NVIDIA Optical Flow Engine-Assisted Frame Rate Up Conversion

Programming Guide

vNVFRUC_PG-09417-001_v03 | February 2023

Table of Contents

Chapter 1. Introduction

NVIDIA GPUs starting from Turing generation contain a hardware-based Optical Flow Accelerator (NVOFA) that gives flow vectors map between the two frames. NVIDIA Optical Flow SDK gives access to NVOFA via Optical Flow APIs.

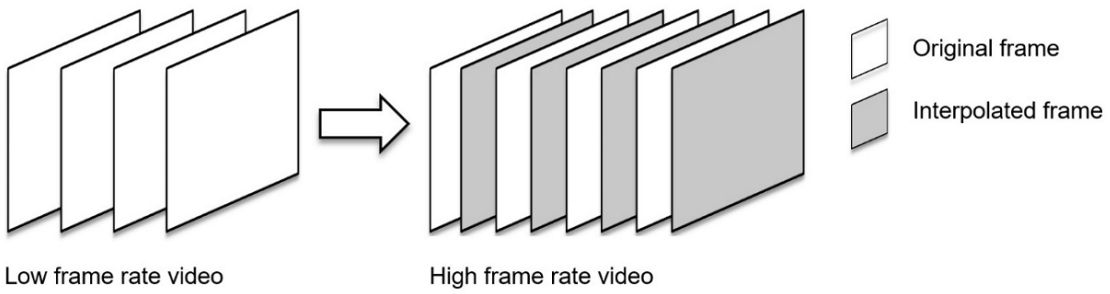
NVIDIA Optical Flow SDK 4.0 and later SDKs include NVOFA assisted Frame Rate Up Conversion (FRUC) library. The library exposes FRUC APIs that can be used for frame rate up conversion of game or video.

This document provides information on how to use the FRUC APIs. It is expected that the developer should have familiarity with Windows and/or Linux development environment.

1.1. Frame Rate Up Conversion

Frame Rate Up Conversion is a technique that generates higher frame rate video from lower frame rate video by inserting interpolated frames into it. Such high frame rate video shows smooth continuity of motion across frames that improves perceived visual quality of video.

Figure 1. Frame Rate Up Conversion



1.2. NVIDIA FRUC library

NVIDIA FRUC library exposes FRUC APIs that take two consecutive frames and generate an interpolated frame in between. These APIs can be used for frame rate up conversion of gaming or video content.

The library internally uses NVOFA hardware engine and CUDA. As a result, the frame interpolation using FRUC library is much faster compared to software-only methods.

The library supports ARGB and NV12 input surface formats. It can be directly integrated into a DirectX game or a CUDA application.

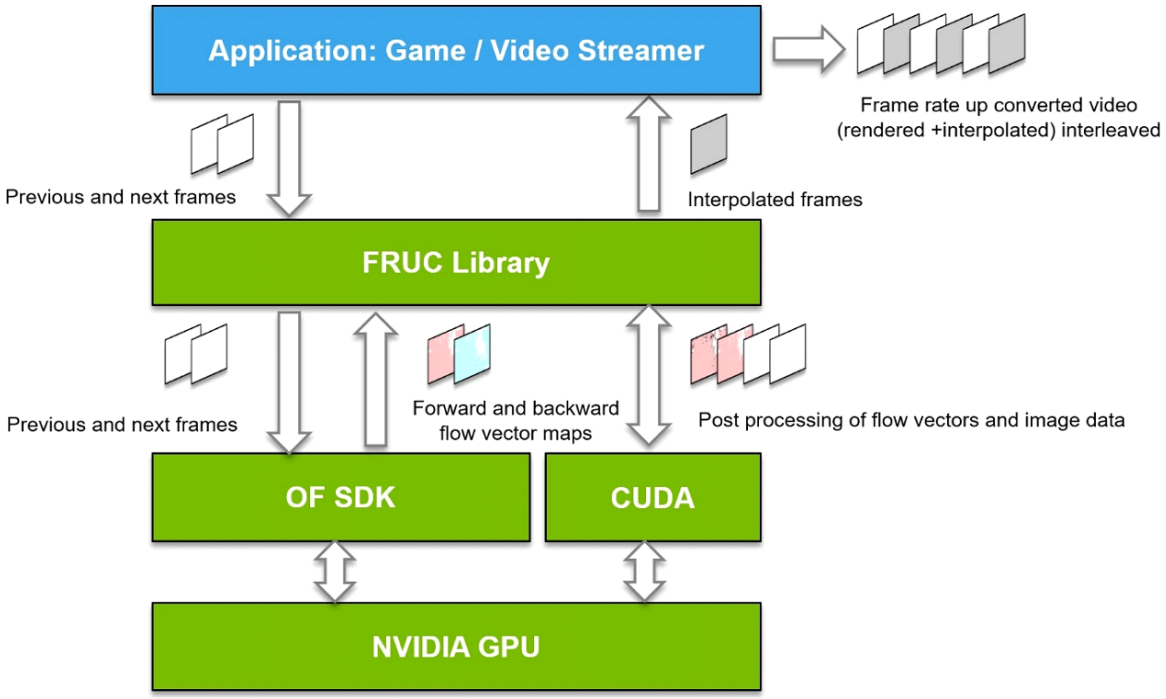
The FRUC library works on Windows OS (Windows 10 and above) and Linux OS (distributions Ubuntu 18 and above). It needs NVIDIA display driver version 511.65 or above on Windows and NVIDIA display driver version 510.47.03 or above on Linux.

Chapter 2. How FRUC Library Works?

2.1. FRUC Library Usage Overview

Here is a block diagram showing how applications can use FRUC library for frame rate up conversion.

Figure 2. FRUC Library Software Stack



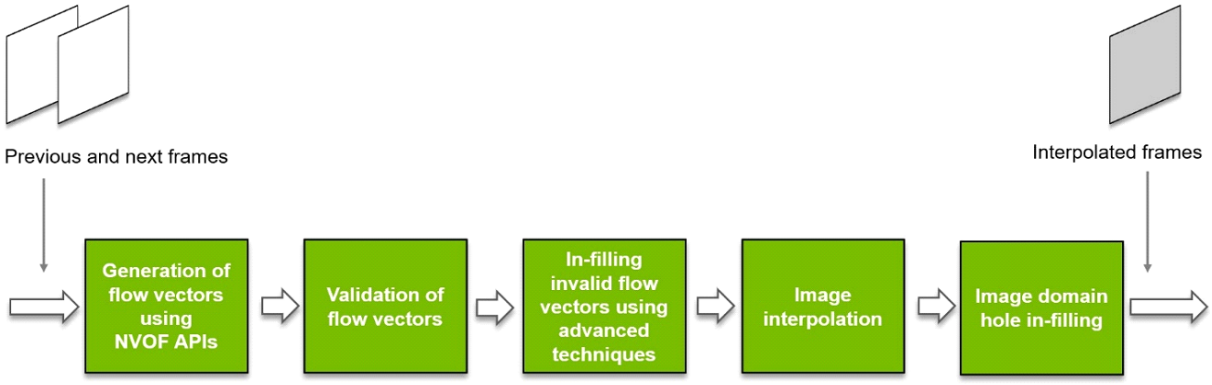
1. Application passes consecutive frames to FRUC library.
2. FRUC library uses frame from current call (next frame) and cached frame from previous call (previous frame) to interpolate intermediate frame. The library first calls NVIDIA Optical flow APIs to get forward and backward flow vector maps between the two frames (previous and next). It then uses CUDA accelerated techniques to generate an interpolated frame between the two frames.

3. The interpolated frame is returned to the application. Application then interleaves the interpolated frames with original frames and generates video with an increased framerate.

2.2. Inside FRUC Library

Here is a simplified functional block diagram of FRUC library.

Figure 3. Simplified block diagram of FRUC library



Here is a brief explanation about how FRUC library processes two consecutive frames and generates an interpolated frame.

Input to the FRUC library are two consecutive frames (previous and next)

1. Generation of flow vectors using NVOFA APIs

   Consecutive frame pairs are sent to NVIDIA Optical flow engine using NVOFA APIs to get forward and backward flow vector maps between them.
2. Validation of flow vectors

   All flow vectors in the flow vectors maps are then validated using forward-backward consistency check. The flow vectors that do not pass the consistency check are rejected, resulting in sparse optical flow vector maps.
3. Infilling invalid flow vectors using advanced techniques

   Using available flow vectors and advanced techniques, accurate flow vectors are generated to fill in the rejected flow vectors, converting the sparse flow vector map into a fully dense flow vector map.
4. Image interpolation

   Using the dense flow vector map, an interpolated frame between the two input frames is generated.

Such an image may contain a few hole regions (pixels that don't have valid color).

5. Image domain hole in-filling

   Holes in the interpolated frame are filled using image domain hole infilling techniques to generate final interpolated image.

   The final interpolated frame is returned to the application.

2.3. FRUC Library Components

Optical Flow SDK includes the following components of FRUC Library:

- `NvFRUC.dll`: DLL that exposes FRUC APIs on Windows.
- `libNvFRUC.so`: .so file that exposes FRUC APIs on Linux (Ubuntu).
- `NvFRUC.h`: NvFRUC API header file.
- `NvFRUCSample`: application source code: Shows how to use NVIDIA FRUC library.
- `ReadMe.pdf`: Includes instructions on how to build and run `NvFRUCSample` application
- `NVOFA_FRUC.pdf`: Contains detailed information about FRUC APIs and how to use those (this document).

Chapter 3. Programming Using FRUC APIs

3.1. Basic Programming Flow

NVIDIA FRUC APIs are designed to accept raw video frames in NV12 or ARGB format and return an interpolated frame between them.

Broadly, the programming flow consists of the following steps:

1. Call API `PtrToFuncNvFRUCCreate` to create FRUC instance.
2. Create input resources (DirectX NV12 or CUDA ARGB surfaces) and register them with FRUC library using API `PtrToFuncNvFRUCRegisterResource`.
3. Call API `PtrToFuncNvFRUCProcess` to process input frames to generate interpolated frames.
4. Call API `PtrToFuncNvFRUCUnregisterResource` to unregister input resources with FRUC library so that they can be destroyed.
5. Call API `PtrToFuncNvFRUCDestroy` to destroy FRUC instance.

Header file `NvFRUC.h` has details of structures used in these functions.

3.2. Background

As seen in Simplified block diagram of FRUC library, the FRUC library uses Optical Flow APIs to get flow vector maps between the two consecutive frames. The client application first needs to allocate buffers to hold input frame pair data and interpolated frame data. Client application then needs to pass the address of these buffers to the FRUC library. The FRUC library then makes use of NVOFA APIs, advanced CUDA algorithms to produce an interpolated frame and copies it to the output buffer shared by the client application.

The client application can create input and output resources using either DirectX APIs or CUDA driver APIs.

- **Resource creation using DirectX 11 API**

  In this case the client application creates shared texture using DirectX 11 APIs and shares the pointers of the device that creates textures and texture itself to FRUC library.

Since the textures are shared between the client application and FRUC library, it is the client application's responsibility to ensure synchronization to avoid race conditions. The synchronization mechanism to be used is ID3D11Fence on Windows OS build number 1703 and above or IDXGIKeyedMutex on rest of Windows OS's. Client application and FRUC library uses the CUDA-DirectX graphics interoperability API for thread safe read-write of buffers. To know more about this, please visit Direct3D 11 Interoperability.

- **Resource creation using CUDA Driver API**

  If the client application uses CUDA API to create the shared resources, it just needs to pass the resource pointer to the FRUC library.

3.3. Example Of FRUC API Usage

Source code of NvFRUCSample application demonstrates use of the APIs exposed by NVIDIA FRUC library for frame rate up conversion. The application accepts input video either as YUV file or as sequence of PNG frames and generates outputs as follows:

- **Input as YUV video sequence**

  In this case the application takes a YUV(YUV420) video sequence, interpolates intermediate frames between the consecutive frames, interleaves interpolated frames with original frames to generate output YUV video. The output video thus generated has double the framerate as that of input video.

- **Input as PNG frame sequence**

  In this case the application takes sequence of frames in PNG format, interpolates intermediate frames between the consecutive frames and saves those in PNG format.

`NvFRUCSample` has the following helper classes to create and handle shared resources. You could reuse these helper classes and other parts of code of `NVFRUCSample` application in your custom application.

- **FrameGeneratorD3D11**

  This class handles the creation of `ID3D11Device`, `IDXGIKeyedMutex` and `ID3D11Fence` interfaces. It also handles reading and writing of shared surfaces with synchronization.

- **FrameGeneratorCUDA**

  This handles the creation of `cuDevicePtr`, `cuArray` interface pointers and sharing these pointers to FRUC library. It also handles reading and writing into CUDA device memory.

- **BufferManager**

  This class handles device-to-host and host-to-device CUDA memory transfers between FRUC `NvFRUCSample` application and FRUC library.

3.4. Using FRUC APIs

3.4.1. Creating FRUC Instance

Load `NvFRUC.dll` (Windows) or `libNvFRUC.so` (on Linux) as follows.

- **Windows**

```
SecureLoadLibrary(L"NvFRUC.dll", &hDLL);
```

  We recommend loading NvFRUC.dll using SecureLoadLibrary() to ensure Nvidia Signed Library is loaded by the application.

- **Linux**

```
hDLL = dlopen("libNvFRUC.so", RTLD_LAZY);
```

Retrieve the addresses of functions exported by FRUC library as follows. Signatures of exported functions are available in header file NvFRUC.h.

```
NvFRUCCreate = (PtrToFuncNvFRUCCreate)GETPROCEDUREADDRESS(
                                        DLL,
                                        CreateProcName);
NvFRUCRegisterResource =(PtrToFuncNvFRUCRegisterResource)GETPROCEDUREADDRESS(
                                        hDLL,
                                        RegisterResourceProcName);
NvFRUCUnregisterResource = (PtrToFuncNvFRUCUnregisterResource)GETPROCEDUREADDRESS(
                                        hDLL,
                                        UnregisterResourceProcName);
NvFRUCProcess = (PtrToFuncNvFRUCProcess)GETPROCEDUREADDRESS(
                                        hDLL,
                                        ProcessProcName);
NvFRUCDestroy = (PtrToFuncNvFRUCDestroy)GETPROCEDUREADDRESS(
                                        hDLL,
                                        DestroyProcName);
```

To create NvFRUC instance call `NvFRUCCreate` function as follows.

```
NvFRUC_CREATE_PARAM createParams = { 0 };
NvFRUCHandle hFRUC;
createParams.pDevice = objFrameGenerator->GetDevice();
createParams.uiHeight = stArgs.m_Height;
createParams.uiWidth = stArgs.m_Width;
createParams.eResourceType = (NvFRUCResourceType)stArgs.m_ResourceType;
createParams.eSurfaceFormat = (NvFRUCSurfaceFormat)stArgs.m_InputSurfaceFormat;
createParams.eCUDAResourceType = (NvFRUCCUDAResourceType)stArgs.m_CudaResourceType;

//Initialize FRUC pipeline which internally initializes Optical flow engine
status = NvFRUCCreate(
        &createParams,
        &hFRUC);
```

Here is a brief explanation about parameters of structure `NvFRUC_CREATE_PARAM` that you need to pass to `NvFRUCCreate` function.

- **pDevice**(input): This is pointer to `ID3D11Device` interface. This pointer is shared with the FRUC library. This is used only if the client is using DirectX API for resource creation. It should be NULL in case you are using CUDA APIs for resource creation.
- **uiHeight**(input): Height of input surface to be created by client application.
- **uiWidth**(input): Width of input surface to be created by client application.
- **eResourceType**(input): Set this to 1 if you are creating shared resources as DirectX 11 texture. In case you are creating a shared resource as `cuDevicePtr` or `cuArray`, set this to 0.
- **eSurfaceFormat**(input): Set this to 0 for surface format NV12 and 1 for surface format ARGB. Surface format is independent of the API being used to create resources.
- **eCUDAResourceType**(input): In case you are using CUDA APIs for resource creation set this parameter to 0 for `cuDevicePtr` and 1 for `cuArray`.

If this function succeeds, it returns handle to FRUC instance that is required in all subsequent functions.

3.4.2. Registering Resources

Register the resources created by client with FRUC library using `NvFRUCRegisterResource` function as follows.

```
NvFRUC_REGISTER_RESOURCE_PARAM regOutParam = { 0 };
objFrameGenerator->GetResource(
                    regOutParam.pArrResource,
                    regOutParam.uiCount);
                    regOutParam.pD3D11FenceObj = objFrameGenerator->GetFenceObj();

status = NvFRUCRegisterResource(
                    hFRUC,
                    &regOutParam);
```

Here the `objFrameGenerator` is an object of class `FrameGeneratorD3D11` or `FrameGeneratorCUDA`. It creates the required resources during initialization. The `objFrameGenerator->GetResource()` function called above arranges these resources in the form of an array of `void` pointers which are then passed to `NvFRUCRegisterResource()` function.

Fill in `NvFRUC_REGISTER_RESOURCE_PARAM` structure as follows.

- **pArrResource**(input): Array of pointers to input and output resources.
- **uiCount**(input): Total number of input and output resources.

If the function call succeeds it returns `NvFRUC_SUCCESS`.

3.4.3. Interpolating Intermediate Frame

Provide the consecutive input frames and get interpolated frame by calling `NvFRUCProcess` in loop as follows.

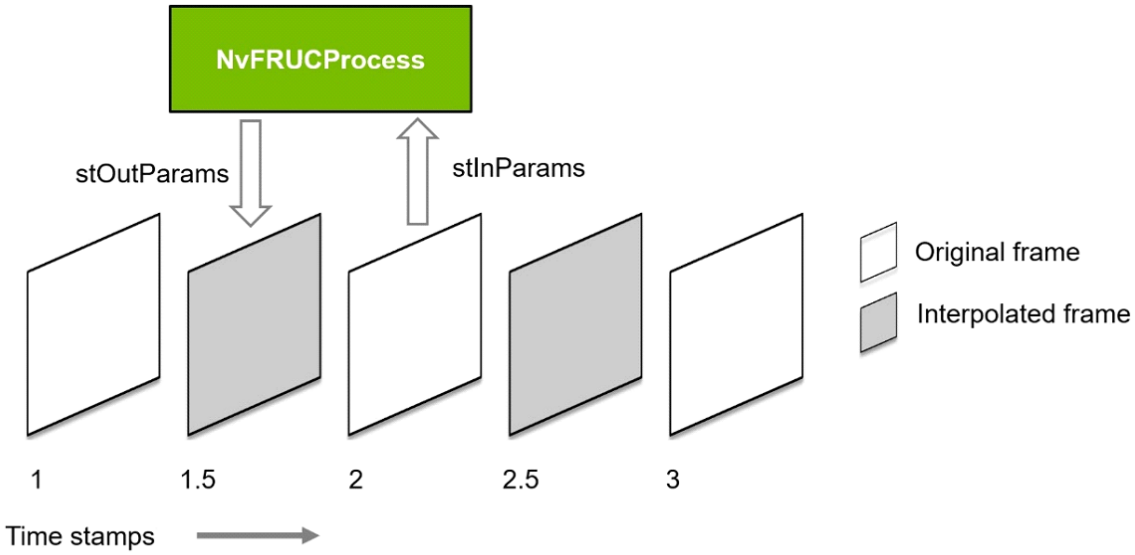
```
NvFRUC_PROCESS_IN_PARAMS stInParams = { 0 };
```

```
NvFRUC_PROCESS_OUT_PARAMS stOutParams = { 0 };

status = NvFRUCProcess(
                hFRUC,
                &stInParams,
                &stOutParams);
```

Here is a figure showing how to use NvFRUCProcess function.

Figure 4. How to use NvFRUCProcess function



Assume that you have a sequence of consecutive frames at timestamps 1, 2, 3 and so on. You wish to interpolate frames in between these frames at timestamps 1.5, 2.5, 3.5 and so on.

Call `NvFRUCProcess` in loop with `stInParams` set to frame with timestamp 1, 2, 3 and timestamp field in `stOutParams` set to 1.5, 2.5, 3.5 and so on. For the first call, `NvFRUCProcess` function returns frame at timestamp 1 itself as it cannot interpolate frame using just one frame. From the next call onwards, this function returns interpolated frames 1.5, 2.5, 3.5 and so on.

`NvFRUCProcess` API can be used to interpolate frame at any time-stamp between the two frames. e.g. 1.25, 1.50, 1.75 etc. Please use the values of `stInParams.nTimeStamp` and `stOutParams.nTimeStamp` accordingly.

`NvFRUCProcess` API uses frame from current call and the cached frame from previous call to interpolate the intermediate frame. Users should not call `NvFRUCProcess` multiple times with the same frame.

Fill in the `stInParams` and `stOutParams` structures before calling `NvFRUCProcess` API as follows:

`stInParams` is a structure of type `NvFRUC_PROCESS_IN_PARAMS` that has the following members:

Fill in `stFrameDataInput` struct as follows.

- **pFrame**(input): Pointer to raw input frame data.
- **nTimeStamp**(input): Timestamp of input frame.
- **bHasFrameRepetitionOccurred**(ignored): The value of this flag is ignored by FRUC library in parameter `stFrameDataInput`.
- **uSyncWait**(output): This member is used for synchronization of CUDA-DirectX interop in case you are using FRUC API in a DirectX application such as a game on Windows. FRUC library supports synchronization using fence on Windows OS build 1703 and above and keyed mutex on other windows OS builds. If you are using `ID3D11Fence`, increment the fence value here so that the library can acquire the input resource, else increment key value. For more details, please refer to graphics-interoperability section in NVIDIA CUDA programming guide.

`stOutParams` is a structure of type `NvFRUC_PROCESS_OUT_PARAMS` that has the following members:

- **pFrame**(output): Pointer to raw output frame data.
- **nTimeStamp**(input): Timestamp of frame to be interpolated.
- **bHasFrameRepetitionOccurred**(output): FRUC library returns the previous frame as interpolated frame in case the interpolated frame does not meet a certain quality bar. In such a case, this flag would be set to true by FRUC library. The application can monitor this flag, if useful..

On success the function returns `NvFRUC_SUCCESS`. If you are using FRUC library in DirectX application, then you need to wait on user thread till `NvFRUCProcess()` completes. For CUDA APIs, the function call is a blocking call.

3.4.4. Unregistering Resources

Unregister the resource using `NvFRUCUnregisterResource` function as follows.

```
NvFRUC_UNREGISTER_RESOURCE_PARAM stUnregisterResourceParam = { 0 };
memcpy(stUnregisterResourceParam.pArrResource,
            regOutParam.pArrResource,
            regOutParam.uiCount * sizeof(IUnknown*));

stUnregisterResourceParam.uiCount = regOutParam.uiCount;

status = NvFRUCUnregisterResource(
            hFRUC,
            &stUnregisterResourceParam);
```

Fill in `NvFRUC_UNREGISTER_RESOURCE_PARAM` structure as follows.

- **pArrResource**(input): Array of pointers to input and output resources.
- **uiCount**(input): Total number of input and output resources.

If the function call succeeds it returns `NvFRUC_SUCCESS`.

3.4.5. Destroying FRUC Instance

In the end, destroy FRUC instance using `NvFRUCDestroy` function as follows:

```
status = NvFRUCDestroy(hFRUC);
```

This function destroys the FRUC instance and returns `NvFRUC_SUCCESS` if it succeeds.

3.4.6. Diagnostics

All FRUC APIs status `NvFRUC_SUCCESS` if they succeed. In case of failure, the APIs return error codes hinting at the possible causes of failure. FRUC header `NvFRUC.h` has a list of all such error codes.

Chapter 4. Prerequisites

Once you integrate FRUC APIs into your application, you can build the application and run it on the target system. Do not run the sample application executable with elevated permission. Target system needs to have the following prerequisites for using FRUC library:

- NVIDIA GPU (Graphics Processing Units): Turing or above, with Optical Flow hardware support
- Windows OS: Windows 10 or above with latest updates
- Linux OS: Distributions Ubuntu 18 or above
- NVIDIA Windows display driver version 511.65 or above
- NVIDIA Linux display driver version 510.47.03 or above

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgment, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform 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 document. NVIDIA accepts no 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 document or (ii) customer product designs.

Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, CUDA Toolkit, cuDNN, DALI, DIGITS, DGX, DGX-1, DGX-2, DGX Station, DLProf, GPU, Jetson, Kepler, Maxwell, NCCL, Nsight Compute, Nsight Systems, NVCaffe, NVIDIA Deep Learning SDK, NVIDIA Developer Program, NVIDIA GPU Cloud, NVLink, NVSHMEM, PerfWorks, Pascal, SDK Manager, Tegra, TensorRT, TensorRT Inference Server, Tesla, TF-TRT, Triton Inference Server, Turing, 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

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com