APPS034-Codensity T408™ Nvidia GPUDirect for Video Application Note

V3.0

2022-11-16

Table of Contents

1 Table of Abbreviations

Abbreviation:	Full Form:
CUDA	parallel computing platform and application programming interface model created by Nvidia
OpenGL	cross-language, cross-platform application programming interface for rendering 2D and 3D vector graphics
AI	Artificial Intelligence
API	Application Programming Interface
AUD	Access Unit Delimiters
AV	Audio Video
AV1	Alliance Open Media Video 1 Codec
CABAC	Context Adaptive Binary Arithmetic Coding
CBR	Constant Bit Rate
CPU	Central Processing Unit
CRF	Constant Rate Factor
DSP	Digital Signal Processing
EOF	End Of File
FIXQB	Fixed Quantization Parameter
FF	Form Factor
FPS	Frames Per Second
GOP	Group of pictures
HDR	High Dynamic Range
HHHL	Half Height, Half Length
HHFL	Half Height, Full Length
HLG	Hybrid Log Gamma
HRD	Hypothetical Reference Decoder
IE	Inference Engine
Mbps	Mega Bit per second
MBps	Mega Byte per second
NLP	Natural Language Processing
NPU	Neural Processing Unit
NVMe	Non-Volatile Memory express
PCIe	Peripheral Component Interconnect express
PPS	Picture Parameter Set
QP	Quoted Printable

RGB	Red Green Blue
RGBA	Red Green Blue Alpha
RDO	Optimization
ROI	Region of interest
SDK	Software Development Kit
SEI	Supplemental Enhancement Information
SRIOV	Single Root I/O Virtualization
SPS	Sequence Parameter Set
VBR	Variable Bit Rat
VCL	Video Coding Layer
VPS	Video Parameter Set
VUI	Video Usability Information

2 References

Technical Notes:
-GPUDIRECT FOR VIDEO Programmer's Guide (PG-06007-001_v05).

Specifications:

3 Background

This document describes how to use Nvidia GPUDirect for Video interface under Linux. There are two groups of APIs for Linux, CUDA and OpenGL. In this document it limits to CUDA.

3.1 Intended Audience

This document is intended to help engineers/technicians/developers who intend to integrate Nvidia GPUDirect for Video APIs into NETINT transcoders software.

3.2 Compatibility

Software Compatibility

APIs' version 1.70.

Hardware Compatibility

The APIs in this guide are tested on Nvidia T4 Tesla GPU.

Operation system

N/A.

4 GPU DMA Operation

Our application case would be to trigger GPU to start a DMA copy from GPU memory to system memory on host side. This would allow to grab GPU memory content with a specific format and send it to NETINT encoder.

Since most of the APIs are well described by the programmer guild, this document would focus on the DMA transfer and synchronization.

4.1 DMA copy

The DMA copy function is defined as follow.

```
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);
```

There are two synchronization Objects provided: srcSync and dstSync. When the srcSync's private count reaches srcAcquireValue, the DMA transfer starts. when the DMA transfer completes, the function updates the dstSync's private count as dstReleaseValue and wakes up the processes which wait for DMA transfer completion by calling dvpSyncObjClientWaitPartial with a condition of dstSync.

The DMA memory copy application is followed:

```
DVP_SAFE_CALL(dvpBegin());
…
        while (copiedSize < fifo->m_size)
        {
            uint32_t copiedChunkSize = (fifo->m_size-copiedSize > chunkSize ? chunkSize :
fifo->m_size-copiedSize);
```

```
            gpuSync->releaseValue++;

            DVP_SAFE_CALL(dvpMemcpy(buf->dvpGpuObjectHandle,
                        extSync->syncObj,
                        extSync->acquireValue,
                        DVP_TIMEOUT_IGNORED,
                        buf->dvpSysMemHandle,
                        gpuSync->syncObj,
                        gpuSync->releaseValue,copiedSize,copiedSize,
                        copiedChunkSize));
            copiedSize += copiedChunkSize;
        }

        DVPStatus status = dvpSyncObjClientWaitPartial(buf->gpuSync.syncObj,
                gpuSync->releaseValue, DVP_TIMEOUT_IGNORED);
…
DVP_SAFE_CALL(dvpEnd());
```

Here extSync->syncObj waits for its private count to be equal to extSync->acquireValue before the DMA transfer can be triggered. The DMA copy is executed asynchronously when the dvpMemcpy call completes. The wait for the DMA completion can be achieved calling dvpSyncObjClientWaitPartial.

The srcSync Object is updated as follow:

```
DVP_SAFE_CALL(dvpBegin());
…
        uint32_t chunkSize = (uint32_t)ceil((float)fifo->m_size/(float)fifo->m_numChunks);
        for (uint32_t i = 0; i < fifo->m_numChunks; i++) {
            // acquire the gpu semaphore before video transfer
            buf->gpuSync.acquireValue++;
            //update the release value
            buf->extSync.releaseValue++;
```

```
                // make sure we are done with the pending GPU DMAs
                DVPStatus status = dvpSyncObjClientWaitPartial(buf->gpuSync.syncObj,
buf->gpuSync.acquireValue, DVP_TIMEOUT_IGNORED);
                assert((status == DVP_STATUS_INVALID_OPERATION) || (status ==
DVP_STATUS_OK));


                //simulation of the actual DMA and the update of the semaphore
                //In reality this should be done asynchronously to the CPU
                //!!!!Start of DMA SIMULATION
                if (g_TestModeEnabled)
                {
                     //memcpy data in chunks
                     uint32_t copiedChunkSize = (fifo->m_size-copiedSize > chunkSize ? chunkSize :
fifo->m_size-copiedSize);
                     copiedSize += copiedChunkSize;
                     while( j < copiedSize/4) //4 bytes
                     {
                          MEM_WR32((uint32_t*)buf->extDevBuffer+j,
*((uint32_t*)buf->sysMemBuffer+j));
                          j++;
                     }
                }
                //update the semaphore when each chunk is DMAed
                MEM_WR32(buf->extSync.sem, buf->extSync.releaseValue);
                //!!!!end of DMA simulation
           }
         assert(buf->extSync.releaseValue == buf->gpuSync.releaseValue);
…
DVP_SAFE_CALL(dvpEnd());
```

Here it simulates a data transfer from system memory to device memory. Once this data transfer is done, the buffer is ready for receiving new data from GPU again. Then the extSync->syncObj private count is updated explicitly and dvpMemcpy should be able to see it.

4.2 Typo in the Programmer Guide

In chapter 4.2.1 DVP Sync Objects, it appears that there is a typo in the pseudo code. In the red block the code should be "WaitOnSemaphore(srcSync, acquireValue);" based on the context. And it is the same about the pseudo code explaining the DMA copy of EXT SDK.

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);
```

5 Notes

The T408 transcoder PCIe bar 4 or 5 space address was mapped as the system memory so that the data would be DMAed directly from the GPU memory to the transcoder DDR memory. But there was an invalid-parameter error when calling dvpBindToCUDACtx to bind the PCIe bar space address memory.

6 Revision History

Version	Change	Create Date	Author
1.0	Document Created	Mar 10,2021	Zheng Lv
2.0	Document updated	Sep 24, 2021	Achraf Jerbi
3.0	Update as per new app note template	December 21, 2021	Balwinder Sidhu

7 Legal Notice

Information in this document is provided in connection with NETINT products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in NETINT's terms and conditions of sale for such products, NETINT assumes no liability whatsoever and NETINT disclaims any express or implied warranty, relating to sale and/or use of NETINT products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right.

A "Mission Critical Application" is any application in which failure of the NETINT Product could result, directly or indirectly, in personal injury or death. Should you purchase or use NETINT's products for any such mission critical application, you shall indemnify and hold NETINT and its subsidiaries, subcontractors and affiliates, and the directors, officers, and employees of each, harmless against all claims costs, damages, and expenses and reasonable attorney's fees arising out of, directly or indirectly, any claim of product liability, personal injury, or death arising in any way out of such mission critical application, whether or not NETINT or its subcontractor was negligent in the design, manufacture, or warning of the NETINT product or any of its parts.

NETINT may make changes to specifications, technical documentation, and product descriptions at any time, without notice. The information here is subject to change without notice. Do not finalize a design with this information. The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications.

NETINT, Codensity, and NETINT Logo are trademarks of NETINT Technologies Inc. All other trademarks or registered trademarks are the property of their respective owners.