nvJPEG

nvJPEG Libary Guide

DA-06762-001_v11.2 | March 2021

Table of Contents

Chapter 1. Introduction

1.1. nvJPEG Decoder

The nvJPEG library provides high-performance, GPU accelerated JPEG decoding functionality for image formats commonly used in deep learning and hyperscale multimedia applications. The library offers single and batched JPEG decoding capabilities which efficiently utilize the available GPU resources for optimum performance; and the flexibility for users to manage the memory allocation needed for decoding.

The nvJPEG library enables the following functions: use the JPEG image data stream as input; retrieve the width and height of the image from the data stream, and use this retrieved information to manage the GPU memory allocation and the decoding. A dedicated API is provided for retrieving the image information from the raw JPEG image data stream.

Tip: Throughout this document, the terms "CPU" and "Host" are used synonymously. Similarly, the terms "GPU" and "Device" are synonymous.

The nvJPEG library supports the following:

JPEG options:

- Baseline and Progressive JPEG decoding/encoding
- 8 bits per pixel
- Huffman bitstream decoding
- Upto 4 channel JPEG bitstreams
- 8- and 16-bit quantization tables
- The following chroma subsampling for the 3 color channels Y, Cb, Cr (Y, U, V):
  - 4:4:4
  - 4:2:2
  - 4:2:0
  - 4:4:0
  - 4:1:1
  - 4:1:0

Features:

- Hybrid decoding using both the CPU (i.e., host) and the GPU (i.e., device).
- Hardware acceleration for baseline JPEG decode on supported platforms.
- Input to the library is in the host memory, and the output is in the GPU memory.
- Single image and batched image decoding.
- Single phase and multiple phases decoding.
- Color space conversion.
- User-provided memory manager for the device and pinned host memory allocations.

1.2. nvJPEG Encoder

The encoding functions of the nvJPEG library perform GPU-accelerated compression of user's image data to the JPEG bitstream. User can provide input data in a number of formats and colorspaces, and control the encoding process with parameters. Encoding functionality will allocate temporary buffers using user-provided memory allocator.

Before calling the encoding functions the user should perform a few prerequisite steps using the helper functions described in nvJPEG Encoder Helper API Reference.

1.3. Thread Safety

Not all nvJPEG types are thread safe.

When using decoder APIs across multiple threads, the following decoder types should be instantiated separately for each thread: nvjpegJpegStream_t, nvjpegJpegState_t, nvjpegBufferDevice_t, nvjpegBufferPinned_t

When using encoder APIs across multiple threads, nvjpegEncoderState_t should be instantiated separately for each thread.

For user provided allocators (inputs to nvJPEGCreateEx()), user needs to ensure thread safety.

1.4. Hardware Accelaration

Starting with CUDA 11.0, hardware accelerated JPEG decode is available on GA100.

Platforms which support hardware accelarated JPEG decode:

- Windows
- Linux (x86_64, PowerPC, ARM64)

Chapter 2. JPEG Decoding

2.1. Using JPEG Decoding

The nvJPEG library provides functions for both the decoding of a single image, and batched decoding of multiple images.

2.1.1. Single Image Decoding

For single-image decoding you provide the data size and a pointer to the file data, and the decoded image is placed in the output buffer.

To use the nvJPEG library, start by calling the helper functions for initialization.

1. Create nvJPEG library handle with one of the helper functions `nvjpegCreateSimple()` or `nvjpegCreateEx()`.
2. Create JPEG state with the helper function `nvjpegJpegStateCreate()`. See nvJPEG Type Declarations and `nvjpegJpegStateCreate()`.

   Below is the list of helper functions available in the nvJPEG library:

   - `nvjpegStatus_t nvjpegGetProperty(libraryPropertyType type, int *value);`
   - `[DEPRECATED] nvjpegStatus_t nvjpegCreate(nvjpegBackend_t backend, nvjpegHandle_t *handle , nvjpeg_dev_allocator allocator);`
   - `nvjpegStatus_t nvjpegCreateSimple(nvjpegHandle_t *handle);`
   - `nvjpegStatus_t nvjpegCreateEx(nvjpegBackend_t backend, nvjpegDevAllocator_t *dev_allocator, nvjpegPinnedAllocator_t *pinned_allocator, unsigned int flags, nvjpegHandle_t *handle);`
   - `nvjpegStatus_t nvjpegDestroy(nvjpegHandle_t handle);`
   - `nvjpegStatus_t nvjpegJpegStateCreate(nvjpegHandle_t handle, nvjpegJpegState_t *jpeg_handle);`
   - `nvjpegStatus_t nvjpegJpegStateDestroy(nvjpegJpegState handle);`
   - Other helper functions such as `nvjpegSet*()` and `nvjpegGet*()` can be used to configure the library functionality on per-handle basis. Refer to the helper API reference for more details.

3. Retrieve the width and height information from the JPEG-encoded image by using the nvjpegGetImageInfo() function.

   Below is the signature of nvjpegGetImageInfo() function:

```
nvjpegStatus_t nvjpegGetImageInfo(
  nvjpegHandle_t              handle,
  const unsigned char         *data,
  size_t                      length,
  int                         *nComponents,
  nvjpegChromaSubsampling_t   *subsampling,
  int                         *widths,
  int                         *heights);
```

   For each image to be decoded, pass the JPEG data pointer and data length to the above function. The nvjpegGetImageInfo() function is thread safe.

4. One of the outputs of the above nvjpegGetImageInfo() function is nvjpegChromaSubsampling_t. This parameter is an enum type, and its enumerator list is composed of the chroma subsampling property retrieved from the JPEG image. See nvJPEG Chroma Subsampling.

5. Use the nvjpegDecode() function in the nvJPEG library to decode this single JPEG image. See the signature of this function below:

```
nvjpegStatus_t nvjpegDecode(
  nvjpegHandle_t          handle,
  nvjpegJpegState_t       jpeg_handle,
  const unsigned char     *data,
  size_t                  length,
  nvjpegOutputFormat_t    output_format,
  nvjpegImage_t           *destination,
  cudaStream_t            stream);
```

   In the above nvjpegDecode() function, the parameters nvjpegOutputFormat_t, nvjpegImage_t, and cudaStream_t can be used to set the output behavior of the nvjpegDecode() function. You provide the cudaStream_t parameter to indicate the stream to which your asynchronous tasks are submitted.

6. **The** nvjpegOutputFormat_t **parameter:**

   The nvjpegOutputFormat_t parameter can be set to one of the output_format settings below:

output_format	Meaning
NVJPEG_OUTPUT_UNCHANGED	Return the decoded image planar format.
NVJPEG_OUTPUT_RGB	Convert to planar RGB.
NVJPEG_OUTPUT_BGR	Convert to planar BGR.
NVJPEG_OUTPUT_RGBI	Convert to interleaved RGB.
NVJPEG_OUTPUT_BGRI	Convert to interleaved BGR.
NVJPEG_OUTPUT_Y	Return the Y component only.
NVJPEG_OUTPUT_YUV	Return in the YUV planar format.

   For example, if the output_format is set to NVJPEG_OUTPUT_Y or NVJPEG_OUTPUT_RGBI, or NVJPEG_OUTPUT_BGRI then the output is written only to channel[0], and the other channels are not touched.

Alternately, in the case of planar output, the data is written to the corresponding channels of the `nvjpegImage_t` destination structure.

Finally, in the case of grayscale JPEG and RGB output, the luminance is used to create the grayscale RGB.

The below table explains the combinations of the output formats and the number of channels supported by the library.

No of Channels in bitstream / **Output Format**	1	2	3	4
NVJPEG_OUTPUT_UNCHANGED	Yes	Yes	Yes	Yes
NVJPEG_OUTPUT_YUV	Only the first channel of the output is populated	No	Yes	No
NVJPEG_OUTPUT_Y	Yes	No	Yes	Yes(a)
NVJPEG_OUTPUT_RGB	Yes(b)	No	Yes	Yes(a)
NVJPEG_OUTPUT_BGR	Yes(b)	No	Yes	Yes(a)
NVJPEG_OUTPUT_RGBI	Yes(b)	No	Yes	Yes(a)
NVJPEG_OUTPUT_BGRI	Yes(b)	No	Yes	Yes(a)

NOTES:

a). (a) Must be enabled using nvjpegDecodeParamsSetAllowCMYK().

b). (b) Luminance is used to create the grayscale RGB.

7. As mentioned above, an important benefit of the `nvjpegGetImageInfo()` function is the ability to utilize the image information retrieved from the the input JPEG image to allocate proper GPU memory for your decoding operation.

   The `nvjpegGetImageInfo()` function returns the `widths`, `heights` and `nComponents` parameters.

```
nvjpegStatus_t nvjpegGetImageInfo(
  nvjpegHandle_t              handle,
  const unsigned char         *data,
  size_t                      length,
  int                         *nComponents,
  nvjpegChromaSubsampling_t   *subsampling,
  int                         *widths,
  int                         *heights);
```

   You can use the retrieved parameters, `widths`, `heights` and `nComponents`, to calculate the required size for the output buffers, either for a single decoded JPEG, or for every decoded JPEG in a batch.

   To optimally set the `destination` parameter for the `nvjpegDecode()` function, use the following guidelines:

For the output_format:	**destination.pitch[0] should be at least:** width[0]	**destination.channel[0] should be at least of size:** destination.pitch[0]*height[0]
NVJPEG_OUTPUT_Y		

For the output_format	destination.pitch[c] should be at least:	destination.channel[c] should be at least of size:
NVJPEG_OUTPUT_YUV	width[c] for c = 0, 1, 2	destination.pitch[c]*height[c] for c = 0, 1, 2
NVJPEG_OUTPUT_RGB and NVJPEG_OUTPUT_BGR	width[0] for c = 0, 1, 2	destination.pitch[0]*height[0] for c = 0, 1, 2
NVJPEG_OUTPUT_RGBI and NVJPEG_OUTPUT_BGRI	width[0]*3	destination.pitch[0]*height[0]
NVJPEG_OUTPUT_UNCHANGED	width[c] for c = [0, nComponents - 1]	destination.pitch[c]*height[c] for c = [0, nComponents - 1]

8. Ensure that the nvjpegImage_t structure (or structures, in the case of batched decode) is filled with the pointers and pitches of allocated buffers. The `nvjpegImage_t` structure that holds the output pointers is defined as follows:

```
typedef struct
{
    unsigned char * channel[NVJPEG_MAX_COMPONENT];
    size_t pitch[NVJPEG_MAX_COMPONENT];
} nvjpegImage_t;
```

   NVJPEG_MAX_COMPONENT is the maximum number of color components the nvJPEG library supports in the current release. For generic images, this is the maximum number of encoded channels that the library is able to decompress.

9. Finally, when you call the `nvjpegDecode()` function with the parameters as described above, the `nvjpegDecode()` function fills the output buffers with the decoded data.

2.1.2. Decode using Decoupled Phases

The nvJPEG library allows further separation of the host and device phases of the decode process. The host phase of the decoding will not need to access to device resources.

A few examples of decoupled APIs can be found under Decode API - Decoupled Multiple Phases.

Below is the sequence of API calls to decode a single image

1. Initialize all the items that are used in the decoding process:
   a). Create the library handle using one of the library handle initialization routines.
   b). Choose decoder implementation nvjpegBackend_t, and create decoder using nvjpegDecoderCreate().
   c). Create JPEG decoder state using nvjpegDecoderStateCreate().
   d). Create JPEG stream using nvjpegJpegStreamCreate().
   e). Create the pinned and device buffers used by the decoder using the below APIs respectively. These buffers are used to store intermediate decoding results.
      - nvjpegBufferPinnedCreate()
      - nvjpegBufferDeviceCreate()
   f). Link the buffers to the JPEG state using the below APIs respectively:
      - nvjpegStateAttachPinnedBuffer()

- nvjpegStateAttachDeviceBuffer()

g). Create decode parameters using the below API. This is used to set the output format, and enable ROI decode:

nvjpegDecodeParamsCreate()

2. Perform decoding:

a). Parse the jpeg bit-stream using nvjpegJpegStreamParse()

- Encoded bitstream information, like channel dimensions, can be retrieved using the below API. This information is used to allocate the output pointers in nvjpegImage_t.
  - nvjpegJpegStreamGetComponentsNum()
  - nvjpegJpegStreamGetComponentDimensions()

b). Call the decode API in the below sequence to decode the image:

- nvjpegDecodeJpegHost()
- nvjpegDecodeJpegTransferToDevice()
- nvjpegDecodeJpegDevice()

2.1.3. Batched Image Decoding

For the batched image decoding you provide pointers to multiple file data in the memory, and also provide the buffer sizes for each file data. The nvJPEG library will decode these multiple images, and will place the decoded data in the output buffers that you specified in the parameters.

2.1.3.1. Single Phase

For batched image decoding in single phase, follow these steps:

1. Call `nvjpegDecodeBatchedInitialize()` function to initialize the batched decoder. Specify the batch size in the `batch_size` parameter. See `nvjpegDecodeBatchedInitialize()`.
2. Next, call `nvjpegDecodeBatched()` for each new batch. Make sure to pass the parameters that are correct to the specific batch of images. If the size of the batch changes, or if the batch decoding fails, then call the `nvjpegDecodeBatchedInitialize()` function again.

2.2. nvJPEG Type Declarations

2.2.1. nvJPEG Backend

```
typedef enum {
 NVJPEG_BACKEND_DEFAULT = 0,
 NVJPEG_BACKEND_HYBRID = 1,
```

```
 NVJPEG_BACKEND_GPU_HYBRID = 2,
 NVJPEG_BACKEND_HARDWARE = 3
} nvjpegBackend_t;
```

The `nvjpegBackend_t` enum is used to select either default back-end by default, or use GPU decoding for baseline JPEG images, or use CPU for Huffman decoding.

Member	Description
NVJPEG_BACKEND_DEFAULT	Back-end is selected internally
NVJPEG_BACKEND_HYBRID	Uses CPU for Huffman decoding
NVJPEG_BACKEND_GPU_HYBRID	Uses GPU for Huffman decoding. `nvjpegDecodeBatched` will use GPU decoding for baseline Jpeg images with interleaved scan when batch size is greater than 100. The decoupled APIs will use GPU assisted Huffman decoding.
NVJPEG_BACKEND_HARDWARE	Uses Hardware Acceleration for decode. Supports baseline JPEG images with single scan with 1 or 3 channels. 410 and 411 chroma subsamplings are not supported.

2.2.2. nvJPEG Bitstream Handle

```
struct nvjpegJpegStream;
typedef struct nvjpegJpegStream* nvjpegJpegStream_t;
```

This handle stores the bit-stream parameters on the host. This helps retrieve bitstream meta-data using APIs defined in nvJPEG Stream API.

2.2.3. nvJPEG Decode Device Buffer Handle

```
struct nvjpegBufferDevice;
typedef struct nvjpegBufferDevice* nvjpegBufferDevice_t;
```

This nvjpegBufferDevice_t is used by decoder states to store the intermediate information in device memory.

2.2.4. nvJPEG Decode Parameter Handle

```
struct nvjpegDecodeParams;
typedef struct nvjpegDecodeParams* nvjpegDecodeParams_t;
```

This decoder parameter handle stores the parameters like output format, and the ROI decode parameters that are set using APIs defined in nvJPEG Chroma Subsampling.

2.2.5. nvJPEG Decode Pinned Buffer Handle

```
struct nvjpegBufferPinned;
typedef struct nvjpegBufferPinned* nvjpegBufferPinned_t;
```

This nvjpegBufferPinned_t handle is used by decoder states to store the intermediate information on pinned memory.

2.2.6. nvJPEG Decoder Handle

```
struct nvjpegJpegDecoder;
typedef struct nvjpegJpegDecoder* nvjpegJpegDecoder_t;
```

This decoder handle stores the intermediate decoder data, which is shared across the decoding stages. This decoder handle is initialized for a given nvjpegBackend_t. It is used as input to the Decode API—Decoupled Decoding.

2.2.7. nvJPEG Host Pinned Memory Allocator Interface

```
typedef int (*tPinnedMalloc)(void**, size_t, unsigned int flags);
typedef int (*tPinnedFree)(void*);
typedef struct
{
tPinnedMalloc pinned_malloc;
tPinnedFree pinned_free;
} nvjpegPinnedAllocator_t;
```

When the `nvjpegPinnedAllocator_t *allocator` parameter in the `nvjpegCreateEx()` function is set as a pointer to the above `nvjpegPinnedAllocator_t` structure, then this structure will be used for allocating and releasing host pinned memory for copying data to/from device. The function prototypes for the memory allocation and memory freeing functions are similar to the `cudaHostAlloc()` and `cudaFreeHost()` functions. They will return 0 in case of success, and non-zero otherwise.

However, if the `nvjpegPinnedAllocator_t *allocator` parameter in the `nvjpegCreateEx()` function is set to NULL, then the default memory allocation functions `cudaHostAlloc()` and `cudaFreeHost()` will be used. When using `nvjpegCreate()` or `nvjpegCreateSimple()` function to create library handle, the default host pinned memory allocator will be used.

2.2.8. nvJPEG Image

```
typedef struct
 {unsigned char * channel[NVJPEG_MAX_COMPONENT];
 size_t pitch[NVJPEG_MAX_COMPONENT];
 } nvjpegImage_t;
```

The `nvjpegImage_t` structure (or structures, in the case of batched decode) is used to fill with the pointers and pitches of allocated buffers. The `nvjpegImage_t` structure that holds the output pointers.

Member	Description
NVJPEG_MAX_COMPONENT	Maximum number of color components the nvJPEG library supports. For generic images, this is the maximum number of encoded channels that the library is able to decompress.

2.2.9. nvJPEG Device Memory Allocator Interface

```
typedef int (*tDevMalloc)(void**, size_t);
```

```
typedef int (*tDevFree)(void*);
typedef struct
  {
    tDevMalloc dev_malloc;
    tDevFree dev_free;
  } nvjpegDevAllocator_t;
```

Users can tell the library to use their own device memory allocator. The function prototypes for the memory allocation and memory freeing functions are similar to the `cudaMalloc()` and `cudaFree()` functions. They should return 0 in case of success, and non-zero otherwise. A pointer to the `nvjpegDevAllocator_t` structure, with properly filled fields, should be provided to the `nvjpegCreate()` function. NULL is accepted, in which case the default memory allocation functions `cudaMalloc()` and `cudaFree()` is used.

When the `nvjpegDevAllocator_t *allocator` parameter in the `nvjpegCreate()` or `nvjpegCreateEx()` function is set as a pointer to the above `nvjpegDevAllocator_t` structure, then this structure is used for allocating and releasing the device memory. The function prototypes for the memory allocation and memory freeing functions are similar to the `cudaMalloc()` and `cudaFree()` functions. They should return 0 in case of success, and non-zero otherwise.

However, if the `nvjpegDevAllocator_t *allocator` parameter in the `nvjpegCreate()` or `nvjpegCreateEx()` function is set to NULL, then the default memory allocation functions `cudaMalloc()` and `cudaFree()` will be used. When using `nvjpegCreateSimple()` function to create library handle the default device memory allocator will be used.

2.2.10. nvJPEG Opaque JPEG Decoding State Handle

```
struct nvjpegJpegState;
typedef struct nvjpegJpegState* nvjpegJpegState_t;
```

The `nvjpegJpegState` structure stores the temporary JPEG information. It should be initialized before any usage. This JPEG state handle can be reused after being used in another decoding. The same JPEG handle should be used across the decoding phases for the same image or batch. Multiple threads are allowed to share the JPEG state handle only when processing same batch during first phase (`nvjpegDecodePhaseOne`) .

2.2.11. nvJPEG Opaque Library Handle Struct

```
struct nvjpegHandle;
typedef struct nvjpegHandle* nvjpegHandle_t;
```

The library handle is used in any consecutive nvJPEG library calls, and should be initialized first.

The library handle is thread safe, and can be used by multiple threads simultaneously.

2.2.12. nvJPEG Output Pointer Struct

```
typedef struct
  {
    unsigned char * channel[NVJPEG_MAX_COMPONENT];
    size_t pitch[NVJPEG_MAX_COMPONENT];
  } nvjpegImage_t;
```

The `nvjpegImage_t` struct holds the pointers to the output buffers, and holds the corresponding strides of those buffers for the image decoding.

See Single Image Decoding on how to set up the `nvjpegImage_t` struct.

2.2.13. nvJPEG Jpeg Encoding

```
typedef enum{
 NVJPEG_ENCODING_UNKNOWN                          = 0x0,
 NVJPEG_ENCODING_BASELINE_DCT                     = 0xc0,
 NVJPEG_ENCODING_EXTENDED_SEQUENTIAL_DCT_HUFFMAN = 0xc1,
 NVJPEG_ENCODING_PROGRESSIVE_DCT_HUFFMAN          = 0xc2
} nvjpegJpegEncoding_t;
```

The `nvjpegJpegEncoding_t` enum lists the JPEG encoding types that are supported by the nvjpeg library The enum values are based on the markers defined in the JPEG specification

Member	Description
NVJPEG_ENCODING_UNKNOWN	This value is returned for all the JPEG markers not supported by the nvjpeg library
NVJPEG_ENCODING_BASELINE_DCT	Corresponds to the JPEG marker 0xc0, refer to the JPEG spec for more details
NVJPEG_ENCODING_EXTENDED_SEQUENTIAL_DCT_HUFFMAN	Corresponds to the JPEG marker 0xc1, refer to the JPEG spec for more details
NVJPEG_ENCODING_PROGRESSIVE_DCT_HUFFMAN	Corresponds to the JPEG marker 0xc2, refer to the JPEG spec for more details

2.2.14. nvJPEG Scale Factor

```
typedef enum{
 NVJPEG_SCALE_NONE = 0,
 NVJPEG_SCALE_1_BY_2 = 1,
 NVJPEG_SCALE_1_BY_4 = 2,
 NVJPEG_SCALE_1_BY_8 = 3
} nvjpegScaleFactor_t;
```

The `nvjpegScaleFactor_t` enum lists all the scale factors supported by the library. This feature is supported when nvjpeg handles are intstaniated using NVJPEG_BACKEND_HARDWARE

Member	Description
NVJPEG_SCALE_NONE	Decoded output is not scaled
NVJPEG_SCALE_1_BY_2	Decoded output width and height are scaled by a factor of 1/2
NVJPEG_SCALE_1_BY_4	Decoded output width and height are scaled by a factor of 1/4
NVJPEG_SCALE_1_BY_8	Decoded output width and height are scaled by a factor of 1/8

2.2.15. nvJPEG Flags

```
#define NVJPEG_FLAGS_DEFAULT 0
#define NVJPEG_FLAGS_HW_DECODE_NO_PIPELINE 1
#define NVJPEG_FLAGS_ENABLE_MEMORY_POOLS   2
#define NVJPEG_FLAGS_BITSTREAM_STRICT      4
```

nvJPEG flags provide additional controls when initializing the library using nvJPEGCreateEx(). It is possible to combine the flags as they are bit fields.

Member	Description
NVJPEG_FLAGS_DEFAULT	Corresponds to default library behavior
NVJPEG_FLAGS_HW_DECODE_NO_PIPELINE	To be used when the library is initialized with NVJPEG_BACKEND_HARDWARE. It will be ignored for other back-ends. nvjpeg in batched decode mode buffers additional images to achieve optimal performance. Use this flag to disable buffering of additional images.
NVJPEG_FLAGS_ENABLE_MEMORY_POOLS [Deprecated]	Starting with CUDA 11.1 this flag will be ignored
NVJPEG_FLAGS_BITSTREAM_STRICT	nvJPEG library will try to decode a bitstream even if it doesn't strictly follow the JPEG specification. Using this flag will return an error in such cases.

2.3. nvJPEG API Reference

This section describes the nvJPEG decoder API.

2.3.1. nvJPEG Helper API Reference

2.3.1.1. nvjpegGetProperty()

Gets the numeric value for the major or minor version, or the patch level, of the nvJPEG library.

Signature:

```
nvjpegStatus_t nvjpegGetProperty(
 libraryPropertyType   type,
 int                   *value);
```

Parameters:

Parameter	Input / Output	Memory	Description
`libraryPropertyType type`	Input	Host	One of the supported `libraryPropertyType` values, that is, MAJOR_VERSION, MINOR_VERSION or PATCH_LEVEL.

`int *value`	Output	Host	The numeric value corresponding to the specific `libraryPropertyType` requested.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.2. nvjpegGetCudartProperty()

Gets the numeric value for the major version, minor version, or the patch level of the CUDA toolkit that was used to build nvJPEG library. For the same information on the nvJPEG library itself, see nvjpegGetProperty().

Signature:

```
nvjpegStatus_t nvjpegGetCudartProperty(
 libraryPropertyType   type,
 int                   *value);
```

Parameters:

Parameter	Input / Output	Memory	Description
`libraryPropertyType type`	Input	Host	One of the supported `libraryPropertyType` values, that is, MAJOR_VERSION, MINOR_VERSION or PATCH_LEVEL.
`int *value`	Output	Host	The numeric value corresponding to the specific `libraryPropertyType` requested.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.3. nvjpegCreate() [DEPRECATED]

Allocates and initializes the library handle.

Note: This function is deprecated. Use either `nvjpegCreateSimple()` or `nvjpegCreateEx()` functions to create the library handle.

Signature:

```
nvjpegStatus_t nvjpegCreate(
 nvjpegBackend_t   backend,
 nvjpegDevAllocator_t   *allocator,
 nvjpegHandle_t   *handle);
```

Parameters:

Parameter	Input / Output	Memory	Description

`nvjpegBackend_t backend`	Input	Host	Backend parameter for nvjpegDecodeBatched() API. If this is set to DEFAULT then it automatically chooses one of the underlying algorithms.
`nvjpegDevAllocator_t *allocator`	Input	Host	Device memory allocator. See `nvjpegDevAllocator_t` structure description. If NULL is provided, then the default CUDA runtime `cudaMalloc()` and `cudaFree()` functions will be used.
`nvjpegHandle_t *handle`	Input/Output	Host	The library handle.

The `nvjpegBackend_t` parameter is an `enum` type, with the below enumerated list values:

```
typedef enum {
 NVJPEG_BACKEND_DEFAULT = 0,
 NVJPEG_BACKEND_HYBRID = 1,
} nvjpegBackend_t;
```

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.4. nvjpegCreateSimple()

Allocates and initializes the library handle, with default codec implementations selected by library and default memory allocators.

Signature:

```
nvjpegStatus_t nvjpegCreateSimple(nvjpegHandle_t *handle);
```

Parameters:

Parameter	Input / Output	Memory	Description
`nvjpegHandle_t *handle`	Input/Output	Host	The library handle.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.5. nvjpegCreateEx()

Allocates and initializes the library handle using the provided arguments.

Signature:

```
nvjpegStatus_t nvjpegCreateEx(nvjpegBackend_t backend,
          nvjpegDevAllocator_t *dev_allocator,
          nvjpegPinnedAllocator_t *pinned_allocator,
          unsigned int flags,
          nvjpegHandle_t *handle);
```

Parameters:

Parameter	Input / Output	Memory	Description
`nvjpegBackend_t backend`	Input	Host	Backend parameter for nvjpegDecodeBatched() API. If this is set to DEFAULT then it automatically chooses one of the underlying algorithms.
`nvjpegDevAllocator_t *dev_allocator`	Input	Host	Device memory allocator. See `nvjpegDevAllocator_t` structure description. If NULL is provided, then the default CUDA runtime functions `cudaMalloc()` and `cudaFree()` will be used.
`nvjpegPinnedAllocator_t *pinned_allocator`	Input	Host	Pinned host memory allocator. See `nvjpegPinnedAllocator_t` structure description. If NULL is provided, then the default CUDA runtime functions `cudaHostAlloc()` and `cudaFreeHost()` will be used.
`unsigned int flags`	Input	Host	Refer to nvJPEG Flags for details
`nvjpegHandle_t *handle`	Input/Output	Host	The library handle.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.6. nvjpegDestroy()

Releases the library handle.

Signature:

```
nvjpegStatus_t nvjpegDestroy(nvjpegHandle_t handle);
```

Parameters:

Parameter	Input / Output	Memory	Description
`nvjpegHandle_t handle`	Input/Output	Host	The library handle to release.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.7. nvjpegSetDeviceMemoryPadding()

Use the provided padding for all device memory allocations with specified library handle. A large number will help to amortize the need for device memory reallocations when needed.

Signature:

```
nvjpegStatus_t nvjpegSetDeviceMemoryPadding(
 size_t padding,
 nvjpegHandle_t handle);
```

Parameters:

Parameter	Input / Output	Memory	Description
`size_t padding`	Input	Host	Device memory padding to use for all further device memory allocations.
`nvjpegHandle_t *handle`	Input/Output	Host	The library handle.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.8. nvjpegGetDeviceMemoryPadding()

Retrieve the device memory padding that is currently used for the specified library handle.

Signature:

```
nvjpegStatus_t nvjpegGetDeviceMemoryPadding(
 size_t *padding,
 nvjpegHandle_t handle);
```

Parameters:

Parameter	Input / Output	Memory	Description
`size_t *padding`	Output	Host	Device memory padding that is currently used for device memory allocations.
`nvjpegHandle_t *handle`	Input/Output	Host	The library handle.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.9. nvjpegSetPinnedMemoryPadding()

Use the provided padding for all pinned host memory allocations with specified library handle. A large number will help to amortize the need for pinned host memory reallocations when needed.

Signature:

```
nvjpegStatus_t nvjpegSetPinnedMemoryPadding(
 size_t padding,
 nvjpegHandle_t handle);
```

Parameters:

Parameter	Input / Output	Memory	Description
`size_t padding`	Input	Host	Pinned host memory padding to use for all further pinned host memory allocations.
`nvjpegHandle_t handle`	Input/Output	Host	The library handle.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.10. nvjpegGetPinnedMemoryPadding()

Retrieve the pinned host memory padding that is currently used for specified library handle.

Signature:

```
nvjpegStatus_t nvjpegGetPinnedMemoryPadding(
 size_t *padding,
 nvjpegHandle_t handle);
```

Parameters:

Parameter	Input / Output	Memory	Description
`size_t *padding`	Output	Host	Pinned host memory padding that is currently used for pinned host memory allocations.
`nvjpegHandle_t handle`	Input/Output	Host	The library handle.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.11. nvjpegJpegStateCreate()

Allocates and initializes the internal structure required for the JPEG processing.

Signature:

```
nvjpegStatus_t nvjpegJpegStateCreate(
 nvjpegHandle_t       handle,
 nvjpegJpegState_t    *jpeg_handle);
```

Parameters:

Parameter	Input / Output	Memory	Description
`nvjpegHandle_t handle`	Input	Host	The library handle.
`nvjpegJpegState_t *jpeg_handle`	Input/Output	Host	The image state handle.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.12. nvjpegJpegStateDestroy()

Releases the image internal structure.

Signature:

```
nvjpegStatus_t nvjpegJpegStateDestroy(nvjpegJpegState handle);
```

Parameters:

Parameter	Input / Output	Memory	Description
`nvjpegJpegState handle`	Input/Output	Host	The image state handle.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.13. nvjpegDecoderCreate()

Creates a decoder handle.

Signature:

```
nvjpegStatus_t nvjpegDecoderCreate(
 nvjpegHandle_t nvjpeg_handle,
 nvjpegBackend_t implementation,
 nvjpegJpegDecoder_t* decoder_handle);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegHandle_t nvjpeg_handle	Input	Host	Library handle.
nvjpegBackend_t backend	Input	Host	Backend parameter for the decoder_handle.The back end applies to all the functions under the decoupled API, when called with this handle.
nvjpegJpegDecoder_t decoder_handle	Input/Output	Host	Decoder state handle.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.14. nvjpegDecoderDestroy()

Destroys the decoder handle.

Signature:

```
nvjpegStatus_t nvjpegDecoderDestroy(
 nvjpegJpegDecoder_t decoder_handle);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegJpegDecoder_t decoder_handle	Input/Output	Host	Decoder handle.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.15. nvjpegDecoderJpegSupported()

Determines whether the decoder_handle is able to handle the bit-stream stored in jpeg_stream.

Signature:

```
nvjpegStatus_t nvjpegDecoderJpegSupported(
 nvjpegJpegDecoder_t decoder_handle,
 nvjpegJpegStream_t jpeg_stream,
 nvjpegDecodeParams_t decode_params,
```

```
 int* is_supported);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegJpegDecoder_t decoder_handle	Input	Host	Decoder state handle
nvjpegJpegStream_t jpeg_stream	Input	Host	Bit stream meta-data
nvjpegDecodeParams_t decode_params	Input	Host	Decoder output configuration
int* is_supported	Output	Host	Return value of 0 indicates bitstream can be decoded by the decoder_handle, non zero value indicates that the bitstream is not supported

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.16. nvjpegDecoderStateCreate()

Creates the `decoder_state` internal structure. The `decoder_state` is associated with the nvjpegBackend_t implementation that was used to create the `decoder_handle`.

Signature:

```
nvjpegStatus_t nvjpegDecoderStateCreate(
 nvjpegHandle_t nvjpeg_handle,
 nvjpegJpegDecoder_t decoder_handle,
 nvjpegJpegState_t* decoder_state);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegHandle_t nvjpeg_handle	Input	Host	Library handle.
nvjpegJpegDecoder_t decoder_handle	Input	Host	Decoder handle.
nvjpegJpegState_t* decoder_state	Input/Output	Host	nvJPEG Image State Handle.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.17. nvjpegJpegStreamCreate()

Creates `jpeg_stream` that is used to parse the JPEG bitstream and store bitstream parameters.

Signature:

```
nvjpegStatus_t nvjpegJpegStreamCreate(
 nvjpegHandle_t handle,
```

```
 nvjpegJpegStream_t *jpeg_stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegHandle_t handle	Input	Host	Library handle
nvjpegJpegStream_t *jpeg_stream	Input	Host	Bitstream handle

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.18. nvjpegJpegStreamDestroy()

Destroys the `jpeg_stream` structure.

Signature:

```
nvjpegStatus_t nvjpegJpegStreamDestroy(
 nvjpegJpegStream_t *jpeg_stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegJpegStream_t *jpeg_stream	Input	Host	Bitstream handle

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.19. nvjpegBufferPinnedCreate()

Creates a pinned buffer handle.

Signature:

```
nvjpegStatus_t nvjpegBufferPinnedCreate(
 nvjpegHandle_t handle,
 nvjpegPinnedAllocator_t* pinned_allocator,
 nvjpegBufferPinned_t* buffer);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegHandle_t handle	Input	Host	Library handle.
nvjpegPinnedAllocator_t* pinned_allocator	Input	Host	Pinned host memory allocator. See nvjpegPinnedAllocator_t structure description.
nvjpegBufferPinned_t* buffer	Input/Output	Host	nvJPEG pinned buffer object.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.20. nvjpegBufferPinnedDestroy()

Destroys a pinned buffer handle.

Signature:

```
nvjpegStatus_t nvjpegBufferPinnedDestroy(
 nvjpegBufferPinned_t buffer);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegBufferPinned_t* buffer	Input/Output	Host	nvJPEG pinned buffer object.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.21. nvjpegStateAttachPinnedBuffer()

Link the nvJPEG pinned buffer handle to `decoder_state`. The `pinned_buffer` is used by the decoder to store the intermediate information that is used across the decoding stages. Pinned buffer can be attached to different decoder states, which helps to switch between implementations without allocating extra memory.

Signature:

```
nvjpegStatus_t nvjpegStateAttachPinnedBuffer(
 nvjpegJpegState_t decoder_state,
 nvjpegBufferPinned_t pinned_buffer);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegJpegState_t decoder_state	Input	Host	nvJPEG decoder state.
nvjpegBufferPinned_t pinned_buffer	Input	Host	nvJPEG pinned buffer container.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.22. nvjpegBufferPinnedRetrieve()

Retrieves the pinned memory pointer and size from the nvJPEG pinned buffer handle. Allows the application to re-use the memory once the decode is complete.

Signature:

```
nvjpegStatus_t nvjpegBufferPinnedRetrieve(
 nvjpegBufferPinned_t buffer,
 size_t* size, void** ptr);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegBufferPinned_t buffer	Input	Host	nvJPEG pinned buffer container.

size_t* size	Input/Output	Host	Size in bytes of the pinned buffer.
void** ptr	Input/Output	Host	Pointer to the pinned buffer.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.23. nvjpegBufferDeviceCreate()

Creates the device buffer handle.

Signature:

```
nvjpegStatus_t nvjpegBufferDeviceCreate(
 nvjpegHandle_t handle,
 nvjpegDevAllocator_t* device_allocator,
 nvjpegBufferDevice_t* buffer);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegHandle_t handle	Input	Host	Library handle.
nvjpegDevAllocator_t* device_allocator	Input	Host	Device memory allocator. See nvjpegDevAllocator_t structure description.
nvjpegBufferDevice_t* buffer	Input/Output	Host	nvJPEG device buffer container.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.24. nvjpegBufferDeviceDestroy()

Destroys the device buffer handle.

Signature:

```
nvjpegStatus_t nvjpegBufferDeviceDestroy(
 nvjpegBufferDevice_t buffer);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegBufferDevice_t* buffer	Input/Output	Host/Device	nvJPEG device buffer container. Device pointers are stored within the host structures.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.25. nvjpegStateAttachDeviceBuffer()

Link the nvJPEG device buffer handle to the `decoder_state`. The `device_buffer` is used by the decoder to store the intermediate information that is used across the decoding stages.

Device buffer can be attached to different decoder states, which helps to switch between implementations without allocating extra memory.

Signature:

```
nvjpegStatus_t nvjpegStateAttachDeviceBuffer(
 nvjpegJpegState_t decoder_state,
 nvjpegBufferDevice_t device_buffer);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegJpegState_t decoder_state	Input	Host	nvJPEG decoder state.
nvjpegBufferDevice_t device buffer	Input	Host/Device	nvJPEG device buffer container. Device pointers are stored within the host structures.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.26. nvjpegBufferDeviceRetrieve()

Retrieve the device memory pointer and size from the nvJPEG device buffer handle. Allows the application to re-use the memory after the decode is complete.

Signature:

```
nvjpegStatus_t nvjpegBufferDeviceRetrieve(
 nvjpegBufferDevice_t buffer,
 size_t* size,
 void** ptr);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegBufferDevice_t buffer	Input	Host	nvJPEG device buffer container.
size_t* size	Input/Output	Host	Device buffer size in bytes.
void** ptr	Input/Output	Host	Pointer to the device buffer.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.27. nvjpegDecodeParamsCreate()

Creates a handle for the parameters. The parameters that can be programmed include: output format, ROI decode, CMYK to RGB conversion.

Signature:

```
nvjpegStatus_t nvjpegDecodeParamsCreate(
 nvjpegHandle_t handle,
 nvjpegDecodeParams_t *decode_params);
```

Parameters:

Parameter	Input / Output	Memory	Description

nvjpegHandle_t handle	Input	Host	Library handle.
nvjpegDecodeParams_t *decode_params	Input/Output	Host	Decode output parameters.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.1.28. nvjpegDecodeParamsDestroy()

Destroys the `decode_params` handle.

Signature:

```
nvjpegStatus_t nvjpegDecodeParamsDestroy(
 nvjpegDecodeParams_t *decode_params);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegDecodeParams_t *decode_params	Input/Output	Host	Decode output parameters.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.2. Retrieve Encoded Image Information API

The helper functions for retrieving the encoded image information.

2.3.2.1. nvjpegGetImageInfo()

Decodes the JPEG header and retrieves the basic information about the image.

Signature:

```
nvjpegStatus_t nvjpegGetImageInfo(
 nvjpegHandle_t              handle,
 const unsigned char         *data,
 size_t                      length,
 int                         *nComponents,
 nvjpegChromaSubsampling_t   *subsampling,
 int                         *widths,
 int                         *heights);
```

Parameters:

Parameter	Input / Output	Memory	Description
`nvjpegHandle_t handle`	Input	Host	The library handle.
`const unsigned char *data`	Input	Host	Pointer to the encoded data.
`size_t length`	Input	Host	Size of the encoded data in bytes.
`int *nComponents`	Output	Host	Chroma subsampling for the 1- or 3- channel encoding.

`int *widths`	Output	Host	Pointer to the first element of array of size NVJPEG_MAX_COMPONENT, where the width of each channel (up to NVJPEG_MAX_COMPONENT) will be saved. If the channel is not encoded, then the corresponding value would be zero.
`int *heights`	Output	Host	Pointer to the first element of array of size NVJPEG_MAX_COMPONENT, where the height of each channel (up to NVJPEG_MAX_COMPONENT) will be saved. If the channel is not encoded, then the corresponding value would be zero.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.2.2. nvJPEG Stream API

These functions store the parsed bit-stream data on the host.

2.3.2.2.1. nvjpegJpegStreamParse()

Parses the bitstream and stores the metadata in the `jpeg_stream` struct.

Signature:

```
nvjpegStatus_t nvjpegJpegStreamParse(
 nvjpegHandle_t handle,
 const unsigned char *data,
 size_t length,
 int save_metadata,
 int save_stream,
 nvjpegJpegStream_t jpeg_stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegHandle_t handle	Input	Host	The library handle.
const unsigned char *data	Input	Host	Pointer to the bit-stream.
size_t length	Input	Host	Bit-stream size.
int save_metadata	Input	Host	(Not enabled. Marked for future use). If not 0, then the JPEG stream metadata (headers, app markers, etc.) will be saved in the internal JpegStream structure for future usage.

			If 0, then the meta data (headers, app markerms etc.) will be discarded.
int save_stream	Input	Host	If not 0, then the whole jpeg stream will be copied to the internal JpegStream structure, and the pointer to the JPEG file data will not be needed after this call. If 0, then JpegStream will just save the pointers (to JPEG file data), and these pointers will be used later during the image decoding.
nvjpegJpegStream_t jpeg_stream	Input/Output	Host/ Device	The nvJPEG bitstream handle that stores the parsed bitstream information.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.2.2.2. nvjpegJpegStreamParseHeader()

Parses only the header of the bit-stream and stores the header information in the `jpeg_stream` struct.

Signature:

```
nvjpegStatus_t nvjpegJpegStreamParseHeader(
 nvjpegHandle_t handle,
 const unsigned char *data,
 size_t length,
 nvjpegJpegStream_t jpeg_stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegHandle_t handle	Input	Host	The library handle.
const unsigned char *data	Input	Host	Pointer to the bit-stream.
size_t length	Input	Host	Bit-stream size.
nvjpegJpegStream_t jpeg_stream	Input/Output	Host/ Device	The nvJPEG bitstream handle that stores the parsed bitstream information.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.2.2.3. nvjpegJpegStreamGetFrameDimensions()

Extracts the JPEG frame dimensions from the bitstream.

Signature:

```
nvjpegStatus_t nvjpegJpegStreamGetFrameDimensions(
 nvjpegJpegStream_t jpeg_stream,
 unsigned int* width,
 unsigned int* height);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegJpegStream_t jpeg_stream	Input	Host	Bitstream handle.
unsigned int* width	Output	Host	Frame height.
unsigned int* height	Output	Host	Frame width.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.2.2.4. nvjpegJpegStreamGetComponentsNum()

Extracts the JPEG frame dimensions from the bitstream.

Signature:

```
nvjpegStatus_t nvjpegJpegStreamGetComponentsNum(
 nvjpegJpegStream_t jpeg_stream,
 unsigned int* components_num);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegJpegStream_t jpeg_stream	Input	Host	Bitstream handle.
unsigned int* components_num	Output	Host	Number of encoded channels in the input.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.2.2.5. nvjpegJpegStreamGetComponentDimensions()

Extracts the component dimensions from the bitstream.

Signature:

```
nvjpegStatus_t nvjpegJpegStreamGetComponentDimensions(
 nvjpegJpegStream_t jpeg_stream,
 unsigned int component,
 unsigned int* width,
 unsigned int* height)
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegJpegStream_t jpeg_stream	Input	Host	Bitstream handle.

unsigned int component	Input	Host	Component index.
unsigned int* width	Output	Host	Component height.
unsigned int* height	Output	Host	Component width.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.2.2.6. nvjpegJpegStreamGetChromaSubsampling()

Gets the chroma subsampling from the `jpeg_stream`. For grayscale (single channel) images it returns NVJPEG_CSS_GRAY. For 3-channel images it tries to assign one of the known chroma sub-sampling values based on the sampling information present in the bitstream, else it returns NVJPEG_CSS_UNKNOWN. If the number of channels is 2 or 4, then it returns NVJPEG_CSS_UNKNOWN.

Signature:

```
nvjpegStatus_t nvjpegJpegStreamGetChromaSubsampling(
 nvjpegJpegStream_t jpeg_stream,
 nvjpegChromaSubsampling_t* chroma_subsampling);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegJpegStream_t jpeg_stream	Input	Host	Bitstream handle
nvjpegChromaSubsampling_t* chroma_subsampling	Output	Host	Chroma subsampling for the 1- or 3- channel encoding.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.2.2.7. nvjpegJpegStreamGetJpegEncoding()

This function obtains the JPEG encoding type from the `jpeg_stream`. For baseline images it returns NVJPEG_ENCODING_BASELINE_DCT. For progressive images it returns NVJPEG_ENCODING_PROGRESSIVE_DCT_HUFFMAN.

Signature:

```
nvjpegStatus_t nvjpegJpegStreamGetJpegEncoding(
 nvjpegJpegStream_t jpeg_stream,
 nvjpegJpegEncoding_t* jpeg_encoding);
```

Parameters:

Parameter	Input / Output	Memory	Description
jpeg_stream	In	Host	Input bitstream handle.
jpeg_encoding	Out	Host	Encoding type obtained—baseline or progressive.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.3. Decode API—Single Phase

Functions for decoding single image or batched images in a single phase.

2.3.3.1. nvjpegDecode()

Decodes a single image, and writes the decoded image in the desired format to the output buffers. This function is asynchronous with respect to the host. All GPU tasks for this function will be submitted to the provided stream.

From CUDA 11 onwards, nvjpegDecode() picks the best available back-end for a given image, user no longer has control on this. If there is a need to select the back-end, then consider using nvjpegDecodeJpeg. This is a new API added in CUDA 11 which allows user to control the back-end

Signature:

```
nvjpegStatus_t nvjpegDecode(
 nvjpegHandle_t          handle,
 nvjpegJpegState_t       jpeg_handle,
 const unsigned char     *data,
 size_t                  length,
 nvjpegOutputFormat_t    output_format,
 nvjpegImage_t           *destination,
 cudaStream_t            stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
`nvjpegHandle_t handle`	Input	Host	The library handle.
`nvjpegJpegState_t jpeg_handle`	Input	Host	The image state handle.
`const unsigned char *data`	Input	Host	Pointer to the encoded data.
`size_t length`	Input	Host	Size of the encoded data in bytes.
`nvjpegOutputFormat_t output_format`	Input	Host	Format in which the decoded output will be saved.
`nvjpegImage_t *destination`	Input/Output	Host/ Device	Pointer to the structure that describes the output destination. This structure should be on the host (CPU), but the pointers in this structure should be pointing to the device (i.e., GPU) memory. See `nvjpegImage_t`.
`cudaStream_t stream`	Input	Host	The CUDA stream where all of the GPU work will be submitted.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.3.2. nvjpegDecodeBatchedInitialize()

This function initializes the batched decoder state. The initialization parameters include the batch size, the maximum number of CPU threads, and the specific output format in which the decoded image will be saved. This function should be called once, prior to decoding the batches of images. Any currently running batched decoding should be finished before calling this function.

Signature:

```
nvjpegStatus_t nvjpegDecodeBatchedInitialize(
  nvjpegHandle_t          handle,
  nvjpegJpegState_t       jpeg_handle,
  int                     batch_size,
  int                     max_cpu_threads,
  nvjpegOutputFormat_t    output_format);
```

Parameters:

Parameter	Input / Output	Memory	Description
`nvjpegHandle_t handle`	Input	Host	The library handle.
`nvjpegJpegState_t jpeg_handle`	Input	Host	The image state handle.
`int batch_size`	Input	Host	Batch size.
`int max_cpu_threads`	Input	Host	Maximum number of CPU threads that can participate in decoding a batch.
`nvjpegOutputFormat_t output_format`	Input	Host	Format in which the decoded output will be saved.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.3.3. nvjpegDecodeBatched()

Decodes the batch of images, and writes them to the buffers described in the `destination` parameter in a format provided to `nvjpegDecodeBatchedInitialize()` function. This function is asynchronous with respect to the host. All GPU tasks for this function will be submitted to the provided stream.

Signature:

```
nvjpegStatus_t nvjpegDecodeBatched(
  nvjpegHandle_t              handle,
  nvjpegJpegState_t           jpeg_handle,
  const unsigned char *const  *data,
  const size_t                *lengths,
  nvjpegImage_t               *destinations,
  cudaStream_t                stream);
```

Parameters:

Parameter	Input / Output	Memory	Description

`nvjpegHandle_t handle`	Input	Host	The library handle.
`nvjpegJpegState_t jpeg_handle`	Input	Host	The image state handle.
`const unsigned char *const *data`	Input	Host	Pointer to the first element of array of the input data. The size of the array is assumed to be batch_size provided to `nvjpegDecodeBatchedInitialize()` batch initialization function.
`const size_t *lengths`	Input	Host	Pointer to the first element of array of input sizes. Size of array is assumed to be batch_size provided to `nvjpegDecodeBatchedInitialize()`, the batch initialization function.
`nvjpegImage_t *destinations`	Input/ Output	Host/ Device	Pointer to the first element of array of output descriptors. The size of array is assumed to be batch_size provided to `nvjpegDecodeBatchedInitialize()`, the batch initialization function. See also `nvjpegImage_t`.
`cudaStream_t stream`	Input	Host	The CUDA stream where all the GPU work will be submitted.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.3.4. nvjpegDecodeBatchedEx()

This API helps to Decodes the batch of images with ROI, and writes them to the buffers described in the `destination` parameter in a format provided to `nvjpegDecodeBatchedInitialize()` function. This function is asynchronous with respect to the host. All GPU tasks for this function will be submitted to the provided stream.

Signature:

```
nvjpegStatus_t nvjpegDecodeBatchedEx(
          nvjpegHandle_t handle,
          nvjpegJpegState_t jpeg_handle,
          const unsigned char *const *data,
          const size_t *lengths,
          nvjpegImage_t *destinations,
          nvjpegDecodeParams_t *decode_params,
          cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegHandle_t handle	Input	Host	nvjpeg library handle
nvjpegJpegState_t jpeg_handle	Input	Host	The image state handle
const unsigned char *const *data	Input	Host	Pointer to the first element of array of the input data. The

			size of the array is assumed to be batch_size provided to nvjpegDecodeBatchedInitialize() batch initialization function
const size_t *lengths	Input	Host	Pointer to the first element of array of input sizes
nvjpegImage_t *destinations	Input/Output	Host/Device	Pointer to the first element of array of output descriptors. The size of array is assumed to be batch_size provided to nvjpegDecodeBatchedInitialize(), the batch initialization function. See also nvjpegImage_t
nvjpegDecodeParams_t *decode_params	Input	Host	Setting ROI Decode parameters
cudaStream_t stream	Input	Host	The CUDA stream where all the GPU work will be submitted

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.3.5. nvjpegDecodeBatchedSupported()

This API helps determine whether an image can be decoded by nvjpegDecodeBatched. User can parse the bitstream header using nvjpegJpegStreamParseHeader and then call this API to determine whether the image can be decoded.

Signature:

```
nvjpegStatus_t nvjpegDecodeBatchedSupported(
 nvjpegHandle_t handle,
 nvjpegJpegStream_t jpeg_stream,
 int* is_supported);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegHandle_t handle	Input	Host	nvjpeg library handle
nvjpegJpegStream_t jpeg_stream	Input	Host	Bit stream meta-data
int* is_supported	Output	Host	Return value of 0 indicates bitstream can be decoded by the decoder_handle, non zero value indicates that the bitstream is not supported

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.3.6. nvjpegDecodeBatchedSupportedEx()

This API helps determine whether an image can be decoded by nvjpegDecodeBatchedEx. User can parse the bitstream header using nvjpegJpegStreamParseHeader and set the ROI in the decode params then call this API to determine whether the image can be decoded.

Signature:

```
nvjpegStatus_t nvjpegDecodeBatchedSupportedEx(
 nvjpegHandle_t handle,
 nvjpegJpegStream_t jpeg_stream,
 nvjpegDecodeParams_t decode_params,
 int* is_supported);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegHandle_t handle	Input	Host	nvjpeg library handle
nvjpegJpegStream_t jpeg_stream	Input	Host	Bit stream meta-data
nvjpegDecodeParams_t decode_params	Input	Host	Setting ROI Decode parameters
int* is_supported	Output	Host	Return value of 0 indicates bitstream can be decoded by the decoder_handle, non zero value indicates that the bitstream is not supported

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.3.7. nvjpegDecodeBatchedPreAllocate()

This is an experimental API that can be used with nvjpegDecodeBatched. When decoding images with varying sizes and chroma subsampling, performance is limited by the repeated cuda calls made by the library to free/allocate device memory. This API attempts to avoid this problem by allocating device memory prior to the actual decoding. Users have the option to call this API with values that are unlikely to be exceeded when nvjpegDecodeBatched is called.

Note:

This functionality is available only when the `nvjpegHandle_t`is instantiated using NVJPEG_BACKEND_HARDWARE. It is currently a No Op for other backends.

This API only provides a hint for initial allocation. If the image dimensions at the time of decode exceed what was provided, then the library will resize the device buffers.

If the images being decoded have different chroma subsamplings, then the `chroma_subsampling` field should be set to NVJPEG_CSS_444 to ensure that the device memory can be reused.

Signature:

```
nvjpegStatus_t nvjpegDecodeBatchedPreAllocate(
  nvjpegHandle_t          handle,
  nvjpegJpegState_t       jpeg_handle,
  int                     batch_size,
  int                     width,
 int                     height,
 nvjpegChromaSubsampling_t chroma_subsampling,
  nvjpegOutputFormat_t    output_format);
```

Parameters:

Parameter	Input / Output	Memory	Description
`nvjpegHandle_t handle`	Input	Host	The library handle.
`nvjpegJpegState_t jpeg_handle`	Input	Host	The image state handle.
`int batch_size`	Input	Host	Batch size.
`int width`	Input	Host	Maximum width of image that will be decoded
`int height`	Input	Host	Maximum height of image that will be decoded
`nvjpegChromaSubsampling_t chroma_subsampling`	Input	Host	Chroma-subsampling of the images
`nvjpegOutputFormat_t output_format`	Input	Host	Format in which the decoded output will be saved

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.4. Decode API—Decoupled Decoding

This set of decoding API works with the bitstream handles, decode parameter handles, pinned and device buffers handles as input, thus decoupling JPEG bitstream parse, buffer management and setting up decoder parameters from the decode process itself.

Currently only multiphase decoding is available. Multiphase decoupled single image decoding consists of three phases:

- Host,
- Mixed, and
- Device,

Each of the above decodings is carried on according to its individual semantics. Phases on different images can be carried out with different decoding state handles simultaneously, while sharing of some helper objects is possible. See the details of semantics in the individual phases descriptions.

Below are a couple of examples of using decoupled API.

The snippet below explains how to use the API to prefetch the host stage of the processing: first do all of the host work on the host, and then submit the rest of decoding work to the device.

```
#define BATCH_SIZE 2
nvjpegHandle_t nvjpeg_handle;
```

```
nvjpegJpegState_t nvjpeg_decoder_state[BATCH_SIZE];
nvjpegBufferPinned_t nvjpeg_pinned_buffer[BATCH_SIZE];
nvjpegBufferDevice_t nvjpeg_device_buffer;
nvjpegJpegStream_t nvjpeg_jpeg_stream[BATCH_SIZE];
nvjpegDecodeParams_t nvjpeg_decode_params;
nvjpegJpegDecoder_t nvjpeg_decoder;
nvjpegBackend_t impl = NVJPEG_BACKEND_DEFAULT;

unsigned char* bitstream[BATCH_SIZE] // pointers jpeg bitstreams
size_t  length[BATCH_SIZE]; // bitstream sizes

nvjpegImage_t output_images[BATCH_SIZE];

// all the images in the batch will be decoded as RGBI
nvjpegDecodeParamsSetOutputFormat(nvjpeg_decode_params,NVJPEG_OUTPUT_RGBI );

// call host phase for two bitstreams
for (int i = 0; i < BATCH_SIZE; i++)
{
 nvjpegJpegStreamParse(nvjpeg_handle, bitstream[i], length[i], 0, 0,
 nvjpeg_jpeg_stream[i]);
 nvjpegStateAttachPinnedBuffer(nvjpeg_decoder_state[i], nvjpeg_pinned_buffer[i]);
 nvjpegDecodeJpegHost(nvjpeg_handle, nvjpeg_decoder, nvjpeg_decoder_state[i],
 nvjpeg_decode_params, nvjpeg_jpeg_stream[i])
}

for (int i = 0; i < BATCH_SIZE; i++)
{
 // same device buffer being used for decoding bitstreams
 nvjpegStateAttachDeviceBuffer(nvjpeg_decoder_state[i], nvjpeg_device_buffer);

 // cuda stream set to NULL
 nvjpegDecodeJpegTransferToDevice(nvjpeg_handle, nvjpeg_decoder,
 nvjpeg_decoder_state[i], nvjpeg_jpeg_stream[i], NULL);
 // cuda stream set to NULL
 nvjpegDecodeJpegDevice(nvjpeg_handle, nvjpeg_decoder, nvjpeg_decoder_state[i],
 &output_images[i], NULL);
 cudaDeviceSynchronize();
}
```

The below snippet explains how pinned and device buffers can be shared across two instances of nvJPEG Decoder Handle.

```
#define BATCH_SIZE 4
nvjpegHandle_t nvjpeg_handle;
nvjpegJpegDecoder_t nvjpeg_decoder_impl1;
nvjpegJpegDecoder_t nvjpeg_decoder_impl2;
nvjpegJpegState_t nvjpeg_decoder_state_impl1;
nvjpegJpegState_t nvjpeg_decoder_state_impl2;
nvjpegBufferPinned_t nvjpeg_pinned_buffer;
nvjpegBufferDevice_t nvjpeg_device_buffer;
nvjpegJpegStream_t nvjpeg_jpeg_stream;
nvjpegDecodeParams_t nvjpeg_decode_params;

unsigned char* bitstream[BATCH_SIZE] // pointers jpeg bitstreams
size_t  length[BATCH_SIZE]; // bitstream sizes

// populate bitstream and length correctly for this code to work
nvjpegImage_t output_images[BATCH_SIZE];

// allocate device memory for output images, for this snippet to work
nvjpegStateAttachPinnedBuffer(nvjpeg_decoder_state_impl1, nvjpeg_pinned_buffer);
nvjpegStateAttachPinnedBuffer(nvjpeg_decoder_state_impl2, nvjpeg_pinned_buffer);
nvjpegStateAttachDeviceBuffer(nvjpeg_decoder_state_impl1, nvjpeg_device_buffer);
nvjpegStateAttachDeviceBuffer(nvjpeg_decoder_state_impl2, nvjpeg_device_buffer);

// all the images in the batch will be decoded as RGBI
```

```
nvjpegDecodeParamsSetOutputFormat(nvjpeg_decode_params,NVJPEG_OUTPUT_RGBI );

for (int i = 0; i < BATCH_SIZE; i++)
{
 nvjpegJpegStreamParse(nvjpeg_handle,bitstream[i],length[i],0,0,nvjpeg_jpeg_stream);

 // decide which implementation to use, based on image size
 unsigned int frame_width;
 unsigned int frame_height;
 nvjpegJpegStreamGetFrameDimensions(nvjpeg_jpeg_stream,&frame_width,
 &frame_height));
 nvjpegJpegDecoder_t& decoder = (frame_height*frame_width > 1024 * 768 ) ?
 nvjpeg_decoder_impl2: nvjpeg_decoder_impl1;
 nvjpegJpegState_t& decoder_state = (frame_height * frame_width > 1024 * 768) ?
 nvjpeg_decoder_state_impl2:nvjpeg_decoder_state_impl1;

 nvjpegDecodeJpegHost(nvjpeg_handle,decoder,decoder_state,nvjpeg_decode_params,nvjpeg_jpeg_stream)

 // cuda stream set to NULL
 nvjpegDecodeJpegTransferToDevice(nvjpeg_handle,decoder,decoder_state,nvjpeg_jpeg_stream,NULL);

 // cuda stream set to NULL
 nvjpegDecodeJpegDevice(nvjpeg_handle,nvjpeg_decoder,decoder_state,&output_images,
 NULL);
 cudaDeviceSynchronize();
}
```

2.3.4.1. nvjpegDecodeJpegHost()

This is the first stage of the decoupled decoding process. It is done entirely on the host, hence it is synchronous with respect of the host.

If a pinned buffer is attached to the decoder state, then the pinned buffer object will be used to allocate the pinned memory required for the host decoding phase. There wouldn't be allocation if the pinned buffer object already handles the required amount of pinned memory.

If pinned buffer object is not attached, then the state will use heap host memory to allocate the memory required for the host processing.

In this phase device is not participating. Hence the device selection, device initialization, and device memory initialization can be done later in the decoding process.

This function works on a parsed stream. The parsed stream handle that is available after calling the nvjpegJpegStreamParse() function should be provided to this function.

Signature:

```
nnvjpegStatus_t nvjpegDecodeJpegHost(
 nvjpegHandle_t handle,
 nvjpegJpegDecoder_t decoder,
 nvjpegJpegState_t decoder_state,
 nvjpegDecodeParams_t decode_params,
 nvjpegJpegStream_t jpeg_stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegHandle_t handle	Input	Host	The library handle.
nvjpegJpegDecoder_t decoder	Input	Host	The nvJPEG decoder handle.
nvjpegJpegState_t decoder_state	Input	Host	The nvJPEG decoder state handle.

nvjpegDecodeParams_t decode_params	Input	Host	Handle to decode the output properties.
nvjpegJpegStream_t jpeg_stream	Input	Host	Handle to the parsed bitstream data.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.4.2. nvjpegDecodeJpegTransferToDevice()

This phase contains both host and device operations. Hence it is a mix of synchronous and asynchronous operations with respect to the host. All the device operations will be submitted to the provided stream.

This phase should be called only after the host phase with the same decoder handle, decoder state handle and parsed jpeg stream handle. Device should be initialized and device buffer should be attached to `decoder_state` handle using nvjpegStateAttachDeviceBuffer() prior to calling this API. This device buffer object will be resized to the required amount of memory if needed. For the host memory buffer, this phase will use whatever was used in the host phase: either the attached pinned buffer or the state's host memory buffer.

Signature:

```
nvjpegStatus_t nvjpegDecodeJpegTransferToDevice(
 nvjpegHandle_t handle,
 nvjpegJpegDecoder_t decoder,
 nvjpegJpegState_t decoder_state,
 nvjpegJpegStream_t jpeg_stream,
 cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegHandle_t handle	Input	Host	The library handle.
nvjpegJpegDecoder_t decoder	Input	Host	The nvJPEG decoder handle.
nvjpegJpegState_t decoder_state	Input	Host	The nvJPEG decoder state handle.
nvjpegJpegStream_t jpeg_stream	Input	Host	Handle to the parsed bitstream data.
cudaStream_t stream	Input	Host	The CUDA stream to which all the GPU tasks will be submitted.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.4.3. nvjpegDecodeJpegDevice()

This phase consists of decode operations that take place mainly on the device (no significant host side computation is done). Hence this phase is asynchronous with respect to the host. This phase should be called after nvjpegDecodeJpegTransferToDevice() for a given `decoder_state` handle and decoder handle.

In this function call the host memory buffers are not used, so if the pinned buffer was attached to the state, then it can be reused somewhere else. Note that at this point the Jpeg stream handle is not needed anymore, since parts that are needed for device decoding will be copied to the device memory in the previous phase.

Signature:

```
nvjpegStatus_t nvjpegDecodeJpegDevice(
 nvjpegHandle_t handle,
 nvjpegJpegDecoder_t decoder,
 nvjpegJpegState_t decoder_state,
 nvjpegImage_t *destination,
 cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegHandle_t handle	Input	Host	The library handle.
nvjpegJpegDecoder_t decoder	Input	Host	The nvJPEG decoder handle.
nvjpegJpegState_t decoder_state	Input	Host	The nvJPEG decoder state handle.
nvjpegImage_t *destination	Input/Output	Host/ Device	Pointer to a structure that describes the output destination. This structure should be on host, but the pointers in this structure should be pointing to the device memory. See nvJPEG Image for details.
cudaStream_t stream	Input	Host	The CUDA stream to which all the GPU tasks will be submitted.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.4.4. nvjpegDecodeJpeg()

This is a single phase API with the flexibility to select nvjpeg back-end when creating nvjpegJpegDecoder_t object. User has the option to call this API instead of making three separate calls to nvjpegDecodeJpegHost(), nvjpegDecodeJpegTransferToDevice(), nvjpegDecodeJpegDevice().

It is required to atttach the device buffer to the decoder state before calling this API. The pinned buffer is optional. If pinned buffer is not attached, then heap memory will be used for host processing

This function works on a parsed stream. The parsed stream handle that is available after calling the nvjpegJpegStreamParse() function should be provided to this function.

Signature:

```
nvjpegStatus_t nvjpegDecodeJpeg(
 nvjpegHandle_t handle,
 nvjpegJpegDecoder_t decoder,
 nvjpegJpegState_t decoder_state,
 nvjpegJpegStream_t jpeg_bitstream,
 nvjpegImage_t *destination,
 nvjpegDecodeParams_t decode_params,
```

```
cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegHandle_t handle	Input	Host	The library handle.
nvjpegJpegDecoder_t decoder	Input	Host	The nvJPEG decoder handle.
nvjpegJpegState_t decoder_state	Input	Host	The nvJPEG decoder state handle.
nvjpegJpegStream_t jpeg_stream	Input	Host	Handle to the parsed bitstream data.
nvjpegImage_t *destination	Input/Output	Host/ Device	Pointer to a structure that describes the output destination. This structure should be on host, but the pointers in this structure should be pointing to the device memory. See nvJPEG Image for details.
nvjpegDecodeParams_t decode_params	Input	Host	Handle which stores the decode output properties.
cudaStream_t stream	Input	Host	The CUDA stream to which all the GPU tasks will be submitted.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.5. nvJPEG Decode Parameters

This category of APIs are used to set the decoding parameters. These APIs should be used with the decode APIs defined in Decode API—Decoupled Decoding.

2.3.5.1. nvjpegDecodeParamsSetOutputFormat()

This function is used to set the decode output format. See `nvjpegOutputFormat_t` described in step 6 of Single Image Decoding. The output parameter of `nvjpegOutputFormat_t` defaults to NVJPEG_OUTPUT_UNCHANGED if not set using this API.

Signature:

```
nvjpegStatus_t nvjpegDecodeParamsSetOutputFormat(
 nvjpegDecodeParams_t decode_params,
 nvjpegOutputFormat_t output_format);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegDecodeParams_t decode_params	Input	Host	Decode output parameter handle.
nvjpegOutputFormat_t output_format	Input	Host	See step 6 of Single Image Decoding.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.5.2. nvjpegDecodeParamsSetROI()

This function enables the region of interest-only (ROI-only) decode. To disable the ROI-only, i.e., to decode the whole image, set:

- `offset_x` = 0,
- `offset_y` = 0,
- `roi_width` = -1, and
- `roi_height` = -1.

Note: ROI decode is disabled by default. It is not supported when nvjpeg decoder handle is created using NVJPEG_BACKEND_HARDWARE

The ROI window cannot go out of image bounds. That is:

- `offset_x` cannot be lower than zero, or
- `offset_x + roi_width` cannot be larger than the JPEG image width.

If the output format is NVJPEG_OUTPUT_YUV or NVJPEG_OUTPUT_UNCHANGED, then the `offset_x` and `offset_y` values have to be multiples of the maximum subsampling factor, as defined in the JPEG standard.

Signature:

```
nvjpegStatus_t nvjpegDecodeParamsSetROI(
 nvjpegDecodeParams_t decode_params,
 int offset_x,
 int offset_y,
 int roi_width,
 int roi_height);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegDecodeParams_t decode_params	Input	Host	The decode output parameter handle.
int offset_x	Input	Host	Image offset along the horizontal direction relative to the top left corner.
int offset_y	Input	Host	Image offset along the vertical direction relative to the top left corner.
int roi_width	Input	Host	Image width relative to offset_x.
int roi_height	Input	Host	Image height relative to offset_y.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.5.3. nvjpegDecodeParamsSetAllowCMYK()

If enabled, the nvJPEG library assumes that the JPEG with 4 encoded color components is in CMYK colorspace, and enables the conversion to RGB/YUV colorspace. The CMYK-to-RGB conversion is disabled by default. The conversion is based on the subtractive scheme—this behaviour matches OpenCV's handling of 4-component JPEGs.

Signature:

```
nvjpegStatus_t nvjpegDecodeParamsSetAllowCMYK(
 nvjpegDecodeParams_t decode_params,
 int allow_cmyk);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegDecodeParams_t decode_params	Input	Host	Decode output parameter handle.
int allow_cmyk	Input	Host	Enable CMYK to RGB conversion.

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.5.4. nvjpegDecodeParamsSetScaleFactor()

Allows the user to scale decode output.

Note: This feature is currently supported only when nvjpeg decoder handle is created using NVJPEG_BACKEND_HARDWARE

Signature:

```
nvjpegStatus_t nvjpegDecodeParamsSetScaleFactor(
 nvjpegDecodeParams_t decode_params,
 nvjpegScaleFactor_t scale_factor);
```

Parameters:

Parameter	Input / Output	Memory	Description
nvjpegDecodeParams_t decode_params	Input	Host	Decode output parameter handle.
nvjpegScaleFactor_t scale_factor	Input	Host	Set the scaling factor for the decode output

The scale factor is set to NVJPEG_SCALE_NONE by default. The supported values are listed here

When setting a scale factor value, the recommended allocation of the destination parameters is as follows:

Use nvjpegGetImageInfo(), or nvjpegJpegStreamGetFrameDimensions() to extract the dimensions of each channel

Let height[NVJPEG_MAX_COMPONENT] and width[NVJPEG_MAX_COMPONENT], be 2 arrays which store the height and width. the index to these arrays correspond to the channel id

for a channel c, the scaled dimensions are calculated as follows:

scaled_height[c] = (height[c] + rounding_factor - 1)/rounding_factor

scaled_width[c] = (width[c] + rounding_factor - 1)/rounding_factor

when scale_factor = NVJPEG_SCALE_NONE, rounding_factor = 1

when scale_factor = NVJPEG_SCALE_1_BY_2, rounding_factor = 2

when scale_factor = NVJPEG_SCALE_1_BY_4, rounding_factor = 4

when scale_factor = NVJPEG_SCALE_1_BY_8, rounding_factor = 8

For the output_format: NVJPEG_OUTPUT_Y	**destination.pitch[0] should be at least:** width[0]	**destination.channel[0] should be at least of size:** destination.pitch[0]*height[0]
For the output_format	**destination.pitch[c] should be at least:**	**destination.channel[c] should be at least of size:**
NVJPEG_OUTPUT_YUV	width[c] for c = 0, 1, 2	destination.pitch[c]*height[c] for c = 0, 1, 2
NVJPEG_OUTPUT_RGB and NVJPEG_OUTPUT_BGR	width[0] for c = 0, 1, 2	destination.pitch[0]*height[0] for c = 0, 1, 2
NVJPEG_OUTPUT_RGBI and NVJPEG_OUTPUT_BGRI	width[0]*3	destination.pitch[0]*height[0]
NVJPEG_OUTPUT_UNCHANGED	width[c] for c = [0, nComponents - 1]	destination.pitch[c]*height[c] for c = [0, nComponents - 1]

Returns:

`nvjpegStatus_t` - An error code as specified in nvJPEG API Return Codes.

2.3.6. nvJPEG API Return Codes

The nvJPEG API adheres to the following return codes and their indicators:

```
typedef enum
{
 NVJPEG_STATUS_SUCCESS = 0,
 NVJPEG_STATUS_NOT_INITIALIZED = 1,
 NVJPEG_STATUS_INVALID_PARAMETER = 2,
 NVJPEG_STATUS_BAD_JPEG = 3,
 NVJPEG_STATUS_JPEG_NOT_SUPPORTED = 4,
 NVJPEG_STATUS_ALLOCATOR_FAILURE = 5,
 NVJPEG_STATUS_EXECUTION_FAILED = 6,
 NVJPEG_STATUS_ARCH_MISMATCH = 7,
 NVJPEG_STATUS_INTERNAL_ERROR = 8,
 NVJPEG_STATUS_IMPLEMENTATION_NOT_SUPPORTED = 9
} nvjpegStatus_t;
```

Description of the returned error codes:

Returned Error (Returned Code)	Description
`NVJPEG_STATUS_SUCCESS (0)`	The API call has finished successfully. Note that many of the calls are asynchronous and

	some of the errors may be seen only after synchronization.
NVJPEG_STATUS_NOT_INITIALIZED (1)	The library handle was not initialized. A call to nvjpegCreate() is required to initialize the handle.
NVJPEG_STATUS_INVALID_PARAMETER (2)	Wrong parameter was passed. For example, a null pointer as input data, or an image index not in the allowed range.
NVJPEG_STATUS_BAD_JPEG (3)	Cannot parse the JPEG stream. Check that the encoded JPEG stream and its size parameters are correct.
NVJPEG_STATUS_JPEG_NOT_SUPPORTED (4)	Attempting to decode a JPEG stream that is not supported by the nvJPEG library.
NVJPEG_STATUS_ALLOCATOR_FAILURE (5)	The user-provided allocator functions, for either memory allocation or for releasing the memory, returned a non-zero code.
NVJPEG_STATUS_EXECUTION_FAILED (6)	Error during the execution of the device tasks.
NVJPEG_STATUS_ARCH_MISMATCH (7)	The device capabilities are not enough for the set of input parameters provided (input parameters such as backend, encoded stream parameters, output format).
NVJPEG_STATUS_INTERNAL_ERROR (8)	Error during the execution of the device tasks.
NVJPEG_STATUS_IMPLEMENTATION_NOT_SUPPORTED (9)	Not supported.

2.3.7. nvJPEG Chroma Subsampling

One of the outputs of the nvjpegGetImageInfo() API is nvjpegChromaSubsampling_t. This parameter is an enum type, and its enumerator list comprises of the chroma subsampling property retrieved from the encoded JPEG image. Below are the chroma subsampling types the nvjpegGetImageInfo() function currently supports:

```
typedef enum
{
 NVJPEG_CSS_444,
 NVJPEG_CSS_422,
 NVJPEG_CSS_420,
 NVJPEG_CSS_440,
 NVJPEG_CSS_411,
 NVJPEG_CSS_410,
 NVJPEG_CSS_GRAY,
 NVJPEG_CSS_UNKNOWN
} nvjpegChromaSubsampling_t;
```

2.3.8. Reference Documents

Refer to the JPEG standard: https://jpeg.org/jpeg/

2.4. Examples of nvJPEG

nvJPEG Decode sample can be found here: https://github.com/NVIDIA/CUDALibrarySamples/tree/master/nvJPEG/nvJPEG-Decoder

Chapter 3. JPEG Encoding

This section describes the encoding functions of the nvJPEG Library.

3.1. Using the Encoder

The user should perform the below prerequisite steps before calling the nvJPEG encoding functions. See also nvJPEG Encoder Helper API Reference.

3.1.1. Encoding the Parameters

The user should create an encoding parameters structure with `nvjpegEncoderParamsCreate()` function. The function will be initialized with default parameters. User can use an appropriate `nvjpegEncoderParamsSet*()` function to set a specific parameter.

The quality parameter can be set, using `nvjpegEncoderParamsSetQuality()` function, to an integer value between 1 and 100, and this quality parameter will be used as a base for generating the JPEG quantization tables.

The parameters structure should be passed to compression functions.

Note: The encoding parameters structure can be reused to compress multiple images simultaneously, but no changes to the parameters should be made during the ongoing encoding, or encoding result will be undefined.

3.1.2. Encoding the State

The user should create the encoding state structure using `nvjpegEncoderStateCreate()` function. This function will hold intermediate buffers for the encoding process. This state should be passed to the compression functions.

Note: The encoding state structure can be reused to encode a series of images, but no encoding should be performed on multiple images with the same encoding state at the same time - otherwise result of the encodings will be undefined.

3.1.3. Encoding the Image

The nvJPEG library provides a few interfaces for compressing the image in different formats and colorspaces. See below.

3.1.3.1. nvjpegEncodeYUV

Input for this function is an image in YUV colorspace. See nvjpegEncodeYUV(). The `source` argument should be filled with the corresponding YUV planar data. The `chroma_subsampling` argument should have the chroma subsampling of the input data. If the chroma subsampling in the encoding parameters is the same as input chroma subsampling, then the user's input data will be directly used in the JPEG compression. Otherwise chroma will be resampled to match the chroma subsampling of the encoding parameters.

Input data should be provided with respect to the subsampling factors. That is, the chrominance image planes should have sizes aligned to the corresponding subsamplings. For example:

- Image dimensions: 123x321
- Input chroma subsampling is: NVJPEG_CSS_410
- Chroma subsampling factor for this chroma subsampling: is 4x2
- Given the above, the encoder library expects the user to provide:
  - Y plane with size: 123 x 321
  - Cb and Cr plane with size: 31 x 161

3.1.3.2. nvjpegEncodeImage

See nvjpegEncodeImage(). Input for this function, i.e., how data should be provided in the `source` argument, is determined by the `input_format` argument. For the interleaved formats (ending with **I**) only the first channel is used. For the non-interleaved formats, all the channels in the input format are used.

For example, if the user has interleaved the RGB image of size `W x H`, stored continuously, and the pointer to it is `pImage`, then `source` should be:

- `source.channel[0] = pImage`
- `source.pitch[0] = W*3`

When the same image is stored in planar format, with image planes pointers stored continuously in the array `pImage[3]`, then `source` should be:

- `source.channel[0] = pImage[0]`
- `source.channel[1] = pImage[1]`
- `source.channel[2] = pImage[2]`

The `pitch` values for each channel in the `source` parameter should be set accordingly to the data layout.

The nvJPEG library will perform the color transformation to the YCbCr, and will compress the result.

3.1.4. Retrieving the Compressed Stream

Often it is not feasible to accurately predict the final compressed data size of the final JPEG stream for any input data and parameters. The nvJPEG library, while encoding, will calculate the size of the final stream, allocate temporary buffer in the encoder state and save the compressed data in the encoding state's buffer. In order to get final compressed JPEG stream, the user should provide the memory buffer large enough to store this compressed data. There are two options for how to do this:

1. Use the upper bound on compressed JPEG stream size for the given parameters and image dimensions:
   a). Use the `nvjpegEncodeRetrieveBitstream()` function to retrieve the maximum possible JPEG stream size at any given time.
   b). Allocate the memory buffer at any given time.
   c). Encode the image using one of the encoding functions.
   d). Retrieve the compressed JPEG stream from the encoder state after successful encoding, using the `nvjpegEncodeRetrieveBitstream()` and the allocated buffer.
2. Wait for the encoding to complete, and retrieve the exact size of required buffer, as below:
   a). Encode the image using one of the encoding functions.
   b). Use the `nvjpegEncodeRetrieveBitstream()` function to retrieve the size in bytes of the compressed JPEG stream.
   c). Allocate the memory buffer of at least this size.
   d). Use the `nvjpegEncodeRetrieveBitstream()` function to populate your buffer with the compressed JPEG stream.

Note: As the same encoding image state can be reused to compress a series of images, the `nvjpegEncodeRetrieveBitstream()` function will return the result for the last compressed image.

3.1.5. JPEG Encoding Example

See below the example code, and the block diagram shown in Figure 1, for encoding with nvJPEG Encoder.

Figure 1. JPEG Encoding Using nvJPEG Encoder

```
nvjpegHandle_t nv_handle;
nvjpegEncoderState_t nv_enc_state;
nvjpegEncoderParams_t nv_enc_params;
cudaStream_t stream;

// initialize nvjpeg structures
```

```
nvjpegCreateSimple(&nv_handle);
nvjpegEncoderStateCreate(nv_handle, &nv_enc_state, stream);
nvjpegEncoderParamsCreate(nv_handle, &nv_enc_params, stream);

nvjpegImage_t nv_image;
// Fill nv_image with image data, let's say 640x480 image in RGB format

// Compress image
nvjpegEncodeImage(nv_handle, nv_enc_state, nv_enc_params,
    &nv_image, NVJPEG_INPUT_RGB, 640, 480, stream);

// get compressed stream size
size_t length;
nvjpegEncodeRetrieveBitstream(nv_handle, nv_enc_state, NULL, &length, stream);
// get stream itself
cudaStreamSynchronize(stream);
std::vector<char> jpeg(length);
nvjpegEncodeRetrieveBitstream(nv_handle, nv_enc_state, jpeg.data(), &length, 0);

// write stream to file
cudaStreamSynchronize(stream);
std::ofstream output_file("test.jpg", std::ios::out | std::ios::binary);
output_file.write(jpeg.data(), length);
output_file.close();
```

3.2. nvJPEG Encoder Type Declarations

This section describes the nvJPEG Encoder Type Declarations.

3.2.1. nvjpegInputFormat_t

```
typedef enum
{
    NVJPEG_INPUT_RGB       = 3,
    NVJPEG_INPUT_BGR       = 4,
    NVJPEG_INPUT_RGBI      = 5,
    NVJPEG_INPUT_BGRI      = 6
} nvjpegInputFormat_t;
```

The `nvjpegInputFormat_t` enum is used to select the color model and pixel format of the input image. It is used for conversion to YCbCr during encoding.

Member	Description
NVJPEG_INPUT_RGB	Input image is in RGB color model. Pixel format is RGB.
NVJPEG_INPUT_BGR	Input image is in RGB color model. Pixel format is BGR.
NVJPEG_INPUT_RGBI	Input image is in RGB color model. Pixel format is interleaved RGB.
NVJPEG_INPUT_BGRI	Input image is in RGB color model. Pixel format is interleaved BGR.

3.2.2. nvjpegEncoderState_t

The `nvjpegEncoderState_t` is a structure that stores intermediate buffers and variables used for compression.

3.2.3. nvjpegEncoderParams_t

The `nvjpegEncoderParams_t` is a structure that stores JPEG encode parameters.

3.3. nvJPEG Encoder Helper API Reference

The nvJPEG Encoder helper functions are used for initializing.

3.3.1. nvjpegEncoderStateCreate()

Creates encoder state that stores intermediate buffers used in compression.

Signature:

```
nvjpegStatus_t nvjpegEncoderStateCreate(
 nvjpegHandle_t handle,
 nvjpegEncoderState_t *encoder_state,
 cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
`handle`	Input	Host	Library handle
`encoder_state`	Output	Host	Pointer to the encoder state structure, where the new state will be placed.
`stream`	Inputt	Host	CUDA stream where all the required device operations will be placed.

3.3.2. nvjpegEncoderStateDestroy()

Destroys the encoder state.

Signature:

```
nvjpegStatus_t nvjpegEncoderStateDestroy(
 nvjpegEncoderState_t encoder_state);
```

Parameters:

Parameter	Input / Output	Memory	Description
`encoder_state`	Input/Output	Host	Encoder state structure that will be released.

3.3.3. nvjpegEncoderParamsCreate()

Creates the structure that holds the compression parameters.

Signature:

```
nvjpegStatus_t nvjpegEncoderParamsCreate(
 nvjpegHandle_t handle,
 nvjpegEncoderParams_t *encoder_params,
 cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
handle	Input	Host	Library handle
encoder_params	Output	Host	Pointer to the location where the new parameters structure will be placed.
stream	Inputt	Host	CUDA stream where all the required device operations will be placed.

3.3.4. nvjpegEncoderParamsDestroy()

Destroys the encoder parameters structure.

Signature:

```
nvjpegEncoderParamsDestroy(
 nvjpegEncoderParams_t encoder_params);
```

Parameters:

Parameter	Input / Output	Memory	Description
encoder_params	Input/Output	Host	Encoder params structure that will be released.

3.3.5. nvjpegEncoderParamsSetEncoding()

Sets the parameter quality in the encoder parameters structure.

Signature:

```
nvjpegStatus_t nvjpegEncoderParamsSetEncoding(
 nvjpegEncoderParams_t encoder_params,
 nvjpegJpegEncoding_t etype,
 cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
encoder_params	Input/Output	Host	Encoder params structure handle.
etype	Input	Host	Encoding type selection (Baseline/ Progressive). Default is Baseline.
stream	Input	Host	CUDA stream where all the required device operations will be placed.

3.3.6. nvjpegEncoderParamsSetQuality()

Sets the parameter quality in the encoder parameters structure.

Signature:

```
nvjpegStatus_t nvjpegEncoderParamsSetQuality(
 nvjpegEncoderParams_t encoder_params,
 const int quality,
 cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
encoder_params	Input/Output	Host	Encoder params structure handle.
quality	Input	Host	Integer value of quality between 1 and 100, where 100 is the highest quality. Default value is 70.
stream	Inputt	Host	CUDA stream where all the required device operations will be placed.

3.3.7. nvjpegEncoderParamsSetOptimizedHuffman()

Sets whether or not to use optimized Huffman. Using optimized Huffman produces smaller JPEG bitstream sizes with the same quality, but with slower performance.

Signature:

```
nvjpegStatus_t nvjpegEncoderParamsSetOptimizedHuffman(
 nvjpegEncoderParams_t encoder_params,
 const int optimized,
 cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
encoder_params	Input/Output	Host	Encoder params structure handle.
optimized	Input	Host	If this value is 0 then non-optimized Huffman will be used. Otherwise optimized version will be used. Default value is 0.
stream	Inputt	Host	CUDA stream where all the required device operations will be placed.

3.3.8. nvjpegEncoderParamsSetSamplingFactors()

Sets which chroma subsampling will be used for JPEG compression.

Signature:

```
nvjpegStatus_t nvjpegEncoderParamsSetSamplingFactors(
```

```
nvjpegEncoderParams_t encoder_params,
const nvjpegChromaSubsampling_t chroma_subsampling,
cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
`encoder_params`	Input/Output	Host	Encoder params structure handle.
`chroma_subsampling`	Input	Host	Chroma subsampling that will be used for JPEG compression. If the input is in YUV color model and `chroma_subsampling` is different from the subsampling factors of source image, then the NVJPEG library will convert subsampling to the value of `chroma_subsampling`. Default value is 4:4:4.
`stream`	Inputt	Host	CUDA stream where all the required device operations will be placed.

3.4. nvJPEG Encoder API Reference

This section describes the nvJPEG Encoder API.

3.4.1. nvjpegEncodeGetBufferSize()

Returns the maximum possible buffer size that is needed to store the compressed JPEG stream, for the given input parameters.

Signature:

```
nvjpegStatus_t nvjpegEncodeGetBufferSize(
 nvjpegHandle_t handle,
 const nvjpegEncoderParams_t encoder_params,
 int image_width,
 int image_height,
 size_t *max_stream_length);
```

Parameters:

Parameter	Input / Output	Memory	Description
`handle`	Input	Host	Library handle
`encoder_params`	Input/Output	Host	Encoder parameters structure handle.
`image_width`	Input	Host	Input image width.
image_height	Input	Host	Input image height.

stream	Inputt	Host	CUDA stream where all the required device operations will be placed.

3.4.2. nvjpegEncodeYUV()

Compresses the image in YUV colorspace to JPEG stream using the provided parameters, and stores it in the state structure.

Signature:

```
nvjpegStatus_t nvjpegEncodeYUV(
 nvjpegHandle_t handle,
 nvjpegEncoderState_t encoder_state,
 const nvjpegEncoderParams_t encoder_params,
 const nvjpegImage_t *source,
 nvjpegChromaSubsampling_t chroma_subsampling,
 int image_width,
 int image_height,
 cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
handle	Input	Host	Library handle
encoder_state	Input/Output	Host	Internal structure that holds the temporary buffers required for the compression and also stores the final compressed JPEG stream.
encoder_params	Input	Host	Encoder parameters structure handle.
source	Input	Host	Pointer to the nvjpeg structure that holds the device pointers to the Y, U(Cb) and V(Cr) image planes and the respective strides.
chroma_subsampling	Input	Host	Chroma subsampling of the input data.
image_width	Input	Host	Input image width.
image_height	Input	Host	Input image height.
stream	Inputt	Host	CUDA stream where all the required device operations will be placed.

3.4.3. nvjpegEncodeImage()

Compresses the image in the provided format to the JPEG stream using the provided parameters, and stores it in the state structure.

Signature:

```
nvjpegStatus_t nvjpegEncodeImage(
 nvjpegHandle_t handle,
```

```
nvjpegEncoderState_t encoder_state,
const nvjpegEncoderParams_t encoder_params,
const nvjpegImage_t *source,
nvjpegInputFormat_t input_format,
int image_width,
int image_height,
cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
handle	Input	Host	Library handle
encoder_state	Input/Output	Host	Internal structure that holds the temporary buffers required for the compression and also stores the final compressed JPEG stream.
encoder_params	Input	Host	Encoder parameters structure handle.
source	Input	Host	Pointer to the nvjpeg structure that holds the device pointers to the Y, U(Cb) and V(Cr) image planes and the respective strides.
input_format	Input	Host	Value of nvjpegInputFormat_t type that describes the input data.
image_width	Input	Host	Input image width.
image_height	Input	Host	Input image height.
stream	Input	Host	CUDA stream where all the required device operations will be placed.

3.4.4. nvjpegEncodeRetrieveBitstream()

Retrieves the compressed stream from the encoder state that was previously used in one of the encoder functions.

- If data parameter is NULL then the encoder will return compressed stream size in the length parameter.
- If data is not NULL then the provided length parameter should contain the data buffer size.
- If the provided length is less than compressed stream size, then an error will be returned. Otherwise the compressed stream will be stored in the data buffer and the actual compressed buffer size will be stored in the length parameter.

Signature:

```
nvjpegStatus_t nvjpegEncodeRetrieveBitstream(
 nvjpegHandle_t handle,
 nvjpegEncoderState_t encoder_state,
 unsigned char *data,
 size_t *length,
 cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
handle	Input	Host	Library handle
encoder_state	Input/Output	Host	The encoder_state that was previously used in one of the encoder functions.
data	Input/Output	Host	Pointer to the buffer in the host memory where the compressed stream will be stored. Can be NULL (see description).
length	Input/Output	Host	Pointer to the input buffer size. On return the NVJPEG library will store the actual compressed stream size in this parameter.
stream	Inputt	Host	CUDA stream where all the required device operations will be placed.

3.4.5. nvjpegEncodeRetrieveBitstreamDevice()

Retrieves the compressed stream from the encoder state that was previously used in one of the encoder functions.

- data parameter should be on device memory
- If data parameter is NULL then the encoder will return compressed stream size in the length parameter.
- If data is not NULL then the provided length parameter should contain the data buffer size.
- If the provided length is less than compressed stream size, then an error will be returned. Otherwise the compressed stream will be stored in the data buffer and the actual compressed buffer size will be stored in the length parameter.

Signature:

```
nvjpegStatus_t nvjpegEncodeRetrieveBitstreamDevice(
 nvjpegHandle_t handle,
 nvjpegEncoderState_t encoder_state,
 unsigned char *data,
 size_t *length,
 cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
handle	Input	Host	Library handle
encoder_state	Input/Output	Host	The encoder_state that was previously used in one of the encoder functions.

`data`	Input/Output	Device	Pointer to the buffer in the device memory where the compressed stream will be stored. Can be NULL (see description).
`length`	Input/Output	Host	Pointer to the input buffer size. On return the NVJPEG library will store the actual compressed stream size in this parameter.
`stream`	Inputt	Host	CUDA stream where all the required device operations will be placed.

Chapter 4. JPEG Transcoding

This section describes the transcoding functions of the nvJPEG Library.

4.1. nvJPEG Transcoder Helper API Reference

This section describes the nvJPEG Transcoder helper API.

4.1.1. nvjpegEncoderParamsCopyMetadata()

Copies the metadata (JFIF, APP, EXT, and COM markers) from the parsed stream.

Signature:

```
nvjpegStatus_t nvjpegEncoderParamsCopyMetadata(
 nvjpegEncoderState_t encoder_state,
 nvjpegEncoderParams_t encode_params,
 nvjpegJpegStream_t jpeg_stream,
 cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
encoder_state	In/Out	Host	Internal structure that stores the temporary buffers required for the compression.
encode_params	Out	Host	Encoder parameters that will be used for compression.
jpeg_stream	In	Host	Input parsed stream.
stream	In	Host	CUDA stream where all the required device operations will be placed.

4.1.2. nvjpegEncoderParamsCopyQuantizationTables()

Copies the quantization tables from the parsed stream.

Signature:

```
nvjpegStatus_t nvjpegEncoderParamsCopyQuantizationTables(
 nvjpegEncoderParams_t encode_params,
```

```
nvjpegJpegStream_t jpeg_stream,
cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
encode_params	Out	Host	Encoder parameters that will be used for compression.
jpeg_stream	In	Host	Input parsed stream.
stream	In	Host	CUDA stream where all the required device operations will be placed.

4.1.3. nvjpegEncoderParamsCopyHuffmanTables()

Copies the Huffman tables from the parsed stream.

Signature:

```
nvjpegStatus_t nvjpegEncoderParamsCopyHuffmanTables(
 nvjpegEncoderState_t encoder_state,
 nvjpegEncoderParams_t encode_params,
 nvjpegJpegStream_t jpeg_stream,
 cudaStream_t stream);
```

Parameters:

Parameter	Input / Output	Memory	Description
encoder_state	In/Out	Host	Internal structure that stores the temporary buffers required for the compression.
encode_params	Out	Host	Encoder parameters that will be used for compression.
jpeg_stream	In	Host	Input parsed stream.
stream	In	Host	CUDA stream where all the required device operations will be placed.

4.2. JPEG Transcoding Example

See below the example code.

```
cudaStream_t stream;
// create library handle
nvjpegHandle_t handle;
nvjpegCreateSimple(&handle);

//////////////////////////////////// nvJPEG
 decoding ////////////////////////////////////////
// create bitstream object
nvjpegJpegStream_t jpeg_stream;
nvjpegJpegStreamCreate(handle, &jpeg_stream);

// parse jpeg stream
nvjpegJpegStreamParse(handle,
    data_ptr,
```

```
    data_size,
    1,  // save metadata in the jpegStream structure
    0,
    jpeg_stream);

// create decoder and decoder state
nvjpegJpegDecoder_t decoder;
nvjpegJpegState_t decoder_state;
nvjpegDecoderCreate(handle, NVJPEG_BACKEND_DEFAULT, &decoder);
nvjpegDecoderStateCreate(handle, decoder, &decoder_state);

// create and set up decoder parameters
nvjpegDecodeParams_t decode_params;
nvjpegDecodeParamsCreate(handle, &decode_params);
nvjpegDecodeParamsSetOutputFormat(decode_params, NVJPEG_OUTPUT_RGBI);

// decode image
nvjpegImage_t output_image;
nvjpegDecodeJpeg(handle, decoder, decode_params, jpeg_stream, decoder_state,
 &output_image, stream);

////////////////////////////////// nvJPEG Transcode and encode
 API ////////////////////////////////
nvjpegEncoderState_t encoder_state;
nvjpegEncoderParams_t encode_params;

// get encoding from the jpeg stream and copy it to the encode parameters
nvjpegJpegEncoding_t jpeg_encoding;
nvjpegJpegStreamGetJpegEncoding(jpeg_stream, &jpeg_encoding);
nvjpegEncoderParamsSetEncoding(encode_params, jpeg_encoding);

// copies according data to the encode parameters
nvjpegEncoderParamsCopyMetadata(encode_params, jpeg_stream, stream);
nvjpegEncoderParamsCopyQuantizationTables(encode_params, jpeg_stream, stream);
nvjpegEncoderParamsCopyHuffmanTables(encode_params, jpeg_stream, stream);

// retrieve frame dimensions
unsigned width, height;
nvjpegJpegStreamGetFrameDimensions(jpeg_stream, &width, &height);

// encode using encode parameters
nvjpegEncodeImage(nvjpeg_handle, encoder_state, encode_params, &output_image,
    input_format, width, height, stream);

// get compressed stream size
size_t length;
nvjpegEncodeRetrieveBitstream(nvjpeg_handle, encoder_state, NULL, &length, stream);
// get stream itself
cudaStreamSynchronize(stream);
std::vector<char> jpeg(length);
nvjpegEncodeRetrieveBitstream(nvjpeg_handle, encoder_state, jpeg.data(), &length,
 0);
```

Chapter 5. List of Dropped APIs

The following APIs are dropped starting CUDA 11.0

```
nvjpegStatus_t nvjpegDecodePhaseOne(
 nvjpegHandle_t          handle,
 nvjpegJpegState_t       jpeg_handle,
 const unsigned char     *data,
 size_t                  length,
 nvjpegOutputFormat_t    output_format,
 cudaStream_t            stream);

nvjpegStatus_t nvjpegDecodePhaseTwo(
 nvjpegHandle_t       handle,
 nvjpegJpegState_t    jpeg_handle,
 cudaStream_t         stream);

nvjpegStatus_t nvjpegDecodePhaseThree(
 nvjpegHandle_t       handle,
 nvjpegJpegState_t    jpeg_handle,
 nvjpegImage_t        *destination,
 cudaStream_t         stream);

nvjpegStatus_t nvjpegDecodeBatchedPhaseOne(
 nvjpegHandle_t          handle,
 nvjpegJpegState_t       jpeg_handle,
 const unsigned char     *data,
 size_t                  length,
 int                     image_idx,
 int                     thread_idx,
 cudaStream_t            stream);

nvjpegStatus_t nvjpegDecodeBatchedPhaseTwo(
 nvjpegHandle_t          handle,
 nvjpegJpegState_t       jpeg_handle,
 cudaStream_t            stream);

nvjpegStatus_t nvjpegDecodeBatchedPhaseThree(
 nvjpegHandle_t            handle,
 nvjpegJpegState_t         jpeg_handle,
 nvjpegImage_t             *destinations,
 cudaStream_t              stream);
```

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 acknowledgement, 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.

VESA DisplayPort

DisplayPort and DisplayPort Compliance Logo, DisplayPort Compliance Logo for Dual-mode Sources, and DisplayPort Compliance Logo for Active Cables are trademarks owned by the Video Electronics Standards Association in the United States and other countries.

HDMI

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

OpenCL

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

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

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