TABLE OF CONTENTS

Chapter 1. Introduction ........................................................................................................ 1
Chapter 2. Getting Started .................................................................................................... 2
  2.1. Requirements ........................................................................................................... 2
  2.2. Build The Samples .................................................................................................. 2
  2.3. Run The Samples .................................................................................................... 3
Chapter 3. Integrating NGX With Your Application ........................................................... 4
  3.1. Adding NGX To Your Application ......................................................................... 4
  3.2. Initializing NGX ..................................................................................................... 4
    3.2.1. Verifying ......................................................................................................... 6
  3.3. Setting And Obtaining Parameters ......................................................................... 7
  3.4. Scratch Buffer Setup .............................................................................................. 7
  3.5. Feature Creation .................................................................................................... 8
  3.6. Feature Evaluation ................................................................................................. 9
  3.7. Feature Disposal ................................................................................................... 10
  3.8. Shutdown ............................................................................................................... 10
Chapter 4. Distributing NGX Features With Your Application .......................................... 12
Chapter 5. NGX Features .................................................................................................. 13
  5.1. DLSS -- Deep Learning Super Sampling (DirectX only) ......................................... 13
  5.2. DLISR - Image Super-Resolution (CUDA Only) ....................................................... 13
  5.3. DLVSRSR (CUDA Only) ...................................................................................... 16
  5.4. DLSLOWMO (CUDA Only) .................................................................................. 18
  5.5. DLINPAINTING (CUDA Only) ............................................................................ 21
Chapter 6. Troubleshooting ............................................................................................... 24
  6.1. Error Codes ......................................................................................................... 24
  6.2. Support ............................................................................................................... 25
Chapter 1.
INTRODUCTION

NVIDIA NGX makes it easy for you to integrate pre-built AI based features into your applications. NVIDIA will be adding new features and also updating the existing ones over time. When an existing feature is updated, the NGX infrastructure will update the feature on all clients that uses it.

There are three main components that make up the system:

**NGX SDK**
- The NGX SDK provides CUDA and DX11/12 APIs for applications to access the AI features.

**NGX Core Runtime**
- All runtime modules are provided with the NVIDIA Graphics Driver that supports RTX hardware. During an advanced driver installation the module is called NGX Core.

**NGX Update Module**
- This module ensures that NGX integrated applications always use the latest version of the NGX features.
Chapter 2.
GETTING STARTED

2.1. Requirements

Ensure you meet the following requirements:

- Download the SDK from NVIDIA Developer Zone.
- After registration, you will receive an installer that will install the SDK materials and samples. By default the SDK will install into the `c:\ProgramData\NVIDIA Corporation\NGX SDK` directory.

The following is needed in order to run NGX applications:

- Windows PC with Windows 10 v1709 (Fall 2017 Creators Update 64-bit) or newer
- NVIDIA RTX GPU (Quadro, Titan or GeForce)
- Latest NVIDIA Graphics Driver, minimum 410.63.

The minimal development environment for integrating the NGX SDK into your application is:

- Microsoft Visual Studio 2015 SP3 with Windows 10 SDK (10.0.10586)

2.2. Build The Samples

To build the SDK Samples, open the NGX Samples solution file in `Samples\NGX Samples` with Visual Studio and select Build Solution from the Build Menu. Select Release x64 from the configuration drop down.

The build will place the sample executables and all required DLLs into the `x64\Release` folder. To step through the source code in the debugger while executing one or more of the samples, select the Debug x64 configuration. In this case, the built samples and all required DLLs are placed into `x64\Debug`. 
2.3. Run The Samples

The samples will build four executables located in the `<NGX SDK>\Samples\NGX Samples\x64\Release` directory:

- `isr.exe`: Image UpRes
- `vsr.exe`: Video UpRes
- `inpaint.exe`: In-Painting
- `Slowmo.exe`: Video slo-mo

They are all command-line applications and operate in a similar way. Run the executables with no commands to see the options.

An example command for the InPainting sample looks similar to:

```
x64\Release>inpaint.exe --input ..\..\SampleImages\inpaint_input.png --mask ..\..\SampleImages\inpaint_mask.png --model 0 --output d:\test.png
```
Chapter 3.
INTEGRATING NGX WITH YOUR APPLICATION

3.1. Adding NGX To Your Application

NGX SDK comes with two header files:

- nvsdk ngx.h
- nvsdk ngx_defs.h

Both files are located in the `<NGX SDK>\include` folder.

In your projects, ensure you include `nvsdk ngx.h`. In addition to including the NGX header files, your project should also link against:

- `nvsdk ngx_s.lib` (if static runtime library (/MT) linking is used in your project), or
- `nvsdk ngx_d.lib` (if dynamic runtime library (/MD) linking is used in your project)

Both files are located in the `<NGX SDK>\lib\x64` folder (NGX is provided as a 64bit only library).

During development, copy `nvngx_* .dll` from `<NGX SDK>\bin\features` to the folder where your executable or DLL is located, so NGX runtime can find DLLs.

For information about how NGX should be distributed with your application, see Distributing NGX Features With Your Application.

3.2. Initializing NGX

The NGX SDK supports the D3D11, D3D12 and CUDA APIs. Generally speaking, each NGX feature supports one API, however, NGX does not limit the number of APIs that
can be supported. The initial features made publicly available support CUDA. For more information, see NGX Features which describes all the APIs that each feature supports.

**Caution**

THE NGX API is not thread safe. The client application must ensure that thread safety is enforced as needed. Invoking NGX APIs from multiple threads can result in unpredictable behavior. For features that use DirectX, the NGX API preserves the state of the immediate D3D11 context, however, that is not the case with D3D12 command lists. For all applications using DirectX 12 and D3D12 command lists, the client application must manage the command list state as needed.

All calls in the NGX SDK are similar for the supported APIs (D3D11, D3D12 and CUDA) so all features can be initialized using code that matches or is very similar to the following sample code. Ensure that you use calls that match the API that your application is using. Any interop between APIs, for example, D3D11 to CUDA, must be handled by the application outside the NGX SDK.

To initialize an SDK instance, use one of the following methods:

```c
// NVSDK_NGX_Init
// -------------------------------------
//
// InApplicationId:
//      Unique Id provided by NVIDIA
//
// InApplicationDataPath:
//      Folder to store logs and other temporary files (write access required)
//
// InDevice: [d3d11/12 only]
//      DirectX device to use
//
// DESCRIPTION:
//      Initializes new SDK instance.
//      
// #ifdef __d3d11_h__
//NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D11_Init(unsigned long InApplicationId, const wchar_t *InApplicationDataPath, ID3D11Device *InDevice, NVSDK_NGX_Version InSDKVersion = NVSDK_NGX_Version_API);
//#elif defined __d3d12_h__
//NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D12_Init(unsigned long InApplicationId, const wchar_t *InApplicationDataPath, ID3D12Device *InDevice, NVSDK_NGX_Version InSDKVersion = NVSDK_NGX_Version_API);
//#else
//NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_CUDA_Init(unsigned long InApplicationId, const wchar_t *InApplicationDataPath,NVSDK_NGX_Version InSDKVersion = NVSDK_NGX_Version_API);
//#endif
```

All the NGX calls have an Application ID as the first parameter which is used mainly for features that have application specific tuning.

Until NVIDIA has assigned you an application ID, use 0. Prior to releasing a product with NGX integrated, notify NVIDIA that you are using NGX here: [https://developer.nvidia.com/sw-notification](https://developer.nvidia.com/sw-notification) and NVIDIA will provide you with an NGX compatible application ID to integrate into your application.

Once an SDK instance is created, create the features as needed by your application. In version 1.0.0, the following deep learning based features are supported:
3.2.1. Verifying

Successful initialization indicates that the target system is capable of running NGX features. However, each feature can have additional dependencies, for example, minimum driver version; therefore, it’s a good practice to check if the specific feature you would like to use is available. For this purpose, NGX provides an `NVSDK_NGX_Parameter` interface which can be used to query read-only parameters provided by NGX runtime and can be obtained using the following API:

```
// NVSDK_NGX_GetParameters
// --------------------------
// Parameters interface used to set any parameter needed by the SDK

For example, to check if `NVSDK_NGX_Feature_InPainting` is available on your system, issue the following:

```c
int InPaintingSupported = 0;
NVSDK_NGX_Parameter *Params = nullptr;
NVSDK_NGX_CUDA_GetParameters(&Params);
Params->Get(NVSDK_NGX_Parameter_InPainting_Available, &InPaintingSupported);
```
3.3. Setting And Obtaining Parameters

Each feature requires certain parameters to be set-up prior to feature creation. This is accomplished by using the `NVSDK_NGX_Parameter` interface.

Parameters are specified as `{name, value}` pairs, for example:

```c
NVSDK_NGX_Parameter *Params = nullptr;
NVSDK_NGX_CUDA_GetParameters(&Params);
Params->Set(NVSDK_NGX_Parameter_Width, 100);
```

For more details on which parameters should be set for specific features, see NGX Features.

3.4. Scratch Buffer Setup

After parameters are set, for each feature, check how much GPU scratch memory is required by calling:

```c
// NVSDK_NGX_GetScratchBufferSize
// ----------------------------------------------------------
// // InFeatureId:
// //   AI feature in question
// // // InParameters:
// //   Parameters used by the feature to help estimate scratch buffer size
// // // OutSizeInBytes:
// //   Number of bytes needed for the scratch buffer for the specified feature.
// // // DESCRIPTION:
// //   SDK needs a buffer of a certain size provided by the client in order to initialize AI feature. Once feature is no longer needed buffer can be released. It is safe to reuse the same scratch buffer for different features as long as minimum size requirement is met for all features. Please note that some features might not need a scratch buffer so return size of 0 is completely valid.
// // NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
// NVSDK_NGX_D3D11_GetScratchBufferSize(NVSDK_NGX_Feature InFeatureId, const NVSDK_NGX_Parameter *InParameters, size_t *OutSizeInBytes);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D12_GetScratchBufferSize(NVSDK_NGX_Feature InFeatureId, const NVSDK_NGX_Parameter *InParameters, size_t *OutSizeInBytes);
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_CUDA_GetScratchBufferSize(NVSDK_NGX_Feature InFeatureId, const NVSDK_NGX_Parameter *InParameters, size_t *OutSizeInBytes);
```
The application is responsible for allocating a scratch buffer of the requested size and passing it as a parameter when creating a specific feature (more details can be found in the source code examples in the NGX SDK).

It is acceptable for the SDK to return 0 as a required scratch buffer size for a specific feature.

### 3.5. Feature Creation

When scratch buffer is allocated and all necessary parameters are set, create the feature. The following code can be used to create a feature:

```c
// NVSDK_NGX_CreateFeature
// -------------------------------------
// InCmdList: [d3d12 only]
// Command list to use to execute GPU commands. Must be:
// - Open and recording
// - With node mask including the device provided in NVSDK_NGX_D3D12_Init
// - Execute on non-copy command queue.
// InDevCtx: [d3d11 only]
// Device context to use to execute GPU commands
// InFeatureID:
// AI feature to initialize
// InParameters:
// List of parameters
// OutHandle:
// Handle which uniquely identifies the feature. If feature with
// provided parameters already exists the "already exists" error code is
// returned.
// DESCRIPTION:
// Each feature needs to be created before it can be used.
// Refer to the sample code to find out which input parameters
// are needed to create specific feature.

NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D11_CreateFeature(ID3D11DeviceContext *InDevCtx, NVSDK_NGX_Feature InFeatureID, const NVSDK_NGX_Parameter *InParameters, NVSDK_NGX_Handle **OutHandle);

NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D12_CreateFeature(ID3D12GraphicsCommandList *InCmdList, NVSDK_NGX_Feature InFeatureID, const NVSDK_NGX_Parameter *InParameters, NVSDK_NGX_Handle **OutHandle);
```
 integrates with your application.

3.6. Feature Evaluation

Features are evaluated by executing inference on specific deep learning models. This can be achieved by calling:

```c
// NVSDK_NGX_EvaluateFeature
// _________________________________
// InCmdList: [d3d12 only]
//   Command list to use to execute GPU commands. Must be:
//     - Open and recording
//     - With node mask including the device provided in NVSDK_NGX_D3D12_Init
//     - Execute on non-copy command queue.
// InDevCtx: [d3d11 only]
//   Device context to use to execute GPU commands
// InFeatureHandle:
//   Handle representing feature to be evaluated
// InParameters:
//   List of parameters required to evaluate feature
// InCallback:
//   Optional callback for features which might take longer
to execute. If specified SDK will call it with progress
// values in range 0.0f - 1.0f. Client application can indicate
// that evaluation should be cancelled by setting OutShouldCancel
to true.
// DESCRIPTION:
//   Evaluates given feature using the provided parameters and
pre-trained NN. Please note that for most features
it can be beneficial to pass as many input buffers and parameters
as possible (for example provide all render targets like color, albedo,
normals, depth etc)
```

```c
typedef void (NVSDK_CONV *PFN_NVSDK_NGX_ProgressCallback)(float
InCurrentProgress, bool &OutShouldCancel);
```

```c
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D11_EvaluateFeature(ID3D11DeviceContext *InDevCtx, const
NVSDK_NGX_Handle *InFeatureHandle, const NVSDK_NGX_Parameter *InParameters,
PFN_NVSDK_NGX_ProgressCallback InCallback = nullptr);
```

```c
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D12_EvaluateFeature(ID3D12GraphicsCommandList *InCmdList, const
NVSDK_NGX_Handle *InFeatureHandle, const NVSDK_NGX_Parameter *InParameters,
PFN_NVSDK_NGX_ProgressCallback InCallback = nullptr);
```
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_CUDA_EvaluateFeature(const NVSDK_NGX_Handle *InFeatureHandle, const NVSDK_NGX_Parameter *InParameters, PFN_NVSDK_NGX_ProgressCallback InCallback = nullptr);

**Caution** DirectX: NGX will modify D3D12 command list state, therefore, the calling process should save or restore its own D3D state before and after calling the NGX evaluate feature.

When evaluating feature input buffers (for example, color, albedo, normals, depth, etc.) they have to be provided as parameters (either as ID3D* resources or CUDA memory buffers). For sample code, see NGX Features.

Some features are not real-time, therefore, it can take a few seconds to finish evaluation (for example, DLSlowMo with high resolution video feed). Progress callback should be used to provide feedback to the user and allow cancellation as needed.

### 3.7. Feature Disposal

When a feature is no longer needed, it should be released by calling the following method:

```c
// NVSDK_NGX_Release
// ---------------------
// InHandle:
//   Handle to feature to be released
// DESCRIPTION:
//   Releases feature with a given handle.
//   Handles are not reference counted so
//   after this call it is invalid to use provided handle.
//
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D11_ReleaseFeature(NVSDK_NGX_Handle *InHandle);

NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_D3D12_ReleaseFeature(NVSDK_NGX_Handle *InHandle);

NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV
NVSDK_NGX_CUDA_ReleaseFeature(NVSDK_NGX_Handle *InHandle);
```

Once released, the feature handle cannot be used any longer.

### 3.8. Shutdown

To release an SDK instance and all resources allocated with it, use the following method:

```c
// NVSDK_NGX_Shutdown
// ---------------------
//
```
// DESCRIPTION:
//      Shuts down the current SDK instance and releases all resources.
//
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D11_Shutdown();
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_D3D12_Shutdown();
NVSDK_NGX_API NVSDK_NGX_Result NVSDK_CONV NVSDK_NGX_CUDA_Shutdown();
Chapter 4. DISTRIBUTING NGX FEATURES WITH YOUR APPLICATION

The NGX SDK contains DLLs for the feature which must be distributed with your application. These basic DLLs are located in the ./bin/features folder and should be installed in the same folder as your application’s executable (or DLL if you are building a plugin).

You need to distribute only DLLs for the features your application is using. For example, if your application is only using InPainting, then you should only include nvngx_inpaint.dll with the rest of your binary files.

<table>
<thead>
<tr>
<th>AI Feature</th>
<th>NGX Feature DLL</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI UpRes -- Stills</td>
<td>DLISR - Image Super-Resolution (CUDA Only)</td>
</tr>
<tr>
<td>AI UpRes -- Video</td>
<td>DLVSR (CUDA Only)</td>
</tr>
<tr>
<td>AI InPainting</td>
<td>DLINPAINTING (CUDA Only)</td>
</tr>
<tr>
<td>AI SloMo</td>
<td>DLSLOWMO (CUDA Only)</td>
</tr>
</tbody>
</table>

The installer for your application should treat these DLL’s in the same way as other components to your application and remove them on uninstall.
5.1. DLSS -- Deep Learning Super Sampling (DirectX only)

DLSS is a new technique developed by NVIDIA Research to apply very high quality image sharpening, enhancement, and optionally, super-resolution to frames rendered by 3D engines and games. This technique uses an autoencoder to extract high dimensional features from the frame and then reconstructs the image with added details and with increased resolution.

DLSS is currently in a closed beta release. NVIDIA is working with partners to enable DLSS in a range of DirectX games. For more information, contact us at NGXSupport@nvidia.com.

5.2. DLISR - Image Super-Resolution (CUDA Only)

What Does This Feature Do?

The DLISR feature increases the resolution of an image by 2, 4 or 8 times using deep learning inference to generate additional pixels.

NGX DLISR Sample

The NGX ISR sample application included in the NGX SDK demonstrates the generation of a high resolution image in 8-bit RGB format from a lower resolution 8-bit RGB input image. This sample uses the `getRgbImage()` and `putRgbImage()` helper functions implemented on top of OpenCV to read and write most common image formats based on the file extension.
The following command reads the `input.png` file and generates an image with x2 scaling factor to `output.png`. The scaling factor could be 2, 4 or 8.

```
isr.exe --input input.png --factor 2 --output output.png
```

### Supported Parameters

The following parameters are supported by DLISR:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>Width of the input image CUDA buffer.</td>
</tr>
<tr>
<td>Height</td>
<td>Height of the input image CUDA buffer.</td>
</tr>
<tr>
<td>Scratch</td>
<td>Scratch buffer allocated with <code>cudaMalloc()</code>.</td>
</tr>
<tr>
<td>Scratch_SizeInBytes</td>
<td>Scratch buffer size.</td>
</tr>
<tr>
<td>Color</td>
<td>CUDA buffer of linear memory (allocated with <code>cudaMalloc()</code>) containing the 8-bit RGB pixels of the input image.</td>
</tr>
<tr>
<td>Color_Format</td>
<td>Color precision of the input image, <code>NVSDK_NGX_Buffer_Format_RGB8UI</code>.</td>
</tr>
<tr>
<td>Color_SizeInBytes</td>
<td>Size of the input image in bytes.</td>
</tr>
<tr>
<td>Output</td>
<td>CUDA buffer of linear memory (allocated with <code>cudaMalloc()</code>) to return the 8-bit RGB pixels of the output image (4x, 16x or 256x which is the size of the source buffer).</td>
</tr>
<tr>
<td>Output_Format</td>
<td>Color precision of the output image, <code>NVSDK_NGX_Buffer_Format_RGB8UI</code>.</td>
</tr>
<tr>
<td>Output_SizeInBytes</td>
<td>Size of the output image in bytes.</td>
</tr>
<tr>
<td>Scale</td>
<td>Super-resolution scale factor (2,4 or 8).</td>
</tr>
</tbody>
</table>

### Sample Code

The following code snippet shows how to use the DLISR feature in NGX. The following steps are involved:

1. Create an NGX Context using `NVSDK_NGX_CUDA_Init()`.  
2. Get structure for setting input parameters using `NVSDK_NGX_CUDA_GetParameters()`.  
3. Get the scratch buffer size required by the DLISR algorithm using `NVSDK_NGX_CUDA_GetScratchBufferSize()`, this depends on the input video width, height, and the scale factor chosen.
5. Create the NGX DLISR feature using `NVSDK_NGX_CUDA_CreateFeature()`.
6. Run DLISR using `NVSDK_NGX_CUDA_EvaluateFeature()`.
7. Release the NGX DLISR feature using `NVSDK_NGX_CUDA_ReleaseFeature()`.
8. Free the NGX resources using `NVSDK_NGX_CUDA_Shutdown()`.

```c
// Initialize NGX.
CK_NGX(NVSDK_NGX_CUDA_Init(app_id, L"./", NVSDK_NGX_Version_API));

// Get the parameter block.
CK_NGX(NVSDK_NGX_CUDA_GetParameters(&params));

// Verify feature is supported
int Supported = 0;
params->Get(NVSDK_NGX_Parameter_ImageSuperResolution_Available, &Supported);
if (!Supported)
{
    std::cerr << "NVSDK_NGX_Feature_ImageSuperResolution Unavailable on this System" << std::endl;
    return 1;
}

// Set the default hyperparameters for inference.
params->Set(NVSDK_NGX_Parameter_Width, in_image_width);
params->Set(NVSDK_NGX_Parameter_Height, in_image_height);
params->Set(NVSDK_NGX_Parameter_Scale, myAppParams.uprez_factor);

// Get the scratch buffer size and create the scratch allocation.
// (if required)
size_t byteSize{ 0u };
void *scratchBuffer{ nullptr };
CK_NGX(NVSDK_NGX_CUDA_GetScratchBufferSize(NVSDK_NGX_Feature_ImageSuperResolution, params, &byteSize));
cudaMalloc(&scratchBuffer, byteSize > 0u ? byteSize : 1u);

// Update the parameter block with the scratch space metadata:
params->Set(NVSDK_NGX_Parameter_Scratch, scratchBuffer);
params->Set(NVSDK_NGX_Parameter_Scratch_SizeInBytes, (uint32_t)byteSize);

// Create the feature
CK_NGX(NVSDK_NGX_CUDA_CreateFeature(NVSDK_NGX_Feature_ImageSuperResolution, params, &DUHandle));

// Pass the pointers to the GPU allocations to the parameter block along with the format and size.
params->Set(NVSDK_NGX_Parameter_Color_SizeInBytes, in_image_row_bytes * in_image_height);
params->Set(NVSDK_NGX_Parameter_Color_Format, NVSDK_NGX_Buffer_Format_RGB8UI);
params->Set(NVSDK_NGX_Parameter_Color_Color, in_image_dev_ptr);
params->Set(NVSDK_NGX_Parameter_Output_SizeInBytes, out_image_row_bytes * out_image_height);
params->Set(NVSDK_NGX_Parameter_Output_Format, NVSDK_NGX_Buffer_Format_RGB8UI);
params->Set(NVSDK_NGX_Parameter_Output_Out, out_image_dev_ptr);

// Execute the feature.
CK_NGX(NVSDK_NGX_CUDA_EvaluateFeature(DUHandle, params, NGXTestCallback));

// Tear down the feature.
CK_NGX(NVSDK_NGX_CUDA_ReleaseFeature(DUHandle));
```
5.3. DLVSR (CUDA Only)

What Does This Feature Do?

Video Super-resolution (VSR) is a technique that constructs high-resolution (HR) video frames from low-resolution (LR) video frames, thereby improving the details within the frame and removing the artifacts caused by the imaging process of the low resolution camera.

VSR can be used for a variety of applications such as increasing the resolution of old content archived at lower resolutions or generating a video for high resolution broadcast (UHD) that was captured using a HD resolution camera.

Conventional Upscaling algorithms upscale the video by using interpolation algorithms but are unable to regenerate the details which could have been captured by a HR camera.

Deep Learning Video Super-Resolution (DLVSR) uses a machine learning approach in order to generate a higher resolution video from a lower resolution video and brings in details that cannot be generated using a conventional upscaling algorithm.

NGX DLVSR Sample

The NGX VSR sample application included in the NGX SDK:

- demonstrates the generation of a high resolution video from a low resolution video directly from a compressed file in a container.
- makes use of FFmpeg demuxer and decodes it using the NVDEC engine on the GPU. It quickly decodes the video file, generates RGB frames and output that can be directly sent to the NGX VSR plugin. It then encodes the VSR output and puts it in mp4 container.

For more information about supported decode/encode resolution by the NVDEC/NVENC engine on a GPU, see NVIDIA Video Codec SDK. If the resolution or codec is not supported by NVDEC/NVENC, use a software decoder and transfer the resulting RGB frames into the video memory to be sent to the NGX VSR plugin and then transfer the resulting high resolution RGB to system memory for use in a software encoder.

The following command reads *input.mp4* file and generates a video with x2 scaling factor to *output.mp4*. The scaling factor could be 2 or 3.

```
DLVSR.exe --input input.mp4 --factor 2 --output output.mp4
```
Supported Parameters

The following parameters are supported by DLVSR:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>The width of the input video.</td>
</tr>
<tr>
<td>Height</td>
<td>The height of the input video.</td>
</tr>
<tr>
<td>Scratch</td>
<td>The CUDA scratch buffer.</td>
</tr>
<tr>
<td>Scratch_SizeInBytes</td>
<td>The CUDA scratch buffer size in bytes.</td>
</tr>
<tr>
<td>Color</td>
<td>The CUDA input video frame buffer.</td>
</tr>
<tr>
<td>Color_Format</td>
<td>The input color precision, for example, NVSDK_NGX_Buffer_Format_RGB16F or NVSDK_NGX_Buffer_Format_RGB32F.</td>
</tr>
<tr>
<td>Color_SizeInBytes</td>
<td>The input video frame size.</td>
</tr>
<tr>
<td>Output</td>
<td>The CUDA output video frame buffer.</td>
</tr>
<tr>
<td>Output_Format</td>
<td>The output color precision, for example, NVSDK_NGX_Buffer_Format_RGB16F or NVSDK_NGX_Buffer_Format_RGB32F.</td>
</tr>
<tr>
<td>Output_SizeInBytes</td>
<td>The size of the output video frame.</td>
</tr>
<tr>
<td>Scale</td>
<td>Super-resolution scale factor (2 or 3).</td>
</tr>
</tbody>
</table>

Sample Code

The following code snippet shows how to use the DLVSR feature in NGX. The following steps are involved:

1. Create an NGX Context using `NVSDK_NGX_CUDA_Init()`.  
2. Get structure for setting input parameters using `NVSDK_NGX_CUDA_GetParameters()`.  
3. Get the scratch buffer size required by the DLSLOMO algorithm using `NVSDK_NGX_CUDA_GetScratchBufferSize()`, this depends on the input video width, height, and the scale factor chosen.  
5. Create the NGX DLSLOMO feature using `NVSDK_NGX_CUDA_CreateFeature()`.  
6. Run DLSLOMO using `NVSDK_NGX_CUDA_EvaluateFeature()`.  
7. Release the NGX DLSLOMO feature using `NVSDK_NGX_CUDA_ReleaseFeature()`.  
8. Free the NGX resources using `NVSDK_NGX_CUDA_Shutdown()`.

// IMPORTANT: ALWAYS CHECK FOR RETURN CODES TO ENSURE NGX CALLS ARE NOT FAILING
// Should be done when application is initialized
NVSDK_NGX_Result Status = NVSDK_NGX_CUDA_Init(MyApplicationID, MyRWAccessFolder, NVSDK_NGX_Version_API));
if(NVSDK_NGX_FAILED(Status))
{
    // Check error code, if NGX is not available on this machine disable it
}

// Once render target size is known we can create feature
NVSDK_NGX_Handle *FeatureHandle = nullptr;
NVSDK_NGX_Parameter *Params = nullptr;
Status = NVSDK_NGX_CUDA_GetParameters(&Params);
if(NVSDK_NGX_FAILED(Status)) { // Handle error
};
Params->Set(NVSDK_NGX_Parameter_Width, Width);
Params->Set(NVSDK_NGX_Parameter_Height, Height);
Params->Set(NVSDK_NGX_Parameter_Scale, ScaleFactor); // Must be 2x or 3x
size_t ByteSize;
Status = NVSDK_NGX_CUDA_GetScratchBufferSize(NVSDK_NGX_Feature_VideoSuperResolution, Params, &ByteSize);
if(NVSDK_NGX_FAILED(Status)) { // Handle error
};
void *ScratchBuffer = MyEngine::CreateCUDAResource(ByteSize);
Params->Set(NVSDK_NGX_Parameter_Scratch, ScratchBuffer);
Params->Set(NVSDK_NGX_Parameter_Scratch_SizeInBytes, (uint32_t)ByteSize);
Status = NVSDK_NGX_CUDA_CreateFeature(NVSDK_NGX_Feature_VideoSuperResolution, Params, &FeatureHandle);
if(NVSDK_NGX_FAILED(Status)) { // Handle error
};

// When needed call to upscale your input
// IMPORTANT: input needs to be 3 channel fp32 in range 0.0-255.0
Params->Set(NVSDK_NGX_Parameter_Color_SizeInBytes, ColorImageSize);
Params->Set(NVSDK_NGX_Parameter_Color_Format, NVSDK_NGX_Buffer_Format_RGB32F);
Params->Set(NVSDK_NGX_Parameter_Color, ColorCUDABuffer);
Params->Set(NVSDK_NGX_Parameter_Output_SizeInBytes, UpscaledImageSize);
Params->Set(NVSDK_NGX_Parameter_Output_Format, NVSDK_NGX_Buffer_Format_RGB32F);
Params->Set(NVSDK_NGX_Parameter_Output, UpscaledCUDABuffer);
Status = NVSDK_NGX_CUDA_EvaluateFeature(FeatureHandle, Params);
if(NVSDK_NGX_FAILED(Status)) { // Handle error
};

// When feature is no longer needed
Status = NVSDK_NGX_CUDA_ReleaseFeature(FeatureHandle);
if(NVSDK_NGX_FAILED(Status)) { // Handle error
};

// During shutdown
Status = NVSDK_NGX_CUDA_Shutdown();
if(NVSDK_NGX_FAILED(Status)) { // Handle error
};

5.4. DLSLOWMO (CUDA Only)

What Does This Feature Do?

The DLSLOWMO feature uses deep learning to create a slow motion video from an original input video. This feature introduces intermediate frames between subsequent frames in the original input video.

NGX DLSLOMO Sample

The NGX DLSLOWMO sample application included in the NGX SDK demonstrates the generation of intermediate frames via deep learning inference between each pair
of frames in a video in mp4 format directly from a compressed file in a container. This sample makes use of FFmpeg demuxer and decodes it using the NVDEC engine on the GPU. It quickly decodes the video file, generates RGB frames and output that can be directly sent to the NGX VSR plugin.

For more information about decode resolution by the NVDEC engine on a GPU, see NVIDIA Video Codec SDK. If the resolution or codec is not supported by NVDEC, use a software decoder and transfer the resulting RGB frames into the video memory to be sent to the NGX VSR plugin.

The following command reads \texttt{input.mp4} file and generates a new slow motion video with two times the number of frames output.mp4. The number of frames can be any integer but is limited by the available GPU memory.

\begin{verbatim}
slomo.exe --input input.mp4 --frames 2 --output output.mp4
\end{verbatim}

**Supported Parameters**

The following parameters are supported by DLSLOWMO:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>Width of a video input frame CUDA buffer.</td>
</tr>
<tr>
<td>Height</td>
<td>Height of a video input frame CUDA buffer.</td>
</tr>
<tr>
<td>Scratch</td>
<td>Scratch buffer allocated with \texttt{cudaMalloc}().</td>
</tr>
<tr>
<td>Scratch_SizeInBytes</td>
<td>Scratch buffer size.</td>
</tr>
<tr>
<td>Input1</td>
<td>CUDA buffer of linear memory (allocated with \texttt{cudaMalloc}()) containing the FP32 RGB pixels of the first input frame in planar CHW format.</td>
</tr>
<tr>
<td>Input1_Format</td>
<td>Color precision of the first input frame, \texttt{NVSDK_NGX_Buffer_Format_RGB32F}.</td>
</tr>
<tr>
<td>Input1_SizeInBytes</td>
<td>Size of the first input frame bytes.</td>
</tr>
<tr>
<td>Input2</td>
<td>CUDA buffer of linear memory (allocated with \texttt{cudaMalloc}()) containing the FP32 RGB pixels of the second input frame in planar CHW format.</td>
</tr>
<tr>
<td>Input2_Format</td>
<td>Color precision of the second input frame, \texttt{NVSDK_NGX_Buffer_Format_RGB32F}.</td>
</tr>
<tr>
<td>Input2_SizeInBytes</td>
<td>Size of the second input frame in bytes.</td>
</tr>
<tr>
<td>OutputX</td>
<td>CUDA buffer of linear memory (allocated with \texttt{cudaMalloc}()) to return the FP32 RGB pixels of output frame X in planar CHW format.</td>
</tr>
</tbody>
</table>
### NGX Features

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputX_Format</td>
<td>Color precision output frame X, NVSDK_NGX_Buffer_Format_RGB32F.</td>
</tr>
<tr>
<td>OutputX_SizeInBytes</td>
<td>Size of output frame X in bytes.</td>
</tr>
<tr>
<td>NumFrames</td>
<td>Number of new frames to generate between two frames in the original video.</td>
</tr>
</tbody>
</table>

### Sample Code

The following code snippet shows how to use the DLSLOMO feature in NGX. The following steps are involved:

1. Create an NGX Context using `NVSDK_NGX_CUDA_Init()`.
2. Get structure for setting input parameters using `NVSDK_NGX_CUDA_GetParameters()`.
3. Get the scratch buffer size required by the DLSLOMO algorithm using `NVSDK_NGX_CUDA_GetScratchBufferSize()`, this depends on the input video width, height, and the scale factor chosen.
5. Create the NGX DLSLOMO feature using `NVSDK_NGX_CUDA_CreateFeature()`.
6. Run DLSLOMO using `NVSDK_NGX_CUDA_EvaluateFeature()`.
7. Release the NGX DLSLOMO feature using `NVSDK_NGX_CUDA_ReleaseFeature()`.
8. Free the NGX resources using `NVSDK_NGX_CUDA_Shutdown()`.

```c
CK_NGX(NVSDK_NGX_CUDA_Init(m_ulAppId, m_wcDataPath, m_SDKVersion));
CK_NGX(NVSDK_NGX_CUDA_GetParameters(&m_pParams));

m_pParams->Set(NVSDK_NGX_Parameter_Width, Width);
m_pParams->Set(NVSDK_NGX_Parameter_Height, Height);
m_pParams->Set(NVSDK_NGX_Parameter_NumFrames, Frames);
CK_NGX(NVSDK_NGX_CUDA_GetScratchBufferSize(NVSDK_NGX_Feature_SlowMotion, m_pParams, &m_uScratchSize));

if (m_uScratchSize)
{
    CK_CUDA(cudaMalloc(&m_ScratchBuffer, m_uScratchSize));
    m_pParams->Set(NVSDK_NGX_Parameter_Scratch, m_ScratchBuffer);
    m_pParams->Set(NVSDK_NGX_Parameter_Scratch_SizeInBytes, m_uScratchSize);
}

CK_NGX(NVSDK_NGX_CUDA_CreateFeature(NVSDK_NGX_Feature_SlowMotion, m_pParams, &m_hSloMo));

m_pParams->Set(NVSDK_NGX_Parameter_Input1_SizeInBytes, InputSize);
m_pParams->Set(NVSDK_NGX_Parameter_Input1_Format, NVSDK_NGX_Buffer_Format_RGB32F);
m_pParams->Set(NVSDK_NGX_Parameter_Input1, Input1);
m_pParams->Set(NVSDK_NGX_Parameter_Input2_SizeInBytes, InputSize);
m_pParams->Set(NVSDK_NGX_Parameter_Input2_Format, NVSDK_NGX_Buffer_Format_RGB32F);
m_pParams->Set(NVSDK_NGX_Parameter_Input2, Input2);
```
for (uint32_t i = 0; i < Frames; i++)
{
    const std::string outputBufName{ std::string(NVSDK_NGX_Parameter_Output) +
        std::to_string(i) };
    const std::string outputBufFormat{ outputBufName + std::string("." +
        std::string(NVSDK_NGX_Parameter_Format) };
    const std::string outputBufSizeInBytes{ outputBufName + std::string("." +
        std::string(NVSDK_NGX_Parameter_SizeInBytes) };
    m_pParams->Set(outputBufName.c_str(), Output[i].get());
    m_pParams->Set(outputBufFormat.c_str(), NVSDK_NGX_Buffer_Format_RGB32F);
    m_pParams->Set(outputBufSizeInBytes.c_str(), OutputSize);
}
CK_NGX(NVSDK_NGX_CUDA_EvaluateFeature(m_hSloMo, m_pParams, NGXTestCallback));
CK_NGX(NVSDK_NGX_CUDA_Shutdown());

5.5. DLINPAINTING (CUDA Only)

What Does This Feature Do?

The DLINPAINTING features applies deep learning to remove parts of an image specified by a mask and replaces the missing pixels with pixels inferred from the surrounding pixels.

NGX DLINPAINTING Sample

The NGX InPaint sample application included in the NGX SDK demonstrates the replacement of image pixels using a mask and deep learning inference in an 8-bit RGB image. This sample uses the `getRgbImage()` and `putRgbImage()` helper functions implemented on top of OpenCV to read and write most common image formats based upon the file extension.

The following command reads `input.png` file and `mask.png` to remove and replace the pixels specified by the mask in the original image using in paint model 0.

```
inpaint.exe --input input.png --mask mask.png --model 0
```

Supported Parameters

The following parameters are supported by DLINPAINTING:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>Width of the input frame CUDA buffer.</td>
</tr>
<tr>
<td>Height</td>
<td>Height of the input frame CUDA buffer.</td>
</tr>
<tr>
<td>Scratch</td>
<td>Scratch buffer allocated with <code>cudaMalloc()</code>.</td>
</tr>
<tr>
<td>Scratch_SizeInBytes</td>
<td>Scratch buffer size.</td>
</tr>
</tbody>
</table>
### NGX Features

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input1</td>
<td>CUDA buffer of linear memory (allocated with cudaMalloc()) containing the 8-bit RGB pixels of the input image.</td>
</tr>
<tr>
<td>Input1_Format</td>
<td>Color precision of the input image, NVSDK_NGX_Buffer_Format_RGBA8UI.</td>
</tr>
<tr>
<td>Input1_SizeInBytes</td>
<td>Size of input image frame in bytes.</td>
</tr>
<tr>
<td>Input2</td>
<td>CUDA buffer of linear memory (allocated with cudaMalloc()) containing the 8-bit RGB pixels of the mask image.</td>
</tr>
<tr>
<td>Input2_Format</td>
<td>Color precision of the mask image, NVSDK_NGX_Buffer_Format_RGBA8UI.</td>
</tr>
<tr>
<td>Input2_SizeInBytes</td>
<td>Size of the mask image in bytes.</td>
</tr>
<tr>
<td>Output</td>
<td>CUDA buffer of linear memory (allocated with cudaMalloc()) to return the 8-bit RGB pixels of output image.</td>
</tr>
<tr>
<td>Output_Format</td>
<td>Color precision output image, NVSDK_NGX_Buffer_Format_RGBA8UI.</td>
</tr>
<tr>
<td>Output_SizeInBytes</td>
<td>Size of the output image in bytes.</td>
</tr>
<tr>
<td>Model</td>
<td>Inpainting model (0 or 1).</td>
</tr>
</tbody>
</table>

### Sample Code

The following code snippet shows how to use the DLINPAINTING feature in NGX. The following steps are involved:

1. Create an NGX Context using `NVSDK_NGX_CUDA_Init()`.
2. Get structure for setting input parameters using `NVSDK_NGX_CUDA_GetParameters()`.
3. Get the scratch buffer size required by the DLINPAINTING algorithm using `NVSDK_NGX_CUDA_GetScratchBufferSize()`, this depends on the input video width, height, and the scale factor chosen.
5. Create the NGX DLINPAINTING feature using `NVSDK_NGX_CUDA_CreateFeature()`.
6. Run DLINPAINTING using `NVSDK_NGX_CUDA_EvaluateFeature()`.
7. Release the NGX DLINPAINTING feature using `NVSDK_NGX_CUDA_ReleaseFeature()`.
8. Free the NGX resources using `NVSDK_NGX_CUDA_Shutdown()`.
// Initialize NGX
CK_NGX(NVSDK_NGX_CUDA_Init(app_id, L"./", NVSDK_NGX_Version_API));

// Get the parameter block.
CK_NGX(NVSDK_NGX_CUDA_GetParameters(&params));

// Verify feature is supported
int Supported = 0;
params->Get(NVSDK_NGX_Parameter_InPainting_Available, &Supported);
if (!Supported)
{
    std::cerr << "NVSDK_NGX_Feature_InPainting Unavailable on this System" << std::endl;
    return 1;
}

// Set the default hyperparameters for inference.
params->Set(NVSDK_NGX_Parameter_Width, image_width);
params->Set(NVSDK_NGX_Parameter_Height, image_height);
params->Set(NVSDK_NGX_Parameter_Model, myAppParams.model);

// Get the scratch buffer size and create the scratch allocation.
size_t byteSize{ 0u };
void *scratchBuffer{ nullptr };
if (NVSDK_NGX_FAILED(NVSDK_NGX_CUDA_GetScratchBufferSize(NVSDK_NGX_Feature_InPainting, params, &byteSize))
{
    std::cerr << "Error Getting NGX Scratch Buffer Size. " << std::endl;
    return 1;
}
cudaMalloc(&scratchBuffer, byteSize > 0u ? byteSize : 1u);

// Update the parameter block with the scratch space metadata.
params->Set(NVSDK_NGX_Parameter_Scratch, scratchBuffer);
params->Set(NVSDK_NGX_Parameter_Scratch_SizeInBytes, (uint32_t)byteSize);
CK_NGX((NVSDK_NGX_CUDA_CreateFeature(NVSDK_NGX_Feature_InPainting, params, &DUHandle)));

// Pass the pointers to the GPU allocations to the parameter block along with
// the format and size.
params->Set(NVSDK_NGX_Parameter_Input1, in_image_dev_ptr);
params->Set(NVSDK_NGX_Parameter_Input1_Format, NVSDK_NGX_Buffer_Format_RGBA8UI);
params->Set(NVSDK_NGX_Parameter_Input1_SizeInBytes, in_image_row_bytes * in_image_height);

params->Set(NVSDK_NGX_Parameter_Input2, in_mask_dev_ptr);
params->Set(NVSDK_NGX_Parameter_Input2_Format, NVSDK_NGX_Buffer_Format_RGBA8UI);
params->Set(NVSDK_NGX_Parameter_Input2_SizeInBytes, in_image_row_bytes * in_image_height);

params->Set(NVSDK_NGX_Parameter_Output, out_image_dev_ptr);
params->Set(NVSDK_NGX_Parameter_Output_Format, NVSDK_NGX_Buffer_Format_RGBA8UI);
params->Set(NVSDK_NGX_Parameter_Output_SizeInBytes, in_image_row_bytes * in_image_height);

// Execute the feature.
CK_NGX(NVSDK_NGX_CUDA_EvaluateFeature(DUHandle, params, NGXTestCallback));

// Tear down the feature.
CK_NGX(NVSDK_NGX_CUDA_ReleaseFeature(DUHandle));
6.1. Error Codes

If an error is detected during NGX execution, one of the following error codes will be reported:

**NVSDK_NGX_Result_FAIL_FeatureNotSupported**
- Feature is not supported on current hardware.

**NVSDK_NGX_Result_FAIL_PlatformError**
- Platform error, for example, check the d3d12 debug layer log for more information.

**NVSDK_NGX_Result_FAIL_FeatureAlreadyExists**
- Feature with given parameters already exists.

**NVSDK_NGX_Result_FAIL_FeatureNotFound**
- Feature with provided handle does not exist.

**NVSDK_NGX_Result_FAIL_InvalidParameter**
- Invalid parameter was provided.

**NVSDK_NGX_Result_FAIL_ScratchBufferTooSmall**
- Provided buffer is too small.

**NVSDK_NGX_Result_FAIL_NotInitialized**
- SDK was not initialized properly.

**NVSDK_NGX_Result_FAIL_UnsupportedInputFormat**
- Unsupported format used for input/output buffers.

**NVSDK_NGX_Result_FAIL_RWFlagMissing**
- Feature input/output needs RW access (UAV) (d3d11/d3d12 specific).

**NVSDK_NGX_Result_FAIL_MissingInput**
- Feature was created with specific input but none is provided at evaluation.
NVSDK_NGX_Result_FAIL_UnableToInitializeFeature
  Feature is misconfigured or not available on the system.

NVSDK_NGX_Result_FAIL_OutOfDate
  NGX runtime libraries require a newer version, update the NVIDIA display driver to a current version.

NVSDK_NGX_Result_FAIL_OutOfGPUMemory
  Feature requires more GPU memory than is available on the system.

NVSDK_NGX_Result_FAIL_UnsupportedFormat
  Format used in input buffer(s) is not supported by feature.

6.2. Support

For any NGX related questions, email NGXSupport@nvidia.com or visit the forum at https://devtalk.nvidia.com/default/board/329/ngx-sdk/.
Notice

THE INFORMATION IN THIS GUIDE AND ALL OTHER INFORMATION CONTAINED IN NVIDIA DOCUMENTATION REFERENCED IN THIS GUIDE IS PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE INFORMATION FOR THE PRODUCT, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the product described in this guide shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

THE NVIDIA PRODUCT DESCRIBED IN THIS GUIDE IS NOT FAULT TOLERANT AND IS NOT DESIGNED, MANUFACTURED OR INTENDED FOR USE IN CONNECTION WITH THE DESIGN, CONSTRUCTION, MAINTENANCE, AND/OR OPERATION OF ANY SYSTEM WHERE THE USE OR A FAILURE OF SUCH SYSTEM COULD RESULT IN A SITUATION THAT THREATENS THE SAFETY OF HUMAN LIFE OR SEVERE PHYSICAL HARM OR PROPERTY DAMAGE (INCLUDING, FOR EXAMPLE, USE IN CONNECTION WITH ANY NUCLEAR, AVIONICS, LIFE SUPPORT OR OTHER LIFE CRITICAL APPLICATION). NVIDIA EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR SUCH HIGH RISK USES. NVIDIA SHALL NOT BE LIABLE TO CUSTOMER OR ANY THIRD PARTY, IN WHOLE OR IN PART, FOR ANY CLAIMS OR DAMAGES ARISING FROM SUCH HIGH RISK USES.

NVIDIA makes no representation or warranty that the product described in this guide will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this guide. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this guide, or (ii) customer product designs.

Other than the right for customer to use the information in this guide with the product, no other license, either expressed or implied, is hereby granted by NVIDIA under this guide. Reproduction of information in this guide is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, cuDNN, cuFFT, cuSPARSE, DIGITS, DGX, DGX-1, DGX Station, GRID, Jetson, Kepler, NGX, NVIDIA GPU Cloud, Maxwell, NCCL, NVLink, Pascal, Tegra, TensorRT, Tesla and Volta are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2019 NVIDIA Corporation. All rights reserved.

www.nvidia.com