天数智芯

IxRT 推理引擎使用指南

版本：V4.0.0-MR

日期：2024.6.6

适用产品：智铠 50| 智铠 100

1 声明

1.1 版权声明

1.2 免责声明

天数智芯可以随时对本文档或本文档中描述的产品进行改进和/或更改。本文档包括与天数智芯产品有关的信息，作为说明典型应用的一种方式, 因此，不一定提供足以进行生产设计的完整信息。对于本文档中内容的准确性或完整性，天数智芯不做任何陈述或保证。

1.3 联系方式

地址：上海闵行区陈行公路 2168 号 3 幢

电话：021-68886607

网址：www.iluvatar.com

Contents

2 修订记录

- COREX01-MR400-UG05-02：2024/6/6
  - 更新” 教程 1：对 ResNet18 模型进行 FP16 推理 (Python)”，统一库名和配置文件变量名
- COREX01-MR400-UG05-01：2024/5/9
  - 删除” 教程 1：对 Bert 模型进行 FP16 推理” 小节，该内容已失效
  - 新增” 教程 6：ResNet18 分类网络动态形状推理下的多运行环境” 小节
- COREX01-MR400-UG05-00：2024/4/3

  文档本次发布内容与 V3.2.1-MR 文档相比 (COREX01-MR321-UG05-00)，有以下更新：
  - 更新本次发布的 IxRT 版本为 v0.9.0
  - 更新”IxRT 推理引擎介绍”：
    * 调整”IxRT 的主要使用流程”
    * 更新”IxRT 的功能特性”
  - ”IxRT 推理引擎示例安装指南” 章节中，删除原” 安装配套辅助软件” 小节
  - ”IxRT 模型推理示例教程” 章节中，删除原” 教程 1：对 Bert 模型进行 FP16 推理” 小节，删除理由：该内容已失效
  - 更新” 使用 IxRT 推理引擎进行模型推理最佳实践” 章节，删除 `static_quantize` 接口中的部分参数，如 `quant_format` 等
  - 新增”IxRT 动态形状推理功能介绍” 章节
  - 更新”IxRT 自定义插件算子使用指南” 章节，新增” 使用插件生成工具” 小节，提供插件生成工具的安装和使用方法
  - ”IxRT-EXEC 自动化模型推理工具使用指南” 章节中，删除 **--batch_size** 运行参数，删除理由：该参数已失效
  - 新增” 配套软件使用指南” 章节，提供 IxRT 配套第三方软件的安装指导、使用示例和 API 参考等信息
  - 新增” 关于 IxRT 使用的高级话题” 章节，提供面向开发者使用 IxRT 的话题
  - 更新” 附录 1：IxRT 适配的模型列表” 章节，新增部分模型
  - 更新” 附录 2：IxRT 支持的算子列表” 章节，删除”addFill” API
  - 更新” 附录 3：IxRT 支持的网络定义 API 列表”，
  - 新增” 附录 4：IxRT 支持的插件列表” 章节，并提供各插件的算子详情

3 IxRT 推理引擎介绍

IxRT (Iluvatar CoreX RunTime) 是天数智芯的专用推理加速引擎，支持对业界主流训练框架的模型进行解析及优化。通过 IxRT 的自动部署工具即可将训练好的模型快速部署到天数智芯加速卡上，实现视觉、语音、推荐、自然语言等领域模型在天数智芯加速卡上的最优推理性能。当前提供的 IxRT 版本是 v0.9.0。

IxRT 提供推理示例方便您快速上手，请参考"IxRT 推理引擎示例安装指南" 了解详情。

3.1 IxRT 的功能模块

IxRT 推理引擎包含以下模块：

- ONNX 模型解析器：从训练框架 (如 PyTorch、TensorFlow、PaddlePaddle 等) 导出 ONNX 模型文件导入到 IxRT 中，IxRT 会自动完成对 ONNX 的解析以及计算图生成。
- 部署工具：支持使用 IxRT 的部署工具对原始模型进行修改 (如修改、添加 Op)，并支持对原始模型进行低精度 (如 FP16、INT8) 量化，在精度损失较小的情况下可以达到更高的推理性能。
- 图优化引擎：支持通过 IxRT 内置的高效图优化引擎，对模型的计算图进行深度优化，从而达到模型的极致性能。
- 序列化工具：支持将深度优化后的计算图在加密后序列化成可执行的引擎文件 (engine file)，直接加载该引擎文件即可快速执行推理。
- 运行时：提供高效的运行时模块，支持识别加密的引擎文件并在加载后执行推理。

3.2 IxRT 的主要使用流程

IxRT 推理引擎支持 C++ 和 Python 接口，使用这两种接口进行推理的流程大致相同，可总结为如下几步：

1. 构建引擎文件
2. (可选) 量化模型
3. (可选) 加载引擎文件
4. 执行模型推理

其中，构建引擎文件时，需要您从一种神经网络模型表示来构建，IxRT 支持从 ONNX 导入模型，也支持使用 API 来自定义搭建模型。详细的 API 使用流程，请参考"(面向开发者) 使用 IxRT 接口进行开发"，另外可参考"IxRT 模型推理示例教程" 了解具体的推理场景教程。

3.3 IxRT 的功能特性

3.3.1 ONNX 格式解析支持

IxRT 推理引擎原生支持 ONNX 格式的解析。ONNX，即 Open Neural Network Exchange，开放神经网络交换，是一种针对机器学习所设计的开放式的文件格式，用于存储训练好的模型。如果您使用第三方训练框架进行训练，请使用对应的转换工具将文件转成 ONNX 格式的文件。下面提供常见训练框架转换至 ONNX 格式的参考链接：

- PaddlePaddle：https:// www.paddlepaddle.org.cn/ documentation/ docs/ zh/ guides/ advanced/ model_to_onnx_cn.html
- PyTorch：https://pytorch.org/docs/stable/onnx.html
- TensorFlow：https://www.tensorflow.org/api_docs/python/tf/compat/v1/lite/TFLiteConverter

建议您使用 onnxsim 工具提前对 ONNX 进行简化，再加载进入 IxRT 推理引擎。

3.3.2 C++ 和 Python API 支持

IxRT 推理引擎原生使用 C++ 实现，其 API 和业界主流推理引擎兼容。IxRT 还提供 Python 绑定接口，从而助您接入 Python 丰富的软件生态，和常见的第三方库，如 Numpy、PyCUDA、OpenCV 等库共用。C++ API 在推理性能上可能会更高效，而 Python API 在开发效率上则占优。您可以根据实际需求选取 API 进行开发。

C++ 和 Python API 的编程指导请参考”(面向开发者) 使用 IxRT 接口进行开发”。

3.3.3 编程范式

IxRT 推理可大体分为两个阶段：

- 构建引擎文件
- 执行引擎文件

上述两个阶段的目的有明显区分：前者离线，后者运行时；前者通过多优化方式，确定天数智芯加速卡上的最佳运行参数，后者则利用这组参数执行推理。

您也可以将这两个阶段写在一个程序中，但不建议您这么做，因为构建引起文件的时间会随着模型规模不同而不等 (几分钟到十几分钟不等)，而有了构建好的引擎文件则可以直接加载快速用于推理。

上述编程范式的详情请参考”(面向开发者) 使用 IxRT 接口进行开发”。

Note

请您确保构建引擎文件和执行引擎文件的 IxRT 版本一致，以避免推理引擎内部调整导致的程序崩溃。

3.3.4 动态形状推理支持

动态形状是指在推理运行时，变换输入形状大小以获得对应输出这一需求。

IxRT 默认使用固定形状推理，比如 ONNX 输入的形状，或是构建 `INetworkDefinition` 指定的输入形状。如果使用动态形状推理，您需要使用 `OptimizationProfile`，在构建引擎文件时指定动态形状的配置，详情请参考”IxRT 动态形状推理功能介绍”。

3.3.5 自定义插件算子支持

IxRT 提供插件接口供您实现高性能自定义算子。实现提供的 C++ 接口插件，并将其注册入 IxRT 后，IxRT 便可以按照您的实现规则去解析 ONNX。当然，实现好 C++ 接口后，您还可以在 Python 上调用这个插件算子。

关于自定义插件算子的详细内容，请参考”IxRT 自定义插件算子使用指南”。

关于 IxRT 支持的插件列表，请参考” 附录 4：IxRT 支持的插件列表”。

3.3.6 部署工具支持

IxRT 提供 INT8 量化工具并支持多种定点算法供您选择，量化工具原生支持导出 QDQ/量化表两种风格的导出方式。

关于量化知识和工具的使用指导，请参考” 使用 IxRT 推理引擎对模型进行 INT8 推理最佳实践”。

3.3.7 类型和精度支持

目前，IxRT 支持 FP16 和 IN8 两种运行精度，部分算子还支持 INT32、BOOL、UINT8 等数据类型。

关于 IxRT 支持的算子列表，请参考” 附录 2：IxRT 支持的算子列表”。

3.3.8 快速验证工具 IxRT-EXEC 支持

IxRT 提供命令行工具 IxRT-EXEC 用于快速验证，该工具目前支持以下功能：

- 快速检验一个 ONNX 是否支持
- 初步得到 IxRT 在这个 ONNX 上的参考性能
- 进行详细性能分析并导出表格
- 基于 ONNX 快速构建引擎文件
- 导出 IxRT 支持的算子列表

在使用 IxRT-EXEC 前，您需要先安装天数智芯适配版的 PyCUDA Python 库，详情请参考” 配套软件使用指南”。

IxRT-EXEC 的具体使用方法，请参考”IxRT-EXEC 自动化模型推理工具使用指南”。

3.3.9 预览特性

自动量化是 IxRT 提供的非常实用的功能，主要是为了完成量化、构建引擎文件、执行推理的一体化流程，主要步骤包括：

- 导入 ONNX 模型或引擎文件，并用校准数据集推理，统计权重的范围以及激活值的分布直方图
- 根据以上的统计信息，计算 Scale 值
- 利用量化好的模型以及量化表信息构建引擎文件，执行 INT8 推理

3.3.10 IxRT Profiling 工具

IxRT Profiling 工具是用于对 IxRT 每次运行的 Kernel 和算子进行 Profiling。该工具对 IxRT 算子的 Profiling 主要依赖 ixTX 和 ixSYS 进行实现。其中，您可以使用 ixTX 来标记算子的运行，使用 ixSYS 的可视化工具查看 Profiling 的结果。

下面简单说明 IxRT Profiling 的使用方法：

- 运行推理脚本前设置环境变量：

```
$ export IXRT_NVTX=ON
```

- 运行推理脚本时添加命令 (支持 Python 和 Shell)：

```
$ ixsys -o profiling.ptrace -t CUDA,NVTX

# 使用 Profiling 运行推理脚本示例
$ ixsys -o profiling.ptrace -t CUDA,NVTX bash inference.sh
```

- 使用 ixSYS 的可视化工具查看 Profiling 的结果。

  打开 ixSYS，通过光标选取一段区域，显示每个 Kernel 的运行时间。

关于 ixTX 和 ixSYS 的详细说明和操作指导，请参考《软件栈工具使用指南》中的” 系统性能分析工具 ixSYS”。

3.3.11 配套软件支持

IxRT 推理引擎核心使用 C++ 进行开发，但使用 IxRT Python API 进行模型部署时依赖第三方软件，因此天数智芯提供了相应的适配版，目前支持 CUDA Python 和 PyCUDA。

- CUDA Python：CUDA Python 可以让您以 Pythonic 的方式轻松访问天数智算软件栈支持的 CUDA API。CUDA Python 为 CUDA Driver 和 Runtime API 提供了 Cython 和 Python 封装，简化了基于 GPU 的并行计算。
- PyCUDA：PyCUDA 可以让您以 Pythonic 的方式轻松访问天数智算软件栈支持的 CUDA API。

关于配套软件的详细内容，请参考” 配套软件使用指南”。

3.4 IxRT 适配的模型

IxRT 推理引擎适配的模型涉及的领域有视觉、自然语言和语音等，详细模型列表请参考” 附录 1：IxRT 适配的模型列表”。

3.5 IxRT 支持的网络定义 API

详细网络定义列表请参考” 附录 3：IxRT 支持的网络定义 API 列表”。

4 IxRT 推理引擎示例安装指南

本章节旨在介绍 IxRT 推理引擎的示例安装过程和展示运行示例。我们提供以下方式安装 IxRT 推理引擎：

- 在 Docker 镜像中使用 IxRT 推理引擎
- 安装 Python 库
- 通过 C++ 库安装 IxRT 推理引擎 - .run 文件安装方式
- 通过 C++ 库安装 IxRT 推理引擎 - tar.gz 文件安装方式

在安装 IxRT 推理引擎前，请您先参考《软件栈安装指南》安装天数智算软件栈。

4.1 在 Docker 镜像中使用 IxRT 推理引擎

如果您在安装天数智算软件栈时使用的是 Docker 镜像安装包，该镜像中已包含 IxRT C++ 库，无需您再手动安装。

如果您需要调用 Python 库，请参考安装 Python 库。

4.2 安装 Python 库

1. 请向您的应用工程师获取 IxRT 的 `.whl` 包并使用 **pip3** 进行安装。
2. 安装完成后，在终端运行如下命令检查安装是否成功：

   ```
   python3 -c "import tensorrt; print(tensorrt.__version__)"
   ```

   如果回显版本号，表示安装成功。
3. 运行示例。

   下面以 `.whl` 包中自带的示例为例，介绍使用 IxRT 进行推理的步骤：

   ```
   $ cd IxRT/oss/samples/python/resnet18/
   $ python3 resnet18.py
   $ python3 resnet18.py --precision int8
   $ python3 resnet18.py --multicontext
   $ python3 resnet18.py --dynamicshape
   ```

4.3 通过 C++ 库安装 IxRT 推理引擎 - .run 文件安装方式

1. 向您的应用工程师获取 IxRT 的 `.run` 包并运行：

   ```
   $ sudo bash ixrt-${ixrt_version}+corex.{v.r.m}-linux_x86_64.run
   ```

通过上述命令即可将 IxRT 安装至系统目录，默认为 /usr/local/corex/，需要管理员权限进行安装。

您也可以使用 **--prefix** 参数来指定安装目录，比如：

```
sudo bash ixrt-${ixrt_version}+corex.{v.r.m}-linux_x86_64.run -- --prefix=~/softwares/ixrt
```

2. 安装完成后，设置如下环境变量或将其写到您的 ~/.bashrc 中 (如果您用的是 bash，配置文件是 ~/.bashrc，其它 shell 请参考对应配置文件)：

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/corex/lib
```

3. 解压后的目录结构如下：

```
$ tree -L 1 ${path_for_run_package}

${path_for_run_package}/IxRT
├── bin
├── cmake
├── CMakeLists.txt
├── data
├── include # 头文件所在路径
├── install
├── lib # 库文件所在路径
├── plugins # IxRT 提供的插件库
├── README.md
├── requirements
├── samples # 示例脚本所在路径
└── scripts

10 directories, 3 files
```

4. 运行示例。

下面以 .run 包中自带的示例为例，介绍使用 IxRT 进行推理的步骤：

```
$ cd ${path_for_run_package}/IxRT

# 编译示例
$ cmake -B build
$ cmake --build build -j

# ResNet18
$ ./build/bin/sampleResNet18 tex
$ ./build/bin/sampleResNet18 tex_s_onnx
$ ./build/bin/sampleResNet18 ten
$ ./build/bin/sampleResNet18 tmc
$ ./build/bin/sampleResNet18 ted
$ ./build/bin/sampleResNet18 tmcd

## YOLOv3
```

```
# 修改 ONNX 文件
$ DATADIR=$(realpath data/yolov3)
$ python3 samples/sampleYoloV3/deploy.py --src ${DATADIR}/
 ↪  quantized_yolov3_without_decoder_shape.onnx --dest ${DATADIR}/
 ↪  quantized_yolov3_with_decoder_plugin.onnx --custom
$ python3 samples/sampleYoloV3/deploy.py --src ${DATADIR}/
 ↪  quantized_yolov3_without_decoder_shape.onnx --dest ${DATADIR}/
 ↪  quantized_yolov3_with_decoder_plugin_dynamic.onnx --custom --dynamic

# 推理
# 示例 1: 插件算子 + 固定形状
$ ./build/bin/sampleYoloV3 --onnx ${DATADIR}/quantized_yolov3_with_decoder_plugin.onnx --
 ↪  engine ${DATADIR}/quantized_yolov3_with_decoder_plugin.engine --demo trt_exe
# 示例 2: 插件算子 + 动态形状
$ ./build/bin/sampleYoloV3 --onnx ${DATADIR}/
 ↪  quantized_yolov3_with_decoder_plugin_dynamic.onnx --engine ${DATADIR}/
 ↪  quantized_yolov3_with_decoder_plugin_dynamic.engine --demo trt_dyn

## YOLOv5
$ export YOLOV5_CHECKPOINTS_DIR=./data/yolov5/
# 修改 ONNX 文件
$ bash samples/sampleYoloV5/deploy.sh samples/sampleYoloV5/config/YOLOV5S_CONFIG --
 ↪  precision float16

# FP16 推理
$ ./build/bin/sampleYoloV5 --precision fp16 --data_dir $YOLOV5_CHECKPOINTS_DIR

## YOLOX
# 修改 ONNX 文件，添加 Plugin 节点
$ DATADIR=$(realpath data/yolox_m)
$ python3 samples/sampleYoloX/create_onnx.py --src ${DATADIR}/yolox_m_offical_focus.onnx --
 ↪  dest ${DATADIR}/yolox_m_with_decoder_nms.onnx
# 推理
$ ./build/bin/sampleYoloX --onnx ${DATADIR}/yolox_m_with_decoder_nms.onnx --engine
 ↪  ${DATADIR}/yolox_m_with_decoder_nms.engine --input ${DATADIR}/dog_640.jpg --demo
 ↪  trt_exec
```

4.4 通过 C++ 库安装 IxRT 推理引擎 - tar.gz 文件安装方式

1. 向您的应用工程师获取 IxRT 的 tar.gz 包，解压并安装 IxRT：

```
$ tar xvzf ixrt-${ixrt_version}+corex.{v.r.m}-linux_x86_64.tar.gz
$ cd IxRT
$ sudo ./install
```

您也可以使用 **--prefix** 参数来指定安装目录，比如：

```
$ sudo ./install --prefix=~/softwares/ixrt
```

2. 安装后的目录结构如下：

```
$ tree -L 1 ${path_for_tar_package}

${path_for_tar_package}/IxRT
├── bin
├── cmake
├── CMakeLists.txt
├── data
├── include # 头文件所在路径
├── install
├── lib # 库文件所在路径
├── plugins # IxRT 提供的插件库
├── README.md
├── requirements
├── samples # 示例脚本所在路径
└── scripts

10 directories, 3 files
```

3. 安装完成后，设置如下环境变量或将其写到您的 ~/.bashrc 中 (如果您用的是 bash，配置文件是 ~/.bashrc，其它 shell 请参考对应配置文件)：

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/corex/lib
```

4. 运行示例。

   下面以 tar.gz 包中自带的示例为例，介绍使用 IxRT 进行推理的步骤：

```
$ cd ${path_for_tar_package}/IxRT

# 编译示例
$ cmake -B build
$ cmake --build build -j

# ResNet18
$ ./build/bin/sampleResNet18 tex
$ ./build/bin/sampleResNet18 tex_s_onnx
$ ./build/bin/sampleResNet18 ten
$ ./build/bin/sampleResNet18 tmc
$ ./build/bin/sampleResNet18 ted
$ ./build/bin/sampleResNet18 tmcd

## YOLOv3
# 修改 ONNX 文件
```

```
$ DATADIR=$(realpath data/yolov3)
$ python3 samples/sampleYoloV3/deploy.py --src ${DATADIR}/
 ↪  quantized_yolov3_without_decoder_shape.onnx --dest ${DATADIR}/
 ↪  quantized_yolov3_with_decoder_plugin.onnx --custom
$ python3 samples/sampleYoloV3/deploy.py --src ${DATADIR}/
 ↪  quantized_yolov3_without_decoder_shape.onnx --dest ${DATADIR}/
 ↪  quantized_yolov3_with_decoder_plugin_dynamic.onnx --custom --dynamic

# 推理
# 示例 1：插件算子 + 固定形状
$ ./build/bin/sampleYoloV3 --onnx ${DATADIR}/quantized_yolov3_with_decoder_plugin.onnx --
 ↪  engine ${DATADIR}/quantized_yolov3_with_decoder_plugin.engine --demo trt_exe
# 示例 2：插件算子 + 动态形状
$ ./build/bin/sampleYoloV3 --onnx ${DATADIR}/
 ↪  quantized_yolov3_with_decoder_plugin_dynamic.onnx --engine ${DATADIR}/
 ↪  quantized_yolov3_with_decoder_plugin_dynamic.engine --demo trt_dyn

## YOLOv5
$ export YOLOV5_CHECKPOINTS_DIR=./data/yolov5/
# 修改 ONNX 文件
$ bash samples/sampleYoloV5/deploy.sh samples/sampleYoloV5/config/YOLOV5S_CONFIG --
 ↪  precision float16

# FP16 推理
$ ./build/bin/sampleYoloV5 --precision fp16 --data_dir $YOLOV5_CHECKPOINTS_DIR

## YOLOX
# 修改 ONNX 文件，添加 Plugin 节点
$ DATADIR=$(realpath data/yolox_m)
$ python3 samples/sampleYoloX/create_onnx.py --src ${DATADIR}/yolox_m_offical_focus.onnx --
 ↪  dest ${DATADIR}/yolox_m_with_decoder_nms.onnx
# 推理
$ ./build/bin/sampleYoloX --onnx ${DATADIR}/yolox_m_with_decoder_nms.onnx --engine
 ↪  ${DATADIR}/yolox_m_with_decoder_nms.engine --input ${DATADIR}/dog_640.jpg --demo
 ↪  trt_exec
```

5 IxRT 模型推理示例教程

使用 IxRT 推理引擎进行模型推理的大致流程如下：

1. 安装 IxRT 环境
2. 获取和量化 ONNX 文件
3. 将 ONNX 文件转换为引擎文件
4. 构建 IxRT 运行时并加载引擎文件
5. 执行推理

本章节提供了以下典型模型推理示例教程来针对性地帮助用户了解 IxRT 推理引擎的使用方法：

教程 1：对 ResNet18 模型进行 FP16 推理 (Python)

教程 2：ResNet18 分类网络同步推理

教程 3：ResNet18 分类网络异步推理

教程 4：ResNet18 多执行环境推理

教程 5：ResNet18 动态形状推理

教程 6：ResNet18 分类网络动态形状推理下的多运行环境

5.1 教程 1：对 ResNet18 模型进行 FP16 推理 (Python)

本教程以使用 Python 接口进行 ResNet18 推理为例进行说明，以下提供的所有代码都可在 `.../samples/python/resnet18/resnet8.py` 中找到。

使用 Python 接口进行 IxRT 推理的流程为：

步骤 1 获取 ONNX 文件

步骤 2 运行推理流程

IxRT 使用开放神经网络交换 (Open Neural Network Exchange, ONNX) 格式模型作为输入，不同训练框架所训练出的模型均可转为 ONNX 格式。下面以 PyTorch 为例，将 ResNet18 模型转换为 ONNX 格式。

5.1.1 步骤 1 获取 ONNX 文件

通过如下代码获取 ONNX 文件：

```
from torch.autograd import Variable
import torch.onnx
import torchvision

# 将模型导出为 ONNX 格式
dummy_input = Variable(torch.randn(1, 3, 224, 224))
model = torchvision.models.resnet18(pretrained=True)
torch.onnx.export(model, dummy_input, "resnet18.onnx")
```

5.1.2 步骤 2 运行推理流程

1. 创建模型表达。

   载入 ONNX 文件并保存为 TensorRT 的表达：

   ```
   onnx_file_path = os.path.join(dir_path, 'resnet18.onnx')
   TRT_LOGGER = tensorrt.Logger()
   builder = tensorrt.Builder(TRT_LOGGER)
   EXPLICIT_BATCH = 1 << (int)(tensorrt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
   network = builder.create_network(EXPLICIT_BATCH)
   parser = tensorrt.OnnxParser(network, TRT_LOGGER)
   parser.parse_from_file(onnx_file_path)
   ```

2. 设定模型配置。

   创建配置对象，并设置运行精度为 float16：

   ```
   build_config = builder.create_builder_config()
   build_config.set_flag(tensorrt.BuilderFlag.FP16)
   ```

3. 创建引擎文件。

   使用模型对象和配置对象可以创建序列化后的引擎文件，这里命名为 plan，可以直接以二进制的方式序列化到磁盘上：

   ```
   plan = builder.build_serialized_network(network, build_config)
   ```

   如果直接使用 engine 对象，需要使用 runtime 对象进行反序列化：

   ```
   logger = tensorrt.Logger(tensorrt.Logger.ERROR)
   runtime = tensorrt.Runtime(logger)
   engine = runtime.deserialize_cuda_engine(plan)
   ```

4. 运行推理流程。

   可以使用 cuda-python 或 PyCUDA 库，获取 GPU 存储资源。假设，目前输入和输出的存储空间地址存储于列表对象 buffers 中，使用 engine 对象，创建执行环境 context：

   ```
   context = engine.create_execution_context()
   ```

   使用 context 对象执行同步推理流程：

   ```
   context.execute_v2(buffers)
   ```

5.2 教程 2：ResNet18 分类网络同步推理

分类网络是最简单的视觉任务网络，通过本教程，您将了解如下操作：

- 创建模型
- 从 ONNX 解析模型

- 构建引擎文件
- 序列化和反序列化引擎文件
- 创建运行时
- 同步推理
- 搭建推理流程
- 访问模型和运行环境属性
- 使用 Logger 接口

5.2.1 示例

样例代码和数据获取方式：

- 代码文件：IxRT OSS 目录下 ${path_for_run_package}/IxRT/samples/sampleResNet/future_classifier.cc
- 数据路径：IxRT OSS 目录下 ${path_for_run_package}/IxRT/data 目录

下面给出整个推理过程的完整代码，对于其中包含的一些自定义函数将在细节解释中给出实现代码：

```
std::string dir_path("data/resnet18/");
std::string image_path(dir_path + "kitten_224.bmp");
std::string model_path(dir_path + "resnet18_shape_opset11.onnx");
std::string quant_file(dir_path + "quantized_resnet18_shape_opset11.json");
std::string input_name("input");
std::string output_name("output");
Logger logger(iluvatar::ILogger::Severity::kVERBOSE);
auto builder = UniquePtr<iluvatar::IBuilder>(iluvatar::createInferBuilder(logger));
if (not builder) {
    std::cout << "Create builder failed" << std::endl;
    return;
} else {
    cout << "Create builder success" << endl;
}
if (builder->platformHasFastInt8()) {
    cout << "Current support Int8 inference" << endl;
} else {
    cout << "Current not support Int8 inference" << endl;
}
const auto explicitBatch = 1U <<
 ↪  static_cast<uint32_t>(iluvatar::NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
auto network = UniquePtr<iluvatar::INetworkDefinition>(builder->createNetworkV2(explicitBatch));
if (not network) {
    std::cout << "Create network failed" << std::endl;
    return;
} else {
    cout << "Create network success" << endl;
}
```

```
auto config = UniquePtr<iluvatar::IBuilderConfig>(builder->createBuilderConfig());
if (not config) {
    std::cout << "Create config failed" << std::endl;
    return;
} else {
    cout << "Create config success" << endl;
}
config->setFlag(iluvatar::BuilderFlag::kINT8);
auto parser = UniquePtr<iluvatar::IParser>(iluvatar::createParser(*network, logger));
if (not parser) {
    std::cout << "Create config failed" << std::endl;
    return;
} else {
    cout << "Create config success" << endl;
}
auto parsed =
    parser->parseFromFile(model_path.c_str(), static_cast<int>(logger.getReportableSeverity()),
    ↪ quant_file.c_str());
if (!parsed) {
    std::cout << "Create onnx parser failed" << std::endl;
    return;
} else {
    cout << "Create onnx parser success" << endl;
}

auto num_input = network->getNbInputs();
cout << "number of input: " << num_input << endl;
auto num_output = network->getNbOutputs();
cout << "number of output: " << num_output << endl;

iluvatar::Dims inputDims = network->getInput(0)->getDimensions();
ASSERT(inputDims.nbDims == 4);
cout << "\nInput dimes: " << endl;
for (auto i = 0; i < inputDims.nbDims; ++i) {
    cout << inputDims.d[i] << " ";
}

cout << "\nOutput dimes: " << endl;
iluvatar::Dims outputDims = network->getOutput(0)->getDimensions();
ASSERT(outputDims.nbDims == 2);
for (auto i = 0; i < outputDims.nbDims; ++i) {
    cout << outputDims.d[i] << " ";
}
cout << endl;

UniquePtr<iluvatar::IHostMemory> plan{builder->buildSerializedNetwork(*network, *config)};
```

```
if (not plan) {
    std::cout << "Create serialized engine plan failed" << std::endl;
    return;
} else {
    cout << "Create serialized engine plan done" << endl;
}

UniquePtr<iluvatar::IRuntime> runtime{iluvatar::createInferRuntime(logger)};
if (not runtime) {
    std::cout << "Create runtime failed" << std::endl;
    return;
} else {
    cout << "Create runtime done" << endl;
}
std::shared_ptr<iluvatar::ICudaEngine> engine;
if (true) {
    // Option operation, not necessary
    DumpBuffer2Disk("/home/work/trt.engine", plan->data(), plan->size());
    std::vector<int8_t> engine_buffer;
    LoadBufferFromDisk("/home/work/trt.engine", &engine_buffer);
    engine = std::shared_ptr<iluvatar::ICudaEngine>(
        runtime->deserializeCudaEngine(engine_buffer.data(), engine_buffer.size()),
 ↪  ObjectDeleter());
} else {
    engine = std::shared_ptr<iluvatar::ICudaEngine>(runtime->deserializeCudaEngine(plan->data(),
 ↪  plan->size()),
                                                    ObjectDeleter());
}

if (not engine) {
    std::cout << "Create engine failed" << std::endl;
    return;
} else {
    std::cout << "Create engine done" << endl;
}

auto input_idx = engine->getBindingIndex(input_name.c_str());
cout << "Input index: " << input_idx << endl;
auto output_idx = engine->getBindingIndex(output_name.c_str());
cout << "Output index: " << output_idx << endl;
auto cpu_fp32_image = LoadImageCPU(image_path, inputDims);
std::vector<void*> binding_buffer(engine->getNbBindings());
auto input_size = GetBytes(inputDims, engine->getBindingDataType(input_idx));
auto output_size = GetBytes(outputDims, engine->getBindingDataType(output_idx));
std::shared_ptr<float> cpu_output(new float[output_size / sizeof(float)], ArrayDeleter());
void* input_gpu{nullptr};
```

```
CHECK(cudaMalloc(&input_gpu, input_size));
void* output_gpu{nullptr};
CHECK(cudaMalloc(&output_gpu, output_size));
CHECK(cudaMemcpy(input_gpu, cpu_fp32_image.get(), input_size, cudaMemcpyHostToDevice));
cout << "User input date prepare done" << endl;
binding_buffer.at(input_idx) = input_gpu;
binding_buffer.at(output_idx) = output_gpu;
auto context = UniquePtr<iluvatar::IExecutionContext>(engine->createExecutionContext());
if (context) {
    cout << "Create execution context done" << endl;
} else {
    cout << "Create execution context failed" << endl;
}

auto status = context->executeV2(binding_buffer.data());
if (not status) {
    cerr << "Execute IxRT failed" << endl;
} else {
    cout << "Execute IxRT success" << endl;
}

CHECK(cudaMemcpy(cpu_output.get(), output_gpu, output_size, cudaMemcpyDeviceToHost));
GetClassificationResult(cpu_output.get(), 1000, 5, 0);
CHECK(cudaFree(input_gpu));
CHECK(cudaFree(output_gpu));
cout << "Resnet-18 with IxRT API demo done" << endl;
```

5.2.2 将 ONNX 文件转换为引擎文件

将 ONNX 文件转换为引擎文件需要经过以下步骤：

1. 创建 Builder
2. 创建模型定义
3. 配置模型运行参数
4. 解析 ONNX 文件

(可选) 5. 访问模型信息

6. 构建引擎文件

IxRT 允许用户的 Logger 接口访问内部返回的日志信息，因此创建 Logger 并给到各个创建的对象，有助于了解运行状态。这里给出最简单的范例，打印到命令行。由于传出的是字符串，也可以保存到文件或转发到中间件。需要继承 ILogger 类并且至少实现 log 方法。

```
using Severity = iluvatar::ILogger::Severity;
class Logger : public iluvatar::ILogger {
   public:
    explicit Logger(Severity severity = Severity::kWARNING) : mReportableSeverity(severity) {}
    iluvatar::ILogger& getIxRTLogger() noexcept { return *this; }

    void log(Severity severity, const char* msg) noexcept override {
        if (severity <= mReportableSeverity) {
            std::cout << severityPrefix(mReportableSeverity) << "[IXRT] " << msg << std::endl;
        }
    }

    void setReportableSeverity(Severity severity) noexcept { mReportableSeverity = severity; }

    Severity getReportableSeverity() const { return mReportableSeverity; }

   private:

    static const char* severityPrefix(Severity severity) {
        switch (severity) {
            case Severity::kINTERNAL_ERROR:
                return "[F] ";
            case Severity::kERROR:
                return "[E] ";
            case Severity::kWARNING:
                return "[W] ";
            case Severity::kINFO:
                return "[I] ";
            case Severity::kVERBOSE:
                return "[V] ";
            default:
                assert(0);
                return "";
        }
    }

    Severity mReportableSeverity;
};  // class Logger
```

5.2.2.1 1. 创建 Builder

主要作用是创建构成引擎文件的各类对象。例如模型的定义和配置：

```
auto builder = UniquePtr<iluvatar::IBuilder>(iluvatar::createInferBuilder(logger));
```

IxRT 中，创建的对象资源都是以裸指针的形式返回，因此这里提供了标准库的 unique_ptr 方便内存管理。这里的"UniquePtr" 是添加了删除器的版本，定义如下：

```
struct ObjectDeleter {
    template <typename T>
    void operator()(T* obj) const {
        delete obj;
    }
};

template <typename T>
using UniquePtr = std::unique_ptr<T, ObjectDeleter>;
```

创建后可以判断当前平台是否支持 INT8 推理：

```
builder->platformHasFastInt8()
```

Note

目前只支持返回 true。

5.2.2.2 2. 创建模型定义

构建保存模型计算图的对象：

```
const auto explicitBatch = 1U <<
 ↪  static_cast<uint32_t>(iluvatar::NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
auto network = UniquePtr<iluvatar::INetworkDefinition>(builder->createNetworkV2(explicitBatch));
```

"explicitBatch" 是一个固定的设置，表示没有隐含维度。

5.2.2.3 3. 配置模型运行参数

指定网络整体以何种精度进行推理：

```
auto config = UniquePtr<iluvatar::IBuilderConfig>(builder->createBuilderConfig());
config->setFlag(iluvatar::BuilderFlag::kINT8);
```

这里指定了使用 INT8 模式进行推理。

5.2.2.4 4. 解析 ONNX 文件

根据 ONNX 文件的内容，初始化模型：

```
auto parser = UniquePtr<iluvatar::IParser>(iluvatar::createParser(*network, logger));
auto parsed = parser->parseFromFile(model_path.c_str(),
 ↪  static_cast<int>(logger.getReportableSeverity()), quant_file.c_str());
```

模型解析器使用 Network 和 Logger 创建。解析方法 `parseFromFile` 的第 3 个参数为量化参数文件路径。

5.2.2.5 (可选) 5. 访问模型信息

这部分不是必须的，只是展示了属性访问的相关功能并打印出来：

```
auto num_input = network->getNbInputs();
cout << "number of input: " << num_input << endl;
auto num_output = network->getNbOutputs();
cout << "number of output: " << num_output << endl;

iluvatar::Dims inputDims = network->getInput(0)->getDimensions();
ASSERT(inputDims.nbDims == 4);
cout << "\nInput dimes: " << endl;
for (auto i = 0; i < inputDims.nbDims; ++i) {
    cout << inputDims.d[i] << " ";
}

cout << "\nOutput dimes: " << endl;
iluvatar::Dims outputDims = network->getOutput(0)->getDimensions();
ASSERT(outputDims.nbDims == 2);
for (auto i = 0; i < outputDims.nbDims; ++i) {
    cout << outputDims.d[i] << " ";
}
cout << endl;
```

5.2.2.6 6. 构建引擎文件

使用模型定义和配置构建引擎文件。这个过程进行了：

1. 规划推理资源
2. 优化计算图
3. 选择最优算法

```
UniquePtr<iluvatar::IHostMemory> plan{builder->buildSerializedNetwork(*network, *config)};
```

构建引擎文件之后，返回的是一个朴素的 CPU 内存对象，IHostMemory 的功能很少，只能访问数据指针和数据长度。因此可以按照写文件的方式，以二进制形式写到磁盘，以及逆过程：

```
DumpBuffer2Disk("/home/work/trt.engine", plan->data(), plan->size());
```

方法定义为：

```
void DumpBuffer2Disk(const std::string& file_path, void* data, uint64_t len) {
    std::ofstream out_file(file_path, std::ios::binary);
    if (not out_file.is_open()) {
        out_file.close();
        return;
    }
    out_file.write((char*)data, len);
    out_file.close();
}
```

如上所示，保存在 plan 对象中的引擎文件内容，可以写到磁盘任意位置，并可以从磁盘加载内容到内存。如果模型优化过程耗时较长的话，推荐您先保存到引擎文件到磁盘，推理时直接加载引擎文件省去了重复进行优化过程。

5.2.3 加载引擎文件并执行推理

加载引擎文件并执行推理需要经过以下步骤：

1. 反序列化引擎文件
2. 创建数据资源
3. 创建运行环境
4. 执行推理
5. 获取输出结果

5.2.3.1 1. 反序列化引擎文件

1. 您需要从已序列化的引擎文件中恢复模型的定义和配置，用于创建运行时环境。首先，需要创建一个类似于 builder 的 Runtime：

   ```
   UniquePtr<iluvatar::IRuntime> runtime{iluvatar::createInferRuntime(logger)};
   ```

2. 使用 Runtime 可以从引擎文件的内存对象中解析出引擎文件对象：

   ```
   std::shared_ptr<iluvatar::ICudaEngine> engine;
   ```

   如果是直接从 IHostMemory 中恢复引擎文件，可以直接通过数据指针访问：

   ```
   engine =
    ↪  std::shared_ptr<iluvatar::ICudaEngine>(runtime->deserializeCudaEngine(plan->data(),
    ↪  plan->size()),
                                                ObjectDeleter());
   ```

如果是从磁盘恢复，需要从磁盘读入二进制内容，然后反序列化：

```
std::vector<int8_t> engine_buffer;
LoadBufferFromDisk("/home/work/trt.engine", &engine_buffer);
engine = std::shared_ptr<iluvatar::ICudaEngine>(
    runtime->deserializeCudaEngine(engine_buffer.data(), engine_buffer.size()),
 ↪  ObjectDeleter());
```

加载定义为：

```
void LoadBufferFromDisk(const std::string& file_path, std::vector<int8_t>* engine_buffer) {
    std::ifstream in_file(file_path, std::ios::binary);
    if (not in_file.is_open()) {
        in_file.close();
        return;
    }
    in_file.seekg(0, std::ios::end);
    uint64_t file_length = in_file.tellg();
    in_file.seekg(0, std::ios::beg);
    engine_buffer->resize(file_length);
    in_file.read((char*)engine_buffer->data(), file_length);
    in_file.close();
}
```

除了 vector 外，可以使用任意的内存对象，只要容量和文件长度相等即可。

5.2.3.2 2. 创建数据资源

1. 为了示例展示的需要，准备图像输入数据。根据输入和输出的名字，获取输入和输出维度，数据访问：

```
auto input_idx = engine->getBindingIndex(input_name.c_str());
auto output_idx = engine->getBindingIndex(output_name.c_str());
```

2. 使用输入和输出名字获取绑定 buffer 的列表索引：

```
auto cpu_fp32_image = LoadImageCPU(image_path, inputDims);
```

3. 使用 Stb lib 库打开图片 (也可以使用 OpenCV)。这里使用了 inputDims 做一些合法性验证，展示代码中将略去此部分：

```
std::shared_ptr<uint8_t> GetImagePtr(const std::string& file_name, int32_t* w, int32_t* h,
 ↪  int32_t* c, int32_t mode) {
    uint8_t* raw_ptr = stbi_load(file_name.c_str(), w, h, c, mode);
    std::shared_ptr<uint8_t> raw_data(raw_ptr, [](uint8_t* p) { stbi_image_free(p); });
    return raw_data;
}

std::shared_ptr<float> LoadImageCPU(const std::string& file_name, const iluvatar::Dims&
 ↪  dims) {
```

```
    std::shared_ptr<uint8_t> image_ptr{nullptr};
    int32_t w, h, c;
    image_ptr = GetImagePtr(file_name, &w, &h, &c, 0);

    uint64_t num_element = w * h * c;

    std::shared_ptr<float> fp32_data(new float[num_element], [](float* p) { delete[] p; });
    auto cpu_data_ptr = fp32_data.get();
    uint8_t* raw_image_ptr = image_ptr.get();

    uint64_t raw_data_area = w * h;
    // From HWC to CHW for real IxRT API
    for (auto i = 0; i < raw_data_area; ++i) {
        for (auto k = 0; k < c; ++k) {
            cpu_data_ptr[i + k * raw_data_area] = static_cast<float>(raw_image_ptr[i * 3 +
↪   k]) / 255.f;
        }
    }

    return fp32_data;
}
```

4. 读取图片，并将 uint8 的数据转换为 float32，除以 255。**注意：原始加载的图片数据时 HWC 的 BGR 数据，ONNX 导出的输入形式是 CHW，读取时也进行了排布的转换，满足输入要求。**
5. 通过引擎文件的信息获取需要绑定列表的大小。这里使用一个 vector 存储列表：

```
std::vector<void*> binding_buffer(engine->getNbBindings());
```

6. 计算待分配输入和输出资源的尺寸，单位是字节：

```
auto input_size = GetBytes(inputDims, engine->getBindingDataType(input_idx));
auto output_size = GetBytes(outputDims, engine->getBindingDataType(output_idx));
```

计算分配容量的函数的定义如下：

```
uint32_t getElementSize(iluvatar::DataType t) noexcept {
    switch (t) {
        case iluvatar::DataType::kINT32:
            return 4;
        case iluvatar::DataType::kFLOAT:
            return 4;
        case iluvatar::DataType::kHALF:
            return 2;
        case iluvatar::DataType::kBOOL:
        case iluvatar::DataType::kINT8:
            return 1;
    }
```

```
    return 0;
}
uint64_t GetBytes(const iluvatar::Dims& dims, iluvatar::DataType t) noexcept {
    uint64_t ret{1};
    for (auto i = 0; i < dims.nbDims; ++i) {
        if (dims.d[i] > 0) {
            ret *= dims.d[i];
        }
    }
    if (ret > 1) {
        return ret * getElementSize(t);
    } else {
        return 0;
    }
}
```

7. 分配一段 CPU 的内存用于接收结果：

```
std::shared_ptr<float> cpu_output(new float[output_size / sizeof(float)], ArrayDeleter());
```

关于 shared_ptr 的删除器定义为：

```
struct ArrayDeleter {
    template <typename T>
    void operator()(T* p) const {
        delete[] p;
    }
};
```

8. 申请 GPU 显存并指定到 buffer 列表中：

```
void* input_gpu{nullptr};
CHECK(cudaMalloc(&input_gpu, input_size));
void* output_gpu{nullptr};
CHECK(cudaMalloc(&output_gpu, output_size));
CHECK(cudaMemcpy(input_gpu, cpu_fp32_image.get(), input_size, cudaMemcpyHostToDevice));

binding_buffer.at(input_idx) = input_gpu;
binding_buffer.at(output_idx) = output_gpu;
```

其中，CHECK 宏用于检查 CUDA 返回的一些错误：

```
#define CHECK(status)                                                \
    do {                                                             \
        auto ret = (status);                                         \
        if (ret != 0) {                                              \
            std::cerr << "Cuda failure: " << ret << std::endl;       \
            abort();                                                 \
```

```
        }                                                   \
    } while (0)
#endif
```

5.2.3.3 3. 创建运行环境

使用引擎文件创建运行环境，IExecutionContext 对象：

```
auto context = UniquePtr<iluvatar::IExecutionContext>(engine->createExecutionContext());
```

注意：引擎文件需要保持和 context 同样的生命周期。

5.2.3.4 4. 执行推理

使用 executeV2 函数执行推理过程，输入参数只有一个，即推理用的输入和输出数据：

```
auto status = context->executeV2(binding_buffer.data());
```

executeV2 方法返回，即表示推理过程结束。

5.2.3.5 5. 获取输出结果

1. 将输出到 GPU 端内存的结果保存到 CPU 端内存：

   ```
   CHECK(cudaMemcpy(cpu_output.get(), output_gpu, output_size, cudaMemcpyDeviceToHost));
   ```

2. 展示输出结果，主要内容是排序。代码内容较长，在此不展示具体代码：

   ```
   GetClassificationResult(cpu_output.get(), 1000, 5, 0);
   ```

3. 最终释放掉 GPU 端使用的资源：

   ```
   CHECK(cudaFree(input_gpu));
   CHECK(cudaFree(output_gpu));
   ```

5.3 教程 3：ResNet18 分类网络异步推理

本章节重点介绍使用 IxRT 对 ResNet18 进行异步推理的过程。其中涉及到的操作细节，请参考” 教程 3：ResNet18 分类网络同步推理”。

5.3.1 代码总览

Note

以下代码去除了有效性判断部分。

```
std::string dir_path("/home/data/cls/");
std::string image_path(dir_path + "kitten_224.bmp");
std::string model_path(dir_path + "resnet18.onnx");
std::string quant_file(dir_path + "quantized_resnet18_shape_opset11.json");
std::string input_name("input");
std::string output_name("output");
Logger logger(iluvatar::ILogger::Severity::kWARNING);
auto builder = UniquePtr<iluvatar::IBuilder>(iluvatar::createInferBuilder(logger));

if (builder->platformHasFastInt8()) {
    cout << "Current support Int8 inference" << endl;
} else {
    cout << "Current not support Int8 inference" << endl;
}
const auto explicitBatch = 1U <<
 ↪  static_cast<uint32_t>(iluvatar::NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
auto network = UniquePtr<iluvatar::INetworkDefinition>(builder->createNetworkV2(explicitBatch));

auto config = UniquePtr<iluvatar::IBuilderConfig>(builder->createBuilderConfig());

config->setFlag(iluvatar::BuilderFlag::kINT8);
auto parser = UniquePtr<iluvatar::IParser>(iluvatar::createParser(*network, logger));

auto parsed =
    parser->parseFromFile(model_path.c_str(), static_cast<int>(logger.getReportableSeverity()),
 ↪  quant_file.c_str());

iluvatar::Dims inputDims = network->getInput(0)->getDimensions();

iluvatar::Dims outputDims = network->getOutput(0)->getDimensions();


UniquePtr<iluvatar::IHostMemory> plan{builder->buildSerializedNetwork(*network, *config)};

UniquePtr<iluvatar::IRuntime> runtime{iluvatar::createInferRuntime(logger)};

auto engine =
 ↪  std::shared_ptr<iluvatar::ICudaEngine>(runtime->deserializeCudaEngine(plan->data(),
 ↪  plan->size()),
                                                ObjectDeleter());
```

```
auto input_idx = engine->getBindingIndex(input_name.c_str());
auto output_idx = engine->getBindingIndex(output_name.c_str());
auto cpu_fp32_image = LoadImageCPU(image_path, inputDims);
std::vector<void*> binding_buffer(engine->getNbBindings());
auto input_size = GetBytes(inputDims, engine->getBindingDataType(input_idx));
auto output_size = GetBytes(outputDims, engine->getBindingDataType(output_idx));

std::shared_ptr<float> cpu_output(new float[output_size / sizeof(float)], ArrayDeleter());
void* input_gpu{nullptr};
CHECK(cudaMalloc(&input_gpu, input_size));
void* output_gpu{nullptr};
CHECK(cudaMalloc(&output_gpu, output_size));
CHECK(cudaMemcpy(input_gpu, cpu_fp32_image.get(), input_size, cudaMemcpyHostToDevice));
binding_buffer.at(input_idx) = input_gpu;
binding_buffer.at(output_idx) = output_gpu;

auto context = UniquePtr<iluvatar::IExecutionContext>(engine->createExecutionContext());

auto stream_ptr = makeCudaStream();
auto event_ptr = makeCudaEvent();

auto status = context->enqueueV2(binding_buffer.data(), *stream_ptr, event_ptr.get());

cudaEventSynchronize(*event_ptr);

cudaStreamSynchronize(*stream_ptr);

CHECK(cudaMemcpy(cpu_output.get(), output_gpu, output_size, cudaMemcpyDeviceToHost));
GetClassificationResult(cpu_output.get(), 1000, 5, 0);
CHECK(cudaFree(input_gpu));
CHECK(cudaFree(output_gpu));
```

5.3.2 异步推理说明

异步调用推理过程在引擎文件创建等模型准备或预处理阶段几乎完全一致，区别在于调用 enqueueV2：

```
context->enqueueV2(binding_buffer.data(), *stream_ptr, event_ptr.get());
```

主机端调用该方法后会立刻返回，推理过程此时还未结束。其中 stream 可以指定在某个流队列中执行，在当前推理流程结束前，请不要重复设置不同的流，否则将有未定义的行为。stream 和 event 的创建如下：

```
auto StreamDeleter = [](cudaStream_t* pStream) {
    if (pStream) {
        cudaStreamDestroy(*pStream);
```

```
        delete pStream;
    }
};

std::unique_ptr<cudaStream_t, decltype(StreamDeleter)> makeCudaStream() {
    std::unique_ptr<cudaStream_t, decltype(StreamDeleter)> pStream(new cudaStream_t,
↪   StreamDeleter);
    if (cudaStreamCreateWithFlags(pStream.get(), cudaStreamNonBlocking) != cudaSuccess) {
        pStream.reset(nullptr);
    }

    return pStream;
}

auto EventDeleter = [](cudaEvent_t* p_event) {
    if (p_event) {
        cudaEventDestroy(*p_event);
        delete p_event;
    }
};

std::unique_ptr<cudaEvent_t, decltype(EventDeleter)> makeCudaEvent(bool timing = false) {
    std::unique_ptr<cudaEvent_t, decltype(EventDeleter)> p_event(new cudaEvent_t, EventDeleter);
    if (timing) {
        if (cudaEventCreate(p_event.get()) != cudaSuccess) {
            p_event.reset(nullptr);
        }
    } else {
        if (cudaEventCreateWithFlags(p_event.get(), cudaEventDisableTiming) != cudaSuccess) {
            p_event.reset(nullptr);
        }
    }
    return p_event;
}
```

如果需要确认推理过程是否结束，请调用：

```
cudaStreamSynchronize(*stream_ptr);
```

异步推理机制方便用户搭建隐蔽数据传输的流程。因此，在输入数据被消耗后即可立即开始下一次数据加载流程。enqueueV2 的第 3 个参数是一个 CUDA event，用于获取和输入数据相关的算子何时执行结束：

```
cudaEventSynchronize(*event_ptr);
```

在主机端同步，这个同步点代表输入相关算子已经执行结束，但推理过程还未结束。

5.4 教程 4：ResNet18 多执行环境推理

多个执行环境，相当于共享模型权重，但每一个执行环境拥有独立的运行空间。本教程提供的内容可以应用于同一模型多线程推理的场景。

5.4.1 示例代码总览

这里创建了两套不同的输入数据，并使用两个不同的 IExecutionContext 对象进行推理：

```
std::string dir_path("/home/data/cls/");
std::string image_path(dir_path + "kitten_224.bmp");
std::string image_path_2("/home/data/cls/robin_224.bmp");
std::string model_path(dir_path + "resnet18.onnx");
std::string quant_file(dir_path + "quantized_resnet18_shape_opset11.json");
std::string input_name("input");
std::string output_name("output");
Logger logger(iluvatar::ILogger::Severity::kVERBOSE);
auto builder = UniquePtr<iluvatar::IBuilder>(iluvatar::createInferBuilder(logger));
if (builder->platformHasFastInt8()) {
    cout << "Current support Int8 inference" << endl;
} else {
    cout << "Current not support Int8 inference" << endl;
}
const auto explicitBatch = 1U <<
 ↪  static_cast<uint32_t>(iluvatar::NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
auto network = UniquePtr<iluvatar::INetworkDefinition>(builder->createNetworkV2(explicitBatch));

auto config = UniquePtr<iluvatar::IBuilderConfig>(builder->createBuilderConfig());
config->setFlag(iluvatar::BuilderFlag::kINT8);
auto parser = UniquePtr<iluvatar::IParser>(iluvatar::createParser(*network, logger));
auto parsed =
    parser->parseFromFile(model_path.c_str(), static_cast<int>(logger.getReportableSeverity()),
 ↪  quant_file.c_str());

iluvatar::Dims inputDims = network->getInput(0)->getDimensions();

cout << "\nOutput dimes: " << endl;
iluvatar::Dims outputDims = network->getOutput(0)->getDimensions();

UniquePtr<iluvatar::IHostMemory> plan{builder->buildSerializedNetwork(*network, *config)};

UniquePtr<iluvatar::IRuntime> runtime{iluvatar::createInferRuntime(logger)};

auto engine =
 ↪  std::shared_ptr<iluvatar::ICudaEngine>(runtime->deserializeCudaEngine(plan->data(),
 ↪  plan->size()),
```

```
                                    ObjectDeleter());

auto input_idx = engine->getBindingIndex(input_name.c_str());
cout << "Input index: " << input_idx << endl;
auto output_idx = engine->getBindingIndex(output_name.c_str());
cout << "Output index: " << output_idx << endl;
auto cpu_fp32_image = LoadImageCPU(image_path, inputDims);
auto cpu_fp32_image_2 = LoadImageCPU(image_path_2, inputDims);
std::vector<void*> binding_buffer(engine->getNbBindings());
std::vector<void*> binding_buffer_2(engine->getNbBindings());
auto input_size = GetBytes(inputDims, engine->getBindingDataType(input_idx));
auto output_size = GetBytes(outputDims, engine->getBindingDataType(output_idx));
std::shared_ptr<float> cpu_output(new float[output_size / sizeof(float)], ArrayDeleter());
std::shared_ptr<float> cpu_output_2(new float[output_size / sizeof(float)], ArrayDeleter());
void* input_gpu{nullptr};
void* input_gpu_2{nullptr};
CHECK(cudaMalloc(&input_gpu, input_size));
CHECK(cudaMalloc(&input_gpu_2, input_size));
void* output_gpu{nullptr};
void* output_gpu_2{nullptr};
CHECK(cudaMalloc(&output_gpu, output_size));
CHECK(cudaMalloc(&output_gpu_2, output_size));
CHECK(cudaMemcpy(input_gpu, cpu_fp32_image.get(), input_size, cudaMemcpyHostToDevice));
CHECK(cudaMemcpy(input_gpu_2, cpu_fp32_image_2.get(), input_size, cudaMemcpyHostToDevice));
cout << "User input date prepare done" << endl;
binding_buffer.at(input_idx) = input_gpu;
binding_buffer.at(output_idx) = output_gpu;
binding_buffer_2.at(input_idx) = input_gpu_2;
binding_buffer_2.at(output_idx) = output_gpu_2;
auto context = UniquePtr<iluvatar::IExecutionContext>(engine->createExecutionContext());
auto context_2 = UniquePtr<iluvatar::IExecutionContext>(engine->createExecutionContext());
auto status = context->executeV2(binding_buffer.data());

auto status_2 = context_2->executeV2(binding_buffer_2.data());

CHECK(cudaMemcpy(cpu_output.get(), output_gpu, output_size, cudaMemcpyDeviceToHost));
CHECK(cudaMemcpy(cpu_output_2.get(), output_gpu_2, output_size, cudaMemcpyDeviceToHost));
cout << "Result 1: " << endl;
GetClassificationResult(cpu_output.get(), 1000, 5);
cout << "Result 2: " << endl;
GetClassificationResult(cpu_output_2.get(), 1000, 5);
CHECK(cudaFree(input_gpu));
CHECK(cudaFree(output_gpu));
CHECK(cudaFree(input_gpu_2));
CHECK(cudaFree(output_gpu_2));
```

5.4.2 多个 context 说明

从引擎文件创建两个 context 对象，共享引擎文件的模型权重部分，但每一个 context 都拥有独立的运行空间：

```
auto context = UniquePtr<iluvatar::IExecutionContext>(engine->createExecutionContext());
auto context_2 = UniquePtr<iluvatar::IExecutionContext>(engine->createExecutionContext());
```

两个不同的 context 可以放到不同的线程中，独立运行。**注意：context 需与引擎文件拥有相同的生命周期。**

5.5 教程 5：ResNet18 动态形状推理

应对输入可变的场景，IxRT 要求 ONNX 输入的维度通过字符串而不是具体的数值表达动态的维度。例如，固定尺寸 ResNet18 的输入维度为 1 x 3 x 224 x 224，如果高度和宽度可变，输入维度修改为 bs x 3 x 224 x 224。动态维度的调整，可以通过 PyTorch 导出为 ONNX 时指定，也可以通过 Python 进行修改：

```
import onnx
import onnx.utils

onnx_path = "./model_bank/resnet18.onnx"
model = onnx.load(onnx_path)
dim_proto0 = model.graph.input[0].type.tensor_type.shape.dim[0]
dim_proto0.dim_param = "bs"
onnx.save(model, "./model_bank/resnet18-dynamic.onnx")
```

上述代码示例，就是将 ResNet18 的输入的第 1 个维度修改为动态的，字符串的内容是任意的，只要设置了就表示这个维度是动态的。

5.5.1 示例代码总览

```
std::string dir_path("/home/data/cls/");
std::string image_path(dir_path + "kitten_224.bmp");
std::string image_path_2(dir_path + "kitten_196.bmp");
std::string model_path(dir_path + "resnet18-all-dynamic.onnx");
std::string quant_file(dir_path + "quantized_resnet18_shape_opset11.json");
std::string input_name("input");
std::string output_name("output");
Logger logger(iluvatar::ILogger::Severity::kVERBOSE);
auto builder = UniquePtr<iluvatar::IBuilder>(iluvatar::createInferBuilder(logger));
const auto explicitBatch = 1U <<
 ↪  static_cast<uint32_t>(iluvatar::NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
auto network = UniquePtr<iluvatar::INetworkDefinition>(builder->createNetworkV2(explicitBatch));
```

```
auto config = UniquePtr<iluvatar::IBuilderConfig>(builder->createBuilderConfig());
config->setFlag(iluvatar::BuilderFlag::kINT8);
auto profile = builder->createOptimizationProfile();
profile->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kMIN, iluvatar::Dims{4,
 ↪  {1, 3, 112, 112}});
profile->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kOPT, iluvatar::Dims{4,
 ↪  {1, 3, 224, 224}});
profile->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kMAX, iluvatar::Dims{4,
 ↪  {1, 3, 448, 448}});
config->addOptimizationProfile(profile);
auto parser = UniquePtr<iluvatar::IParser>(iluvatar::createParser(*network, logger));
auto parsed =
    parser->parseFromFile(model_path.c_str(), static_cast<int>(logger.getReportableSeverity()),
 ↪  quant_file.c_str());

iluvatar::Dims inputDims = network->getInput(0)->getDimensions();

iluvatar::Dims outputDims = network->getOutput(0)->getDimensions();

UniquePtr<iluvatar::IHostMemory> plan{builder->buildSerializedNetwork(*network, *config)};

UniquePtr<iluvatar::IRuntime> runtime{iluvatar::createInferRuntime(logger)};

auto engine =
 ↪  std::shared_ptr<iluvatar::ICudaEngine>(runtime->deserializeCudaEngine(plan->data(),
 ↪  plan->size()),
                                                    ObjectDeleter());

auto input_idx = engine->getBindingIndex(input_name.c_str());
auto output_idx = engine->getBindingIndex(output_name.c_str());
iluvatar::Dims dynamic_input_dims{4, {1, 3, 224, 224}};
auto cpu_fp32_image = LoadImageCPU(image_path, dynamic_input_dims);
std::vector<void*> binding_buffer(engine->getNbBindings());
auto input_size = GetBytes(dynamic_input_dims, engine->getBindingDataType(input_idx));
void* input_gpu{nullptr};
CHECK(cudaMalloc(&input_gpu, input_size));

CHECK(cudaMemcpy(input_gpu, cpu_fp32_image.get(), input_size, cudaMemcpyHostToDevice));

auto context = UniquePtr<iluvatar::IExecutionContext>(engine->createExecutionContext());

context->setBindingDimensions(input_idx, dynamic_input_dims);
auto context_input_dims = context->getBindingDimensions(input_idx);
auto context_output_dims = context->getBindingDimensions(output_idx);
```

```
auto output_size = GetBytes(context_output_dims, engine->getBindingDataType(output_idx));
std::shared_ptr<float> cpu_output(new float[output_size / sizeof(float)], ArrayDeleter());
std::shared_ptr<float> cpu_output_2(new float[output_size / sizeof(float)], ArrayDeleter());
void* output_gpu{nullptr};
CHECK(cudaMalloc(&output_gpu, output_size));

binding_buffer.at(input_idx) = input_gpu;
binding_buffer.at(output_idx) = output_gpu;

auto status = context->executeV2(binding_buffer.data());
CHECK(cudaMemcpy(cpu_output.get(), output_gpu, output_size, cudaMemcpyDeviceToHost));
GetClassificationResult(cpu_output.get(), 1000, 5);
CHECK(cudaFree(input_gpu));
CHECK(cudaFree(output_gpu));

// Prepare image with small size
iluvatar::Dims dynamic_input_dims_2{4, {1, 3, 196, 196}};
auto cpu_fp32_image_2 = LoadImageCPU(image_path_2, dynamic_input_dims_2);
auto input_size_2 = GetBytes(dynamic_input_dims_2, engine->getBindingDataType(input_idx));
void* input_gpu_2{nullptr};
CHECK(cudaMalloc(&input_gpu_2, input_size_2));
CHECK(cudaMemcpy(input_gpu_2, cpu_fp32_image_2.get(), input_size_2, cudaMemcpyHostToDevice));

context->setBindingDimensions(input_idx, dynamic_input_dims_2);

auto context_output_dims_2 = context->getBindingDimensions(output_idx);

auto output_size_2 = GetBytes(context_output_dims_2, engine->getBindingDataType(output_idx));
void* output_gpu_2{nullptr};
CHECK(cudaMalloc(&output_gpu_2, output_size_2));
std::vector<void*> binding_buffer_2(engine->getNbBindings());
binding_buffer_2.at(input_idx) = input_gpu_2;
binding_buffer_2.at(output_idx) = output_gpu_2;
auto status_2 = context->executeV2(binding_buffer_2.data());
CHECK(cudaMemcpy(cpu_output_2.get(), output_gpu_2, output_size_2, cudaMemcpyDeviceToHost));
GetClassificationResult(cpu_output_2.get(), 1000, 5);
CHECK(cudaFree(input_gpu_2));
CHECK(cudaFree(output_gpu_2));
```

需要注意的是，含有动态维度的模型，动态的维度值使用"-1" 表示。如果访问 networkDefinition 中的 Tensor 维度值，例如，batch size 是动态的，就会返回，输出为-1 x 3 x 224 x 224。经过形状推导之后，输出的维度是-1 x 1000。

5.5.2 动态形状推理

使用动态形状推理，相比固定形状推理，需要增加如下步骤：

5.5.2.1 1. 创建 optimization profile

示例代码标识了动态的范围：

```
auto profile = builder->createOptimizationProfile();
profile->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kMIN, iluvatar::Dims{4,
 ↪  {1, 3, 112, 112}});
profile->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kOPT, iluvatar::Dims{4,
 ↪  {1, 3, 224, 224}});
profile->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kMAX, iluvatar::Dims{4,
 ↪  {1, 3, 448, 448}});
config->addOptimizationProfile(profile);
```

5.5.2.2 2. 在运行前设置输入维度

此时要求明确的输入维度，不能含有-1，例如：

```
iluvatar::Dims dynamic_input_dims{4, {1, 3, 224, 224}};
context->setBindingDimensions(input_idx, dynamic_input_dims_2);
```

在维度不变的情况下，可以直接调用 executeV2 或 enqueueV2。一旦维度发生变化，就需要调用一次 setBindingDimensions 方法。在上述示例中，分类网络分别对以下两种尺寸执行了推理：

- 1 x 3 x 224 x 224
- 1 x 3 x 196 x 196

5.6 教程 6：ResNet18 分类网络动态形状推理下的多运行环境

本章节将展示在多个线程中，使用动态形状共享一个模型的权重的推理方案。

5.6.1 示例代码总览

```
std::string dir_path("/home/data/cls/");
std::string image_path(dir_path + "kitten_224.bmp");
std::string image_path_2(dir_path + "robin_224.bmp");
std::string model_path(dir_path + "resnet18-all-dynamic.onnx");
std::string quant_file(dir_path + "quantized_resnet18_shape_opset11.json");
```

```
std::string input_name("input");
std::string output_name("output");

Logger logger(iluvatar::ILogger::Severity::kVERBOSE);
auto builder = UniquePtr<iluvatar::IBuilder>(iluvatar::createInferBuilder(logger));
const auto explicitBatch = 1U <<
 ↪  static_cast<uint32_t>(iluvatar::NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
auto network = UniquePtr<iluvatar::INetworkDefinition>(builder->createNetworkV2(explicitBatch));

auto config = UniquePtr<iluvatar::IBuilderConfig>(builder->createBuilderConfig());
config->setFlag(iluvatar::BuilderFlag::kINT8);
auto profile = builder->createOptimizationProfile();
profile->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kMIN, iluvatar::Dims{4,
 ↪  {1, 3, 112, 112}});
profile->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kOPT, iluvatar::Dims{4,
 ↪  {1, 3, 224, 224}});
profile->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kMAX, iluvatar::Dims{4,
 ↪  {1, 3, 448, 448}});
config->addOptimizationProfile(profile);
// Add second profile
auto profile_2 = builder->createOptimizationProfile();
profile_2->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kMIN,
                         iluvatar::Dims{4, {1, 3, 112, 112}});
profile_2->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kOPT,
                         iluvatar::Dims{4, {1, 3, 224, 224}});
profile_2->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kMAX,
                         iluvatar::Dims{4, {1, 3, 448, 448}});
config->addOptimizationProfile(profile_2);
// Add third profile, same as first
config->addOptimizationProfile(profile);
auto parser = UniquePtr<iluvatar::IParser>(iluvatar::createParser(*network, logger));
auto parsed =
    parser->parseFromFile(model_path.c_str(), static_cast<int>(logger.getReportableSeverity()),
 ↪  quant_file.c_str());

iluvatar::Dims inputDims = network->getInput(0)->getDimensions();

iluvatar::Dims outputDims = network->getOutput(0)->getDimensions();

UniquePtr<iluvatar::IHostMemory> plan{builder->buildSerializedNetwork(*network, *config)};

UniquePtr<iluvatar::IRuntime> runtime{iluvatar::createInferRuntime(logger)};

auto engine =
 ↪  std::shared_ptr<iluvatar::ICudaEngine>(runtime->deserializeCudaEngine(plan->data(),
 ↪  plan->size()),
```

```
                                                ObjectDeleter());

auto input_idx = engine->getBindingIndex(input_name.c_str());
auto output_idx = engine->getBindingIndex(output_name.c_str());
int32_t profile_index = 1;

iluvatar::Dims dynamic_input_dims{4, {1, 3, 224, 224}};
auto cpu_fp32_image = LoadImageCPU(image_path, dynamic_input_dims);
auto num_binding = engine->getNbBindings();
auto num_profile = engine->getNbOptimizationProfiles();
auto binding_cell_size = num_binding / num_profile;
std::vector<void*> binding_buffer(num_binding);
auto input_size = GetBytes(dynamic_input_dims, engine->getBindingDataType(input_idx));
void* input_gpu{nullptr};
CHECK(cudaMalloc(&input_gpu, input_size));

CHECK(cudaMemcpy(input_gpu, cpu_fp32_image.get(), input_size, cudaMemcpyHostToDevice));

auto context = UniquePtr<iluvatar::IExecutionContext>(engine->createExecutionContext());

context->setBindingDimensions(input_idx, dynamic_input_dims);
auto context_input_dims = context->getBindingDimensions(input_idx);
auto context_output_dims = context->getBindingDimensions(output_idx);

auto output_size = GetBytes(context_output_dims, engine->getBindingDataType(output_idx));
std::shared_ptr<float> cpu_output(new float[output_size / sizeof(float)], ArrayDeleter());
std::shared_ptr<float> cpu_output_2(new float[output_size / sizeof(float)], ArrayDeleter());
void* output_gpu{nullptr};
CHECK(cudaMalloc(&output_gpu, output_size));

binding_buffer.at(input_idx) = input_gpu;
binding_buffer.at(output_idx) = output_gpu;

auto status = context->executeV2(binding_buffer.data());
CHECK(cudaMemcpy(cpu_output.get(), output_gpu, output_size, cudaMemcpyDeviceToHost));
GetClassificationResult(cpu_output.get(), 1000, 5);
CHECK(cudaFree(input_gpu));
CHECK(cudaFree(output_gpu));

// Prepare image with small size
auto context_2 = UniquePtr<iluvatar::IExecutionContext>(engine->createExecutionContext());
auto stream = makeCudaStream();
context_2->setOptimizationProfileAsync(1, *stream);
// Switch optimization profile by set profile index, context resource will be updated
//    context_2->setOptimizationProfileAsync(2, *stream);
```

```
iluvatar::Dims dynamic_input_dims_2{4, {1, 3, 224, 224}};
auto cpu_fp32_image_2 = LoadImageCPU(image_path_2, dynamic_input_dims_2);
auto input_size_2 = GetBytes(dynamic_input_dims_2, engine->getBindingDataType(input_idx +
 ↪  binding_cell_size));
void* input_gpu_2{nullptr};
CHECK(cudaMalloc(&input_gpu_2, input_size_2));
CHECK(cudaMemcpy(input_gpu_2, cpu_fp32_image_2.get(), input_size_2, cudaMemcpyHostToDevice));

context_2->setBindingDimensions(input_idx, dynamic_input_dims_2);

auto context_output_dims_2 = context_2->getBindingDimensions(output_idx);

auto output_size_2 = GetBytes(context_output_dims_2, engine->getBindingDataType(output_idx));
void* output_gpu_2{nullptr};
CHECK(cudaMalloc(&output_gpu_2, output_size_2));
binding_buffer.at(input_idx + binding_cell_size) = input_gpu_2;
binding_buffer.at(output_idx + binding_cell_size) = output_gpu_2;
auto status_2 = context_2->executeV2(binding_buffer.data());
CHECK(cudaMemcpy(cpu_output_2.get(), output_gpu_2, output_size_2, cudaMemcpyDeviceToHost));
GetClassificationResult(cpu_output_2.get(), 1000, 5);
CHECK(cudaFree(input_gpu_2));
CHECK(cudaFree(output_gpu_2));
```

5.6.2 示例说明

需要注意的是，如果创建多个 context，则每个 context 需要对应一个 optimization profile。optimization profile 的数量决定了 context 可以创建的上限。每个 profile 都有一组单独的索引用于应对输入和输出数据的位置。例如，这里创建了 3 个 profile，第 1 个和第 3 个是相同的：

```
auto profile = builder->createOptimizationProfile();
profile->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kMIN, iluvatar::Dims{4,
 ↪  {1, 3, 112, 112}});
profile->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kOPT, iluvatar::Dims{4,
 ↪  {1, 3, 224, 224}});
profile->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kMAX, iluvatar::Dims{4,
 ↪  {1, 3, 448, 448}});
config->addOptimizationProfile(profile);
// Add second profile
auto profile_2 = builder->createOptimizationProfile();
profile_2->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kMIN,
                         iluvatar::Dims{4, {1, 3, 112, 112}});
profile_2->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kOPT,
                         iluvatar::Dims{4, {1, 3, 224, 224}});
profile_2->setDimensions(input_name.c_str(), iluvatar::OptProfileSelector::kMAX,
```

```
                        iluvatar::Dims{4, {1, 3, 448, 448}});
config->addOptimizationProfile(profile_2);
// Add third profile, same as first
config->addOptimizationProfile(profile);
```

在创建 context 时，默认使用的是 0 号 profile。如果创建更多的 context，需要显式地指定 profile 序号：

```
auto stream = makeCudaStream();
context_2->setOptimizationProfileAsync(1, *stream);
```

Important

如果某一个 profile 已经被某个 context 占用，就无法被别的 context 共享。

需要注意 context 的 binding 索引，除了 0 号 profile 绑定的 context，其它的 context 在使用时都需要加上 profile 的偏移量。例如，第 2 个 context 的输入数据为：

```
binding_buffer.at(input_idx + binding_cell_size) = input_gpu_2;
binding_buffer.at(output_idx + binding_cell_size) = output_gpu_2;
```

6 使用 IxRT 推理引擎进行模型推理最佳实践

本章节旨在提供使用 IxRT 推理引擎进行模型推理的最佳实践指导，帮助用户可以更好地利用 IxRT 的模型推理功能和 IxRT-EXEC 的算子检验及推理性能测试功能。

我们推荐您使用如下步骤进行 IxRT 模型推理：

步骤 1 获取 ONNX 模型

步骤 2 使用 IxRT-EXEC 工具快速检验算子支持情况和推理性能

步骤 3 (For INT8 推理) 量化

步骤 4 构建引擎文件

步骤 5 加载引擎文件并执行推理

Note

- INT8 推理时必须进行量化，FP16 推理时不需要进行量化。
- 步骤 4 和步骤 5 使用的是 IxRT 的核心功能。

6.1 步骤 1 获取 ONNX 模型

IxRT 使用开放神经网络交换 (Open Neural Network Exchange, ONNX) 格式模型作为输入，不同训练框架所训练出的模型均可转为 ONNX 格式。以 PyTorch 为例，使用如下代码将 ResNet18 模型转换为 ONNX 格式：

```
$ pip3 install onnx-simplifier matplotlib [-i http://pypi.douban.com/simple --trusted-host
↪  pypi.douban.com] # 网络不好时，可使用 pip 镜像源
$ mkdir -p models # 创建 models 目录用于存放后续导出的 ONNX 文件
```

```
# 引入必备库
from torch.autograd import Variable
import torch.onnx
import torchvision

# 将模型导出为 ONNX 格式
dummy_input = Variable(torch.randn(1, 3, 224, 224))
model = torchvision.models.resnet18(pretrained=True)
torch.onnx.export(model, dummy_input, "models/resnet18.onnx")
```

6.2 步骤 2 使用 IxRT-EXEC 工具快速检验算子支持情况和推理性能

IxRT-EXEC 工具的使用方法和参数说明请参考"IxRT-EXEC 自动化模型推理工具使用指南"。

测试模型推理性能，无需编程，您只需要运行一行命令即可。下面是模型推理示例：

6.2.1 示例 1：对 ResNet18 进行 FP16 推理

该 ONNX 模型经由 PyTorch 导出，使用 FP16 精度推理。不选择 **--precision** 时，参数默认是 fp16，等价于 **ixrtexec --onnx models/resnet18.onnx --precision float16** 命令。

执行如下命令进行模型推理：

```
$ ixrtexec --onnx models/resnet18.onnx
```

6.2.2 示例 2：对 ResNet18 进行 INT8 推理

该 ONNX 模型经由 PyTorch 导出，使用 INT8 精度推理。

执行如下命令进行模型推理：

```
$ ixrtexec --onnx models/resnet18.onnx --precision int8
```

6.2.3 示例 3：对 YOLOv7 进行 FP16 推理

1. 获取 `unit_test_yolov7_bin.zip` YOLOv7 推理测试包。
2. 将 `unit_test_yolov7_bin.zip` 移动至 `models` 目录。
3. 执行如下命令进行模型推理：

```
$ cd models
$ unzip unit_test_yolov7_bin.zip
$ ixrtexec --onnx models/unit_test_yolov7_bin/yolov7_without_decoder.onnx
```

6.2.4 示例 4：对 YOLOv7 进行 INT8 推理

继续使用 `unit_test_yolov7_bin.zip` 测试包进行推理性能测试，执行如下命令进行模型推理：

```
$ ixrtexec --onnx models/unit_test_yolov7_bin/yolov7_without_decoder.onnx --precision int8
```

6.3 步骤 3 (For INT8 推理) 量化

IxRT 提供的量化工具，支持对输入的 ONNX 文件进行量化并最终加载用于模型推理。通过该量化工具，可构建用于推理的 IxRT 运行时并直接进行模型推理。

IxRT 提供静态量化方案，支持直接量化 PyTorch 的 `nn.Module` 文件和 ONNX 文件。

下面介绍静态量化的使用方法。

首先，引用 IxRT 静态量化工具：

```
from tensorrt.deploy import static_quantize
```

静态量化的 static_quantize 接口定义如下：

```
def static_quantize(
    model: [str, torch.nn.Module],
    calibration_dataloader: torch.utils.data.DataLoader,
    observer: str = "minmax",
    disable_bias_correction: bool = False,
    analyze: bool = False,
    save_quant_onnx_path=None,
    save_quant_params_path=None,
    data_preprocess=None,
    device=0,
    disable_quant_names: list = None,
    disable_quant_types: list = None,
    operator_config_by_type: Dict[str, QuantOperatorObserverConfig] = None,
    operator_config_by_name: Dict[str, QuantOperatorObserverConfig] = None,
    passes: Union[str, List, BasePass] = "default",
    **kwargs,
):
    ...
```

- 接口参数说明：
  - model：ONNX 文件路径或 PyTorch 的 nn.Module
  - calibration_dataloader：校准数据集 Dataloader
  - observer：量化策略，可选项有：
    - minmax：默认选项，直接使用 Tensor 的最大值和最小值 (推荐在 Tensor 的数据较少以及校准数据集较小的情况下使用)
    - percentile：使用 Tensor 的百分位数上的数作为最值，默认使用 99.99% (通常能取得不错的结果，但计算百分数过程较慢)
    - hist_percentile：使用 Tensor 在统计直方图上的百分位数作为最值，默认使用 99.99% (该方法在"percentile" 上使用了直方图，通常能取得较好的结果，速度也较快)
    - entropy：使用 KL 散度去最小化浮点数类型和量化类型之间的数据分布 (该方法会在生成的直方图上进行搜索，找出损失最小的区间，但速度较慢)
    - ema：使用滑动平均去计算最值 (不推荐)
  - disable_bias_correction：禁用偏置修正，若为 True，则不会修改偏置，否则修改部分偏置
  - analyze：量化分析
  - save_quant_onnx_path：保存量化后的 ONNX 文件到本地，若为 NONE，则不保存
  - save_quant_params_path：保存量化参数到本地，若为 NONE，则不保存
  - data_preprocess：数据预处理方法
  - device：运行的设备号
  - disable_quant_names：禁止量化的算子名称
  - disable_quant_types：禁止量化的算子类型
- 返回：量化后的中间表示，类型为 tensorrt.deploy.ir.graph.Graph

- 输出：若 save_quant_onnx_path 和 save_quant_params_path 不为空，则保存到相应路径下

以下是使用示例,量化后的模型文件为 results/quantized_resnet18.onnx 和 results/quantized_resnet18_params.json：

```
$ mkdir -p results # 创建 results 目录用于存放后续生成的模型文件
```

```
# 生成随机的校准数据集
from tensorrt.cli import generate_dummy_calibration_loader
loader = generate_dummy_calibration_loader([('input', [1, 3, 224, 224])], dict(input="float32"))

# 使用静态量化
from tensorrt.deploy import static_quantize
quant_onnx_path = "results/quantized_resnet18.onnx"
quant_param_path = "results/quantized_resnet18_params.json"
resnet18_ir_graph = static_quantize("models/resnet18.onnx",
                calibration_dataloader=loader,
                save_quant_onnx_path=quant_onnx_path,
                save_quant_params_path=quant_param_path)
```

6.4 步骤 4 构建引擎文件

构建引擎文件代码如下：

```
import os
import sys

import tensorrt

quant_onnx_path = "results/quantized_resnet18.onnx"
quant_param_path = "results/quantized_resnet18_params.json"
```

下面分别提供构建 INT8 推理的引擎文件和 FP16 推理的引擎文件的代码示例：

构建 INT8 引擎文件

```
# set config
IXRT_LOGGER = tensorrt.Logger(tensorrt.Logger.WARNING)
builder = tensorrt.Builder(IXRT_LOGGER)
EXPLICIT_BATCH = 1 << (int)(tensorrt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
network = builder.create_network(EXPLICIT_BATCH)
build_config = builder.create_builder_config()
build_config.set_flag(tensorrt.BuilderFlag.INT8)
parser = tensorrt.OnnxParser(network, IXRT_LOGGER)
parser.parse_from_files(quant_onnx_path, quant_param_path)
```

```
# build engine
plan = builder.build_serialized_network(network, build_config)
with open("results/resnet18_int8.engine", "wb") as f:
    f.write(plan)

print("Engine has been saved at", "results/resnet18_int8.engine")
```

构建 FP16 引擎文件

```
# set config
onnx_model = "models/resnet18.onnx"
IXRT_LOGGER = tensorrt.Logger(tensorrt.Logger.WARNING)
builder = tensorrt.Builder(IXRT_LOGGER)
EXPLICIT_BATCH = 1 << (int)(tensorrt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
network = builder.create_network(EXPLICIT_BATCH)
build_config = builder.create_builder_config()
build_config.set_flag(tensorrt.BuilderFlag.FP16)
parser = tensorrt.OnnxParser(network, IXRT_LOGGER)
parser.parse_from_file(onnx_model)

# build engine
plan = builder.build_serialized_network(network, build_config)
with open("results/resnet18_fp16.engine", "wb") as f:
    f.write(plan)

print("Engine has been saved at", "results/resnet18_fp16.engine")
```

6.5 步骤 5 加载引擎文件并执行推理

调用推理接口代码如下：

```
import cv2
import numpy as np
import torch
from tensorrt.utils import topk
import pycuda.autoinit
import pycuda.driver as cuda


def show_cls_result(result, k=5):
    data = result
    from imagenet_labels import labels

    vals, idxs = topk(data, k, axis=1)
```

```
    # n_samples = len(idxs)
    n_samples = 1
    for i in range(n_samples):
        idx0 = idxs[i]
        val0 = vals[i]
        print("---Python inference result---")
        for i, (val, idx) in enumerate(zip(val0, idx0)):
            print(f"Top {i+1}:   {val}  {labels[idx]}")


def check_cls_result(result, answer, k=5):
    data = result[0][1]
    from imagenet_labels import labels

    vals, idxs = topk(data, k, axis=1)
    idx0 = idxs[0]
    val0 = vals[0]
    answer_ids = []
    answer_names = []
    for _, (val, idx) in enumerate(zip(val0, idx0)):
        answer_ids.append(val)
        answer_names.append(labels[idx])
    if answer not in answer_names:
        return False
    else:
        return True

def setup_io_bindings(engine):
    # Setup I/O bindings
    inputs = []
    outputs = []
    allocations = []
    for i in range(engine.num_bindings):
        is_input = False
        if engine.binding_is_input(i):
            is_input = True
        name = engine.get_binding_name(i)
        dtype = engine.get_binding_dtype(i)
        shape = engine.get_binding_shape(i)
        if is_input:
            batch_size = shape[0]
        size = np.dtype(tensorrt.nptype(dtype)).itemsize
        for s in shape:
            size *= s
        allocation = cuda.mem_alloc(size)
        binding = {
```

```
            "index": i,
            "name": name,
            "dtype": np.dtype(tensorrt.nptype(dtype)),
            "shape": list(shape),
            "allocation": allocation,
        }
        allocations.append(allocation)
        if engine.binding_is_input(i):
            inputs.append(binding)
        else:
            outputs.append(binding)
    return inputs, outputs, allocations

def run_load_from_cpu_v1(runtime, engine, file, print_result=True):
    context = engine.create_execution_context()
    inputs, outputs, allocations = setup_io_bindings(engine)
    input_io_buffers = []
    output_io_buffers = []
    # Prepare the output data
    output = np.zeros(outputs[0]["shape"], outputs[0]["dtype"])

    data_batch = (
        np.flip(cv2.imread(file) / 255.0, axis=2)
        .astype("float32")
        .transpose(2, 0, 1)
    )
    data_batch = data_batch.reshape(1, *data_batch.shape)
    data_batch = np.ascontiguousarray(data_batch)

    # Process I/O and execute the network
    cuda.memcpy_htod(inputs[0]["allocation"], data_batch)
    context.execute_v2(allocations)
    cuda.memcpy_dtoh(output, outputs[0]["allocation"])

    show_cls_result(output)

    # Free Gpu Memory
    inputs[0]["allocation"].free()
    outputs[0]["allocation"].free()
```

下面分别提供加载 INT8 引擎文件并推理和加载 FP16 引擎文件并推理的代码示例：

加载 INT8 引擎文件并推理

```
# 加载引擎文件
runtimei8 = tensorrt.Runtime(IXRT_LOGGER)
```

```
with open("results/resnet18_int8.engine", "rb") as f:
    enginei8 = runtimei8.deserialize_cuda_engine(f.read())
```

现在，请您下载一只小猫的图片，对 ResNet18 推理进行测试。

Note

注意，ResNet18 推理需要 224x224，因此您需要提前将图片缩放到该尺寸。

```
# 创建 imgs 目录并通过 URL 下载小猫图片到 imgs 目录下
$ declare -x http_proxy="http://10.101.3.14:3128"
$ declare -x https_proxy="http://10.101.3.14:3128"

$ mkdir -p imgs
$ wget -q https://raw.githubusercontent.com/dmlc/mxnet.js/main/data/cat.png -P imgs
```

```
# 加载图片缩放并显示
import cv2
import matplotlib.pyplot as plt
im_path = "imgs/cat.png"
im = cv2.imread(im_path)
im = cv2.resize(im, (224, 224))

plt.imshow(im[...,::-1])
plt.show()
_ = cv2.imwrite(im_path, im)
```

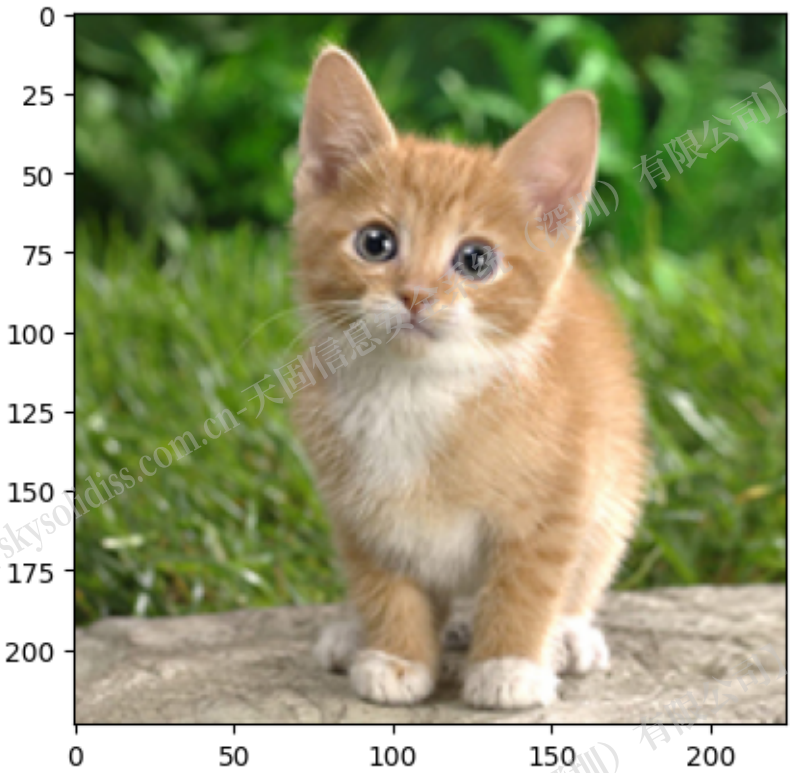

Figure 1: 缩放后的测试图片

```
# 执行推理
_ = run_load_from_cpu_v1(runtimei8, enginei8, im_path)

# 推理结果
---Python inference result---
Top 1:   4.405557632446289  n03223299 doormat, welcome mat
Top 2:   4.405557632446289  n02123045 tabby, tabby cat
Top 3:   4.405557632446289  n02119022 red fox, Vulpes vulpes
Top 4:   4.405557632446289  n02119789 kit fox, Vulpes macrotis
Top 5:   4.405557632446289  n15075141 toilet tissue, toilet paper, bathroom tissue
```

加载 FP16 引擎文件并推理

```
# 加载引擎文件
runtimef16 = tensorrt.Runtime(IXRT_LOGGER)
with open("results/resnet18_fp16.engine", "rb") as f:
    enginefp16 = runtimef16.deserialize_cuda_engine(f.read())

# 执行推理
_ = run_load_from_cpu_v1(runtimef16, enginefp16, im_path)

# 推理结果
```

```
---Python inference result---
Top 1:   9.8046875  n02123159 tiger cat
Top 2:   9.6484375  n02123394 Persian cat
Top 3:   9.40625  n02123045 tabby, tabby cat
Top 4:   9.109375  n02124075 Egyptian cat
Top 5:   8.921875  n02127052 lynx, catamount
```

7 使用 IxRT 推理引擎对模型进行 INT8 推理最佳实践

本章节将详细介绍量化相关的知识以及指导您正确使用 IxRT 推理引擎对模型进行 INT8 推理。

7.1 关于量化

在进行推理操作前，请您先参考本小节提供的内容了解什么是量化。您将从以下方面获取量化相关信息：

- 量化相关知识
- 量化的优点
- INT8 量化推理流程
- IxRT 量化工具

7.1.1 量化相关知识

对于一个函数 y = w * x，其中 y, w, x 为 FP32 类型，为了使用低精度类型来计算出 y，您可以将 y 的计算定义为：y 约等于 dequantize(quantize(w) * quantize(x))，其中 quantize (量化) 是将高精度类型量化为低精度类型，dequantize (反量化) 是将低精度类型还原到高精度类型。

量化通常使用基于范围映射的方法实现。该方法分为非对称量化 和对称量化 两种映射模式，非对称量化 (Affine Quantization) 和对称量化 (Scale Quantization) 的区间映射如下图所示：

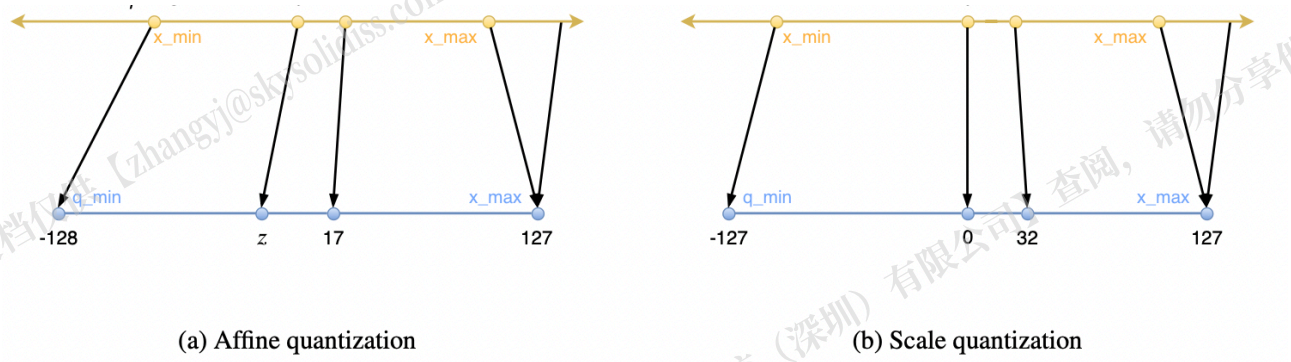

Figure 2: 非对称量化和对称量化的区间映射图

其中，[x_min, x_max] 作为真实数值的范围；[q_min, q_max] 作为需要进行量化的目标区间。比如，如果需要量化为 INT8 类型，则可以表示的区间为 [-128, 127]。

7.1.1.1 非对称量化

在基于范围映射的量化方法中，

- 量化函数：quantize 被定义为 quantize(x, s, z) = clip(round(s*x + z), q_min, q_max)，其中 s 为缩放系数 (scale)，z 为中心点位置 (zero-point)，clip 是将值裁剪至 [q_min, q_max] 的区间内。
- 反量化函数：dequantize 被定义为 dequantize(x_q, s, z) = (x_q - z) / s。

其中，量化和反量化的 s 和 z 的计算方式分别为：

- s = (q_max - q_min) / (x_max - x_min)
- z = q_min - round(x_min * s)

7.1.1.2 对称量化

对称量化是非对称量化的一种特例，即：

- z = 0
- s = (q_max - q_min) / (2 * max(| x_min | , | x_max |))

7.1.2 量化的优点

- 计算上：低精度的计算效率远高于高精度的计算效率，比如 INT8 的计算效率要优于 FP32 的计算效率
- 内存压力上：由于低精度所需的比特位要小于高精度类型的所需，因此当模型整体上采用低精度类型的存储时，所消耗的内存也将得到相应的减少
- 内存带宽上：低精度的数据格式可以减少内存带宽压力，提高带宽有限计算的性能

7.1.3 INT8 量化推理流程

对于一个 Float32 的 ONNX 模型，希望使用 INT8 进行推理，需要经历以下主要步骤：

1. 使用 IxRT-EXEC 进行支持情况和性能快速验证
2. 编写量化脚本进行量化
3. 基于 IxRT API 编写引擎生成和推理程序

建议您使用 IxRT deploy 工具提供的量化工具进行量化，从而获得在天数智芯加速卡上高效的推理体验。当然，您也可以使用第三方量化工具进行量化，并使用 IxRT 推理引擎在天数智芯加速卡上进行推理。

IxRT 使用 QDQ 格式的 onnx 文件执行 INT8 推理，因此 **请您确保使用的 onnx 文件是 QDQ 格式**。详情请参考基于 QDQ ONNX 的 INT8 推理。

7.1.4 IxRT 量化工具

IxRT 量化工具是 deploy 工具提供的功能之一，下表总结了 IxRT 量化工具的支持情况：

量化模式	导出平台
PTQ 和 QAT 的静态量化	QDQ 格式、PPQ 格式

训练后量化 (Post-training Quantization, PTQ) 又被称为隐式量化，这种模式下用户控制哪些算子量化的权限有限，用户可以通过校准 (Calibration) 和直接设置 Dynamic Range 两种方式来实现。这种模式主要的目的是加速算子运算，因此适用的是 Conv、ConvTranspose、Gemm、FullyConnected、Attention 这类的 Compute-bound 算子。

训练感知量化 (Quantization Aware Training, QAT) 又被称为显式量化，这种模式下用户有更高的自由度来控制 Q/DQ 的边界，因此理论上可以适用于所有算子。该模式在保证性能的同时可以更好地用于精度调优。

7.1.4.1 训练后量化 PTQ

首先引用 IxRT 静态工具：

```
from tensorrt.deploy import static_quantize
```

静态量化 `static_quantize` 接口定义如下：

```
def static_quantize(
    model: [str, torch.nn.Module],
    calibration_dataloader: torch.utils.data.DataLoader,
    observer: str = "minmax",
    disable_bias_correction: bool = False,
    analyze: bool = False,
    save_quant_onnx_path=None,
    save_quant_params_path=None,
    data_preprocess=None,
    quant_format="ppq",
    device=0,
    disable_quant_names: list = None,
    disable_quant_types: list = None,
    operator_config_by_type: Dict[str, QuantOperatorObserverConfig] = None,
    operator_config_by_name: Dict[str, QuantOperatorObserverConfig] = None,
    passes: Union[str, List, BasePass] = "default",
    **kwargs,
):
    ...
```

- 接口参数说明：
  - `model`：ONNX 文件路径或 PyTorch 的 `nn.Module`
  - `calibration_dataloader`：校准数据集 Dataloader
  - `observer`：量化策略，可选项有：
    * `minmax`：默认选项，直接使用 Tensor 的最大值和最小值 (推荐在 Tensor 的数据较少以及校准数据集较小的情况下使用)
    * `percentile`：使用 Tensor 的百分位数上的数作为最值，默认使用 99.99% (通常能取得不错的结果，但计算百分数过程较慢)
    * `hist_percentile`：使用 Tensor 在统计直方图上的百分位数作为最值，默认使用 99.99% (该方法在”percentile” 上使用了直方图，通常能取得较好的结果，速度也较快)

    * entropy：使用 KL 散度去最小化浮点数类型和量化类型之间的数据分布 (该方法会在生成的直方图上进行搜索，找出损失最小的区间，但速度较慢)
    * ema：使用滑动平均去计算最值 (不推荐)
  - disable_bias_correction：禁用偏置修正，若为 True，则不会修改偏置，否则修改部分偏置
  - analyze：量化分析
  - save_quant_onnx_path：保存量化后的 ONNX 文件到本地，若为 NONE，则不保存
  - save_quant_params_path：保存量化参数到本地，若为 NONE，则不保存
  - data_preprocess：数据预处理方法
  - quant_format：量化格式，可选项有：
    * qdq：以 QDQ 格式到处量化的 onnx 文件
    * ppq：以 PPQ 风格的量化表格式导出量化参数
  - device：运行的设备号
  - disable_quant_names：禁止量化的算子名称
  - disable_quant_types：禁止量化的算子类型
  - operator_config_by_type：通过算子类型来设置对算子的量化方法
  - operator_config_by_name：通过算子名称来设置对算子的量化方法
  - passes：量化前对 Graph 进行处理的 Pass
- 返回：量化后的中间表示，类型为 tensorrt.deploy.ir.graph.Graph
- 输出：若 save_quant_onnx_path 和 save_quant_params_path 不为空，则保存到相应路径下

使用样例 1

```
from tensorrt.deploy.api import *

resnet18_model = "resnet18.onnx"
resnet18_ir_graph = static_quantize(
    model=resnet18_model,
    save_quant_onnx_path="quantized_resnet18.onnx",
    save_quant_params_path="quantized_resnet18_params.json"
)
```

使用样例 2：设置算子的量化方法

```
from tensorrt.deploy.api import *
from tensorrt.deploy.ir.operator_type import OperatorType as OP

static_quantize(
    model="resnet18.onnx",
    save_quant_onnx_path="quantized_resnet18.onnx",
    save_quant_params_path="quantized_resnet18_params.json",
    operator_config_by_type={OP.GEMM: QuantOperatorObserverConfig(
        activation=create_observer("minmax", quant_policy=QuantPolicy(QuantGrain.PER_TENSOR)),
        weight=create_observer("minmax", quant_policy=QuantPolicy(QuantGrain.PER_TENSOR)),
    )}
)
```

7.1.4.2 训练感知量化 QAT

在 QAT 中，IxRT deploy 会将计算图转为 `torch.nn.Module` 的实例，从而可以使用 PyTorch 对模型进行微调。如果导出模型的训练框架来自于 PyTorch，则只需要在之前的训练代码上加上如下函数即可：

```
def convert_to_qat(
    graph: Graph,
    calibration_dataloader: torch.utils.data.DataLoader,
    qconfig: QuantizerConfig = None,
    preprocess: Callable=None,
    executor: BaseExecutor=None
) -> torch.nn.Module:
    ...
```

- 参数说明如下：
  - `graph tensorrt.deploy.ir.graph.Graph`：IxRT deploy 中的计算图
  - `calibration_dataloader`：校准数据集
  - `qconfig`：量化器的配置
  - `preprocess`：数据预处理方法
  - `executor`：计算图的执行器
- 返回值：

  返回 `torch.nn.Module` 的实例

Note

- 对于 TorchVision 中的分类模型进行 QAT，请参考 `qat_classification.py` 脚本 (请向您的应用工程师获取该脚本)。
- 现阶段，QAT 暂不支持在多卡上进行微调。
- 为了对 ONNX 进行微调，可以使用 `create_source(model: str, example_inputs)` 来创建 Graph 的一个实例。
- 在对模型进行微调完成之后，需要使用静态量化器 (Post Training Static Quantizer) 来完成对模型的整体量化。

7.1.4.3 量化的粒度选择

在进行量化时，有很多种粒度，由粗到细为：per tensor > per kernel, per channel > per row, per col > per element，其中，

- per row 和 per col 主要针对的是二维矩阵
- per channel 主要针对的是三维的 Tensor，比如图像等
- per kernel 主要针对的是卷积中的卷积核，在输出通道维度上的计算
- 量化的粒度越细，计算的复杂度就会变得越大
- IxRT 推理引擎目前支持 per tensor 和 per channel 两种模式

7.1.4.4 定点算法选择

定点算法的目的是计算出 x_min 和 x_max，不同的定点算法对量化后准确度的影响是非常大的，IxRT deploy 支持的定点算法有：

- Minmax：直接使用 Tensor 的最小值和最大值 (推荐在 Tensor 的数据比较少，以及校准数据集比较小的情况下使用)
- EMA：使用 KL 散度去最小化 浮点数类型和 量化类型之间的数据分布，TensorRT 默认使用这种方式 (该方法会在生成的直方图上进行搜索，找出损失最小的区间，速度较慢)
- Percentile：使用 Tensor 的百分位数上的数作为最值，默认使用 99.99% (通常能取得比较不错的结果，但计算百分数的过程较慢)
- BasedHistogramPercentile：使用 Tensor 在统计直方图上的百分位数作为最值，默认使用 99.99% (该方法在"Percentile" 方法上使用了直方图，通常也能取得比较好的结果，速度上也会更快)

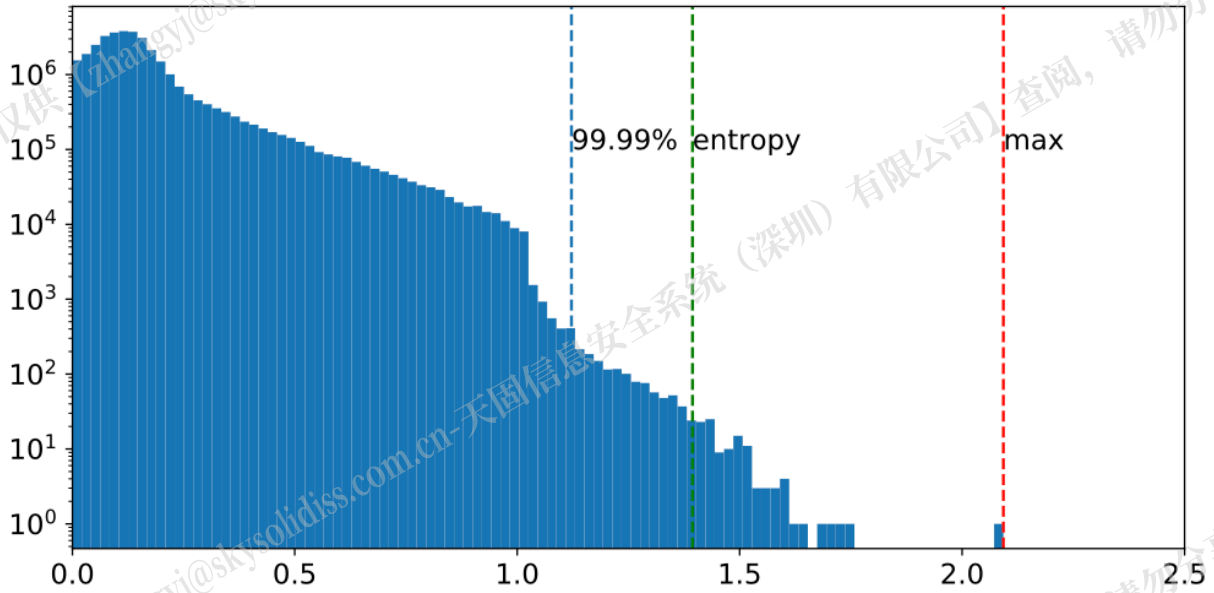

Figure 3: 定点算法示意图

Note

- 在使用 hist_percentile 和 percentile 时，可以使用不同的百分位数 (percentile)，默认是 99.99，可以设置为 99.9, 99.99, 99.999 等。
- 在使用 entropy 时，可以设置 start_bin 为 128, 256, 512, ... num_bins，也可以设置 num_bins 为 2048, 4096, 8192。
- 在 IxRT deploy 中设置定点算法的方法是：

```
from tensorrt.depoly.api import *
# 其中名字为定点算法的名字，可参考 IxRT 的文档
observer = create_observer(name: str, *args, **kwargs)
```

```
# 在 static_quantize 中使用
static_quantize(..., observer = observer)
```

7.2 基于 QDQ ONNX 的 INT8 推理

IxRT 使用 QDQ 格式的 onnx 文件执行 INT8 推理，因此 **请您确保使用的 onnx 文件是 QDQ 格式**：

```
def static_quantize(
        ...
    quant_format="qdq",
):
```

当 IxRT 检测到计算图中包含 Q/DQ 对时，会执行 QDQ 优化，并最终综合决策出运行时要使用 INT8 推理的算子。要使用 INT8 精度推理一个网络模型，您需要在配置时进行如下设定：

```
config->setFlag(BuilderFlag::kINT8);
```

7.2.1 QDQ ONNX 的权重

对于量化算子的权重，可以是量化的，也可以是非量化的。如果是量化的，其结构应为：

```
int8 weight --> DQ --> OP
```

如果是非量化的，其结构应为：

```
weight --> Q --> DQ --> OP
```

7.2.2 IxRT INT8 自定义算子

对于自定义的 INT8 算子，要求该算子在 ONNX 中必须严格遵循 QDQ 模式。假设您实现了一个 INT8 插件 `MyInt8Plugin`，那么您需要将该插件置于 DQ/Q 对之间：

```
DQ --> MyInt8Plugin --> Q
```

该模式将被 IxRT 识别、优化并以 INT8 精度运行。

下面是一个示例：

- 原模式：

```
DQ --> Conv --> Q --> DQ --> MyInt8Plugin --> Q
```

- 优化后的模式：

```
Conv --> MyInt8Plugin
```

优化后的 Conv 和 MyInt8Plugin 均以 INT8 模式运行。

8 (面向开发者) 使用 IxRT 接口进行开发

您获得一个 ONNX 模型后，应该先构建引擎文件，再加载该引擎文件进行推理。通常的推理场景中，构建引擎文件和加载引擎文件进行推理是两个独立的程序。构建引擎文件的时间会随模型尺寸的增长而增长。构建好引擎文件后，运行时的最佳状态便被确定下来。在模型推理时，加载该引擎文件即可。

本章节针对开发者提供，分别介绍使用 C++ API 和 Python API 构建引擎文件后加载引擎文件进行推理的说明。

8.1 C++ API

IxRT 的 C++ API 中的接口类似以前缀"I" 开头，例如：ILogger, IBuilder 等。建议您在第一次调用 IxRT 之前，先自行创建和配置 CUDA 上下文。为了说明对象生命周期，本小节中的代码不使用智能指针。建议您在 IxRT 接口中使用智能指针。

IxRT 头文件默认安装在 `/usr/local/corex/include/` 目录下，库默认安装在 `/usr/local/corex/lib/` 目录下。为使推理程序能够链接到 `libixrt.so`，请确保以下环境变量已设置：

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/corex/lib
```

使用以下代码引用必备的头文件：

```
#include "NvInfer.h"
#include "NvOnnxParser.h"

using namespace nvinfer1;
```

推理流程将分为两步：1. 构建引擎文件 - C++ 和2. 加载引擎文件并执行推理 - C++。其中，在构建引擎文件阶段，IxRT 对计算图进行必要的图优化和底层算法寻优，从而得到推理时最佳状态。构建引擎文件的时间会随模型的大小而不同。

Tip

建议您对照 ResNet18 C++ API 推理案例 (IxRT OSS 路径：`${path_for_run_package}/samples/sampleResNet18/future_classifier.cc`) 来学习 C++ API 的使用。

8.1.1 1. 构建引擎文件 - C++

1. 首先，需要实例化 ILogger 接口和创建生成器，进而创建推理所需的相关对象。

   IxRT 允许用户的 logger 接口访问内部返回的日志信息，因此创建 logger 并给到各个创建的对象，有助于了解运行状态。这里给出最简单的范例，打印到命令行。由于传出的是字符串，也可以保存到文件或转发到中间件。需要继承 ILogger 类并且至少实现 log 方法。

```
using Severity = nvinfer1::ILogger::Severity;
class Logger : public nvinfer1::ILogger {
public:
    explicit Logger(Severity severity = Severity::kWARNING) : mReportableSeverity(severity)
    {}
    nvinfer1::ILogger& getIxRTLogger() noexcept { return *this; }

    void log(Severity severity, const char* msg) noexcept override {
        if (severity <= mReportableSeverity) {
            std::cout << severityPrefix(mReportableSeverity) << "[IXRT] " << msg <<
    std::endl;
        }
    }

    void setReportableSeverity(Severity severity) noexcept { mReportableSeverity =
        severity; }

    Severity getReportableSeverity() const { return mReportableSeverity; }

private:

    static const char* severityPrefix(Severity severity) {
        switch (severity) {
            case Severity::kINTERNAL_ERROR:
                return "[F] ";
            case Severity::kERROR:
                return "[E] ";
            case Severity::kWARNING:
                return "[W] ";
            case Severity::kINFO:
                return "[I] ";
            case Severity::kVERBOSE:
                return "[V] ";
            default:
                assert(0);
                return "";
        }
    }

    Severity mReportableSeverity;
};  // class Logger
```

2. 然后，您可以创建生成器的实例：

```
IBuilder* builder = createInferBuilder(logger);
```

3. 创建网络定义：

```
uint32_t flag = 1U << static_cast<uint32_t>
↪ (NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);

INetworkDefinition* network = builder->createNetworkV2(flag);
```

为了使用 ONNX 解析器导入模型，需要 kEXPLICIT_BATCH 标志。

4. 使用 ONNX 解析器导入模型。

有了 Network 后，可以从 ONNX 解析导入模型。使用 #include "NvOnnxParser.h" 头文件中的 createParser API 创建一个 ONNX 解析器：

```
IParser* parser = createParser(*network, logger);
```

接着，对 ONNX 文件进行解析，并处理可能出现的错误。有如下四种解析 ONNX 文件的方法：

方法一

```
bool parsed = parser->parseFromFile(model_path.c_str(),
↪ static_cast<int>(logger.getReportableSeverity()));
if (!parsed) {
    std::cout << "Create onnx parser failed" << std::endl;
    return;
} else {
    cout << "Create onnx parser success" << endl;
}
```

parseFromFile(const char* onnxModelFile, int verbosity) 的参数含义如下：

- onnxModelFile：ONNX 模型的路径。
- verbosity：日志等级。

方法二

```
std::ifstream onnx_file(model_path, std::ios::ate | std::ios::binary);
std::streamsize onnx_size = onnx_file.tellg();
onnx_file.seekg(0, std::ios::beg);
std::vector<char> onnx_buffer(onnx_size);
if (!onnx_file.read(onnx_buffer.data(), onnx_size)) {
    std::cout << "Failed to read from file: " << model_path << std::endl;
    return;
}
auto parsed = parser->parse(onnx_buffer.data(), onnx_size,
↪ external_weight_location.c_str());
if (!parsed) {
    std::cout << "Create onnx parser failed" << std::endl;
    return;
} else {
    cout << "Create onnx parser success" << endl;
}
```

parse(void const* serialized_onnx_model, size_t serialized_onnx_model_size, const char* model_path = nullptr) 的参数含义如下：

- serialized_onnx_model：指向 serialized onnx 的指针。
- serialized_onnx_model_size：serialized onnx 的大小。
- model_path：外部权重文件的绝对路径，以便在需要时加载外部权重。

方法三

```
bool parsed = parser->parseFromFiles(model_path.c_str(), quant_param_path.c_str(),
↪  static_cast<int>(logger.getReportableSeverity()));
if (!parsed) {
    std::cout << "Create onnx parser failed" << std::endl;
    return;
} else {
    cout << "Create onnx parser success" << endl;
}
```

parseFromFiles(const char* onnxModelFile, const char* quantParamFile, int verbosity) 的参数含义如下：

- onnxModelFile：ONNX 模型的路径。
- quantParamFile：量化参数文件的路径 (使用 deploy 工具对模型进行量化)。
- verbosity：日志等级。

方法四

```
std::ifstream onnx_file(model_path, std::ios::ate | std::ios::binary);
std::streamsize onnx_size = onnx_file.tellg();
onnx_file.seekg(0, std::ios::beg);
std::vector<char> onnx_buffer(onnx_size);
if (!onnx_file.read(onnx_buffer.data(), onnx_size)) {
    std::cout << "Failed to read from file: " << model_path << std::endl;
    return;
}
auto parsed = parser->parseFromFiles(onnx_buffer.data(), onnx_size,
↪  quant_param_path.c_str(), external_weight_location.c_str());
if (!parsed) {
    std::cout << "Create onnx parser failed" << std::endl;
    return;
} else {
    cout << "Create onnx parser success" << endl;
}
```

parseFromFiles(void const* serialized_onnx_model, size_t serialized_onnx_model_size, const char* quant_param_file, const char* model_path = nullptr) 的参数含义如下：

- serialized_onnx_model：指向 serialized onnx 的指针。
- serialized_onnx_model_size：serialized onnx 的大小。
- quant_param_file：量化参数文件的路径 (使用 deploy 工具对模型进行量化)。

- `model_path`：外部权重文件的绝对路径，以便在需要时加载外部权重。

5. 构建引擎文件。

   创建一个配置，指示 IxRT 如何优化这个模型：

   ```
   IBuilderConfig* config = builder->createBuilderConfig();
   ```

   您可以设置使用 INT8 推理或 FP16 推理：

   ```
   config->setFlag(nvinfer1::BuilderFlag::kINT8);
   config->setFlag(nvinfer1::BuilderFlag::kHALF);
   ```

   配置确定后，开始构建引擎文件：

   ```
   IHostMemory* = builder->buildSerializedNetwork(*network, *config);
   ```

   这个过程中，会经历以下步骤：

   1. 规划推理资源
   2. 优化计算图
   3. 选择最优算法

8.1.2 2. 加载引擎文件并执行推理 - C++

1. 您需要从已序列化的引擎文件中恢复模型定义和配置，用于创建运行时环境。首先，需要创建一个类似于 builder 的 Runtime：

   ```
   IRuntime* runtime = nvinfer1::createInferRuntime(logger)
   ```

   将模型读入内存后，可以将其反序列化以获得引擎文件：

   ```
   ICudaEngine* engine = runtime->deserializeCudaEngine(modelData, modelSize);
   ```

2. 执行推理。

   使用引擎文件创建运行环境，即 IExecutionContext 对象：

   ```
   IExecutionContext *context = engine->createExecutionContext();
   ```

   一个引擎文件可以有多个执行上下文，允许一组权重用于多个重叠的推理任务。

   使用 `executeV2` 函数执行推理过程，输入参数只有一个，表示推理用的输入和输出数据。从 `binding_buffer` 中分配好输入输出内存，并拷贝输出数据到内存中，接着送入 `executeV2` 函数便可执行推理。执行结束后，结果会写入到输出内存：

   ```
   bool status = context->executeV2(binding_buffer.data());
   ```

8.2 Python API

使用 Python API 推理的流程分为两步：1. 构建引擎文件 - Python 和2. 加载引擎文件并执行推理 - Python。

您可以搭配 IxRT 开源代码中的 `${path_for_run_package}/samples/python/resnet18/resnet18.py` 脚本一起使用。

IxRT Python 包安装在系统 Python 环境下，使用以下代码引用必备库：

```
import tensorrt
```

8.2.1 1. 构建引擎文件 - Python

1. 创建日志记录器。

   如果要创建生成器 (builder)，必须首先创建日志记录器 (logger)。IxRT Python 绑定接口提供了一个简单的 logger，用于将日志信息输出到终端 (stdout)。

   ```
   IXRT_LOGGER = tensorrt.Logger(tensorrt.Logger.WARNING)
   ```

   当然，您也可以实现自己的日志类，用于更加定制化的操作：

   ```
   class MyLogger(tensorrt.ILogger):
       def __init__(self):
       trt.ILogger.__init__(self)

       def log(self, severity, msg):
           pass # 实现您的日志记录方式

   logger = MyLogger()
   ```

2. 创建生成器。

   有了 logger 之后，用其创建一个 builder。

   ```
   builder = tensorrt.Builder(IXRT_LOGGER)
   ```

   Note

   引擎文件的构建通常作为离线的、单独的一个步骤，用于与运行时的推理分开。构建一个引擎文件可能会花费较久的时间，因此，建议您在推理环节直接使用构建好的引擎文件进行推理。

3. 创建一个 NetworkDefinition。

   在创建 builder 之后，首先需要创建一个 NetworkDefinition。

   ```
   network = builder.create_network(EXPLICIT_BATCH)
   ```

   `EXPLICIT_BATCH` 标志用于 ONNX 解析器解析过程。

4. 用 ONNX 解析器导入这个模型。

   ```
   parser = tensorrt.OnnxParser(network, IXRT_LOGGER)
   parser.parse_from_file(onnx_model)
   ```

5. 创建引擎文件。

   下一步是创建一个构建配置，指定 TensorRT 应该如何优化模型。

   ```
   build_config = builder.create_builder_config()
   ```

   您可以对构建配置的相关标识进行设置，比如设置推理类型：

   - 如果设定推理类型为 INT8，要求输入的 onnx 文件是 QDQ 格式，详情请参考" 使用 IxRT 推理引擎对模型进行 INT8 推理最佳实践" > " 基于 QDQ ONNX 的 INT8 推理" 并设置：

     ```
     build_config.set_flag(tensorrt.BuilderFlag.INT8)
     ```

   - 如果设定推理类型为 FP16，设置为：

     ```
     build_config.set_flag(tensorrt.BuilderFlag.FP16)
     ```

   配置完成后，即可序列化引擎文件至本地：

   ```
   serialized_engine = builder.build_serialized_network(network, build_config)
   with open("sample.engine", "wb") as f:
       f.write(serialized_engine)
   ```

   Note

   不同版本的 IxRT 间可能不支持移植对应版本的引擎文件，因此建议您使用同一版本的 IxRT 来构建引擎文件并进行推理。

8.2.2 2. 加载引擎文件并执行推理 - Python

1. 创建运行时。

   执行推理阶段，需要基于 logger 和 `Runtime` 接口类来创建一个运行时

   ```
   runtime = trt.Runtime(logger)
   ```

2. 导入引擎文件。

   从文件或者内存中导入这个引擎文件。

   - 从内存中导入：

     ```
     engine = runtime.deserialize_cuda_engine(serialized_engine)
     ```

   - 从文件中导入：

     ```
     with open("sample.engine", "rb") as f:
         serialized_engine = f.read()
     engine = runtime.deserialize_cuda_engine(serialized_engine)
     ```

3. 执行推理。

   引擎文件保存优化后的模型参数，但要执行推理，需要创建一个上下文 (Context)：

```
context = engine.create_execution_context()
```

一个引擎文件可以用来创建多个 Context，继而共享权重用于多个推理任务。接着，使用 execute_v2 接口执行推理。

```
context.execute_v2(allocations)
```

您需要使用 pycuda 这个库为输入和输出分配足够的内存。一个分配内存的示例代码如下：

```
import pycuda.autoinit
import pycuda.driver as cuda
allocations = []
for i in range(engine.num_bindings):
    dtype = engine.get_binding_dtype(i)
    size = np.dtype(tensorrt.nptype(dtype)).itemsize
    for s in shape:
        size *= s
    allocation = cuda.mem_alloc(size)
    allocations.append(allocation)
```

在执行前，需要将输入拷贝至输入 GPU 地址：

```
cuda.memcpy_htod(inputs[0]["allocation"], data_batch)
```

执行后，可以选择将输出拷贝至 CPU 进行结果分析：

```
cuda.memcpy_dtoh(output, outputs[0]["allocation"])
```

上述的 inputs 和 outputs 表示输入和输出的相关信息，output 表示输出的 CPU 地址，详情请查看 ${path_for_run_package}/IxRT/oss/samples/python/resnet18/resnet18.py 脚本进行学习。

9 IxRT 动态形状推理功能介绍

动态形状推理是指在构建引擎文件时仅提供输入的维度范围，在推理时才指定真实的输入维度。动态形状可以通过 C++ 和 Python 接口使用。

构建和执行具有动态形状的引擎文件的步骤如下：

1. 创建具有显式批次维度的网络定义。

   - C++

   ```
   const auto explicitBatch = 1U <<
   ↪  static_cast<uint32_t>(nvinfer1::NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
   auto network =
   ↪  UniquePtr<nvinfer1::INetworkDefinition>(builder->createNetworkV2(explicitBatch));
   ```

   - Python

   ```
   EXPLICIT_BATCH = 1 << (int)(tensorrt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
   network = builder.create_network(EXPLICIT_BATCH)
   ```

2. 使用-1 作为输入维度占位符。
3. 创建一个或多个优化配置文件，为推理时输入指定允许的维度范围。

   - C++

   ```
   auto profile = builder->createOptimizationProfile();
   profile->setDimensions(input_name.c_str(), nvinfer1::OptProfileSelector::kMIN,
   ↪  nvinfer1::Dims{4, {1, 3, 112, 112}});
   profile->setDimensions(input_name.c_str(), nvinfer1::OptProfileSelector::kOPT,
   ↪  nvinfer1::Dims{4, {1, 3, 224, 224}});
   profile->setDimensions(input_name.c_str(), nvinfer1::OptProfileSelector::kMAX,
   ↪  nvinfer1::Dims{4, {1, 3, 448, 448}});
   config->addOptimizationProfile(profile);
   ```

   - Python

   ```
   profile = builder.create_optimization_profile()
   profile.set_shape(
       input_name, Dims([1, 3, 112, 112]), Dims([1, 3, 224, 224]), Dims([1, 3, 448, 448])
   )
   config.add_optimization_profile(profile)
   ```

4. 根据引擎文件创建执行上下文，与非动态形状的步骤相同。
5. 指定一个步骤 3 中所创建的优化配置文件。

   - C++

```
context.setOptimizationProfileAsync(profile_idx, stream)
```

- Python

```
context.set_optimization_profile_async(profile_idx, stream)
```

Note

- 创建多个执行上下文时，也需要创建多份优化配置文件。每个执行上下文必须使用单独的优化配置文件。优化配置文件的数量决定了 Context 可以创建的上限。
- `stream` 是用于后续 `enqueueV2()` 调用的 CUDA 流。

6. 指定执行上下文的输入维度。

- C++

```
context->setBindingDimensions(input_idx, input_dims);
```

- Python

```
context.set_binding_shape(input_idx, input_dims)
```

7. 调用 `executeV2` 或 `enqueueV2()` 执行推理，与非动态形状的步骤相同。

10 IxRT 自定义插件算子功能介绍

IxRT 内置多种算子和融合算子供您使用，但有时您可能会遇到 IxRT 不支持某算子的情况，又或者您有自定义高性能算子的需求。在这些场景下，您可以使用 IxRT 的插件功能来满足您的业务需求。

IxRT 开放的源代码包含了部分插件算子供您学习与使用，这些插件算子被编译为 `libixrt_plugin.so`，默认安装在 `/usr/local/corex/lib/` 目录下。使用 IxRT 提供的插件，您需要链接 `libixrt_plugin.so`。如果插件仍无法满足您的需求，您可以实现自己的插件算子。

IxRT 提供的插件既可以被单独调用，也可以作为整个模型的一部分，在推理模型时调用。IxRT 提供的插件均需要使用 C++ 实现，其中算子运行部分需要您使用 CUDA 或者第三方库实现，详情请参考 IxRT 开放源代码提供的插件实现。

Tip

您可以在 IxRT 的 `tar.gz` 包或 `.run` 包中获取 IxRT 开放的源代码。

IxRT 提供天数智芯适配版的 TPG (TensorRT Plugin Generator) 工具，方便在您开发自定义插件算子时，基于 ONNX 或 YAML 文件，轻松生成 IxRT 插件算子。详情请参考使用插件生成工具。

关于 IxRT 支持的插件列表，请参考” 附录 4：IxRT 支持的插件列表”。

10.1 使用自定义插件算子的工作流

使用 IxRT 提供的自定义插件算子推理模型将经过如下步骤：

1. 获得包含插件的 ONNX 模型
2. 实现插件
3. 测试计算的正确性
4. 使用插件算子推理模型

10.1.1 1. 获得包含插件的 ONNX 模型

首选，您需要确保待推理的 ONNX 模型中包含插件算子。**注意：该算子的类型 (op_type) 不能与 ONNX 标准算子类型同名。**您可以使用 IxRT 提供的部署工具修改 ONNX，用于自定义算子。以下是以 YOLOv3 模型为例，自定义 Decoder 算子的代码示例：

```
# !/usr/bin/env python
# -*- coding: utf-8 -*-
import argparse
from os.path import basename, dirname, join

from ixrt.deploy.api import DataType, GraphTransform, create_source, create_target
```

```
class YoloV3Transform:
    def __init__(self, graph, custom):
        self.t = GraphTransform(graph)
        self.graph = graph
        if custom:
            self.op_type = "YoloV3Decoder_IxRT"
        else:
            self.op_type = "YoloV3Decoder"

    def AddYoloDecoderOp(self, inputs: list, outputs: list, **attributes):
        self.t.make_operator(self.op_type, inputs=inputs, outputs=outputs, **attributes)
        # The end of original onnx is output, so replace it
        for dest, src in zip(outputs, inputs):
            self.t.replace_output(dest, src)
            self.t.get_variable(dest).dtype = "FLOAT"
        return self.graph

    def SetDynamicInput(self):
        self.t.get_variable("images").set_shape([1, 3, "bs", "bs"])
        return self.graph


def add_yolov3_decoder(graph, custom, dynamic=False):
    t = YoloV3Transform(graph, custom)
    graph = t.AddYoloDecoderOp(
        inputs=["output"],
        outputs=["decoder_13"],
        anchor=[116, 90, 156, 198, 373, 326],
        num_class=80,
        stride=32,
    )
    graph = t.AddYoloDecoderOp(
        inputs=["664"],
        outputs=["decoder_26"],
        anchor=[30, 61, 62, 45, 59, 119],
        num_class=80,
        stride=16,
    )
    graph = t.AddYoloDecoderOp(
        inputs=["692"],
        outputs=["decoder_52"],
        anchor=[10, 13, 16, 30, 33, 23],
        num_class=80,
        stride=8,
    )
    if dynamic:
```

```
        graph = t.SetDynamicInput()
    return graph


def parse_args():
    file = join(
        dirname(__file__),
        "../../data/yolov3/quantized_yolov3_without_decoder_shape.onnx",
    )
    dest = join(
        dirname(__file__),
        "../../data/yolov3/quantized_yolov3_with_decoder.onnx",
    )
    parser = argparse.ArgumentParser("Add yolov3 decoder")
    parser.add_argument("--src", type=str, default=file)
    parser.add_argument("--dest", type=str, default=dest)
    parser.add_argument("--dynamic", action="store_true", default=False)
    parser.add_argument(
        "--custom", action="store_true", default=False, help="Use IxRT plugin"
    )
    config = parser.parse_args()
    return config


if __name__ == "__main__":
    config = parse_args()
    graph = create_source(config.src)()
    graph = add_yolov3_decoder(graph, config.custom, config.dynamic)
    create_target(saved_path=config.dest).export(graph)
    print("Surged onnx lies on", config.dest)
```

10.1.2 2. 实现插件

如果 IxRT 提供的插件无法满足您的需求，您可以实现自己的插件算子。要实现一个自定义插件，您需要：

1. 首先继承以下两个接口类：
    - 插件算子类 `IPluginV2DynamicExt`
    - 插件创建类 `IPluginCreator`
2. 实现两个类对应的方法后，再使用宏 `REGISTER_TENSORRT_PLUGIN` 对插件创建类进行注册。

10.1.3 3. 测试计算的正确性

实现算子后，建议单独测试并验证正确性，再将其运用于整体模型的推理。您可以使用如下方法调用实现的插件算子进行测试：

```
// Look up the plugin in the registry
auto creator = getPluginRegistry()->getPluginCreator(pluginName, pluginVersion);
const PluginFieldCollection* pluginFC = creator->getFieldNames();
// Populate the fields parameters for the plugin layer
PluginFieldCollection *pluginData = parseAndFillFields(pluginFC, layerFields);
// Create the plugin object using the layerName and the plugin meta data
IPluginV2 *pluginObj = creator->createPlugin(layerName, pluginData);
// Run the plugin
pluginObj->enqueue(inputDesc, outputDesc, inputs, outputs, workspace, stream)
```

10.1.4 4. 使用插件算子推理模型

实现插件算子后，当 IxRT 解析 ONNX 模型并与到您实现过的算子类型时，IxRT 便会将该算子作为运行图的一部分，令其参与运行时计算。因此，您不需要修改推理流程的代码即可执行推理。参考” 使用 IxRT 推理模型通用教程” > ” 教程 3：(面向开发者) 使用 IxRT 接口进行开发” > ”C++ API”，了解如何编写推理代码。

10.2 插件算子的 C++ API

本小节介绍插件算子的实现方法。如2. 实现插件 所述，继承的两个接口类分别是：

- 插件算子类 `IPluginV2DynamicExt`
- 插件创建类 `IPluginCreator`

其中，插件算子类 `IPluginV2DynamicExt` 支持动态形状。假设实现一个 `FooPlugin`，其有 3 个输入和 3 个输出，并且：

- 第 1 个输出是第 2 个输出的拷贝
- 第 2 个输出是 3 个输入沿着第 1 个维度的拼接，所有类型/格式必须是相同的且是线性格式的
- 第 3 个输出的 [N, 1]

`FooPlugin` 可以按照以下方式派生：

```
class FooPlugin : public IPluginV2DynamicExt
{
        ...override virtual methods inherited from IPluginV2DynamicExt.
};
```

受动态形状影响的 4 个方法分别是：

- `getOutputDimensions`
- `supportsFormatCombination`
- `configurePlugin`
- `enqueue`

其中，`getOutputDimensions` 是用于计算关于输入形状的输出形状的 符号表示。您可以使用传入的 `IExprBuilder` 来构建新的表达式。对于本例中，`getOutputDimensions` 的实现为：

```
DimsExprs FooPlugin::getOutputDimensions(int outputIndex,
    const DimsExprs* inputs, int nbInputs,
    IExprBuilder& exprBuilder)
{
    switch (outputIndex)
    {
    case 0:
    {
        return inputs[0];
    }
    case 1:{
        // First dimension of output is sum of input along first dimensions.
        DimsExprs output(inputs[0]);
        auto two_sum = exprBuilder.operation(DimensionOperation::kSUM,
                inputs[0].d[0], inputs[1].d[0]);
        output.d[0] = exprBuilder.operation(DimensionOperation::kSUM,
                two_sum, inputs[2].d[0]);
    return output;
    }
    case 2:{
        DimsExprs output;
        output.nbDims=2;
        output.d[0] = exprBuilder.constant(inputs[2].d[0]);
        output.d[1] = exprBuilder.constant(1);
        return output;
    }
    default:
        throw std::invalid_argument( “invalid output” );
}
```

supportsFormatOrganization 的重写必须指明是否允许格式组合。接口将输入/输出统一地用连接编号 pos 进行索引。第 1 个输入从 0 开始，其余的输入按照顺序排列，然后对输出进行编号。在本例中，输入为连接 0, 1 和 2，输出为连接 3, 4 和 5：

```
bool FooPlugin::supportsFormatCombination(int pos, const PluginTensorDesc* inOut, int nbInputs,
↪   int nbOutputs) override
{
    assert(0 <= pos && pos < 4);
    const auto* in = inOut;
    const auto* out = inOut + nbInputs;
    switch (pos)
    {
    case 0: return in[0].format == TensorFormat::kLINEAR;
    case 1: return in[1].type == in[0].type &&
                in[1].format == TensorFormat::kLINEAR;
    case 2: return in[0].format == TensorFormat::kLINEAR;
```

```
    case 3: return out[1].type == in[0].type &&
                out[1].format == TensorFormat::kLINEAR;
    case 4: return out[1].type == in[0].type &&
                out[1].format == TensorFormat::kLINEAR;
    case 5: return out[1].type == in[0].type &&
                out[1].format == TensorFormat::kLINEAR;
    }
    throw std::invalid_argument( “invalid connection number” );
}
```

Note

这里的 in 和 out 局部变量使能通过输入或输出编号，而不是连接编号索引 inOut。

IxRT 使用 configurePlugin 在运行时设置插件。由于 FooPlugin 插件不需要 configurePlugin 做任何事情，所以这里是个空实现：

```
void FooPlugin::configurePlugin(
    const DynamicPluginTensorDesc* in, int nbInputs,
    const DynamicPluginTensorDesc* out, int nbOutputs) override
{
}
```

如果插件需要知道它可能会遇到的最小或最大尺寸，您可以对任何输入或输出调用 DynamicPluginTensorDesc: : min 或 DynamicPluginTensorDesc: : max 字段。格式和构建时维度信息可以在 DynamicPluginTensorDesc: : desc 中找到。任何运行时维度显示为-1，实际的维度会提供给 FooPlugin: : enqueue。

最后，重写 FooPlugin: : enqueue。由于形状是动态的，因此会向 enqueue 方法传递一个 PluginTensorDesc。该 PluginTensorDesc 描述每个输入和输出的实际尺寸、类型和格式。

10.3 使用插件生成工具

1. 向您的应用工程师获取天数智芯适配版的 TPG 工具的 .whl 包并使用 **pip** 安装，例如：

   ```
   $ pip3 install trtpg-1.3.0+corex.{v.r.m}-cp310-cp310-linux_x86_64.whl
   ```

2. 假设您的 ONNX 中包含 IxRT 不支持的 custom_add 和 custom_conv 算子，使用如下命令转为 YAML 配置文件：

   ```
   $ trtpg extract --onnx model.onnx --custom_operators custom_add,custom_conv --yaml
   ↪  plugin.yml
   ```

3. 检查生成的 YAML 配置文件中是否包含 need_user_to_specify 项。如果包含，说明 TPG 无法从 ONNX 中推断出该项信息，需要您手动更改为对应的值。参考 官方文档 了解 YAML 文件的修改规则。

4. 生成插件代码：

```
$ trtpg generate --yaml plugin.yml --output ./plugin_codes
```

在生成插件的过程中，您可能需要指定 IxRT 头文件安装路径、库文件安装路径和 CUDA 安装路径。参考如下命令了解对应的传参方式：

```
$ trtpg generate -h
```

5. 完成插件推理代码。

   您需要实现以下方法：

方法名	是否必须要实现	说明
getOutputDimensions()	必须	形状推导，计算输出的形状
enqueue()	必须	最重要的部分，该算子的计算逻辑
initialize()	可选	为运行时额外分配的内存
terminate()	可选	释放所有 `initialize()` 中分配的内存
configurePlugin()	可选	如果动态形状下，不同形状对应不同的最优实现，您可调用此方法
getWorkspaceSize()	可选	`initialize()` 需要分配的内存大小

关于 TPG 的更多信息，请参考 https://github.com/NVIDIA-AI-IOT/tensorrt_plugin_generator。

11 IxRT 图优化功能介绍

IxRT 内置的高效图优化引擎，支持对模型的计算图进行深度优化，主要包括自动类型转换、自动通道对齐、可复用内存管理、算子融合等，从而达到模型的极致性能。

11.1 自动类型转换

当一个算子的输入数据类型与算子计算所需数据类型不符时，IxRT 能提供自动类型转换功能，使二者的数据类型保持一致。例如，将 FP32 转换为 FP16。

11.2 自动通道对齐

部分算子 (例如 conv2d) 在天数智芯加速卡上需要通道数为 32 或 64 倍数才能达到最优的运行效率。IxRT 支持对数据进行自动通道对齐，更改对应数据通道数量以及算子通道属性，保证算子能够在天数智芯加速卡上以最优的效率运行。

11.3 可复用内存管理

在推理模型时，需要分配的存储空间分为如下两类：

- 常量存储：用来记录模型的权重和参数配置，在推理过程中一直保持不变，因此这部分存储很难有优化空间。
- 可变存储：用来存储临时生成的数据，常见的有算子的输入和输出数据，在推理过程中会被不断更新。IxRT 对于显存使用的优化主要是针对这部分存储。

IxRT 的内存管理模块可以做到在正确运行模型推理过程的同时尽可能少地占用显存空间。其主要使用了如下两种优化策略：

- 原地复用：部分算子 (例如激活算子、reshape 算子) 可以进行 inplace 操作，这类算子的输出复用输入的内存空间即可。
- 时分复用：模型在推理时，不同算子的执行顺序存在依赖关系，同一个任务队列中，同一时刻正在执行的算子只有一个。因此，同一任务队列中的不同算子可以使用同一块内存 (例如 Op2 的 output buffer 可以复用 Op1 的 input buffer)，从而做到内存的时分复用。

11.4 算子融合

算子融合是推理框架中常见的图优化方式，通过将多个算子组成的子图替换为一个或多个新的算子。算子融合的主要目的在于：

- 减少算子数量：通过减少算子数量，可以减少 kernel launch 的次数，从而减少计算时间。
- 减少数据搬运：通过融合算子减少数据搬运的次数，从而减少运行时间。

IxRT 中常见的算子融合方法有：

- Sigmoid + Mul：将相邻的 sigmoid 和 mul 算子融合为一个 silu 算子
- Conv + activation：将相邻的卷积核激活算子融合成一个卷积算子
- Conv + activation + Elementwise：将相邻的卷积、激活、逐元素操作算子融合成一个卷积算子
- parallel conv：将具有相同输入数据且卷积核形状相同的多个卷积替换为一个卷积 +split 算子的组合
- Bert layer：将 attention、matmul、layernormalization、gelu 组成的子图融合为 bert 算子
- Continuous Bert layer：将多个连续的 bert 算子融合为一个 bert 算子
- Matmul + Add：将相邻的 Matmul、Add 算子融合为一个 linear 算子

12 IxRT-EXEC 自动化模型推理工具使用指南

IxRT-EXEC 是一个集成在 IxRT 推理引擎中的命令行工具，支持用户在不同开发代码的场景下使用 IxRT 推理引擎快速运行 ONNX 模型。IxRT-EXEC 工具集主要提供了一系列经典模型的算子支持情况检验和推理性能测试能力。

12.1 IxRT-EXEC 功能说明

特性	说明
多精度支持	支持 INT8、FP16 精度推理测试
模型支持	目前测试通过 ResNet50, InceptionNetV2, YOLOv5m, YOLOv7 等模型

12.2 IxRT-EXEC 使用指南

1. 安装 IxRT 引擎后，IxRT-EXEC 工具的所在目录将由您环境中的 Python 所决定：
   - Python 在系统环境中时，IxRT-EXEC 所在目录为 `/usr/local/bin/ixrtexec`
   - 使用 Conda 管理 Python 环境时，IxRT-EXEC 所在目录为 `<path-to-Conda>/envs/<env-name>/bin/ixrtexec`
2. 进入 IxRT-EXEC 工具所在的目录，执行如下命令查看工具的使用帮助：

```
$ ixrtexec -h
```

3. 执行如下命令使用 IxRT-EXEC 工具：

```
$ ixrtexec --onnx <path-to-onnx-file> [option 1] [option 2] ...
```

12.3 IxRT-EXEC 运行参数说明

IxRT-EXEC 推理工具的运行参数说明如下：

参数	说明
--onnx	必选，指定要运行的 ONNX 模型路径
--precision	可选，指定模型运行时的精度，可以指定为 int8 或者 fp16，默认为 fp16

参数	说明
--quant_file	可选，模型运行精度为 INT8 时，可指定量化 JSON 文件的路径，未指定时将自动优先调用量化模块自动得到的临时量化 JSON 文件
--input_types	可选，指定模型输入数据类型，示例：input0:float16,input1:int32，如未指定则使用 ONNX 中默认输入数据类型
--output_types	可选，指定模型输出数据类型，示例：output0:float32, output1:int32，如未指定则使用 ONNX 中默认输出数据类型
--warmUp	可选，指定模型 warmup 次数，默认为 10
--iterations	可选，指定模型运行次数，默认为 10
--log_level	可选，指定 IxRT 运行时显示日志的等级，可指定为 verbose、info、warning、error、internal_error，默认为 warning
--dump_graph	可选，以 ONNX 形式保存计算图的文件位置，默认为 None
--save_engine	可选，指定保存引擎文件的位置，默认为 None，表示不保存引擎文件
--load_engine	可选，指定加载引擎文件的位置，默认为 None，表示不加载引擎文件
--run_profiler	可选，指定进行性能分析，结果会显示在窗口中，默认不进行性能分析
--export_profiler	可选，保存性能分析结果为指定的 csv 文件，示例：result.csv
--min_shape,--opt_shape,--max_shape	可选，指定动态形状输入的范围，如果勾选，需要将三个形状均设置好，格式：name:shape,name1:shape1
--shapes	可选，指定动态形状模型推理时的输入数据形状
--load_inputs	可选，从自定义的输入文件加载输入数据，格式：input_name1:file1,input_name2:file2
--verify_acc	可选，调用精度对比工具，逐层比较 IxRT 和 ONNXRuntime 的计算结果，以调试精度
--cosine_sim	可选，在精度对比中使用，余弦相似度阈值，默认为 0.999，余弦相似度阈值越接近 1，表明 IxRT 计算结果和比较结果越接近，比如 **--cosine_sim 0.99999**

参数	说明
--diff_max	可选，在精度对比中使用，Tensor 中最大差值的绝对值，默认不参与对比，此阈值越大，表明 IxRT 计算结果和比较结果的差异越大，比如 **--diff_max 5.0**
--diff_sum	可选，在精度对比中使用，Tensor 差值和的绝对值，默认不参与对比，此阈值越大，表明 IxRT 计算结果和比较结果的差异越大，比如 **--diff_sum 5.0**
--ort_onnx	可选，在精度对比中使用，指定 ONNXRuntime 加载原来的 ONNX，**--ort_onnx origin_model.onnx**
--ort_cpu	可选，在精度对比中使用，指定 ONNXRuntime 以 CPU 运行，不指定默认以 GPU 运行
--inject_tensors	可选，在精度对比中使用，指定某条输入边从外界加载到 IxRT 中，比如使用 **--inject_tensors 170** 指定 170 从 ONNXRuntime 的运行结果输入，**--inject_tensors 170:170.npy** 指定从本地文件加载，多条边使用英文逗号隔开
--verify_input	可选，在精度对比中使用，鉴定 IxRT 的输入边是否和上次的输出边一致
--plugins	可选，指定提前加载的插件列表，比如 **--plugins liba.so libb.so**

Note

使用 `shapes`、`input_types`、`output_types` 等参数时，请使用 ONNX 模型中数据对应的名称进行设置。

12.4 使用 IxRT-EXEC 运行示例

IxRT-EXEC 提供了一系列经典模型的算子支持情况检验和推理性能测试，下面是部分运行示例：

12.4.1 对 ResNet50 模型进行 INT8 推理

```
$ ixtrexec --onnx ./resnet50.onnx --precision int8
```

12.4.2 对 ResNet50 模型进行 FP16 推理

```
$ ixtrexec --onnx ./resnet50.onnx
```

12.4.3 更改模型的输入数据类型

```
$ ixrtexec --onnx ./bert-base-optimized.onnx --precision fp16 --input_types
↪  input_ids:int32,token_type_ids:int32
```

12.4.4 进行性能分析并保存结果文件

```
$ ixrtexec --onnx ./resnet18.onnx --precision fp16 --run_profiler --export_profiler result.csv
```

运行完上述命令后，会在界面显示每层的性能分析数据，如下图所示：

```
=== Profile Layer (10 iterations) ===
                                  Layer Name    Time(ms)   Avg. Time(ms)   Median Time(ms)    Time(%)
------------------------------------------  ----------  --------------  -----------------  ---------
                      n::type_images_Conv_0       0.213           0.021              0.021      0.692
n::reformat_e::type_images_Conv_0_Conv_0          0.170           0.017              0.017      0.553
                                    Conv_0        0.749           0.075              0.076      2.432
                                 MaxPool_2        0.426           0.043              0.043      1.383
                                    Conv_3        0.529           0.053              0.053      1.717
                                    Conv_5        0.570           0.057              0.056      1.851
                                    Conv_8        0.514           0.051              0.052      1.667
                                   Conv_10        0.554           0.055              0.056      1.798
                                   Conv_16        0.210           0.021              0.021      0.683
                                   Conv_13        0.411           0.041              0.041      1.335
                                   Conv_15        0.803           0.080              0.080      2.607
                                   Conv_19        0.621           0.062              0.061      2.016
                                   Conv_21        0.783           0.078              0.078      2.544
                                   Conv_27        6.555           0.656              0.662     21.283
                                   Conv_24        0.919           0.092              0.092      2.982
                                   Conv_26        1.624           0.162              0.162      5.273
                                   Conv_30        1.416           0.142              0.142      4.597
                                   Conv_32        1.623           0.162              0.162      5.269
                                   Conv_35        1.397           0.140              0.140      4.534
                                   Conv_38        0.334           0.033              0.033      1.083
                                   Conv_37        3.682           0.368              0.368     11.953
                                   Conv_41        2.394           0.239              0.239      7.771
                                   Conv_43        3.673           0.367              0.367     11.925
                       GlobalAveragePool_46       0.291           0.029              0.029      0.944
n::reformat_onnx::Flatten_189_Flatten_47          0.000           0.000              0.000      0.000
                                  Flatten_47      0.000           0.000              0.000      0.000
                                    Gemm_48       0.231           0.023              0.023      0.749
                  n::type_output_output           0.111           0.011              0.011      0.359
```

Figure 4: 性能分析数据示例 1

显示每层 Layer 和 Kernel 的关系以及性能分析数据，如下图所示：

```
=== Profile Kernel (10 iterations) ===
                          Layer Name  Kernel Name                                                                  Time(ms)    Avg. Time(ms)   Median Time(ms)   Time(%)
------------------------------------  ---------------------------------------------------------------------------  ---------  --------------  ----------------  --------
                n::type_images_Conv_0  Fp32ToFp16KernelI3O4                                                          0.213          0.021             0.021     0.708
n::reformat_e::type_images_Conv_0_Conv_0  BatchTransposeKernelHalf_Small_H<16u, 512u, 16u, 64u>                      0.170          0.017             0.017     0.566
                             Conv_0  filterPadKAlign32<__half>                                                     0.140          0.014             0.014     0.465
                             Conv_0  implConvolution2DTcuKernelSmallBatchFP16Template<256u, 64u, 32u, 32u,        0.555          0.055             0.055     1.844
                          MaxPool_2  MaxPoolingForwardKernelNhwcFp16x2<float>                                     0.426          0.043             0.043     1.416
                             Conv_3  inputZeroPadding<__half>                                                     0.170          0.017             0.017     0.565
                             Conv_3  implConvolution2DTcuKernelSmallBatchFP16Template<256u, 64u, 32u, 32u,        0.305          0.030             0.031     1.012
                             Conv_5  inputZeroPadding<__half>                                                     0.178          0.018             0.017     0.593
                             Conv_5  implConvolution2DTcuKernelSmallBatchFP16Template<256u, 64u, 32u, 32u,        0.333          0.033             0.033     1.108
                             Conv_8  inputZeroPadding<__half>                                                     0.156          0.016             0.015     0.519
                             Conv_8  implConvolution2DTcuKernelSmallBatchFP16Template<256u, 64u, 32u, 32u,        0.303          0.030             0.031     1.008
                            Conv_10  inputZeroPadding<__half>                                                     0.170          0.017             0.017     0.566
                            Conv_10  implConvolution2DTcuKernelSmallBatchFP16Template<256u, 64u, 32u, 32u,        0.329          0.033             0.033     1.094
                            Conv_16  implConvolution2DTcuKernelSmallBatchFP16Template<256u, 128u, 32u, 64u,       0.210          0.021             0.021     0.699
                            Conv_13  implConvolution2DTcuKernelHalfCommon<128, 128, 64, 32, 32, true, false      0.411          0.041             0.041     1.367
                            Conv_15  inputZeroPadding<__half>                                                     0.176          0.018             0.017     0.585
                            Conv_15  implConvolution2DTcuKernelSmallBatchFP16Template<256u, 128u, 32u, 64u,       0.573          0.057             0.057     1.904
                            Conv_19  implConvolution2DTcuKernelHalfCommon<128, 128, 64, 32, 32, true, false      0.621          0.062             0.061     2.064
                            Conv_21  inputZeroPadding<__half>                                                     0.154          0.015             0.015     0.512
                            Conv_21  implConvolution2DTcuKernelSmallBatchFP16Template<256u, 128u, 32u, 64u,       0.573          0.057             0.057     1.904
                            Conv_27  implConvolution2DTcuKernelSmallBatchFP16Template<256u, 256u, 32u, 64u,       6.555          0.656             0.662    21.788
                            Conv_24  ImgToCol_FP16<128u>                                                         0.190          0.019             0.019     0.633
                            Conv_24  MatrixMulTensorCore<256u, 256u, 32u, 32u, 32u, true, false, false, 1,       0.674          0.067             0.067     2.239
                            Conv_26  inputZeroPadding<__half>                                                     0.150          0.015             0.015     0.500
                            Conv_26  implConvolution2DTcuKernelSmallBatchFP16Template<256u, 256u, 32u, 64u,       1.419          0.142             0.142     4.715
                            Conv_30  ImgToCol_FP16<256u>                                                         0.192          0.019             0.019     0.640
                            Conv_30  MatrixMulTensorCore<256u, 256u, 32u, 32u, 32u, true, false, false, 1,       1.169          0.117             0.118     3.886
                            Conv_32  inputZeroPadding<__half>                                                     0.150          0.015             0.015     0.500
                            Conv_32  implConvolution2DTcuKernelSmallBatchFP16Template<256u, 256u, 32u, 64u,       1.418          0.142             0.142     4.713
                            Conv_35  ImgToCol_FP16<256u>                                                         0.171          0.017             0.017     0.568
                            Conv_35  MatrixMulTensorCore<256u, 256u, 32u, 32u, 32u, true, false, false, 1,       1.171          0.117             0.118     3.893
                            Conv_38  implConvolution2DTcuKernelSmallBatchFP16Template<256u, 256u, 32u, 64u,       0.334          0.033             0.033     1.109
                            Conv_37  implConvolution2DTcuKernelSmallBatchFP16Template<256u, 256u, 32u, 64u,       3.682          0.368             0.368    12.237
                            Conv_41  ImgToCol_FP16<512u>                                                         0.175          0.017             0.017     0.580
                            Conv_41  MatrixMulTensorCore<256u, 256u, 32u, 32u, 32u, true, false, false, 1,       2.164          0.216             0.216     7.192
                            Conv_43  implConvolution2DTcuKernelSmallBatchFP16Template<256u, 256u, 32u, 64u,       3.673          0.367             0.367    12.208
                 GlobalAveragePool_46  AveragePoolingForwardKernelNhwc<float, __half, float>                      0.291          0.029             0.029     0.966
                            Gemm_48  mr_fast_gemm_f_f_h_h_h_tcuh<32u, 64u, 32u, 16u, 32u, true, false,           0.231          0.023             0.023     0.767
                n::type_output_output  CudaFp16ToFp32Kernel                                                       0.111          0.011             0.011     0.368

Profile file saved : result.csv
```

Figure 5: 性能分析数据示例 2

同时，会在本地生成一个.csv 文件。

12.4.5 使用精度对比工具

参考” 关于 IxRT 使用的高级话题 > (面向开发者) 精度对比工具” 了解工具的使用方法和原理。

13 配套软件使用指南

IxRT 推理引擎核心使用 C++ 进行开发，但使用 IxRT Python API 进行模型部署时依赖第三方软件，因此天数智芯提供了相应的适配版，目前有如下软件：

- CUDA Python
- PyCUDA

Note

配套软件的使用不依赖 IxRT 推理引擎，因此可以独立安装并使用。

13.1 CUDA Python

CUDA Python 可以让您以 Pythonic 的方式轻松访问天数智算软件栈支持的 CUDA API。CUDA Python 为 CUDA Driver 和 Runtime API 提供了 Cython 和 Python 封装，简化了基于 GPU 的并行计算。在使用 IxRT Python API 进行部署模型时，您可以使用 CUDA Python 来实现数据 H2D、D2H 和同步等操作。

本次发布适配来自 https://github.com/NVIDIA/cuda-python。

13.1.1 安装 CUDA Python

向您的应用工程师获取 `.whl` 安装包并使用 **pip3** 命令安装：

```
$ pip3 install cuda_python-{v.r.m}+corex.{v.r.m}-cp310-cp310-linux_x86_64.whl
```

13.1.2 使用示例

使用 Runtime API 示例如下：

```
import cuda.cudart as cudart
import cuda.cuda as cuda
import numpy as np
import ctypes
import base64

def cudaGetErrorEnum(error):
    if isinstance(error, cuda.CUresult):
        err, name = cuda.cuGetErrorName(error)
        return name if err == cuda.CUresult.CUDA_SUCCESS else "<unknown>"
    elif isinstance(error, cudart.cudaError_t):
        return cudart.cudaGetErrorName(error)[1]
    else:
```

```
        raise RuntimeError('Unknown error type: {}'.format(error))

def checkCudaErrors(result):
    if result[0].value:
        raise RuntimeError("CUDA error code={}({})".format(result[0].value,
        ↪  cudaGetErrorEnum(result[0])))
    if len(result) == 1:
        return None
    elif len(result) == 2:
        return result[1]
    else:
        return result[1:]

def assertSuccess(err):
    assert(err == cudart.cudaError_t.cudaSuccess)

def test_cudart_memcpy():
    # Set device
    devID = 0
    checkCudaErrors(cudart.cudaSetDevice(devID))

    # Allocate dev memory
    size = 1024 * np.uint8().itemsize
    err, dptr = cudart.cudaMalloc(size)
    assertSuccess(err)

    # Set h1 and h2 memory to be different
    h1 = np.full(size, 1).astype(np.uint8)
    h2 = np.full(size, 2).astype(np.uint8)
    assert(np.array_equal(h1, h2) is False)

    # mem set
    cudart.cudaMemset(dptr, 0, size)
    assertSuccess(err)

    # h1 to D
    err, = cudart.cudaMemcpy(dptr, h1, size, cudart.cudaMemcpyKind.cudaMemcpyHostToDevice)
    assertSuccess(err)

    # D to h2
    err, = cudart.cudaMemcpy(h2, dptr, size, cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost)
    assertSuccess(err)

    # Validate h1 == h2
    assert(np.array_equal(h1, h2))
```

```
    # Cleanup
    err, = cudart.cudaFree(dptr)
    assertSuccess(err)

def test_cudart_stream():
    stream_legacy = cudart.cudaStream_t(cudart.cudaStreamLegacy)
    stream_per_thread = cudart.cudaStream_t(cudart.cudaStreamPerThread)

    err_rt, stream_with_flags = cudart.cudaStreamCreateWithFlags(cudart.cudaStreamNonBlocking)
    assertSuccess(err_rt)
    err_rt, = cudart.cudaStreamSynchronize(stream_with_flags)
    assertSuccess(err_rt)
    err_rt, = cudart.cudaStreamDestroy(stream_with_flags)
    assert(err_rt == cudart.cudaError_t.cudaSuccess)

    error = cudart.cudaGetLastError()[0]
    assert error in [
            cudart.cudaError_t.cudaSuccess,
            cudart.cudaError_t.cudaErrorPeerAccessAlreadyEnabled,
            cudart.cudaError_t.cudaErrorPeerAccessNotEnabled
    ]

def test_cuda_device_access_peer():
    # Number of GPUs
    print("Checking for multiple GPUs...")
    gpu_n = checkCudaErrors(cudart.cudaGetDeviceCount())
    print("CUDA-capable device count: {}".format(gpu_n))

    if gpu_n < 2:
        print("Two or more GPUs with Peer-to-Peer access capability are required")
        return

    prop = [checkCudaErrors(cudart.cudaGetDeviceProperties(i)) for i in range(gpu_n)]
    # Check possibility for peer access
    print("\nChecking GPU(s) for support of peer to peer memory access...")

    p2pCapableGPUs = [-1, -1]
    for i in range(gpu_n):
        p2pCapableGPUs[0] = i
        for j in range(gpu_n):
            if i == j:
                continue
            i_access_j = checkCudaErrors(cudart.cudaDeviceCanAccessPeer(i, j))
            j_access_i = checkCudaErrors(cudart.cudaDeviceCanAccessPeer(j, i))
            print("> Peer access from {} (GPU{}) -> {} (GPU{}) : {}\n".format(
                    prop[i].name, i, prop[j].name, j, "Yes" if i_access_j else "No"))
```

```
         print("> Peer access from {} (GPU{}) -> {} (GPU{}) : {}\n".format(
               prop[j].name, j, prop[i].name, i, "Yes" if i_access_j else "No"))
         if i_access_j and j_access_i:
            p2pCapableGPUs[1] = j
            break
   if p2pCapableGPUs[1] != -1:
         break

if p2pCapableGPUs[0] == -1 or p2pCapableGPUs[1] == -1:
   print("Two or more GPUs with Peer-to-Peer access capability are required.")
   print("Peer to Peer access is not available amongst GPUs in the system, waiving test.")
   return

# Use first pair of p2p capable GPUs detected
gpuid = [p2pCapableGPUs[0], p2pCapableGPUs[1]]

# Enable peer access
print("Enabling peer access between GPU{} and GPU{}...".format(gpuid[0], gpuid[1]))
checkCudaErrors(cudart.cudaSetDevice(gpuid[0]))
checkCudaErrors(cudart.cudaDeviceEnablePeerAccess(gpuid[1], 0))
checkCudaErrors(cudart.cudaSetDevice(gpuid[1]))
checkCudaErrors(cudart.cudaDeviceEnablePeerAccess(gpuid[0], 0))

# Allocate buffers
buf_size = 1024 * 1024 * 16 * np.dtype(np.float32).itemsize
print("Allocating buffers ({}MB on GPU{}, GPU{} and CPU Host)...".format(int(buf_size / 1024
↪ / 1024), gpuid[0], gpuid[1]))
checkCudaErrors(cudart.cudaSetDevice(gpuid[0]))
g0 = checkCudaErrors(cudart.cudaMalloc(buf_size))
checkCudaErrors(cudart.cudaSetDevice(gpuid[1]))
g1 = checkCudaErrors(cudart.cudaMalloc(buf_size))
h0 = checkCudaErrors(cudart.cudaMallocHost(buf_size)) # Automatically portable with UVA

# Create CUDA event handles
print("Creating event handles...")
eventflags = cudart.cudaEventBlockingSync
start_event = checkCudaErrors(cudart.cudaEventCreateWithFlags(eventflags))
stop_event = checkCudaErrors(cudart.cudaEventCreateWithFlags(eventflags))

# P2P memcopy() benchmark
checkCudaErrors(cudart.cudaEventRecord(start_event, cudart.cudaStream_t(0)))

for i in range(100):
   # With UVA we don't need to specify source and target devices, the
   # runtime figures this out by itself from the pointers
   # Ping-pong copy between GPUs
```

```
        if i % 2 == 0:
            checkCudaErrors(cudart.cudaMemcpy(g1, g0, buf_size,
↪   cudart.cudaMemcpyKind.cudaMemcpyDefault))
        else:
            checkCudaErrors(cudart.cudaMemcpy(g0, g1, buf_size,
↪   cudart.cudaMemcpyKind.cudaMemcpyDefault))

    checkCudaErrors(cudart.cudaEventRecord(stop_event, cudart.cudaStream_t(0)))
    checkCudaErrors(cudart.cudaEventSynchronize(stop_event))
    time_memcpy = checkCudaErrors(cudart.cudaEventElapsedTime(start_event, stop_event))
    print("cudaMemcpyPeer / cudaMemcpy between GPU{} and GPU{}: {:.2f}GB/s".format(gpuid[0],
↪   gpuid[1],
            (1.0 / (time_memcpy / 1000.0)) * ((100.0 * buf_size)) / 1024.0 / 1024.0 / 1024.0))

    # Prepare host buffer and copy to GPU 0
    print("Preparing host buffer and memcpy to GPU{}...".format(gpuid[0]))

    h0_local = (ctypes.c_float * int(buf_size / np.dtype(np.float32).itemsize)).from_address(h0)
    for i in range(int(buf_size / np.dtype(np.float32).itemsize)):
        h0_local[i] = i % 4096

    checkCudaErrors(cudart.cudaSetDevice(gpuid[0]))
    checkCudaErrors(cudart.cudaMemcpy(g0, h0, buf_size, cudart.cudaMemcpyKind.cudaMemcpyDefault))

    # Disable peer access (also unregisters memory for non-UVA cases)
    print("Disabling peer access...")
    checkCudaErrors(cudart.cudaSetDevice(gpuid[0]))
    checkCudaErrors(cudart.cudaDeviceDisablePeerAccess(gpuid[1]))
    checkCudaErrors(cudart.cudaSetDevice(gpuid[1]))
    checkCudaErrors(cudart.cudaDeviceDisablePeerAccess(gpuid[0]))

    # Cleanup and shutdown
    print("Shutting down...")
    checkCudaErrors(cudart.cudaEventDestroy(start_event))
    checkCudaErrors(cudart.cudaEventDestroy(stop_event))
    checkCudaErrors(cudart.cudaSetDevice(gpuid[0]))
    checkCudaErrors(cudart.cudaFree(g0))
    checkCudaErrors(cudart.cudaSetDevice(gpuid[1]))
    checkCudaErrors(cudart.cudaFree(g1))
    checkCudaErrors(cudart.cudaFreeHost(h0))

    for i in range(gpu_n):
        checkCudaErrors(cudart.cudaSetDevice(i))


def test_interop_graphExec():
```

```
    err_dr, = cuda.cuInit(0)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
    err_dr, device = cuda.cuDeviceGet(0)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
    err_dr, ctx = cuda.cuCtxCreate(0, device)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
    err_dr, graph = cuda.cuGraphCreate(0)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
    err_dr, node = cuda.cuGraphAddEmptyNode(graph, [], 0)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)

    # DRV to RT
    err_dr, graphExec, errorNode = cuda.cuGraphInstantiate(graph, b'', 0)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
    err_rt, = cudart.cudaGraphExecDestroy(graphExec)
    assert(err_rt == cudart.cudaError_t.cudaSuccess)

    # RT to DRV
    err_rt, graphExec, errorNode = cudart.cudaGraphInstantiate(graph, b'', 0)
    assert(err_rt == cudart.cudaError_t.cudaSuccess)
    err_dr, = cuda.cuGraphExecDestroy(graphExec)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)

    err_rt, = cudart.cudaGraphDestroy(graph)
    assert(err_rt == cudart.cudaError_t.cudaSuccess)
    err_dr, = cuda.cuCtxDestroy(ctx)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)

def test_cudart_cudaStreamGetCaptureInfo():
    # create stream
    err, stream = cudart.cudaStreamCreate()
    assertSuccess(err)

    # validate that stream is not capturing
    err, status, pid = cudart.cudaStreamGetCaptureInfo(stream)
    assertSuccess(err)
    assert(status == cudart.cudaStreamCaptureStatus.cudaStreamCaptureStatusNone)

    # start capture
    err, = cudart.cudaStreamBeginCapture(
        stream, cudart.cudaStreamCaptureMode.cudaStreamCaptureModeGlobal
    )
    assertSuccess(err)

    # validate that stream is capturing now
    err, status, pid = cudart.cudaStreamGetCaptureInfo(stream)
```

```
    assertSuccess(err)
    assert(status == cudart.cudaStreamCaptureStatus.cudaStreamCaptureStatusActive)

    # clean up
    err, pgraph = cudart.cudaStreamEndCapture(stream)
    assertSuccess(err)


def test_ipc():
    BUFFER_SIZE = 1024 * np.uint8().itemsize

    err,  = cudart.cudaSetDevice(0)
    assertSuccess(err)

    host_buffer = np.full(BUFFER_SIZE, 1).astype(np.uint8)

    err, device_buffer_ptr = cudart.cudaMalloc(BUFFER_SIZE)
    assertSuccess(err)

    err, ipc_mem_handle = cudart.cudaIpcGetMemHandle(device_buffer_ptr)
    assertSuccess(err)
    memory_handle_str = base64.b64encode(ipc_mem_handle.reserved).decode('utf-8')

    err, = cudart.cudaMemcpy(device_buffer_ptr, host_buffer,BUFFER_SIZE,
↪  cudart.cudaMemcpyKind.cudaMemcpyHostToDevice)
    assertSuccess(err)

    host_buffer = np.full(BUFFER_SIZE, 0).astype(np.uint8)
    err, = cudart.cudaMemcpy(host_buffer, device_buffer_ptr,BUFFER_SIZE,
↪  cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost)
    assertSuccess(err)

    err, = cudart.cudaFree(device_buffer_ptr)
    assertSuccess(err)

    err, = cudart.cudaDeviceReset()
    assertSuccess(err)

    err_dr, = cuda.cuInit(0)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)

    err_dr, device = cuda.cuDeviceGet(0)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)

    err_dr, ctx = cuda.cuCtxCreate(0, device)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
```

```
    new_mem_hdl = cudart.cudaIpcMemHandle_t()
    new_mem_hdl.reserved = base64.b64decode(memory_handle_str)
    err, devPtr =  cudart.cudaIpcOpenMemHandle(new_mem_hdl,cudart.cudaIpcMemLazyEnablePeerAccess)
    assertSuccess(err)

    host_buffer = np.full(BUFFER_SIZE, 0).astype(np.uint8)
    err, = cudart.cudaMemcpy(host_buffer, devPtr,BUFFER_SIZE,
↪  cudart.cudaMemcpyKind.cudaMemcpyDeviceToHost)
    assertSuccess(err)

    host_buffer = np.full(BUFFER_SIZE, 8).astype(np.uint8)
    err, = cudart.cudaMemcpy(devPtr, host_buffer,BUFFER_SIZE,
↪  cudart.cudaMemcpyKind.cudaMemcpyHostToDevice)
    assertSuccess(err)

    err, = cudart.cudaIpcCloseMemHandle(devPtr)
    assertSuccess(err)

    err, = cudart.cudaDeviceReset()
    assertSuccess(err)

    err_dr, = cuda.cuCtxDestroy(ctx)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
```

使用 Driver API 示例如下：

```
import cuda.cuda as cuda
import cuda.cudart as cudart
import numpy as np

def ASSERT_DRV(err):
    if isinstance(err, cuda.CUresult):
        if err != cuda.CUresult.CUDA_SUCCESS:
            raise RuntimeError('Cuda Error: {}'.format(err))
    elif isinstance(err, cudart.cudaError_t):
        if err != cudart.cudaError_t.cudaSuccess:
            raise RuntimeError('Cudart Error: {}'.format(err))
    else:
        raise RuntimeError('Unknown error type: {}'.format(err))

def test_cuda_memcpy():
    # Init CUDA
    err, = cuda.cuInit(0)
    assert(err == cuda.CUresult.CUDA_SUCCESS)
```

```
    # Get device
    err, device = cuda.cuDeviceGet(0)
    assert(err == cuda.CUresult.CUDA_SUCCESS)

    # Construct context
    err, ctx = cuda.cuCtxCreate(0, device)
    assert(err == cuda.CUresult.CUDA_SUCCESS)

    # Allocate dev memory
    size = int(1024 * np.uint8().itemsize)
    err, dptr = cuda.cuMemAlloc(size)
    assert(err == cuda.CUresult.CUDA_SUCCESS)

    # Set h1 and h2 memory to be different
    h1 = np.full(size, 1).astype(np.uint8)
    h2 = np.full(size, 2).astype(np.uint8)
    assert(np.array_equal(h1, h2) is False)

    # h1 to D
    err, = cuda.cuMemcpyHtoD(dptr, h1, size)
    assert(err == cuda.CUresult.CUDA_SUCCESS)

    # D to h2
    err, = cuda.cuMemcpyDtoH(h2, dptr, size)
    assert(err == cuda.CUresult.CUDA_SUCCESS)

    # Validate h1 == h2
    assert(np.array_equal(h1, h2))

    # Cleanup
    err, = cuda.cuMemFree(dptr)
    assert(err == cuda.CUresult.CUDA_SUCCESS)
    err, = cuda.cuCtxDestroy(ctx)
    assert(err == cuda.CUresult.CUDA_SUCCESS)


def test_interop_stream():
    err_dr, = cuda.cuInit(0)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
    err_dr, device = cuda.cuDeviceGet(0)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
    err_dr, ctx = cuda.cuCtxCreate(0, device)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)

    # DRV to RT
    err_dr, stream = cuda.cuStreamCreate(0)
```

```
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)

    # Event
    err_dr, event0 = cuda.cuEventCreate(0)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
    err_dr, event1 = cuda.cuEventCreate(0)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
    cuda.cuEventRecord(event0, stream)
    cuda.cuEventRecord(event1, stream)
    _, ms = cuda.cuEventElapsedTime(event0, event1)

    err_rt, = cudart.cudaStreamDestroy(stream)
    assert(err_rt == cudart.cudaError_t.cudaSuccess)

    # RT to DRV
    err_rt, stream = cudart.cudaStreamCreate()
    assert(err_rt == cudart.cudaError_t.cudaSuccess)
    err_dr, = cuda.cuStreamDestroy(stream)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)

    err_dr, = cuda.cuCtxDestroy(ctx)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)

def test_interop_event():
    err_dr, = cuda.cuInit(0)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
    err_dr, device = cuda.cuDeviceGet(0)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
    err_dr, ctx = cuda.cuCtxCreate(0, device)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)

    # DRV to RT
    err_dr, event = cuda.cuEventCreate(0)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
    err_rt, = cudart.cudaEventDestroy(event)
    assert(err_rt == cudart.cudaError_t.cudaSuccess)

    # RT to DRV
    err_rt, event = cudart.cudaEventCreate()
    assert(err_rt == cudart.cudaError_t.cudaSuccess)
    err_dr, = cuda.cuEventDestroy(event)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)

    err_dr, = cuda.cuCtxDestroy(ctx)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
```

```
def test_device_Get_attribute():
    err_dr, = cuda.cuInit(0)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
    err_dr, device = cuda.cuDeviceGet(0)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
    err_dr, ctx = cuda.cuCtxCreate(0, device)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)

    err, major =
↪  cuda.cuDeviceGetAttribute(cuda.CUdevice_attribute.CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MAJOR,
↪  device)
    ASSERT_DRV(err)
    err, minor =
↪  cuda.cuDeviceGetAttribute(cuda.CUdevice_attribute.CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MINOR,
↪  device)
    ASSERT_DRV(err)

    err_dr, = cuda.cuCtxDestroy(ctx)
    assert(err_dr == cuda.CUresult.CUDA_SUCCESS)
```

13.1.3 API 参考

当前支持以下 CUDA Python Runtime API：

```
cudart.cudaSetDevice
cudart.cudaMalloc
cudart.cudaMallocHost
cudart.cudaMallocAsync
cudart.cudaMemset
cudart.cudaMemcpy
cudart.cudaMemcpyAsync
cudart.cudaMemcpyKind
cudart.cudaFree
cudart.cudaFreeHost
cudart.cudaFreeAsync
cudart.cudaStream_t
cudart.cudaStreamLegacy
cudart.cudaStreamPerThread
cudart.cudaStreamCreateWithFlags
cudart.cudaStreamNonBlocking
cudart.cudaStreamSynchronize
cudart.cudaStreamCreate
cudart.cudaStreamDestroy
cudart.cudaStreamGetCaptureInfo
```

```
cudart.cudaStreamCaptureStatus
cudart.cudaStreamCaptureMode
cudart.cudaStreamGetCaptureInfo
cudart.cudaStreamBeginCapture
cudart.cudaStreamEndCapture
cudart.cudaGraphInstantiate
cudart.cudaGraphExecDestroy
cudart.cudaGraphDestroy
cudart.cudaGetLastError
cudart.cudaError_t
cudart.cudaGetDeviceCount
cudart.cudaGetDeviceProperties
cudart.cudaDeviceCanAccessPeer
cudart.cudaDeviceEnablePeerAccess
cudart.cudaDeviceReset
cudart.cudaEventBlockingSync
cudart.cudaEventCreateWithFlags
cudart.cudaEventRecord
cudart.cudaEventSynchronize
cudart.cudaEventElapsedTime
cudart.cudaEventDestroy
cudart.cudaIpcGetMemHandle
cudart.cudaIpcMemHandle_t
cudart.cudaIpcMemLazyEnablePeerAccess
cudart.cudaIpcOpenMemHandle
cudart.cudaIpcCloseMemHandle
```

当前支持以下 CUDA Python Driver API：

```
cuda.cuInit
cuda.CUresult
cuda.cuDeviceGet
cuda.cuCtxCreate
cuda.cuCtxDestroy
cuda.cuDeviceGetAttribute
cuda.CUdevice_attribute
cuda.cuMemAlloc
cuda.cuMemcpyHtoD
cuda.cuMemcpyHtoDAsync
cuda.cuMemcpyDtoH
cuda.cuMemcpyDtoHAsync
cuda.cuMemFree
cuda.cuStreamCreate
cuda.cuStreamDestroy
cuda.cuStreamSynchronize
cuda.cuEventCreate
```

```
cuda.cuEventRecord
cuda.cuEventElapsedTime
```

13.2 PyCUDA

PyCUDA 可以让您以 Pythonic 的方式轻松访问天数智算软件栈支持的 CUDA API。在使用 IxRT Python API 进行部署模型时，您可以使用 PyCUDA 来实现数据 H2D、D2H 和同步等操作。

本次发布适配来自 https://github.com/inducer/pycuda。

13.2.1 安装 PyCUDA

向您的应用工程师获取 `.whl` 安装包并使用 **pip3** 命令安装：

```
$ pip3 install pycuda-{v.r.m}+corex.{v.r.m}-cp310-cp310-linux_x86_64.whl
```

13.2.2 使用示例

使用示例如下：

```
import pycuda.autoinit
import pycuda.driver as drv
import pycuda.driver as cuda
import numpy as np
import numpy.linalg as la

from pycuda.compiler import SourceModule


mod = SourceModule(
   """
__global__ void multiply_them(float *dest, float *a, float *b)
{
   const int i = threadIdx.x*blockDim.y + threadIdx.y;
   dest[i] = a[i] * b[i];
}
"""
)

multiply_them = mod.get_function("multiply_them")

shape = (32, 8)
```

```
a = drv.pagelocked_zeros(shape, dtype=np.float32)
b = drv.pagelocked_zeros(shape, dtype=np.float32)
a[:] = np.random.randn(*shape)
b[:] = np.random.randn(*shape)

m = a*b

a_gpu = drv.mem_alloc(a.nbytes)
b_gpu = drv.mem_alloc(b.nbytes)
c_gpu = drv.mem_alloc(b.nbytes)

strm = drv.Stream()
drv.memcpy_htod_async(a_gpu, a, strm)
drv.memcpy_htod_async(b_gpu, b, strm)
strm.synchronize()

drv.memcpy_dtod_async(c_gpu, b_gpu, b.nbytes, strm)

dest = drv.pagelocked_empty_like(a)

start = cuda.Event()
end = cuda.Event()
start.record()
multiply_them(drv.Out(dest), a_gpu, c_gpu, block=shape + (1,), stream=strm)
end.record()
strm.synchronize()
secs = start.time_till(end)*1e-3
print("time: %fs" % (secs))

drv.memcpy_dtoh_async(a, a_gpu, strm)
drv.memcpy_dtoh_async(b, b_gpu, strm)
strm.synchronize()

assert la.norm(dest - m) == 0
```

13.2.3 API 参考

当前支持以下 PyCUDA API。

13.2.3.1 Device

```
pycuda.driver.Device(number)
pycuda.driver.Device.count()
```

```
pycuda.driver.Device.name()
pycuda.driver.Device.make_context()
```

13.2.3.2 Context

```
pycuda.driver.Context.detach()
pycuda.driver.Context.push()
pycuda.driver.Context.pop()
pycuda.driver.Context.get_device()
pycuda.driver.Context.synchronize()
```

13.2.3.3 Memory

```
pycuda.driver.mem_alloc(bytes)
pycuda.driver.to_device(buffer)
pycuda.driver.from_device(devptr, shape, dtype, order='C')
pycuda.driver.from_device_like(devptr, other_ary)
pycuda.driver.pagelocked_empty(shape, dtype, order='C', mem_flags=0)
pycuda.driver.pagelocked_zeros(shape, dtype, order='C', mem_flags=0)
pycuda.driver.pagelocked_empty_like(array, mem_flags=0)
pycuda.driver.pagelocked_zeros_like(array, mem_flags=0)
pycuda.driver.aligned_empty(shape, dtype, order='C', alignment=4096)
pycuda.driver.aligned_zeros(shape, dtype, order='C', alignment=4096)
pycuda.driver.aligned_empty_like(array, alignment=4096)
pycuda.driver.aligned_zeros_like(array, alignment=4096)
```

13.2.3.4 Memory Transfers

```
pycuda.driver.memcpy_htod(dest, src)
pycuda.driver.memcpy_htod_async(dest, src, stream=None)
pycuda.driver.memcpy_dtoh(dest, src)
pycuda.driver.memcpy_dtoh_async(dest, src, stream=None)
pycuda.driver.memcpy_dtod(dest, src, size)
pycuda.driver.memcpy_dtod_async(dest, src, size, stream=None)
```

13.2.3.5 Stream

```
pycuda.driver.Stream(flags=0)
pycuda.driver.Stream.synchronize()
```

13.2.3.6 Event

```
pycuda.driver.Event(flags=0)
pycuda.driver.Event.record(stream=None)
pycuda.driver.Event.synchronize()
pycuda.driver.Event.time_since(event)
pycuda.driver.Event.time_till(event)
```

14 关于 IxRT 使用的高级话题

在本章节中，您将了解如下有关使用 IxRT 的高级话题：

- 使用 IxRT API 自定义搭建网络
- 使用钩子 (Hook) 函数跟踪逐层计算结果
- (面向开发者) 精度对比工具
- 错误处理

14.1 使用 IxRT API 自定义搭建网络

除了使用 ONNX 解析器，您还可以直接使用网络定义 API 从零开始逐层搭建网络。当前支持的网络定义 API 列表，请参考” 附录 3：IxRT 支持的网络定义 API 列表”。

以下示例展示使用 Input、Convolution、Pooling、Activation、ElementWise、Shuffle 和 MatrixMultiply 层搭建 ResNet18 模型。

14.1.1 C++ API

本示例中的全部代码均可在 `.../oss/samples/sampleAPIToResnet/` 中找到。

在开始前，将 PyTorch 模型的权重导出为文件，以便在 C++ 中将权重数据加载为 `std::map<std::string, nvinfer1::Weights>`。

```
import copy
import struct
import time
from os.path import basename, dirname, join
from typing import Any, Dict, Iterable, Tuple, Type

import torch
import torch.fx as fx
import torch.nn as nn


def fuse_conv_bn_eval(conv, bn):
    assert not (conv.training or bn.training), "Fusion only for eval!"
    fused_conv = copy.deepcopy(conv)

    fused_conv.weight, fused_conv.bias = fuse_conv_bn_weights(
        fused_conv.weight,
        fused_conv.bias,
        bn.running_mean,
        bn.running_var,
```

```
        bn.eps,
        bn.weight,
        bn.bias,
    )

    return fused_conv


def fuse_conv_bn_weights(conv_w, conv_b, bn_rm, bn_rv, bn_eps, bn_w, bn_b):
    if conv_b is None:
        conv_b = torch.zeros_like(bn_rm)
    if bn_w is None:
        bn_w = torch.ones_like(bn_rm)
    if bn_b is None:
        bn_b = torch.zeros_like(bn_rm)
    bn_var_rsqrt = torch.rsqrt(bn_rv + bn_eps)

    conv_w = conv_w * (bn_w * bn_var_rsqrt).reshape(
        [-1] + [1] * (len(conv_w.shape) - 1)
    )
    conv_b = (conv_b - bn_rm) * bn_var_rsqrt * bn_w + bn_b

    return torch.nn.Parameter(conv_w), torch.nn.Parameter(conv_b)


def _parent_name(target: str) -> Tuple[str, str]:
    *parent, name = target.rsplit(".", 1)
    return parent[0] if parent else "", name


def replace_node_module(
    node: fx.Node, modules: Dict[str, Any], new_module: torch.nn.Module
):
    assert isinstance(node.target, str)
    parent_name, name = _parent_name(node.target)
    setattr(modules[parent_name], name, new_module)


def fuse(model: torch.nn.Module) -> torch.nn.Module:
    model = copy.deepcopy(model)
    fx_model: fx.GraphModule = fx.symbolic_trace(model)
    modules = dict(fx_model.named_modules())

    for node in fx_model.graph.nodes:
        if (
            node.op != "call_module"
```

```
        ):
            continue
        if (
            type(modules[node.target]) is nn.BatchNorm2d
            and type(modules[node.args[0].target]) is nn.Conv2d
        ):
            if len(node.args[0].users) > 1:  # Output of conv is used by other nodes
                continue
            conv = modules[node.args[0].target]
            bn = modules[node.target]
            fused_conv = fuse_conv_bn_eval(conv, bn)
            replace_node_module(node.args[0], modules, fused_conv)
            node.replace_all_uses_with(node.args[0])
            fx_model.graph.erase_node(node)
    fx_model.graph.lint()
    fx_model.recompile()
    return fx_model


def benchmark(inp, model, iters=20):
    for _ in range(10):
        model(inp)
    begin = time.time()
    for _ in range(iters):
        model(inp)
    return str(time.time() - begin)


if __name__ == "__main__":
    import torchvision

    net = torchvision.models.resnet18(pretrained=True).cuda()
    net.eval()
    fused_net = fuse(net)
    print(fused_net.code)
    inp = torch.randn(10, 3, 224, 224).cuda()
    torch.testing.assert_allclose(fused_net(inp), net(inp))
    print("Unfused time: ", benchmark(inp, net))
    print("Fused time: ", benchmark(inp, fused_net))

    save_path = join(
        dirname(__file__),
        "../../data/resnet18/resnet18_fusebn.wts",
    )
    f = open(save_path, "w")
    f.write("{}\n".format(len(fused_net.state_dict().keys())))
```

```
idx = 0
for k, v in fused_net.state_dict().items():
    print("key: ", k)
    print("value: ", v.shape)
    vr = v.reshape(-1).cpu().numpy()
    f.write("{} {}".format(k, len(vr)))
    idxx = 0
    for vv in vr:
        f.write(" ")
        f.write(struct.pack(">f", float(vv)).hex())
    f.write("\n")
```

1. 首先，创建 IBuilder 和 INetworkDefinition 对象：

```
Logger logger(nvinfer1::ILogger::Severity::kVERBOSE);
auto builder = UniquePtr<nvinfer1::IBuilder>(nvinfer1::createInferBuilder(logger));
const auto explicitBatch = 1U <<
 ↪  static_cast<uint32_t>(nvinfer1::NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
auto network =
 ↪  UniquePtr<nvinfer1::INetworkDefinition>(builder->createNetworkV2(explicitBatch));
```

2. 添加输入层到网络，并传入指定输入张量的名称、数据类型和完整维数：

```
nvinfer1::ITensor* data =
    network->addInput(input_name.c_str(), nvinfer1::DataType::kFLOAT, nvinfer1::Dims{4, {1,
 ↪  3, 224, 224}});
```

3. 添加 Convolution 层到网络，并传入 filter 的大小、权重和偏置数据，设置 stride 和 padding 参数：

```
nvinfer1::IConvolutionLayer* conv1 = network->addConvolutionNd(*data, 64, nvinfer1::Dims{2,
 ↪  {7, 7}}, weight_map["conv1.weight"], weight_map["conv1.bias"]);
assert(conv1);
conv1->setStrideNd(nvinfer1::Dims{2, {2, 2}});
conv1->setPaddingNd(nvinfer1::Dims{2, {3, 3}});
```

Note

权重和偏置数据的类型为 nvinfer1::Weights。

4. 添加 Activation 层到网络，并传入激活函数类型：

```
nvinfer1::IActivationLayer* relu1 = network->addActivation(*conv1->getOutput(0),
 ↪  nvinfer1::ActivationType::kRELU);
assert(relu1);
```

5. 添加 Pooling 层到网络，并传入池化类型和窗口大小，设置 stride 和 padding 参数：

```
nvinfer1::IPoolingLayer* pool1 =
    network->addPoolingNd(*relu1->getOutput(0), nvinfer1::PoolingType::kMAX,
 ↪  nvinfer1::Dims{2, {3, 3}});
```

```
assert(pool1);
pool1->setStrideNd(nvinfer1::Dims{2, {2, 2}});
pool1->setPaddingNd(nvinfer1::Dims{2, {1, 1}});
```

6. 添加 ResNet block 到网络中：

```
nvinfer1::IActivationLayer* relu2 = basicBlock(network, weight_map, *pool1->getOutput(0),
↪  64, 64, 1, "layer1.0.");
nvinfer1::IActivationLayer* relu3 = basicBlock(network, weight_map, *relu2->getOutput(0),
↪  64, 64, 1, "layer1.1.");
nvinfer1::IActivationLayer* relu4 = basicBlock(network, weight_map, *relu3->getOutput(0),
↪  64, 128, 2, "layer2.0.");
nvinfer1::IActivationLayer* relu5 = basicBlock(network, weight_map, *relu4->getOutput(0),
↪  128, 128, 1, "layer2.1.");
nvinfer1::IActivationLayer* relu6 = basicBlock(network, weight_map, *relu5->getOutput(0),
↪  128, 256, 2, "layer3.0.");
nvinfer1::IActivationLayer* relu7 = basicBlock(network, weight_map, *relu6->getOutput(0),
↪  256, 256, 1, "layer3.1.");
nvinfer1::IActivationLayer* relu8 = basicBlock(network, weight_map, *relu7->getOutput(0),
↪  256, 512, 2, "layer4.0.");
nvinfer1::IActivationLayer* relu9 = basicBlock(network, weight_map, *relu8->getOutput(0),
↪  512, 512, 1, "layer4.1.");
```

Note

basicBlock 方法的实现，请参考 .../oss/samples/sampleAPIToResnet/sampleAPIToResnet.cc。

7. 添加第二个 Pooling 层到网络，并传入池化类型和窗口大小，设置 stride 和 padding 参数：

```
nvinfer1::IPoolingLayer* pool2 =
    network->addPoolingNd(*relu9->getOutput(0), nvinfer1::PoolingType::kAVERAGE,
↪  nvinfer1::Dims{2, {7, 7}});
assert(pool2);
pool2->setStrideNd(nvinfer1::Dims{2, {1, 1}});
```

8. 添加 Shuffle 层到网络，并 reshape dimensions 参数 (该 Shuffle 层替代了 Flatten 层)：

```
nvinfer1::IShuffleLayer* flatten = network->addShuffle(*pool2->getOutput(0));
assert(flatten);
flatten->setReshapeDimensions(nvinfer1::Dims{2, {1, 512}});
```

9. 添加 MatrixMultiply 层到网络，注意这里使用 MatrixMultiply 和 ElementWise 层替代 Gemm 层：

```
nvinfer1::IConstantLayer* gemm_weight =
    network->addConstant(nvinfer1::Dims{2, {1000, 512}}, weight_map["fc.weight"]);
nvinfer1::IMatrixMultiplyLayer* matmul =
    network->addMatrixMultiply(*flatten->getOutput(0), nvinfer1::MatrixOperation::kNONE,
↪  *gemm_weight->getOutput(0),
                               nvinfer1::MatrixOperation::kTRANSPOSE);
assert(matmul);
```

10. 添加 ElementWise 层到网络：

```
nvinfer1::IConstantLayer* gemm_bias = network->addConstant(nvinfer1::Dims{2, {1, 1000}},
↪  weight_map["fc.bias"]);
nvinfer1::IElementWiseLayer* gemm_bias_add =
    network->addElementWise(*matmul->getOutput(0), *gemm_bias->getOutput(0),
↪  nvinfer1::ElementWiseOperation::kSUM);
assert(gemm_bias_add);
```

11. 最后标记 gemm_bias_add 层输出为网络的输出。

```
network->markOutput(*gemm_bias_add->getOutput(0));
```

至此，整个网络便搭建完成，参考"(面向开发者) 使用 IxRT 接口进行开发 > C++ API" 了解如何使用该网络构建引擎文件并进行推理。

14.1.2 Python API

本示例中的全部代码均可在 .../oss/samples/python/APIToResnet/ 中找到。

在开始前，将 PyTorch 模型的权重导出为文件，参考"C++ API" 中的步骤即可。

1. 首先，创建 IBuilder 和 INetworkDefinition 对象：

```
IXRT_LOGGER = tensorrt.Logger(tensorrt.Logger.WARNING)
builder = tensorrt.Builder(IXRT_LOGGER)
EXPLICIT_BATCH = 1 << (int)(tensorrt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
network = builder.create_network(EXPLICIT_BATCH)
```

2. 添加输入层到网络，并传入指定输入张量的名称、数据类型和完整维数：

```
data = network.add_input(
    name="input", dtype=model_info["dtype"], shape=model_info["input_shape"]
)
```

3. 添加 Convolution 层到网络，并传入 filter 的大小、权重和偏置数据，设置 stride 和 padding 参数：

```
conv1 = network.add_convolution_nd(
    input=data,
    num_output_maps=64,
    kernel_shape=(7, 7),
    kernel=weight["conv1.weight"],
    bias=weight["conv1.bias"],
)
assert conv1
conv1.stride_nd = (2, 2)
conv1.padding_nd = (3, 3)
```

Note

权重和偏置数据的类型为 np.float32。

4. 添加 Activation 层到网络，并传入激活函数类型：

```
relu1 = network.add_activation(conv1.get_output(0),
                               type=tensorrt.ActivationType.RELU)
assert relu1
```

5. 添加 Pooling 层到网络，并传入池化类型和窗口大小，设置 stride 和 padding 参数：

```
pool1 = network.add_pooling_nd(input=relu1.get_output(0),
                               window_size=(3, 3),
                               type=tensorrt.PoolingType.MAX)
assert pool1
pool1.stride_nd = (2, 2)
pool1.padding_nd = (1, 1)
```

6. 添加 ResNet block 到网络中：

```
relu2 = basic_block(network, weight, pool1.get_output(0), 64, 64, 1, "layer1.0.")
relu3 = basic_block(network, weight, relu2.get_output(0), 64, 64, 1, "layer1.1.")
relu4 = basic_block(network, weight, relu3.get_output(0), 64, 128, 2, "layer2.0.")
relu5 = basic_block(network, weight, relu4.get_output(0), 128, 128, 1, "layer2.1.")
relu6 = basic_block(network, weight, relu5.get_output(0), 128, 256, 2, "layer3.0.")
relu7 = basic_block(network, weight, relu6.get_output(0), 256, 256, 1, "layer3.1.")
relu8 = basic_block(network, weight, relu7.get_output(0), 256, 512, 2, "layer4.0.")
relu9 = basic_block(network, weight, relu8.get_output(0), 512, 512, 1, "layer4.1.")
```

Note

basicBlock 方法的实现，请参考 .../oss/samples/python/APIToResnet/APIToResnet.py。

7. 添加第二个 Pooling 层到网络，并传入池化类型和窗口大小，设置 stride 和 padding 参数：

```
pool2 = network.add_pooling_nd(input=relu9.get_output(0),
                               window_size=(7, 7),
                               type=tensorrt.PoolingType.AVERAGE)
assert pool2
pool2.stride_nd = (1, 1)
```

8. 添加 Shuffle 层到网络，并 reshape dimensions 参数 (该 Shuffle 层替代了 Flatten 层)：

```
flatten = network.add_shuffle(input=pool2.get_output(0))
assert flatten
flatten.reshape_dims = (1, 512)
```

9. 添加 MatrixMultiply 层到网络，注意这里使用 MatrixMultiply 和 ElementWise 层替代 Gemm 层：

```
gemm_weight = network.add_constant(shape=(1000, 512), weights=weight["fc.weight"])
matmul = network.add_matrix_multiply(input0=flatten.get_output(0),
    op0=tensorrt.MatrixOperation.NONE, input1=gemm_weight.get_output(0),
    op1=tensorrt.MatrixOperation.TRANSPOSE)
```

```
assert matmul
```

10. 添加 ElementWise 层到网络：

```
gemm_bias = network.add_constant(shape=(1, 1000), weights=weight["fc.bias"])
gemm_bias_add = network.add_elementwise(matmul.get_output(0), gemm_bias.get_output(0),
                                        tensorrt.ElementWiseOperation.SUM)
assert gemm_bias_add
```

11. 最后标记 gemm_bias_add 层输出为网络的输出。

```
network.mark_output(tensor=gemm_bias_add.get_output(0))
```

至此，整个网络便搭建完成，参考"(面向开发者) 使用 IxRT 接口进行开发 > Python API" 了解如何使用该网络构建引擎文件并进行推理。

14.2 使用钩子 (Hook) 函数跟踪逐层计算结果

IxRT 允许您注册 Hook 函数，进而在运行时调用这些函数。通常情况下，您不会用到这个功能，但也许您需要在推理的过程中检查或调试网络中间层的计算结果，抑或是自定义特定层的输入，那么 Hook 机制将会是您的好帮手。

14.2.1 C++ API

您需要实现一个 Hook 函数，这个函数只有一个参数 `nvinfer1::ExecutionContextInfo const* info`，用于表示当前运行的网络层相关上下文信息。您可以在 `.../samples/sampleResNet/classification.cc` 中的 `TensorRTAPIExecuteWithHook` 函数中学习 Hook 的使用。

下面是一个简单的 Hook 函数，用于将每一层的信息进行打印：

```
void MyHook(nvinfer1::ExecutionContextInfo const* info) {
    cudaDeviceSynchronize();
    std::cout << "Running callback function " << info->hookName << std::endl;
    std::cout << "Running op " << info->opName << std::endl;
    std::cout << "NbInputs " << info->nbInputs << std::endl;
    std::cout << "NbOutputs " << info->nbOutputs << std::endl;
}
```

想要定义 Hook 函数，请记得第一行首先进行同步，以确保获取到正确的内存数据。定义好 Hook 函数后，使用 `IExecutionContext::registerHook(AsciiChar const* name, ExecutionHook hook, int32_t flag)` 注册这个函数：

```
context->registerHook("print_basic_info", MyHook,
↪  int32_t(nvinfer1::ExecutionHookFlag::kPRERUN));
```

Note

关于 context 的使用方法，请参考"(面向开发者) 使用 IxRT 接口进行开发 > C++ API" 了解 IxRT 基础推理 API。

- 第一个参数用于表示这个 Hook 函数的名字，每个 Hook 函数使用一个独立的字符串表示
- 第二个参数是刚刚定义的 Hook 函数
- 第三个参数用于指示 Hook 函数的运行时刻，`ExecutionHookFlag::kPRERUN` 表示在层运行之前运行，`ExecutionHookFlag::kPOSTRUN` 表示在层之后运行

您可以参考 `nvinfer1::ExecutionContextInfo` 的定义，对 IxRT 暴露出的中间信息进行您的自定义操作：

```
struct ExecutionContextInfo {
    //! The name of the op being executed
    AsciiChar const* opName;
    //! The type of the op being executed
    LayerType type;
    //! The op type defined in onnx fashion, optional
    AsciiChar const* op_type;
    //! The number of inputs of the layer
    int32_t nbInputs;
    //! The number of outputs of the layer
    int32_t nbOutputs;
    //! The input names of the executed op
    AsciiChar const* const* inputNames;
    //! The output names of the executed op
    AsciiChar const* const* outputNames;
    //! The input tensors of the executed op
    ExecutionContextTensorDesc const* inputTensors;
    //! The output tensors of the executed op
    ExecutionContextTensorDesc const* outputTensors;
    //! The hook being executed
    AsciiChar const* hookName;
};
```

其中，`ExecutionContextTensorDesc` 为 IxRT 的内部 Tensor 表示，定义为：

```
struct ExecutionContextTensorDesc {
    //! device ptr of the tensor
    void const* data;
    //! internal paddings, for better performance if used
    Dims paddings;
    //! dimension after internal paddings
    Dims dims;
    //! data type of the tensor
    DataType type;
    //! data format of the tensor
    TensorFormat format;
    int32_t itemsize;
```

```
    //! for int8, the scale factor
    float* scale;
    //! number of scales, should be a non-negative number
    int32_t nb_scales;
    //! If the tensor is an initializer
    bool is_initializer;
};
```

其中 data 是运行时 GPU 内存地址，您可以选择在运行时访问这一块内存，并修改 (仅限于) 输入内存。**注意，运行时修改输入内存内容仅限于调试阶段使用，请您谨慎使用该选项，请勿在生产环境中使用。**如果您希望修改输入内存，则需要指定 Hook 函数标志包含 int32_t(nvinfer1::ExecutionHookFlag::kPRERUN)。

14.2.2 Python API

IxRT 也支持注册来自 Python 的回调函数，从而可以方便地利用 Python 丰富的第三方生态，比如 numpy。您可以在安装包的 .../samples/python/resnet18/sample_hook.py 中学习 Hook 函数的使用。

和 C++ 类似，您需要首先定义 Hook 函数：

```
import tensorrt
import cuda.cuda as cuda
import cuda.cudart as cudart

def my_callback(info: tensorrt.ExecutionContextInfo):
    cudart.cudaDeviceSynchronize()
    print("Layer Type: ", info.layer_type)
    print("OpType: ", info.op_type)
    print("Op: ", info.op_name)
```

想要定义 Hook 函数，请记得第一行首先进行同步，以确保获取到正确的内存数据，接着注册回调函数：

```
context.register_hook("simple_print", my_callback, tensorrt.ExecutionHookFlag.POSTRUN)
```

上面一行代码，将 my_callback 取名为 "sample_print" 注册，并指定在每层运行完之后运行 Hook 函数。借助 Python 的动态语言特性，您可以使用许多方便的特性，比如，在运行时打断点，以检查网络中间层结果：

```
def my_callback(info: tensorrt.ExecutionContextInfo):
    if info.op_name == "Conv_0":
        import pdb; pdb.set_trace()
```

Python 的 Hook 信息 ExecutionContextInfo 定义如下：

```
class ExecutionContextInfo:
  op_name: str
  type: LayerType
  op_type: str
  nb_inputs: int
  nb_outputs: int
  input_names: List[str]
  output_names: List[str]
  input_tensors: List[ExecutionContextTensorDesc]
  output_tensors: List[ExecutionContextTensorDesc]
  hook_name: str
```

其中，ExecutionContextTensorDesc 为 IxRT 的内部 Tensor 表示，定义为：

```
class ExecutionContextTensorDesc:
  data: int
  paddings: List[int]
  dims: List[int]
  type: DatatType
  format: TensorFormat
  itemsize: int
  scale: List[int]
  is_initializer: bool
```

由于传递给 Hook 函数的是一个处于 GPU 的 IxRT Tensor，因此您可以通过一些方便的函数用于将数据拷贝 numpy array：

```
from tensorrt.hook.utils import copy_ixrt_tensor_as_np, copy_ixrt_io_tensors_as_np
def my_view_tensor_callback(info):
    # 直接得到一个 IxRT 内部表示的 Tensor
    np_array = copy_ixrt_tensor_as_np(info.input_tensors[0], ort_style=False)
    # 转换成一个 kLINEAR、FP32 的标准 Tensor
    np_array = copy_ixrt_tensor_as_np(info.input_tensors[0], ort_style=True)
    # 把所有的输入、输出都拷贝到 CPU，且使用 IxRT 内部表示格式
    np_array = copy_ixrt_io_tensors_as_np(info, ort_style=False)
    # 把所有的输入、输出都拷贝到 CPU，且使用 kLINEAR、FP32 的标准
    np_array = copy_ixrt_io_tensors_as_np(info, ort_style=True)
```

- ort_style 选取为 False 时，您可以更直接访问 IxRT 计算的中间结果
- ort_style 选取为 True 时，您可以更方便地和第三方框架 (如 ONNXRuntime、PyTorch) 进行比较

除此以外，IxRT 还内置了一些 Hook 函数供您使用：

```
from tensorrt.hook import create_hook

print_info_hook = create_hook("print_info")
```

```
inject_input_hook = create_hook("inject_external_input", external_input="path/to/input.npy")

context.register_hook(...)
```

14.3 (面向开发者) 精度对比工具

极少数情况下，您使用 IxRT 的推理结果可能不符预期，导致此结果的原因有多种，比如计算图逻辑错误、插件实现错误，或者 IxRT 内部算子计算错误等。当这种情况出现时，建议您使用 IxRT 提供的精度对比工具，快速找出首先计算错误的节点。精度对比工具的大致原理是：分别计算 IxRT 和第三方框架的逐层输出结果，并对输出结果进行数值比较。这里，使用 ONNXRuntime 作为基础比较框架。

14.3.1 开始之前

精度对比工具集成在 IxRT-EXEC 中，如果您需要使用精度对比相关的内容，需要先安装依赖包：

```
$ pip3 install onnx onnxruntime scipy
```

14.3.2 使用精度对比工具

您只需要对 IxRT-EXEC 多增加一个 **--verify_acc** 参数即可调用精度对比工具，并开始比较精度：

```
$ ixrtexec --onnx model.onnx --verify_acc
```

下图是输出日志的一种可能：

```
105 (from Conv_0) was calculated RIGHT in IxRT, diff_max=0.00045239925, diff_sum=53.33425, cosine_sim=0.9999999403953552
106 (from MaxPool_2) was calculated RIGHT in IxRT, diff_max=0.00045239925, diff_sum=13.502668, cosine_sim=0.9999999403953552
109 (from Conv_3) was calculated RIGHT in IxRT, diff_max=0.00040978193, diff_sum=13.37693, cosine_sim=1
113 (from Conv_5) was calculated RIGHT in IxRT, diff_max=0.0010695457, diff_sum=37.315254, cosine_sim=0.9999999403953552
116 (from Conv_8) was calculated RIGHT in IxRT, diff_max=0.0007882714, diff_sum=20.432032, cosine_sim=0.9999995231628418
120 (from Conv_10) was calculated RIGHT in IxRT, diff_max=0.0025637746, diff_sum=69.31664, cosine_sim=0.9999995231628418
193 (from Conv_15) was calculated RIGHT in IxRT, diff_max=0.002123803, diff_sum=18.745705, cosine_sim=0.9999989867210388
123 (from Conv_13) was calculated RIGHT in IxRT, diff_max=0.0013795346, diff_sum=6.7809424, cosine_sim=0.9999992251396179
129 (from Conv_16) was calculated RIGHT in IxRT, diff_max=0.001935035, diff_sum=17.159203, cosine_sim=0.9999992251396179
132 (from Conv_19) was calculated RIGHT in IxRT, diff_max=0.0018842816, diff_sum=5.784069, cosine_sim=0.9999982118606567
136 (from Conv_21) was calculated RIGHT in IxRT, diff_max=0.0034091473, diff_sum=17.00099, cosine_sim=0.9999987483024597
208 (from Conv_26) was calculated RIGHT in IxRT, diff_max=0.000839293, diff_sum=3.6347923, cosine_sim=0.9999996423721313
139 (from Conv_24) was calculated RIGHT in IxRT, diff_max=0.0019032955, diff_sum=3.760725, cosine_sim=0.9999992847442627
145 (from Conv_27) was calculated RIGHT in IxRT, diff_max=0.005443454, diff_sum=6.640604, cosine_sim=0.9999989867210388
148 (from Conv_30) was calculated RIGHT in IxRT, diff_max=0.0040744543, diff_sum=3.3018556, cosine_sim=0.9999989867210388
152 (from Conv_32) was calculated RIGHT in IxRT, diff_max=0.005547166, diff_sum=7.7737575, cosine_sim=0.9999986886978149
155 (from Conv_35) was calculated RIGHT in IxRT, diff_max=0.0033253265, diff_sum=1.64303, cosine_sim=0.9999976754188538
223 (from Conv_37) was calculated RIGHT in IxRT, diff_max=0.006100893, diff_sum=8.707741, cosine_sim=0.9999989867210388
161 (from Conv_38) was calculated RIGHT in IxRT, diff_max=0.0054211617, diff_sum=2.765906, cosine_sim=0.999994158744812
164 (from Conv_41) was calculated RIGHT in IxRT, diff_max=0.002550736, diff_sum=1.252594, cosine_sim=0.9999954104423523
168 (from Conv_43) was calculated RIGHT in IxRT, diff_max=0.029663444, diff_sum=14.346548, cosine_sim=0.9999947547912598
169 (from GlobalAveragePool_46) was calculated RIGHT in IxRT, diff_max=0.011210799, diff_sum=0.21766344, cosine_sim=0.9999977350234985
170 (from Flatten_47) was calculated RIGHT in IxRT, diff_max=0.011210799, diff_sum=0.21766344, cosine_sim=0.9999977350234985
output (from n::type_output_output) was calculated RIGHT in IxRT, diff_max=0.008860171, diff_sum=2.000992, cosine_sim=0.999997615814209
---------Comparison report---------
Accuracy compare program has finished!
Compared with onnxruntime, result calculated from IxRT is RIGHT
-------------end-------------
```

Figure 6: 输出精度对比结果

输出日志根据节点的执行顺序，展示了输出和 ONNXRuntime 的比较结果。每个输出有三个指标可以供比较，每个指标如果合乎阈值，则在终端输出为绿色，反之则为红色。三个指标均可通过命令行传参指定，三个指标的详细描述如下：

- `cosine_sim`：基于 余弦相似度，您可以通过 **--cosine_sim** 参数指定余弦相似度阈值，默认为 0.999，余弦相似度阈值越接近 1，表明 IxRT 计算结果和比较结果越接近。
- `diff_max`：Tensor 中的最大差值的绝对值，**默认不参与对比**，您可以通过 **--diff_max** 参数开启。此阈值越大，表明 IxRT 计算结果和比较结果的差异越大。
- `diff_sum`：Tensor 差值和的绝对值，**默认不参与对比**，您可以通过 **--diff_sum** 参数开启。此阈值越大，表明 IxRT 计算结果和比较结果的差异越大。

您还可以将 IxRT INT8 计算结果和 ONNXRuntime Float32 计算结果进行比较：

```
$ ixrtexec --onnx model.onnx --verify_acc --precision int8
```

通过上述命令，IxRT-EXEC 会调用 IxRT Deploy 工具对输入的 ONNX 进行随机量化，再产生随机输入送入 IxRT 和 ONNXRuntime，并分别进行对比。

14.3.3 IxRT 使用的 ONNX 包含自定义部分

对于训练框架导出的 ONNX，您可能首先会用量化、图融合等技巧进行加速推理。量化的 ONNX 为 QDQ 格式，图融合涉及修改子图结构、算子合并等。这些都会修改原有的 ONNX，导致在 ONNXRuntime 上无法直接运行。

对于这种情况，您可以通过 IxRT 提供的一个额外参数 **--ort_onnx** 传入原始的 ONNX，并最终提供给 ONNXRuntime 计算 FP32 结果；使用 **--onnx** 参数，可传入自定义的 ONNX，并最终提供给 IxRT 执行推理。

如下命令可以对比 IxRT INT8 与 ONNXRuntime FP32 的计算结果，其中，`model.onnx` 的量化结果是 `model_qdq.onnx`：

```
$ ixrtexec --onnx model_qdq.onnx --ort_onnx model.onnx --verify_acc --precision int8
```

除此以外，您还可以指定 IxRT 直接加载引擎文件输入，ONNXRuntime 加载对应的 ONNX：

```
$ ixrtexec --load_engine model.engine --ort_onnx model.onnx --verify_acc
```

14.4 错误处理

您可以通过继承 `IErrorRecorder` 类并注册进入 IxRT 对象当中，得到不同阶段的错误信息。如果不进行注册，错误信息默认从 Logger 输出。

14.4.1 C++ 使用方法

下面提供一个参考的 ErrorRecorder C++ 实现，详情请参考安装包中 `.../samples/sampleResNet/classfication.cc` 中的 `TensorRTAPILoadEngine()`：

```
class SampleErrorRecorder : public IErrorRecorder
{
    using errorPair = std::pair<ErrorCode, std::string>;
    using errorStack = std::vector<errorPair>;

public:
    SampleErrorRecorder() = default;

    ~SampleErrorRecorder() noexcept override {}
    int32_t getNbErrors() const noexcept final
    {
        return mErrorStack.size();
    }
    ErrorCode getErrorCode(int32_t errorIdx) const noexcept final
    {
        return invalidIndexCheck(errorIdx) ? ErrorCode::kINVALID_ARGUMENT : (*this)
        ↪ [errorIdx].first;
    };
    IErrorRecorder::ErrorDesc getErrorDesc(int32_t errorIdx) const noexcept final
    {
        return invalidIndexCheck(errorIdx) ? "errorIdx out of range." : (*this)
        ↪ [errorIdx].second.c_str();
    }
    bool hasOverflowed() const noexcept final
    {
        return false;
    }

    void clear() noexcept final
    {
        try
        {
            std::lock_guard<std::mutex> guard(mStackLock);
            mErrorStack.clear();
        }
        catch (const std::exception& e)
        {
            throw  e.what();
        }
    };

    bool empty() const noexcept
    {
        return mErrorStack.empty();
    }
```

```
    bool reportError(ErrorCode val, IErrorRecorder::ErrorDesc desc) noexcept final
    {
        try
        {
            std::lock_guard<std::mutex> guard(mStackLock);
            std::cerr << "Error[" << static_cast<int32_t>(val) << "]: " << desc << std::endl;
            mErrorStack.push_back(errorPair(val, desc));
        }
        catch (const std::exception& e)
        {
            throw e.what();
        }
        return true;
    }

    IErrorRecorder::RefCount incRefCount() noexcept final
    {
        return ++mRefCount;
    }
    IErrorRecorder::RefCount decRefCount() noexcept final
    {
        return --mRefCount;
    }

private:
    const errorPair& operator[](size_t index) const noexcept
    {
        return mErrorStack[index];
    }

    bool invalidIndexCheck(int32_t index) const noexcept
    {
        size_t sIndex = index;
        return sIndex >= mErrorStack.size();
    }
    std::mutex mStackLock;

    std::atomic<int32_t> mRefCount{0};

    errorStack mErrorStack;
};     // class SampleErrorRecorder
```

- 该对象的内存是基于引用计数管理的，IxRT 内部每次引用 ErrorRecorder 时，都会调用一次 `incRefCount()`；当 IxRT 对象声明周期结束时，会调用一次 `decRefCount()`。
- 您需要基于引用计数，管理对象的内存释放。
- 错误通过 reportError 由推理引擎传递给 ErrorRecorder，您需要在这里实现记录方式。

- 在多线程场景中，请注意给读写操作加锁。

使用方式举例：

```
SampleErrorRecorder my_error_recorder;
builder->setErrorRecorder(&my_error_recorder);
runtime->setErrorRecorder(&my_error_recorder);
engine->setErrorRecorder(&my_error_recorder);
network->setErrorRecorder(&my_error_recorder);
context->setErrorRecorder(&my_error_recorder);
if (auto* er = runtime->getErrorRecorder()) {
    if (er->getNbErrors()) {
        std::cerr << "Got " << er->getNbErrors() << " errors from IxRT:" << std::endl;
        for (int i = 0; i < er->getNbErrors(); ++i) {
            std::cerr << "Error #" << i << std::endl;
            std::cerr << "Error code:" << int32_t(er->getErrorCode(i)) << std::endl;
            std::cerr << "Error desc:" << er->getErrorDesc(i) << std::endl;
        }
        throw "Failed to load engine";
    }
}
```

14.4.2 Python 使用方法

下面提供一个参考的 ErrorRecorder Python 实现，详情请参考安装包中的 .../samples/python/error_handling/main.py：

```
import tensorrt


class MyErrorRecorder(tensorrt.IErrorRecorder):
    def __init__(self):
        self.errors = []
        super().__init__()

    def report_error(self, val, desc):
        self.errors.append((val, desc))

    def num_errors(self):
        return len(self.errors)

    def get_error_code(self, idx):
        if idx < self.num_errors():
            return self.errors[idx][0]
        else:
```

```
            raise IndexError(
                f"Wrong index {idx}, which is out of [0, {self.num_errors()}]"
            )

    def get_error_desc(self, idx):
        if idx < self.num_errors():
            return self.errors[idx][1]
        else:
            raise IndexError(
                f"Wrong index {idx}, which is out of [0, {self.num_errors()}]"
            )

    def has_overflowed(self):
        return False

    def clear(self):
        self.errors.clear()
```

使用举例：

```
error_recorder = MyErrorRecorder()
runtime.error_recorder = error_recorder
for i in range(error_recorder1.num_errors()):
    print(f"#{i} error code:", error_recorder1.get_error_code(i))
    print(f"#{i} error desc:", error_recorder1.get_error_desc(i))
```

15 附录 1：IxRT 适配的模型列表

以下模型目前已验证 IxRT 推理引擎的适配，模型列表将随验证推进而继续增加。

15.1 AI-Generated Content (AIGC)

15.1.1 Text2Image 类

模型	框架	精度
StableDifussion_V1.5	PyTorch	FP16

15.2 Computer Vision (CV)

15.2.1 Action 类

模型	框架	精度
I3D	ONNX	FP16
TSM	PyTorch	FP16

15.2.2 Classification 类

模型	框架	精度
Deit_s	PyTorch	FP16
EfficientNet	ONNX	INT8
Inception_V3	ONNX	FP16
Inception_Resnet_V2	PyTorch	FP16/INT8
Mobilenet_V2	Caffe	FP16/INT8
ResNet101	ONNX	INT8
ResNet18	ONNX	INT8

模型	框架	精度
ResNet34	ONNX	FP16/INT8
ResNet50	ONNX	FP16/INT8
ResNet50	MMClassification	FP16/INT8
ResNet_V1_D50	ONNX	FP16
SqueezeNet	ONNX	FP16
Swin_Transformer	ONNX	INT8
Swin_Transformer_V2	ONNX	INT8
SwinB	PyTorch	FP16
Vision_Transformer	ONNX	FP16
AlexNet	Torchvision	FP16/INT8
VGG16	ONNX	INT8
GoogLeNet	Torchvision	FP16/INT8
Mobilenet_V3	MMClassification	FP16
Shufflenet_V1	MMClassification	FP16
DenseNet	MMClassification	FP16/INT8
EfficientNet_b0	MMClassification	FP16/INT8
CSPDarkNet50	MMClassification	FP16/INT8
Res2Net50	PyTorch	FP16/INT8
CSPResNet50	MMClassification	FP16/INT8
RepVGG	PyTorch	FP16
EfficientNet_b1	MMClassification	FP16/INT8
HRNet_w18	MMClassification	FP16/INT8
Deit_b	PyTorch	FP16/INT8
ResNet18	Caffe	FP16/INT8
MobileNet_v1	Caffe	FP16/INT8
Inception_v1	Caffe	FP16/INT8
SqueezeNet_v1.1	Caffe	FP16/INT8
SwinS	ONNX	FP16/INT8
SwinT	ONNX	FP16/INT8

模型	框架	精度
RepViT	ONNX	FP16/INT8

15.2.3 Detection 类

模型	框架	精度
Detr	ONNX	FP16
Fcos	MMDetection	FP16
SoloV1	MMDetection	FP16
YOLOV3	Caffe	FP16
YOLOV3	MMDetection	FP16
YOLOV4	ONNX	INT8
YOLOV5m	ONNX	INT8
YOLOV5s	ONNX	FP16/INT8
YOLOV5s	MMDetection	FP16
YOLOV5s_v5.0	ONNX	FP16
YOLOV7_m	PyTorch	INT8
YOLOV7_x	ONNX	FP16
YOLOX_m	ONNX	FP16/INT8
YOLOX_s	ONNX	FP16/INT8
YOLOX_x	ONNX	FP16/INT8
YOLOV3-608	Caffe	FP16/INT8
YOLOV2-416x416	Caffe	FP16/INT8
YOLOV2-1920x1080	Caffe	FP16
YOLOV4_DARKNET	ONNX	INT8
Detr	PyTorch	FP16

15.2.4 Face 类

模型	框架	精度
FaceNet	PyTorch	FP16/INT8
RetinaFace	PyTorch	FP16/INT8

15.2.5 OCR 类

模型	框架	精度
DBnet	PaddleOCR	FP16/INT8
ResNet-BiLSTM	ONNX	FP16
CRNN	ONNX	FP16

15.2.6 Pose 类

模型	框架	精度
HRNet	ONNX	FP16
OpenPose	ONNX	FP16

15.2.7 Segmentation 类

模型	框架	精度
AOT	ONNX	FP16
DDRNet	ONNX	INT8
DeepLab_V3	MMSegmentation	FP16
FCN	ONNX	FP16
PSPNet	ONNX	FP16
UNet	MMSegmentation	FP16
Yolact	ONNX	INT8

15.2.8 Tracking 类

模型	框架	精度
DeepSort	PyTorch	FP16

15.3 Natural Language Processing (NLP)

15.3.1 LanguageModel 类

模型	框架	精度
ChatGLM	PyTorch	FP16
GPT2	PyTorch	FP16
LLaMA_65B	PyTorch	FP16
LLaMA_7B	PyTorch	FP16
Vicuna_7B	PyTorch	FP16

15.3.2 NER 类

模型	框架	精度
ALBert	PyTorch	FP16
Bert	PyTorch	FP16/INT8

15.3.3 Question_Answering 类

模型	框架	精度
BertBase	PyTorch	FP16/INT8
BertLarge	PyTorch	FP16

15.3.4 Fill_Blanks 类

模型	框架	精度
BertBase	PyTorch	FP16/INT8

15.3.5 Text_Classification 类

模型	框架	精度
Electra	PyTorch	FP16
Bert	PyTorch	FP16

15.3.6 Text_Matching 类

模型	框架	精度
SBert	PyTorch	FP16

15.3.7 Translation 类

模型	框架	精度
Transformer_LN_Before	PyTorch	FP16
Transformer_LN_After	PyTorch	FP16

15.3.8 Recommendation 类

模型	框架	精度
Wide&Deep	PaddlePaddle	FP16

15.4 Speech

15.4.1 Speaker_Recognition 类

模型	框架	精度
2SSpeed	ONNX	FP16
Ecapa_Tdnn	SpeechBrain	FP16
RepVGG	ONNX	FP16
Res2Net50	ONNX	FP16
ResNetTDF112	ONNX	FP16
TDNN	ONNX	FP16

15.4.2 Speech_Recognition 类

模型	框架	精度
Conformer	WeNet	FP16/INT8
Transformer	SpeechBrain	FP16
Transformer	ONNX	FP16
Transformer	WeNet	FP16
DeepSpeech2	ONNX	FP16
Wenetspeech_Conformer	WeNet	FP16

15.4.3 Speech_Synthesis

模型	框架	精度
Tacotron2	ONNX	FP16

16 附录 2：IxRT 支持的算子列表

IxRT 推理引擎支持的算子包含 ONNX 通用算子和常见模型的自定义算子，详细列表如下 (算子列表可以通过 **ixrtexec --support pipe** 命令生成)：

序号	算子	支持精度
0	Abs	BOOL
1	Acos	BOOL
2	Acosh	BOOL
3	Add	INT8, FP16
4	AlignChannel	INT8, FP16
5	And	INT8, FP16, BOOL
6	ArgMax	INT8, FP16
7	ArgMin	INT8, FP16
8	Asin	BOOL
9	Asinh	BOOL
10	Assertion	BOOL
11	Atan	BOOL
12	Atanh	BOOL
13	AveragePool	INT8, FP16
14	BatchNormalization	INT8, FP16
15	BertLayer	INT8
16	BertPoolerFP16	FP16
17	Cast	INT8, FP32, FP16
18	Ceil	BOOL
19	Clip	INT8, FP16
20	Concat	INT8, FP16
21	ConformerEncoderCTC	INT8
22	ConformerEncoderCTCFp16	FP16
23	Constant	INT8, I64, I32, FP64, FP32, FP16
24	ConstantOfShape	INT8, FP16
25	Conv	INT8, FP32, FP16

序号	算子	支持精度
26	ConvTranspose	INT8, FP16
27	Cos	BOOL
28	Cosh	BOOL
29	DequantizeLinear	INT8, FP16
30	Div	INT8, FP16
31	Einsum	FP16
32	ElectraLnLinearResidual	FP16
33	Elu	INT8, FP16
34	Embedding	FP16
35	Equal	FP32, FP16
36	Erf	FP16
37	Exp	BOOL
38	Expand	INT8, FP32, FP16
39	Fill	INT8
40	Flatten	INT8, FP32, FP16
41	Floor	BOOL
42	Focus	INT8, FP16
43	Gather	INT8, FP32, FP16
44	GatherElements	INT8, FP32, FP16
45	GatherND	INT8, FP32, FP16
46	Gelu	INT8, FP16
47	Gemm	INT8, FP32, FP16
48	GlobalAveragePool	INT8, FP16
49	Greater	FP32, FP16
50	GreaterOrEqual	FP32, FP16
51	GroupNorm	FP16
52	HardSigmoid	INT8, FP16
53	HardSwish	INT8, FP16
54	Identity	INT8, FP32, FP16

序号	算子	支持精度
55	InstanceNormalization	FP16
56	LSTM	FP16
57	LayerNormalization	INT8, FP32, FP16
58	LeakyRelu	INT8, FP16
59	Less	FP32, FP16
60	LessOrEqual	FP32, FP16
61	Linear	FP32
62	Log	BOOL
63	LogSoftmax	FP16
64	MakeMaskByRadio	FP16
65	MatMul	FP16
66	Max	FP16
67	MaxPool	INT8, FP16
68	MemoryTransfer	INT8, I64, I32, FP64, FP32, FP16, BOOL
69	Min	FP16
70	Mish	INT8, FP16
71	Mul	INT8, FP16
72	NMS	INT8, FP32, FP16
73	Neg	BOOL
74	NonZero	INT8, FP32, FP16, BOOL
75	Normalization	FP16
76	Not	BOOL
77	Or	INT8, FP16, BOOL
78	PRelu	INT8, FP16
79	Pad	INT8, FP16
80	PluginV2	INT8, FP32, FP16
81	PosEncodeSinCos	FP16
82	Pow	INT8, FP16
83	QuantizeLinear	INT8

序号	算子	支持精度
84	Range	INT8, I64, I32, FP64, FP32, FP16
85	Reciprocal	BOOL
86	ReduceL1	INT8, FP16
87	ReduceL2	INT8, FP16
88	ReduceLogSum	INT8, FP16
89	ReduceLogSumExp	INT8, FP16
90	ReduceMax	INT8, FP16
91	ReduceMean	INT8, FP16
92	ReduceMin	INT8, FP16
93	ReduceProd	INT8, FP16
94	ReduceSum	INT8, FP16
95	ReduceSumSquare	INT8, FP16
96	Relu	INT8, FP16
97	Reshape	INT8, FP32, FP16
98	Resize	INT8, FP16
99	ReverseMask	I32, FP16
100	Round	BOOL
101	ScaledTanh	INT8, FP16
102	ScatterElements	INT8, FP32, FP16
103	ScatterND	INT8, FP32, FP16
104	Search	FP16
105	Selu	INT8, FP16
106	Shape	INT8, FP16
107	Shuffle	INT8, FP32, FP16
108	Sigmoid	INT8, FP16
109	Sign	BOOL
110	Silu	INT8, FP16
111	Sin	BOOL
112	Sinh	BOOL

序号	算子	支持精度
113	Slice	INT8, FP16
114	Softmax	INT8, FP16
115	Softplus	INT8, FP16
116	Softsign	INT8, FP16
117	Split	INT8, FP16
118	Sqrt	BOOL
119	Square	BOOL
120	Squeeze	INT8, FP16
121	Sub	INT8, FP16
122	TRT_MatMul	INT8, FP16
123	Tan	BOOL
124	Tanh	INT8, FP16
125	TemporalShift	FP16
126	ThresholdedRelu	INT8, FP16
127	Tile	INT8, FP16
128	TopK	FP32, FP16
129	TransformerDecoderGreedySearchFp16	FP16
130	TransformerDecoderGreedySearchI8	INT8
131	TransformerEncoderEmbFp16	FP16
132	TransformerEncoderFp16	FP16
133	Transpose	INT8, FP32, FP16
134	Trilu	INT8, I64, I32, FP64, FP32, FP16, BOOL
135	UnAlignChannel	INT8, FP16
136	Unfold	FP16
137	Unsqueeze	INT8, FP16
138	Where	INT8, FP32, FP16
139	Xor	INT8, FP16, BOOL

17 附录 3：IxRT 支持的网络定义 API 列表

IxRT 推理引擎支持的网络定义 API 列表如下：

	Network Definition API (C++)	Network Definition API (Python)	Layer/Tensor
0	addActivation	add_activation	IActivationLayer
1	addAssertion	add_assertion	IAssertionLayer
2	addCast	add_cast	ICastLayer
3	addConcatenation	add_concatenation	IConcatenationLayer
4	addConstant	add_constant	IConstantLayer
5	addConvolutionNd	add_convolution_nd	IConvolutionLayer
6	addDeconvolutionNd	add_deconvolution_nd	IDeconvolutionLayer
7	addDequantize	add_dequantize	IDequantizeLayer
8	addEinsum	add_einsum	IEinsumLayer
9	addElementWise	add_elementwise	IElementWiseLayer
10	addFullyConnected	add_fully_connected	IFullyConnectedLayer
11	addGather	add_gather	IGatherLayer
12	addGatherV2	add_gatherv2	IGatherLayer
13	addIdentity	add_identity	IIdentityLayer
14	addInput	add_input	ITensor
15	addMatrixMultiply	add_matrix_multiply	IMatrixMultiplyLayer
16	addNormalization	add_normalization	INormalizationLayer
17	addParametricReLU	add_parametric_relu	IParametricReLULayer
18	addPluginV2	add_plugin_v2	IPluginV2Layer
19	addPoolingNd	add_poolingnd	IPoolingLayer
20	addQuantize	add_quantize	IQuantizeLayer
21	addReduce	add_reduce	IReduceLayer
22	addResize	add_resize	IResizeLayer
23	addSelect	add_select	ISelectLayer
24	addShape	add_shape	IShapeLayer
25	addShuffle	add_shuffle	IShuffleLayer

	Network Definition API (C++)	Network Definition API (Python)	Layer/Tensor
26	addSlice	add_slice	ISliceLayer
27	addSoftMax	add_softmax	ISoftMaxLayer
28	addTopK	add_topk	ITopKLayer
29	addUnary	add_unary	IUnaryLayer

18 附录 4：IxRT 支持的插件列表

18.1 插件列表

IxRT 推理引擎支持的插件列表如下：

序号	插件名	版本号
1	ViterbiDecode	1
2	Focus	1
3	FacenetNorm	1
4	RoiAlign	1
5	YolactDecoder	1
6	YoloV3Decode	1
7	YoloV5Decoder	1
8	YoloV7Decoder	1
9	YoloxDecoder	1
10	BertLnResidual	1
11	EcapaPool	1
12	EcapaScorePool	1
13	EcapaAspAttn	1
14	WindowPartition	1
15	WindowReverse	1
16	Cmvn	1
17	Conv2dSubsampling4	1
18	CTCGreedySearch	1
19	Reorg	1

18.2 ViterbiDecode

算子描述

ViterbiDecode 算子实现了维特比解码算法，常用于 CRF (Condition Random Field)，可以和 Bert 搭配，进行 NER (Name Entity Recognition) 任务。该算子有 5 个输入和 1 个输出。

参数信息：支持的精度

支持的精度组合 1：

输入	精度	输出	精度
crf_end_transitions	float32	output	int32
crf_start_transitions	float32		
crf_transitions	float32		
fc_output	float32		
i8_mask	INT8		

支持的精度组合 2：

输入	精度	输出	精度
fc_output	half	output	int32
mask	int32		
crf_start_transitions	half		
crf_transitions	half		
crf_end_transitions	half		

参数信息：支持的排布

输入	数据排布	输出	数据排布
fc_output	kLINEAR	output	kLINEAR
i8_mask	kLINEAR		
crf_start_transitions	kLINEAR		
crf_transitions	kLINEAR		
crf_end_transitions	kLINEAR		

参数信息：支持的形状

输入	形状	输出	形状
i8_mask	(N,L)	output	(N,L)

输入	形状	输出	形状
fc_output	(N, L, H)		
crf_transitions	(7, 7)		
crf_start_transitions	7		
crf_end_transitions	7		

参考 BERT 原论文 了解更多关于该算子的信息。

18.3 Focus

算子描述

Focus 算子实现了数据在内存中的重新排布。该算子有 1 个输入和 1 个输出。

参数信息：属性

类型	属性名	说明
int32	in_c	默认值为 4
int32	out_c	默认值为 16
int32	size	默认值为 2

参数信息：支持的精度

支持的精度组合 1：

输入	精度	输出	精度
input	int8	output	int8

支持的精度组合 2：

输入	精度	输出	精度
input	float16	output	float16

参数信息：支持的排布

输入	数据排布	输出	数据排布
input	kHWC	output	kHWC

参数信息：支持的形状

输入	形状	输出	形状
input	(N,C,H,W)	output	(N,C*4,H/2,W/2)

参考 官方文档 了解更多关于该算子的信息。

18.4 FacenetNorm

算子描述

Facenet Norm 算子是将最后一层的特征图做 L2 正则化的算子 (fixed outdim=512)。该算子有 1 个输入和 1 个输出。

参数信息：属性

类型	属性名	说明
int32	size	size=512

参数信息：支持的精度

支持的精度组合 1：

输入	精度	输出	精度
input	int8	output	float32

支持的精度组合 2：

输入	精度	输出	精度
input	float16	output	float32

参数信息：支持的排布

输入	数据排布	输出	数据排布
input	kHWC	output	kLINEAR

参数信息：支持的形状

输入	形状	输出	形状
input	(N,1,1,512)	output	(N,512)

参考 官方文档 了解更多关于该算子的信息。

18.5 RoiAlign

算子描述

Facenet Norm 算子实现了特征图的提取处理。该算子有 3 个输入和 1 个输出。

参数信息：属性

类型	属性名
int32	output_height
int32	output_width
int32	sampling_ratio
int32	mode
int32	coordinate_transformation_mode
float32	spatial_scale

参数信息：支持的精度

支持的精度组合 1：

输入	精度	输出	精度
input	float16	output	float16
rois	float16		
batch_indices	int32		

支持的精度组合 2：

输入	精度	输出	精度
input	float32	output	float32
rois	float32		
batch_indices	int32		

参数信息：支持的排布

输入	数据排布	输出	数据排布
input	kLINEAR	output	kLINEAR
rois	kLINEAR		
batch_indices	kLINEAR		

参数信息：支持的形状

输入	形状	输出	形状
input	(BS, C, H, W)	output	(nb_rois, C, out_H, out_W)
rois	(nb_rois, 4)		
batch_indices	(nb_rois)		

参考 Mask RCNN 原论文 了解更多关于该算子的信息。

18.6 YolactDecoder

算子描述

YolactDecoder 算子用于处理特征图，生成目标框的算子。该算子有 2 个输入和 1 个输出。

参数信息：属性

类型	属性名
float32*	anchor
int32	num_class
float32	stride

参数信息：支持的精度

支持的精度组合 1：

输入	精度	输出	精度
cls_map	int8	output	float32
bbox_map	int8		

支持的精度组合 2：

输入	精度	输出	精度
cls_map	float16	output	float32
bbox_map	float16		

参数信息：支持的排布

输入	数据排布	输出	数据排布
cls_map	kHWC	output	kLINEAR
bbox_map	kHWC		

参数信息：支持的形状

输入	形状	输出	形状
cls_map	(N, 81*3, H, W)	output	(N, (H * W)*3, 6)
bbox_map	(N, 4*3, H, W)		

参考 官方文档 了解更多关于该算子的信息。

18.7 YoloV3Decode

算子描述

Yolov3 Decoder 算子用于处理特征图，生成目标框的算子。该算子有 1 个输入和 1 个输出。

参数信息：属性

类型	属性名	说明
int32	num_class	目标检测类别数量
int32	stride	特征图大小相对网络输入缩小的倍数
float*	anchor	目标检测使用锚点预设值
int32	faster_impl	目前仅支持 1，表示检测目标仅输出分值最高的类对应的置信度

参数信息：支持的精度

支持的精度组合 1：

输入	精度	输出	精度
feature map	int8	output	float16

支持的精度组合 2：

输入	精度	输出	精度
feature map	float16	output	float16

参数信息：支持的排布

输入	数据排布	输出	数据排布
feature map	kHWC	output	kLINEAR

参数信息：支持的形状

- class_number：检测目标种类
- anchor_number：每个 feature map 点关联的 anchor 数量

输入	形状	输出	形状
feature map	(N, (class_number+5)*anchor_number, H, W)	output	(N, (H*W)*anchor_number, 6)

参考 官方文档 了解更多关于该算子的信息。

18.8 YoloV5Decoder

算子描述

Yolov5 Decoder 算子用于处理特征图，生成目标框的算子。该算子有 1 个输入和 1 个输出。

参数信息：属性

类型	属性名	说明
int32	num_class	目标检测类别数量
int32	stride	特征图大小相对网络输入缩小的倍数
float*	anchor	目标检测使用锚点预设值
int32	faster_impl	目前仅支持 1，表示检测目标仅输出分值最高的类对应的置信度

参数信息：支持的精度

支持的精度组合 1：

输入	精度	输出	精度
feature map	int8	output	float16

支持的精度组合 2：

输入	精度	输出	精度
feature map	float16	output	float16

参数信息：支持的排布

输入	数据排布	输出	数据排布
feature map	kHWC	output	kLINEAR

参数信息：支持的形状

- class_number：检测目标种类
- anchor_number：每个 feature map 点关联的 anchor 数量

输入	形状	输出	形状
feature map	(N, (class_number+5)*anchor_number, H, W)	output	(N, (H*W)*anchor_number, 6)

参考 官方文档 了解更多关于该算子的信息。

18.9 YoloV7Decoder

算子描述

Yolov7 Decoder 算子用于处理特征图，生成目标框的算子。该算子有 1 个输入和 1 个输出。

参数信息：属性

类型	属性名	说明
int32	num_class	目标检测类别数量
int32	stride	特征图大小相对网络输入缩小的倍数
float*	anchor	目标检测使用锚点预设值
int32	faster_impl	目前仅支持 1，表示检测目标仅输出分值最高的类对应的置信度

参数信息：支持的精度

支持的精度组合 1：

输入	精度	输出	精度
feature map	int8	output	float16

支持的精度组合 2：

输入	精度	输出	精度
feature map	float16	output	float16

参数信息：支持的排布

输入	数据排布	输出	数据排布
feature map	kHWC	output	kLINEAR

参数信息：支持的形状

- class_number：检测目标种类
- anchor_number：每个 feature map 点关联的 anchor 数量

输入	形状	输出	形状
feature map	(N, (class_number+5)*anchor_number, H, W)	output	(N, (H*W)*anchor_number, 6)

参考 官方文档 了解更多关于该算子的信息。

18.10 YoloxDecoder

算子描述

Yolox Decoder 算子用于处理特征图，生成目标框的算子。该算子有 3 个输入和 1 个输出。

参数信息：属性

类型	属性名	说明
int32	num_class	目标检测类别数量
int32	stride	特征图大小相对网络输入缩小的倍数
int32	faster_impl	目前仅支持 1，表示检测目标仅输出分值最高的类对应的置信度

参数信息：支持的精度

支持的精度组合 1：

输入	精度	输出	精度
location map	int8	output	float16
confidence map	int8		
classification map	int8		

支持的精度组合 2：

输入	精度	输出	精度
location map	float16	output	float16
confidence map	float16		
classification map	float16		

参数信息：支持的排布

输入	数据排布	输出	数据排布
location map	kHWC	output	kLINEAR
confidence map	kHWC		
classification map	kHWC		

参数信息：支持的形状

- class_number：检测目标种类

输入	形状	输出	形状
location map	(N, 4, H, W)	output	(N, (H*W), 6)
confidence map	(N, 1, H, W)		
classification map	(N, class_number, H, W)		

参考 官方文档 了解更多关于该算子的信息。

18.11 BertLnResidual

算子描述

BertLnResidual 算子融合 LayerNorm 和残差结构。融合后有 4 个输入和 4 个输出。

参数信息：属性

类型	属性名
float32	l0_qkv_in_amax
int32	hidden_size

类型	属性名
int32	head_num
int32	inner_size

参数信息：支持的精度

输入	精度	输出	精度
hidden_states	float16	q_0	int8
ln_weights_first	float16	residual_0	float16
ln_bias_first	float16	ffd1_0	int8
l0_self_attn_out_proj_bias	float16	ffd2_0	int8

参数信息：支持的排布

输入	数据排布	输出	数据排布
hidden_states	kLINEAR	q_0	kLINEAR
ln_weights_first	kLINEAR	residual_0	kLINEAR
ln_bias_first	kLINEAR	ffd1_0	kLINEAR
l0_self_attn_out_proj_bias	kLINEAR	ffd2_0	kLINEAR

参数信息：支持的形状

- class_number：检测目标种类

输入	形状	输出	形状
hidden_states	(N, L, H)	q_0	(N, L, H)
ln_weights_first		residual_0	(N, L, H)
ln_bias_first		ffd1_0	(L1)
l0_self_attn_out_proj_bias		ffd2_0	(L2)

其中，L1 和 L2 的计算方式如下：

```
size1 = 3*N*L*H
size2 = inner_size*N*L
size3 = head_num*N*L*H

L1 = max(size1, size2)
L2 = max(L1, size3)
```

参考 BERT 原论文 了解更多关于该算子的信息。

18.12 EcapaPool

算子描述

EcapaPool 算子在 ECAPA-TDNN 模型中使用，是融合了 pool 之后的融合算子。该算子有 2 个输入和 1 个输出。

参数信息：支持的精度

输入	精度	输出	精度
input	float16	output	float16
length	float32		

参数信息：支持的排布

输入	数据排布	输出	数据排布
input	kHWC	output	kHWC
length	kHWC		

参数信息：支持的形状

输入	形状	输出	形状
input	(N, H, C)	output	(N, H, 1)
length	N		

参考 官方文档 了解更多关于该算子的信息。

18.13 EcapaScorePool

算子描述

EcapaScorePool 在 ECAPA-TDNN 模型中使用，是融合了 pool 之后的融合算子。该算子有 3 个输入和 1 个输出。

参数信息：支持的精度

输入	精度	输出	精度
input	float16	output	float16
score	float16		
length	float32		

参数信息：支持的排布

输入	数据排布	输出	数据排布
input	kHWC	output	kHWC
score	kHWC		
length	kHWC		

参数信息：支持的形状

输入	形状	输出	形状
input	(N, H, W)	output	(N, 2*H, 1)
score	(N, H, W)		
length	N		

参考 官方文档 了解更多关于该算子的信息。

18.14 EcapaAspAttn

算子描述

EcapaAspAttn 在 ECAPA-TDNN 模型中使用，是关于 Attention 的融合算子。该算子有 2 个输入和 1 个输出。

参数信息：支持的精度

输入	精度	输出	精度
input	float16	output	float16
length	float32		

参数信息：支持的排布

输入	数据排布	输出	数据排布
input	kHWC	output	kHWC
length	kHWC		

参数信息：支持的形状

输入	形状	输出	形状
input	(N, H, W)	output	(N, 3*H, W)
length	N		

参考 官方文档 了解更多关于该算子的信息。

18.15 WindowPartition

算子描述

WindowPartition 在 Swint-Transformer 模型中使用，该算子实现了 Windows 的 roll+partition 功能。该算子有 1 个输入和 1 个输出。

参数信息：支持的精度

输入	精度	输出	精度
input	int8	output	int8
input	float16	output	float16
input	float32	output	float32

参数信息：支持的排布

输入	数据排布	输出	数据排布
input	kLinear	output	kLinear

参数信息：支持的形状

输入	形状	输出	形状
input	(N, H, W, C)	output	(number_widnows, window_size, window_size, C)

参考 官方文档 了解更多关于该算子的信息。

18.16 WindowReverse

算子描述

WindowReverse 在 Swint-Transformer 模型中使用，该算子实现了 Windows 的 merge+roll 功能。该算子有 1 个输入和 1 个输出。

参数信息：支持的精度

输入	精度	输出	精度
input	int8	output	int8
input	float16	output	float16
input	float32	output	float32

参数信息：支持的排布

输入	数据排布	输出	数据排布
input	kLinear	output	kLinear

参数信息：支持的形状

输入	形状	输出	形状
input	(number_widnows, window_size, window_size, C)	output	(N, H, W, C)

参考 官方文档 了解更多关于该算子的信息。

18.17 Cmvn

算子描述

Cmvn 实现倒谱均值归一化 (Cepstral Mean Variance Normalization)，用于处理语音识别任务数据。该算子有 3 个输入和 1 个输出。

参数信息：支持的精度

输入	精度	输出	精度
input	float16	output	float16
mean	float16		
variance	float16		

参数信息：支持的排布

输入	数据排布	输出	数据排布
input	kLINEAR	output	kLINEAR
mean	kLINEAR		
variance	kLINEAR		

参数信息：支持的形状

- class_number：检测目标种类
- anchor_number：每个 feature map 点关联的 anchor 数量

输入	形状	输出	形状
cls_map	(N, feat_len, feat_dim)	output	(N, feat_len, feat_dim)
mean	(feat_dim)		
variance	(feat_dim)		

参考 官方文档 了解更多关于该算子的信息。

18.18 Conv2dSubsampling4

算子描述

Conv2dSubsampling4 通过两个 conv2d 算子和一个 linear 算子实现对网络的特征的 4 倍下采样，目前用于 WeNet 模型中，用于进行语音识别任务。该算子有 8 个输入和 3 个输出。

参数信息：属性

类型	属性名	说明
int32	kernel_size_1	第一个 conv 的 kernel 尺寸
int32	kernel_size_2	第二个 conv 的 kernel 尺寸
int32	stride_1	第一个 conv 的 stride
int32	stride_2	第二个 conv 的 stride
int32	odim_1	第一个 conv 的输出通道数
int32	odim_2	第二个 conv 的输出通道数
int32	hidden_size	矩阵乘法输出的 dim

参数信息：支持的精度

输入	精度	输出	精度
input	float16	output	float16
mask	int32	pos_embed	float16
conv1_weight	float16	out_mask	int32
conv1_bias	float32		
conv2_weight	float16		
conv2_bias	float32		
linear_weight	float16		
linear_bias	float16		

参数信息：支持的排布

输入	数据排布	输出	数据排布
input	kLINEAR	output	kLINEAR
mask	kLINEAR	pos_embed	kLINEAR

输入	数据排布	输出	数据排布
conv1_weight	kLINEAR	out_mask	kLINEAR
conv1_bias	kLINEAR		
conv2_weight	kLINEAR		
conv2_bias	kLINEAR		
linear_weight	kLINEAR		
linear_bias	kLINEAR		

参数信息：支持的形状

- new_feat_len = ((feat_len - (kernel_size_1 - 1) - 1) / stride_1 - (kernel_size_2 - 1)) / stride_2 + 1
- new_feat_dim = ((feat_dim- (kernel_size_1 - 1) - 1) / stride_1 - (kernel_size_2 - 1)) / stride_2 + 1

输入	形状	输出	形状
input	(N, feat_len, feat_dim)	output	(N, new_feat_len, hidden_size)
mask	(N, 1, feat_dim)	pos_embed	(1, new_feat_len, hidden_size)
conv1_weight	(odim_1, 2, kernel_size_1, kernel_size_1)	out_mask	(N, new_feat_len)
conv1_bias	odim_1		
conv2_weight	(odim_2, odim_1, kernel_size_2, kernel_size_2)		
conv2_bias	odim_2		
linear_weight	(hidden_size, odim_2*new_feat_dim)		
linear_bias	hidden_size		

参考 官方文档 了解更多关于该算子的信息。

18.19 CTCGreedySearch

算子描述

CTCGreedySearch 算子实现了贪心搜索，目前用于 WeNet 模型中，用于处理语音识别任务数据。该算子有 1 个输入和 1 个输出。

参数信息：属性

类型	属性名	说明
int32	vocab_size	词汇表包含词汇数量
int32	eos_id	用于表示句子结尾的 ID
int32	pad_id	取 Top1 时会避开的 ID

参数信息：支持的精度

输入	精度	输出	精度
input	float16	output	int32

参数信息：支持的排布

输入	数据排布	输出	数据排布
input	kLINEAR	output	kLINEAR

参数信息：支持的形状

输入	形状	输出	形状
input	(N, feat_len, feat_dim)	output	(N, feat_len)

参考 官方文档 了解更多关于该算子的信息。

18.20 Reorg

算子描述

Reorg 算子实现了对特征的重新排布，主要用于 YOLOv2 检测模型。该算子有 1 个输入和 1 个输出。

参数信息：属性

类型	属性名	说明
int32	stride	特征图尺寸缩小的步长

参数信息：支持的精度

支持的精度组合 1：

输入	精度	输出	精度
input	float16	output	float16

支持的精度组合 2：

输入	精度	输出	精度
input	int8	output	int8

参数信息：支持的排布

输入	数据排布	输出	数据排布
input	kLINEAR	output	kLINEAR

参数信息：支持的形状

输入	形状	输出	形状
input	(N, input_channel, input_h, input_w)	output	(N, input_channel * stride * stride, input_h/stride, input_w/stride)

参考 官方文档 了解更多关于该算子的信息。

19 商标声明

- 天数智芯、天数智芯 logo、Iluvatar CoreX 等商标、标识、组合商标为上海天数智芯半导体有限公司之注册商标或商标，受法律保护。
- 除了天数智芯的注册商标外，本内容中使用的其他产品名称及标志仅用于识别目的，该等名称及标志可能是归属于其各自公司的商标。我们否认对该等名称及标志的所有权利。
- CentOS 标识为 Red Hat 公司的商标。
- Docker 为 Docker 公司在美国和其他国家的商标或注册商标。
- Linux 为 Linus Torvalds 在美国和其它国家的注册商标。
- NVIDIA 和 CUDA 为 NVIDIA 公司在美国和/或其它国家的商标和/或注册商标。
- PyTorch 为 Facebook 公司的商标。
- TensorFlow 为 Google 公司的商标。
- Ubuntu 为 Canonical 公司的注册商标。