



Denglin Hamming™ V2

dJPEG Programming Guide

DL-DG/SW-034A-01

2023-4-30

Copyright©苏州登临科技有限公司，2019 - 2025，版权所有。

未经苏州登临科技有限公司事先书面同意，不得以任何形式或方式复制或传播本文件的任何部分。

商标和许可



和其它苏州登临科技有限公司的其它登临科技的图标为苏州登临科技有限公司的商标。本手册中提及的所有其他商标均为其各自所有者的财产。

通知

所购买的产品、服务和特性由苏州登临科技有限公司与客户签订的合同规定。本文件中描述的所有或部分产品、服务和特性可能不在采购范围或使用范围内。除非合同中另有规定，本文件中的所有声明、信息和建议均按“原样”提供，无任何明示或暗示的保证或陈述。

本手册中的信息如有更改，恕不另行通知。本文件在编制过程中已尽一切努力确保内容的准确性，本文件中的所有声明、信息和建议不构成任何明示或暗示的保证。

苏州登临科技有限公司

苏州工业园区扬富路11号南岸新地一期商务楼5号1101室，江苏，中国

<http://www.denglin.ai>

Email : support@denglin.ai

Change History

Version	Change description
01	Initial version.

Table of Contents

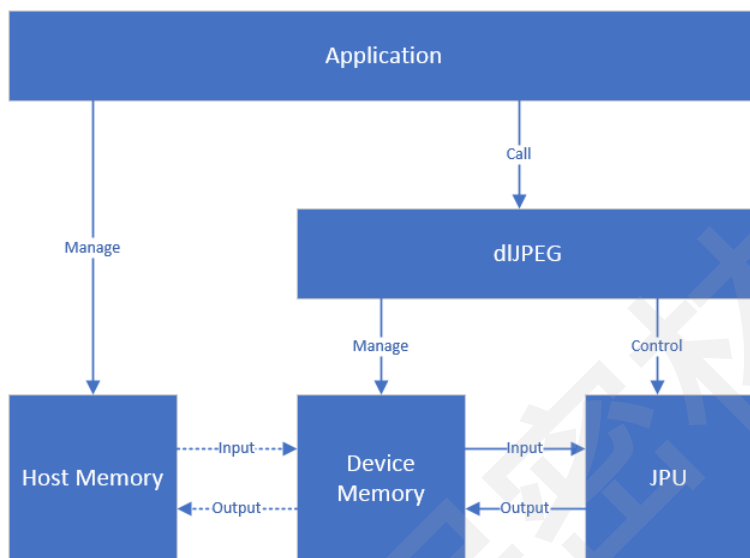
Table of Contents

- 1. Overview
- 2. Main Codec Feature
- 3. Work Flow
 - 3.1 Encode Overview
 - 3.2 Encode Flow
 - 3.3 Decode Overview
 - 3.4 Decode Flow
- 4. Using dJPEG APIs
 - 4.1 Getting a JPEG Device
 - 4.2 Getting session count
 - 4.3 Creating a JPEG Session
 - 4.4 Parsing JPEG Header (Decode Only)
 - 4.5 Device Memory
 - 4.6 Decode
 - 4.7 Async Decode
 - 4.8 Encode
 - 4.9 Creating a JPEG Header (Encode Only)
 - 4.10 Custom Huffman Table (Encode Only)
 - 4.11 Custom Quantization table (Encode Only)
 - 4.12 Cleanup
- 5. dJPEG FAQs
 - 5.1 Is it possible there is difference between OpenCV decoding output and dJPEG decoding output of YUV data?
 - 5.2 dJPEG decoding output is stored in the device memory allocated by calling dljpegMalloc(). Can this kind of device memory also be used directly by dINNE/dICU?

1. Overview

A DengLin Goldwasser™ AI accelerator card contains multiple JPEG codec engines (referred to as JPUs). Different JPUs can do codec simultaneously.

The dJPEG library provides software APIs (referred to as dJPEG APIs) for programming all available JPUs.



Encoding: Encode YUV data into compressed JPEG data.

Decoding: Decode compressed JPEG data to YUV data.

JPUs can access the device memory only. Both the input and output data for JPU processing are on the device memory. The specific device memory can be managed through a set of dedicated dJPEG APIs.

2. Main Codec Feature

The detailed codec features for a Goldwasser AI accelerator card are as follows.

- Baseline ISO/IEC 10918-1 JPEG compliance
- Support 1 or 3 color components
- 8-bit samples for each component
- Support 4:2:0, 4:2:2, 4:4:0, 4:4:4 and 4:0:0 color formats
- Minimum decode/encode size 16x16
- Maximum decode/encode size 32768x32768
- Support ROI, Scale, Rotate

3. Work Flow

3.1 Encode Overview

Encoding: Encode YUV data into compressed JPEG data.

To encode, first of all, get a JPU device and create a session on the device. Then, allocate the device memory as input and output buffers for the JPU. After that, copy YUV data to the input buffer. Then, start to encode. Encode outputs the compressed JPEG data. Combine the generated JPEG header and the compressed JPEG data to get a complete JPEG image.

3.2 Encode Flow

At a high level, the following steps should be followed for encoding by using dJPEG APIs.

1. Get a JPEG device.
2. Create a JPEG session on the JPEG device.
3. Prepare input and output buffers.
4. Encode.
5. Generate the JPEG header.
6. Clean up.

The above steps are explained in [Using dJPEG APIs](#) and demonstrated in the sample application included in Hamming™ SDK.

3.3 Decode Overview

Decoding: Decode JPEG data to YUV data.

To decode, first of all, get a JPU device and create a session on the device. JPEG header information needs to be parsed to get the output buffer size. Then, allocate input and output buffers on the device memory for the JPU. After that, copy JPEG compressed data to the input buffer. Then, start to decode and get the output YUV data.

3.4 Decode Flow

At a high level, the following steps should be followed for decoding using dJPEG APIs.

1. Get a JPEG device.
2. Create a JPEG session on the JPEG device.
3. Parse the JPEG header.
4. Prepare input and output buffers.
5. Decode.
6. Clean up.

The above steps are explained in [Using dJPEG APIs](#) and demonstrated in the sample application included in Hamming™ V2 SDK.

4. Using dJPEG APIs

4.1 Getting a JPEG Device

The first step for JPEG codec is to get DL_JPEG_DEVICE.

Call **dljpegGetDevice()** to get one JPEG device from GPU. Setting `clusterMask` and `channelMask` to `0xf` means that you let the dJPEG library decide which JPEG device to use. Caller can set an exact mask to get the JPU.

CudaSetDevice() must be called before **dljpegGetDevice()**.

```
// Select a DLGPU device
cudaSetDevice(0);

// Select a JPU in this GPU device
info.clusterMask = 0xf;
info.channelMask = 0xf;

result = dljpegGetDevice(&device, &info);
```

4.2 Getting session count

Call **dljpegGetSessionCount()** to get how many sessions is created on a JPEG device.

```
result = dljpegGetSessionCount(&session_count, device);
```

4.3 Creating a JPEG Session

The second step for JPEG codec is to create a JPEG session. Codec task must be processed in the JPEG session.

Call **dljpegCreateSession()** to create a JPEG session.

```
// Create a session for encode
result = dljpegCreateSession(&session, device);
```

4.4 Parsing JPEG Header (Decode Only)

For decoding, the image information is in the JPEG header. The information needs to be parsed out from the JPEG header since it is needed by decode and used to calculate the output buffer size.

A JPEG image is made up of the JPEG header and compressed image data. The parse APIs also retrieve the compressed image data offset by using parameter `sosoffset` which is needed by decode.

Call **dljpegParseHeader()** to parse the JPEG header information.

```
// Parse the JPEG header
result = dljpegParseHeader(&headerInfo, &sosOffset, JPEGData, JPEGDataLenth);
```

4.5 Device Memory

Both input and output data for JPUs are in the device memory. If the input data is in the host memory, it needs to be copied to the device memory first. Also the output data may need to be copied to the host memory.

Note: Driver may reserve more device memory for all JPUs in the same cluster when allocating device memory for one JPU. DengLin recommends that you keep JPUs' load balance to avoid wasting device memory.

Call **dljpegMalloc()** to allocate the device memory for the JPU.

Call **cudaMemcpy()** to copy data between the host memory and the device memory.

Call **cudaFree()** to free the device memory allocated by **dljpegMalloc()**.

```
// allocate device memory for the JPU
dljpegMalloc(&buffer, device, bufferSize);

// copy input data to device memory
cudaMemcpy(buffer, hostBuffer, inputSize, cudaMemcpyHostToDevice);

// free device memory
cudaFree(buffer);
```

4.6 Decode

Decode is in synchronous mode. Some transformations can be set in decode parameters such as scale, rotation, and ROI. Output YUV data is in 1-3 buffers depending on the output format plane count. For example, YUV420P output data are in 3 buffers, NV12 output data are in 2 buffers.

Call **dljpegDecode()** to start to decode.

```
// decode
dljpegDecode(session, &decodeParams, inputBuffer,
              JPEGDataLenth - sosOffset, outputBuffer,
              outputBufferStride, DL_JPEG_PIXEL_FORMAT_YUV420P);
```

4.7 Async Decode

An asynchronous-mode decode is supplied with a CUDA event in the parameter. The function will return immediately. The CUDA event will be triggered when decode is finished. Caller should keep the input and output buffers until decode is done.

Call **dljpegDecodeAsync()** to start to asynchronously decode.

```
// async decode
dljpegDecodeAsync(session, &decodeParams, inputBuffer,
                  JPEGDataLenth - sosOffset, outputBuffer,
                  outputBufferStride, DL_JPEG_PIXEL_FORMAT_YUV420P, event);
```


4.8 Encode

Encode is in synchronous mode. Some transformations can be set in encode parameters such as scale and rotation. Encode quality also can be set in parameters if needed. Input YUV data is in different buffers depending on the format plane count. Output data size will be retrieved by parameter `scanline_size`.

Call `dljpegEncode()` to start to encode.

```
//encode
dljpegEncode(session, &encodeParams, inputBuffer, strides,
              DL_JPEG_PIXEL_FORMAT_YUV420P, outputBuffer, &scanline_size);
```

Note: `outputBuffer` must align to 256 byte.

4.9 Creating a JPEG Header (Encode Only)

For encoding, a JPU outputs compressed image data. dJPEG provides APIs to generate the JPEG header. Combine the JPEG header and the compressed image data into a complete JPEG image.

Call `dljpegFillFileHeader()` to create the JPEG header.

```
// header buffer
uint8_t headerBuffer[1024];

// fill JPEG header
result = dljpegFillFileHeader(headerBuffer, &headerSize, 1024, &encodeParams);
```

4.10 Custom Huffman Table (Encode Only)

An advanced user can set the custom Huffman table for encode. The default table is used if no custom table is set.

The custom table can be set at `DL_JPEG_ENCODE_PARAMS.headerInfo -> dcHuffTables` and `DL_JPEG_ENCODE_PARAMS.headerInfo -> acHuffTables`.

Note: The custom table is only for the advanced user.

4.11 Custom Quantization table (Encode Only)

The advanced user can set custom quantization table for encode. The default table is used if no custom table is set.

The custom table can be set at `DL_JPEG_ENCODE_PARAMS.headerInfo -> quantTables`.

Note: The custom table is only for the advanced user.

4.12 Cleanup

Call `cudaFree()` to free buffers allocated by `dljpegMalloc()`.

Call `dljpegFreeHeader()` to free the header information allocated by `dljpegParseHeader()`.

Call `dljpegDestroySession()` to destroy the JPEG session.

There is no need to destroy the JPEG device.

5. dJPEG FAQs

5.1 Is it possible there is difference between OpenCV decoding output and dJPEG decoding output of YUV data?

Could be. The dJPEG always outputs the exact same YUV format as the original version of the input image. But OpenCV doesn't -- it may also do an extra format conversion internally after decoding.

For example, to decode a YUVJ420P JPEG image, the djpeg will output YUVJ420P data. However OpenCV will output YUV420P data. Because OpenCV will use the YUV420P format via `imread`, `cvtColor(COLOR_BGR2YUV_I420)` and other functions.

So please be careful of this difference when you are porting OpenCV apps.

5.2 dJPEG decoding output is stored in the device memory allocated by calling `djpegMalloc()`. Can this kind of device memory also be used directly by `dINNE/dICU`?

No. This kind of special device memory allocated by `djpegMalloc()` is so-called "channel memory", which is different from the normal device memory allocated by `cudaMalloc()`. This kind of device channel memory used by dJPEG has its special layout, which is different from the normal device memory.

Before accessing data in any channel memory allocated by `djpegMalloc`, you must copy this channel memory to other normal memory -- either device memory allocated by `cudaMalloc()` or the normal host memory allocated by `malloc()`. And please note that `dINNE` and `dICU` can only access the normal device memory allocated by the `cudaMalloc()`.