Atlas 300 AI 加速卡

应用软件开发指南 (型号 3000)

文档版本	01
发布日期	2020-05-09

华为技术有限公司

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。

目 录

1 开发前必读 1
2 软件开发说明（重要） 3
3 开发前准备 4
3.1 获取样例程序 4
3.2 准备环境 4
4 安装驱动和开发工具 5
4.1 安装升级驱动 5
4.2 升级 MCU 固件 5
4.3 安装 DDK 开发工具 6
5 开发环境介绍 7
5.1 DDK 目录分布 7
5.2 头文件与库介绍 8
5.2.1 头文件 8
5.2.2 动态库 8
5.3 编译工具介绍 8
5.3.1 Host 侧编译 9
5.3.2 Device 侧编译 9
5.4 开发工具介绍 9
6 通用推理业务流程介绍 11
6.1 硬件业务流程 11
6.2 软件业务流程 12
6.3 软件模块与硬件模块对应关系示例 13
7 代码运行样例（HelloDavinci 程序） 14
7.1 获取 HelloDavinci 代码 14
7.2 HelloDavinci 文件说明 14
7.3 HelloDavinci 流程框架介绍 15
7.4 HelloDavinci 编译运行 16
8 软件代码开发 18
8.1 配置 Matrix 框架 18
8.1.1 配置、创建与销毁 Graph 19

8.1.2 配置 Engine..... 21
8.1.3 配置数据传输..... 22
8.2 调用 DVPP 接口..... 23
8.2.1 使用 DVPP 接口..... 23
8.2.2 申请 DVPP 内存..... 25
8.3 离线模型推理..... 25
8.3.1 配置 AIPP..... 25
8.3.2 转换离线模型..... 26
8.3.3 执行模型推理..... 27
8.4 优化软件性能..... 29
8.5 软件日志调测..... 29
8.5.1 日志注册..... 29
8.5.2 日志打印..... 30
8.5.3 日志查看..... 30
8.5.4 日志重启..... 33
8.6 业务软件编译..... 33
8.7 Demo 样例解析..... 34
8.7.1 离线视频推理（InferOfflineVideo）..... 34
9 附录..... 37
9.1 Graph 关键字..... 37
9.2 修订记录..... 41

1 开发前必读

本章节主要介绍使用Atlas 300 AI加速卡 （ 型号 3000 ）进行业务开发时需要了解的基础知识、要求和注意事项。

建议开发人员仔细阅读本章节内容，确保了解各项要求和注意事项之后再启动开发。

使用场景

适用于使用Atlas 300 AI加速卡 （ 型号 3000 ）+华为服务器进行推理任务的场景。

关键概念

表 1-1 关键概念

概念	解释
Ascend 310	Ascend 310是一款华为专门为图像识别、视频处理、推理计算及机器学习等领域设计的高性能、低功耗AI芯片。芯片内置2个AI core，可支持128位宽的LPDDR4x，最高可提供16TOPS（ Float16/INT8 ） 的计算能力。
Atlas 300 AI加速卡（ 型号 3000 ）	Atlas 300 AI加速卡 （ 型号 3000 ）采用4片海思Ascend 310处理器，是标准的PCIe HHHL卡，配合主设备，实现快速高效的模型推理、图像识别及处理等工作。
Mind Studio	Mind Studio是一套基于华为NPU（ Neural-network Processing Unit ）开发的AI全栈开发平台，包括基于芯片的算子开发、调试、调优以及自定义算子开发，同时还包括网络层的网络移植、优化和分析，另外在业务引擎层提供了一套可视化的AI引擎拖拽式编程服务。如果想进一步了解Mind Studio，请参考文档《 Ascend 310 Mind Studio 快速入门 》和《 Ascend 310 Mind Studio 基础操作 》。
DDK	数字开发套件（ Digital Development Kit ），DDK是Mind Studio解决方案提供的开发者套件包，Mind Studio通过安装DDK后获得Mind开发必需的API、库、工具链等开发组件。

概念	解释
Graph	Graph是HiAI框架中的概念，而非深度学习框架中计算图的概念。Graph是指HiAI框架中用于描述整个业务处理流程的图，由多个Engine组成，是一个程序处理流程。
HiAI Engine	HiAI Engine是一个通用业务流程执行引擎，主要包含Agent（运行在Host侧）和Manger（运行在Device侧）两个部分。每个Engine完成一个由用户代码实现的功能，即Engine的处理程序是由用户实现的。
Host侧	对于服务器+Atlas 300 AI加速卡 （型号 3000）来说，Host侧为服务器侧CPU的操作系统。
Device侧	对于服务器+Atlas 300 AI加速卡 （型号 3000）来说，Device侧为Atlas 300 AI加速卡 （型号 3000）侧的操作系统。
DVPP	数字视觉预处理（Digital Vision Pre-Process），提供对特定格式的视频和图像进行解码、缩放等预处理操作，同时具有对处理后的视频、图像进行编码再输出的能力。
AIPP	AI（AI Pre Process）预处理，支持格式转换、Padding/Crop、CSC色域转换（YUV2RGB或者RGB2YUV）、Scale UP/Down、通道数据交换等。
OMG	离线模型生成（Offline Model Generator），用户使用Caffe/TensorFlow等框架训练好的模型，通过OMG将其转换为华为芯片支持的离线模型，实现算子调度的优化，权值数据重排、压缩，内存使用优化等可以脱离设备完成的模型优化功能。
OME	离线模型执行（Offline Model Inference Executor），转换完成的离线模型，使用OME进行模型的加载和推理。
Ctrl CPU	一个Ascend 310芯片中有4个Ctrl CPU，主要负责业务逻辑处理。
AI CPU	一个Ascend 310芯片中有4个AI CPU，主要用于算子任务调度、部分算子的实现。
AI Core	一个Ascend 310芯片中有2个AI Core，主要负责矩阵运算。
IPC	IP摄像机，提供RTSP数据流。
YUV420SP	有损图像颜色编码格式，常用YUV420SP_UV、YUV420SP_VU两种格式。

2 软件开发说明（重要）

Atlas 300 AI加速卡(型号 3000)是配套华为泰山服务器（Arm架构）开发的AI加速卡，针对泰山服务器进行硬件调整，不支持其他类型的服务器。在业务软件开发上，Atlas 300 AI加速卡(型号 3000)与Atlas 300 AI加速卡(型号 3010)使用相同的软件框架（Matrix）和API接口，业务代码可以通用，两者的区别如表2-1所示。

表 2-1 Atlas 300 AI 加速卡(型号 3010)与 Atlas 300 AI 加速卡(型号 3000)差异表

项目	Atlas 300 AI 加速卡(型号 3010)	Atlas 300 AI 加速卡(型号 3000)	差异说明
HOST处理器架构	x86处理器	Armv8处理器	Host侧软件的编译工具、动态链接库（第三方动态链接库、Ascend 310芯片相关的动态链接库）等差异。
跨侧数据传输带宽	PCIe3.0x4总线	PCIe3.0x2总线	通信带宽差异。

开发与执行环境的搭建请参考《Atlas 300 AI加速卡 用户指南 (型号 3000)》和《Atlas 300 AI加速卡 1.0.0 DDK 安装指南(*XXXX*，ARM) (型号 3000)》，请在华为泰山服务器上搭建。由Atlas发布的Sample和Demo，编译目标指定为Atlas 300 AI加速卡 (型号 3000)时，Host侧编译器为系统默认编译器，用户仅需提供支持华为泰山服务器的第三方动态链接库和头文件，即可在Atlas 300 AI加速卡 (型号 3000)上使用Sample和Demo。

3 开发前准备

3.1 获取样例程序
3.2 准备环境

3.1 获取样例程序

- Sample代码
  获取地址：**https://gitee.com/HuaweiAtlas/samples**
  使用方法：请参考gitee各Sample README.md文件。
- Demo代码
  获取地址：**https://gitee.com/HuaweiAtlas**
  使用方法：请参考gitee各Demo README.md文件。

3.2 准备环境

- 已将Atlas 300 AI加速卡 （型号 3000）插到兼容的服务器上并上电。
  Atlas 300 AI加速卡 （型号 3000）兼容的服务器请通过访问**华为服务器兼容性查询助手**查询所在服务器的“部件兼容性”。
- 已安装服务器的OS。
  确认Atlas 300 AI加速卡 （型号 3000）支持的操作系统请参见《Atlas 300 AI加速卡 用户指南 (型号 3000)》中的“规划与准备”章节。
- 已配置IP并接入互联网。

4 安装驱动和开发工具

Atlas 300 AI加速卡 （ 型号 3000 ）的版本升级分为如**表4-1**所示的三部分。

表 4-1 版本升级说明表

名称	开发环境	说明
驱动	服务器+Atlas 300 AI加速卡 （ 型号 3000 ）	执行环境下，需在Host系统中安装芯片驱动。
MCU固件	Atlas 300 AI加速卡 （ 型号 3000 ）的管理芯片	用于管理NPU芯片、采集温度。
DDK开发包	服务器+Ubuntu/CentOS对应版本	提供API、动态链接库、编译器等开发工具。

4.1 安装升级驱动

4.2 升级MCU固件

4.3 安装DDK开发工具

4.1 安装升级驱动

根据主机架构、系统版本和内核版本的不同，分别选择不同的软件包和安装指导书配合安装，驱动版本升级可直接覆盖安装（ 无需卸载旧版本驱动 ），详细的安装步骤请参考文档《 Atlas 300 AI加速卡 用户指南 (型号 3000) 》。

4.2 升级 MCU 固件

MCU是Atlas 300 AI加速卡 （ 型号 3000 ）板卡上的外带管理芯片，用于管理板卡上NPU芯片、采集板卡温度、与服务器的BMC管理软件交互等。

MCU是带外管理芯片，并不会影响业务软件的开发和运行，如需要使用监控板卡信息、管理NPU芯片等管理功能时，建议先升级MCU固件，固件升级方法请参考文档《 Atlas 300 AI加速卡 1.0.0.SPC200及以上 升级指导书（ 型号 3000, 3010 ）》或

《Atlas 300 AI加速卡 1.0.0.SPC200以下 升级指导书（型号 3000, 3010）》中的“升级MCU”章节。

4.3 安装 DDK 开发工具

DDK开发工具用于开发能使用NPU的业务程序，部署到开发环境，可与执行环境分离。

DDK工具包包含了Device侧的编译工具、头文件、动态链接库和Host侧的头文件、动态链接库。因此，DDK工具包对开发、执行和Device三个环境的操作系统及架构有强制要求，请用户根据需求选择合适的安装包安装。为了降低开发的复杂度，当前发布的DDK工具包的开发环境和执行环境要求是相同系统和架构。

DDK工具包的详细安装步骤请参考文档《Atlas 300 AI加速卡 1.0.0 DDK安装指南 (*XXXX*，*XX*) (型号 3000)》。

说明

- 以普通用户安装DDK，以root用户安装run包。
- *XXXX*，*XX*表示操作系统及主控芯片架构。

5 开发环境介绍

5.1 DDK目录分布
5.2 头文件与库介绍
5.3 编译工具介绍
5.4 开发工具介绍

5.1 DDK 目录分布

- 安装完DDK，需要将DDK安装路径导出至环境变量DDK_HOME，此处以用户HwHiAiUser为例（具体以用户环境的安装路径为准）：
  ```
  export DDK_HOME=/home/HwHiAiUser/tools/che/ddk/ddk
  ```
- 在“$DDK_HOME”中查看目录结构如下，其中“include”、“host/lib”、“device/lib”分别为头文件目录、host侧库目录、device侧库目录。
  ```
  ├─── bin
  ├─── conf
  ├─── ddk_info
  ├─── device
  │    └─── lib -> ../lib/aarch64-linux-gcc6.3
  ├─── host
  │    ├─── bin -> ../bin/x86_64-linux-gcc5.4
  │    └─── lib -> ../lib/x86_64-linux-gcc5.4
  ├─── include
  │    ├─── inc
  │    ├─── libc_sec
  │    └─── third_party
  ├─── lib
  │    ├─── aarch64-linux-gcc6.3
  │    └─── x86_64-linux-gcc5.4
  ├─── packages
  ├─── python3.5
  ├─── sample
  ├─── scripts
  ├─── site-packages
  ├─── toolchains
  └─── uihost
     ├─── bin -> ../bin/x86_64-linux-gcc5.4
     ├─── lib -> ../lib/x86_64-linux-gcc5.4
     └─── toolchains -> ../toolchains
  ```

5.2 头文件与库介绍

5.2.1 头文件

表 5-1 头文件说明

头文件	说明
include/inc	包含Matrix框架，DVPP，模型推理等头文件。
include/libc_sec	包含安全函数头文件。
include/third_party	包含第三方库头文件，omg工具使用，请用户自行安装使用。

头文件目录结构如下：

```
├─── include
│    ├─── inc
│    ├─── libc_sec
│    └─── third_party
```

5.2.2 动态库

host/lib：存放Host侧库文件

device/lib：存放Device侧库文件

其中也包含了protobuf，opencv等第三方库文件，显示如下：

```
├─── host/lib
│    ├─── libcrypto.so -> libcrypto.so.1.1
│    ├─── libcrypto.so.1.1
│    ├─── libc_sec.so
│    ├─── libdrvdevdrv.so
│    ├─── libdrvdsmi_host.so
│    ├─── libdrvdsmi.so
│    ├─── libdrvhdc_host.so
│    ├─── libdrvhdc.so
│    ├─── liblog.so
│    ├─── libmatrix.so
│    ├─── libmemory.so
│    ├─── libmmpa.so
│    ├─── libprofilerclient.so
│    ├─── libprofilerserver.so
│    ├─── libprotobuf.so -> libprotobuf.so.15
│    ├─── libprotobuf.so.15
│    ├─── libsecurec.so
│    ├─── libslog.so
│    ├─── libssl.so -> libssl.so.1.1
│    └─── libssl.so.1.1
```

5.3 编译工具介绍

5.3.1 Host 侧编译

Host侧代码编译，需要使用操作系统默认的gcc编译器来编译。

5.3.2 Device 侧编译

Device侧的代码编译，DDK提供了整套基于aarch64的gcc编译链工具，需要使用“$DDK_HOME/toolchains/aarch64-linux-gcc6.3”目录下的编译链工具进行交叉编译。

```
├─── aarch64-linux-gnu
├─── bin
│    ├─── aarch64-linux-gnu-addr2line
│    ├─── aarch64-linux-gnu-ar
│    ├─── aarch64-linux-gnu-as
│    ├─── aarch64-linux-gnu-c++
│    ├─── aarch64-linux-gnu-c++filt
│    ├─── aarch64-linux-gnu-cpp
│    ├─── aarch64-linux-gnu-elfedit
│    ├─── aarch64-linux-gnu-g++
│    ├─── aarch64-linux-gnu-gcc
│    ├─── aarch64-linux-gnu-gcc-6.3.0
│    ├─── aarch64-linux-gnu-gcc-ar
│    ├─── aarch64-linux-gnu-gcc-nm
│    ├─── aarch64-linux-gnu-gcc-ranlib
│    ├─── aarch64-linux-gnu-gcov
│    ├─── aarch64-linux-gnu-gcov-tool
│    ├─── aarch64-linux-gnu-gprof
│    ├─── aarch64-linux-gnu-ld
│    ├─── aarch64-linux-gnu-ld.bfd
│    ├─── aarch64-linux-gnu-nm
│    ├─── aarch64-linux-gnu-objcopy
│    ├─── aarch64-linux-gnu-objdump
│    ├─── aarch64-linux-gnu-ranlib
│    ├─── aarch64-linux-gnu-readelf
│    ├─── aarch64-linux-gnu-size
│    ├─── aarch64-linux-gnu-strings
│    └─── aarch64-linux-gnu-strip
├─── include
├─── lib64
├─── libexec
├─── libs.tar.gz
└─── sysroot
```

5.4 开发工具介绍

DDK工具目录：$DDK_HOME/bin/x86_64-linux-gcc5.4

工具功能说明请参考**表5-2**。

表 5-2 工具说明

名称	功能	说明
IDE-daemon-client	IDE Daemon命令工具集	具体说明请参考《IDE-daemon-client命令参考》。

名称	功能	说明
IDE-daemon-hiai	数据回传工具	• 图片预处理的时候，数据回传。 • 算子的数据从Device回传到Host侧。
omg	模型转换工具	可将Caffe或者Tensorflow模型转换为DDK支持的om模型文件，详情请参考《模型转换指导》。
protoc	第三方库protobuf的工具	用于将proto文件转换为各个语言支持的协议。

6 通用推理业务流程介绍

6.1 硬件业务流程
6.2 软件业务流程
6.3 软件模块与硬件模块对应关系示例

6.1 硬件业务流程

Atlas 300 AI加速卡 （型号 3000）（NPU卡）作为PCIe板卡，插在服务器的PCIe插槽上使用（一台服务器最多支持16张Atlas 300 AI加速卡 （型号 3000））。Atlas 300 AI加速卡 （型号 3000）卡内有4片NPU芯片，每个芯片有独立的操作系统。因此，业务软件根据运行环境分为Host侧与Device侧，需用户采用相对应API、动态链接库和编译工具进行开发。

- Host侧：即服务器侧。
- Device侧：即NPU芯片侧，每个芯片作为一个PCIE设备挂接在主机上。

Host与Device数据传输通过HDC驱动进行传输，硬件通道为PCIe通道，如图6-2所示，芯片内部CtrlCPU，DVPP、AIPP、AICore、AICPU等硬件模块功能介绍，详情请参见**关键概念**。

图 6-1 PCIe 拓扑图

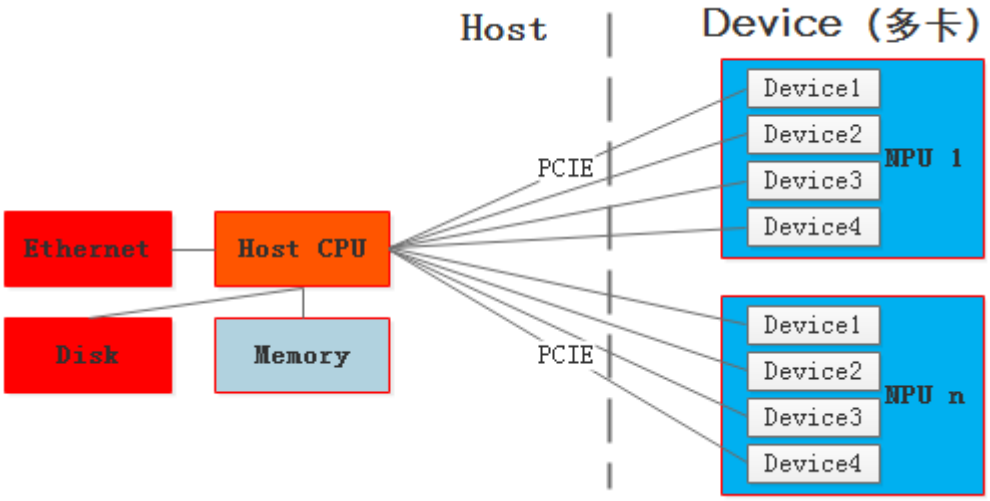
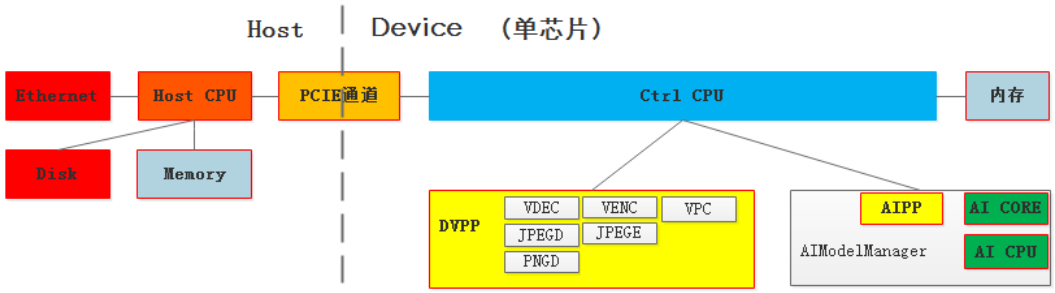


图 6-2 硬件模块图



6.2 软件业务流程

图 6-3 业务流程图

图6-3展示了软件业务流程，具体流程说明请参考表6-1。

表 6-1 业务流程说明

过程	描述	说明
数据源	视频或图片来源	可以使IPC摄像机提供RTSP数据流，或者磁盘离线视频文件，图片文件等。
数据获取	实现数据获取	可以选择开源FFmpeg函数库拉流，或自行实现代码拉流、读文件。运行在HostCPU。
数据预处理	实现图像解码、缩放、色域转换等图片预处理功能	• 选择软解码时，调用开源OpenCV函数库，运行在HostCPU或CtrlCPU。 • 选择硬解码时，调用DVPP解码缩放，调用AIPP实现色域转换，完成预处理后，在Device侧运行对应硬件模块。
推理	实现模型推理功能	运行在Device侧AI Core和AI CPU。
后处理	实现模型结果后处理	可以选择运行在Device侧Ctrl CPU或Host CPU。
输出	实现结果呈现	-

6.3 软件模块与硬件模块对应关系示例

Device侧的CtrlCPU为Arm处理器，由于处理能力有限，用户可根据业务流程、Host-Device数据传输及CtrlCPU负载情况，在Device侧实现通用的逻辑运算（预处理、后处理等）。

说明

用户可根据业务需求，选择使用软件预处理或基于DVPP的硬件预处理，两者具体差异请参考图6-5。

图 6-4 软件模块图

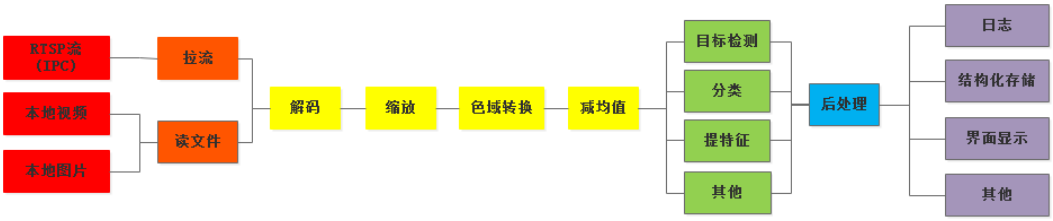
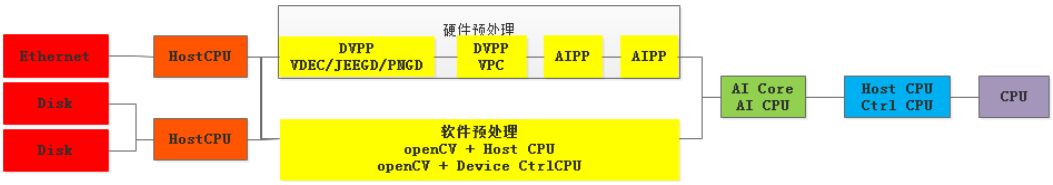


图 6-5 硬件模块图



7 代码运行样例（HelloDavinci 程序）

7.1 获取HelloDavinci代码
7.2 HelloDavinci文件说明
7.3 HelloDavinci流程框架介绍
7.4 HelloDavinci编译运行

7.1 获取 HelloDavinci 代码

获取地址：**https://gitee.com/HuaweiAtlas/samples**

7.2 HelloDavinci 文件说明

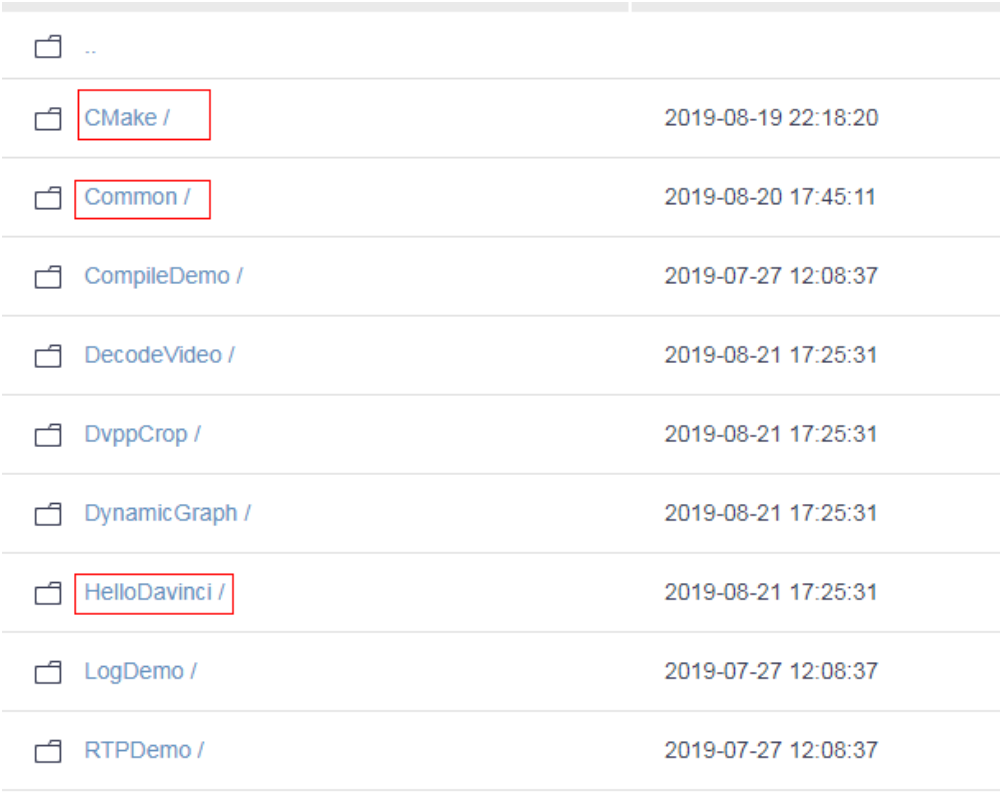
获取的文件samples包含了编译配置和Atlas 300 AI加速卡 （型号 3000）各个程序样例（HelloDavinci在其中）。如果单独运行HelloDavinci，需要Samples/Cmake，Samples/Common和Samples/HelloDavinci三个文件夹，如**图7-1**所示。

- CMake：存放cmake配置文件。
- Common：存放公共代码。
- HelloDavinci：HelloDavinci工程目录，主要包括build文件、源码文件、graph配置文件及README.md。

  说明

  请保持这三个文件的相对路径不变。

图 7-1 文件目录



编译工具文件Cmake目录结构如下所示：

```
├────Ascend.cmake    //Device侧编译链
└────FindDDK.cmake   //cmake寻找DDK模块
```

HelloDavinci的目录结构如下所示：

```
├──── build              //编译文件夹，包括Host和Device侧的编译
│     ├──── CMakeLists.txt
│     ├──── device
│     └──── host
├──── build.sh           //编译脚本
├──── README.md          //README.md
├──── main.cpp           //主函数入口
├──── include            //HelloDavinci公共模块
├──── DstEngine          //DstEngine(host侧)
│     ├──── DstEngine.cpp
│     └──── DstEngine.h
├──── graph.config       //graph配置文件
├──── HelloDavinci       //HelloDavinci Engine(device侧)
│     ├──── HelloDavinci.cpp
│     └──── HelloDavinci.h
└──── SrcEngine          //SrcEngine Engine(host侧)
│     ├──── SrcEngine.cpp
│     └──── SrcEngine.h
```

7.3 HelloDavinci 流程框架介绍

本章节介绍HelloDavinci样例代码的流程。

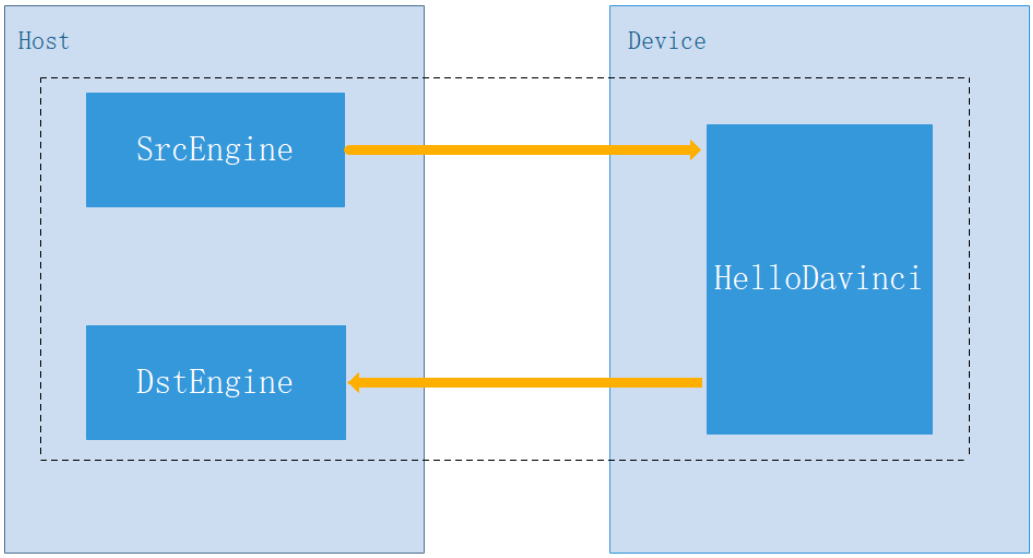
说明

如无需了解本章节内容可跳过本节到**7.4 HelloDavinci编译运行**直接进行编译运行，查看运行结果。

本开发样例主要是演示从Host侧发送数据到Device侧，再从Device侧获取生成的字符串发送回Host侧，保存结果，并且打印到终端。如图7-2所示，整个程序分为两部分运行，Host侧（包括SrcEngine和DstEngine）和Device侧（包括HelloDavinci），运行过程如下：

1. 运行从main开始，向SrcEngine发数据。
2. SrcEngine收到数据之后，转发给HelloDavinci，HelloDavinci在内部生成字符串“This message is from HelloDavinci”并发送到DstEngine。
3. DstEngine收到数据后将信息保存在目录“${workPath}/out/dacvinci_log_info.txt”(${workPath}为工程根目录)下，并向main发送结束信号。
4. main函数收到结束信号后，销毁graph，在终端打印结束信息并退出程序。

图 7-2 HelloDavinci 流程框架图



7.4 HelloDavinci 编译运行

前提条件

- 已完成环境配置。
- 已安装Atlas产品驱动。
- 已安装第三方CMake编译工具（2.8.4版本以上）。
  若无CMake编译工具，请自行安装。
- 已安装DDK并配置好DDK_HOME。

操作步骤

步骤1 将7.2 HelloDavinci文件说明章节获取到的samples文件拷贝到开发主机的某一个目录下。

步骤2 进入“HelloDavinci”工程目录。

步骤3 执行命令**bash build.sh**进行编译，编译成功后将在“${workPath}/out/”目录下生成可执行文件main和Device侧的so文件（libDevice.so）。

其中“${workPath}”为“HelloDavinci”目录。

步骤4 执行命令 **./out/main**，查看结果输出，如返回如下结果，则表示运行成功。

```
[root@host build]# ./out/main
Hello Davinci!
The sample end!!
[root@host build]# cat out/davinci_log_info.txt
This message is from HelloDavinci
```

说明

- Atlas 300 AI加速卡（型号 3000）驱动安装后，root及HwHiAiUser两个用户组的用户可以调用。
- 运行可执行文件时，请确认用户归属。

----结束

8 软件代码开发

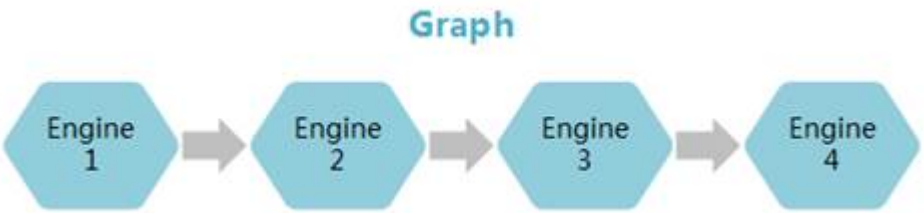
8.1 配置Matrix框架
8.2 调用DVPP接口
8.3 离线模型推理
8.4 优化软件性能
8.5 软件日志调测
8.6 业务软件编译
8.7 Demo样例解析

8.1 配置 Matrix 框架

为了高效使用Ascend 310芯片的算力，提供了Matrix框架来完成推理业务迁移，Matrix框架提供的功能如下：

- 流程编排：
  a. 定义Engine为流程的基本功能单元，同时允许用户自定义Engine的实现（输入图片数据、对图片进行分类处理、输出对图片数据的分类预测结果等）。每个Engine在Ascend 310端默认对应一个线程来运行处理。
  b. 定义关于Graph管理若干Engine的流程。每个Graph在Ascend 310端对应一个进程运行处理。Graph与Engine关系如图8-1所示。

  图 8-1 Graph 与 Engine 关系

  在Graph配置文件中配置Engine节点间的串接和节点属性（运行该节点所需的参数），节点间数据的实际流向根据具体业务在节点中实现，通过向业务的开始节点灌入数据启动整个Engine计算流程。

- 媒体预处理：

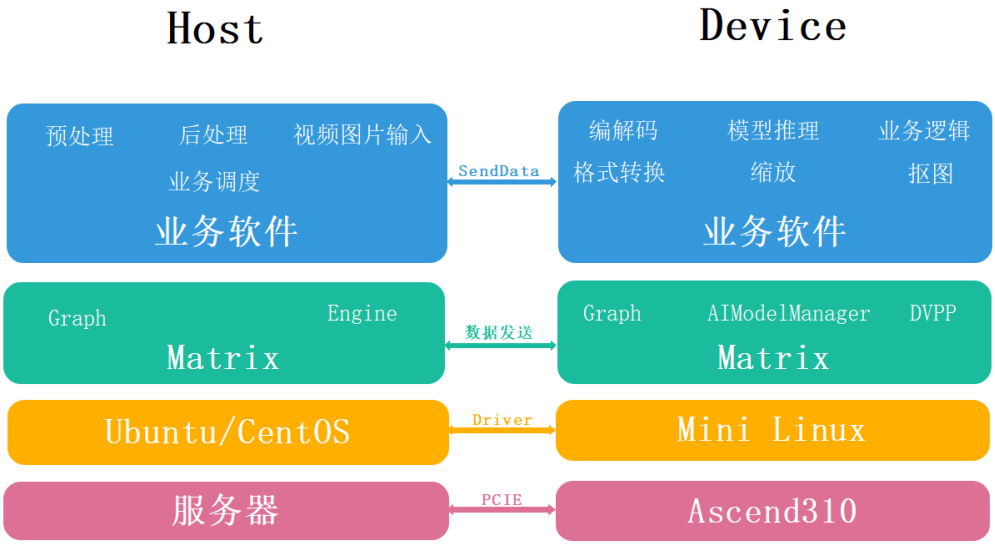
  运行在Ascend 310上的Engine可直接调用DVPP提供的API接口实现媒体预处理能力。

- 离线模型加载和运行：

  运行在Ascend 310上的Engine可直接调用模型管家（AIModelManger）提供的API进行离线模型加载和推理功能。

  基于Matrix框架，用户的业务软件结构如下图所示，用户创建自定义的Engine组成业务流（Graph），运行在Device侧的Engine可以调用DVPP和AIModelManager的API，使用Ascend 310的媒体预处理和模型推理的硬件加速功能。Host侧的Engine主要实现业务软件逻辑及与Device侧Engine间的数据传输功能。

图 8-2 业务软件框架



8.1.1 配置、创建与销毁 Graph

Graph主要功能是描述业务包含的Engine和Engine间的数据传输关系，Matrix框架通过Protobuf定义了Graph的数据结构，用户以配置文件的方式来定义Graph配置。

说明

Graph关键字含义请参考**9.1 Graph关键字**，详情请参考《Matrix API参考》。

下文提供了一个简单的graph配置文件，该graph文件会创建一个id为1000的graph业务流，该业务流包含三个Engine，Engine间的数据传输关系如**图8-3**所示。

图 8-3 Engine 间的数据传输关系


```
graphs {
    graph_id: 1000          # 单芯片支持多graph，graph id需大于0，且不能重复
    device_id: "0"          # graph运行的芯片id
    priority: 1             # 优先级
```

```
  # 用户自定义Engine，可实例化多个Engine，以id区分
  engines {
    id: 2001                              # 引擎ID
    engine_name: "ObjectDetectionEngine"     # 用户自定义Engine的类名
    side: DEVICE                          # 指定引擎运行在Host/Device侧
    so_name: "./libObjectDetectionEngine.so" # 动态链接库名称及在Host侧的路径，由Matrix框架拷贝到Device侧
    # 用户可自定义配置参数
    ai_config{
      items{
        name: "model"                  # 模型名称
        value: "./FaceDetection.om"     # 模型路径在Host侧的路径，由Matrix框架拷贝到Device侧
      }
      items{
        name: "mode"
        value: "test"
      }
    }
  }
  engines {
    id: 1000
    engine_name: "DecodeEngine"
    side: DEVICE
    so_name: "./libDecodeEngine.so"
  }
  engines {
    id: 2002
    engine_name: "PostProcess"
    side: HOST
  }

  # 描述Engine间的端口连接
  connects {
    src_engine_id: 1000                  #数据传输源引擎ID
    src_port_id: 0                      #数据传输源端口号
    target_engine_id: 2001                #数据传输目的引擎ID
    target_port_id: 0                   #数据传输目的端口号
  }
  connects {
    src_engine_id: 2001
    src_port_id: 0
    target_engine_id: 2002
    target_port_id: 0
  }
}
```

Matrix框架会将Device侧Engine依赖的文件动态链接库（由so_name指定），模型文件（在ai_config内添加）等拷贝到Device侧，供Engine线程使用。Matrix提供的三个常用Graph接口如**表8-1**所示，详细介绍请参考《Matrix API参考》中的“流程编排接口”章节。

表 8-1 接口说明

接口	说明
HIAI_StatusT HIAI_Init(uint32_t deviceID)	Ascend 310芯片初始化，请注意此处的芯片ID编号是Ascend 310芯片的绝对编号。 一台服务器支持16卡，64芯片，DeviceID范围为[0， 63]，npu-smi查询的芯片ID是相对编号。
static HIAI_StatusT Graph::CreateGraph(const std::string& configFile)	读取Graph配置文件，初始化Engine，创建线程和数据传输通道，完成业务流初始化。
static HIAI_StatusT Graph::DestroyGraph(uint32_t graphID)	销毁Graph，运行Engine的析构函数等。

8.1.2 配置 Engine

Engine是Matrix框架定义的业务软件基本功能单元，用户可继承Matrix定义的Engine模板类，创建业务中各功能模块的Engine（读取输入文件，图像预处理，神经网络推理，推理结果后处理，host/device数据传输等），详情请参考《Matrix API参考》。每一个Engine定义了函数Init()和Process()，在Graph初始化时，会自动运行Init()，从而实现Engine的参数初始化（包含内存分配和模型加载）。Process()接口实现数据的传输和业务逻辑。

- 常用接口介绍如表8-2所示。

表 8-2 接口说明

接口	描述	说明
HIAI_DEFINE_PROCESS（inputPortNum, outputPortNum）	端口数量（Engine的数据输入和输出数据通道）	Matrix框架为每一端口创建了队列用于缓存数据，Engine间端口的传输关系在Graph配置文件内指定，请参考9.1 Graph关键字。
HIAI_StatusT Engine::Init(const AIConfig &config, const vector<AIModelDescription>&modelDesc)	Engine初始化	在创建Graph时此接口会被调用，初始化Engine，Graph配置文件中定义的ai_config会传给函数入参config，用户可在配置文件内添加自定义item，并在初始化函数内使用此item。

接口	描述	说明
HIAI_IMPL_ENGINE_PROCESS(name, engineClass, inPortNum)	Engine的Process，在Device侧对应线程	由数据驱动，即输入端口收到数据后，由框架启动Process运行，如果设置了多个输入端口，每一个输入端口接收到数据后，都会触发一次Process运行，因此如果业务处理依赖多个输入，用户需自行实现多输入的同步逻辑。框架已将Process封装成宏定义，用户实现该宏定义即可。

- Engine读取输入端口的数据，框架支持最大16个输入端口（arg0~arg15），用户可直接使用，从业务应用角度看，数据传输的是共享指针，传输代码示例如下：

```
// 发送Engine：传输自定义数据USER_DEFINE_TYPE到目标Engine
std::shared_ptr<USER_DEFINE_TYPE > streamData= std::make_shared< USER_DEFINE_TYPE >();
// 对streamData进行赋值后，调用Senddata发送，发送需要将共享指针转换为void类型
hiai::Engine::SendData(0, " USER_DEFINE_TYPE ", std::static_pointer_cast<void>(deviceStreamData));
// 接收Engine：接收数据，并将数据转换为用户自定义类型USER_DEFINE_TYPE
std::shared_ptr< USER_DEFINE_TYPE > inputArg = std::static_pointer_cast< USER_DEFINE_TYPE >(arg0);
```

8.1.3 配置数据传输

Matrix框架定义的数据传输，根据业务应用分为以下三类，接口类似、传输的对象都为共享指针，但是根据场景不同，接口使用方式有明显区别。

- Graph外传输数据到Engine的输入端口：请参考《Matrix API参考》中的“Graph::SendData”章节。
- Graph内的Engine输出端口传输数据到Graph外：请参考《Matrix API参考》中的“Graph::SetDataRecvFunctor”和“Engine::SetDataRecvFunctor”章节。
- Graph内Engine间的数据传输：请参考《Matrix API参考》中的“Engine::SendData”章节。

Engine 间的数据传输

Matrix框架将业务软件分为Host侧（X86/Arm服务器）和Device侧（Ascend 310芯片）两部分软件，因此Engine间的数据传输分为跨侧传输和同侧传输，说明如下：

- 跨侧传输：传输的数据需要序列化为二进制，通过PCIE或DMA等硬件完成数据传输后，反序列为有效数据。因此对于自定义数据结构，需要用户自定义序列化与反序列化函数，Matrix框架主要提供两种序列化/反序列函数定义方式，分别为普通接口和高速接口，普通接口适合256K以下的数据传输，高速接口适合256K以上的数据传输（高速接口传输小内存块，速度与普通接口类似），详情请参考《Matrix API参考》中的“数据类型序列化和反序列化（C++语言）”章节。
- 同侧传输：传输的数据即为共享指针的地址，并未拷贝，由于Engine是线程运行，该方式即为线程间用共享内存的方式，实现数据传输，需要用户保证不出现非法访问。Matrix框架推荐用于传输的共享指针，在传输给下一个Engine后，本Engine不再修改共享指针。
- 跨侧传输在实现序列化与反序列化函数之后，在业务应用上采用的接口与同侧传输一致。


```
HIAI_StatusT Engine::SendData(uint32_t portId, const std::string& messageName,
const shared_ptr<void>& dataPtr， uint32_t timeOut = TIME_OUT_VALUE);
```

Graph 外传数据到 engine 的输入端口

- 由Engine组成的业务流，由数据驱动，数据从Graph外传输到Graph内，即通过该接口实现。
- 以下接口可实现跨侧传输和同侧传输，要求与Engine间的数据传输一致，详情请参考《 Matrix API参考 》中的“Graph::SendData”章节。

```
HIAI_StatusT Graph::SendData(const EnginePortID& targetPortConfig, const
std::string& messageName, const std::shared_ptr<void>& dataPtr，const uint32_t
timeOut = 500)
```

Graph 内的 Engine 输出端口传输数据到 Graph 外

框架提供了回调函数的模板类DataRecvInterface，框架要求回调函数与输出Engine运行于同侧，即支持同侧传输，不支持跨侧传输。Matrix采用以下两种设置回调函数的方式将Engine输出端口的数据传输到Graph外：

- 业务流节点Engine输出端口回调，请参考《 Matrix API参考 》中的“Graph::SetDataRecvFunctor”章节。
- 其他场景，Engine输出端口回调，请参考《 Matrix API参考 》中的“Engine::SetDataRecvFunctor”章节。

8.2 调用 DVPP 接口

DVPP是Ascend 310芯片提供的图像预处理硬件加速模块，该模块集成的六个功能如下所示，接口介绍和使用方法请参考《 DVPP API参考 》。

- 格式转换，抠图与缩放（ VPC ）
- H264/H265视频解码（ VDEC ）
- H264/H265视频编码 （ VENC ）
- Jpeg图片解码（ JPEGD ）
- Jpeg图片编码（ JPEGE ）
- Png图片解码（ PNGD ）

8.2.1 使用 DVPP 接口

DVPP采用句柄方式提供接口，主要包含三类接口：创建，使用，销毁。

VPC、JPEGE、JPEGD、PNGD共用一套接口，输入的参数有差异。VDEC和VENC各使用一套接口。

- VDEC解码使用的三个接口如下所示，用户以异步方式调用，使用VdecCtl接口传入配置（包含回调函数）、H264/H265数据等，硬件解码后，Matrix框架调用回调函数将结果返回。

  说明

  VDEC解码后的数据为HFBC（内部格式），用户需要使用VPC将HFBC转换为YUV420SP格式，详情请参考《 DVPP API参考 》的“实现 VDEC 功能”章节中的相关示例。

```
int CreateVdecApi(IDVPPAPI *&pIDVPPAPI, int singleton)
int VdecCtl(IDVPPAPI *&pIDVPPAPI, int CMD, dvppapi_ctl_msg *MSG, int singleton)
int DestroyVdecApi(IDVPPAPI *&pIDVPPAPI, int singleton)
```


- VPC、JPEGE、JPEGD、PNGD使用如下所示相同的接口，不同功能传输的配置参数不同，请用户参考《 DVPP API参考 》中的“VPC/JPEGE/JPEGD/PNGD功能接口”章节。

```
int CreateDvppApi(IDVPPAPI *&pIDVPPAPI)
int DvppCtl(IDVPPAPI *&pIDVPPAPI, int CMD, dvppapi_ctl_msg *MSG)
int DestroyDvppApi(IDVPPAPI *&pIDVPPAPI)
```

- 由于硬件限制，DVPP使用过程中存在部分限制。为了加快读写速度，图片长宽需要对齐到指定大小，但不影响有效区域，采用向左、向下填充0的方式，对齐到指定大小。

  例如：对于300*300的YUV420SP_UV图片，需要对齐到304*300（宽要求16对齐，高要求2对齐），有效区域仍为[0, 0]至[300, 300]，需要用户在图片右侧补零，对齐到304列。

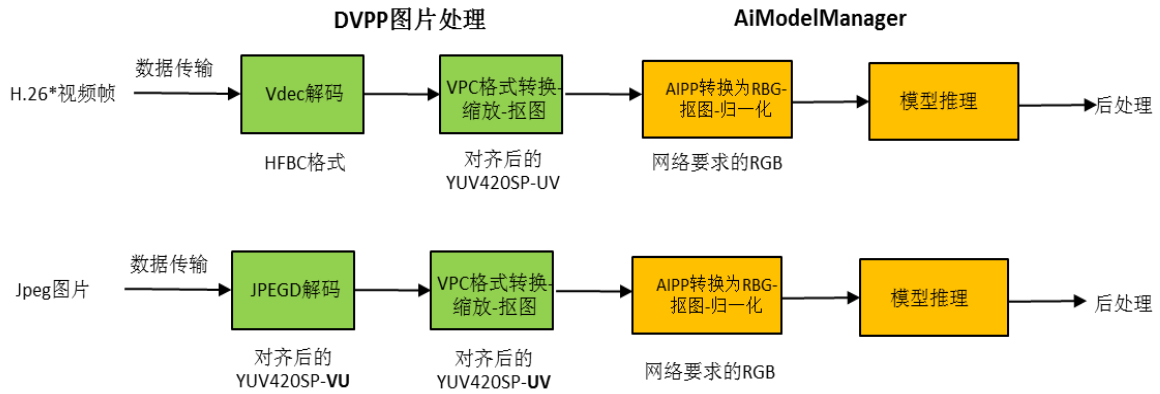
  - 当用户使用DVPP的JPEGD、VDEC、PNGD组件读取输入图片时，解码后的图片需要满足长宽对齐要求（实际长宽），此时用户需要注意，以对齐后的图片大小申请输出图片内存。

    例如：对于300*300的YUV420SP_UV图片，需要申请304*300*3/2Byte（YUV420SP一个像素需要1.5Byte存储）。

    - VPC输入和输出内存地址，16字节对齐。
    - VPC输出图片的宽度，16字节对齐。
    - VPC输出图片的高度，2字节对齐。
    - VPC输入图片的宽度，16字节对齐。
    - VPC输入图片的高度，2字节对齐。
    - JPEGD输出图片的宽度，128字节对齐。
    - JPEGD输出图片的高度，16字节对齐。
  - DVPP各组件基于处理速度和内存占用量的考虑，对输出图片有诸多限制，如输出图片需要长宽对齐，输出格式必须为YUV420SP等，但模型输入通常为RGB或BGR，且输入图片尺寸各异。因此，Ascend310芯片提供AIPP（Ai Preprocess），用于图片格式转换及抠图功能，具体说明请参考《 模型转换指导 》。

    例如：Jpeg图片输入和H26*视频输入的处理流程如图8-4所示。

图 8-4 视频与图片输入处理流程



8.2.2 申请 DVPP 内存

Matrix针对DVPP提供了内存申请接口HIAI_DVPP_DMalloc和释放接口HIAI_DVPP_DFree，申请接口用于申请满足DVPP对齐要求的内存。用户在使用时注意该接口需要成对使用，为有效预防内存泄露问题，推荐使用共享指针管理申请的内存，实现代码如下：

```
uint8_t* buffer = nullptr;
HIAI_StatusT ret = hiai::HIAIMemory::HIAI_DVPP_DMalloc(dataSize,  (void*&) buffer);
std::shared_ptr<uint8_t> dataBuffer =  std::shared_ptr<uint8_t>( buffer, \
[](std::uint8_t* data) hiai::HIAIMemory::HIAI_DVPP_DFree(data);});
```

此外，该接口仅在Device侧使用，Host侧无该接口。HIAI_DVPP_DMalloc申请的内存可用于Device到Host的高速数据传输，在此方式下，内存由Matrix框架释放，详情请参考sample代码DvppDecodeResize。

8.3 离线模型推理

Ascend 310芯片有利于加速Caffe和Tensorflow框架的推理。用户在完成训练后，需要先将训练后的模型转换成Ascend 310支持的模型文件（om文件），再编写业务代码，然后调用Matrix框架提供的接口完成业务功能。

Matrix框架将AIPP（Ai Preprocess）和模型推理功能封装至一个模块内， 用户调用推理接口后，Matrix框架会先调用AIPP完成输入图片的预处理，再将预处理后的图片输入到模型推理模块内，最后返回推理后的结果。

8.3.1 配置 AIPP

AIPP是Ascend 310提供的硬件图像预处理功能，包括色域转换，图像归一化（减均值/乘系数）和抠图（指定抠图起始点，抠出神经网络需要大小的图片）等功能。

AIPP分为静态AIPP和动态AIPP：

- 静态AIPP：在模型转换时设置参数，模型推理过程采用固定的AIPP预处理（无法修改），静态AIPP的关键字含义和配置文件模板请参考《模型转换指导》中的“配置文件模板”章节。
- 动态AIPP：在模型转换时，设置AIPP为动态模式，每次模型推理前，根据需求，设置AIPP预处理参数。动态AIPP在根据业务要求改变预处理参数的场合下使用（如不同摄像头采用不同的归一化参数，输入图片格式需要兼容YUV420和RGB等）。动态AIPP的使用方法请参考《Matrix API参考》中的 “AIPP配置接口”章节。

须知

- AIPP的输入格式为“YUV420SP_U8”（默认为“YUV420SP_UV”格式），若格式为“YUV420SP_VU”，请修改参数“rbuv_swap_switch”，否则会影响输出结果。
- 模型输入为“RGB_U8”或“BGR_U8”，对应不同的色域转换矩阵。

DVPP模块输出的图片多为对齐后的YUV420SP类型，不支持输出RGB图片。因此，业务流需要使用AIPP模块转换对齐后YUV420SP类型图片的格式，并抠出模型需要的输入图片。

例如：模型要求输入为300*300的RGB图片，用户调用DVPP接口处理（如Jpeg解码和缩放）后，由于对齐要求，输出的图片为384*304（图片有效区域为300*300，向右向下填充零补齐）的YUV420SP_UV格式图片。

静态AIPP的配置如下所示，文件配置了抠图的起始坐标，抠图的长宽默认为模型输入，图像归一化参数，即均值和方差的倒数（减均值后，乘以该系数）。

```
aipp_op{
# AIPP为静态模式
aipp_mode: static
# 使能抠图
crop: true
# 输入图片的格式及大小
input_format : YUV420SP_U8
src_image_size_w : 384
src_image_size_h : 304
# 抠图的起始坐标，抠图的宽与高默认为模型输入
load_start_pos_h : 0
load_start_pos_w : 0

# 使能格式转换，该转换矩阵将YUV420SP_UV转换为RGB888
csc_switch : true
matrix_r0c0 : 298
matrix_r0c1 : 516
matrix_r0c2 : 0
matrix_r1c0 : 298
matrix_r1c1 : -100
matrix_r1c2 : -208
matrix_r2c0 : 298
matrix_r2c1 : 0
matrix_r2c2 : 409
input_bias_0 : 16
input_bias_1 : 128
input_bias_2 : 128

#开启数据归一化，配置均值和方差的倒数
mean_chn_0 : 125
mean_chn_1 : 125
mean_chn_2 : 125
var_reci_chn_0 : 0.0039
var_reci_chn_1 : 0.0039
var_reci_chn_2 : 0.0039
}
```

8.3.2 转换离线模型

训练后的Caffe和Tensorflow模型，有两种初步评估的方法评估Ascend 310芯片是否支持用户模型，评估方法如下：

1. 模型使用的算子是否全部包含在算子清单内，包含即支持。
2. 使用模型转换工具直接转换训练好的模型，确认是否可以转换成功，转换成功即支持，不支持的算子将返回错误信息。

说明

1. 算子清单可参考《算子清单》。
2. 如存在不支持的算子，请参考《TE自定义算子开发指导》开发不支持的算子。

离线模型转换工具包含在DDK工具包内，位于目录“<$DDK_HOME>/uihost/bin/omg”下，omg为命令行工具（可通过**-h**获得参数信息），用于实现Caffe和

Tensorflow模型转换成Ascend 310支持的om文件。Omg工具的使用说明请参考《模型转换指导》中的“使用omg工具转换模型”章节。

1. Caffe模型的转换：
```
#omg --framework 0 --model <model.prototxt> --weight <model.caffemodel> --output <output name> --insert_op_conf <aipp.cfg>
```
2. Tensorflow模型的转换：
```
#omg --framework 3 --model <model.pb> --input_shape “input_name:1,112,112,3” --output <output_name> --insert_op_conf <aipp.cfg>
```

表 8-3 参数说明

参数	说明
framework 0	指定为Caffe模型。
framework 3	指定为Tensorflow模型。
model	指定模型文件。
weight	指定Caffe的权重文件。
output	支持输出om文件的文件名。
input_shape	指定输入层的名字和大小，Tensorflow默认为“input_layer_name: n, h, w, c”。
insert_op_conf	指定AIPP的配置文件。

8.3.3 执行模型推理

Matrix框架提供AIModelManager类，实现模型加载和推理功能，详情请参考《Matrix API参考》。

模型推理初始化

步骤1 在自定义推理模型Engine的Graph配置文件内设置模型路径（在ai_config内，添加items，设置模型在host路径）。

步骤2 使用Matrix框架将模型文件传输到Device侧。

步骤3 用户在自定义Engine内解析自定义items，获得模型在Device侧的路径。

步骤4 调用AIModelManager::Init()完成初始化，具体实现如下：

```
/* 用户自定义Engine的成员变量 */
std::shared_ptr<hiai::AIModelManager> modelManager;
/* 自定义Engine Init函数内实现AIModelManager初始化 */
std::vector<hiai::AIModelDescription> model_desc_vec;
hiai::AIModelDescription model_desc_;
......
/* 从Graph配置文件的ai_config结构内，解析出模型路径 */
model_desc_.set_path(model_path);// 设置模型路径
model_desc_vec.push_back(model_desc_);
ret = modelManager->Init(config, model_desc_vec);// config无意义，将Engine::Init的入参传入即可
```

----结束

设置模型推理输入与输出

- Matrix框架定义IAITensor类，用于管理模型推理的输入与输出矩阵。为了便于使用，Matrix框架基于IAITensor，派生出了AISimpleTensor和AINeuralNetworkBuffer。
- 模型推理的输入和输出采用HIAI_DMalloc接口申请，可减少一次内存拷贝。
- Matrix框架可自动释放AISimpleTensor管理的内存，建议由用户申请和释放，防止出现内存泄露或重复释放。
- 模型转换过程中，如启用了AIPP的抠图、格式转换、图片归一化等功能，则输入数据需经AIPP模块处理，再将处理后的数据进行真正的模型推理。

输入与输出的具体实现

输入输出的具体实现代码如下：

```
/* 获取推理模型的输入输出Tensor描述 */
std::vector<hiai::TensorDimension> inputTensorDims;
std::vector<hiai::TensorDimension> outputTensorDims;
ret = modelManager->GetModelIOTensorDim(modelName, inputTensorDims,
outputTensorDims);

/* 设置输入, 如果设置多个输入，需要依次创建并设置 */
std::shared_ptr<hiai::AISimpleTensor> inputTensor =
std::shared_ptr<hiai::AISimpleTensor>(new hiai::AISimpleTensor());
inputTensor->SetBuffer(<输入数据的内存地址>, <输入数据的长度>);
inputTensorVec.push_back(inputTensor);

/* 设置输出 */
for (uint32_t index = 0; index < outputTensorDims.size(); index++) {
hiai::AITensorDescription outputTensorDesc = hiai::AINeuralNetworkBuffer::GetDescription();
uint8_t* buf = (uint8_t*)HIAI_DMalloc(outputTensorDims[index].size);
.....
std::shared_ptr<hiai::IAITensor> outputTensor = hiai::AITensorFactory::GetInstance()-
>CreateTensor(
outputTensorDesc, buf, outputTensorDims[index].size);
outputTensorVec.push_back(outputTensor);
}
```

模型推理

- Matrix框架提供了同步推理和异步推理两种方式，默认为同步推理，可通过设置AIContext配置项，使用回调函数实现异步推理。

  ```
  /* 模型推理 */
  hiai::AIContext aiContext;
  HIAI_StatusT ret = modelManager->Process(aiContext, inputTensorVec, outputTensorVec,
  0);
  ```

- AIModelManager对象加载多个模型，可通过设置AIContext配置项，指定模型（初始化阶段，指定的模型名），详情请参考《Matrix API参考》中的“离线模型管家”章节。

模型推理后处理

模型推理的结果矩阵，以内存+描述信息的方式保存在IAITensor对象内，用户需要根据模型的实际输出格式（数据类型和数据顺序），将内存解析成有效的输出。

```
/* 推理结果解析 */
for (uint32_t index = 0; index < outputTensorVec.size(); index++) {
```



```
shared_ptr<hiai::AINeuralNetworkBuffer> resultTensor =
std::static_pointer_cast<hiai::AINeuralNetworkBuffer>(outputTensorVec[i]);
// resultTensor->GetNumber()  -- N
// resultTensor->GetChannel() -- C
// resultTensor->GetHeight()  -- H
// resultTensor->GetWidth()   -- W
// resultTensor->GetSize()    -- 内存大小
// resultTensor->GetBuffer()  -- 内存地址
}
```

说明

常见的分类模型后处理可参考示例代码“InferClassification”，SSD目标检测模型的后处理可参考示例代码“InferObjectDetection”。

8.4 优化软件性能

软件业务流开发过程中，需要考虑性能优化（内存使用，数据传输，高性能算子选择等），高性能软件编程建议请参考《高性能应用编程用户手册》。

8.5 软件日志调测

本章节介绍Atlas平台软件开发过程中如何编写日志打印和日志查看的代码，这些代码用于软件开发过程中打印调测日志，软件运行过程中记录异常。

说明

Device侧代码在Ascend 310芯片上的操作系统上运行，无法使用printf或cout将调试信息输出到Host的终端界面，需使用本节介绍的日志系统。

8.5.1 日志注册

在使用日志接口之前，先进行错误码注册（使用日志注册相关接口需要在头文件“hiaiengine/status.h”中定义）。

操作步骤

步骤1 自定义日志模块ID，实现如下：

```
#define USE_DEFINE_ERROR 0x6001  //0x6001为自定义值，值不能跟status.h文件中的MODID_GRAPH等重复。
```

步骤2 自定义错误码的名称，实现如下：

```
enum{
    HIAI_INVALID_INPUT_MSG_CODE = 0 // 错误码的名称，0为自定义值
};
```

步骤3 注册错误码，实现示例如下，参数说明如表8-4所示。

```
HIAI_DEF_ERROR_CODE(moduleId, logLevel, codeName, codeDesc)
```

表 8-4 参数说明

参数	说明
moduleId	模块ID。

参数	说明
logLevel	错误级别，级别说明如下： ● 调试： HIAI_DEBUG ● 正常： HIAI_INFO ● 警告：HIAI_WARNNING ● 一般错误：HIAI_ERROR
codeName	错误码的名称。
codeDesc	错误码的描述，字符串。

步骤4 日志注册调用，实现示例如下：

```
HIAI_DEF_ERROR_CODE(USE_DEFINE_ERROR, HIAI_ERROR, HIAI_INVALID_INPUT_MSG, "invalid input message pointer");
```

----结束

8.5.2 日志打印

调用HIAI_ENGINE_LOG打印日志，根据传入不同的参数，调用对应HIAI_ENGINE_LOG_IMPL函数进行日志打印，打印相关的接口在log.h中定义。日志打印格式分为八种，本章结合日志注册的内容，介绍其一。在软件开发调试的过程中，如果在Device侧调用了“cout”“printf”等输出函数，由于Host和Device是两个独立的系统，故在Host是无法看到打印内容的。因此，请用户使用HIAI_ENGINE_LOG来代替此类方法。

- 函数格式显示如下，参数说明如表8-5所示。

```
#define HIAI_ENGINE_LOG(...) \
HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)
void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber, const uint32_t  errorCode, const char* format, ...);
```

表 8-5 参数说明

参数	说明
errorCode	错误码
format	日志描述
...	format中的可变参数，根据日志内容添加

- 日志打印调用，实现示例如下：

```
HIAI_ENGINE_LOG(HIAI_INVALID_INPUT_MSG, "RUNNING OK");
```

8.5.3 日志查看

更改日记级别

框架提供5种日志级别：ERROR>WARNING>INFO>DEBUG>EVENT

EVENT级别的日志输出全系统最关键日志，需要单独设置。其余级别设置了日志对应级别后，此级别及高于此级别等级的日志都能打印出来。

1. 通过执行命令行方式修改日志等级。

   命令格式如下，其中的参数说明如表8-6所示。

   IDE-daemon-client --host <host_ip>:<port> [--device <deviceID>] --log '<params>'

   表 8-6 参数说明

参数		描述	是否为必选	默认值
--host <host_ip>:<port>	host_ip	host ip地址	是	无
	port	指定端口	是	22118
--device <deviceID>	deviceID	device ID号，在多个device中指定特定device	否	0
--log '<params>' **说明** params包括以下两种格式： 1. SetLogLevel(scope)[moduleName:level] 2. SetLogLevel(scope)[flag]	scope	日志模块范围，说明如下： ● 0：全局 ● 1：模块 ● 2：event	是	无
	moduleName	日志模块名称 例如：dlog、slog、cce等	否	无
	level	指定模块日志级别 例如：error、info、warning、debug、null	否	无
	flag	当scope=2时： ● enable：开启event日志 ● disable：不开启event日志	否	无

   此方法的操作仅在普通用户下执行（本例中的用户为HwHiAiUser）。操作前，以HwHiAiUser用户登录Host侧服务器。

   操作示例如下：

   a. 设置全局日志等级为error级别。

      IDE-daemon-client --host xx.xx.xx.xx:22118 --log 'SetLogLevel(0)[error]'

   b. 设置slog模块的日志级别为error级别。

IDE-daemon-client --host xx.xx.xx.xx:22118 --log 'SetLogLevel(1)[slog:error]'

c. 打开event日志级别。

IDE-daemon-client --host xx.xx.xx.xx:22118 --log 'SetLogLevel(2)[enable]'

2. 通过修改配置文件来修改日志等级：

在“/etc/slog.conf”目录文件下， 更改“global_level”的值，即可修改日志的等级。各个日志等级的对应“global_level”的值如表8-7所示。

例如：若“global_level”的值为“3”，则打印的日志等级为ERROR，此时只有ERROR级别的日志被打印，Host侧和Device侧需要单独配置 。

表 8-7 日志等级说明

参数值	级别
0	DEBUG
1	INFO
2	WARNING
3	ERROR

代码实现示例如下：

```
# Global log level
global_level=3

# User
user=HwHiAiUser

# Share memory size of node 512k * 32 biggest support
maxNodeSize=524272

# Share memory count of queue
maxQueueCount=40

# log-agent-host #
logAgentMaxFileNum=8
# set host one log file max size, range is (0, 104857600]
logAgentMaxFileSize=10485760
# set host log dir
logAgentFileDir=/var/dlog
...
```

查看日志

- 日志路径：“/var/dlog”
- 日志命名格式如表8-8所示：

表 8-8 日志命名格式

日志位置	命名格式
Device侧	device-x_*.log（x代表芯片号）

日志位置	命名格式
Host侧	host-x_*.log（x代表芯片号）

- 日志压缩：日志文件默认是压缩的，无法直接查看，若需要直接查看，需要设置slog.conf文件或者联系产品支持技术人员申请获取专用解压工具进行解压再查看。用户需要在“/etc/slog.conf”文件中， 查看“zip_switch”的值是否为“0”，若不为“0”，将其值设置为“0”后重启slogd进程即可生效直接查看日志功能，slogd进程重启方法请参考8.5.4 日志重启。

  “/etc/slog.conf”文件内容：

  ```
  #zip switch, default : not zip
  #0 : not zip
  #1 : zip
  zip_switch = 0
  #zip level: 0 ~ 9, others err and exit, default:0
  #zip_level=0
  ```

8.5.4 日志重启

步骤1 输入如下命令查找slogd和IDE-daemon-host进程ID：

```
HwHiAiUser@test-2288H-V5:/var/dlog$ ps aux | grep slogd
HwHiAiU+    667  0.0  0.0 237652  4952 ?        Ssl  15:37   0:00 ./slogd
syslog     1732  0.0  0.0 256400  4320 ?        Ssl  Aug14   0:11 /usr/sbin/rsyslogd -n
HwHiAiU+  44956  0.0  0.0  21292   944 pts/14   S+   15:51   0:00 grep --color=auto slogd
HwHiAiUser@test-2288H-V5:/var/dlog$ ps -aux | grep IDE
HwHiAiUser       2885  0.1  0.0 1270180 18556 ?       Ssl  Aug21   4:20 ./IDE-daemon-host
HwHiAiUser      26567  0.0  0.0  21292   924 pts/8    S+   10:47   0:00 grep --color=auto IDE
```

步骤2 执行如下命令将用户切换为该进程的所属用户（HwHiAiUser）。

su HwHiAiUser

步骤3 执行如下命令结束slogd和IDE-daemon-host进程。

kill 667

kill 2885

步骤4 执行如下命令重新启动这两个进程，重新启动后即生效。

cd /usr/local/HiAI/driver/tools

./slogd

./IDE-daemon-host

----结束

8.6 业务软件编译

业务软件分为在Host侧和Device侧两部分软件，两部分软件分别在服务器和Ascend 310芯片上运行，因此，两部分软件依赖不同的编译工具和动态链接库，故需要分别编译。

- 推荐将Host侧编译为一个可执行文件，Device侧编译为一个动态连接库，用户可以根据需要，将业务划分为一个可执行文件（Host侧）和多个动态链接库（Host和Device侧）。

例如：将main编译为可执行文件，将每一个Engine编译成动态链接库。

- Device侧采用DDK工具包内提供的编译器、头文件以及动态链接库。
- Host侧采用开发环境自带的编译器（当前DDK指定开发环境的系统版本和架构），DDK内的头文件以及动态链接库。

- 业务软件的编译系统推荐采用CMake搭建。CMake的编译系统，请参考HelloDavinci的例程。

8.7 Demo 样例解析

8.7.1 离线视频推理（InferOfflineVideo）

本demo主要实现离线视频推理功能，针对输入的H264/H265数据，实现视频解码、图片格式转换、目标检测、抠图与缩放、属性检测、JPEG编码，主要涉及VDEC、VPC、JPEGE、模型推理等功能，业务流程如图8-5所示，demo的使用指导请参考功能根目录的Readme。

图 8-5 业务流程图

说明

输入的H264/H265数据可来源于摄像头或通过ffmpeg解封装的视频文件（MP4文件），本样例以H264文件为样例。

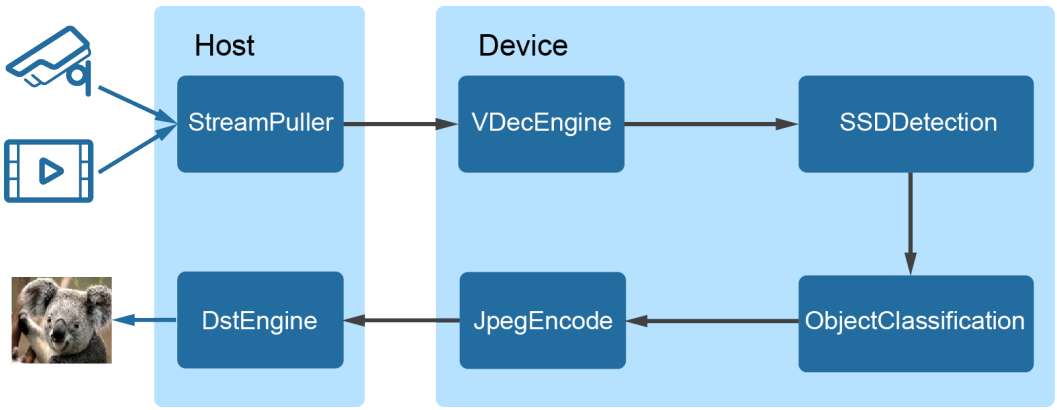
获取方式

离线视频推理（InferOfflineVideo）获取地址：**https://gitee.com/HuaweiAtlas/samples**

功能解析

本demo的Graph业务流由7个Engine组成，使用了目标检测（vgg_ssd）和属性识别（resnet50）两个模型。Engine间的输入与输出端口连接关系如图8-6所示。

图 8-6 处理流程



- StreamPuller

  StreamPuller的数据来源主要包括以下两个方面：

  a. H264/H265格式的离线视频

     通过文件系统函数打开视频文件，读取H264/H265数据到buff后发送给VDecEngine进行硬件解码。

  b. MP4、AVI等封装格式离线视频

     通过ffmpeg命令行或API代码将MP4、AVI等封装格式解封装为H264/H265格式。

     - 通过ffmpeg命令行：

       ffmpeg -i infile.mp4 -an -vcodec libx264 -crf 23 outfile.h264

       若通过命令行方式将MP4解封装为H264/H265文件，则后面解码流程与上面第1点处理流程相同。

     - 本样例代码还实现了通过ffmpeg API代码将MP4、AVI等封装格式解封装为H264/H265文件。

       主要涉及以下几个步骤，将读取出的H264/H265数据发送给vdec进行解码，ffmpeg的API接口调用流程如下：

       1) av_register_all
       2) avformat_open_input
       3) avformat_find_stream_info
       4) av_find_best_stream
       5) av_init_packet
       6) h264_mp4toannexbav_read_frame

- VDecEngine

  VDecEngine主要用于完成H264/H265视频流的解码，主要由初始化和周期解码两部分组成。

  – 初始化：资源申请和创建，例如CreateVdecApi，创建H26*解码句柄，同时由于VDEC输出的是HFBC的格式，创建VPC句柄，用于将HFBC转换成YUV。
  – 周期解码：周期性对发送过来的H264/H265进行解码，VDEC解码采用异步方式解码，调用VdecCtl传入原始数据，Matrix框架完成解码后，调用回调函数vdec_frame_return，将解码后的数据返回给用户，用户在vdec ctrl的入参中将回调函数注册给vdec模块。

    说明

    本样例在回调函数中实现HFBC格式到YUV格式的转换，并将转换后的数据发送给下一个Engine。

- SSDDetection

  实现SSD检测的推理和结果输出，包括分类ID，置信度和框位置。

- ObjectClassification

  将目标检测的结果进行进一步详细分类，输出包括分类ID和置信度。

- JpegEncode

  负责将解码过的YUV图片编码成JPEG图片。

- DstEngine

将Device侧的处理结果数据保存为文件和图片。

9 附录

9.1 Graph关键字
9.2 修订记录

9.1 Graph 关键字

参数		说明	必选（M）/可选（O）
graph_id	-	Graph ID，为正整数。	O
priority	-	优先级，无需调整。	O
device_id	-	设备ID。不同Graph可以运行在一个或多个芯片上，也可以运行在不同PCIe卡的芯片上，如果运行在不同芯片上，需要在graph的配置中增加device_id项，用于指定Graph运行的device id。如果不指定，默认运行在device id为0的芯片上。device id的id值从0开始，到N-1结束（N表示Device个数）。	O
engines	id	Engine ID。	M
	engine_name	Engine名称。	M
	side	Engine运行target，取值为“HOST”或“DEVICE”，需要注意根据业务需求，配置Engine运行在HOST或DEVICE。	M

参数		说明	必选（M）/可选（O）
说明 多路解码时推荐配置多个Engine，一个Engine对应一个线程，如果一个Engine对应多个线程，解码时无法保证顺序。	so_name	Engine运行时需要从Host侧拷贝动态库so文件名到Device侧。如果FrameworkerEngine运行时，需要依赖第三方库文件或自定义库文件时，依赖的so文件也需要配置在graph文件中（请将以下示例中的“xxx”替换为实际依赖库名。） so_name: "./libFrameworkerEngine.so" so_name: "./libxxx.so" so_name: "./libxxx.so"	O
	thread_num	线程数量。多路解码时，该参数值推荐设置为“1”，如果“thread_num”的值大于“1”，线程之间的解码将无法保证顺序。	M
	thread_priority	线程优先级。取值范围从“1”到“99”，用于设置engine对应的数据处理线程的优先级，采用SCHED_RR调度策略，再根据策略将对应Engine设置为较高优先级。	O
	queue_size	队列大小，默认为“200”。 用户需要根据业务负载的波动、Engine接收数据大小、系统内存进行合理配置。	O

	ai_config	配置示例：
		ai_config{
		items{
		name: "model_path"
		value: "./test_data/model/resnet18.om"
		}
		}
		● name值无需配置。
		● value值配置为模型文件所在的路径，包含文件名（名称中只允许包含数字、字母、下划线和小数点），可以将参数值配置为单个模型文件的路径（“./test_data/model/resnet18.om”）；也可以将模型文件打包成tar包后，将参数值配置为tar包所在的路径（“./test_data/model/resnet18.tar”）。若存在多个AIConfigItem，在配置tar包路径时，不允许在同一个目录下同时存在名称相同但格式不同的文件，例如“./test_data/model/test.zip”和“./test_data/model/test.tar”。
			O

参数		说明	必选（M）/可选（O）

	ai_model	配置示例：
		ai_model{
		name: "" //模型名称
		type："" //模型类型
		version："" //模型版本
		size："" //模型大小
		path："" //模型路径
		sub_path："" //辅助模型路径
		key："" //模型密钥
		sub_key："" //辅助模型密钥
		frequency："UNSET" //设备频率，取值UNSET或0表示取消设置；LOW或1表示低频；MEDIUM或2表示中频；HIGH或3表示高频。
		device_frameworktype："NPU" //模型运行设备类型，取值为NPU/IPU/MLU/CPU。
		framework："OFFLINE" //执行框架，取值为OFFLINE/CAFFE/TENSORFLOW。
		}
			O

	oam_config	配置dump算法数据的方式。当推理不准确时，可以查看某些层或全部层的算法结果。
		配置示例：
		oam_config{
		items{
		model_name: "" //相对路径+模型名称+后缀
		is_dump_all："" //是否全部dump，取值为"true"或"false"。
		layer: "" //层
		}
		dump_path："" //dump路径
		}
			O

	internal_so_name	内置在Device侧的动态库文件，用户可直接使用，无需从Host侧拷贝到Device侧。	O
	wait_inputdata_max_time	当前收到数据后等待下一个数据的最大超时时间。 ● 单位：毫秒。 ● 默认为“0”，表示数据不超时。	O
	is_repeat_timeout_flag	针对Engine未收到数据，是否进行循环超时处理（唤醒），与wait_inputdata_max_time配合使用，取值说明如下： ● 0：不重复，默认为0。 ● 1：重复。	O
	holdModelFileFlag	是否保留本Engine的模型文件，取值说明如下： ● 0：不保留。 ● 非0：保留。	O

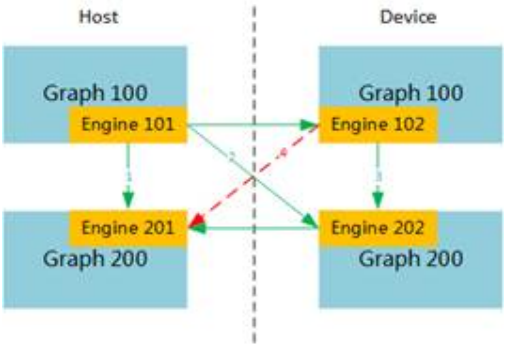
参数		说明	必选（M）/可选（O）
connects	src_engine_id	源Engine ID。	M

	src_port_id	源Engine的发送端口号。
		对于某个Engine，其端口号从“0”开始顺次增加。端口个数定义一般在相应的头文件中以HIAI_DEFINE_PROCESS宏来定义，代码实现如下：
		`#define SOURCE_ENGINE_INPUT_SIZE 1`
		`#define SOURCE_ENGINE_OUTPUT_SIZE 1`
		`class SrcEngine : public Engine {`
		`HIAI_DEFINE_PROCESS(SOURCE_ENGINE_INPUT_SIZE, SOURCE_ENGINE_OUTPUT_SIZE)`
		`}`
			M

	target_engine_id	目的Engine ID。	M
	target_port_id	目的Engine的接收端口号。	M
	target_graph_id	目的Graph ID。 Engine支持跨Graph串接，在该场景下，需要在connects的配置中增加“target_graph_id”项来表示接收端的Graph ID，如不配置，则默认在同一个graph中串接。 在有多个Ascend 310芯片情况下，可以使用跨Graph串接将各个模型运行在不同的Ascend 310芯片上，故尽量将相同的模型放在一个芯片上进行推理，减少不必要的内存消耗。 【场景说明】 • 如表格下图9-1所示，Engine在以下场景中支持跨Graph串接： 1. Graph A的Host Engine直接给Graph B的Host Engine发送数据。 2. Graph A的Host Engine直接给Graph B的Device Engine发送数据。 3. Graph A的Device Engine直接给Graph B的Device Engine发送数据。 • Engine在以下场景中不支持跨Graph串接：Graph A的Device Engine直接给Graph B的Host Engine发送数据，如表格下图9-1所示。	O

参数		说明	必选（M）/可选（O）
	receive_memory_without_dvpp	• 参数默认值为“0”，表示Deivce上运行的目标Engine的接收内存需要满足4G地址空间限制； • 若将参数值配置为“1”，表示Device上运行的目标Engine的接收内存不用满足4G地址空间限制。 • 用户可在Graph配置文件中将所有运行在Device上的目标Engine的参数“receive_memory_without_dvpp”（在connects属性下）设置为“1”，也可以将所有connects属性下的“receive_memory_without_dvpp”参数都配置为“1”，由于在Host上运行的目标Engine即使设置了该参数也无影响，故Matrix接收内存池无4G地址空间限制，从而提高系统内存使用率。 **说明** 当前DVPP中，VPC、JPEGE、JPEGD、PNGD功能的输入内存需要满足4G地址空间限制，VDEC、VENC功能的输入内存可以无4G地址空间限制。	O

图 9-1 Engine 跨 Graph 串接场景



9.2 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。

