云天励飞DESDK应用软件开发指南

版本：v1.3.0

更新记录

版本	修改日期	修改说明
V1.0.0	2020.9.7	初始版本
V1.1.0	2020.12.1	合并原用户手册和快速上手手册内容，适配v1.6.0版本交付结构修改
V1.1.1	2021.2.4	增加同步模型推理（同步接口）、芯片RPC服务程序代码示例，简化目录结构
V1.2.0	2021.4.28	增加模型精度测试示例，更新过时的路径
V1.3.0	2021.8.20	更新python接口示例，使用InferEngine接口

目录

1 概述

本文档适用于基于DESDK接口进行应用开发的人员，通过本文档您可以：

- 了解DESDK应用开发的基本方法和典型开发流程
- 了解DESDK的高效率编程方法，充分利用芯片硬件资源

2 DESDK介绍

2.1 软件架构

如下图所示，DESDK整体框架分为以下几层：

- 硬件抽象层（HAL）

  实现对芯片硬件资源 (如USB、I/O、Ai驱动适配等) 的通用配置，隐藏具体的操作细节，为上层提供简单清晰的调用接口。

- 功能模块层（FML）

  通过调用 HAL，实现项目中所涉及到的各片外功能模块，隐藏具体的模块操作细节，并为上层提供简单清晰的调用接口。

- 定制组合模块层（CML）

  提供定制化功能层以及组合的FML层（便于用户更快捷的开发）。

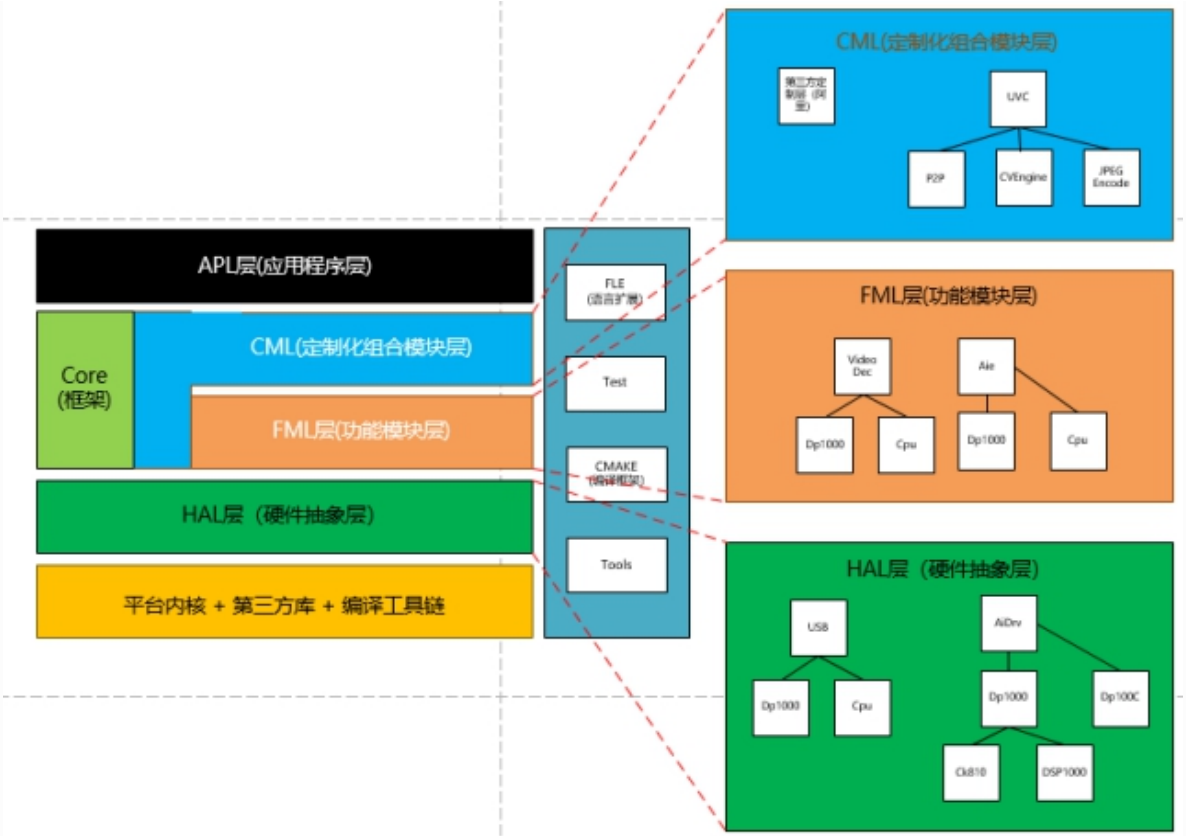
- 应用程序层（APL）

  用户的应用程序所在层，通过调用 HAL 与 FML，实现最终的应用功能。

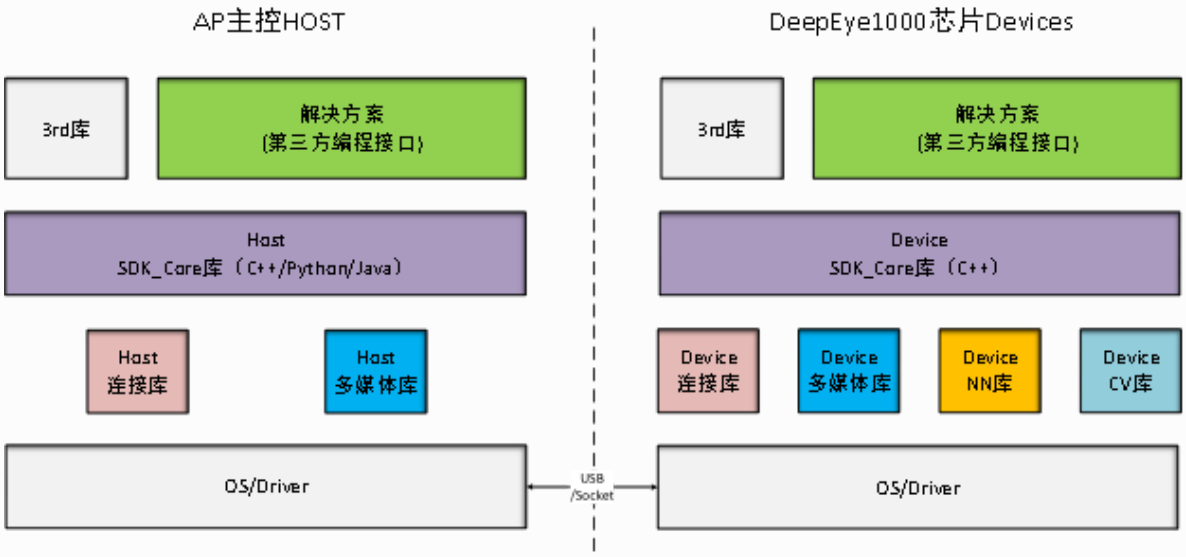
- 内核模块（CORE）

  指各层公用的软件中间模块，包含P2P传输，log打印等。

目前DESDK主要提供CORE和FML层功能接口。



2.2 编程接口



SDK编程接口如上图所示，支持两种编程接口方式，用户根据具体应用需求选择合适的编程接口。

Host主控SDK编程接口，提供C++、Python语言接口。用户通过主控SDK来控制Host和Device端的连接通信方式、芯片的图片编解码以及视频流的解码、芯片的NN网络同步调用以及芯片的视觉CV同步调用；该方式编程接口提供基本的同步调用功能，简单易用，但受限于协处理器的控制、数据交互效率，芯片的资源利用率低。

Device芯片SDK编程接口，提供C++语言接口。用户可以通过该接口直接调用芯片的图片编解码以及视频流的解码、芯片的NN网络异步调用以及芯片视觉的CV异步调用；该方式下基于多线程流水异步调度，避免数据在Host端和Device频繁交互，充分利用芯片的资源。

对于Host端/Device端的编程接口选择而言，建议如下：

1. Host端的编程接口主要适用于数据交互少、延迟要求不高的接口设计，比如资源管理、模块初始化参数配置、控制Device端的多线程流水设计等；
2. Device端的编程接口主要适用于实时延迟低的、数据交互频繁的接口设计，比如实时视频流的人脸检测、跟踪算法，芯片的视频流解码、检测NN调度和跟踪CV调度完全适合流水设计，只能在Device端接口进行编程控制；

2.3 语言支持

芯片侧仅支持C++语言接口，gcc 6.3.0以上，支持C++11特性。

主控侧目前支持C++语言，gcc 4.8.5以上，支持C++11特性。支持Python语言模型推理。

2.4 平台支持

芯片侧为中天微csky平台，操作系统为linux，不可修改，可进行二次开发。

主控侧默认支持linux-x64，支持ubuntu16.04及以上系统。其他平台可通过平台增量包定制获取。

linux-x64平台建议直接使用DeAI中提供的docker环境进行开发部署。

2.5 接口调用方式

DESDK是异构SDK系统，主控侧和芯片侧接口基本相同，所有接口都在de命名空间下。有两种调用方式：

2.5.1 GetFunc方式

以函数名作为参数通过GetFunc函数调用，该函数必须通过DE_CLASS_REGISTER(OpName, Type)进行注册。这种方式既可以本地调用，也可以跨系统调用，跨系统调用相当于RPC调用，当device端函数返回后，host端函数再返回。

在主控端调用Local::GetFunc时获取的是本地函数，调用Device::GetFunc时获取的是芯片侧函数。在芯片侧只能调用Local::GetFunc，获取的是芯片本地的函数。

例如，从主控侧创建芯片侧JPEG编码器，代码如下：

```
auto enc = Device::GetFunc(0, "de.mm.jpeg.encoder.create")(7);
```

2.5.2 直接调用方式

如传统SDK的调用方式一样，直接使用方法名调用，此种方式只能用在芯片侧。

例如，从芯片侧创建JPEG编码器，代码如下：

```
auto enc = new de::mm::JpegEncoder(7);
```

3 应用开发介绍

3.1 开发流程

3.1.1 环境搭建

使用DESDK开发基于DeepEye芯片的人工智能应用，建议在DEngine_docker下进行，在docker环境下已经配置好了链接和运行的环境变量。docker环境的搭建参见《云天励飞DEngine集成开发指南》环境搭建章节。

所有代码示例均可在DEngine_docker环境下成功编译运行。如果是其他环境，需要自行安装依赖软件，可参考/DEngine/tools/env下的环境配置脚本。示例需要依赖的软件主要有：

软件名	版本	说明
cmake	3.10	用于编译c++示例代码
gcc/g++	4.8.5	用于编译c++示例代码
opencv	3.1.0	用于图片读取、解码、打框和保存，主要在图片检测、视频检测示例中使用
ffmpeg	4.1.4	用于视频取流显示，主要在视频检测、视频检测跟踪示例中使用
gstreamer	1.14.5	用于VideoInput标准Node取流，如自行取流可不依赖
python	3.6	用于python推理示例
opencv-python	4.1.2	用于python推理示例
decorator	4.4	用于python推理示例
xlsxwriter	1.3.7	用于生成benchmark统计表格

3.1.2 工程编译

3.1.2.1 创建工程目录

原则上用户可以在DEngine下的任何位置创建工程目录，并自行搭建工程框架。我们推荐在DEngine/solution下创建工程目录，或者将其他目录映射到DEngine/solution下。以HelloWorld工程为示例，src目录存放应用软件源码，bin目录存放编译后的可执行文件，lib目录存放编译后的依赖库。原始模型文件和编译后的模型文件均放在model文件夹下，以模型名区分。

```
├── DEngine
│   ├── solution
│   │   ├── HelloWorld //该目录下存放应用软件相关的源码、脚本、配置文件
│   │   │   ├── host //该目录下存放应用软件主控侧相关源码、库和可执行文件
│   │   │   │   ├── src
│   │   │   │   ├── bin
│   │   │   │   ├── lib
│   │   │   ├── dev //该目录下存放应用软件芯片侧相关源码、库和可执行文件
│   │   │   │   ├── src
│   │   │   │   ├── bin
│   │   │   │   ├── lib
│   │   │   ├── model //该目录下存放模型转换相关的配置文件、模型文件
│   │   │   │   ├── xxx //该目录下为具体模型
│   │   │   │   │   ├── xxx.ini
│   │   │   │   │   ├── xxx.prototxt
```

3.1.2.2 编译框架

DESDK中的示例代码均使用CMake作为编译框架。DEngine_docker中已经安装了可用的CMake版本。

用户在编译自己的工程时，可以直接基于示例代码的CMakeLists.txt文件修改，也可自行使用其他编译框架。编译框架文件中的主要内容是设置编译使用的编译链，依赖的头文件和库文件。

3.1.2.3 设置编译链

如果编译平台与部署平台相同，可以直接编译。docker中gcc有两个版本，gcc-4.8和gcc-7，默认为gcc-4.8，可通过以下命令切换到gcc-7：

```
# update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-7 60
# update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-7 60
```

然后可通过以下命令还原到gcc-4.8

```
# update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-7 40
# update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-7 40
```

在linux-x64环境下编译非linux-x64环境的程序需要使用交叉编译工具链，请根据《云天励飞DEngine集成开发指南》中的指导单独下载工具链并配置环境。

编译芯片侧程序和库需要用到DeepEye芯片交叉工具链csky-toolchains，编译脚本默认其放在/opt/csky-toolchains目录下，DEngine_docker在进入时会将宿主机的/opt/csky-toolchains映射到容器内，可直接使用。

在CMakeLists中指定编译器

```
if("${TARGET_CPU}" STREQUAL "dp1000")
  add_definitions(-DUSE_DP1000_PLATFORM)
  SET(CMAKE_SYSTEM_NAME Linux)
  set(CROSS_PREFIX "csky-abiv2-linux-")
endif()

if(DEFINED CROSS_PREFIX)
  set(CROSS_TOOLCHAIN_PREFIX ${CROSS_PREFIX})
  set(CMAKE_CXX_COMPILER "${CROSS_TOOLCHAIN_PREFIX}g++")
  set(CMAKE_C_COMPILER "${CROSS_TOOLCHAIN_PREFIX}gcc")
  set(CMAKE_STRIP "${CROSS_TOOLCHAIN_PREFIX}strip")
endif()
```

3.1.2.4 设置链接库

DEngine包中，平台的链接库在DEngine/desdk/platform文件夹中按照平台名存放。在DEngine_docker中，可通过环境变量切换编译平台，支持的环境变量有：

- TARGET_TYPE，目标类型，支持以下2种：
    - host：表示主控侧
    - dev：表示设备侧，设备可能是芯片或者仿真环境
- TARGET_OS，目标操作系统，可能为以下几种：
    - linux：表示linux内核的系统，与平台相关，在通用的服务器和ARM上默认为ubuntu，在专用的嵌入式平台上一般为该平台的默认linux系统。
    - android：表示android系统，使用ndk编译。

    - win7/win10：表示windows各版本系统。
    - kylin/uos：麒麟/统信等国产操作系统。
    - 其他定制的操作系统。
- TARGET_CPU，目标CPU，可能为以下几种：
    - x64/x86：通用x64/x86架构处理器。
    - armv7/armv8：通用armv7/armv8架构处理器，及支持的嵌入式平台，如海思、瑞芯微等。
    - dp1000：表示DeepEye1000芯片平台。

目前标准DEngine包中的platform中默认提供host_linux_x64和dev_linux_dp1000两个平台。平台的命名规则为<TARGET_TYPE>_<TARGET_OS>-<TARGET_CPU>。

- 如果要编译主控linux x64平台（默认平台，平台名称为host_linux-x64）下的代码，可以设置

```
# export TARGET_TYPE=host
# export TARGET_OS=linux
# export TARGET_CPU=x64
```

  或者修改docker_enter.sh中的环境变量参数

```
--env TARGET_TYPE=host --env TARGET_OS=linux --env TARGET_CPU=x64
```

  此时example会使用platform/host_linux-x64目录下的库进行链接，编译出host_linux-x64平台下的可执行程序和库。

- 如果要编译DeepEye1000芯片平台下的代码，可以设置

```
# export TARGET_TYPE=dev
# export TARGET_OS=linux
# export TARGET_CPU=dp1000
```

  或者修改docker_enter.sh中的环境变量参数

```
--env TARGET_TYPE=dev --env TARGET_OS=linux --env TARGET_CPU=dp1000
```

  此时example会使用platform/dev_linux-dp1000目录下的库进行链接，编译出dev_linux-dp1000平台下的可执行程序和库。

- 如果要编译其他平台下的代码，参考上面的示例设置环境变量。目标系统和目标CPU的代号可从定制平台命名中根据命名规则获取，或者与云天励飞DeepEye芯片技术支持人员联系。

3.2 代码开发

3.2.1 开发方式

DESDK目前提供两种开发方式：基本开发模式和Graph模式。

3.2.1.1 基本开发模式

基本开发模式使用一般的SDK接口调用的方式，为应用提供基本的初始化、数据传输、设备管理以及图像处理、模型推理等业务接口，参见《云天励飞DESDK API手册》。

3.2.1.2 Graph开发模式

Graph模式主要为业务开发提供基于图的实现方式。参见《云天励飞DESDK Graph编程手册》，将所有过程封装为一个个功能Node，通过Pin和Pout串接起来。串接Node有两种方式：

1. 通过调用Graph API接口串接
2. 通过加载Graph配置文件串接

两种方法效果是一样的，用户可以自己选择。Graph配置文件是Json格式，如果开始不会写可以通过API接口写个初始的，运行后会生成对应的Graph配置文件，以后在此基础上修改。

Graph模式下的开发步骤为编写业务Node和串接Graph。

3.2.1.2.1 编写业务Node

每个Node封装为一个Node类，所有Node类都继承于de::Thread类。

以jpeg_dec的example中的代码为例，主控侧创建两个Node用于发送和接收数据，分别为Sender和Receiver。在构造函数中指定Node输入输出的数据结构及序列化、反序列化和析构方法。

```
//发送node，透传数据到设备node
class Sender : de::Thread {
 public:
 Sender(){
  pin.SetTypeInfo(0, "de::JpegDecTask", de::TaskDeSerializeCustom<de::JpegDecTask>,
de::TaskDeleter<de::JpegDecTask>);
  pout[0].SetTypeInfo("de::JpegDecTask", de::TaskSerializeCustom<de::JpegDecTask>);
 }
 void Proc(void* rx_task, int32_t task_type){
  de::JpegDecTask* pTask = static_cast<de::JpegDecTask*>(rx_task);
   FILE *file = fopen("tmp.jpg", "w");
   fwrite((char*)pTask->array.GetTensorData(), pTask->array.GetTensorDataSize(), 1,
file);
   fclose(file);
  pout[0].SendTask(pTask);
 }
 void Stop(void){pin.DestroyQueue();}
};
//接收node，从设备node透传数据
class Receiver : de::Thread {
 public:
 Receiver(){
  pin.SetTypeInfo(0, "de::JpegDecTask", de::TaskDeSerializeCustom<de::JpegDecTask>,
de::TaskDeleter<de::JpegDecTask>);
  pout[0].SetTypeInfo("de::JpegDecTask", de::TaskSerializeCustom<de::JpegDecTask>);
 }
 void Proc(void* rx_task, int32_t task_type){
  de::JpegDecTask* pTask = static_cast<de::JpegDecTask*>(rx_task);
  pout[0].SendTask(pTask);
 }

 void Stop(void){pin.DestroyQueue();}
```

```
};
```

通过DE_CLASS_REGISTER注册为全局Node，以便Graph通过名称调用。

```
namespace de
{
DE_CLASS_REGISTER("Sender", Sender);
DE_CLASS_REGISTER("Receiver", Receiver);
}
```

3.2.1.2.2 串接Graph

目前可以通过两种方法将Node串接为Graph，完成业务流程。

3.2.1.2.2.1 通过Graph API接口串接

以jpeg_dec的example中的代码为例，首先创建de::Graph对象graph，使用graph对象创建核间通信Bridge，再创建往返2条通信通道。创建主控和芯片侧Node，设置连接，最后将输入Node的输入和输出Node的输出设置到Graph的输入输出上，启动业务，示例代码如下：

```
 //创建graph
 de::Graph graph("jpeg_dec");

 //创建bridge，连接输入和解码2个node
 graph.CreateBridge(0);
 graph.AddBridgeH2DChan(0, "Host2Device");
 graph.AddBridgeD2HChan(1, "Device2Host");

 //创建node
 graph.CreateHostNode("Sender", "jpeg_in");
 graph.CreateHostNode("Receiver", "yuv_out");
 graph.CreateDevNode("de::JpegDecoderNode", "jpeg_dec");

 //设置graph内node连接，jpeg_in的Pout[0]连接到jpeg_dec的Pin，使用bridge通道0
 graph.LinkNode("jepg_in", 0, "jpeg_dec", 0);

 //设置graph内node连接，jpeg_dec的Pout[0]连接yuv_out的Pin，使用bridage通道1
 graph.LinkNode("jpeg_dec", 0, "yuv_out", 1);

 //设置graph输出，graph的Pout[0]为yuv_out的Pout[0]
 graph.SetOutputNode(0, "yuv_out", 0);

 //设置graph输入，graph的Pin[0]为jpeg_in的Pin
 graph.SetInputNode(0, "jpeg_in");

 //启动业务
 graph.Start();
```

3.2.1.2.2.2 通过json配置文件读取

编辑好Graph配置的json文件，通过Graph的FromFile方法读取后启动，示例代码如下：

```
//创建graph
de::Graph graph("jpeg_dec") = de::Graph::FromFile(cfg_file);

//启动业务
graph.Start();
```

3.2.2 内存管理

3.2.2.1 基本数据结构

DESDK使用de::NDArray作为图像和数据保存的基本数据结构，相关API接口都是用此结构作为数据传输的参数类型。de::NDArray支持核间引用管理和零拷贝序列化，参见《云天励飞DESDK API手册》数据类型章节。

3.2.2.2 芯片内存管理

DESDK提供了芯片端内存申请和释放接口管理芯片端内存，通过接口申请的内存是连续的而且可以获取到物理地址，可以直接传递给硬件进行读取，效率很高，适合较大的后继需要硬件加入处理的数据，例如待处理的图像数据等。参见《云天励飞DESDK API手册》内存接口章节。

主控端可以通过“方法获取”接口通过RPC调用芯片端的内存管理接口实现对芯片端内存的管理，参见《云天励飞DESDK API手册》流程接口中的方法获取章节。

3.2.2.3 核间内存管理

DESDK支持主控通过引用管理芯片端内存。例如，主控端de::NDArray对象中保存的可能是芯片端的引用，当主控端对象析构时会通知芯片端释放引用，当引用计数为0时芯片端会自动释放内存。因此，对带引用的对象进行传递时需要考虑引用对象的保存问题，避免指向的内存被自动释放。

3.2.3 视频取流

DESDK支持H264/H265视频取流，可以在主控或芯片端执行，使用软件实现，只支持Graph模式，作为解码Node的前置Node。

Graph模式下封装为VideoInput标准算子，支持本地文件、RTSP流地址输入，参见《云天励飞DESDK Graph编程手册》标准算子-视频输入算子章节。

3.2.4 图像编解码

DeepEye1000硬件提供了硬件的编解码能力，包括视频解码、JPEG图片的编解码。

3.2.4.1 视频解码

DeepEye1000芯片包含视频解码硬件模块，支持H264/H265视频解码，能力可达1080P 120fps或4K 50fps。Graph模式下封装为VideoDecoder标准算子，参见《云天励飞DESDK Graph编程手册》标准算子-视频解码章节。

解码功能也可以通过API调用，参见《云天励飞DESDK API手册》视频解码章节。

3.2.4.2 JPEG解码

DeepEye1000芯片包含JPEG解码硬件模块，能力可达1080P 130fps或4K 45fps。

Graph模式下封装为JpegDecoder标准算子，参见《云天励飞DESDK Graph编程手册》标准算子-JPEG解码章节。

解码功能也可以通过API调用，参见《云天励飞DESDK API手册》JPEG解码章节。

3.2.4.3 JPEG编码

DeepEye1000芯片包含JPEG编码硬件模块，能力可达1080P 75fps或4K 20fps。

Graph模式下封装为JpegEncoder标准算子，参见《云天励飞DESDK Graph编程手册》标准算子-JPEG编码章节。

编码功能也可以通过API调用，参见《云天励飞DESDK API手册》JPEG编码章节。

3.2.5 图像和数据处理

DESDK在芯片端提供了一些常用的图像和数据处理接口，包括图像尺寸格式转换、翻转旋转、抠图，数据的TopN、向量乘，特征值比对等，使用专用硬件和DSP进行加速。参见《云天励飞DESDK API手册》图像和数据处理接口章节。

此外，DeepEye1000芯片提供了独立的KCF硬件对KCF算法进行硬加速，参见《云天励飞DESDK API手册》KCF跟踪章节。

3.2.6 模型推理

模型推理首先需要将模型加载到内存中，再通过推理对象调用硬件进行推理。多个模型推理对象可同时使用同一个加载的模型对象，只占用一份内存空间，但模型推理的中间结果需要分别占用不同的内存空间。

3.2.6.1 模型加载

DESDK通过模型加载接口将模型文件加载到芯片内存中供推理使用。目前加载的模型必须是通过DETVM编译好的离线模型文件，文件可以放在主控或芯片端，如果是在主控端文件路径前需要添加host:前缀。参见《云天励飞DESDK API手册》流程接口的模型加载和卸载章节。

3.2.6.2 模型推理

3.2.6.2.1 基本开发模式

基本开发模式下需要使用AiEngine对象进行推理，步骤为：

1. 创建模型推理类AiEngine的对象，stream_mode需要设置成false，不产生新的线程。
2. 定义前后处理函数，并通过设置接口设置为回调函数。（如果不需要前后处理可省略）
3. 调用对象的LoadModel方法加载模型。（内部会调用上节所述的模型加载接口，如果之前未加载会在此时加载，同一模型只会加载一次）
4. 加载完成后调用Call接口，返回值为推理结果。输入和返回值默认都是NNTask类型，如果定义了前后处理则为前后处理中指定的类型。

3.2.6.2.2 Graph模式

Graph模式下AiEngine是一个标准Node，用户可以直接使用或对其进行继承扩展。如果需要前后处理需要重新定义一个类继承AiEngine，定义前后处理函数，并在新类的构造函数中通过前后处理设置接口设置为回调函数。不需要显式创建对象和加载模型，Node会根据定义好的属性自动进行，参见《云天励飞DESDK Graph编程手册》标准算子的模型推理算子章节。

3.2.7 数据传输

数据在主控CPU和芯片CPU间传输需要进行序列化和反序列化。如果用户需要传输自定义数据类型，需要同时定义序列化和反序列化函数。

如下面的示例代码，用户需要自定义数据类型MediaTaskExt，类型中必须包含1个de::NDArray类的对象array，并定义序列化和反序列化函数。

de::NDArray中预留了一段内存空间供用户使用，默认为256字节，可在de::NDArray构造时设置大小，通过GetBearData方法可以获取到地址。用户可将自定义数据拷贝到该空间，然后通过Shrink方法压缩到最小需要的空间进行序列化。在接收端需要进行反序列化，方法和序列化时必须相对应，以保证可以恢复本来的数据结构。

```
typedef struct{

  de::NDArray array;   ///< data object
  int   streamid; ///< stream id used for multi-stream
  uint64_t pts;    ///< timestamp
  uint32_t frameid;  ///< frame id
  char picname[32];  ///pic name

  //序列化函数
  inline de::NDArray Serialize() {
    int32_t *ptr = static_cast<int32_t*>(array.GetBearData());
    ptr[0] = streamid;
    ptr[1] = pts & 0xFFFFFFFF;
    ptr[2] = (pts >> 32) & 0xFFFFFFFF;
    ptr[3] = frameid;
    memcpy((char *)ptr + 16, picname, 32);
    std::vector<int64_t> shapes;
    array.Shrink(shapes, 12*sizeof(int32_t));
    return array;
  }

  //反序列化函数
  inline void DeSerialize(de::NDArray data) {
    int32_t *ptr = static_cast<int32_t*>(data.GetBearData());
    streamid = ptr[0];
    pts = (((int64_t)ptr[2])<<32) | ((int64_t)ptr[1] & 0xFFFFFFFF);
    frameid = ptr[3];
    memcpy(picname, (char *)ptr + 16, 32);
    array = data;
  }

} MediaTaskExt;
```

在Node间传输自定义数据时，在Node的构造函数中通过de::TaskDeSerializeCustom和de::TaskSerializeCustom模板指定输入输出脚使用自定义的序列化函数。

```
pin.SetTypeInfo(1, "MediaTaskExt", de::TaskDeSerializeCustom<MediaTaskExt>,
de::TaskDeleter<MediaTaskExt>);
pouts_[1]->SetTypeInfo("MediaTaskExt", de::TaskSerializeCustom<MediaTaskExt>);
```

3.2.8 设备管理

DESDK通过设备管理接口（DCMI）实现对芯片设备的控制和管理，包括版本升级和下载、复位、获取设备状态等，参见《云天励飞DESDK API手册》设备管理接口章节。

3.2.9 标准算子

DESDK在Graph模式下提供了一系列标准Node供用户直接使用或扩展，如视频输入、视频解码、JPEG解码、JPEG编码、模型推理等算子。参见《云天励飞DESDK Graph编程手册》标准算子章节。

3.2.10 自定义算子

标准算子的输入和输出是固定的，如果用户想扩展自定义的输入和输出，同时仍使用标准算子提供的基本功能，可以在标准算子基础上进行扩展。参见《云天励飞DESDK Graph编程手册》自定义算子章节。

如果想从头开始定义一个算子，需要以下几步：

1. 继承de::Thread算子基类
2. 构造函数中设置输入和输出Pin的数据类型，以及序列化、反序列化和析构方法。
3. 重写Proc函数，定义新算子的行为。Proc函数将在每个输入到来时被框架调用。
4. 注册新算子

以发送算子为例，该算子从主控端读取图片文件并发送到芯片端解码。

```
//定义算子
class Sender : de::Thread {
  public:
  Sender(){
    //设置输入输出
    pin.SetTypeInfo(0, "de::MediaTask", de::TaskDeSerializeCustom<de::MediaTask>,
de::TaskDeleter<de::MediaTask>);
    pouts_[0]->SetTypeInfo("de::MediaTask",
de::TaskSerializeCustom<de::MediaTask>);
  }
  //定义算子行为
  void Proc(void* rx_task, int32_t task_type, POutType pout_type){
    de::MediaTask* pTask = static_cast<de::MediaTask*>(rx_task);
      FILE *file = fopen("tmp.jpg", "w");
      fwrite((char*)pTask->array.GetTensorData(), pTask->array.GetTensorDataSize(),
1, file);
      fclose(file);
    pouts_[0]->SendTask(pTask);
  }
  void Stop(void){pin.DestroyQueue();}
};

//注册算子
DE_CLASS_REGISTER("Sender", Sender);
```

3.3 软件部署

软件部署从方式上可分为动态加载和静态加载两种方法，从平台上可分为主控和芯片两个部分。

3.3.1 动态加载

动态加载是通过主控端SDK接口加载芯片的动态库和资源运行的方式。特点有：

- 芯片端软件保存在主控端，需要时加载到芯片
- 芯片端修改升级不需要重新烧写版本
- 芯片重启需要从主控重新加载软件，启动时间较长

此方式由于修改升级方便，适合调试阶段以及功能不固定的部署场景。

动态加载流程：

1. 主控端初始化DESDK，与芯片建立通信
2. 主控端调用DESDK的加载接口，将芯片端的库和模型等资源加载到芯片端内存中（注：Graph接口会通过Node属性获取是主控还是芯片端，芯片端会自动加载）
3. 主控端启动业务

3.3.2 静态加载

静态加载是先将芯片端软件烧写或拷贝到芯片端加载运行的方式。特点有：

- 芯片端软件保存在芯片端，上电自动加载
- 芯片端修改升级需要重新制作芯片端版本，烧写或拷贝到芯片端
- 芯片重启可以从本地直接加载软件，启动时间较短

此方式由于对主控资源要求小，适合主控存储小，功能固定的部署场景。

静态加载版本烧写流程：

1. 将芯片端的可执行文件、库和模型等资源打包为新的版本
2. 通过烧写工具或芯片升级接口传输到芯片上，覆盖原有版本
3. 重启芯片

烧写完成后启动流程：

1. 芯片上的启动脚本根据配置执行新版本可执行程序
2. 主控端初始化DESDK，与芯片建立通信
3. 主控端启动业务

3.3.3 主控端

在linux-x64主控下部署同样建议在DEngine_docker下进行，依然可以设置上述平台环境变量。

将desdk的平台库路径加入到环境变量GST_PLUGIN_PATH和LD_LIBRARY_PATH中。

由于USB通信需要使用root权限，请在root权限下执行可执行程序。

3.3.4 芯片端

动态加载时芯片端启动默认的rpc_server，接受主控端控制，动态加载库和资源。

静态加载时先通过烧写工具或升级接口将版本传输到芯片端，覆盖原有的芯片端软件。重启后芯片根据配置文件启动可执行程序运行。

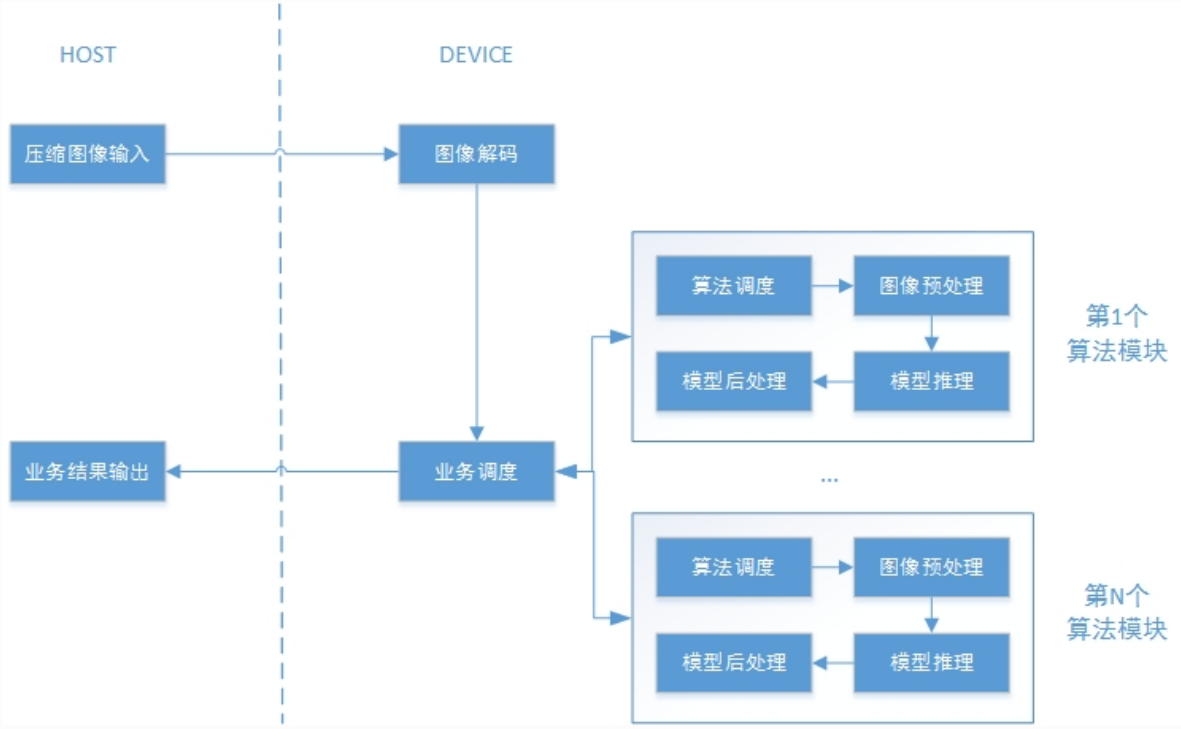
4 高性能编程

提升基于DeepEye芯片开发的应用的性能，需要了解DeepEye芯片和DESDK的一些特性，才能充分利用芯片硬件能力。通常来说，开发者需要遵循以下原则：

4.1 原则：尽量减少主控与芯片间的数据交互

由于DeepEye1000芯片是协处理芯片，与主控CPU间的通信主要通过USB口进行，而USB口通常需要处理视频帧传输、控制信息下发，芯片结果上报等多种数据，加上USB口本身传输速率不高，容易出现拥堵和延迟。因此，应尽量减少主控与协处理芯片间的交互，让芯片主要做推理加速。一般来说，数据预处理、后处理和模型推理都放在芯片侧，主控侧仅做数据输入和结果接收。

推荐的业务开发方式如下图所示，HOST端以压缩图像输入，在DEVICE芯片端解码后交给业务调度模块，由业务调度模块调度不同算法模块工作，并收集最终结果发给HOST端的业务结果输出模块。这样主控与芯片间的数据交互很少，调度过程基本上在芯片端完成，执行效率很高。



4.2 原则：使用DESDK提供的接口管理芯片侧内存

由于DeepEye1000芯片侧的CPU处理能力较弱，芯片侧需要尽量减少内存拷贝，并提高内存申请释放的效率。DESDK提供的内存操作接口使用内存池进行管理，使用连续的内存空间，同时可以获取到物理地址，方便进行零拷贝跨核传输和硬件处理。因此大内存操作一定要使用DESDK提供的内存操作接口，参见《云天励飞DESDK API手册》内存接口章节。

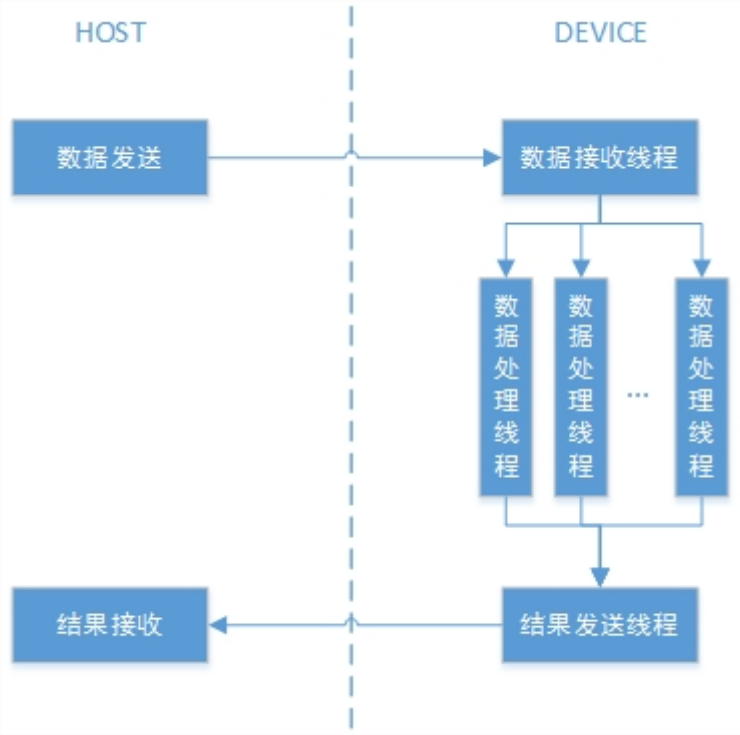
4.3 原则：使用零拷贝序列化

DeepEye芯片与主控CPU间的通信数据需要进行序列化，序列化应尽量使用零拷贝。DESDK提供的基本数据结构de::NDArray本身支持零拷贝序列化，用户的自定义数据需要通过自定义序列化函数指定序列化方式。参见本文**数据传输**章节。

4.4 原则：使用单独线程进行核间传输

一般来说，核间传输的速度与芯片侧处理的速度是不匹配的，因此在处理核间发送和接收数据时，最好使用单独的线程以免阻塞流水。**此原则主要在用户使用非Graph编程时考虑，Graph编程模式下Node间的传输已经为异步，不需要考虑这个问题。**

如下图所示，DEVICE端的数据接收使用单独的线程，接收后将数据发送给数据处理线程，数据处理线程处理完成后将结果发给结果发送线程，由结果发送线程发送给HOST端 的结果接收模块，可防止数据处理和传输速度不匹配导致的同步等待阻塞。



4.5 原则：使用芯片进行视频和图片的编解码

DeepEye芯片中内置了H264/H265视频解码器和JPEG图片的编解码器，使芯片端可以接收压缩的H264/H265视频流或者JPEG图片直接解码，并将需要输出的图片进行编码后输出，这样可以极大的降低USB传输的数据量。同时，硬件的编解码效率很高，也能大大提升系统的性能。

注：芯片的硬编解码器对图像格式和大小有一些要求，不符合要求的图像会使用软件处理甚至会出错，请在使用前确认图像是否满足要求。

4.6 原则：使用芯片进行图像处理

DeepEye芯片中有2个DSP，可以进行数据转换和图像处理等工作，性能很高。DESDK在芯片侧提供了多种图像和数据处理接口供用户调用，使用户可以高效率地串接和执行业务。参考《云天励飞DESDK API手册》图像和数据处理接口章节。

4.7 原则：模型推理尽量使用batch模式和多RES，合理使用多AiEngine

DeepEye芯片模型推理在Graph模式下支持batch模式，多RES调度，以及使用多个AiEngine进行多线程调度。AiEngine是模型推理Node，每个Node拥有1个单独的线程，batch和RES是每个AiEngine的属性。

- batch模式可一次性处理多张图像，1次调度多NNP同时工作，效率最高，限制是多张图像需要同时输入输出。
- 多RES是内部采用异步轮流调度，也可以将多NNP轮流调度起来，但多RES间调度有间隔，调度开销稍大于batch模式。
- 针对多RES的调度间隔，可以使用多AiEngine多线程调度来填充，以保证硬件资源被充分利用。限制是AiEngine过多会导致芯片侧CPU线程过多，性能下降。

三种调度方式都需要额外消耗内存资源，系统实际占用的内存资源为batch数×RES数×线程数，所以在追求性能的同时要考虑系统的可用资源。单个模型在不同线程/batch/RES数下的性能可通过benchmark测试获得。

根据3种调度方式的特点，针对不同业务场景，采用不同的调度组合。

- 对于大量图像批量处理，对延迟要求不高的场景，使用batch模式+多AiEngine调度
- 对于图像流水处理，对延迟有一定要求的场景，使用多RES+多AiEngine调度

5 代码示例说明

本章主要介绍DESDK接口代码示例的编译和运行。

5.1 C++ Graph示例

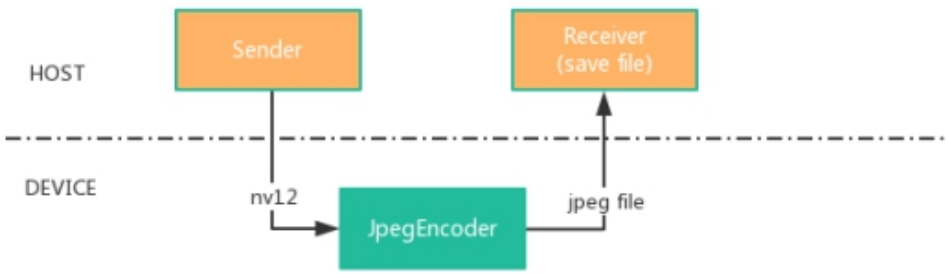
本节介绍如何使用desdk 标准算子及自定义算子串接完成应用开发。

5.1.1 JPEG编码

样例说明

本样例演示JPEG图片编码功能。

流程框架



如何编译

1. ./docker_enter.sh进入docker
2. 进入/DEngine/desdk/cpp目录
3. 执行./build.sh 生成jpeg_enc（ /DEngine/desdk/cpp/example/jpeg_enc/host/bin目录 ）

如何运行

1. docker进入/DEngine/desdk/cpp/example/jpeg_enc/host/bin
2. 执行/DEngine/run.sh ./jpeg_enc
3. 本例使用1080P的nv12数据文件做为输入（ /DEngine/data/1080p/1080p.jpg ），输出jpeg_enc_out.jpg文件。

约束条件

Dp1000图片编码器对于输入原图像的格式要求：

支持格式：NV12/NV21/UYVY/YUYV

最小尺寸：92x32

最大尺寸：8192x8192

水平步进：16

垂直步进：16

运行结果判断

bin目录输出的jpg文件打开画面显示正常。

代码获取

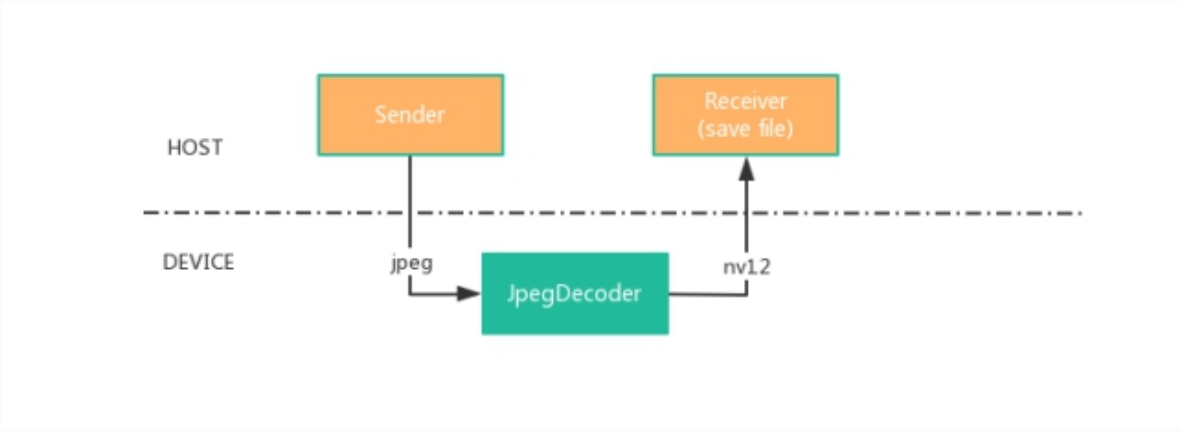
/DEngine/desdk/cpp/example/jpeg_enc/jpeg_enc.cc

5.1.2 JPEG解码

样例说明

本样例演示JPEG图片解码功能。

流程框架



如何编译

1. ./docker_enter.sh进入docker
2. 进入/DEngine/desdk/cpp目录
3. 执行./build.sh 生成jpeg_dec（/DEngine/desdk/cpp/example/jpeg_dec/host/bin目录）

如何运行

1. docker进入/DEngine/desdk/cpp/example/jpeg_dec/host/bin
2. 执行/DEngine/run.sh ./jpeg_dec
3. 本例使用1080P的jpg文件做为输入（/DEngine/data/1080p/1080p.yuv），输出jpeg_dec_out.yuv文件。

约束条件

Dp1000解码器对于输入原图像的格式要求：

支持格式：YCbCr_420SP/YCbCr_422SP/YCbCr_440/YCbCr_400

最小尺寸：48x48

最大尺寸：16386x16386

水平步进：8

垂直步进：8

编码模式：baseline

若原图宽高不满足8像素对齐，软件会自动做补齐，此时输出图像会有花边。

运行结果判断

bin目录输出的nv12文件打开画面显示正常。

代码获取

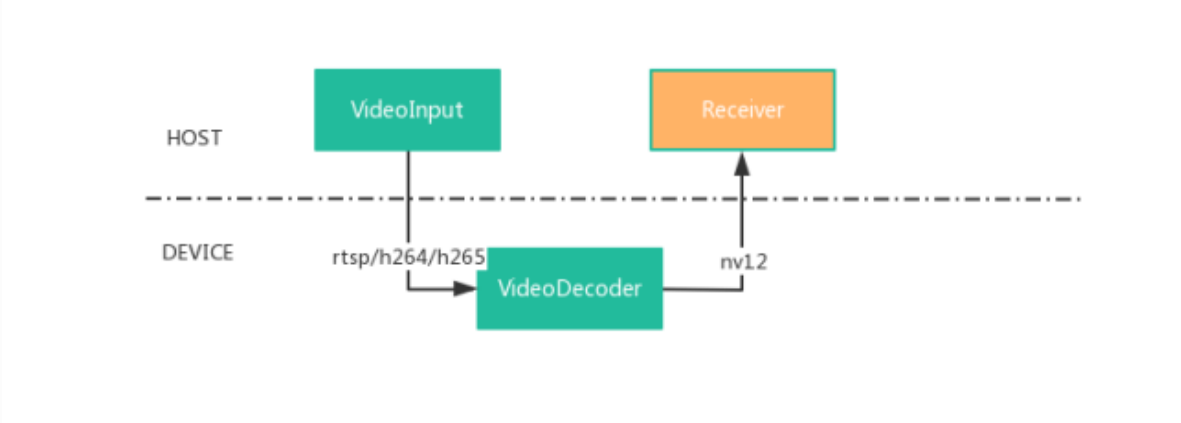
/DEngine/desdk/cpp/example/jpeg_dec/jpeg_dec.cc

5.1.3 视频解码

样例说明

本样例演示视频解码功能。

流程框架



图示绿色模块为desdk标准算子，橙色模块为自定义算子或基于标准算子的扩展算子。

1. 视频输入标准算子将rtsp或h264流连接到芯片侧标准解码算子。
2. Receiver算子接收解码后的nv12数据。

如何编译

1. ./docker_enter.sh进入docker
2. 进入/DEngine/desdk/cpp目录
3. 执行./build.sh 生成video_dec（自动安装到/DEngine/desdk/cpp/example/video_dec/host/bin目录）

如何运行

1. docker进入/DEngine/desdk/cpp/example/video_dec/host/bin
2. 执行/DEngine/run.sh ./video_dec rtsp://admin:introcks1234@192.168.33.124
3. 上例为标准算子VideoInput配置rtsp视频流作为输入。也可以配置h264文件作为输入（如file:///DEngine/data/1080p/1080p.h264）。

约束条件

当前dp1000视频解码只支持h264/h265裸流文件作为输入，或直接配置rtsp流地址（注意本例代码配置为h264流，不能接h265输入）。

运行结果判断

主控持续打印 frame decode finished. 本例只为展示解码结果上报，若同时有打印drop package，是因为nv12数据持续上报引起传输堵塞，为正常现象。

代码获取

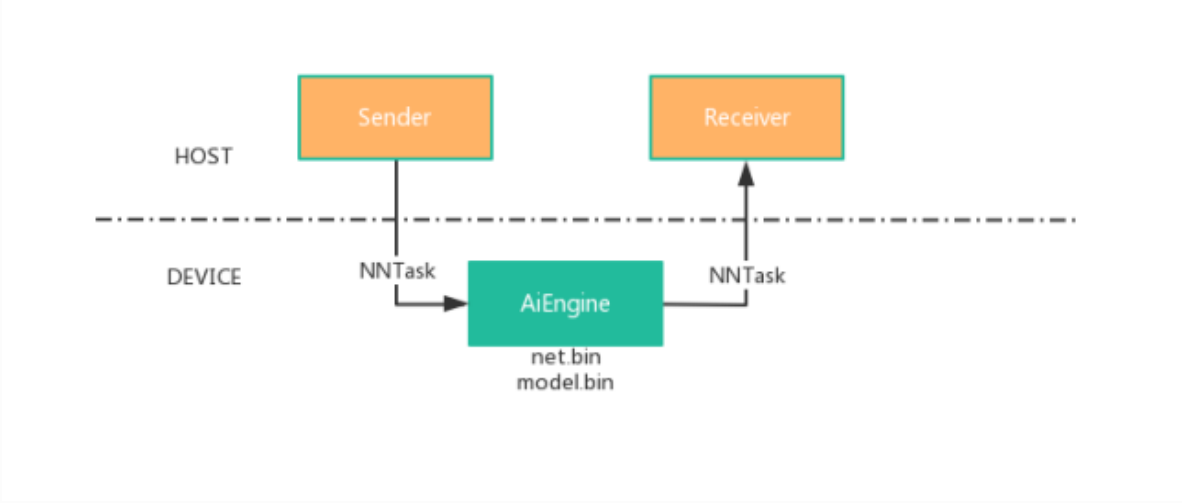
/DEngine/desdk/cpp/example/video_dec/video_dec.cc

5.1.4 模型推理（Graph接口）

样例说明

本样例演示模型推理功能。

流程框架



图示绿色模块为desdk标准算子，橙色模块为自定义算子或基于标准算子的扩展算子。

1. Sender算子构造模型输入发送到标准推理算子。
2. Receiver算子接收推理结果。

如何编译

1. ./docker_enter.sh进入docker
2. 进入/DEngine/desdk/cpp目录
3. 执行./build.sh生成model_pred（自动安装到/DEngine/desdk/cpp/example/model_pred/host/bin目录）

如何运行

1. 查看/DEngine/model/dp1000/caffe_squeezenet_v1.1目录是否存在，是否包含net.bin和model.bin。
2. 进入/DEngine/desdk/cpp/example/model_pred/host/bin
3. 执行/DEngine/run.sh ./model_pred
4. 本例将caffe_squeezenet_v1.1模型加载到芯片侧，并执行模型推理，推理输入为打桩数据，连续压入100个输入。

约束条件

如果模型编译时没有将resize_en开关打开，样例中输入的数据必须符合模型的输入要求。

运行结果判断

主控持续打印100条带有"<====AiEngine result"字样的推理结果。

代码获取

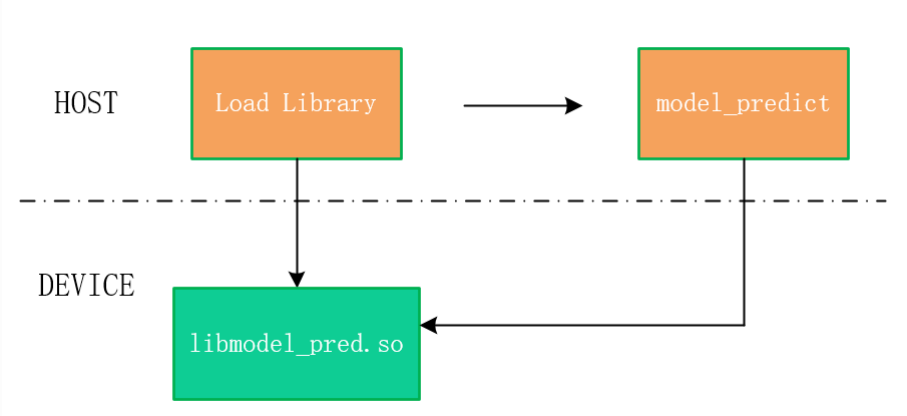
/DEngine/desdk/cpp/example/model_pred/model_pred.cc

5.1.5 模型推理（同步接口）

样例说明

本样例演示非Graph接口模型推理功能。由于跨核同步调用效率较低，非Graph接口主要用于在芯片侧直接串接推理逻辑，主控仅用于驱动芯片侧运行。因此，本例中推理使用的模型和库均提前放到芯片侧。

流程框架



1. 主控将芯片的lib库加载至芯片侧。
2. 芯片侧的libmodel_pred库执行模型加载、图片读取、推理流程。

如何编译

1. ./docker_enter.sh进入docker
2. 进入/DEngine/desdk/cpp目录
3. 执行./build.sh生成model_pred（自动安装到/DEngine/desdk/cpp/example/model_pred_normal/host/bin目录）
4. 执行./build_device.sh生成libmodel_pred.so（自动安装到/DEngine/desdk/cpp/example/model_pred_normal/dev/lib目录）

如何运行

1. 查看/DEngine/model/dp1000/caffe_squeezenet_v1.1目录是否存在，是否包含net.bin和model.bin。
2. 将编译好的net.bin和model.bin文件通过工具拷贝至芯片的/root/data目录。

```
# sh /DEngine/tools/transfer.sh -i 0 -u
/DEngine/model/dp1000/caffe_squeezenet_v1.1/model.bin /root/data/model.bin
# sh /DEngine/tools/transfer.sh -i 0 -u
/DEngine/model/dp1000/caffe_squeezenet_v1.1/net.bin /root/data/net.bin
```

3. 准备一张图片1.jpg，将其拷贝至芯片的/root/app/bin目录

```
# sh /DEngine/tools/transfer.sh -i 0 -u /DEngine/data/pic/1.jpg
/root/app/bin/1.jpg
```

4. 进入/DEngine/desdk/cpp/example/model_pred_normal/host/bin
5. 执行/DEngine/run.sh ./model_pred_normal

约束条件

如果模型编译时没有将resize_en开关打开，样例中输入的数据必须符合模型的输入要求。

运行结果判断

芯片会打印“tensor count: 1”字样的推理结果。

代码获取

/DEngine/desdk/cpp/example/model_pred_normal/host/src/model_pred_host.cc

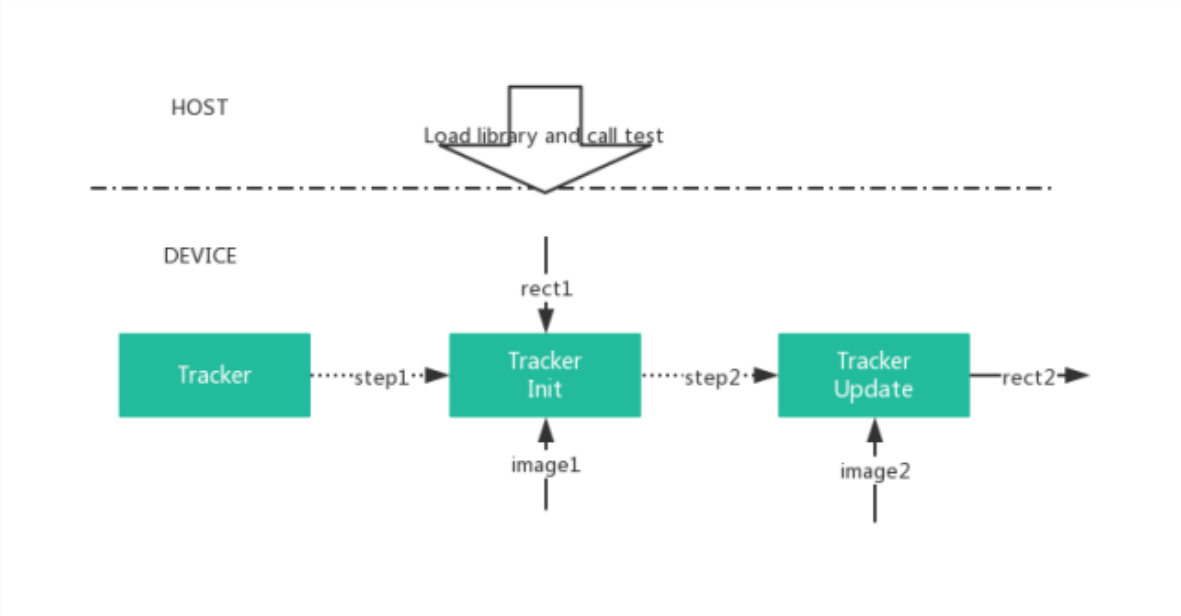
/DEngine/desdk/cpp/example/model_pred_normal/dev/src/model_pred_dev.cc

5.1.6 KCF跟踪

样例说明

本样例演示KCF跟踪功能。

流程框架



图示绿色模块仅为cv转换函数接口的输入输出表示。

本例编译生成将包含测试函数的芯片侧测试库和主控侧测试程序。主控测试程序将测试库下载到芯片并执行测试函数。

如何编译

1. ./docker_enter.sh进入docker
2. 进入/DEngine/desdk/cpp目录
3. 执行./build.sh 生成kcf_proc（自动安装到/DEngine/desdk/cpp/example/kcf_proc/host/bin目录）
4. 执行./build_device.sh生成libkcf_proc.so（自动安装到/DEngine/desdk/cpp/example/kcf_proc/dev/lib）

如何运行

1. 本示例所需yuv文件位于/DEngine/data/1080p，请将此目录下track0.nv12.yuv、track1.nv12.yuv、track2.nv12.yuv复制至DP1000侧/root/data/track_test/目录。

2. 进入/DEngine/desdk/cpp/example/kcf_proc/host/bin

3. 执行/DEngine/run.sh ./kcf_proc

约束条件

kcf跟踪接口详见《云天励飞DESDK API手册》kcf跟踪章节的约定。

运行结果判断

通过串口观察芯片侧log，输出类似打印
Tracker: 0 1 791 51 837 915

Tracker: 0 1 ...

表明三次图片输入有对应的跟踪结果，认为结果正确。

同时可通过芯片侧/root/data/image_test目录下生成的跟踪小图（yuv）查看跟踪目标图像。

代码获取

/DEngine/desdk/cpp/example/kcf_proc/kcf_proc_host.cc

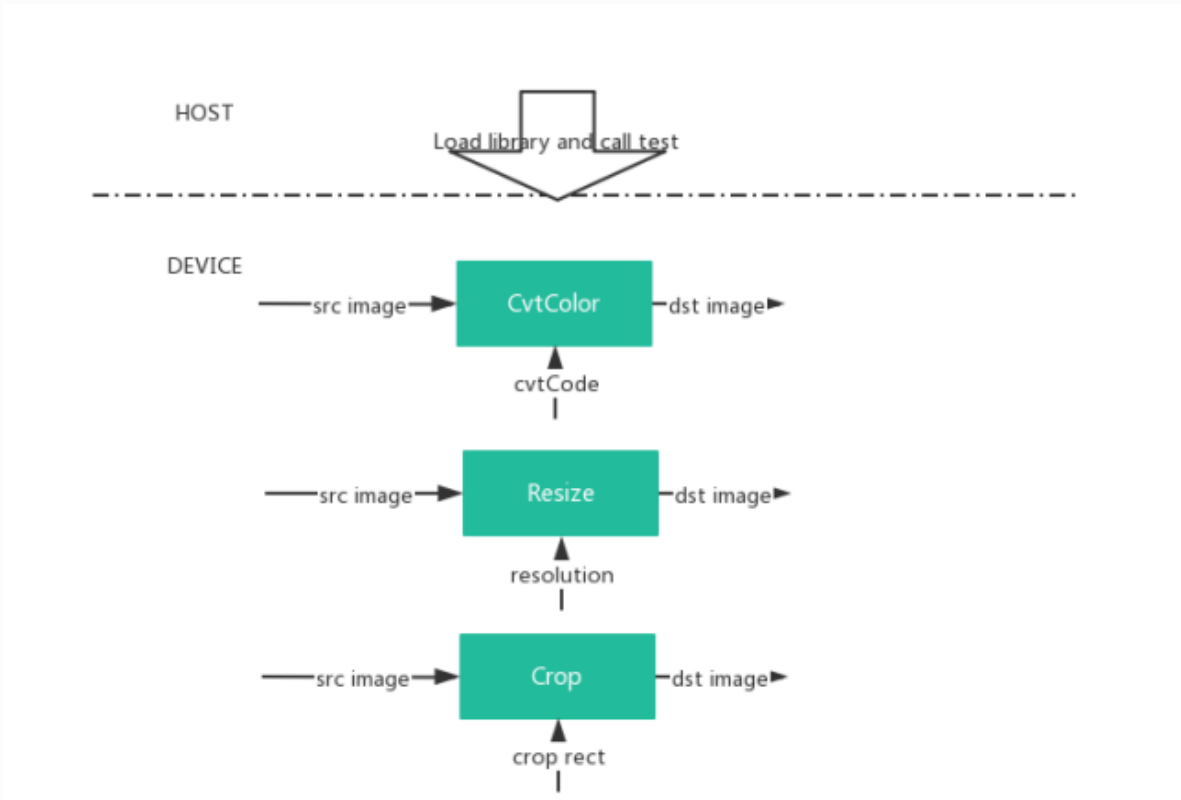
/DEngine/desdk/cpp/example/kcf_proc/kcf_proc_dev.cc

5.1.7 图像处理

样例说明

本样例演示几种常用图像处理功能。

流程框架



以上图示绿色模块仅为cv转换函数接口的输入输出表示。

本例编译生成将包含测试函数的芯片侧测试库和主控侧测试程序。主控测试程序将测试库下载到芯片并执行测试函数。

如何编译

1. ./docker_enter.sh进入docker
2. 进入/DEngine/desdk/cpp目录
3. 执行./build.sh生成image_proc（自动安装到/DEngine/desdk/cpp/example/image_proc/host/bin目录）
4. 执行./build_device.sh生成libimage_proc.so（自动安装到/DEngine/desdk/cpp/example/image_proc/dev/lib）

如何运行

1. 本示例所需yuv文件位于/DEngine/data/1080p，请将此目录下1080p.yuv复制至DP1000侧/root/data/image_test/目录。
2. 进入/DEngine/desdk/cpp/example/image_proc/host/bin
3. 执行/DEngine/run.sh ./image_proc

约束条件

格式转换支持的类型详见《云天励飞DESDK API手册》图像格式转换章节约定。

resize转换支持的类型详见《云天励飞DESDK API手册》图像尺寸转换章节约定。

crop接口使用详见《云天励飞DESDK API手册》图像抠图章节约定。

运行结果判断

本样例：

1. 将test.yuv（1080P，nv12）转换为RGB888格式并保存为/root/data/image_test/test_1080p_888.rgb
2. 将test_1080p_888.rgb resize到720p并保存为/root/data/image_test/test_720p_888.rgb
3. 将test_1080p_888.rgb resize到CIF（通用影像传输格式）并保存为/root/data/image_test/test_cif_888.rgb
4. 将test.yuv（1080P，nv12）进行crop并保存为/root/data/image_test/test_720p_crop.nv12.yuv
5. 将test_1080p_888.rgb resize到720p同时crop并保存为/root/data/image_test/test_640x360_crop.rgb

代码获取

/DEngine/desdk/cpp/example/image_proc/image_proc_host.cc

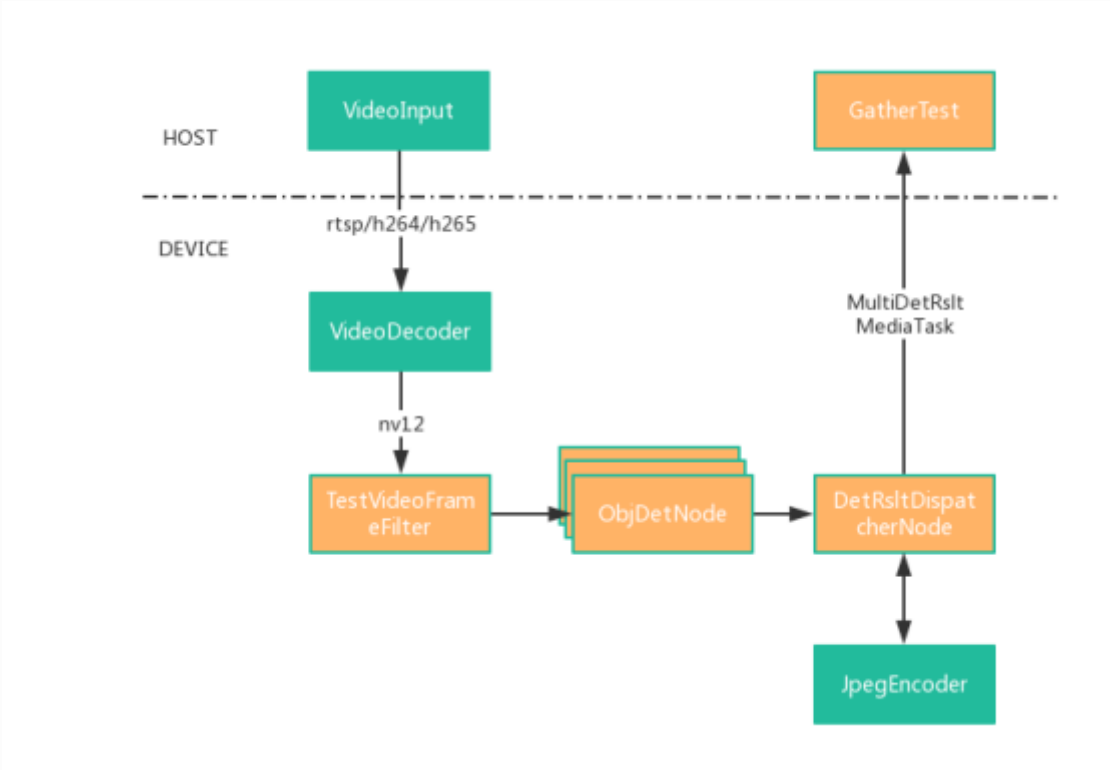
/DEngine/desdk/cpp/example/image_proc/image_proc_dev.cc

5.1.8 视频检测

样例说明

本样例以公版模型yolov3为例，演示使用检测模型做视频检测的简单抓拍流程。

流程框架



图示绿色模块为desdk标准算子，橙色模块为自定义算子或基于标准算子的扩展算子。

1. TestVideoFrameFilter算子接收解码后的视频帧，按简单的策略分发给一个或多个检测算子（ObjDetNode）
2. ObjDetNode算子执行模型推理和前后处理，将检测结果发送到DetRsltDispather算子
3. DetRsltDispather将视频帧编码为jpg联通检测结果一通上报给主控GatherTest算子。
4. GatherTest算子对收到的jpg图片和检测结果进行匹配并打框保存图片。

如何编译

1. ./docker_enter.sh进入docker
2. 进入/DEngine/desdk/cpp目录
3. 执行./build.sh 生成video_det（自动安装到/DEngine/desdk/cpp/example/video_det/host/bin目录）
4. 执行./build_device.sh 生成libdevnodes.so（自动安装到/DEngine/desdk/cpp/example/nodebase/dev/lib）

如何运行

1. 进入/DEngine/desdk/cpp/example/video_det/host/bin
2. 本例使用模型路径ftp://113.100.143.90:821/release/models/dp1000_v1.1.0/1nnp/caffe_yolov3_416，请下载caffe_yolov3_416拷贝到/DEngine/model/dp1000/目录
3. 执行/DEngine/run.sh ./video_det rtsp://admin:introcks1234@192.168.33.124 1920 1080

4. 视频参数可以为rtsp也可以为h264本地文件，如 file:///DEngine/data/1080p/1080p.h264

约束条件

芯片解码限制视频输入必须为h264裸流。

运行结果判断

在/DEngine/desdk/cpp/example/video_det/host/bin目录下创建org和img目录，则运行后会在目录下保存有抓拍图片（带打框）和检测结果（json）文件。

代码获取

/DEngine/desdk/cpp/example/video_det/host/video_det.cc

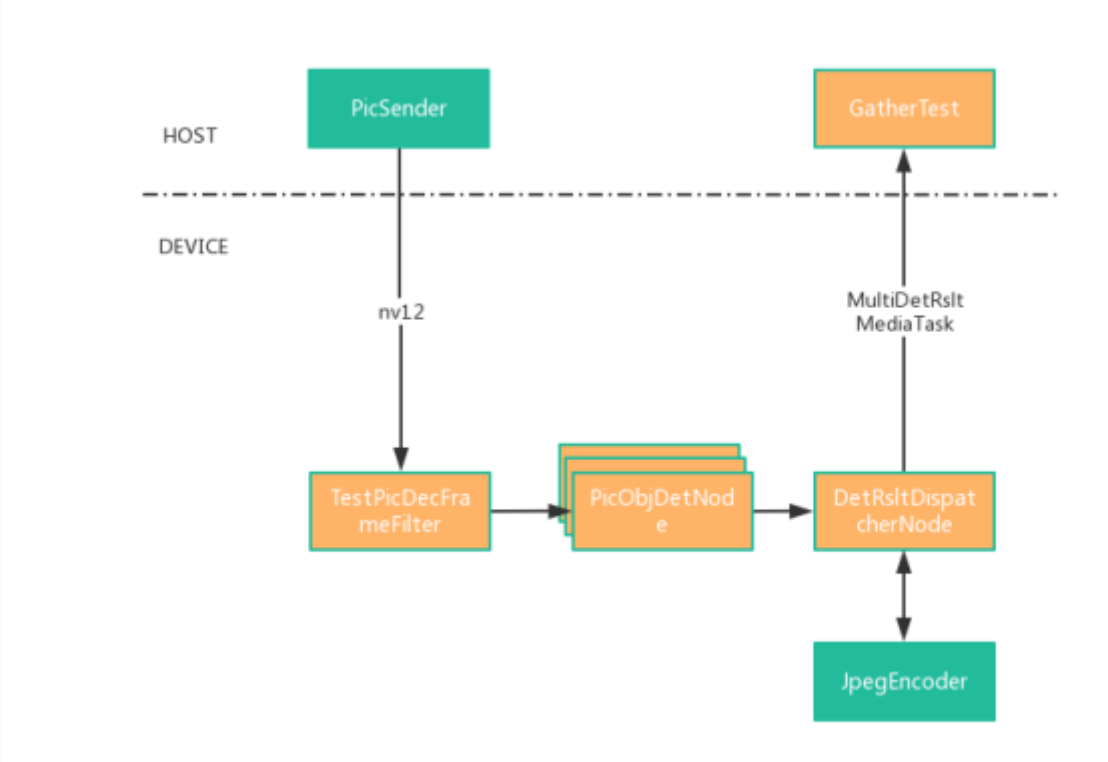
/DEngine/desdk/cpp/example/nodebase/dev/yolov3/*.cc

5.1.9 图片检测

样例说明

本样例以公版模型yolov3为例，演示图片检测流程。

流程框架



图示绿色模块为desdk标准算子，橙色模块为自定义算子或基于标准算子的扩展算子。

1. TestPicDecFrameFilter算子接收解码后的视频帧，按简单的策略分发给一个或多个检测算子（PicObjDetNode）。
2. PicObjDetNode算子执行模型推理和前后处理，将检测结果发送到DetRsltDispather算子。
3. DetRsltDispather将视频帧编码为jpg联通检测结果一通上报给主控GatherTest算子。
4. GatherTest算子对收到的jpg图片和检测结果进行匹配并打框保存图片。

如何编译

1. ./docker_enter.sh进入docker
2. 进入/DEngine/desdk/cpp目录
3. 执行./build.sh 生成pic_det（自动安装到/DEngine/desdk/cpp/example/pic_det/host/bin目录）
4. 执行./build_device.sh 生成libdevnodes.so（自动安装到/DEngine/desdk/cpp/example/nodebase/dev/lib）

如何运行

1. 进入/DEngine/desdk/cpp/example/pic_det/host/bin。
2. 本例使用模型路径ftp://113.100.143.90:821/release/models/dp1000_v1.1.0/1nnp/caffe_yolov3_416，请下载caffe_yolov3_416拷贝到/DEngine/model/dp1000/目录
3. 执行/DEngine/run.sh ./pic_det jpg://图片目录实际路径。
4. 程序将遍历图片目录下的文件作为输入。

约束条件

仅支持部分格式硬解码，具体请参考《云天励飞DESDK Graph编程手册》中JpegDecoder的描述。

运行结果判断

在/DEngine/desdk/cpp/example/pic_det/host/bin目录下创建org和img目录则运行后会在目录下保存打框图片和检测结果（json）文件。

代码获取

/DEngine/desdk/cpp/example/video_det/host/pic_det.cc

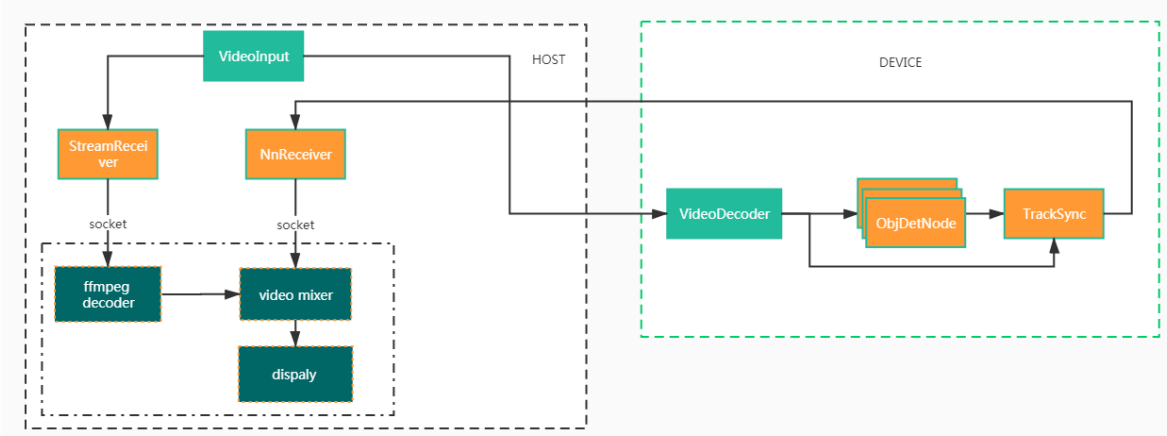
/DEngine/desdk/cpp/example/nodebase/dev/yolov3/*.cc

5.1.10 视频检测跟踪（可视化）

样例说明

本样例以公版模型yolov3为例，演示可视化视频检测跟踪示例流程。由于显示组件的限制，该示例仅在ubuntu-x64环境可视化环境下编译运行。

流程框架



图示绿色模块为desdk标准算子，橙色模块为自定义算子或基于标准算子的扩展算子，墨绿色模块为一个单独显示进程的主要过程。

1. VideoInput算子将h264/h265视频接入StreamReceiver算子，同时送往芯片侧视频解码算子（VideoDecoder）解码。

2. 解码后的视频帧送到一个或多个检测算子（ObjDetNode）做推理，同时送往跟踪同步算子（TrackSync）。

3. 主控NNReceiver算子接收检测跟踪结果通过socket发给display显示进程。

4. 主控StreamReceiver算子将视频流通过socket发给display显示进程。

5. display进程对视频流进行解码、转换、打框并显示。

如何编译

1. ./docker_enter.sh进入docker
2. 进入/DEngine/desdk/cpp目录
3. 执行./build.sh生成video_det_track（自动安装到/DEngine/desdk/cpp/example/video_det_track/host/bin目录）
4. 执行./build_device.sh生成libdevnodes.so（自动安装到/DEngine/desdk/cpp/example/nodebase/dev/lib）

如何运行

1. 进入/DEngine/desdk/cpp/example/video_det_track/host/bin。
2. bin目录下的run.sh脚本可配置参数：

   -c 芯片个数, 默认1(不配置)

   -s 流数量, 默认1

   -t 流类型，可选项rtsp、file、usbcamera, 默认rtsp

   -f 帧率, 默认25

   -w 流像素宽, 默认1920

   -h 流像素高, 默认1080

   -m 模型路径，默认host:/DEngine/model/dp1000/caffe_yolov3_416, 本例使用模型路径ftp://113.100.143.90:821/release/models/dp1000_v1.1.0/1nnp/caffe_yolov3_416，请下载caffe_yolov3_416拷贝到/DEngine/model/dp1000/目录

   -l 待加载芯片库路径，默认host:/DEngine/desdk/cpp/example/nodebase/dev/lib/libdevnodes.so;如有多个库用逗号分开

   -u 流地址路径, 默认rtsp://admin:introcks1234@192.168.33.124, 多个地址需要写在双引号中并用空格隔开

3. 根据上述参数选择配置并执行run.sh

   执行样例：

   sh run.sh -t rtsp -u rtsp://username:passwd@192.168.33.124

   sh run.sh -t file -c 1 -s 2 -w 640 -h 480 -f 25 -u "file://1.h264 file://2.h264"

   sh run.sh -t usbcamera -c 1 -s 1 -w 640 -h 480 -f 30 -u v4l2:///dev/video0

约束条件

本例使用的检测模型为yolov3-416，libdevnodes.so已集成此公版模型的算法后处理，请确保模型路径下放有yolov3-416对应的model.bin和net.bin，可到集成开发指南指定的ftp下载。

运行结果判断

窗口显示正常，打框正常。

代码获取

/DEngine/desdk/cpp/example/video_det/host/video_det_track.cc

/DEngine/desdk/cpp/example/nodebase/dev/yolov3/*.cc

/DEngine/desdk/cpp/example/nodebase/dev/tracksync/*.cc

5.1.11 芯片RPC服务程序

样例说明

本样例展示如何重写芯片侧RPC服务程序。

desdk_rpc_server是芯片端开机自启的服务程序，负责与主控的sdk进行RPC通信、执行主控下发的命令等基本功能。如果用户需要在主控sdk连接前进行一些操作，就必须重写RPC服务程序。

流程框架

1. 读取配置文件。
2. 初始化系统、初始化DESDK。

如何编译

1. ./docker_enter.sh进入docker
2. 进入/DEngine/desdk/cpp目录
3. 执行./build_device.sh生成desdk_rpc_server（自动安装到/DEngine/desdk/cpp/example/rpc_server/dev/bin目录）

如何运行

1. 在芯片侧备份desdk_rpc_server文件：

```
# cd /root/app/bin
# mv desdk_rpc_server desdk_rpc_server_bak
```

2. 在主控侧将编译好的desdk_rpc_server拷贝至芯片：

```
# sh /DEngine/tools/transfer.sh -i 0 -u
/DEngine/desdk/cpp/example/rpc_server/dev/bin/desdk_rpc_server
/root/app/bin/desdk_rpc_server
```

3. 在芯片侧为desdk_rpc_server添加可执行权限：

```
# chmod +x desdk_rpc_server
```

4. 重启芯片后，desdk_rpc_server自动运行。

约束条件

无

运行结果判断

芯片串口会每个一段时间打印“SDK Alive!”字样。

代码获取

/DEngine/desdk/cpp/example/rpc_server/dev/src/rpc_server.cc

5.1.12 下载

样例说明

本样例演示如何下载版本到开发板。

流程框架

1. 主控准备待下载的文件到目录/DEngine/desdk/archive/evb/file。

其中evb目录是开发板目录，实际使用时，按照项目的实际情况指定版本文件的目录。

file目录中的文件如下：

config.ini 下载配置文件
images.tar.bz2 版本文件

如何编译

1. ./docker_enter.sh进入docker
2. 进入/DEngine/desdk/cpp目录
3. 执行./build.sh生成dcmi_download（自动安装到/DEngine/desdk/cpp/example/dcmi_download/host/bin目录）

如何运行

1. 将芯片调整至烧录模式，具体操作参考各硬件平台快速上手手册
2. 进入/DEngine/desdk/cpp/example/dcmi_download/host/bin目录，执行/DEngine/run.sh ./dcmi_download
3. 下载完成后，将芯片调整至工作模式。

约束条件

下载只支持烧录模式。

运行结果判断

出现“download dp1000 -> OK”字样，下载成功后，切换拨码开关，重启板子，版本正常启动。

代码获取

/DEngine/desdk/cpp/example/dcmi_download/host/src/dcmi_download.cc

5.1.13 升级

样例说明

本样例演示如何升级版本到开发板。

流程框架

1. 主控准备待升级的文件到目录/DEngine/desdk/archive/evb/file。

其中evb目录是开发板目录，实际使用时，按照项目的实际情况指定版本文件的目录。

file目录中的文件如下：

images.tar.bz2 待升级的版本文件

如何编译

1. ./docker_enter.sh进入docker
2. 进入/DEngine/desdk/cpp目录
3. 执行./build.sh生成dcmi_upgrade（自动安装到/DEngine/desdk/cpp/example/dcmi_upgrade/host/bin目录）

如何运行

1. 将芯片调整至工作模式，具体操作参考各硬件平台快速上手手册
2. 进入/DEngine/desdk/cpp/example/dcmi_upgrade/host/bin目录，执行/DEngine/run.sh ./dcmi_upgrade

约束条件

升级只支持工作模式。

运行结果判断

出现“>>>>>> success to send cmd reboot”字样，升级成功后，版本正常启动。

代码获取

/DEngine/desdk/cpp/example/dcmi_upgrade/host/src/dcmi_upgrade.cc

5.2 Python示例

本章主要介绍使用python接口进行神经网络模型推理，包括得到推理结果和进行精度测试。

注：示例中有一些模型在套件包中未直接提供，可通过/DEngine/model/get_models.sh脚本下载。同时部分模型参数文件并非真实数据集训练，只展示使用方法，无需关注执行精度结果。

5.2.1 模型推理

5.2.1.1 resnet50模型推理

示例说明

在芯片上运行resnet50模型，对目标测试图片目录下的图片进行分类。

1. 配置模型和测试图片目录

```
net_file = "/DEngine/model/dp1000/public/4nnp/caffe_resnet50/net.bin"
model_file = "/DEngine/model/dp1000/public/4nnp/caffe_resnet50/model.bin"
img_dir = "/DEngine/data/datasets/ILSVRC2012/ILSVRC2012_img_val"
```

2. 创建InferEngine推理对象，max_batch参数为batchsize的最大值。

```
engine = InferEngine(net_file, model_file, max_batch=8)
```

3. 准备模型输入。输入数据必须符合模型输入的类型和shape。如果模型编译时未配置resize且测试图片尺寸不等于模型输入，需要配置resize。如果不使用芯片解码，可以使用load_image接口解码图片并resize。data为每个样本的输入数组，每个(format, shape, img)为一个输入。如果是batch模式，配置多个data数组。

```
# 如果模型不配置resize，需要resize到模型输入大小
resize = None
format = de.PixelFormat.DE_PIX_FMT_RGB888_PLANE
img = load_image(img_path, resize=resize)
shape = img.shape
data = [(format, shape, img)]
```

4. 执行模型，获取执行结果。data_out即模型的输出结果，是一个numpy数组，与浮点模型的输出内容一致，格式需要关注data_out[i].dtype，根据类型匹配相应后处理。如果是batch模式，predict内可配置多个data参数。

```
data_out = engine.predict(data)
```

5. 模型后处理，模型输出结果为每个id的置信度，将结果从大到小排序得到最大置信度对应的id=a[0]，从标注表中得到label并显示。

```
# 模型后处理
a = np.argsort(-data_out[0].flatten())
# 获取label
datasets_path = "/DEngine/data/"
synset_path = os.path.join(datasets_path, "synset_1000.txt")
synsets_label = get_labels(synset_path)
print("predict id = {}, label = {}, prob = {}".format(a[0],
synsets_label[a[0]], data_out[0][a[0]]))
```

6. 显示本次推理过程端到端执行分布时间profile，非必要。

```
engine.profile()
```

如何运行

1. 从ftp的/release/models/dp1000/4nnp/caffe_resnet50下载模型放到/DEngine/model/dp1000/caffe_resnet50目录下。可通过get_models.sh脚本一次性下载全部模型。
2. 进入/DEngine/desdk/python/example目录，执行sh /DEngine/run.sh test_resnet50.py

运行结果

打印显示图片集中每张图片的推理结果，top1的分类ID，label和置信度。最后profile显示端到端执行每步的平均耗时。

```
==========================================
test id 1, image ILSVRC2012_val_00000001.JPEG
predict result: data_out len=1
data_out[0], shape=(1000,), dtype=float16
predict id = 65, label = n01751748 sea snake, prob = 0.939453125

==========================================
test id 2, image ILSVRC2012_val_00000002.JPEG
predict result: data_out len=1
data_out[0], shape=(1000,), dtype=float16
predict id = 795, label = n04228054 ski, prob = 0.966796875

==========================================
test id 3, image ILSVRC2012_val_00000003.JPEG
predict result: data_out len=1
data_out[0], shape=(1000,), dtype=float16
predict id = 230, label = n02105855 Shetland sheepdog, Shetland sheep dog,
Shetland, prob = 0.91748046875

==========================================
test id 4, image ILSVRC2012_val_00000004.JPEG
predict result: data_out len=1
data_out[0], shape=(1000,), dtype=float16
predict id = 809, label = n04263257 soup bowl, prob = 0.892578125

==========================================
test id 5, image ILSVRC2012_val_00000005.JPEG
predict result: data_out len=1
data_out[0], shape=(1000,), dtype=float16
predict id = 520, label = n03131574 crib, cot, prob = 0.375732421875

==========================================
test id 6, image ILSVRC2012_val_00000006.JPEG
predict result: data_out len=1
data_out[0], shape=(1000,), dtype=float16
predict id = 67, label = n01755581 diamondback, diamondback rattlesnake, Crotalus
adamanteus, prob = 0.4267578125

==========================================
test id 7, image ILSVRC2012_val_00000007.JPEG
predict result: data_out len=1
data_out[0], shape=(1000,), dtype=float16
predict id = 334, label = n02346627 porcupine, hedgehog, prob = 0.9990234375

==========================================
test id 8, image ILSVRC2012_val_00000008.JPEG
predict result: data_out len=1
data_out[0], shape=(1000,), dtype=float16
predict id = 415, label = n02776631 bakery, bakeshop, bakehouse, prob =
0.5634765625

==========================================
test id 9, image ILSVRC2012_val_00000009.JPEG
```

```
predict result: data_out len=1
data_out[0], shape=(1000,), dtype=float16
predict id = 674, label = n03794056 mousetrap, prob = 0.94482421875

========================================
test id 10, image ILSVRC2012_val_00000010.JPEG
predict result: data_out len=1
data_out[0], shape=(1000,), dtype=float16
predict id = 153, label = n02085936 Maltese dog, Maltese terrier, Maltese, prob =
0.363525390625

profile:
[2021-08-19 03:24:36.295621] load library cost 0.016 ms
[2021-08-19 03:24:37.971466] engine create cost 1675.845 ms
[2021-08-19 03:24:39.150597] data to ndarray cost 3.569 ms
[2021-08-19 03:24:39.185615] predict batch=1 cost 35.018 ms
[2021-08-19 03:24:39.185791] get output cost 0.176 ms
```

代码获取

/DEngine/desdk/python/example/test_resnet50.py

5.2.2 resnet50分类模型top1/top5精度测试

示例说明

以ILSVRC2012作为测试集，在芯片上运行resnet50模型，计算分类模型推理top1/top5精度。

1. 配置被测模型、测试集路径和测试样本数。测试样本数越大结果越准确，如果配置的测试样本数小于测试集，会提示[warning] test num x is lower than dataset num y, may affect the results!

   ```
   datasets_path = "/DEngine/data/datasets/ILSVRC2012"
   model_path = "/DEngine/model/dp1000/public/4nnp/caffe_resnet50/model.bin"
   net_path = "/DEngine/model/dp1000/public/4nnp/caffe_resnet50/net.bin"
   test_img_nums=5000
   ```

2. 创建InferEngine推理对象，max_batch参数为batchsize的最大值

   ```
   engine = InferEngine(net_file, model_file, max_batch=8)
   ```

3. 准备模型输入。输入数据必须符合模型输入的类型和shape。如果不使用芯片解码，可以使用load_image接口解码图片并resize。data为每个样本的输入数组，每个(format, shape, img)为一个输入。如果是batch模式，配置多个data数组。

   ```
   format = de.PixelFormat.DE_PIX_FMT_RGB888_PLANE
   img = load_image(img_path, resize=resize)
   shape = img.shape
   data = [(format, shape, img)]
   ```

4. 执行模型，获取执行结果。data_out即模型的输出结果，是一个numpy数组，与浮点模型的输出内容一致，格式需要关注data_out[i].dtype，根据类型匹配相应后处理。如果是batch模式，predict内可配置多个data参数。

   ```
   data_out = engine.predict(data)
   ```

5. 模型后处理，根据数据集标注文件判断每个测试样本top1/top5正确性，最终计算出准确率。

```
# 后处理计算top1/top5正确性
flag1, flag5, model_info = graph_eval_result_proc(data_out[0], imgname,
val_label_dict, synsets_label, topn)
```

如何运行

1. 从ftp的/release/models/dp1000/4nnp/caffe_resnet50下载模型，放到/DEngine/model/dp1000/caffe_resnet50目录下。可通过get_models.sh脚本一次性下载全部模型。
2. 下载完整的ILSVRC2012数据集放到/DEngine/data/datasets/ILSVRC2012目录下。
3. 进入/DEngine/desdk/python/example/acctest目录，执行sh /DEngine/run.sh acctest_caffe_resnet50.py

运行结果

运行完成后，log显示测试的样本数和top1/top5准确度如下：

```
finish total = 5000

top1 = 0.7494

top5 = 0.9192
```

代码获取

/DEngine/desdk/python/example/acctest/acctest_caffe_resnet50.py

5.2.3 yolov3检测模型mAP精度测试

示例说明

以coco_val2014作为测试集，在芯片上运行yolov3模型，计算检测模型推理mAP精度。

1. 配置被测模型、测试集路径和测试样本数。测试样本数越大结果越准确，如果配置的测试样本数小于测试集，会提示[warning] test num x is lower than dataset num y, may affect the results!

```
datasets_path = "/DEngine/data/datasets/coco_val2014"
model_path = "/DEngine/model/dp1000/public/4nnp/caffe_yolov3_416/model.bin"
net_path = "/DEngine/model/dp1000/public/4nnp/caffe_yolov3_416/net.bin"
test_img_nums=5000

coco_class = coco(data_path, ann_file, '2014')
num_images = len(coco_class.image_index)
class_num = coco_class.num_classes
```

2. 创建InferEngine推理对象，max_batch参数为batchsize的最大值

```
engine = InferEngine(net_file, model_file, max_batch=8)
```

3. 准备模型输入。输入数据必须符合模型输入的类型和shape。如果不使用芯片解码，可以使用load_image接口解码图片并resize。data为每个样本的输入数组，每个(format, shape, img)为一个输入。如果是batch模式，配置多个data数组。

```
size = 416
resize = [size, size]

format = de.PixelFormat.DE_PIX_FMT_RGB888_PLANE
img = load_image(img_path, resize=resize)
shape = img.shape
data = [(format, shape, img)]
```

4. 执行模型，获取执行结果。data_out即模型的输出结果，是一个numpy数组，与浮点模型的输出内容一致，格式需要关注data_out[i].dtype，根据类型匹配相应后处理。如果是batch模式，predict内可配置多个data参数。

```
data_out = engine.predict(data)
```

5. 模型后处理，根据数据集标注文件判断每个测试样本每种检测类型的AP，最终计算出mAP。

```
# 计算mAP
result += coco_class.evaluate_detections(all_boxes, output_dir)
```

如何运行

1. 从ftp的/release/models/dp1000/4nnp/caffe_yolov3_416下载模型到/DEngine/model/dp1000/caffe_yolov3_416目录下。可通过get_models.sh脚本一次性下载全部模型。
2. 下载完整的coco_val2014数据集放到/DEngine/data/datasets/coco_val2014目录下
3. 进入/DEngine/desdk/python/example/acctest目录，执行sh /DEngine/run.sh acctest_caffe_yolov3.py

运行结果

运行完成后，log显示测试的mAP和AP准确度如下（第1个为mAP，后面为各分类AP）

```
~~Mean and per-category AP @ IoU=[0.50,0.50] ~~

66.4 80.2 55.7 64.1 74.6 90.5 ……
```

代码获取

/DEngine/desdk/python/example/acctest/acctest_caffe_yolov3.py

6 FAQ

6.1 运行模型期间出现内存耗尽错误

问题现象

出现如下打印：dp1000 dataspace out of memory! size = 2296996028

问题原因

```
    1. 模型存储和运行需要都需要使用连续内存，目前dp1000支持的最大连续内存为512M，超出则无法分配内存导致运行失败。
```

2. 程序运行过程中内存没有及时释放也会导致内存耗尽。

解决方法

1. 检查模型的大小以及运行内存需求。如果设置了batch，请减小batch数尝试。如果没有设置batch，则说明模型参数或中间变量过大，需要缩减，否则无法支持。

2. 检查程序是否有内存泄漏。

6.2 运行时出现JPEG解码错误

问题现象

出现如下打印：Can not get jpeg info:-2, jpeg decode failed:-2

问题原因

原因是程序中配置使用了芯片解码器，目标图片格式不符合芯片解码器对于JPEG格式的要求。要求详见《云天励飞DEngine集成开发指南》业务开发准备—业务开发约束章节。

解决方法

更换符合要求的JPEG图片，或者用户自行解码后送给芯片。