GPUDIRECT FOR VIDEO

PG-06007-001_v05 | February 2013

Programmer's Guide

DOCUMENT CHANGE HISTORY

PG-06007-001_v05

Version	Date	Authors	Description of Change
01	September 14, 2011	AA, SM	Initial Release
02	December 22, 2011	AA, SM	•Updated Section 3.1 and Section 3.2 •Updated Sub-section 4.2.1
03	April 27, 2012	AA, SM	Updated Chapter 3 and Chapter 4
04	November 8, 2012	TT, SM	•Updated Section 3.3 •Minor edits
05	February 21, 2013	AA, SM	Updated Chapter 3 and Chapter 4

TABLE OF CONTENTS

–

LIST OF FIGURES

LIST OF TABLES

Chapter 1. INTRODUCTION

NVIDIA® GPUDirect for Video™ is a user mode API that allows vendors of video capture or output devices to facilitate low latency transfers of video and other data in to and out of NVIDIA graphics processing units (GPUs) for high performance processing using available graphics and compute libraries.

The GPUDirect for Video data transfer technique permits sharing of a lockable system memory buffer between video I/O device and the GPU and permits direct DMA transfers between system memory and GPU memory providing a CPU-independent way to transfer data in to and out of the GPU (Figure 1-1).

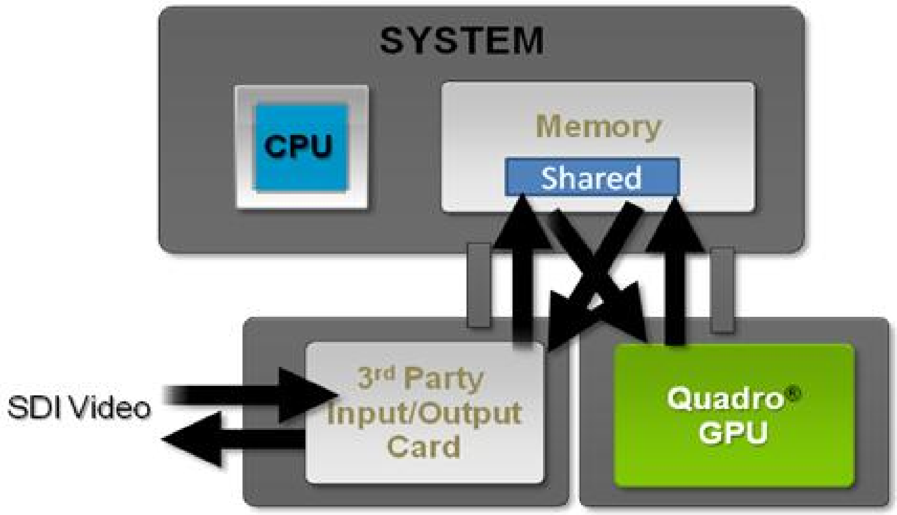

Figure 1-1. GPUDirect for Video Pipeline

This API is only exposed to video device vendors for use in device drivers, SDKs or applications. As a result, end user applications sitting on top of the vendor software stack and NVIDIA graphics and compute software stacks will automatically make use of the low latency transfers (Figure 1-2).

Note: It is up to the SDK vendor which graphics and compute APIs will be supported.

This programmer's guide describes programming methodologies for 3rd party video I/O driver/SDK developers who intend to integrate GPUDirect for Video I/O into their software.

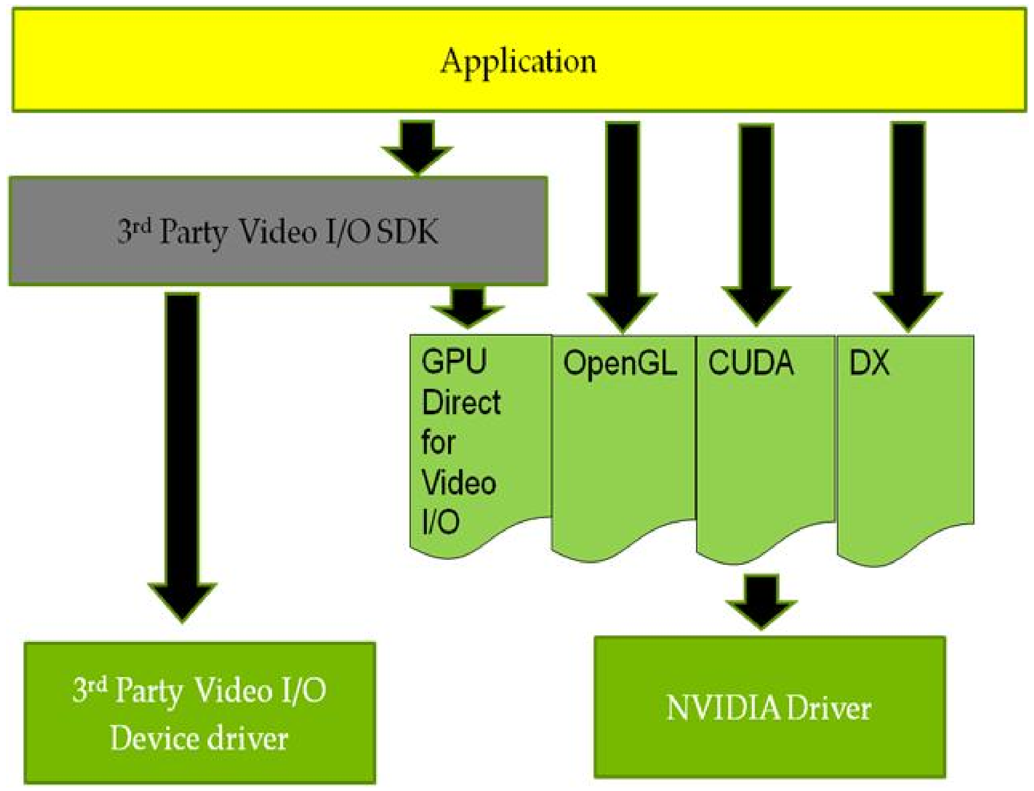

Figure 1-2. Architectural Overview

Chapter 2. END USER WORKFLOW

Following is a sample outline of an end user application taking advantage of the GPUDirect for Video while using 3rd party video I/O device SDK.

Note: : From now on the 3rd party video I/O driver/SDK API will be referred to as EXT SDK and all the EXT SDK function calls and data types will have the EXT prefix.

OpenGL/DirectX/CUDA will be referred to as GPU API.

The GPUDirect for Video I/O library will be referred to as the DVP library.

- Identify the GPU for the target and source of the data transfer in the EXT SDK.
- Create the GPU object(s) in the GPU API of choice (OpenGL, DirectX, NVIDIA CUDA®, etc) and the matching system memory that will facilitate the data transfers.
- Create an input or output EXT SDK ring buffer with the allocated GPU objects and the allocated system memory.
- Set up a loop where the application gets the next available GPU object from the EXT SDK processes the object and subsequently releases the object back to the EXT SDK.

Chapter 3. SETUP

This chapter details all the aspects of configuring GPUDirect for Video transfers within the EXT SDK.

3.1 INITIALIZATION

Prior to making any calls into the DVP library and allocating any library resources, the library must be initialized with a GPU API context/device. The EXT SDK should generally allow the end user application to specify these.

The allocation is performed with one of the `dvpInit*` calls and the corresponding de-allocation is done with its `dvpClose*` counterpart:

Note: Special care must be taken when the application initializes a DirectX device. The user must make sure that the device is initialized with multithreading. For D3D9, only an IDirect3DDevice9Ex can be used. For D3D10 and D3D11 the device must be created using D3D10CreateDeviceAndSwapChain and D3D11CreateDeviceAndSwapChain respectively.

```
DVPAPI_INTERFACE dvpInitGLContext(uint32_t flags);

DVPAPI_INTERFACE dvpInitCUDAContext(uint32_t flags);

DVPAPI_INTERFACE dvpInitD3D9Device(IDirect3DDevice9 *pD3D9Device,
                            uint32_t flags);
```

```
DVPAPI_INTERFACE dvpInitD3D10Device(ID3D11Device *pD3D11Device,
                             uint32_t flags);

DVPAPI_INTERFACE dvpInitD3D11Device(ID3D11Device *pD3D11Device,
                             uint32_t flags);
DVPAPI_INTERFACE dvpCloseGLContext();

DVPAPI_INTERFACE dvpCloseCUDAContext();

DVPAPI_INTERFACE dvpCloseD3D9Device(IDirect3DDevice9 *pD3D9Device);

DVPAPI_INTERFACE dvpCloseD3D10Device(ID3D11Device *pD3D11Device);

DVPAPI_INTERFACE dvpCloseD3D11Device(ID3D11Device *pD3D11Device);
```

Note: If OpenGL or CUDA are used, then the OpenGL and CUDA context must be current at the time of the `dvpInit*/dvpClose*` calls, and `dvpInit*/dvpClose*` must be called for each of the contexts that contain resources used by the library.

The `dvpInit*` call tells the library which context (for OpenGL or CUDA) or device (for DirectX) the GPU objects will reside within. The DVP library currently has two flags available: 0 or `DVP_DEVICE_FLAGS_SHARE_APP_CONTEXT`. If the value of `flags` is 0, then an internal context will be created, which shares all the resources with the current context. If the value of `flags` is `DVP_DEVICE_FLAGS_SHARE_APP_CONTEXT`, then the library will use whichever context is current at the time for a context dependent DVP function call.

The behavior of the library is different for every GPU API:

- For OpenGL, both flags are supported.
- For DirectX, there is no option of sharing the EXT SDK/application device and 0 is the only supported flag.
- For CUDA, sharing of the context is the only supported behavior and `DVP_DEVICE_FLAGS_SHARE_APP_CONTEXT` is the only supported flag.

3.2 SYSTEM MEMORY

Once the context/device has been established the EXT SDK needs to allocate system memory to be used for transferring video data in/out of the 3rd party device and the GPU.

Depending on the EXT SDK vendor, the memory can be allocated either by the EXT SDK or by the end user application and passed to the EXT SDK.

Each system memory buffer is required to fill out the `DVPSysmemBufferDesc` structure describing the dimensions and storage format of the buffer and allocate a `DVPBufferHandle` handle in the library.

```
typedef struct DVPSysmemBufferDescRec {
    uint32_t width;
    uint32_t height;
    uint32_t stride;
    uint32_t size;
    DVPBufferFormats format;
    DVPBufferTypes type;
    void *bufAddr;
} DVPSysmemBufferDesc;
```

For system memory buffers that correspond to a texture object on the GPU, the width, height, and stride fields will be used. And if a system memory buffer corresponds to a GPU buffer object, then the size field will be used.

Note: When performing a DMA between system memory and a 3rd party board, it is important to note that the buffer stride might have some padding at the end of each line for system memory buffers corresponding to GPU texture objects.

For optimal performance when graphics APIs are used, some forms of data translation can be performed by the driver during data transfer. These transformations are defined by the format and layout of the system memory and the format and layout of the GPU buffer defined by the graphics API. When possible, GPU hardware features will be used for these transformations.

Note: The driver will not perform any color space conversion as part of the data translation step. These must be done separately, in a shader or in a compute program.

`DVPBufferFormats` describe all the supported pixel formats and `DVPBufferTypes` describe all the supported pixel component storage type and bit layout of the system memory buffer.

```
typedef enum
 {
     DVP_BUFFER,
     DVP_DEPTH_COMPONENT,
     DVP_RGBA,
     DVP_BGRA,
     DVP_RED,
     DVP_GREEN,
     DVP_BLUE,
     DVP_ALPHA,
     DVP_RGB,
     DVP_BGR,
     DVP_LUMINANCE,
     DVP_LUMINANCE_ALPHA,
     DVP_CUDA_1_CHANNEL,
     DVP_CUDA_2_CHANNELS,
     DVP_CUDA_4_CHANNELS,
 } DVPBufferFormats;

typedef enum
 {
     DVP_UNSIGNED_BYTE,
     DVP_BYTE,
     DVP_UNSIGNED_SHORT,
     DVP_SHORT,
     DVP_UNSIGNED_INT,
     DVP_INT,
     DVP_FLOAT,
     DVP_HALF_FLOAT,
     DVP_UNSIGNED_BYTE_3_3_2,
     DVP_UNSIGNED_BYTE_2_3_3_REV,
     DVP_UNSIGNED_SHORT_5_6_5,
     DVP_UNSIGNED_SHORT_5_6_5_REV,
     DVP_UNSIGNED_SHORT_4_4_4_4,
     DVP_UNSIGNED_SHORT_4_4_4_4_REV,
     DVP_UNSIGNED_SHORT_5_5_5_1,
     DVP_UNSIGNED_SHORT_1_5_5_5_REV,
     DVP_UNSIGNED_INT_8_8_8_8,
     DVP_UNSIGNED_INT_8_8_8_8_REV,
     DVP_UNSIGNED_INT_10_10_10_2,
     DVP_UNSIGNED_INT_2_10_10_10_REV,
 } DVPBufferTypes;
```

Note: The GPU operates with data formats that are multiples of 8-bit. Therefore, textures with 12-bit pixel formats need to be packed into multiples of 8-bit words.

The following tables describe the bit layout of several buffer formats and types in the big endian bit order:

Table 3-1. 32-Bit Layout

Format	Type	1st Byte	2nd Byte	3rd Byte	4th Byte
RGBA	UNSIGNED_BYTE	RRRRRRRR	GGGGGGGG	BBBBBBBB	AAAAAAAA
	UNSIGNED_INT_8_8_8_8	RRRRRRRR	GGGGGGGG	BBBBBBBB	AAAAAAAA
	UNSIGNED_INT_8_8_8_8_REV	AAAAAAAA	BBBBBBBB	GGGGGGGG	RRRRRRRR
	UNSIGNED_INT_10_10_10_2	RRRRRRRR	RRGGGGGG	GGGGBBBB	BBBBBBAA
	UNSIGNED_INT_2_10_10_10_REV	AABBBBBB	BBBBGGGG	GGGGGGRR	RRRRRRRR
BGRA	UNSIGNED_BYTE	BBBBBBBB	GGGGGGGG	RRRRRRRR	AAAAAAAA
	UNSIGNED_INT_8_8_8_8	BBBBBBBB	GGGGGGGG	RRRRRRRR	AAAAAAAA
	UNSIGNED_INT_8_8_8_8_REV	AAAAAAAA	RRRRRRRR	GGGGGGGG	BBBBBBBB
	UNSIGNED_INT_10_10_10_2	BBBBBBBB	BBGGGGGG	GGGGRRRR	RRRRRRAA
	UNSIGNED_INT_2_10_10_10_REV	AARRRRRR	RRRRGGGG	GGGGGGBB	BBBBBBBB

Table 3-2. 24-Bit Layout

Format	Type	1st Byte	2nd Byte	3rd Byte
RGB	UNSIGNED_BYTE	RRRRRRRR	GGGGGGGG	BBBBBBBB
BGR	UNSIGNED_BYTE	BBBBBBBB	GGGGGGGG	RRRRRRRR

Table 3-3. 16-Bit Layout

Format	Type	1st Byte	2nd Byte
RGB	UNSIGNED_SHORT_5_6_5	RRRRRGGG	GGGBBBBB
	UNSIGNED_SHORT_5_6_5_REV	BBBBBGGG	GGGRRRRR
RGBA	UNSIGNED_SHORT_5_5_5_1	RRRRRGGG	GGBBBBBA
	UNSIGNED_SHORT_1_5_5_5_REV	ABBBBBGG	GGGRRRRR
	UNSIGNED_SHORT_4_4_4_4	RRRRGGGG	BBBBAAAA
	UNSIGNED_SHORT_4_4_4_4_REV	AAAABBBB	GGGGRRRR
BGRA	UNSIGNED_SHORT_5_5_5_1	BBBBBGGG	GGRRRRRA
	UNSIGNED_SHORT_1_5_5_5_REV	ARRRRRGG	GGGBBBBB
	UNSIGNED_SHORT_4_4_4_4	BBBBGGGG	RRRRAAAA
	UNSIGNED_SHORT_4_4_4_4_REV	AAAARRRR	GGGGBBBB

Table 3-4. 8-Bit Layout

Format	Type	1st Byte
RGB	UNSIGNED_SHORT_3_3_2	RRRGGGBB
	UNSIGNED_SHORT_3_3_2_REV	BBGGGRRR

The `DVP_BUFFER` buffer format provides an unspecified format type – in this case the buffer is copied to GPU memory without any interpretation of the stored bytes and shaders or compute programs must be used by the EXTSDK or the end user application to interpret the data.

Note: If `DVP_BUFFER` buffer format is used then the `DVPSysmemBufferDesc` size field must be correctly set. The width and height fields are ignored.

Once the `DVPSysmemBufferDesc` has been configured, the buffer must be registered with the DVP library using `dvpCreateBuffer` function, and prior to destruction it must be unregistered using the `dvpDestroyBuffer.`

```
DVPAPI_INTERFACE dvpCreateBuffer(DVPSysmemBufferDesc *desc,
                   DVPBufferHandle *hBuf);
DVPAPI_INTERFACE dvpDestroyBuffer(DVPBufferHandle  hBuf);
```

The buffer then must be bound to the GPU API of choice using one of `dvpBindTo*` functions:

```
DVPAPI_INTERFACE  dvpBindToGLCtx(DVPBufferHandle hBuf);

DVPAPI_INTERFACE dvpBindToCUDACtx(DVPBufferHandle hBuf);

DVPAPI_INTERFACE dvpBindToD3D9Device(DVPBufferHandle hBuf,
IDirect3DDevice9 *pD3D9Device);

DVPAPI_INTERFACE  dvpBindToD3D10Device(DVPBufferHandle hBuf,
ID3D10Device *pD3D10Device);

DVPAPI_INTERFACE  dvpBindToD3D11Device(DVPBufferHandle hBuf,
ID3D11Device *pD3D11Device);

 DVPAPI_INTERFACE
    dvpUnbindFromGLCtx(DVPBufferHandle hBuf);

 DVPAPI_INTERFACE
    dvpUnbindFromCUDACtx(DVPBufferHandle hBuf);
```

```
DVPAPI_INTERFACE
   dvpUnbindFromD3D9Device(DVPBufferHandle hBuf,
                           IDirect3DDevice9 *pD3D9Device);
DVPAPI_INTERFACE
   dvpUnbindFromD3D10Device(DVPBufferHandle hBuf,
                            ID3D10Device *pD3D10Device);
DVPAPI_INTERFACE
   dvpUnbindFromD3D11Device(DVPBufferHandle hBuf,
                            ID3D11Device *pD3D11Device);
```

Note: If OpenGL or CUDA are used, then the OpenGL and CUDA context must be current at the time of the `dvpBind*/dvpUnbind*` calls.

The DVP library constrains the address alignment to be used by the GPU DMA engines. These constraints tend to be architecture and driver specific as well as GPU API specific and can be queried using the `dvpGetRequiredConstantsXXX` call:

```
DVPAPI_INTERFACE
dvpGetRequiredConstantsGLCtx(uint32_t *bufferAddrAlignment,
                             uint32_t *bufferGPUStrideAlignment,
                             uint32_t *semaphoreAddrAlignment,
                             uint32_t *semaphoreAllocSize,
                             uint32_t *semaphorePayloadOffset,
                             uint32_t *semaphorePayloadSize);

DVPAPI_INTERFACE
dvpGetRequiredConstantsCUDACtx(uint32_t *bufferAddrAlignment,
                               uint32_t *bufferGPUStrideAlignment,
                               uint32_t *semaphoreAddrAlignment,
                               uint32_t *semaphoreAllocSize,
                               uint32_t *semaphorePayloadOffset,
                               uint32_t *semaphorePayloadSize);

DVPAPI_INTERFACE
dvpGetRequiredConstantsD3D9Device(uint32_t *bufferAddrAlignment,
                                  uint32_t *bufferGPUStrideAlignment,
                                  uint32_t *semaphoreAddrAlignment,
                                  uint32_t *semaphoreAllocSize,
                                  uint32_t *semaphorePayloadOffset,
                                  uint32_t *semaphorePayloadSize,
                                  IDirect3DDevice9 *pD3D9Device);

DVPAPI_INTERFACE
dvpGetRequiredConstantsD3D10Device(uint32_t *bufferAddrAlignment,
                                   uint32_t *bufferGPUStrideAlignment,
                                   uint32_t *semaphoreAddrAlignment,
                                   uint32_t *semaphoreAllocSize,
                                   uint32_t *semaphorePayloadOffset,
                                   uint32_t *semaphorePayloadSize,
                                   ID3D10Device *pD3D10Device);

DVPAPI_INTERFACE
dvpGetRequiredConstantsD3D11Device(uint32_t *bufferAddrAlignment,
                                   uint32_t *bufferGPUStrideAlignment,
                                   uint32_t *semaphoreAddrAlignment,
                                   uint32_t *semaphoreAllocSize,
                                   uint32_t *semaphorePayloadOffset,
                                   uint32_t *semaphorePayloadSize,
                                   ID3D11Device *pD3D11Device);
```

This call also provides an alignment recommendation for the stride of the system memory buffer. The performance of the data transfer could vary depending on the stride alignment so aligning the stride to certain values, for example, 64 bytes, may give optimal data rate. The value returned from `dvpGetRequiredConstantsXXX` will yield optimal performance, but smaller values may do the same. These values may vary from GPU to GPU, so testing for the optimal alignment is always recommended. This is important should the 3rd party hardware require a stride not aligned with the value from `dvpGetRequiredConstantsXXX`.

Note: If OpenGL or CUDA APIs are used, then the OpenGL and CUDA context must be current at the time of the `dvpGetRequiredConstantsXXX` call.

In addition, the allocated memory usually needs to be pinned in place for 3rd party device DMAs which can place additional restrictions on the allocations. Pinning can be done with the use of OS specific functions. On Windows, `MmProbeAndLockPages` or `VirtualLock` can be used. The application can increase the allowable amount of pinned memory if necessary by calling `SetProcessWorkingSetSize.`On Linux, `get_user_pages` will provide the necessary functionality.

The following code sample demonstrates how to setup a system memory buffer for DVP library usage on Linux. In this example the memory is setup for OpenGL usage as a 1920 × 1080 RGBA texture.

```
dvpGetRequiredConstantsGLCtx(&g_bufferAddrAlignment,
                                &g_bufferGPUStrideAlignment,
                                &g_semaphoreAddrAlignment,
                                &g_semaphoreAllocSize

                                &g_semaphorePayloadOffset
                                &g_semaphorePayloadSize);
DVPSysmemBufferDesc sysMemBuffersDesc;
```

```
//Use GPU constrains
bufferWidth = 1920;
bufferHeight = 1080;
bufferStride = bufferWidth*4;
bufferStride += g_bufferGPUStrideAlignment-1;
bufferStride &= ~(g_bufferGPUStrideAlignment-1);

sysMemBuffersDesc.width    = bufferWidth;
sysMemBuffersDesc.height   = bufferHeight;
sysMemBuffersDesc.stride   = bufferStride;
sysMemBuffersDesc.format   = DVP_RGBA;
sysMemBuffersDesc.type     = DVP_UNSIGNED_INT_8_8_8_8;
size = bufferHeight*bufferStride;
//allocate host memory
void     *sysMemAlloc, *sysMemBuffer;
sysMemAlloc=calloc(1, size + g_bufferAddrAlignment - 1);
//Align buffer
sysMemBuffer=(void *)(((uint64_t)sysMemAlloc+g_bufferAddrAlignment-1)
                        &  ~((uint64_t)g_bufferAddrAlignment - 1));
sysMemBuffersDesc.bufAddr = sysMemBuffer;
//allocate a library handle
DVPBufferHandle sysMemHandle;
dvpCreateBuffer(sysMemBuffersDesc, &sysMemHandle);
//bind the library handle to the GL context
dvpBindToGLCtx(sysMemHandle);
```

3.3 SYNCHRONIZATION OBJECTS

The DVP sync objects are used for bi-directional synchronization between the GPU and the 3rd party device. This way the 3rd device party SDK and driver knows when the GPU is done transferring data and the GPU knows when the data is ready to be transferred.

The sync objects defined in this API are implemented as increasing semaphores with two types of operations: a release and an acquire operation. In this document a semaphore refers to a shared memory primitive containing a 32-bit integer value. A release operation on a semaphore writes an integer value to the memory location, while an acquire operation blocks until the semaphore value is greater than or equal to a specified value.

To setup, the following `DVPSyncObjectDesc` structure must be initialized:

```
  typedef struct DVPSyncObjectDescRec {
       uint32_t *sem;
       uint32_t  flags;
       DVPStatus (*externalClientWaitFunc) (DVPSyncObjectHandle sync,
                                            uint32_t value,
                                            bool GEQ,
                                            uint64_t timeout);
   } DVPSyncObjectDesc;
```

DVP library provides two semaphore acquire methods for the EXT SDK.

```
    DVPAPI_INTERFACE
    dvpSyncObjClientWaitComplete(DVPSyncObjectHandle syncObject,
                                 uint64_t timeout);
    DVPAPI_INTERFACE
    dvpSyncObjClientWaitPartial(DVPSyncObjectHandle syncObject,
                                uint32_t value,
                                uint64_t timeout);
```

The object must be imported and then later released from the library using the following functions:

```

    DVPAPI_INTERFACE
    dvpImportSyncObject(DVPSyncObjectDesc *desc,
                              DVPSyncObjectHandle *syncObject);
    DVPAPI_INTERFACE
    dvpFreeSyncObject(DVPSyncObjectHandle syncObject);
```

The library provides a way to query sync object for the time of completion of the last GPU release operation. This time is in nanoseconds and it is in the GPU domain:

```

    DVPAPI_INTERFACE
         dvpSyncObjCompletion(DVPSyncObjectHandle syncObject,
                      uint64_t *timeStamp);
```

Note: If OpenGL is being used and the library was initialized with `DVP_DEVICE_FLAGS_SHARE_APP_CONTEXT`, then an OpenGL context must be current when calling `dvpSyncObjCompletion`. This call bounds the sync object to the context, and therefore `dvpFreeSyncObject` must also be called with each application context to unbind the object. Otherwise, if there are outstanding contexts from which the sync object needs to freed, `dvpFreeSyncObject` will return `DVP_STATUS_SYNC_STILL_BOUND`.

`dvpSyncObjClientWaitComplete` blocks until the corresponding sync object's semaphore is greater than or equal to the last release value issued by the DVP library whereas `dvpSyncObjClientWaitPartial` blocks until the semaphore value is greater than or equal to the supplied value. Both methods must be encapsulated in `dvpBegin()` and `dvpEnd()` calls.

Note: The first time `dvpSyncObjClientWaitComplete` function is called it will return `DVP_STATUS_INVALID_OPERATION` if the syncObject has yet to be used by the library. The `dvpSyncObjClientWait*` calls will silently fail if they are not encapsulated in `dvpBegin/dvpEnd` function calls.

When the flags field is set to `DVP_SYNC_OBJECT_FLAGS_USE_EVENTS` and the library gets a call to `dvpSyncObjClientWait*`, it will prefer to use native operating system event waits instead of spin-loops.

Note: GPU hardware level semaphore and wait operations are supported on Windows XP and Linux but not supported on Windows 7.

EXT SDK can specify a callback for the DVP library semaphore wait operations using `externalClientWaitFunc` function. This allows the application or the EXT SDK to create events, which can be triggered on device interrupts and consequently, these events can be waited upon instead of using spin loops inside the DVP library. This function should return `DVP_STATUS_OK` on success, non-zero for failure and `DVP_STATUS_TIMEOUT` on timeout.

The DVP library constrains the alignment and allocation size of the system memory devoted for the sync object in the same way it constrains the allocation properties of the system memory used for transfers. These constraints are also obtained during the `dvpGetRequiredConstantsXXX` call as it can be seen by the function call signature.

The following code sample demonstrates how to setup a DVP sync object and import it into the DVP library.

```
DVPSyncObjectDesc syncObjectDesc;
DVPSyncObjectHandle syncObj;
uint32_t semOrg = (uint32_t *)
malloc(g_semaphoreAllocSize+g_semaphoreAddrAlignment-1);
// Correct alignment
uint64_t val = (uint64_t)semOrg;
val += g_semaphoreAddrAlignment-1;
val &= ~(g_semaphoreAddrAlignment-1);

// setup the DVPSyncObjectDesc structure
syncObjectDesc.sem = sem;
syncObjectDesc.externalClientWaitFunc = NULL;
syncObjectDesc.flags = 0;
// Import the DVP Sync Object
dvpImportSyncObject(&syncObjectDesc, &syncObj);
```

The EXT SDK should allocate two sync objects for every buffer. Each sync object corresponds to a DMA operation: one for the video device DMA, and another for the GPU DMA.

Note: It is legal to use the same memory by two separate sync objects, but could yield unpredictable results.

Also, a sync object should not be used across GPU APIs. For example, it is illegal to use it for synchronization while doing data transfers with source or target GL texture and then use it for synchronization for data transfers with source or target CUDA buffer.

3.4 GPU OBJECTS

Once the end user application has finished setting up all the GPU objects (buffers or textures) which will be used as sources or targets of data transfers, it must pass them to the EXT SDK so they can be registered with the DVP library using one of the `dvpCreateGPU*` functions listed in the following code sample. Those objects must later be unregistered using the `dvpFreeBuffer` function.

```
DVPAPI_INTERFACE  dvpCreateGPUBufferGL(GLuint bufferID,
DVPBufferHandle *bufferHandle);

 DVPAPI_INTERFACE dvpCreateGPUTextureGL(GLuint texID,
DVPBufferHandle *bufferHandle);

 DVPAPI_INTERFACE dvpCreateGPUCUDAArray(CUarray array,
DVPBufferHandle *bufferHandle);

DVPAPI_INTERFACE  dvpCreateGPUCUDADevicePtr(CUdeviceptr devPtr,
DVPBufferHandle *bufferHandle);

 DVPAPI_INTERFACE  dvpCreateGPUD3D9Resource(IDirect3DResource9
*pD3DResource,   DVPBufferHandle *bufferHandle);

 DVPAPI_INTERFACE dvpCreateGPUD3D10Resource(ID3D10Resource
*pD3DResource,  DVPBufferHandle *bufferHandle);
```

```
 DVPAPI_INTERFACE dvpCreateGPUD3D11Resource(ID3D11Resource
*pD3DResource,  DVPBufferHandle *bufferHandle);

 DVPAPI_INTERFACE
    dvpFreeBuffer(DVPBufferHandle  hBuf);
```

Notes: If OpenGL or CUDA APIs are used, then the OpenGL and CUDA context must be current at the time of the dvpCreate* calls.

If DirectX API is used, then the texture or buffer resources must be created in `D3DPOOL_DEFAULT` memory pool for D3D9 and `D3D10_USAGE_DEFAULT` and `D3D11_USAGE_DEFAULT` for D3D10 and D3D11 respectively.

The following are the only DirectX resource types that are accepted:

```
For D3D9: IDirect3DSurface9, IDirect3DTexture9,
IDirect3DVolumeTexture9,IDirect3DCubeTexture9,
IDirect3DIndexBuffer9, IDirect3DVertexBuffer9

For D3D10:ID3D10Texture1D, ID3D10Texture2D, ID3D10Texture3D,
ID3D10Buffer

For D3D11:ID3D11Texture1D, ID3D11Texture2D, ID3D11Texture3D,
ID3D11Buffer
```

Chapter 4.
MECHANISM OF OPERATION

This chapter will delve into the DVP library transfers and various synchronization mechanisms.

4.1 DVP MEMORY COPY

As it was previously mentioned in this programmer's guide, data transfer between the shared system memory and the GPU happens as a DMA operation which gets scheduled when the EXT SDK uses one of the dvpMemcpy* calls. The following is a list of available dvpMemcpy* functions:

```
DVPAPI_INTERFACE  dvpMemcpy(DVPBufferHandle      srcBuffer,
      DVPSyncObjectHandle  srcSync,
      uint32_t             srcAcquireValue,
      uint64_t             timeout,
      DVPBufferHandle      dstBuffer,
      DVPSyncObjectHandle  dstSync,
      uint32_t             dstReleaseValue,
      uint32_t             srcOffset,
      uint32_t             dstOffset,
      uint32_t             count);

DVPAPI_INTERFACE dvpMemcpyLined(DVPBufferHandle      srcBuffer,
                   DVPSyncObjectHandle  srcSync,
                   uint32_t             srcAcquireValue,
                   uint64_t             timeout,
                   DVPBufferHandle      dstBuffer,
                   DVPSyncObjectHandle  dstSync,
                   uint32_t             dstReleaseValue,
                   uint32_t             startingLine,
                   uint32_t             numberOfLines);

DVPAPI_INTERFACE dvpMemcpy2D(DVPBufferHandle      srcBuffer,
                DVPSyncObjectHandle  srcSync,
                uint32_t             srcAcquireValue,
                uint64_t             timeout,
                DVPBufferHandle      dstBuffer,
                DVPSyncObjectHandle  dstSync,
                uint32_t             dstReleaseValue,
                uint32_t             startY,
                uint32_t             startX,
                uint32_t             height,
                uint32_t             width);
```

`dvpMemcpy` is designed to only work with a GPU API pure buffer and a DVP system buffer of `DVP_BUFFER` format, whereas `dvpMemcpyLined` and `dvpMemcpy2D` only work with a GPU API texture and a DVP system buffer of any format with the exception of `DVP_BUFFER`.

Note: GPU API texture refers to a texture for OpenGL and DirectX, and CUDA array for CUDA. GPU API pure buffer refers to a buffer object for OpenGL, an IDirect3DVertexBuffer9 for Direct X 9, a buffer resource for Direct X 10 and 11 and unformatted CUDA heap memory for CUDA.

4.2 SYNCHRONIZATION

GPUDirect for Video has two different synchronization mechanisms built into the API: The DVP sync objects and the `dvpMap*` APIs. Both mechanisms should be utilized by the EXT SDK to achieve correctness of data in the GPU APIs.

4.2.1 DVP Sync Objects

The DVP Sync Objects exist for synchronization between the NVIDIA driver and EXT SDK to synchronize DMA operations. It is preferable for performance that the acquire and the release operations for these objects are done in hardware, but the driver can also perform the operations on behalf of the hardware.

`dvpMemcpy*` APIs are used for copying data between system memory and the GPU memory. They take two DVP sync objects as arguments: `srcSync` and `dstSync`, and two pairs of values `acquireValue` and `releaseValue`.

`srcSync` should get signaled by the EXT device/driver when the DMA operation between the device and `sysmem` is completed. The GPU will wait until `srcSync` value reaches the `acquireValue`. `dstSync` will get signaled and its value will be set to `releaseValue` upon the GPU DMA operation completion. It is up to the EXT SDK to accurately keep track of the acquire and release values of the sync objects. The sync objects should be used in a similar fashion for the EXT `memcpy` operation.

The following is pseudo code for a `dvpMemcpy*` operation for capture:

```
WaitOnSemaphore(dstSync, acquireValue);
InitiateGPUMemcpy(sysmem, gpumem, ... );
```

The NVIDIA GPU will execute the following in pseudo code once the DMA is complete:

```
ReleaseSemaphore(dstSync, releaseValue);
```

A similar code outline should appear in the EXT SDK when scheduling of the 3rd party device DMA:

```
WaitOnSemaphore(srcSync, acquireValue);
InitiateEXTMemcpy(sysmem, devicemem, ... );
```

The EXT SDK should execute the following in pseudo code once the DMA is complete:

```
ReleaseSemaphore(srcSync, releaseValue);
```

4.2.2 dvpMap* APIs

`dvpMap*` APIs are used to synchronize DVP operations with application-level GPU API operations.

```
DVPAPI_INTERFACE
    dvpMapBufferEndDVP(DVPBufferHandle gpuBufferHandle);
DVPAPI_INTERFACE
    dvpMapBufferWaitDVP(DVPBufferHandle gpuBufferHandle);

DVPAPI_INTERFACE
    dvpMapBufferEndAPI(DVPBufferHandle gpuBufferHandle);
DVPAPI_INTERFACE
    dvpMapBufferWaitAPI(DVPBufferHandle gpuBufferHandle);

DVPAPI_INTERFACE
    dvpMapBufferEndCUDAStream(DVPBufferHandle gpuBufferHandle, CUstream
stream);
DVPAPI_INTERFACE
    dvpMapBufferWaitCUDAStream(DVPBufferHandle gpuBufferHandle,
CUstream stream);
```

The `dvpMapBufferEndAPI` call sets up a signal for a buffer in the callers GPU API context or device. The signal follows all previous GPU API operations up to this point, thus allowing subsequent DVP calls to know when this buffer is ready for use within the DVP library. `dvpMapBufferWaitDVP` allows the DVP command stream to wait for the recently set up signal to be triggered by the GPU API command stream.

Similarly, `dvpMapBufferEndDVP` call sets up a signal for a buffer in the DVP command stream, and this signal follows all previous DVP operations up to this point, thus allowing subsequent GPU API calls to know when this buffer is ready for use within the GPU API. `dvpMapBufferWaitAPI` allows the GPU API command stream to wait for the recently set up signal for the buffer to be triggered by the DVP command stream. For best performance, `dvpMapBufferEndCUDAStream/dvpMapBufferWaitCUDAStream` should be used instead of `dvpMapBufferEndAPI/dvpMapBufferWaitAPI` in CUDA. These calls allow the DVP library to synchronize with a particular computation stream instead of the default stream, thus providing a finer grained way of doing synchronization.

Note: Each set of `dvpMapBufferWaitDVP/dvpMapBufferEndDVP/dvpSyncObjClientWait*` and `dvpMemcpy*` function calls should be encapsulated in the `dvpBegin` / `dvpEnd` function calls. If there is a thread specifically devoted to GPU DMA in any one direction, it is enough to call dvpBegin in the beginning of this thread’s lifetime and dvpEnd at the end.

4.3 PERFORMANCE CONSIDERATIONS

- **Latency Reduction**
  Simply by using the DVP library for transfers, which uses shared system memory for GPU and 3rd party device DMA operations, the EXT SDK reduces latency by eliminating an expensive `memcpy` between the 3rd party device DMA-able system memory and GPU DMA-able system memory. Further improvements can be achieved by overlapping 3rd party device DMA with the GPU DMA. This can be done by implementing sub-frame DMA transfers in the EXT SDK and scheduling the `dvpMemcpy*` operations in sub-frame chunks.
  For 3rd party devices with significantly slower DMA operation than the GPU DMA speed, it's preferred to have smaller chunks so there is less time waiting for the device. For devices with DMA speeds matching that of the GPU, two chunks are recommended. However, it is always best to test to determine for the optimal number of chunks.
- **Threads of Execution**
  Since any DMA operation is designed to occur in a CPU asynchronous manner the EXT SDK most likely already provides ways of synchronizing this operation to the end user application. Therefore, in order to spare the application from coordinating additional DMA operations when using the GPU, the DVP library provides convenient ways for the EXT SDK to take on the responsibility of scheduling and coordinating these DMA operations.
  It is recommended for the EXT SDK to either initiate the GPU DMAs on its own, or provide an API to the application that will invoke the GPU DMAs in the same threads that are responsible for initiating the EXT device DMAs. By doing so, these threads will incur any synchronization overhead instead of the end user application, and help the end user applications to utilize both of the DMA engines on the GPU when graphics APIs are used.
  With the introduction of the Fermi architecture, all Quadro and NVIDIA Tesla® GPUs feature two asynchronous DMA engines that enable asynchronous data transfers with concurrent 3-way overlap such that the current set of data can be processed while the previous set can be readback from the GPU, and the next set is uploaded.
  The DVP library already utilizes separate data transfer engines for CUDA, but for graphics APIs the driver will only utilize both engines if all API processing calls, data transfer calls into GPU, and data transfer calls out of the GPU reside in separate threads.

Note: Introduction of additional threads in the EXT SDK that are responsible for GPU DMA operations will lift the burden off of the end user to restructure the application for asynchronous DMA engines utilization.

For example, the capture data transfer thread would contain the following DVP library calls:

```
    dvpBegin();
    dvpMapBufferWaitDVP(dvpTextureHandle);
    dvpMemcpyLined(dvpSysMemHandle, dvpTextureHandle,..)
    …
    dvpMapBufferEndDVP(dvpTextureHandle);
    dvpEnd();
```

Meanwhile `dvpMapBufferWaitAPI()` and `dvpMapBufferEndAPI()` would be called by the EXT SDK from the application GPU processing thread.

The following pseudo code illustrates the DVP library calls that should be implicitly made from the application GPU processing thread:

```
GLuint EXTBegin()
{
    dvpTextureHandle = GetCapturedFrame();
       dvpMapBufferWaitAPI(dvpTextureHandle);
       return GLTexture(dvpTextureHandle);
}

void EXTEnd(GLuint TextureHandle)
{
    dvpTextureHandle = LocateDVPHandle(TextureHandle);
    dvpMapBufferEndAPI(dvpTextureHandle);
}

main()
{
     TextureHandle2 = EXTBegin();
     Render(TextureHandle2);
     EXTEnd(TextureHandle2);
}
```

- Various Optimizations
  It is important to note that when performing a batched copy of multiple frames, there is no need to call `dvpMap*` operations for each buffer in the batch. It is enough to call `dvpMap*` once in the beginning of the batch and once in the end of the batch, as the frames will be copied sequentially.