寒武纪 EasyDK 用户手册

版本 *4.1.0*

2022 年 12 月 09 日

Cambricon 寒武纪

目录

插图目录

表格目录

1 版权声明

免责声明

中科寒武纪科技股份有限公司（下称“寒武纪”）不代表、担保（明示、暗示或法定的）或保证本文件所含信息，并明示放弃对可销售性、所有权、不侵犯知识产权或特定目的适用性做出任何和所有暗示担保，且寒武纪不承担因应用或使用任何产品或服务而产生的任何责任。寒武纪不应对因下列原因产生的任何违约、损害赔偿、成本或问题承担任何责任：（1）使用寒武纪产品的任何方式违背本指南；或（2）客户产品设计。

责任限制

在任何情况下，寒武纪都不对因使用或无法使用本指南而导致的任何损害（包括但不限于利润损失、业务中断和信息损失等损害）承担责任，即便寒武纪已被告知可能遭受该等损害。尽管客户可能因任何理由遭受任何损害，根据寒武纪的产品销售条款与条件，寒武纪为本指南所述产品对客户承担的总共和累计责任应受到限制。

信息准确性

本文件提供的信息归寒武纪所有，且寒武纪保留不经通知随时对本文件信息或对任何产品和服务做出任何更改的权利。本指南所含信息和本指南所引用寒武纪文档的所有其他信息均“按原样”提供。寒武纪不担保信息、文本、图案、链接或本指南内所含其他项目的准确性或完整性。寒武纪可不经通知随时对本指南或本指南所述产品做出更改，但不承诺更新本指南。

本指南列出的性能测试和等级要使用特定芯片或计算机系统或组件来测量。经该等测试，本指南所示结果反映了寒武纪产品的大概性能。系统硬件或软件设计或配置的任何不同会影响实际性能。如上所述，寒武纪不代表、担保或保证本指南所述产品将适用于任何特定用途。寒武纪不代表或担保测试每种产品的所有参数。客户全权承担确保产品适合并适用于客户计划的应用以及对应用程序进行必要测试的责任，以期避免应用程序或产品的默认情况。

客户产品设计的脆弱性会影响寒武纪产品的质量和可靠性并导致超出本指南范围的额外或不同的情况和/或要求。

知识产权通知

寒武纪和寒武纪的标志是中科寒武纪科技股份有限公司在中国和其他国家的商标和/或注册商标。其他公司和产品名称应为与其关联的各自公司的商标。

式复制、重制、修改、出版、上传、发布、传输或分发本指南。除了客户使用本指南信息和产品的权利，根据本指南，寒武纪不授予其他任何明示或暗示的权利或许可。未免疑义，寒武纪不根据任何专利、版权、商标、商业秘密或任何其他寒武纪的知识产权或所有权对客户授予任何（明示或暗示的）权利或许可。

- 版权声明
- © 2022 中科寒武纪科技股份有限公司保留一切权利。

2 前言

2.1 版本记录

表 2.1: 版本记录

文档名称	寒武纪 EasyDK 用户手册
版本号	V4.1.0
作者	Cambricon
修改日期	2022.12.8

2.2 更新历史

- v4.1.0

  更新时间：2022 年 12 月 8 日

  更新内容：

  - 增加多路示例 easy_pipeline，位于 `samples/easy_pipeline` 目录下。
  - 移动 Detection 和 Classification 示例到 `samples/simple_demo` 目录下。
  - 修改 MLU370、MLU590 和 MLU585 平台上，输出 buffer 数目。对于视频解码器改为 min_output_buf_num + 1，对于 JPEG 解码器改为 2。
  - 增加接口 `BufSurfaceWrapper::GetNumFilled` 、`BufSurfaceWrapper::GetMemType`、`infer_server::RemovePreprocHandler` 和 `infer_server::RemovePostprocHandler`。
  - 支持 InferServer Python 接口。
  - 修改 `IPreproc::OnPreproc` 接口参数 `src` 和 `dst` 数据类型 `CnedkBufSurface*` 为 `cnedk::BufSurfWrapperPtr` 。
  - 修改 InferServer 后处理方法设置方式。详情请查看*后处理*。
  - 修改 `infer_server::NetworkInputFormat` 枚举值为 `RGB`、`BGR`、`RGBA`、`BGRA`、`ARGB`、`ABGR`、`GRAY`、`TENSOR` 和 `INVLAID`。

- v4.0.0

更新时间：2022 年 10 月 17 日

更新内容：

- 支持云端 MLU370、MLU590 和 MLU585 平台。
- 支持边缘端 CE3226 平台。
- 不再兼容 MLU200 系列（MLU270/MLU220.M.2/MLU220.Edge/MLU220.SOM）平台。
- 移除 v3.1.0 所有接口和 InferServer Contrib 所有接口，其他 InferServer 接口除外。
- 硬件解码不再支持复用解码器输出 buffer。
- 硬件编码不再支持获取编码器输入 buffer。
- 不再支持追踪功能。
- InferServer 增加 `InferSessionVideo` 类。
- InferServer 自定义前后处理函数部分有较大改动。以 batch 为单位处理数据，数据将被保存在 `CnedkBufSurface` 中。
- 支持硬件解码、编码、Transform、OSD、Vin、Vout 功能（其中 OSD，Vin 和 Vout 仅在边缘端支持）。
- 硬件编码支持将解码后的帧 resize 到其他分辨率。
- 硬件编码支持将待解码的帧 resize 到其他分辨率再进行编码。
- 新增一系列接口：
  * `CnedkBufPoolCreate`
  * `CnedkBufPoolDestroy`
  * `CnedkBufSurfaceCreateFromPool`
  * `CnedkBufSurfaceCreate`
  * `CnedkBufSurfaceDestroy`
  * `CnedkBufSurfaceSyncForCpu`
  * `CnedkBufSurfaceSyncForDevice`
  * `CnedkBufSurfaceCopy`
  * `CnedkBufSurfaceMemSet`
  * `CnedkVdecCreate`
  * `CnedkVdecDestroy`
  * `CnedkVdecSendStream`
  * `CnedkVencCreate`
  * `CnedkVencDestroy`
  * `CnedkVencSendFrame`
  * `CnedkDrawRect`
  * `CnedkFillRect`
  * `CnedkDrawBitmap`
  * `CnedkPlatformInit`
  * `CnedkPlatformUninit`
  * `CnedkPlatformGetInfo`
  * `CnedkTransformSetSessionParams`
  * `CnedkTransformGetSessionParams`

    * CnedkTransform
    * CnedkVinCaptureCreate
    * CnedkVinCaptureDestroy
    * CnedkVinCapture
    * CnedkVoutRender
  - 增加类 cnedk::BufSurfaceWrapper 、 IBufDeleter 、 BufPool 。
- v3.1.0

  更新时间：2022 年 8 月 26 日

  更新内容：
  - 使用 Glog，统一 log 格式。
  - 适配 MagicMind v0.8.2 版本。
  - InferServer 支持输入可变的模型（LoadModel 时设置输入 shape，设置后输入 shape 不可改变）。
  - InferServer Python 接口提供 get_device_core_version 接口。
  - 修复 InferServer CNCV（CambriconComputer Vision Library，寒武纪计算机视觉库）二级网络预处理 keep aspect ratio 时存在的问题。
  - 修复应用在使用 EasyDecode 时可能碰到的时序问题。
  - 修复 EasyEncode MLU200 编码参数 bit_rate 的注释描述。其单位为 bps 不是 kbps。
  - 修复 MLU370 平台编码时序问题。
  - 修复对输出进行 TransOrder 的问题。
  - 修复 Classification 示例的问题。
  - InferServer samples 新增输入多路视频 demo，支持 OPENCV、CNCV 两种预处理。
  - InferServer Python API samples，支持在 MLU370 平台上运行。samples 会自动选择平台，无需传递参数-p。另外增加-dev 参数可以选择设备。
- v3.0.0

  更新时间：2022 年 1 月 5 日

  更新内容：
  - 合并 InferServer 到 EasyDK。
  - 增加 InferServer 的 Python 接口，支持同步和异步推理。
  - 支持 InferServer 处理输出形状可变的模型。
  - 增加 InferServer::Preprocessor 预处理器，在预处理函数中处理 batch 数据。
  - 修改 InferServer::PreprocessorHost 的 ProcessFunction 类型。
  - 修改 InferServer::Postprocessor 的 ProcessFunction 类型。
  - 增加 WITH_BACKWARD 编译选项。
  - 兼容 CNRT 4.x 和 5.x 版本。
  - 优化 EasyDecode，支持 MLU300 系列解码。
  - 优化 EasyEncode，支持 MLU300 系列编码。
  - 支持 EasyEncode ，从编码器获取 MLU 内存，减少内存拷贝。
  - 修改部分 EasyEncode::Attr 参数。
  - 增加三种解码方式示例，支持使用 EasyDecode、FFmpeg 和 FFmpeg-MLU 三种解码方式。

  - 替换示例中的推理部分，使用 `InferServer` 进行推理。
  - 增加以下接口:
    * `EasyEncode::FeedData`
    * `EasyEncode::FeedEos`
    * `EasyEncode::RequestFrame`
  - 废弃接口 `EasyEncode::SendDataCPU`，使用 `EasyEncode::FeedData` 和 `EasyEncode::FeedEos`。
  - 废弃 `EasyEncode::FeedData` 接口的 `integral_frame` 参数。

- v2.6.0

  更新时间：2021 年 10 月 11 日

  更新内容：

  - 移除 `easy_plugin` 模块。
  - 优化 `FeatureMatchTrack` 性能。
  - 移除日志颜色。
  - 修复 JPEG 编码未处理错误帧。
  - `EasyInfer` 构造函数不再需要绑定设备。
  - 修正 `MluMemoryOp` 日志中错误的内存大小。
  - 移除以下废弃接口：
    * `EasyDecode::SendData`
    * `EasyInfer::WithRGB0Output`
    * `EasyInfer::WithYUVInput`

- v2.5.0

  更新时间：2021 年 3 月 23 日

  更新内容：

  - 参考 Google log（glog）实现日志系统，替换 glog。
  - 增加 `TimeMark` 用于测量硬件时间。
  - 设置 `cnrtSetDeviceFlag`，使能驱动代理同步队列，降低 CPU 占用。
  - 移除 `EasyCodec` 模块以下繁琐参数，简化 Codec 模块使用：
    * `EasyDecode::Attr::buf_strategy`
    * `EasyEncode::Attr::crop_config`
    * `EasyEncode::Attr::ir_count`
    * `EasyEncode::Attr::max_mb_per_slice`
    * `EasyEncode::Attr::cabac_init_idc`

- v2.4.0

  更新时间：2020 年 12 月 31 日

  更新内容：

  - 移除 v2.2.0 废弃的接口。
  - 丰富示例代码，支持 rtsp 拉流解码，保存推理结果至本地视频文件等功能。
  - [*Early Access*] 增加推理服务模块，详见*推理服务*。
  - 提升追踪模块 `FeatureMatchTrace` 精确度。

  - `EasyInfer` 增加异步推理接口。
  - 修复 `MluResizeConvertOp` 未凑齐 batch 时运行算子发生段错误的问题（该问题由 v2.3.0 引入）。
- v2.3.0

  更新时间：2020 年 11 月 30 日

  更新内容：
  - 支持交叉编译 MLU220EDGE 平台。
  - 使用 ShapeEx 代替 Shape，支持任意维度数据。
  - 废弃接口 `ModelLoader::InputShapes` 和 `ModelLoader::OutputShapes` 。
  - 增加接口 `ModelLoader::InputShape(uint32_t index)` 和 `ModelLoader::OutputShape(uint32_t index)` 。
  - 优化 EasyCodec 实现。
- v2.2.0

  更新时间：2020 年 11 月 5 日

  更新内容：
  - 废弃以下接口（将于 v2.4.0 版本删除）：
    * `MluContext::(ConfigureForThisThread, ChannelId, SetChannelId)`
    * `MluResize::InvokeOp`
    * `EasyDecode::Create, EasyEncode::Create`
    * `EasyInfer::(Init, Loader, BatchSize)`
    * `ModelLoader::InitLayout`
    * `MluMemoryOp::(SetLoader, Loader, AllocCpuInput, AllocCpuOutput, AllocMluInput, AllocMluOutput, AllocMlu, FreeArrayMlu, MemcpyInputH2D, MemcpyOutputD2H, MemcpyH2D, MemcpyD2H)`
  - 替换示例代码中使用的上述废弃接口。
  - EasyDecode 支持解码 Progressive-JPEG。
  - 增加查询设备数量的接口 `edk::MluContext::GetDeviceNum()`。
  - 修复 MluResizeConvertOp 部分规模不支持的问题。
  - [*Early Access*] 使用统一的异常类型。
  - 增加 Resize（yuv2yuv）的算子。
  - 重构 MluTaskQueue 为 C++ 风格。
- v2.1.0

  更新时间：2020 年 10 月 26 日

  更新内容：
  - 初始版本。

3 介绍

EasyDK（Cambricon Easy Development Kit）提供了一套面向寒武纪硬件设备的接口，用于快速开发和部署人工智能应用。

EasyDK 支持如下特性:

- BufSurface：描述及管理 buffer。
- Platform：初始化和去初始化平台，获取平台相关信息。
- Decode：视频与图片的硬件解码及解码后缩放。
- Encode：视频与图片的硬件编码及编码前缩放。
- Transform：转换图片。
- OSD：绘制框及位图。
- Vin：捕捉摄像头输入。
- Vout：渲染图片。

下图为 EasyDK 的模块结构：

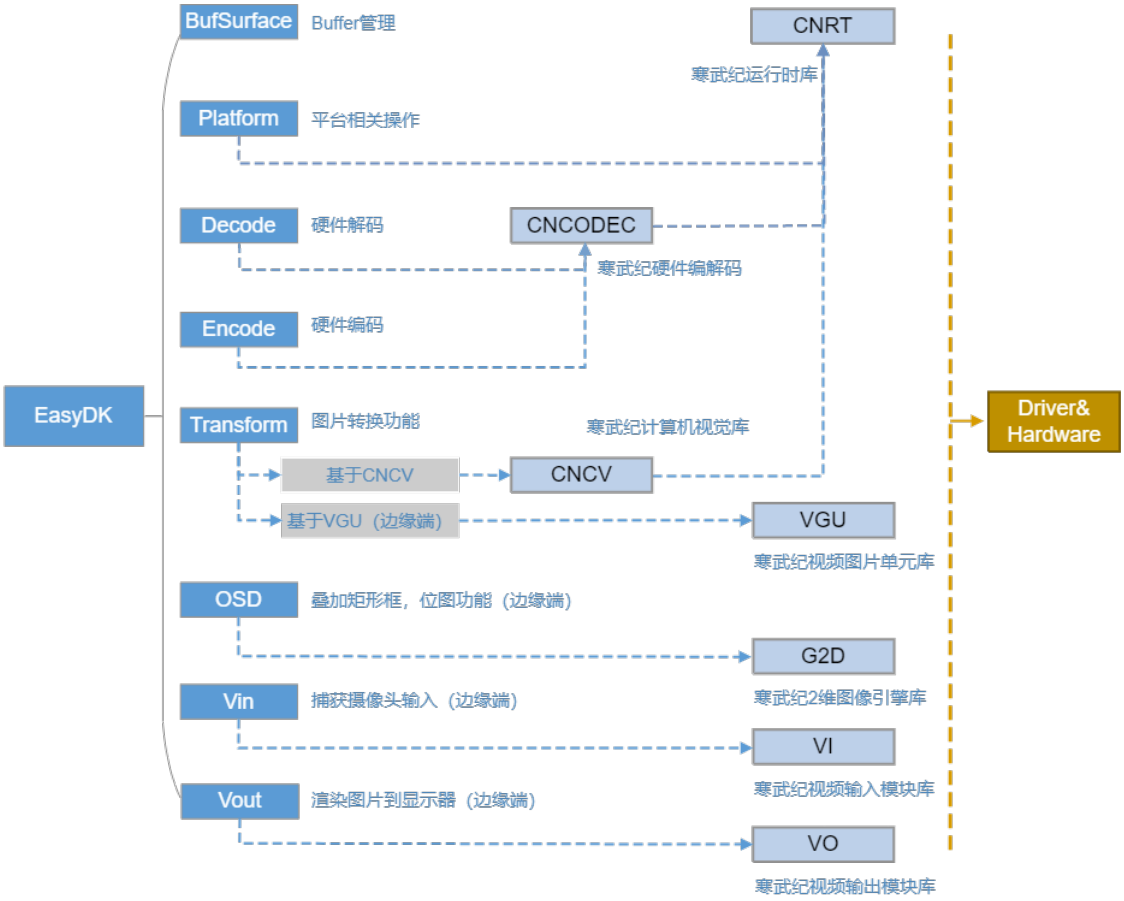

图 3.1: EasyDK 结构

EasyDK 还包含推理服务组件：提供了一套面向寒武纪硬件设备的类似服务器的推理接口（C++11 标准），以及模型加载与管理，推理任务调度等功能，极大地简化了面向寒武纪硬件平台高性能人工智能应用的开发和部署工作。

推理服务在寒武纪软件栈中的位置，如下图所示：

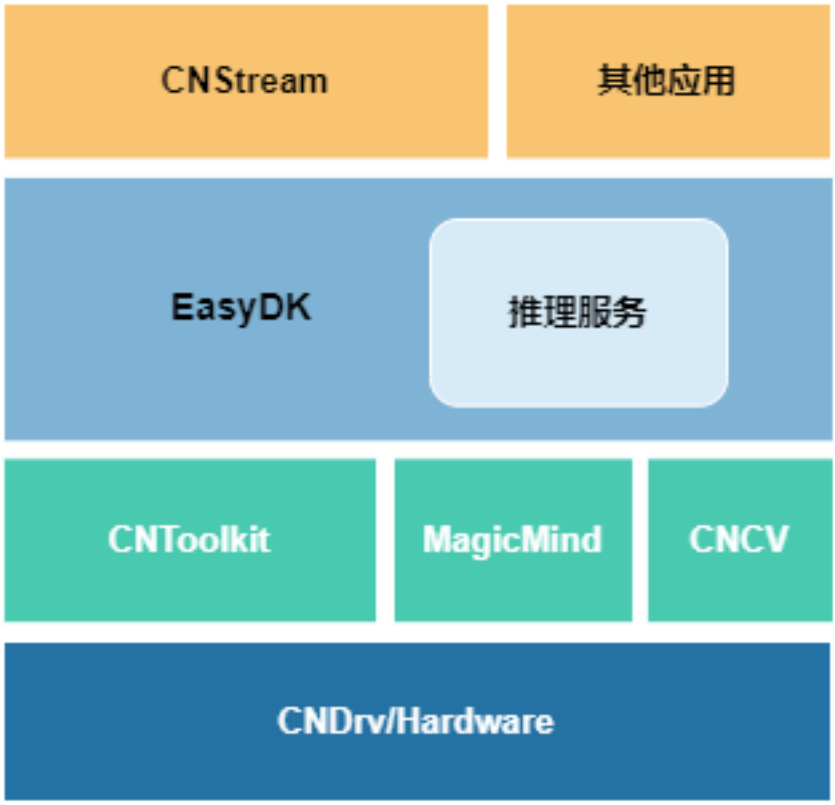

图 3.2: 寒武纪软件栈

推理服务共包含以下 3 个模块的用户接口:

- Model: 模型加载与管理。
- Processor: 可自定义的后端处理单元。
- InferServer: 执行推理任务。

4 快速入门

4.1 构建和运行环境要求

构建和运行 EasyDK 有如下依赖：

- CMake 3.5+
- GCC 5.4+
- GLog 0.3.4
- libcurl-dev (可选)

测试程序和示例有以下额外依赖：

- OpenCV 2.4.9+
- GFlags 2.1.2
- FFmpeg 2.8 3.4 4.2

4.1.1 寒武纪安装包

4.1.1.1 Cloud

EasyDK 支持云端 MLU590、MLU585 和 MLU370 平台，依赖于寒武纪 CNToolkit 安装包中 CNDev、CNDrv、CNRT 和 CNCodec-V3 库，以及 CNCV、MagicMind、Cambricon CNNL、CNNL_Extra 和 CNLight 库。

用户在使用 EasyDK 之前需要安装 CNToolkit、CNCV 以及 MagicMind。发送邮件到 service@cambricon.com，联系寒武纪工程师获得相关的软件包和安装指南。

MLU590 和 MLU585 具体版本依赖如下:

表 4.1: MLU590 和 MLU585 版本依赖

EasyDK	CNToolkit	CNCV	Driver	MagicMind	Cambricon CNNL	CNNL_ EXTRA	CNLight
v4.1.0	v3.1.1	v1.1.0	v5.3.6	v0.14.0	v1.13.1	v0.19.1	v0.16.1
v4.0.0	v3.1.1	v1.1.0	v5.3.6	v0.14.0	v1.13.1	v0.19.1	v0.16.1

MLU370 具体版本依赖如下:

表 4.2: MLU370 版本依赖

EasyDK	CNToolkit	CNCV	Driver	MagicMind	Cambricon CNNL	CNNL_ EXTRA	CNLight
v4.1.0	v3.0.2	v1.0.0	v4.20.9	v0.13.0	v1.11.1	v0.17.0	v0.15.0
v4.1.0	v3.0.2	v1.0.0	v4.20.9	v0.13.0	v1.11.1	v0.17.0	v0.15.0

注解:

CNToolkit 与 Driver，MagicMind 与 Cambricon CNNL、CNNL_EXTRA、CNLight 之间的更详细的版本依赖关系请查看《寒武纪 CNToolkit 安装升级使用手册》和《寒武纪 MagicMind 用户手册》。

4.1.1.2 Edge

EasyDK 支持边缘端 CE3226 平台。

在 CE3226 平台，EasyDK 依赖 `ce3226v100-sdk-x.y.z.tar.gz` 软件包中的 inference 和 mps 部分，该软件包的安装和使用请参阅《寒武纪 CE3226V100&101 SDK 使用开发指南》。

用户在使用 EasyDK 之前需要安装 ce3226v100-sdk。发送邮件到 service@cambricon.com，联系寒武纪工程师获得相关的软件包和安装指南。

CE3226 具体版本依赖如下:

EasyDK	ce3226v100-sdk
v4.1.0	v1.1.0
v4.0.0	v1.0.0

4.2 源码编译 EasyDK

EasyDK 仅支持源码编译的方式使用，按如下步骤编译 EasyDK（*${EasyDK_DIR}*，代表 EasyDK 源码目录）：

1. 创建编译文件夹存储编译结果。

```
cd ${EasyDK_DIR}
mkdir build
```

2. 运行 CMake 配置编译选项，并生成编译指令。

   寒武纪 EasyDK 提供了一个 CMakeLists.txt 描述编译流程，用户可以从 CMake 官网[1] 免费下载和使用 CMake。

表 4.3: EasyDK 编译选项

cmake 选项	范围	默认值	描述
PLATFORM	MLU590/ MLU370/ CE3226	MLU370	选择平台
BUILD_PYTHON_API	ON / OFF	OFF	编译 InferServer Python 接口
BUILD_SAMPLES	ON / OFF	ON	编译 sample
BUILD_TESTS	ON / OFF	ON	编译 tests
CNIS_WITH_CURL	ON / OFF	ON	使能 CURL，支持推理服务从网络下载模型
CNIS_RECORD_PERF	ON / OFF	ON	使能性能数据记录
SANITIZE_MEMORY	ON / OFF	OFF	检查内存
SANITIZE_ADDRESS	ON / OFF	OFF	检查地址
SANITIZE_THREAD	ON / OFF	OFF	检查多线程
SANITIZE_UNDEFINED	ON / OFF	OFF	检查未定义行为

   示例:

```
cd build
cmake ${EasyDK_DIR} -D{cmake option}={ON/OFF}
```

[1] http://www.cmake.org/

注解：

1. PLATFORM 编译选项的值不是 ON/OFF，用户需要根据目标平台选择 PLATFORM，例如 -DPLATFORM=CE3226。
2. 当目标平台是 MLU585 时选择 -DPLATFORM=MLU590。

1. 运行编译指令。

```
make
```

2. 编译后的库文件存放在 ${EasyDK_DIR}/lib ；头文件存放在 ${EasyDK_DIR}/include 。
3. 如果想要交叉编译 EasyDK，则需要事先交叉编译并安装第三方依赖库，并配置 CMAKE_TOOLCHAIN_FILE 文件，以 CE3226 为例：

```
export MPS_HOME=/your/path/to/mps
export PATH=$PATH:/your/path/to/cross-compiler/bin
cmake ${EasyDK_DIR} -DCMAKE_FIND_ROOT_PATH=/your/path/to/3rdparty-libraries-install-path -DCM
↪AKE_TOOLCHAIN_FILE=${EasyDK_DIR}/cmake/cross-compile.cmake  -DCNIS_WITH_CURL=OFF -DPLATFORM
↪=CE3226
```

更多信息可以参考交叉编译 。

4.3 EasyDK 开发示例

人工智能应用开发示例为用户提供了离线模型、视频文件、运行脚本以及开发示例代码，帮助用户快速了解如何使用 EasyDK 完成简单的人工智能应用部署。用户可以直接通过脚本运行示例代码，无需修改任何设置。

4.3.1 开发示例目录结构

人工智能应用和转码开发示例存放于 *${EasyDK_DIR}/samples/* 文件夹下，支持在 MLU590、MLU585、MLU370 和 CE3226 平台下运行。开发示例包含的文件如下:

- easy_pipeline: 多路异步推理示例。
  - framework: 核心框架。
  - decode：解码模块。
  - encode：编码模块。
  - inference：同步推理模块和异步推理模块。
  - OSD：绘图模块。
  - common：公共代码。
- simple_demo: 单路推理示例。
  - common：OSD、视频解析等示例通用功能。

  - classification：分类示例。
  - detection：检测示例。
- CMakelists.txt：CMake 文件，编译示例时使用，用户无需做任何设置和改动。
- data：运行示例使用的视频文件。

4.3.2 编译和运行开发示例

1. 编译样例。编译 EasyDK 时打开 *BUILD_SAMPLES* 选项即可编译示例代码。
2. 运行人工智能应用样例。运行如下命令:

```
pushd samples/simple_demo/classification
# ./run_resnet50.sh  [mlu590/mlu370/ce3226]   支持 MLU590、MLU585、MLU370 和 CE3226 平台
./run_resnet50.sh mlu370
popd
```

```
pushd samples/simple_demo/detection
# ./run_yolov3.sh  [mlu590/mlu370/ce3226]   支持 MLU590、MLU585、MLU370 和 CE3226 平台
./run_yolov3.sh mlu370
popd
```

```
pushd samples/easy_pipeline
# ./run_yolov3.sh  [mlu590/mlu370/ce3226]   支持 MLU590、MLU585、MLU370 和 CE3226 平台
./run_yolov3.sh mlu370
# ./run_resnet50.sh  [mlu590/mlu370/ce3226]   支持 MLU590、MLU585、MLU370 和 CE3226 平台
./run_resnet50.sh mlu370
popd
```

运行结束后程序会自动退出。

注解:

当目标平台是 MLU585 时，在执行脚本时传入平台参数 `mlu590` 。

5 模块

本章详细介绍 EasyDK 的各个模块。

5.1 BufSurface

BufSurface 提供了描述及管理 batched buffer 的功能。buffer 信息保存在 CnedkBufSurface 结构体内，包括内存类型，batched buffer 指针（指针指向每个 buffer 的描述，包括大小、颜色空间、内存指针、plane 信息）等。

另外 EasyDK 还提供了一些接口，用于创建、销毁及拷贝 BufSurface、内存填充以及同步边缘端内存，创建和销毁 BufSurface 内存池以及从内存池创建一份 BufSurface。

在云端平台上，可申请的内存类型有 CNEDK_BUF_MEM_DEVICE 和 CNEDK_BUF_MEM_SYSTEM 。

在边缘端平台上，可申请的内存类型有 CNEDK_BUF_MEM_VB 、CNEDK_BUF_MEM_VB_CACHED 、CNEDK_BUF_MEM_UNIFIED 、CNEDK_BUF_MEM_UNIFIED_CACHED 和 CNEDK_BUF_MEM_SYSTEM。

```
// 绑定设备。
int dev_id = 0;
cnrtSetDevice(dev_id);

// 创建 BufSurface。
CnedkBufSurface *surf = nullptr;

CnedkBufSurfaceCreateParams params;
params.device_id = dev_id;
params.batch_size = 1;
params.color_format = CNEDK_BUF_COLOR_FORMAT_NV12;
params.width = width;
params.height = height;
// MLUxxx device 内存。
params.mem_type = CNEDK_BUF_MEM_DEVICE;
CnedkBufSurfaceCreate(&surf, &params);

// 拷贝 BufSurface 到 BufSurface (CPU 内存)。
```



```
CnedkBufSurface *dst_surf = nullptr;
// System 内存 (CPU)。
params.mem_type = CNEDK_BUF_MEM_SYSTEM;
CnedkBufSurfaceCreate(&dst_surf, &params);
// 拷贝 BufSurface 中的数据
CnedkBufSurfaceCopy(surf, dst_surf);

// 销毁 BufSurface。
CnedkBufSurfaceDestroy(surf);
CnedkBufSurfaceDestroy(dst_surf);

// 创建 BufSurface 内存池。
void* surf_pool;
// CE3226 hardware VB 内存。
params.mem_type = CNEDK_BUF_MEM_VB_CACHED;
uint32_t pool_size = 16;
CnedkBufPoolCreate(&surf_pool, params);
CnedkBufSurface* surf_from_pool;
CnedkBufSurfaceCreateFromPool(&surf_from_pool, surf_pool);

// memset 填充内存。
uint8_t filled_val = 128;
CnedkBufSurfaceMemSet(surf_from_pool, -1, -1, filled_val);
// 同步内存到 CPU。
CnedkBufSurfaceSyncForCpu(surf_from_pool, -1, -1);
// 访问 CPU 内存。
memset(surf_from_pool->surface_list[0].mapped_data_ptr, 0,
       surf_from_pool->surface_list[0].data_size);
// 同步内存到 Device。
CnedkBufSurfaceSyncForDevice(surf_from_pool, -1, -1);

// 销毁内存池。
CnedkBufPoolDestroy(surf_pool);
```

为了使 BufSurface 的使用更加便利，提供类 `cnedk::BufSurfaceWrapper` 用于管理 BufSurface。另外支持创建 BufSurfaceWrapper 内存池。

```
// 绑定设备。
int dev_id = 0;
cnrtSetDevice(dev_id);

// 创建 BufSurface。
```



```
CnedkBufSurface *surf = nullptr;

CnedkBufSurfaceCreateParams params;
params.device_id = dev_id;
params.batch_size = 1;
params.color_format = CNEDK_BUF_COLOR_FORMAT_NV12;
params.width = width;
params.height = height;
// MLUxxx device 内存。
params.mem_type = CNEDK_BUF_MEM_DEVICE;
CnedkBufSurfaceCreate(&surf, &params);

// 创建一个 BufSurfaceWrapper，无需手动释放 surf，BufSurfaceWrapper 析构函数中自动调用。
surf_wrapper = std::make_shared<cnedk::BufSurfaceWrapper>(surf);

// 获取宽、高
surf_wrapper->GetWidth();
surf_wrapper->GetHeight();
// 获取 plane 数目
surf_wrapper->GetPlaneNum();
// 获取 plane0 的 stride
uint32_t plane_idx = 0;
surf_wrapper->GetStride(plane_idx);
// 获取 plane0 的字节数
surf_wrapper->GetPlaneBytes(plane_idx);
// 获取 batch 中第一份数据的 plane0 的数据。
uint32_t batch_idx = 0;
surf_wrapper->GetData(plane_idx, batch_idx);

// 设置及获取 pts
surf_wrapper->SetPts(1000);
surf_wrapper->GetPts();

// 获取 CPU 侧数据指针
void* cpu_ptr = GetHostData(plane_idx, batch_idx);
uint32_t data_size = surf->surface_list[batch_idx].data_size;
memset(cpu_ptr, 0 , data_size);
// 同步 CPU 数据到 Device。当 plane_idx，batch_idx 设置为-1 时，代表同步全部数据。
SyncHostToDevice(plane_idx, batch_idx);


// 创建 BufSurfaceWrapper 内存池
```



```
cnedk::BufPool pool;
uint32_t pool_size = 16;
pool.CreatePool(params, pool_size);

// 从内存池中获取一个 BufSurfaceWrapper。如获取超时则返回 nullptr。
int timeout_ms = 10;
pool.GetBufSurfaceWrapper(timeout_ms);

// 销毁 BufSurfaceWrapper 内存池。
int destroy_timeout_ms = 5000;
pool.DestroyPool(destroy_timeout_ms);
```

5.2 Platform

本模块提供了初始化平台、去初始化平台、获取平台信息功能。

注意：

在边缘端平台上，必须先对平台进行初始化，包括 VB、SYS 等模块的初始化。在云端平台上，无需对平台进行初始化。

```
CnedkPlatformConfig config;
// 设置编解码的起始 id。
config.codec_id_start = 0;
// 如使用 vout，设置其参数。
config.vout_params.max_input_width = 1920;
config.vout_params.max_input_height = 1080;
// 如使用 vin，设置其参数。
config.sensor_num = 1;
config.sensor_params[0].sensor_type = 6;
config.sensor_params[0].mipi_dev = 1;
config.sensor_params[0].bus_id = 0;
config.sensor_params[0].sns_clk_id = 1;
config.sensor_params[0].out_width = 1920;
config.sensor_params[0].out_height = 1080;

// 初始化平台
CnedkPlatformInit(&config);

// 反初始化平台
CnedkPlatformUninit();
```



```
// 获取平台信息
CnedkPlatformInfo info;
int dev_id = 0;
CnedkPlatformGetInfo(dev_id, &info);
// 获取平台名称
info.name;
```

5.3 Decode

Decode 封装了硬件解码及缩放功能。

对于 MLU590 和 MLU585 平台，封装了 CNCodec 中解码的功能，支持视频和图片解码，格式如下:

- HEVC（H.265）
- H.264
- JPEG

视频支持的最大分辨率为 8192 x 4320，JPEG 支持的最大分辨率为 16384 x 16384。视频和 JPEG 解码支持输出 NV12、NV21 格式的数据。

对于 MLU370 平台，封装了 CNCodec 中解码的功能，支持视频和图片解码，格式如下:

- HEVC（H.265）
- H.264
- JPEG

视频支持的最大分辨率为 8192 x 4320，JPEG 支持的最大分辨率为 16384 x 16384。视频和 JPEG 解码支持输出 NV12、NV21 格式的数据。

对于 CE3226 平台，封装了 CNCodec 中解码的功能，支持视频和图片解码，格式如下:

- HEVC（H.265）
- H.264
- JPEG

视频支持的最大分辨率为 8192 x 8192，JPEG 支持的最大分辨率为 16384 x 16384。视频解码仅支持输出 NV21 格式的数据。JPEG 解码支持输出 NV12、NV21 格式的数据。

支持对解码后的数据进行缩放，云端平台基于 CNCV，边缘端平台基于硬件 VGU。

解码可配置选项:

- 输出数据格式，具体支持格式见上述。
- 解码器设备号，可指定对应设备解码。

注解：

解码器仅支持输入完整帧数据进行解码，建议使用 *FFmpeg* 进行解封装和解析后再送入解码器。

解码流程如下:

1. 设计解码数据回调函数 `OnFrame` ，获取 BufSurface 回调函数 `GetBufSurf` ，EOS 回调函数 `OnEos` 和 ERROR 回调函数 `OnError` 。
2. 调用 `CnedkVdecCreate` 按 params 参数创建一个解码器实例。
3. 调用 `CnedkVdecSendStream` 方法发送数据给解码器解码。
4. 通过设置的 `GetBufSurf` 回调函数获取一个 BufSurface。
5. 随后通过设置的解码数据回调函数 `OnFrame` 获取填入解码后的数据的 BufSurface，数据使用完毕后调用 `CnedkBufSurfaceDestroy` 释放 BufSurface。
6. 在最后一帧发送给解码器后，调用 `CnedkVdecSendStream` 送空数据（stream->bits 设置为 nullptr），告知解码器解码结束。
7. 解码最后一帧结束后，解码器通过设置的 EOS 回调函数 `OnEos` 通知应用程序。
8. 解码出错时，通过设置的 ERROR 回调函数 `OnError` 获取错误码。
9. 解码结束后，调用 `CnedkVdecDestroy` 释放解码器资源。

解码示例程序见 `samples/simple_demo/common/video_decoder.cpp` 。

5.4 Encode

Encode 封装了硬件编码功能以及输入预处理功能，支持对输入图片进行缩放（云端平台还支持颜色空间转换）。

对于 MLU590 和 MLU585 平台，封装了 CNCodec 中编码的功能，支持图片编码，格式如下:

- JPEG

JPEG 编码器支持输入 NV12、NV21 格式的数据。

对于 MLU370 平台，封装了 CNCodec 中编码的功能，支持视频和图片编码，格式如下:

- HEVC（H.265）
- H.264
- JPEG

视频支持的最大分辨率为 8192 x 8192，JPEG 支持的最大分辨率为 16384 x 16384。视频编码器（H.264，H.265）支持输入 NV12、NV21 格式的数据。JPEG 编码器支持输入 NV12、NV21 格式的数据。

对于 CE3226 平台，封装了 CNCodec 中编码的功能，支持视频和图片编码，格式如下:

- HEVC（H.265）
- H.264
- JPEG

视频支持的最大分辨率为 8192 x 8192，JPEG 支持的最大分辨率为 16384 x 16384。视频和 JPEG 编码器支持输入 NV12、NV21、GRAY 格式的数据（编码器根据每帧数据输入的格式进行编码）。

注意：

在边缘端平台上，输入内存类型必须是 `CNEDK_BUF_MEM_VB` 或 `CNEDK_BUF_MEM_VB_CACHED` 。

编码可配置选项:

- 输入数据格式，具体支持格式见上述。
- 编码器设备号，可指定对应设备编码。
- 视频编码支持配置帧率、GOP 大小和码率。

编码流程如下:

1. 设计编码数据回调函数 `OnFrameBits` ，EOS（End Of Stream）回调函数 `OnEos` 和 ERROR 回调函数 `OnError` 。
2. 调用 `CnedkVencCreate` 按 params 参数创建一个编码器实例。
3. 调用 `CnedkVencSendFrame` 方法发送数据给编码器编码。
4. 通过设置的编码数据回调函数 *OnFrameBits* 获取编码后的数据，数据使用完毕后数据自动释放。
5. 在最后一帧发送给解码器后，调用 `CnedkVencSendFrame` 送空数据（surf 设置为 nullptr），告知编码器编码结束。
6. 编码最后一帧结束后，编码器通过设置的 EOS 回调函数 *OnEos* 通知应用程序。
7. 编码出错时，通过设置的 ERROR 回调函数 `OnError` 获取错误码。
8. 编码结束后，调用 `CnedkVencDestroy` 释放编码器资源。

编码示例程序见 `easydk/samples/easy_pipeline/encode/encode_handler_mlu.cpp` 。

5.5 Transform

Transform 封装了硬件图片转换功能。提供颜色空间转化，截取有效区域，减均值除方差等功能。

对于云端平台，支持基于 CNCV 的图片转换加速，包括：

1. YUV420spToRGBx、缩放（可选）、CROP（可选）以及减均值除方差（可选）。
2. RGBxToYUV420sp、缩放（可选）以及减均值除方差（可选）。
3. Yuv420spResize。
4. 减均值除方差。

对于边缘端平台，支持基于硬件 VGU 的图片转换加速，包括：

1. YUV420spToRGB、缩放以及 ROI。

另外，边缘端平台还支持基于 CNCV 的图片转换加速，包括：

1. YUV420spToRGBx、缩放（可选）、CROP（可选）以及减均值除方差（可选）。
2. RGBxToYUV420sp、缩放（可选）以及 CROP（可选）。

3. Yuv420spResize。
4. 减均值除方差。

注解：

YUV420sp 指 YUV420sp NV12 和 NV21。RGBx 指 RGB 家族，包括 RGB、BGR、ARGB、ABGR、BGRA 和 RGBA。

注意：

在边缘端平台上使用 VGU 硬件做图片转换时，输入输出的内存类型必须是 `CNEDK_BUF_MEM_VB` 或 `CNEDK_BUF_MEM_VB_CACHED` 。

5.6 OSD

OSD 封装了硬件图片叠加矩形框，填充矩形及位图功能（仅边缘端支持）。

仅支持 YUV420sp NV12 图片格式。

注意：

数据内存类型必须是 `CNEDK_BUF_MEM_VB` 或 `CNEDK_BUF_MEM_VB_CACHED` 。

5.7 Vin

Vin 封装了硬件捕捉摄像头传感器输入的功能（仅边缘端支持）。

捕捉流程如下:

1. 设计编码数据回调函数 `OnFrame` ，获取 BufSurface 回调函数 `GetBufSurf` 和 ERROR 回调函数 `OnError` 。
2. 调用 `CnedkVinCaptureCreate` 按 params 参数创建一个捕捉器实例。
3. 调用 `CnedkVinCapture` 方法捕捉传感器数据。
4. 通过设置的 `GetBufSurf` 回调函数获取一个 BufSurface。
5. 通过设置的捕捉数据回调函数 `OnFrame` 获取已经填入捕捉数据的 BufSurface。
6. 捕捉出错时，通过设置的 ERROR 回调函数 `OnError` 获取错误码。
7. 捕捉结束后，调用 `CnedkVinCaptureDestroy` 释放资源。

注意：

必须先调用 `CnedkPlatformInit` 接口，初始化平台。

5.8 Vout

Vout 封装了硬件渲染图片到显示器的功能（仅边缘端支持）。

支持 YUV420sp 图片格式。

注意：

1. 必须先调用 `CnedkPlatformInit` 接口，初始化平台。
2. 数据内存类型必须是 `CNEDK_BUF_MEM_VB` 或 `CNEDK_BUF_MEM_VB_CACHED` 。

6 推理服务

6.1 基本概念

本节描述推理服务中所涉及的具体概念。

6.1.1 InferServer

其整体架构如下图所示：

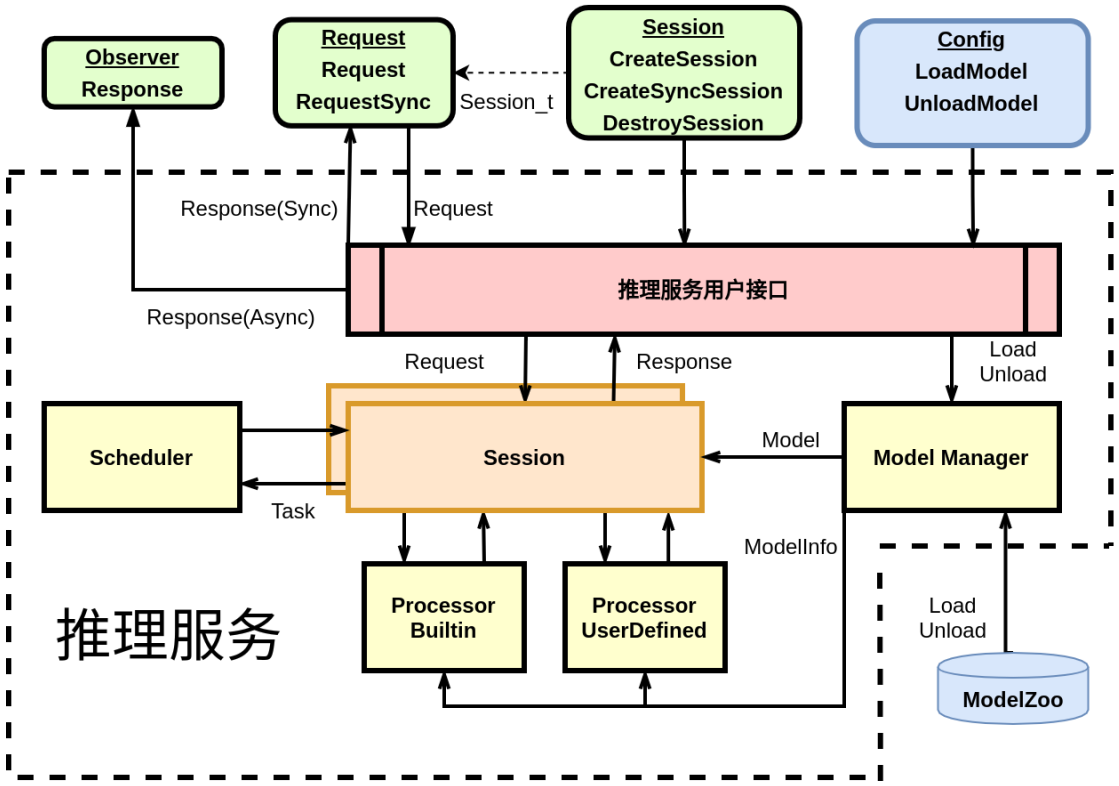

图 6.1: 推理服务架构

`InferServer` 是推理服务暴露给用户的功能入口，用户通过此入口进行加载或卸载模型（Model）、创建

推理节点（Session）、请求推理任务（Request）等操作。推理任务划分为预处理，推理，后处理三个环节，分别交由不同的后端处理单元（Processor）完成。每个推理服务实例维护一个线程池（Scheduler），以处理环节（Task）作为最小调度单元，调度执行在该推理服务实例中运行的所有推理任务。

InferServer 使用 pimpl 指针隔离接口与实现，内部保证每个设备上仅有一个 pimpl 实例，同一设备号下创建的 InferServer 链接同一个 pimpl 指针，提供对应功能。

```
InferServer s_0(0);
InferServer s_1(1);

// another_s_0 和 s_0 共用同一个任务调度器和相同的推理资源
InferServer another_s_0(0);
```

使用 InferServer 异步接口进行推理的步骤如下所示：

1. 加载离线模型 InferServer::LoadModel()。
2. 创建异步推理节点 InferServer::CreateSession()。
3. 准备输入数据。
4. 提交推理请求 InferServer::Request()，推理任务完成后将结果通过 Observer::Response 发送给用户。
5. 完成所有推理任务后，释放推理节点 InferServer::DestroySession()。

```
class MyObserver : public Observer {
  void Response(Status status, PackagePtr output, any user_data) { ... }
};

class MyPreprocHandler : public IPreproc {
 public:
  ~MyPreprocHandler() { }
  int OnTensorParams(const CnPreprocTensorParams *params) override { ... return 0; }
  int OnPreproc(cnedk::BufSurfWrapperPtr src, cnedk::BufSurfWrapperPtr dst,
                const std::vector<CnedkTransformRect> &src_rects) override { ... retu
↪rn 0; }
}

class MyPostprocHandler : public IPostproc {
 public:
  ~MyPostprocHandler() { }
  int OnPostproc(const std::vector<infer_server::InferData*>& data_vec,
                 const infer_server::ModelIO& model_output,
                 const infer_server::ModelInfo* model_info) override { ... return 0;␣
↪}
}
```



```
// prepare resources
InferServer server(0);

SessionDesc desc;
desc.name = "sample infer";
desc.model = InferServer::LoadModel(model_path);
desc.model_input_format = NetworkInputFormat::RGB;

// create processors
desc.preproc = Preprocessor::Create();
auto preproc_handler = new MyPreprocHandler();
SetPreprocHandler(desc.model->GetKey(), preproc_handler);

desc.postproc = Postprocessor::Create();
auto postproc_handler = new MyPostprocHandler();
SetPostprocHandler(desc.model->GetKey(), postproc_handler);

Session_t session = server.CreateSession(desc, std::make_shared<MyObserver>());

// run inference
// create an input package with tag "stream_0", containing two piece of InferData
auto input = Package::Create(2, "stream_0");
// Create two BufSurface
CnedkBufSurface *surf_1, *surf_2;
CnedkBufSurfaceCreateParams params;
params.device_id = dev_id;
params.batch_size = 1;
params.color_format = CNEDK_BUF_COLOR_FORMAT_NV12;
params.width = width;
params.height = height;
// MLUxxx device 内存。
params.mem_type = CNEDK_BUF_MEM_DEVICE;
CnedkBufSurfaceCreate(&surf_1, &params)
CnedkBufSurfaceCreate(&surf_2, &params)

PreprocInput input_1, input_2;
input_1.surf = std::make_shared<cnedk::BufSurfaceWrapper>(surf_1);
input_1.has_bbox = false;
input->data[0].Set(std::move(input_1));

input_2.surf = std::make_shared<cnedk::BufSurfaceWrapper>(surf_2);
```



```
input_2.has_bbox = false;
input->data[1].Set(std::move(input_2));

server.Request(session, input, nullptr);
// result will be passed to MyObserver::Response after finishing process

// wait until the "stream_0" tagged inference task done
server.WaitTaskDone(session, "stream_0");

// release resources
RemovePreprocHandler(desc.model->GetKey());
RemovePostprocHandler(desc.model->GetKey());
server.DestroySession(session);
delete preproc_handler;
```

6.1.2 ModelInfo

ModelInfo 提供了易用的用户侧模型管理和信息读取接口，是各种模型实现的基类。由 InferServer::LoadModel() 加载模型，得到模型基类的智能指针。当所有实例生命周期结束后，模型自动卸载。用户可随时获取模型的各种信息，包含输入输出个数、输入输出数据形状以及 batch_size 等。

6.1.3 Session

Session 是推理任务节点的抽象，接收用户的推理请求并完成响应。一个 Session 为一个处理节点，内部的后端处理单元的顺序结构是固定的，处理同一类请求。

使用 InferServer::CreateSession 创建异步 Session，异步 Session 只能使用异步 Request 接口，处理完毕后通过 Observer::Response 发送响应给用户；使用 InferServer::CreateSyncSession 创建同步 Session，同步 Session 只能使用同步 RequestSync 接口，响应作为 RequestSync 的输出参数返回。

```
InferServer s(0);
SessionDesc desc;
/*
 * set desc params
 * ...
 */
Session_t async_session = s.CreateSession(desc, std::make_shared<MyObserver>());
Session_t sync_session = s.CreateSyncSession(desc);
```



```
s.Request(async_session, input, user_data);
s.RequestSync(sync_session, input, &status, output);
```

Session 根据传入的参数准备 SessionDesc::engine_num 份推理资源，使每份推理资源可以独立执行任务，达成并行推理。

注解：

模型键值拼接预处理和后处理单元类型名称（modelkey_preproc_postproc）作为 Session 的键值，键值相同的 Session 共用同一簇推理资源。

6.1.4 Processor

后端处理单元，负责处理推理任务中的一个环节，由多个 Processor 链接起来构成整个推理任务处理流程。

BaseObject 基类为 Processor 提供设置任意参数的功能。

```
MyProcessor p;
p.SetParams("some int param", 1,
            "some string param", "some string");
int a = p.GetParam<int>("some int param");
const char* b = p.GetParam<const char*>("some string");
```

6.1.5 InferData

InferData 表示一份推理数据，基于 C++11 编译器实现的 any 类使任意类型的推理数据都可以在 InferData 中设置。

```
InferData data;
cv::Mat opencv_image;
// 填充数据到 InferData
data.Set(opencv_image);
// 获取一份数据的复制
auto image = data.Get<cv::Mat>();
// 获取一份数据的引用
auto& image_ref = data.GetLref<cv::Mat>();
try {
  // 类型不匹配，抛出异常 bad_any_cast!
  auto non_sense = data.Get<int>();
} catch (bad_any_cast&) {
  std::cout << "Data stored in any cannot convert to given type!";
```



```
}

PreprocInput preproc_input;
// 重新设置 data 后 image_ref 非法化
data.Set(preproc_input);
// cv::imwrite("foo.jpg", image_ref);  // segment fault
auto input = data.Get<PreprocInput>();
auto& input_ref = data.GetLref<PreprocInput>();
```

6.1.6 Package

Package 是一组推理数据的集合，既是 Request 的输入，也是 Response 的输出。用户通过 Package 可以一次请求多份数据进行处理。

使能 CNIS_RECORD_PERF 编译选项时，输出中 Package::perf 包含每一个处理环节的性能数据，未使能时为空表。

6.1.7 Observer

进行异步请求需要在创建 Session 时，设置 Observer 实例。Session 完成推理任务后，以通知 Observer 的方式完成 Response。

```
class MyObserver : public Observer {
  void Response(Status status, PackagePtr output, any user_data) {
    std::cout << "Get one response\n";
  }
};

InferServer s(0);
SessionDesc desc;
Session_t async_session = s.CreateSession(desc, std::make_shared<MyObserver>());
```

6.2 功能

本节详细介绍推理服务的功能。

6.2.1 模型加载与管理

推理服务提供模型加载和管理功能，并在全局保有唯一一份模型缓存。

推理服务使用 MagicMind 后端进行推理。

使用 `InferServer::LoadModel(const std::string& model_uri, const std::vector<Shape>& in_shapes = {})` model_uri 代表模型模型 URI，当模型输入 shape 可变时，通过 in_shapes 设置模型输入形状。

若使能 `CNIS_WITH_CURL` 编译选项，上述接口可以从远端下载模型，并加载至模型存储路径（由 `InferServer::SetModelDir` 设置，默认值为当前目录）。当检测到模型存储路径中已存在同名模型，则跳过下载，直接加载本地模型（请注意不要使用相同的模型名，可能会导致使用错误的模型）。URI 形如 `http://some.web.site/foo.graph` 或 `../../bar.data`。

使用 `InferServer::LoadModel(void* ptr, size_t size, const std::vector<Shape>& in_shapes = {})` 从内存中加载模型。

将两个模型相关字符串进行拼接作为键值，对模型进行区分（从内存加载的模型路径是内存地址字符串）。若加载模型时发现缓存中已存在该模型，则直接返回缓存模型。模型缓存存在上限，超出上限自动卸载未在使用的模型。上限默认值是 10，可通过环境变量 `CNIS_MODEL_CACHE_LIMIT` 更改默认值。支持运行时清除模型缓存，从缓存中清除不会直接卸载模型，仅在无其他模型的智能指针实例时才会卸载模型（确保模型已经没有在使用，避免功能性错误）。

```
ModelPtr local_model = InferServer::LoadModel("../../resnet50.magicmind");
ModelPtr net_model = InferServer::LoadModel("http://foo.com/resnet50.magicmind");
// input shape is mutable, set input shape: 4, 416, 416, 3
ModelPtr mutable_input_model = InferServer::LoadModel("../../resnet50.magicmind", {infer_server
↪::Shape({4, 416, 416, 3})});

void *model_mem, *decoded_model_mem;
size_t file_len = ReadFromFile(model_mem, ...);
size_t model_size = DecodeModel(decoded_model_mem, model_mem, file_len, ...);
ModelPtr mem_model = InferServer::LoadModel(decoded_model_mem, model_size);
// input shape is mutable. Assume the model has two inputs.
// Set input0 shape: 4, 416, 416, 3 and input2 shape: 4, 256, 256, 3
ModelPtr mutable_input_mem_model =
    InferServer::LoadModel(decoded_model_mem, model_size,
                           {infer_server::Shape({4, 416, 416, 3}), infer_server::Shape({4, 256,
↪ 256, 3})});
```


6.2.2 推理任务调度

推理服务对所有请求的推理任务进行调度，以在保证通用性的前提下尽量达到最优性能。推理服务使用三种共同作用的调度方式：批处理、优先级和并行处理。

6.2.2.1 批处理（Batch）

推理服务提供两种批处理模式，Dynamic 模式（`BatchStrategy::DYNAMIC`）和 Static 模式（`BatchStrategy::STATIC`），在创建 Session 时通过 `SessionDesc::strategy` 指定。

- Dynamic 模式会跨 Request 拼凑批数据，尽可能使用性能最优的批大小进行处理，达到较为理想的吞吐。但由于跨 Request 拼凑数据会存在等待数据集齐的时间，单次 Request 的时延较高。达到设置的 timeout 时间后，即使未集齐批也会进行处理，以避免时延过高。
- Static 模式以用户每次输入的 Request 数据为一批，不跨 Request 拼凑数据，可以达到较为理想的单次 Request 时延。但相较于 Dynamic 模式更难达到性能较优的批大小，总吞吐量较 Dynamic 模式略低。

注解：

目前底层尚未支持带状态的离线模型，待后端支持后，会增量支持 Sequence 模式的批处理策略。

6.2.2.2 优先级（Priority）

每个设备上的所有推理任务共用同一个调度器。用户可以在创建 Session 时通过 `SessionDesc::priority` 设置优先级，高优先级的 Session 中的任务将会优先被处理。优先级限制为 0~9 的整数，数值越大优先级越高，低于 0 的按 0 处理，高于 9 的按 9 处理。

6.2.2.3 并行处理（Parallel）

为达到最大性能，通常需要在一个设备上并行执行多组推理任务。若某个 Session 代表的一类推理任务负载较重，可以通过设置 `SessionDesc::engine_num` 增大该类推理任务的并行度，使该 Session 共占用 engine_num * model_core_number 个计算核，和对应份推理资源（内存，模型指令等）。超出上限后，继续增加 engine_num，可能出现由于资源竞争导致的总吞吐下降的情况。

6.2.3 后端处理单元

推理服务内置三种后端处理单元（Processor）。

6.2.3.1 预处理

预处理单元完成输入数据预处理的功能，推理服务内置了通用的预处理单元 Preprocessor 。

预处理单元

用户需要继承 IPreproc 基类，重写 OnTensorParams 和 OnPreproc 接口，通过 SetPreprocHandler 设置给模型。OnTensorParams 仅会被回调一次，用户可以通过该函数获取模型相关信息，包括模型形状，数据类型以及输入颜色空间。OnPreproc 回调函数的输入 src 储存了一个 batch 的数据，src_rects 描述每份数据的有效区域，当 src_rects.size() 为 0 时，代表当前 batch 中的数据的有效区域为全图。处理完毕后，用户需要将结果填入 dst 中。最后通过 RemovePreprocHandler 移除 handler。

```
class MyPreprocHandler : public IPreproc {
 public:
  ~MyPreprocHandler() { }
  int OnTensorParams(const CnPreprocTensorParams *params) override { ... return 0; }
  int OnPreproc(cnedk::BufSurfWrapperPtr src, cnedk::BufSurfWrapperPtr dst,
                const std::vector<CnedkTransformRect> &src_rects) override { ... return 0; }
}

InferServer s(0);
SessionDesc desc;
desc.preproc = Preprocessor::Create();
preproc_handler = new MyPreprocHandler();
SetPreprocHandler(desc.model->GetKey(), preproc_handler);
desc.model_input_format = NetworkInputFormat::RGB;
/*
 * set desc params
 * ...
 */
Session_t sync_session = s.CreateSyncSession(desc);
...
RemovePreprocHandler(desc.model->GetKey());
delete preproc_handler;
```


6.2.3.2 推理

推理单元完成推理任务（暂不支持用户设置推理单元，所有 Session 默认使用内置的推理单元）。推理单元接受固定类型的输入，输出固定类型的推理结果（`ModelIO`），输入输出数据均在 MLU 内存上。推理完成后，将推理结果数据转至后处理单元解析。

每个推理单元实例从模型获取一份包含指令和参数数据的 Context。当 engine_num 大于 1 时，同一个模型存在多个推理单元实例，多份 Context 共享部分数据，以避免指令的多份拷贝。

6.2.3.3 后处理

后处理单元完成推理结果解析，推理服务内置了通用的后处理单元 Postprocessor。用户需要继承 `IPostproc` 基类，重写 `OnPostproc` 接口，通过 `SetPostprocHandler` 设置给模型。`OnPostproc` 回调函数的 `output` 参数储存了一个 batch 的推理结果，即后处理的输入，`info` 描述了模型信息，处理完毕后，用户需要将结果填入 `data_vec` 中。最后通过 `RemovePostprocHandler` 移除 handler。

```
class MyPostprocHandler : public IPostproc {
 public:
  ~MyPostprocHandler() { }
  int OnPostproc(const std::vector<InferData*> &data_vec, const ModelIO& model_output,
                 const ModelInfo* model_info) override { ... return 0; }
};

InferServer s(0);
SessionDesc desc;
desc.postproc = Postprocessor::Create();
postproc_handler = new MyPostprocHandler();
SetPostprocHandler(desc.model->GetKey(), postproc_handler);
/*
 * set desc params
 * ...
 */
Session_t sync_session = s.CreateSyncSession(desc);
...
RemovePostprocHandler(desc.model->GetKey());
delete postproc_handler;
```

若用户未设置后处理方法，则仅执行拷贝操作，后处理单元输出 CPU 上的 `ModelIO` 数据。此种情况下，用户无需设置 `IPostproc`。

很多模型需要额外的数据才能够完成后处理，不同的模型需要的数据不同。`InferData` 中可以设置任意类型的补充数据（`user_data`）供后处理使用。用户在发起 Request 前，通过 `InferData::SetUserData` 设置额外数据：


```
struct Meta {
  ObjTracker* tracker;
};

// create package containing 4 piece of infer data
auto input = Package::Create(4);
for (auto& item : input->data) {
  // Create BufSurface set its wrapper to PreprocInput object and to InferData as inference
↪input
  CnedkBufSurface *surf;
  CnedkBufSurfaceCreateParams params;
  params.device_id = dev_id;
  params.batch_size = 1;
  params.color_format = CNEDK_BUF_COLOR_FORMAT_NV12;
  params.width = width;
  params.height = height;
  // MLUxxx device 内存。
  params.mem_type = CNEDK_BUF_MEM_DEVICE;
  CnedkBufSurfaceCreate(&surf, &params)

  PreprocInput input;
  input.surf = std::make_shared<cnedk::BufSurfaceWrapper>(surf);
  input.has_bbox = false;
  item->Set(std::move(input));

  // set user data for postprocessor
  Meta meta;
  meta.tracker = GetTracker(...);
  item->SetUserData(meta);
}
server->Request(session, input, nullptr);
```

用户在后处理方法中通过 `InferData::GetUserData` 获取该数据：

```
class MyPostprocHandler : public IPostproc {
 public:
  ~MyPostprocHandler() { }
  // for a batch infer data
  int OnPostproc(const std::vector<InferData*> &data_vec, const ModelIO& model_output,
                 const ModelInfo* model_info) override {
    // network output idx 0
    cnedk::BufSurfWrapperPtr output = model_output.surfs[0];
```



```
    // For Edge
    CnedkBufSurfaceSyncForCpu(output->GetBufSurface(), -1, -1);

    for (size_t batch_idx = 0; batch_idx < data_vec.size(); batch_idx++) {
      Meta meta = data_vec[batch_idx]->GetUserData<Meta>();
      // 根据额外参数对模型输出后处理
      auto postproc_out = FooPostproc(output->GetHostData(0, batch_idx));
      // 用户侧额外需要的特定逻辑，如给推理目标打上标签，更新追踪器的状态等
      meta.tracker->Update(postproc_out);
      // 设置用户需要在 response 获取到的输出
      data_vec[batch_idx]->Set(postproc_out);
    }
    return 0;
  }
};

InferServer s(0);
// 创建 session 前，设置 handler 给后处理单元
SessionDesc desc;
desc.postproc = Postprocessor::Create();
postproc_handler = new MyPostprocHandler();
SetPostprocHandler(desc.model->GetKey(), postproc_handler);
/*
 * set desc params
 * ...
 */
Session_t sync_session = s.CreateSyncSession(desc);
...
delete postproc_handler;
```

6.3 编程指南

6.3.1 同步或异步请求

请求任务（Request）有同步接口和异步接口两种选择。

```
bool Request(Session_t session, PackagePtr input, any user_data, int timeout = -1) noexcept;
bool RequestSync(Session_t session, PackagePtr input,
                 Status* status, PackagePtr output, int timeout = -1) noexcept;
```

异步请求中保留了 user_data 字段，支持用户设置任意数据，推理任务中不会对该字段进行处理，任务

完成后将 Response 返回给用户，完成透传。

为避免数据处理能力不一致导致内存堆积，推理服务对处理任务缓存进行了限制，缓存超出一定数量则进入繁忙状态，Request 和 RequestSync 接口会被阻塞，直至推理服务空闲后再将请求的数据推入队列中。因此同步和异步接口都可以选择设置 timeout，同步和异步接口在等待超时后直接退出，不进行请求。请求后，同步接口阻塞直至得到 Response，若等待过程总时长超过 timeout 设置的值，则将该份 Request 数据标记为 discarded，直接退出。

当 Session 不属于该 InferServer，或等待入队列超时，未发送请求，则接口返回 false，其他情况返回 true。若同步接口发送请求后，等待处理完成超时，则接口返回 true，status 值为 `Status::TIMEOUT`。

`Package::data` 是一个 vector，其中的每一个元素代表一份数据。每次请求可以包含一份数据，或者多份数据（`Package::data.size()`），但请求-响应对与数据份数无关，始终是一份响应对应一份请求。换言之，对异步接口调用时，`Observer::Response` 被调用次数与请求次数一致，且数据一一对应。

异步请求收到响应的顺序与发送请求的顺序保持一致（如请求顺序为 req_0、req_1、req_2, 则响应顺序一定是 resp_0、resp_1、resp_2），以避免有序数据出现紊乱。

6.3.2 等待处理结束或遗弃任务

在某些数据处理正常结束时，如流数据到达 EOS，如果需等待相同类型数据全部完成后再退出处理或销毁 Session，需在请求时输入 Package 设置 tag（`std::string`），调用 `InferServer::WaitTaskDone(session, tag)`，这样就会在函数中等待数据处理完成，直到最后一个持有该 tag 的请求处理完毕后退出函数，若无持有该 tag 的请求则立即返回。

如需快速停止某些数据处理，遗弃未处理数据，需在请求时输入 Package 设置 tag（`std::string`），在需要快速停止时调用 `InferServer::DiscardTask(session, tag)`，该函数不阻塞，仅将所有持有该 tag 的请求标记为 discarded，然后返回。标记为 discarded 的数据，若在等待处理的缓存中，则直接抛弃，若已在处理，则等待处理完毕后废弃。所有被标记为 discarded 的数据都不会返回给用户。

6.3.3 自定义数据结构

用户必须设置 `PreprocInput` 保存到 `InferData` 中，交给推理服务进行推理，但在后处理函数中可自定义任意数据结构保存到 `InferData` 中，如下所示：

```
struct FooStruct {
  ...
};

InferServer s(0);

class MyPostprocHandler : public IPostproc {
 public:
```



```
  ~MyPostprocHandler() { }
  int OnPostproc(const std::vector<InferData*> &data_vec,
                 const ModelIO& model_output,
                 const ModelInfo* model_info) override {
    ...

    for (size_t batch_idx = 0; batch_idx < data_vec.size(); batch_idx++) {
      // 根据额外参数对模型输出后处理
      FooStruct postproc_out = FooPostproc(...);
      // 设置用户需要在 response 获取到的输出
      data_vec[batch_idx]->Set(postproc_out);
    }
    return 0;
  }
};

// 创建 session 前，设置 handler 给后处理单元
SessionDesc desc;
desc.postproc = Postprocessor::Create();
postproc_handler = new MyPostprocHandler();
SetPostprocHandler(desc.model->GetKey(), postproc_handler);
/*
 * set desc params
 * ...
 */
Session_t session = s.CreateSession(desc, std::make_shared<MyObserver>());
...
delete postproc_handler;
```

6.3.4 自定义后端处理单元

用户可自定义后端处理单元，以满足定制化需求。自定义的后端处理单元 MyProcessor 需继承自 ProcessorForkable<MyProcessor> （CRTP）以获得 Create 和 Fork 的能力，并重写 Init 和 Process 两个纯虚函数。Session 调用虚函数 Fork 复制 Processor 实例，用来并行处理推理任务。参数拷贝自原有 Processor，并调用 Init 来初始化。自定义后端处理单元对纯虚函数 Init 的重写需要实现解析参数并初始化推理资源，对纯虚函数 Process 的重写需要实现任务数据的处理。

```
class MyProcessor : public ProcessorForkable<MyProcessor> {
 public:
  MyProcessor() noexcept : ProcessorForkable<MyProcessor>("MyProcessor") { ... }
  Status Init() noexcept override { ... }
```



```
  Status Process() noexcept override { ... }
}
```

自定义 Processor 的实现应满足：

- 传递给 ProcessorForkable 基类的构造函数一个唯一的类型名称（用于日志打印，区分 Executor，性能数据表键值）。
- Processor::Init 函数解析所有参数，并准备好推理资源。
- Processor::Process 函数在 Init 后可直接调用，实现无需线程安全，交由调用方（内部友元 TaskNode ）通过私有方法 Processor::Lock 锁住处理资源。

6.3.5 特殊用法

如果无需进行后处理，可以不设置后处理的 process_func 函数，PostProcessor 内部会对推理结果进行拷贝，如果为异步推理用户可通过 Response 回调函数的参数 PackagePtr data 获取，如果为同步推理可通过 RequestSync 接口的输出参数 PackagePtr response 获取。

6.3.6 性能调优方法

仅在打开编译选项 CNIS_RECORD_PERF 时进行性能统计，未使能时 InferServer::GetPerformance 接口返回空表。使能 CNIS_RECORD_PERF 后创建 Session 时设置 SessionDesc::show_perf 为 true，则每隔 2 秒自动打印 Session 的性能数据。

1. 每个 Processor 时延计时。
2. 锁等待计时。
3. 批处理等待时间计时，每份批处理数据数量记录。
4. 单次 Request 时延计时，整体吞吐计数。

通过 std::map<string, LatencyStatistic> GetPerformance(Session_t session) 获取时延性能信息，其中 LatencyStatistic 如下：

```
struct LatencyStatistic {
  // 由于每个批处理中数据量可能不一致，以单份数据为单元计数
  uint32_t unit_cnt;  // 数据单元总数
  uint64_t total_time;  // 总时延
  uint32_t max_time;  // 单元最大时延
  uint32_t min_time;  // 单元最小时延
};
```

通过 ThroughoutStatistic GetThroughout(Session_t session) 获取吞吐性能信息，其中 ThroughoutStatistic 如下：


```
struct ThroughoutStatistic {
  uint32_t request_cnt{0};  // 请求总数
  uint32_t unit_cnt{0};  // 数据单元总数
  float rps{0};  // 每秒请求数
  float ups{0};  // 每秒数据单元数
  float rps_rt{0};  // 实时每秒请求数
  float ups_rt{0};  // 实时每秒数据单元数
};
```

影响性能的参数有如下三个：

1. `BatchStrategy`：DYNAMIC 模式总吞吐较高，单个 Request 时延较长；Static 模式单个 Request 时延较短，总吞吐较低。
2. `SessionDesc::engine_num`：engine_num == card core number / model core number 时可达到最大吞吐，继续增大 engine_num 吞吐基本无提升，或可能下降。engine_num 越大占用的 MLU 内存越多，并行处理能力越强。
3. `SessionDesc::batch_timeout`：仅在 Dynamic 模式下生效，一般应调整至大多数情况都能凑齐一组批处理的时长，然后根据要求的吞吐和时延上下微调。

根据数据调优性能的建议如下：

- 若要降低单帧数据总时延，可以选择使用 Static 策略，或者使用 batchsize 更小的离线模型。
- 若要提高总吞吐量，可以选择使用 Dynamic 策略，或者使用 batchsize 更大的模型。
- 批处理等待时间过长可能表示 timeout 设置时间过长，模型的 batchsize 过大。
- 批处理中数据达不到模型的 batchsize 值，可能表示 timeout 设置时间过短，或模型的 batchsize 过大。
- 单个 Processor 处理时间过长表示可能遇到性能瓶颈，资源占用达到上限（算力、带宽、CPU），或者处理并行度较低，在 engine_num * model_core_number 小于 MLU 单个设备核数的时候，可以适当调大 engine_num，以加快处理速度。
- MagicMind 后端模型默认占用所有 MLU 设备核，engine_num 选 1 时延较低，选 2 吞吐更高，设置更大的 engine_num 没有性能收益。
- 使用 cnmon 获取到 MLU 占用率较低的情况，可能是任务不饱和，确认 engine_num 足够大。若使用的是 CPU 侧的预处理，可以考虑更换成 MLU 上的预处理以加快处理速度，提高 MLU 占用率。
- 为保证 Response 顺序与 Request 顺序一致，`Observer::Response` 对单个 Session 的 Request 在时序上是严格串行执行的，若 `Response` 函数耗时 10ms，则相当于限制该 Session 处理速率上限是 100 个请求/秒。降低 `Response` 耗时或将推理任务分给多个不同的 Session 能够有效地降低该限制，提升性能（使用相同模型的多个 Session 使用同一套推理资源，不会有额外的资源开销）。

6.4 Python 封装

InferServer 基于 pybind11 封装了 Python 接口，帮助用户通过 Python 编程语言快速开发实际业务。目前该套接口仅支持 Python 3 以上版本调用。支持 MLU370 平台。

6.4.1 使用说明

编译 CNStream，制作 Python 包并安装。

```
cd {EASYDK_DIR}/python
python setup.py install
```

${EASYDK_DIR} 代表 EasyDK 源码目录。当前 setup.py 脚本还不支持编译选项配置，采用项目中 CMakeLists.txt 默认配置。该脚本包含了编译、打包、安装等全部步骤，如果用户只需要编译 EasyDK Python 库，执行以下命令在 {EASYDK_DIR}/python/lib 目录下获得相应的动态库。

```
cd {EASYDK_DIR}/build
# For mlu370
cmake .. -DBUILD_PYTHON_API=ON -DPLATFORM=MLU370
make
```

卸载 InferServer Python 包：

```
pip uninstall cnis -y
```

注解：

如果此前通过 setup.py 安装过 InferServer Python 包，需要将其卸载后再进行编译。否则默认使用安装的 cnis 包。

编译及执行 InferServer Python 接口单元测试：

通过执行以下命令，编译单元测试，

```
cd {EASYDK_DIR}/;
mkdir build; cd build;
# For mlu370
cmake .. -DBUILD_PYTHON_API=ON -DPLATFORM=MLU370 -DBUILD_TESTS=ON
```

执行单元测试依赖于 pytest、Numpy 和 OpenCV，安装依赖：

```
pip install -U pytest
cd {EASYDK_DIR}/python
pip install -r requirements.txt
```


注解：

pytest 要求 Python 版本高于 3.6。

另外，提供三个示例应用：

示例依赖于 Numpy 和 OpenCV，通过执行以下命令安装依赖：

```
cd {EASYDK_DIR}/python
pip install -r requirements.txt
```

- cnis_sync_demo.py，该应用演示了如何创建推理服务并请求同步推理。前处理和后处理使用 Python 代码实现。
- cnis_async_demo.py，该应用演示了如何创建推理服务并请求异步推理。前处理和后处理使用 C++ 代码实现。前处理使用 CnedkTransform 接口，底层调用 CNCV 算子。
- cnis_infer_only_demo.py，该应用演示了使用推理服务进行推理，前后处理在业务层 Python 代码中进行。

注解：

假如碰到未能找到 cnis 动态库错误，则需要将 Python 虚拟环境中 cnis 动态库路径添加到 LD_LIBRARY_PATH 。

6.4.2 编程模型

6.4.2.1 同步推理

1. 创建 InferServer。

```
import cnis

# Set device.
cnis.set_current_device(self.dev_id)

# Create InferServer.
infer_server = cnis.InferServer(dev_id=0)
```

2. 创建 SessionDesc，通过该对象描述 Session。

```
# Create SessionDesc.
session_desc = cnis.SessionDesc()
session_desc.name = "test_session_sync"
session_desc.engine_num = 1
session_desc.strategy = cnis.BatchStrategy.STATIC
```



```
# Load model (yolov5).
model_path = "http://video.cambricon.com/models/magicmind/v0.13.0/yolov5m_v0.13.0_4b_rgb_uint
↪8.magicmind"
session_desc.model = infer_server.load_model(model_path)
session_desc.model_input_format = cnis.NetworkInputFormat.RGB

# Set preprocessing.
# Custom preprocessing.
class PreprocYolov5(cnis.IPreproc):
  def __init__(self):
    super().__init__()
  def on_tensor_params(self, params):
    self.params = params
    return 0
  def on_preproc(self, src, dst, src_rects):
    # Do preprocessing.
    return 0

session_desc.preproc = cnis.Preprocessor()
preproc = PreprocYolov5()
cnis.set_preproc_handler(session_desc.model.get_key(), preproc)

# Set postprocessing.
# Custom postprocessing.
class PostprocYolov5(cnis.IPostproc):
  def __init__(self, threshold):
    super().__init__()
    self.threshold = threshold
  def on_postproc(self, data_vec, model_output, model_info):
    # Do postprocessing.
    return 0

session_desc.postproc = cnis.Postprocessor()
postproc = PostprocYolov5(0.5)
cnis.set_postproc_handler(session_desc.model.get_key(), postproc)
```

3. 创建同步 Session。

```
session = infer_server.create_sync_session(session_desc)
```

4. 准备输入数据。


```
import cv2

img = cv2.imread("your_image.jpg")

preproc_input = cnis.PreprocInput()
preproc_input.surf = cnis.convert_to_buf_surf(img, cnis.CnedkBufSurfaceColorFormat.BGR)

# Create package with one frame and set preproc_input to it.
tag = "stream_0"
input_pak = cnis.Package(1, tag)
input_pak.data[0].set(preproc_input)
input_pak.data[0].set_user_data({"image_width": img.shape[1], "image_height": img.shape[0]})
```

5. 准备输出数据。

```
output_pak = cnis.Package(1)
```

6. 请求同步推理。

```
status = cnis.Status.SUCCESS
ret = infer_server.request_sync(session, input_pak, status, output_pak, timeout=20000)
```

7. 销毁 Session。

```
# Remove Preproc and Postproc handlers
cnis.remove_preproc_handler(session_desc.model.get_key())
cnis.remove_postproc_handler(session_desc.model.get_key())

# Destroy Session.
infer_server.destroy_session(session)
```

6.4.2.2 异步推理

1. 创建 InferServer。

```
import cnis

# Set device.
cnis.set_current_device(self.dev_id)

# Create InferServer.
infer_server = cnis.InferServer(dev_id=0)
```

2. 创建 SessionDesc，通过该对象描述 Session。


```
# Create SessionDesc.
session_desc = cnis.SessionDesc()
session_desc.name = "test_session_async"
session_desc.engine_num = 1
session_desc.strategy = cnis.BatchStrategy.DYNAMIC

# Load model (yolov5).
model_path = "http://video.cambricon.com/models/magicmind/v0.13.0/yolov5m_v0.13.0_4b_rgb_uint
↪8.magicmind"
session_desc.model = infer_server.load_model(model_path)
session_desc.model_input_format = cnis.NetworkInputFormat.RGB

# Set preprocessing.
# The custom preprocessing PreprocYolov5 is written in C++ code as an sample.
session_desc.preproc = cnis.Preprocessor()
preproc = cnis.PreprocYolov5()
cnis.set_preproc_handler(session_desc.model.get_key(), preproc)

# Set postprocessing.
# The custom postprocessing PostprocYolov5 is written in C++ code as an sample.
session_desc.postproc = cnis.Postprocessor()
postproc = cnis.PostprocYolov5(0.5)
cnis.set_postproc_handler(session_desc.model.get_key(), postproc)
```

3. 定义一个观察者，用于接收输出数据。

```
class MyObserver(cnis.Observer):
  def __init(self):
    super().__init__()
  def response(self, status, data, user_data):
    # Process outputs here.

obs = MyObserver()
```

4. 创建异步 Session。

```
session = infer_server.create_session(session_desc, obs)
```

5. 准备输入数据。

```
import cv2

cv_image = cv2.imread(cur_file_dir + "/../data/test.jpg")
img_width = cv_image.shape[1]
```



```
img_height = cv_image.shape[0]

# Create a BufSurface
params = cnis.CnedkBufSurfaceCreateParams()
params.mem_type = cnis.CnedkBufSurfaceMemType.DEVICE
params.width = img_width
params.height = img_height
params.color_format = cnis.CnedkBufSurfaceColorFormat.NV12
params.batch_size = 1
params.device_id = 0
params.force_align_1 = True
mlu_buffer = cnis.cnedk_buf_surface_create(params)

# Convert image from BGR24 TO YUV NV12
data_size = int(img_width * img_height * 3 / 2)
i420_img = cv2.cvtColor(cv_image, cv2.COLOR_BGR2YUV_I420)
i420_img = i420_img.reshape(data_size)
img_y = i420_img[:img_width*img_height]
img_uv = i420_img[img_width*img_height:]
img_uv = img_uv.reshape((int(img_width * img_height / 4), 2), order="F").reshape(int(img_widt
↪h * img_height /2))
img = np.concatenate((img_y, img_uv), axis=0)

# Copy to mlu buffer
mlu_buffer.get_surface_list(0).copy_from(img, data_size)

# Set buffer wrapper to PreprocInput
mlu_buffer_wrapper = cnis.CnedkBufSurfaceWrapper(mlu_buffer)
preproc_input = cnis.PreprocInput()
preproc_input.surf = mlu_buffer_wrapper

# Create input package
tag = "stream_0"
input_pak = cnis.Package(1, tag)
input_pak.data[0].set(preproc_input)

# Set user data to input data, as PostprocYolov5 need to known the image width and height
input_pak.data[0].set_user_data({"image_width": img_width, "image_height": img_height})
```

6. 请求异步推理。


```
ret = infer_server.request(session, input_pak, {"user_data":"test"}, timeout=20000)
```

7. 等待所有任务结束后，销毁 Session。

```
# Wait all tasks done.
infer_server.wait_task_done(session, tag)

# Remove Preproc and Postproc handlers
cnis.remove_preproc_handler(session_desc.model.get_key())
cnis.remove_postproc_handler(session_desc.model.get_key())

# Destroy Session.
infer_server.destroy_session(session)
```

6.4.2.3 仅推理

支持使用默认前后处理，InferServer 内部仅进行推理。

- 不设置 SessionDesc::preproc 时，使用默认前处理。将输入数据拷贝至 InferServer 指定的模型输入内存中。
- 不设置 SessionDesc::postproc 时，使用默认后处理。返回模型输出的原始数据。
- 输入数据为经过预处理后，符合模型输入的数据。

```
# Prepare input data.
img = cv2.imread("your_image.jpg")
resized_img = cv2.resize(img, (width, height))
bgra_img = cv2.cvtColor(resized_img, cv2.COLOR_BGR2RGB)

tag = "stream_0"
preproc_input = cnis.PreprocInput()
preproc_input.surf = cnis.convert_to_buf_surf(src_img, cnis.CnedkBufSurfaceColorFormat.RGB)

# Create input package and set PreprocInput to input package
input_pak = cnis.Package(1, tag)
input_pak.data[0].set(preproc_input)
input_pak.data[0].set_user_data({"image_width": src_img.shape[1], "image_height": src_img.shape
↪[0]})
```

- 输出数据为模型输出，用户需要在业务层 Python 代码完成后处理。

```
# Get output data.
for data in output_pak.data:
  model_io = data.get_model_io()
```



```
output0 = model_io.surfs[0]
output1 = model_io.surfs[1]
data = output0.get_data(plane_idx = 0, batch_idx = 0, dtype=cnis.DataType.FLOAT32)
box_num = output1.get_data(plane_idx = 0, batch_idx = 0)[0]
```

6.4.3 自定义前后处理

InferServer 内置了 Yolov5 的前后处理代码。同时也支持 Python 语言创建自定义前后处理逻辑并关联至 Session 中。虽然 InferServer 框架为了提高处理效率一直并行调度前后处理，但是由于 Python GIL（Global Interpreter Lock，全局解释器锁）的存在，前后处理的 python 代码段都会被串行执行，大大降低整体的处理效率。因此对性能敏感场景请谨慎使用该功能，建议使用 C++ 编写自定义前后处理代码，或者在业务层 Python 代码完成预处理，然后喂给推理服务，推理服务则返回推理后原始数据。

6.4.3.1 自定义前处理

1. 首先继承前处理的基类 `IPreproc`。
2. 定义 `on_tensor_params` 函数，获取模型输入信息。
3. 定义 `on_preproc` 函数，实现前处理逻辑。如果处理成功则返回 0，处理失败则返回-1。

```
import cnis

class CustomPreprocess(cnis.IPreproc):
  def __init__(self):
    super().__init__()

  def on_tensor_params(self, params):
    self.params = params
    return 0

  def on_preproc(self, src, dst, src_rects):
    # Do preprocessing.
    return 0
```

4. 设置前处理。

```
...
session_desc.preproc = cnis.Preprocessor()
preproc = CustomPreprocess()
cnis.set_preproc_handler(session_desc.model.get_key(), preproc)
```


6.4.3.2 自定义后处理

1. 首先继承后处理的基类 IPostproc 。
2. 定义 on_postproc 函数，实现后处理逻辑。如果处理成功则返回 0，处理失败则返回-1。

```
import cnis

class CustomPostprocess(cnis.IPostproc):
  def __init__(self):
    super().__init__()

  def on_postproc(self, data_vec, model_output, model_info):
    # Doing postprocessing.
    return 0
```

3. 设置后处理。

```
...
session_desc.postproc = cnis.Postprocessor()
postproc = CustomPostprocess()
cnis.set_postproc_handler(session_desc.model.get_key(), postproc)
```


7 示例代码说明

7.1 人工智能应用示例

本章提供了单路和多路视频流输入的人工智能应用示例代码。分别采用了串行和并行的处理方式，用户可根据业务需要进行参考。

7.2 单路视频流人工智能应用示例

单路视频流人工智能应用示例，提供了简单的同步使用方式，并且输入的视频源只能是单一的视频源，不支持多路视频源同时工作。

```
static std::queue<CnedkBufSurface*> g_frames;
static std::mutex g_mut;
static std::condition_variable g_cond;
static std::atomic<bool> g_receive_eos{false};
static std::atomic<bool> g_running{true};

static std::unique_ptr<infer_server::InferServer> g_infer_server;
static infer_server::Session_t g_session;

constexpr const char *g_model_path =
    "http://video.cambricon.com/models/magicmind/v0.13.0/resnet50_v0.13.0_4b_rgb_uint8.magicmin
↪d";

// The decode get BufSurface callback
int GetBufSurface(CnedkBufSurface **surf,
                  int width, int height, CnedkBufSurfaceColorFormat fmt,
                  int timeoutMs, void* userdata) {
  // Create BufSurface
  CnedkBufSurfaceCreateParams params;
  ...
  CnedkBufSurfaceCreate(surf, params);
}
```



```
// The decode frame callback
int OnFrame(CnedkBufSurface *surf, void *userdata) {
  std::unique_lock<std::mutex> lk(g_mut);
  g_frames.push(surf);
  g_cond.notify_one();
}

// The eos frame callback
void OnEos() {
  std::unique_lock<std::mutex> lk(g_mut);
  g_frames.push(nullptr);
  g_cond.notify_one();
  g_receive_eos = true;
  g_cond.notify_one();
}

class MyObserver : public infer_server::Observer {
  void Response(infer_server::Status status, infer_server::PackagePtr data,
                infer_server::any user_data) noexcept override {
    auto surf = infer_server::any_cast<CnedkBufSurfaceCreate*>(user_data);
    CnedkBufSurfaceDestroy(surf);
    if (status != infer_server::Status::SUCCESS) {
      std::cout << "infer sever process failed\n";
    }
  }
}

class PreprocHandler : public infer_server::IPreproc {
 public:
  int OnTensorParams(const infer_server::CnPreprocTensorParams *params) override { return 0; }
  int OnPreproc(cnedk::BufSurfWrapperPtr src, cnedk::BufSurfWrapperPtr dst,
                const std::vector<CnedkTransformRect> &srcRects) override {
    // Do preproc
    ...
    return 0;
  }
}
class PostprocHandler : public infer_server::IPostproc {
 public:
  int OnPostproc(const std::vector<infer_server::InferData*>& data_vec,
                 const infer_server::ModelIO& model_output,
```



```
                const infer_server::ModelInfo* model_info) {
    // Do postproc
    ...
    return 0;
  }
};

static PreprocHandler g_preproc_handler = PreprocHandler();
static PostprocHandler g_postproc_handler = PostprocHandler();

bool DecodeLoop() {
  // Init decoder
  CnedkVdecCreateParams create_params;
  // Set paramters
  ...
  // Set callbacks
  create_params.GetBufSurf = GetBufSurface;
  create_params.OnFrame = OnFrame;
  create_params.OnEos = OnEos;

  void* vdec;
  CnedkVdecCreate(&vdec, &create_params);

  // For example, read a jpeg file
  std::string img_path = "your_image.jpg";
  FILE *fp = fopen(img_path.c_str(), "rb");
  if (!fp) {
    std::cerr << "img_path not exist, path: " << img_path << "\n";
    return false;
  }
  fseek(fp, 0, SEEK_END);
  uint64_t len = ftell(fp);
  rewind(fp);
  void* data = malloc(len);
  memset(data, 0, len);
  fread(data, 1, len ,fp);
  fclose(fp);

  // Decode loop
  uint64_t frame_index = 0;
  uint64_t frame_cnt = 10;
  while ((frame_cnt--) > 0 && g_running) {
```



```
    CnedkVdecStream stream;
    stream.bits = data;
    stream.len = len;
    pkt.pts = frame_index++;

    // Feed data to codec
    bool ret = g_decoder->FeedData(pkt);
    int ret = CnedkVdecSendStream(vdec_, &stream, 5000);
    if (ret < 0) {
      return false;
    }
  }
  free(data);
  // Feed eos (end of stream)
  CnedkVdecStream stream;
  stream.bits = nullptr;
  stream.len = 0;
  stream.pts = 0;
  CnedkVdecSendStream(vdec, &stream, 5000);
  return true;
}

bool InitInfer(int dev_id) {
  g_infer_server.reset(new infer_server::InferServer(dev_id));
  infer_server::SessionDesc desc;
  desc.strategy = infer_server::BatchStrategy::STATIC;
  desc.engine_num = 1;
  desc.priority = 0;
  desc.show_perf = true;
  desc.name = "infer session";
  desc.model_input_format = infer_server::NetworkInputFormat::BGR;

  // Load offline model
  desc.model = infer_server::InferServer::LoadModel(g_model_path);
  // set preproc and postproc
  desc.preproc = infer_server::Preprocessor::Create();
  desc.postproc = infer_server::Postprocessor::Create();

  infer_server::SetPreprocHandler(desc.model->GetKey(), &g_preproc_handler);
  infer_server::SetPostprocHandler(desc.model->GetKey(), &g_postproc_handler);

  g_session = g_infer_server->CreateSession(desc, std::make_shared<MyObserver>());
```



```
  if (g_session) {
    return true;
  } else {
    return false;
  }
}

bool Process(CnedkBufSurface* surf) {
  std::string tag = "stream_0";
  if (!surf) {
    g_infer_server->WaitTaskDone(g_session, tag);
    return true;
  }
  infer_server::PackagePtr request = infer_server::Package::Create(1, tag);
  infer_server::PreprocInput tmp;
  tmp.surf = std::make_shared<cnedk::BufSurfaceWrapper>(surf, false);
  tmp.has_bbox = false;
  request->data[0]->Set(std::move(tmp));
  request->data[0]->SetUserData(surf);

  if(!infer_server_->Request(session_, std::move(request), frame)) {
    CnedkBufSurfaceDestroy(surf);
    std::cout << "Call infer_server request error" << std::endl;
    return false;
  }

  return true;
}

bool RunLoop() {
  while (true) {
    std::unique_lock<std::mutex> lk(g_mut);
    g_cond.wait(lk, []() { return !g_frames.empty() || !g_running; });
    if (!g_running) return false;
    CnedkBufSurface* frame = g_frames.front();
    g_frames.pop();
    lk.unlock();

    if (!Process(std::move(frame))) return false;

    lk.lock();
    if (g_frames.size() == 0 && g_receive_eos.load()) { break; }
```



```
    lk.unlock();
  }
  return true;
}

void Destroy() {
  if (g_decoder) {
    g_decoder.reset();
  }
  if (g_infer_server && g_session) {
    infer_server::RemovePreprocHandler(g_infer_server->GetModel(g_session)->GetKey());
    infer_server::RemovePostprocHandler(g_infer_server->GetModel(g_session)->GetKey());
    g_infer_server->DestroySession(g_session);
    g_infer_server.reset();
  }
}

int main() {
  int dev_id = 0;
  cnrtSetDevice(dev_id);

  bool ret = true;
  // Init InferServer
  if (!InitInfer(dev_id)) {
    std::cout << "InitInfer failed!!" << std::endl;
    return -1;
  }

  // Start a thread to do inference
  std::future<bool> process_loop_return = std::async(std::launch::async, RunLoop);

  // Decode loop
  ret = DecodeLoop();
  if (!ret) {
    std::unique_lock<std::mutex> lk(g_mut);
    g_running = false;
    g_cond.notify_one();
  }

  // Wait all done
  ret = process_loop_return.get();
  if (ret) {
```



```
    std::cout << "\n>>>>>>>>>>>>>>>> Run succeed!! >>>>>>>>>>>>>>>>" << std::endl;
  } else {
    std::cout << "\n>>>>>>>>>>>>>>>> Run failed!! >>>>>>>>>>>>>>>>" << std::endl;
  }
  Destroy();
  return 0;
}
```

完整流程示例代码见路径：easydk/samples/simple_demo

- samples/simple_demo/detection 集成了视频解码、图像预处理（颜色空间转换和图像缩放）、人工智能推理、显示等功能，实现了一个以 Yolov3 网络为基础的目标检测应用程序。
- samples/simple_demo/classification 集成了视频解码、图像预处理（颜色空间转换和图像缩放）、人工智能推理、显示等功能，实现了一个以 ResNet50 网络为基础的目标识别应用程序。

7.3 推理服务示例

7.3.1 推理示例

使用 `InferServer` 的推理示例如下，示例包含同步接口和异步接口的使用。

```
namespace infer_server {

class MyObserver : public Observer {
  void Response(Status status, PackagePtr output, any user_data) {
    std::cout << "Get one response\n";
  }
};

class MyPreprocHandler : public IPreproc {
  int OnTensorParams(const infer_server::CnPreprocTensorParams* params) override {
    std::cout << "On Tensor Params\n";
    return 0;
  };
  int OnPreproc(cnedk::BufSurfWrapperPtr src, cnedk::BufSurfWrapperPtr dst,
                const std::vector<CnedkTransformRect>& src_rects) override {
    std::cout << "On Preproc\n";
    return 0;
  };
}
```



```
class MyPostprocHandler : public IPostproc {
  int OnPostproc(const std::vector<infer_server::InferData*>& data_vec,
                 const infer_server::ModelIO& model_output,
                 const infer_server::ModelInfo* model_info) override {
    std::cout << "On Postproc\n";
    return 0;
  };
}

// Load offline model
ModelPtr model = InferServer::LoadModel(model_path);

// Create session
SessionDesc desc;
desc.model = model;
desc.model_input_format = NetworkInputFormat::RGB;
desc.preproc = Preprocessor::Create();
desc.postproc = Postprocessor::Create();
desc.strategy = BatchStrategy::DYNAMIC;
desc.batch_timeout = 200;
desc.show_perf = false;
desc.priority = 1;
desc.engine_num = 1;

MyPreprocHandler preproc_handler;
SetPreprocHandler(model->GetKey(), &preproc_handler);
MyPostprocHandler postproc_handler;
SetPostprocHandler(model->GetKey(), &postproc_handler);

int device_id = 0;
InferServer s(device_id);

// Create session
#if ASYNC
Session_t session = s.CreateSession(desc, std::make_shared<MyObserver>());
#else
Session_t session = s.CreateSyncSession(desc);
#endif

// Prepare input data with tag "stream_0"
// for wait or discard task with specified tag
auto package = Package::Create(1, "stream_0");
```



```
CnedkBufSurface *surf;
CnedkBufSurfaceCreateParams params;
params.device_id = device_id;
params.batch_size = 1;
params.color_format = CNEDK_BUF_COLOR_FORMAT_NV12;
params.width = width;
params.height = height;
// MLUxxx device 内存。
params.mem_type = CNEDK_BUF_MEM_DEVICE;
CnedkBufSurfaceCreate(&surf, &params)

PreprocInput input;
input.surf = std::make_shared<cnedk::BufSurfaceWrapper>(surf);
input.has_bbox = false;

package->data[0]->Set(std::move(input));

#if ASYNC
// run inference async
s.Request(session, std::move(package), nullptr);
// wait all tasks are finished
s.WaitTaskDone(session, "stream_0");
#else
// run inference
auto out = std::make_shared<Package>();
Status stat;
s.RequestSync(session, std::move(package), &stat, out);
#endif

RemovePreprocHandler(model->GetKey());
RemovePostprocHandler(model->GetKey());
s.DestroySession(session);

}  // namespace infer_server
```


7.3.2 自定义后处理单元示例

该示例展示了如何自定义后处理单元。模型输出数据以连续 batch 数据的形式存在，例如 | unit_1 unit_2 unit3 unit_4 |，该后处理单元简单地将模型输出数据拆分为分散的 unit，| unit_1 | unit_2 | unit_3 | unit_4 |，设置给输出的 `InferData`。

```
class MyPostprocessor : public ProcessorForkable<MyPostprocessor> {
public:
 MyPostprocessor() : ProcessorForkable<MyPostprocessor>("MyPostprocessor") {}
 ~MyPostprocessor() {}

 Status Init() noexcept override {
   // 检查是否所有必需参数已设置
   constexpr const char* params[] = {"model_info", "device_id"};
   for (auto p : params) {
     if (!HaveParam(p)) {
       LOG(ERROR) << p << " has not been set!";
       return Status::INVALID_PARAM;
     }
   }
   // 读取参数
   try {
     model_ = GetParam<ModelPtr>("model_info");
     dev_id_ = GetParam<int>("device_id");
   } catch (bad_any_cast&) {
     LOG(ERROR) << "Unmatched data type";
     return Status::WRONG_TYPE;
   }
   return Status::SUCCESS;
 }

Status Process(PackagePtr pack) noexcept override {
  // 检查输入，后处理单元的输入 package 中 predict_io 应该包含模型输出数据
  if (!pack->predict_io || !pack->predict_io->HasValue()) return Status::ERROR_BACKEND;

  auto output = pack->predict_io->Get<ModelIO>();
  auto& shape = output.shapes[0];
  for (uint32_t idx = 0; idx < pack->data.size(); ++idx) {
    // 将整个 batch 的模型输出数据逐份拷贝到分离的内存上
    CnedkBufSurface *surf = nullptr;
    CnedkBufSurfaceCreateParams params;
    params.device_id = dev_id_;
```

```
    params.batch_size = 1;
    params.color_format = CNEDK_BUF_COLOR_FORMAT_TENSOR;
    params.size = output.surfs[0]->GetSurfaceParams(idx)->data_size;;
    // MLUxxx device 内存。
    params.mem_type = CNEDK_BUF_MEM_DEVICE;
    CnedkBufSurfaceCreate(&surf, &params);
    auto wrapper = std::make_shared<cnedk::BufSurfaceWrapper>(surf);

    cnrtMemcpy(wrapper->GetData(0), output.surfs[0]->GetData(0, idx), params.size, cnrtMemcpyDe
↪vToDev);

    pack->data[idx]->Set(wrapper);
  }
  return Status::SUCCESS;
}

private:
 ModelPtr model_;
 int dev_id_;
};
```

7.4 多路视频流人工智能应用示例

多路视频流人工智能应用示例，代码位于 easydk/samples/easy_pipeline ，提供了一个简易的 pipeline，可以充分的提高整体硬件吞吐能力，提高硬件利用率。

此示例中内置了 decode、inference、OSD、encode 四个模块，用户可以自由选择搭配（decode 作为输入源，属于必要模块）。

用户在此示例中还可以配置视频源路数。

7.4.1 stream_app

stream_app 是简单的一个多路应用。用户可以参考此示例并结合自己的需求来构建自己的 pipeline。代码位于 easydk/samples/easy_pipeline/stream_app.cpp。

构建 pipeline 的步骤：

1. 实例化一个 pipeline。

```
std::shared_ptr<EasyPipeline> easy_pipe;
easy_pipe = std::make_shared<EasyPipeline>();
```


2. 实例化模块。

   可以根据需求实例化不同的模块。

```
std::shared_ptr<EasyModule> source =
    std::make_shared<SampleDecode>("source", 0, FLAGS_device_id, FLAGS_data_path, i, FLAGS_fr
↪ame_rate);

std::shared_ptr<EasyModule> infer =
    std::make_shared<SampleAsyncInference>("infer", 1, FLAGS_device_id, FLAGS_model_path, FLA
↪GS_model_name);

std::shared_ptr<EasyModule> osd = std::make_shared<SampleOsd>("osd", 1, FLAGS_label_path);
std::shared_ptr<EasyModule> encode = std::make_shared<SampleEncode>("encode", 1, FLAGS_device
↪_id, FLAGS_output_name);
```

3. 将模块添加至 pipeline。

```
easy_pipe->AddModule(source);
easy_pipe->AddModule(infer);
easy_pipe->AddModule(osd);
easy_pipe->AddModule(encode);
```

4. 根据模块名字来进行模块连接。

   可以根据需求来连接不同的模块。

```
easy_pipe->AddLink("source", "infer");
easy_pipe->AddLink("infer", "osd");
easy_pipe->AddLink("osd", "encode");
```

   如果只想要解码和编码，连接 source 和 encode 模块。

```
easy_pipe->AddLink("source", "encode");
```

5. 启动 pipeline 并等待结束。

```
easy_pipe->Start();
easy_pipe->WaitForStop();
easy_pipe->Stop();
```


7.4.2 framework

framework 中包含了 EasyPipeline 、EasyModule 和 EasyFrame 类。代码位于 easydk/samples/easy_pipeline/framework 。

- EasyPipeline: 主要负责数据转运，以及数据分发。
- EasyModule: 是所有模块的基类，如果用户想要实现在即的模块，需要继承这个类。用户在实现自己的模块时，需要注意必须手动调用 Transmit()，这样才能保证数据流转给下一个模块。
- EasyFrame: pipeline 中流转的 frame，主要包含 MLU 侧的图像数据，经过处理过后的 bounding box，以及追踪 ID 等。

```
class EasyModule {
 public:
  EasyModule(std::string name, int parallelism) : module_name_(name), parallelism_(parallelis
↪m) {
  }

  ~EasyModule() = default;

  void SetProcessDoneCallback(std::function<int(std::shared_ptr<EdkFrame>)> callback) {
    callback_ = callback;
  }

  virtual int Transmit(std::shared_ptr<EdkFrame> frame) final {  // NOLINT
    if (callback_) return callback_(frame);
    return 0;
  }

  int GetParallelism() {
    return parallelism_;
  }

  std::string GetModuleName() {
    return module_name_;
  }

 public:
  virtual int Open() = 0;
  virtual int Process(std::shared_ptr<EdkFrame> frame) = 0;
  virtual int Close() = 0;

 private:
```



```
  std::string module_name_ = "";
  int parallelism_ = 1;
  std::function<int(std::shared_ptr<EdkFrame>)> callback_;
  std::shared_ptr<EasyModule> next_module_ = nullptr;
};
```

```
struct BoundingBox {
  float x = 0.0;
  float y = 0.0;
  float w = 0.0;
  float h = 0.0;
};

struct DetectObject {
  float score;
  int label;
  BoundingBox bbox;
  std::string track_id;
  std::map<std::string, std::string> attributes;  // add info into bbox, secondary infer cla
↪ssfication
};

class EdkFrame {
public:
  int stream_id;
  uint64_t frame_idx;
  bool is_eos;
  std::vector<DetectObject> objs;
  std::string track_id;
  std::map<std::string, std::string> attributes;  // add info into bbox, secondary infer cla
↪ssfication
  cnedk::BufSurfWrapperPtr surf;
};
```


7.4.3 decode 模块

decode 模块是整个 pipeline 的数据源头，负责产生数据。模块使用的是寒武纪硬件解码产生的 YUV 数据，数据存放在 MLU 侧。代码位于 `easydk/samples/easy_pipeline/decode` ，包含以下特性：

- 模块支持 FFmpeg RTSP 流、H.264、H.265 文件以及 JPEG 文件。
- 模块支持帧率控制。
- 解码后的图片数据保存在内存池中。

7.4.4 inference 模块

inference 模块主要展示了 InferServer 的使用方式。代码位于 `easydk/samples/easy_pipeline/inference` ，包含以下特性：

- 同步和异步请求推理模块（SampleSyncInference 和 SampleAsyncInference）。
- 实现了 Yolov3 的前后处理。
- 实现了 ResNet50 的前后处理。

7.4.5 OSD 模块

OSD 模块使用 OpenCV 将检测结果 `EasyFrame::objs` 绘制在原图 `EasyFrame::surf` 上。代码位于 `easydk/samples/easy_pipeline/osd` ，包含以下特性：

- 支持绘制检测结果和分类结果。
- 同步绘制结果到设备内存。

7.4.6 encode 模块

encode 模块使用寒武纪硬件进行编码。可以将图像数据编码成 H.264 及 H.265 码流。代码位于 `easydk/samples/easy_pipeline/encode` ，主要包含以下特性：

- 支持编码成不同的码流（H.264/H.265）。

7.4.7 用户自定义模块

自定义模块，参考以下代码。

```
class FakeModule : public Module {
 public:
  FakeModule(std::string name, int parallelism) : Module(name, parallelism) { }
```



```
  ~FakeModule() = default;

  int Open() override {
    return 0;
  }

  int Process(std::shared_ptr<EdkFrame> frame) override {
    // user code
    std::cout << "frame stream id: "<< frame->stream_id << ", frame idx: " << frame->frame_idx
↪<< std::endl;
    // std::this_thread::sleep_for(std::chrono::microseconds(200));
    // std::cout << "stream_id: " << frame->stream_id << ", is eos: " << frame->is_eos << std:
↪:endl;
    Transmit(frame);
    return 0;
  }

  int Close() override {
    return 0;
  }
};
```


Cambricon 寒武纪

8 FAQ

8.1 有没有交叉编译 EasyDK 的指导?

可下载 edge 编译压缩包 http://video.cambricon.com/models/edge.tar.gz，解压后按照 README 文档提供的步骤进行编译。

8.2 如何将 Log 打印到终端以及如何带颜色显示?

可以通过环境变量 GLOG_alsologtostderr 或者命令行参数 alsologtostderr 将日志打印到终端。可以通过环境变量 GLOG_colorlogtostderr 或者命令行参数 colorlogtostderr 使不同等级的日志显示为不同的颜色。

8.3 怎么调整 Log 打印等级?

可以通过环境变量 GLOG_minloglevel 或者命令行参数 minloglevel 调整日志输出等级，范围 [0-3]，数字越小输出的 log 内容越多。

- 0 代表打印 LOG(INFO)
- 1 代表打印 LOG(WARNING)
- 2 代表打印 LOG(ERROR)
- 3 代表打印 LOG(FATAL)

另外，如果希望打印更多日志信息，在 minloglevel=0 的前提下，设置环境变量 GLOG_v 或者命令行参数 v 调整自定义日志等级，范围 [1-5]，数字越大输出的 log 内容越多。

