



# VIDEO CAPTURE, ENCODING, AND STREAMING IN A MULTI- GPU SYSTEM

TB-05566-001\_v01 | November 2010

**Technical Brief**



## TABLE OF CONTENTS

<b>Video Capture, Encoding, and Streaming in a Multi-GPU System .....</b>	<b>4</b>
System Overview .....	5
Video Capture .....	6
Connecting the Capture Card with a GPU .....	6
Video Processing with CUDA .....	8
3D video (Stereo) .....	10
Ancillary Data .....	10
Encoding .....	11
Encoder Performance Considerations .....	12
System Considerations .....	14
Data Considerations .....	14
Capture .....	15
Stereo Content Handling .....	15
Image Formats and Format Conversions .....	15
Image resampling .....	18
Data Movement .....	19
Streaming .....	19
3D Client .....	20
Video Stream Publishing .....	21
References: .....	21

## LIST OF FIGURES

Figure 1.	High level system diagram .....	5
Figure 2.	Capture GPU Tasks .....	14
Figure 3.	Encoding GPU Tasks .....	14
Figure 4.	YUY3 Format .....	15
Figure 5.	NV12 Format .....	16
Figure 6.	YV12 Format .....	17
Figure 7.	Lanczos Filter .....	18
Figure 8.	Streaming Data .....	19
Figure 9.	Microsoft Silverlight SMF 2.0 Video Player Data Format .....	20

## LIST OF TABLES

Table 1.	Hardware Components .....	6
Table 2.	Encoder Performance Chart .....	13

# VIDEO CAPTURE, ENCODING, AND STREAMING IN A MULTI-GPU SYSTEM

Nowadays, compression plays a major role in any media delivery infrastructure. In video streaming it is especially important as high-definition uncompressed video can consume as much as one gigabit per second for a single stream. Video codecs such as H.264 and VC-1 have made viewing high-quality video at low bit rates possible. However, for the best viewing experience, content providers are required to produce multiple versions of the captured stream at various bit rates for adaptive streaming, and at various resolutions to fit the screens of many different viewer devices.

Currently there is a need for efficient and affordable solutions that allow content providers to capture multiple SDI video feeds (or video file inputs) and produce multiple bitrates of each feed for internet delivery. There is also a growing demand for systems that are capable of capturing and streaming live 3D content. NVIDIA® GPUs are incorporated into all aspects of image and video processing thanks to the tremendous processing power available through the GPUs highly parallel architecture.

The purpose of this document is to outline some of the design and programming considerations required to build a real-time video encoder and server using NVIDIA technology. It details the fundamentals of programming for the NVIDIA Quadro® SDI video capture card, the efficiencies of GPU-based h.264 encoding, and how client applications can stream and watch 3D video.

## SYSTEM OVERVIEW

The described video encoder and video server system allows capturing several video feeds and it harnesses the power of multiple GPUs to deliver multiple compressed video streams to internet clients.

The figure below is a high level diagram for the system.

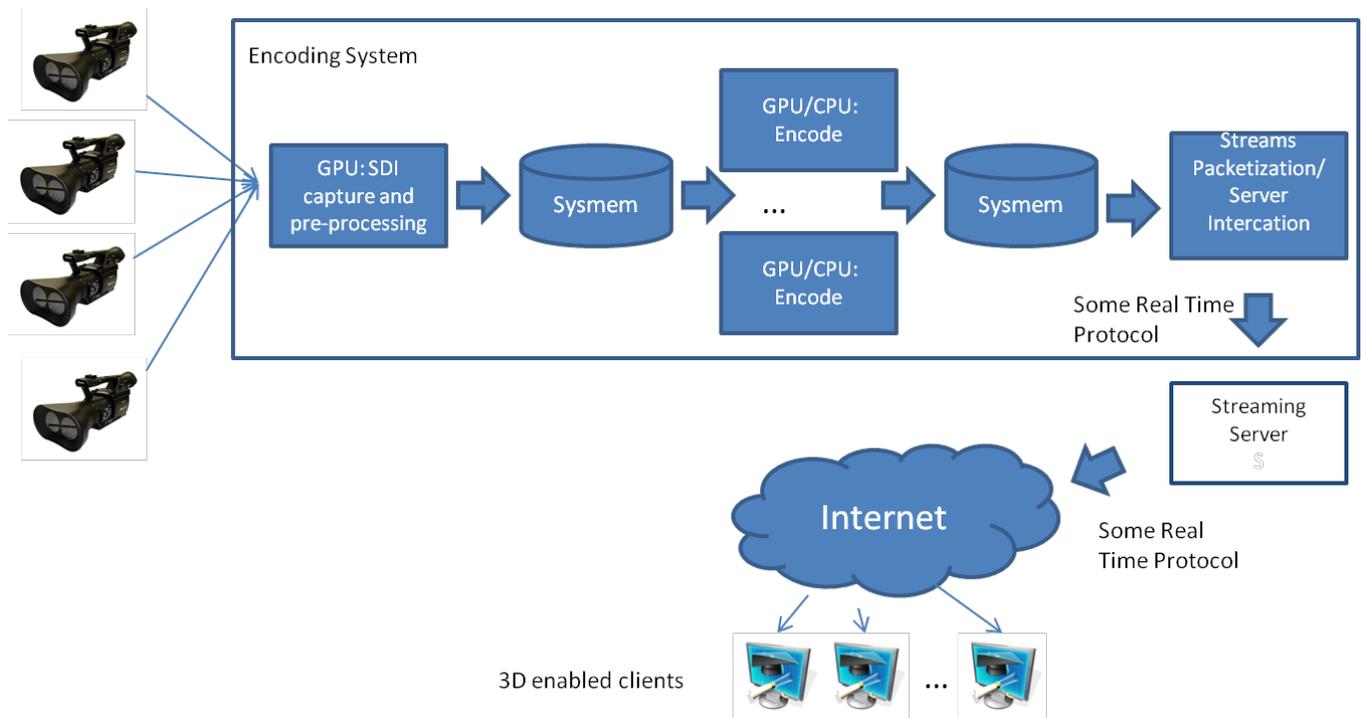


Figure 1. High level system diagram

The encoding portion of the system is implemented using the NVIDIA Quadro SDI capture card that provides the ability to capture up to four SDI video feeds with the lowest possible latency directly to an NVIDIA Quadro GPU and multiple Quadro and NVIDIA Tesla™ GPUs. The GPUs are used to accelerate video compression of the captured feeds. Table 1 lists the hardware components used to build the system.

Table 1. Hardware Components

Component	Description
 Quadro SDI Capture Card	PCI Express ×8 interface card capable of capturing up to four single-link, or two dual-link HD SDI, or two 3G SDI video streams directly into GPU video memory.
GPU	Quadro – GT200 and GF100 class
	Tesla – GT200 and GF100 class

## VIDEO CAPTURE

Video capture is done by the Quadro SDI Capture card. The device is capable of capturing up to four single-link, or two dual-link HD SDI, or two 3G SDI video streams directly into GPU video memory. This method delivers the lowest latency input to the GPU. To perform the capture, the device must be bound to one (and only one) of the GPUs that are supported for capture. Both the capture device and the GPU must be programmatically configured using the combination of NVIDIA I/O API and OpenGL capture extension (**NvAPI** with GL/WGL extension on windows, and **NVCtrl** with GL/GLX extension on Linux).

## Connecting the Capture Card with a GPU

Transfer of the SDI video data to the GPU is enabled by the **GL\_NV\_video\_capture** extension to OpenGL. The connection of the SDI card with the GPU is established using an OpenGL rendering context. Prior to creating the rendering context the application must select a device context on Windows and an XScreen on Linux to address a particular GPU. On Windows, GPU affinity extension must be used to create a device context corresponding to a particular GPU. This device context should then be used throughout capture configuration code.

## Code Listing 1: Addressing a Particular GPU on Windows:

```

HGPUNV gpuList[MAX_GPUS];

//populating a GPU affinity handle list.
int i = 0;
HGPUNV hGPU;
while(wglEnumGpusNV(GPUIdx,&hGPU))
{
    gpuList[i++] = hGPU;
    //hGPU and the affinity extension can be used for further GPU
    identification
}
...
HGPUNV handles[2];
handles[0] = gpuList[CaptureGPU];
handles[1] = NULL;

HDC videoDC = wglCreateAffinityDCNV(handles);

//Use the affinity device context when configuring capture and creating
OpenGL rendering context
UINT numDevices = wglEnumerateVideoCaptureDevicesNV(videoDC, NULL);
...

```

## Code Listing 2: Addressing a Particular GPU on Linux

On Linux, an XScreen associated with the chosen GPU must be used throughout capture configuration code. There might be cases where there is no one-to-one GPU $\leftrightarrow$ XScreen correspondence in the system. **NVCTRL** API must be used to determine the GPU to XScreen mapping.

```

//determine GPU->XScreen mapping
ret = XNVCTRLQueryTargetCount(dpy, NV_CTRL_TARGET_TYPE_GPU, &num_gpus);
if (ret) {
    for (gpu = 0; gpu < num_gpus; gpu++) {
        /* X Screens driven by this GPU */
        ret = XNVCTRLQueryTargetBinaryData
            (dpy,
             NV_CTRL_TARGET_TYPE_GPU,
             gpu, // target_id
             0, // display_mask
             NV_CTRL_BINARY_DATA_XSCREENS_USING_GPU,
             (unsigned char **) &pData,
             &len);
        if (ret) {
            if(pData[0])
                xscreen[gpu] = pData[1];
        }
    }
}

```

```

//NVCtrl API can be used for further GPU identification
    }
}
...
//The selected XScreen should be used when configuring capture and
creating OpenGL rendering context
VideoInDevices = glXEnumerateVideoCaptureDevicesNV(dpy,
xscreen[captureGPU],
                                                    &numDevices);
...

```

The **GL\_NV\_video\_capture** extension provides a mechanism for direct capture and streaming of the incoming SDI video into either OpenGL video buffer objects (VBO), which are an extension to the OpenGL pixel buffer objects(PBO) or texture objects in GPU memory.

## VIDEO PROCESSING WITH CUDA

Captured OpenGL objects already in the GPU memory can be mapped to CUDA memory space and further processed by the GPU using the NVIDIA CUDA®-OpenGL interoperability mechanism.

To do that, the CUDA device must be initialized for OpenGL interoperability. This can be done using **cuGLCtxCreate** call when creating a CUDA context using the driver API or **cuGLSetGLDevice** when setting up the device using the runtime API.



**Note:** OpenGL capture context must be current before creating a CUDA context with OpenGL interoperability.

### Code Listing 3: Creating CUDA Context for OpenGL Interoperability

```

CUdevice cuDevice;
CUcontext cuContext;
int selectedDevice = 0;
CUresult cerr = cuDeviceGet(&cuDevice, selectedDevice);
CheckError(cerr);
cerr = cuGLCtxCreate(&cuContext,
CU_CTX_MAP_HOST|CU_CTX_BLOCKING_SYNC, cuDevice);
CheckError(cerr);

```

A graphics object containing the video frame must be registered with CUDA in the beginning of the program execution and mapped to CUDA address space every frame prior to CUDA's usage. The object must be unmapped before it can be used again for capture. Code Listing 4 illustrates this.

#### Code Listing 4: CUDA Processing of a Video Buffer Object Using CUDA Driver API

```
GLint buf = m_vidBufObj[objInd];

CUgraphicsResource  cudaResource;

//Registering is done only once in the beginning
cuGraphicsGLRegisterBuffer(&cudaResource, buf,
CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE);
unsigned char *dptr;
// Buffer object mapping:Done every frame
cuGraphicsMapResources(1, &cudaResource, 0);
size_t num_bytes;
cuGraphicsResourceGetMappedPointer((void**)&dptr, &num_bytes,
cudaResource);

// Call the CUDA kernel here
// Buffer object unmapping:Done every frame
cuGraphicsUnmapResources(1, &cudaResource, 0);

// Unregistering is done only once in the end
cuGraphicsGLUnregisterBuffer(cudaResource)
```

Note that CUDA – OpenGL interop does not require for the CUDA context and OpenGL context to reside on the same device. When there are several GPUs present in the system, there is a possibility that the CUDA context and the OpenGL context reside on two separate devices because OpenGL and CUDA enumerate devices independent of each other. In this case the driver moves the buffer object from one device to the next via system memory every frame for interop.

Data movement can be avoided when the CUDA context and OpenGL context are made to reside on the same GPU. On Windows it is possible to achieve this by using GPU affinity and **cuWGLGetDevice** as shown in code listing 5.

## Code Listing 5: Using GPU Affinity for CUDA OpenGL Interoperability on Windows

```
HGPUNV gpuList[MAX_GPUS];
//see code listing 24 for gpuList setup
cuWGLGetDevice(&cuDevice, gpuList[CaptureGPU]);
// now create the CUDA context
result = cuGLCtxCreate(&cuContext,
CU_CTX_MAP_HOST|CU_CTX_BLOCKING_SYNC, cuDevice);
```

Currently on Linux there is no counterpart to **cuWGLGetDevice** (such as **cuGLXGetDevice** call), but it is planned for a future release of CUDA. In place of this, other techniques should be used in making sure that the CUDA context with OpenGL interop is created on the GPU with a particular XScreen. For example; one can make sure the capture GPU differs from the other GPUs in the system by name, then it would be possible to identify it in CUDA using **cuDeviceGetName** and only create a context with OpenGL interoperability for the device that has a particular name.

## 3D VIDEO (STEREO)

Stereoscopy is the most widely accepted method for capturing and delivering 3D video. It involves capturing stereo pairs in a two-view setup, with cameras mounted side by side and separated by the same (or close to the same) distance as between a person's pupils. When 3D video feed capture is mentioned in this paper it assumes stereoscopic capture and it also assumes that both left and right camera views arrive into the system as two separate, unprocessed video feeds.

## ANCILLARY DATA

In addition to video, the Quadro SDI Capture device captures ancillary data in both the horizontal and vertical blanking regions of the video streams. It can be configured to capture various types of data such as time-code, audio and other custom data packets. Each frame, the data ends up in a structure in system memory. This data can be accessed using the Ancillary Data API that is provided as a part of the Quadro SDI Capture SDK (refer to the SDK for details) and it should be packaged together with the already encoded frame prior to encoded stream distribution.

## ENCODING

For the purposes of this analysis, video encoding is performed using the NVIDIA CUDA Video Encoder (NVCUVENC). Note that there are several other examples of CUDA accelerated video encoding software products and tools that can be used instead (for example MainConcept CUDA H.264/AVC Encoder).

NVCUVENC is compliant with AVC/H.264<sup>1</sup> (MPEG-4 Part 10 AVC, ISO/IEC 14496-10). Its' intent, like that of all of MPEG-4, was to produce video compression of acceptable quality and very low bit-rate—around half the bit rate of its predecessors MPEG-2 and H.263) and it is using the GPU to accelerate the encoding process. It is supported on all CUDA enabled GPUs.

On the low level, an encoder takes raw YUV frames in NV12 format as input (*Image Formats and Format Conversions* on page 15) and generates Network Abstraction Layer (NAL) packets.



**Note:** To aid in providing efficient and error resilient transport, the AVC specification defines a Network Abstraction Layer (NAL) that encapsulates the output of the encoder. NAL Units consist of video slices: independently-decodable groups of macro blocks with positioning, quantization and other data. NAL Units form the basic fragments of video that are transmitted to clients

The H.264 encoder operates on a frame in units of macroblock (16x16 pixels). Each macroblock is encoded in *intra* (first picture or a reference frame of a sequence) or *inter* (all the other pictures or pictures between the reference frames) mode. In either case, a prediction macroblock P is formed based on a reconstructed frame. In intra mode, P is formed from samples in the current frame that have been previously encoded, decoded, and reconstructed. In inter mode, P is formed by motion-compensated prediction from one or more reference frame(s).

There are four compute-intensive portions in the H.264 encode process and it is important to consider those when using the GPU for the encode process acceleration:

- ▶ **Motion Estimation (ME)**  
Examining the reference frame for similarities to the input macroblock.
- ▶ **Motion Compensation (MC)**  
Block prediction by block reconstruction from previously encoded pictures using motion vectors (if there are any).

---

<sup>1</sup> H.264, is a subset of MPEG-4 also known as MPEG-4 Advanced Video Coding (AVC).

▶ **Digital Signal Processing (DSP)**

Transformation, scaling and quantization of the difference between the original and the predicted block (the residual).

▶ **Variable Length Coding (VLC)**

Examining the frequency of patterns within the quantized residual block and its coding.

NVCUVENC can use the GPU in two different modes:

- ▶ Partial GPU offload
- ▶ Full GPU offload.

In Partial Offload mode only the motion estimation task is executed on the GPU. For the Full Offload mode, the bulk of computation is executed on the GPU, with just the variable length coding running on the CPU.

## Encoder Performance Considerations

The purpose of this section is to provide an example of the kind of performance analysis that needs to be done when building a multi GPU encoding system and estimating the maximum encoding load that can be handled by the system. It also goes over some rudimentary encoder performance aspects that should be taken into consideration. For the sake of simplicity, the encoder is configured with performance settings (baseline profile, no B frames. and etc.). All of the video feeds are being encoded into 15 Mbps bit streams.

Performance is measured in various system configurations.

The system contains two four core CPUs (Intel Xeon x5550). At first, performance of the encoder on a single stream accelerated by a single NVIDIA GPU is considered and later multiple GPU/multiple encode configurations are examined.

## Encoding a Clip

Encoding a clip gives a good idea of the encoder throughput and the utilization of all the resources because in this scenario the encoder is not being bound by the input rate.

A clip at 800 frames long (approximately 13 seconds of video at 60 fps) and a 1920x1080 resolution will be encoded. The clip was captured at a sports event so it represents a good encoding challenge.

Table 2 shows the performance of the encoder in both modes of operation, running on a GT200 class GPU, Quadro FX 4800.



**Note:** GPU utilization in the following chart means which portion of the time the GPU is running kernels. In current GPU architectures, whenever the GPU is running a kernel it is utilized at a 100%. This might not be the case in future architectures.

Table 2. Encoder Performance Chart

Partial Offload		Full Offload	
FPS	120	FPS	60
GPU Utilization (%)	82	GPU Utilization (%)	90
CPU Utilization (%)	40 - 45	CPU Utilization (%)	6 - 10

The rest of the analysis will be performed in the partial offload mode in a system with one or more GPUs of the GT200s class.

### Real-time Single Stream Encoding:

According to the chart in Table 2, the encode process is operating at twice the video input rate when the incoming video is 1080p (a 1920x1080 resolution at 60 fps):

$$\text{Encode rate} = 120 \text{ fps} = 2 * 60 \text{ fps} = 2 * \text{Input rate}$$

Hence the encode process should utilize about a half of the CPU resource, which puts it at 40-45%/2, which is 20-22% utilization, and a half of its GPU resource, which achieves 40% utilization.

### Real-time Multiple Stream Encoding on a single GPU:

The number of 1080p video streams that can be encoded simultaneously (each stream is a 1920x1080 resolution at 60 fps) in the system with a single GPU will scale linearly until we reach encoder throughput. This means that we can run up to 2 simultaneous encodes of 1080p video.

### Real-time Multiple Stream Encoding on multiple GPUs:

By adding more GPUs to the system, the number of encodes running simultaneously scales linearly until limited by CPU utilization. Since each encoding process utilizes the CPU at about 22%, it means that the number of simultaneous full resolution 1080p encodes in a system will be bound by 5. Note that this is only a theoretical maximum at this point as other aspects of the system were not taken into consideration (for example: OS scheduling, video ingest overhead, initial processing overhead, and etc.).

## SYSTEM CONSIDERATIONS

It is important to consider the PCI bandwidth requirements of the capture board. In current motherboard architectures, two PCI slots can belong to two different PCI controllers, which can be detrimental to the GPU $\leftrightarrow$ Capture board bandwidth. As a result, care should be taken in slot placements of the capture board and the GPU designated for capture.

## DATA CONSIDERATIONS

Due to the fact that one of the GPUs is involved in the image acquisition, it makes sense for that GPU to do some data pre-processing. The resulting images are then passed to other GPUs for encoding. Figure 2 and Figure 3 show possible task distribution between the GPUs. Note, that depending on the GPU load during the encode process; the capture GPU can be available for encoding tasks in addition to its pre-processing tasks.

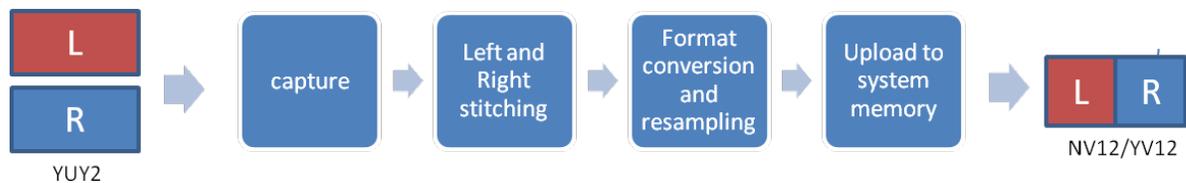


Figure 2. Capture GPU Tasks

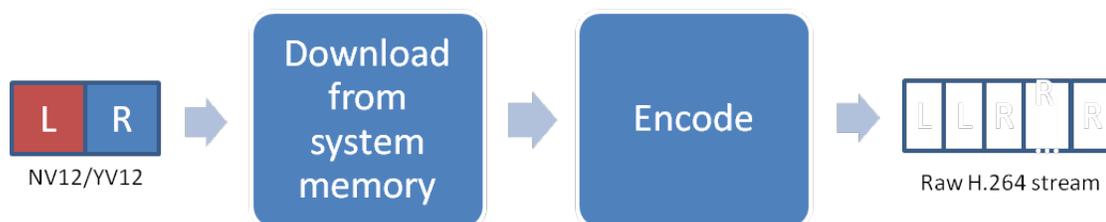


Figure 3. Encoding GPU Tasks

## Capture

Capture is happening almost entirely without GPU involvement with the exception of populating the OpenGL objects with the content from the ring buffer shared between the capture board and the GPU. This task is negligible compared to all the other tasks in the process.

## Stereo Content Handling

When stereo content is captured, it arrives at the GPU as two different buffers representing left and right eye. To be correctly processed by the encoder and downstream of the encoder, the frames must be packaged and formatted together.

A typical packaging and formatting is side-by-side. In side-by-side 3D, a full 1080p or 720p frame consists of two halves: one on the left and the other on the right. The entire frame for the left eye is scaled down horizontally to fit the left-half of the frame, and the entire frame for the right eye is scaled down horizontally to fit the right side of the frame. In the case of 720p content (resolution of 1280 x 720), each frame actually consists of the horizontally scaled frame for the left eye with a resolution of 640 x 720 and adjacent to it, the corresponding frame for the right eye at the same 640x720 resolution.

## Image Formats and Format Conversions

The image(s) arrive at the capture GPU and end up as a buffer in YUY2 format.

YUY2 is a packed 4:2:2 YUV format where every scan line contains four Y samples for every two U or V samples. Figure 4 shows the memory component arrangement of YUY2 data.

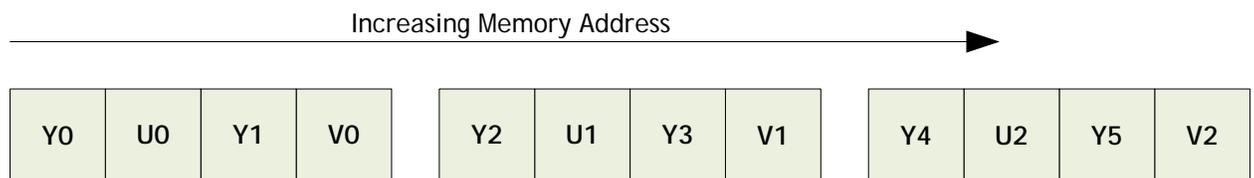


Figure 4. YUY3 Format

An image with this component arrangement is not very suitable for per-pixel operations and it usually undergoes a format transformation to an image of a planar format prior to any processing.

At the low level, the NVIDIA encoder requires video to be in NV12 format, so even though the encoder API accepts video in other YUV formats, the encoder will internally convert the video to be in NV12 format prior to encoding. NV12 is a 4:2:0 quasi-planar format with four samples of Y for every two U or V samples horizontally and two samples of Y for every sample of U or V vertically (see Figure 5).

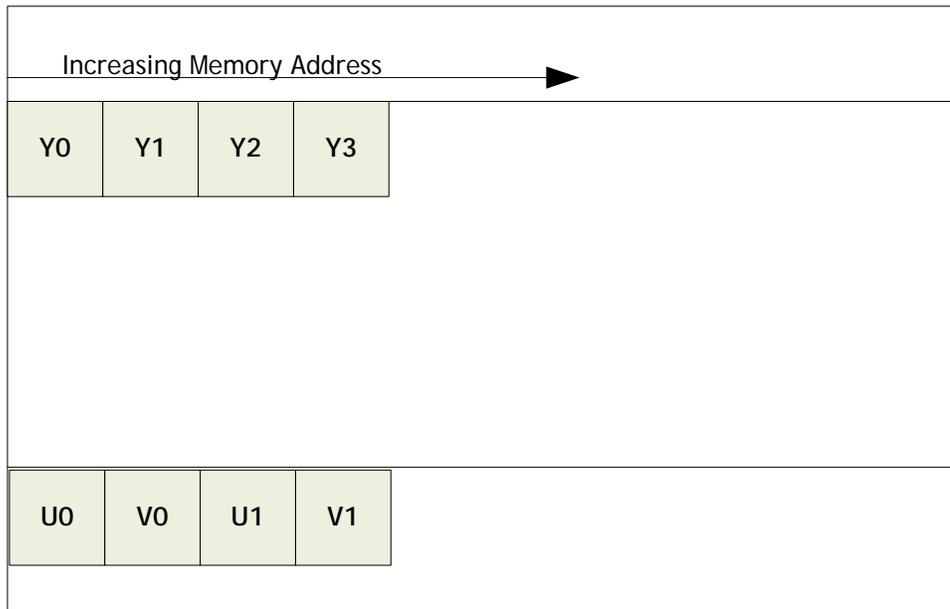


Figure 5. NV12 Format

An image with this component arrangement is not very suitable for per-pixel operations and it usually undergoes a format transformation to an image of a planar format prior to any processing.

A more common format that is typically used for encoding is YV12. YV12 is a 4:2:0 planar format with the same component sampling as NV12. The component arrangement is shown in Figure 6.

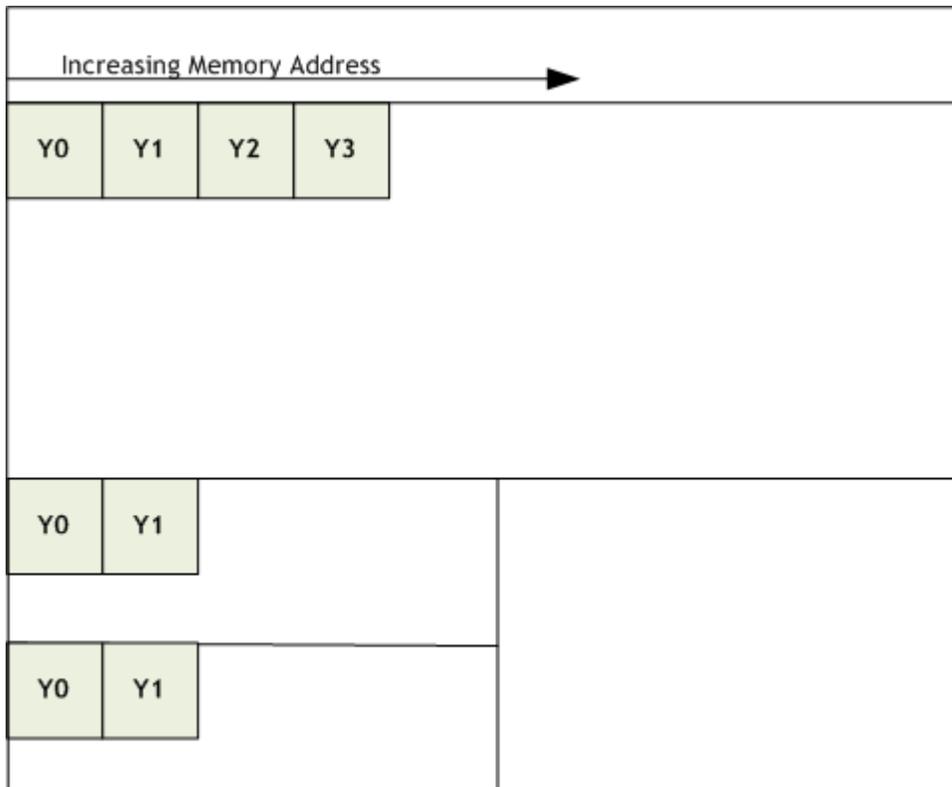


Figure 6. YV12 Format

To arrive at the encoding image format, the image data must undergo component reordering and vertical resampling of the Chroma planes. These operations can be conducted on the encoding GPUs but since some down sampling is involved, it might be more beneficial to have them done on the capture GPU. This way less data is being transferred between the GPUs. Resampling is discussed in more detail in the following section.

## Image resampling

As discussed, some component planes of the captured image must undergo down sampling prior to encoding (vertically for Chroma plane sub-sampling and horizontally for all component planes when a packaged stereo content is being processed). Because high quality image resampling is a compute intensive task that is inherently parallel and well suited for the GPU, the resampling is done on the GPU.

The implementation used is *Lanczos windowed sinc* function (Figure 7) because it offers the best compromise in terms of reduction of aliasing, sharpness, and minimal ringing.

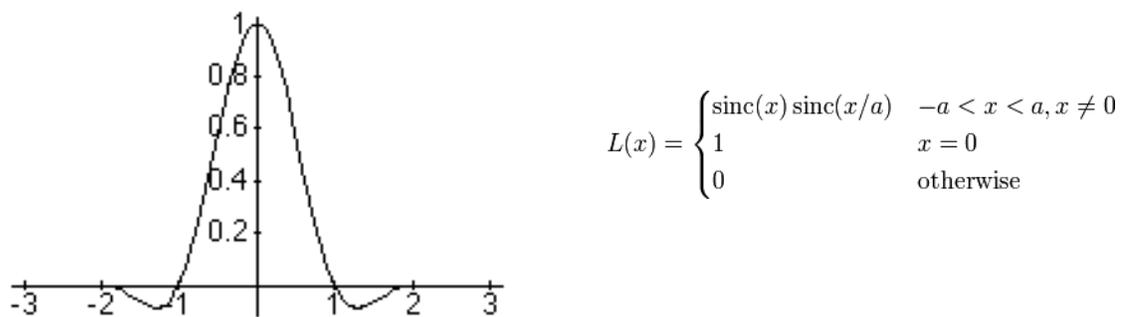


Figure 7. Lanczos Filter

The algorithm walks through the pixels in the downsampled image and looks at each pixel as a fractionally-positioned pixel in the original image. The color value of the pixel comprises of a weighted sum of the neighboring pixels in a 4x4 neighborhood. The weights are calculated using the formula shown in Figure 7 and each weight is normalized by the sum of the weight.

For packaged stereo content this procedure is conducted horizontally when down sampling the Luma plane, and both vertically and horizontally when down sampling the Chroma planes. The image should also undergo some blurring (low pass filtering), and possibly de-interlacing if the incoming video is interlaced prior to resampling. This reduces aliasing and preserves image integrity. The same function can be used for the low pass filter.

## Data Movement

The SDI capture board delivers the video frames to a single GPU, where in a multi GPU environment has to propagate them to other GPUs in the system for further processing (for example when encoding).

Data movement is happening after all the initial processing had taken place on the capture GPU; particularly stereo content packaging, format conversion, and resampling. Once the initial processing is done, the capture GPU uploads the data to system memory for encoding GPUs to consume.

## STREAMING

To accomplish streaming, a publishing process that packages encoded frames together with audio and other ancillary data in some network-friendly container then publishes the multiplexed streams to a content distribution server must be present downstream of the encoder. The end user can connect to the server through a URL and view the content with the appropriate client (the user must have a player and in the case of 3D content, also 3D ready hardware). There are several content delivery techniques and frameworks that implement this available today. But at the time that this paper is written there is only one officially available client for viewing streaming 3D content and it is NVIDIA 3D Vision™-modified Microsoft Silverlight SMF 2.0 Video Player. Figure 8 shows the streaming data going to the Microsoft Silverlight player.

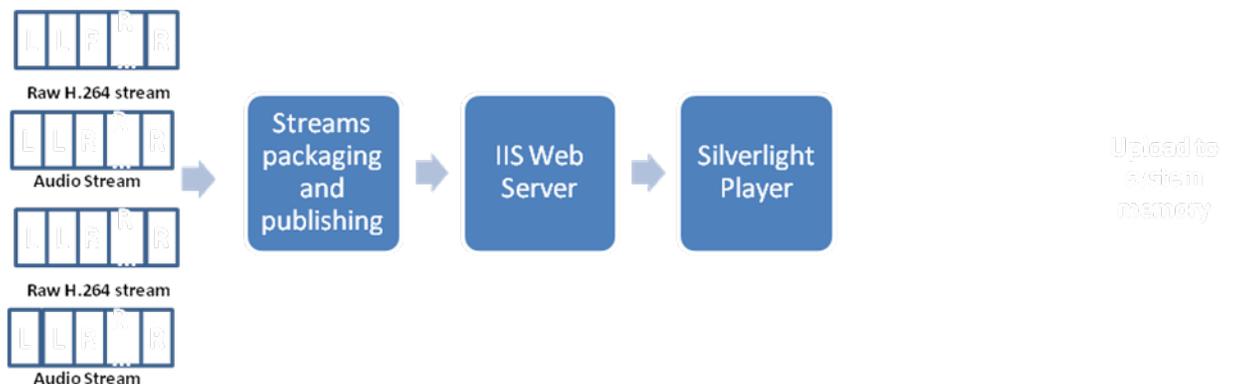


Figure 8. Streaming Data

## 3D Client

NVIDIA 3D Vision-modified Microsoft Silverlight SMF 2.0 Video Player is built to receive 3D content in a frame-packed format (side/side, top/bottom, etc.). It decodes the stream, splits each frame to extract the frame for each eye, and then rescales these individual frames to a full HD resolution using up-scaling algorithms. The player then displays these up-scaled individual frames alternately in a frame-sequential manner that is in sync with your active shutter 3D glasses.

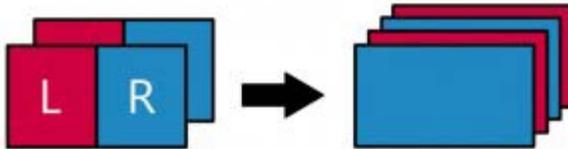


Figure 9. Microsoft Silverlight SMF 2.0 Video Player Data Format

Silverlight is powered by IIS Smooth Streaming technology. This allows the client to request stream chunks from the Web server in a linear fashion and download them using plain HTTP progressive download. As the chunks are downloaded to the client, the client plays back the sequence of chunks in linear order. When the video/audio source is encoded at multiple bit rates the client can now choose between chunks of different sizes. Because Web servers usually deliver data as fast as network bandwidth allows them to, the client can easily estimate user bandwidth and decide to download larger or smaller chunks ahead of time thus allowing smooth user viewing experience.

The next section briefly explains what needs to consist of a publishing process to be able to deliver the encoded streams to the server.

## Video Stream Publishing

IIS Smooth Streaming technology uses MPEG-4 Part 14 (ISO/IEC 14496-12) as a transport format. This format consists of basic units called a *box* containing both metadata and data. It is also referred to as *Fragmented MP4* (or *f-MP4*) because its specification is designed to allow these MP4 boxes to be organized in a fragmented manner, where the stream can be created "as you go" as a series of short metadata/data box pairs, rather than one long metadata/data pair.

The Smooth Streaming Format SDK is intended to be used by encoding applications to package compressed video and audio payloads into the fragmented-MP4 container, and to generate the required manifests that describe the bit-streams. These generated files include file extensions **\*.ismv** that contain H.264 elementary streams and audio data, and manifest XML files with file extensions **\*.ism** and **\*.ismc** that contain detailed description of the streams for the server and the clients to consume.

The publishing process is responsible for setting up and managing long-running HTTP POST connections to the IIS server's Live Smooth Streaming Publishing Point. Once the process establishes an HTTP connection with the server, it can call into the SSF library and provide settings information about the input video and audio compressed bit-streams. Subsequently, the application will feed compressed video and audio payload into the SSF library until enough data has been provided to package one fragment/chunk. The process will then request the f-MP4 buffer for transmitting to the IIS server.

When the entire stream has been processed an empty *mfra* box should be sent to the IIS server to signal end-of-stream and stop the publishing point.

## REFERENCES:

- ▶ *NVIDIA Quadro SDI Capture Programming Guide*
- ▶ *Microsoft IIS Smooth Streaming Technical Overview*
- ▶ *NVIDIA CUDA Video Encoder Specification*

## Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

## HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

## ROVI Compliance Statement

NVIDIA Products that support Rovi Corporation's Revision 7.1.L1 Anti-Copy Process (ACP) encoding technology can only be sold or distributed to buyers with a valid and existing authorization from ROVI to purchase and incorporate the device into buyer's products.

This device is protected by U.S. patent numbers 6,516,132; 5,583,936; 6,836,549; 7,050,698; and 7,492,896 and other intellectual property rights. The use of ROVI Corporation's copy protection technology in the device must be authorized by ROVI Corporation and is intended for home and other limited pay-per-view uses only, unless otherwise authorized in writing by ROVI Corporation. Reverse engineering or disassembly is prohibited.

## OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

## Trademarks

NVIDIA, the NVIDIA logo, CUDA, NVIDIA 3D Vision, Quadro, and Tesla are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

## Copyright

© 2010 NVIDIA Corporation. All rights reserved.