# 高性能应用编程用户手册

文档版本 01

发布日期 2020-05-09

华为技术有限公司

**商标声明**

HUAWEI和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

**注意**

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

# 目 录

# 1 简介

## 目的

本文档详细描述在Ascend 310上实现高性能应用的约束和建议，帮助用户快速理解样例，使用户能够快速构建起自己的高性能应用。

## 范围

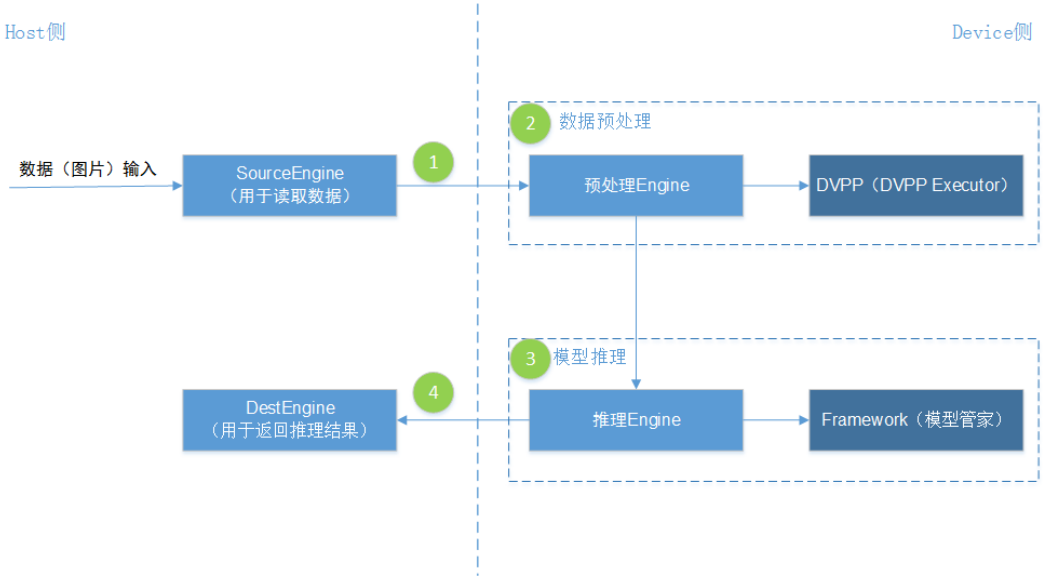
本文档主要描述Ascend 310上编程实现高性能的要点，适合在Ascend 310上使用芯片硬件解码、推理、以及通用图像处理来完成应用的用户。

# 2 要点介绍

应用开发过程中，Matrix对数据的处理流程分为以下几个阶段：

1. Matrix调用接口将数据（图片）从Host侧搬运到Device侧。
2. Matrix调用DVPP的接口对数据进行编解码、缩放等处理。
3. Matrix调用Framwork提供的模型管家接口对数据进行推理。
4. Matrix调用接口将推理结果从Device侧回传到Host侧。

**图 2-1** 处理流程图




在Matrix的数据处理过程中，主要的消耗在搬运数据、DVPP处理以及模型推理上，本文档主要介绍这几个处理环节及它们之间的衔接环节如何进行较高性能的处理。

关于各接口的详细说明，请分别参见《Matrix API参考》、《DVPP API参考》。

# 2.1 内存管理

Ascend 310支持以下两种内存管理方式：原生语言的内存管理接口和Matrix框架提供的内存管理接口。

## 2.1.1 原生语言的内存管理接口

原生语言的内存管理接口包括malloc、free、memcpy、memset、new、delete等接口，支持C/C++等语言，由此类接口申请的内存，用户可以自行管理和控制内存使用的生命周期。用户申请内存空间小于256k时，使用原生语言的内存接口与Matrix框架提供的内存管理接口在性能上区别不大，基于简单便捷考虑，建议使用原生语言的内存管理接口。

使用原生语言的内存管理接口代码示例：

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

## 2.1.2 Matrix 框架提供的内存管理接口

框架单独提供了一套内存分配和释放接口，包括HIAI_DMalloc/HIAI_DFree、HIAI_DVPP_DMalloc/HIAI_DVPP_DFree，支持C/C++语言。其中，HIAI_DMalloc/HIAI_DFree接口主要用于申请内存，再配合SendData接口从Host侧搬运数据到Device侧；HIAI_DVPP_DMalloc/HIAI_DVPP_DFree接口主要用于申请Device侧DVPP使用的内存。通过调用HIAI_DMalloc/HIAI_DFree、 HIAI_DVPP_DMalloc/HIAI_DVPP_DFree接口申请内存，能够尽量少拷贝，减少流程处理时间。

### 接口描述

关于HIAI_DMalloc/HIAI_DFree、 HIAI_DVPP_DMalloc/HIAI_DVPP_DFree接口的功能，如**表2-1**所示。

**表 2-1** 接口描述

| 接口名称 | 接口功能 |
| --- | --- |
| HIAIMemory::HIAI_DMalloc（C++专用接口） | 申请内存接口， 该内存类似普通内存，在用跨侧传输（Host-Device/Device-Host）以及模型推理处理时，性能更优。 |

| 接口名称 | 接口功能 |
| --- | --- |
| HIAIMemory::HIAI_DFree（C++专用接口） | 与HIAIMemory::HIAI_DMalloc配合使用， 用于释放HIAIMemory::HIAI_DMalloc分配的内存。<br>如果在调用HIAIMemory::HIAI_DMalloc接口时，将flag参数值设置为MEMORY_ATTR_AUTO_FREE，表示如果分配了内存，且通过SendData接口发送数据到对端，则无需调用HIAIMemory::HIAI_DFree，对端接收到数据后，内存会自动释放；如果分配了内存，但没有通过SendData接口发送数据到对端或者通过SendData接口发送数据失败，则需要调用HIAIMemory::HIAI_DFree释放内存。 |
| HIAI_DMalloc（C语言与C++通用接口） | 申请内存接口， 该内存类似普通内存，在用跨侧传输（Host-Device/Device-Host）以及模型推理处理时，性能更优。 |
| HIAI_DFree（C语言与C++通用接口） | 与HIAI_DMalloc配合使用， 用于释放HIAI_DMalloc分配的内存。<br>如果在调用HIAI_DMalloc接口时，将flag参数值设置为MEMORY_ATTR_AUTO_FREE，表示如果分配了内存，且通过SendData接口发送数据到对端，则无需调用HIAI_DFree，程序运行结束后，内存会自动释放；如果分配了内存，但没有通过SendData接口发送数据到对端或者通过SendData接口发送数据失败，则需要调用HIAI_DFree释放内存。 |
| HIAIMemory::HIAI_DVPP_DMalloc（C++专用接口） | 申请内存接口，该接口主要给Device端分配DVPP使用的内存。 |
| HIAIMemory::HIAI_DVPP_DFree（C++专用接口） | 用于释放内存， 该接口主要用于释放HIAIMemory::HIAI_DVPP_DMalloc接口分配的内存。 |
| HIAI_DVPP_DMalloc（C语言与C++通用接口） | 申请内存接口，该接口主要给Device端分配DVPP使用的内存。 |
| HIAI_DVPP_DFree（C语言与C++通用接口) | 用于释放内存， 该接口主要用于释放HIAI_DVPP_DMalloc接口分配的内存。 |

## 接口调用流程

**图 2-2** 接口调用流程

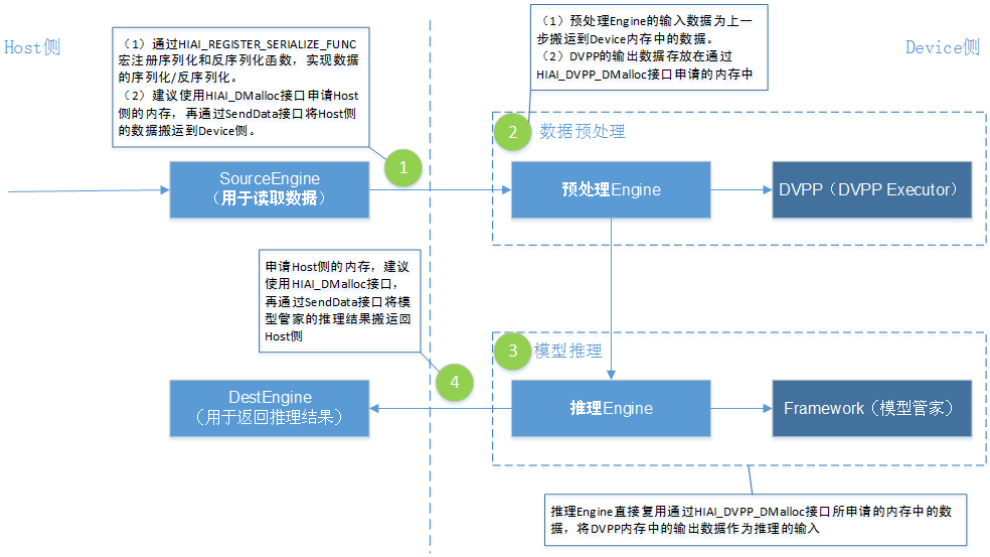



图2-2中关于接口使用的说明如下：

- 通过HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口申请的内存可用于端到端数据传输以及模型推理时使用。调用HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口，同时配合使用HIAI_REGISTER_SERIALIZE_FUNC宏（对用户自定义数据类型进行序列化或反序列化），可使数据传输效率更高，性能更优。

  通过HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口申请内存，**有以下优势**：

  - 申请的内存是可以直接给到HDC做数据搬运的， 这样可以避免Matrix与HDC间的数据拷贝。
  - 申请的内存可以直接使能模型推理零拷贝机制，减少数据拷贝时间。
- 通过HIAI_DVPP_DMalloc或HIAIMemory::HIAI_DVPP_DMalloc接口申请的内存可以给DVPP使用， 同时也可以在DVPP使用完后透传给模型推理时使用。如果不需要做模型推理，HIAI_DVPP_DMalloc接口申请的内存中的数据可以直接回传给Host侧。
- 通过HIAI_DMalloc、HIAIMemory::HIAI_DMalloc、HIAI_DVPP_DMalloc、HIAIMemory::HIAI_DVPP_DMalloc接口申请的内存兼容原生语言的内存管理接口， 可以当做普通内存使用，但是不能使用free、delete等来释放。一般来说，通过HIAI_DMalloc、HIAIMemory::HIAI_DMalloc、HIAI_DVPP_DMalloc、HIAIMemory::HIAI_DVPP_DMalloc接口申请的内存，需要分别通过调用HIAI_DFree、HIAIMemory::HIAI_DFree、HIAI_DVPP_DFree、HIAIMemory::HIAI_DVPP_DFree接口释放内存。

  如果在调用HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口时，将flag参数值设置为MEMORY_ATTR_AUTO_FREE，且通过SendData接口发送数据到对端，则无需调用HIAI_DFree或HIAIMemory::HIAI_DFree接口，对端接收到数据后，内存会自动释放；如果仅将flag参数值设置为MEMORY_ATTR_AUTO_FREE，但没有通过SendData接口发送数据到对端或者通过SendData接口发送数据失败，则需要调用HIAIMemory::HIAI_DFree释放内存。
- 通过HIAI_DVPP_DMalloc、HIAIMemory::HIAI_DVPP_DMalloc接口申请的内存是满足DVPP要求的内存，所以资源较为有限，建议仅当DVPP处理场景时使用。

## 接口使用要点

通过HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口申请内存，关于内存管理，**请注意以下要点**：

- 申请自动释放内存，用于Host到Device或Device到Host的数据传输时，如果是智能指针，由于Matrix框架自动释放内存，所以智能指针指定的析构器必须是空的；如果非智能指针，则Matrix框架自动释放。
- 申请手动释放内存，用于Host到Device或Device到Host的数据传输时，如果是智能指针，则需要指定析构器为HIAI_DFree或HIAIMemory::HIAI_DFree；如果非智能指针，则数据发送完成后需要调用HIAI_DFree或HIAIMemory::HIAI_DFree释放内存。
- 申请自动释放内存，对于该内存中的数据，不允许多次调用SendData接口发送数据。
- 申请手动释放内存时，如果用于Host到Device或Device到Host的数据传输时，在内存释放前，不可复用内存中的数据；如果用于Host到Host或Device到Device的数据传输时，在内存释放前，可以复用内存中的数据。
- 申请手动释放内存时，如果调用SendData接口异步传输数据，发送数据后，不允许修改内存中的数据。

如果调用HIAI_DVPP_MAlloc或HIAIMemory::HIAI_DVPP_DMalloc接口申请内存，用于Device到Host的数据传输时，由于HIAI_DVPP_MAlloc或HIAIMemory::HIAI_DVPP_DMalloc没有自动释放标签，所以一定需要调用HIAI_DVPP_DFree或HIAIMemory::HIAI_DVPP_DFree接口手动释放内存。如果使用智能指针存放申请的内存地址， 必须指定析构器为HIAI_DVPP_DFree或HIAIMemory::HIAI_DVPP_DFree。

## 接口调用示例

```
(1)使用性能优化方案传输数据，必须对发送数据的接口进行手动序列化和反序列化：
// 注：序列化函数在发送端使用，反序列化在接收端使用，所以这个注册函数最好在接收端和发送端都注册一遍；
//数据结构
typedef struct
{
    uint32_t left_offset = 0;
    uint32_t right_offset = 0;
    uint32_t top_offset = 0;
    uint32_t bottom_offset = 0;
    //下面serialize函数用于序列化结构体
    template <class Archive>
    void serialize(Archive & ar)
    {
        ar(left_offset,right_offset,top_offset,bottom_offset);
    }
} crop_rect;



// 注册Engine将流转的结构体
typedef struct EngineTransNew
{
    std::shared_ptr<uint8_t> trans_buff = nullptr;    // 传输Buffer
    uint32_t buffer_size = 0;                         // 传输Buffer大小
    std::shared_ptr<uint8_t> trans_buff_extend = nullptr;
    uint32_t buffer_size_extend = 0;
    std::vector<crop_rect> crop_list;
    //下面serialize函数用于序列化结构体
    template <class Archive>
```

```
    void serialize(Archive & ar)
    {
        ar(buffer_size, buffer_size_extend, crop_list);
    }
}EngineTransNewT;

//序列化函数
/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,        序列化Trans数据
* @param [in] : data_ptr         结构体指针
* @param [out]：struct_str       结构体buffer
* @param [out]：data_ptr         结构体数据指针buffer
* @param [out]：struct_size      结构体大小
* @param [out]：data_size        结构体数据大小
*/
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;
    // 获取结构体buffer和size
    buffer_size = dataLen + dataLen_extend;
    buffer = (uint8_t*)engine_trans->trans_buff.get();

    // 序列化处理
    std::ostringstream outputStr;
    cereal::PortableBinaryOutputArchive archive(outputStr);
    archive((*engine_trans));
    struct_str = outputStr.str();
}

//反序列化函数
/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,            反序列化Trans数据
* @param [in] : ctrl_ptr             结构体指针
* @param [in] : data_ptr             结构体数据指针
* @param [out]：std::shared_ptr<void> 传给Engine的指针结构体指针
*/
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>();
    // 给engine_trans_ptr赋值
    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);
        // 因为trans_buff和trans_buff_extend指向的是一块以dataPtr为首地址的连续内存空间，
        // 因此只需要trans_buff挂载析构器释放一次即可
        (engine_trans_ptr->trans_buff_extend).reset((const_cast<uint8_t*>(dataPtr + offsetLen)),
SearDeleteNothing);
    }
    return std::static_pointer_cast<void>(engine_trans_ptr);
}

// 注册EngineTransNewT
HIAI_REGISTER_SERIALIZE_FUNC("EngineTransNewT", EngineTransNewT, GetTransSearPtr, GetTransDearPtr);
```

(2) 在发送数据时，需要使用注册的数据类型，另外配合使用HIAI_DMalloc分配数据内存，可以使性能更优
注：在从host侧向Device侧搬运数据时，使用HIAI_DMalloc方式将很大的提供传输效率，建议优先使用

```
HIAI_DMalloc，该内存接口目前支持0 – (256M Bytes - 96 Bytes)的数据大小，如果数据超出该范围，则需要使用malloc接口进行分配；
    // 使用Dmalloc接口申请数据内存，10000为时延，为10000毫秒，表示如果内存不足，等待10000毫秒；
    HIAI_StatusT get_ret =  HIAIMemory::HIAI_DMalloc(width*align_height*3/2,(void*&)align_buffer, 10000);

    // 发送数据，调用该接口后无需调用HIAI_DFree接口，10000为时延
    graph->SendData(engine_id_0, "TEST_STR", std::static_pointer_cast<void>(align_buffer), 10000);
```

# 2.2 Host-Device 数据传输

```
// EngineTransNewT结构体
typedef struct EngineTransNew
{
    std::shared_ptr<uint8_t> trans_buff = nullptr;    // 传输Buffer
    uint32_t buffer_size = 0;                         // 传输Buffer大小
    std::shared_ptr<uint8_t> trans_buff_extend = nullptr;
    uint32_t buffer_size_extend = 0;
    std::vector<crop_rect> crop_list;
    //下面serialize函数用于序列化结构体
    template <class Archive>
    void serialize(Archive & ar)
    {
        ar(buffer_size, buffer_size_extend, crop_list);
    }
}EngineTransNewT;

/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,        序列化Trans数据
* @param [in] : dataPtr          结构体指针
* @param [out]：structStr        结构体buffer
* @param [out]：buffer           结构体数据指针buffer
* @param [out]：buffSize         结构体数据大小
*/
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;
    // 获取结构体buffer和size
    buffer_size = dataLen + dataLen_extend;
    buffer = (uint8_t*)engine_trans->trans_buff.get();

    // 序列化处理
    std::ostringstream outputStr;
    cereal::PortableBinaryOutputArchive archive(outputStr);
    archive((*engine_trans));
    struct_str = outputStr.str();
}

/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,            反序列化Trans数据
* @param [in] : ctrlPtr              结构体指针
* @param [in] : ctrlLen              数据结构中控制信息大小
* @param [in] : dataPtr              结构体数据指针
* @param [in] : dataLen              结构中数据信息存储空间大小，仅用于校验，不表示原始数据信息大小
* @param [out]：std::shared_ptr<void> 传给Engine的指针结构体指针
*/
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>();
```

```
    // 给engine_trans_ptr赋值
    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);
        // 因为trans_buff和trans_buff_extend指向的是一块以dataPtr为首地址的连续内存空间，
        // 因此只需要trans_buff挂载析构器释放一次即可
        (engine_trans_ptr->trans_buff_extend).reset((const_cast<uint8_t*>(dataPtr + offsetLen)),
SearDeleteNothing);
    }
    return std::static_pointer_cast<void>(engine_trans_ptr);
}
// 注册EngineTransNewT
HIAI_REGISTER_SERIALIZE_FUNC("EngineTransNewT", EngineTransNewT, GetTransSearPtr, GetTransDearPtr);
```

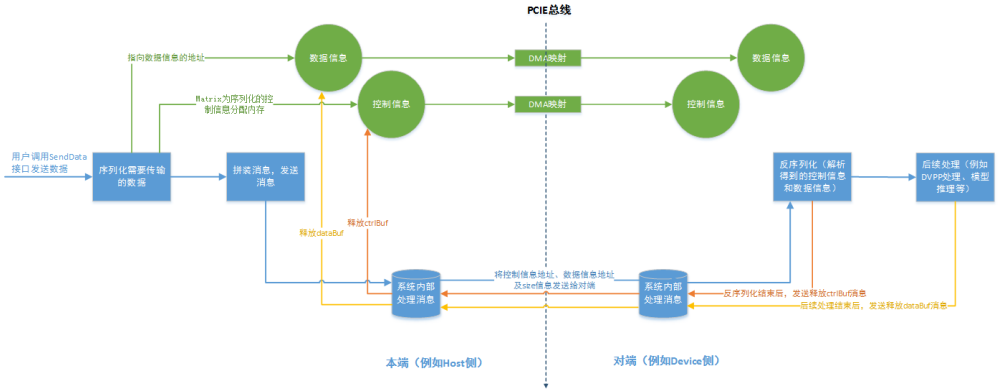
在较大图像数据或码流传输的情况下，用HIAI_REGISTER_SERIALIZE_FUNC对自定义数据类型进行序列化/反序列化，可以实现高性能数据传输，节省传输时间。

Matrix通过“控制信息 + 数据信息”的形式描述要传输的数据，控制信息指用户自定义的数据类型，数据信息指需要传输的数据内容。为保证Host和Device之间的数据传输，Matrix提供如下机制：

1. 在传输数据前，用户可调用HIAI_REGISTER_SERIALIZE_FUNC宏注册用户自定义数据类型、用户自定义序列化函数、用户自定义反序列化函数。
2. 用户在本端调用SendData接口发送数据后，Matrix会做如下处理，处理流程如图2-3所示：
   a. 调用用户自定义的序列化函数对控制信息序列化，将序列化后的控制信息放入内存（ctrlBuf）。
   b. 通过DMA（Direct Memory Access）映射将控制信息拷贝一份存放到对端的内存中，并保持本端与对端之间控制信息的映射关系。

      指向数据信息的内存（dataBuf）指针已通过SendData接口的入参传入，dataBuf是由用户调用HIAI_DMalloc/HIAI_DVPP_DMalloc接口申请的，申请该内存后系统将本端的数据信息通过DMA映射拷贝一份存放到对端的内存中，并保持本端与对端之间数据信息的映射关系。
   c. 拼装消息发送给对端，主要将ctrlBuf的地址及大小、dataBuf的地址及大小发送到对端。
   d. 对端收到消息后，Matrix会调用用户自定义的反序列化函数解析对端已获取到的控制信息和数据信息，并将解析后的数据发送给对应的接收Engine进行处理。
   e. 对端解析数据后，控制信息已使用完成，因此可以释放掉存放控制信息的内存（ctrlBuf），但由于存放控制消息的内存是在本端申请的，因此对端需要给本端发送释放ctrlBuf的消息。
   f. 本端收到消息后，释放ctrlBuf。
   g. Engine在收到数据并完成所有处理后，便可以释放dataBuf了，但由于Matrix并不知道用户何时会使用完dataBuf，因此需要用户在实现反序列化函数时，dataBuf以智能指针返回并绑定析构器hiai::Graph::ReleaseDataBuffer。当智能指针结束生命周期析构时，便会自动调用析构器给本端发送释放dataBuf内存消息。
   h. 本端收到消息后，释放dataBuf。

**图 2-3** 数据处理流程




示例场景：以下代码声明了序列化/反序列化函数，自定义了数据类型（结构体），并将自定义数据类型、序列化函数、反序列化函数通过HIAI_REGISTER_SERIALIZE_FUNC宏注册。在数据传输发起侧的数据传输前，框架调用注册的序列化函数，在数据接收侧的数据传输后，调用注册的反序列化函数。

```
// EngineTransNewT结构体
struct EngineTransNewT
{
    std::shared_ptr<uint8_t> transBuff;
    uint32_t bufferSize;   // buffer大小
    std::string url;
    //下面serialize函数用于序列化结构体
    template <class Archive>
    void serialize(Archive & ar)
    {
        ar(url);
    }
}

/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,        序列化Trans数据
* @param [in] : dataPtr          结构体指针
* @param [out]: structStr        结构体buffer
* @param [out]: buffer           结构体数据指针buffer
* @param [out]: buffSize         结构体数据大小
*/
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,        反序列化Trans数据
* @param [in] : ctrlPtr          结构体指针
```

```
* @param [in] : ctrlLen            数据结构中控制信息大小
* @param [in] : dataPtr            结构体数据指针
* @param [in] : dataLen            结构中数据信息存储空间大小，仅用于校验，不表示原始数据信息大小
* @param [out]: std::shared_ptr<void> 传给Engine的指针结构体指针
*/
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);
}
// 注册EngineTransNewT
HIAI_REGISTER_SERIALIZE_FUNC("EngineTransNewT", EngineTransNewT, GetTransSearPtr, GetTransDearPtr);
```

# 2.3 DVPP 使用

## 2.3.1 图像/视频编解码

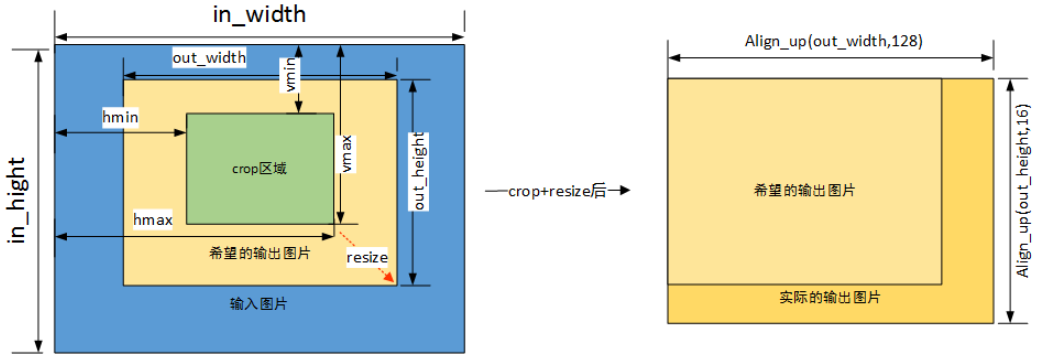
框架提供了图像处理单元以及视频编解码能力的调用接口，用户可以根据实际情况，将图像的解码/视频的解码放到Device上，以减少从Host到Device传输的数据量，同时降低数据传输时间开销和带宽压力。

在Host侧，通过调用Matrix框架提供HIAI_DMalloc申请Device侧的内存，作为图像/视频编解码的输入使用，数据存放的内存位置建议起始地址128对齐。在Device侧，DVPP完成图像/视频预处理后，调用Matrix框架提供HIAI_DVPP_DMalloc申请内存，作为图像预处理后的输出使用。

## 2.3.2 图像 Crop/Resize

在Ascend 310上编程，图像crop/resize推荐使用DVPP来实现。

**图 2-4** crop/resize 运行示意图




crop/resize运行示意图如图2-4所示，它可以完成在图像中对ROI区域进行截图并使用这个截图进行重采样的过程。截图称为crop，重采样称之为resize。当resize系数为1时，相当于只做crop。当crop为原图时，相当于只做resize。

crop/resize对输入的限制如下：

- 输入数据地址（os虚拟地址）：16字节对齐。
- 输入图像宽度内存限制：16字节对齐。
- 输入图像高度内存限制：2字节对齐。

基于上述限制，高性能的编程方式要实现“0拷贝”则需要满足从Device（接收端）给用户的内存地址开始就满足限制。一般的做法根据输入不同分为以下两种做法。

- 方法一：在Host进行解码或者在其他硬件进行解码的应用，在Host发送端将数据就做好裁剪或者padding，满足16*2对齐，这样框架在数据接收端会自动的申请满足上述限制的数据内存。

  **说明**

  示例的输入数据是YUV图片，所以计算图片SIZE都是通过(ImageWidth * ImageHeight) * 3/2的方式，其他格式的数据，根据图片格式自行调整。

```
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;

// 进行宽高对齐
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;

// 读取文件数据， 拷贝对齐内存
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);

// 进行拷贝
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
     }
}
```

- 方法二：在Device进行图片解码、视频解码以后输出为16*2对齐，可直接作为DVPP VPC(Crop&&Resize)输入。

## 2.4 模型转换预处理配置

从图1 crop/resize运行示意图中可以看到，crop/resize输出的图像是经过align_up对齐的，这种对齐会导致部分图像是经过padding的，就不是原始模型需要的输入。为了得到希望输出的图片，可以经过将这部分数据拷贝出来，放在一个新的缓冲区输入到模型推理模块，但这样引入了数据拷贝的开销。为了降低这类开销，框架提供了机制，允许输入到模型管家（modelManager）的图像带padding边，模型管家的AIPP模块会根据用户设定的宽高（在模型转换时设置），对图像进行crop，输出满足模型输入要求的图片，送到模型推理，不需要进行数据拷贝，性能得到了较大提高。

如下，假定模型推理需要输入的图像为224*224，而从DVPP的获取的数据是128*16对齐的，即256*224。

## 2.5 Batch 和超时

对于大部分模型，特别是小模型，一个批量的输入组成一个batch交给芯片做模型的推理可获得性能收益。使用batch推理将大大提高数据的吞吐率，同时也将提高芯片的利用率，在损失一定的时延情况下提升了整体的性能。因此，构建一个高性能应用应当在时延允许的情况下尽可能使用大batch。

框架为了用户能更方便、更灵活的使用batch，引入了超时机制，用户在config文件配置“is_repeat_timeout_flag”设置是否重复做超时处理，配置“wait_inputdata_max_time”设置超时时间。如果设置了超时参数，在超时后，系统调用Process函数时传入的全部是空指针，需要由用户自行编写超时处理的代码逻辑。用户编写代码逻辑时，可以在接收到数据时，将数据存储在队列中，等到足够的数据组成batch后再一并进行推理。这个队列可以使用框架提供的hiai::MultiTypeQueue。为了防止数据“饿死”在队列中，用户使用超时的设置接口，根据应用对时延的要求设置超时时间。超时时间到达时，框架会主动再次调用engine的主要处理流程。此时用户将队列中的数据取出处理，这样，数据就不会“饿死”在队列中了。

如果模型的输入是多Batch且用户分批发送各Batch的数据给模型管家（推理Engine）做推理，则需要用户添加如下代码逻辑：

1. 需要用户在Device侧单独申请缓存空间存放各Batch的数据。
2. 当Device侧的推理Engine接收到各Batch的数据后，用户需要将各Batch的数据拼接起来存放1申请的缓存空间中。
3. 等Device侧的推理Engine接收的Batch个数与模型推理需要的Batch个数相等后，用户才可以使用缓存空间中的多Batch数据进行推理。

## 2.6 算法推理输入输出数据处理

为了避免算法推理内部可能出现的内存拷贝，在调用模型管家Process接口时，建议输入数据（输入数据一般可直接使用框架传入的内存，该内存是由框架通过HIAI_DMalloc申请得到）及输出数据都通过HIAI_DMalloc接口申请，这样就能够使能算法推理的零拷贝机制，优化Process时间。如果在推理前需要进行DVPP处理，DVPP的输入内存使用框架传入的内存，输出内存可通过HIAI_DVPP_DMalloc接口分配，并将输出内存传给推理Engine当做输入内存。

## 2.7 回传数据优化处理

当推理计算完成后，需要将推理结果或者推理结束信号发送给Host端，如果在推理Engine内部调用SendData回传数据到Host端，将会消耗推理Engine的时间。建议单独开一个专门负责回传数据的Engine（例如：DataOptEngine），当推理结束后，推理Engine将处理数据透传给DataOptEngine，由DataOptEngine负责将数据回传给Host侧，再由Host侧的Engine（例如：DstEngine）负责接收传过来的推理结果。

```
//Device侧的DataOptEngine负责将数据回传给Host侧
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);
}
// Host侧的DstEngine负责接收从Device侧传过来的推理结果
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 算子使用建议

## 总体原则

基于Ascend 310芯片的特点，要提升算法的性能，就要尽量提升Cube的使用效率，相应的需减小数据搬移和Vector运算的比例。总体原则有以下几点。

1. 网络结构
   - 推荐使用主流的网络拓扑，包括ResNet、MobileNet，性能已做过调优。
   - 不推荐使用早期的网络拓扑，包括VGG、AlexNet，网络模型偏大，带宽压力大。
   - 矩阵乘法的MKN，尽量取16的倍数。算法上可以考虑适当增加channel个数，而不是分group的方式减少channel数量。
   - 增加数据复用率：一个参数的利用次数越多带宽的瓶颈越小，所以算法上可以考虑增加filter的复用次数，比如增加feature map大小，避免过大的stride或dilation。
2. Conv算子
   - 非量化模式下，Conv的输入和输出通道数建议采用16的整数倍。
   - 量化模式下，Conv的输入和输出通道数建议采用32的整数倍。
   - 量化模式下，多个Conv算子之间，建议少插入Pooling算子。
3. FC（FullConnection）算子

   当网络存在FC算子，尽量使用多batch同时推理。
4. Concat算子
   - 非量化模式下，Concat的输入通道建议采用16的整数倍。
   - 量化模式下，Concat的输入通道建议采用32的整数倍。
5. Conv融合算子

   推荐使用Conv+BatchNorm+Scale+Relu/Relu6的组合，性能已做过调优。
6. Norm算子
   - 推荐使用BatchNorm算子，使用预训练的Norm参数。
   - 不推荐使用需要在线计算Norm参数的算子，比如LRN等。
7. 检测算子

   建议使用主流的检测网络拓扑，包括FasterRCNN、SSD，性能已做过调优。

## 部分算子使用技巧

1. Conv+（BatchNorm+Scale）+Relu性能较Conv+(BatchNorm+Scale)+Tanh等激活算子好；尽量避免过于复杂的激活函数。
2. Concat算子在C维度进行拼接时，输入Tensor的Channel数均为16倍数时，性能较好。
3. FC算子在Batch数为16倍数时，性能较好。
4. 连续卷积结构性能较好，如果卷积层间反复插入较多Vector算子（如Pooling），则性能较差；这点在INT8模型中较明显。
5. 在早期AlexNet、 GoogleNet中使用了LRN作为normalization算子，该算子计算十分复杂，在算法演进过程中也逐渐被替换为BatchNorm等其他算子，在目前ResNet、Inception等主流网络结构中不再使用。针对Ascend310平台，推荐在网络中替换为BatchNorm等算子。

# 4 示例说明

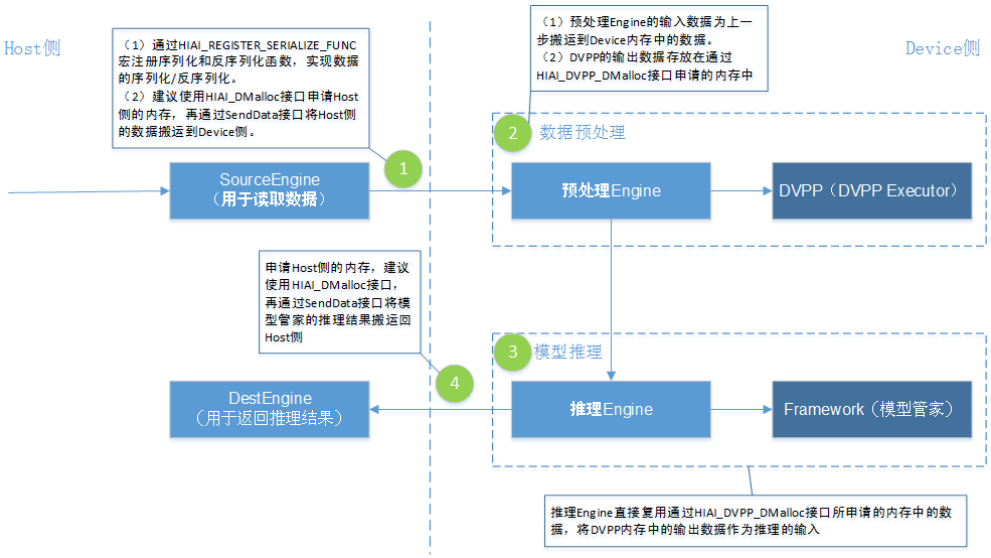
需要同时运行多个模型推理时，不建议使用多Graph，建议单Graph多batch或者单Graph多Engine。



## 4.1 数据流

示例代码完成了通过分类网络（resnet-18）处理图片的功能，数据流如图4-1所示。

**图 4-1** 接口调用流程




## 4.2 ”0“拷贝

示例代码的整个流程展示了“0”拷贝的思想。所谓的“0”拷贝，指用户在整个流程中对图像数据不做任何显式的拷贝动作。为了实现“0”拷贝，需要执行以下操作。

1. 通过HIAI_REGISTER_SERIALIZE_FUNC注册序列化函数（GetSerializeFunc）和反序列化函数（GetDeserializeFunc），实现数据类型的序列化/反序列化。

2. 使用Matrix框架提供的HIAI_Dmalloc接口申请内存，再调用SendData接口，用于将Host侧数据向Device侧搬运。
3. 通过HIAI_DMalloc接口申请的内存，作为图像/视频编解码的输入使用，无需进行数据拷贝。
4. 使用Matrix框架提供的HIAI_DVPP_DMalloc接口申请的内存，内存地址满足DVPP的输入/输出要求，可直接作为图像/视频输出的使用。调用HIAI_DVPP_DMalloc接口申请内存后，必须使用HIAIMemory::HIAI_DVPP_DFree接口释放内存。

   在DVPP内部，VPC模块的输入可直接复用内存中JPEGD模块的输出数据。
5. 通过HIAI_DVPP_DMalloc接口申请的内存，可直接作为模型推理首层的输入，无需进行数据拷贝。

# 5 FAQ



## 5.1 申请自动释放内存时使用智能指针，但析构器不为空，导致异常

### 问题现象

通过HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口申请内存（用于Host到Device或Device到Host的数据传输），将flag参数值设置为MEMORY_ATTR_AUTO_FREE（表示自动释放内存），使用智能指针存放申请的内存地址，但不定义析构器（调用默认析构器）或调用其他释放析构器（如：HIAI_DFree，会重复释放），会导致程序异常、内存泄露等情况。

### 解决方法

通过HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口申请内存（用于Host到Device或Device到Host的数据传输），将flag参数值设置为MEMORY_ATTR_AUTO_FREE（表示自动释放内存），使用智能指针存放申请的内存地址，由于Matrix框架自动释放内存，所以指定智能指针的析构器为空即可。

示例代码如下：

```
* @申请自动释放内存，如果使用智能指针，则析构器必须是空的（框架自动释放）
**/
int main()
{
    // 使用DMalloc申请自动释放
    unsigned char* inbuf = nullptr;
    uint32_t bufferLen = 1080*1920;
    // 默认是自动释放方式， flag = hiai::HIAI_MEMORY_ATTR::MEMORY_ATTR_ATUO_FREE
    HIAI_StatusT getRet = hiai::HIAIMemory::HIAI_DMalloc(bufferLen, (void*&)inbuf, 10000);
    // todo getRet 异常判断
    // todo 把需要传输的数据放到 inbuf中，大小为bufferLen。
    //组装发送数据的结构体engineTranData
    EngineTransNewT   engineTranData;
    engineTranData->bufferSize = bufferLen;
    engineTranData->transBuff.reset(std::static_pointer_cast<uint8_t *> (inbuf), [](uint8_t *)(addr )
        {
```

```
        // 析构器不做任何操作，由框架自动释放内存，如果不定义析构器，则调用默认析构器，会异常
        // 如果调用其他释放析构器，HIAI_DFree，则重复释放，也是异常
      });
   //SandData 发送数据到对端
   HIAI_StatusT sendRet =
SendData(DEFAULT_PORT,"EngineTransNewT",std::static_pointer_cast<void>(engineTranData));
// todo sendRet异常判断
}
```

# 5.2 申请手动释放内存时使用智能指针， 但未将析构器指定为 HIAI_DFree，导致异常

## 问题现象

通过HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口申请内存（用于Host到Device或Device到Host的数据传输），将flag参数值设置为MEMORY_ATTR_MANUAL_FREE（表示手动释放内存），使用智能指针存放申请的内存地址，若指定的智能指针析构器不对，会导致程序异常、内存泄露等情况。

## 解决方法

通过HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口申请内存（用于Host到Device或Device到Host的数据传输），将flag参数值设置为MEMORY_ATTR_MANUAL_FREE（表示手动释放内存），使用智能指针存放申请的内存地址，需指定智能指针析构器为HIAI_DFree或HIAIMemory::HIAI_DFree。

示例代码为：

```
/**
* @申请手动释放内存，如果使用智能指针， 需要指定析构器为 HIAI_DFree
**/
int main()
{
   // 使用DMalloc手动释放内存
   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);
   // todo getRet 异常判断

   // todo 把需要传输的数据放到 inbuf中，大小为bufferLen。
   //组装发送数据的结构体engineTranData
   EngineTransNewT   engineTranData;
   engineTranData->bufferSize = bufferLen;
   engineTranData->transBuff.reset(std::static_pointer_cast<uint8_t *> (inbuf), [](uint8_t *)(addr )
      {
         // 析构器必须调用HIAI_DFree，否则内存泄漏
        hiai::HIAIMemory::HIAI_DFree(void * addr)
      });
   //SandData 发送数据到对端
   HIAI_StatusT sendRet =
SendData(DEFAULT_PORT,"EngineTransNewT",std::static_pointer_cast<void>(engineTranData));
   // todo sendRet 异常判断
}
```

# 6 附录



## 6.1 修订记录

| 文档版本 | 发布日期 | 修改说明 |
| --- | --- | --- |
| 01 | 2020-05-09 | 第一次正式发布。 |