nvJPEG Library 0.1

Developer Documentation

June 2018

**Note:** For a list of the package contents of the nvJPEG library 0.1, and the instructions for compiling, see the end of this document.
1. Introduction

2. Using the nvJPEG API
   The nvJPEG library provides APIs for both the decoding of a single image and batched decoding of multiple images.
   2.1. Single Image Decoding
       2.1.1. Decode by Phases
   2.2. Batched Image Decoding
       2.2.1. Single Phase
       2.2.2. Multiple Phases

3. nvJPEG Type Declarations
   3.1. nvJPEG Memory Allocator Interface
   3.2. nvJPEG Opaque Library Handle Struct
   3.3. nvJPEG Opaque JPEG Decoding State Handle
   3.4. nvJPEG Output Pointer Struct

4. nvJPEG API Reference
   4.1. nvJPEG Helper API Reference
       4.1.1. nvjpegGetProperty()
       4.1.2. nvjpegCreate()
       4.1.3. nvjpegDestroy()
       4.1.4. nvjpegJpegStateCreate()
       4.1.5. nvjpegJpegStateDestroy()
   4.2. Retrieve Encoded Image Information API
       4.2.1. nvjpegGetImageInfo()
   4.3. Decode API -- Single Phase
       4.3.1. nvjpegDecode()
       4.3.2. nvjpegDecodeBatchedInitialize()
       4.3.3. nvjpegDecodeBatched()
   4.4. Decode API -- Multiple Phases
       4.4.1. nvjpegDecodePhaseOne()
       4.4.2. nvjpegDecodePhaseTwo()
       4.4.3. nvjpegDecodePhaseThree()
       4.4.4. nvjpegDecodeBatchedPhaseOne()
       4.4.5. nvjpegDecodeBatchedPhaseTwo()
       4.4.6. nvjpegDecodeBatchedPhaseThree()
   4.5. nvJPEG API Return Codes
   4.6. nvJPEG Chroma Subsampling
   4.7. Reference Documents
5. Package Contents and Instructions

5.1 Package Contents
5.2 Using the Library
5.3 Example
1. Introduction

The nvJPEG 0.1 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.

Note: Throughout this document, the terms “CPU” and “Host” are used synonymously. Similarly, the terms “GPU” and “Device” are synonymous.

The nvJPEG decoder library supports the following:

**JPEG options:**
- Baseline and Progressive JPEG decoding.
- 8 bits per sample.
- Huffman encoded streams.
- 3 color channels (YCbCr) or 1 color channel (Grayscale).
- 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, and
  - 4:1:0
- 8- and 16-bit quantization tables.

**Features:**
- Hybrid decoding using both the CPU (i.e., host) and the GPU (i.e., device).
- Input to the library is in the host memory, and the output is in the GPU memory.
- Single image and batched image decoding.
- Decode in a single phase and in multiple phases.
- Color space conversion.
- User-provided memory manager for the device allocations.
2. Using the nvJPEG API

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

2.1. Single Image Decoding

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

- To use the nvJPEG library, start by first calling the helper APIs for initialization.
- Create nvJPEG library handle with the helper API `nvjpegCreate()`.
- Create JPEG state with the helper API `nvjpegJpegStateCreate()`. See the nvJPEG Type Declarations and the the `nvjpegJpegStateCreate()` in the nvJPEG Helper API reference.

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

- `nvjpegStatus_t nvjpegCreate(nvjpegHandle_t *handle, nvjpeg_dev_allocator allocator);`
- `nvjpegStatus_t nvjpegDestroy(nvjpegHandle_t handle);`
- `nvjpegStatus_t nvjpegJpegStateCreate(nvjpegHandle_t handle, nvjpegJpegState_t *jpeg_handle);`
- `nvjpegStatus_t nvjpegJpegStateDestroy(nvjpegJpegState handle);`

- Retrieve the width and height information from the JPEG encoded image by using the `nvjpegGetImageInfo()` method. See also the `nvjpegGetImageInfo()` API reference.
Below is the signature of `nvjpegGetImageInfo()` method:

```c
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 method. This `nvjpegGetImageInfo()` method is thread safe.

- Note that one of the outputs of the above `nvjpegGetImageInfo()` method is `nvjpegChromaSubsampling_t`. This parameter is an enum type, and its enumerator list comprises of the chroma subsampling property retrieved from the JPEG image. See the [nvJPEG Chroma Subsampling](nvJPEG API Reference).

- Next, use the `nvjpegDecode()` method in the nvJPEG library to decode this single JPEG image. See the signature of this method below:

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

- **The `nvjpegOutputFormat_t` parameter**

  The `nvjpegOutputFormat_t` parameter can be set to one of the below output_format settings:
<table>
<thead>
<tr>
<th>output_format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVJPEG_OUTPUT_UNCHANGED</td>
<td>Return the decoded image planar format</td>
</tr>
<tr>
<td>NVJPEG_OUTPUT_RGB</td>
<td>Convert to planar RGB</td>
</tr>
<tr>
<td>NVJPEG_OUTPUT_BGR</td>
<td>Convert to planar BGR</td>
</tr>
<tr>
<td>NVJPEG_OUTPUT_RGBI</td>
<td>Convert to interleaved RGB</td>
</tr>
<tr>
<td>NVJPEG_OUTPUT_BGRI</td>
<td>Convert to interleaved BGR</td>
</tr>
<tr>
<td>NVJPEG_OUTPUT_Y</td>
<td>Return the Y component only</td>
</tr>
<tr>
<td>NVJPEG_OUTPUT_YUV</td>
<td>Return in the YUV planar format</td>
</tr>
</tbody>
</table>

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 `nvjpegImage_t` destination structure.

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

- As mentioned above, an important benefit of the `nvjpegGetImageInfo()` method is you can utilize the image information the method retrieves from the the input JPEG image to allocate proper GPU memory for your decoding operation.

The `nvjpegGetImageInfo()` method, as shown below, returns the widths, heights and `nComponents` parameters. See `nvjpegGetImageInfo()`.

```c
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 these 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.

Refer to the below table to optimally set the 'destination' parameter for the nvjpegDecode() method:

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

- 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 below:

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

The NVJPEG_MAX_COMPONENT is the maximum number of color components the JPEG library supports in the current release. For the generic images, this is the maximum number of encoded channels.
Finally, when you call the `nvjpegDecode()` method with the parameters as discussed above, the `nvjpegDecode()` method will fill the output buffers with the decoded data.

2.1.1. Decode by Phases

Alternately, you can decode a single image in multiple phases. This gives you flexibility in controlling the flow, and to optimize the decoding process.

Follow the below sequence of API calls:

- Just as in decode in a single phase, create the JPEG state with the helper API `nvjpegJpegStateCreate()`.
- Next, call the APIs in this below sequence (see the Decode API -- Multiple Phases reference):
  - `nvjpegDecodePhaseOne()`
  - `nvjpegDecodePhaseTwo()`
  - `nvjpegDecodePhaseThree()`

At the conclusion of the third phase, the `nvjpegDecodePhaseThree()` method will write the decoded output at the memory location pointed to by its *destination parameter.

2.2. 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.

2.2.1. Single Phase

For the batched image decoding in single phase, the below applies:

- You call the `nvjpegDecodeBatchedInitialize()` API to initialize the batched decoder. Specify the batch size in the batch_size parameter of this API. See API Reference for `nvjpegDecodeBatchedInitialize()`.
Next, call the `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()` API again.

### 2.2.2. Multiple Phases

To decode a batch of images in multiple phases, follow these below steps:

**Note:** This is the only case where the JPEG state could be used by the multiple threads at the same time.

1. Create the JPEG state with the helper API `nvjpegJpegStateCreate()`.
2. Call the `nvjpegDecodeBatchedInitialize()` API to initialize the batched decoder. Specify the batch size in the `batch_size` parameter of this API, and specify the `max_cpu_threads` parameter to set the maximum number of CPU threads that work on single batch.
3. Batched processing is done by calling the APIs for the specific phases in an order:
   a. In the first phase, call `nvjpegDecodeBatchedPhaseOne()` for each image in the batch, according to the index of the image in the batch. Note that this could be done using multiple threads. If multiple threads are used then the thread index in the range [0, max_cpu_threads-1] should be provided to the `nvjpegDecodeBatchedPhaseOne()` function. Before proceeding to the next phase you need to ensure that the `nvjpegDecodeBatchedPhaseOne()` calls for every image have finished.
   b. Next, call `nvjpegDecodeBatchedPhaseTwo()`
   c. Finally, call `nvjpegDecodeBatchedPhaseThree()`
4. If you have another batch of images of the same size to process, then repeat from 3.

### 3. nvJPEG Type Declarations

#### 3.1. nvJPEG Memory Allocator Interface

```c
typedef int (*tDevMalloc)(void**, size_t);
typedef int (*tDevFree)(void*);
typedef struct
```
tDevMalloc dev_malloc;
tDevFree dev_free;
} nvjpegDevAllocator_t;

When the ‘nvjpegDevAllocator_t *allocator’ parameter in the nvjpegCreate() function is set as a pointer to the above nvjpegDevAllocator_t structure, then this structure will be used for allocating and releasing 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() function is set to NULL, then the default memory allocation functions cudaMalloc() and cudaFree() will be used.

### 3.2. nvJPEG Opaque Library Handle Struct

```c
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 could be used by multiple threads simultaneously.

### 3.3. nvJPEG Opaque JPEG Decoding State Handle

```c
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).
3.4. nvJPEG Output Pointer Struct

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

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

See the section [Single Image Decoding](#) on how to set up the `nvjpegImage_t` struct.

4. nvJPEG API Reference

This section describes the nvJPEG API Reference.

4.1. nvJPEG Helper API Reference

The nvJPEG helper APIs are used for initializing.

4.1.1. nvjpegGetProperty()

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

**Signature:**

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

**Parameters:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Input / Output</th>
<th>Memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libraryPropertyType type</td>
<td>Input</td>
<td>Host</td>
<td>One of the supported libraryPropertyType values, i.e. MAJOR_VERSION, MINOR_VERSION or PATCH_LEVEL</td>
</tr>
<tr>
<td>int *value</td>
<td>Output</td>
<td>Host</td>
<td>The numeric value</td>
</tr>
</tbody>
</table>
corresponding to the specific libraryPropertyType requested.

**Returns:**

nvjpegStatus_t - An error code as specified in Section 4.5. nvJPEG API Return Codes.

### 4.1.2. nvjpegCreate()

Allocates and initializes the library handle.

**Signature:**

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

**Parameters:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Input / Output</th>
<th>Memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvjpegBackend_t backend</td>
<td>Input</td>
<td>Host</td>
<td>A backend parameter for the library. If this is set to &quot;DEFAULT&quot; then it will choose one of underlying algorithm automatically.</td>
</tr>
<tr>
<td>nvjpegDevAllocator_t *allocator</td>
<td>Input</td>
<td>Host</td>
<td>Device memory allocator. See nvjpegDevAllocator_t structure description. If NULL is provided, then the default CUDA runtime cudaMalloc() and cudaFree() functions will be used.</td>
</tr>
<tr>
<td>nvjpegHandle_t *handle</td>
<td>Input/Output</td>
<td>Host</td>
<td>The library handle</td>
</tr>
</tbody>
</table>

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

```c
typedef enum
{
    NVJPEG_BACKEND_DEFAULT = 0,
    NVJPEG_BACKEND_HYBRID  = 1,
} nvjpegBackend_t;
```
4.1.3. nvjpegDestroy()

Releases the library handle.

**Signature:**

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

**Parameters:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Input / Output</th>
<th>Memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvjpegHandle_t handle</td>
<td>Input/Output</td>
<td>Host</td>
<td>The library handle that should be released.</td>
</tr>
</tbody>
</table>

4.1.4. nvjpegJpegStateCreate()

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

**Signature:**

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

**Parameters:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Input / Output</th>
<th>Memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvjpegHandle_t handle</td>
<td>Input</td>
<td>Host</td>
<td>The library handle</td>
</tr>
<tr>
<td>nvjpegJpegState_t *jpeg_handle</td>
<td>Input/Output</td>
<td>Host</td>
<td>The image state handle</td>
</tr>
</tbody>
</table>
Returns:

nvjpegStatus_t - An error code as specified in Section 4.5. nvJPEG API Return Codes.

4.1.5. nvjpegJpegStateDestroy()

Releases the image internal structure.

Signature:

nvjpegStatus_t nvjpegJpegStateDestroy(nvjpegJpegState handle);

Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Input / Output</th>
<th>Memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvjpegJpegState handle</td>
<td>Input/Output</td>
<td>Host</td>
<td>The image state handle.</td>
</tr>
</tbody>
</table>

Returns:

nvjpegStatus_t - An error code as specified in Section 4.5. nvJPEG API Return Codes.

4.2. Retrieve Encoded Image Information API

The helper API for retrieving the encoded image information.

4.2.1. 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:
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Input / Output</th>
<th>Memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvjpegHandle_t handle</td>
<td>Input</td>
<td>Host</td>
<td>The library handle.</td>
</tr>
<tr>
<td>const unsigned char *data</td>
<td>Input</td>
<td>Host</td>
<td>Pointer to the encoded data.</td>
</tr>
<tr>
<td>size_t length</td>
<td>Input</td>
<td>Host</td>
<td>Size of the encoded data in bytes.</td>
</tr>
<tr>
<td>int *nComponents</td>
<td>Output</td>
<td>Host</td>
<td>Number of encoded channels in the input.</td>
</tr>
<tr>
<td>nvjpegChromaSubsampling_t *subsampling</td>
<td>Output</td>
<td>Host</td>
<td>Chroma subsampling for the 1- or 3- channel encoding.</td>
</tr>
<tr>
<td>int *widths</td>
<td>Output</td>
<td>Host</td>
<td>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.</td>
</tr>
<tr>
<td>int *heights</td>
<td>Output</td>
<td>Host</td>
<td>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.</td>
</tr>
</tbody>
</table>
Returns:

nvjpegStatus_t - An error code as specified in Section 4.5. nvJPEG API Return Codes.

4.3. Decode API -- Single Phase

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

4.3.1. nvjpegDecode()

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

Signature:

```c
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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Input / Output</th>
<th>Memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvjpegHandle_t handle</td>
<td>Input</td>
<td>Host</td>
<td>The library handle.</td>
</tr>
<tr>
<td>nvjpegJpegState_t jpeg_handle</td>
<td>Input</td>
<td>Host</td>
<td>The image state handle.</td>
</tr>
<tr>
<td>const unsigned char *data</td>
<td>Input</td>
<td>Host</td>
<td>Pointer to the encoded stream.</td>
</tr>
<tr>
<td>size_t length</td>
<td>Input</td>
<td>Host</td>
<td>Size of the encoded stream.</td>
</tr>
</tbody>
</table>
4.3.2. 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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Input / Output</th>
<th>Memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvjpegHandle_t handle</td>
<td>Input</td>
<td>Host</td>
<td>The library handle.</td>
</tr>
<tr>
<td>nvjpegJpegState_t jpeg_handle</td>
<td>Input</td>
<td>Host</td>
<td>The image state handle.</td>
</tr>
<tr>
<td>int batch_size</td>
<td>Input</td>
<td>Host</td>
<td>Batch size</td>
</tr>
<tr>
<td>int max_cpu_threads</td>
<td>Input</td>
<td>Host</td>
<td>Maximum number of CPU threads that can participate in decoding a batch.</td>
</tr>
<tr>
<td>nvjpegOutputFormat_t output_format</td>
<td>Input</td>
<td>Host</td>
<td>The format in which decoded image will be saved.</td>
</tr>
</tbody>
</table>

Returns:

nvjpegStatus_t - An error code as specified in Section 4.5. nvJPEG API Return Codes.

4.3.3. 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 method is asynchronous with respect to the host. All GPU tasks for this method will be submitted to the provided stream.

Signature:

```c
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:
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Input / Output</th>
<th>Memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvjpegHandle_t handle</td>
<td>Input</td>
<td>Host</td>
<td>The library handle.</td>
</tr>
<tr>
<td>nvjpegJpegState_t jpeg_handle</td>
<td>Input</td>
<td>Host</td>
<td>The image state handle.</td>
</tr>
<tr>
<td>const unsigned char *const *data</td>
<td>Input</td>
<td>Host</td>
<td>Pointer to the first element of array of the input data. The size of the array is assumed to be batch_size provided to nvjpegDecodeBatch edInitialize(), the batch initialization function.</td>
</tr>
<tr>
<td>const size_t *lengths</td>
<td>Input</td>
<td>Host</td>
<td>Pointer to the first element of array of input sizes. Size of array is assumed to be batch_size provided to nvjpegDecodeBatch edInitialize(), the batch initialization function.</td>
</tr>
<tr>
<td>nvjpegImage_t *destinations</td>
<td>Input/Output</td>
<td>Host/Device</td>
<td>Pointer to the first element of array of output descriptors. The size of array is assumed to be batch_size provided to nvjpegDecodeBatch edInitialize(), the batch initialization function. See also nvjpegImage_t description.</td>
</tr>
<tr>
<td>cudaStream_t stream</td>
<td>Input</td>
<td>Host</td>
<td>The CUDA stream where all the GPU work will be performed.</td>
</tr>
</tbody>
</table>
Returns:

nvjpegStatus_t - An error code as specified in Section 4.5. nvJPEG API Return Codes.

4.4. Decode API -- Multiple Phases

The nvJPEG library provides an ability to control the decoding process in phases. In the simple case of a single image decode you can split the decoding into phases. For decoding multiple images, on the other hand, you can overlap the decoding phases of separate images within a single thread.

Finally, for the batched decode you can use multiple threads to split the host tasks. Synchronization between phases should be handled with CUDA events and CUDA stream synchronization mechanisms, by the user.

Note that first phases are synchronous with the respect to the host and second and third one - asynchronous for both single image and batched decode.

4.4.1. nvjpegDecodePhaseOne()

The first phase of a single-image decode. You should provide all the inputs, and the nvJPEG library performs any required preprocessing on the host. Any previous calls to nvjpegDecodePhaseOne() and nvjpegDecodePhaseTwo() with the same 'nvjpeg_handle' parameter should be finished prior to this call.

Signature:

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

Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Input / Memory</th>
<th>Description</th>
</tr>
</thead>
</table>

21
Returns:

nvjpegStatus_t - An error code as specified in Section 4.5. nvJPEG API Return Codes.

4.4.2. nvjpegDecodePhaseTwo()

In this second phase of decoding process, the GPU (i.e., device) is involved. The decoding task is transferred to the device memory. Any required preprocessing will be performed on the device. Any previous calls to nvjpegDecodePhaseTwo() and nvjpegDecodePhaseThree() with the same jpeg_handle parameter should be finished prior to this call.

Signature:

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

Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Input / Output</th>
<th>Memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvjpegHandle_t handle</td>
<td>Input</td>
<td>Host</td>
<td>The library handle.</td>
</tr>
<tr>
<td>nvjpegJpegState_t jpeg_handle</td>
<td>Input</td>
<td>Host</td>
<td>The image state handle.</td>
</tr>
<tr>
<td>const unsigned char *data</td>
<td>Input</td>
<td>Host</td>
<td>Pointer to the encoded stream.</td>
</tr>
<tr>
<td>size_t length</td>
<td>Input</td>
<td>Host</td>
<td>Size of the encoded stream.</td>
</tr>
<tr>
<td>nvjpegOutputFormat_t output_format</td>
<td>Input</td>
<td>Host</td>
<td>Format in which the decoded image will be saved.</td>
</tr>
<tr>
<td>cudaStream_t stream</td>
<td>Input</td>
<td>Host</td>
<td>The CUDA stream where all the GPU work will be submitted.</td>
</tr>
</tbody>
</table>
4.4.3. nvjpegDecodePhaseThree()

In this third phase of the decoding process, the decoded image will be written to the output, in the specified decoding format.

**Note:** If the same jpeg_handle is shared for decoding multiple images simultaneously, then these multiple images should be of the same output_format.

**Signature:**

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

**Parameters:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Input / Output</th>
<th>Memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvjpegHandle_t handle</td>
<td>Input</td>
<td>Host</td>
<td>The library handle.</td>
</tr>
<tr>
<td>nvjpegJpegState_t jpeg_handle</td>
<td>Input</td>
<td>Host</td>
<td>The image state handle.</td>
</tr>
<tr>
<td>nvjpegImage_t *destination</td>
<td>Input/Output</td>
<td>Host/Device</td>
<td>Pointer to the structure that describes the output</td>
</tr>
</tbody>
</table>
destination. This structure should be on host, but the pointers in this structure should be pointing to the device memory. See nvjpegImage_t description for details.

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 Section 4.5. nvJPEG API Return Codes.

4.4.4. nvjpegDecodeBatchedPhaseOne()

This first phase of the batched decoding should be called separately for each image in the batch. The batch initialization API, with appropriate batch parameters, should be called prior to starting the task with the batch.

If the batch parameters (batch size, number of threads, output format) did not change, then there is no need to initialize the batch again before starting the task.

It is possible to use multiple threads to split this first phase of the task. In which case, each thread should have a unique index. Provide the index of the image in the batch, and use the same JPEG decoding state parameter.

The thread index for the batch should be in the range of [0, max_cpu_threads-1]. The image index should be in the range of [0, batch_size-1].

Any previous calls to nvjpegDecodeBatchedPhaseOne() and nvjpegDecodeBatchedPhaseTwo() on a different batch with the same JPEG state handle parameter should be completed prior to this call.

Signature:

```
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);

Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Input / Output</th>
<th>Memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvjpegHandle_t handle</td>
<td>Input</td>
<td>Host</td>
<td>The library handle.</td>
</tr>
<tr>
<td>nvjpegJpegState_t jpeg_handle</td>
<td>Input</td>
<td>Host</td>
<td>The image state handle.</td>
</tr>
<tr>
<td>const unsigned char *data</td>
<td>Input</td>
<td>Host</td>
<td>Pointer to the encoded stream.</td>
</tr>
<tr>
<td>size_t length</td>
<td>Input</td>
<td>Host</td>
<td>Size of the encoded stream.</td>
</tr>
<tr>
<td>int image_idx</td>
<td>Input</td>
<td>Host</td>
<td>Image index in the batch. Should be in the range from 0 to batch_size-1.</td>
</tr>
<tr>
<td>int thread_idx</td>
<td>Input</td>
<td>Host</td>
<td>Thread index that calls this phase. Should be in the range from 0 to max_cpu_threads-1</td>
</tr>
<tr>
<td>cudaStream_t stream</td>
<td>Input</td>
<td>Host</td>
<td>The CUDA stream where all the GPU work will be submitted.</td>
</tr>
</tbody>
</table>

Returns:

nvjpegStatus_t - An error code as specified in Section 4.5. nvJPEG API Return Codes.

4.4.5. nvjpegDecodeBatchedPhaseTwo()
This phase should be called once per batch. It should be called only after the
nvjpegDecodeBatchedPhaseOne() calls for every image in the batch have finished. Any prior
calls to nvjpeg DecodeBatchedPhaseTwo() and nvjpegDecodeBatchedPhaseThree() for other
batches with the same JPEG state handle parameter should be finished prior this call.

**Signature:**

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

**Parameters:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Input / Output</th>
<th>Memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvjpegHandle_t handle</td>
<td>Input</td>
<td>Host</td>
<td>The library handle.</td>
</tr>
<tr>
<td>nvjpegJpegState_t jpeg_handle</td>
<td>Input</td>
<td>Host</td>
<td>The image state handle.</td>
</tr>
<tr>
<td>cudaStream_t stream</td>
<td>Input</td>
<td>Host</td>
<td>The CUDA stream where all the GPU work will be submitted.</td>
</tr>
</tbody>
</table>

**Returns:**

nvjpegStatus_t - An error code as specified in [Section 4.5. nvJPEG API Return Codes](#).

### 4.4.6. nvjpegDecodeBatchedPhaseThree()

This phase should be called once per batch. It should be called only after the
nvjpegDecodeBatchedPhaseTwo() call for the same batch has finished.

Between a call to nvjpegDecodeBatchedPhaseTwo() and a call to this method, no calls are
allowed to nvjpegDecodeBatchedPhaseTwo() or nvjpegDecodeBatchedPhaseThree() for any
other batch with the same JPEG state handle parameter.

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

Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Input / Output</th>
<th>Memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvjpegHandle_t handle</td>
<td>Input</td>
<td>Host</td>
<td>The library handle.</td>
</tr>
<tr>
<td>nvjpegJpegState_t jpeg_handle</td>
<td>Input</td>
<td>Host</td>
<td>The image state handle.</td>
</tr>
<tr>
<td>nvjpegImage_t *destinations</td>
<td>Input/Output</td>
<td>Host/Device</td>
<td>Pointer to the first element of the array of output descriptors. The size of the array is assumed to be the “batch_size” parameter that was provided to the batch initialization function. See nvjpegImage_t description for details.</td>
</tr>
<tr>
<td>cudaStream_t stream</td>
<td>Input</td>
<td>Host</td>
<td>The CUDA stream to which all the GPU tasks will be submitted.</td>
</tr>
</tbody>
</table>

Returns:

nvjpegStatus_t - An error code as specified in Section 4.5. nvJPEG API Return Codes.

4.5. 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,
};

Description of the returned error codes:

<table>
<thead>
<tr>
<th>Returned Error (Returned Code)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVJPEG_STATUS_SUCCESS (0)</td>
<td>The API call has finished successfully. Note that many of the calls are asynchronous and some of the errors might be seen only after synchronization.</td>
</tr>
<tr>
<td>NVJPEG_STATUS_NOT_INITIALIZED (1)</td>
<td>The library handle was not initialized. A call to nvjpegCreate() is required to initialize the handle.</td>
</tr>
<tr>
<td>NVJPEG_STATUS_INVALID_PARAMETER (2)</td>
<td>Wrong parameter is passed to the API. For example, a null pointer as input data or image index is not in the allowed range.</td>
</tr>
<tr>
<td>NVJPEG_STATUS_BAD_JPEG (3)</td>
<td>Cannot parse the JPEG stream. Check that the encoded JPEG stream and its size parameters are correct.</td>
</tr>
<tr>
<td>NVJPEG_STATUS_JPEG_NOT_SUPPORTED (4)</td>
<td>Attempting to decode a JPEG stream that is not supported by the nvJPEG library.</td>
</tr>
<tr>
<td>NVJPEG_STATUS_ALLOCATOR_FAILURE (5)</td>
<td>The user-provided allocator functions, for either memory allocation or for releasing the memory, returned a non-zero code.</td>
</tr>
<tr>
<td>NVJPEG_STATUS_EXECUTION_FAILED (6)</td>
<td>Error during the execution of the device tasks.</td>
</tr>
<tr>
<td>NVJPEG_STATUS_ARCH_MISMATCH (7)</td>
<td>The device capabilities are not enough for the set of input parameters provided (input parameters such as i.e. backend, encoded stream parameters, output format).</td>
</tr>
</tbody>
</table>
4.6. nvJPEG Chroma Subsampling

One of the outputs of the \texttt{nvjpegGetImageInfo()} API is \texttt{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 \texttt{nvjpegGetImageInfo()} method currently supports:

```c
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;
```

4.7. Reference Documents

Refer to the JPEG standard: \url{https://jpeg.org/jpeg/}

5. Package Contents and Instructions

5.1 Package Contents

cuda-linux64-nvjpeg/doc/nvJPEG Library 0.1 Documentation.pdf
cuda-linux64-nvjpeg/include/nvjpeg.h
cuda-linux64-nvjpeg/lib64/libnvjpeg_static.a
5.2 Using the Library

The library in this package is compiled and linked against the CUDA 9.0 runtime libraries.

This package contains the library header and a set of libraries - static and shared. Shared libraries (libnvjpeg.so and the respective versioned libraries) have all of the CUDA toolkit dependencies statically linked. However, if you want to link against the static library (libnvjpeg.a) you also need to link the other dependencies from the CUDA Toolkit 9.0 library - namely the NPP libraries, the CUDA runtime library (libcudart.a), and the CUDA tools library (libculibos.a).

Example of linking shared library:
```bash
  g++ -Icuda-linux64-nvjpeg/include -lnvjpeg
  -Lcuda-linux64-nvjpeg/lib64 my_example.cpp -o my_example
```

Example of linking static library, considering CUDA 9.0 is installed by the path:
```bash
/usr/local/cuda-9.0:
  g++ -Icuda-linux64-nvjpeg/include -lnvjpeg_static
  -Lcuda-linux64-nvjpeg/lib64 -lcudart_static -lculibos
  -lnppicom_static -L/usr/local/cuda-9.0/lib64 my_example.cpp -o
  my_example
```

5.3 Example

Below example shows how to use the various nvJPEG APIs. Compile with the following command from the examples folder, considering CUDA 9.0 is installed by the path
```bash
/usr/local/cuda-9.0:
```
This below example shows how to decode the JPEG files using either single or batched API, and write the decoded files as BMP images:

To decode a single image:

```
./nvjpeg_example -i /tmp/my_image.jpg -fmt rgb -o /tmp
```

To decode multiple images in the folder using the batched API in separate phases:

```
./nvjpeg_example -i /tmp/my_images/ -fmt rgb -b 32 -pipelined -batched -o /tmp
```

Run `"./nvjpeg_example -h"` for the description of the parameters.