NVIDIA VR SLI

PG-07774-001_v05 | March 2017

Programming Guide

DOCUMENT CHANGE HISTORY

PG-07774-001_v05

Version	Date	Authors	Description of Change
01	8/11/2015	NR/CC	Initial release
02	11/11/2015	NR	Updated for v1.0 release
03	12/11/2015	NR	Updated for v1.1 release
04	04/26/2016	RP/JK	Added earliest supported interface version info for `ID3D11MultiGPUDevice` methods. Added Unmap resource to the list of D3D11 calls to which the GPU mask applies. Added `GetVideoBridgeStatus`.
05	03/07/2017	JK	Added MGPU constant buffer support in V3 interface.

TABLE OF CONTENTS

Chapter 1. INTRODUCTION TO NVIDIA VR SLI

1.1 NVIDIA VR SLI GOALS

NVIDIA® VR SLI®, also known as Explicit Multi-GPU, provides the ability to send different work to each GPU in an SLI system, subject to these restrictions:

- A single command stream is presented to all GPUs.
- The state of the GPUs cannot diverge except for a few restricted pieces of state that are handled specially.

This API doesn't attempt to solve the problem of efficient stereo rendering on a single GPU, or automatically distributing views across GPUs. It provides only these capabilities:

- To broadcast a command stream to several GPUs with minimal per-GPU state
- To synchronize GPUs and transfer data between them

Currently, the API is developed only in DirectX 11.

1.2 GPU MASKING AND STATE

This API provides the ability to mask off commands that perform actions such as clear, draw, and dispatch so that they apply only to a subset of the GPUs. It does not provide the ability to mask commands for setting arbitrary state, such as binding textures, setting or rasterizing state. The ability to mask commands for setting state is omitted because it would add a great deal of CPU overhead in handling an independent state vector for each GPU.

Technically, masking provides a great deal of flexibility because it is possible to set the mask to target a single GPU at a time, and generate a unique command stream for each GPU. While this approach achieves the required result, it is somewhat inefficient for these reasons:

- It requires many API calls, many of which send the same data to all GPUs.
- Additionally, some GPU performance may be lost due to extra state processing.

Despite these issues, masking can be a useful first step in integrating multi-GPU support into a game. If the overhead of this approach is acceptable for your purposes, no further steps are required.

The API also provides the ability to broadcast commands to multiple GPUs with the following limited subset of state for each GPU:

- Constant buffer bindings
- Viewports
- Scissor rectangles

This way is the more efficient way of using the API when you require most or all objects to be drawn on all GPUs, for example, for stereo rendering. Ideally, most of your rendering will be done with the mask set to specify all GPUs, using constant buffers to configure different view matrices and so forth for individual GPUs. It is entirely reasonable for the game to still use GPU masking occasionally, but the less GPU masking is used, the more efficient rendering will be.

To integrate multi-GPU support into an application, a reasonable first step is to add `SetGPUMask` calls where the application iterates over views. This first step should enable a working implementation with a limited amount of effort, although performance is likely to be suboptimal. Next, individual rendering passes can be converted one-by-one to the more efficient broadcast approach.

Chapter 2. USING VR SLI

2.1 MEMORY MODEL

The VR SLI API allocates memory based on the usage with which a resource is created:

- **Resources with immutable or default usage** implicitly have a separate instance for each GPU. These resources reside in video memory. While they have the same address, size, and format on all GPUs, they may contain different data on each GPU.
- **Resources with dynamic or staging usage** have only a single instance, which is accessible to all GPUs. These resources reside in system memory.

However, constant buffers are always treated as having a single instance, regardless of their usage type. Constant buffers can't have different data for individual GPUs. Instead, you must explicitly create a separate constant buffer for each GPU, and bind the resulting buffers by using `ID3D11MultiGPUDevice::SetConstantBuffers()`.

For clarity and simplicity, this description of the memory model of the VR SLI API omits some details that are handled by the OS and driver, not by the user:

- Video memory resources may be made resident or nonresident by the OS at any time.
- In some circumstances, system memory resources may be allocated with separate copies for each GPU.

2.1.1 Uploading to Multiple GPUs

Data that is intended to be shared across all GPUs can be uploaded by using the ordinary D3D11 mechanisms. For example, constant buffers and dynamic vertex and index buffers can be mapped and written to as usual, and their data will be accessible to all GPUs.

It's also possible to upload different data to each GPU by using the following technique:

1. Writing all GPUs' data to a large dynamic or staging resource or several such resources
2. Calling the `ID3D11MultiGPUDevice::CopySubresourceRegion()` method to copy into the individual GPUs' instances of a default resource

   For more information, see Reading Data from One GPU and Multiple GPUs.

Note that this technique does not work for constant buffers.

2.1.2 Reading Data from One GPU and Multiple GPUs

You can use the `ID3D11MultiGPUDevice::CopySubresourceRegion()` method in two different ways to read data from one GPU and from multiple GPUs.

- To read from an individual GPU, use the `ID3D11MultiGPUDevice::CopySubresourceRegion()` method to copy data from the GPU's instance of a default resource into a staging resource. Then map the staging resource as usual.
- To read from multiple GPUs, use a larger staging resource and copy each GPU's data into a different region of the resource, or use several staging resources.

2.1.3 Cross-GPU Memory Copies

The `ID3D11MultiGPUDevice::CopySubresourceRegion()` method can also be used to copy data between default-usage resources on two different GPUs. When copied this way, the data is transferred directly between GPUs whenever possible, without staging through a buffer in CPU memory.

If a resource is created with `D3D11_BIND_SHADER_RESOURCE` as the **only** enabled bind flag, it can't be copied across GPUs. This restriction is intended to reduce the internal resource consumption associated with resources that are visible across GPUs. Resources created with any other bind flags enabled can be copied across GPUs.

Depending on the bus available, copying across GPUs can be slow. For example, PCIe 2.0 × 16 provides only 8 GB/s of bandwidth, compared with the hundreds of GB/s available for a GPU to access its own memory. To mitigate this slowness, the API enables explicit synchronization for cross-GPU copies. This explicit synchronization allows applications to issue a copy request, then continue doing unrelated rendering work that will execute concurrently with the copy, and later wait for the copy to finish.

Synchronization is done with fences, much like in D3D12. One GPU can write a fence, and other GPUs can wait on it.

For convenience, `nvapi.h` provides a pair of macros for synchronizing the start and end of an asynchronous copy:

```
#define FENCE_SYNCHRONIZATION_START(pMultiGPUDevice, hFence, Value,
srcGpu, dstGpu) \
    pMultiGPUDevice->SetFence(dstGpu, hFence, Value); \
    pMultiGPUDevice->WaitForFence(1 << (srcGpu), hFence, Value); \
    Value++;

#define FENCE_SYNCHRONIZATION_END(pMultiGPUDevice, hFence, Value,
srcGpu, dstGpu) \
    pMultiGPUDevice->SetFence(srcGpu, hFence, Value); \
    pMultiGPUDevice->WaitForFence(1 << (dstGpu), hFence, Value); \
    Value++;
```

To prevent hazards in an asynchronous copy, these macros should be called as follows:

- To prevent a write-after-write hazard between previous rendering and the copy, `FENCE_SYNCHRONIZATION_START()` should be called after any previous writes to the destination resource, and before the asynchronous copy.
- To prevent a read-after-write hazard, `FENCE_SYNCHRONIZATION_END()` should be called after the asynchronous copy and before any subsequent reads from the destination resource.

Note that asynchronous operation is optional and not the default. All cross-GPU copies synchronize the commands streams of the GPUs involved before performing the copy unless the asynchronous flag is specified explicitly.

2.2 GENERAL NVAPI OPERATION

NVAPI is the NVIDIA API for accessing many vender-specific capabilities of NVIDIA GPUs. NVAPI covers a very wide range of functionality. The most relevant aspects of this functionality for the operation of multi-GPU functionality are listed in the subsections that follow. For full details, see the general NVAPI library documentation.

2.2.1 Accessing the Library

NVAPI consists of the following layers:

- **The API layer** is a header, or set of headers, and a minimal static library.

  The static library is just a minimal thunking layer used to access the NVAPI runtime provided by the driver. It does not add any dependencies to the code that require the presence of additional components.

The header and the static library must be integrated into any application that is to use NVAPI functionality.

- **The runtime layer** is a DLL delivered and installed with NVIDIA drivers.

During startup, an application should call `NvAPI_Initialize()` to set up the library before calling any other NVAPI functions.

2.2.2 Return Values

Almost all NVAPI functions return a value of type `NvAPI_Status`.

- If no error occurs, the return value is `NVAPI_OK`.
- If the function is not available with the currently installed driver, the return value is `NVAPI_NO_IMPLEMENTATION`.

Other return values are defined in `nvapi.h`.

2.2.3 Threading

NVAPI functions generally do not perform any automatic synchronization. If an application is calling NVAPI functions from multiple threads, the application is responsible for synchronizing the calls as necessary to ensure that two threads cannot be in NVAPI at the same time.

2.3 GENERAL INTERACTIONS WITH D3D11

Wherever possible, the `ID3D11MultiGPUDevice` interface has been designed to behave like a direct extension of an `ID3D11Device`. However, in some instances this behavior is not feasible or would result in an inefficient implementation.

2.3.1 Statelessness

By design, `ID3D11MultiGPUDevice` appears stateless to users. Therefore, it does not offer calls for getting multi-GPU state. D3D methods such as `CSGetConstantBuffers()` and `RSGetViewports()` return only the GPU0 state that D3D itself is aware of. Any application that requires tracking of API state must handle the tracking of multi-GPU API state itself.

2.3.2 Reference Counting

Like D3D, `ID3D11MultiGPUDevice` holds references to any constant buffers currently bound to the pipeline across all GPUs, and releases them when they are unbound. However, due to implementation constraints, in some cases the `Release()` calls do not

happen immediately when constant buffers are unbound. They may be deferred to the next call to `ID3D11MultiGPUDevice::SetConstantBuffers()`. Specifically, the calls are deferred when constant buffers that were bound by using `ID3D11MultiGPUDevice` are unbound by using the standard D3D11 API.

The result is that the multi-GPU API sometimes keeps references to a few constant buffers for longer than you might expect. Generally, you do not need to worry about this subtlety in the behavior of the API, and it is described here only for completeness.

2.3.3 Top-Level D3D11 API

The top-level API for VR SLI provides the interface for querying and initializing multi-GPU functionality. This portion of the API is designed to be used at application initialization. After initialization, an `ID3D11MultiGPUDevice` object created through this interface is used for the actual rendering calls.

Chapter 3.
API REFERENCE

3.1 STRUCTURES

3.1.1 NV_MULTIGPU_CAPS

```
typedef struct
{
      NvU32 multiGPUVersion;
      NvU32 reserved;
      NvU32 nTotalGPUs;
      NvU32 nSLIGPUs;
      NvU32 videoBridgePresent;
} NV_MULTIGPU_CAPS, *PNV_MULTIGPU_CAPS;
```

3.1.1.1 Members

`multiGPUVersion`

Type: `NvU32`

The version of the Multi-GPU functionality supported on the system.

`reserved`

Type: `NvU32`

Reserved for future use.

`nTotalGPUs`

Type: `NvU32`

The number of NVIDIA GPUs in the system.

`nSLIGPUs`

Type: `NvU32`

The number of GPUs participating in the SLI group. Generally, it is the number of GPUs the user can expect to be available for access through the Multi-GPU interface.

`videoBridgePresent`

Type: `NvU32`

Whether the system supports using a video bridge for transferring data between GPUs and the screen.

3.1.1.2 Remarks

The `NV_MULTIGPU_CAPS` structure provides information about the supported capabilities of the system. The data can be obtained by calling `NvAPI_D3D11_MultiGPU_GetCaps()`. This data can be queried before initializing multi-GPU support to allow the application to determine how to proceed.

3.2 FUNCTIONS

3.2.1 NvAPI_D3D11_MultiGPU_GetCaps

```
NVAPI_INTERFACE NvAPI_D3D11_MultiGPU_GetCaps (
     [out] NV_MULTIGPU_CAPS *pMultiGPUCaps
);
```

3.2.1.1 Parameters

`pMultiGPUCaps [out]`

Type: `NV_MULTIGPU_CAPS*`

A pointer to the structure that holds the query about the capabilities of the system.

3.2.1.2 Return Value

- Returns `NVAPI_OK` on success.
- Returns `NVAPI_INVALID_ARGUMENT` if `pMultiGPUCaps` is `NULL`.

3.2.1.3 Remarks

This function fills the provided `NV_MULTIGPU_CAPS` structure with the current system capabilities.

3.2.2 NvAPI_D3D11_MultiGPU_Init

```
NVAPI_INTERFACE NvAPI_D3D11_MultiGPU_Init(
     [in]  bool bEnable
);
```

3.2.2.1 Parameters

`bEnable [in]`

Type: `bool`

Whether to enable or disable support for the multi-GPU API in D3D devices created subsequently.

3.2.2.2 Return Value

Returns `NVAPI_OK` on success.

3.2.2.3 Remarks

This function initializes and configures the API. It takes a Boolean parameter that indicates whether to turn the API on or off.

- If `true`, the application specifies to the driver that it will use explicit multi-GPU, as opposed to relying on SLI profiles.
- If `false`, the application will use the default (automatic) SLI behavior.

To determine an application's one-time configuration settings, `GetCaps` can be called before `Init` to examine the system's capabilities.

An application must call `Init` before creating the D3D11 device. The result of the call to `Init` applies to all subsequent D3D11 devices created by the current process. Previously created devices are unaffected and continue to work normally.

3.2.3 NvAPI_D3D11_CreateMultiGPUDevice

```
NVAPI_INTERFACE NvAPI_D3D11_CreateMultiGPUDevice(
     [in]              ID3D11Device *pDevice,
     [in]              ULONG version,
     [out]             ULONG *currentVersion,
     [out]             ID3D11MultiGPUDevice **ppD3D11MultiGPUDevice,
     [in, optional]    UINT maxGpus = ALL_GPUS
);
```

3.2.3.1 Parameters

`pDevice [in]`

Type: `ID3D11Device*`

A pointer to the D3D device for which to create an `ID3D11MultiGPUDevice`.

`version [in]`

Type: `ULONG`

The version of `ID3D11MultiGPUDevice` to create. The version must be less than or equal to the version supported by installed driver.

This version is `ID3D11MultiGPUDevice_VER` defined in the SDK that corresponds to the installed driver. For more information, see `nvapi.h`.

`currentVersion [out]`

Type: `ULONG*`

Returns the version of the `ID3D11MultiGPUDevice` implemented in the installed driver. This parameter cannot be `NULL`.

`ppD3D11MultiGPUDevice [out]`

Type: `ID3D11MultiGPUDevice**`

Returns a pointer to the `ID3D11MultiGPUDevice` object created. This parameter cannot be `NULL`.

`maxGpus [in, optional]`

Type: `UINT`

The maximum number of GPUs that the application will use. This parameter can be used to restrict the number of GPUs to fewer than all the GPUs configured for SLI on the system. For example, if an application is designed to use two GPUs, but is run on a system with three or four GPUs, unexpected behavior could

result, such as the application mistakenly broadcasting commands to GPUs it doesn't intend to use. To avoid this unexpected behavior, the application can set this parameter to 2. Then, only the first two GPUs will be available to the application, and all operations will automatically be restricted to those GPUs.

If this parameter is set to `ALL_GPUS` (the default), all GPUs configured for SLI on the system will be available to the application.

3.2.3.2 Return Value

- Returns `NVAPI_OK` on success.
- Returns `NVAPI_INVALID_ARGUMENT` if `NvAPI_D3D11_MultiGPU_Init()` was not enabled before the creation of the D3D11 device, or if `maxGpus` is greater than the number of GPUs in the SLI group.
- Returns `NVAPI_NO_ACTIVE_SLI_TOPOLOGY` on single-GPU systems, or if SLI has not been enabled in the NVIDIA control panel.
- Returns `NVAPI_INVALID_CALL` if an `ID3D11MultiGPUDevice` is already created for the specified `ID3D11Device`.
- Returns `NVAPI_INCOMPATIBLE_STRUCT_VERSION` if the requested interface version is greater than the version that is supported by installed driver.

  In this case, `currentVersion` will contain the version supported by installed driver.

3.2.3.3 Remarks

This function creates an `ID3D11MultiGPUDevice` interface. The interface is tied to the specified `ID3D11Device`, and internally holds a reference to it.

Creating more than one `ID3D11MultiGPUDevice` for the same `ID3D11Device` isn't allowed. However, if you use multiple instances of `ID3D11Device` in an application, you can create a separate `ID3D11MultiGPUDevice` for each instance.

Because `currentVersion` will always be the latest version supported by the driver installed on the system, it is likely that `currentVersion` will return a value greater than the version you requested as required for your application.

3.3 ID3D11MULTIGPUDEVICE METHODS

The `ID3D11MultiGPUDevice` interface provides the methods used to manage multi-GPU rendering operations and state. Each `ID3D11MultiGPUDevice` is attached to a specific D3D device when it is created, and any D3D objects passed to its methods must be owned by that same D3D device. Note that `ID3D11MultiGPUDevice` is **not** a COM interface, so it lacks reference-counting and interface query functionality. The `Destroy()` method will immediately free the interface and any references it holds.

Most methods of `ID3D11MultiGPUDevice` take an `ID3D11DeviceContext*` to specify on which context the operation should be performed. The interaction with this context should be considered as altering that context, in the same way as a calling a method on the context. As a result, the same thread-safety rules from D3D apply: The application is responsible for ensuring that only one thread at a time operates on the context, either through the context's own methods or through `ID3D11MultiGPUDevice` methods.

Additionally, all methods that do not accept an `ID3D11DeviceContext*` except `Destroy()` can be considered to implicitly operate on the immediate context of the D3D11 device on which the `ID3D11MultiGPUDevice` was created. This behavior means that calls to such methods as `SetGPUMask()` can be thought of as operating as if they took the immediate mode context as an argument. This behavior governs both the sequencing of operations and the multithreading rules.

If an application uses multiple D3D devices and is intended to use multi-GPU functionality on them, it can create a separate `ID3D11MultiGPUDevice` interface for each D3D device.

3.3.1 SetGPUMask

```
UINT SetGPUMask(
     [in]  UINT GPUMask
);
```

3.3.1.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER1`

3.3.1.2 Parameters

`GPUMask [in]`

Type: `UINT`

A bitmask specifying which GPUs should be considered active for subsequent action commands. See Remarks for additional details.

3.3.1.3 Return value

Returns the previous GPU mask.

3.3.1.4 Remarks

This method sets the active GPU mask. All subsequent commands that perform an action such as clear, draw, or dispatch apply only to those GPUs whose bits are set in the

mask. Additionally, any calls to all versions of `SetConstantBuffers()` will also only set the constant buffers for the active GPU.

Commands that set state and commands that begin or end a query are always applied to **all** GPUs, regardless of the value of this mask.

If any bits set in the mask are higher than the number of GPUs, an error is reported.

A mask value of zero disables masking. When the mask is set to 0, all commands are sent to all GPUs. The API contains the symbolic constant `NVAPI_ALL_GPUS` to reflect this special behavior.

This mask applies to the following D3D-level commands:

- D3D11.0 commands:
  - `ClearDepthStencilView`
  - `ClearRenderTargetView`
  - `ClearUnorderedAccessViewFloat`
  - `ClearUnorderedAccessViewUint`
  - `CopyResource`
  - `CopyStructureCount`
  - `CopySubresourceRegion`
  - `Dispatch`
  - `DispatchIndirect`
  - `Draw`
  - `DrawAuto`
  - `DrawIndexed`
  - `DrawIndexedInstanced`
  - `DrawIndexedInstancedIndirect`
  - `DrawInstanced`
  - `DrawInstancedIndirect`
  - `GenerateMips`
  - `ResolveSubresource`
  - `SetPredication`
  - `Unmap`
- D3D11.1 commands:
  - `ClearView`
  - `CopySubresourceRegion1`
  - `DiscardResource`
  - `DiscardView`
  - `DiscardView1`
- D3D11.2 commands:
  - `CopyTiles`

3.3.2 SetContextGPUMask

```
NvAPI_Status SetContextGPUMask(
     [in] ID3D11DeviceContext *pContext,
     [in]  UINT GPUMask
);
```

3.3.2.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER2`

3.3.2.2 Parameters

`pContext [in]`

Type: `ID3D11DeviceContext*`

The rendering context on which to set the GPU mask.

`GPUMask [in]`

Type: `UINT`

A bitmask specifying which GPUs should be considered active for subsequent action commands on the specified context.

3.3.2.3 Return Value

Returns `NVAPI_OK` on success.

3.3.2.4 Remarks

This method sets the GPU mask on the given device context. On the immediate context, this command is equivalent to `SetGPUMask()`. This command allows the GPU mask to be set on deferred contexts also. For details, see Remarks for `SetGPUMask()`.

3.3.3 SetConstantBuffers

```
NvAPI_Status VSSetConstantBuffers(
     [in]              ID3D11DeviceContext *pContext,
     [in]              UINT GPUMask,
     [in]              UINT StartSlot,
     [in]              UINT NumBuffers,
     [in, optional]    ID3D11Buffer *const *ppConstantBuffers,
     [in, optional]    const UINT *pFirstConstant = NULL,
     [in, optional]    const UINT *pNumConstants = NULL
);
```

```
NvAPI_Status PSSetConstantBuffers(
     [in]              ID3D11DeviceContext *pContext,
     [in]              UINT GPUMask,
     [in]              UINT StartSlot,
     [in]              UINT NumBuffers,
     [in, optional]    ID3D11Buffer *const *ppConstantBuffers,
     [in, optional]    const UINT *pFirstConstant = NULL,
     [in, optional]    const UINT *pNumConstants = NULL

);

NvAPI_Status GSSetConstantBuffers(
     [in]              ID3D11DeviceContext *pContext,
     [in]              UINT GPUMask,
     [in]              UINT StartSlot,
     [in]              UINT NumBuffers,
     [in, optional]    ID3D11Buffer *const *ppConstantBuffers,
     [in, optional]    const UINT *pFirstConstant = NULL,
     [in, optional]    const UINT *pNumConstants = NULL
);

NvAPI_Status HSSetConstantBuffers(
     [in]              ID3D11DeviceContext *pContext,
     [in]              UINT GPUMask,
     [in]              UINT StartSlot,
     [in]              UINT NumBuffers,
     [in, optional]    ID3D11Buffer *const *ppConstantBuffers,
     [in, optional]    const UINT *pFirstConstant = NULL,
     [in, optional]    const UINT *pNumConstants = NULL

);

NvAPI_Status DSSetConstantBuffers(
     [in]              ID3D11DeviceContext *pContext,
     [in]              UINT GPUMask,
     [in]              UINT StartSlot,
     [in]              UINT NumBuffers,
     [in, optional]    ID3D11Buffer *const *ppConstantBuffers,
     [in, optional]    const UINT *pFirstConstant = NULL,
     [in, optional]    const UINT *pNumConstants = NULL

);

NvAPI_Status CSSetConstantBuffers(
     [in]              ID3D11DeviceContext *pContext,
     [in]              UINT GPUMask,
     [in]              UINT StartSlot,
     [in]              UINT NumBuffers,
```

```
    [in, optional]    ID3D11Buffer *const *ppConstantBuffers,
    [in, optional]    const UINT *pFirstConstant = NULL,
    [in, optional]    const UINT *pNumConstants = NULL
);
```

3.3.3.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER1`

3.3.3.2 Parameters

`pContext [in]`

Type: `ID3D11DeviceContext*`

The rendering context in which to receive the change to constant buffer state.

`GPUMask [in]`

Type: `UINT`

A bit field indicating the GPUs for which the constant buffer state should be altered.

`StartSlot [in]`

Type: `UINT`

The first constant buffer slot to modify.

`NumBuffers [in]`

Type: `UINT`

The number of constant buffers to set.

`ppConstantBuffers [in, optional]`

Type: `ID3D11Buffer* const *`

A pointer to the array of constant buffers to apply to the current state of the GPUs listed in the mask. If the pointer is `NULL`, the bindings are cleared for all referenced constant buffers.

`pFirstConstant [in, optional]`

Type: `const UINT*`

An array of offsets into the constant buffers provided with the `ppConstantBuffers` parameter. These offsets specify where in the buffer the shader should consider index zero to be. The offset is measured in 16-byte constant slots, not in bytes. An offset of zero means that the constants start at the beginning of the buffer, and an offset of 16 means that the constants start 256

bytes (16, 16-byte slots) into the buffer. The value must be a multiple of 16. If the parameter is `NULL`, all offsets are zero.

This parameter has a default value of `NULL`, so it can be omitted. The ability to omit this parameter allows the function signature to closely match both the D3D11 `SetConstantBuffers()` function and the D3D11.1 `SetConstantBuffers1()` function.

`pNumConstants [in, optional]`

Type: const `UINT*`

An array that holds the number of constants that each element in `ppConstantBuffers` specifies. If this argument is `NULL`, each constant buffer is understood to be bound in its entirety. This argument may be `NULL` only if `pFirstConstant` is also `NULL`.

This parameter has a default value of `NULL`, so it can be excluded. The ability to omit this parameter allows the function signature to closely match both the D3D11 `SetConstantBuffers()` function and the D3D11.1 `SetConstantBuffers1()` function.

3.3.3.3 Return Value

- Returns `NVAPI_OK` on success.
- Returns any NVAPI error if an error occurs.

3.3.3.4 Remarks

These members are multi-GPU analogues of the D3D11 `SetConstantBuffers()` and D3D11.1 `SetConstantBuffers1()` function calls. Instead of providing a single set of constant buffers for all active GPUs, these members allow independent configuration of constant buffers across all GPUs in the group. The constant buffers in the array are ordered so that the first `NumBuffers` elements go to the first GPU with a bit set in the mask, the next `NumBuffers` elements go to the next GPU in the mask, and so on. The following pseudocode demonstrates the operation in the D3D11 case.

```
// Multiview Implementation of constant buffer setup
void CSSetConstantBuffers(
ID3D11DeviceContext *pContext,
UINT GPUMask,
UINT StartSlot,
UINT NumBuffers,
ID3D11Buffer *const *ppConstantBuffers
)
{
  UINT SavedGPUMask = CurrentGPUMask;
  ID3D11Buffer *const *ppBuffers = ppConstantBuffers;
```

```
  // iterate over all GPUs in the multiview setup
  for (UINT gpu = 0; gpu < NumMultiViewGPUs; gpu++)
  {
    // if the mask bit for this GPU was set, set constant buffers for
it
    if ((GPUMask >> gpu) & 0x1)
    {
      // Set the GPU mask for just the current GPU
      SetGPUMask(0x1 << gpu);

      pContext->CSSetConstantBuffers(StartSlot, NumBuffers, ppBuffers);

      // if the buffer array is non-NULL, advance by the number of
buffers
      if (ppBuffers != NULL)
        ppBuffers += NumBuffers;
    }
  }

  // Return the GPU mask to its previous value
  SetGPUMask(SavedGPUMask);
}
```

As shown by the pseudocode, this function primarily serves as an optimization where all constant buffers for all GPUs can be set to different values with a single call rather than having to iterate over all GPUs. It should be thought of as similar to setting a batch of constant buffers rather than making one call for each constant buffer to set.

The `ppConstantBuffers`, `pFirstConstant`, and `pNumConstants` arrays must all contain `NumBuffers * count_bits(GPUMask)` elements when the parameters are not `NULL`.

Note that if any of the ordinary D3D11 `SetConstantBuffers()` or D3D11.1 `SetConstantBuffers1()` methods are called, they set the constant buffers for all GPUs, regardless of the current `SetGPUMask()` setting.

3.3.4 SetViewports

```
NvAPI_Status SetViewports(
     [in]  ID3D11DeviceContext *pContext,
     [in]  UINT GPUMask,
     [in]  UINT NumViewports,
     [in]  const D3D11_VIEWPORT *pViewports
);
```

3.3.4.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER1`

3.3.4.2 Parameters

`pContext [in]`

Type: `ID3D11DeviceContext*`

The context on which to apply the viewport state for individual GPUs.

`GPUMask [in]`

Type: `UINT`

A bit field indicating the GPUs on which the viewport state should be changed.

`NumViewports [in]`

Type: `UINT`

The number of viewports to set for each GPU identified in `GPUMask`.

`pViewports [in]`

Type: `D3D11_VIEWPORT*`

An array of viewport structures to set to the GPUs. The array should contain `NumViewports * count_bits(GPUMask)` elements.

3.3.4.3 Return Value

Returns `NVAPI_OK` on success.

3.3.4.4 Remarks

This member behaves like `RSSetViewports()`, except that it allows for setting different viewports for each GPU. The `pViewports` array is ordered so that the first `NumViewports` elements are the viewports for the first GPU with a bit set in the mask, and so on. For more information, see the pseudocode in `SetConstantBuffers`. GPUs for which the corresponding `GPUMask` bit is not set will retain their prior viewport state.

Note that if the ordinary D3D11 `RSSetViewports()` method is called, it sets the viewports for all GPUs, regardless of the current `SetGPUMask()` setting.

3.3.5 SetScissorRects

```
NvAPI_Status SetScissorRects(
     [in]  ID3D11DeviceContext *pContext,
     [in]  UINT GPUMask,
     [in]  UINT NumRects,
     [in]  const D3D11_RECT *pRects
);
```

3.3.5.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER1`

3.3.5.2 Parameters

`pContext [in]`

Type: `ID3D11DeviceContext*`

The context on which to apply the scissor rectangles for individual GPUs.

`GPUMask [in]`

Type: `UINT`

A bit field indicating the GPUs on which the scissor state should be changed.

`NumRects [in]`

Type: `UINT`

Number of viewports to set for each GPU identified in `GPUMask`.

`pRects [in]`

Type: `D3D11_RECT*`

An array of scissor rectangles to set. The array should contain `NumRects * count_bits(GPUMask)` elements.

3.3.5.3 Return Value

Returns `NVAPI_OK` on success.

3.3.5.4 Remarks

This method sets different scissor rectangles for each GPU in the system. It accepts a mask indicating which GPUs' scissor rectangles should be updated. This mask is independent of the `SetGPUMask()` setting. This method accepts an array of `D3D11_RECT` structures containing all the scissor rectangles being set for the first GPU with a bit set in the mask, followed by the scissor rectangles for the second GPU in the mask, and so on.

Note that if the ordinary D3D11 `RSSetScissorRects()`method is called, it sets the viewports for **all** GPUs, regardless of the `SetGPUMask()` setting.

3.3.6 GetData

```
HRESULT GetData(
     [in]              ID3D11DeviceContext *pContext,
     [in]              ID3D11Asynchronous *pAsync,
     [in]              UINT GPUIndex,
     [out, optional]   void *pData,
     [in]              UINT DataSize,
     [in]              UINT GetDataFlags
);
```

3.3.6.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER1`

3.3.6.2 Parameters

`pContext [in]`

Type: `ID3D11DeviceContext*`

The context to use to query the data.

`pAsync [in]`

Type: `ID3D11Asynchronous*`

The object to query data from.

`GPUIndex [in]`

Type: `UINT`

The GPU from which the data should be queried.

`pData [out, optional]`

Type: `void*`

A pointer to the memory used to store the results from the object. If this parameter is `NULL`, no results are provided, but the status of the object (available or not) is returned.

`DataSize [in]`

Type: `UINT`

The size in bytes of the memory backing the pointer `pData`.

`GetDataFlags [in]`

Type: `UINT`

Flags used to control the operation of the `GetData()` method.

3.3.6.3 Return Value

Type: `HRESULT`

- If the data is ready, `S_OK` is returned.
- If the data is not ready, `S_FALSE` is returned.
- If `GPUIndex` is not a valid index, `ERROR_INVALID_PARAMETER` is returned.

3.3.6.4 Remarks

This method provides the functionality of `ID3D11DeviceContext::GetData()`, with the ability to specify the GPU from which to get data. The standard `D3D11 GetData()` method returns only the result from GPU 0.

An example use of this method when rendering stereo views would be to check the results of an occlusion query in both eyes. You could then use the sum or maximum of the two values for culling and LOD computations.

3.3.7 CopySubresourceRegion

```
NvAPI_Status CopySubresourceRegion(
     [in]              ID3D11DeviceContext *pContext,
     [in]              ID3D11Resource *pDstResource,
     [in]              UINT DstSubresource,
     [in]              UINT DstGPUIndex,
     [in]              UINT DstX,
     [in]              UINT DstY,
     [in]              UINT DstZ,
     [in]              ID3D11Resource *pSrcResource,
     [in]              UINT SrcSubresource,
     [in]              UINT SrcGPUIndex,
     [in]              const D3D11_BOX *pSrcBox,
     [in, optional]    UINT ExtendedFlags = 0
);
```

3.3.7.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER1`

3.3.7.2 Parameters

`pContext [in]`

Type: `ID3D11DeviceContext*`

The context on which to perform the copy.

`pDstResource [in]`

Type: `ID3D11Resource*`

A pointer to the destination resource.

`DstSubresource [in]`

Type: `UINT`

The subresource index of the destination, such as the mipmap level.

`DstGPUIndex [in]`

Type: `UINT`

The index of the GPU to receive the data.

`DstX [in]`

Type: `UINT`

The x-coordinate of the upper left corner of the destination region.

`DstY [in]`

Type: `UINT`

The y-coordinate of the upper left corner of the destination region. This value must be zero if the resource is one-dimensional.

`DstZ [in]`

Type: `UINT`

The z-coordinate of the upper left corner of the destination region. This value must be zero if the resource is one-dimensional or two-dimensional.

`pSrcResource [in]`

Type: `ID3D11Resource*`

A pointer to the source resource.

`SrcSubresource [in]`

Type: `UINT`

The subresource index of the source, such as the mipmap level.

`SrcGPUIndex [in]`

Type: `UINT`

The index of the GPU from which to copy the data.

`pSrcBox [in, optional]`

Type: `D3D11_BOX*`

The region of the source resource to copy into the destination resource. If the pointer is `NULL`, the entire subresource is to be copied into the destination resource.

`ExtendedFlags [in, optional]`

Type: `UINT`

Accepts flags specific to multi-GPU operation. The only accepted flag in this release is `NVAPI_COPY_ASYNCHRONOUSLY`. If asynchronous copying is specified, the copy proceeds in parallel with other commands in the rendering streams of all GPUs, and the application must use fences to ensure proper ordering.

3.3.7.3 Return Value

Returns `NVAPI_OK` on success.

3.3.7.4 Remarks

This method copies data between two GPUs, or between a GPU and system memory. It accepts the same parameters as the D3D11 `CopySubresourceRegion()` method, but also includes explicit source and destination GPU indices. For example, you can copy data rendered on GPU1 back to GPU0. You can also issue two calls to copy both directions, ensuring that GPU0 and GPU1 have the same data going forward.

This method also enables copying of data between CPU and GPU resources. For example, it can copy data from a specified GPU's instance of a default resource (in `vidmem`) to a staging resource (in `sysmem`). In this case, you must pass `NVAPI_CPU_RESOURCE` as the destination GPU index. Similarly, you can also copy data from a dynamic resource (in `sysmem`) to a default resource (in `vidmem`). In this case, `NVAPI_CPU_RESOURCE` should be passed for the source GPU index.

The copy is done by using the GPU copy engine, which runs asynchronously with 3D rendering. By default, this method automatically performs synchronization: The two GPUs involved wait for each other before the copy, and any subsequent rendering operations that touch the source or destination resources will wait for the copy to finish. However, if the `NVAPI_COPY_ASYNCHRONOUSLY` flag is provided, automatic synchronization is disabled. In that case, the user is responsible for placing appropriate fences to ensure that the copy doesn't overlap with dependent rendering.

Note that if the user maps a CPU resource after issuing a copy to it or from it, `Map()`behaves normally, that is, it waits for the copy to finish if required by the map type and flags.

To be copied across GPUs, a resource must have been created with at least one enabled bind flag other than `D3D11_BIND_SHADER_RESOURCE`.

3.3.8 CopySubresourceRegion1

```
NvAPI_Status CopySubresourceRegion1(
     [in]              ID3D11DeviceContext1 *pContext,
     [in]              ID3D11Resource *pDstResource,
     [in]              UINT DstSubresource,
     [in]              UINT DstGPUIndex,
     [in]              UINT DstX,
     [in]              UINT DstY,
     [in]              UINT DstZ,
     [in]              ID3D11Resource *pSrcResource,
     [in]              UINT SrcSubresource,
     [in]              UINT SrcGPUIndex,
     [in]              const D3D11_BOX *pSrcBox,
     [in]              UINT CopyFlags,
     [in, optional]    UINT ExtendedFlags = 0
);
```

3.3.8.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER1`

3.3.8.2 Parameters

`pContext [in]`

Type: `ID3D11DeviceContext1*`

The context on which to perform the copy.

`pDstResource [in]`

Type: `ID3D11Resource*`

A pointer to the destination resource.

`DstSubresource [in]`

Type: `UINT`

The subresource index of the destination, such as the mipmap level.

`DstGPUIndex [in]`

Type: `UINT`

The index of the GPU to receive the data.

`DstX [in]`

Type: `UINT`

The x-coordinate of the upper left corner of the destination region.

`DstY [in]`

Type: `UINT`

The y-coordinate of the upper left corner of the destination region. This value must be zero if the resource is one-dimensional.

`DstZ [in]`

Type: `UINT`

The z-coordinate of the upper left corner of the destination region. This value must be zero if the resource is one-dimensional or two-dimensional.

`pSrcResource [in]`

Type: `ID3D11Resource*`

A pointer to the source resource.

`SrcSubresource [in]`

Type: `UINT`

The subresource index of the source, such as the mipmap level.

`SrcGPUIndex [in]`

Type: `UINT`

The index of the GPU to receive the data.

`pSrcBox [in, optional]`

Type: `D3D11_BOX*`

The region of the source resource to copy into the destination resource. If the pointer is `NULL`, the entire subresource is to be copied into the destination resource.

`CopyFlags [in]`

Type: `UINT`

Accepts any flags allowed by `CopySubresourceRegion1` in D3D11.1.

`ExtendedFlags [in]`

Type: `UINT`

Accepts flags specific to multi-GPU operation. The only accepted flag in this release is `NVAPI_COPY_ASYNCHRONOUSLY`. If asynchronous copying is specified, the copy proceeds in parallel with other commands in the rendering streams of all GPUs, and the application must use fences to ensure proper ordering.

3.3.8.3 Return Value

Returns `NVAPI_OK` on success.

3.3.8.4 Remarks

This method copies data between two GPUs, or between a GPU and system memory. It accepts the same parameters as the D3D11 `CopySubresourceRegion1()` method, but also includes explicit source and destination GPU indices. For example, you can copy data rendered on GPU1 back to GPU0. You can also issue two calls to copy both directions, ensuring that GPU0 and GPU1 have the same data going forward.

This method also enables copying data between CPU and GPU resources. For example, it can copy data from a specified GPU's instance of a default resource (in `vidmem`) to a staging resource (in `sysmem`). In this case, you must pass `NVAPI_CPU_RESOURCE` as the destination GPU index. Similarly, you can also copy data from a dynamic resource (in `sysmem`) to a default resource (in `vidmem`). In this case, `NVAPI_CPU_RESOURCE` should be passed for the source GPU index.

The copy is done by using the GPU copy engine, which runs asynchronously with 3D rendering. By default, this method automatically performs synchronization: The two GPUs involved will wait for each other before the copy, and any subsequent rendering operations that touch the source or destination resources will wait for the copy to finish. However, if the `NVAPI_COPY_ASYNCHRONOUSLY` flag is provided, automatic

synchronization is disabled. In that case, the user is responsible for placing appropriate fences to ensure that the copy doesn't overlap with dependent rendering.

Note that if the user maps a CPU resource after issuing a copy to it or from it, `Map()`behaves normally, that is, it waits for the copy to finish if required by the map type and flags.

To be copied across GPUs, a resource must have been created with at least one enabled bind flag other than `D3D11_BIND_SHADER_RESOURCE`.

3.3.9 UpdateSubresource

```
NvAPI_Status UpdateSubresource(
     [in]              ID3D11DeviceContext *pContext,
     [in]              ID3D11Resource *pDstResource,
     [in]              UINT DstSubresource,
     [in]              UINT DstGPUIndex,
     [in, optional]    const D3D11_BOX *pDstBox,
     [in]              const void *pSrcData,
     [in]              UINT SrcRowPitch,
     [in]              UINT SrcDepthPitch
);
```

3.3.9.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER1`

3.3.9.2 Parameters

`pContext [in]`

Type: `ID3D11DeviceContext*`

The context on which to perform the update.

`pDstResource [in]`

Type: `ID3D11Resource*`

A pointer to the resource receiving the update.

`DstSubresource [in]`

Type: `UINT`

The subresource index of the destination, such as the mipmap level. For more information, see the D3D11 documentation for `D3D11CalcSubResource()`.

`DstGPUIndex [in]`

Type: `UINT`

The index of the GPU to receive the data.

`pDstBox [in, optional]`

Type: `const D3D11_BOX*`

A pointer to a box that defines the portion of the destination subresource into which to copy the resource data. Coordinates are in bytes for buffers and in texels for textures. If `pDstBox` is `NULL`, the update region is the entire subresource.

`pSrcData [in]`

Type: `const void*`

A pointer to the data to use for updating the resource.

`SrcRowPitch [in]`

Type: `UINT`

The size of one row in the source data.

`SrcDepthPitch [in]`

Type: `UINT`

The size of one slice in the source data.

3.3.9.3 Return Value

Returns `NVAPI_OK` on success.

3.3.9.4 Remarks

This method updates a specified GPU's instance of a resource. It provides all the functionality of the D3D11 `UpdateSubresource()` method, but also accepts an explicit GPU index.

This method operates only on default resources, and cannot operate on constant buffers. Calling this method on non-default resources or constant buffers results in an error. In the multi-GPU memory model, constant buffers and dynamic and staging resources aren't instanced across GPUs, so they should be updated by using the ordinary D3D11 `UpdateSubresource()` method.

Note that if the ordinary D3D11 `UpdateSubresource()` method is called on a default resource, it updates the data for **all** GPUs, regardless of the `SetGPUMask()` setting.

3.3.10 UpdateTiles

```
NvAPI_Status UpdateTiles(
     [in]  ID3D11DeviceContext2 *pContext2,
     [in]  ID3D11Resource *pDestTiledResource,
     [in]  UINT GPUMask,
     [in]  const D3D11_TILED_RESOURCE_COORDINATE
                 *pDestTileRegionStartCoordinate,
     [in]  const D3D11_TILE_REGION_SIZE *pDestTileRegionSize,
     [in]  const void *pSourceTileData,
     [in]  UINT Flags
);
```

3.3.10.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER1`

3.3.10.2 Parameters

`pContext2 [in]`

Type: `ID3D11DeviceContext2*`

The device context on which to update tiles.

`pDestTiledResource [in]`

Type: `ID3D11Resource*`

The tiled resource to accept the updates.

`GPUMask [in]`

Type: `UINT`

A bitmask describing the GPUs on which to update the tile data.

`pDestTileRegionStartCoordinate [in]`

Type: `const D3D11_TILED_RESOURCE_COORDINATE*`

A pointer to the coordinate defining the starting location for the update.

`pDestTileRegionSize [in]`

Type: `const D3D11_TILE_REGION_SIZE*`

A pointer to a structure describing the size of the region to update.

`pSourceTileData [in]`

Type: `const void*`

A pointer to data to be used to update the tiles.

`Flags [in]`

Type: `UINT`

For valid flag values, see the D3D11 documentation for `UpdateTiles()`.

3.3.10.3 Return value

Returns `NVAPI_OK` on success.

3.3.10.4 Remarks

This method updates a specified GPU's instance of a tiled resource. It is supported only on D3D11.2+ devices. It provides all the functionality of the D3D11.2 `UpdateTiles()` method, but also accepts an explicit GPU index.

Note that if the ordinary D3D11.2 `UpdateTiles()` method is called, it updates the data for **all** GPUs, regardless of the `SetGPUMask()` setting.

3.3.11 CreateFences

```
NvAPI_Status CreateFences(
     [in]  UINT count,
     [out] void **ppFences
);
```

3.3.11.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER1`

3.3.11.2 Parameters

`count [in]`

Type: `UINT`

The number of fences to create.

`ppFences [out]`

Type: `void**`

A pointer to the memory used to return the fences. The amount of memory must be at least `sizeof(void *) * count` bytes.

3.3.11.3 Return Value

Returns NVAPI_OK on success.

3.3.11.4 Remarks

This method accepts a count, and creates the requested number of fences. The fences are identified by opaque handles, represented as void* values. The handles are returned in an output array. Each fence represents a UINT64 value. The initial value of all the fences is zero.

3.3.12 SetFence

```
NvAPI_Status SetFence(
     [in]  UINT GPUIndex,
     [in]  void *hFence,
     [in]  UINT64 value
);
```

3.3.12.1 Earliest Supported Interface Version

ID3D11MultiGPUDevice_VER1

3.3.12.2 Parameters

GPUIndex [in]

Type: UINT

The GPU on which to record the fence.

hFence [in]

Type: void*

The fence object to use for recording the fence operation.

value [in]

Type: UINT64

The value to record to the fence object on completion.

3.3.12.3 Return Value

Returns NVAPI_OK on success.

3.3.12.4 Remarks

This method accepts a GPU index, which is the handle of a fence, and a `UINT64` value to write to the fence. It issues a command to the specified GPU to write the value to the fence after completing all previous rendering operations.

Note that setting a fence is a somewhat expensive operation. It causes the current GPU command packet to be flushed, followed by a fence-setting packet. Therefore, in a real-time setting, fences should not be used more than a few times per frame.

3.3.13 WaitForFence

```
NvAPI_Status WaitForFence(
     [in]  UINT GPUMask,
     [in]  void *hFence,
     [in]  UINT64 value
);
```

3.3.13.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER1`

3.3.13.2 Parameters

`GPUMask [in]`

Type: `UINT`

A bitmask specifying which GPUs should wait for the specified fence.

`hFence`

Type: `void*`

The fence on which to wait.

`value`

Type: `UINT64`

The value that the fence must reach to be marked as achieved.

3.3.13.3 Return Value

Returns `NVAPI_OK` on success.

3.3.13.4 Remarks

This method accepts a GPU mask, which is the handle of a fence, and a `UINT64` value to wait for. It issues commands to all GPUs enabled in the mask, irrespective of the

`SetGPUMask()` setting, to wait until the value of the fence is greater than or equal to the specified value. The function returns immediately on the CPU. The wait is entirely contained within the GPU's execution of its command stream.

Note that waiting for a fence is a somewhat expensive operation. It causes the current GPU command packet to be flushed, followed by a fence-waiting packet. Therefore, in a real-time setting, fences should not be used more than a few times per frame.

3.3.14 FreeFences

```
NvAPI_Status FreeFences(
     [in]  UINT count,
     [in]  void **ppFences
);
```

3.3.14.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER1`

3.3.14.2 Parameters

`count [in]`

Type: `UINT`

The number of fences to free.

`ppFences [in]`

Type: `void**`

A pointer to an array of the handles of the fences to release. Memory pointed to must be at least `sizeof(void*) * count` bytes.

3.3.14.3 Return Value

Returns `NVAPI_OK` on success.

3.3.14.4 Remarks

This method accepts an array of the handles of the fences to release. Fences are released immediately from the CPU. When the fences are released, they may no longer be used in subsequent calls to `SetFence()` or `WaitForFence()`. However, the fences are **not** released immediately from the GPU. Previously submitted `SetFence()` and `WaitForFence()` commands are guaranteed to complete.

3.3.15 PresentCompositingConfig

```
NvAPI_Status PresentCompositingConfig(
[in]       IDXGISwapChain *pSwapChain,
[in]       UINT GPUMask,
[in, optional]   const D3D11_RECT *pRects,
[in]       UINT flags
);
```

3.3.15.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER1`

3.3.15.2 Parameters

`pSwapChain [in]`

Type: `IDXGISwapChain*`

The swap chain for which to configure presentation composition.

`GPUMask [in]`

Type: `UINT`

A bitmask identifying which GPUs contribute to the composition. If the value is 0, automatic composition is disabled, and only the back buffer of GPU0 is presented.

`pRects [in, optional]`

Type: `const D3D11_RECT*`

A pointer to an array of `D3D11_RECT` structures identifying the regions of the back buffer for each GPU to composite into the presentation buffer. The array must contain `countbits(GPUMask)` elements. If `GPUMask` is zero, this parameter must be `NULL`.

`Flags [in]`

Type: `UINT`

Flags controlling optional compositing modes. This parameter is reserved for future use.

3.3.15.3 Return Value

Returns `NVAPI_OK` on success.

3.3.15.4 Remarks

This method configures inter-GPU compositing to happen automatically on the swap chain when the `Present()` method is called. The array of rectangles provides a subregion for each GPU with a bit set in `GPUMask`. When the swap chain is presented, each of these rectangles is automatically copied from the back buffer on its corresponding GPU to the same pixel location in the back buffer on GPU 0.

For example, to render one eye per GPU, you could configure two rectangles for the left half and right half of the back buffer. The left rectangle would be taken from the back buffer of GPU0, and the right rectangle from the back buffer GPU1. When presenting the swap chain, the driver automatically copies the right rectangle from GPU1 to GPU0 and then presents the composite image.

Note that you could also do the necessary copy operations manually by using `CopySubresourceRegion()`. This method is provided as a convenience, and as a placeholder for additional future functionality.

If presentation composition is disabled, the swap chain displays only the back buffer of GPU0.

3.3.16 Destroy

```
void Destroy();
```

3.3.16.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER1`

3.3.16.2 Parameters

None.

3.3.16.3 Return Value

None.

3.3.16.4 Remarks

This method immediately frees the `ID3D11MultiGPUInterface`, including the reference to the D3D device with which it was created. After `Destroy()` has been called, calling any member function of the interface will produce undefined results, including the possible termination of the application. It is important to free the interface, because it holds a reference to the D3D device from which it was created.

3.3.17 GetVideoBridgeStatus

```
void GetVideoBridgeStatus(
     [in] IUnknown *pSwapChain,
     [in] UINT* pVideoBridgeStatus
);
```

3.3.17.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER2`

3.3.17.2 Parameters

`pSwapChain [in]`

Type: `IUnknown*`

The swap chain for which status is requested.

`pVideoBridgeStatus [in]`

Type: `UINT*`

The address of the variable that will receive current bridge status, which can be one of the following values:

```
#define NVAPI_VIDEO_BRIDGE_STATUS_AVAILABLE       0
#define NVAPI_VIDEO_BRIDGE_STATUS_NOT_AVAILABLE   1
#define NVAPI_VIDEO_BRIDGE_STATUS_FAILED_ACCESS   2
#define NVAPI_VIDEO_BRIDGE_STATUS_UNKNOWN         3
```

3.3.17.3 Return Value

None.

3.3.17.4 Remarks

The meaning of each bridge status is as follows:

- `NVAPI_VIDEO_BRIDGE_STATUS_AVAILABLE` - The application is running in full-screen mode and the video bridge is available.
- `NVAPI_VIDEO_BRIDGE_STATUS_NOT_AVAILABLE` - The application is running in full-screen mode and the video bridge is not available.
- `NVAPI_VIDEO_BRIDGE_STATUS_FAILED_ACCESS` – The application is running in full-screen mode, but the video bridge status could not be accessed.
- `NVAPI_VIDEO_BRIDGE_STATUS_UNKNOWN` - The application is running in windowed mode and, therefore, video bridge usage is not allowed.

3.3.18 CreateMultiGPUConstantBuffer

```
NvAPI_Status CreateMultiGPUConstantBuffer (
     [in]        const D3D11_BUFFER_DESC *pDesc,
     [in, optional const D3D11_SUBRESOURCE_DATA **ppInitialData,
     [out]       ID3D11Buffer **ppBuffer
);
```

3.3.18.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER3`

3.3.18.2 Parameters

`pDesc [in]`

Type: `D3D11_BUFFER_DESC *`

A constant buffer description.

`ppInitialData [in_opt]`

Type: `D3D11_SUBRESOURCE_DATA**`

An optional pointer to the initial data of the constant buffer.

`ppBuffer [out]`

Type: `ID3D11Buffer**`

A pointer to the constant buffer interface.

3.3.18.3 Return Value

- Returns `NVAPI_OK` on success.
- Returns `NVAPI_INVALID_POINTER` if `pDesc` or `ppBuffer` is `NULL`.
- Returns `NVAPI_INVALID_ARGUMENT` if `pDesc->BindFlags` does not contain a `D3D11_BIND_CONSTANT_BUFFER` bit.

3.3.18.4 Remarks

This call creates constant buffers that share an API handle for individual GPUs. As a result, the buffers can be separately updated by using the GPU mask, but set simultaneously with different underlying contents for each GPU by using the methods described in SetMGPUConstantBuffers.

The number of underlying constant buffers allocated depends on number of GPUs in the SLI configuration.

The constant buffer handle created can be used **only** in `MultiGPUDevice` interface functions. Use of this handle in normal DX APIs is likely to cause errors.

3.3.19 ReleaseMultiGPUConstantBuffer

```
NvAPI_Status ReleaseMultiGPUConstantBuffer (
     [in] ID3D11Buffer *pBuffer
);
```

3.3.19.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER3`

3.3.19.2 Parameters

`pBuffer [out]`

Type: `ID3D11Buffer**`

A constant buffer interface.

3.3.19.3 Return Value

- Returns `NVAPI_OK` on success.
- Returns `NVAPI_INVALID_POINTER` if `pBuffer` is `NULL`.
- Returns `NVAPI_INVALID_ARGUMENT` if `pBuffer` is not `MultiGPUConstantBuffer`.

3.3.19.4 Remarks

This method releases the constant buffers for individual GPUs that were created when `pBuffer` was created by using the `CreateMultiGPUConstantBuffer` method.

This method is the only way to release a `MultiGPUConstantBuffer` handle. The normal `ID3D11Buffer::Release()` method will not work.

3.3.20 SetMGPUConstantBuffers

```
NvAPI_Status VSSetMGPUConstantBuffers(
     [in]              ID3D11DeviceContext *pContext,
     [in]              UINT StartSlot,
     [in]              UINT NumBuffers,
     [in]              ID3D11Buffer *const *ppMGPUConstantBuffers,
);
```

```
NvAPI_Status PSSetMGPUConstantBuffers(
     [in]              ID3D11DeviceContext *pContext,
     [in]              UINT StartSlot,
     [in]              UINT NumBuffers,
     [in]              ID3D11Buffer *const *ppMGPUConstantBuffers,

);

NvAPI_Status GSSetMGPUConstantBuffers(
     [in]              ID3D11DeviceContext *pContext,
     [in]              UINT StartSlot,
     [in]              UINT NumBuffers,
     [in]              ID3D11Buffer *const *ppMGPUConstantBuffers,
);

NvAPI_Status HSSetMGPUConstantBuffers(
     [in]              ID3D11DeviceContext *pContext,
     [in]              UINT StartSlot,
     [in]              UINT NumBuffers,
     [in]              ID3D11Buffer *const *ppMGPUConstantBuffers,
);

NvAPI_Status DSSetMGPUConstantBuffers(
     [in]              ID3D11DeviceContext *pContext,
     [in]              UINT StartSlot,
     [in]              UINT NumBuffers,
     [in]              ID3D11Buffer *const *ppMGPUConstantBuffers,

);

NvAPI_Status CSSetMGPUConstantBuffers(
     [in]              ID3D11DeviceContext *pContext,
     [in]              UINT StartSlot,
     [in]              UINT NumBuffers,
     [in]              ID3D11Buffer *const *ppMGPUConstantBuffers,
);
```

3.3.20.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER3`

3.3.20.2 Parameters

`pContext [in]`

Type: `ID3D11DeviceContext*`

The rendering context to receive the change to the constant buffer state.

`StartSlot [in]`

Type: `UINT`

The first constant buffer slot to modify.

`NumBuffers [in]`

Type: `UINT`

The number of constant buffers to set.

ppMGPUConstantBuffers [in]

Type: `ID3D11Buffer* const *`

An array of constant buffers to apply to the current state of the GPUs listed in the mask. If the pointer is `NULL`, the bindings are cleared for all referenced constant buffers.

3.3.20.3 Return value

- Returns `NVAPI_OK` on success.
- Returns any NVAPI error if an error occurs.

3.3.20.4 Remarks

These members are multi-GPU analogues of the D3D11 `SetConstantBuffers()` function. Constant buffers are set for those GPUs that are in the current GPU mask with corresponding per-GPU constant buffer.

The current GPU mask is set with the `SetGPUMask` method, which contains all GPUs in the SLI configuration by default.

The semantics of `XXSetMGPUConstantBuffers` interface calls are similar to the semantics of `XXSetConstantBuffers` interface calls. The difference is that in `XXSetConstantBuffers`, each GPU receives a unique constant buffer handle associated with it, while the `XXSetMGPUConstantBuffers` call takes only one handle, which already contains a predefined association of a unique constant buffer for each GPU. This approach provides a similar interface of a single handle backed by per-GPU copies for constant buffers as is provided for other resources, such as textures.

A `MultiGPU` constant buffer can be passed as the `XXSetConstantBuffers` parameter `ppConstantBuffers`. In this case, `XXSetConstantBuffers` sets the underlying constant buffer for a specific GPU if that GPU is enabled in the mask, but only for the GPU corresponding to its array entry. That is, if the `MultiGPU` constant buffer is array entry index *N*, the function reads the *N*th enabled bit in the mask, which we will call *M*, and then bind the associated underlying constant buffer for GPU *M* to GPU *M*. It is

possible to mix `MultiGPU` constant buffers and regular constant buffers in the `ppConstantBuffers` array argument.

3.3.21 UpdateConstantBuffer

```
NvAPI_Status UpdateConstantBuffer(
     [in]              ID3D11DeviceContext *pContext,
     [in]              ID3D11Buffer *pBuffer,
     [in]              const void *pSrcData,
     [in, optional]    UINT GPUMask,
);
```

3.3.21.1 Earliest Supported Interface Version

`ID3D11MultiGPUDevice_VER3`

3.3.21.2 Parameters

`pContext [in]`

Type: `ID3D11DeviceContext*`

The context on which to perform the update.

`pBuffer [in]`

Type: `ID3D11Buffer*`

A pointer to the constant buffer receiving the update.

`GPUMask [in optional]`

Type: `UINT`

A GPU mask that specifies which GPU constant buffer must be updated.

3.3.21.3 Return Value

- Returns `NVAPI_OK` on success.
- Returns `NVAPI_ERROR` if an error occurs.

3.3.21.4 Remarks

This method updates a specified GPU's instance of a constant buffer. It supports `D3D11_USAGE_DYNAMIC`.

Notice

HDMI

OpenCL

Trademarks

Copyright