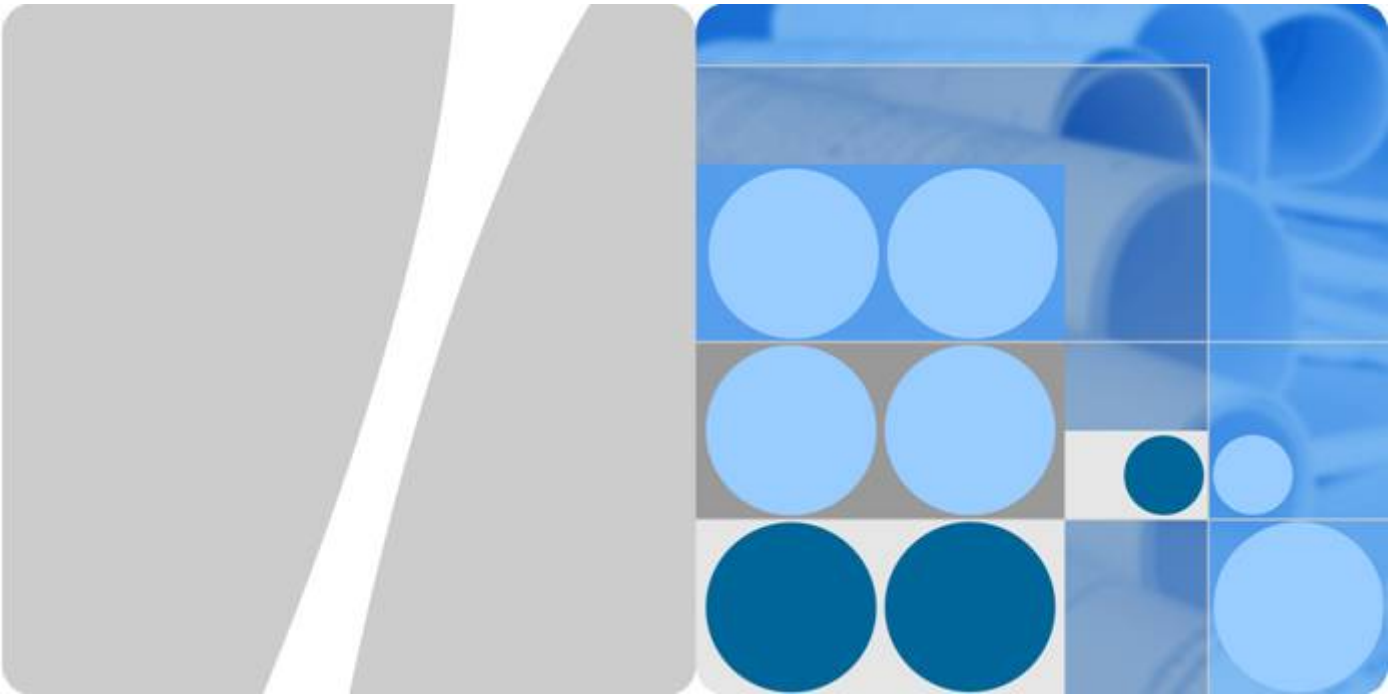


Ascend 310

V100R001

高性能应用编程用户手册

文档版本　01

发布日期　2019-03-12

华为技术有限公司

HUAWEI

商标声明


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。

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： http://www.huawei.com

客户服务邮箱： support@huawei.com

客户服务电话： 4008302118

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

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框架提供的内存管理接口。

原生语言的内存管理接口

支持原生语言的内存管理接口，如malloc、free、memcpy、memset、new、delete等，支持C/C++/python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。

使用原生语言的内存管理接口代码示例：

```
注册数据结构
HIAI_REGISTER_DATA_TYPE("EngineTransT", EngineTransT);
使用原生语言的内存管理malloc接口
unsigned char* inbuf = (unsigned char*)malloc( fileLen );
调用发送接口发送数据
ret = SendData(0, "EngineTransT", output);
```

框架提供的内存管理接口

框架还提供了一种新的内存管理方式，用户通过调用框架提供的HIAI_DMalloc（HIAIMemory::DMalloc）、HIAI_DFree（HIAIMemory::DFree）接口进行内存的获取和释放，用户可以自行管理和控制内存的使用生命周期。该接口申请的内存兼容原生语言的内存管理接口，可以当做普通内存直接使用，但不能使用free、delete来释放内存。

表 2-1 HIAI_DMalloc 使用场景

使用场景	是否需要HIAI_Dfree
使用HIAI_DMalloc申请内存， 并通过SendData发送数据到对端	可以不调用HIAI_Dfree, 框架会自动管理该内存
使用HIAI_DMalloc申请内存，但不调用SendData发送数据到对端	需要用户自己调用HIAI_Dfree 去释放该内存

使用场景	是否需要HIAI_Dfree
使用HIAI_DMalloc申请内存， 传入Flag = HIAI_MEMORY_ATTR_MANUAL_FREE	用户手动管理， 无论是否调用 SendData 发送数据到对端，都需要调用HIAI_Dfree 去释放该内存

框架推荐使用智能指针来管理内存，并推荐用户在需要做数据较大的Host到Device数据搬运时（例如图像数据），通过框架提供的接口申请和释放内存，调用SendData接口进行Host和Device之间的数据传输。由于在底层进行了优化，在Host和Device的数据传输上，框架将拥有更强大的性能。

使用框架提供的内存管理接口代码示例：

```
注册数据结构的序列化和反序列化函数
HIAI_REGISTER_SERIALIZE_FUNC("EngineTransNewT", EngineTransNewT, GetTransSearPtr, GetTransDearPtr);
使用框架的内存管理DMalloc接口
HIAI_StatusT get_ret = hiai::HIAIMemory::DMalloc(fileLen, (void*&)inbuf, 10000);
调用发送接口发送数据
ret = SendData(0, "EngineTransNewT", output);
```

2.2 Host-Device 数据传输

Host和Device之间的数据传输，框架提供如下机制。

- 发送端和接收端都需要通过HIAI_REGISTER_SERIALIZE_FUNC注册需要发送的数据。
- 发送端源地址的内存由用户申请。
- 接收端目的地址的内存由框架自动申请。

为保证跨平台通用性，框架会将传输的数据默认在发送端序列化并在接收端反序列化出来，但这种方式对于已知数据可兼容的部署平台来说，数据较大的图像/码流数据的序列化/反序列化是没有必要的且会产生大量的开销。为避免上述没必要的开销，框架提供了用户自定义序列化/反序列化数据块的功能，通过宏HIAI_REGISTER_SERIALIZE_FUNC注册。该宏为我们需要传输的数据载体（结构体）指定了一个序列化和反序列化的函数指针，用户在实现这两个函数的时候，对于对象本身进行序列化/反序列化（若平台一致可直接强转），但数据的指针可根据host和Device的平台考虑是否直接赋值。

例如，以下代码声明了序列化/反序列化函数，并注册到了某结构体上。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´óÐ¡
}

/**
* @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]: dataSize         结构体数据大小
```



```
*/
void GetTransSearPtr(void* dataPtr, std::string& structStr, uint8_t*& buffer, uint32_t& bufferSize)
{
    EngineTransNewT* engineTrans = (EngineTransNewT*)dataPtr;
    structStr  = std::string((char*)dataPtr, sizeof(EngineTransNewT));
    buffer = (uint8_t*)engineTrans->transBuff.get();
    bufferSize = engineTrans->bufferSize;
}

/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,              反序列化Trans数据
* @param [in] : ctrl_ptr              结构体指针
* @param [in] : ctrlLen               数据结构中控制信息大小
* @param [in] : data_ptr              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>();
    engineTransPtr->bufferSize = ((EngineTransNewT*)ctrlPtr)->bufferSize;
    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。

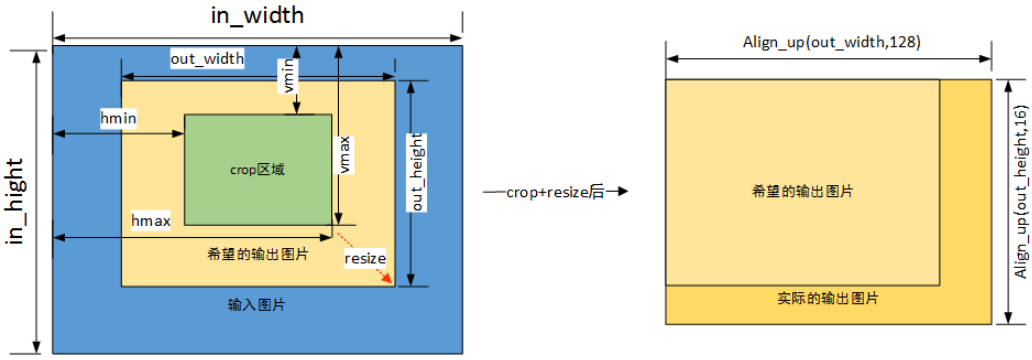
图像/视频编解码的输入数据存放的内存位置建议起始地址128对齐，通过**2.1章节**的介绍，我们知道接收端的内存是由框架申请的，该内存已经满足前面两个限制，可直接作为图像/视频编解码的输入使用。除需要对Host传输上来的数据做改变等特殊场景，框架不推荐用户直接使用mmap映射内存空间。

详细请参见**3.3示例代码**sample_device.cpp中JpegdEngine类的实现。

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 2-1 crop/resize 运行示意图



crop/resize运行示意图如图1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。截图我们称之为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：

- 输入图像宽度：128像素对齐。
- 输入图像高度：16像素对齐。
- 输入数据地址：虚拟地址128字节对齐（1024位对齐），若不满足该限制，DVPP内部会进行内存拷贝。

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128*16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 = 128;
static const uint32_t ALIGN_H = 16;
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进行解码的应用，图片解码输出为128*16对齐，直接作为输入。视频解码输出因此也需要通过 vpc进行图片的剪裁缩放，输出为128*16对齐，可直接作为DVPP VPC(Crop&&Resize)输入。详情请参见**3.3示例代码**src文件sample_device.cpp中JpegdEngine类的实现。

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，这就意味着会有部分数据是经过padding出来的，它不是NN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，模型管家根据用户设定的宽高自动的只将需要的图像数据送到模型推理。用户设定的宽高在通过模型转换过程中转换参数设置实现。

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。配置如下.

```
    int32_t jpegdoutWidth = 256;
    int32_t jpegdoutHeight = 224;
    int32_t outImageSize = JpegdoutWidth * JpegdoutHeight * 3/2;
    hiai::Rectangle<hiai::Point2D> rectangle;
    rectangle.anchor_lt.x = 0;
    rectangle.anchor_lt.y = 0;
    rectangle.anchor_rb.x = JpegdinWidth - 1;
    rectangle.anchor_rb.y = JpegdinHeight - 1;
    // 使用AutoBuffer
    if (outBuffer_ == nullptr) {
        outBuffer_ = make_shared<AutoBuffer>();
    }
    uint32_t outWidth = MODEL_WIDTH;  // 224
    uint32_t outHeight = MODEL_WIDTH; // 224
    auto ret = CropAndResizeByDvpp(dvpp_api, inImage,(char*)vpcOutBuffer, outImageSize, rectangle,
outWidth, outHeight);
```

调用模型管家时，图像宽高为256*224，模型推理首层输入的宽高为224*224，并且不需要进行数据拷贝，性能得到了较大提高。

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会使我们获得性能收益。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“timeout_info”设置是否超时等待，配置“wait_inputdata_max_time”设置超时时间。用户在接受到数据时可以暂时不处理，直接返回。仅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静态或者动态的设置超时时间。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。

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，并通过CreateTensor接口将内存传给推理模块， 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_DMalloc(bufferSize, buffer, 1000, HIAI_MEMORY_HUGE_PAGE) 分配， 并将输出内存传给推理Engine当做输入内存。

以下为模型推理Engine使能算法推理零拷贝的样例代码：

```
// 推理Engine Process函数实现
HIAI_IMPL_ENGINE_PROCESS("ClassifyNetEngine", ClassifyNetEngine, CLASSIFYNET_ENGINE_INPUT_SIZE)
{
    // 获取从上个Engine传入的数据
    std::shared_ptr<EngineTransNewT> input_arg = std::static_pointer_cast<EngineTransNewT>(arg0);
    // 如果传入数据为空指针，直接返回
    if (nullptr == input_arg){
        HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG, "fail to process invalid message");
        return HIAI_INVALID_INPUT_MSG;
    }
    // 准备输入数据， 如果确保传入的数据是通过HIAI_DMalloc申请或者是直接从框架传入的，则直接给到模型推理输入
    hiai::AITensorDescription inputTensorDesc = hiai::AINeuralNetworkBuffer::GetDescription();
    shared_ptr<hiai::IAITensor> inputTensor = hiai::AITensorFactory::GetInstance()->CreateTensor(inputTensorDesc,
        input_arg->trans_buff.get(), input_arg->buffer_size);
    // AIModelManager填充输入数据
    inDataVec.push_back(inputTensor);
    // 准备输出数据， 输出数据使用HIAI_DMalloc接口分配内存， 并通过CreateTensor给到算法推理，如果使用同步的机制
    // 调用推理的Process，则只需要分配一次输出内存
    if (preOutBuffer == false) {
        std::vector<hiai::TensorDimension> inputTensorVec;
        std::vector<hiai::TensorDimension> outputTensorVec;
        // 获取输出Vec的SIZE
        ret = ai_model_manager_->GetModelIOTensorDim(modelName, inputTensorVec, outputTensorVec);
        if (ret != hiai::SUCCESS)
        {
            HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_INIT_FAIL,
                "hiai ai model manager init fail");
            return HIAI_AI_MODEL_MANAGER_INIT_FAIL;
        }
        // 预分配OutData;
        HIAI_StatusT hiai_ret = HIAI_OK;
        for (uint32_t index = 0; index < outputTensorVec.size(); index++) {
            hiai::AITensorDescription outputTensorDesc = hiai::AINeuralNetworkBuffer::GetDescription();
            uint8_t* buffer = nullptr;
            // HIAI_Dmalloc分配内存， 该内存主要是给算法进行推理，需要调用HIAI_DFree释放，在这边记录内存地址，
            // Engine析构时释放
            hiai_ret = hiai::HIAIMemory::HIAI_DMalloc(outputTensorVec[index].size, (void*&)buffer, 1000);
```



```
            if (hiai_ret != HIAI_OK || buffer == nullptr) {
                continue;
            }
            outData_.push_back(buffer);
            shared_ptr<hiai::IAITensor> outputTensor =
                hiai::AITensorFactory::GetInstance()->CreateTensor(outputTensorDesc, buffer,
outputTensorVec[index].size);
            outDataVec_.push_back(outputTensor);
        }
        preOutBuffer = true;
    }
}
 ClassifyNetEngine::~ClassifyNetEngine() {
    // 释放outData预分配内存
    HIAI_StatusT ret = HIAI_OK;
    for (auto buffer : outData_) {
        if (buffer != nullptr) {
            ret = hiai::HIAIMemory::HIAI_DFree(buffer);
            buffer = nullptr;
        }
    }
 }
```

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。建议单独开一个传输Engine专门负责回传数据，当推理结束后， 推理Engine将处理数据透传给传输Engine，由传输Engine负责回传给Host端。

```
// 该Engine专门负责回传数据给到Host端， 这样将减少推理Engine回传的时间消耗;
HIAI_IMPL_ENGINE_PROCESS("TransEngine", TransEngine, CLASSIFYNET_ENGINE_INPUT_SIZE) {
    std::shared_ptr<std::string> data_result_ptr =
        std::static_pointer_cast<std::string>(arg0);
    // 判断data_result_ptr是否有效
    if (nullptr == data_result_ptr)
    {
        HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG, "fail to process invalid message");
        return HIAI_INVALID_INPUT_MSG;
    }
    hiai::Engine::SendData(0, "string", std::static_pointer_cast<void>(data_result_ptr));
}
```


3 算子使用建议

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. 提升Cube的数据利用效率

   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的方式来减小。

2. 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，反之（比如不带batch的全连接，一个参数只用一次）则带宽瓶颈越大。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。

3. 减小vector的计算比例

   尽量避免过于复杂的normalization（如LRN）和激活函数。

部分算子使用技巧.

- Conv+(BatchNorm+Scale)+Relu性能比Conv+(BatchNorm+Scale)+Tanh等激活算子好。
- 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。
- FC算子在Batch数为16倍数时，性能较好。
- 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，在INT8模型中较明显。
- 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

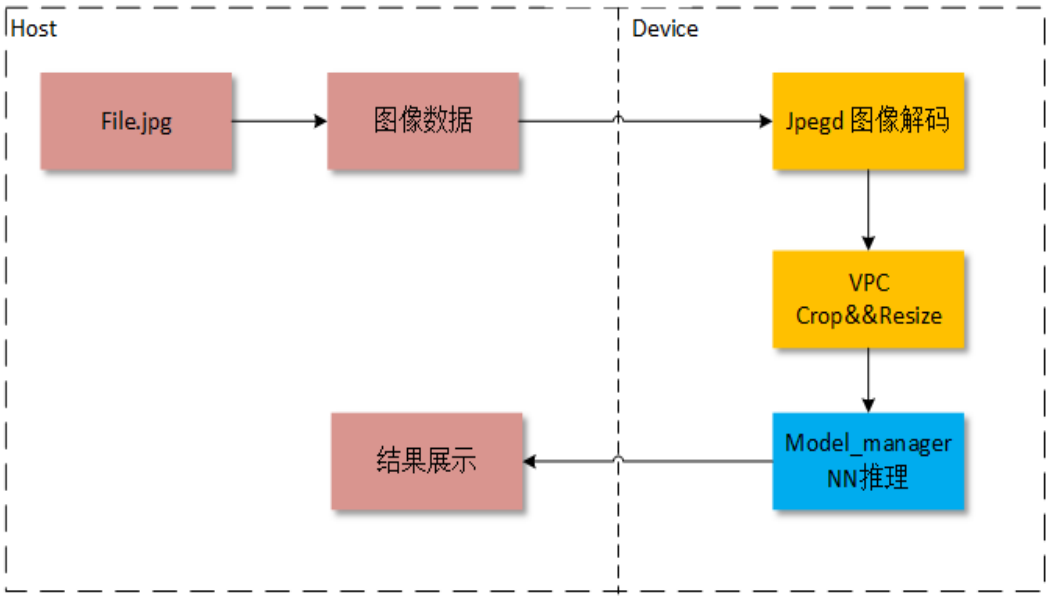
说明

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-50）处理图片的功能，数据流如图4-1所示。

图 4-1 处理图片数据流



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注册序列化和反序列化函数。

2. 实现结构体序列化/反序列化（GetSerializeFunc&&GetDeserializeFunc）函数。
3. 使用“HIAI_Dmalloc”分配发送数据的内存。

4.3 示例代码

请参见该文档zip包中resource文件夹下的performance_sample.rar文件。

A 缩略语

D		
Device	Device	设备，这里指Mini
DVPP	Digital Vision Pre-Process	数字视觉预处理
H		
host	host	主机，这里指搭载mini的计算板
J		
JPEGD	Joint Photographic Experts Group Decoder	JPEG格式图像解码器
P		
PNGD	Portable Network Graphics Decoder	便携式网络图形解码器
V		
VPC	Vision preprocess core	视觉处理核

