

`OPTIX_INSTANCE_FLAG_FLIP_TRIANGLE_FACING`
: Flips the triangle orientation during intersection. Also affects any culling of front and back faces.

`OPTIX_INSTANCE_FLAG_DISABLE_ANYHIT`
: Disables any-hit calls for primitive intersections. Can be overridden by ray flags.

`OPTIX_INSTANCE_FLAG_ENFORCE_ANYHIT`
: Forces any-hit calls for primitive intersections. Can be overridden by ray flags.

The visibility mask is combined with the ray mask to determine visibility for this instance. If the condition `rayMask & instance.mask == 0` is true, the instance is culled. The visibility flags may be interpreted as assigning rays and instances to one of eight groups. Instances are traversed only when the instance and ray have at least one group in common. (See "Trace" (page 134).)

The sbtOffset is an offset into the SBT for hit groups (intersection, any-hit, closest-hit) specified with the `hitgroupRecordBase` parameter of `OptixShaderBindingTable`. It is used as a simple additive offset into the SBT to select the hit group programs run in case of an intersection of a primitive part of this instance. See "Acceleration structures" (page 83) for more detail. If the child of the instance is a transform object — an `OptixStaticTransform`, `OptixMatrixMotionTransform`, or `OptixSRTMotionTransform` traversable object instead of a geometry-AS — the instance's sbtOffset value still applies when hitting a primitive of the geometry-AS at the end of the chain of transforms. The maximal supported SBT offset can be queried using `optixDeviceContextGetProperty` with `OPTIX_DEVICE_PROPERTY_LIMIT_MAX_SBT_OFFSET`. In a traversable graph with multiple levels of instance acceleration structure (IAS) objects the offsets are added together. That is, the offset at a GAS is the sum of the offsets of all ancestor instances in the traversable graph. The maximal supported summed SBT offset is equal to the maximum SBT offset for a single instance.

5.5 Build flags

An acceleration structure build can be controlled using the values of the `OptixBuildFlags` enum. To enable random vertex access on an acceleration structure, use `OPTIX_BUILD_FLAG_ALLOW_RANDOM_VERTEX_ACCESS`. (See "Vertex random access" (page **??**).)

To steer trade-offs between build performance, runtime traversal performance and acceleration structure memory usage, use `OPTIX_BUILD_FLAG_PREFER_FAST_TRACE` and `OPTIX_BUILD_FLAG_PREFER_FAST_BUILD`. For curve primitives in particular, these flags control splitting; see "Splitting curve segments" (page 110).

The flags `OPTIX_BUILD_FLAG_PREFER_FAST_TRACE` and `OPTIX_BUILD_FLAG_PREFER_FAST_BUILD` are mutually exclusive. To combine multiple flags that are not mutually exclusive, use the logical "or" operator.

5.6 Dynamic updates

Building an acceleration structure can be computationally costly. Applications may choose to update an existing acceleration structure using modified vertex data or bounding boxes. Updating an existing acceleration structure is generally much faster than rebuilding. However, the quality of the acceleration structure may degrade if the data changes too much with an update, for example, through explosions or other chaotic transitions — even if for only parts of the mesh. The degraded acceleration structure may result in slower traversal performance as compared to an acceleration structure built from scratch from the modified input data.

To allow for future updates of an acceleration structure, set `OPTIX_BUILD_FLAG_ALLOW_UPDATE` in the build flags when building the acceleration structure initially.

For example:

Listing 5.8

```
accelOptions.buildFlags = OPTIX_BUILD_FLAG_ALLOW_UPDATE;
accelOptions.operation  = OPTIX_BUILD_OPERATION_BUILD;
```

To update the previously built acceleration structure, set the operation to `OPTIX_BUILD_OPERATION_UPDATE` and then call `optixAccelBuild` on the same output data. All other options are required to be identical to the original build. The update is done in-place on the output data.

For example:

Listing 5.9

```
accelOptions.buildFlags = OPTIX_BUILD_FLAG_ALLOW_UPDATE;
accelOptions.operation = OPTIX_BUILD_OPERATION_UPDATE;

void* d_tempUpdate;
cudaMalloc( &d_tempUpdate, bufferSizes.tempUpdateSizeInBytes );

optixAccelBuild( optixContext, cuStream, &accelOptions,
   buildInputs, 2, d_tempUpdate,
   bufferSizes.tempUpdateSizeInBytes, d_output,
   bufferSizes.outputSizeInBytes, &outputHandle, nullptr, 0 );
```

Updating an acceleration structure usually requires a different amount of temporary memory than the original build.

When updating an existing acceleration structure, only the device pointers and/or their buffer content may be changed. You cannot change the number of build inputs, the build input types, build flags, traversable handles for instances (for an instance-AS), or the number of vertices, indices, AABBs, instances, SBT records or motion keys. Changes to any of these things may result in undefined behavior, including GPU faults.

Note, however, that in the following two cases it is more efficient to re-build the geometry-AS and/or the instance-AS, or to use the respective masking and flags:

- When using indices, changing the connectivity or, in general, using shuffled vertex positions will work, but the quality of the acceleration structure will likely degrade substantially.
- During an animation operation, geometry that should be invisible to the camera should not be "removed" from the scene, either by moving it very far away or by converting it into a degenerate form. Such changes to the geometry will also degrade the acceleration structure.

Setting the acceleration structure flag `OPTIX_BUILD_FLAG_ALLOW_UPDATE` may also degrade the performance of the acceleration structure when processing curve primitives.

Updating an acceleration structure requires that any other acceleration structure that is using this acceleration structure as a child directly or indirectly also needs to be updated or rebuild.

5.7 Relocation

Geometry acceleration structures can be copied and moved, however they may not be used until `optixAccelRelocate` has been called to update the copied acceleration structure and generate the new traversable handle. Any acceleration structure may be relocated, including compacted acceleration structures.

The copy does not need to be on the original device. This enables the copying of acceleration structure data to compatible devices without rebuilding the acceleration structure.

To relocate an acceleration structure, an `OptixRelocationInfo` object is filled using `optixAccelGetRelocationInfo` and the traversable handle of the source acceleration structure. This object can then be used to determine if relocation to a device (as specified with an `OptixDeviceContext`) is possible. This is done using `optixCheckRelocationCompatibility`. If the target device is compatible, the source acceleration structure may be copied to that device with a subsequent call of `optixAccelRelocate`.

The traversables referenced by an IAS and the OMMs referenced by a triangle GAS may themselves require relocation. The arguments `relocateInputs` and `numRelocateInputs` to `optixAccelRelocate` should be used to specify the relocated traversables and OMMs. After relocation, the relocated acceleration structure will reference these relocated traversables and OMMs instead of their sources. The number of relocate inputs `numRelocateInputs` must match the number of build inputs `numBuildInputs` used to build the source acceleration structure. Relocate inputs correspond with build inputs used to build the source acceleration structure and should appear in the same order (see `optixAccelBuild`). relocateInputs and numRelocateInputs may be zero, preserving any references to traversables and OMMs from the source acceleration structure. An `OptixRelocateInputInstanceArray` specifies a device buffer `OptixRelocateInputInstanceArray::traversableHandles` of handles to relocated traversables, one per instance. `OptixRelocateInputInstanceArray::traversableHandles` may be zero, preserving any references to traversables and OMMs from the source input. Note that the geometric bounds of the acceleration structure are not updated, so `OptixRelocateInputInstanceArray::traversableHandles` should correspond to relocated source traversables. An `OptixRelocateInputTriangleArray` may specify a relocated OMM `OptixRelocateInputTriangleArray::opacityMicromap`. `OptixRelocateInputTriangleArray::opacityMicromap` may be zero, preserving any references to the OMM from the source input. `OptixRelocateInputTriangleArray::numSbtRecords` must equal the corresponding value `OptixBuildInputTriangleArray::numSbtRecords`.

The following example relocates the geometry and instance acceleration structure to new CUDA allocations on the same device:

Listing 5.10

```
OptixRelocationInfo gasInfo = {};
optixAccelGetRelocationInfo( context, gasHandle, &gasInfo );
```

```
int compatible = 0;
optixCheckRelocationCompatibility(
       context, &gasInfo, &compatible );
if( compatible != 1 ) {
       fprintf( stderr,
       "Device isn’t compatible for relocation "
              "of geometry acceleration structures." );
       exit( 2 );
}
```

This is unnecessary because the copy operation’s source and destination are on the same device, but is here to illustrate the API.

```
CUdeviceptr d_relocatedGas = 0;
cudaMalloc( ( void** )&d_relocatedGas,
   gasBufferSizes.outputSizeInBytes );
cudaMemcpy( ( void* )d_relocatedGas,
   ( void* )d_gasOutputBuffer,
   gasBufferSizes.outputSizeInBytes,
   cudaMemcpyDeviceToDevice );
```

Copy and relocate the geometry acceleration structure

```
OptixTraversableHandle relocatedGasHandle = 0;
optixAccelRelocate(
   context, 0,
   &gasInfo,
   0, 0,
   d_relocatedGas,
   gasBufferSizes.outputSizeInBytes,
   &relocatedGasHandle );
```

```
CUdeviceptr d_relocatedIas = 0;
cudaMalloc( ( void** )&d_relocatedIas,
    iasBufferSizes.outputSizeInBytes );
cudaMemcpy( ( void* )d_relocatedIas,
    ( void* )d_iasOutputBuffer,
    iasBufferSizes.outputSizeInBytes,
    cudaMemcpyDeviceToDevice );
```

Copy and relocate the instance acceleration structure

```
OptixRelocationInfo iasInfo = {};
optixAccelGetRelocationInfo(
    context, iasHandle, &iasInfo );
OptixTraversableHandle relocatedIasHandle = 0;

std::vector<OptixTraversableHandle>
    instanceHandles( g_instances.size() );

CUdeviceptr d_instanceTravHandles = 0;
cudaMalloc(
    ( void** )&d_instanceTravHandles,
    sizeof( OptixTraversableHandle ) * instanceHandles.size() );

for( unsigned int i = 0; i < g_instances.size(); ++i )
    instanceHandles[i] = relocatedGasHandle;

cudaMemcpy( ( void* )d_instanceTravHandles, instanceHandles.data(),
    sizeof( OptixTraversableHandle ) * instanceHandles.size(),
    cudaMemcpyHostToDevice );

OptixRelocateInput relocateInput = {};
relocateInput.type = OPTIX_BUILD_INPUT_TYPE_INSTANCES;
relocateInput.instanceArray.numInstances = instanceHandles.size();
relocateInput.instanceArray.traversableHandles = d_instanceTravHandles;

optixAccelRelocate(
    context, 0,
    &iasInfo,
    &relocateInput, 1,
    d_relocatedIas,
   iasBufferSizes.outputSizeInBytes,
    &relocatedIasHandle );
```

5.8 Compacting acceleration structures

A post-process can compact an acceleration structure after construction. This process can significantly reduce memory usage, but it requires an additional pass. The build and compact operations are best performed in batches to ensure that device synchronization does not degrade performance. The compacted size depends on the acceleration structure type and its properties and on the device architecture.

To compact the acceleration structure as a post-process, do the following:

1. Build flag `OPTIX_BUILD_FLAG_ALLOW_COMPACTION` must be set in the `OptixAccelBuildOptions` as passed to `optixAccelBuild`.
2. The emit property `OPTIX_PROPERTY_TYPE_COMPACTED_SIZE` must be set in the `OptixAccelEmitDesc` as passed to `optixAccelBuild`. This property is generated on the device and it must be copied back to the host if it is required for allocating the new output buffer. The application may then choose to compact the acceleration structure using `optixAccelCompact`.
3. The `optixAccelCompact` call should be guarded by an `if( compactedSize < outputSize )` (or similar) to avoid the compacting pass in cases where it is not beneficial. Note that this check requires a copy of the compacted size (as queried by `optixAccelBuild`) from the device memory to host memory.

Just like an uncompacted acceleration structure, it is possible to traverse, update, or relocate a compacted acceleration structure.

For example:

Listing 5.11

```
size_t *d_compactedSize;
OptixAccelEmitDesc property = {};
property.type = OPTIX_PROPERTY_TYPE_COMPACTED_SIZE;
property.result = d_compactedSize;

OptixTraversableHandle accelHandle = 0;
OptixTraversableHandle compactedAccelHandle = 0;

accelOptions.buildFlags = OPTIX_BUILD_FLAG_ALLOW_COMPACTION;

optixAccelBuild( optixContext, cuStream, &accelOptions,
    buildInputs, 2, d_tempUpdate, bufferSizes.tempSizeInBytes,
    d_output, bufferSizes.outputSizeInBytes, &accelHandle,
    &property, 1 );

size_t compactedSize;
cudaMemcpy( &compactedSize, d_compactedSize,
    sizeof( size_t ),
    cudaMemcpyDeviceToHost );

void *d_compactedOutputBuffer;
cudaMalloc( &d_compactedOutputBuffer, compactedSize );

if( compactedSize < bufferSizes.outputSizeInBytes ) {
    optixAccelCompact(
        optixContext, cuStream,
        accelHandle,
        d_compactedOutputBuffer, compactedSize,
        &compactedAccelHandle );
}
```

A compacted acceleration structure does not reference the uncompacted input data. The application is free to reuse the memory of the uncompacted acceleration structure without invalidating the compacted acceleration structure. However, the memory for the compacted acceleration structure must not overlap memory of the uncompacted acceleration structure as the compaction operation does not work in-place.

A compacted acceleration structure supports dynamic updates only if the uncompacted source acceleration structure was built with the `OPTIX_BUILD_FLAG_ALLOW_UPDATE` build flag. (See "Dynamic updates" (page 29).) The amount of temporary memory required for a dynamic update is the same for the uncompacted acceleration structure and compacted acceleration structure. Note that using the following build flags will lead to less memory savings when enabling the compacting post-process:

- `OPTIX_BUILD_FLAG_ALLOW_UPDATE`
- `OPTIX_BUILD_FLAG_PREFER_FAST_BUILD`

5.9 Traversable objects

The instances in an instance-AS may reference transform traversables, as well as geometry-ASs. Transform traversables are fully managed by the application. The application needs to create these traversables manually in device memory in a specific form. The function `optixConvertPointerToTraversableHandle` converts a raw pointer into a traversable handle of the specified type. The traversable handle can then be used to link traversables together.

In device memory, all traversable objects need to be 64-byte aligned. Note that moving a traversable to another location in memory invalidates the traversable handle. The application is responsible for constructing a new traversable handle and updating any other traversables referencing the invalidated traversable handle.

The traversable handle is considered opaque and the application should not rely on any particular mapping of a pointer to the traversable handle.

For example:

Listing 5.12

```
OptixMatrixMotionTransform transform = {};

...          Setup motion transform

cudaMemcpy( d_transform, &transform,
    sizeof( OptixMatrixMotionTransform ),
    cudaMemcpyHostToDevice );
```

```
OptixTraversableHandle transformHandle = 0;
optixConvertPointerToTraversableHandle(
    optixContext, d_transform,
    OPTIX_TRAVERSABLE_TYPE_MATRIX_MOTION_TRANSFORM,
    &transformHandle );

OptixInstance instance = {};
instance.traversableHandle = transformHandle;

...           Setup instance description
```

5.9.1 Traversal of a single geometry acceleration structure

The traversable handle passed to `optixTrace` can be a traversable handle created from a geometry-AS. This can be useful for scenes where single geometry-AS objects represent the root of the scene graph.

If the modules and pipeline only need to support single geometry-AS traversables, it is beneficial to change the `OptixPipelineCompileOptions::traversableGraphFlags` from `OPTIX_TRAVERSABLE_GRAPH_FLAG_ALLOW_ANY` to `OPTIX_TRAVERSABLE_GRAPH_FLAG_ALLOW_SINGLE_GAS`.

This signals to NVIDIA OptiX that no other traversable types require support during traversal.

5.10 Motion blur

Motion support in OptiX targets the rendering of images with motion blur using a stochastic sampling of time. OptiX supports two types of motion as part of the scene: transform motion and vertex motion, often called deformation motion. When setting up the scene traversal graph and building the acceleration structures, motion options can be specified per acceleration structure as well as per motion transform traversable. At run time, a time parameter is passed to the trace call to perform the intersection of a ray against the scene at the selected point in time.

The general design of the motion feature in OptiX tries to strike a balance between providing many parameters to offer a high degree of freedom combined with a simple mapping of scene descriptions to these parameters but also delivering high traversal performance at the same time. As such OptiX supports the following key features:

- Vertex and transformation motion
- Matrix as well as SRT (scale rotation translation) transformations
- Arbitrary time ranges (ranges not limited to [0,1]) and flags to specify behavior outside the time range
- Arbitrary concatenations of transformations (for example, a matrix transformation on top of a SRT transformation)
- Per-ray timestamps

Scene descriptions with motion need to map easily to traversable objects and their motion options as offered by OptiX. As such, the idea is that the motion options are directly derived by the scene description, delivering high traversal performance without the need for any performance-driven adjustments. However, due to the complexity of the subject, there are a few exceptions that are discussed in this section.

This section details the usage of the motion options on the different traversable types and how to map scene options best to avoid potential performance pitfalls.

5.10.1 Basics

Motion is supported by `OptixMatrixMotionTransform`, `OptixSRTMotionTransform` and acceleration structure traversables. The general motion characteristics are specified per traversable as motion options: the number of motion keys, flags, and the beginning and ending motion times corresponding to the first and last key. The remaining motion keys are evenly spaced between the beginning and ending times. The motion keys are the data at specific points in time and the data is interpolated in between neighboring keys. The motion options are specified in the `OptixMotionOptions` struct.

The motion options are always specified per traversable (acceleration structure or motion transform). There is no dependency between the motion options of traversables; given an instance referencing a geometry acceleration structure with motion, it is not required to build an instance acceleration structure with motion. The same goes for motion transforms. Even if an instance references a motion transform as child traversable, the instance acceleration structure itself may or may not have motion.

Motion transforms must specify at least two motion keys with the motion beginning time strictly smaller than the ending time. Acceleration structures, however, also accept `OptixAccelBuildOptions` with field `OptixMotionOptions` set to zero. This effectively disables motion for the acceleration structure and ignores the motion beginning and ending times, along with the motion flags.

OptiX also supports static transform traversables in addition to the static transform of an instance. Static transforms are intended for the case of motion transforms in the scene. Without any motion transforms (`OptixMatrixMotionTransform` or `OptixSRTMotionTransform`) in the traversable graph, any static transformation should be baked into the instance transform. However, if there is a motion transform, it may be required to apply a static transformation on a traversable (for example, on a geometry-AS) first before applying the motion transform. For example, a motion transform may be specified in world coordinates, but the geometry it applies to needs to be placed into the scene first (object-to-world transformation, which is usually done using the instance transform). In this case, a static transform pointing at the geometry acceleration structure can be used for the object-to-world transformation and the instance transform pointing to the motion transform has an identity matrix as transformation.

Motion boundary conditions are specified by using flags. By default, the behavior for any time outside the time range, is as if time was clamped to the range, meaning it appears static and visible. Alternatively, to remove the traversable before the beginning time, set `OPTIX_MOTION_FLAG_START_VANISH`; to remove it after the ending time, set `OPTIX_MOTION_FLAG_END_VANISH`.

For example:

Listing 5.13

```
OptixMotionOptions motionOptions = {};
motionOptions.numKeys = 3;
motionOptions.timeBegin = -1f;
motionOptions.timeEnd = 1.5f;
motionOptions.flags = OPTIX_MOTION_FLAG_NONE;
```

OptiX offers two types of motion transforms, SRTs (scale-rotation-translation) as well as 3x4 affine matrices, each specifying one transform (SRT or matrix) per motion key. The transformations are always specified as object-to-world transformation just like the instance transformation. During traversal OptiX performs a per-component linear interpolation of the two nearest keys. The rotation component (expressed as a quaternion) of the SRT is an exception, OptiX ensures that the interpolated quaternion of two SRTs is of unit length by using nlerp interpolation for performance reasons. This results in a smooth, scale-preserving rotation in Cartesian space though with non-constant velocity.

For vertex motion, OptiX applies a linear interpolation between the vertex data that are provided by the application. If intersection programs are used and AABBs are supplied for the custom primitives, the AABBs are also linearly interpolated for intersection. The AABBs at the motion keys must therefore be big enough to contain any motion path of the underlying custom primitive.

There are several device-side functions that take a time parameter such as `optixTrace` and respect the motion options as set at the traversables. The result of these device-side functions is always that of the specified point in time, e.g., the intersection of the ray with the scene at the selected point in time. Device-side functions are discussed in detail in "Device-side functions" (page 129).

5.10.2 Motion geometry acceleration structure

Use `optixAccelBuild` to build a motion acceleration structure. The motion options are part of the build options (`OptixAccelBuildOptions`) and apply to all build inputs. Build inputs must specify primitive vertex buffers (for `OptixBuildInputTriangleArray`, `OptixBuildInputCurveArray`, and `OptixBuildInputSphereArray`), radius buffers (for `OptixBuildInputCurveArray` and `OptixBuildInputSphereArray`), and AABB buffers (for `OptixBuildInputCustomPrimitiveArray` and `OptixBuildInputInstanceArray`) for all motion keys. These are interpolated during traversal to obtain the continuous motion vertices and AABBs between the begin and end time.

For example:

Listing 5.14

```
CUdeviceptr d_motionVertexBuffers[3];

OptixBuildInputTriangleArray buildInput = {};
```

```
buildInput.vertexBuffers = d_motionVertexBuffers;
buildInput.numVertices = numVertices;
```

The motion options are typically defined by the mesh data which should directly map to the motion options on the geometry acceleration structure. For example, if a triangle mesh has three per-vertex motion values, the geometry acceleration structure needs to have three motion keys. Just as for non-motion meshes, it is possible to combine meshes within a single geometry acceleration structure to potentially increase traversal performance (this is generally recommended if there is only a single instance of each mesh and the meshes overlap or are close together). However, these meshes need to share the same motion options (as they are specified per geometry acceleration structure). The usual trade-offs apply in case meshes need to be updated from one frame to another as in an interactive application. The entire geometry acceleration structure needs to be rebuilt or refitted if the vertices of at least one mesh change.

It is possible to use a custom intersection program to decouple the actual vertex data and the motion options of the geometry acceleration structure. Intersection programs allow any kind of intersection routine. For example, it is possible to implement a three-motion-key-triangle intersection, but build a static geometry acceleration structure over AABBs by passing AABBs to the geometry acceleration structure build that enclose the full motion path of the triangles. However, this is generally not recommended for two reasons: First, the AABBs tend to increase in size very quickly even with very little motion. Second, it prevents the use of hardware intersection routines. Both of these effects can have a tremendous impact on performance.

5.10.3 Motion instance acceleration structure

Just as for a geometry acceleration structure, the motion options for an instance acceleration structure are specified as part of the build options. The notable difference to a geometry acceleration structure is that the motion options for an instance acceleration structure almost only impact performance. Hence, whether or not to build a motion instance acceleration structure has no impact on the correctness of the rendering (determining which instances can be intersected), but impacts memory usage as well as traversal performance. The only exception to that are the vanish flags as these force any instance of the instance acceleration structure to be non-intersectable for any ray time outside of the time range of the instance acceleration structure.

In the following, guidelines are provided on setting the motion options to achieve good performance and avoid pitfalls. We will focus on the number of motion keys, usually the main discriminator for traversal performance and the only factor for memory usage. The optimal number of motion keys used for the instance acceleration structure build depends on the amount and linearity of the motion of the traversables referenced by the instances. The time beginning and ending range are usually defined by what is required to render the current frame. The recommendations given here may change in the future.

The following advice should be considered a simplified heuristic. A more detailed derivation of whether or not to use motion is given below. For RTCores version 1.0 (Turing architecture), do not use motion for instance acceleration structure, but instead build a static instance acceleration structure that can leverage hardware-accelerated traversal. For any other device (devices without RTCores or RTCores version >= 2.0), build a motion instance acceleration structure if any of the instances references a motion transform or a motion acceleration structure as traversable child.

If a motion instance acceleration structure is built, it is often sufficient to use a low number of motion keys (two or three) to avoid high memory costs. Also, it is not required to use a large number of motion keys just because one of the referenced motion transforms has many motion keys (such as the maximum motion keys of any referenced traversable by any of the instances). The motion options have no dependency between traversable objects and a high number of motion keys on the instance acceleration structure causes a high memory overhead. Clearly, motion should not be used for an instance acceleration structure if the instances only reference static traversables.

Further considerations when using motion blur:

Is motion enabled?

An instance acceleration structure should be built with motion on (the number of motion keys larger than one) if the overall amount of motion of the instanced traversables is non-minimal. For a single instance this can be quantified by the amount of change of its AABB over time. Hence, in case of a simple translation (for example, due to a matrix motion transform), the metric is the amount of the translation in comparison to the size of the AABB. In case of a scaling, it is the ratio of the size of the AABB at different points in times. If sufficiently many instanced traversables exhibit a non-minimal amount of change of their AABB over time, build a motion instance acceleration structure. Inversely, a static instance acceleration structure can yield higher traversal performance if many instanced traversables have no motion at all or only very little. The latter can happen for rotations. A rotation around the center of an object causes a rather small difference in the AABB of the object. However, if the rotational pivot point is not the center, it is likely to cause a big difference in the AABB of the object.

As it is typically hard to actually quantify the amount of motion for the instances, switch to motion if sufficiently many instanced traversables have or are expected to have motion. Yet it is difficult to predict when exactly it pays off to use or not use motion on the instance acceleration structure.

If motion is enabled, how many keys should be defined?

A reasonable metric to determine the required number of motion keys for an instance acceleration structure is the linearity of the motion of the instanced traversables. If there are motion transforms with many motion keys, rotations, or a hierarchical set of motion transforms, more motion keys on the instance acceleration structure may increase traversal performance. Transformations like a simple translation, rotation around the center of an object, a small scale, or even all of those together are usually handles well by a two-motion-key instance acceleration structure.

Finally, the quality of the instance acceleration structure is also affected by the number of motion keys of the referenced traversables of the instances. As such, it is desirable to have the motion options of the instance acceleration structure match the motion options of any referenced motion transform. For example, if all instances reference motion transforms with three keys, it is reasonable to also use three motion keys for the instance acceleration structure. Note that also in this case the statement from above still applies that using more motion keys only helps if the underlying transformation results in a non-linear motion.

5.10.4 Motion matrix transform

The motion matrix transform traversable (`OptixMatrixMotionTransform`) transforms the ray during traversal using a motion matrix. The traversable provides a 3x4 row-major object-to-world transformation matrix for each motion key. The final motion matrix is constructed during traversal by interpolating the elements of the matrices at the nearest motion keys.

The `OptixMatrixMotionTransform` struct has a dynamic size, dependent on the number of motion keys. The struct specifies the header and the first two motion keys for convenience; when using more than than two keys, compute the size required for additional keys.

For example:

Listing 5.15

```
#define NUM_MOTION_KEYS 3
float matrixKeys[NUM_MOTION_KEYS][12];
...

size_t transformSizeInBytes = sizeof( OptixMatrixMotionTransform )
    + ( NUM_MOTION_KEYS-2 ) * 12 * sizeof( float );

OptixMatrixMotionTransform *transform =
    ( OptixMatrixMotionTransform* ) malloc( transformSizeInBytes );

transform->motionOptions.numKeys = NUM_MOTION_KEYS;
transform->motionOptions.timeBegin = -1f;
transform->motionOptions.timeEnd = 1.5f;
transform->motionOptions.flags = 0;
memcpy( transform->transform, matrixKeys,
NUM_MOTION_KEYS * 12 * sizeof( float ) );
```

5.10.5 Motion scale/rotate/translate transform

The behavior of the motion transform `OptixSRTMotionTransform` is similar to the matrix motion transform `OptixMatrixMotionTransform`. In `OptixSRTMotionTransform` the object-to-world transforms per motion key are specified as a scale, rotation and translation (SRT) decomposition instead of a single 3x4 matrix. Each motion key is a struct of type `OptixSRTData`, which consists of 16 floats:

Listing 5.16

```
typedef struct OptixSRTData {
   float sx, a, b, pvx, sy, c, pvy, sz, pvz, qx, qy, qz, qw, tx, ty, tz;
} OptixSRTData;
```

These 16 floats define the three components for scaling, rotation and translation whose product is the motion transformation:

- The scaling matrix S

$$S = \begin{bmatrix} sx & a & b & pvx \\ 0 & sy & c & pvy \\ 0 & 0 & sz & pvz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

defines an affine transformation that can include scale, shear, and a translation. The translation enables a pivot point to be defined for the scale, shear as well as the subsequent rotation.

- The quaternion R

  $$R = \begin{bmatrix} qx & qy & qz & qw \end{bmatrix}$$

  describes a rotation with angular component `qw` $= cos(\theta/2)$ and other components qx, qy and qz, where

  $$\begin{bmatrix} qx & qy & qz \end{bmatrix} = sin(\theta/2) * \begin{bmatrix} ax & ay & az \end{bmatrix}$$

  and where the axis $\begin{bmatrix} ax & ay & az \end{bmatrix}$ is normalized.

- The translation T

  $$T = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \end{bmatrix}$$

  defines another translation that is applied after the rotation. Typically, this translation includes the inverse translation from the matrix S to undo the pivot point transformation.

To obtain the effective transformation at time t, the elements of the components of S, R, and T are interpolated linearly and R is normalized afterwards. The components are then multiplied to obtain the combined transformation $C = T \times R \times S$. The transformation C is the effective object-to-world transformations at time t, and C^{-1} is the effective world-to-object transformation at time t.

Example 1 – Rotation about the origin

Use two motion keys. Set the first key to identity values. For the second key, define a quaternion from an axis and angle, for example, a 60-degree rotation about the z axis is given by:

$$Q = \begin{bmatrix} 0 & 0 & sin(\pi/6) & cos(\pi/6) \end{bmatrix}$$

Example 2 – Rotation about a pivot point

Use two motion keys. Set the first key to identity values. Represent the pivot point as a translation P, and define the second key as follows:

$$S' = P^{-1} \times S$$

$$T' = T \times P$$

$$C = T' \times R \times S'$$

Example 3 – Scaling about a pivot point

Use two motion keys. Set the first key to identity values. Represent the pivot as a translation $G = \begin{bmatrix} G_x & G_y & f G_z \end{bmatrix}$ and modify the pivot point described above:

$$P'_x = P_x + (-S_x * G_x + G_x)$$

$$P'_y = P_y + (-S_y * G_y + G_y)$$

$$P'_z = P_z + (-S_z * G_z + G_z)$$

5.10.6 Transforms trade-offs

Several trade-offs must be considered when using transforms.

SRTs compared to matrix motion transforms

Use SRTs for any transformations containing a rotation. Only SRTs produce a smooth rotation without distortion. They also avoid any oversampling of matrix transforms to approximate a rotation. However, note that the maximum angle of rotation due to two neighboring SRT keys needs to be less than 180 degrees, hence, the dot product of the quaternions needs to be positive. This way the rotations are interpolated using the shortest path. If a rotation of 180 degrees or more is required, additional keys need to be specified such that the rotation between two keys is less than 180 degrees. OptiX uses nlerp to interpolate quaternion at runtime. While nlerp produces the best traversal performance, it causes non-constant velocity in the rotation. The variation of rotational velocity is directly dependent on the amount of the rotation. If near constant rotation velocity is required, more SRT keys can be used.

Due to the complexity of the rotation, instance acceleration structure builds with instances that reference SRT transforms can be relatively slow. For real-time or interactive applications, it can be advantageous to use matrix transforms to have fast rebuilds or refits of the instance acceleration structure.

Motion options for motion transforms

The motion options for motion transforms should be derived by the scene setup and used as needed. The number of keys is defined by the number of transformations specified by the scene description. The beginning, ending times should be as needed for the frame or tighter if specified by the scene description.

Avoid duplicating instances of motion transforms to achieve a motion behavior that can also be expressed by a single motion transform but many motion keys. An example is the handling of irregular keys, which is discussed in the following section.

Dealing with irregular keys

OptiX only supports regular time intervals in its motion options. Irregular keys should be resampled to fit regular keys, potentially with a much higher number of keys if needed.

A practical example for this is a motion matrix transform that performs a rotation. Since the matrix elements are linearly interpolated between keys, the rotation is not an actual rotation, but a scale/shear/translation. To avoid visual artifacts, the rotation needs to be sampled with potentially many matrix motion keys. Such a sampling bounds the maximum error in the approximation of the rotation by the linear interpolation of matrices. The sampling should not try to minimize the number of motion keys by outputting irregular motion keys, but rather oversample the rotation with many keys.

Duplicate motion transforms should not be used as a workaround for irregular keys, where each key has varying motion beginning and ending times and vanish motion flags set. This duplication creates traversal overhead as all copies need to be intersected and their motion times compared to the ray's time.

5.11 Opacity micromaps

Very high quality, high-definition opacity (alpha) content is usually very coherent, or locally similar. Typically, there are larger regions that are completely transparent or opaque within meshes that do not necessarily coincide with triangle boundaries. Consequently, any-hit programs are often invoked for ray-triangle hits that could be trivially categorized as either a miss or a hit. To reduce the overhead of redundant and potentially expensive any-hit programs, OptiX *opacity micromaps* (OMMs) can be used to cull any-hit program invocations in regions within a triangle known to be completely opaque or transparent.

The OMM is defined on a sub-triangle detail level, encoded in a uniformly subdivided mesh of 4^N *microtriangles*, laid out on a $2^N \times 2^N$ barycentric grid. Figure 5.1 shows the first few levels of the subdivision scheme.

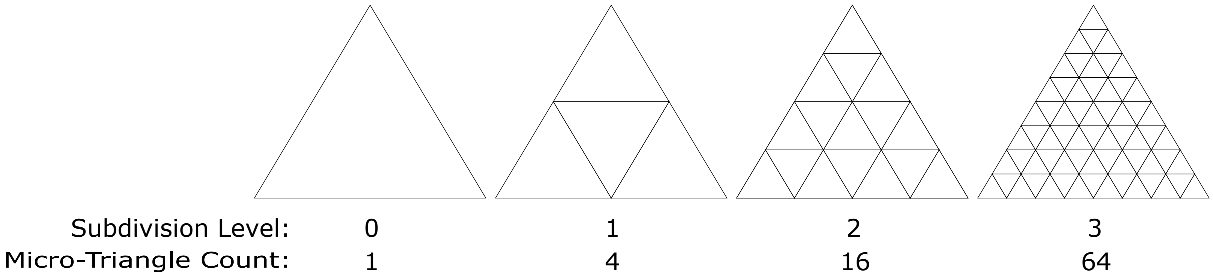

Fig. 5.1 - Microtriangle subdivision

The OMM specifies one of four opacity states per microtriangle: opaque, transparent, unknown-opaque or unknown-transparent. An OMM is applied to one or more base triangles in a GAS to add extra opacity detail, much like traditional texture mapping.

Figure 5.2 shows examples of OMM detail applied to a base triangle.

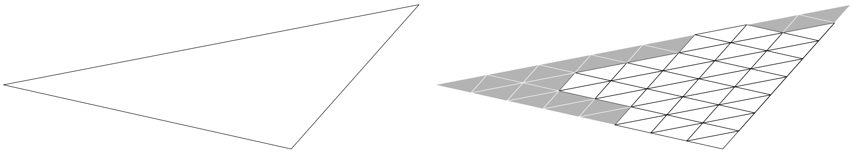

Fig. 5.2 - Opacity micromap detail on base triangle

5.11.1 Opacity micromap arrays

Unlike triangles, the OMMs are not stored directly in the geometry acceleration structures (GAS), but instead reside in a separate resource, an *opacity micromap array*. Individual OMMs may be referenced by triangles from within a GAS. Because OMM storage is separate from the GASes, they can be reused within and across multiple GASes in a scene. Like acceleration structures, OMM arrays are created on the device using the functions `optixOpacityMicromapArrayComputeMemoryUsage` and `optixOpacityMicromapArrayBuild`.

Listing 5.17

```
OptixMicromapBufferSizes bufferSizes = {};
optixOpacityMicromapArrayComputeMemoryUsage(
   optixContext, &buildInput, &bufferSizes );

void* d_micromapArray;
void* d_tmp;

cudaMalloc( &d_micromapArray, bufferSizes.outputSizeInBytes );
cudaMalloc( &d_tmp, bufferSizes.tempSizeInBytes );

OptixMicromapBuffers buffers = {};
buffers.output            = d_micromapArray;
buffers.outputSizeInBytes = bufferSizes.outputSizeInBytes;
buffers.temp              = d_tmp;
buffers.tempSizeInBytes   = bufferSizes.tempSizeInBytes;

OptixResult results = optixOpacityMicromapArrayBuild(
   optixContext, cuStream, &buildInput, &buffers );
```

The functions use a single `OptixOpacityMicromapArrayBuildInput` struct specifying the set of OMMs in the array. The build input specifies a device buffer of raw OMM data and a device buffer with per OMM `OptixOpacityMicromapDesc` structs. The descriptors specify the format and size of each OMM and its location within the raw OMM data buffer. The input also specifies a host buffer of `OptixOpacityMicromapHistogramEntry` structs. This specifies a histogram over opacity micromaps in the build input, binned by input format and subdivision level combinations. Counts of histogram bins with equal format and subdivision combinations are added together. The descriptors in the input buffer don't need to appear in any particular order, as long as the counts match the input histogram.

Listing 5.18

```
OptixOpacityMicromapHistogramEntry histogram[2];
```

Create micromap histogram

```
histogram[0].count            = 2;
histogram[0].subdivisionLevel = 9;
histogram[0].format           =
   OPTIX_OPACITY_MICROMAP_FORMAT_2_STATE;
```

Two 1-bit OMMs of level 9

```
histogram[1].count            = 1;
histogram[1].subdivisionLevel = 8;
histogram[1].format           =
   OPTIX_OPACITY_MICROMAP_FORMAT_4_STATE;
```

One 2-bit OMM of level 8

```
OptixOpacityMicromapDesc hostPerMicromapDescBuffer[2] = {...};

unsigned micromapUsageCounts = histogram[0].count + histogram[1].count;
size_t perMicromapDescBufferSizeInBytes =
   micromapUsageCounts * sizeof( OptixOpacityMicromapDesc );
size_t inputBufferSizeInBytes =
   histogram[0].count *
      (1 << (max( 3,2 * histogram[0].subdivisionLevel ) - 3)) +
   histogram[1].count *
      (1 << (max( 2,2 * histogram[1].subdivisionLevel ) - 2));
```

All OMMs are byte alinged

```
std::vector<OptixOpacityMicromapDesc> h_perMicromapDescBuffer(
   micromapUsageCounts );
std::vector<char> h_inputBuffer( inputBufferSizeInBytes );

...
```

Setup OMM descriptors and input data.

```
void* d_perMicromapDescBuffer;
void* d_inputBuffer;

cudaMalloc( d_perMicromapDescBuffer, perMicromapDescBufferSizeInBytes );
cudaMalloc( d_inputBuffer, inputBufferSizeInBytes );

cudaMemcpy(
   d_perMicromapDescBuffer, h_perMicromapDescBuffer.data(),
   perMicromapDescBufferSizeInBytes, cudaMemcpyHostToDevice );
cudaMemcpy(
   d_inputBuffer, h_inputBuffer.data(),
   inputBufferSizeInBytes, cudaMemcpyHostToDevice );

OptixOpacityMicromapArrayBuildInput buildInput = {};
buildInput.flags                       = OPTIX_OPACITY_MICROMAP_FLAG_NONE;
buildInput.inputBuffer                 = d_inputBuffer;
buildInput.perMicromapDescBuffer       = d_perMicromapDescBuffer;
buildInput.numMicromapHistogramEntries = 2;
buildInput.micromapHistogramEntries    = histogram;
```

The OMM arrays are opaque data structures, but the application is responsible for memory management. The amount of memory required for a OMM array can be queried by passing the build input to `optixOpacityMicromapArrayComputeMemoryUsage`.

The OMM array constructed by `optixOpacityMicromapArrayBuild` does not reference any of the device buffers referenced in the build input. All relevant data is copied from these buffers into the OMM array output buffer, possibly in a different format. The application is free to release the input memory after the build without invalidating the OMM array.

Figure 5.3 shows an example of a collection of OMMs in an OMM array.

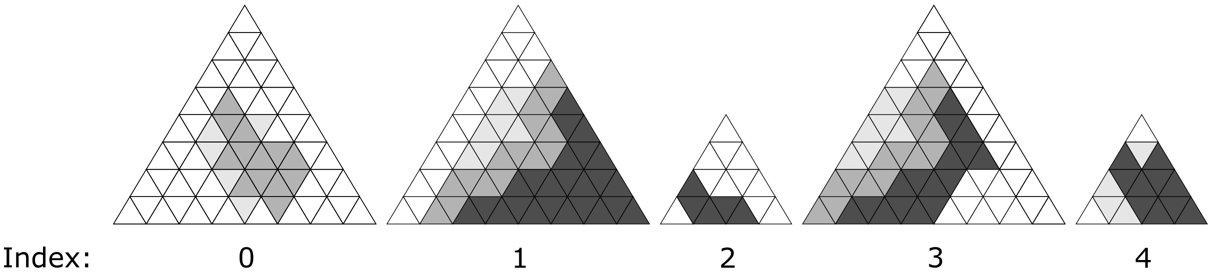

Fig. 5.3 - Collection of opacity micromaps on a base triangle

Like acceleration structures, OMM arrays can be copied freely in memory, but the `optixOpacityMicromapArrayRelocate` function must be called with the target location address before the relocated OMM array is used in any bounding volume hierarchy (BVH). (See "Relocation" (page 30).)

5.11.2 Usage

5.11.2.1 Construction of the geometry acceleration structure

The application can attach one OMM array per triangle geometry input to the GAS build. Figure 5.4 shows the relationship between OMM arrays and the acceleration structure hierarchy.

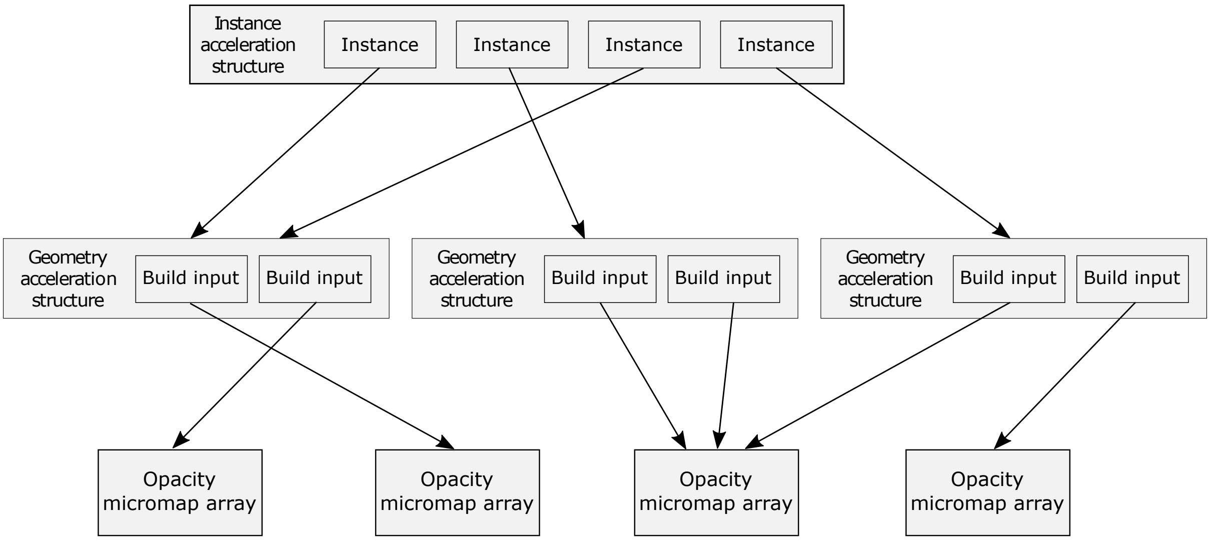

Fig. 5.4 - Opacity micromap arrays and the acceleration structure hierarchy

The OMM array is specified using the `OptixBuildInputOpacityMicromap` struct. OMM can be indexed using an index buffer in device memory, with one OMM index per triangle. Instead of an actual index, predefined indices (with names in the form `OPTIX_OPACITY_MICROMAP_PREDEFINED_INDEX_*`) can be used to indicate that there is no OM for this triangle, but the triangle has a uniform opacity state, and the selected behavior is applied to the entire triangle.

The input also specifies a host buffer of `OptixOpacityMicromapUsageCount` structs. This specifies the usage counts over opacity micromaps in the acceleration structure build input, binned by input format and subdivision level combinations. Counts of bins with equal format and subdivision combinations are added together. Duplicate use of OMMs in the GAS build input must be included in this count, while OMMs that occur in the OMM array but are not referenced by the GAS build input are not to be included in these counts. Note that this buffer differs from the histogram passed to the OMM array build, which only specifies the occurences of OMMs in the OMM array, irrespective of use by GAS build inputs.

Listing 5.19

```
OptixBuildInputOpacityMicromap opacityMicromap = {};

int h_indexBuffer[] = {
   1, OPTIX_OPACITY_MICROMAP_PREDEFINED_INDEX_FULLY_OPAQUE 0, ...
};

cudaMemcpy(
   d_indexBuffer, h_indexBuffer, sizeof( h_indexBuffer ),
   cudaMemcpyHostToDevice );

OptixOpacityMicromapUsageCount count[2] = { ... };

opacityMicromap.indexingMode =
   OPTIX_OPACITY_MICROMAP_ARRAY_INDEXING_MODE_INDEXED;
opacityMicromap.indexBuffer            = d_indexBuffer;
opacityMicromap.opacityMicromapArray   = d_micromapArray;
opacityMicromap.indexSizeInBytes       = 4;
opacityMicromap.numMicromapUsageCounts = 2;
opacityMicromap.micromapUsageCounts    = count;

OptixBuildInputTriangleArray triangleInput = {};
...
triangleInput.opacityMicromap = opacityMicromap;
```

Create micromap histogram

If a GAS has been built with the `OPTIX_BUILD_FLAG_ALLOW_OPACITY_MICROMAP_UPDATE` build flag, it is possible to assign a different OMM array and OMM indices when updating a GAS. GAS builds using OMM arrays as input will continue to refer to these OMM Arrays. If an OMM array is overwritten with new OMM data, any GASes referencing it become stale and must be updated.

5.11.2.2 Traversal

To render any triangles with OMMs, OMMs must be enabled in the pipeline using allowOpacityMicromaps in the `OptixPipelineCompileOptions` struct. If OMMs are known not to be used, it is more efficient to not specify the flag. If the flag is omitted and an OMM is encountered during ray traversal, behavior is undefined.

Listing 5.20

```
OptixPipelineCompileOptions options = {};
...
options.allowOpacityMicromaps = true;
```

When a ray intersects a triangle with an OMM attached, the intersection point within the barycentric space of the triangle is used to look up the opacity at that location through the OMM. OMMs classify microtriangles as *opaque*, *transparent*, or *unknown*.

Opaque The hit is treated as a hit against a geometry with the `OPTIX_GEOMETRY_FLAG_DISABLE_ANYHIT` flag set.

Transparent The hit is ignored and traversal resumes.

Unknown The hit is treated as a hit against a geometry without the `OPTIX_GEOMETRY_FLAG_DISABLE_ANYHIT` flag set.

The two microtriangles states unknown-opaque and unknown-transparent are both treated as unknown. However, using the `OPTIX_RAY_FLAG_FORCE_OPACITY_MICROMAP_2_STATE` ray flag or `OPTIX_INSTANCE_FLAG_FORCE_OPACITY_MICROMAP_2_STATE` instance flag they can be forced to be opaque and transparent, respectively. This redefinition affords some flexibility of interpretation: in some ray-traced effects, exact resolution is not required and eliminating all any-hit program invocation is visually acceptable. For example, soft shadows may be resolved using a lower resolution proxy. The `OPTIX_GEOMETRY_FLAG_DISABLE_ANYHIT` flag is ignored for triangles which have OMMs attached, as OMMs give much more fine-grained control and are intended to replace the geometry-wide state where possible.

Ray flags and instance flags may still alter the state of opaque hits, but note that any such flags are only applied *after* the OMM hit classification has occurred. This means that once the OMM has been evaluated, there is no way to turn a transparent microtriangle miss into a hit even by using the `OPTIX_RAY_FLAG_DISABLE_ANYHIT` ray flag or `OPTIX_INSTANCE_FLAG_DISABLE_ANYHIT` instance flag.

It is still possible to turn off OMMs and revert to the geometry-specified behavior for individual instances using the `OPTIX_INSTANCE_FLAG_DISABLE_OPACITY_MICROMAPS` instance flag. This flag is only valid if the referenced BLAS was originally built with the `OPTIX_BUILD_FLAG_ALLOW_DISABLE_OPACITY_MICROMAPS` build flag. Disabling OMMs on a per-instance basis may be useful to implement certain level-of-detail schemes.

5.11.3 Encoding

OMM are bit masks of one or two bits per microtriangle.

A 1-bit OMM encodes each microtriangle as either transparent (`OPTIX_OPACITY_MICROMAP_STATE_TRANSPARENT`) or opaque (`OPTIX_OPACITY_MICROMAP_STATE_OPAQUE`) and never requires any-hit program invocation during the tracing of a ray.

A 2-bit OMM is used if there are portions of the opacity that need to be resolved in an any-hit program. The 2-bit OMM encodes each microtriangles as either transparent, opaque, unknown-transparent (`OPTIX_OPACITY_MICROMAP_STATE_UNKNOWN_TRANSPARENT`) or unknown-opaque (`OPTIX_OPACITY_MICROMAP_STATE_UNKNOWN_OPAQUE`).

The OMM microtriangle states are organized along a space-filling curve in barycentric space, as illustrated in Figure 5.5.

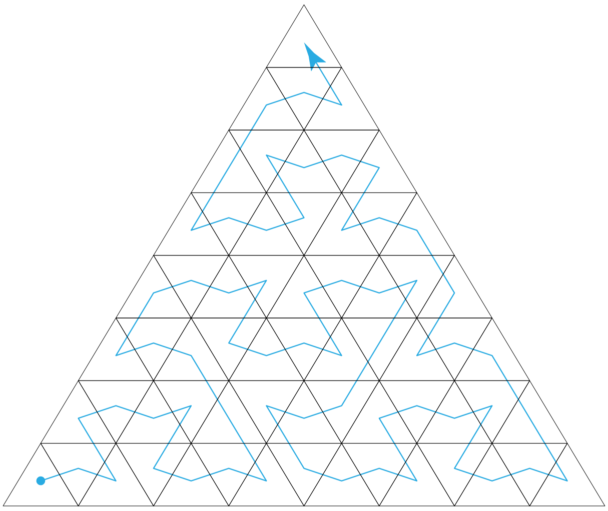

Fig. 5.5 - Order of microtriangle states in a base triangle

The mapping of microtriangle into base triangle barycentric space is implemented by the helper function `optixMicromapIndexToBaseBarycentrics`, made available by including the header file `optix_micromap.h`.

5.12 Clusters

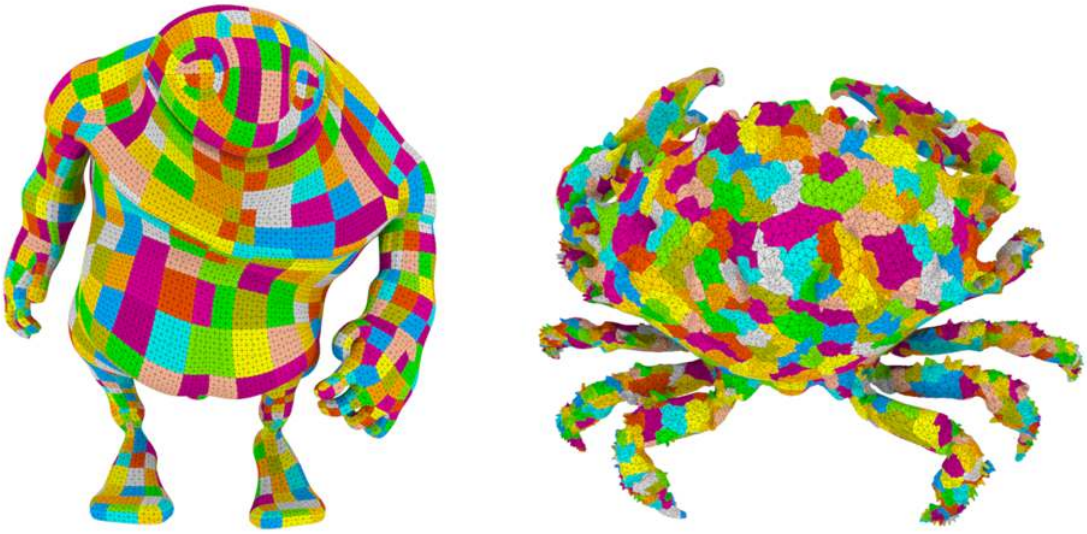

Fig. 5.6 - Meshes made of many small, user-defined clusters of triangles. (Big Guy model courtesy Bay Raitt of Valve Software. Reef crab model from three3dscans.com.)

Acceleration structure build times can become a bottleneck in interactive ray-tracing applications that require large amounts of dynamic geometry. Example scenarios include new geometry being streamed in from disk, high numbers of animated objects, level-of-detail systems, and dynamic tessellation. When building a geometry acceleration structure, OptiX has to discover which triangles are near one another in space; however, the application may already have this information readily available.

We provide a lower-level API for an application to define clusters of triangles directly for acceleration structures, removing a substantial portion of the GAS build cost, while also assuming part of the responsibility for build quality. Use `optixDeviceContextGetProperty` with `OPTIX_DEVICE_PROPERTY_CLUSTER_ACCEL` to check whether the cluster API is supported on a given device. For general hardware version support see "Limits" (page 127).

5.12.1 Overview

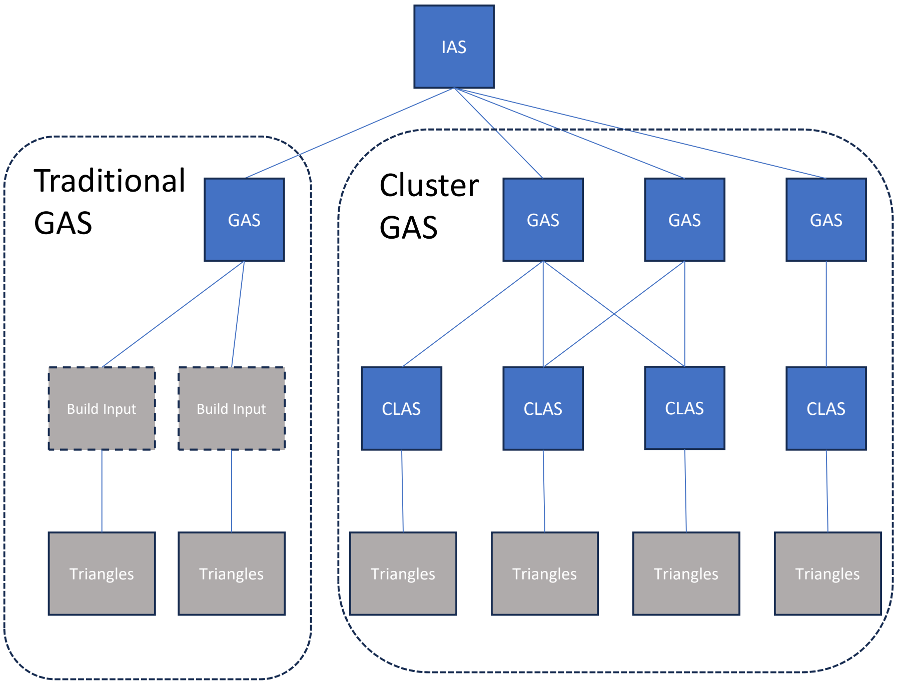

Figure 5.7

The Cluster API provides two additional acceleration structure types and one additional data structure:

- A Cluster Acceleration Structure (CLAS) built over triangles.
- A Cluster GAS, built over CLAS primitives.
- A Cluster Template built over vertex indices. This is an intermediate data structure that helps with cluster reuse and will be described in more detail below.

All three of these are built by providing different arguments to `optixClusterAccelBuild`.

Listing 5.21

```
optixClusterAccelBuild(
    OptixDeviceContext context,
    CUstream stream,
    const OptixClusterAccelBuildModeDesc* buildModeDesc,
```

Where should the outputs be written?

```
    const OptixClusterAccelBuildInput* buildInput,
```

What type of inputs will be passed in argsArray?

```
    CUdeviceptr argsArray,
```

Array of inputs in device memory

```
    ...)
```

In all three cases, `optixClusterAccelBuild` builds a list of output items. There are two input structs of type `OptixClusterAccelBuildModeDesc` and `OptixClusterAccelBuildInput` that configure the build. These structs do not contain per-item input data, which is instead provided in device memory via `argsArray`, similar to "multi-indirect" draw calls used in graphics APIs.

The `OptixClusterAccelBuildModeDesc::mode` enum controls how output items are written into device memory:

`OPTIX_CLUSTER_ACCEL_BUILD_MODE_IMPLICIT_DESTINATIONS`
: Write the output into a single buffer, with no guarantees on order or padding between items. Also return auxiliary output buffers with handles and sizes for each output item, in the original order.

`OPTIX_CLUSTER_ACCEL_BUILD_MODE_EXPLICIT_DESTINATIONS`
: Write each output item to an address provided by the user.

`OPTIX_CLUSTER_ACCEL_BUILD_MODE_GET_SIZES`
: Return a size estimate for each output item, but don't write anything else. These sizes are only valid for future builds using explicit destinations.

For each mode above, the build mode struct also has pointers to the matching output and temp buffers for the mode. These buffers need to be sized appropriately beforehand using `optixClusterAccelComputeMemoryUsage()` or the result of call using `OPTIX_CLUSTER_ACCEL_BUILD_MODE_GET_SIZES`.

The `OptixClusterAccelBuildInput` struct is a tagged union that describes the type of the build, as well as bounds for the input array for that build type. Bounds can be such things as, for example, the maximum number of triangles per cluster in a build that takes triangles as input and produces clusters. Bounds allow OptiX to take the greatest advantage of device parallelism. Making the bounds as tight as possible (i.e. not making them overly conservative) will yield the best performance.

The `OptixClusterAccelBuildInput::type` enum has one of the following values:

- `OPTIX_CLUSTER_ACCEL_BUILD_TYPE_CLUSTERS_FROM_TRIANGLES`
- `OPTIX_CLUSTER_ACCEL_BUILD_TYPE_CLUSTERS_FROM_TEMPLATES`
- `OPTIX_CLUSTER_ACCEL_BUILD_TYPE_TEMPLATES_FROM_TRIANGLES`
- `OPTIX_CLUSTER_ACCEL_BUILD_TYPE_TEMPLATES_FROM_GRIDS`
- `OPTIX_CLUSTER_ACCEL_BUILD_TYPE_GASES_FROM_CLUSTERS`

Each of these builds produces one of the cluster data types described below.

5.12.2 Cluster Acceleration Structure

Build type: `CLUSTERS_FROM_TRIANGLES` or `CLUSTERS_FROM_TEMPLATES`

A CLAS behaves like a tiny GAS but with the following differences:

- A CLAS contains a limited number of triangles and vertices (currently up to 256 of each).
- A CLAS can only be used in a Cluster GAS; it cannot be directly included in an Instance Acceleration Structure (IAS) nor traced on its own.
- SBT indices are directly specified per CLAS, or optionally per triangle, and may be non-consecutive. This is different from the way SBT indices are specified in an ordinary GAS build.
- A CLAS is assigned a user-defined 32-bit cluster ID that can be fetched from a hit shader. These values do not have to be unique or contiguous.
- Cluster primitive indices returned by `optixGetPrimitiveIndex` are relative to the cluster that was hit. To locate a triangle within the Cluster GAS, the cluster ID must be used in combination with the primitive index.
- Vertex data fetch is only supported for the current triangle that was hit – there is no random access to other triangles in the GAS.

Currently there is no support for motion blur in either a CLAS or a Cluster GAS.

A large number of CLASes can be built with a single call to `optixClusterAccelBuild` with a build type of `CLUSTERS_FROM_TRIANGLES`. The build input describes the number of clusters, and bounds on the number of triangles, vertices, and SBT indices:

Listing 5.22

```
// Set up the build input description
OptixClusterAccelBuildInput buildInputs = {};
buildInputs.type = OPTIX_CLUSTER_ACCEL_BUILD_TYPE_CLUSTERS_FROM_TRIANGLES;
buildInputs.triangles.maxArgCount = clusterCount;
buildInputs.triangles.vertexFormat = OPTIX_VERTEX_FORMAT_FLOAT3;
buildInputs.triangles.maxSbtIndexValue = materialCount-1;
buildInputs.triangles.maxUniqueSbtIndexCountPerArg = 1;
buildInputs.triangles.maxTriangleCountPerArg = maxTrianglesPerCluster;
buildInputs.triangles.maxVertexCountPerArg  = maxVerticesPerCluster;
...
```

Recall that the input triangle data is passed completely in device memory. The creation of this data is application specific, but let's assume we have a user-defined CUDA kernel that generates the vertices, indices, etc., for all clusters (perhaps using tessellation or precomputed clusters). You can find example kernels in the OptiX SDK samples. The following code sketches how to create the remaining input data and to build CLASes for all the clusters in a single call.

Listing 5.23

```
// Generate input argument data for all clusters
OptixClusterAccelBuildInputTrianglesArgs *d_inputTrianglesArgs = nullptr;
cudaMalloc( reinterpret_cast<void**>( &d_inputTrianglesArgs ),
    clusterCount * sizeof( OptixClusterAccelBuildInputTrianglesArgs ) ) );

dim3 blockSize( 32, 1, 1 );
dim3 gridSize( ( clusterCount + blockSize.x - 1 ) / blockSize.x, 1, 1 );
makeInputTrianglesArgs_kernel<<<gridSize, blockSize»>( clusterCount,
    d_inputTrianglesArgs );

// Set up the build input description
OptixClusterAccelBuildInput buildInputs = {};
buildInputs.type = OPTIX_CLUSTER_ACCEL_BUILD_TYPE_CLUSTERS_FROM_TRIANGLES;
buildInputs.triangles.maxArgCount = clusterCount;
...           fill in other bounds

// Allocate output and temp buffers
OptixAccelBufferSizes bufferSizes = {};
optixClusterAccelComputeMemoryUsage( context,
    OPTIX_CLUSTER_ACCEL_BUILD_MODE_IMPLICIT_DESTINATIONS,
    &buildInputs, &bufferSizes );

CUdeviceptr d_tempBuffer = 0, d_clasBuffer = 0;
CUdeviceptr* d_clasPointersBuffer = nullptr;
uint32_t* d_clasSizesBuffer = nullptr;

cudaMalloc( reinterpret_cast<void**>( &d_clasBuffer ),
    bufferSizes.outputSizeInBytes ) );
...           allocate other buffers

// Set up the build mode description (where to write the output)
OptixClusterAccelBuildModeDesc buildModeDesc = {};
buildModeDesc.mode = OPTIX_CLUSTER_ACCEL_BUILD_MODE_IMPLICIT_DESTINATIONS;
buildModeDesc.implicitDest.outputBuffer = d_clasBuffer;
buildModeDesc.implicitDest.outputBufferSizeInBytes =
    bufferSizes.outputSizeInBytes;
...           fill in other fields

// Build CLASes for all clusters.  The argsCount and argsStrideInBytes
// are optional and can be set to 0 in this case; the maxArgCount from
// the build input will be used instead.
optixClusterAccelBuild( context, 0, &buildModeDesc, &buildInput,
    d_inputTrianglesArgs, 0, 0 );
```

While we leave the details of the CUDA kernel undefined, its job is to write the `OptixClusterAccelBuildInputTrianglesArgs` input data per cluster:

Listing 5.24

```
__global__ void makeInputTrianglesArgs_kernel(
    uint32_t clusterCount,
    OptixClusterAccelBuildInputTrianglesArgs* d_inputTrianglesArgs,
    ...           other app-defined inputs
)
{
    const uint32_t clusterIndex = blockIdx.x * blockDim.x + threadIdx.x;
    if( clusterIndex >= clusterCount )
        return;

    float3* vertices = foo();      Application logic to generate or look up vertices,
    uint32_t* indices = bar();     indices, etc.

    // Write the input data for one cluster
    OptixClusterAccelBuildInputTrianglesArgs inputTrianglesArg = {};
    inputTrianglesArg.clusterId = clusterIndex;
    inputTrianglesArg.triangleCount = triangleCount;
    inputTrianglesArg.indexBuffer = CUdeviceptr( indices );
    inputTrianglesArg.vertexCount = vertexCount;
    inputTrianglesArg.positionTruncateBitCount = 0;    Optional vertex
                                                       quantization
    inputTrianglesArg.vertexBuffer = CUdeviceptr( vertices );
    ...           fill in other fields

    d_inputTrianglesArgs[clusterIndex] = inputTrianglesArg;
}
```

Please consult the OptiX header files for other fields not shown here, such as an optional Opacity Micromap that may be assigned per triangle of the cluster. The `positionTruncateBitCount` field provides a way to quantize the input vertex positions, which can reduce the size of the resulting clusters. It specifies the number of low-order bits of the position mantissa values that will be considered to be zero.

5.12.3 Cluster Templates

Build type: `TEMPLATES_FROM_TRIANGLES`, `TEMPLATES_FROM_GRIDS`

A Cluster Template captures the topology of a cluster of triangles (i.e. the vertex index list) in a form that can be very efficiently instantiated into CLASes with that same topology, each with unique vertex positions, cluster ID, and SBT indices. This speeds up builds that reuse a set of cluster patterns, either within a frame or from frame to frame. An important case for this is *adaptive tessellation*, where a coarse base mesh (e.g., the control cage of a subdivision surface) is refined into a detailed mesh using a small set of grid cluster patterns (e.g., 8x8 grids of points evaluated on the limit surface).

Figure 5.8 (page 55) shows how a Cluster is instantiated directly from triangles (a), versus using Cluster Templates (b, c). Part (c) shows a special case of a Cluster Template that is a regular grid, parameterized by the grid dimensions rather than vertex indices.

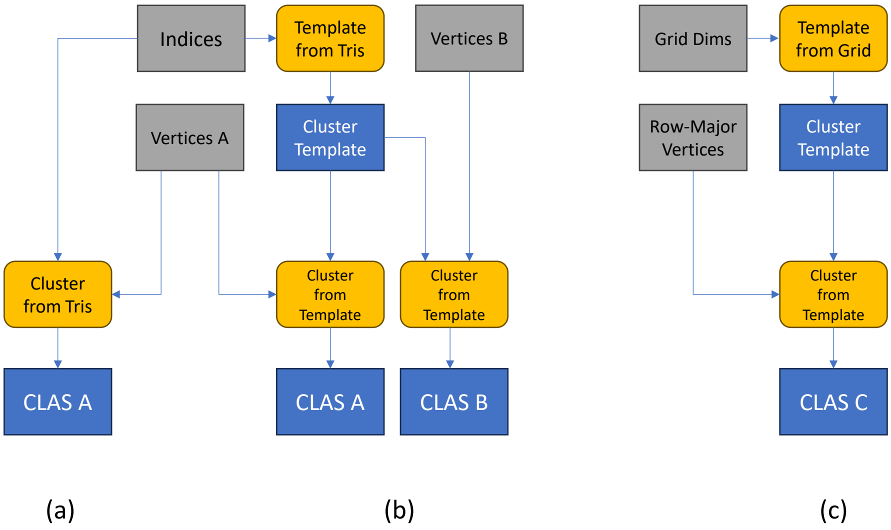

Figure 5.8

OptiX can optimize grid templates because their topology is particularly simple. Use `optixDeviceContextGetProperty` with `OPTIX_DEVICE_PROPERTY_LIMIT_MAX_STRUCTURED_GRID_RESOLUTION` for the max supported dimension in either axis. Rectangular grids are supported too. As an example, a grid template with 4x2 edge segments per side would expect the following row-major order for its input vertices (a) and would have topology (b), with winding order (c):

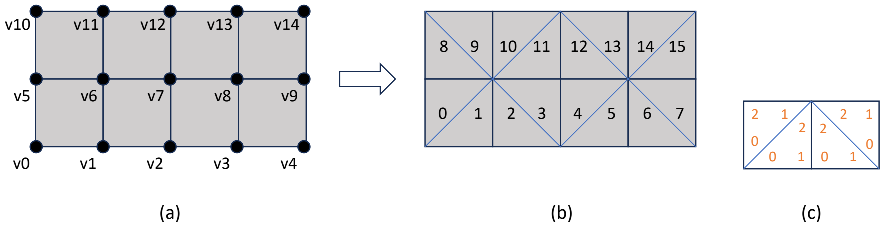

Figure 5.9

Here is one way an application could use Cluster Templates for tessellated, animated geometry:

- Precompute a Cluster Template for each possible cluster topology that might be output. This is hopefully a small set (dozens or hundreds) compared to the total number of clusters (thousands or millions). This step happens only once, at startup or after loading a scene.
- For each frame:
  - Use compute kernels to create the vertices of each cluster, and select a Cluster Template per cluster. This step could involve animation, tessellation, and displacement.
  - Instantiate Cluster Templates using the vertices from above to produce a CLAS per cluster.

- Create a Cluster GAS over the CLASes (see below). This is a new Cluster GAS, not a refit; the Cluster GAS is typically small and quick to compute because its leaf nodes are clusters rather than triangles.

This workflow provides build performance comparable with traditional refit but is more flexible because it permits topology changes. It also has lower memory usage and more stable trace performance versus traditional refit, and only clusters that are animated need to be rebuilt. It does, however, rely on a good clustering of the mesh by the application, where triangles in a cluster are near each other in space!

The kernel for building template arguments is similar to the one for building CLAS arguments, except that the vertex positions in device memory for each template are optional; if passed, they are considered to be examples, and are not required to match the final vertex positions, although the template may be more optimized if the two are similar.

Listing 5.25

```
__global__ void makeInputTrianglesArgsForTemplates_kernel(
    uint32_t templateCount,
    OptixClusterAccelBuildInputTrianglesArgs* d_inputTrianglesArgs,
    ...
)
{
    const uint32_t templateIndex = ...

    uint32_t* indices = bar();

    OptixClusterAccelBuildInputTrianglesArgs arg = {};   Input data for one
                                                         template
    ...
    arg.indexBuffer = CUdeviceptr( indices );
    arg.vertexBuffer = 0;   Optional sample vertices

    arg.positionTruncateBitCount = 0;          Optional vertex quantization/bounds
    arg.instantiationBoundingBoxLimit = 0;     for future clusters
    ...

    d_inputTrianglesArgs[templateIndex] = arg;
}
```

Note the optional vertex quantization and bounding box fields. These form a contract: all future vertices will have their lower-order bits considered to be zero, and the bounding box must bound the vertices of any future clusters instantiated from the template. Passing this extra information is not required but may reduce the predicted memory usage for the clusters.

The kernel for building grid template arguments is simpler since it takes the grid dimensions rather than indices:

Listing 5.26

```
__global__ void makeInputGridsArgsForTemplates_kernel(
    uint32_t templateCount,
    OptixClusterAccelBuildInputGridsArgs* d_inputGridsArgs,
    ...
)
{
    ...

    OptixClusterAccelBuildInputGridsArgs arg = {};
    ...
    arg.dimensions[0] = 4;
    arg.dimensions[1] = 2;
```

In place of indices, set the number of edge segments per grid

```
    arg.positionTruncateBitCount = 0;
```

Optional vertex quantization for future clusters

```
    ...

    d_inputGridsArgs[templateIndex] = arg;
}
```

Once the template is built, it can be used to instantiate a CLAS. The template replaces the input indices and the build type switches to `CLUSTERS_FROM_TEMPLATES`.

Listing 5.27 – Device code for template instantiation

```
__global__ void makeInputClustersFromTemplatesArgs_kernel(
    uint32_t clusterCount,
    CUdeviceptr* templatesPerCluster,
    OptixClusterAccelBuildInputTemplatesArgs* d_inputArgs,
    ...
)
{
    const uint32_t clusterIndex = ...

    float3* vertices = foo();
```

Application logic to generate or look up vertices, which must match the topology of the template

```
    OptixClusterAccelBuildInputTemplatesArgs arg = {};
    arg.clusterIdOffset = clusterIndex;
    arg.clusterTemplate = templatesPerCluster[clusterIndex];
    arg.vertexBuffer = CUdeviceptr( vertices );
    ...

    d_inputArgs[clusterIndex] = arg;
}
```

Listing 5.28 – Host code for template instantiation

```
OptixClusterAccelBuildInputTemplatesArgs *d_inputArgs = nullptr;
cudaMalloc( reinterpret_cast<void**>( &d_inputArgs ),
   clusterCount * sizeof( OptixClusterAccelBuildInputTemplatesArgs ) ) );

dim3 blockSize( 32, 1, 1 );
dim3 gridSize( ( clusterCount + blockSize.x - 1 ) / blockSize.x, 1, 1 );
...
makeInputClustersFromTemplatesArgs_kernel<<<gridSize, blockSize»>(
   clusterCount, d_templates, d_inputArgs );

OptixClusterAccelBuildInput buildInput = {};
buildInputs.type = OPTIX_CLUSTER_ACCEL_BUILD_TYPE_CLUSTERS_FROM_TEMPLATES;
buildInputs.triangles.maxArgCount = clusterCount;
...          fill in other bounds, allocate memory, do the build
```

There is one other subtle point relating to memory usage. Templates can be used to compute upper bounds on the size of the output CLASes, which are often the largest chunk of the acceleration structure data. Recall that a build mode of `OPTIX_CLUSTER_ACCEL_BUILD_MODE_GET_SIZES` will return the output sizes without writing other data. These sizes are expected to be more accurate than when using `optixClusterAccelComputeMemoryUsage`, but cost more to compute. There is a twist however: when using `GET_SIZES` with a `CLUSTERS_FROM_TEMPLATES` build, you may pass a null vertex pointer to return sizes that are an upper bound for any possible CLAS instantiated from that template. Thus you can run GET_SIZES just once per unique template using null vertex pointers, cache the sizes, and reuse them for the lifetime of the template.

5.12.4 Cluster GAS

Build type: `GASES_FROM_CLUSTERS`

- A Cluster GAS is a type of GAS, and so can be passed to an IAS build operation, etc. However, there are some differences in its build and usage.
- A Cluster GAS can only be built with `optixClusterAccelBuild`, not with `optixAccelBuild`.
- A Cluster GAS only accepts Clusters as input, not triangles, curves, etc.
- Motion blur is not supported.
- Multiple Cluster GASes can be built in a single call.
- A Cluster GAS contains explicit references to its CLAS inputs. If CLASes are moved or become invalid then any Cluster GAS referencing them will likewise become invalid.

Note that this is a fundamental difference from other types of GAS, where all input primitive information is stored in the GAS itself.

Listing 5.29 – Host code for a Cluster GAS

```
// Allocate input argument data for all GASes
OptixClusterAccelBuildInputClustersArgs *d_inputArgs = nullptr;
cudaMalloc( reinterpret_cast<void**>( &d_inputArgs ),
   gasCount * sizeof( OptixClusterAccelBuildInputClustersArgs ) ) );

// Generate input argument data in device memory for GASes
...          see listing below

// Set up the build input description
OptixClusterAccelBuildInput gasBuildInputs = {};
gasBuildInputs.type = OPTIX_CLUSTER_ACCEL_BUILD_TYPE_GASES_FROM_CLUSTERS;
gasBuildInputs.clusters.flags = OPTIX_CLUSTER_ACCEL_BUILD_FLAG_NONE;
gasBuildInputs.clusters.maxArgCount = gasCount;
gasBuildInputs.clusters.maxTotalClusterCount = maxClusterCount;
gasBuildInputs.clusters.maxClusterCountPerArg = maxClustersPerGas;

// Allocate output and temp buffers, similar to other builds
...

// Set up the build mode description
...

// Build GASes
optixClusterAccelBuild( context, 0, &buildModeDesc, &buildInput,
   d_inputArgs, 0, 0 );
```

The input argument data points to a buffer of CLAS device pointers per GAS; these are the same pointers that were output from earlier CLAS builds. When building a single GAS, there is only one input argument but it still needs to be created in device memory with a kernel or cudaMemcpy:

Listing 5.30

```
// Generate input argument data for a single GAS (gasCount=1) and
// copy arg to device memory
OptixClusterAccelBuildInputClustersArgs h_inputArg = {};
h_inputArg.clusterCount = numClusters;
h_input.clusters = d_clasPointerBuffer;

cudaMemcpy( reinterpret_cast<void*>( d_inputArgs ), h_inputArg,
   sizeof(OptixClusterAccelBuildInputClustersArgs),
   cudaMemcpyHostToDevice )
```

5.12.5 Cluster Shading Data

During shading, the cluster ID, local triangle ID and triangle vertices for the current ray hit within a CLAS can be retrieved in a closest-hit shader using the built-in functions `optixGetClusterId`, `optixGetPrimitiveIndex` and `optixGetTriangleVertexData`. Other user data such as vertex normals, texture coordinates, or material parameters need to be stored separately from the acceleration structure and indexed using either the Shading Binding Table (SBT), or the user-defined cluster ID and/or instance ID.

Each cluster maps to one or more consecutive records in the SBT. These indices are set directly as part of the per-cluster device input data, e.g., in `OptixClusterAccelBuildInputTrianglesArgs`. There is no concept of a "build input" for Cluster GASes other than the CLAS inputs. When converting an existing GAS to a Cluster GAS, it should be straightforward to translate SBT indices that are already working well in the regular case.

Another alternative is to keep the SBT indexing as simple as possible, e.g., a single program binding over an entire Cluster GAS, and handle geometry and material variation directly using ClusterId and InstanceId in the closest-hit program. For example, a "shading data" buffer with one entry per cluster could hold the material ID for the cluster plus an offset into a buffer of vertex normals, etc.

6 Program pipeline creation

The following API functions are described in this section:

```
optixModuleCreate
optixModuleCreateWithTasks
optixModuleDestroy
optixModuleGetCompilationState
optixProgramGroupCreate
optixProgramGroupGetStackSize
optixPipelineCreate
optixPipelineDestroy
optixPipelineSetStackSize
optixTaskExecute
```

Programs are first compiled into *modules* of type `OptixModule`. One or more modules are combined to create a *program group* of type `OptixProgramGroup`. Those program groups are then linked into an `OptixPipeline` on the GPU. This is similar to the compile and link process commonly found in software development. The program groups are also used to initialize the header of the SBT record associated with those programs.

The three create methods, `optixModuleCreate`, `optixProgramGroupCreate`, and `optixPipelineCreate` take an optional log string. This string is used to report information about any compilation that may have occurred, such as compile errors or verbose information about the compilation result. To detect truncated output, the size of the log message is reported as an output parameter. It is not recommended that you call the function again to get the full output because this could result in unnecessary and lengthy work, or different output for cache hits. If an error occurred, the information that would be reported in the log string is also reported by the device context log callback (when provided).

Both mechanisms are provided for these create functions to allow a convenient mechanism for pulling out compilation errors from parallel creation operations without having to determine which output from the logger corresponds to which API invocation.

Symbols in `OptixModule` objects may be unresolved and contain `extern` references to variables and `__device__` functions.

These symbols can be resolved during pipeline creation using the symbols defined in the pipeline modules. Duplicate symbols will trigger an error.

A *pipeline* contains all programs that are required for a particular ray-tracing launch. An application may use a different pipeline for each launch, or may combine multiple ray-generation programs into a single pipeline.

Most NVIDIA OptiX API functions do not own any significant GPU state; Streaming Assembly (SASS) instructions, which define the executable binary programs in a pipeline, are an exception. The `OptixPipeline` owns the CUDA resource associated with the compiled SASS and it is held until the pipeline is destroyed. This allocation is proportional to the amount of compiled code in the pipeline, typically tens of kilobytes to a few megabytes. However, it is possible to create complex pipelines that require substantially more memory, especially if large static initializers are used. Wherever possible, exercise caution in the number and size of the pipelines.

Module lifetimes need to extend to the lifetimes of program groups that reference them. After using modules to create an `OptixPipeline` through the `OptixProgramGroup` objects, modules may be destroyed with `optixModuleDestroy`.

6.1 Program input

NVIDIA OptiX programs are encoded by the compiler into either *OptiX-IR*, a proprietary intermediate representation, or into *PTX*,[1] an instruction set designed for parallel thread execution. OptiX-IR contains a richer representation of the code that enables symbolic debugging and provides opportunities for enhanced optimizations and future features. OptiX-IR is a binary format that can only be read by NVIDIA tools, unlike PTX, which is stored in plain-text format.

To create PTX programs, compile CUDA source files using the NVIDIA `nvcc` offline compiler[2] or `nvrtc` JIT compiler.[3] To create OptiX-IR programs, compile the CUDA source files with `nvcc`. OptiX device headers should be included in the source files to provide the device API for OptiX programs.

Transitioning from PTX to OptiX-IR input is recommended when using `nvcc` to generate code for OptiX. PTX will continue to be supported but may not provide all OptiX-IR features, such as symbolic debugging.

The following example uses `nvcc` to create an OptiX-IR program:

```
nvcc -optix-ir -Ipath-to-optix-sdk/include --use_fast_math myprogram.cu \
     -o myprogram.optixir
```

The `nvcc` command-line options are explained in more detail as part of the usage description of the compiler options displayed with `nvcc --help`.

Note the following requirements for `nvcc` and `nvrtc` compilation:

- The streaming multiprocessor (SM) target of the input OptiX program must be less than or equal to the SM version of the GPU for which the module is compiled.
- To generate code for the minimum supported GPU (Maxwell), use architecture targets for SM 5.0, for example, `--gpu-architecture=compute_50`. Because OptiX rewrites the code internally, those targets will work on any newer GPU as well.

[1] https://docs.nvidia.com/cuda/parallel-thread-execution/

[2] https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/

[3] https://docs.nvidia.com/cuda/nvrtc/

- CUDA Toolkits 10.2 and newer throw deprecation warnings for SM 5.0 targets. These can be suppressed with the compiler option `-Wno-deprecated-gpu-targets`.

  If support for Maxwell GPUs is not required, you can use the next higher GPU architecture target SM 6.0 (Pascal) to suppress these warnings.
- Use `--machine=64` (`-m64`). Only 64-bit code is supported in OptiX.
- Define the output type with `--optix-ir` or `--ptx`. Do not compile to `obj` or `cubin`.
- For debugging, use the debug flag `-G`. Symbolic debugging is currently only supported for OptiX-IR and not PTX, though PTX compiled with debug support can be used as input to OptiX. It may also be necessary to set the environment variable `OPTIX_FORCE_DEPRECATED_LAUNCHER` to 1. If breakpoints are unable to be hit, try setting this environment variable before starting your application.
- Enable `--relocatable-device-code=true` (`-rdc`). Command `nvcc` can also use the option `--keep-device-functions`, which is not supported by `nvrtc`. These flags prevent the CUDA compiler from eliminating direct or continuation callables as dead code.
- To get smaller and faster code, enable `--use_fast_math`. This flag enables `.approx` instructions for trigonometric functions and reciprocals, avoiding inadvertent use of slow double-precision floats. For performance reasons, it is recommended that you set this flag; the only exception is use cases that require more precision.
- To profile your code with Nsight Compute,[4] enable `--generate-line-info` and set `debugLevel = OPTIX_COMPILE_DEBUG_LEVEL_MODERATE` in the `OptixModuleCompileOptions` in your application host code.

6.2 Programming model

The NVIDIA OptiX programming model supports the *multiple instruction, multiple data* (MIMD) subset of CUDA. Execution must be independent of other threads. For this reason, shared memory usage and warp-wide or block-wide synchronization — such as barriers — are not allowed in the input PTX code. All other GPU instructions are allowed, including math, texture, atomic operations, control flow, and loading data to memory. Special warp-wide instructions like vote and ballot are allowed, but can yield unexpected results as the locality of threads is not guaranteed and neighboring threads can change during execution, unlike in the full CUDA programming model. Still, warp-wide instructions can be used safely when the algorithm in question is independent of locality by, for example, implementing warp-aggregated atomic adds. The special registers in PTX - defined in the PTX IR[5] - are interpreted in OptiX as follows:

PTX Special Register	*Interpretation in OptiX*
`%tid`	launch index
`%ntid`	launch dimension
`%ctaid`	always equals to 0

[4] https://developer.nvidia.com/nsight-compute

[5] http://cuda-internal/docs/cuda/gpgpu/current/parallel-thread-execution/index.html#special-registers

`%nctaid`	always equals to 1
`%laneid`	unchanged, but volatile across trace calls
`%warpid`	unchanged
`%nwarpid`	unchanged
`%smid`	unchanged
`%nsmid`	unchanged
`%gridid`	unchanged
`%lanemask_eq, ...`	unchanged, but volatile across trace calls
`%clock, ...`	unchanged
`%pm0, ...`	invalid PTX error
`%pm0_64, ...`	invalid PTX error
`%envreg0, ...`	invalid PTX error
`%globaltimer, ...`	unchanged
`%reserved_smem_offset_begin, ...`	invalid PTX error
`%total_smem_size`	invalid PTX error
`%dynamic_smem_size`	invalid PTX error

While the first four registers can be accessed through CUDA intrinsics, namely `threadIdx`, `blockDim`, `blockIdx` and `gridDim` respectively, all other special registers require the use of PTX inline assembly. The usage of either CUDA intrinsics or PTX inline assembly applies to both PTX or OptiX-IR input.

The memory model is consistent only within the execution of a single launch index, which starts at the ray-generation invocation and only with subsequent programs reached from any `optixTrace` or callable program. This includes writes to stack allocated variables. Writes from other launch indices may not be available until after the launch is complete. If needed, atomic operations may be used to share data between launch indices, as long as an ordering between launch indices is not required. Memory fences are not supported.

The input PTX should include one or more NVIDIA OptiX programs. The type of program affects how the program can be used during the execution of the pipeline. These program types are specified by prefixing the program's name with the following:

Program type	*Function name prefix*
Ray generation	`__raygen__`
Intersection	`__intersection__`
Any hit	`__anyhit__`
Closest hit	`__closesthit__`
Miss	`__miss__`
Direct callable	`__direct_callable__`
Continuation callable	`__continuation_callable__`
Exception	`__exception__`

If a particular function needs to be used with more than one type, then multiple copies with corresponding program prefixes should be generated.

In addition, each program may call a specific set of device-side intrinsics that implement the actual ray-tracing-specific features. (See "Device-side functions" (page 129).)

6.3 Module creation

A module may include multiple programs of any program type. Two option structs control the parameters of the compilation process:

`OptixPipelineCompileOptions`
: Must be identical for all modules used to create program groups linked in a single pipeline.

`OptixModuleCompileOptions`
: May vary across the modules within the same pipeline.

These options control general compilation settings, for example, the level of optimization. `OptixPipelineCompileOptions` controls features of the API such as the usage of custom any-hit programs, curve primitives, sphere primitives, motion blur, exceptions, ray payload and primitive attributes. For example:

Listing 6.1

```
OptixModuleCompileOptions moduleCompileOptions = {};
moduleCompileOptions.maxRegisterCount =
    OPTIX_COMPILE_DEFAULT_MAX_REGISTER_COUNT;
moduleCompileOptions.optLevel =
    OPTIX_COMPILE_OPTIMIZATION_DEFAULT;
moduleCompileOptions.debugLevel =
    OPTIX_COMPILE_DEBUG_LEVEL_MINIMAL;
moduleCompileOptions.numPayloadTypes = 0;
moduleCompileOptions.payloadTypes = 0;

OptixPipelineCompileOptions pipelineCompileOptions = {};
pipelineCompileOptions.usesMotionBlur = false;
pipelineCompileOptions.traversableGraphFlags =
    OPTIX_TRAVERSABLE_GRAPH_FLAG_ALLOW_SINGLE_LEVEL_INSTANCING;
pipelineCompileOptions.numPayloadValues = 2;
pipelineCompileOptions.numAttributeValues = 2;
pipelineCompileOptions.exceptionFlags = OPTIX_EXCEPTION_FLAG_NONE;
pipelineCompileOptions.pipelineLaunchParamsVariableName = "params";
pipelineCompileOptions.usesPrimitiveTypeFlags = 0;

OptixModule module = nullptr;
char* ptxData = ...;
size_t logStringSize = sizeof( logString );

OptixResult res = optixModuleCreate(
    optixContext,
    &moduleCompileOptions,
    &pipelineCompileOptions,
    ptxData, ptx.size(),
    logString, &logStringSize,
    &module );
```

The `numAttributeValues` field of `OptixPipelineCompileOptions` defines the number of 32-bit words that are reserved to store the attributes. This corresponds to the attribute definition in `optixReportIntersection`. See "Reporting intersections and attribute access" (page 137).

The `numPayloadValues` field of `OptixPipelineCompileOptions` defines the number of 32-bit words that are reserved to store the ray payload. Alternatively ray payload usage can be specified in more detail using the `numPayloadTypes` and `payloadTypes` fields in `OptixModuleCompileOptions`. See "Payload" (page 149).

Note: For best performance when your scene contains nothing but built-in triangles, set `OptixPipelineCompileOptions::usesPrimitiveTypeFlags` to just `OPTIX_PRIMITIVE_TYPE_FLAGS_TRIANGLE`.

6.4 Task API

While multiple invocations of `optixModuleCreate` can be executed in parallel, some large modules can still take a long time to compile. There are opportunities for parallelism within a single module, specifically those with multiple functions. The `OptixTask` API exposes parallel work in a way that allows the developer to manage and integrate the work into their own threading environment.

6.4.1 API

Start Compilation by creating an `OptixTask` object. An `OptixTask` is an opaque type used to break down compilation into independent parallel tasks.

Listing 6.2

```
typedef struct OptixTask_t* OptixTask;
```

You get an OptixTask with a new API function:

Listing 6.3

```
optixModuleCreateFromPTXWithTasks(..., OptixTask* firstTask );
```

All the other arguments are the same as optixModuleCreate with the addition of a new return parameter, OptixTask* firstTask. The OptixModule returned will be in a partially created state (more about this later). This function is designed to be as quick as possible, doing only the minimal amount of work to set up the module for compilation and create the initial OptixTask and is expected to be fast enough to not be necessary to run in parallel. A single thread could generate all the OptixTask objects for multiple OptixModule objects for further processing in a work pool.

Tasks are executed using a new API function:

Listing 6.4

```
OptixResult optixTaskExecute( OptixDeviceContext context, OptixTask task,
                              OptixTask** additionalTasks,
                              int* numAdditionalTasks );
```

Work will be done on the thread executing optixTaskExecute up to the point where OptiX can find additional parallelism. At that point the function will return an array of OptixTask objects. Each OptixTask can be executed in parallel (including between different modules as before). If no additional work is required, optixTaskExecute will set numTasks to 0.

During OptixModule creation, you can query the state of the module with a new API function that returns an enum.

Listing 6.5

```
OptixResult optixModuleGetCompilationState(
                OptixModule module,
                OptixModuleCompileState* state );

typedef enum OptixModuleCompileState
{
    /// No OptixTask objects have started
    OPTIX_MODULE_COMPILE_STATE_NOT_STARTED,

    /// Started, but not all OptixTask objects have completed. No detected
    /// failures.
    OPTIX_MODULE_COMPILE_STATE_STARTED,
    /// Not all OptixTask objects have completed, but at least one has
    /// failed.
    OPTIX_MODULE_COMPILE_STATE_IMPENDING_FAILURE,

    /// All OptixTask objects have completed, and at least one has failed
    OPTIX_MODULE_COMPILE_STATE_FAILED,

    /// All OptixTask objects have completed. The OptixModule is ready to
    /// be used.
    OPTIX_MODULE_COMPILE_STATE_COMPLETED,
} OptixModuleCompileState;
```

If an invocation of `optixTaskExecute` fails

- `OPTIX_MODULE_COMPILE_STATE_IMPENDING_FAILURE` if there are outstanding tasks
- `OPTIX_MODULE_COMPILE_STATE_FAILED` if there are no outstanding tasks.

Subsequent calls to `optixTaskExecute` will not execute additional work aside from cleanup. Currently executing tasks will not necessarily be terminated immediately but at the next opportunity. If the compilation state is `OPTIX_MODULE_COMPILE_STATE_IMPENDIND_FAILURE`, any unstarted `OptixTask` objects can be discarded. They will be cleaned up when `optixModuleDestroy` is called.

Note that there is no explicit destroy method for `OptixTask` objects. They are owned by the `OptixModule` and will be cleaned up either after execution of the task or when the `OptixModule` is destroyed regardless of the success of the compilation. Note that if compilation fails, you still need to call `optixModuleDestroy`. It is illegal to call `optixModuleDestroy` if any `OptixTask` objects are currently being executed. In that case `OPTIX_ERROR_ILLEGAL_DURING_TASK_EXECUTE` will be returned.

Here is a diagram of the execution flow:

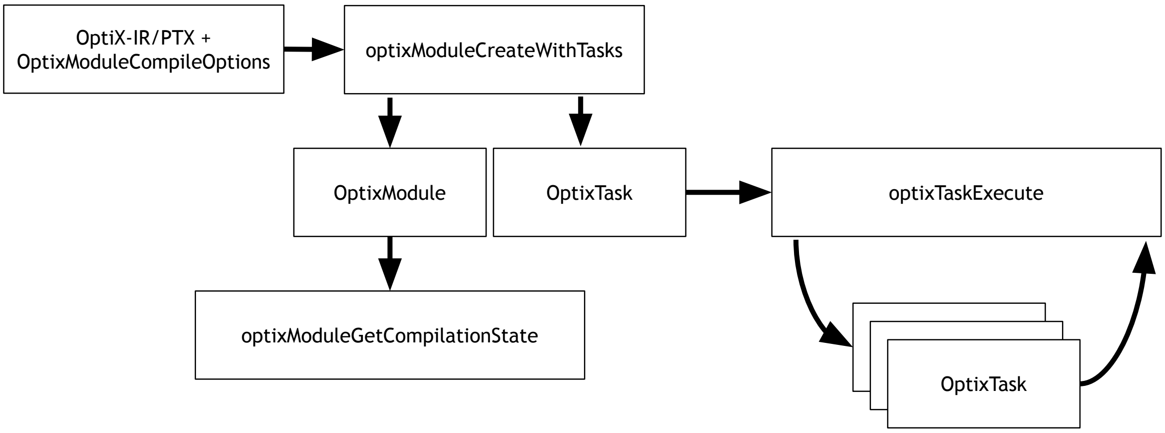
OptiX-IR/PTX +
OptixModuleCompileOptions
optixModuleCreateWithTasks
OptixModule
OptixTask
optixTaskExecute
optixModuleGetCompilationState
OptixTask

Figure 6.1

6.4.2 Liveness of input parameters

All inputs to `optixModuleCreateWithTasks` should be live until the compilation state is either `OPTIX_MODULE_COMPILE_STATE_FAILURE` or `OPTIX_MODULE_COMPILE_STATE_COMPLETED`. In the case of the log string pointed to by logString, logging will continue to be directed to logString at the end of each task that is executed. Errors from each task will appear first before any other feedback. This implies the output will be "resorted" when the logString is updated to keep errors above feedback. The log can be read at any time, but may be changed if there are in-flight invocations to `optixTaskExecute`. logString does not need to be live if there are unexecuted `OptixTask` objects and `optixModuleDestroy` is called.

There is no requirement to run `optixTaskExecute` in parallel or on a separate thread. Execution could be performed serially on the same thread.

Listing 6.6

```
OptixModule module;
std::queue<OptixTask> tasks( 1 );
optixModuleCreateWithTasks( ..., &module, &tasks.front() );

while( !tasks.empty() )
{
    OptixTask task = tasks.front();
    tasks.pop();
    OptixTask* newTasks;
    unsigned int numTasks;
    optixTaskExecute( task, &newTasks, &numTasks );
    for( unsigned int i = 0; i < numTasks; ++i )
        tasks.push( newTasks[i] );
}
```

6.5 Pipeline launch parameter

You specify launch-varying parameters or values that must be accessible from any module through a user-defined variable named in `OptixPipelineCompileOptions`. In each module that needs access, declare this variable with `extern` or `extern "C"` linkage and the `__constant__` memory specifier. The size of the variable must match across all modules in a pipeline. Variables of equal size but differing types may trigger undefined behavior.

For example, the header file in Listing 6.7 defines the variable to shared, named `params`, as an instance of the `Params` struct:

Listing 6.7 – Struct defined in header file `params.h`

```
struct Params
{
    uchar4* image;
    unsigned int image_width;
};
extern "C" __constant__ Params params;
```

Listing 6.8 shows that by including header file `params.h`, programs can access and set the values of the shared `params` struct instance:

Listing 6.8 – Use of header file `params.h` *in OptiX program*

```
#include "params.h"

extern "C"
__global__ void draw_solid_color()
{
    ...
    unsigned int image_index =
        launch_index.y * params.image_width + launch_index.x];
    params.image[image_index] = make_uchar4( 255, 0, 0 );
}
```

6.5.1 Parameter specialization

In some cases it could be beneficial to specialize modules in a pipeline to toggle specific features on and off. For example, users may wish to compile support in their shaders for calculating shadow rays, but may wish to disable this support if the scene parameters do not require them. Users could support this with either multiple versions of the PTX program or by reading a value from the pipeline launch parameters to indicate whether shadows are supported. Multiple versions of the PTX program would allow for the best performance, but come at the cost of maintaining and storing all those program versions. NVIDIA OptiX provides a mechanism for specializing values in the pipeline launch parameters.

During compilation of the module, NVIDIA OptiX will attempt to find loads to the pipeline launch parameter struct that are specified by `OptixPipelineCompileOptions::pipelineLaunchParamsVariableName` within a given range. These specified loads are then each replaced with a predefined value. Compiler optimization passes use those constant values.

The struct `OptixModuleCompileBoundValueEntry` in Listing 6.9 can specify an array of bytes that will replace a portion of the pipeline parameters. The array of bytes to use for specialization is passed as an offset+size pair where the offset and size are measured in bytes and the offset is relative to the beginning of the launch params. An optional annotation string can be provided as well, which will be used to report every instance of specialization found in the pipeline compiler feedback. This allows users to verify the compiler is specializing every instance of bound value usage in the code.

Listing 6.9

```
struct OptixModuleCompileBoundValueEntry {
    size_t pipelineParamOffsetInBytes;
    size_t sizeInBytes;
    const void* boundValuePtr;
    const char* annotation;   Optional string to display
};
```

Listing 6.10 shows how an array of `OptixModuleCompileBoundValueEntry` structs can be specified in `OptixModuleCompileOptions` during module compilation:

Listing 6.10

```
struct OptixModuleCompileOptions {
    ...
    const OptixModuleCompileBoundValueEntry* boundValues;
    unsigned int numBoundValues;
};
```

Listing 6.11 is an example of specializing a module to disable shadow rays:

Listing 6.11 – Device code that references the pipeline launch parameters to determine if shadows are enabled

```
struct LP {
    bool useShadows;
};

extern "C" {
__constant__ LP params;
}

extern "C"
__global__ void __closesthit__ch()
{
    float3 shadowColor = make_float3( 1.f, 1.f, 1.f );
    if( params.useShadows ) {
        shadowColor = traceShadowRay( ... );
    }
    ...
}
```

On the host side, Listing 6.12 shows the implementation of the launch parameters:

Listing 6.12 – Host code to specialize the pipeline launch parameters

```
LP launchParams = {};
launchParams.useShadows = false;

OptixModuleCompileBoundValueEntry useShadow = {};
useShadow.pipelineParamOffsetInBytes = offsetof( LP, useShadows );
useShadow.sizeInBytes = sizeof( LP::useShadows );
useShadow.boundValuePtr = &launchParams.useShadows;

OptixModuleCompileOptions moduleCompileOptions = {};
moduleCompileOptions.boundValues = &useShadow;
moduleCompileOptions.numBoundValues = 1;
...
optixModuleCreate( ..., moduleCompileOptions, ... );
```

This pipeline launch parameter specialization makes the code of Listing 6.13 (page 73) possible:

Listing 6.13

```
extern "C"
__global__ void __closesthit__ch()
{
    float3 shadowColor = make_float3( 1.f, 1.f, 1.f );
    if( false )
    {
        shadowColor = traceShadowRay( ... );
    }
    ...
}
```

Subsequent optimization would remove unreachable code, producing Listing 6.14:

Listing 6.14

```
extern "C"
__global__ void __closesthit__ch()
{
    float3 shadowColor = make_float3( 1.f, 1.f, 1.f );
    ...
}
```

The bound values are intended to represent a constant value in the `pipelineParams`. NVIDIA OptiX will attempt to locate all loads from the `pipelineParams` and correlate them to the appropriate bound value. However, there are cases where these loads and bound values cannot be safely or reliably correlated. For example, correlation is not possible if the pointer to the `pipelineParams` is passed as an argument to a non-inline function or if the offset of the load to the `pipelineParams` cannot be statically determined due to access in a loop. No module should rely on the value being specialized in order to work correctly. The values in the `pipelineParams` specified on `optixLaunch` should match the bound value. If validation mode is enabled on the context, NVIDIA OptiX will verify that the bound values that are specified match the values in `pipelineParams` specified to `optixLaunch`.

If caching is enabled, changes in these values will result in newly compiled modules.

The `pipelineParamOffsetInBytes` and `sizeInBytes` must be within the bounds of the `pipelineParams` variable or `OPTIX_ERROR_INVALID_VALUE` will be returned from `optixModuleCreate`.

If more than one bound value overlaps or the size of a bound value is equal to 0, an `OPTIX_ERROR_INVALID_VALUE` will be returned from `optixModuleCreate`.

The same set of bound values do not need to be used for all modules in a pipeline, but overlapping values between modules must have the same value. `OPTIX_ERROR_INVALID_VALUE` will be returned from `optixPipelineCreate` otherwise.

6.6 Program group creation

OptixProgramGroup objects are created from one to three OptixModule objects and are used to fill the header of the SBT records. (See "Shader binding table" (page 81).) There are five types of program groups.

```
OPTIX_PROGRAM_GROUP_KIND_RAYGEN
OPTIX_PROGRAM_GROUP_KIND_MISS
OPTIX_PROGRAM_GROUP_KIND_EXCEPTION
OPTIX_PROGRAM_GROUP_KIND_HITGROUP
OPTIX_PROGRAM_GROUP_KIND_CALLABLES
```

Modules can contain more than one program. The program in the module is designated by its entry function name as part of the OptixProgramGroupDesc struct passed to optixProgramGroupCreate. Four program groups can contain only a single program; only OPTIX_PROGRAM_GROUP_KIND_HITGROUP can designate up to three programs for the closest-hit, any-hit, and intersection programs.

Programs from modules can be used in any number of OptixProgramGroup objects. The resulting program groups can be used to fill in any number of SBT records. Program groups can also be used across pipelines as long as the compilation options match.

The lifetime of a module must extend to the lifetime of any OptixProgramGroup that references that module.

A hit group specifies the intersection program used to test whether a ray intersects a primitive, together with the hit shaders to be executed when a ray does intersect the primitive. For built-in primitive types, a built-in intersection program should be obtained from optixBuiltinISModuleGet() and used in the hit group. As a special case, the intersection program is not required – and is ignored – for triangle primitives, including triangles in clusters.

The following examples show how to construct a single hit-group program group:

Listing 6.15 – Construct hit-group for custom primitives

```
OptixModule shadingModule, intersectionModule;
...           shadingModule and intersectionModule created here by optixModuleCreate

OptixProgramGroupDesc pgDesc = {};
pgDesc.kind = OPTIX_PROGRAM_GROUP_KIND_HITGROUP;
pgDesc.hitgroup.moduleCH = shadingModule;
pgDesc.hitgroup.entryFunctionNameCH = "__closesthit__shadow";
pgDesc.hitgroup.moduleAH = shadingModule;
pgDesc.hitgroup.entryFunctionNameAH = "__anyhit__shadow";
pgDesc.hitgroup.moduleIS = intersectionModule;
pgDesc.hitgroup.entryFunctionNameIS = "__intersection__sphere";

OptixProgramGroupOptions pgOptions = {};
OptixProgramGroup sphereGroup = nullptr;
```

```
optixProgramGroupCreate(
    optixContext,
    &pgDesc,    programDescriptions
    1,    numProgramGroups
    &pgOptions,    programOptions
    logString, sizeof( logString ),
    &sphereGroup );    programGroup
```

Listing 6.16 – Construct hit-group for built-in curves primitives

```
OptixModule shadingModule, intersectionModule;
...    shadingModule created here by optixModuleCreate

OptixBuiltinISOptions builtinISOptions = {};
builtinISOptions.builtinISModuleType =
    OPTIX_PRIMITIVE_TYPE_ROUND_CUBIC_BSPLINE;
OptixResult res = optixBuiltinISModuleGet(
    optixContext,
    &moduleCompileOptions,
    &pipelineCompileOptions,
    &builtinISOptions,
    &intersectionModule );

OptixProgramGroupDesc pgDesc= {};
pgDesc.kind = OPTIX_PROGRAM_GROUP_KIND_HITGROUP;
pgDesc.hitgroup.moduleCH = shadingModule;
pgDesc.hitgroup.entryFunctionNameCH = "__closesthit__curves";
pgDesc.hitgroup.moduleAH = nullptr;    Any-hit shader is optional
pgDesc.hitgroup.entryFunctionNameAH = nullptr;
pgDesc.hitgroup.moduleIS = intersectionModule;
pgDesc.hitgroup.entryFunctionNameIS = nullptr;    No name for built-in IS

OptixProgramGroupOptions pgOptions = {};
OptixProgramGroup curvesGroup = nullptr;
optixProgramGroupCreate(
    optixContext,
    &pgDesc,
    1,
    &pgOptions,
    logString, sizeof( logString ),
    &curvesGroup );
```

A hit-group construction for sphere primitives would be similar to the built-in curves example, replacing the module type `OPTIX_PRIMITIVE_TYPE_ROUND_CUBIC_BSPLINE` by `OPTIX_PRIMITIVE_TYPE_SPHERE`.

Multiple program groups of varying kinds can be constructed with a single call to `optixProgramGroupCreate`. The following code demonstrates the construction of a ray-generation and miss program group.

Listing 6.17

```
OptixModule rg, miss;
...         Ray-generation and miss programs created here by optixModuleCreate

OptixProgramGroupDesc pgDesc[2] = {};
pgDesc[0].kind = OPTIX_PROGRAM_GROUP_KIND_MISS;
pgDesc[0].miss.module = miss1;
pgDesc[0].miss.entryFunctionName = "__miss__radiance";
pgDesc[1].kind = OPTIX_PROGRAM_GROUP_KIND_RAYGEN;
pgDesc[1].raygen.module = rg;
pgDesc[1].raygen.entryFunctionName = "__raygen__pinhole_camera";
OptixProgramGroupOptions pgOptions = {};
OptixProgramGroup raygenMiss[2];
optixProgramGroupCreate(
    optixContext,
    &pgDesc,  programDescriptions
    2,  numProgramGroups
    &pgOptions,  programOptions
    logString, sizeof( logString ),
    raygenMiss );  programGroup
```

Options defined in `OptixProgramGroupOptions` may vary across program groups linked into a single pipeline, similar to `OptixModuleCompileOptions`.

6.7 Pipeline linking

After all program groups of a pipeline are defined, they must be linked into an `OptixPipeline`. The resulting `OptixPipeline` object is then used to invoke a ray-generation launch.

When the `OptixPipeline` is linked, some fixed function components may be selected based on `OptixPipelineLinkOptions` and `OptixPipelineCompileOptions`. These options were previously used to compile the modules in the pipeline. The link options consist of the maximum recursion depth setting for recursive ray tracing, along with pipeline level settings for debugging. However, the value for the maximum recursion depth has an upper limit that overrides an limit set by the link options. (See "Limits" (page 127).)

For example, the following code creates and links an `OptixPipeline`:

Listing 6.18

```
OptixPipeline pipeline = nullptr;

OptixProgramGroup programGroups[3] =
    { raygenMiss[0], raygenMiss[1], sphereGroup };

OptixPipelineLinkOptions pipelineLinkOptions = {};
pipelineLinkOptions.maxTraceDepth = 1;
optixPipelineCreate(
    optixContext,
    &pipelineCompileOptions,
    &pipelineLinkOptions,
    programGroups,
    3,
    logString, sizeof( logString ),
    &pipeline );
```

After calling `optixPipelineCreate`, the fully linked module is loaded into the driver.

NVIDIA OptiX uses a small amount of GPU memory per pipeline. This memory is released when the pipeline or device context is destroyed.

6.8 Pipeline stack size

The programs in a module may consume two types of stack structure : a *direct stack* and a *continuation stack*. The resulting stack needed for launching a pipeline depends on the resulting call graph, so the pipeline must be configured with the appropriate stack size. These sizes can be determined by the compiler for each program group. A pipeline may be reused for different call graphs as long as the set of programs is the same. For this reason, the pipeline stack size is configured separately from the pipeline compilation options.

The direct stack requirements resulting from ray-generation, miss, exception, closest-hit, any-hit and intersection programs and the continuation stack requirements resulting from exception programs are calculated internally and do not need to be configured. The direct stack requirements resulting from direct-callable programs, as well as the continuation stack requirements resulting from ray-generation, miss, closest-hit, any-hit, intersection, and continuation-callable programs need to be configured. If these are not configured explicitly, an internal default implementation is used. When the maximum depth of call trees of continuation-callable programs is two or less, and no direct-callable programs or motion transforms are used, the default implementation is correct (but not necessarily optimal) Even in cases where the default implementation is correct, Users can always provide more precise stack requirements based on their knowledge of a particular call graph structure.

To query individual program groups for their stack requirements, use `optixProgramGroupGetStackSize`. Use this information to calculate the total required stack sizes for a particular call graph of NVIDIA OptiX programs. To set the stack sizes for a particular pipeline, use `optixPipelineSetStackSize`. For other parameters, helper functions are available to implement these calculations. The following is an explanation about how to compute the stack size for `optixPipelineSetStackSize`, starting from a very conservative approach, and refining the estimates step by step.

Let `cssRG` denote the maximum continuation stack size of all ray-generation programs; similarly for miss, closest-hit, any-hit, intersection, and continuation-callable programs. Let `dssDC` denote the maximum direct stack size of all direct-callable programs. Let maxTraceDepth denote the maximum trace depth (as in `OptixPipelineLinkOptions::maxTraceDepth`), and let `maxCCDepth` and `maxDCDepth` denote the maximum depth of call trees of continuation-callable and direct-callable programs, respectively. Then a simple, conservative approach to compute the three parameters of `optixPipelineSetStackSize` is:

Listing 6.19

```
directCallableStackSizeFromTraversal = maxDCDepth * dssDC;
directCallableStackSizeFromState     = maxDCDepth * dssDC;

cssCCTree =
    maxCCDepth * cssCC;
```

Upper bound on continuation stack used by call trees of continuation callables

```
cssCHOrMSPlusCCTree =
    max( cssCH, cssMS ) + cssCCTree;
```

Upper bound on continuation stack used by closest-hit or miss programs, including the call tree of continuation-callable programs

```
continuationStackSize = cssRG + cssCCTree
    + maxTraceDepth * cssCHOrMSPlusCCTree
    + cssIS + cssAH;
```

This computation can be improved in several ways. For the computation of continuationStackSize, the stack sizes `cssIS` and `cssAH` are not used on top of the other summands, but can be offset against one level of `cssCHOrMSPlusCCTree`. This gives a more complex but better estimate:

Listing 6.20

```
continuationStackSize =
    cssRG
  + cssCCTree
  + max( 1, maxTraceDepth ) - 1 ) * cssCHOrMSPlusCCTree
  + min( maxTraceDepth, 1 ) * max( cssCHOrMSPlusCCTree, cssIS+cssAH );
```

The preceding formulas are implemented by the helper function `optixUtilComputeStackSizes`.

The computation of the first two terms can be improved if the call trees of direct-callable programs are analyzed separately based on the semantic type of their call site. In this context, call sites in any-hit and intersection programs count as traversal, whereas call sites in ray-generation, miss, and closest-hit programs count as state.

Listing 6.21

```
directCallableStackSizeFromTraversal =
    maxDCDepthFromTraversal * dssDCFromTraversal;
directCallableStackSizeFromState =
    maxDCDepthFromState * dssDCFromState;
```

This improvement is implemented by the helper function `optixUtilComputeStackSizesDCSplit`.

Depending on the scenario, these estimates can be improved further, sometimes substantially. For example, imagine there are two call trees of continuation-callable programs. One call tree is deep, but the involved continuation-callable programs need only a small continuation stack. The other call tree is shallow, but the involved continuation-callable programs needs a quite large continuation stack. The estimate of `cssCCTree` can be improved as follows:

Listing 6.22

```
cssCCTree = max( maxCCDepth1 * cssCC1, maxCCDepth2 * cssCC2 );
```

This improvement is implemented by the helper function `optixUtilComputeStackSizesCssCCTree`.

Similar improvements might be possible for all expressions involving `maxTraceDepth` if the ray types are considered separately, for example, camera rays and shadow rays.

6.8.1 Constructing a path tracer

A simple path tracer can be constructed from two ray types: camera rays and shadow rays. The path tracer will consist only of ray-generation, miss, and closest-hit programs, and will not use any-hit, intersection, continuation-callable, or direct-callable programs. The camera rays will invoke only the miss and closest-hit programs `MS1` and `CH1`, respectively. `CH1` might trace shadow rays, which invoke only the miss and closest-hit programs `MS2` and `CH2`, respectively. That is, the maximum trace depth is two and the initial formulas simplify to:

Listing 6.23

```
directCallableStackSizeFromTraversal = maxDCDepth * dssDC;
directCallableStackSizeFromState     = maxDCDepth * dssDC;
continuationStackSize =
    cssRG + 2 * max( cssCH1, cssCH2, cssMS1, cssMS2 );
```

However, from the call graph structure it is clear that `MS2` or `CH2` can only be invoked from `CH1`. This restriction allows for the following estimate:

Listing 6.24

```
continuationStackSize
    = cssRG + max( cssMS1, cssCH1 + max( cssMS2, cssCH2 ) );
```

This estimate is never worse than the previous one, but often better, for example, in the case where the closest-hit programs have different stack sizes (and the miss programs do not dominate the expression).

The helper function `optixUtilComputeStackSizesSimplePathTracer` implements this formula by permitting two arrays of closest-hit programs instead of two single programs.

6.9 Compilation cache

Compilation work is triggered automatically when calling `optixModuleCreate` or `optixProgramGroupCreate`, and also potentially during `optixPipelineCreate`. This work is automatically cached on disk if enabled on the `OptixDeviceContext`. Caching reduces compilation effort for recurring programs and program groups. While it is enabled by default, users can disable it through the use of `optixDeviceContextSetCacheEnabled`. See "Context" (page 15) for other options regarding the compilation cache.

Generally, cache entries are compatible with the same driver version and GPU type only.

7 Shader binding table

The *shader binding table* (SBT) is an array that contains information about the location of programs and their parameters. The SBT resides in device memory and is managed by the application.

7.1 Records

A *record* is an array element of the SBT that consists of a header and a data block. The header content is opaque to the application, containing information accessed by traversal execution to identify and invoke programs. The data block is not used by NVIDIA OptiX and holds arbitrary program-specific application information that is accessible in the program. The header size is defined by the `OPTIX_SBT_RECORD_HEADER_SIZE` macro (currently 32 bytes).

The API function `optixSbtRecordPackHeader` and a given `OptixProgramGroup` object are used to fill the header of an SBT record. The SBT records must be uploaded to the device prior to an NVIDIA OptiX launch. The contents of the SBT header are opaque, but can be copied or moved. If the same program group is used in more than one SBT record, the SBT header can be copied using plain device-side memory copies. For example:

Listing 7.1

```
template <typename T>
struct Record
{
    __align__( OPTIX_SBT_RECORD_ALIGNMENT )
        char header[OPTIX_SBT_RECORD_HEADER_SIZE];
    T data;
};

typedef Record<RayGenData> RayGenSbtRecord;

OptixProgramGroup raygenPG;
...
RayGenSbtRecord rgSBT;
rgSBT.data.color = make_float3( 1.0f, 1.0f, 0.0f );
optixSbtRecordPackHeader( raygenPG, &rgSBT );
CUdeviceptr deviceRaygenSbt;
cudaMalloc( ( void** )&deviceRaygenSbt, sizeof( RayGenSbtRecord ) );
cudaMemcpy( ( void** )deviceRaygenSbt, &rgSBT,
    sizeof( RayGenSbtRecord ), cudaMemcpyHostToDevice );
```

SBT headers can be reused between pipelines as long as the compile options match between modules and program groups. The data section of an SBT record can be accessed on the device using the `optixGetSbtDataPointer` device function.

7.2 Layout

A shader binding table is split into five sections, where each section represents a unique program-group type:

Group	*Program types in group*	*Value of enum* `OptixProgramGroupKind`
Ray generation	ray-generation	`OPTIX_PROGRAM_GROUP_KIND_RAYGEN`
Exception	exception	`OPTIX_PROGRAM_GROUP_KIND_EXCEPTION`
Miss	miss	`OPTIX_PROGRAM_GROUP_KIND_MISS`
Hit	intersection, any-hit, closest-hit	`OPTIX_PROGRAM_GROUP_KIND_HITGROUP`
Callable	direct-callable, continuation-callable	`OPTIX_PROGRAM_GROUP_KIND_CALLABLES`

OptiX program groups

See also “Program group creation” (page 74).

Pointers to the SBT sections are passed to the NVIDIA OptiX launch:

Listing 7.2

```
typedef struct OptixShaderBindingTable
{
    CUdeviceptr raygenRecord;
```

Device address of the SBT record of the ray generation program to start launch

```
    CUdeviceptr  exceptionRecord;
```

Device address of the SBT record of the exception shader

```
    CUdeviceptr  missRecordBase;
    unsigned int missRecordStrideInBytes;
    unsigned int missRecordCount;
```

Arrays of SBT records. The base address, stride in bytes and maximum index are defined.

```
    CUdeviceptr  hitgroupRecordBase;
    unsigned int hitgroupRecordStrideInBytes;
    unsigned int hitgroupRecordCount;

    CUdeviceptr  callablesRecordBase;
    unsigned int callablesRecordStrideInBytes;
    unsigned int callablesRecordCount;
} OptixShaderBindingTable;
```

All SBT records on the device are expected to have a minimum memory alignment, defined by `OPTIX_SBT_RECORD_ALIGNMENT` (currently 16 bytes). Therefore, the stride between records must also be a multiple of `OPTIX_SBT_RECORD_ALIGNMENT`. Each section of the SBT is an independent memory range and is not required to be allocated contiguously.

The selection of an SBT record depends on the program type and uses the corresponding base pointer. Since there can only be a single call to both ray-generation and exception programs, a stride is not required for these two program group types and the passed-in pointer is expected to point to the desired SBT records.

For other types, the SBT record at index *sbt-index* for a program group of type *group-type* is located by the following formula:

group-type`RecordBase` + *sbt-index* `*` *group-type*`RecordStrideInBytes`

For example, the third record (index 2) of the miss group would be:

```
missRecordBase + 2 * missRecordStrideInBytes
```

The index to records in the shader binding table is used in different ways for the miss, hit, and callables groups:

Miss

Miss programs are selected for every `optixTrace` call using the `missSBTIndex` parameter.

Callables

Callables take the index as a parameter and call the direct-callable when invoking `optixDirectCall` and continuation-callable when invoking `optixContinuationCall`.

Any hit, closest hit, intersection

The computation of the index for the hit group (intersection, any-hit, closest-hit) is done during traversal. See "Acceleration structures" (page 83) for more detail.

7.3 Acceleration structures

The selection of the SBT hit group record for the instance is slightly more involved to allow for a number of use cases such as the implementation of different ray types. The SBT record index sbtIndex is determined by the following index calculation during traversal:

```
sbt-index =
    sbt-instance-offset
    + (sbt-geometry-acceleration-structure-index * sbt-stride-from-trace-call)
    + sbt-offset-from-trace-call
```

The index calculation depends upon the following SBT indices and offsets:

- Instance offset
- Geometry acceleration structure index
- Trace offset
- Trace stride

7.3.1 SBT instance offset

Instance acceleration structure instances (type `OptixInstance`) store an SBT offset that is applied during traversal. This is zero for single geometry-AS traversable because there is no corresponding instance-AS to hold the value. (See Traversal of a single geometry acceleration structure (page 35).) This value is limited to 28 bits (see the declaration of `OptixInstance::sbtOffset`).

7.3.2 SBT geometry-AS index

Each geometry acceleration structure build input references at least one SBT record. The first SBT geometry acceleration structure index for each geometry acceleration structure build input is the prefix sum of the number of SBT records. Therefore, the computed SBT geometry acceleration structure index is dependent on the order of the build inputs.

The following example demonstrates a geometry acceleration structure with three build inputs. Each build input references one SBT record by specifying `numSBTRecords=1`. When intersecting geometry at trace time, the SBT geometry acceleration structure index used to compute the sbtIndex to select the hit group record will be organized as follows:

SBT geometry-AS index	*0*	*1*	*2*
Geometry-AS build input	Build input[0]		
		Build input[1]	
			Build input[2]

In this simple example, the index for the build input equals the SBT geometry acceleration structure index. Hence, whenever a primitive from “Build input [1]” is intersected, the SBT geometry acceleration structure index is one.

When a single build input references multiple SBT records (for example, to support multiple materials per geometry), the mapping corresponds to the prefix sum over the number of referenced SBT records.

For example, consider three build inputs where the first build input references four SBT records, the second references one SBT record, and the last references two SBT records:

SBT geometry-AS index	*0*	*1*	*2*	*3*	*4*	*5*	*6*
Geometry-AS build input	Build input[0] `numSBTRecords=4`						
					Build input[1] `numSBTRecords=1`		
						Build input[2] `SBTIndexOffset2`	

These three build inputs result in the following possible SBT geometry acceleration structure indices when intersecting the corresponding geometry acceleration structure build input:

- One index in the range of [0,3] if a primitive from “Build input [0]” is intersected
- Four if a primitive from “Build input [1]” is intersected
- One index in the range of [5,6] if a primitive from “Build input [2]” is intersected

The per-primitive SBT index offsets, as specified by using sbtIndexOffsetBuffer, are local to the build input. Hence, per-primitive offsets in the range [0,3] for the build input 0 and in the range [0,1] for the last build input, map to the SBT geometry acceleration structure index as follows:

SBT geometry-AS index	0	1	2	3	4	5	6
Build input[0] `SBTIndexOffset:`	[0]						
		[1]					
			[2]				
				[3]			
Build input[1] `SBTIndexOffset=nullptr`							
Build input[2] `SBTIndexOffset:`						[0]	
							[1]

Because build input 1 references a single SBT record, a sbtIndexOffsetBuffer does not need to be specified for the geometry acceleration structure build. See “Acceleration structures” (page 83).

7.3.3 SBT trace offset

The `optixTrace` function takes the parameter `SBToffset`, allowing for an SBT access shift for this specific ray. It is required to implement different ray types.

7.3.4 SBT trace stride

The parameter `SBTstride`, defined as an index offset, is multiplied by `optixTrace` with the SBT geometry acceleration structure index. It is required to implement different ray types.

7.3.5 Example SBT for a scene

In this example, a shader binding table implements the program selection for a simple scene containing one instance acceleration structure and two instances of the same geometry acceleration structure, where the geometry acceleration structure has two build inputs:

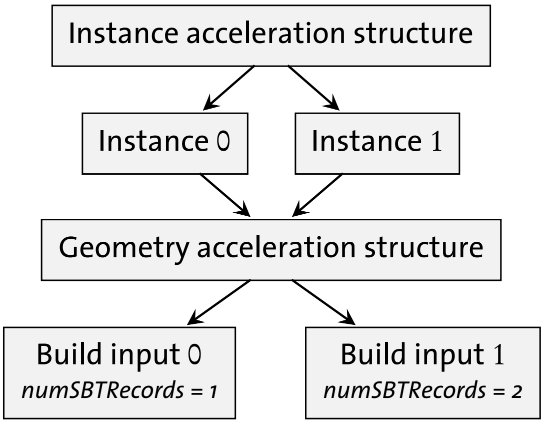

Fig. 7.1 - Structure of a simple scene

The first build input references a single SBT record, while the second one references two SBT records. There are two ray types: one for forward path tracing and one for shadow rays (next event estimation). The two instances of the geometry acceleration structure have different transforms and SBT offsets to allow for material variation in each instance of the same geometry acceleration structure. Therefore, the SBT needs to hold two miss records and 12 hit group records (three for the geometry acceleration structure, ×2 for the ray types, ×2 for the two instances in the instance acceleration structure).

The SBT is structured in the following way:

Raygen	Miss		`instance0.sbtOffset = 0`						`instance1.sbtOffset = 6`					
RG0	MS0	MS1	hit0	hit1	hit2	hit3	hit4	hit5	hit6	hit7	hit8	hit9	hit10	hit11
The preceding programs are called for the following combination of geometry and ray types														
Instance:			0	0	0	0	0	0	1	1	1	1	1	1
Build input:			0	0	1	1	1	1	0	0	1	1	1	1
SBT index:			0	1	2	3	4	5	6	7	8	9	10	11
SBT instance offset:			0	0	0	0	0	0	6	6	6	6	6	6
SBT geometry-AS index:			0	0	1	1	2	2	0	0	1	1	2	2
Build input SBT index offset:			-	-	0	0	1	1	-	-	0	0	1	1
Trace offset/ray type:	0	1	0	1	0	1	0	1	0	1	0	1	0	1

To trace a ray of type 0 (for example, for path tracing):

Listing 7.3

```
optixTrace( IAS_handle,
    ray_org, ray_dir,
    tmin, tmax, time,
    visMask, rayFlags,
    0,   sbtOffset
    2,   sbtStride
    0,   missSBTIndex
    rayPayload0, ... );
```

Shadow rays need to pass in an adjusted sbtOffset as well as `missSBTIndex`:

Listing 7.4

```
optixTrace( IAS_handle,
    ray_org, ray_dir,
    tmin, tmax, time,
    visMask, rayFlags,
    1,   sbtOffset
    2,   sbtStride
    1,   missSBTIndex
    rayPayload0, ... );
```

Program groups of different types (ray generation, miss, intersection, and so on) do not need to be adjacent to each other as shown in the example. The pointer to the first SBT record of each program-group type is passed to `optixLaunch`, as described previously, which allows for arbitrary spacing in the SBT between the records of different program-group types.

7.4 SBT record access on device

To access the SBT data section of the currently running program, request its pointer by using an API function:

Listing 7.5

```
CUdeviceptr optixGetSbtDataPointer();
```

Typically, this pointer is cast to a pointer that represents the layout of the data section. For example, for a closest-hit program, the application gets access to the data associated with the SBT record that was used to invoke that closest-hit program:

Listing 7.6 – Data for closest-hit program

```
struct CHData {
    int meshIdx;   Triangle mesh build input index
    float3 base_color;
};
```

```
CHData* material_info = ( CHData* )optixGetSbtDataPointer();
```

The program is encouraged to rely on the alignment constraints of the SBT data section to read this data efficiently.

8 Shader execution reordering

8.1 Introduction

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*). Consider 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 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 streaming multiprocessor (SM) and memory system specifically targeted at efficient thread reordering.

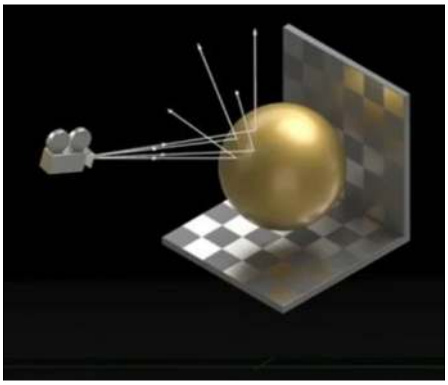

Fig. 8.1 - Rays bounce off an object in different directions, hitting different materials

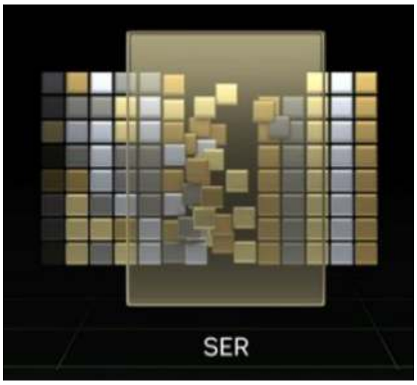

Fig. 8.2 - SER reorders threads, grouping similar work together

Fig. 8.3 - SMs execute shaders with increased coherence

Using SER, performance can be improved up to two times in raytracing regimes of real-world applications, achieved with only a small amount of developer effort. In 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.

8.2 API overview

8.2.1 optixReorder

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

Listing 8.1

```
void optixReorder(
   unsigned int coherenceHint,
   unsigned int numCoherenceHintBitsFromLSB );
```

This function is available in ray-generation shaders. The function does what its name suggests: it asks the system to reorder the calling thread, along with other threads that also call `optixReorder`, across the physical execution units of the GPU. After the function returns, threads that execute together (for example, in the same warp or on the same SM) will be more coherent than before `optixReorder`. Coherence is measured with respect to the argument passed to `optixReorder`, which can be thought of as a sort key. In most cases, that key will be a hit object (described below), which represents the hit location of a ray that was traced into the scene. However, there are also variants of `optixReorder` where “key” an additional coherence hint expressed as a simple number, allowing the function to be used for situations where more information is known about the coherence beyond what is represented in the hit object.

8.2.2 optixReorder and raytracing

To see how `optixReorder` fits into the raytracing programming model, recall that the standard control flow for a ray tracing operation can be depicted as in Figure 8.4.

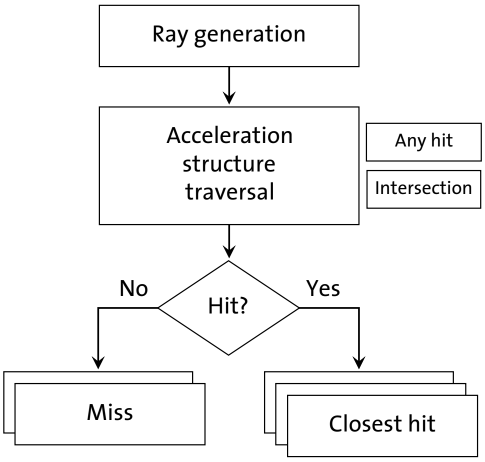

Fig. 8.4 - Control flow of `optixTrace`

The most common and significant source of shading divergence is closest-hit shading, or more generally, any code that performs material or scene-object-specific computations. Ideally, reordering should therefore occur after a ray has been cast (that is, 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 8.5 (page 91).

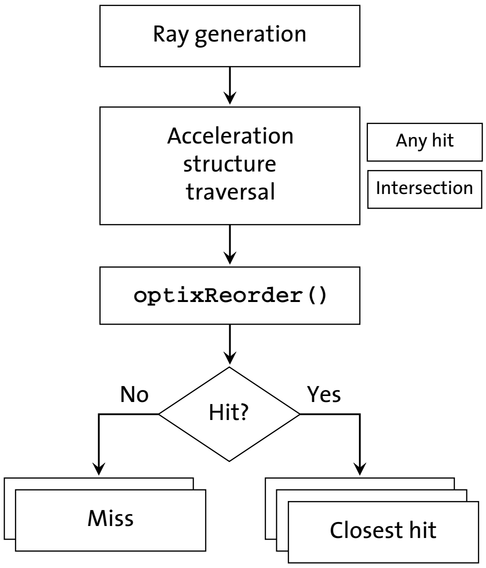

Fig. 8.5 - A good position for reordering is after acceleration structure traversal

Note that the standard APIs do allow systems to perform such reordering "under the hood" but this has a downside: it does not allow an 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 the hit object, combined with `optixReorder`, comes into play.

8.2.3 Hit objects

OptiX does not have an explicit object representation of the data associated with a hit during ray tracing. Instead, functions such as `optixGetRayTmin` and `optixGetInstanceId` return values corresponding to the current hit. Collectively, this data is called the *hit object*. The hit object is available during traversal and shading. It represents either the currently intersected object or the closest intersected object, if any. Several of the hit object values are initialized at the beginning of traversal, while others are set when a hit is accepted by the any hit program.

The hit object's role in the ray-tracing process can be divided into four major phases:

1. *A call to* `optixTrace` — The current hit object is saved if there is one. A new hit object is created.
2. *Traversal* — The hit object is modified and set based on what is intersected and the results of any hit calls.
3. *Shading* — The hit object is available to the closest-hit or miss programs to read.
4. *The return to the caller* — If the previous hit object was saved, it is now restored.

Because there is no explicit representation of the hit object, there can only ever be one hit object active at a time. If a secondary ray is traced during shading, the current hit object will be saved so that it may be restored later.

With the introduction of `optixTraverse` and `optixInvoke`, an application must be able to interact with the hit object that is produced by traversal and consumed by shading. For the purposes of this chapter, the hit object available during traversal and shading will be referred to as the *incoming hit object* while the one available outside of traversal and shading will be referred to as the *outgoing hit object*. Both the incoming and outgoing hit objects can be live at the same time, but there is only ever one of each at a time. Accessor functions for the outgoing hit object are distinguished from the incoming hit object with the inclusion of the string "HitObject" in the device side API functions. For example, `optixGetRayTmin` accesses the incoming hit object, while `optixHitObjectGetRayTmin` accesses the outgoing hit object.

There are three types of hit objects:

- hit
- miss
- nop ("no operation" — neither a hit nor miss will be executed if `optixInvoke` is called)

You can query the type of hit object through the following functions:

- `optixHitObjectIsHit`
- `optixHitObjectIsMiss`
- `optixHitObjectIsNop`

There are several functions that can set the outgoing hit object:

Function	*Produces hit*	*Produces miss*	*Produces nop*
`optixTraverse`	✓	✓	
`optixMakeHitObject`	✓		
`optixMakeMissHitObject`		✓	
`optixMakeNopHitObject`			✓
`optixInvoke`			✓
`optixTrace`			✓

Since there can be only one outgoing hit object at a time, any function that sets it will overwrite the existing hit object.

Shading through the invocation of the outgoing hit object (i.e. calling `optixInvoke`) "consumes" the outgoing hit object and the outgoing hit object is set to nop. You can consider `optixTrace` as effectively a pair of calls to `optixTraverse` and `optixInvoke`, which overwrites the outgoing hit object and subsequently sets it to nop after the invocation of shading.

Note that after certain calls such as `optixTraverse`, the hit object can only be of type hit or miss, so shaders can be written with a single comparison to determine if there is a hit. Listing 8.2 shows such a comparison:

Listing 8.2

```
optixTraverse( ... );
if( optixHitObjectIsHit() )
    printf( "Ray hit" );
else
    printf( "Ray missed" );
```

For a given function's scope, there is always a single outgoing hit object which is initialized to a nop hit object. The function's scope also extends to any inlined functions. In other words, if a function is inlined it does not maintain its own scope but rather becomes part of the caller's scope.

A nop hit object has no data associated with it and queries against it will return zero.

Note that the following hit object accessors will return undefined values when the hit object is a miss:

- `optixHitObjectGetInstanceId`
- `optixHitObjectGetInstanceIndex`
- `optixHitObjectGetPrimitiveIndex`
- `optixHitObjectGetTransformListHandle`
- `optixHitObjectGetSbtGASIndex`
- `optixHitObjectGetHitKind`
- `optixHitObjectGetAttribute_0...7`

After an outgoing hit object is set, the values in the hit object will remain unchanged until a subsequent call creates a new outgoing hit object.

Setting the outgoing hit object will not affect the incoming hit object. For example, calling `optixTraverse` inside a closest-hit shader will not affect the results of the incoming hit object associated with that closest-hit shader. Note that in the regions of a function where the incoming hit object is not used, OptiX does not need to save or restore the components of that hit object, with an improvement in performance as a result.

The role of incoming and outgoing hit objects in function `optixTraverse` and their context in the closest-hit program can be summarized as follows.

```
__ch__
{
  The incoming hit object is set by trace/invoke and is accessible throughout the function.
  float tmax = optixGetRayTmax();   Returns the incoming hit object tmax.

  The outgoing hit object is set to NOP at the start of the function.
  Any access to a NOP-value outgoing hit object returns 0.
  assert( optixHitObjectGetRayTmax() == 0 );

  optixTrace() {
    Actions during the traversal phase:
      Save the incoming hit object.
      Set the incoming hit object.
      Traverse.
    Actions during the invocation phase:
      Invoke shading (closest-hit and miss programs).
      Set the outgoing hit object to NOP.
      Restore the incoming hit object.
  }
  The incoming hit object is preserved across calls to optixTrace.
  assert( optixGetRayTmax() == tmaxIncoming );
  The outgoing hit object is still NOP.
  assert( optixHitObjectGetRayTmax() == 0 );
}
```

When the traversal and shader invocation phases are performed separately with `optixTraverse` and `optixInvoke`, the hit object plays the same role as in `optixTrace`:

```
__ch__
{
  The incoming hit object is set by trace/invoke and is accessible throughout the function.
  float tmax = optixGetRayTmax();   Returns the incoming hit object tmax.

  The outgoing hit object is set to NOP at the start of the function.
  Any access to an outgoing NOP-value hit object returns 0.
  assert( optixHitObjectGetRayTmax() == 0 );

  optixTraverse() {
    Actions taken by the function:
      Save the incoming hit object.
      Set the incoming hit object.
      Traverse.
      Set the outgoing hit object to the results of the traversal.
      Restore the incoming hit object.
  }

  The incoming hit object is preserved across calls to optixTraverse.
  assert( optixGetRayTmax() == tmaxIncoming );
  You can access the outgoing hit object set by optixTraverse.
  float tmaxOutgoing = optixHitObjectGetRayTmax();

  optixInvoke {
    Actions taken by the function:
      Save the incoming hit object.
      Set the incoming hit object from the outgoing hit object.
      Invoke shading (closest-hit and miss programs).
      Set the outgoing hit object to NOP.
      Restore the incoming hit object.
  }

  The incoming hit object is preserved across calls to optixInvoke.
  assert( optixGetRayTmax() == tmaxIncoming );
  The outgoing hit object is set to NOP by calling optixInvoke.
  assert( optixHitObjectGetRayTmax() == 0 );
}
```

8.2.4 Coherence hints

Using the pattern described in the previous section as a basis, an 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 hit object. When numCoherenceHintBitsFromLSB is zero, no user hint bits are used and coherenceHint is ignored.

Here is the full `optixReorder` signature:

Listing 8.3

```
void optixReorder(
    unsigned int coherenceHint,
    unsigned int numCoherenceHintBitsFromLSB );
```

In `optixReorder`, `NumCoherenceBitsFromLSB` specifies the number of bits that `optixReorder` should take into account, starting from the least significant bit.

There is a convenience version of `optixReorder` where the coherenceHint does not need to be specified. It is equivalent to calling `optixReorder` with numCoherenceHintBitsFromLSB and coherenceHint set to zero.

Listing 8.4

```
void optixReorder();
```

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

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

With this key design, an application can use coherence hints to incorporate knowledge about execution divergence into the reordering that isn't already represented in the hit object. One example of this is branches within hit shader code, which are often based on material parameters — whether the material is emissive, has a clear coat layer, casts a shadow ray, etc. An application may inspect 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. A hit object provides a convenient method to read data from the shader table entry that the hit object represents.

Here is an example pattern that uses the hit object and `optixReorder`:

Listing 8.5

```
optixTraverse( sceneIAS, ..., payload );
```

```
unsigned int materialFlags =
    *reinterpret_cast<unsigned int*>(
        optixHitObjectGetSbtDataPointer() );
```

Inspect the material flags encoded as a root constant at offset 0 in the Shader table entry of our hit. Note this requires all closest-hit and miss programs to encode this integer.

```
optixReorder( materialFlags, 4 );
```

Reorder by hit point, using material flags as additional coherence hints. Assumes we have four bits worth of flags.

```
optixInvoke( payload );
```

Invoke shading. Thanks to reordering with coherence hints, this will be coherent both with respect to which shader is executed as well as which path is taken through the shaders.

Coherence hints can also be used to reorder threads based on control flow in the ray-generation shader itself. In the simple loop depicted in Listing 8.6, threads in a warp can be expected to stay on the same iteration without `optixReorder` (for example, first i=0, then i=1). With `optixReorder`, after executing the call, the value of increment variable `i` can be different among threads. This depends on the result of the reordering, which in turn depends on the contents of the hit object. 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.

Listing 8.6

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

    unsigned int coherenceHints = i == 3 ? 1 : 0;
    optixReorder( coherenceHints, 1 );

    ...
}
```

Reorder based on a hit object, while also grouping threads that will exit the loop.

A related situation can be found in path tracers or multi-bounce reflections. In those processes, 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 `optixTrace` is called.

Listing 8.7 is a pseudocode example of shader reordering based on coherence hints:

Listing 8.7

```
while( ... ) {
```

Loop over light bounces

```
    optixTraverse( sceneIAS, ..., payload );
    float albedo = *reinterpret_cast<float*>(
        optixHitObjectGetSbtDataPointer() );
    bool done = russianRoulette( payload, albedo ) ||
                bounceCount >= maxBounces;
    unsigned int coherenceHints = done ? 1 : 0;
```

Before we reorder, figure out if this will be the last loop iteration, and encode that information in a coherence hint bit.

```
    optixReorder( coherenceHints, 1 );
    optixInvoke( payload );
```

Reorder and shade

```
    if( done )
        break;
}
```

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.

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. A confident estimate is sufficient because reordering only affects performance, not correctness, so that an approximate coherence hint is often better than none at all.

8.2.5 More ways to use the hit object

The hit object is the mechanism that enables the splitting of `optixTrace` into two phases and by extension the combination of application-side and system-side knowledge to inform reordering. It is worth noting, however, that hit object is a versatile tool in and of itself even without support for reordering. It allows an 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 the following.

Not executing closest-hit 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 `OPTIX_RAY_FLAG_DISABLE_CLOSESTHIT` ray flag, but using it offers no way to obtain information about the closest hit. The typical solution is to write a trivial closest-hit shader that only fills the payload with the desired hit information. A hit object provides a simpler and more efficient path: one can simply trace the ray using `optixTraverse` and then use state functions to inspect the resulting hit object without ever calling `optixInvoke`. See "The hit object's state" (page 105).

Listing 8.8

```
optixTraverse( sceneIAS, ..., payload );
bool hitDistance = MAX_FLT;
if( optixHitObjectIsHit() )
   hitDistance = optixHitObjectGetRayTmax();
```

Custom indexing into the shader table

The shader table record that `optixTrace` invokes is found using a formula that takes into account a number of factors from various sources, described in "Acceleration structures" (page 83). The current formula provides some amount of flexibility, but it is fixed and imposes certain constraints. A hit object may be modified to reference an arbitrary shader record with `optixHitObjectSetSbtRecordIndex`. This opens new possibilities when it comes to shader table organization. Altering the SBT index can be interesting particularly in applications that only use a single combined any-hit and intersection shader across all objects in the scene. In such cases, all instances can refer to the same shader table range used only to trigger any-hit and intersection shaders, while the closest-hit or miss shader is selected using a manually computed index.

8.3 Best practices

8.3.1 When to use (and when not to use) reordering

The implementation of `optixReorder` is highly optimized but its execution is not without cost. 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.

Furthermore, work following `optixReorder` 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 reduce the performance of reading or writing data indexed by the launch index, which can turn reordering into a net loss when the rest of the workload is particularly cheap.

Reordering is most beneficial in situations that exhibit non-trivial hit shading (irrespective of whether the shading code lives in the ray-generation or closest-hit shader), 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 they access (textures, vertex attributes, environment probes, etc.), the more potential there is for reordering to improve performance. Note, however, that a high shader count isn't necessarily required. Using coherence hints to decrease data 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. For example, 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 the light path is not continued.

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 `optixTrace` calls with `optixTraverse`/`optixReorder`/`optixInvoke` 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.

8.3.2 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 for SER optimization. Warp coherence, that is, the number of threads active in a warp on average, is a good proxy for the general execution coherence improvements that an application will achieve by using `optixReorder`.

8.3.3 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 an `optixReorder` call. The smaller that state is, the lower the overhead of the `optixReorder` 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 an `optixReorder` 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 do not affect performance to begin with.

Applications should target OptiX-IR (page 62) and use CUDA's `__restrict__` keyword wherever possible. OptiX-IR provides more state information to the compiler than PTX, which improves the accuracy of the compiler's live-state analysis. This, in turn, enables additional optimizations and in many cases reduces migrated state. The `__restrict__` keyword also carries additional information in OptiX-IR modules, enabling the compiler to avoid live-state migration caused by function calls. Note that the `__restrict__` keyword improves live-state analysis in OptiX-IR only; it is ignored in PTX. For more information about the role of the `__restrict__` keyword, see the CUDA documentation.[1]

[1] `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=__restrict__#restrict`

8.3.4 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 an 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 trade-off 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.

8.3.5 Tailoring payload types to invoked shaders

When replacing a call to `optixTrace` with its hit object equivalents, it is most straightforward to use the same payload type in both `optixTraverse` and `optixInvoke`. 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 `optixTraverse` (any-hit and intersection) and those invoked from `optixInvoke` (closest-hit and miss) can differ. A frequent example is that any-hit shaders require a smaller payload than closest-hit, because any-hit shaders are only used for simple alpha testing. In cases like these, an application should use different actual payload types — ideally combined with a payload type as described in the API reference. Using different payload types guarantee that register pressure from unnecessary payload fields is avoided as much as possible, which can help increase overall performance. (See "Interaction with payload semantic types" (page 107).)

8.4 API Reference

8.4.1 Querying `optixReorder` behavior

The `optixReorder` API is available on all raytracing-capable GPUs, but its behavior depends on the GPU architecture. In most cases, an application can ignore this fact, and can simply use `optixReorder` as if reordering was always supported. This works because thread reordering affects performance, but not correctness. `optixReorder` 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 `optixReorder`. 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 `optixReorder` can be queried using `optixDeviceContextGetProperty`.

Listing 8.9

```
unsigned int SERFlags = 0;
optixDeviceContextGetProperty(
    s_context,
    OPTIX_DEVICE_PROPERTY_SHADER_EXECUTION_REORDERING,
    &SERFlags, sizeof( unsigned int ) );
if( SERFlags ==
    OPTIX_DEVICE_PROPERTY_SHADER_EXECUTION_REORDERING_FLAG_NONE ) {
    std::cout << "Device does not support SER";
}
```

Check to see if this device can support reordering.

8.4.1.1 optixTraverse

Similar to `optixTrace`, but does not invoke the closest-hit or miss shader. Instead, it overwrites the current outgoing hit object with the results of traversing the ray. The outgoing hit object may be invoked at some later point with `optixInvoke` or used with `optixReorder`. The outgoing hit object can also be queried through various functions such as `optixHitObjectIsHit` or `optixHitObjectGetAttribute_0`.

Listing 8.10

```
template <typename... Payload>
void optixTraverse(
    OptixTraversableHandle handle,
    float3                 rayOrigin,
    float3                 rayDirection,
    float                  tmin,
    float                  tmax,
    float                  rayTime,
    OptixVisibilityMask    visibilityMask,
    unsigned int           rayFlags,
    unsigned int           SBToffset,
    unsigned int           SBTstride,
    unsigned int           missSBTIndex,
    Payload&... payload );
```

Listing 8.11

```
template <typename... Payload>
void optixTraverse(
    OptixPayloadTypeID     type,
    OptixTraversableHandle handle,
    float3                 rayOrigin,
    float3                 rayDirection,
    float                  tmin,
    float                  tmax,
    float                  rayTime,
    OptixVisibilityMask    visibilityMask,
    unsigned int           rayFlags,
    unsigned int           SBToffset,
    unsigned int           SBTstride,
    unsigned int           missSBTIndex,
    Payload&... payload );
```

8.4.1.2 optixMakeHitObject

Constructs an outgoing hit object from the hit object data provided. The traverseData needs to be collected from a previous hit. This hit object will now become the current outgoing hit object and will overwrite the current outgoing hit object.

Listing 8.12

```
template <typename... RegAttributes>
void optixMakeHitObject(
    OptixTraversableHandle        handle,
    float3                        rayOrigin,
    float3                        rayDirection,
    float                         tmin,
    float                         rayTime,
    unsigned int                  rayFlags,
    OptixTraverseData             traverseData,
    const OptixTraversableHandle* transforms,
    unsigned int                  numTransforms );
```

8.4.1.3 optixMakeMissHitObject

Creates a hit object 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.

Listing 8.13

```
void optixMakeMissHitObject(
    unsigned int missSBTIndex,
    float3       rayOrigin,
    float3       rayDirection,
    float        tmin,
    float        tmax,
    float        rayTime,
    unsigned int rayFlags );
```

8.4.1.4 optixMakeNopHitObject

Constructs an outgoing hit object that when invoked with `optixInvoke` does nothing (neither the miss nor the closest-hit shader will be invoked). This hit object will now become the current outgoing hit object and will overwrite the current outgoing hit object. Accessors such as `optixHitObjectGetInstanceId` will return 0 or 0 filled structs. Only `optixHitObjectIsNop` will return a non-zero result.

Listing 8.14

```
void optixMakeNopHitObject();
```

8.4.1.5 optixHitObjectSetSbtRecordIndex

Sets the SBT record index in the current outgoing hit object.

Listing 8.15

```
void optixHitObjectSetSbtRecordIndex( unsigned int sbtRecordIndex );
```

8.4.1.6 optixInvoke

Invokes the closest-hit shader, the miss shader or nop, based on the current outgoing hit object. After execution the current outgoing hit object will be set to nop. An implied nop hit object is always assumed to exist even if there are no calls to `optixTraverse`, `optixMakeHitObject`, `optixMakeMissHitObject`, or `optixMakeNopHitObject`.

Listing 8.16

```
template <typename... Payload>
void optixInvoke( Payload&... payload );
```

Listing 8.17

```
template <typename... Payload>
void optixInvoke( OptixPayloadTypeID type, Payload&... payload );
```

8.4.1.7 The hit object's state

Retrieve information from the current outgoing hit object. An implied nop hit object is always assumed to exist even if there are no calls to `optixTraverse`, `optixMakeHitObject`, `optixMakeMissHitObject`, or `optixMakeNopHitObject`.

Listing 8.18

```
void optixHitObjectGetTraverseData(
    OptixTraverseData* data );
bool optixHitObjectIsMiss();
bool optixHitObjectIsHit();
bool optixHitObjectIsNop();
unsigned int optixHitObjectGetInstanceId();
unsigned int optixHitObjectGetInstanceIndex();
unsigned int optixHitObjectGetPrimitiveIndex();
unsigned int optixHitObjectGetTransformListSize();
OptixTraversableHandle optixHitObjectGetTransformListHandle(
   unsigned int index );
unsigned int optixHitObjectGetSbtGASIndex();
unsigned int optixHitObjectGetHitKind();
float3 optixHitObjectGetWorldRayOrigin();
float3 optixHitObjectGetWorldRayDirection();
float optixHitObjectGetRayTmin();
float optixHitObjectGetRayTmax();
float optixHitObjectGetRayTime();
unsigned int optixHitObjectGetRayFlags();
unsigned int optixHitObjectGetAttribute_0();
unsigned int optixHitObjectGetAttribute_1();
unsigned int optixHitObjectGetAttribute_2();
unsigned int optixHitObjectGetAttribute_3();
unsigned int optixHitObjectGetAttribute_4();
unsigned int optixHitObjectGetAttribute_5();
unsigned int optixHitObjectGetAttribute_6();
unsigned int optixHitObjectGetAttribute_7();
unsigned int optixHitObjectGetSbtRecordIndex();
CUdeviceptr optixHitObjectGetSbtDataPointer();
OptixTraversableHandle optixHitObjectGetGASTraversableHandle();
void optixHitObjectGetTriangleVertexData( float3 data[3] );
void optixHitObjectGetLinearCurveVertexData( float4 data[2] );
void optixHitObjectGetQuadraticBSplineVertexData( float4 data[3] );
void optixHitObjectGetQuadraticBSplineRocapsVertexData( float4 data[3] );
void optixHitObjectGetCubicBSplineVertexData( float4 data[4] );
void optixHitObjectGetCubicBSplineRocapsVertexData( float4 data[4] );
void optixHitObjectGetCatmullRomVertexData( float4 data[4] );
void optixHitObjectGetCatmullRomRocapsVertexData( float4 data[4] );
void optixHitObjectGetCubicBezierVertexData( float4 data[4] );
void optixHitObjectGetCubicBezierRocapsVertexData( float4 data[4] );
void optixHitObjectGetRibbonVertexData( float4 data[3] );
float3 optixHitObjectGetRibbonNormal( float2 ribbonParameters );
void optixHitObjectGetSphereData( float4 data[1] );
void optixHitObjectGetWorldToObjectTransformMatrix( float m[12] );
void optixHitObjectGetObjectToWorldTransformMatrix( float m[12] );
float3 optixHitObjectTransformPointFromWorldToObjectSpace( float3 point );
float3 optixHitObjectTransformVectorFromWorldToObjectSpace( float3 vec );
float3 optixHitObjectTransformNormalFromWorldToObjectSpace( float3 normal );
float3 optixHitObjectTransformPointFromObjectToWorldSpace( float3 point );
float3 optixHitObjectTransformVectorFromObjectToWorldSpace( float3 vec );
float3 optixHitObjectTransformNormalFromObjectToWorldSpace( float3 normal );
unsigned int optixHitObjectGetClusterId();
float2 optixHitObjectGetTriangleBarycentrics();
float optixHitObjectGetCurveParameter();
float2 optixHitObjectGetRibbonParameters();
```

8.4.2 optixReorder

Reorders threads based on the current outgoing hit object and a coherence hint value. Parameter `NumCoherenceHintBits` indicates how many of the least significant bits of `CoherenceHint` should be considered during reordering, with a maximum of 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`.

Listing 8.19

```
void optixReorder(
    unsigned int coherenceHint,
    unsigned int numCoherenceHintBits );
```

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

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

This ordering implies that `optixReorder` will first attempt to group threads whose hit object references the same shader ID. (Miss shaders and nop hit objects are grouped separately). Within each of these groups, it will attempt to order threads by the value of their coherence hints. Within ranges of equal coherence hints, it will also attempt to maximize locality in 3D space of the ray hit (if any).

You can use optixReorder without arguments as a convenience function that supplies 0 for numCoherenceHintBits thereby ignoring any values in coherenceHint. Only information in the current outgoing hit object will be considered when reordering.

Listing 8.20

```
void optixReorder();
```

8.4.2.1 Interaction with payload semantic types

Payload semantic types describe how information flows between shader stages in optixTrace. Figure 8.6 shows this information flow:

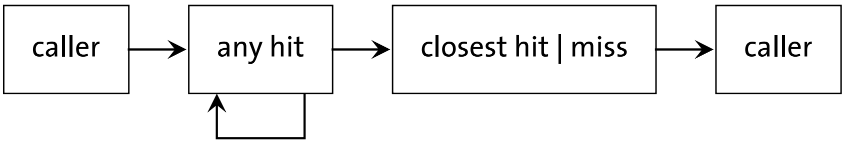

Fig. 8.6 - Information flow between optixTrace shader stages

With the hit object, control is returned to the caller after optixTrace has finished. Therefore, [caller] is inserted between [any hit] and [closest hit | miss]:

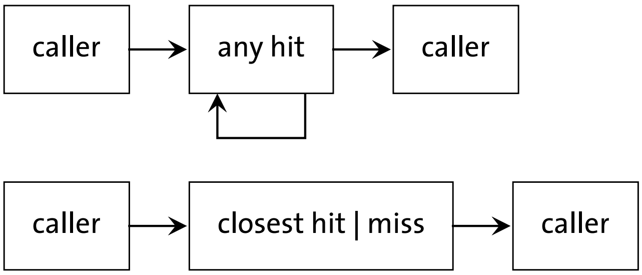

Fig. 8.7 - Information flow change with hit object

To allow interchangeability of payload types between optixTrace and the optixTraverse/optixInvoke execution model, the following additional rules apply:

- At the call to optixTraverse, any field declared as OPTIX_PAYLOAD_SEMANTICS_MS_READ or OPTIX_PAYLOAD_SEMANTICS_CH_READ is treated as OPTIX_PAYLOAD_SEMANTICS_TRACE_CALLER_READ .
- At the call to optixInvoke, any field declared as OPTIX_PAYLOAD_SEMANTICS_AH_WRITE is treated as OPTIX_PAYLOAD_SEMANTICS_TRACE_CALLER_WRITE .

Note, these rules equally apply to the READ_WRITE flags.

9 Curves and spheres

Because curves and spheres are commonly used primitives, OptiX provides a set of built-in intersectors for these primitives, and users do not have to provide their own. Starting with the Nvida Blackwell architecture, spheres and linear curves are hardware accelerated. These primitives are primarily intended for things like particles, hair, and fur.

9.1 Differences between curves, spheres, and triangles

Ray tracing curves or spheres with NVIDIA OptiX is similar to the procedure for ray tracing triangles; see "Curve build inputs" (page 25) . The differences between curves, spheres, and triangles include the following:

- In the triangle build input, the index buffer is optional. In the curves build input, it is mandatory. A spheres build input does not take an index buffer.
- Each curves build input references just a single SBT record. Unlike triangles or spheres, there is no per-primitive SBT index. It is still possible to use multiple materials for curves in the same BVH, by using multiple build inputs, one per material. (Because there is only one SBT record, the `OptixGeometryFlags` are specified by only one int, rather than an array of ints.)
- There is no `preTransform` field for curves or spheres. If there were, a nonuniform scale or shear transformation would yield different results as a pre-transform than as an instance transform. Nonuniform instance transforms of curves create elliptical cross sections, while `preTransform` would still have a circular cross section.
- Each shader binding table record for curves requires a hit group that uses a built-in intersection program for curves. In the `OptixProgramGroupHitgroup`, you must use a module returned by `optixBuiltinISModuleGet()` for `moduleIS`, and `nullptr` for `entryFunctionNameIS`. This also applies to spheres.
- Curves or spheres must be explicitly enabled in the pipeline to be rendered; triangles and custom primitives are enabled by default. This is enabled in the `OptixPipelineCompileOptions::usesPrimitiveTypeFlags` by setting the relevant bits from `OptixPrimitiveTypeFlags`.

Listing 9.1

```
OptixPipelineCompileOptions pipelineCompileOptions = {};
...
pipelineCompileOptions.usesPrimitiveTypeFlags =
    OPTIX_PRIMITIVE_TYPE_FLAGS_TRIANGLE |
    OPTIX_PRIMITIVE_TYPE_FLAGS_CUSTOM |
    OPTIX_PRIMITIVE_TYPE_FLAGS_ROUND_LINEAR |
    OPTIX_PRIMITIVE_TYPE_FLAGS_ROUND_QUADRATIC_BSPLINE |
    OPTIX_PRIMITIVE_TYPE_FLAGS_ROUND_QUADRATIC_BSPLINE_ROCAPS |
    OPTIX_PRIMITIVE_TYPE_FLAGS_FLAT_QUADRATIC_BSPLINE |
    OPTIX_PRIMITIVE_TYPE_FLAGS_ROUND_CUBIC_BSPLINE |
    OPTIX_PRIMITIVE_TYPE_FLAGS_ROUND_CUBIC_BSPLINE_ROCAPS |
    OPTIX_PRIMITIVE_TYPE_FLAGS_ROUND_CATMULLROM |
    OPTIX_PRIMITIVE_TYPE_FLAGS_ROUND_CATMULLROM_ROCAPS |
    OPTIX_PRIMITIVE_TYPE_FLAGS_ROUND_CUBIC_BEZIER |
    OPTIX_PRIMITIVE_TYPE_FLAGS_ROUND_CUBIC_BEZIER_ROCAPS |
    OPTIX_PRIMITIVE_TYPE_FLAGS_SPHERE;
```

- To render motion blur for any primitive, motion blur must be enabled in the pipeline. But for curves or spheres, motion blur must also be enabled in the built-in intersection program, by setting the `OptixBuiltinISOptions::usesMotionBlur` flag. (Note this flag should only be set to true when using vertex motion blur, not when using motion transform blur.) The values of the `OptixBuiltinISOptions::buildFlags` must also match the corresponding build flags in `OptixAccelBuildOptions::buildFlags` that were used for building the acceleration structure, and `OptixBuiltinISOptions::curveEndcapFlags` must match `OptixBuildInputCurveArray::endcapFlags`. Note that motion blur for spheres and linear curves is not hardware accelerated, and uses a software fallback instead, so be aware that enabling motion blur with those primitives may have additional performance implications.

9.2 Splitting curve segments

NVIDIA OptiX can split curve segments into multiple sub-segments, and bound the sub-segments separately. This gives faster performance but costs more memory. Splitting can be controlled via `OptixBuildFlags`, using `OPTIX_BUILD_FLAG_PREFER_FAST_TRACE` for a splitting factor higher than default, and `OPTIX_BUILD_FLAG_PREFER_FAST_BUILD` for a lower splitting factor.

> **Note:** Splitting means that the same primitive (the same curve segment) can be hit multiple times by a ray. Geometry with `OPTIX_GEOMETRY_FLAG_REQUIRE_SINGLE_ANYHIT_CALL` set is not split; in this case, each segment can only be hit once, but a longer curve strand composed of multiple segments can still be hit more than once.

9.3 Curves and the hit program

In hit programs, non-ribbon curve hits provide a single attribute: the curve parameter (“u”) within the segment corresponding to the intersection, returned by `optixGetCurveParameter()`. If end caps are enabled, the “u” parameter returned is exactly 0.0 or 1.0 when and only when an end cap is hit. For ribbon hits, two attributes are provided by `optixGetRibbonParameters()`. In addition to parameter “u” along the curve axis, parameter “v” along the width is returned. The range of parameter “v” is -1.0 to 1.0. As with all hit programs, `optixGetRayTmax()` returns the ray parameter (“t”), from which the hit point can be computed, and `optixGetPrimitiveIndex()` returns the segment’s primitive index. To maximize performance, no other geometry attributes are passed or precomputed. Instead, the program must compute whatever curve geometry it requires using the vertex data. For ribbons, you can compute the geometric normal by passing in the primitive index and the ribbon parameters to `optixGetRibbonNormal`().

```
optixGetRibbonNormal
```

To make vertex data available to your OptiX shaders (for example your closest-hit program), the geometry acceleration structure build flag must include `OPTIX_BUILD_FLAG_ALLOW_RANDOM_VERTEX_ACCESS`. (See "Build flags" (page 28).) To fetch the vertex positions and radius values for the segment, pass the primitive index to the function appropriate for the curve type:

```
optixGetLinearCurveVertexDataFromHandle
optixGetQuadraticBSplineVertexDataFromHandle
optixGetQuadraticBSplineRocapsVertexDataFromHandle
optixGetCubicBSplineVertexDataFromHandle
optixGetCubicBSplineRocapsVertexDataFromHandle
optixGetCatmullRomVertexDataFromHandle
optixGetCatmullRomRocapsVertexDataFromHandle
optixGetCubicBezierVertexDataFromHandle
optixGetCubicBezierRocapsVertexDataFromHandle
optixGetRibbonVertexDataFromHandle
```

If only the vertex data of a currently intersected curve is required, it is recommended to use functions without 'FromHandle'. For example, `optixGetCubicBSplineRocapsVertexData` should be used instead of `optixGetCubicBSplineRocapsVertexDataFromHandle`. This data fetch of the currently intersected curve does not require building the corresponding geometry acceleration structure with flag `OPTIX_BUILD_FLAG_ALLOW_RANDOM_VERTEX_ACCESS`.

Sample code is provided to interpolate the curve points and radii, and to compute tangents, normals, and derivatives; see header file `SDK/cuda/curve.h`, for Rocaps normal computation see `optixHair.cu` in the `optixHair` code sample. A single hit program can (and should) handle multiple curve primitive types by checking the value of `optixGetPrimitiveType`(). For example, see `optixHair.cu` .

The `optixGetCurveParameter` and `optixGetRibbonParameters` functions return the "u" parameter value relative to a single polynomial segment of the curve. If needed, the hit program can map this to the parameter value relative to the entire multi-segment strand. For instance, this can be used to interpolate between two colors specified at the root and tip of a hair. This segment-to-strand mapping is the application's responsibility; see the `optixHair` code sample for an example implementation.

9.4 Spheres and the hit program

Hit programs for spheres report the front face intersection of a ray with a sphere. The function `optixGetRayTmax`() returns the ray parameter ("t"), from which the hit point can be computed. The function `optixGetPrimitiveIndex`() returns the sphere's primitive index. No geometry attributes are passed or precomputed. Instead, the program must compute whatever sphere geometry it requires (surface normals, etc.) using the vertex data.

To make vertex data available to OptiX shaders (for example the closest-hit program), the build flag of the geometry acceleration structure must include `OPTIX_BUILD_FLAG_ALLOW_RANDOM_VERTEX_ACCESS`. (See "Build flags" (page 28).) By including this flag, the function `optixGetSphereDataFromHandle` can acquire the vertex positions and radius values for the sphere. Similar to curves, if only the vertex data of a currently intersected sphere is used, it is recommended to use the function without 'FromHandle'. This does not require build flag `OPTIX_BUILD_FLAG_ALLOW_RANDOM_VERTEX_ACCESS`.

9.5 Interpolating curve endpoints

A B-spline curve typically does not interpolate (that is, does not touch) its control points. In particular, in contrast to a Bézier curve, it does not reach as far as its first and last control points:

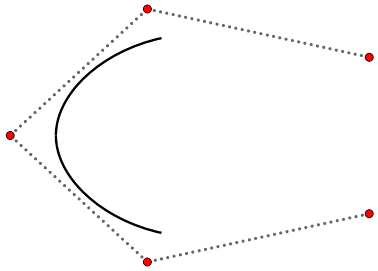

Fig. 9.1 - The curve does not reach the first and last control points

If desired, the application can modify the control point sequence to interpolate these points, by adding an additional control point at each end:

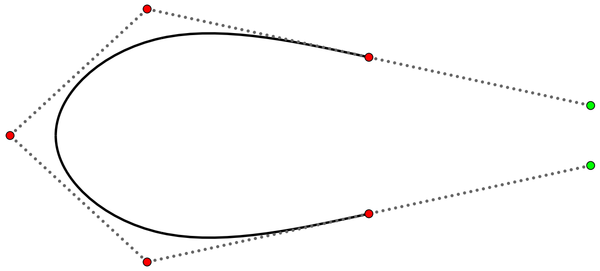

Fig. 9.2 - Adding points (in green) to extend the curve

Following the Universal Scene Description documentation,[1] let's call these "phantom" points, resulting in "pinned" curves. A phantom point is constructed by reflecting through the endpoint the point that preceded it. Given control points $p_1, p_2 ... p_n$, the phantom points can be defined geometrically by:

$$p_0 = p_1 + (p_1 - p_2)$$

$$p_{n+1} = p_n + (p_n - p_{n-1})$$

[1] `https://graphics.pixar.com/usd/docs/api/class_usd_geom_basis_curves.html#UsdGeomBasisCurves_Segment`

It is also possible to interpolate endpoints by repeating them. For cubic B-spline curves, a control point that is repeated 3 times will be interpolated. OptiX allows this, but the phantom point method is preferred, as it is numerically more stable, and permits control of curvature.

Note: The additional points must be added to the build input by the application. These points are not added by NVIDIA OptiX itself.

9.6 Back-face culling

A sphere primitive or the tube of a curve primitive is considered hollow, and back faces are culled. Thus if a ray starts inside a sphere primitive or a curve primitive (inside the tube), it will not hit that primitive. This is convenient for secondary rays and for transparency. However, see “Limitations” (page 113) below.

9.7 Rocaps curves

Rocaps (roving capsules) has been introduced for faster rendering of cubic and round quadratic curves. It is usually faster for non-straight curves and especially for lower splitting factors.

The Rocaps type variants use the same input representation as the non-Rocaps curve types (called Phantom). Rocaps and Phantom compute intersections with slightly different surfaces. Phantom uses a swept circle whereas Rocaps uses a swept sphere. The swept circle for Phantom is defined by the set of circles in the plane orthogonal to the curves at each point. The reported surface intersection for Rocaps will be on the surface of the union if all 3D spheres defined for each point on the curve.

The difference in shape between swept-sphere and swept-circle surfaces varies based on curvature and relative thickness (radius-length ratio). For constant-width curves the surfaces coincide. The difference between the shapes gets larger when curvature and thickness increase. Practically speaking for curves used to represent hair and fur (thin low curvature geometry), the shapes of Phantom and Rocaps are interchangeable.

Surface normals of Rocaps and Phantom need to be computed differently. For Rocaps, besides at flat end caps the surface normal is simply the vector from the point on the curve at curve parameter “u” to the hit point on the surface. `optixHair.cu` in the `optixHair` code sample describes the normal computation, see also “Limitations” (page 113) below.

9.8 Limitations

There are some caveats to be aware of when using curves and curve hit programs.

Multiple hits

For a ray that intersects a single curve segment more than once, NVIDIA OptiX does not guarantee that all intersections will be reported as hits. In particular, the any-hit program will in general not be called more than once for a given segment. To implement transparency, the alternative approach to using an any-hit program is to use closest-hit and relaunch a continuation ray from each hit.

Back-face culling

While the tube is mostly hollow, in the current implementation a ray can hit an internal endcap, meaning an endcap between two segments of a strand. For typical curves that are much longer than they are wide, this is often not noticeable. To avoid hitting internal endcaps, adjust secondary rays to launch from outside the tube.

Triple control points

As noted in "Interpolating curve endpoints" (page 112), duplicating control points is possible in NVIDIA OptiX, but the use of phantom points is preferred. Duplicating the ending control points is numerically challenging (the derivative is zero) and the first segment is forced to be straight.

Convoluted cases

Curves for hair, fur, cloth fibers, etc., are typically thin with relatively gentle curvature. It is possible to construct curve segments with tight curvature (relative to width), self-intersections, or rapidly varying radii that will exhibit artifacts, particularly if rendered up close. In order to avoid precision problems with curves it is recommended to keep the ratio of length to average width of the curve below 100 and to avoid cases where the radius of curvature is smaller than the radius of the curve. In many cases, these issues can be remedied by splitting the segment into smaller segments.

Ribbon basis type and representation

Ribbons are oriented curves, where the basis type is restricted to uniform quadratic B-splines. As described in "Curve build inputs" (page 25), it is assumed that the segments of a ribbon strand have overlapping control points. The first index of a ribbon build input should be 0; there are no unused control points at the beginning of the vertex data.

Ribbons with user-specified normals (orientations)

Ribbons can be used without any specified orientations. The orientation is computed automatically from the shape of the curve axis, taking its tangent along the strand into account. For many applications, such as modeling grass, this computation produces the intended orientations and should be the preferred method. Alternatively, normals can be provided for orienting the ribbons. The normals are not control points of a quadratic B-spline, but are linearly interpolated along the segment. The normals need to fit the curve geometry sufficiently, otherwise the connections between segments may not be smooth. Non-smooth segment borders may also show up for straight ribbon strands. These can be addressed by slightly perturbing the control points of the strand.

Accuracy of Rocaps curve parameter

For improving the efficiency of ray tracing Rocaps computes the ray parameter ("t") of the intersection with higher accuracy than the curve parameter ("u"). If the found curve parameter is used directly, to compute the normal for example, higher accuracy might be necessary. When using it directly for normal computation it can introduce visible artifacts on rapidly varying surfaces. If a high accuracy parameter is desired, it can be optionally computed in a hit shader. Example code is provided in `optixCurves.cu` in the `optixCurves` code sample.

Negative radii

Rocaps does not support negative radii, curve segments with negative radius will be skipped.

Rocaps with rays originating inside the surface

Both, Rocaps and Phantom will not intersect the surface if the ray origin lies inside the curve. In contrast to Phantom, rocaps will also not intersect the surface if the ray origin lies inside the spherical caps of the curve ends, even if the end caps are specified to be omitted or flat.

10 Ray generation launches

The API function described in this section is:

`optixLaunch`

A ray generation launch is the primary workhorse of the NVIDIA OptiX API. A launch invokes a 1D, 2D or 3D array of threads on the device and invokes ray generation programs for each thread. When the ray generation program invokes `optixTrace`, other programs are invoked to execute traversal, intersection, any-hit, closest-hit, miss and exception programs until the invocations are complete.

A pipeline requires device-side memory for each launch. This space is allocated and managed by the API. Because launch resources may be shared between pipelines, they are only guaranteed to be freed when the `OptixDeviceContext` is destroyed.

To initiate a pipeline launch, use the `optixLaunch` function. All launches are asynchronous, using CUDA streams. When it is necessary to implement synchronization, use the mechanisms provided by CUDA streams and events.

In addition to the pipeline object, the CUDA stream, and the launch state, it is necessary to provide information about the SBT layout, including: This includes:

- The base addresses for sections of the SBT that hold the records of different types
- The stride, in bytes, along with the maximum valid index for arrays of SBT records. The stride is used to calculate the SBT address for a record based on a given index. (See "Layout" (page 82).)

The value of the pipeline launch parameter is specified by the `pipelineLaunchParamsVariableName` field of the `OptixPipelineCompileOptions` struct. It is determined at launch with a CUdeviceptr parameter, named `pipelineParams`, that is provided to `optixLaunch`. Note the following restrictions:

- If the size specified by the `pipelineParamsSize` argument of `optixLaunch` is smaller than the size of the variable specified by the modules, the non-overlapping values of the parameter will be undefined.
- If the size is larger, an error will occur.

(See "Pipeline launch parameter" (page 69).)

The kernel creates a copy of `pipelineParams` before the launch, so the kernel is allowed to modify `pipelineParams` values during the launch. This means that subsequent launches can run with modified pipeline parameter values. Users cannot synchronize with this copy between the invocation of `optixLaunch` and the start of the kernel.

> **Note:** Concurrent launches with different values for `pipelineParams` in the same pipeline triggers serialization of the launches. Concurrency requires a separate pipeline for each concurrent launch.

The dimensions of a launch must also be specified. If one-dimensional launches are required, use the width as the dimension of the launch and set both a height and a depth of 1. If two-dimensional launches are required, set the width and the height as the dimension of the launch and set a depth of 1.

Dimension	*Width*	*Height*	*Depth*
1D	*width*	1	1
2D	*width*	*height*	1
3D	*width*	*height*	*depth*

Specifying different dimensions for a launch

For example:

Listing 10.1

```
CUstream stream = nullptr;
cuStreamCreate( &stream );
CUdeviceptr raygenRecord, hitgroupRecords;

...          Generate acceleration structures and SBT records

unsigned int width = ...;
unsigned int height = ...;
unsigned int depth = ...;
OptixShaderBindingTable sbt = {};
sbt.raygenRecord = raygenRecord;
sbt.hitgroupRecords = hitgroupRecords;
sbt.hitgroupRecordStrideInBytes = sizeof( HitGroupRecord );
sbt.hitgroupRecordCount = numHitGroupRecords;
MyPipelineParams pipelineParams = ...;
CUdeviceptr d_pipelineParams = 0;

...          Allocate and copy the params to the device

optixLaunch( pipeline, stream,
    d_pipelineParams, sizeof( MyPipelineParams ),
    &sbt, width, height, depth );
```

11 Neural Rendering with Cooperative Vectors

Cooperative vectors are an OptiX data type and API for high-performance machine learning computations during OptiX shader programs. They facilitate use of the Tensor Cores on NVIDIA RTX GPUs. Tensor Core operations require coordination between the threads in a warp. Cooperative vectors are designed to enable this warp-wide work within OptiX's single-threaded programming model.

Support for cooperative vectors extends to all RTX devices along with a selection of server class GPUs. Support can be queried using `optixDeviceContextGetProperty` with `OPTIX_DEVICE_PROPERTY_COOP_VEC`. Using cooperative vectors on devices that do not support it will generate errors; no fallback support is available.

The basic building block of many neural network algorithms, such as multi-layer perceptrons (MLPs), is a matrix-vector multiply followed by addition of a bias value. In the cooperative vector API this is done with an intrinsic called `optixCoopVecMatMul`. It performs the following operation:

Listing 11.1

```
matrix * inputVector + bias = output
 [NxK]     [Kx1]      [Nx1] = [Nx1]
```

Cooperative vectors are designed for cases where the matrices are relatively small. While the OptiX intrinsics support a wide range of matrix dimensions, they are most optimized for sizes up to 64 elements. They also support a range of element types, from int8 up to fp32. The sizes of the weights matrix and activation vectors have a direct effect on the register pressure, which can significantly impact performance.

While neural network design is beyond the scope of this programming guide, we describe here how to implement the common algorithms.

The input and output of `optixCoopVecMatMul` are vectors. OptiX provides a templated C++ vector class called `OptixCoopVec`, found in `optix_device.h`. `OptixCoopVec` is templated with both the element type and the number of elements in the vector. Here is a declaration of a 20-element vector of type half (16-bit floating point).

Listing 11.2

```
OptixCoopVec<half, 20> vec;
```

In addition to `optixCoopVecMatMul`, there are a variety of device functions that operate on vectors, which are useful for computing activation functions between successive calls to `optixCoopVecMatMul`. For example, the operator for component-wise multiplication of two vectors is declared as:

Listing 11.3

```
template <typename VecT>
__device__ VecT optixCoopVecMul( const VecT& vecA, const VecT& vecB );
```

To implement a ReLU activation, a maximum intrinsic can be used to clamp negative values to zero:

Listing 11.4

```
OptixCoopVec<half, 16> a = optixCoopVecMax( vec, half( 0.0f ) );
```

There are several more operators found in `optix_device.h`.

The call to `optixCoopVecMatMul` has several template and standard parameters that describe the input and output data along with its layout.

Listing 11.5

```
using InputVecType = OptixCoopVec<float, 16>;
using OutputVecType = OptixCoopVec<int, 4>;
InputVecType inputLayer = generateInputs();
OutputVecType outputLayer = optixCoopVecMatMul<
    OutputVecType,  output vector type

    InputVecType,  input vector type

    OPTIX_COOP_VEC_ELEM_TYPE_INT8,  input interpretation

    OPTIX_COOP_VEC_MATRIX_LAYOUT_COLUMN_MAJOR,  matrix layout

    false,  transpose

    OutputVecType::size,  matrix size N

    InputVecType::size,  matrix size K

    OPTIX_COOP_VEC_ELEM_TYPE_INT8,  matrix element type

    OPTIX_COOP_VEC_ELEM_TYPE_INT32  bias element type
    >( inputVector, matrix, /*matrixOffset*/ 0, /*bias*/ nullptr,
       /*biasOffset*/ 0 );
```

The input and output vector types are specified as part of the template parameters. This is to allow OptiX to efficiently extract the vector components. This is also why the matrix sizes (N and K) are passed as template parameters. For now, N and K must match the sizes of the input and output vector types.

When the element type of the input vector does not match the input interpretation, casting is done. In the above example the 32-bit floating point input value is cast to an eight-bit signed integer before being multiplied by the matrix. In the current implementation the input interpretation must match the matrix element type.

The bias type needs to match the type of the output vector as no casting or changing of types are supported at this stage of the operation. The output type is based on the matrix type. The following type combinations are supported.

inputType	*inputInterpretation*	*matrixElementType*	*biasElementType*	*outputType*
FLOAT16	FLOAT16	FLOAT16	FLOAT16	FLOAT16
FLOAT16	FLOAT8_E4M3	FLOAT8_E4M3	FLOAT16	FLOAT16
FLOAT16	FLOAT8_E5M4	FLOAT8_E5M4	FLOAT16	FLOAT16
FLOAT16	UINT8/INT8	UINT8/INT8	UINT32/INT32	UINT32/INT32
FLOAT32	UINT8/INT8	UINT8/INT8	UINT32/INT32	UINT32/INT32
UINT8/INT8	UINT8/INT8	UINT8/INT8	UINT32/INT32	UINT32/INT32

Note that for eight-bit integer types, the matrix multiply will be signed if either the inputInterpretation or the matrixElementType is signed. If either the matrix multiply or the biasElementType is signed, the outputType must be signed.

The layout of the matrix is specified as a template parameter for OptiX to be able to efficiently load the matrix data into Tensor Core layout. See "Memory Layout" (page 122) below.

The transpose flag indicates the matrix data is stored transposed from the specified matrix dimensions. This is intended for neural network training, and is allowed only when the matrix element type is `FLOAT16` and the matrix layout is one of the two optimal types. See "Training with OptiX Cooperative Vectors" (page 124) for more details.

The standard parameters specify the input vector, a pointer to the matrix stored in global memory, and optionally a pointer to the bias stored in global memory. There are two additional parameters for specifying offsets from the matrix and bias pointers. These offsets are designed for successive calls to `optixCoopVecMatMul` where the matrices and biases are stored in the same arrays.

11.1 Performance Considerations

When evaluating a neural network, it is important to restrict operations between successive `optixCoopVecMatMul` calls to other `optixCoopVec` intrinsics. To make use of the Tensor Cores when calling `optixCoopVecMatMul`, per-thread data is shuffled across the warp before the call and then unshuffled afterwards. When only supported vector operations are found between calls to `optixCoopVecMatMul`, the shuffling and unshuffling can be removed, improving performance. If a program needs to perform operations on the vector that are not supported by the set of intrinsics, it can do so with the caveat that there will be increased data movement.

While full warps are not required to call `optixCoopVecMatMul`, performance will be best when using full warps. Use SER ("Shader execution reordering" (page 89)) to coalesce threads, and avoid calling `optixCoopVecMatMul` inside dynamic conditionals. As an example, using SER to sort the rays based on whether something is hit or not can improve performance because it gathers all the threads that call the closest-hit program into coherent warps. Also, if there is a conditional call to `optixCoopVecMatMul`, making the call unconditional with dummy input and later ignoring the output can improve performance.

When issuing successive calls to `optixCoopVecMatMul`, the weights should be stored in the same global memory buffer for best performance. The pointer argument should remain the same and the offset to each matrix changed. When the offset can be a compile-time constant, performance can also be improved. The device intrinsic `optixCoopVecGetMatrixSize` will calculate the size of the matrix at compile time. There is a corresponding host side function, `optixCoopVecMatrixComputeSize`, that can be used to pack the matrices into device memory.

Experimentation with the maximum register count can yield better performance. `optixCoopVecMatMul` can take a lot of registers, especially for matrices of dimensions 64 and larger.

11.2 Memory Layout

The layout of weight matrices is defined with `OptixCoopVecMatrixLayout`. In addition to the familiar row-major and column-major layouts, two optimized formats are supported. These layouts are optimized to take best advantage of the Tensor Cores. There are two optimal layouts, `OPTIX_COOP_VEC_MATRIX_LAYOUT_INFERENCING_OPTIMAL` for use during inferencing and `OPTIX_COOP_VEC_MATRIX_LAYOUT_TRAINING_OPTIMAL` for use during training. The principal difference between inference and training layouts is that inference layouts may create multiple copies of matrix elements to improve performance, while the training layout avoids duplicate elements. The training layout can be used for both forward and backward propagation. The inference and training layouts are opaque and may be unique to each driver and hardware architecture. While `optixCoopVecMatMul` can support matrices in all layouts, some options require the matrices to be in optimal layouts. For example, transpose is only supported for optimal layout matrices.

The host function `optixCoopVecMatrixConvert` converts matrices between layouts and types. Data should be stored outside of the API in row- or column-major and converted into the optimal layouts using `optixCoopVecMatrixConvert`. Type conversion to `FLOAT8` types can also be done at this time, though `FLOAT8` types can also be produced outside of the OptiX API and imported from row- or column-major layouts.

The `optixCoopVecMatrixConvert` function is designed to convert an array of networks with the same topology. The network topology is specified with the `OptixNetworkDescription` struct, which lists the layout, size, and element type of each matrix in the network. Conversion is performed on the device, with both the source and destination specified by device pointers. Because the optimal layouts are opaque and specific to device and driver, the utility function `optixCoopVecMatrixComputeSize` is provided to calculate the sizes of matrices in OptiX host code.

Here is an example of host code to fill in a three-layer MLP with the identity matrix. The data is filled in using column-major layout (`srcLayer`), then converted to inferencing-optimal layout (`dstLayer`). The size of each matrix is computed with `optixCoopVecMatrixComputeSize`. Tightly packing matrices with this size is useful, because on the device `optixCoopVecGetMatrixSize` can be used to adjust the offset into the matrix when calling `optixCoopVecMatMul` for successive layers.

Once the conversion is performed, the source matrices are no longer needed, and are deleted.

Listing 11.6

```
// number of channels (activations) between each layer
std::vector<int> channels =
    {INPUT_SIZE, HIDDEN_SIZE, HIDDEN_SIZE, OUTPUT_SIZE};
int n_layers  = (int) channels.size() - 1;

int num_weights = 0;
for( int i = 0; i < n_layers; i++ )
    num_weights += channels[i] * channels[i + 1];

std::vector<half> w(num_weights);
int w_idx = 0;
size_t dst_size = 0;

using MD = std::vector<OptixCoopVecMatrixDescription>;
MD srcLayerDesc( n_layers, OptixCoopVecMatrixDescription{} );
MD dstLayerDesc( n_layers, OptixCoopVecMatrixDescription{} );

for( int i = 0; i < n_layers; i++ )
{
    const int K = channels[i];      // num input channels
    const int N = channels[i + 1];  // num output channels

    for( int j = 0; j < K; ++j )
        for( int i = 0; i < N; ++i )
        {
            // non-square identity matrix
            w[w_idx + j*N + i] = (i%K) == j ? half( 1.f ) : half( 0.f );
        }

    OptixCoopVecMatrixLayout optMatrixLayout =
        OPTIX_COOP_VEC_MATRIX_LAYOUT_TRAINING_OPTIMAL;
    unsigned int             optRowColumnStrideInBytes = 0;
    size_t                   optMatrixDataSize         = 0;
    optixCoopVecMatrixComputeSize(
        state.context, N, K, OPTIX_COOP_VEC_ELEM_TYPE_FLOAT16,
        optMatrixLayout, optRowColumnStrideInBytes, &optMatrixDataSize );
    OptixCoopVecMatrixDescription& srcLayer = srcLayerDesc[i];
    OptixCoopVecMatrixDescription& dstLayer = dstLayerDesc[i];
    srcLayer.N = dstLayer.N = N;
    srcLayer.K = dstLayer.K = K;
    srcLayer.offsetInBytes  = w_idx * sizeof( half );
    dstLayer.offsetInBytes  = dst_size;
    srcLayer.elementType    = OPTIX_COOP_VEC_ELEM_TYPE_FLOAT16;
    dstLayer.elementType    = OPTIX_COOP_VEC_ELEM_TYPE_FLOAT16;
    srcLayer.layout         = OPTIX_COOP_VEC_MATRIX_LAYOUT_COLUMN_MAJOR;
    dstLayer.layout         = optMatrixLayout;
    srcLayer.rowColumnStrideInBytes = N * sizeof( half );
    dstLayer.rowColumnStrideInBytes = optRowColumnStrideInBytes;
    srcLayer.sizeInBytes = N * K * sizeof( half );
    dstLayer.sizeInBytes = static_cast<unsigned int>( optMatrixDataSize );

    w_idx += N * K;
    dst_size += optMatrixDataSize;
}

size_t src_size = num_weights * sizeof(half);
CUdeviceptr d_srcMatrix;
cudaMalloc( reinterpret_cast<void**>( &d_srcMatrix ), src_size );
cudaMemcpy( reinterpret_cast<void*>( d_srcMatrix ), w.data(),
           src_size, cudaMemcpyHostToDevice );
CUdeviceptr d_optMatrix;
cudaMalloc( reinterpret_cast<void**>( &d_optMatrix ), dst_size );

OptixNetworkDescription inputNetworkDescription =
   {srcLayerDesc.data(), static_cast<unsigned int>( srcLayerDesc.size() )};
OptixNetworkDescription outputNetworkDescription =
   {dstLayerDesc.data(), static_cast<unsigned int>( dstLayerDesc.size() )};

optixCoopVecMatrixConvert(
  state.context, state.stream, /*numNetworks*/ 1, &inputNetworkDescription,
  d_srcMatrix, /*stride*/ 0, &outputNetworkDescription, d_optMatrix,
  /*stride*/ 0 );

cudaFree( reinterpret_cast<void*>( d_srcMatrix ) );

d_weights = reinterpret_cast<half*>( d_optMatrix );
```

11.3 Training with OptiX Cooperative Vectors

While the focus of this API is inferencing in OptiX programs, training is also supported. To facilitate training there are two additional device-side intrinsics as well as some options to `optixCoopVecMatMul`, such as transpose.

Consider a forward pass that calls `optixCoopVecMatMul` with an [NxK] matrix:

Listing 11.7

```
optixCoopVecMatMul<OptixCoopVec<T, N>, // VecTOut
                   OptixCoopVec<T, K>, // VecTIn
                   ...
                   false,              // transpose
                   N,                  // N
                   K,                  // K
                   ...>( ... );
```

Later, when propagating the loss backward through the same matrix, swap the input and output sizes and set the transpose flag. (The matrix is stored in the same forward layout, but the layout is specified for the backward multiplication.)

Listing 11.8

```
optixCoopVecMatMul<OptixCoopVec<T, K>, // VecTOut
                   OptixCoopVec<T, N>, // VecTIn
                   ...
                   true,               // transpose
                   K,                  // N
                   N,                  // K
                   ...>( ... );
```

There are two device functions for helping to accumulate loss values over the bias vectors and weight matrices. They both store their results in global memory using atomic adds to facilitate accumulating the loss across multiple threads.

`optixCoopVecReduceSumAccumulate`: Adds the input vector to a vector stored in memory.

`optixCoopVecOuterProductAccumulate`: Computes the outer product of the two input vectors and adds it to the matrix stored in memory. Only `OPTIX_COOP_VEC_MATRIX_LAYOUT_TRAINING_OPTIMAL` matrices are currently supported. Specify the vector types as you would for forward propagation. (There is no transpose flag.)

Listing 11.9

```
optixCoopVecOuterProductAccumulate< OptixCoopVec<T, N>,
                                    OptixCoopVec<T, K,...>( ... );
```

There is a 1:1 mapping between the output of `optixCoopVecOuterProductAccumulate` and a weight matrix of the same size and layout. Note that for training-optimal matrices, there may be padding in the data, but it is safe to read and write the padding values as these are ignored by `optixCoopVecMatMul`. Thus the following code can be used to apply the loss gradients to the weights being trained.

Listing 11.10

```
for( unsigned int j = 0; j < matrix_size_in_bytes/sizeof(T); j++ )
  weight[j] += weight_gradient[j];
```

11.4 Further Considerations

`OptixCoopVec` intrinsics can operate on any custom class that implements the `OptixCoopVec` public interface. They do not strictly require `OptixCoopVec` objects.

12 Limits

The previous chapters described properties that have an upper limit lower than the limit implied by the data type of the property. The values of these limits depend on the GPU generation for which the device context is created. These values can be queried at runtime using `optixDeviceContextGetProperty`.

Limit values may change from one OptiX SDK version to the next but not internally with updated NVIDIA drivers. Updated drivers use the limit values of the SDK version employed during application compilation as well as the GPU generation. The following table lists the NVIDIA OptiX limit values for the currently supported GPU generations, including Turing:

Type of limit	*Limit*	*Value*
Acceleration structure	Maximum number of primitives per geometry acceleration structure including motion keys	2^{29}
	Maximum number of referenced SBT records per geometry acceleration structure	2^{24}
	Maximum number of instances per instance acceleration structure	2^{28*}
	Number of bits for SBT offset	28*
	Number of bits for instance ID	28*
	Number of bits for visibility mask	8
	Maximum number of vertices per cluster acceleration structure	256
	Maximum number of triangles per cluster acceleration structure	256
	Maximum number of edges per side of a grid cluster template	11
Pipeline	Maximum trace (recursion) depth	31
	Maximum traversable graph depth	31
Device functions	Number of bits for `optixTrace` – `visibilityMask`	8
	Number of bits for `optixTrace` – `SBTstride`	4
	Number of bits for `optixTrace` – `SBToffset`	4
	Number of bits for `optixReportIntersection` – `hitKind`	7

NVIDIA OptiX limits

Note: In the table, values marked with * were raised beginning in NVIDIA OptiX version 7.1. For code compiled with OptiX SDK version 7.0, the limit is 24.

		non-RTX cards	*RTX-enabled Turing*	*Ampere*	*Ada Lovelace*	*Blackwell*
Hardware version	RT Cores version (read as x.x)	00	10	20	30	40
Feature Support	Cluster Acceleration Structures	no	yes	yes	yes	yes
	Shader Execution Reordering	no-op	no-op	no-op	yes	yes
	Cooperative Vector	no*	yes	yes	yes	yes

NVIDIA OptiX Feature Support

Note: * Cooperative Vectors are enabled in select server class chips that do not have RT Cores.

For the instance properties SBT offset, instance ID, and visibility mask, the higher bits of the 32-bit struct member must be set to zero. In case of the device functions, any bits higher than those specified in the table are ignored. Limits for device functions cannot be queried at runtime.

For more information on CUDA limits and supported features, see Appendix K: Compute Capabilities[1] in the Cuda Toolkit documentation. However, note that the upper limit for the size of an OptiX launch requires $width \times height \times depth \leq 2^{30}$.

[1] `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities`

13 Device-side functions

The NVIDIA OptiX device runtime provides functions to set and get the ray tracing state and to trace new rays from within user programs. The following functions are available in all program types:

```
optixGetTransformTypeFromHandle
optixGetInstanceChildFromHandle
optixGetInstanceIdFromHandle
optixGetInstanceTransformFromHandle
optixGetInstanceInverseTransformFromHandle
optixGetStaticTransformFromHandle
optixGetMatrixMotionTransformFromHandle
optixGetSRTMotionTransformFromHandle
optixGetGASMotionTimeBegin
optixGetGASMotionTimeEnd
optixGetGASMotionStepCount
optixGetGASPointerFromHandle
optixGetPrimitiveType(hitKind)
optixIsFrontFaceHit(hitKind)
optixIsBackFaceHit(hitKind)
optixGetTriangleVertexDataFromHandle
optixGetLinearCurveVertexDataFromHandle
optixGetQuadraticBSplineVertexDataFromHandle
optixGetQuadraticBSplineRocapsVertexDataFromHandle
optixGetCubicBSplineVertexDataFromHandle
optixGetCubicBSplineRocapsVertexDataFromHandle
optixGetCatmullRomVertexDataFromHandle
optixGetCatmullRomRocapsVertexDataFromHandle
optixGetCubicBezierVertexDataFromHandle
optixGetCubicBezierRocapsVertexDataFromHandle
optixGetRibbonVertexDataFromHandle
optixGetRibbonNormalFromHandle
optixGetSphereDataFromHandle
optixGetInstanceTraversableFromIAS
optixGetLaunchIndex
optixGetLaunchDimensions
```

`optixCoopVec*`, including `optixCoopVecMatMul`

Other functions are available only in specific program types. The following tables identify the program types within which the OptiX device functions are valid. The tables use these abbreviations for the program types:

RG	Ray generation
IS	Intersection
AH	Any hit
CH	Closest hit
MS	Miss
EX	Exception
DC	Direct callable
CC	Continuation callable

Program types that are available for a function are printed in black; unavailable functions are printed in red.

Note that all OptiX enabled functions support the same set of features as direct-callable programs except for `optixGetSbtDataPointer`. See list of non-inlined functions (page 154).

Function name	*RG*	*IS*	*AH*	*CH*	*MS*	*EX*	*CC*	*DC*
`optixGetSbtDataPointer`	RG	IS	AH	CH	MS	EX	CC	DC
`optixThrowException` `optixDirectCall`	RG	IS	AH	CH	MS		CC	DC

`optixTraverse`
`optixHitObjectGetTraverseData`
`optixHitObjectIsHit`
`optixHitObjectIsMiss`
`optixHitObjectIsNop`
`optixHitObjectGetSbtRecordIndex`
`optixHitObjectGetGASTraversableHandle`
`optixHitObjectGetWorldRayOrigin`
`optixHitObjectGetWorldRayDirection`
`optixHitObjectGetRayTmin`
`optixHitObjectGetRayTmax`
`optixHitObjectGetRayTime`
`optixHitObjectGetRayFlags`
`optixHitObjectGetTriangleVertexData`
`optixHitObjectGetLinearCurveVertexData`
`optixHitObjectGetQuadraticBSplineVertexData`
`optixHitObjectGetQuadraticBSplineRocapsVertexData`
`optixHitObjectGetCubicBSplineVertexData`
`optixHitObjectGetCubicBSplineRocapsVertexData`
`optixHitObjectGetCatmullRomVertexData`
`optixHitObjectGetCatmullRomRocapsVertexData`
`optixHitObjectGetCubicBezierVertexData`
`optixHitObjectGetCubicBezierRocapsVertexData`
`optixHitObjectGetRibbonVertexData`
`optixHitObjectGetRibbonNormal`
`optixHitObjectGetSphereData`
`optixHitObjectGetWorldToObjectTransformMatrix`
`optixHitObjectGetObjectToWorldTransformMatrix`
`optixHitObjectGetTransformListSize`
`optixHitObjectGetTransformListHandle`
`optixHitObjectGetClusterId`
`optixHitObjectGetPrimitiveIndex`
`optixHitObjectGetSbtGASIndex`
`optixHitObjectGetInstanceId`
`optixHitObjectGetInstanceIndex`
`optixHitObjectGetHitKind`
`optixHitObjectGetTriangleBarycentrics`
`optixHitObjectGetCurveParameter`
`optixHitObjectGetRibbonParameters`
`optixHitObjectGetSbtDataPointer`
`optixHitObjectGetAttribute_0...7`
	RG			CH	MS		CC	DC

Function name	*RG*	*IS*	*AH*	*CH*	*MS*	*EX*	*CC*	*DC*

`optixTrace`
`optixInvoke`
`optixMakeHitObject`
`optixMakeMissHitObject`
`optixMakeNopHitObject`
`optixHitObjectSetSbtRecordIndex`
`optixContinuationCall`
	RG			CH	MS		CC	

`optixSetPayloadTypes`
`optixGetWorldRayOrigin`
`optixGetWorldRayDirection`
`optixGetRayTmin`
`optixGetRayTmax`
`optixGetRayTime`
`optixGetRayFlags`
`optixGetRayVisibilityMask`
`optixSetPayload_0...31`
`optixGetPayload_0...31`
		IS	AH	CH	MS			

`optixGetGASTraversableHandle`
`optixGetWorldToObjectTransformMatrix`
`optixGetObjectToWorldTransformMatrix`
`optixTransformPointFromWorldToObjectSpace`
`optixTransformVectorFromWorldToObjectSpace`
`optixTransformNormalFromWorldToObjectSpace`
`optixTransformPointFromObjectToWorldSpace`
`optixTransformVectorFromObjectToWorldSpace`
`optixTransformNormalFromObjectToWorldSpace`
`optixHitObjectTransformPointFromWorldToObjectSpace`
`optixHitObjectTransformVectorFromWorldToObjectSpace`
`optixHitObjectTransformNormalFromWorldToObjectSpace`
`optixHitObjectTransformPointFromObjectToWorldSpace`
`optixHitObjectTransformVectorFromObjectToWorldSpace`
`optixHitObjectTransformNormalFromObjectToWorldSpace`
`optixGetTransformListSize`
`optixGetTransformListHandle`
`optixGetPrimitiveIndex`
`optixGetSbtGASIndex`
`optixGetInstanceId`
`optixGetInstanceIndex`
		IS	AH	CH				

`optixGetObjectRayOrigin` `optixGetObjectRayDirection`		IS	AH					

Function name	*RG*	*IS*	*AH*	*CH*	*MS*	*EX*	*CC*	*DC*

`optixGetTriangleVertexData`
`optixGetLinearCurveVertexData`
`optixGetQuadraticBSplineVertexData`
`optixGetQuadraticBSplineRocapsVertexData`
`optixGetCubicBSplineVertexData`
`optixGetCubicBSplineRocapsVertexData`
`optixGetCatmullRomVertexData`
`optixGetCatmullRomRocapsVertexData`
`optixGetCubicBezierVertexData`
`optixGetCubicBezierRocapsVertexData`
`optixGetRibbonVertexData`
`optixGetRibbonNormal`
`optixGetSphereData`
`optixGetClusterId`
`optixGetHitKind`
`optixGetPrimitiveType`
`optixIsFrontFaceHit`
`optixIsBackFaceHit`
`optixIsTriangleHit`
`optixIsTriangleFrontFaceHit`
`optixIsTriangleBackFaceHit`
`optixGetTriangleBarycentrics`
`optixGetCurveParameter`
`optixGetRibbonParameters`
`optixGetAttribute_0...7`
			AH	CH				

`optixReorder`	RG							
`optixReportIntersection`		IS						
`optixTerminateRay` `optixIgnoreIntersection`			AH					

`optixGetExceptionCode`
`optixGetExceptionLineInfo`
`optixGetExceptionDetail_0...7`
						EX		

Any function in the module that calls an NVIDIA OptiX device-side function is inlined into the caller (with the exceptions noted below). This process is repeated until only the outermost function contains these function calls.

For example, consider a closest-hit program that calls a function called `computeValue`, which calls `computeDeeperValue`, which itself calls `optixGetTriangleBarycentrics`. The inlining process inlines the body of `computeDeeperValue` into `computeValue`, which in turn, is inlined into the closest-hit program. Recursive functions that call device-side API functions will generate a compilation error.

The following functions do not trigger inlining:

```
optixTexFootprint2D
```

```
optixGetInstanceChildFromHandle
optixTexFootprint2DGrad
optixTexFootprint2DLod
optixCoopVecLoad
optixCoopVecExp2
optixCoopVecLog2
optixCoopVecTanh
optixCoopVecCvt
optixCoopVecMin
optixCoopVecMax
optixCoopVecMul
optixCoopVecAdd
optixCoopVecSub
optixCoopVecStep
optixCoopVecFFMA
optixCoopVecGetMatrixSize
```

13.1 Launch index

The *launch index* identifies the current thread, within the launch dimensions specified by `optixLaunch` on the host. The launch index is available in all programs.

Listing 13.1

```
uint3 optixGetLaunchIndex();
```

Typically, the ray generation program is only launched once per launch index.

In contrast to the CUDA programming model, program execution of neighboring launch indices is not necessarily done within the same warp or block, so the application must not rely on the locality of launch indices.

13.2 Trace

The `optixTrace` function initiates a ray tracing query starting with the given traversable and the provided ray origin and direction. If the given `OptixTraversableHandle` is null, only the miss program is invoked.

The `tmin` and `tmax` arguments set the extent associated with the current ray. Any reported hits with `hitT` outside of this range are ignored. The `tmin` value must be equal or greater than zero. The `optixTrace` function's behavior is undefined for negative `tmin` values.

An arbitrary payload is associated with each ray that is initialized with this call; the payload is passed to all the intersection, any-hit, closest-hit and miss programs that are executed during this invocation of trace. The payload can be read and written by each program using the pairs of `optixGetPayload` and `optixSetPayload` functions (for example, `optixGetPayload_0` and `optixSetPayload_0`). The payload is subsequently passed back to the caller of `optixTrace` and follows a copy-in/copy-out semantic. See "Payload" (page 149).

The rayTime argument sets the time allocated for motion-aware traversal and material evaluation. If motion is not enabled in the pipeline compile options, the ray time is ignored and removed by the compiler. To request the ray time, use the optixGetRayTime function. In a pipeline without motion, optixGetRayTime always returns 0.

The rayFlags argument can represent a combination of OptixRayFlags. The following flags are supported. Illegal combinations are noted.

OPTIX_RAY_FLAG_NONE

No change from the behavior configured for the individual acceleration structure.

OPTIX_RAY_FLAG_DISABLE_ANYHIT

Disables any-hit programs for the ray. Overrides potential instance flag OPTIX_INSTANCE_FLAG_ENFORCE_ANYHIT when intersecting instances. This flag is mutually exclusive with OPTIX_RAY_FLAG_ENFORCE_ANYHIT, OPTIX_RAY_FLAG_CULL_DISABLED_ANYHIT, OPTIX_RAY_FLAG_CULL_ENFORCED_ANYHIT.

OPTIX_RAY_FLAG_ENFORCE_ANYHIT

Forces any-hit program execution for the ray. Overrides OPTIX_GEOMETRY_FLAG_DISABLE_ANYHIT and OPTIX_INSTANCE_FLAG_DISABLE_ANYHIT.

OPTIX_RAY_FLAG_TERMINATE_ON_FIRST_HIT

Terminates the ray after the first hit and executes the closest-hit program of that hit.

OPTIX_RAY_FLAG_DISABLE_CLOSESTHIT

Disables closest-hit programs for the ray, but still executes the miss program in case of a miss.

OPTIX_RAY_FLAG_CULL_BACK_FACING_TRIANGLES

Prevents intersection of triangle back faces (respects a possible face change due to instance flag OPTIX_INSTANCE_FLAG_FLIP_TRIANGLE_FACING). This flag is mutually exclusive with OPTIX_RAY_FLAG_CULL_FRONT_FACING_TRIANGLES.

OPTIX_RAY_FLAG_CULL_FRONT_FACING_TRIANGLES

Prevents intersection of triangle front faces (respects a possible face change due to instance flag OPTIX_INSTANCE_FLAG_FLIP_TRIANGLE_FACING). This flag is mutually exclusive with OPTIX_RAY_FLAG_CULL_BACK_FACING_TRIANGLES.

OPTIX_RAY_FLAG_CULL_DISABLED_ANYHIT

Prevents intersection of geometry which disables any-hit programs (due to setting geometry flag OPTIX_GEOMETRY_FLAG_DISABLE_ANYHIT or instance flag OPTIX_INSTANCE_FLAG_DISABLE_ANYHIT). This flag is mutually exclusive with OPTIX_RAY_FLAG_CULL_ENFORCED_ANYHIT, OPTIX_RAY_FLAG_ENFORCE_ANYHIT, OPTIX_RAY_FLAG_DISABLE_ANYHIT.

OPTIX_RAY_FLAG_CULL_ENFORCED_ANYHIT

Prevents intersection of geometry which have an enabled any-hit program (due to not setting geometry flag OPTIX_GEOMETRY_FLAG_DISABLE_ANYHIT or setting instance flag OPTIX_INSTANCE_FLAG_ENFORCE_ANYHIT). This flag is mutually exclusive with OPTIX_RAY_FLAG_CULL_DISABLED_ANYHIT, OPTIX_RAY_FLAG_ENFORCE_ANYHIT, OPTIX_RAY_FLAG_DISABLE_ANYHIT.

Ray flags modify traversal behavior. For example, setting ray flag `OPTIX_RAY_FLAG_TERMINATE_ON_FIRST_HIT` causes the very first hit (that is not ignored in any-hit) to abort further traversal, defining it as the closest hit. This particular flag can be useful for shadow rays to allow for the early termination of a traversal without the need for a special any-hit program that calls `optixTerminateRay`.

The visibility mask controls intersection against configurable masks of instances. (See "Instance build inputs" (page 27).) Intersections are computed if there is at least one matching bit in both masks. The same limit applies in the number of available bits as for the instance visibility mask. See "Limits" (page 127).

The SBT offset and stride adjust the SBT indexing when selecting the SBT record for a ray intersection. Like the visibility mask, both parameters are limited. See "Limits" (page 127) and "Acceleration structures" (page 83).

The specified miss SBT index is used to identify the program that is invoked on a miss. (See "Layout" (page 82).) The argument must be a valid index for a SBT record for a miss program.

Listing 13.2

```
__device__ void optixTrace( OptixTraversableHandle handle,
    float3 rayOrigin,
    float3 rayDirection,
    float  tmin,
    float  tmax,
    float  rayTime,
    OptixVisibilityMask visibilityMask,
    unsigned int rayFlags,
    unsigned int SBToffset,
    unsigned int SBTstride,
    unsigned int missSBTIndex,
    unsigned int& p0,
    ...
    unsigned int& p7 );
```

13.3 Payload access

In intersection, any-hit, closest-hit, and miss programs, the payload is used to communicate values from the `optixTrace` that initiates the traversal, to and from other programs in the traversal, and back to the caller of `optixTrace`. There are up to thirty-two 32-bit payload values available. Getting and setting the thirty-two payload values use functions whose names end with a payload index. For example, payload 0 is set and accessed by these two functions:

Listing 13.3

```
__device__ void optixSetPayload_0( unsigned int p );
__device__ unsigned int optixGetPayload_0();
```

If the user configured payload types with the `OptixModuleCompileOptions::numPayloadTypes` and

OptixModuleCompileOptions::payloadTypes compile options, the optixSetPayloadTypes function associates a program with particular payload types. When used, this function must be called unconditionally at the top of the program. See "Payload" (page 149).

Listing 13.4

```
__device__ void optixSetPayloadTypes( unsigned int typeMask );
```

13.4 Reporting intersections and attribute access

To report an intersection with the current traversable, the intersection program can use the optixReportIntersection function. The hitKind of the given intersection is communicated to the associated any-hit and closest-hit program and allows the any-hit and closest-hit programs to customize how the attributes should be interpreted. The lowest 7 bits of the hitKind are interpreted; values [128, 255] are reserved for internal use.

Up to eight 32-bit primitive attribute values are available. Intersection programs write the attributes when reporting an intersection using optixReportIntersection. Then closest-hit and any-hit programs are able to read these attributes. For example:

Listing 13.5

```
__device__ bool optixReportIntersection(
    float hitT,
    unsigned int hitKind,
    unsigned int a0, ... , unsigned int a7 );

__device__ unsigned int optixGetAttribute_0();
```

To reject a reported intersection in an any-hit program, an application calls optixIgnoreIntersection. The closest-hit program is called for the closest accepted intersection with the attributes reported for that intersection. An any-hit program may not be called for all possible hits along the ray. When an intersection is accepted (not discarded by optixIgnoreIntersection), the interval for intersection and traversal is updated. Further intersections outside the new interval are not performed.

Although the type of the attributes is exclusively integer data, it is expected that users will wrap one or more of these data types into more readable data structures using __int_as_float and __float_as_int, or other data types where necessary.

Triangle intersections return two attributes, the barycentric coordinates (u,v) of the hit, which may be read with the convenience function optixGetTriangleBarycentrics. Analoguously, ribbon primitive intersections return two attributes, the coordinates (u,v) of the hit with the ribbon segment, which may be read by optixGetRibbonParameters. Other curve primitive intersections return one attribute, the curve parameter (u) within the polynomial curve segment, which may be read with the convenience function optixGetCurveParameter. Sphere intersections do not return an attribute.

No more than eight values can be used for attributes. Unlike the ray payload that can contain pointers to local memory, attributes should not contain pointers to local memory. This memory may not be available in the closest-hit or intersection programs when the attributes are consumed. More sophisticated attributes are probably better handled in the closest-hit program. There are generally better memory bandwidth savings by deferring certain calculations to the closest-hit program or reloading values once in the closest-hit program.

13.5 Ray information

To query the properties of the currently active ray, use the following functions:

`optixGetWorldRayOrigin / optixGetWorldRayDirection`

Returns the ray's origin and direction passed into `optixTrace`. It may be more expensive to call these functions during traversal (that is, in intersection or any-hit) than their object space counterparts.

`optixGetObjectRayOrigin / optixGetObjectRayDirection`

Returns the object space ray direction or origin based based on the current transformation stack. These functions are only available in intersection and any-hit programs.

`optixGetRayTmin`

Returns the minimum extent associated with the current ray. This is the `tmin` value passed into `optixTrace`.

`optixGetRayTmax`

Returns the maximum extent associated with the current ray. Note the following:

- In intersection and closest-hit programs, this is the smallest reported `hitT` or if no intersection has been recorded yet the `tmax` that was passed into `optixTrace`.
- In any-hit programs, this returns the `hitT` value as passed into `optixReportIntersection`.
- In miss programs, the return value is the `tmax` that was passed into `optixTrace`.

`optixGetRayTime`

Returns the time value passed into `optixTrace`. Returns 0 if motion is disabled in the pipeline.

`optixGetRayFlags`

Returns the ray flags passed into `optixTrace`.

`optixGetRayVisibilityMask`

Returns the visibility mask passed into `optixTrace`.

Note: In ray-generation and exception programs, these functions are not supported because there is no currently active ray.

13.6 Undefined values

Advanced application writers seeking more fine-grained control over register usage may want to reduce total register use in heavy intersect and any-hit programs by writing an unknown value to payload slots not used during traversal. NVIDIA OptiX provides the following function to make this straightforward:

Listing 13.6

```
__device__ unsigned int optixUndefinedValue();
```

13.7 Intersection information

The primitive index of the current intersection point can be queried using `optixGetPrimitiveIndex`. The primitive index is local to its build input.

In the case of clusters, the primitive index is local to its Cluster Acceleration Structure (CLAS). The user-defined cluster ID may be queried using `optixGetClusterId`.

The SBT index of the current intersection point can be queried using `optixGetSbtGASIndex`. The SBT index is local to its build input. (See "Shader binding table" (page 81).)

The application can query the 8-bit hit kind by using `optixGetHitKind`. The hit kind is analyzed by calling `optixGetPrimitiveType`, which tells whether a custom primitive, built-in triangle, built-in sphere, or built-in curve primitive was hit, as well as the curve type. For built-in primitives, `optixIsFrontFaceHit` and `optixIsBackFaceHit` tell whether the ray hit a front or back face of the primitive. The front face of a built-in triangle primitive is defined by the counter-clockwise winding of the vertices. For custom primitives, the hit kind is the value reported by `optixReportIntersection` when it was called in the intersection.

The default counter-clockwise winding that defines the front face can be changed to clockwise winding by setting the `OPTIX_INSTANCE_FLAG_FLIP_TRIANGLE_FACING` instance flag. The value of this flag in an instance overrides the flag's value that may have been set during the traversal of the acceleration structures of parent instances. (For a description of instance flags, see "Instance build inputs" (page 27).)

Note: It is generally more efficient to have one hit shader handle multiple primitive types (by switching on the value of `optixGetPrimitiveType`), rather than have several hit shaders that implement the same ray behavior but differ only in the type of geometry they expect.

For triangle hits, there are several notational shortcuts. The hit kind is either `OPTIX_HIT_KIND_TRIANGLE_FRONT_FACE` or `OPTIX_HIT_KIND_TRIANGLE_BACK_FACE`, depending on whether the ray came from the front or back of the triangle.

The device functions `optixIsTriangleHit`, `optixIsTriangleFrontFaceHit`, and `optixIsTriangleBackFaceHit` may also be used.

The functions `optixGetPrimitiveType`, `optixIsFrontFaceHit`, and `optixIsBackFaceHit` each have two forms. Without an argument, they analyze the current ray intersection; this form can only be used in a closest-hit or any-hit program. Or, an explicit hit kind argument may be used; this form can be called from any type of program.

When traversing a scene with instances, that is, a scene containing instance acceleration structure objects, two properties of the most recently visited instance can be queried in intersection and any-hit programs. In the case of closest-hit programs, the properties reference the instance most recently visited when the hit was recorded with `optixReportIntersection`. Using `optixGetInstanceId` the value supplied to the `OptixInstance::instanceId` can be retrieved. Using `optixGetInstanceIndex` the zero-based index within the instance acceleration structure's instances associated with the instance is returned. If no instance has been visited between the geometry primitive and the target for `optixTrace`, `optixGetInstanceId` returns `~0u` (the bitwise complement of a `unsigned int` zero) and `optixGetInstanceIndex` returns an `unsigned int` zero.

13.8 SBT record data

The data section of the current SBT record can be accessed using `optixGetSbtDataPointer`. It returns a pointer to the data, omitting the header of the SBT record (see "Shader binding table" (page 81)).

13.9 Vertex access

Triangle vertices are baked into the triangle data structure of the geometry acceleration structure. The application can thus safely release its own triangle data buffers on the device, thereby lowering overall memory usage, and continue to trace rays and fetch vertex data for hits.

The function `optixGetTriangleVertexData` returns the three triangle vertices for the currently intersected triangle at the current ray time:

Listing 13.7

```
void optixGetTriangleVertexData( float3 data[3] );
```

Additionally, if a triangle geometry acceleration structure is built with the `OPTIX_BUILD_FLAG_ALLOW_RANDOM_VERTEX_ACCESS` flag set, the application can query in object space the triangle vertex data of *any* triangle in the geometry acceleration structure using the function `optixGetTriangleVertexDataFromHandle`; this returns the three triangle vertices at the `rayTime` passed in. Motion interpolation is performed if motion is enabled on the geometry acceleration structure and pipeline.

Listing 13.8

```
void optixGetTriangleVertexDataFromHandle(
    OptixTraversableHandle gas,
    unsigned int primIdx,
    unsigned int sbtGasIdx,
    float rayTime,
    float3 data[3] );
```

The user can call functions `optixGetGASTraversableHandle`, `optixGetPrimitiveIndex`, `optixGetSbtGASIndex` and `optixGetRayTime` to obtain the geometry acceleration structure traversable handle, primitive index, geometry acceleration structure local SBT index and motion time associated with an intersection in the closest-hit and any-hit programs. The function `optixGetTriangleVertexDataFromHandle` also performs motion vertex interpolation for triangle position data.

For example:

Listing 13.9

```
OptixTraversableHandle gas = optixGetGASTraversableHandle();
unsigned int primIdx = optixGetPrimitiveIndex();
unsigned int sbtIdx = optixGetSbtGASIndex();
float time = optixGetRayTime();

float3 data[3];
optixGetTriangleVertexDataFromHandle( gas, primIdx, sbtIdx, time, data );
```

NVIDIA OptiX may remove degenerate (unintersectable) triangles from the acceleration structure during construction. Calling `optixGetTriangleVertexDataFromHandle` on a degenerate triangle returns `NaN` as triangle data, not the original triangle vertices.

The potential decompression step of triangle data may come with significant runtime overhead. Enabling random access may cause the geometry acceleration structure to use slightly more memory.

Care has to be taken if `optixGetTriangleVertexDataFromHandle` is used with a primitive index other than the value returned by `optixGetPrimitiveIndex`. `optixGetTriangleVertexDataFromHandle` expects a local primitive index corresponding to the build input / sbtGASIndex plus the primitive index offset as specified in the build input at the acceleration structure build.

Curve primitive vertices and radii are also stored in the geometry acceleration structure, and may be retrieved in an analogous way using functions depending on the type of the curve. For example, use `optixGetCubicBSplineVertexData` or `optixGetCubicBSplineVertexDataFromHandle` for cubic B-Splines. The convenience functions `optixGetRibbonNormal` or `optixGetRibbonNormalFromHandle` return the ribbon normal at a specified (u,v).

Sphere vertices and radii can be retrieved by using the function `optixGetSphereData`.

These functions' behavior is undefined when the traversable handle doesn't reference a valid geometry acceleration structure traversable, the geometry acceleration structure wasn't built with the `OPTIX_BUILD_FLAG_ALLOW_RANDOM_VERTEX_ACCESS` flag set, or the primitive index or geometry acceleration structure local SBT index are not within the valid range. Random vertex access is also not supported for Cluster GASes.

13.10 Geometry acceleration structure motion options

In addition to the motion vertex interpolation performed by `optixGetTriangleVertexDataFromHandle`, interpolation may also be desired for other user-managed vertex data, such as interpolating vertices in a custom motion intersection, or interpolating user-provided shading normals in the closest-hit shader. NVIDIA OptiX provides the following functions to obtain the motion options for a geometry acceleration structure:

```
optixGetGASMotionTimeBegin
```

`optixGetGASMotionTimeEnd`

`optixGetGASMotionStepCount`

For example, if the number of motion keys for the user vertex data equals the number of motion keys in the geometry acceleration structure, the user can compute the left key index and intra-key interpolation time as follows:

Listing 13.10

```
OptixTraversableHandle gas = optixGetGASTraversableHandle();

float currentTime = optixGetRayTime();
float timeBegin = optixGetGASMotionTimeBegin( gas );
float timeEnd = optixGetGASMotionTimeEnd( gas );
int numIntervals = optixGetGASMotionStepCount( gas ) - 1;

float time =
    ( globalt - timeBegin ) * numIntervals / ( timeEnd - timeBegin );
time = max( 0.f, min( numIntervals, time ) );
float fltKey = floorf( time );

float intraKeyTime = time - fltKey;
int leftKey = ( int )fltKey;
```

13.11 Transform list

In a multi-level/IAS scene graph, one or more transformations are applied to each primitive. NVIDIA OptiX provides intrinsics to read a transform list at the current primitive. The transform list contains all transforms on the path through the scene graph from the root traversable (passed to `optixTrace`) to the current primitive. Function `optixGetTransformListSize` returns the number of entries in the transform list and `optixGetTransformListHandle` returns the traversable handle of the transform entries.

Function `optixGetTransformTypeFromHandle` returns the type of a traversable handle and can be of one of the following types:

`OPTIX_TRANSFORM_TYPE_INSTANCE`

An instance in an instance acceleration structure. Function `optixGetInstanceIdFromHandle` returns the user-supplied instance ID. Functions `optixGetInstanceTransformFromHandle` and `optixGetInstanceInverseTransformFromHandle` return the instance transform and its inverse. Function `optixGetInstanceChildFromHandle` returns the traversable handle referenced by the instance via `OptixInstance::traversableHandle`.

`OPTIX_TRANSFORM_TYPE_STATIC_TRANSFORM`

A transform corresponding to the `OptixStaticTransform` traversable. Function `optixGetStaticTransformFromHandle` returns a pointer to the traversable.

`OPTIX_TRANSFORM_TYPE_MATRIX_MOTION_TRANSFORM`

A transform corresponding to the `OptixMatrixMotionTransform` traversable. Function `optixGetMatrixMotionTransformFromHandle` returns a pointer to the traversable.

`OPTIX_TRANSFORM_TYPE_SRT_MOTION_TRANSFORM`

A transform corresponding to the `OptixSRTMotionTransform` traversable. Function `optixGetSRTMotionTransformFromHandle` returns a pointer to the traversable.

Only use these pointers to read data associated with these nodes. Writing data to any traversables that are active during a launch produces undefined results.

For example (note this can be called directly with `optixGetWorldToObjectTransformMatrix` found in `optix_device.h`):

Listing 13.11 – Generic world to object transform computation

```
float4 mtrx[3];
for( unsigned int i = 0; i < optixGetTransformListSize(); ++i ) {
    OptixTraversableHandle handle = optixGetTransformListHandle( i );
    float4 trf[3];
    switch( optixGetTransformTypeFromHandle( handle ) ) {
    case OPTIX_TRANSFORM_TYPE_INSTANCE: {
        const float4* trns =
            optixGetInstanceInverseTransformFromHandle( handle );
        trf[0] = trns[0];
        trf[1] = trns[1];
        trf[2] = trns[2];
    } break;
    case OPTIX_TRANSFORM_TYPE_STATIC_TRANSFORM : {
        const OptixStaticTransform* traversable =
            optixGetStaticTransformFromHandle( handle );
        ...         Compute trf
    } break;
    case OPTIX_TRANSFORM_TYPE_MATRIX_MOTION_TRANSFORM : {
        const OptixMatrixMotionTransform* traversable =
            optixGetMatrixMotionTransformFromHandle( handle );
        ...         Compute trf
    } break;
    case OPTIX_TRANSFORM_TYPE_SRT_MOTION_TRANSFORM : {
        const OptixSRTMotionTransform* traversable =
            optixGetSRTMotionTransformFromHandle( handle );
        ...         Compute trf
    } break;
    default:
        continue;
    }
    if( i == 0 ) {
        mtrx[0] = trf[0];
        mtrx[1] = trf[1];
        mtrx[2] = trf[2];
    } else {
        float4 m0 = mtrx[0], m1 = mtrx[1], m2 = mtrx[2];
        mtrx[0] = rowMatrixMul( m0, m1, m2, trf[0] );
        mtrx[1] = rowMatrixMul( m0, m1, m2, trf[1] );
        mtrx[2] = rowMatrixMul( m0, m1, m2, trf[2] );
    }
}
```

Right multiply rows with pre-multiplied matrix

An application can implement a customized transformation evaluation function (for example, to get the world-to-object transformation matrix) using these intrinsics. Doing so can be beneficial as one can take advantage of the particular structure of the scene graph, for example, when a scene graph features a known maximum of transforms and only a subset of transform types are used. However, it is more important to properly specify the `OptixPipelineCompileOptions::traversableGraphFlags` and `OptixPipelineCompileOptions::usesMotionBlur` compile options based on which subset of scene graphs need to be supported. These compile options allow for optimizations to the intrinsics by compile-time removal of non-supported cases. For example, if only one level of instancing is necessary and no motion blur transforms need to be supported, set `traversableGraphFlags` to `OPTIX_TRAVERSABLE_GRAPH_FLAG_ALLOW_SINGLE_LEVEL_INSTANCING`. If set, device functions such as `optixGetWorldToObjectTransformMatrix` and `optixGetObjectToWorldTransformMatrix` are equally performant as a custom implementation.

Handles passed back from `optixGetTransformListHandle` can be stored or passed to other functions in which they can be decoded. Functions `optixGetWorldToObjectTransformMatrix` and `optixGetObjectToWorldTransformMatrix` are only available in the context of an intersection (IS, AH, CH programs). As such, it may also be required to use a customized transformation evaluation function, fetching the transformations from previously stored handles.

13.12 Instance random access

Transform lists are the preferred way to access data of an NVIDIA OptiX instance associated with the current intersection in intersection, any-hit and closest-hit programs. However, a user may require random access to instance data of any other instance in the instance acceleration structure, not associated with the current intersection. When an instance acceleration structure is built with the `OPTIX_BUILD_FLAG_ALLOW_RANDOM_INSTANCE_ACCESS` flag set, the application can query the instance traversable handle of any instance in the instance acceleration structure. Because the instance acceleration structure contains the instance data, the application can safely release its own instance data buffers on the device, thereby lowering overall memory usage.

The function `optixGetInstanceTraversableFromIAS` returns the traversable handle of an instance in the instance acceleration structure.

Listing 13.12

```
OptixTraversableHandle optixGetInstanceTraversableFromIAS(
    OptixTraversableHandle ias,
    unsigned int instIdx );
```

The user can call functions `optixGetInstanceIdFromHandle`, `optixGetInstanceChildFromHandle`, `optixGetInstanceTransformFromHandle` and `optixGetInstanceInverseTransformFromHandle` to obtain the user instance id, child traversable handle, object-to-world transform and world-to-object transform from the instance traversable handle.

For example:

Listing 13.13

```
OptixTraversableHandle instanceHandle =
    optixGetInstanceTraversableFromIAS( iasHandle, instanceIdx );
const float4* transform =
    optixGetInstanceTransformFromHandle( instanceHandle );
```

This functions' behavior is undefined when the traversable handle doesn't reference a valid instance acceleration structure traversable, the instance acceleration structure wasn't built with the `OPTIX_BUILD_FLAG_ALLOW_RANDOM_INSTANCE_ACCESS` flag set or the instance index is not within the valid range.

13.13 Terminating or ignoring traversal

In any-hit programs, use the following functions to control traversal:

`optixTerminateRay`
: Causes the traversal execution associated with the current ray to immediately terminate. After termination, the closest-hit program associated with the ray is called.

`optixIgnoreIntersection`
: Causes the current potential intersection to be discarded. This intersection will not become the new closest-hit intersection associated with the ray.

These functions do not return to the caller and they immediately terminate the program. Any modifications to ray payload values must be set before calling these functions.

13.14 Exceptions

Exceptions allow NVIDIA OptiX to check for invariants and to report details about violations.

To enable exception checks, set the `OptixPipelineCompileOptions::exceptionFlags` field with a bitwise combination of `OptixExceptionFlags`. Depending on the scenario and combination of flags, enabling exceptions can lead to severe overhead, so some flags should be mainly used in internal and debug builds.

There are several different kinds of exceptions, which are enabled based on the set of flags specified:

OPTIX_EXCEPTION_FLAG_NONE

No exception set (default).

OPTIX_EXCEPTION_FLAG_STACK_OVERFLOW

Checks for overflow in the continuation stack specified with the continuationStackSize parameter to optixPipelineSetStackSize. When this exception is enabled, the overhead is usually negligible.

OPTIX_EXCEPTION_FLAG_TRACE_DEPTH

Before tracing a new ray, checks to see if the ray depth exceeds the value specified with OptixPipelineLinkOptions::maxTraceDepth. Some stack overflows are only detected if the exception for trace depth is enabled as well (or the value of OptixPipelineLinkOptions::maxTraceDepth is correct). When this exception is enabled, the overhead is usually negligible.

OPTIX_EXCEPTION_FLAG_USER

Enables the use of optixThrowException().

If an exception occurs, the exception program is invoked. The exception program can be specified with an SBT record set in OptixShaderBindingTable::exceptionRecord. If exception flags are specified but no exception program is provided, a default exception program is provided by NVIDIA OptiX. This built-in exception program prints the first five exceptions that occurred to stdout to limit the amount of exception printing. Control does not return to the location that triggered the exception, and execution of the launch index ends.

In exception programs, the kind of exception that occurred can be queried with optixGetExceptionCode. OptiX defines two exception codes:

OPTIX_EXCEPTION_CODE_STACK_OVERFLOW

Stack overflow of the continuation stack. No information functions.

OPTIX_EXCEPTION_CODE_TRACE_DEPTH_EXCEEDED

The trace depth was exceeded. No information functions.

User exceptions can be thrown with values between 0 and $2^{30} - 1$. Zero to eight 32-bit-value details can also be used to pass information to the exception program using a set of functions of one to nine arguments, as shown in Listing 13.14.

Listing 13.14 – Function signatures for user exceptions

```
optixThrowException(
    unsigned int code );

optixThrowException(
    unsigned int code,
    unsigned int detail0 );

optixThrowException(
    unsigned int code,
    unsigned int detail0,
    unsigned int detail1 );
```

`...` *Exceptions for three to seven detail arguments*

```
optixThrowException(
    unsigned int code,
    unsigned int detail0,
    unsigned int detail1,
    unsigned int detail2,
    unsigned int detail3,
    unsigned int detail4,
    unsigned int detail5,
    unsigned int detail6,
    unsigned int detail7 );
```

The details can be queried in the exception program with eight functions, `optixGetExceptionDetail_0`() to `optixGetExceptionDetail_7`().

These functions’ behavior is undefined when exception detail for another exception code is queried or if user exception detail that is queried was not set with `optixThrowException`.

14 Payload

The *ray payload* is used to pass data between `optixTrace` and the programs invoked during ray traversal. Payload values are passed to and returned from `optixTrace`, and follow a copy-in/copy-out semantic. The payload is passed to all the intersection, any-hit, closest-hit and miss programs that are invoked during the execution of `optixTrace`. The payload can be read and written by each program using the thirty-two pairs of `optixGetPayload` and `optixSetPayload` functions (for example, `optixGetPayload_0` and `optixSetPayload_0`). Setting a payload value using `optixSetPayload` causes the updated value to be visible in any subsequent `optixGetPayload` calls until the return to the caller of `optixTrace`. Payload values that are not explicitly set in a program remain unmodified. Payload values can be set anywhere in a program. Payloads are limited in size and are encoded in a maximum of thirty-two 32-bit integer values, which are held in registers where possible. To hold additional state, these values may also encode pointers to stack-based variables or application-managed global memory.

The number of available payload values can be configured globally for the entire pipeline using the `numPayloadValues` field of `OptixPipelineCompileOptions`. When configured globally per pipeline, all intersection, any-hit, closest-hit and miss programs agree on the number of values in the payload and have read and write access to all payload values. The lifetime of all payload values therefore extends over the entire `optixTrace` call. Configuring a larger payload will thus generally increase register consumption.

Alternatively, the user can specify more fine-grained payload type semantics per program. Each payload type is defined as some number of 32-bit integer values, with specific read/write semantics for each value. These semantics declare which programs may read and/or write to a particular payload value. OptiX can use these semantics to limit the lifetime of payload values and possibly improve register consumption. Each `optixTrace` call is associated with a single payload type. Similarly, each intersection, any-hit, closest-hit and miss program is associated with one or more payload types. Up to eight payload types can be specified per module using the `OptixModuleCompileOptions` as follows:

Listing 14.1

```
unsigned int semantics[2] = {
  OPTIX_PAYLOAD_SEMANTICS_TRACE_CALLER_WRITE
    | OPTIX_PAYLOAD_SEMANTICS_CH_READ,
  OPTIX_PAYLOAD_SEMANTICS_TRACE_CALLER_READ
    | OPTIX_PAYLOAD_SEMANTICS_CH_WRITE,
};

OptixPayloadType payloadType;
payloadType.numPayloadValues = 2;
payloadType.payloadSemantics = semantics;

OptixModuleCompileOptions moduleCompileOptions = {};
...          Option assignment
moduleCompileOptions.numPayloadTypes = 1;
moduleCompileOptions.payloadTypes = &payloadType;

OptixPipelineCompileOptions pipelineCompileOptions = {};
...          Option assignment
pipelineCompileOptions.numPayloadValues = 0;
```

Note: Shader output payload values that are exclusively written in closest-hit and/or miss shaders and read by the caller generally have the shortest lifetime (`OPTIX_PAYLOAD_SEMANTICS_TRACE_CALLER_READ | OPTIX_PAYLOAD_SEMANTICS_CH_WRITE | OPTIX_PAYLOAD_SEMANTICS_MS_WRITE`), thus providing OptiX with the most optimization potential.

Setting the `OptixPipelineCompileOptions::numPayloadValues` to zero signals that the modules in a pipeline use payload types. Modules compiled with different payload types may be freely combined in a pipeline. While the maximum number of payload types per module is eight, the pipeline itself can exceed this limit due to the ability to link modules with disjoint payload types. Within a module the payload types are referenced by ID, where the ID of a type equals its index in the payload type array `payloadTypes` specified in `OptixModuleCompileOptions`. Each program in a module specifies one or more supported payload types using the `optixSetPayloadTypes` function. When called, this function must be invoked unconditionally at the top of the program. Omitting the call to `optixSetPayloadTypes` or with `OPTIX_PAYLOAD_TYPE_DEFAULT` as argument indicates that the program supports all payload types specified in `OptixModuleCompileOptions`. If the `OptixModuleCompileOptions` specify multiple payload types each `optixTrace` call must supply a single payload type. All programs invoked during the execution of an `optixTrace` call must be associated with the payload type passed to that `optixTrace`. It is the responsibility of the user to setup the Shader Binding Table so the payload type of any programs invoked in a trace execution match the type associated with the `optixTrace` call. See "Shader binding table" (page 81). Type mismatches result in undefined behavior. When validation mode is enabled on the context, NVIDIA OptiX will verify program payload types and report any detected mismatches. The payload types argument passed to `optixTrace` and `optixSetPayloadTypes` must be a compile time constant. Note that a user may assign matching payload types to different IDs in different modules. Payload types are considered to match if they agree on the number of payload values and the semantics of each value, irrespective of their IDs within a module.

Listing 14.2

```
extern "C" __global__ void __anyhit__ah()
{
    optixSetPayloadTypes(
        OPTIX_PAYLOAD_TYPE_ID_1 |
        OPTIX_PAYLOAD_TYPE_ID_
    3 );
```

Support only types 1 and 3 specified in `OptixModuleCompileOptions`*:*

```
}

extern "C" __global__ void __intersection__default()
{
    optixSetPayloadTypes(
        OPTIX_PAYLOAD_TYPE_DEFAULT );
```

Support all types specified in `OptixModuleCompileOptions`*:*

```
}

extern "C" __global__ void __raygen__rg()
{
    unsigned int p0, p1;
    optixTrace(
        OPTIX_PAYLOAD_TYPE_ID_0, ..., p0, p1 );
```

All IS, AH, CH and MS programs associated with this call need to be associated with payload type 0.

```
}
```

Note that reading or writing a payload value inside a program without the appropriate read or write semantics for that value will result in a compilation failure. Programs associated with multiple types will be compiled once for each type. Programs in a program group are associated with a single specific type only. The user configures the target type with `OptixProgramGroupOptions::payloadType`. If there is only one unique payload type that is supported by all programs specified in the group description OptiX will deduce the unique type and `OptixProgramGroupOptions::payloadType` may be left zero.

Listing 14.3

```
OptixProgramGroupDesc desc = {};
desc.kind = OPTIX_PROGRAM_GROUP_KIND_HITGROUP;
desc.hitgroup.moduleCH = moduleA;
desc.hitgroup.entryFunctionNameCH = "__closesthit__ch"
desc.hitgroup.moduleAH = moduleB;
desc.hitgroup.entryFunctionNameAH = "__anyhit__ah"

OptixProgramGroupOptions options = {};
options.payloadType = &payloadTypes[1];
```

The `payloadType` *can be defined explicitly...*

```
options.payloadType = nullptr;
```

...or let OptiX try to find a unique type match:

```
optixProgramGroupCreate(
    ..., &desc, 1, options, ..., &programGroup );
```

15 Callables

15.1 Callable programs

Callable programs allow for additional programmability within the standard set of NVIDIA OptiX programs. They are invoked using their index in the shader binding table. Invoking a function with its index enables a function call to change its target at runtime without having to recompile the program. This increase in flexibility can enable, for example, different shading effects in response to user input or programs that can be customized based on the scene setup.

Two types of callable programs exist in NVIDIA OptiX: *direct callables* and *continuation callables*. Tracing rays is supported in both types of callable programs, though the two differ in their capabilities. Like a closest-hit program, a continuation callable program can call `optixTrace` with full recursive tracing from subsequent shading programs. A direct callable program can trace rays by calling `optixTraverse`, but shading programs and recursion are not supported. A direct callable cannot call `optixTrace` or `optixInvoke`, directly or indirectly. See “`optixTraverse`” (page 102).

Another difference between direct and continuation callables is their method of invocation. Direct callables are called immediately, whereas continuation callables need to be executed by the scheduler to support recursive rays with `optixTrace`. This support of recursion may result in additional overhead when using continuation callables.

Since continuation-callable programs can only be called from other continuation-callable programs, nested callables that need to call `optixTrace` must be marked as continuation callables. This has implications when creating a shader network from callables that trace rays. In such cases, refactoring the shader network to avoid calling `optixTrace` is recommended to improve overall performance.

The use of continuation callables can also lead to better overall performance, especially if the code block in question is part of divergent code execution. A simple example is a variety of material parameter inputs, such as different noise functions or a complex bitmap network. Using a continuation callable for each of these inputs allows the scheduler to more efficiently execute these complex snippets and to potentially resolve most of the divergent code execution.

Direct callables can be called from any program type except exception. Continuation callables can be called from ray-generation, closest-hit, and miss programs.

15.2 Implementing a callable program

Three steps are required to implement a callable program:

1. Implement the program you wish to call. That program needs to be annotated with the appropriate name prefix, either `__direct_callable__` or `__continuation_callable__`.
2. Include a shader binding table record for the program group that contains the callable program. The program group for a callable can contain both types of callables, though both types share a single record in the shader binding table.
3. Invoke the callable in a program using the functions `optixDirectCall` or `optixContinuationCall`, shown in Listing 15.1. (The device-code compilation of these variadic templates requires C++11 or later.)

Listing 15.1 – Functions that can invoke callable programs

```
template<typename ReturnT, typename... ArgTypes>
ReturnT optixDirectCall( unsigned int sbtIndex, ArgTypes... args );

template<typename ReturnT, typename... ArgTypes>
ReturnT optixContinuationCall( unsigned int sbtIndex, ArgTypes... args );
```

The `sbtIndex` argument is an index of the array of callable shader binding table entries specified with the `OptixShaderBindingTable::callablesRecordBase` parameter to `optixTrace`.

15.3 Non-inlined functions

While the shortest execution time is often achieved by aggressively inlining functions, inlining can lengthen compilation time. OptiX supports the use of non-inlined functions that can be referenced by name and participate in cross-module linking. However, non-inlined functions cannot use OptiX device-side API functions without being automatically inlined during module creation.

To disable this automatic inlining, define the function as *OptiX enabled* by adding the prefix `__optix_enabled__` to the function name. Since OptiX-enabled functions do not have entries in the shader binding table, they do not have any associated data. They can be called from other OptiX-enabled functions and any entry function except for within exception programs.

As with all functions, taking the address of the function is not permitted. If function pointer semantics are required, direct callable functions should be used.

OptiX-enabled functions may incur some compile and runtime overhead similar to direct callables. See the table in "Device-side functions" (page 129) for a list of OptiX device-side API functions that are callable from OptiX-enabled functions.

16 NVIDIA AI Denoiser

Image areas that have not yet fully converged during rendering will often exhibit pixel-scale noise due to the insufficient amount of information gathered by the renderer. This grainy appearance in an image may be caused by low iteration counts, especially in scenes with complex lighting environments and material calculations.

The NVIDIA AI Denoiser can estimate the converged image from a partially converged image. Instead of improving image quality through a larger number of path tracing iterations, the denoiser can produce images of acceptable quality with far fewer iterations by post-processing the image.

The denoiser is based on statistical data sets that guide the denoising process. These data, represented by a binary blob called a *training model*, are produced from a large number of rendered images in different stages of convergence. The images are used as input to an underlying *deep learning* system. (See the NVIDIA Developer article "Deep Learning"[1] for more information about deep-learning systems.)

Because deep-learning training needs significant computational resources — even obtaining a sufficient number of partially converged images can be difficult — a general-purpose model is included with the OptiX software. This model is suitable for many renderers. However, the model may not yield optimal results when applied to images produced by renderers with very different noise characteristics compared to those used in the original training data.

Post-processing rendered images includes image filters, such as blurring or sharpening, or reconstruction filters, such as box, triangle, or Gaussian filters. Custom post-processing performed on a noisy image can lead to unsatisfactory denoising results. During post-processing, the original high-frequency, per-pixel noise may become smeared across multiple pixels, making it more difficult to detect and be handled by the model. Therefore, post-processing operations should be done *after* the denoising process, while reconstruction filters should be implemented by using filter importance-sampling.

In general, the pixel color space of an image that is used as input for the denoiser should match the color space of the images on which the denoiser was trained. However, slight variations, such as substituting sRGB with a simple gamma curve, should not have a noticeable impact. Images used for the training model included with the NVIDIA AI Denoiser distribution were output directly as HDR data.

[1] `https://developer.nvidia.com/deep-learning`

16.1 Functions and data structures for denoising

Calling function optixDenoiserCreate creates a denoiser object:

Listing 16.1

```
OptixResult optixDenoiserCreate(
    OptixDeviceContext          context,
    OptixDenoiserModelKind      modelKind,
    const OptixDenoiserOptions* options,
    OptixDenoiser*              denoiser );
```

The OptixDenoiserModelKind is one of the following enum values:

OPTIX_DENOISER_MODEL_KIND_LDR

The input image data contains values of limited dynamic range with RGB values in the range 0.0 to 1.0.

OPTIX_DENOISER_MODEL_KIND_HDR

The input image data contains values of high dynamic range with RGB values in the range of 0.0 to approximately 10,000.0 or more.

OPTIX_DENOISER_MODEL_KIND_AOV

Separate passes can be defined by many rendering systems using *arbitrary output variables*, or AOVs. AOV images from a rendering system can contain diffuse, emission, glossy, specular or other types of data. High dynamic range AOV images can be passed as input layers to optixDenoiserInvoke in addition to the beauty, RGB, albedo and normal images. The AOV mode can be used to denoise multiple layers simultaneously and may reduce processing time.

After denoising, the denoised output layers can be composited to a noise-free beauty layer, typically by adding all RGB values from the denoised layers.

The first OptixDenoiserLayer in the list of layers passed to optixDenoiserInvoke is the noisy beauty image. Subsequent layers in this list are noisy AOVs. A noisy beauty image is required in all modes of OptixDenoiserModelKind and the denoised image for each given layer is written back in OptixDenoiserLayer::output.

OPTIX_DENOISER_MODEL_KIND_TEMPORAL

In this mode, a sequence of images is denoised to eliminate temporal noise. It requires the denoised beauty image from the previous frame in OptixDenoiserLayer::previousOutput. The flow vector image is specified by OptixDenoiserGuideLayer::flow. OptixDenoiserParams::hdrIntensity must be set in this mode.

In the first frame of a sequence, previousOutput could be set to the noisy beauty image of the first frame instead of the denoised version and all flow vectors set to zero. previousOutput is read in optixDenoiserInvoke before writing a new output, so previousOutput could be set to output (the same buffer) for efficiency if useful in the application.

`OPTIX_DENOISER_MODEL_KIND_TEMPORAL_AOV`

In this mode, the following two additional fields are defined:

- `OptixDenoiserGuideLayer::previousOutputInternalGuideLayer`
- `OptixDenoiserGuideLayer::outputInternalGuideLayer`

The image type for both fields is a *guide layer*, an `OptixImage2D` of type `OPTIX_PIXEL_FORMAT_INTERNAL_GUIDE_LAYER`. Temporal denoising relies on these internal guide layers to carry information from the previous denoising pass, specified with `previousOutputInternalGuideLayer` and `outputInternalGuideLayer` in the `OptixDenoiserGuideLayer` struct. A set of guide layers must be provided to the denoising pass as both input and output. These layers have the format `OPTIX_PIXEL_FORMAT_INTERNAL_GUIDE_LAYER` and should be sized according to the field `internalGuideLayerPixelSizeInBytes` returned by `optixDenoiserComputeMemoryResources`.

An internal guide layer is a densely packed format, where each pixel has size `OptixDenoiserSizes::internalGuideLayerPixelSizeInBytes`, and each row is `width` × pixel size. The width and height of the guide layers are set to the same dimensions that are used for the other layer images. The `pixelStrideInBytes` field of the guide layer is set to `internalGuideLayerPixelSizeInBytes`. The `rowStrideInBytes` field is set to `width` × `internalGuideLayerPixelSizeInBytes`.

The parameter `OptixDenoiserParams::temporalModeUsePreviousLayers` controls the use of previous layers in denoising calculations. In temporal mode, setting this to 1 indicates the denoiser should read the values of the previous frame. Set to 0 for initial frames or when you want to reset the temporal sequence. In the first frame, when using temporal denoising modes, `OptixDenoiserGuideLayer::flow` must either contain valid motion vectors if available, otherwise the xy vectors must be set to zero (no motion). In temporal upscaling mode `OptixDenoiserLayer::previousOutput` is not accessed when `OptixDenoiserParams::temporalModeUsePreviousLayers` is not set.

Each new sequence should contain the noisy beauty image as the value of `previousOutput`. The `previousOutputInternalGuideLayer` image content must be set to zero for the first frame.

Motion vectors must be set to zero if they are not available for the first frame.

Fields `previousOutputInternalGuideLayer` and `outputInternalGuideLayer` must refer to two separate buffers; memory cannot be shared between these two buffers, regardless of the tiling mode. The buffer memory locations must start at a 16-byte aligned address or `optixDenoiserInvoke` will return an error code. After denoising a frame, these two buffers must be exchanged, so that `outputInternalGuideLayer` becomes `prevousOutputInternalGuideLayer` for the next frame. Instead of copying data into the previous guide layer, you should use a double-buffering strategy by swapping the content of the two `OptixImage2D` structs. You should also use double buffering for the `output` and `previousOutput` fields of the `OptixDenoiserLayer` struct.

`OPTIX_DENOISER_MODEL_KIND_UPSCALE2X`
`OPTIX_DENOISER_MODEL_KIND_TEMPORAL_UPSCALE2X`

These modes upscale the input image by a factor of two in both dimensions. All output images must be allocated accordingly using twice the original size of the input width and height:

- `OptixDenoiserLayer::previousOutput`
- `OptixDenoiserLayer::output`
- `OptixDenoiserGuideLayer::previousOutputInternalGuideLayer`
- `OptixDenoiserGuideLayer::outputInternalGuideLayer`

When using `OPTIX_DENOISER_MODEL_KIND_TEMPORAL_UPSCALE2X` mode for the first frame, `previousOutputInternalGuideLayer` image content must be set to zero. The `previousOutput` field is ignored in this mode and just the memory needs to be allocated. Internally, the noisy beauty image will be upscaled by a factor of two and used as the previous output (similar to the non-upcaling mode where the noisy beauty image should be passed as the previously denoised output).

The denoiser options determine the type of guide layers used:

Listing 16.2

```
typedef struct OptixDenoiserOptions {
   unsigned int           guideAlbedo;
   unsigned int           guideNormal;
   OptixDenoiserAlphaMode denoiseAlpha;
} OptixDenoiserOptions;
```

The `denoiseAlpha` field of `OptixDenoiserOptions` defines how the optional alpha channel of the noisy beauty image should be treated. The possible values of the `OptixDenoiserAlphaMode` enum are:

`OPTIX_DENOISER_ALPHA_MODE_COPY`

This is the default, alpha is copied from input to output and not denoised.

`OPTIX_DENOISER_ALPHA_MODE_DENOISE`

The alpha channel is also denoised.

A denoiser object can also be created using a user-defined model:

Listing 16.3

```
OptixResult optixDenoiserCreateWithUserModel(
   OptixDeviceContext context,
   const void*        userData,
   size_t             userDataSizeInBytes,
   OptixDenoiser*     denoiser );
```

After denoising, `optixDenoiserDestroy` should be called to destroy the denoiser object along with associated host resources:

Listing 16.4

```
OptixResult optixDenoiserDestroy(
   OptixDenoiser denoiser );
```

16.1.1 Structure and use of image buffers

An RGB image buffer that contain a noisy image is provided as input to the denoising process. The optional fourth (alpha) channel of the image can be configured so that it is ignored by the denoiser. Note that this buffer must contain values between 0 and 1 for each of the three color channels (for example, as the result of tone-mapping) and should be encoded in sRGB or gamma space with a gamma value of 2.2 when working with low dynamic range (LDR) input.

When working with high dynamic range (HDR) input instead, RGB values in the color buffer should be in a range from zero to 10,000, and on average not too close to zero, to match the built-in model. Images in HDR format can contain single, extremely bright, nonconverted pixels, called *fireflies*. Using a preprocess pass that corrects drastic under- or over-exposure along with clipping or filtering of fireflies on the HDR image can improve the denoising quality dramatically. Note, however, that no tone-mapping or gamma correction should be performed on HDR data.

An optional, noise-free normal buffer contains the surface normal vectors of the primary hit in camera space. The normal buffer is enabled by `OptixDenoiserOptions::guideNormal` and specified by `OptixDenoiserGuideLayer::normal`. This buffer contains data in format `OPTIX_PIXEL_FORMAT_HALF3` or `OPTIX_PIXEL_FORMAT_FLOAT3`. These two formats can represent the x, y and z components of a vector. The normal buffer must have the same type and dimensions as the input buffer. The vectors in the normal buffer are normalized.

The camera space is assumed to be right-handed such that the camera is looking down the negative z axis, and the up direction is along the y axis. The x axis points to the right.

Visualizing the normal vectors as colors for debugging purposes should produce images similar to Figure 16.1 (page 160) and Figure 16.2 (page 160). The scanlines of the normal buffer proceed from top to bottom in the images. The surface normal vector components have values in the range [-1.0, 1.0]. Figure 16.1 (page 160) maps the component values to the range

[0.0, 1.0]. Figure 16.2 (page 160) clamps the component values to [0.0,1.0]. The value of a normal buffer pixel for which there was no primary hit is (0,0,0).

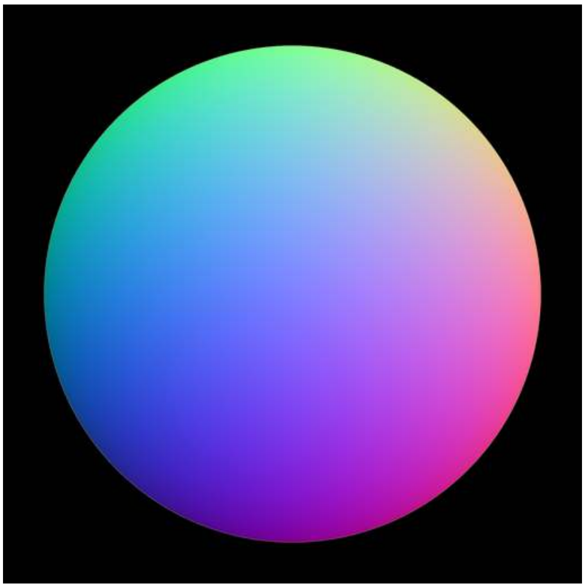

Fig. 16.1 - Surface normal vector components mapped to [0.0, 1.0]

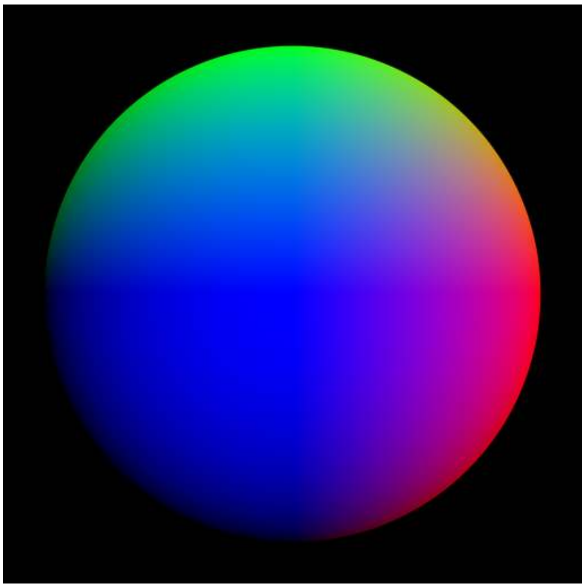

Fig. 16.2 - Surface normal vector components clamped to [0.0, 1.0]

The normal buffer can improve denoising quality for scenes containing a high degree of geometric detail. This detail may be part of the geometric complexity of the surface itself, or may be the result of high-resolution images used in normal or bump mapping.

The optional, noise-free albedo image represents an approximation of the color of the surface of the object, independent of view direction and lighting conditions. In physical terms, the albedo is a single color value approximating the ratio of radiant exitance to the irradiance under uniform lighting. The albedo value can be approximated for simple materials by using the diffuse color of the first hit, or for layered materials, by using a weighted sum of the albedo values of the individual BRDFs. For some objects such as perfect mirrors or highly glossy materials, the quality of the denoising result might be improved by using the albedo value of a subsequent hit instead. The fourth channel of this buffer is ignored, but must have the same type and dimensions as the input buffer. Specifying albedo can dramatically improve denoising quality, especially for very noisy input images.

16.1.2 Temporal denoising modes

In temporal denoising modes, pixels in the two-dimensional flow image represents the motion from the previous to the current frame for that pixel. Pixels in the flow image can be in one of two formats:

- `OPTIX_PIXEL_FORMAT_FLOAT2`
- `OPTIX_PIXEL_FORMAT_HALF2`

Figure 16.3 (page 161) shows the definition of the flow vector based on the hit points of the current and previous frames.

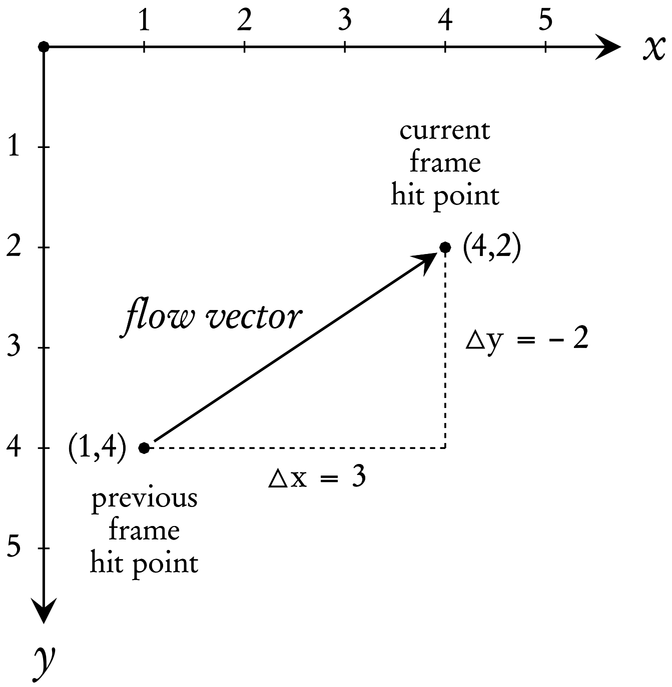

Fig. 16.3 - Definition of the flow vector based on the previous and current hit points

If the hit point of the surface at position (4,2) in the current frame has moved from position (1,4) in the previous frame, the flow vector at position (4,2) would be x=3, y=-2. This flow vector definition assumes that images define y=0 at the top of the image. If images define y=0 at the bottom, this flow vector would be x=3, y=2. The flow vector coordinates must be described with subpixel (fractional) precision.

The OptiX SDK provides the `OptixDenoiser` sample which could be used to verify properly specified flow vectors. When invoked with -z, the tool looks up the position in the previous frame of each pixel using the flow vector. The pixel value from the previous frame position is looked up in the current frame position and written. When denoising a sequence with the -z option, frame *N* with motion applied should look similar to frame *N+1* in the noisy input sequence. There should be no major jumps when comparing these images, just shading differences as well as differences due to disocclusion and so forth.

16.1.3 Allocating denoiser memory

To allocate the required, temporary device memory to run the denoiser, call function `optixDenoiserComputeMemoryResources`:

Listing 16.5

```
OptixResult optixDenoiserComputeMemoryResources(
    const OptixDenoiser denoiser,
    unsigned int        outputWidth,
    unsigned int        outputHeight,
    OptixDenoiserSizes* returnSizes );
```

The memory requirements are returned in the fields of struct `OptixDenoiserSizes`:

Listing 16.6

```
typedef struct OptixDenoiserSizes {
    size_t       stateSizeInBytes;
    size_t       withOverlapScratchSizeInBytes;
    size_t       withoutOverlapScratchSizeInBytes;
    unsigned int overlapWindowSizeInPixels;
    size_t       computeAverageColorSizeInBytes
    size_t       computeIntensitySizeInBytes
    size_t       internalGuideLayerPixelSizeInBytes;
} OptixDenoiserSizes;
```

The `OptixDenoiserSizes` values are used as input to function `optixDenoiserSetup`:

Listing 16.7

```
OptixResult optixDenoiserSetup(
    OptixDenoiser denoiser,
    CUstream      stream,
    unsigned int  inputWidth,
    unsigned int  inputHeight,
    CUdeviceptr   denoiserState,
    size_t        denoiserStateSizeInBytes,
    CUdeviceptr   scratch,
    size_t        scratchSizeInBytes );
```

In tiling mode, the `inputWidth` and `inputHeight` values should include two times the overlap size in the tile window dimensions.

16.1.4 Using the denoiser

To execute denoising on a given image, call function `optixDenoiserInvoke`:

Listing 16.8

```
OptixResult optixDenoiserInvoke(
   OptixDenoiser                 denoiser,
   CUstream                      stream,
   const OptixDenoiserParams*    params,
   CUdeviceptr                   denoiserState,
   size_t                        denoiserStateSizeInBytes,
   const OptixDenoiserGuideLayer* guideLayer,
   const OptixDenoiserLayer*     layers,
   unsigned int                  numLayers,
   unsigned int                  inputOffsetX,
   unsigned int                  inputOffsetY,
   CUdeviceptr                   scratch,
   size_t                        scratchSizeInBytes );
```

When guide layers are enabled in `OptixDenoiserOptions`, the corresponding image buffers are specified by the `OptixDenoiserGuideLayer` struct. In temporal modes a flow vector image is specified by the `OptixDenoiserGuideLayer::flow` field.

Listing 16.9

```
typedef struct OptixDenoiserGuideLayer {
   OptixImage2D albedo;
   OptixImage2D normal;
   OptixImage2D flow;
   OptixImage2D previousOutputInternalGuideLayer;
   OptixImage2D outputInternalGuideLayer;
   OptixImage2D flowTrustworthiness;
} OptixDenoiserGuideLayer;
```

Beauty and AOV layers are specified by the `OptixDenoiserLayer::input`. The denoised output of the beauty image in the layer is written to `output`. In temporal modes the denoised beauty image from the previous frame for a layer is specified by `OptixDenoiserLayer::previousOutput`.

Listing 16.10

```
typedef struct OptixDenoiserLayer {
   OptixImage2D input;
   OptixImage2D previousOutput;
   OptixImage2D output;
} OptixDenoiserLayer;
```

The denoiser layers are defined by sets of struct `OptixImage2D`:

Listing 16.11

```
typedef struct OptixImage2D {
   CUdeviceptr      data;
   unsigned int     width;
   unsigned int     height;
   unsigned int     rowStrideInBytes;
   unsigned int     pixelStrideInBytes;
   OptixPixelFormat format;
} OptixImage2D;
```

Function `optixDenoiserInvoke` executes the denoiser on a set of input data and produces one output image. State memory must be available during the execution of the denoiser or until `optixDenoiserSetup` is called with a new state memory pointer. Scratch memory passed as a CUdeviceptr must be exclusively available to the denoiser during execution. It is only used for the duration of `optixDenoiserInvoke`. Scratch and state memory sizes must have a size greater than or equal to the sizes returned by `optixDenoiserComputeMemoryResources`.

Input and output to the denoiser uses struct `OptixImage2D` to represent pixel data:

Listing 16.12

```
typedef struct OptixImage2D {
   CUdeviceptr      data;
   unsigned int     width;
   unsigned int     height;
   unsigned int     rowStrideInBytes;
   unsigned int     pixelStrideInBytes;
   OptixPixelFormat format;
} OptixImage2D;
```

Parameters `inputOffsetX` and `inputOffsetY` are pixel offsets in the `inputLayers` image. These offsets specify the beginning of the image without overlap. When denoising an entire image without tiling, there is no overlap and `inputOffsetX` and `inputOffsetY` must be zero. When denoising a tile which is adjacent to one of the four sides of the entire image the corresponding offsets must also be zero; there is no overlap at the side adjacent to the image border. Parameters `inputWidth` and `inputHeight` correspond to the values passed to `optixDenoiserComputeMemoryResources`.

The structure of pixel data is defined by the enum `OptixPixelFormat`:

Field name	*Description*
OPTIX_PIXEL_FORMAT_HALF2	Two halfs, XY
OPTIX_PIXEL_FORMAT_HALF3	Three halfs, RGB/XYZ
OPTIX_PIXEL_FORMAT_HALF4	Four halfs, RGBA/XYZW
OPTIX_PIXEL_FORMAT_FLOAT2	Two floats, XY
OPTIX_PIXEL_FORMAT_FLOAT3	Three floats, RGB/XYZ
OPTIX_PIXEL_FORMAT_FLOAT4	Four floats, RGBA/XYZW
OPTIX_PIXEL_FORMAT_INTERNAL_GUIDE_LAYER	internal format

The `OptixPixelFormat` value of the `inputLayers` must match the format that was specified during `optixDenoiserCreate`. The `outputLayer` must have the same width, height and pixel format as the input RGB(A) layer. All input layers must have the same width and height.

Further control over denoising is provided by the `OptixDenoiserParams` struct:

Listing 16.13

```
typedef struct OptixDenoiserParams {
   CUdeviceptr          hdrIntensity;
   float                blendFactor;
   CUdeviceptr          hdrAverageColor;
   unsigned int         temporalModeUsePreviousLayers;
} OptixDenoiserParams;
```

The `blendFactor` field specifies an interpolation weight between the noisy input image (1.0) and the denoised output image (0.0). Calculation of `hdrIntensity` is described in "Calculating the HDR intensity parameter" (page 166).

Parameter `hdrAverageColor` is used when the `OPTIX_DENOISER_MODEL_KIND_AOV` model kind is set. This parameter is a pointer to three floats that are the average log color of the RGB channels of the input image. The default value (the null pointer) will produce results that are not optimal.

When denoising entire images without tiling, the same scratch memory as passed to `optixDenoiserInvoke` could be used.

16.1.5 Calculating the HDR average color of the AOV model

The function `optixDenoiserComputeAverageColor` computes the average logarithmic value for each of the first three channels of the input image. When denoising tiles the intensity of the entire image should be computed — not per tile — for consistent results.

The `inputImage` parameter must contain three or four components of type half or float. The data type `unsigned char` is not supported.

The required scratch memory sizes for the API functions `optixDenoiserComputeAverageColor` and `optixDenoiserComputeIntensity` should be queried with `optixDenoiserComputeMemoryResources`. The scratch memory size for these functions is stored in `OptixDenoiserSizes::computeAverageColorSizeInBytes` and `OptixDenoiserSizes::computeIntensitySizeInBytes`

Listing 16.14

```
OptixResult optixDenoiserComputeAverageColor(
    OptixDenoiser       denoiser,
    CUstream            stream,
    const OptixImage2D* inputImage,
    CUdeviceptr         outputAverageColor,
    CUdeviceptr         scratch,
    size_t              scratchSizeInBytes );
```

16.1.6 Calculating the HDR intensity parameter

The value of `hdrIntensity` in `OptixDenoiserParams` can be calculated in one of two ways. A custom application-side, preprocessing pass on the image data could provide a result value of type `float`.

Intensity can also be calculated by calling function `optixDenoiserComputeIntensity`:

Listing 16.15

```
OptixResult optixDenoiserComputeIntensity(
    OptixDenoiser       denoiser,
    CUstream            stream,
    const OptixImage2D* inputImage,
    CUdeviceptr         outputIntensity,
    CUdeviceptr         scratch,
    size_t              scratchSizeInBytes );
```

Function `optixDenoiserComputeIntensity` calculates the logarithmic average intensity of parameter `inputImage`. The calculated value is returned in parameter `outputIntensity` as a pointer to a single value of type `float`.

Intensity calculation can be dependent on `optixDenoiserInvoke`. The `params` parameter of `optixDenoiserInvoke` is a struct of type `OptixDenoiserParams`. By default, the `hdrIntensity` field of `OptixDenoiserParams` is a null pointer. If `hdrIntensity` is not a null pointer, it points to an array of RGB values that are multiplied by the output values in `outputIntensity`. This is useful for denoising HDR images which are very dark or bright. When denoising with tiles, the intensity of the entire image should first be computed for consistent results across tiles. (Tiling is described in the "Using image tiles with the denoiser" (page 167).)

For each RGB pixel in the inputImage the intensity is calculated and summed if it is greater than 1e-8f as follows:

$$intensity = log(r \times 0.212586 + g \times 0.715170 + b \times 0.072200)$$

The function returns:

$$\frac{0.18}{exp\left(\frac{sum\_of\_intensities}{number\_of\_summed\_pixels}\right)}$$

Execution of `optixDenoiserComputeIntensity` requires a scratch memory size in bytes of at least:

```
sizeof(int) * (2 + inputImage::width * inputImage::height)
```

When denoising entire images (without tiling) the same scratch memory that is passed to `optixDenoiserInvoke` can be used.

Further information about this tone-mapping strategy can be found in "Photographic Tone Reproduction for Digital Images"[2] by Erik Reinhard, et al.

16.2 Using image tiles with the denoiser

Denoising can be performed incrementally on subregions of an image to limit memory usage during the denoising process. An image is divided in to a set of *tiles* defined by their width and height, as well as a surrounding region of overlapping pixels to avoid discontinuities at tile boundaries.

For example, Figure 16.4 (page 168) describes a possible structure for tiles and their overlapping regions using `OptixImage2D`:

[2] https://dl.acm.org/doi/10.1145/566654.566575

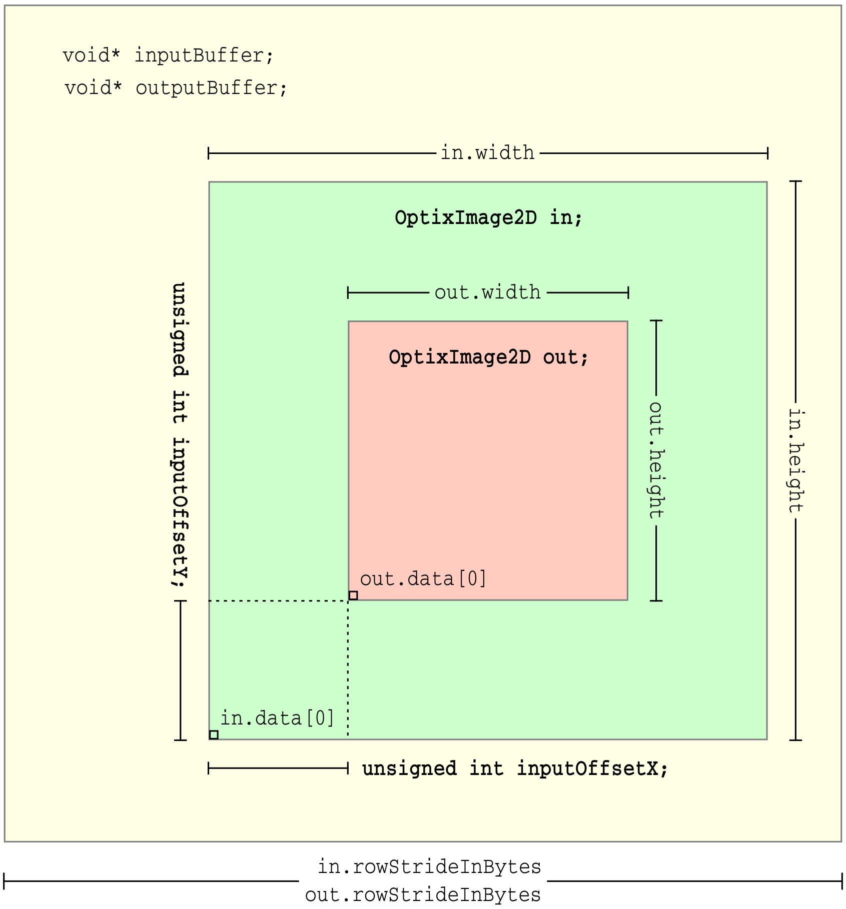

Fig. 16.4 - An example of a denoiser tiling layout

The tiling code must accommodate tile sizes that do not evenly divide the image buffers. Figure 16.5 shows a tile that with a consistent overlap size on all edges:

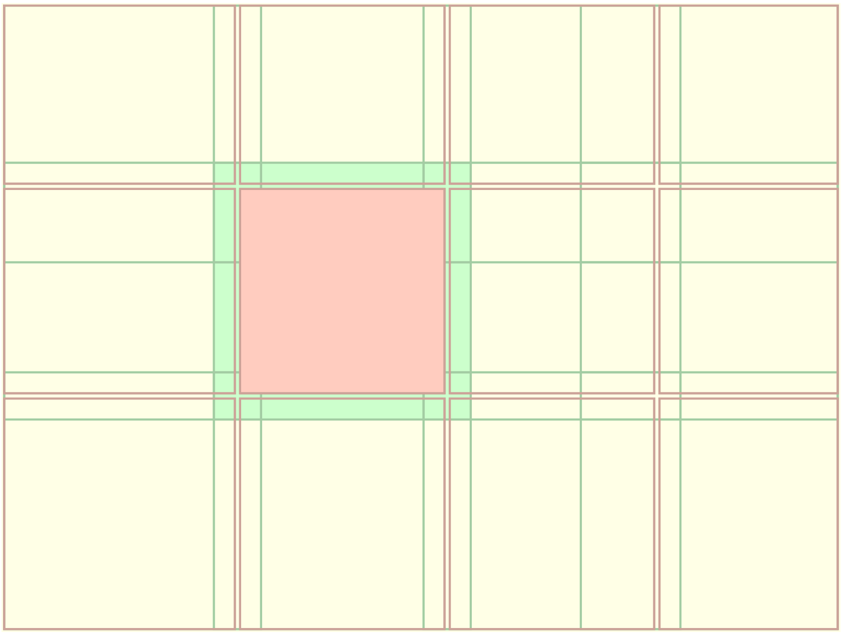

Fig. 16.5 - Output tile with overlap

Figure 16.6 (page 169) shows how a tiling procedure could vary the tile sizes based on their position in the input and output buffers.

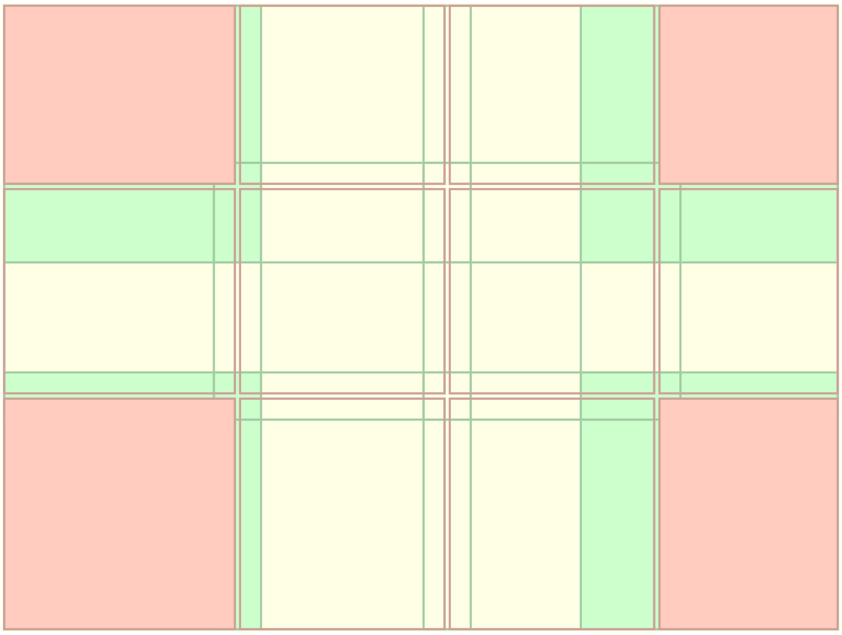

Fig. 16.6 - Tiles in corners showing variable overlap

The OptiX SDK includes an example of the tiling process. See header file `optix_denoiser_tiling.h` in the SDK examples.