NVIDIA OPTICAL FLOW SDK

Programming Guide

vNVOF_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 (hereafter referred to as NVOFA). The NVOFA hardware accepts a pair of YUV/RGB frames as input and generates a map of flow vectors between the two frames. NVOFA engine's capabilities can be accessed using the NVIDIA Optical Flow APIs (hereafter referred to as NVOF APIs), exposed via NVIDIA Optical Flow SDK.

This document provides information on how to program the NVOFA using the NVOF APIs exposed in the SDK. The NVOF APIs are supported on Windows (Windows 7 and above) and Linux.

It is expected that the developers should have familiarity with Windows and/or Linux development environment.

NVOF API guarantees backward compatibility (and will make explicit reference whenever backward compatibility is broken). This means that applications compiled with older versions of released API will continue to work on future driver versions released by NVIDIA.

Chapter 2. Basic Programming Flow

The NVIDIA OFAPI is designed to accept raw video frames (8-bit YUV or RGB format) and output the flow vectors.

Broadly, the programing flow consists of the following steps:

1. Create the optical flow context
2. Initialize the NVOF API interface
3. Allocate input/output buffers
4. Set up the desired parameters
5. Kick off the NVOFA engine
6. Clean-up - release all allocated input/output buffers
7. Close the session

These steps are explained in the rest of the document and demonstrated in the sample applications included in the Optical Flow SDK package.

Chapter 3. Initializing NVOF API

3.1. Opening the session

Developers can create a client application that calls NVOF APIs exposed by `nvOFAPI.dll` for Windows or `libnvidia-opticalflow.so` for Linux. These libraries are installed as part of the NVIDIA display driver. The client application can link to these libraries at run-time using `GetProcAddress()` on Windows and `dlsym()` on Linux.

The NVIDIA OFAPI supports use of the following types of interfaces:

- **DirectX 11** - DirectX 11 is supported on Windows 8 and above.
- **DirectX 12** - DirectX 12 is supported on Windows 10 20H1 and above.
- **CUDA** – CUDA interface is supported on Linux and Windows (Windows 7 and above).
- **Vulkan** – Vulkan interface is supported on Linux and Windows (Windows 10 and above).

3.2. Initialization

Depending upon the interface in use, the initialization steps are different.

3.2.1. DirectX 11 interface

Follow the steps listed below for initializing the DirectX 11 interface for Optical Flow.

1. Load the optical flow module `nvofapi.dll/nvofapi64.dll`.
2. Create an instance of ID3D11Device and ID3D11DeviceContext.
3. Retrieve the address of exported function `NvOFAPICreateInstanceD3D11` from the loaded optical flow module.
4. Populate API function pointer list.
   - Call `NvOFAPICreateInstanceD3D11` to populate NVOF API function pointer list.
   - The API accepts `NV_OF_API_VERSION` and a pointer `NV_OF_D3D11_API_FUNCTION_LIST*` to a memory block of function pointer list `NV_OF_D3D11_API_FUNCTION_LIST` that receives the NVOF APIs addresses.
   - The function pointers enable the access to the optical flow functionality.

5. Create optical flow instance
   - Call `NvCreateOpticalFlowD3D11` and pass the instance of `ID3D11Device` and `ID3D11DeviceContext` created in the previous step.
   - The API returns an `NvOFHandle` which should be preserved and used throughout the session.
6. Call `NvOFGetCaps` to query the capabilities supported by the NVIDIA display driver and the GPU.
7. Call `NvOFInit` after filling up `NV_OF_INIT_PARAMS`.

3.2.2. DirectX 12 interface

Follow the steps listed below for initializing the DirectX 12 interface for Optical Flow.

1. Load the optical flow module `nvofapi.dll/nvofapi64.dll`.
2. Create an instance of ID3D12Device.
3. Retrieve the address of exported function `NvOFAPICreateInstanceD3D12` from the loaded optical flow modue.
4. Populate API function pointer list.
   - Call `NvOFAPICreateInstanceD3D12` to populate API function pointer list.
   - The API accepts `NV_OF_API_VERSION` and a pointer `NV_OF_D3D12_API_FUNCTION_LIST*` to a memory block of function pointer list `NV_OF_D3D12_API_FUNCTION_LIST` that receives the API addresses.
   - The function pointers enable the access to the optical flow functionality.
5. Create optical flow instance
   - Call `NvCreateOpticalFlowD3D12` and pass the instance of `ID3D12Device` created in the previous step.
   - The API returns an `NvOFHandle` which should be preserved and used throughout the session.
6. Call `NvOFGetCaps` to query the capabilities supported by the NVIDIA display driver and the GPU.
7. Call `NvOFInit` after filling up `NV_OF_INIT_PARAMS`.

3.2.3. Cuda interface

Follow the steps listed below for initializing the CUDA interface.

1. Create a CUDA context.
2. Load optical flow module libnvidia-opticalflow.so.
3. Retrieve the address of exported function `NvOFAPICreateInstanceCuda` from the loaded optical flow module.
4. Populate API function pointer list.
   - Call `NvOFAPICreateInstanceCuda` to populate API function poitner list.

- The API accepts `NV_OF_API_VERSION` and a pointer `NV_OF_CUDA_API_FUNCTION_LIST*` to a memory block of function pointer list `NV_OF_CUDA_API_FUNCTION_LIST` that receives the API address.
- The function pointers enable the access to the optical flow functionality.

5. Call `NvCreateOpticalFlowCuda` using the CUDA context created in the earlier step.
   - The client must pass `NV_OF_API_VERSION` as the first parameter of `NvCreateOpticalFlowCuda`.
   - The API returns a `NvOFHandle` which should be preserved and used all throughout the session.
6. It is recommended to create CUDA streams and call `NvOFSetIOCudaStreams` for enabling internal preprocessing and post processing on the respective streams.
7. Call `NvOFGetCaps` to query the capabilities supported by the NVIDIA display driver and the GPU.
8. Call `NvOFInit` after filling up `NV_OF_INIT_PARAMS`.

3.2.4. Vulkan interface

Follow the steps listed below for initializing the Vulkan interface for Optical Flow.

1. Load the optical flow module `nvofapi.dll/nvofapi64.dll`.
2. Initialize Vulkan by creating a VkInstance with the `VK_KHR_get_physical_device_properties2` extension enabled, and ensure that VkApplicationInfo::apiVersion is set to VK_API_VERSION_1_3 or higher.
3. Select a physical device which supports `VK_KHR_timeline_semaphore` and `VK_NV_optical_flow` extensions.
4. Query the queue family which supports VK_QUEUE_OPTICAL_FLOW_BIT_NV.
5. Creating a logical device which enables
   - `VK_KHR_timeline_semaphore`, `VK_NV_optical_flow` extensions.
   - `VkPhysicalDeviceTimelineSemaphoreFeatures`, `VkPhysicalDeviceSynchronization2Features` and `VkPhysicalDeviceOpticalFlowFeaturesNV` features.
   - A device queue for optical flow family along with other devices queue(s) for graphics, compute and/or other tasks.
6. Retrieve the address of exported function `NvOFAPICreateInstanceVk` from the loaded optical flow module.
7. Populate API function pointer list.
   - Call `NvOFAPICreateInstanceVk` to populate Vulkan API function pointer list.
   - The API accepts `NV_OF_API_VERSION` and a pointer `NV_OF_VK_API_FUNCTION_LIST*` to a memory block of function pointer list `NV_OF_VK_API_FUNCTION_LIST` that receives the API addresses.
   - The function pointers enable the access to the optical flow functionality.
8. Create optical flow instance

- Call `nvCreateOpticalFlowVk` and pass the `VkInstance,VkPhysicalDevice,VkDevice` created in the previous step.
- The API returns an `NvOFHandle` which should be preserved and used throughout the session.

9. Call `NvOFGetCaps` to query the capabilities supported by the NVIDIA display driver and the GPU.
10. Call `NvOFInit` after filling up `NV_OF_INIT_PARAMS`.

After querying the capabilities, populate the following parameters of `NV_OF_INIT_PARAMS:`

- **Image height**: The height of the image/video sequence for which the vectors will be evaluated.
- **Image width**: The width of the image/video sequence for which the vectors will be evaluated.
- **Flag to enable external hints**: NVOF API provides clients the flexibility for feeding external flow vectors which will be used as hints by NVOFA while performing the motion search.
- **Output grid size**: Specify `NV_OF_INIT_PARAMS::outGridSize`. The flow vectors will be generated for the specified grid size.
- **Hint grid size**: In case the client is using external hints, the hint grid size should be specified.
- **Preset** – The NVOF API exposes three presets to trade quality vs performance.
- **Operating mode:** The NVOFA can be used for generating optical flow vectors (which comprise X and Y components and is referred to as optical flow mode) or stereo vectors (which comprise only the X component and is referred to as stereo mode).
- **Enable output cost:** Set this flag to generate confidence associated with the flow vectors. Higher cost value implies the flow vector to be less accurate and vice-versa. It is recommended to use the 8-bit cost (by allocating a buffer of type `NV_OF_BUFFER_FORMAT_UINT8`) instead of 32-bit cost as that saves CUDA bandwidth.
- **Disparity Range:** This is valid from Ampere and later GPUs and applicable in `NV_OF_MODE_STEREODISPARITY` mode. It specifies maximum stereo disparity range value.
- **Enable ROI:** Set the flag to estimate the flow for a rectangular region with in the image. Available on Ampere and later GPUs
- **Prediction direction:** Direction of the prediction. Supports either forward or forward-backward directions.
- **Enable Global Flow:** Global flow estimation is enabled when this flag is set.
- **Input buffer format:** `NV_OF_BUFFER_FORMAT` for the input buffer.

Chapter 4. Buffer Management

After the session creation and initialization, client needs to allocate buffers needed for generating the optical flow/stereo vectors.

4.1. Buffer allocation for DirectX 11 interface

Follow these steps for allocating buffers when DirectX 11 interface is used.

1. Call `NvOFGetSurfaceFormatCountD3D11`.
   - While making the call, the client should specify NV_OF_BUFFER_USAGE and NV_OF_MODE.
   - The function returns the count of DXGI_FORMATs supported by the interface.
2. Allocate an array of `DXGI_FORMAT`s. The size of the array should be at least equal to the count of `DXGI_FORMAT`s returned in `NvOFGetSurfaceFormatCountD3D11`.
3. Call `NvOFGetSurfaceFormatD3D11`.
   - While making the call, the client should specify NV_OF_BUFFER_USAGE, NV_OF_MODE and pointer to the memory that receives the supported `DXGI_FORMAT`s.
4. Allocate `ID3D11Resource` using `CreateResource` DirectX 11 API with required DXGI_FORMAT that is supported.
5. Each of the allocated buffers should be registered with NVOF API by calling `NvOFRegisterResourceD3D11`.
   - The API returns a handle to the registered resource through `NvOFGPUBufferHandle` which will be used for accessing the buffer.

4.2. Buffer allocation for DirectX 12 interface

Follow these steps for allocating buffers when DirectX 12 interface is used.

1. Call `NvOFGetSurfaceFormatCountD3D12`.

- While making the call, the client should specify NV_OF_BUFFER_USAGE and NV_OF_MODE.
- The function returns the count of DXGI_FORMATs supported by the interface.

2. Allocate an array of `DXGI_FORMAT`s. The size of the array should be at least equal to the count of `DXGI_FORMAT`s returned in `NvOFGetSurfaceFormatCountD3D12`.
3. Call `NvOFGetSurfaceFormatD3D12`.
   - While making the call, the client should specify NV_OF_BUFFER_USAGE, NV_OF_MODE and pointer to the memory that receives the supported `DXGI_FORMAT`s.
4. Allocate `ID3D12Resource` using `ID3D12Device::CreateCommittedResource` DirectX 12 API with required DXGI_FORMAT.
5. Each of the allocated buffers should be registered with NVOF API by calling `NvOFRegisterResourceD3D12`.
   - The API `NvOFRegisterResourceD3D12` accepts two `NV_OF_FENCE_POINT` objects (A fence point is a pair of `ID3D12Fence` pointer and a value), `NV_OF_REGISTER_RESOURCE_PARAMS_D3D12::inputFencePoint` and `NV_OF_REGISTER_RESOURCE_PARAMS_D3D12::outputFencePoint`. NVOFA waits until the `NV_OF_REGISTER_RESOURCE_PARAMS_D3D12::inputFencePoint::fence` reaches or exceeds the `NV_OF_REGISTER_RESOURCE_PARAMS_D3D12::inputFencePoint::value` before start of processing the `resource`. After processing of the `resource`, the `NV_OF_REGISTER_RESOURCE_PARAMS_D3D12::outputFencePoint::fence` is updated with the `NV_OF_REGISTER_RESOURCE_PARAMS_D3D12::outputFencePoint::value`.
   - The API returns a handle to the registered resource through `NvOFGPUBufferHandle` which will be used for accessing the buffer.

4.3. Buffer allocation for CUDA interface

Follow these steps for allocating buffers when CUDA interface is used.

1. Fill up the `NV_OF_BUFFER_DESCRIPTOR` with the dimensions of the surface to be allocated, type of buffer (enumerated in `NV_OF_BUFFER_USAGE`) and available formats (enumerated in `NV_OF_BUFFER_FORMAT`).
2. Specify `NV_OF_CUDA_BUFFER_TYPE`.
3. Call `NvOFCreateGPUBufferCuda`.

The API returns a handle to the allocated buffer through `NvOFGPUBufferHandle` which can be used for accessing the buffer.

4.4. Buffer allocation for Vulkan interface

Follow these steps for allocating buffers when vulkan interface is used.

1. Call `nvOFGetSurfaceFormatCountVk`.

- While making the call, the client should specify NV_OF_BUFFER_USAGE and NV_OF_MODE.
- The function returns the count of VkFormat supported by the interface.

2. Allocate an array of `VkFormats`. The size of the array should be at least equal to the count of `VkFormats` returned in `nvOFGetSurfaceFormatCountVk`.
3. Call `nvOFGetSurfaceFormatVk`.
   - While making the call, the client should specify NV_OF_BUFFER_USAGE, NV_OF_MODE and pointer to the memory that receives the supported `VkFormat`.
4. Allocate `VkImage` using `vkCreateImage` where Vulkan API with required VkFormat.
5. Use `vkCmdPipelineBarrier` Vulkan API for the image memory barrier pipelining.
6. Each of the allocated buffers should be registered with NVOF API by calling `nvOFRegisterResourceVk`.
   - The API `nvOFRegisterResourceVk` accepts vulkan image and its format.
   - After processing of the VkImage
   - The API returns a handle to the registered resource through `NvOFGPUBufferHandle` which will be used for accessing the buffer.

Chapter 5. Generating Flow Vectors

The process is same in DirectX 11 and CUDA interfaces. Due to explicit synchornization in DirectX 12, Vulkan the process is slightly different.

Depending on the client requirements and supported features, client can program additional fields to generate more data.

Flow vector is preresented by a 32-bit value with each horizontal and vertical compoment being 16-bit value. The lowest 5 bits holding fractional value, followed by a 10-bit integer value and the most significant bit being a sign bit.

5.1. DirectX 11 and CUDA

After the NVOF API is initialized and all buffers are allocated, client needs to follow the following steps to generate the flow vectors:

1. Fill up the following fields in `NV_OF_EXECUTE_INPUT_PARAMS`
   - Handle to the input frame.
   - Handle to reference frame.
   - Handle to external motion vector hints buffer (in case NV_OF_INIT_PARAMS::enableExternalHints is enabled).
   - Flag to invalidate past temporal hints.
2. Fill up the following fields in `NV_OF_EXECUTE_OUTPUT_PARAMS`
   - Handle to the output buffer where NVOFA will dump the flow vectors.
   - Handle to the cost buffer (in case NV_OF_INIT_PARAMS::enableOutputCost is enabled).
3. Call `NvOFExecute`.
   - This API must be called for every pair of frames/images.
4. The NVOFA will generate and store the flow vectors in `NV_OF_EXECUTE_OUTPUT_PARAMS::outputBuffer`.

5.2. DirectX 12

After the NVOF API is initialized and all buffers are allocated, client needs to follow the following steps to generate the flow vectors:

1. Fill up the following fields in `NV_OF_EXECUTE_INPUT_PARAMS_D3D12`
   - Handle to the input frame.
   - Handle to reference frame.
   - Handle to external motion vector hints buffer (in case NV_OF_INIT_PARAMS::enableExternalHints is enabled).
   - Flag to invalidate past temporal hints.
   - A pointer to an array of input fence points `NV_OF_FENCE_POINT`. Since there are multiple input buffers and the data can be rendered to these buffers by different engines, application can use these fence points to synchronize NVOFA with other engines.
2. Fill up the following fields in `NV_OF_EXECUTE_OUTPUT_PARAMS_D3D12`
   - Handle to the output buffer where NVOFA will dump the flow vectors.
   - Handle to the cost buffer (in case NV_OF_INIT_PARAMS::enableOutputCost is enabled).
   - A pointer to a `NV_OF_FENCE_POINT` which will be signaled when NVOFA is done with all processing.
3. Call `NvOFExecuteD3D12`.
   - This API must be called for every pair of frames/images.
4. The NVOFA will generate and store the flow vectors in `NV_OF_EXECUTE_OUTPUT_PARAMS::outputBuffer`.

5.3. Vulkan

After the NVOF API is initialized and all buffers are allocated, follow the following steps to generate the flow vectors:

1. Fill up the following fields in `NV_OF_EXECUTE_INPUT_PARAMS_VK`
   - Handle to the input frame.
   - Handle to reference frame.
   - Handle to external motion vector hints buffer (in case NV_OF_INIT_PARAMS::enableExternalHints is enabled).
   - Flag to invalidate past temporal hints.

- A pointer to an array of input semaphores `NV_OF_SYNC_VK`. Since there are multiple input buffers and the data can be rendered to these buffers by different engines, application can use these semaphores to synchronize NVOFA with other engines.

2. Fill up the following fields in `NV_OF_EXECUTE_OUTPUT_PARAMS_VK`
   - Handle to the output buffer where NVOFA will dump the flow vectors.
   - Handle to the cost buffer (in case NV_OF_INIT_PARAMS::enableOutputCost is enabled).
   - A pointer to a `NV_OF_SYNC_VK` which will be signaled when NVOFA is done with all processing.
3. Call `NvOFExecuteVk`.
   - This API must be called for every pair of frames/images.
4. The NVOFA will generate and store the flow vectors in `NV_OF_EXECUTE_OUTPUT_PARAMS::outputBuffer`.

Chapter 6. Releasing the Resources

After completion of the session, client needs to free all allocated resources as explained below.

6.1. DirectX 11 Mode

Follow the steps listed below for doing all necessary cleanup:

1. Unregister all the allocated resources by calling `NvOFUnregisterResourceD3D11`.
2. Free all allocated resources.
3. Call `NvOFDestroy`.
   - This NVOF API destroys the NVOF session and frees up all resources allocated internally by NVIDIA display driver.
4. Destroy the DirectX 11 device context.
5. Destroy the DirectX 11 device.

6.2. DirectX 12 Mode

Follow the steps listed below for doing all necessary cleanup:

1. Unregister all the allocated resources by calling `NvOFUnregisterResourceD3D12`.
2. Free all allocated resources.
3. Call `NvOFDestroy`.
   - This NVOF API destroys the NVOF session and frees up all resources allocated internally by NVIDIA display driver.
4. Destroy the DirectX 12 device.

6.3. CUDA Mode

Follow the steps listed below for doing all necessary clean up:

1. Deallocate all allocated buffers by calling `NvOFDestroyGPUBufferCuda`.
2. Call `NvOFDestroy`.

- This NVOF API destroys the NVOF session and frees up all resources allocated internally by NVIDIA display driver.

3. Destroy the CUDA streams if allocated.
4. Destroy the CUDA context.

6.4. Vulkan Mode

Follow the steps listed below for doing all necessary cleanup:

1. Unregister all the allocated resources by calling `NvOFUnregisterResourceVk`.
2. Free all allocated resources.
3. Call `NvOFDestroy`.
   - This NVOF API destroys the NVOF session and frees up all resources allocated internally by NVIDIA display driver.
4. Destroy the VkDevice, VkInstance

Chapter 7. Other Supported Features

The NVOF API supports the following additional features.

7.1. Query NVOF API Capabilities

Client should follow the steps below for querying various capabilities of NVOF API:

1. Call `NvOFGetCaps` specifying the capability to be queried in `NV_OF_CAPS`.
2. The NVOF API will fill up the supported value corresponding to the queried capability.

This NVOF API is supported for DirectX 11/12, Vulkan and CUDA modes. In order to make the application future-proof, it is strongly recommended that the clients check the capabilities returned by this API before using those capabilities.

7.2. Get last error encountered

Client can call `NvOFGetLastError` to query the last error encountered inside NVIDIA optical flow API/driver.

The NVOF API populates the output buffer with the description of the last error encountered.

This NVOF API is supported for both DirectX 11 and CUDA modes and could be helpful while troubleshooting.

7.3. Query maximum supported NVOF API version

Client application can use `NvOFGetMaxSupportedApiVersion` to retrieve the maximum NVOF API version supported by the underlying NVIDIA display driver.

Using this NVOF API a client application can support multiple versions of the NVIDIA driver and only use functionality supported by the specific NVOF API header version in the underlying NVIDIA display driver.

In other words, NvOFGetMaxSupportedApiVersion enables the clients to build applications which can work across different NVIDIA display driver versions supporting different NVOF API versions.

NvOFGetMaxSupportedApiVersion is supported for both DirectX 11/12, Vulkan and CUDA modes.

This NVOF API is supported in NVIDIA Optical Flow SDK 1.1 and above.

7.4. Region of Interest (ROI)

Starting GA100 GPU, NVOFA can generate flow vectors for a specified region within a frame which results to increased performance.

Follow the below steps for using the feature:

1. Query the capability by calling NvOFGetCaps.
2. If the feature is supported, then NV_OF_INIT_PARAMS::enableRoi should be set to 1.
3. Specify the number of ROI and ROI coordinates for which the flow vectors need to be evaluated in NV_OF_EXECUTE_INPUT_PARAMS::numRois and NV_OF_EXECUTE_INPUT_PARAMS::roiData respectively.

7.5. Forward and backward flow (FB flow)

When NV_OF_INIT_PARAMS::predDirection is set to NV_OF_PRED_DIRECTION_BOTH, forward and backward flow will be generated in a single NvOFExecute/NvOFExecuteD3D12/NvOFExecuteVk API call.

Along with setting NV_OF_INIT_PARAMS::predDirection, client also needs to set the NV_OF_EXECUTE_OUTPUT_PARAMS::bwdOutputBuffer/ NV_OF_EXECUTE_OUTPUT_PARAMS_D3D12::bwdOutputBuffer and NV_OF_EXECUTE_OUTPUT_PARAMS::bwdOutputCostBuffer/ NV_OF_EXECUTE_OUTPUT_PARAMS_D3D12::bwdOutputCostBuffer if NV_OF_INIT_PARAMS::enableOutputCost flag is set which receives the backward flow output and cost respectively.

Forward flow represents the movement of pixels from input frame to reference frame. Backward flow represents the movement of pixels from reference frame to input frame.

7.6. Global flow vector

When NV_OF_INIT_PARAMS::enableGlobalFlow is set to NV_OF_TRUE, a global flow vector is estimated from forward flow in the same NvOFExecute/NvOFExecuteD3D12/NvOFExecuteVk API call.

Along with setting NV_OF_INIT_PARAMS::enableGlobalFlow, client also needs to set the NV_OF_EXECUTE_OUTPUT_PARAMS::globalFlowBuffer/

`NV_OF_EXECUTE_OUTPUT_PARAMS_D3D12::globalFlowBuffer` which receives the global flow vector.

Chapter 8. Guidelines for Efficient Usage of NVOF API

1. Minimize the number of CUDA contexts created.
   - As far as possible, use a shared CUDA context across multiple NVOF sessions. This helps avoid the memory and initialization overhead associated with CUDA context creation. If it's not possible to use a single CUDA context, try to minimize the number of CUDA contexts created (e.g. use pooled contexts).
2. Create and use distinct CUDA streams for preprocessing and postprocessing.
   - The NVOF API internally does pre-processing and post-processing using CUDA. Hence, creating and using distinct CUDA streams for preprocessing and postprocessing results in better pipelining and improves the throughput. If distinct CUDA streams are not specified, the NVOF API will use the NULL stream for all internal preprocessing and postprocessing operations which may adversely impact the overall throughput.
3. Keep temporal hints enabled, which NVOF API enables by default.
   - NVOFA uses the flow vectors of earlier frame as hints for doing motion search, to take advantage of temporal correlation in a video sequence. Disable the temporal hints only if there is a-priori knowledge of no temporal correlation (e.g. a scene change, independent successive frame pairs).
4. Pass good quality flow vectors as hints.
   - NVOF API provides the option for passing external hints. The external hints are given highest importance while performing the motion search by the hardware. Hence it is recommended to pass good quality flow vectors as hints, and not use the feature in case the hints are not of good quality.
5. The client application should keep a pool of buffers for input and reference and use them in a round robin fashion.
   - NVOF API needs the input and reference frame for generating the flow vectors.
   - It is recommended for the client application to keep a pool of buffers for input and reference and use them in a round robin fashion. This ensures better pipelining by avoiding resource hazards at the cost of slightly increased memory footprint. For example, for first `NvOFExecute`, if buffer 1 is used as reference and buffer 2 is used as input, the second `NvOFExecute` should use buffer 3 as input and buffer 4 as reference. This avoids performance impact from synchronization overhead of reusing buffers. Generally, a buffer pool of 4-6 input buffers should suffice to saturate the hardware.

Lower resolutions (say, below 480p) may require larger number of buffers to achieve high efficiency.

6. Minimize PCIe transfers of raw image buffers (RGB/YUV) to improve performance.
   - If the input frames are available as raw YUV/RGB frames in system memory, it will be beneficial to preload as many frames in the video memory as possible. Typically, PCIe transfers of raw images requires large bandwidth and becomes a bottleneck in achieving high performance with optical flow. If preloading of the frames in video memory is not an option, try to pipeline the loading of the next batch of frames into the video memory while the optical flow engine is computing the flow vectors on the current set of frames.
   - Another option is to consider using H.264 or HEVC-encoded bitstreams for the frames and use GPU's NVDEC engines to decode the frames just in time before sending to optical flow computation.
7. Use SLOW preset in scenarios only where enough graphics engine bandwidth is available.
   - Using the SLOW preset may result in increase in graphics/CUDA engine utilization.
8. Enable output cost to get the confidence on the flow vectors.
   - Higher cost value implies the flow vector to be less accurate and vice-versa.
9. NVOF APIs are not thread safe.
   - If a NVOF API context is shared among multiple threads in a client application, then application needs to use appropriate synchronization mechanism (such as critical sections) to synchronize access to the shared NVOF API context.
10. If `NV_OF_MODE::NV_OF_MODE_STEREODISPARITY` is specified, then the input image pair should be rectified i.e. vertical displacement across the input frame pair should be zero.
11. When passing input frames to the NVOF API, no padding should be added to the frames.
12. `NvOFGetMaxSupportedApiVersion` can be used to develop applications which can work across NVIDIA display driver versions supporting different NVOF API versions.
13. It is recommended to use the 8-bit cost (by allocating a buffer of type `NV_OF_BUFFER_FORMAT_UINT8)` instead of 32-bit cost as that saves CUDA bandwidth. Support for 32-bit cost will be deprecated in future.

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