High Performance Application Programming Guide

Issue 01

Date 2020-05-30

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are trademarks of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Contents

1 Overview

Purpose

This document describes the constraints and suggestions for building high-performance applications on the Ascend 310, helping users understand the samples and quickly build high-performance applications.

Scope

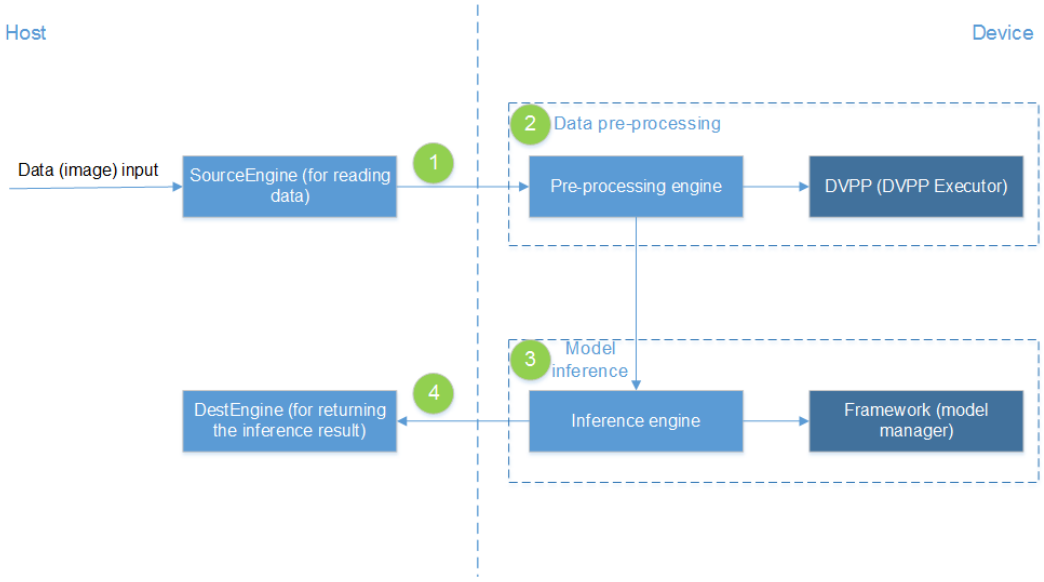
This document describes the key knowledge points for high-performance programming on the Ascend 310. It is intended for users who use chip hardware decoding, inference, and general image processing on the Ascend 310.

2 Key Points

During application development, the Matrix module processes data in the following phases:

1. Calls an API to transfer data (images) from the host to the device.
2. Calls DVPP APIs to encode, decode, and resize data.
3. Calls the model manager APIs provided by Framework to infer data.
4. Calls an API to send the inference result from the device to the host.

Figure 2-1 Data processing



During Matrix module processing, data transfer, DVPP processing, and model inference are most time-consuming. This document mainly describes how to streamline these phases and boost the performance.

For details about each API, see *Matrix API Reference* and *DVPP API Reference*

2.1 Memory Management

The Ascend 310 supports two types of APIs for memory management: memory management APIs provided by the native language and memory management APIs provided by the Matrix module.

2.1.1 Memory Management APIs Provided by the Native Language

The native language (C/C++) provides the **malloc**, **free**, **memcpy**, **memset**, **new**, and **delete** APIs for memory management. You can manage and control the lifecycle of memory allocated by using these APIs. If the memory to be allocated is less than 256 KB, memory management APIs provided by the native language and those provided by the Matrix module show similar performance. Therefore, you are advised to use a memory management API provided by the native language to simplify programming.

The following code shows how to use memory management APIs provided by the native language:

```
// Use malloc to alloc buffer
unsigned char* inbuf = (unsigned char*)malloc( fileLen );
// free buffer
free(inbuf);
inbuf = nullptr;
```

2.1.2 Memory Management APIs Provided by the Matrix Module

The Matrix module provides a set of C/C++ APIs for allocating and freeing memory, including **HIAI_DMalloc**/**HIAI_DFree** and **HIAI_DVPP_DMalloc**/**HIAI_DVPP_DFree**. Among these APIs, **HIAI_DMalloc** and **HIAI_DFree** are used to apply for memory and transfer data from the host to the device by working with **SendData**. While **HIAI_DVPP_DMalloc** and **HIAI_DVPP_DFree** are used to allocate memory for the DVPP on the device. You can call the **HIAI_DMalloc**/**HIAI_DFree** and **HIAI_DVPP_DMalloc**/**HIAI_DVPP_DFree** APIs to allocate memory to reduce copy operations and save time.

API Description

Table 2-1 describes the functions of the **HIAI_DMalloc**/**HIAI_DFree** and **HIAI_DVPP_DMalloc**/**HIAI_DVPP_DFree** APIs.

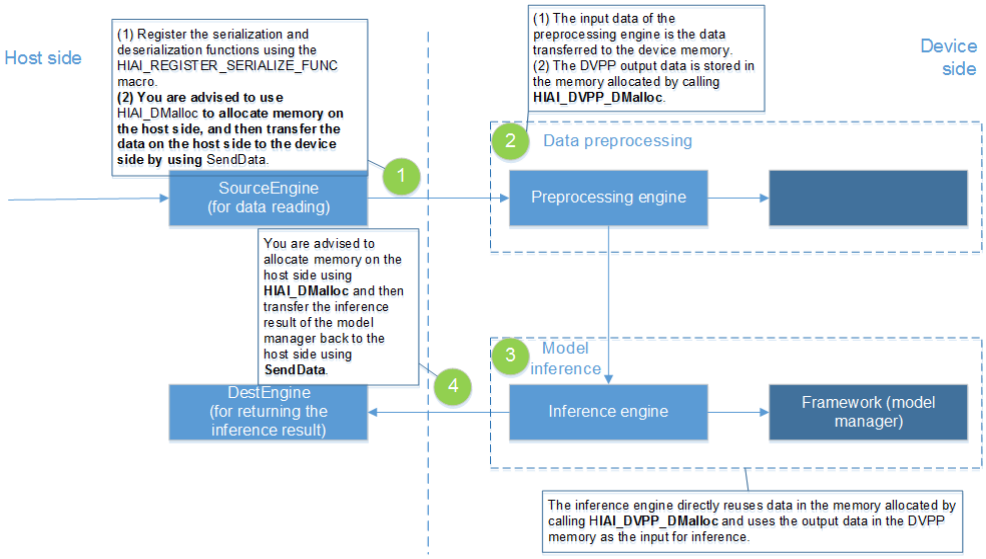
Table 2-1 API description

API Name	Function
HIAIMemory::HIAI_DMalloc (for C++ only)	Allocates memory. The memory is similar to common memory but offers better performance in cross-side transmission (host-device/device-host) and model inference.
HIAIMemory::HIAI_DFree (for C++ only)	Frees the memory allocated by **HIAIMemory::HIAI_DMalloc**. This API is used together with **HIAIMemory::HIAI_DMalloc**. When calling the **HIAIMemory::HIAI_DMalloc** API, you can set **flag** to **MEMORY_ATTR_AUTO_FREE**. In this case, if data is sent to the peer end by calling the **SendData** API, the allocated memory is automatically freed without calling the **HIAIMemory::HIAI_DFree** API after the program is complete. However, if the **SendData** API is not called to send data to the peer end after memory is allocated, you need to call **HIAIMemory::HIAI_DFree** to free the memory.
HIAI_DMalloc (for C/C++)	Allocates memory. The memory is similar to common memory but offers better performance in cross-side transmission (host-device/device-host) and model inference.
HIAI_DFree (for C/C++)	Frees the memory allocated by **HIAI_DMalloc**. This API is used together with **HIAI_DMalloc**. When calling the **HIAI_DMalloc** API, you can set **flag** to **MEMORY_ATTR_AUTO_FREE**. In this case, if data is sent to the peer end by calling the **SendData** API, the allocated memory is automatically freed without calling the **HIAI_DFree** API after the program is complete. However, if the **SendData** API is not called to send data to the peer end after memory is allocated, you need to call **HIAI_DFree** to free the memory.
HIAIMemory::HIAI_DVPP_DMalloc (for C++ only)	Allocates memory for the DVPP on the device.

API Name	Function
HIAIMemory::HIAI_DVPP_DFree (for C++ only)	Frees the memory allocated by the **HIAIMemory::HIAI_DVPP_DMalloc** API.
HIAI_DVPP_DMalloc (for C/C++)	Allocates memory for the DVPP on the device.
HIAI_DVPP_DFree (for C/C++)	Frees the memory allocated by the **HIAI_DVPP_DMalloc** API.

API Calling Process

Figure 2-2 API Calling Process



The usage of APIs in **Figure 2-2** is described as follows:

- The memory allocated by using the **HIAI_DMalloc** or **HIAIMemory::HIAI_DMalloc** API can be used in end-to-end data transmission and model inference. The data transmission efficiency and performance can be improved by calling the **HIAI_DMalloc** or **HIAIMemory::HIAI_DMalloc** API and using the **HIAI_REGISTER_SERIALIZE_FUNC** macro that serializes or deserializes user-defined data types.

  Allocating memory by using the **HIAI_DMalloc** or **HIAIMemory::HIAI_DMalloc** API has the following advantages:

  – The allocated memory can be directly used by host-device communication (HDC) module for data transmission to avoid data copy between the Matrix module and HDC.

  – You can use the allocated memory for zero-copy inference to reduce data copy time.

- The memory allocated by using the **HIAI_DVPP_DMalloc** or **HIAIMemory::HIAI_DVPP_DMalloc** API can be used by the DVPP. After being

used by the DVPP, data in the memory can be transparently transmitted to the inference model. If model inference is not required, data in the memory allocated by using the **HIAI_DVPP_DMalloc** API can be directly sent back to the host.

- The memory allocated by using the **HIAI_DMalloc**, **HIAIMemory::HIAI_DMalloc**, **HIAI_DVPP_DMalloc**, and **HIAIMemory::HIAI_DVPP_DMalloc** APIs is compatible with memory management APIs provided by the native language. It can be used as common memory, but cannot be freed by using APIs such as **free** and **delete**. Generally, memory allocated by using the **HIAI_DMalloc**, **HIAIMemory::HIAI_DMalloc**, **HIAI_DVPP_DMalloc**, and **HIAIMemory::HIAI_DVPP_DMalloc** APIs needs to be freed by calling **HIAI_DFree**, **HIAIMemory::HIAI_DFree**, **HIAI_DVPP_DFree**, and **HIAIMemory::HIAI_DVPP_DFree**, respectively.

  When calling the **HIAI_DMalloc** or **HIAIMemory::HIAI_DMalloc** API, you can set **flag** to **MEMORY_ATTR_AUTO_FREE**. In this case, if data is sent to the peer end by calling the **SendData** API, the allocated memory is automatically freed without calling the **HIAIMemory::HIAI_DFree** API after the program is complete. However, if the **SendData** API is not called to send data to the peer end after memory is allocated, you need to call **HIAIMemory::HIAI_DFree** to free the memory.
- The memory allocated by using the **HIAI_DVPP_DMalloc** or **HIAIMemory::HIAI_DVPP_DMalloc** API meets the requirements of the DVPP. Therefore, when the resources are limited, you are advised to use these APIs only for the DVPP.

Precautions for API Usage

When allocating memory by using **HIAI_DMalloc** or **HIAIMemory::HIAI_DMalloc**, pay attention to the following issues about memory management:

- When allocating memory to be automatically freed for host-device or device-host data transmission, if a smart pointer is used, the Matrix module automatically frees the memory. Therefore, the destructor specified by the smart pointer must be empty. If the pointer is not a smart pointer, the Matrix module automatically frees the memory.
- When allocating memory to be manually freed for host-device or device-host data transmission, if a smart pointer is used, you need to set the destructor to **HIAI_DFree** or **HIAIMemory::HIAI_DFree**. If the pointer is not a smart pointer, you need to call **HIAI_DFree** or **HIAIMemory::HIAI_DFree** to free the memory after data transmission is complete.
- When memory to be manually freed is allocated, the **SendData** API cannot be called repeatedly to send data in the memory.
- When allocating memory to be manually freed, if the memory is used for data transmission between the host and device, do not reuse the data in the memory before the memory is freed. If the memory is used for host-host or device-device data transmission, the data in the memory can be reused before the memory is freed.
- When allocating memory to be manually freed, if the **SendData** API is called to asynchronously send data, data in the memory cannot be modified after data is sent.

If the **HIAI_DVPP_MAlloc** or **HIAIMemory::HIAI_DVPP_DMalloc** API is called to allocate memory for device-host data transmission, you need to call the **HIAI_DVPP_DFree** or **HIAIMemory::HIAI_DVPP_DFree** API to manually free the memory, because the **HIAI_DVPP_MAlloc** or **HIAIMemory::HIAI_DVPP_DMalloc** API does not automatically free the memory. If a smart pointer is used to store the allocated memory address, the destructor must be set to **HIAI_DVPP_DFree** or **HIAIMemory::HIAI_DVPP_DFree**.

API Calling Example

```
(1) When the performance optimization solution is used to transmit data, the data transmit API must be manually serialized and deserialized.
// Note: The serialization function is used at the transmit end and the deserialization function is used at the receive end. Therefore, you are advised to register this function with both transmit and receive ends.
// Data structure
typedef struct
{
    uint32_t left_offset = 0;
    uint32_t right_offset = 0;
    uint32_t top_offset = 0;
    uint32_t bottom_offset = 0;
    // The serialize function is used to serialize a structure.
    template <class Archive>
    void serialize(Archive & ar)
    {
        ar(left_offset,right_offset,top_offset,bottom_offset);
    }
} crop_rect;



// Registers the structure to be transferred between engines.
typedef struct EngineTransNew
{
    std::shared_ptr<uint8_t> trans_buff = nullptr;    // Transfer buffer
    uint32_t buffer_size = 0;                  // Transfer buffer size
    std::shared_ptr<uint8_t> trans_buff_extend = nullptr;
    uint32_t buffer_size_extend = 0;
    std::vector<crop_rect> crop_list;
    // The serialize function is used to serialize a structure.
    template <class Archive>
    void serialize(Archive & ar)
    {
        ar(buffer_size, buffer_size_extend, crop_list);
    }
}EngineTransNewT;
// Serialization function
/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,        // Serializes the Trans data.
* @param [in]: data_ptr         // Structure pointer
* @param [out]: struct_str       // Structure buffer
* @param [out]: data_ptr         // Structure data pointer buffer
* @param [out]: struct_size      // Structure size
* @param [out]: data_size        // Structure data size
*/
void GetTransSearPtr(void* data_ptr, std::string& struct_str,
    uint8_t*& buffer, uint32_t& buffer_size)
{
    EngineTransNewT* engine_trans = (EngineTransNewT*)data_ptr;
    uint32_t dataLen = engine_trans->buffer_size;
    uint32_t dataLen_extend = engine_trans->buffer_size_extend;
    // Obtains the structure buffer and size.
    buffer_size = dataLen + dataLen_extend;
    buffer = (uint8_t*)engine_trans->trans_buff.get();

    // Serialization
```



```
    std::ostringstream outputStr;
    cereal::PortableBinaryOutputArchive archive(outputStr);
    archive((*engine_trans));
    struct_str = outputStr.str();
}
// Deserialization function
/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,             // Deserializes the Trans data.
* @param [in]: ctrl_ptr              // Structure pointer
* @param [in]: data_ptr              // Structure data pointer
* @param [out]: std::shared_ptr<void> // Structure pointer assigned to the engine
*/
std::shared_ptr<void> GetTransDearPtr(
    const char* ctrlPtr, const uint32_t& ctrlLen,
    const uint8_t* dataPtr, const uint32_t& dataLen)
{
    if(ctrlPtr == nullptr) {
        return nullptr;
    }
    std::shared_ptr<EngineTransNewT> engine_trans_ptr = std::make_shared<EngineTransNewT>();
    // Assigns a value to engine_trans_ptr.
    std::istringstream inputStream(std::string(ctrlPtr, ctrlLen));
    cereal::PortableBinaryInputArchive archive(inputStream);
    archive((*engine_trans_ptr));
    uint32_t offsetLen = engine_trans_ptr->buffer_size;
    if(dataPtr != nullptr) {
        (engine_trans_ptr->trans_buff).reset((const_cast<uint8_t*>(dataPtr)), ReleaseDataBuffer);
        // trans_buff and trans_buff_extend point to a contiguous memory space whose address starts with
dataPtr;
        // therefore, you only need to bind trans_buff to the destructor, and then the destructor will free the
contiguous memory space after being used.
        (engine_trans_ptr->trans_buff_extend).reset((const_cast<uint8_t*>(dataPtr + offsetLen)),
SearDeleteNothing);
    }
    return std::static_pointer_cast<void>(engine_trans_ptr);
}
// Registers EngineTransNewT
HIAI_REGISTER_SERIALIZE_FUNC("EngineTransNewT", EngineTransNewT, GetTransSearPtr, GetTransDearPtr);
```

(2) When sending data, you can use only the registered data types. Use **HIAI_DMalloc** to allocate memory to optimize performance.

Note: When transferring data from the host to the device, you are advised to use **HIAI_DMalloc** to optimize transmission efficiency. The data size supported by the **HIAI_DMalloc** API ranges from 0 bytes to (256 MB – 96 bytes). If the data size exceeds this range, use the **malloc** API to allocate memory.

```
    // Allocates the data memory by calling the HIAI_DMalloc API. The value 10000 indicates the delay in
microseconds, that is, if the memory space is insufficient, the program waits 10000 ms.
    HIAI_StatusT get_ret =  HIAIMemory::HIAI_DMalloc(width*align_height*3/2,(void*&)align_buffer, 10000);
    // Sends data. After the SendData API is called, the HIAI_DFree API does not need to be called. The
value 10000 indicates the delay.
    graph->SendData(engine_id_0, "TEST_STR", std::static_pointer_cast<void>(align_buffer), 10000);
```

2.2 Host-Device Data Transmission

```
// EngineTransNewT structure
typedef struct EngineTransNew
{
    std::shared_ptr<uint8_t> trans_buff = nullptr;    // Transfer buffer
    uint32_t buffer_size = 0;                  // Transfer buffer size
    std::shared_ptr<uint8_t> trans_buff_extend = nullptr;
    uint32_t buffer_size_extend = 0;
    std::vector<crop_rect> crop_list;
    // The serialize function is used to serialize a structure.
    template <class Archive>
    void serialize(Archive & ar)
    {
        ar(buffer_size, buffer_size_extend, crop_list);
    }
```



```
}EngineTransNewT;

/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,        // Serializes the Trans data.
* @param [in]: dataPtr          // Structure pointer
* @param [out]: structStr        // Structure buffer
* @param [out]: buffer           // Structure data pointer buffer
* @param [out]: buffSize         // Structure data size
*/
void GetTransSearPtr(void* data_ptr, std::string& struct_str,
    uint8_t*& buffer, uint32_t& buffer_size)
{
    EngineTransNewT* engine_trans = (EngineTransNewT*)data_ptr;
    uint32_t dataLen = engine_trans->buffer_size;
    uint32_t dataLen_extend = engine_trans->buffer_size_extend;
    // Obtains the structure buffer and size.
    buffer_size = dataLen + dataLen_extend;
    buffer = (uint8_t*)engine_trans->trans_buff.get();

    // Serialization
    std::ostringstream outputStr;
    cereal::PortableBinaryOutputArchive archive(outputStr);
    archive((*engine_trans));
    struct_str = outputStr.str();
}

/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,             // Deserializes the Trans data.
* @param [in] : ctrlPtr               // Structure pointer
* @param [in]: ctrlLen               // Control information size of the data structure
* @param [in] : dataPtr               // Structure data pointer
* @param [in]: dataLen               // Data storage size of the structure, which is used only for verification
and does not represent the original data size
* @param [out]: std::shared_ptr<void> // Structure pointer assigned to the engine
*/
std::shared_ptr<void> GetTransDearPtr(
    const char* ctrlPtr, const uint32_t& ctrlLen,
    const uint8_t* dataPtr, const uint32_t& dataLen)
{
    if(ctrlPtr == nullptr) {
        return nullptr;
    }
    std::shared_ptr<EngineTransNewT> engine_trans_ptr = std::make_shared<EngineTransNewT>();
    // Assigns a value to engine_trans_ptr.
    std::istringstream inputStream(std::string(ctrlPtr, ctrlLen));
    cereal::PortableBinaryInputArchive archive(inputStream);
    archive((*engine_trans_ptr));
    uint32_t offsetLen = engine_trans_ptr->buffer_size;
    if(dataPtr != nullptr) {
        (engine_trans_ptr->trans_buff).reset((const_cast<uint8_t*>(dataPtr)), ReleaseDataBuffer);
        // trans_buff and trans_buff_extend point to a contiguous memory space whose address starts with
dataPtr;
        // therefore, you only need to bind trans_buff to the destructor, and then the destructor will free the
contiguous memory space after being used.
        (engine_trans_ptr->trans_buff_extend).reset((const_cast<uint8_t*>(dataPtr + offsetLen)),
SearDeleteNothing);
    }
    return std::static_pointer_cast<void>(engine_trans_ptr);
}
// Registers EngineTransNewT
HIAI_REGISTER_SERIALIZE_FUNC("EngineTransNewT", EngineTransNewT, GetTransSearPtr, GetTransDearPtr);
```

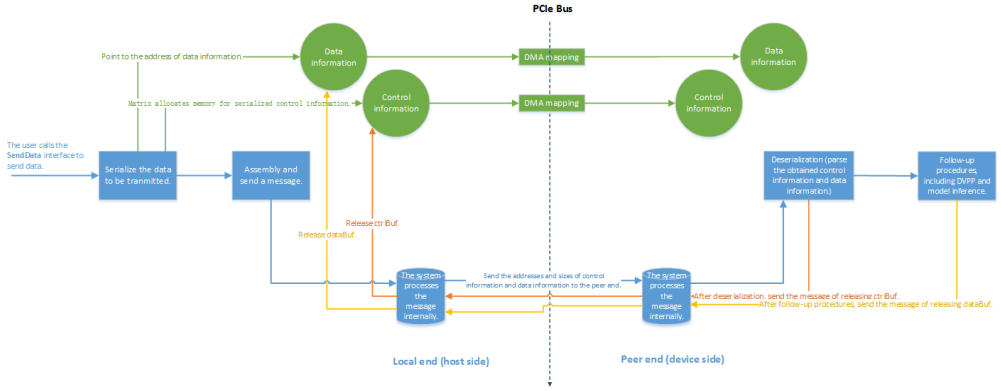
In the case of a large amount of image data or code stream transmission, **HIAI_REGISTER_SERIALIZE_FUNC** is used to serialize or deserialize the user-defined data types, which can implement high-performance data transmission and save the transmission time.

The Matrix module describes the data to be transmitted in the form of "control information+data information". The control information refers to a user-defined data type, and the data information refers to content to be transmitted. The Matrix module provides the following mechanism to ensure data transmission between the host and device.

1. Before data transmission, you can call the **HIAI_REGISTER_SERIALIZE_FUNC** macro to register user-defined data types, user-defined serialization functions, and user-defined deserialization functions.
2. After you call the **SendData** API to send data at the local end, the Matrix module performs the operations. **Figure 2-3** shows the processing process.
    a. Call the user-defined serialization function to serialize the control information and save the serialized control information to the memory (**ctrlBuf**).
    b. Copy the control information and store it in the memory of the peer end through the Direct Memory Access (DMA) mapping, and maintain the mapping between the control information of the local end and that of the peer end.

       The pointer to the memory (**dataBuf**) has been transferred through the input parameter of the **SendData** API. **dataBuf** is allocated by calling the **HIAI_DMalloc/HIAI_DVPP_DMalloc** API. After the memory is allocated, the system copies the local data information through DMA mapping and stores it to the memory of the peer end, and maintains the mapping between the local end and the peer end.
    c. A message including the addresses and sizes of **ctrlBuf** and **dataBuf** is sent to the peer end.
    d. After the peer end receives the message, the Matrix module calls the user-defined deserialization function to parse the control information and data information obtained by the peer end, and sends the parsed data to the corresponding receiving engine for processing.
    e. After the peer end parses the data, the control information is used out. Therefore, the memory (**ctrlBuf**) for storing the control information can be freed. This memory is allocated at the local end, so the peer end needs to send the local end a message for freeing **ctrlBuf**.
    f. After receiving the message, the local end frees **ctrlBuf**.
    g. After the engine receives and processes the data, **dataBuf** can be freed. The Matrix module does not know when **dataBuf** is used up. Therefore, when the deserialization function is implemented, **dataBuf** needs to return a smart pointer and bound to the destructor **hiai::Graph::ReleaseDataBuffer**. When the smart pointer ends the lifecycle destruction, the destructor is automatically called to send a message to the local end for freeing **dataBuf**.
    h. After receiving the message, the local end frees **dataBuf**.

Figure 2-3 Data processing process



Example scenario: The following code declares the serialization/deserialization function, defines the data type (struct), and registers the user-defined data type, serialization function, and deserialization function using the **HIAI_REGISTER_SERIALIZE_FUNC** macro. Before data transmission, the Matrix module calls the registered serialization function. After data transmission, the Matrix module calls the registered deserialization function.

```
// EngineTransNewT structure
struct EngineTransNewT
{
    std::shared_ptr<uint8_t> transBuff;
    uint32_t bufferSize;   // Buffer size
    std::string url;
    // The serialize function is used to serialize a structure.
    template <class Archive>
    void serialize(Archive & ar)
    {
        ar(url);
    }
}

/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,        // Serializes the Trans data.
* @param [in]: dataPtr          // Structure pointer
* @param [out]: structStr        // Structure buffer
* @param [out]: buffer           // Structure data pointer buffer
* @param [out]: buffSize         // Structure data size
*/
void GetTransSearPtr(void* dataPtr, std::string& structStr, uint8_t*& buffer, uint32_t& bufferSize)
{
    EngineTransNewT* engineTrans = (EngineTransNewT *)dataPtr;
    std::shared_ptr<uint8_t> transBuff = ((EngineTransNewT *)dataPtr)->transBuff;
    buffer = (uint8_t*)engineTrans->transBuff.get();
    bufferSize = engineTrans->bufferSize;

    engineTrans->transBuff = nullptr;
    engineTrans->buffSize = 0;

    std::ostringstream outputStr;
    cereal::PortableBinaryOutputArchive archive(outputStr);
    archive((*engineTrans));
    struct_str = outputStr.str();

    ((EngineTransNewT*)dataPtr)->transBuff = transBuff;
    engineTrans->buffSize = bufferSize;
}

/**
```

```
* @ingroup hiaiengine
* @brief GetTransSearPtr,            // Deserializes the Trans data.
* @param [in] : ctrlPtr              // Structure pointer
* @param [in]: ctrlLen               // Control information size of the data structure
* @param [in] : dataPtr              // Structure data pointer
* @param [in]: dataLen               // Data storage size of the structure, which is used only for verification
and does not represent the original data size
* @param [out]: std::shared_ptr<void> // Structure pointer assigned to the engine
*/
std::shared_ptr<void> GetTransDearPtr(char* ctrlPtr, const uint32_t& ctrlLen, uint8_t* dataPtr, const
uint32_t& dataLen)
{
    std::shared_ptr<EngineTransNewT> engineTransPtr = std::make_shared<EngineTransNewT>();
    std::istringstream inputStream(std::string(ctrlPtr, ctrlLen));
    cereal::PortableBinaryInputArchive archive(inputStream);
    archive((*engineTransPtr));
    engineTransPtr->bufferSize = dataLen;
    engineTransPtr->transBuff.reset(dataPtr, hiai::Graph::ReleaseDataBuffer);
    return std::static_pointer_cast<void>(engineTransPtr);
}
// Registers EngineTransNewT
HIAI_REGISTER_SERIALIZE_FUNC("EngineTransNewT", EngineTransNewT, GetTransSearPtr, GetTransDearPtr);
```

2.3 Usage of DVPP

2.3.1 Image and Video Encoding and Decoding

The Matrix module provides APIs for image processing and video encoding/decoding. You can perform image/video decoding on the device to reduce the data traffic between the host and device, the data transmission time, and the bandwidth pressure.

On the host, the **HIAI_DMalloc** API provided by the Matrix module is called to allocate memory on the device. The allocated memory is used as the input of image and video encoding and decoding. It is recommended that the start address of the memory for data storage be 128-byte aligned. On the device, after image or video pre-processing, the DVPP calls the **HIAI_DVPP_DMalloc** API provided by the Matrix module to allocate memory. The memory is used to store the image/video pre-processing output.

2.3.2 Image Cropping and Resizing

When programming with Ascend 310, you are advised to use digital vision pre-processing (DVPP) for image cropping and resizing.

Figure 2-4 Cropping and resizing process

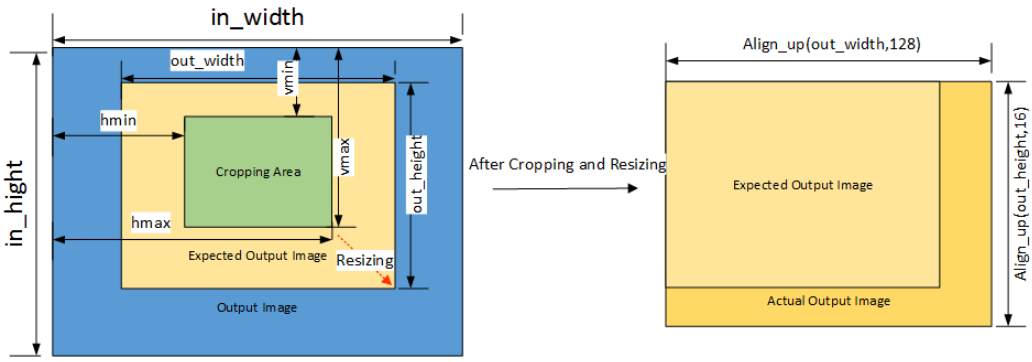


Figure 2-4 shows the cropping/resizing process. This process crops the region of interest (ROI) and uses the cropped area for re-sampling. The process of cutting out the ROI is called cropping, and the process of re-sampling is called resizing. When the resizing coefficient is 1, only cropping is performed. When the original image is the cropping output, only resizing is performed.

The restrictions on cropping/resizing input are as follows:

- Input data address (OS virtual address): 16-byte aligned
- Buffer restriction for the input image width: 16-byte aligned
- Buffer restriction for the input image height: 2-byte aligned

To achieve zero-copy during high performance programming, the memory addresses assigned to users on the device (receive end) must meet the preceding restrictions when the memory is allocated. Generally, you can use either of the following methods depending on different input types:

- Method 1: If the app uses the host or other hardware for decoding, perform data cropping or padding on the host (transmit end) to meet the 16 x 2 alignment requirement. In this way, the Matrix module automatically allocates data memory that meets the preceding restrictions at the data receive end.

  NOTE

  The input data in the sample is a YUV image. Therefore, the size of the image is calculated by using the following formula: (*Image width* x *Image height*) x 3/2. You need to change the formula according to the image format.

```
static const uint32_t ALIGN_W = 16;
static const uint32_t ALIGN_H = 2;
uint32_t imageWidth = 500;
uint32_t imageHeight = 333;
uint32_t imageSize = imageWidth * imageHeight *3/2;
uint32_t align_width = image_width, align_height = image_height;

// Performs width and height alignment.
alignWidth  = (alignWidth % ALIGN_W) ? alignWidth : (imageWidth + ALIGN_W)/ALIGN_W*
ALIGN_W;
alignHeight = (alignHeight % ALIGN_H) ? alignHeight : (imageHeight + ALIGN_H)/ALIGN_H *
ALIGN_H;
uint32_t alignSize = alignWidth * alignHeight *3/2;

// Reads the file data and copies the aligned memory.
FILE *fpIn = fopen(filePath.data(), "rb");
Uint8_t* imageBuffer = (uint8_t*) malloc(imageSize);
HIAI_StatusT getRet = hiai::HIAIMemory::HIAI_DMalloc(fileLen, (void*&)alignBuffer, 10000);

size_t size = fread(imageBuffer, 1, imageSize, fpIn);
// Copies data.
Uint32_t tempLen = imageWidth * imageHeight;
if (alignWidth == imageWidth)
{
    if(alignHeight > imageHeight)
    {
        memcpy_s(alignBuffer, tempLen, imageBuffer, tempLen);
        memcpy_s(alignBuffer + alignHeight * alignWidth,
            tempLen/2, imageBuffer + tempLen, tempLen/2);
    }
} else {
    for(int32_t n=0;n<imageHeight;n++)
    {
       memcpy_s(alignBuffer+n*alingWidth,imageWidth,
          imageBuffer+n*imageWidth,imageWidth);
```



```
    }
    for(int32_t n=0;n<imageHeight/2;n++)
    {
        memcpy_s(alignBuffer+n*alignWidth+alignHeight* alignWidth,
            imageWidth, imageBuffer+n* imageWidth + imageHeight*imageWidth,imageWidth);//UV
     }
}
```

- Method 2: Decode the image or video on the device to generate 16 x 2 aligned output that can be used as the input for DVPP VPC cropping and resizing.

2.4 Model Conversion Pre-Processing Configuration

As shown in **Figure 2-4**, the cropped or resized image are aligned. As a result, some areas are padded and are not the input required by the original model. To obtain the required image, you can copy the data to a new buffer and input the data to the model inference module. However, this process causes overheads. To reduce such overheads, the Matrix module provides a mechanism that allows padded areas in input images to the model manager (**modelManager**). The AIPP module of the model manager crops the image based on the user-defined width and height to generate an image that meets the model input requirements. When the image generated by the AIPP module is used, the inference module does not need to copy data, greatly improving the performance.

In the following example, the input image size for module inference is 224 pixels x 224 pixels, and the data obtained from DVPP is 128 pixels x 16 pixels aligned (the aligned image size is 256 pixels x 224 pixels).

2.5 Batch and Timeout

For most models, especially small models, chip inference using batch input brings performance improvement. Batch inference greatly improves the data throughput and chip utilization. Although a certain amount of latency is generated, the overall performance of the system is improved. Therefore, to build a high-performance application, the input batch needs to be the largest batch allowed by the latency.

To make batch operations more simple and flexible, the Matrix module supports the timeout mechanism. Users can configure **is_repeat_timeout_flag** in the **config** file to set whether to enable timeout waiting, and configure **wait_inputdata_max_time** to set the timeout duration. If timeout parameters are set, the system transfers null pointers when calling the **Process** function after timeout. You need to compile the code logic for timeout processing, which stores the received data in a queue and performs inference until the received data is sufficient to form a batch. You can use the **hiai::MultiTypeQueue** queue provided by the Matrix module. To prevent data starvation, a timeout duration must be specified by using the timeout setting API based on the latency requirement of the application. When the timeout duration expires, the Matrix module calls the main processing flow of the engine. Users can obtain and process the data in the queue to avoid starvation.

If the model input is multiple batches and you want to send data of each batch to the model manager (inference engine) for inference, you need to add the following code logic:

1. You need to apply for a buffer space on the device side to store the data of each batch.
2. When the inference engine on the device side receives data of all batches, you need to combine the data and store it in the buffer space allocated in **1**.
3. You can use the data in batches stored in the buffer space for inference only when the number of received batches on the device side is the same as that required for model inference.

2.6 Processing the Input and Output Data of Algorithm Inference

To avoid memory copy during algorithm inference, you are advised to use the **HIAI_DMalloc** API to allocate memory for the input and output data when calling the process API of the model manager. In this way, zero-copy can be achieved and the processing time is optimized. If DVPP processing is required before inference, use the memory transferred by the Matrix module as the DVPP input memory, use the **HIAI_DVPP_DMalloc** API to allocate the DVPP output memory, and use the DVPP output memory as the input memory of the inference engine.

2.7 Optimization of Data Backhaul

After the inference is complete, the inference result or inference end signal needs to be sent to the host. If the inference engine calls **SendData** to send data back to the host, the time of the inference engine is consumed. It is recommended that an independent engine (for example, DataOptEngine) be deployed to return data. After the inference is completed, the inference engine transparently transmits the processed data to DataOptEngine, which forwards the data to an engine (for example, DstEngine) on the host side.

```
// DataOptEngine on the device side sends data to the host side.
HIAI_IMPL_ENGINE_PROCESS("DataOptEngine", DataOptEngine, 1)
{
    HIAI_StatusT hiaiRet = HIAI_OK;
    if (arg0 == nullptr) {
        HIAI_ENGINE_LOG(HIAI_INVALID_INPUT_MSG, "get inference result timeout");
        return HIAI_INVALID_INPUT_MSG;
    }
    hiaiRet = SendData(0, "EngineTransNewT", arg0);
}
// DataOptEngine on the device side sends data to the host side.
HIAI_IMPL_ENGINE_PROCESS("DstEngine", DstEngine, 1)
{
    HIAI_StatusT ret = HIAI_OK;
    if (nullptr != arg0) {
        printf("dest engine had receive data already\n");
        std::shared_ptr<std::string> result =
            std::static_pointer_cast<std::string>(arg0);
        ret = SendData(0, "string", result);
        if (HIAI_OK != ret) {
            HIAI_ENGINE_LOG(ret, "DstEngine SendData to recv failed");
            return ret;
        }
    }
    else {
        HIAI_ENGINE_LOG(HIAI_INVALID_INPUT_MSG, "DestEngine Fail to receive data");
        printf("destengine do not receive data arg0 is null\n");
        return HIAI_INVALID_INPUT_MSG;
```



```
    }
    return HIAI_OK;
}
```

3 Operator Usage Suggestions

Principles

On the Ascend 310 chip, you can improve the efficiency of the Cube to optimize the algorithm performance. In this case, you need to reduce data transmission and vector calculation. The general principles are as follows:

1. Network structure
   - Mainstream network topologies such as ResNet and MobileNet are recommended because their performance has been optimized.
   - Earlier network topologies such as VGG and AlexNet use large network models and bring high bandwidth pressure.
   - In matrix multiplication, set the values of **M**, **K**, **N** to multiples of 16. Increase the number of channels in the algorithm if possible and do not reduce the number of channels by creating groups.
   - Increasing the parameter reuse rate can reduce the bandwidth pressure. Therefore, you can increase the filter reuse rate to improve the performance. For example, use greater feature map sizes, smaller stride values, and smaller dilation values.
2. Conv operator
   - In non-quantization mode, it is recommended that the number of input and output channels of a Conv operator be integral multiples of 16.
   - In quantization mode, it is recommended that the number of input and output channels of a Conv operator be integral multiples of 32.
   - In quantization mode, it is recommended that fewer pooling operators be inserted between multiple Conv operators.
3. Full connection (FC) operator

   If FC operators exist on the network, use multiple batches to perform inference at the same time.
4. Concat operator
   - In non-quantization mode, it is recommended that the number of input channels of the Concat operator be integral multiples of 16.
   - In quantization mode, it is recommended that the input channel of the Concat operator be integral multiples of 32.

5. Conv fusion operator

   The Conv+BatchNorm+Scale+Relu/Relu6 combination is recommended. The performance has been optimized.

6. Norm operator
   - The BatchNorm operator is recommended, which uses the pre-trained **Norm** parameter.
   - Operators (such as LRN) that require the online calculation of the **Norm** parameter are not recommended.

7. Detection operator

   Mainstream detection network topologies such as Faster R-CNN and SSD are recommended because their performance has been optimized.

Tips

1. The performance of Conv+(BatchNorm+Scale)+Relu is better than that of Conv+(BatchNorm+Scale)+Tanh. Avoid using complex activation functions.
2. When Concat operators are assembled in the C dimension, the performance is better if the values of **Tensor** and **Channel** are multiples of 16.
3. When the value of **Batch** is a multiple of 16, the performance is better.
4. The continuous convolution structure shows better performance. If multiple vector operators (such as pooling) are inserted between the convolution layers, the performance is poor. This is obvious in the INT8 model.
5. In early versions of AlexNet and GoogleNet, LRN is used as the normalization operator. The calculation of this operator is complex. During the evolution of the algorithm, the operator is replaced with other operators such as BatchNorm. The LRN operator is no longer used in mainstream network structures such as ResNet and Inception. For the Ascend310 platform, it is recommended BatchNorm be used on the network.

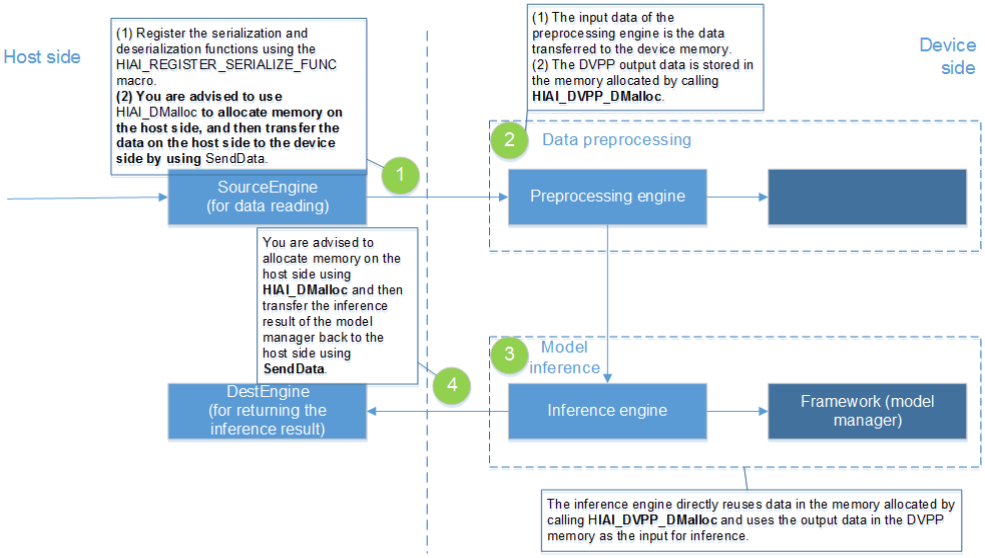
4 Samples

If multiple inference models are run at the same time, multiple graphs are not recommended. You are advised to use single graph multiple batches or single graph multiple engines.

4.1 Data Flow

The sample code uses the classification network (ResNet-18) to process images. **Figure 4-1** shows the data flow.

Figure 4-1 API Calling Process



4.2 Zero-Copy

The sample code shows the idea of zero-copy. Zero-copy means that the user does not perform any explicit copy operation on the image data in the entire process. Zero-copy requires the following operations:

1. Use **HIAI_REGISTER_SERIALIZE_FUNC** to register the serialization function (**GetSerializeFunc**) and deserialization function (**GetDeserializeFunc**) to implement serialization and deserialization of data types, respectively.
2. Call the **HIAI_Dmalloc** API provided by the Matrix module to allocate memory and then call the **SendData** API to transfer data from the host to the device.
3. The memory allocated by calling the **HIAI_DMalloc** API is used as the input of image/video encoding and decoding without data copying.
4. The memory allocated by calling the **HIAI_DVPP_DMalloc** API provided by the Matrix module meets the input/output address requirements of the DVPP and can be used directly as the image/video output. After the **HIAI_DVPP_DMalloc** API is called to allocate memory, the **HIAIMemory::HIAI_DVPP_DFree** API must be called to free the memory.

   In the DVPP, the input of the VPC module can directly reuse the output data of the JPEGD module in the memory.
5. The memory allocated by calling the **HIAI_DVPP_DMalloc** API can be used directly as the input of the first-layer model inference without data copying.

5 FAQs

5.1 What Do I Do If an Exception Occurs Because a Smart Pointer Is Used to Automatically Free Memory but the Destructor Is Not Empty?

Symptom

HIAI_DMalloc or **HIAIMemory::HIAI_DMalloc** is called to allocate memory for host-device or device-host data transmission. **flag** is set to **MEMORY_ATTR_AUTO_FREE** to automatically free memory. A smart pointer is used to store the address of the allocated memory, but no destructor is defined (the default destructor is called) or another destructor is called to free the memory (such as **HIAI_DFree**, which causes repeated memory freeing). As a result, the program becomes abnormal and memory leakage occurs.

Solution

The Matrix module automatically frees the memory. Therefore, set the destructor of the smart pointer to empty to resolve the problem.

The reference code is as follows:

```
* @ Allocates memory to be automatically freed. If a smart pointer is used, the destructor must be empty (the Matrix module automatically frees the memory).
**/
int main()
{
    // Uses DMalloc to allocate memory to be automatically freed.
    unsigned char* inbuf = nullptr;
    uint32_t bufferLen = 1080*1920;
    // The allocated memory is automatically freed. In this case, set flag to hiai::HIAI_MEMORY_ATTR::MEMORY_ATTR_ATUO_FREE.
    HIAI_StatusT getRet = hiai::HIAIMemory::HIAI_DMalloc(bufferLen, (void*&)inbuf, 10000);
```



```
    // getRet exception judgment
    // Stores the data to be transmitted in inbuf. The size of the data is bufferLen.
    // Assembles the engineTranData structure for sending data
    EngineTransNewT   engineTranData;
    engineTranData->bufferSize = bufferLen;
    engineTranData->transBuff.reset(std::static_pointer_cast<uint8_t *> (inbuf), [](uint8_t *)(addr )
        {
            // The destructor does not perform any operation. The Matrix module automatically frees the
memory. If the destructor is not defined, the default destructor is called, causing an exception.
                // If another destructor such as HIAI_DFree is called, repeated memory freeing occurs, causing
an exception.
        });
    // Sends data to the peer end by calling SendData.
    HIAI_StatusT sendRet =
SendData(DEFAULT_PORT,"EngineTransNewT",std::static_pointer_cast<void>(engineTranData));
// sendRet exception judgment
}
```

5.2 What Do I Do If an Exception Occurs Because a Smart Pointer Is Used to Manually Free Memory but the Destructor Is Not HIAI_DFree?

Symptom

HIAI_DMalloc or **HIAIMemory::HIAI_DMalloc** is called to allocate memory for host-device or device-host data transmission. **flag** is set to **MEMORY_ATTR_MANUAL_FREE** to manually free memory. A smart pointer is used to store the address of the allocated memory. If the destructor of the smart pointer is incorrect, the program becomes abnormal and memory leakage occurs.

Solution

Set the destructor of the smart pointer to **HIAI_DFree** or **HIAIMemory::HIAI_DFree**.

An example of the code is as follows:

```
/**
* @ Allocates memory to be manually freed. If a smart pointer is used, use HIAI_DFree as the destructor.
**/
int main()
{
    // Uses DMalloc to manually free memory.
    unsigned char* inbuf = nullptr;
    uint32_t bufferLen = 1080*1920;
    HIAI_StatusT getRet = hiai::HIAIMemory::HIAI_DMalloc(bufferLen, (void*&)inbuf,
10000,hiai::HIAI_MEMORY_ATTR::MEMORY_ATTR_MANUAL_FREE);
    // getRet exception judgment

    // Stores the data to be transmitted in inbuf. The size of the data is bufferLen.
    // Assembles the engineTranData structure for sending data
    EngineTransNewT   engineTranData;
    engineTranData->bufferSize = bufferLen;
    engineTranData->transBuff.reset(std::static_pointer_cast<uint8_t *> (inbuf), [](uint8_t *)(addr )
        {
            // Uses HIAI_DFree as the destructor. Otherwise, memory leakage occurs.
            hiai::HIAIMemory::HIAI_DFree(void * addr)
        });
    // Sends data to the peer end by calling SendData.
    HIAI_StatusT sendRet =
SendData(DEFAULT_PORT,"EngineTransNewT",std::static_pointer_cast<void>(engineTranData));
```

```
    // sendRet exception judgment
}
```

6 Appendix

6.1 Change History

Release Date	Description
2020-05-30	This issue is the first official release.

