天数智芯

IGIE 推理框架使用指南

版本：V4.0.1-MR

日期：2024.5.31

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-MR401-UG03-00：2024/5/31

  文档本次发布内容与 V4.0.0-MR 文档相比 (COREX01-MR400-UG03-00)，有以下更新：

  - 更新” 快速上手：IGIE 推理框架使用教程” 章节中的” 教程 1: 如何导入一个模型到 IGIE” 小节，建议用户优先使用 `import_model_to_igie` 接口导入模型

3 IGIE 推理框架功能说明

IGIE (Iluvatar GPU Inference Engine) 是一个针对天数智芯加速卡研发的高性能、高通用、全流程的神经网络推理框架。可为推理场景提供易部署、高吞吐量、低延迟的完整方案。当前提供的 IGIE 版本是 v0.9.1。

3.1 IGIE 推理框架模块

IGIE 主要提供以下功能模块：

- 模型导入：支持将 PyTorch、TensorFlow、ONNX、PaddlePaddle、MXNet 等框架模型导入到 IGIE 中。
- 模型量化：支持 FP16 和 INT8 量化，支持 KL 和 Percentile 量化校准算法，也支持客户使用任何开源量化框架（例如 PyTorch、ONNXRuntime 量化），只要能将量化后模型导出为 ONNX 模型文件即可。
- 自动图优化: 对模型进行自动优化，包括自动算子融合，冗余算子消除，常量折叠，内存重用，layout 转换，自动 padding 等，最终转换为性能更优的 IR Module。
- 编译：将优化好的模型编译成可执行的引擎文件（engine file）, 供模型部署使用。引擎文件 (engine file) 是指通过模型构建的，经过 IGIE 优化的可部署的二进制文件。
- 运行时：用户使用 IGIE 轻量 Runtime，加载并运行引擎文件。
- AutoTune：IGIE 通过 AutoTune 可以自动寻找 ixInfer、ixDNN、ixBLAS、IGIE 算子等生成各种融合算子的最优实现，并应用于模型。
- 多后端支持：支持 CPU、天数智芯加速卡 (Native)、IxRT 等后端的支持。当前 IxRT 兼容 TensorRT API，IGIE 增加了 IxRT 后端的支持，导入模型时可以指定 IxRT 后端，来使用 IxRT 推理引擎编译和运行模型。
- Fallback 机制：支持当 IxRT 后端模型编译失败后自动 Fallback 到原有的 IGIE Backend 执行。

3.2 使用 IGIE 部署模型步骤

当您得到训练好的模型后，使用 IGIE 推理框架的步骤如下：

1. 安装 IGIE 环境
2. 将模型导入 IGIE
3. 对导入模型进行量化，IGIE 还会对量化后的模型自动进行转换和优化
4. 将优化后的模型序列化为引擎文件 (engine file)，用于部署
5. 加载引擎文件，执行推理

3.3 IGIE 功能特性

3.3.1 框架模型的支持

您可直接把主流框架模型直接导入 IGIE。前端框架及算子支持情况如下：

框架	支持算子数量
ONNX	191
PyTorch	230
TensorFlow	174
Caffe 和 Caffe2	53
Paddle	163
TFlite	102
MXNet	255
Keras	70
DarkNet	28

详细算子支持信息请查看" 附录：IGIE 框架算子支持列表"。

3.3.2 精度支持

IGIE 支持以下数据类型：

- FP32
- FP16
- INT8

3.3.3 自动图优化支持

IGIE 图优化模块能够自动地对导入模型进行优化，优化手段主要包括自动算子融合、冗余算子消除、常量折叠、内存重用、layout 转换、自动 padding 等，最终转换为符合天数智芯硬件特性且性能更优的 IR Module，用于最终推理。

具体内容请参考"IGIE 自动图优化介绍"。

3.3.4 INT8 和 FP16 量化支持

目前 IGIE 支持 FP16 量化和 INT8 量化：

- FP16 量化是在 FP32 的基础上对 FP32 数据直接进行截断，无需提供校准数据，因此量化较为简单。
- INT8 量化主要分为两类：训练后量化（Post-Training Quantization，简称 PTQ）和量化感知训练（Quantization-Aware Training, 简称 QAT）。
  - 训练后量化 PTQ，是指将模型中的权重由浮点数量化到低比特整数（当前支持 INT8/INT4），并通过少量校准数据对数据（activation）进行校准量化。

- 量化感知训练 QAT，是指借助用户完整训练数据集，在训练过程中引入量化和反量化算子，通过在训练前向计算中对数据和权重进行伪量化（Q/DeQ），引入量化误差损失，从而在训练过程中提高模型对量化效应的适应能力，提高最终的量化模型精度。

IGIE 内置了训练后量化 PTQ，提供了 FP16 和 INT8 类型的量化，而对于量化感知训练 QAT 可以通过任何其他量化框架（PyTorch、ONNXRuntime quantization）进行 QAT 训练后，再导入到 IGIE。

3.3.5 Runtime 支持

IGIE 提供了一个非常轻量的 runtime 用于模型部署。目前支持 C++ 和 Python 两种 API 接口用于模型部署。

3.3.6 模型多线程多 stream 并发推理

得益于天数智芯加速卡的 Stream 特性，IGIE 可以为每个模型设置不同的 stream，以支持模型在单卡上并发推理。

3.3.7 Pipeline 支持

IGIE 可以作为 pipeline 中的推理组件，嵌入到 pipeline 中运行。

Pipeline 中 IGIE GPU 内存持久化的支持：

- 在视频分析 pipeline 应用中，一般会希望数据尽可能放在 GPU 内存中，即解码后的图像数据一直保存在 GPU 内存中，持续到该帧在 pipeline 的整个生命周期，解码后的数据直接用于推理，以避免内存拷贝带来的开销。
- IGIE 支持将解码后的视频帧通过零拷贝的方式，直接构造 IGIE Tensor 进行模型推理，无需数据搬运，该特性也使得 IGIE 可以作为框架后端进行集成。

3.3.8 IGIE Framework Integrations

将 IGIE 作为框架的后端，部分算子可以使用 IGIE 进行加速。例如：

- TensorFlow-igie
- Torch-igie
- onnxruntime-igie
- paddle-igie

3.3.9 IxRT 后端和 Fallback 机制支持

- 当前 IxRT 兼容 TensorRT API，IGIE 增加了 IxRT 后端的支持，可以指定 IxRT 后端编译和运行模型。
- IGIE Fallback 机制：当使用 ONNX 模型进行推理时，默认使用 IxRT Backend 进行推理。如果 IxRT 不支持该模型，则自动 Fallback 到原有的 IGIE Backend 执行，该过程将自动化完成。

3.3.10 igie-exec 命令行工具

igie-exec 是 IGIE 提供的一套基于命令行调用的模型快速部署和验证工具，方便用户一键运行 FP32/FP16/INT8 模型推理和生成相应的引擎文件。

igie-exec 的具体使用方法请参考”igie-exec 模型快速部署和验证工具使用指南”。

3.3.11 IGIE ModelZoo 支持的模型列表

IGIE ModelZoo 目前支持的模型列表分类如下：

Task	ModelZoo	Nums
Classification	Classification-ModelZoo	89
ObjectDetection	ObjectDetection-ModelZoo	20
SemanticSegmentation	SemanticSegmentation-ModelZoo	3
Trace	Trace-ModelZoo	3
NLP	NLP-ModelZoo	6
Audio	Audio-ModelZoo	2
TotalNum	/	**123**

Note

IGIE ModelZoo 支持的模型列表将持续更新。

4 IGIE 快速上手

本章节将从使用 igie-exec 命令行工具快速验证性能 和使用 Python 脚本端到端推理 两方面的操作说明指导用户快速上手 IGIE。

4.1 使用 igie-exec 命令行工具快速验证性能

igie-exec 工具提供了基于命令行调用的方式，使用一行命令可以直接生成 FP32/FP16/INT8 (使用随机数据集或用户提供的校准数据集进行量化) 的引擎文件，并进行性能测试，可以帮助用户快速验证模型的性能。

Note

引擎文件 (engine file) 是指通过模型构建的、经过 IGIE 优化的可部署的二进制文件。

通过.whl 包安装完 IGIE 后，即可在任意目录下执行 **igie-exec** 命令。

下面是使用 igie-exec 进行 ResNet18 模型的 INT8 推理的示例，模型将从 TorchVision 自动下载，INT8 量化使用随机数据进行校准：

```
$ igie-exec --model_path resnet18 --input input:32,3,224,224 --precision int8

# 推理结果 (MR50 1.0GHz 的设备)
[2023-03-22 12:52:39,986 igie-exec line:64] INFO: Evaluate inference igie time cost...
[2023-03-22 12:52:48,832 igie-exec line:67] INFO: Mean inference time (std dev): 2.438 ms (0.011
↪  ms)
[2023-03-22 12:52:48,833 igie-exec line:69] INFO: Mean fps: 13126.796
```

关于 igie-exec 的参数介绍和更多示例，请参考”igie-exec 模型快速部署和验证工具使用指南”。

4.2 使用 Python 脚本端到端推理

IGIE 提供了非常简单且友好的 Python/C++ 的调用方式，基本上遵循 导入模型 -> INT8 量化（可选）-> 编译 -> 导出引擎文件 -> 推理模型等推理步骤。

下面以 ResNet50 模型为例，展示在 Python 中使用 IGIE 生成 FP32/FP16/INT8 的引擎文件并进行推理的示例。

4.2.1 1. 准备测试模型和测试图片

IGIE 支持导入 PyTorch/ONNX/TensorFlow/PaddlePaddle 等各种框架的模型。此处以 PyTorch 举例，从 TorchVision 下载预训练的 ResNet50 模型：

```
import numpy as np
from PIL import Image

import torch
import torchvision
from torchvision import transforms
from tvm.contrib.download import download_testdata

import ssl
ssl._create_default_https_context = ssl._create_unverified_context

model_name = "resnet50"
model = getattr(torchvision.models, model_name)(pretrained=True).eval()
input_data = torch.randn([1, 3, 224, 224])
scripted_model = torch.jit.trace(model, input_data).eval()

img_url = "https://github.com/dmlc/mxnet.js/blob/main/data/cat.png?raw=true"
img_path = download_testdata(img_url, "cat.png", module="data")
img = Image.open(img_path)

img_preprocess = transforms.Compose([
   transforms.Resize(256),
   transforms.CenterCrop(224),
   transforms.ToTensor(),
   transforms.Normalize(mean=[0.485, 0.456, 0.406],
                        std=[0.229, 0.224, 0.225])])
img = img_preprocess(img)
img = np.expand_dims(img, 0)
print("image shape:", img.shape)
```

测试图片如下：

Figure 1: 推理测试图片

4.2.2 2. FP32/FP16 推理

IGIE 对 FP32/FP16 的切换非常简单，通过 precision 参数就可以自动切换模型的推理精度，并且在编译时选择对应的图优化策略。

对于上述步骤中下载的 ResNet50 模型，IGIE 只需如下步骤就可以完成 FP32/FP16 的引擎文件生成与推理。

```
import tvm
from tvm import relay
from tvm.contrib import graph_executor

# 设置 GPU 设备
target = tvm.target.iluvatar(model="MR", options="-libs=cudnn,cublas,ixinfer")
dev = tvm.device(target.kind.name, 0)

# 导入模型
input_name = "input0"
shape_list = [(input_name, img.shape)]
mod, params = relay.frontend.from_pytorch(scripted_model, shape_list)

# 编译
precision = "fp16" # 设置 precision = "fp32" 就会使用 FP32 推理
engine = relay.build(mod, target=target, params=params, precision=precision)

# 导出引擎文件
```

```
engine.export_library("resnet50.so")
# engine = tvm.runtime.load_module("resnet50.so") # 也可直接导入之前生成的引擎文件

# 推理
m = graph_executor.GraphModule(engine["default"](dev)) # 加载引擎文件到 GPU Device

igie_input = tvm.nd.array(img.astype("float32"))
m.set_input(input_name, igie_input)
m.run()
igie_output = m.get_output(0)
print(igie_output.numpy().shape)
```

验证 IGIE 推理结果，和 PyTorch 推理结果进行比较：

```
# 使用 PyTorch 推理该图片
with torch.no_grad():
    torch_img = torch.from_numpy(img)
    torch_output = model(torch_img)

# Get top-1 result
top1_igie = np.argmax(igie_output.numpy()[0])
top1_torch = np.argmax(torch_output.numpy())

print("IGIE top-1 id: {}.".format(top1_igie))
print("Torch top-1 id: {}.".format(top1_torch))
```

最终输出，IGIE 和 Torch 输出结果一致 (label: 282='tiger cat')：

IGIE top-1 id: 282.

Torch top-1 id: 282.

4.2.3 3. INT8 推理

INT8 推理区别于 FP32 和 FP16 推理，需要对模型进行 INT8 量化。IGIE 内置了量化接口，同时也支持使用第三方量化框架。关于量化的更多详细信息，请参考" 教程 2: 如何对模型进行量化"。

在此示例中，我们使用 IGIE 内置量化接口对导入模型进行 INT8 量化，具体代码如下：

```
import tvm
from tvm import relay
from tvm.contrib import graph_executor

# 设置 GPU 设备
target = tvm.target.iluvatar(model="MR", options="-libs=cudnn,cublas,ixinfer")
dev = tvm.device(target.kind.name, 0)
```

```
# 导入模型
input_name = "input0"
shape_list = [(input_name, img.shape)]
mod, params = relay.frontend.from_pytorch(scripted_model, shape_list)


# 设置校准数据集
def calibrate_dataset(batched_image, input_name):
   calibration_data_list = []
   data = tvm.nd.array(batched_image)
   # 为了方便演示，这里重复使用一张图片组成校准数据集，进行校准
   for i in range(16):
      calibration_data_list.append({input_name: data})
   return calibration_data_list

# 使用 IGIE 内置量化接口量化模型
with relay.quantize.qconfig(calibrate_mode="percentile", weight_scale="max"):
   mod = relay.quantize.quantize(mod, params, dataset=calibrate_dataset(img, input_name))

# 编译
precision = "int8" # 设置 int8，编译时会进行针对 INT8 的专用的图优化
engine = relay.build(mod, target=target, params=params, precision=precision)

# 导出引擎文件
engine.export_library("resnet50.so")
# engine = tvm.runtime.load_module("resnet50.so") # 也可直接导入已生成的引擎文件

# 推理
m = graph_executor.GraphModule(engine["default"](dev)) # 加载引擎文件到 GPU Device

igie_input = tvm.nd.array(img.astype("float32"))
m.set_input(input_name, igie_input)
m.run()
igie_output = m.get_output(0)
print(igie_output.numpy().shape)
```

5 快速上手：IGIE 推理框架使用教程

本教程提供了一系列场景式的使用文档来帮助用户快速熟悉 IGIE，加速深度学习模型推理部署。

5.1 教程 0：如何开始深度学习模型推理部署

开始之前，我们来简单了解下 IGIE 推理部署深度学习模型的基本流程：

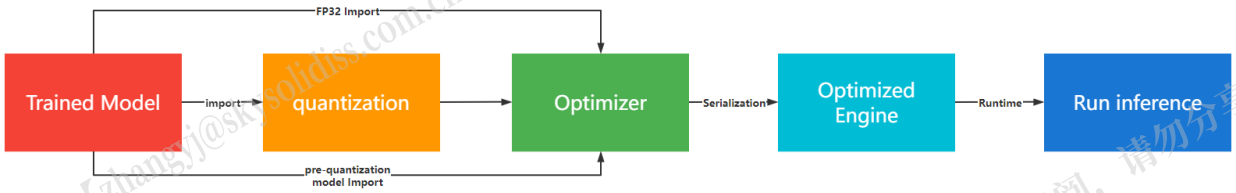

Figure 2: IGIE 推理模型流程

如上图所示，主要经过以下流程：

阶段	描述	教程
模型导入	将训练好的深度学习模型导入到 IGIE。	教程 1: 如何导入一个模型到 IGIE
模型量化	对模型进行量化（可选，只对 INT8 推理适用）。	教程 2: 如何对模型进行量化
自动图优化	对模型进行自动图优化（IGIE 在编译时会自动完成）。	教程 3: 如何自动优化模型并导出引擎文件
导出引擎文件	将优化后的模型导出为引擎文件用于部署。引擎文件 (engine file) 是指通过模型构建的，经过 IGIE 优化的可部署的二进制文件。	教程 3: 如何自动优化模型并导出引擎文件
推理部署	导入引擎文件进行推理部署。	教程 4: 如何加载引擎文件进行推理 (Python) 教程 5: 如何加载引擎文件进行推理 (C++)

5.1.1 接口支持情况

阶段	Python 接口	C++ 接口
模型导入	支持	开发中
模型量化	支持	开发中
自动图优化	支持	整理中
导出引擎文件	支持	整理中
推理部署	支持	支持

5.2 教程 1: 如何导入一个模型到 IGIE

IGIE (Iluvatar GPU Inference Engine) 是针对天数智芯加速卡研发的高性能深度学习推理引擎，通过对模型进行一系列图优化，算子融合、量化、集成多个加速算子库、AutoTune 等优化手段。您可使用 IGIE 进行深度学习模型推理，在天数智芯加速卡上可以达到低延迟、高吞吐的最佳性能。

首先，我们需要将训练好的深度学习模型导入到 IGIE 中，使用 IGIE 对模型做加速优化。

5.2.1 前端框架模型导入 IGIE 的支持情况

目前主流深度学习框架的模型（TensorFlow、PyTorch、ONNX）都可以直接导入 IGIE，无需额外转换到 ONNX 模型。

各个前端框架使用的导入接口如下：

支持框架	支持算子数量	使用导入 API
ONNX	191	relay.frontend.from_onnx()
PyTorch	230	relay.frontend.from_pytorch()
Tensorflow	174	relay.frontend.from_tensorflow()
Caffe/Caffe2	53	relay.frontend.from_caffe()
Paddle	163	relay.frontend.from_paddle()
TFlite	102	relay.frontend.from_tflite()
MXNet	255	relay.frontend.from_mxnet()
Keras	70	relay.frontend.from_keras()
DarkNet	28	relay.frontend.from_darknet()

IGIE 详细的算子支持情况请查看” 附录：IGIE 框架算子支持列表”。

5.2.2 不同框架的模型导入 IGIE 接口使用介绍

下面针对不同框架的模型，介绍如何导入模型到 IGIE。

5.2.2.1 导入 ONNX 模型到 IGIE

导入 ONNX 模型时，需要使用 `onnx.load` 加载 ONNX 模型，然后使用 `relay.frontend.from_onnx` 将模型导入进 IGIE。

```
import tvm
from tvm import relay
import onnx

# 输入模型信息
model_path="./resnet50.onnx"
input_name="input0"
input_shape=[32, 224, 224, 3]

onnx_model = onnx.load(model_path)
shape_dict = {input_name: input_shape}
mod, params = relay.frontend.from_onnx(onnx_model, shape_dict)
print(" 查看模型 graph op: ", mod)
```

参数说明：

- `model_path`：模型的路径。
- `input_name`：模型输入节点的名称。
- `input_shape`：模型输入的 shape。
- `shape_dict`：input_name 和 input_shape 组成的 list。

5.2.2.2 导入 PyTorch 模型到 IGIE

导入 PyTorch 模型时，需要使用 `torch.jit.load` 加载 pt 模型，然后使用 `relay.frontend.from_pytorch` 将模型导入进 IGIE。

以导入 `resnet50.pt` 模型为例：

```
from tvm import relay
import torch

scripted_model = torch.jit.load("./ResNet50.pt").eval()
shape_list = [("input0", ([32, 224, 224, 3], "float32"))]
mod, params = relay.frontend.from_pytorch(scripted_model, shape_list)
print(mod)
```

参数说明：

- scripted_model：导入的 PT 模型。
- shape_list：模型输入名和模型输入的 shape 组成的 list。

5.2.2.3 导入 Tensorflow 模型到 IGIE

导入 TensorFlow 模型时，需要使用 graph_def 加载 pb 模型，然后使用 relay.frontend.from_tensorflow 将模型导入进 IGIE。

```
# 导入 Tensorflow 包
import tensorflow as tf
import tvm.relay.testing.tf as tf_testing

# 使用 graph_def 加载 pb 模型
with tf.io.gfile.GFile(model_path, "rb") as tf_graph:
    graph_def = tf.compat.v1.GraphDef()
    graph_def.ParseFromString(tf_graph.read())
    graph_def = tf_testing.ProcessGraphDefParam(graph_def)

    # 导入 TF graph 到 IGIE
    shape_dict = {input_name: input_shape}
    mod, params = relay.frontend.from_tensorflow(graph_def, shape=shape_dict)
    print(mod)
```

参数说明：

- model_path：模型的路径。
- input_name：模型输入节点的名称。
- input_shape：模型输入的 shape。
- shape_dict：input_name 和 input_shape 组成的 list。

5.2.3 统一导入接口

为了更方便地导入模型，我们整合了所有框架地导入接口，您仅需使用一个 import_model_to_igie 函数即可导入 ONNX, PyTorch, TensorFlow, PaddlePaddle, IxRT 等模型到 IGIE 中。因此，您可以使用不同框架的模型导入 IGIE 接口使用介绍 中介绍的 from_onnx 导入接口指定导入的 ONNX 模型，也可以直接使用 import_model_to_igie 导入任意类型的模型。

建议您优先使用 import_model_to_igie 接口导入模型。

import_model_to_igie 使用方法

```
# 首先导入函数
from tvm.relay.import_model import import_model_to_igie

### 设置模型信息
batch_size = 32
```

```
model_path_or_name = "./resnet50.onnx"
input_dict = {"input": ([batch_size, 3, 224, 224], "float32")}
output_dict = {"output": "float32"}
precision = "fp16"
mod, params = import_model_to_igie(model_path_or_name, inputs_info=input_dict,
↪  outputs_info=output_dict, precision=precision, backend="default")
```

import_model_to_igie 函数说明

使用该接口可以直接导入 ONNX, TensorFlow, PyTorch, TorchVision, PaddlePaddle 模型到 IGIE 中。

```
def import_model_to_igie(model_path_or_name, inputs_info, outputs_info=None, precision=None,
↪  backend="default", dynamic_info=None)
```

参数说明

- `model_path_or_name`：根据自己的模型类型设置 模型路径或 模型目录或 模型名称等字符串。详细示例如下：
  - 模型文件路径：
    * ONNX 模型：例如 `/path/to/resnet50.onnx`
    * PyTorch 模型：例如 `/path/to/resnet50.pt`
    * TensorFlow 模型：例如 `/path/to/resnet50.pb`
  - 模型文件目录：以 目录/模型名称的形式指定 model_path_or_name 参数。例如：
    * PaddlePaddle 模型文件：resnet50/ model。resnet50 目录下需要包含 `resnet50/model.pdiparams`、`model.pdiparams.info` 和 `model.pdmodel` 文件。
  - 模型名称：如果传入参数为模型名称，如 resnet50、vgg16，该接口会自动从 TorchVision 中寻找并加载 torch pre-train 模型。
- `inputs_info`：模型的输入信息。

  期望以 `inputs_info={'name': (shape, dtype)}` 的形式设置 inputs_info。例如：

  ```
  inputs_info = {
        'input0':  ([32, 3, 224, 224], "float32"),
        'input1':  ([32, 256], "int32")
  }
  ```

- `outputs_info`：仅在使用 IxRT Backend 时需要设置，设置模型的输出信息，默认值为 None。

  期望以 `outputs_info={'name': dtype}` 的形式设置 outputs_info。例如：

  ```
  outputs_info={'output0': "float32", 'output1':  "int32"}
  ```

- `precision`：仅在使用 IxRT Backend 时需要设置，设置模型推理精度，默认值为 None。

  例如：`precision="fp16"`。

- `dynamic_info`：仅在使用 IxRT Backend 时需要设置，设置模型的输入为动态 shape，默认值为 None。

  例如：

```
dynamic_info={
            'min_shape': {'input0': ([1, 3, 224, 224], "float32"), 'input1': ([1, 256],
            ↪ "int32")}
            'opt_shape': {'input0': ([32, 3, 224, 224], "float32"), 'input1': ([32,
            ↪ 256], "int32")}
            'max_shape': {'input0': ([128, 3, 224, 224], "float32"), 'input1': ([128,
            ↪ 256], "int32")}
}
```

- backend：直接指定模型推理后端，默认值为 default。
  - backend="default"，表示默认使用 IxRT Backend，IxRT 运行失败后，Fallback 到 IGIE Backend 执行。
  - backend="ixrt"，表示使用 IxRT Backend 执行模型。
  - backend="igie"，表示使用 IGIE Backend 执行模型。

返回值

返回 mod 和 params 两个对象，用来下一步进行优化和编译引擎文件。

- mod：模型结构对象。
- params：模型参数对象。

5.3 教程 2: 如何对模型进行量化

之前我们介绍了" 教程 1: 如何导入模型到 IGIE"，现在我们介绍如何量化模型。

5.3.1 1. 为什么要做量化?

量化是推理引擎常用的性能加速手段，量化是指对模型的权重（weight）和数据（activation）进行低比特处理，让最终生成的网络模型更加轻量化，从而达到节省网络模型存储空间、降低传输时延、提高计算效率，达到性能提升与优化的目标。

一般训练好的模型输入和权重都是 FP32 类型的，将模型量化到更低比特（INT8）后，能够有以下好处：

- 减少 GPU 内存占用量（FP32 4Byte -> INT8 1Byte，内存占用量减少 3/4）。
- 目前很多模型的算子都是 memory bound，量化到 INT8，能大幅度提升访存带宽。
- 天数智芯加速卡为不同数据类型配置了不同的算力，INT8 算力是 FP32 算力的 16 倍。使用 INT8 推理能充分利用 GPU 的算力资源。

5.3.2 2. 使用 IGIE 对模型进行 FP16 量化

FP16 量化是直接对 FP32 数据直接进行截断，在编译构建时，通过指定 precision=fp16 参数可以直接将模型编译为 FP16 的引擎文件。

```
import tvm
from tvm import relay
import onnx

# 输入模型信息
model_path="./resnet50.onnx"
input_name="input0"
input_shape=[32, 224, 224, 3]


# 1) 导入模型到 IGIE
onnx_model = onnx.load(model_path)
shape_list = {input_name: input_shape}
mod, params = relay.frontend.from_onnx(onnx_model, shape_list)

# 2) 设置模型使用 FP16 推理
precision = "fp16"

# 3) 自动优化模型，编译模型为引擎文件
print("Strat build engine...")
engine = relay.build(mod, target=target, params=params, precision=precision)
```

5.3.3 3. 对模型进行 INT8 量化

INT8 量化分为对权重（weight）量化和数据（activation）量化。

- 因为模型的权重已知，可以直接对权重进行量化，统计量化系数。
- 而模型的输入 (feature map) 是未知的，因此需要使用校准数据集：使用一部分校准数据集来代表模型整个推理过程中数据，统计数据分布情况，计算激活数据的量化系数。

为了更好满足用户需求，IGIE 支持两种量化方式：

- 支持使用 IGIE 自身内置的量化框架，对导入后的模型进行量化，对于 CV 类模型有较好效果，但有一定的局限性。目前内置的量化框架暂不支持 Per Channel 量化。
- 支持直接导入已量化好的 ONNX 模型，即：将模型量化好之后再导入到 IGIE。用户可以选择使用任何量化框架，包括 ONNXRuntime Quantization、PyTorch Quantization、TFLite 等开源量化框架，只要能将量化后模型导出为 ONNX 格式即可。

下图是 IGIE 支持的两种量化方式流程图：

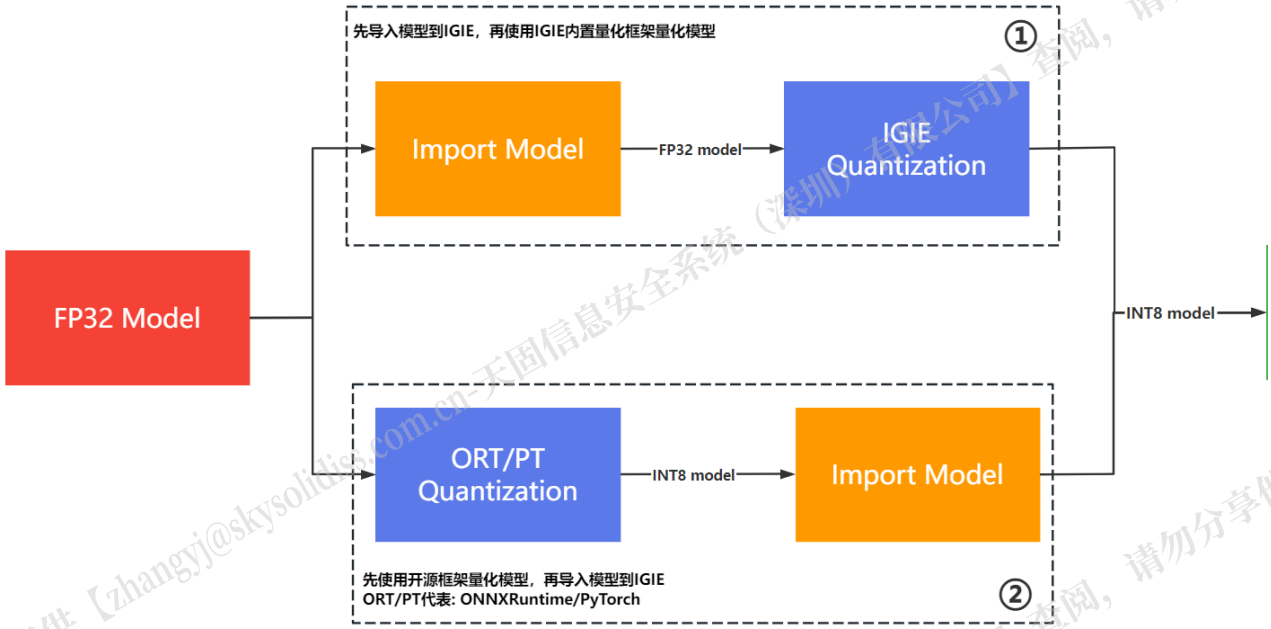

Figure 3: IGIE 支持的量化方式

1）使用 IGIE 自身内置的量化框架进行 INT8 量化：

```
import tvm
from tvm import relay
import onnx

# 1) 导入模型到 IGIE
onnx_model = onnx.load(model_path)
shape_dict = {input_name: input_shape}
mod, params = relay.frontend.from_onnx(onnx_model, shape_dict)

# 2) 对模型进行 INT8 量化
# 使用 16 张图片，作为 ResNet50 INT8 量化校准数据集，
def calibrate_dataset(batched_image, input_name):
    calibration_data_list = []
    data = tvm.nd.array(batched_image)
    for i in range(16):
        calibration_data_list.append({input_name: data})
    return calibration_data_list

# 量化模型
with relay.quantize.qconfig(calibrate_mode="percentile", weight_scale="max"):
    mod = relay.quantize.quantize(mod, params, dataset=calibrate_dataset(batched_image,
↪   input_name))
```

通过对 relay.quantize.qconfig 进行配置，可以控制 IGIE 的量化行为。比如：

- `calibrate_mode`：选择校准算法"percentile" 或者"kl_divergence"。
- `weight_scale`：选择权重的量化方式 max 或者 power2。
- `skip_conv_layers=[]`：选择哪些 conv layer 跳过，不进行 INT8 量化。
- `skip_dense_layer=False`：是否跳过所有的 dense 量化。
- `skip_group_conv_layers=False`：是否跳过所有的 group conv2d 量化。
- `import_scale_file=scale_file_path`：导出并加载量化系数到指定文件。

2）使用深度学习框架自带的开源量化框架将模型量化好后，再导入 IGIE，（框架量化工具例如：ONNXRuntime 的 quantization，PyTorch 的 quantization tools）。

推荐使用 ONNXRuntime 进行量化，IGIE 封装了 ONNXRuntime 的量化实现，可参考 ONNXRuntime 量化官方网站。

5.3.4 4. 量化总结

IGIE 提供了两套量化方案可供用户灵活选择：

- 一个是 IGIE 支持直接导入已量化好的 ONNX 模型，即将模型量化好之后再导入到 IGIE。用户可以选择使用任何量化框架，包括 ONNXRuntime Quantization、PyTorch Quantization、TFLite 等开源量化框架，只要能将量化后模型导出为 ONNX 格式即可。
- 另一个是使用 IGIE 自身内置的量化框架，对导入后的 FP32 模型进行量化，对于 CV 类模型有较好效果，但有一定的局限性，不支持 Per Channel 量化，量化的控制粒度较大。

对于一些检测模型、NLP 模型，对精度要求更高，需要对模型中的部分算子进行更细粒度的量化控制，甚至需要进行 QAT（量化感知训练）。IGIE 支持第三方开源量化框架，只要量化后的模型能够以 ONNX 格式导出，便可以导入到 IGIE，通过这种方式支持更细粒度、更全面的量化控制。

比如：使用 ONNXRuntime 可以控制哪些 op node 做量化，哪些 op node 不做量化。当发现模型 INT8 量化导致精度损失时，推荐 ONNXRunTime 进行量化排查。

总结：

- FP16 量化，建议使用 IGIE 量化，指定 `precision=fp16` 即可。
- INT8 量化，ONNX 模型推荐使用 ONNXRuntime 量化，对 PyTorch/TensorFlow/PaddlePaddle 等框架的模型，推荐使用 IGIE 内置量化。

5.3.5 附录：量化术语介绍

量化是指对模型的权重（weight）和数据（activation）进行低比特处理，让最终生成的网络模型更加轻量化，从而达到节省网络模型存储空间、降低传输时延、提高计算效率，达到性能提升与优化的目标。

量化根据是否需要重训练，分为训练后量化（Post-Training Quantization）和量化感知训练（Quantization-Aware Training）。

量化	训练后量化 PTQ	量化感知训练 QAT
概念	训练后量化（Post-Training Quantizati on，简称 PTQ），是指将训练后模型中的权重由浮点数（当前支持 floa t32）量化到低比特整数（当前支持 int8），并通过少量校准数据基于推理过程对数据（activ ation）进行校准量化。	量化感知训练（Quantization-Aware Training，简称 QAT），是指借助用户完整训练数据集，在训练过程中引入量化操作，通过在训练前向计算中对数据和权重进行伪量化（Q/DeQ），引入量化误差损失，从而在训练过程中提高模型对量化效应的适应能力，提高最终的量化模型精度。
对于激活数据（activation）的量化	需要校准。对于激活数据的量化，只有在推理过程中才知道输入数据的分布情况，因此需要校准的过程：使用一部分推理数据来代表推理过程中整体的数据分布情况，计算激活数据的量化系数。	在模型中加入伪量化层，将待量化层（如卷积）的数据做伪量化处理后，再进行对应的计算（如卷积计算）。在训练过程中，训练伪量化层的参数和模型原有的参数，使得模型在量化的处理下也能达到一个较好的精度。此方法耗时长，但量化精度损失小，适用于做训练后量化精度损失大的场景。
对于权重（weight）的量化	由于是训练后的模型，权重是固定的，因此可以根据权重数据的分布情况直接计算出权重的量化系数。	一样需要在模型中加入伪量化层。

校准

训练后量化场景中，做前向推理获取数据量化因子的过程。由于是训练后的模型，权重是固定的，因此可以根据权重数据的分布情况直接计算出权重的量化因子。对于激活数据的量化，只有在推理过程中才知道输入数据的分布情况，因此需要校准的过程：使用一部分推理数据来代表推理过程中的数据分布情况，进行计算激活数据的量化因子或者系数。

校准数据集

训练后量化场景中，做前向推理使用的数据集。该数据集的分布代表着所有激活数据集的分布，获取校准集时应该具有代表性，推荐使用测试集的子集作为校准数据集。如果数据集不是模型匹配的数据集或者代表性不够，则根据校准集计算得到的量化因子，在全数据集上表现较差，量化损失大，量化后精度低。

5.4 教程 3: 如何自动优化模型并导出引擎文件

之前我们介绍了” 教程 1: 如何导入模型到 IGIE” 和” 教程 2：如何进行模型量化”。

现在介绍如何使用 IGIE 自动优化模型并导出引擎文件。

IGIE 拥有自动图优化机制，能够自动地对导入模型进行优化。主要包括自动算子融合，冗余算子消除，常量折叠，内存重用，layout 转换，自动 padding 等，编译得到符合天数智芯加速卡硬件特性，性能更优的引擎文件，用于最终部署推理。

而这一切的优化都是由 IGIE 自动完成的，用户只需调用 `relay.build` 将导入的模型编译成引擎文件即可。

5.4.1 1. 自动优化模型并编译成引擎文件

```
# 输入模型信息
model_path="./resnet50.onnx"
input_name="input0"
input_shape=[32, 224, 224, 3]

# 1) 导入模型到 IGIE
onnx_model = onnx.load(model_path)
shape_dict = {input_name: input_shape}
mod, params = relay.frontend.from_onnx(onnx_model, shape_dict)

# 2) 设置运行模型的设备信息
target = tvm.target.iluvatar(model="MR", options="-libs=cudnn,cublas,ixinfer")
device = tvm.device(target.kind.name, 0)

# 3) 自动优化模型，编译模型为引擎文件
precision = "fp16"
engine = relay.build(mod, params=params, target=target, precision=precision, verbose=False)

# 4) 查看编译后的 graph function
print(engine.function_metadata["__tvm_main__"])
```

我们通过 `tvm.target.iluvatar` 设置 IGIE 运行的天数智芯目标设备信息。使用 `device = tvm.device(target.kind.name, 0)` 创建 device，指定了 IGIE 运行在第几个 GPU 设备上，这里设置为设备 0。接下来将以上参数传递给 `relay.build` 函数，开始优化和编译模型的引擎文件。

`relay.build` 函数说明

编译和优化模型为引擎文件。

```
def relay.build(mod, parmas=None, target=None, precision="", device=None, verbose=False)
```

参数说明

- `mod`：`import_model_to_igie` 接口返回的模型 module 对象。
- `params`：`import_model_to_igie` 接口返回的模型参数对象。
- `target`：IGIE 运行的天数智芯目标设备信息。

  一般固定设置为 `target = tvm.target.iluvatar(model="MR", options="-libs=cudnn,cublas,ixinfer")`。

- precision：模型运行精度，可设置 fp16 和 int8。
- device：IGIE 运行的 GPU 设备信息，可以指定运行在第几个设备上。

  一般固定设置为 device = tvm.device(target.kind.name, 0)。
- verbose：输出编译 log 信息。如果设置为 True，则保存编译过程 log 文件到当前 module 目录。默认为 False。

返回值

返回优化和编译好的引擎文件对象，用于下一步推理。

engine：模型引擎文件对象。

5.4.2 2. 导出或导入引擎文件

```
# 导出引擎文件
engine.export_library("./resnet50_engine.so")

# 导入引擎文件
engine = tvm.runtime.load_module("./resnet50_engine.so")
```

5.5 教程 4: 如何加载引擎文件进行推理（Python）

在之前" 教程 3：如何自动优化模型并导出引擎文件" 中，介绍了如何自动优化模型并导出引擎文件。现在我们可以直接使用引擎文件用于部署推理。

```
import tvm
import numpy as np
from tvm.contrib import graph_executor

input_name="input0"
input_shape=[32, 224, 224, 3]

# 设置测试 target 和 GPU 设备
target = tvm.target.iluvatar(model="MR", options="-libs=cudnn,cublas,ixinfer")
dev = tvm.device(target.kind.name, 0)

# 加载引擎文件
engine_file = "./resnet50_engine.so"
engine = tvm.runtime.load_module(engine_file)
m = graph_executor.GraphModule(engine["default"](dev)) # 加载引擎文件到 GPU Device

# 设置模型输入
image = np.random.uniform(size=input_shape).astype("float32")
```

```
image = tvm.nd.array(image, dev)
m.set_input(input_name, image)

# 推理模型
m.run()

# 输出推理结果
igie_output = m.get_output(0)
print(igie_output.numpy())
```

5.6 教程 5: 如何加载引擎文件进行推理（C++）

回顾之前几节教程，使用 Python 开发完整的模型推理流程为：

1. 模型导入
2. 模型量化
3. 自动图优化
4. 导出引擎文件
5. 推理部署

目前 IGIE 的 C++ 接口更倾向于用于推理部署。即：

- 先使用 Python 对模型进行量化、导入、自动图优化等操作，最终生成可部署的引擎文件（即第 1 步到第 4 步）。
- 再使用 C++ 接口直接加载生成好的引擎文件进行模型推理（第 5 步）。

这样仅需要一个引擎文件便能使用 C++ 接口快速部署模型，一是方便开发者在 Python 侧调试模型，二是简化模型部署开发的代码量。

C++ 接口使用示例如下：

5.6.1 步骤 1：引入必要的头文件

```
#include <dlpack/dlpack.h>
#include <tvm/runtime/module.h>
#include <tvm/runtime/packed_func.h>
#include <tvm/runtime/registry.h>
#include <cstdio>
```

5.6.2 步骤 2：创建 Target 和 Device

创建一个 Iluvatar Target，用于编译模型使用。

```
auto target = Target("iluvatar")
```

创建 Device，用于指定 op 和 tensor 的运行设备。

```
DLDevice dev{kDLILUVATAR, 0};
```

Target 接收一个字符串来创建 Target，这里使用"iluvatar" 来表示创建 Iluvatar gpu 后端，使用"llvm" 来表示创建 LLVM CPU 后端。
DLDevice 接收两个参数，第一个参数是设备标志符，kDLILUVATAR 代表 Iluvatar GPU 设备, 第 2 个参数是设备 ID，这里设为 GPU 设备 0。

下面是 DLDevice 的结构定义：

```
/*!
 * \brief A Device for Tensor and operator.
 */
typedef struct {
  /*! \brief The device type used in the device. */
  DLDeviceType device_type;
  /*! \brief The device index */
  int device_id;
} DLDevice;
```

5.6.3 步骤 3：创建 Tensor

Tensor 是一个 N 维的数组，主要用于存放模型的输入和输出数据。

1. 使用 tvm::runtime::NDArray::Empty(ShapeTuple shape, DLDataType dtype, Device dev) 创建一个空 Tensor。

   例如：创建一个 shape=[4]，dtype 类型为 float32，在 Iluvatar GPU 上的 Tensor。

   ```
   // 设置 Tensor 位于的设备
   DLDevice dev{kDLILUVATAR, 0};

   // 设置 Tensor 的类型
   DLDataType dtype{kDLFloat, 32, 1};
   tvm::runtime::NDArray input = tvm::runtime::NDArray::Empty({4}, dtype, dev);

   // copy data from cpu to gpu
   float data[4] = {1.0, 2.0, 3.0, 4.0};
   input.CopyFromBytes(data, 4 * sizeof(float));
   ```

2. 使用 tvm::runtime::NDArray::FromDLPack 创建 Tensor。IGIE 支持以零拷贝的方式创建 Tensor：

```
# 该接口支持以零拷贝的方式构造 IGIE Tensor，void *data 可以是 CUDA Memory 数据指针。

tvm::runtime::NDArray CreateNDArray(void *data,
                                    ShapeTuple &shape,
                                    const DLDataType &dtype,
                                    const DLDevice &dev) {
  // 创建 DLTensor
  DLTensor tensor;
  tensor.data = data;
  tensor.device = dev;
  tensor.ndim = static_cast<int>(shape.size());
  tensor.shape = const_cast<int64_t *>(shape.data());
  tensor.dtype = dtype;
  tensor.strides = nullptr;
  tensor.byte_offset = 0;

  // 创建 DLManagerTensor（DLPack 结构体）
  DLManagedTensor managed_tensor;
  managed_tensor.dl_tensor = tensor;
  managed_tensor.deleter = nullptr;

  tvm::runtime::NDArray x_array  = tvm::runtime::NDArray::FromDLPack(&managed_tensor);
  return x_array;
}

// GPU 内存的 data
float *d_data;
cudaMalloc((void**)&d_data, size * sizeof(float));
cudaMemcpy(d_data, h_data, size * sizeof(float), cudaMemcpyHostToDevice);

// 调用 CreateNDArray 函数，零拷贝方式创建 IGIE Tensor，
ShapeTuple in_shape{1, 224, 224, 3};
tvm::runtime::NDArray in_tensor;
in_tensor = CreateNDArray(mag_tensor, d_data, in_shape, DLDataType{kDLFloat, 32, 1}, dev);
```

5.6.4 步骤 4：加载引擎文件

使用 tvm::runtime::Module::LoadFromFile(file_name, format) 加载引擎文件。

```
// 加载引擎文件到 IGIE 中
tvm::runtime::Module mod_factory = tvm::runtime::Module::LoadFromFile("resnet50.so");

// 创建 runtime context
tvm::runtime::Module gmod = mod_factory.GetFunction("default")(dev);
```

5.6.5 步骤 5：获取模型推理函数

使用 gmod.GetFunction(function_name) 获取推理函数。

```
// 设置模型输入函数
tvm::runtime::PackedFunc set_input = gmod.GetFunction("set_input");
// 获取模型输出函数
tvm::runtime::PackedFunc get_output = gmod.GetFunction("get_output");
// 模型运行函数
tvm::runtime::PackedFunc run = gmod.GetFunction("run");
```

5.6.6 步骤 6：设置模型输入

```
set_input("input_0", in_tensor);
```

5.6.7 步骤 7：创建计时器

```
tvm::runtime::Timer timer = tvm::runtime::Timer::GetTimer(dev);
```

5.6.8 步骤 8：运行模型

```
timer->Start();
run();
timer->Stop();
```

5.6.9 步骤 9：获取模型输出并统计耗时

```
float latency = timer->SyncAndGetElapsedNanos() / 1e6;   // ns->ms
float FPS = 1000 / time;

// 创建 Output Tensor
ShapeTuple out_shape{1, 1000};
out_tensor = tvm::runtime::NDArray::Empty(out_shape, DLDataType{kDLFloat, 32, 1}, dev);
get_output(0, out_tensor);
```

5.6.10 IGIE C++ Inference 完整代码示例

Note

IGIE C++ 示例代码不在本发布包中。如有需要，请向您的应用工程师获取示例代码。

下面是完整的 C++ 代码示例：

```
#include <dlpack/dlpack.h>
#include <tvm/runtime/module.h>
#include <tvm/runtime/packed_func.h>
#include <tvm/runtime/registry.h>
#include <tvm/runtime/profiling.h>
#include <cstdio>
#include <stdlib.h>
#include <iostream>
#include <iomanip>
#include <string>
#include <fstream>
#include <errno.h>
#include <exception>

// 读取图片数据到 data
bool GetImageData(std::string file_path, float *data, uint32_t size) {
  try {
    std::ifstream file;
    file.exceptions(file.failbit | file.badbit);
    file.open(file_path);
    if (file.is_open()) {
      float tmp;
      uint32_t count = 0;
      while (file >> tmp) {
        *data++ = tmp;
        count++;
        if (count >= size) {
          break;
        }
      }
      file.close();
      return true;
    }
  } catch (std::ifstream::failure &e) {
    std::cout << "open file failed: " << e.what() << std::endl;
    return false;
  }
  return false;
}
```

```
// 保存推理结果
bool SaveOutput(const std::string &file_path, float *data, uint32_t size) {
  try {
    std::ofstream out_file;
    out_file.exceptions(out_file.failbit | out_file.badbit);
    out_file.open(file_path);
    if (out_file.is_open()) {
      float tmp;
      for (uint32_t i = 0; i < size; ++i) {
        tmp = *data++;
        out_file << tmp;
        out_file << '\n';
      }
      out_file.close();
      return true;
    }
  } catch (std::ifstream::failure &e) {
    std::cout << "open file failed: " << e.what() << std::endl;
    return false;
  }
  return false;
}

// 使用引擎文件部署模型
void DeployResnet50(std::string in_file, std::string out_file, std::string dev_type = "gpu") {
  LOG(INFO) << "Running C++ resnet50 graph executor...";

  // 读取测试图片数据
  float *data = new float[1 * 3 * 224 * 224];
  if (!GetImageData(in_file, data, 1*3*224*224)) {
    std::cout << "Get data failed!" << std::endl;
    return;
  }

  // 设置 device
  DLDevice dev{kDLILUVATAR, 0};
  tvm::runtime::Module mod_factory;

  // 加载引擎文件
  if (dev_type == "cpu") {
    dev.device_type = kDLCPU;
    mod_factory = tvm::runtime::Module::LoadFromFile("model/resnet50_b1_llvm.so");
  } else if (dev_type == "gpu") {
    dev.device_type = kDLILUVATAR;
    mod_factory = tvm::runtime::Module::LoadFromFile("model/resnet50_b1_iluvatar.so");
  }else {
```

```
    std::cout << "ERROR: unrecognized device type!" << std::endl;
    return;
  }

  //  获取模型推理函数
  tvm::runtime::Module gmod = mod_factory.GetFunction("default")(dev);
  tvm::runtime::PackedFunc set_input = gmod.GetFunction("set_input");
  tvm::runtime::PackedFunc get_output = gmod.GetFunction("get_output");
  tvm::runtime::PackedFunc run = gmod.GetFunction("run");

  // 创建空的 Tensor
  tvm::runtime::NDArray input = tvm::runtime::NDArray::Empty({1, 3, 224, 224},
↪  DLDataType{kDLFloat, 32, 1}, dev);
  tvm::runtime::NDArray output_gpu = tvm::runtime::NDArray::Empty({1, 1000},
↪  DLDataType{kDLFloat, 32, 1}, dev);

  // 创建 Timer
  tvm::runtime::Timer timer = tvm::runtime::Timer::GetTimer(dev);

  // Warm Up
  std::cout << "Start Warmup ..." << std::endl;
  for(size_t i = 0; i < 3; i++)
  {
    run();
  }

  // 拷贝 Host data 到 GPU input tensor 中 (HToD)
  input.CopyFromBytes(data, 1 * 3 * 224 * 224 * sizeof(float));

  // 设置模型输入
  set_input("data", input);

  // 推理模型
  timer->Start();
  run();
  timer->Stop();

  // 计算耗时
  float latency = timer->SyncAndGetElapsedNanos() / 1e6;   // ns->ms
  float FPS = 1000 / time;
  std::cout << "Consume time: " << latency << " ms" << std::endl;
  std::cout << "FPS:  " << FPS << std::endl;

  // 获取模型输出
  get_output(0, output_gpu);
```

```
  // 拷贝 GPU Tensor 数据到 Host output 中（DtoH）
  float output[1000];
  output_gpu.CopyToBytes((void*)output, 1000 * sizeof(float));

  for (int32_t i = 0; i < 10; ++i) {
    std::cout << ((float*)output)[i] << '\t';
  }
  std::cout << std::endl;

  // save inference output to postprecess
  SaveOutput(out_file, output, 1000);

  if (data != nullptr) {
    LOG(INFO) << "free data when device type is cuda!!!";
    delete[] data;
  }
  LOG(INFO) << "End running resnet50 graph executor...";
}


int main(int argc, char *argv[]) {
  if (argc != 4) {
    std::cout << "ERROR: two arguments for main, but get " << argc - 1 << std::endl;
    return 1;
  }
  std::string in_file(argv[1]);
  std::string out_file(argv[2]);
  std::string dev_type(argv[3]);

  std::cout << "in_file: " << in_file << ", out_file: " << out_file << std::endl;
  DeployResnet50(in_file, out_file, dev_type);
  return 0;
}
```

执行：

```
$ cd igie/tests/test_iluvatar/test_cpp/
$ bash run.sh
```

5.7 教程 6：如何使用 igie-exec 生成引擎并推理

igie-exec 是 IGIE 提供的一套基于命令行调用的模型快速部署和验证工具，方便用户使用一行命令运行 FP32/FP16/INT8 模型推理和生成相应的引擎文件，并进行性能验证。

当前 igie-exec 内置了 imagenet 和 coco2017 数据集，方便对使用这两种标准数据集的模型进行快速精度测试，而其它模型则使用随机数据集进行性能测试。以下是 igie-exec 的部分使用示例，具体说明请参考”igie-exec 模型快速部署和验证工具使用指南”。

5.7.1 示例 1：运行 PyTorch ResNet50 模型

```
$ igie-exec --model_path data/models/resnet/resnet50-fp32.pt --input input:32,3,224,224 --
 ↪  precision int8 --use_imagenet True

# 执行结果
{'acc@1': 0.7604633482714469,
 'acc@5': 0.9275568181818182,
 'acc_result': 0.7604633482714469,
 'fps_result': 7623.5377158913625}
```

5.7.2 示例 2：运行 YOLOv5 ONNX 模型

```
$ igie-exec --model_path data/models/yolov5/yolov5m.onnx --input images:32,3,640,640 --precision
 ↪  int8 --use_coco2017 True --automatic_yolo_quantization True

# 执行结果
{'acc_result': 0.6236765361287956, 'fps_result': 1094.5590025580452}

# YOLOv5m + GPU NMS 处理
$ igie-exec --model_path data/models/yolov5/yolov5m.onnx --input images:32,3,640,640 --precision
 ↪  int8 --use_coco2017 True --automatic_yolo_quantization True --custom_option with_nms:True
 ↪  iou_thres:0.65 conf_thres:0.001 topk:100
```

5.7.3 示例 3：运行 ONNX BERT-Base Squad 模型

BERT-Base Squad 是一个多输入 NLP 模型，下面展示生成其 FP16 的引擎文件，并使用随机数据进行性能测试的过程。

向您的应用工程师获取数据集。源模型来源于 https://huggingface.co/csarron/bert-base-uncased-squad-v1。

```
# 运行模型
$ igie-exec --model_path data/models/bert/bert-base-uncased-squad-v1.onnx --input input_ids:8,
 ↪  256 attention_mask:8,256 token_type_ids:8,256 --precision fp16

# 执行结果
```

```
[2023-05-18 03:36:24,299 igie-exec line:67] INFO: Evaluate inference igie time cost...
[2023-05-18 03:36:24,638 igie-exec line:70] INFO: Mean inference time (std dev): 9.773 ms (0.016
↪  ms)
[2023-05-18 03:36:24,638 igie-exec line:72] INFO: Mean fps: 818.612
```

5.8 教程 7：IGIE 完整推理说明及示例

5.8.1 1. IGIE Fallback 机制说明

目前 IGIE 已经增加了后端选择的自动 Fallback 机制，支持特性如下：

- 当前 IxRT 兼容 TensorRT API，IGIE 也增加了 IxRT 后端的支持，导入模型时可以指定 IxRT 后端，来使用 IxRT 推理引擎编译和运行模型。
- 当使用 ONNX 模型进行推理时，默认使用 IxRT Backend。如果 IxRT 不支持该模型，则自动 Fallback 到 IGIE Backend 执行，该过程将自动化完成，无需您操作。

Note

当前 IxRT Backend 只支持 ONNX 模型的导入推理，因此上述的 IGIE Fallback 机制只支持 ONNX 模型自动 Fallback，其他类型格式的模型 (如 TensorFlow, PyTorch, PaddlePaddle) 会直接使用 IGIE Backend 执行。

IGIE 增加 Fallback 机制后的最终效果示例如下：

```
# 测试 ResNet50 FP16 推理，默认使用 IxRT Backend 运行
$ python3 inference.py --model_path=resnet50.onnx --input input:32,3,224,224 --precision fp16 --
↪  use_imagenet True

# 测试 ResNet50 INT8 推理，默认使用 IxRT Backend 运行，如果检测到 IxRT 编译该模型失败，自动
↪  Fallback 到 IGIE Backend 执行
$ python3 inference.py --model_path=resnet50.onnx --input input:32,3,224,224 --precision int8 --
↪  use_imagenet True
```

Note

参考2.4 完整推理代码示例 获取上述测试的详细代码。

5.8.2 2. 如何进行推理

IGIE 推理流程分为 导入模型到 `IGIE` -> 优化和编译模型引擎文件 -> 推理模型等步骤。下面我们一一进行使用介绍。

5.8.2.1 2.1 导入模型到 IGIE

当前 IGIE 支持各大主流深度学习框架模型 (如 TensorFlow、PyTorch、ONNX 模型) 的直接导入，在” 教程 1: 如何导入一个模型到 IGIE” 中，我们介绍了各个框架模型的导入方法。

为了更方便地导入模型，我们整合了所有框架地导入接口，您仅需使用一个 import_model_to_igie 函数即可导入 ONNX、PyTorch、TensorFlow、PaddlePaddle、IxRT 等模型到 IGIE。

import_model_to_igie 的使用方法

```
# 首先 import 函数
from tvm.relay.import_model import import_model_to_igie

### 设置模型信息
batch_size = 32
model_path_or_name = "./resnet50.onnx"
input_dict = {"input": ([32, 3, 224, 224], "float32")}
precision = "fp16"

# 使用 IGIE Backend 推理
mod, params = import_model_to_igie(model_path_or_name, inputs_info=input_dict,
 ↪  outputs_info=None, precision=precision, backend="igie")

# 使用 IxRT Backend 推理
output_dict = {"output": "float32"}
mod, params = import_model_to_igie(model_path_or_name, inputs_info=input_dict,
 ↪  outputs_info=output_dict, precision=precision, backend="ixrt")
```

import_model_to_igie 的函数说明

使用该接口，可直接导入 ONNX、TensorFlow、PyTorch、TorchVision、PaddlePaddle 模型到 IGIE 中。

```
def import_model_to_igie(model_path_or_name, inputs_info, outputs_info=None, precision=None,
 ↪  backend="default", dynamic_info=None)
```

参数说明

- model_path_or_name：根据自己的模型类型设置 模型路径或 模型目录或 模型名称等字符串。详细示例如下：
  - 模型文件路径：
    * ONNX 模型：例如 /path/to/resnet50.onnx
    * PyTorch 模型：例如 /path/to/resnet50.pt
    * TensorFlow 模型：例如 /path/to/resnet50.pb
  - 模型文件目录：以 目录/模型名称的形式指定 model_path_or_name 参数。例如：
    * PaddlePaddle 模型文件：resnet50/ model。resnet50 目录下需要包含 resnet50/model.pdiparams、model.pdiparams.info 和 model.pdmodel 文件。
  - 模型名称：如果传入参数为模型名称，如 resnet50、vgg16，该接口会自动从 TorchVision 中寻找并加载 torch pre-train 模型。
- inputs_info：模型的输入信息。

  期望以 inputs_info={'name': (shape, dtype)} 的形式设置 inputs_info。例如：

```
inputs_info = {
        'input0':  ([32, 3, 224, 224], "float32"),
        'input1':  ([32, 256], "int32")
}
```

- outputs_info：仅在使用 IxRT Backend 时需要设置，设置模型的输出信息，默认值为 None。

  期望以 outputs_info={'name': dtype} 的形式设置 outputs_info。例如：

```
outputs_info={'output0': "float32", 'output1':  "int32"}
```

- precision：仅在使用 IxRT Backend 时需要设置，设置模型的推理精度，默认值为 None。

  例如：precision="fp16"。

- dynamic_info：仅在使用 IxRT Backend 时需要设置，设置模型的输入为动态 shape，默认值为 None。

  例如：

```
dynamic_info={
            'min_shape': {'input0':  ([1, 3, 224, 224], "float32"),  'input1':  ([1, 256],
            ↪  "int32")}
            'opt_shape': {'input0':  ([32, 3, 224, 224], "float32"),  'input1':  ([32,
            ↪  256], "int32")}
            'max_shape': {'input0':  ([128, 3, 224, 224], "float32"),  'input1':  ([128,
            ↪  256], "int32")}
}
```

- backend：直接指定模型推理后端，默认值为 default。
  - backend=”default”，表示默认使用 IxRT Backend。IxRT 运行失败后，Fallback 到 IGIE Backend 执行。
  - backend=”ixrt”，表示使用 IxRT Backend 执行模型推理。。
  - backend=”igie”，表示使用原有的 IGIE Backend 执行模型推理。

返回值

返回 mod 和 params 两个对象，用来下一步进行优化和编译引擎文件。

- mod：模型结构对象。
- params：模型参数对象。

5.8.2.2 2.2 优化和编译模型引擎文件

当导入的模型完成后，开始第 2 步，对导入模型进行优化和编译，生成引擎文件。

```
# 1) 导入模型
from tvm.relay.import_model import import_model_to_igie
model_path_or_name = "./resnet50.onnx"
input_dict = {"input": ([32, 3, 224, 224], "float32")}
precision = "fp16"
```

```
mod, params = import_model_to_igie(model_path_or_name, inputs_info=input_dict,
↪  outputs_info=None, precision=precision)

# 2) 设置运行模型的设备信息
target = tvm.target.iluvatar(model="MR", options="-libs=cudnn,cublas,ixinfer")
device = tvm.device(target.kind.name, 0)

# 3) 编译引擎文件
engine = relay.build(mod, params=params, target=target, precision=precision, device=device)
```

首先，我们通过 `tvm.target.iluvatar` 设置 IGIE 运行的天数智芯目标设备信息。使用 `device = tvm.device(target.kind.name, 0)` 创建 device，指定了 IGIE 运行在第几个 GPU 设备上，这里设置为设备 0。

Note

创建 Target 时 options 参数设置：

当使用 IGIE Backend 时，可以通过 `options="-libs=cudnn,cublas,ixinfer"` 参数来选择使用哪些算子库实现(关于 IGIE 使用算子库的详细说明，请查阅" 附录：IGIE 框架算子支持列表")。

而当使用 IxRT Backend 时，该 `options` 参数不会生效。

接下来，将以上参数传递给 `relay.build` 函数，开始优化和编译模型的引擎文件。

relay.build 函数说明

编译和优化模型为引擎文件。

```
def relay.build(mod, parmas=None, target=None, precision="", device=None)
```

参数说明

- `mod`：`import_model_to_igie` 接口返回的模型 module 对象。
- `params`：`import_model_to_igie` 接口返回的模型参数对象。
- `target`：IGIE 运行的天数智芯目标设备信息。

  一般固定设置为 `target = tvm.target.iluvatar(model="MR", options="-libs=cudnn,cublas,ixinfer")`。
- `precision`：模型运行精度，可设置 fp16 和 int8。
- `device`：IGIE 运行的 GPU 设备信息，可以指定运行在第几个设备上。

  一般固定设置为 `device = tvm.device(target.kind.name, 0)`。

返回值

返回优化和编译好的引擎文件对象，用于下一步推理。

`engine`：模型引擎文件对象。

支持导入或导入引擎文件

```
# 导出引擎文件
engine.export_library("./resnet50.engine")
# 导入引擎文件
engine = tvm.runtime.load_module("./resnet50.engine")
```

5.8.2.3 2.3 推理模型

在编译完引擎文件后，可以使用该引擎文件推理模型，推理过程分为 设置模型输入 -> 运行模型 -> 获取模型输出等步骤。使用方法如下：

```
engine = relay.build(mod, params=params, target=target, precision=precision, device=device)
# 加载引擎文件到 GPU 设备
module = graph_executor.GraphModule(engine["default"](device))

print("\n预热模型推理...")
for i in range(3):
  module.run()

print("\n开始推理模型...")
for i in range(10):
  # 设置模型输入
  np_data = np.random.randint(low=0, high=255, size=[batch_size, 224, 224, 3]).astype("float32")
  module.set_input("input", np_data)

  # 推理模型
  module.run()

  # 输出推理结果，以 numpy 数组的形式返回模型输出结果 output
  output = module.get_output(0)
  device.sync()
  print(" 输出推理结果: ", output)
```

首先，通过 `module = graph_executor.GraphModule(engine["default"](device))` 将引擎文件加载到 GPU 设备中，并创建 GraphModule 用于执行模型。

接下来需要使用 `set_input`、`run()` 和 `get_output` 接口来运行模型。

1. set_input() 函数

函数功能

设置模型输入。

当模型有多个输入时，可以多次 `set_input(key, value)`，也可以使用 dict 的形式一次性传入所有输入。

```
def set_input(key=None, value=None, **input_dict)
```

参数说明

- key：模型输入的 name 字符串。
- value：模型输入 name 所对应的 data numpy 数据。
- input_dict：一次性传入所有输入，方式为 key 和 value 组成的字典，例如：

```
input_dict = {'input0': np_data1, 'input1': np_data2}
module.set_input(input_dict)
```

返回值

无返回值。

2. run() 函数

函数功能

运行模型单次推理。

```
def run()
```

参数说明

无参数。

返回值

无返回值。

3. get_output() 函数

函数功能

获取模型单次推理的输出结果。

```
def get_output(index==-1)
```

参数说明

index：返回模型第几个输出。默认为-1，返回所有输出的 list。

返回值

返回模型的输出结果，以 numpy 数组的形式返回。

5.8.2.4 2.4 完整推理代码示例

Note

IGIE C++ 代码示例不在本发布包中。如有需要，请向您的应用工程师获取示例代码。

完成代码参考示例如下 (IGIE 使用 IxRT Backend 测试 ResNet50 模型)：

```
import numpy as np
import os
import sys
from pprint import pprint
from tqdm import tqdm

import tvm
from tvm import relay
from tvm.contrib import graph_executor
from tvm.relay.import_model import import_model_to_igie

# set util_dir python path
util_dir = os.path.dirname(__file__)
sys.path.insert(1, util_dir + "..")

import util
from util import Timer

def main():
    args = util.get_args()
    target, device = util.get_target(args.target)

    print("Run Model args: ")
    pprint(vars(args), indent=2)

    batch_size = args.batch_size
    precision = args.precision

    input_dict = args.input_dict
    model_path_or_name = args.model_path

    if not os.path.isfile(args.engine_path):
        if precision == "int8" and args.model_framework == "onnx":
            model_path_or_name = util.onnx_quantize_model_from_args(args)

        mod, params = import_model_to_igie(model_path_or_name, input_dict, outputs_info=None,
↪   precision=precision)
        lib = relay.build(mod, target=target, params=params, precision=precision, device=device,
↪   verbose=True)
        lib.export_library(args.engine_path)

    lib = tvm.runtime.load_module(args.engine_path)
    module = graph_executor.GraphModule(lib["default"](device))

    print("\nWarm-up...")
    for _ in range(args.warmup):
```

```
        module.run()

    print("\Strat inference...")
    batch_size = args.batch_size
    dataloader = util.get_dataloader_from_args(args)

    total_num = 0
    timer = Timer(device)

    if args.use_imagenet:
        top1_acc = 0
        top5_acc = 0

        for image, label in tqdm(dataloader):
            image = np.array(image)

            module.set_input(args.input_name_list[0], image)

            timer.start()
            module.run()
            timer.stop()

            pred = module.get_output(0)

            if not hasattr(module, "runtime"):
                pred = pred.numpy()

            ## batch accuracy
            batch_top1_acc, batch_top5_acc = util.get_topk_accuracy(pred, label)
            top1_acc += batch_top1_acc
            top5_acc += batch_top5_acc
            total_num += batch_size

            ## batch performance
            batch_infernec_time = timer.last_duration # ms
            batch_infernec_fps = (1000 * batch_size / batch_infernec_time)

            logging.info(f"* Inference Latency: {batch_infernec_time:.3f} ms, Inference FPS:
↪ {batch_infernec_fps:.3f}")
            logging.info(f"* Acc @1 {top1_acc/total_num:.5f} Acc @5 {top5_acc/total_num:.5f}")

if __name__ == "__main__":
    main()
```

执行代码：

```
#! /bin/bash
current_dir=`pwd`

if [[ ! -d ../data ]]; then
    cd ..
    bash prepare_data.sh
    cd ${current_dir}
fi

# 测试 ResNet50 FP16 推理，默认使用 IxRT Backend 运行
$ python3 inference.py --model_path=../data/models/resnet50/resnet50.onnx --input input:32,3,
 ↪  224,224 --precision fp16 --use_imagenet True

# 测试 ResNet50 INT8 推理，默认使用 IxRT Backend 运行，如果检测到 IxRT 编译该模型失败，自动
 ↪  Fallback 到 IGIE Backend 执行
$ python3 inference.py --model_path=../data/models/resnet50/resnet50.onnx --input input:32,3,
 ↪  224,224 --precision int8 --use_imagenet True
```

6 igie-exec 模型快速部署和验证工具使用指南

igie-exec 项目是 IGIE 提供的一套基于命令行调用的模型快速部署和验证工具，方便用户一键运行 FP32/FP16/INT8 模型推理和生成相应的引擎文件。

6.1 igie-exec 功能说明

特性	说明
多框架支持	支持 ONNX、PyTorch、TensorFlow 多框架模型导入
多 batch 支持	支持设置不同 batch 推理测试
多精度支持	支持 FP32、FP16 、INT8 精度推理测试
多个量化框架支持	目前支持两种 INT8 量化方式：IGIE 自身的 INT8 8 量化；使用 ONNXRuntime 进行 INT8 量化（仅支持 ONNX 模型）
引擎文件导出和导入	支持引擎文件导出和导入，方便测试部署
CV 类模型支持	对大多数 CV 类模型 FP32/ FP16/INT8 推理支持较好，部分模型使用 IGIE INT8 量化可能会出错，建议尝试 ONNXRuntime 量化
NLP 模型支持	支持 FP32、FP16 的 NLP 模型，目前适配部分 INT8 NLP 模型，正在增加更多模型适配
动态 shape 支持	目前 igie-exec 对于有动态 shape 的模型需要填充成固定输入进行测试

6.2 igie-exec 的使用指南

6.2.1 开始之前：准备环境

通过安装天数智算软件栈安装 igie-exec：

```
# 使用.run 安装包或 Docker 镜像安装包安装天数智算软件栈
# (仅.run 安装包形式需要) 通过.whl 包安装 IGIE 框架，igie-exec 将随.whl 包同时安装
# 确认是否已成功安装 igie-exec，在任意目录下执行
$ igie-exec -h # igie-exec 代码通常被放在 IGIE 的安装目录下，可以通过 pip3 show igie 查看

# 向您的应用工程师获取数据集 data.tar.gz 并解压，内含 imagenet/coco2017 标准数据集以及 ResNet,
↪  YOLO 等常见模型
$ tar zxf data.tar.gz
```

6.2.2 测试模型推理性能

测试模型推理性能，无需编程，您只需要在任意目录下运行一行命令即可。下面是模型推理示例：

6.2.2.1 示例 1: 运行 PyTorch ResNet50

```
$ igie-exec --model_path data/models/resnet/resnet50-fp32.pt --input input:32,3,224,224 --
 ↪  precision int8 --use_imagenet True

# 执行结果:
执行结果:
{'acc@1': 0.7604633482714469,
 'acc@5': 0.9275568181818182,
 'acc_result': 0.7604633482714469,
 'fps_result': 7623.5377158913625}
```

6.2.2.2 示例 2: 运行 PyTorch TorchVision

对于 PyTorch 模型，igie-exec 也内置了 torchVision.model 来下载预训练模型，只要使用 `--model_path efficientnet_b0` 指定模型名称，即可下载模型进行推理。TorchVision 支持预训练的模型列表可参考 torchVision Models。

- 运行 TorchVision ResNet18 模型

```
$ igie-exec --model_path resnet18  --input input:32,3,224,224 --precision fp16 --
 ↪  use_imagenet True

# 执行结果:
{'acc@1': 0.6070342509603073,
 'acc@5': 0.8355873879641486,
 'acc_result': 0.6070342509603073,
 'fps_result': 19526.346486512382}
```

- 运行 TorchVision EfficientNet 模型：

```
$ igie-exec --model_path efficientnet_b0  --input input:32,3,224,224 --precision fp16 --
 ↪  use_imagenet True

# 执行结果:
执行结果:
{'acc@1': 0.7768285851472471,
 'acc@5': 0.9357394366197183,
```

```
 'acc_result': 0.7768285851472471,
 'fps_result': 2800.5860386306513}
```

6.2.2.3 示例 3: 运行 TensorFlow VGG16

```
$ igie-exec --model_path data/models/vgg/vgg16.pb --input input:32,224,224,3 --precision int8 --
 ↪  use_imagenet True --input_layout NHWC

# 执行结果:
{'acc@1': 0.7006442061459667,
 'acc@5': 0.8942661651728553,
 'acc_result': 0.7006442061459667,
 'fps_result': 4093.361278897936}
```

6.2.2.4 示例 4: 运行 YOLOv3 ONNX

```
$ igie-exec --model_path data/models/yolov3/yolov3_nhwc.onnx --input "input/input_data:0":"32,
 ↪  416,416,3" --precision int8 --use_coco2017 True --automatic_yolo_quantization True --
 ↪  input_layout NHWC

# 执行结果:
{'acc_result': 0.5967312564573963, 'fps_result': 1479.6737448863978}
```

6.2.2.5 示例 5: 运行 YOLOv5 ONNX

```
$ igie-exec --model_path data/models/yolov5/yolov5m.onnx --input images:32,3,640,640 --precision
 ↪  int8 --use_coco2017 True --automatic_yolo_quantization True --custom_option
 ↪  required_pass:MutatePadMod --verbose True

# 执行结果:
{'acc_result': 0.6236765361287956, 'fps_result': 1094.5590025580452}
```

YOLOv5m + GPU NMS 处理:

```
$ igie-exec --model_path data/models/yolov5/yolov5m.onnx --input images:32,3,640,640 --precision
 ↪  fp16 --use_coco2017 True --automatic_yolo_quantization True --custom_option with_nms:True
 ↪  iou_thres:0.65 conf_thres:0.001 topk:100
```

6.2.2.6 示例 6: 运行 YOLOv7 ONNX

```
$ igie-exec --model_path data/models/yolov7/yolov7_dynamic.onnx --input images:32,3,640,640 --
↪  precision int8 --use_coco2017 True --automatic_yolo_quantization True

# 执行结果:
{'acc_result': 0.6851086809169135, 'fps_result': 643.7612413146849}
```

6.2.2.7 示例 7: 运行 ONNX BERT-Base Squad

BERT-Base Squad 是一个多输入 NLP 模型，下面展示生成其 FP16 的引擎文件，并使用随机数据进行性能测试的过程。

源模型来源于 https://huggingface.co/csarron/bert-base-uncased-squad-v1。

```
# 运行模型
$ igie-exec --model_path data/models/bert/bert-base-uncased-squad-v1.onnx --input input_ids:8,
↪  256 attention_mask:8,256 token_type_ids:8,256 --precision fp16

# 执行结果
[2023-05-18 03:36:24,299 igie-exec line:67] INFO: Evaluate inference igie time cost...
[2023-05-18 03:36:24,638 igie-exec line:70] INFO: Mean inference time (std dev): 9.773 ms (0.016
↪  ms)
[2023-05-18 03:36:24,638 igie-exec line:72] INFO: Mean fps: 818.612
```

6.2.2.8 示例 8: 运行 ONNX BERT-Large Squad

BERT-Large Squad 是一个多输入 NLP 模型，下面展示生成其 FP16 的引擎文件，并使用随机数据进行性能测试的过程。

源模型来源于 https://huggingface.co/neuralmagic/bert-large-uncased-finetuned-squadv1。

```
# 运行模型
$ igie-exec --model_path data/models/bert/bert-large-uncased-finetuned-squadv1.onnx --input
↪  input_ids:8,384 attention_mask:8,384 token_type_ids:8,384 --precision fp16

# 执行结果:
[2023-05-18 03:38:44,399 igie-exec line:67] INFO: Evaluate inference igie time cost...
[2023-05-18 03:38:45,098 igie-exec line:70] INFO: Mean inference time (std dev): 44.010 ms
↪  (0.021 ms)
[2023-05-18 03:38:45,099 igie-exec line:72] INFO: Mean fps: 181.778
```

6.2.2.9 示例 9: 运行 ONNX Electra

Electra 是一个多输入 NLP 模型，下面展示生成其 FP16 的引擎文件，并使用随机数据进行性能测试的过程。

源模型来源于 https://huggingface.co/bhadresh-savani/electra-base-emotion。

```
# 运行模型
$ igie-exec --model_path data/models/electra/electra-base-emotion.onnx --input input_ids:32,64
 ↪  attention_mask:32,64 token_type_ids:32,64 --precision fp16

# 执行结果：
[2023-05-18 03:39:49,471 igie-exec line:67] INFO: Evaluate inference igie time cost...
[2023-05-18 03:39:49,743 igie-exec line:70] INFO: Mean inference time (std dev): 2.751 ms (0.003
 ↪  ms)
[2023-05-18 03:39:49,743 igie-exec line:72] INFO: Mean fps: 11630.930
```

6.2.2.10 示例 10: 运行 ONNX ViT

ViT 是一个 Transformer 架构的 CV 模型，下面展示生成其 FP16 的引擎文件，并使用随机数据进行性能测试的过程。

源模型来源于 https://huggingface.co/google/vit-base-patch16-224。

```
# 运行模型
$ igie-exec --model_path data/models/vit/vit.onnx --input pixel_values:32,3,224,224 --precision
 ↪  fp16

# 执行结果：
[2023-05-18 03:41:11,573 igie-exec line:67] INFO: Evaluate inference igie time cost...
[2023-05-18 03:41:12,519 igie-exec line:70] INFO: Mean inference time (std dev): 56.862 ms
 ↪  (0.037 ms)
[2023-05-18 03:41:12,519 igie-exec line:72] INFO: Mean fps: 562.767
```

6.2.2.11 示例 11: 运行 ONNX Swin Transformer

Swin Transformer 是一个 Transformer 架构的 CV 模型，下面展示生成其 FP16 的引擎文件，并使用随机数据进行性能测试的过程。

源模型来源于 https://huggingface.co/microsoft/swin-small-patch4-window7-224。

```
# 运行模型
$ igie-exec  --model_path data/models/swin_transformer/swin_transformer.onnx --input
 ↪  pixel_values:32,3,224,224 --precision fp16

# 执行结果：
[2023-05-18 03:46:26,360 igie-exec line:67] INFO: Evaluate inference igie time cost...
```

```
[2023-05-18 03:46:27,028 igie-exec line:70] INFO: Mean inference time (std dev): 32.053 ms
↪  (0.031 ms)
[2023-05-18 03:46:27,028 igie-exec line:72] INFO: Mean fps: 998.360
```

6.2.2.12 示例 12: 运行 ONNX Conformer

Conformer 是一个 Audio 类的语音识别模型，下面展示生成其 FP16 的引擎文件，并使用随机数据进行性能测试的过程：

```
$ igie-exec  --model_path data/models/conformer/encoder_bs24_seq384_static_opt_matmul.onnx  --
↪  input speech:24,384,80 speech_lengths:24 --precision fp16

# 执行结果：
[2023-05-18 03:48:20,999 igie-exec line:67] INFO: Evaluate inference igie time cost...
[2023-05-18 03:48:21,720 igie-exec line:70] INFO: Mean inference time (std dev): 39.498 ms
↪  (0.026 ms)
[2023-05-18 03:48:21,720 igie-exec line:72] INFO: Mean fps: 607.627
```

Tip

如果把最后一个 topk 算子 offload 到模型外，性能将提升到 1200FPS 左右。

6.2.2.13 示例 13: 使用自定义数据集进行 INT8 量化

igie-exec 支持用户提供自定义的 pkl 数据用于 INT8 量化校准，自定义数据要求后缀名为 `.pkl`。对于 `.pkl` 格式的输入，igie-exec 内部会通过 `pickle` 读取，格式要求如下：

- 单输入，比如 resnet@imagnet。如果我们准备 100 个数据，首先创建一个以模型输入名为 key，校准数据集为 value 的字典，然后将其保存为 `pkl` 格式的文件。

```
a = np.random.randn(100, 3, 224, 224).astype(np.float32)
data = {'input': a}

with open('calibration.pkl', 'wb') as pkl_file:
    pickle.dump(data, pkl_file)
```

- 多输入，假设有两个输入，比如也是 100 个数据，按照模型不同输入，保存对应的校准数据集即可：

```
a = np.random.randn(100, 3, 224, 224).astype(np.float32)
b = np.random.randn(100, 3, 100).astype(np.float32)

data = {'input_1': a, 'input_2': b}

with open('calibration.pkl', 'wb') as pkl_file:
    pickle.dump(data, pkl_file)
```

下面是 PyTorch ResNet50 使用 .pkl 量化的例子：

```
# 获取.pkl 格式的校准数据集

# 使用自定义数据进行 INT8 量化并生成引擎文件
# 对于除 ONNX 外的模型，这里可以指定 batchsize 为 1，用来加速量化系数表的生成
# 后续生成其他 batch 的引擎文件时，可以重复利用该量化系数表，以节省时间和内存
igie-exec --model_path data/models/resnet/resnet50-fp32.pt --input input:1,3,224,224 --precision
 ↪  int8  --calibration_file_path calibration.npy

# 使用已有的量化系数表，生成大 batch 的引擎文件，并在完整的数据集上测试
igie-exec --model_path data/models/resnet/resnet50-fp32.pt --input input:32,3,224,224 --
 ↪  precision int8 --use_imagenet True


# 执行结果：
{'acc@1': 0.744538252240717,
 'acc@5': 0.9224951984635084,
 'acc_result': 0.744538252240717,
 'fps_result': 5753.188203730447}
```

6.3 igie-exec 测试工具的参数定义

您可通过 **./igie-exec -h** 命令获取全部参数信息，具体参数介绍如下：

必填参数

- `--model-path`：模型的路径，目前 igie-exec 支持导入 TensorFlow/PyTorch/ONNX/PaddlePaddle 四种框架的模型以及从 TorchVision 直接下载模型。
  - `--model_path resnet50.pb`：导入 TensorFlow 模型，以 `*.pb` 为后缀名。
  - `--model_path resnet50.pt`：带入 PyTorch 模型，以 `*.pt` 为后缀名。
  - `--model_path resnet50.onnx`：导入 ONNX 模型，以 `*.onnx` 为后缀名。
  - `--model_path resnet50/inference`：导入 PaddlePaddle 模型，注意 PaddlePaddle 以目录的方式给出，如 `resnet50`，内含名为 `inference.pdiparams`，`inference.pdiparams.info` 和 `inference.pdmodel` 的文件。
  - `--model_path resnet18`：导入 TorchVision 预训练模型，igie-exec 内置了 TorchVision 的模型，可通过指定 TorchVision 中的模型名字自动下载模型并生成引擎。
- `--input`：模型的输入信息，支持多输入，包含 name、shape 和 dtype 三种信息，格式为 `name:shape/dtype`。其中，name 和 shape 为必填，dtype 不填写默认为 float32，常见的 dtype 包含 float32/int32/int64。例如：
  - 单输入：`--input input1:32,3,224,224`，此时默认 dtype 为 float32。
  - 单输入指定 dtype：`--input input1:32,3,224,224/int32`
  - 多输入：`--input input1:32,3,224,224 input2:32,100`
  - 多输入指定 dtype：`--input input1:32,3,224,224/float32 input2:32,100/int64`

- `--precision`：模型的推理精度，根据该精度会选择对应的量化策略以及图优化选项。
  - 使用 `--precision fp16` 参数会自动将大部分的计算转换为 FP16。
  - 使用 `--precision int8` 参数会根据模型的原生框架以及是否提供校准数据集来进行对应的量化策略。
    * 模型框架：
      · 如果是 ONNX 模型，将自动采用 ONNXRuntime 量化。
      · 如果是 TensorFlow/PyTorch/PaddlePaddle 模型，将采用 IGIE 进行量化。
    * 校准数据：
      · 如果提供校准数据集 (参考后续 `--calibration_file_path` 的说明)，或是使用内置的 imagenet/coco2017 数据 (参考后续的 `--use_imagenet` 和 `--use_coco2017` 的说明)，将采用内置的校准数据集进行量化。
      · 如果没有指定数据集，将根据输入的 input 信息生成随机数据进行量化和推理。

可选参数

- `--target`：模型推理的目标设备，默认值为 `--target iluvatar_with_all_libs`，可选项以及对应说明如下：
  - `--target llvm`：CPU 推理。
  - `--target iluvatar`：GPU 推理。
  - `--target iluvatar_with_cudnn_cublas`：GPU 推理，且启用 cuBLAS/cudnn 加速库。
  - `--target iluvatar_with_all_libs`：GPU 推理，且启用 cuBLAS/cudnn/ixInfer 加速库。ixInfer 是天数智芯加速卡底层的高性能算子库，能极大幅度地提高推理速度。
- `--engine_path`：生成的引擎文件的保存路径，后缀名必须以 `.so` 结尾，默认值为 None，此时则保存在当前工作路径。用例如下：

  ```
  --engine_path resent50.so
  ```

- `--warmup`：性能测试前的 warmup 次数，默认值为 `--warmup 3`，一般不需要特殊指定。
- `--verbose`：是否保存编译期间的模型结构对应的 mod 信息，默认为 `--verbose False`，使用 `--verbose True` 可将编译期间的 mod 保存在当前目录的 module 目录中，方便 debug。
- `--num_workers`：加载数据集使用的线程数，默认值为 `--num_workers 16`，在 CPU 核数较少的情况下建议设置成 `--num_workers 1` 或 `--num_workers 4`。
- `--batch_size`：一个 batch 的大小，默认值为 None，会自动从 `--input` 中传入的 shape 信息自动解析。但是对于一些特殊的多输入的情况，如 `--input input_ids:1000,22 pixel_values:32,3,224,224 attention_mask:1000,22`，实际一个 batch 的数据应该为 32，此时可通过该 `--batch_size 32` 参数进行显式指定。
- `--use_imagenet`：使用标准的 imagenet 的 val 数据集进行量化校准以及精度验证，通过 `--use_imagenet True` 启用，适用于 ResNet/VGG/ShuffleNet/MobileNet 等经典模型，输入格式默认为 NCHW，模型预处理按照 PyTorch 格式进行。
- `--use_coco2017`：使用标准的 coco2018 的 val 数据集进行量化校准以及精度验证，通过 `--use_coco2017 True` 启用，适用于 YOLO 系列经典模型，输入格式默认为 NCHW，模型预处理按照 PyTorch 格式进行。
- `--input_layout`：在使用 imagenet/coco2017 数据集时的输入数据格式，默认值为 `--input_layoyut NCHW`，可改变为 `--input_layout NHWC`。此时 imagnet 的输入预处理会采用 TensorFlow 格式。

- `--calibration_file_path`：用户定义的量化校准数据集，需要以 numpy 的格式提供，具体可参考示例 13: 使用自定义数据集进行 INT8 量化。
- `--automatic_yolo_quantization`：是否自动跳过 YOLO 系列 ONNX 模型的 detect 部分的节点的量化，通过开启该选项可大幅提高 YOLO 系列模型的量化精度，默认值为 `--automatic_yolo_quantization False`。
- `--quantization_config_path`：精细配置 ONNXRuntime/IGIE 量化参数的 json 文件，具体可参考示例 13: 使用自定义数据集进行 INT8 量化。
- `--acc_target`：使用 imagenet/coco2017 数据集进行精度评价的目标值，其中 imagenet 使用 `acc@1`，coco2017 使用 `map@.5` 作为目标。
- `--fps_target`：使用 imagenet/coco2017 数据集进行性能评价的目标值，fps 的计算公式为 (batch_size / batch_infer_time) 的平均值。
- `--perf_only`：是否仅使用随机数据做性能测试，默认值为 `--perf_only False`，但是在未使用 imagenet/coco2017 的情况下，该选项默认会被自动打开。
- `--just_export`：是否仅生成引擎文件，默认值为 `--just_export False`。
- `--custom_option`:拓展选项,使用类似 `--input`,支持复数拓展选项,格式为 `--custom_option key1:value1 key2:value2 key3:value3`。目前开放的拓展参数如下：
  - `with_nms`：YOLO 系列模型是否将 nms 也加入到生成的引擎中，可通过 `with_nms:True` 的方式指定。
  - `iou_thres`：nms 中的 iou 阈值，可通过 `iou_thres:0.65` 的方式指定。
  - `conf_thres`：nms 中的 conf 阈值，可通过 `conf_thres:0.001` 的方式指定。
  - `topk`：nms 生成的最大的 box 数量，可通过 `topk:100` 的方式指定。

7 从 TensorRT 迁移到 IGIE

IGIE 与 TensorRT 拥有一模一样的推理流程，推理代码都遵循 导入模型 -> 量化（可选）-> 自动图优化（编译）-> 导出或导入引擎文件 -> 设置模型输入 -> 推理模型等步骤。从使用 API 角度看，IGIE 的调用接口更加简洁。

下面以 ResNet50 模型为例，比较 TensorRT 和 IGIE 推理过程：

7.1 使用 TensorRT 推理 ResNet50 模型

TensorRT 推理模型的流程分为 导入模型 -> 量化（量化）-> 自动图优化（编译）-> 导出或导入引擎文件 -> 设置模型输入 -> 推理模型等步骤。

```
# 1) 导入 ONNX 模型到 TRT
network = trt.builder.create_network(batch_size)
parser = trt.OnnxParser(network)
with open(model_file, "rb") as model:
    parser.parse(model.read())


# 2) 模型量化，以 FP16 为例
config = trt.builder.create_builder_config()
config.set_flag(trt.BuilderFlag.FP16)


# 3) 自动优化模型，编译模型为引擎文件
config.max_workspace_size = 1 << 30 # 1GB
engine = builder.build_engine(network, config)


# 4) 导出或导入引擎文件
plan = trt.builder.build_serialized_network(network, config) # 导出引擎文件
with open("ResNet50.engine", "wb") as f:
    f.write(plan)
engine = trt.runtime.deserialize_cuda_engine(plan) # 导入引擎文件

context = engine.create_execution_context() # 设置 module context，加载引擎文件到 GPU device


# 5) 设置模型输入
inputs = []
outputs = []
bindings = []
stream = cuda.Stream()
for binding in engine:
```

```
    size = trt.volume(engine.get_binding_shape(binding)) * engine.max_batch_size
    dtype = trt.nptype(engine.get_binding_dtype(binding))
    # Allocate host and device buffers
    host_mem = cuda.pagelocked_empty(size, dtype)
    device_mem = cuda.mem_alloc(host_mem.nbytes)
    # Append the device buffer to device bindings.
    bindings.append(int(device_mem))
    # Append to the appropriate list.
    if engine.binding_is_input(binding):
        inputs.append(HostDeviceMem(host_mem, device_mem))
    else:
        outputs.append(HostDeviceMem(host_mem, device_mem))

cuda.memcpy_htod_async(inputs.device, inputs.host, stream)


# 6) 执行推理
context.execute_async_v2(bindings=bindings, stream_handle=stream.handle)


# 7) 获取 TRT 推理输出
cuda.memcpy_dtoh_async(outputs.host, outputs.device, stream)
stream.synchronize()
print(outputs.host)
```

7.2 使用 IGIE 推理 ResNet50 模型

使用 IGIE 推理模型依然遵循此流程：导入模型 -> 量化 -> 自动图优化（编译）-> 导出或导入引擎文件 -> 设置模型输入 -> 推理模型。

```
# 1) 导入 ONNX 模型到 IGIE
input_name = "input0"
shape_list = [(input_name, input_shape)]
mod, params = relay.frontend.from_pytorch(scripted_model, shape_list)


# 2) 设置模型使用 FP16 推理
precision = "fp16"

# 3) 自动优化模型，编译模型为 engine
print("Strat build engine...")
engine = relay.build(mod, target=target, params=params, precision=precision)
```

```
# 4) 导出或导入引擎文件
engine.export_library("resnet50.so")              # 导出引擎文件
engine = tvm.runtime.load_module("resnet50.so")  # 导入引擎文件

m = graph_executor.GraphModule(engine["default"](dev)) # 设置 model context，加载引擎文件到 GPU
↪  device


# 5) 设置模型输入
image = np.random.uniform(size=input_shape).astype("float32")
image = tvm.nd.array(image, device)
m.set_input(input_name, image)


# 6) 推理模型
m.run()


# 7) 输出推理结果
igie_output = m.get_output(0)
print(igie_output)
```

7.3 总结

IGIE 与 TensorRT 推理对比总结：

IGIE 与 TensorRT 拥有一模一样的推理流程，推理代码都遵循导入模型 -> 量化 -> 自动图优化 (编译) -> 导出或导入引擎文件 -> 设置模型输入 -> 推理模型等步骤。从使用 API 角度看，IGIE 的调用接口更加简洁。

当然，在部署模型时，更常用的方法是直接拿编好的引擎文件进行部署推理，即下图的 4，5，6，7 步骤，使得适配迁移 TensorRT 更加方便。

IGIE与TensorRT模型推理对比

	TensorRT模型推理	IGIE模型推理	

1	`# 1) 导入ONNX模型到TRT`
	`network = trt.builder.create_network(batch_size)`
	`parser = trt.OnnxParser(network)`
	`with open(model_file, "rb") as model:`
	`  parser.parse(model.read())`
		`# 1) 导入ONNX模型到IGIE`
		`input_name = "input0"`
		`shape_list = [(input_name, input_shape)]`
		`mod, params = relay.frontend.from_pytorch(model, shape_list)`
			导入模型

2	`# 2) 模型量化，以FP16为例`
	`config = trt.builder.create_builder_config()`
	`config.set_flag(trt.BuilderFlag.FP16)`
		`# 2) 模型量化，以FP16为例`
		`from tvm.relay.transform import ToMixedPrecision`
		`mod = ToMixedPrecision(mixed_precision_type="float16")(mod)`
			量化模型

3	`# 3) 自动优化模型，编译模型为engine`
	`config.max_workspace_size = 1 << 30 # 1GB`
	`engine = builder.build_engine(network, config)`
		`# 3) 自动优化模型，编译模型为engine`
		`engine = relay.build(mod, target=target, params=params)`
			编译engine

4	`# 4) 导出 or 导入 engine文件`
	`plan = trt.builder.build_serialized_network(network, config)`
	`with open("ResNet50.engine", "wb") as f:`
	`  f.write(plan)`
	`engine = trt.runtime.deserialize_cuda_engine(plan)`
	`context = engine.create_execution_context()`
		`# 4) 导出 or 导入engine文件`
		`engine.export_library("resnet50.so")`
		`engine = tvm.runtime.load_module("resnet50.so")`
		`m = graph_executor.GraphModule(engine["default"](dev))`
			导入&导出engine

5	`# 5) 设置模型输入`
	`inputs = []`
	`outputs = []`
	`bindings = []`
	`stream = cuda.Stream()`
	`for binding in engine:`
	`  size = trt.volume(engine.get_binding_shape(binding)) * engine.max_batch_size`
	`  dtype = trt.nptype(engine.get_binding_dtype(binding))`
	`  # Allocate host and device buffers`
	`  host_mem = cuda.pagelocked_empty(size, dtype)`
	`  device_mem = cuda.mem_alloc(host_mem.nbytes)`
	`  # Append the device buffer to device bindings.`
	`  bindings.append(int(device_mem))`
	`  # Append to the appropriate list.`
	`  if engine.binding_is_input(binding):`
	`  inputs.append(HostDeviceMem(host_mem, device_mem))`
	`  else:`
	`  outputs.append(HostDeviceMem(host_mem, device_mem))`
	`inputs, outputs, bindings, stream = allocate_buffers(engine)`
	`cuda.memcpy_htod_async(inputs.device, inputs.host, stream)`
		`# 5) 设置模型输入`
		`image = np.random.uniform(size=input_shape)`
		`image = tvm.nd.array(image, device)`
		`m.set_input(input_name, image)`
			设置模型输入

6	`# 6) 执行推理` `context.execute_async_v2(bindings=bindings, stream_handle=stream.handle)`	`# 6) 执行推理` `m.run()`	执行推理

7	`# 7) 获取TRT推理输出`
	`cuda.memcpy_dtoh_async(outputs.host, outputs.device, stream)`
	`stream.synchronize()`
	`print(outputs.host)`
		`# 7) 输出推理结果`
		`igie_output = m.get_output(0)`
		`print(igie_output)`
			输出结果

Figure 4: IGIE 与 TensorRT 推理对比

8 IGIE 自动图优化介绍

IGIE 图优化模块能够自动地对导入模型进行优化，主要包括自动算子融合，冗余算子消除，常量折叠，内存重用，layout 转换，自动 padding 等，最终转换为符合天数智芯硬件特性，性能更优的 IR Module，用于最终推理。

8.1 算子融合

算子融合是推理引擎加速重要的手段，将零散的算子融合成一个算子后，有两大好处：

- 减少 kernel launch 的个数。
- 目前大多数模型的算子都是 memory bound，融合算子后会节省很多 kernel 结果写回 global memory 的时间，这在拥有大量小算子的模型上尤为明显。

下面列举 IGIE 中几个常用的算子融合：

Conv2d+BatchNorm+Bias 算子融合：

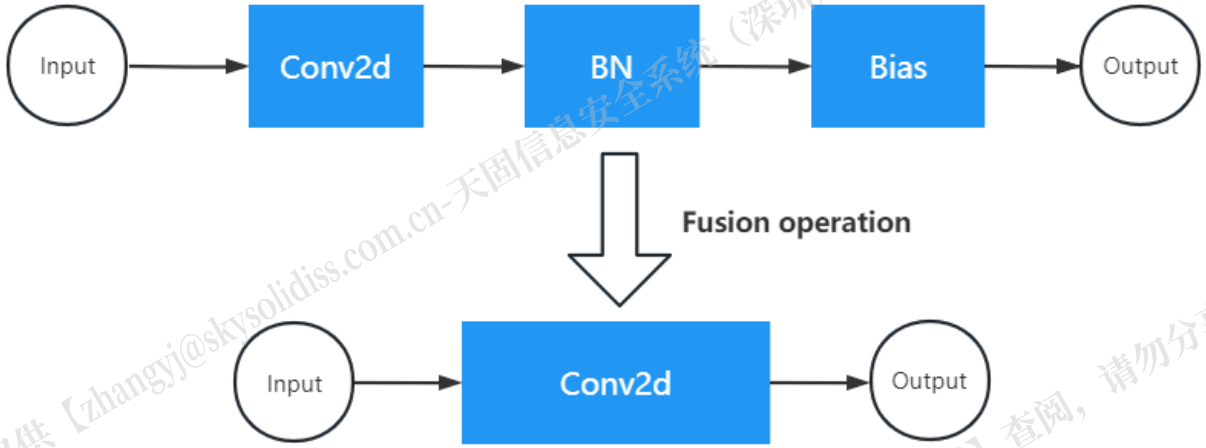

Figure 5: 算子融合

Conv2d+BatchNorm+Add+Relu 算子融合：

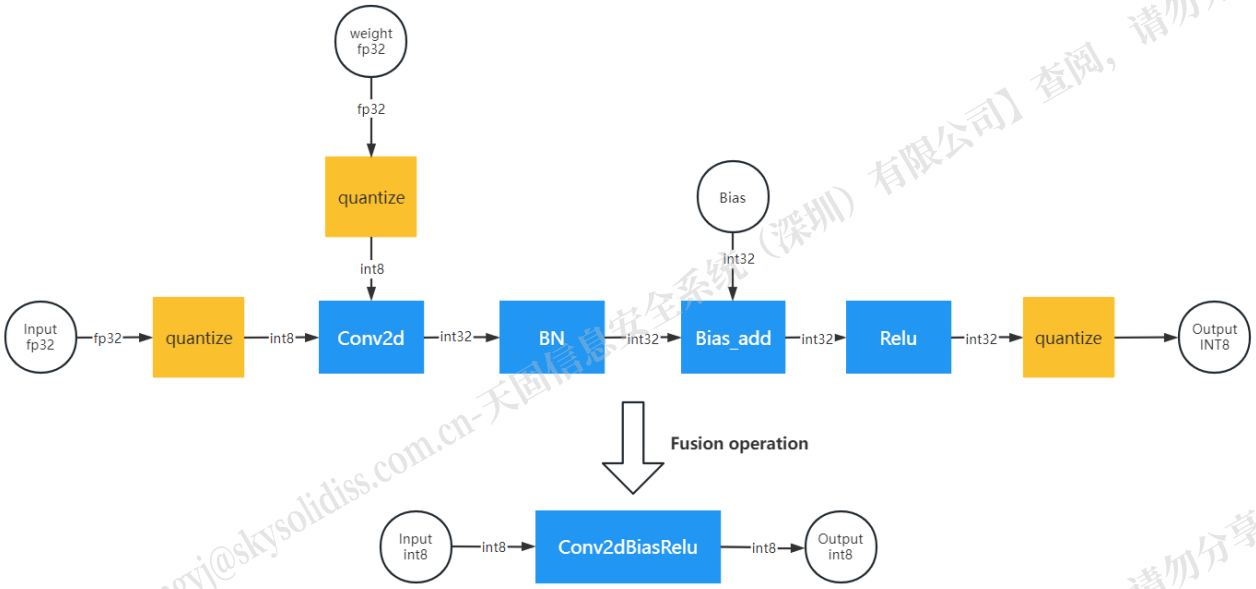

Figure 6: 算子融合

Conv2d+Sigmoid+multiply 算子融合：

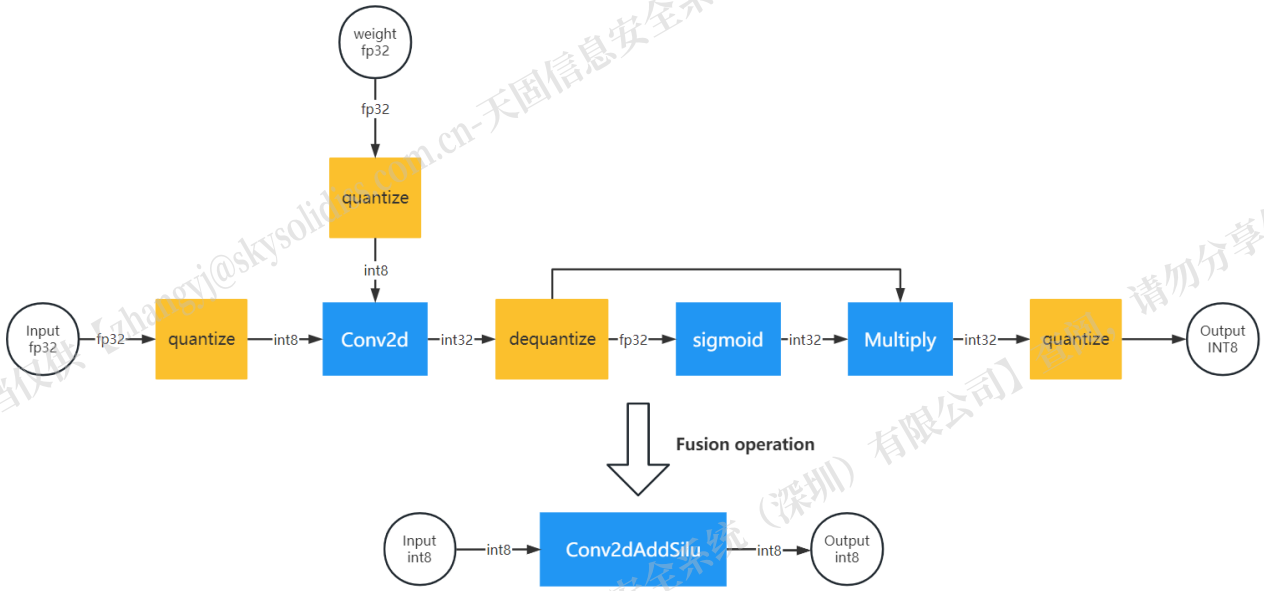

Figure 7: 算子融合

两个 Conv2d+Add+Relu 算子融合

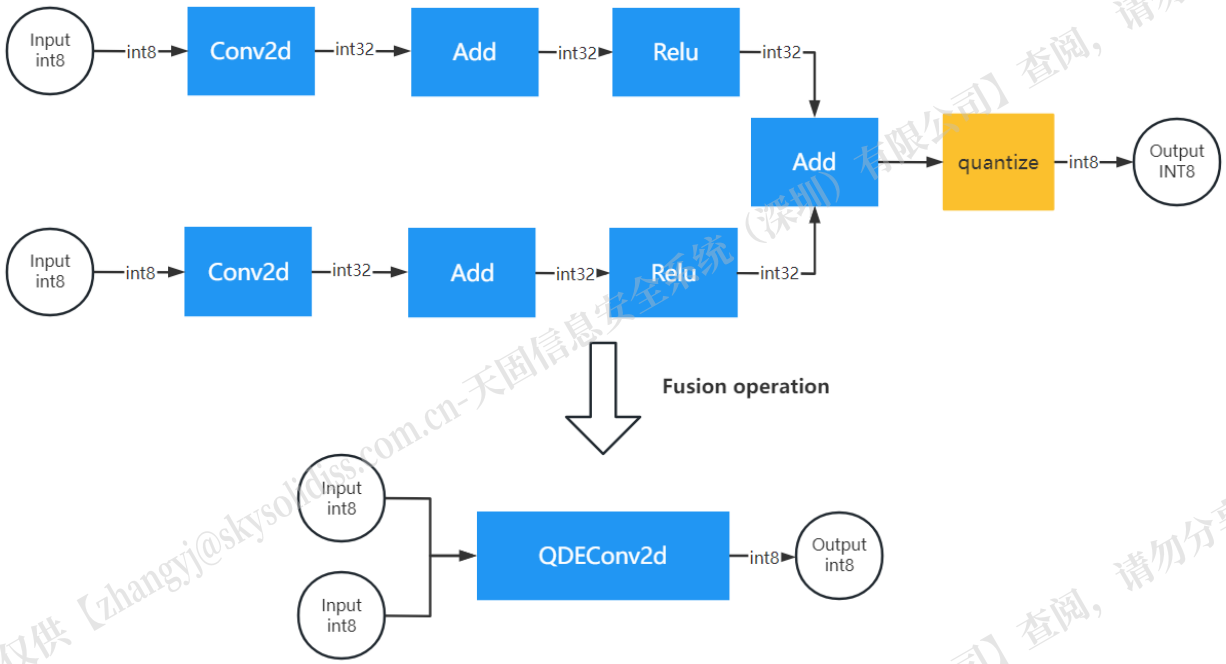

Figure 8: 算子融合

量化 Concat 算子融合

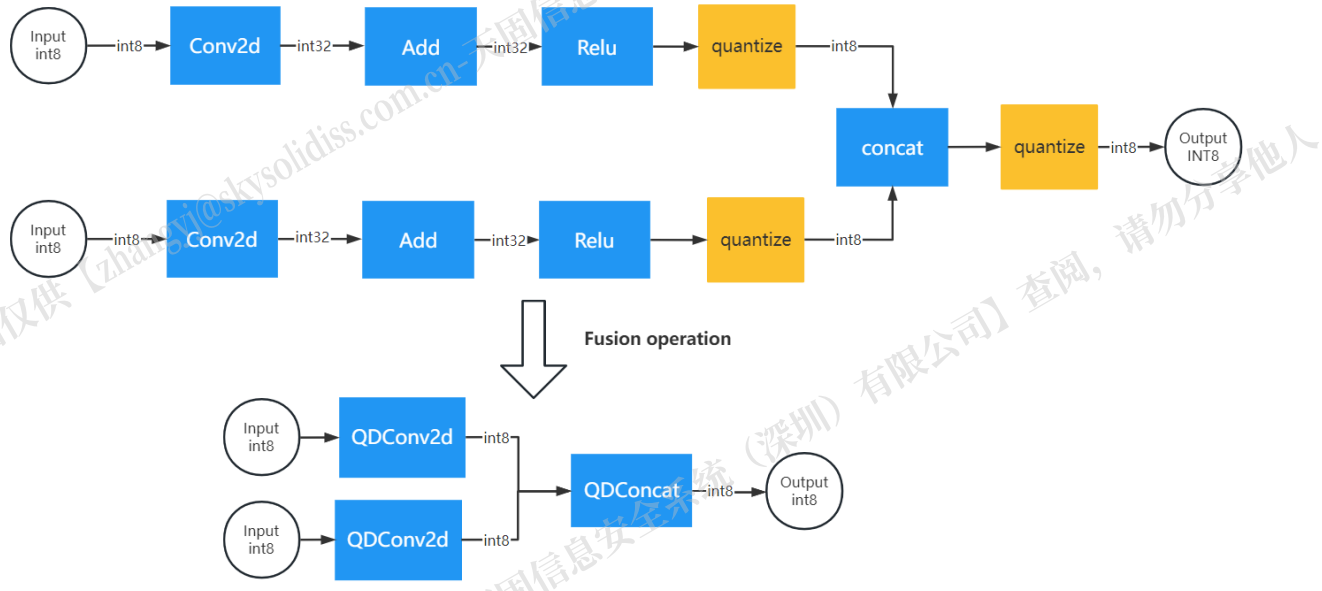

Figure 9: 算子融合

8.2 Layout 转换和自动 padding

IGIE 可以方便的对 Tensor 进行内存布局转换，以优化数据局部性，提升访存效率。也可以根据天数智芯加速卡中 TCU 的硬件特性进行自动 padding。

例如：

1) 模型 Layout 自动转换为算子库性能最佳方式（NCHW <-> NHWC）。
2) 重新 Packing data，方便对接外部算子库的实现。（NCHW->NCHWC64）
3) 将整个模型的 conv2d channel 自动填充成适合天数 GPU 计算的大小。

8.3 内存复用

IGIE 通过对模型中部分算子的内存进行复用，以减少内存开销。

例如下图，颜色一样的算子内存会进行复用（relu+relu+add）。

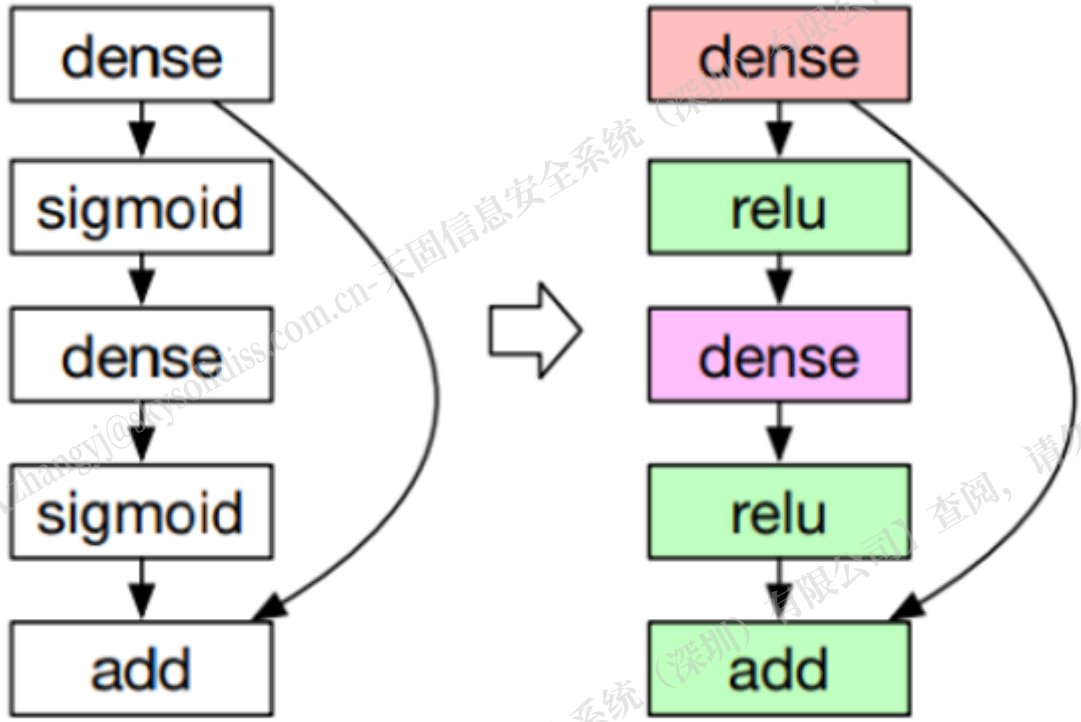

Figure 10: 内存复用

8.4 常量折叠和常量预计算

对于模型中的常量或者 weights 运算，IGIE 会在编译的时候，自动将常量提前计算好。

对于多个相关的常量，会进行折叠处理。

如下图，权重所有的运算操作，全部在模型运行前计算好。

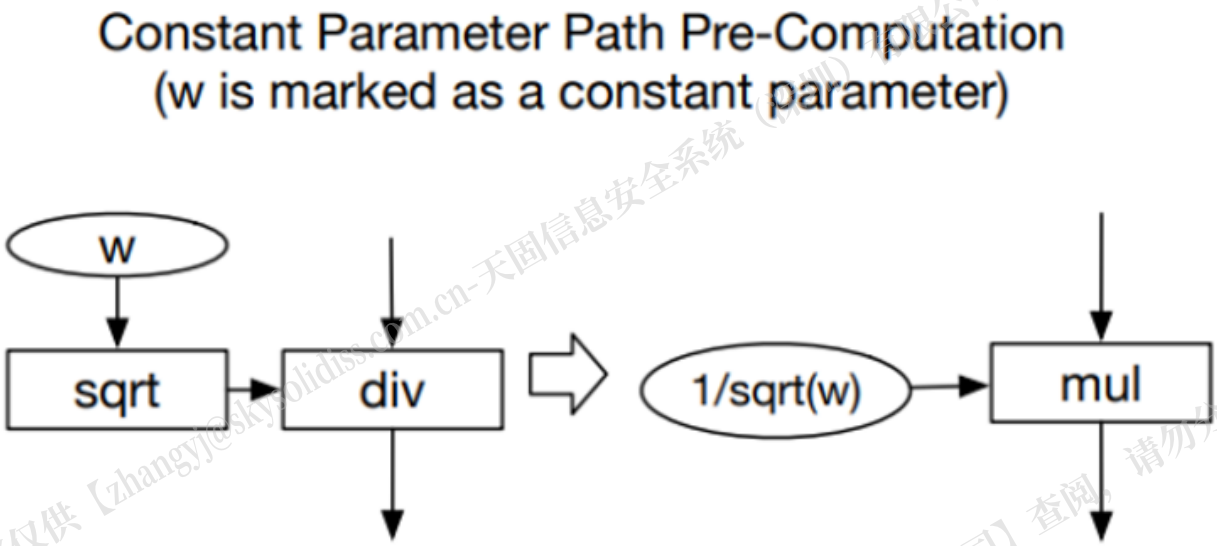

Figure 11: 常量计算

9 IGIE C++ API 说明

主要介绍 IGIE 模型部署相关 C++ API，包括：

- Target 相关类和函数
- Device 类和获取设备信息相关函数
- 内存操作和 Stream 设置相关函数
- 构造 Tensor 相关类和函数
- 加载模型引擎相关类和函数
- 模型输入输出信息相关函数
- 模型运行相关函数
- 模型耗时统计
- 推理代码参考示例

9.1 引入必要的头文件

```
#include <dlpack/dlpack.h>
#include <tvm/runtime/device_api.h>
#include <tvm/runtime/module.h>
#include <tvm/runtime/packed_func.h>
#include <tvm/runtime/registry.h>
#include <tvm/runtime/ndarray.h>
#include <tvm/runtime/profiling.h>
```

9.2 Target 相关类

9.2.1 Target()

功能概述

构造函数，根据传入 string 名称创建 Target。

函数原型

```
tvm::Target::Target(const String &target_str)
```

参数说明

`target_str`: 传入 target name 字符串，用于构造 Target。

返回值

返回创建的 Target。

使用示例

```
auto target = Target("iluvatar")
```

9.2.2 Target::Current()

功能概述

获得当前上下文的 Target。

函数原型

```
static tvm::Target tvm::Target::Current(bool allow_not_defined=true)
```

参数说明

- allow_not_defined：如果该参数设置为 true，允许返回 undefined Target。否则，当 context stack 为 empty 时，会导致运行时错误。

返回值

返回当前上下文的 Target。

9.3 Device 相关类

设备 Device 相关的类和函数。主要包括：

- Device 类的构造
- DeviceAPI 获取设备信息，该接口是 cudaDeviceGetAttribute 的包装，可以获取以下设备信息：
  - kWarpSize：Warp size 的数量。
  - kComputeVersion：CUDA 计算版本。
  - kDeviceName：设备名称。
  - kMaxClockRate：最大时钟频率。
  - kMultiProcessorCount：CU 个数。
  - kMaxThreadsPerBlock：GPGPU 设备中每个 Block 可用的最大线程数。
  - kMaxSharedMemoryPerBlock：GPGPU 设备中每个 Block 可用的最大 shape memory 大小。
  - kMaxRegistersPerBlock：每个 Block 可用最大寄存器的数量。
  - kMaxThreadDimensions：Block 能设置的最大 X、Y、Z 维度。

9.3.1 DLDevice 结构体

功能概述

创建 DLDevice，用于指定 op 和 tensor 的运行设备。

函数原型

```
typedef struct {
  DLDeviceType device_type;
  int device_id;
} DLDevice;

DLDevice dev{kDLILUVATAR, 0};
```

参数说明

- DLDeviceType：指定设备类型，天数智芯加速卡设备的宏为：kDLILUVATAR；CPU 设备的宏为：kDLCPU。
- device_id: 指定第几个设备，例如一台机器上有 8 卡 GPU，device_id 可以指定为 0~7 。

返回值

返回 DLDevice 结构体。

9.3.2 DeviceAPI 类

功能概述

DeviceAPI 是一个获取 device 信息和对 device 进行操作的基础类，通过使用 DeviceAPI 类的成员函数对 Device 进行操作。您需要使用 device 构建 DeviceAPI 对象。

函数原型

```
static DeviceAPI* DeviceAPI::Get(const DLDevice& dev)
```

参数说明

DLDevice& dev: 通过 DLDevice 创建的 device 结构体。

返回值

返回 DeviceAPI 类对象。

使用示例

```
DLDevice dev{kDLILUVATAR, 0};
DeviceAPI* api = DeviceAPI::Get(dev);
```

9.3.3 DeviceAPI::GetAttr()

功能概述

获取 Device 设备属性信息，该接口是对 cudaDeviceGetAttribute 的包装，可以通过 DeviceAttrKind 获取以下设备信息：

- kWarpSize：Warp size 的数量。

- kComputeVersion：CUDA 计算版本。
- kDeviceName：设备名称。
- kMaxClockRate：最大时钟频率。
- kMultiProcessorCount：CU 个数。
- kMaxThreadsPerBlock：GPGPU 设备中每个 Block 可用的最大线程数。
- kMaxSharedMemoryPerBlock：GPGPU 设备中每个 Block 可用的最大 shape memory 大小。
- kMaxRegistersPerBlock：每个 Block 可用最大寄存器的数量。
- kMaxThreadDimensions：Block 能设置的最大 X、Y、Z 维度。

函数原型

```
virtual void GetAttr(Device dev, DeviceAttrKind kind, TVMRetValue* rv)
```

参数说明

- `Device& dev`: 通过 DLDevice 创建的 device 结构体。
- `DeviceAttrKind kind`: 指定获取的设备属性 Kind，DeviceAttrKind 结构体定义如下：

```
enum DeviceAttrKind : int {
  kExist = 0,
  kMaxThreadsPerBlock = 1,
  kWarpSize = 2,
  kMaxSharedMemoryPerBlock = 3,
  kComputeVersion = 4,
  kDeviceName = 5,
  kMaxClockRate = 6,
  kMultiProcessorCount = 7,
  kMaxThreadDimensions = 8,
  kMaxRegistersPerBlock = 9,
  kGcnArch = 10,
  kApiVersion = 11,
  kDriverVersion = 12
};
```

- `TVMRetValue* rv`: GetAttr 函数的返回值会存放在 rv 中。

返回值

无返回值。

使用示例

```
DLDevice dev{kDLILUVATAR, 0};
DeviceAPI* api = DeviceAPI::Get(dev);
kind = tvm::runtime::DeviceAttrKind::kWarpSize;
tvm::runtime::TVMRetValue ret;
api->GetAttr(dev, kind, &ret);
int warp_size = ret;
```

9.4 Device 内存、流相关操作和设置

9.4.1 SetDevice()

功能概述

设置模型的运行设备 Device id，底层调用为 cudaSetDevice 函数。

函数原型

```
virtual void DeviceAPI::SetDevice(Device dev)
```

参数说明

Device& dev: 通过 DLDevice 创建的 device 结构体。

返回值

无返回值。

9.4.2 CreateStream()

功能概述

创建一个新的执行 stream。

函数原型

```
virtual TVMStreamHandle DeviceAPI::CreateStream(Device dev)
```

参数说明

Device& dev: 通过 DLDevice 创建的 device 结构体。

返回值

返回一个 TVMStreamHandle 对象。

使用示例

```
DLDevice dev{kDLILUVATAR, 0};
DeviceAPI* api = DeviceAPI::Get(dev);
TVMStreamHandle stream = api->CreateStream(dev);
```

9.4.3 SetStream()

功能概述

设置模型推理到指定 stream。

函数原型

```
virtual void tvm::runtime::DeviceAPI::SetStream(Device dev, TVMStreamHandle stream)
```

参数说明

- `Device& dev`: 通过 DLDevice 创建的 device 结构体。
- `TVMStreamHandle stream`: 通过 CreateStream() 创建的 Stream 对象。

返回值

无返回值。

使用示例

```
DLDevice dev{kDLILUVATAR, 0};
DeviceAPI* api = DeviceAPI::Get(dev);
TVMStreamHandle stream = api->CreateStream(dev);
api->SetStream(dev, stream);
```

9.4.4 StreamSync()

功能概述

同步指定 stream。

函数原型

```
virtual void tvm::runtime::DeviceAPI::StreamSync(Device dev, TVMStreamHandle stream)
```

参数说明

- `Device& dev`: 通过 DLDevice 创建的 device 结构体。
- `TVMStreamHandle stream`: 通过 CreateStream() 创建的 Stream 对象。

返回值

无返回值。

使用示例

```
DLDevice dev{kDLILUVATAR, 0};
DeviceAPI* api = DeviceAPI::Get(dev);
TVMStreamHandle stream = api->CreateStream(dev);
...
api->StreamSync(dev, stream);
```

9.4.5 DeviceSync()

功能概述

同步当前 Device。

函数原型

```
virtual void tvm::runtime::DeviceAPI::DeviceSync(Device dev)
```

参数说明

- Device& dev: 通过 DLDevice 创建的 device 结构体。

返回值

无返回值。

使用示例

```
DLDevice dev{kDLILUVATAR, 0};
DeviceAPI* api = DeviceAPI::Get(dev);
...
api->DeviceSync(dev);
```

9.4.6 FreeStream()

功能概述

释放 stream 对象。

函数原型

```
virtual void tvm::runtime::DeviceAPI::FreeStream(Device dev, TVMStreamHandle stream)
```

参数说明

- Device& dev: 通过 DLDevice 创建的 device 结构体。
- TVMStreamHandle stream: 通过 CreateStream() 创建的 Stream 对象。

返回值

无返回值。

使用示例

```
DLDevice dev{kDLILUVATAR, 0};
DeviceAPI* api = DeviceAPI::Get(dev);
TVMStreamHandle stream = api->CreateStream(dev);
...
api->FreeStream(dev, stream);
```

9.4.7 AllocDataSpace()

功能概述

申请一块设备内存。

函数原型

```
virtual void* tvm::runtime::DeviceAPI::AllocDataSpace(Device dev, size_t nbytes, size_t
↪  alignment, DLDataType type_hint)
```

参数说明

- Device& dev: 通过 DLDevice 创建的 device 结构体。
- size_t nbytes: 申请内存的 byte 大小。
- size_t alignment: 内存对齐大小。
- DLDataType type_hint: 申请单个 elements 的类型。对于 Float 类型 Element 为: DLDataType{kDLFloat, 32, 1};

DLDataType 定义如下:

```
typedef struct {
  /*!
   * \brief Type code of base types.
   * We keep it uint8_t instead of DLDataTypeCode for minimal memory
   * footprint, but the value should be one of DLDataTypeCode enum values.
   * */
  uint8_t code;
  /*!
   * \brief Number of bits, common choices are 8, 16, 32.
   */
  uint8_t bits;
  /*! \brief Number of lanes in the type, used for vector types. */
  uint16_t lanes;
} DLDataType;
```

返回值

void* ptr: 返回申请的内存空间指针。

使用示例

```
DLDevice dev{kDLILUVATAR, 0};
DeviceAPI* api = DeviceAPI::Get(dev);
DLDataType dtype = DLDataType{kDLFloat, 32, 1};
void* ptr;
ptr = api->AllocDataSpace(dev, 1024, 64, dtype);
```

9.4.8 CopyDataFromTo()

功能概述

拷贝内存，从一个设备内存 place 到另一个设备内存 place。例如拷贝 cpu 内存数据到 GPU 内存，(可以是 cpu->gpu，也可以是 gpu->gpu、gpu->cpu 的内存数据拷贝)。

函数原型

```
virtual void tvm::runtime::DeviceAPI::CopyDataFromTo(DLTensor* from, DLTensor* to,
↪  TVMStreamHandle stream)
```

参数说明

- `DLTensor* from`: 通过 DLDevice 创建的 device 结构体。
- `DLTensor* to`: 申请内存的 byte 大小。
- `TVMStreamHandle stream`: 指定内存操作的流，可以是默认流 nullptr。

DLTensor 定义如下：

```
typedef struct {
  /*!
   * \brief The opaque data pointer points to the allocated data. This will be
   * CUDA device pointer or cl_mem handle in OpenCL. This pointer is always
   * aligned to 256 bytes as in CUDA.
   */
  void* data;
  /*! \brief The device of the tensor */
  DLDevice device;
  /*! \brief Number of dimensions */
  int ndim;
  /*! \brief The data type of the pointer*/
  DLDataType dtype;
  /*! \brief The shape of the tensor */
  int64_t* shape;
  /*!
   * \brief strides of the tensor (in number of elements, not bytes)
   *  can be NULL, indicating tensor is compact and row-majored.
   */
  int64_t* strides;
  /*! \brief The offset in bytes to the beginning pointer to data */
  uint64_t byte_offset;
} DLTensor;
```

返回值

无返回值。

使用示例

```
DLDevice dev{kDLILUVATAR, 0};
DeviceAPI* api = DeviceAPI::Get(dev);

DLTensor from;
from.data = const_cast<void*>(data);
from.device = DLDevice{kDLILUVATAR, 0};
from.dtype = DLDataType{kDLFloat, 32, 1};
ShapeTuple shape{1, 224, 224, 3};
from.ndim = static_cast<int>(shape.size());
from.shape = const_cast<int64_t *>(shape.data());
from.strides = nullptr;
from.byte_offset = 0;

DLTensor to;
...
api->CopyDataFromTo(&from, &to, nullptr);
```

9.4.9 FreeDataSpace()

功能概述

释放一块设备内存。

函数原型

```
virtual void tvm::runtime::DeviceAPI::FreeDataSpace(Device dev, void* ptr)
```

参数说明

- Device& dev: 通过 DLDevice 创建的 device 结构体。
- void* ptr: 设备指针。

返回值

无返回值。

使用示例

```
DLDevice dev{kDLILUVATAR, 0};
DeviceAPI* api = DeviceAPI::Get(dev);
void* ptr;
api->FreeDataSpace(dev, ptr);
```

9.5 Tensor 相关类和函数

Tensor 是深度学习常用的数据结构，通常表示为一个 N 维的数组。

9.5.1 DLTensor 结构体

功能概述：

DLTensor 是 IGIE 表示 Tensor 的基础结构体，在 NDArray 中被使用。该结构体储存了 Tensor 的数据指针、设备类型、维度、shape 信息、内存 stride 和内存 offset 等 Memory 层面的信息。

结构体原型

DLTensor 定义如下：

```
typedef struct {
/*!
   * \brief The opaque data pointer points to the allocated data. This will be
   * CUDA device pointer or cl_mem handle in OpenCL. This pointer is always
   * aligned to 256 bytes as in CUDA.
   */
void* data;
/*! \brief The device of the tensor */
DLDevice device;
/*! \brief Number of dimensions */
int ndim;
/*! \brief The data type of the pointer*/
DLDataType dtype;
/*! \brief The shape of the tensor */
int64_t* shape;
/*!
   * \brief strides of the tensor (in number of elements, not bytes)
   *  can be NULL, indicating tensor is compact and row-majored.
   */
int64_t* strides;
/*! \brief The offset in bytes to the beginning pointer to data */
uint64_t byte_offset;
} DLTensor;
```

参数说明

- `void* data`: 数据指针，可以是 CPU Memory 指针，也可以是 CUDA Memory 指针。
- `DLDevice device`：指定 tensor 所在设备。
- `int ndim`：Tensor 维度。
- `DLDataType dtype`: Tensor 的数据类型。
- `int64_t* shape`: Tensor 的 shape 信息。
- `int64_t* strides`: Tensor 的 stride，stride 的单位是 element 的个数，而非 nbyte。
- `uint64_t byte_offset`: 相当于 data 数据开始位置的偏移量。

使用示例

```
DLTensor from;
from.data = const_cast<void*>(data);
from.device = DLDevice{kDLILUVATAR, 0};
from.dtype = DLDataType{kDLFloat, 32, 1};
ShapeTuple shape{1, 224, 224, 3};
from.ndim = static_cast<int>(shape.size());
from.shape = const_cast<int64_t *>(shape.data());
from.strides = nullptr;
from.byte_offset = 0;
```

9.5.2 NDArray 类

NDArray 是表示 N 维的数组的类，主要用于创建模型的输入和输出。

9.5.3 NDArray::Empty()

功能概述

创建一个空 NDArray。

函数原型

```
static NDArray tvm::runtime::NDArray::Empty(ShapeTuple shape, DLDataType dtype, Device dev,
↪  Optional<String> mem_scope = NullOpt)
```

参数说明

- `ShapeTuple shape`: 新 array 的 shape 大小。
- `DLDataType dtype`: 新 array 的数据类型。
- `Device& dev`: 通过 DLDevice 创建的 device 结构体。
- `mem_scope`: 内存类型。

返回值

返回 NDArray 对象。

使用示例

```
DLDevice dev{kDLILUVATAR, 0};
tvm::runtime::NDArray x;
x = tvm::runtime::NDArray::Empty({2, 2}, DLDataType{kDLFloat, 32, 1}, dev);
```

9.5.4 NDArray::Shape()

功能概述

获取 NDArray 的 Shape 信息。

函数原型

```
ShapeTuple tvm::runtime::NDArray::Shape()
```

参数说明

无参数

返回值

返回 ShapeTuple 对象。

使用示例

```
tvm::runtime::NDArray input;
tvm::runtime::ShapeTuple input_shape = input.Shape();

// 打印 shape 信息
std::cout << "Input Shape: [";
for (int i=0; i < input_shape.size(); i++){
std::cout << input_shape.data()[i]  << ",";
}
std::cout << "]" << std::endl;
```

9.5.5 NDArray::DataType()

功能概述

获取 NDArray 的 Type 信息。

函数原型

```
DataType tvm::runtime::NDArray::DataType()
```

参数说明

无参数

返回值

返回 DataType 对象。

使用示例：

```
tvm::runtime::NDArray input;
tvm::runtime::DataType input_type = input.DataType();

// 打印 Dtype 信息
std::string dtype =  tvm::runtime::DLDataType2String(input_type);
std::cout << "Input Type: " << dtype << std::endl;
```

9.5.6 NDArray::CopyFromTo()

功能概述

拷贝一个 DLTensor 数据到另一个 DLTensor，例如，拷贝 CPU 上的 DLTensor 数据到 GPU DLTensor。

函数原型

```
static void tvm::runtime::NDArray::CopyFromTo(const DLTensor* from, DLTensor* to,
↪  TVMStreamHandle stream = nullptr)
```

参数说明

- `DLTensor* from`: 源 DLTensor。
- `DLTensor* to`: 目标 DLTensor。
- `TVMStreamHandle stream`: 执行拷贝操作的 stream，默认是 nullptr stream。

返回值

无返回值。

使用示例

```
DLTensor from;
from.data = const_cast<void*>(data);
from.device = DLDevice{kDLCPU, 0};
from.dtype = DLDataType{kDLFloat, 32, 1};
ShapeTuple shape{1, 224, 224, 3};
from.ndim = static_cast<int>(shape.size());
from.shape = const_cast<int64_t *>(shape.data());
from.strides = nullptr;
from.byte_offset = 0;

DLTensor to;
to.device = DLDevice{kDLILUVATAR, 0};

tvm::runtime::NDArray::CopyFromTo(from, to)
```

9.5.7 NDArray::CopyToBytes()

功能概述

拷贝数据内容到另一个 NDArray，例如，拷贝 CPU 上的 data 数据到 GPU DLTensor。

函数原型

```
void tvm::runtime::NDArray::CopyToBytes(void* data, size_t nbytes)
```

参数说明

- `void* data`: 数据指针。
- `size_t nbytes`: 拷贝数据大小 bytes。

返回值

无返回值。

使用示例

```
// 例如：创建一个 shape=[4]，dtype 类型为 float32，在 Iluvatar GPU 上的 Tensor

// 设置 Tensor 位于的设备
DLDevice dev{kDLILUVATAR, 0};
// 设置 Tensor 的类型
DLDataType dtype{kDLFloat, 32, 1};
tvm::runtime::NDArray input = tvm::runtime::NDArray::Empty({4}, dtype, dev);
// 拷贝 CPU 数据 data 到 GPU Tensor 中
float data[4] = {1.0, 2.0, 3.0, 4.0};
input.CopyFromBytes(data, 4 * sizeof(float));
```

9.5.8 NDArray::FromDLPack()

功能概述

从一个 dlpack tensor 创建 NDArray。

函数原型

```
static NDArray tvm::runtime::NDArray::FromDLPack(DLManagedTensor* tensor)
```

参数说明

`DLManagedTensor* tensor`: dlpack tensor。

返回值

返回 NDArray。

使用说明：

DLPack 是一种通用的 Tensor 数据结构，它允许不同框架之间的 Tensor 进行相互转换。DLPack 相关介绍可以查看 DLPack 文档。

因此，`tvm::runtime::NDArray::FromDLPack` 接口可以将 支持 `DLPack` 的其他框架 `Tensor` 转换为 `IGIE NDArray` 进行推理，支持的框架包括：Torch、TensorFlow、Paddle、ONNXRuntime。

转换过程如下：**torch tensor-> dlpack tensor -> igie tensor(NDArray)**

使用 `tvm::runtime::NDArray::FromDLPack` 创建 Tensor，支持零拷贝的方式创建 Tensor：

```
// 该接口支持以零拷贝的方式构造 IGIE Tensor, void *data 可以是 CUDA Memory 数据指针。
tvm::runtime::NDArray CreateNDArray(void *data,
                                    ShapeTuple &shape,
                                    const DLDataType &dtype,
                                    const DLDevice &dev) {
  // 创建 DLTensor
  DLTensor tensor;
  tensor.data = data;
  tensor.device = dev;
  tensor.ndim = static_cast<int>(shape.size());
  tensor.shape = const_cast<int64_t *>(shape.data());
  tensor.dtype = dtype;
  tensor.strides = nullptr;
  tensor.byte_offset = 0;

  // 创建 DLManagerTensor（DLPack 结构体）
  DLManagedTensor managed_tensor;
  managed_tensor.dl_tensor = tensor;
  managed_tensor.deleter = nullptr;

  tvm::runtime::NDArray x_array  = tvm::runtime::NDArray::FromDLPack(&managed_tensor);
  return x_array;
}

// GPU 内存的 data
float *d_data;
cudaMalloc((void**)&d_data, size * sizeof(float));
cudaMemcpy(d_data, h_data, size * sizeof(float), cudaMemcpyHostToDevice);

// 调用 CreateNDArray 函数，零拷贝方式创建 IGIE Tensor,
ShapeTuple in_shape{1, 224, 224, 3};
tvm::runtime::NDArray in_tensor;
in_tensor = CreateNDArray(mag_tensor, d_data, in_shape, DLDataType{kDLFloat, 32, 1}, dev);
```

9.6 加载模型引擎相关类和函数

9.6.1 Module 类

IGIE 中表示运行时模型的类。

9.6.1.1 Module::LoadFromFile()

功能概述

从引擎文件构造 Module 对象，用于推理。

函数原型

```
static Module tvm::runtime::Module::LoadFromFile(const std::string& file_name, const
↪  std::string& format = "")
```

参数说明

- const std::string& file_name: 引擎文件路径。
- const std::string& format = "": 引擎文件类型，默认为空。

返回值

返回 Module 对象。

使用说明:

```
// 加载引擎文件到 IGIE 中
tvm::runtime::Module mod_factory = tvm::runtime::Module::LoadFromFile("./resnet50.so");
```

9.6.1.2 Module::GetFunction("default")()

功能概述

创建 GraphExecutor 对象，加载 Module 到 Device GPU。

函数原型

```
PackedFunc tvm::runtime::Module::GetFunction("default")(DLDevice device)
```

参数说明

DLDevice device: 设备对象。

返回值

GraphExecutor 对象。

使用说明:

因为 GraphExecutor 的父类是 ModuleNode（class GraphExecutor : public ModuleNode()），因此，mod_factory.GetFunction("default")(dev) 返回的 GraphExecutor 对象可以是 Module 类型。

```
DLDevice dev{kDLILUVATAR, 0};
// 加载引擎文件到 IGIE 中
tvm::runtime::Module mod_factory = tvm::runtime::Module::LoadFromFile("./resnet50.so");
// 创建 GraphExecutor 对象，加载 Module 到 Device GPU。
tvm::runtime::Module graph_executor = mod_factory.GetFunction("default")(dev);
```

9.7 模型输入输出信息相关函数

9.7.1 GraphExecutor::GetFunction()

功能概述

通过统一的 GetFunction 接口，获取 GraphExecutor 的所有成员函数。比如：

- `get_input()`: 获取模型输入信息。
- `set_input()`: 设置模型输入。
- `set_input_zero_copy()`: 以零拷贝的形式设置模型输入。
- `get_output()`: 获取模型输出。
- `get_num_outputs()`：获取模型的输出个数。
- `get_num_inputs()`：获取模型的输入个数。
- `get_input_names()`：获取模型的输入 name。
- `get_output_names()`: 获取模型的输出 name。

函数原型

```
PackedFunc GraphExecutor::GetFunction(const std::string& name)
```

参数说明

`const std::string& name`: 函数名字符串。

返回值

返回函数对象，可以直接调用。

使用说明：

```
DLDevice dev{kDLILUVATAR, 0};
// 加载引擎文件到 IGIE 中
tvm::runtime::Module mod_factory = tvm::runtime::Module::LoadFromFile("./resnet50.so");
// 创建 GraphExecutor 对象，加载 Module 到 Device GPU。
tvm::runtime::Module graph_executor = mod_factory.GetFunction("default")(dev);

//获取模型输入个数
tvm::runtime::PackedFunc get_num_inputs = graph_executor.GetFunction("get_num_inputs");

//获取模型输出个数
tvm::runtime::PackedFunc get_num_outputs = graph_executor.GetFunction("get_num_outputs");

// 获取模型全部输入 names
tvm::runtime::PackedFunc get_input_names = graph_executor.GetFunction("get_input_names");

// 获取模型全部输出 names
tvm::runtime::PackedFunc get_output_names = graph_executor.GetFunction("get_output_names");
```

```
// 获取模型输入信息
tvm::runtime::PackedFunc set_input = graph_executor.GetFunction("get_input");

// 设置模型输入函数
tvm::runtime::PackedFunc set_input = graph_executor.GetFunction("set_input");

// 设置模型输入函数
tvm::runtime::PackedFunc set_input_zero_copy =
 ↪  graph_executor.GetFunction("set_input_zero_copy");

// 获取模型输出函数
tvm::runtime::PackedFunc get_output = graph_executor.GetFunction("get_output");
```

9.7.2 get_num_inputs()

功能概述

获取模型输入的个数。

函数原型

```
GraphExecutor::GetFunction("get_num_inputs")()
```

参数说明

无

返回值

返回模型输入的个数。

使用说明:

```
DLDevice dev{kDLILUVATAR, 0};
// 加载引擎文件到 IGIE 中
tvm::runtime::Module mod_factory = tvm::runtime::Module::LoadFromFile("./resnet50.so");
// 创建 GraphExecutor 对象，加载 Module 到 Device GPU。
tvm::runtime::Module graph_executor = mod_factory.GetFunction("default")(dev);

//获取模型输入个数
tvm::runtime::PackedFunc get_num_inputs = graph_executor.GetFunction("get_num_inputs");
int input_num = get_num_inputs();
std::cout << "get_num_inputs: " << input_num << std::endl;
```

9.7.3 get_num_outputs()

功能概述

获取模型输出的个数。

函数原型

```
int GraphExecutor::GetFunction("get_num_outputs")()
```

参数说明

无

返回值

返回模型输出的个数。

使用说明：

```
DLDevice dev{kDLILUVATAR, 0};
// 加载引擎文件到 IGIE 中
tvm::runtime::Module mod_factory = tvm::runtime::Module::LoadFromFile("./resnet50.so");
// 创建 GraphExecutor 对象，加载 Module 到 Device GPU。
tvm::runtime::Module graph_executor = mod_factory.GetFunction("default")(dev);

//获取模型输出个数
tvm::runtime::PackedFunc get_num_outputs = graph_executor.GetFunction("get_num_outputs");
int output_num = graph_executor.NumOutputs();
std::cout << "get_num_outputs: " << output_num << std::endl;
```

9.7.4 get_input_names()

功能概述

获取模型所有输入 Name。

函数原型

```
std::vector<std::string> GraphExecutor::GetFunction("get_input_names")()
```

参数说明

无

返回值

返回模型输入名数组。

使用说明：

```
DLDevice dev{kDLILUVATAR, 0};
// 加载引擎文件到 IGIE 中
tvm::runtime::Module mod_factory = tvm::runtime::Module::LoadFromFile("./resnet50.so");
```

```
// 创建 GraphExecutor 对象，加载 Module 到 Device GPU。
tvm::runtime::Module graph_executor = mod_factory.GetFunction("default")(dev);

// 获取模型全部输入 names
tvm::runtime::PackedFunc get_input_names = graph_executor.GetFunction("get_input_names");
void* input_names = get_input_names();
std::vector<std::string>* vector_names = static_cast<std::vector<std::string>*>(input_names);
for (const auto& str : *vector_names) {
  std::cout << "Input Name: " << str << std::endl;
}
```

9.7.5 get_output_names()

功能概述

获取模型所有输出 Name。

函数原型

```
std::vector<std::string> GraphExecutor::GetFunction("get_output_names")()
```

参数说明

无

返回值

返回模型输出名数组。

使用说明：

```
DLDevice dev{kDLILUVATAR, 0};
// 加载引擎文件到 IGIE 中
tvm::runtime::Module mod_factory = tvm::runtime::Module::LoadFromFile("./resnet50.so");
// 创建 GraphExecutor 对象，加载 Module 到 Device GPU。
tvm::runtime::Module graph_executor = mod_factory.GetFunction("default")(dev);

// 获取模型全部输出 names
tvm::runtime::PackedFunc get_output_names = graph_executor.GetFunction("get_output_names");
void* output_names = get_output_names();
vector_names = static_cast<std::vector<std::string>*>(output_names);
for (const auto& str : *vector_names) {
  std::cout << str << std::endl;
}
```

9.7.6 get_input_index()

功能概述

根据输入名称获取输入的 index。

函数原型

```
int GraphExecutor::GetFunction("get_input_index")(const std::string& name)
```

参数说明

const std::string& name: 模型输入名。

返回值

输入的 index。

使用说明:

```
DLDevice dev{kDLILUVATAR, 0};
// 加载引擎文件到 IGIE 中
tvm::runtime::Module mod_factory = tvm::runtime::Module::LoadFromFile("./resnet50.so");
// 创建 GraphExecutor 对象，加载 Module 到 Device GPU。
tvm::runtime::Module graph_executor = mod_factory.GetFunction("default")(dev);
tvm::runtime::PackedFunc get_output_names = graph_executor.GetFunction("get_input_index");

int index = get_output_names("input_0");
```

9.7.7 get_output_index()

功能概述

根据输出名称获取输出的 index。

函数原型

```
int GraphExecutor::GetFunction("get_output_index")(const std::string& name)
```

参数说明

const std::string& name: 模型输出名。

返回值

输出的 index。

使用说明:

```
DLDevice dev{kDLILUVATAR, 0};
// 加载引擎文件到 IGIE 中
tvm::runtime::Module mod_factory = tvm::runtime::Module::LoadFromFile("./resnet50.so");
// 创建 GraphExecutor 对象，加载 Module 到 Device GPU。
tvm::runtime::Module graph_executor = mod_factory.GetFunction("default")(dev);
tvm::runtime::PackedFunc get_output_names = graph_executor.GetFunction("get_output_index");

int index = get_output_index("softmax_0");
```

9.7.8 get_input()

功能概述

设置模型输入，获取输入的 Shape、Dtype 和 data 信息。

函数原型

```
NDArray GraphExecutor::GetFunction("get_input")(int index)
```

参数说明

int index: 第几个输入。

返回值

返回 NDArray。

使用说明：

```
DLDevice dev{kDLILUVATAR, 0};
// 加载引擎文件到 IGIE 中
tvm::runtime::Module mod_factory = tvm::runtime::Module::LoadFromFile("./resnet50.so");
// 创建 GraphExecutor 对象，加载 Module 到 Device GPU。
tvm::runtime::Module graph_executor = mod_factory.GetFunction("default")(dev);
// 获取模型输入信息
tvm::runtime::PackedFunc get_input = gmod.GetFunction("get_input");
tvm::runtime::NDArray input_ = get_input(0);
tvm::runtime::ShapeTuple input_shape = input_.Shape();
tvm::runtime::DataType input_type = input_.DataType();

// 打印 shape 信息
std::cout << "Input Shape: [";
for (int i=0; i < input_shape.size(); i++){
  std::cout << input_shape.data()[i]  << ",";
}
std::cout << "]" << std::endl;
// 打印 Dtype 信息
```

```
std::string dtype =  tvm::runtime::DLDataType2String(input_type);
std::cout << "Input Type: " << dtype << std::endl;
```

9.7.9 set_input()

功能概述

设置模型输入。

函数原型

```
void GraphExecutor::GetFunction("set_input")(int index, DLTensor* data_in)
```

参数说明

- int index: 第几个输入。
- DLTensor* data_in: 输入的 DLTensor。

返回值

无返回值。

使用说明:

```
DLDevice dev{kDLILUVATAR, 0};
// 加载引擎文件到 IGIE 中
tvm::runtime::Module mod_factory = tvm::runtime::Module::LoadFromFile("./resnet50.so");
// 创建 GraphExecutor 对象，加载 Module 到 Device GPU。
tvm::runtime::Module graph_executor = mod_factory.GetFunction("default")(dev);
// 获取模型输入信息
tvm::runtime::PackedFunc set_input = graph_executor.GetFunction("get_input");

// 设置模型输入函数
tvm::runtime::PackedFunc set_input = graph_executor.GetFunction("set_input");

DLTensor* input;
// 通过指定 input name 设置输入
set_input("data", input);
// 或者 通过指定 index 设置输入
set_input(0, input);
```

9.7.10 set_input_zero_copy()

功能概述

以零拷贝的方式设置模型输入。如果是在 GPU 运行模型，则必须保证 data_ref 中的 data 指针指向 GPU 内存，该接口本质是只改变 DLTensor 的 data 指针。

函数原型

```
void GraphExecutor::GetFunction("set_input_zero_copy")(int index, DLTensor* data_ref)
```

参数说明

- `int index`: 第几个输入。
- `DLTensor* data_ref`: 输入的 DLTensor。如果是在 GPU 运行模型，则必须保证 data_ref 中的 data 指针指向 GPU 内存。

返回值

无返回值。

使用说明:

```
DLDevice dev{kDLILUVATAR, 0};
// 加载引擎文件到 IGIE 中
tvm::runtime::Module mod_factory = tvm::runtime::Module::LoadFromFile("./resnet50.so");
// 创建 GraphExecutor 对象，加载 Module 到 Device GPU。
tvm::runtime::Module graph_executor = mod_factory.GetFunction("default")(dev);
// 设置模型输入函数
tvm::runtime::PackedFunc set_input_zero_copy =
 ↪  graph_executor.GetFunction("set_input_zero_copy");

DLTensor* data_ref;
int index = set_input_zero_copy(data_ref);
```

9.7.11 get_output()

功能概述

获取模型的输出。

函数原型

```
NDArray GraphExecutor::GetFunction("get_output")(int index)
```

参数说明

`int index`: 第几个输出。

返回值

返回 NDArray。

使用说明:

```
DLDevice dev{kDLILUVATAR, 0};
// 加载引擎文件到 IGIE 中
tvm::runtime::Module mod_factory = tvm::runtime::Module::LoadFromFile("./resnet50.so");
// 创建 GraphExecutor 对象，加载 Module 到 Device GPU。
tvm::runtime::Module graph_executor = mod_factory.GetFunction("default")(dev);
// 获取模型输出函数
tvm::runtime::PackedFunc get_output = graph_executor.GetFunction("get_output");

tvm::runtime::NDArray output = get_output(0);

tvm::runtime::ShapeTuple output_shape = output.Shape();
tvm::runtime::DataType output_type = output.DataType();

// 打印 shape 信息
std::cout << "Output Shape: [";
for (int i=0; i< output_shape.size(); i++){
  std::cout << output_shape.data()[i]  << ",";
}
std::cout << "]" << std::endl;

// 打印 Dtype 信息
std::string dtype =  tvm::runtime::DLDataType2String(output_type);
std::cout << "Output Type: " << dtype << std::endl;
```

9.8 模型运行相关函数

9.8.1 run()

功能概述

模型运行函数。

函数原型

```
GraphExecutor::GetFunction("run")()
```

参数说明

无。

返回值

无返回。

使用说明：

```
DLDevice dev{kDLILUVATAR, 0};
// 加载引擎文件到 IGIE 中
tvm::runtime::Module mod_factory = tvm::runtime::Module::LoadFromFile("./resnet50.so");
// 创建 GraphExecutor 对象，加载 Module 到 Device GPU。
tvm::runtime::Module graph_executor = mod_factory.GetFunction("default")(dev);
tvm::runtime::PackedFunc run = graph_executor.GetFunction("run");

run();
```

9.9 性能统计相关函数

9.9.1 GetTimer()

功能概述

创建 Timer 对象，用于耗时统计。

函数原型

```
tvm::runtime::Timer timer = tvm::runtime::Timer::GetTimer(dev);
```

参数说明

`Device& dev`: 通过 DLDevice 创建的 device 结构体。

返回值

Timer 对象。

使用说明:

通过传入不同类型的 Device 创建 Timer 对象，控制开启 GPU/CPU 耗时统计。

```
// CPU 耗时统计
DLDevice dev{kDLCPU, 0};
tvm::runtime::Timer timer = tvm::runtime::Timer::GetTimer(dev);

// GPU 耗时统计
DLDevice dev{kDLILUVATAR, 0};
tvm::runtime::Timer timer = tvm::runtime::Timer::GetTimer(dev);
```

9.9.2 Start()

功能概述

开启耗时统计。

参数说明

无。

返回值

无返回值。

使用说明：

```
timer->Start();
```

9.9.3 Stop()

功能概述

关闭耗时统计。

参数说明

无。

返回值

无返回值。

使用说明：

```
timer->Stop();
```

9.9.4 SyncAndGetElapsedNanos()

功能概述

计算 Start 和 Stop 之间的耗时。

参数说明

无。

返回值

返回耗时统计，类型为 int64_t，单位为 ns。

使用说明：

```
int64_t time = timer->SyncAndGetElapsedNanos();
```

9.10 推理完整参考示例

Note

IGIE C++ 示例代码不在本发布包中。如有需要，请向您的应用工程师获取示例代码。

下面是完整的 C++ 代码：

```
#include <dlpack/dlpack.h>
#include <tvm/runtime/module.h>
#include <tvm/runtime/packed_func.h>
#include <tvm/runtime/registry.h>
#include <tvm/runtime/ndarray.h>
#include <tvm/runtime/profiling.h>
#include <cstdio>
#include <stdlib.h>
#include <iostream>
#include <iomanip>
#include <string>
#include <fstream>
#include <errno.h>
#include <exception>

// 读取图片数据到 data
bool GetImageData(std::string file_path, float *data, uint32_t size) {
try {
   std::ifstream file;
   file.exceptions(file.failbit | file.badbit);
   file.open(file_path);
   if (file.is_open()) {
      float tmp;
      uint32_t count = 0;
      while (file >> tmp) {
      *data++ = tmp;
      count++;
      if (count >= size) {
         break;
      }
      }
      file.close();
      return true;
   }
} catch (std::ifstream::failure &e) {
   std::cout << "open file failed: " << e.what() << std::endl;
   return false;
}
return false;
```

```
}

// 保存推理结果
bool SaveOutput(const std::string &file_path, float *data, uint32_t size) {
try {
   std::ofstream out_file;
   out_file.exceptions(out_file.failbit | out_file.badbit);
   out_file.open(file_path);
   if (out_file.is_open()) {
      float tmp;
      for (uint32_t i = 0; i < size; ++i) {
      tmp = *data++;
      out_file << tmp;
      out_file << '\n';
      }
      out_file.close();
      return true;
   }
} catch (std::ifstream::failure &e) {
   std::cout << "open file failed: " << e.what() << std::endl;
   return false;
}
return false;
}

// 使用引擎文件部署模型
void DeployResnet50(std::string in_file, std::string out_file, std::string dev_type = "gpu") {
LOG(INFO) << "Running C++ resnet50 graph executor...";

//读取测试图片数据
float *data = new float[1 * 3 * 224 * 224];
if (!GetImageData(in_file, data, 1*3*224*224)) {
   std::cout << "Get data failed!" << std::endl;
   return;
}

// 设置 device
DLDevice dev{kDLILUVATAR, 0};
tvm::runtime::Module mod_factory;

// 加载引擎文件
if (dev_type == "cpu") {
   dev.device_type = kDLCPU;
   mod_factory = tvm::runtime::Module::LoadFromFile("model/resnet50_b1_llvm.so");
} else if (dev_type == "gpu") {
   dev.device_type = kDLILUVATAR;
```

```
    mod_factory = tvm::runtime::Module::LoadFromFile("model/resnet50_b1_iluvatar.so");
}else {
    std::cout << "ERROR: unrecognized device type!" << std::endl;
    return;
}

// create the graph executor module
tvm::runtime::Module gmod = mod_factory.GetFunction("default")(dev);

//获取模型输入个数
tvm::runtime::PackedFunc get_num_inputs = gmod.GetFunction("get_num_inputs");
int input_num = get_num_inputs();
std::cout << "get_num_inputs: " << input_num << std::endl;

//获取模型输出个数
tvm::runtime::PackedFunc get_num_outputs = gmod.GetFunction("get_num_outputs");
int output_num = get_num_outputs();
std::cout << "get_num_outputs: " << output_num << std::endl;

// 获取模型全部输入 names
tvm::runtime::PackedFunc get_input_names = gmod.GetFunction("get_input_names");
void* input_names = get_input_names();

std::vector<std::string>* vector_names = static_cast<std::vector<std::string>*>(input_names);
for (const auto& str : *vector_names) {
    std::cout << str << std::endl;
}

// 获取模型全部输出 names
tvm::runtime::PackedFunc get_output_names = gmod.GetFunction("get_output_names");
void* output_names = get_output_names();

vector_names = static_cast<std::vector<std::string>*>(output_names);
for (const auto& str : *vector_names) {
    std::cout << str << std::endl;
}

tvm::runtime::PackedFunc get_input = gmod.GetFunction("get_input");
tvm::runtime::NDArray input_ = get_input(0);
tvm::runtime::ShapeTuple input_shape = input_.Shape();
tvm::runtime::DataType input_type = input_.DataType();
std::cout << "Input Shape: [";
for (int i=0; i< input_shape.size(); i++){
    std::cout << input_shape.data()[i]  << ",";
}
std::cout << "]" << std::endl;
```

```
std::string dtype =  tvm::runtime::DLDataType2String(input_type);
std::cout << "Input Type: " << dtype << std::endl;

//  获取模型推理函数
tvm::runtime::PackedFunc set_input = gmod.GetFunction("set_input");
tvm::runtime::PackedFunc get_output = gmod.GetFunction("get_output");
tvm::runtime::PackedFunc run = gmod.GetFunction("run");

// 创建空的 Tensor
tvm::runtime::NDArray input = tvm::runtime::NDArray::Empty({1, 3, 224, 224},
 ↪  DLDataType{kDLFloat, 32, 1}, dev);

// 创建 Timer
tvm::runtime::Timer timer = tvm::runtime::Timer::GetTimer(dev);

// Warm Up
std::cout << "Start Warmup ..." << std::endl;
for(size_t i = 0; i < 3; i++)
{
   run();
}

// 拷贝 Host data 到 GPU input tensor 中 (HToD)
input.CopyFromBytes(data, 1 * 3 * 224 * 224 * sizeof(float));

// 设置模型输入
set_input("data", input);

// 推理模型
timer->Start();
run();
timer->Stop();

// 计算耗时
float latency = timer->SyncAndGetElapsedNanos() / 1e6;   // ns->ms
float FPS = 1000 / time;
std::cout << "Consume time: " << latency << " ms" << std::endl;
std::cout << "FPS:  " << FPS << std::endl;

// get the output
tvm::runtime::NDArray output_gpu;
output_gpu = get_output(0);

tvm::runtime::ShapeTuple output_shape = output_gpu.Shape();
tvm::runtime::DataType output_type = output_gpu.DataType();
```

```
std::cout << "Output Shape: [";
for (int i=0; i< output_shape.size(); i++){
   std::cout << output_shape.data()[i]  << ",";
}
std::cout << "]" << std::endl;

// 打印 Dtype 信息
dtype =  tvm::runtime::DLDataType2String(output_type);
std::cout << "Output Type: " << dtype << std::endl;

// 拷贝 GPU Tensor 数据到 Host output 中 (DtoH)
float output[1000];
output_gpu.CopyToBytes((void*)output, 1000 * sizeof(float));

for (int32_t i = 0; i < 10; ++i) {
   std::cout << ((float*)output)[i] << '\t';
}
std::cout << std::endl;

// save inference output to postprecess
SaveOutput(out_file, output, 1000);

if (data != nullptr) {
   LOG(INFO) << "free data when device type is cuda!!!";
   delete[] data;
}
LOG(INFO) << "End running resnet50 graph executor...";
}


int main(int argc, char *argv[]) {
if (argc != 4) {
   std::cout << "ERROR: two arguments for main, but get " << argc - 1 << std::endl;
   return 1;
}
std::string in_file(argv[1]);
std::string out_file(argv[2]);
std::string dev_type(argv[3]);

std::cout << "in_file: " << in_file << ", out_file: " << out_file << std::endl;
DeployResnet50(in_file, out_file, dev_type);
return 0;
}
```

执行：

```
$ cd igie/tests/test_iluvatar/test_cpp/
$ bash run.sh
```

10 常见问题

问题 1：IGIE 的 Relay OP 是什么?

Relay 是 IGIE Graph 和 OP 的集合，所有算子的定义都在 Relay Package 下面。因此 IGIE 的算子有时候也会被称为 Relay OP。

您在使用 IGIE 时，可以通过 Relay OP 构造网络，例如：

```
data = relay.var("data", shape=input_shape, dtype=dtype)
conv = relay.nn.conv2d(data, weight)
add = relay.add(conv, bias)
relu = relay.nn.relu(add)
```

您也可以使用 `relay.build` 进行模型 Graph 编译：

```
engine = relay.build(mod, params=params, target=target)
```

问题 2：推理模型时，使用什么 Layout 性能最好?

目前天数智芯高性能算子库的开发，针对 `NHWC Layout` 算子适配程度和性能优化都是最好的，对于 `NCHW Layout` 的算子目前优化支持较少。

因此，建议您导入模型后尽量使用 `NHWC Layout` 进行模型推理。

如果模型内部使用的是 `HCHW Layout`，可以在导入模型后使用下面代码对 Layout 进行转换，注意输入输出的 Layout 是不会改变的：

```
from tvm.relay.transform.iluvatar.optimize_graph import convert_graph_layout_simplify
# NCHW -> NHWC
convert_layout="NHWC"
mod = convert_graph_layout_simplify(mod, convert_layout=convert_layout)
```

问题 3：运行模型时，出现" 现在不支持，以后会支持" 的屏幕输出，是怎么回事?

目前 ixInfer 算子库有部分 Shape 还没有完全支持，遇到不支持的 `conv2d` 实现会输出" 现在不支持，以后会支持" 字样，您需要将该 Shape 提交给您的应用工程师，应用工程师将进一步提交给对应研发进行处理；或者您可以将该 Shape 暂时回退使用 IGIE 的默认实现。

问题 4：igie-exec 工具量化表生成

问题描述

IGIE 对于不同 Batch 的 INT8 推理，是否每个 Batch 都需要重新量化？量化表是否生成一次即可?

问题解答

目前 igie-exec 提供了两种量化方案供您选择：IGIE 量化和 ONNXRuntime 量化。

- 对于 IGIE 量化 (**--igie-quantize** 参数)：
  - 优点：适用大部分模型，支持 FP16 和 INT8 量化
  - 缺点：无法控制每一个量化节点，只能跳过一类节点

- 对于 ONNXRuntime 量化 (**--onnx-quantize** 参数)：
  - 优点：对于 ONNX 模型支持较好，可以配置参数选择哪些节点不量化，哪些节点量化，控制粒度更细
  - 缺点：只支持 ONNX 模型

使用 **--igie-quantize** 参数进行 INT8 量化时，会自动将量化系数 (量化表) 保存在当前目录的 `quantize_scale` 目录中，下次进行其他 Batch 推理时，会自动加载量化表，无需再次量化。当 Batch 较大时，会导致模型量化很慢或者内存溢出 OOM，推荐使用 `batch=1` 进行 INT8 量化，然后再执行其他 Batch 推理，因为不同的 Batch 使用的量化系数是一样的。

使用 **--onnx-quantize** 参数进行 INT8 量化时，igie-exec 也推荐先对 1 batch 进行量化，再次进行 32/64/128 batch 推理时即不再需要量化。igie-exec 内部有针对 ONNX 模型的 Batch 自动修改工具，可以在 1 batch 已量化模型的基础上修改 Batch。

问题 5：在 ARM 设备上安装 onnxoptimizer 和 onnxsim 报错

问题描述

在 ARM 设备上安装 igie-exec 的 requirement.txt 后报错。

问题解答

如果是编译安装 onnxoptimizer，可能是因为 CMake 版本过低导致的编译失败，需要升级 CMake。一般错误信息如下：

```
CMake Error at CMakeLists.txt:1 (cmake_minimum_required):
        CMake 3.22 or higher is required.  You are running version 3.16.9
```

请参考下列步骤解决问题：

1. 升级 CMake 版本：

```
# 向您的应用工程师获取 cmake.git 文件

$ ./build_cmake.sh
$ ./install_cmake.sh
```

2. 升级完成后，查看 CMake 版本：

```
$ cmake --version
cmake version 3.25.2-corex.3.1.0.20230322.534
```

3. 重新安装 onnxoptimizer。

问题 6：如何使用 IGIE DEBUG 模式?

您可以通过编译选项 **ILUVATAR_LOG_DEBUG** 来控制是否打印 DEBUG 信息，在 `igie/cmake/config.cmake` 文件中将该选项设置为 `ON`，重新编译 IGIE 后即可看到 DEBUG 信息。

问题 7：如何开启 IGIE 多算子库支持?

IGIE 可以在以下 4 种模式下工作，分别能够支持不同的算子库：

- **llvm**：模型中的所有 OP 全部走 IGIE CPU 实现，性能最差，一般用于精度对比
- **iluvatar**：模型中的所有 OP 全部走 IGIE GPU 实现，性能相对较差，通用性高
- **iluvatar_with_cudnn_cublas**：从 ixDNN、ixBLAS 库中寻找性能相对较高的实现，通用性取决于 ixDNN、ixBLAS 的实现情况
- **iluvatar_with_all_libs**：从 ixInfer、ixDNN、ixBLAS 库中寻找性能最好的实现，能够获取最佳性能，通用性取决于 ixInfer、ixDNN、ixBLAS 的实现情况

llvm 模式使用 CPU 计算，在实际模型适配中很少用到。其余三种模式：

- 通用性和精度：iluvatar > iluvatar_with_cudnn_cublas > iluvatar_with_all_libs
- 性能：iluvatar_with_all_libs > iluvatar > iluvatar_with_cudnn_cublas

在编译生成引擎文件时，您可以通过设置 **target** 参数来控制 IGIE 工作在何种模式下：

```
# llvm
target = tvm.target.Target("llvm")

# iluvatar
target = tvm.target.iluvatar()

# iluvatar_with_cudnn_cublas
target = tvm.target.iluvatar(options="-libs=cudnn,cublas")

# iluvatar_with_all_libs
target = tvm.target.iluvatar(options="-libs=cudnn,cublas,ixinfer")
```

模型适配中遇到无法运行或者是精度问题，您可以逐步回退到 iluvatar 工作模式来验证模型的支持情况。

问题 8：IGIE 推理程序 Hang，如何解决?

IGIE INT8 推理可能会出现程序 Hang 住，这有可能是因为调用了 ixInfer 算子库时遇到了未完全支持的实现，可以调整 **target** 参数中的选项，运行不使用 ixInfer 算子库的引擎文件来进一步确认问题：

```
target = tvm.target.iluvatar(options="-libs=cudnn,cublas")
```

确认为 ixInfer 问题后，在 IGIE 层面还可以进行如下尝试：

- 尝试切换 ixInfer 算法实现：ixInfer 提供了两种算法选择，在调用 ixInfer INT8 算子时，IGIE 默认选择算法 0 的实现。将 `python/tvm/contrib/ixinfer.py` 脚本下 `qdconv_forward` 和 `qdeconv_forward` 函数中的 algo 设置为 1，可快速切换到 ixInfer 的另一种算法实现。另外，切换算法实现后，需要重新编译生成引擎文件。
- 尝试对模型进行 Pad：当模型中的 OP 输入、输出 Channel 为 64 的倍数时，ixInfer 在精度和性能上往往会有较好的表现。您可以调用 `IGIE PadNet` 接口对整个模型进行 64 倍数的 Pad，接口调用非常简单，在构建引擎文件时额外传参即可：

  ```
  engine = relay.build(mod, target=target, params=params, required_pass=['MutatePadMod'])
  ```

  该功能也包含在 igie-exec 中，您只需额外输入 **--custom_option required_pass:MutatePadMod**

参数即可开启。下面以 YOLOv5 模型为例：

```
$ igie-exec --model_path yolov5m.onnx --input images:32,3,640,640 --precision int8 --
↪  use_coco2017 True --automatic_yolo_quantization True --custom_option
↪  required_pass:MutatePadMod
```

问题 9：IGIE 推理出现精度问题，如何快速定位解决？

下面按照推理的精度类型分别讨论。

FP16 推理

FP16 推理不涉及到模型量化，出现精度问题的可能性比较小。如果出现精度问题，可以遵循以下步骤分析定位问题：

1. 使用模型原生框架，确认模型预处理、后处理计算无误；
2. 逐步剥离算子库支持，通过修改 **target** 依次回退，确认是否为算子库计算错误。

INT8 推理

INT8 推理首先需要排除模型量化本身出现错误的可能：

1. 同样需要确定模型预处理、后处理计算无误，尤其是预处理。如果模型预处理错误，量化会带来极大的精度损失。
2. IGIE 目前推荐使用 ONNXRuntime 量化，量化后的模型可以使用 ONNXRuntime 进行精度验证。

如果量化后的模型本身有精度损失，这个在问题 10 中单独讨论。确认量化后的模型精度无误，导入 IGIE 后精度错误，一般也可以通过逐步剥离算法库支持来分析定位问题。总之，IGIE 在 **iluvatar target** 会有最佳的精度表现，可以优先尝试修改 **target**。

假如您已经通过上述步骤，定位到了问题出现在 ixInfer 算法库上，那么同样可以参考问题 8 中提到的两种方法尝试解决：

1. 切换 ixInfer 的算法实现
2. 调用 IGIE `PadNet` 接口

问题 10：如何解决 INT8 量化本身带来的精度损失？

关于 IGIE INT8 量化的介绍，可以参考” 快速上手：IGIE 推理框架使用教程” > ” 教程 2：如何对模型进行量化”。目前 IGIE 更推荐使用 ONNXRuntime 进行量化，并提供以下配置选项控制量化行为：

- `op_types_to_quantize`：指定要量化的 OP 类型，例如 `['Conv']` 仅量化 Conv OP
- `nodes_to_exclude`：指定需要跳过量化的 Node List，设置该参数后，List 中的 Node 会跳过量化，例如 `['Conv_0', 'Conv_1']`
- `per_channel`：是否开启 per-channel 量化选项，相比 per-tensor，per-channel 一般能获得更好的精度

下面列出一些可能会引起量化精度损失的情况和解决方法：

序号	问题描述	解决方法
1	模型中存在 GroupConv、DepthWiseConv 量化带来精度损失	设置 per_channel=True，使用 per-channel 量化
2	YOLO 系列模型后处理部分量化后，带来精度损失	将后处理部分的相关 Node 添加到 `nodes_to_exclude` 中
3	不清楚哪些层可能会带来精度损失	极端情况下，可以只量化 Conv

另外，igie-exec 提供了 **--automatic_yolo_quantization** 参数，用于自动跳过 YOLO 后处理的部分 OP。

问题 11：IGIE 是否支持非官方模型?

目前 IGIE ModelZoo 已经支持 120+ 个模型，CV 类包括但不限于：torchvision、MMClassification、MMDetection 的大多数模型，并且在持续扩充中。

在适配过程中，IGIE 展现了较强的通用性，即使模型并非官方模型，相信 IGIE 也可以覆盖。唯一需要注意的是，当模型中存在 Dynamic Input/Output 时，IGIE 的支持还不是那么完善。

另外，您也可以使用 igie-exec 工具来快速验证模型的支持情况，详情请参考”igie-exec 模型快速部署和验证工具使用指南”。

问题 12：如何解决 ONNX infer shape 报错?

首先解释下为何会出现该报错：igie-exec 内置自动修改模型 Batch 的功能，但是并非支持 ONNX 模型中的所有 OP。因此，修改后的模型在调用 ONNX shape inference 接口时可能会出现 `[ShapeInferenceError] Inferred shape and existing shape differ in dimension 0: (64) vs (2)`。

由于 ONNX shape inference 仅用作模型中间层 Tensor Shape 的推断，不是模型推理的必要执行条件，并且 IGIE 自身存在 Shape 推断逻辑，通常注释掉该接口调用并不影响模型推理，可以注释掉 `rewrite_batch_size` 函数中的 `onnx.shape_inference.infer_shapes` 调用。

另外，igie-exec 自动 Batch 修改并非适用于所有模型，如果 ONNX 模型来自于 PyTorch 框架，您可以尝试指定 Batch 或将 Batch 维度设置为 `dynamic_axes`。

问题 13：IGIE 能否提供 Dynamic Shape 支持?

在 Transformer 模型上，往往会遇到 Dynamic Shape 的情况，目前 IGIE 提供了一定范围的 Dynamic Shape 支持。具体的做法是：将 Dynamic Shape 的范围分档，编译多个引擎文件，对输入进行 Pad 以匹配对应的引擎文件，实现一定程度的支持。

问题 14：在 ARM 设备上编译模型出现 Compilation Error，如何解决?

受 GCC 限制，目前在 ARM 设备上编译和导出超过 2GB 的模型时，会出现 Compilation Error。您可以通过以下方式解决该问题：

1. 将 engine 和 params 分开导出：

```
# Export model
graph, lib, params = relay.build(mod, target, params=params, precision="fp32")

lib.export_library("deploy_lib.so")

with open("deploy_graph.json", "w") as fo:
   fo.write(graph)

with open("deploy_param.params", "wb") as fo:
   fo.write(relay.save_param_dict(params))
```

2. 加载导出的 engine 和 params：

```
# Load model
lib = tvm.runtime.load_module("deploy_lib.so")
graph = open("deploy_graph.json", "rb").read()
params = bytearray(open("deploy_param.params", "rb").read())

dev = tvm.device("iluvatar", 0)
module = graph_executor.create(graph, lib, dev)
module.load_params(params)
module.run()
```

3. 使用 C++ 接口加载导出的 engine 和 params：

```
// Load lib
tvm::runtime::Module lib = tvm::runtime::Module::LoadFromFile(module_file);

// Parser graph json
std::ifstream json_in(json_file.c_str());
if(json_in.fail())
{
throw std::runtime_error("could not open json file");
}

const std::string json_data((std::istreambuf_iterator<char>(json_in)),
                            std::istreambuf_iterator<char>());
json_in.close();

// Parser params
std::ifstream params_in(params_file.c_str(), std::ios::binary);
if(params_in.fail())
{
throw std::runtime_error("could not open params file");
}

const std::string params_data((std::istreambuf_iterator<char>(params_in)),
                              std::istreambuf_iterator<char>());
params_in.close();

TVMByteArray params_arr;
params_arr.data = params_data.c_str();
params_arr.size = params_data.length();

// Create Module
DLDevice dev{kDLILUVATAR, 0};
const auto runtime_create = *tvm::runtime::Registry::Get("tvm.graph_executor.create");
tvm::runtime::Module gmod = runtime_create(json_data, lib,
 ↪  static_cast<int>(dev.device_type), dev.device_id);

// Load params
tvm::runtime::PackedFunc load_params = gmod.GetFunction("load_params");
load_params(params_arr);
```

11 附录：IGIE 框架算子支持列表

11.1 IGIE 算子支持

算子	备注
abs	Get absolute value of the input element-wise.
acos	Take acos of input x.
acosh	Take acos of input x.
adaptive_avg_pool1d	1D adaptive average pooling operator.
adaptive_avg_pool2d	2D adaptive average pooling operator.
adaptive_avg_pool3d	3D adaptive avg pooling operator.
adaptive_max_pool1d	1D adaptive max pooling operator.
adaptive_max_pool2d	2D adaptive max pooling operator.
adaptive_max_pool3d	3D adaptive max pooling operator.
add	Generic add operator.
adv_index	Numpy style advanced indexing.
affine_grid	affine_grid operator that generates 2D sampling grid.
all	Computes the logical AND of boolean array elements over given axes.
all_class_non_max_suppression	Non-maximum suppression operator for object detection, corresponding to ONNX NonMaxSuppression and TensorFlow combined_non_max_suppression.
any	Computes the logical OR of boolean array elements over given axes.
arange	Return evenly spaced values within a given interval.
argmax	Returns the indices of the maximum values along an axis.
argmin	Returns the indices of the minimum values along an axis.
argsort	Performs sorting along the given axis and returns an array of indices having same shape as an input array that index data in sorted order.
argwhere	Find the indices of elements of a tensor that are non-zero.
asin	Compute elementwise asin of data.

算子	备注
asinh	Take asinh of input x.
atan	Take atan of input x.
atanh	Take atanh of input x.
avg_pool1d	1D average pooling operator.
avg_pool2d	2D average pooling operator.
avg_pool2d_grad	Gradient of 2D average pooling operator.
avg_pool3d	3D average pooling operator.
batch_flatten	BatchFlatten.
batch_matmul	Compute batch matrix multiplication of tensor_a and tensor_b.
batch_norm	Batch normalization layer .
batch_to_space_nd	Reshape the batch dimension into spatial dimensions.
bias_add	add_bias operator.
bind	Bind an free variables in expr or function arguments.
bitpack	Tensor packing for bitserial operations.
bitserial_conv2d	2D convolution using bitserial computation.
bitserial_dense	Bitserial Dense operator.
bitwise_and	bitwise AND with numpy-style broadcasting.
bitwise_not	Compute element-wise bitwise not of data.
bitwise_or	bitwise OR with numpy-style broadcasting.
bitwise_xor	bitwise XOR with numpy-style broadcasting.
broadcast_to	Return a scalar value array with the same type, broadcasted to the provided shape.
broadcast_to_like	Return a scalar value array with the same shape and type as the input array.
cast	Cast input tensor to data type.
cast_like	Cast input tensor to data type of another tensor.
ceil	Compute element-wise ceil of data.
clip	Clip the elements in a between a_min and a_max.
collapse_sum_like	Return a scalar value array with the same shape and type as the input array.

算子	备注
collapse_sum_to	Return a summation of data to the specified shape.
comm_reducer	Create a commutative reducer for reduction.
concatenate	Concatenate the input tensors along the given axis.
const	Create a new constant with specified name and dtype
conv1d	1D convolution.
conv1d_transpose	One dimensional transposed convolution operator.
conv2d	2D convolution.
conv2d_backward_weight	The gradient of conv2d with respect to weight.
conv2d_transpose	Two dimensional transposed convolution operator.
conv3d	3D convolution.
conv3d_transpose	3D transpose convolution.
copy	Copy a tensor.
copy_shape_func	Shape function for copy op.
correlation	Applies correlation to inputs.
cos	Take cos of input x.
cosh	Take cosh of input x.
contrib_conv2d _gemm_weight_transform	Weight Transformation part for 2D convolution with gemm algorithm.
contrib_conv2d_gemm _without_weight_transform	2D convolution with gemm algorithm.
contrib_conv2d_nchwc	Variant of 2D convolution.
contrib_conv2d_winograd _nnpack_weight_transform	Weight Transformation part for 2D convolution with winograd algorithm.
contrib_conv2d_w inograd_weight_transform	Weight Transformation part for 2D convolution with winograd algorithm.
contrib_conv2d_winograd without_weight_transform	2D convolution with winograd algorithm.
contrib_conv3d winograd_weight_transform	Weight Transformation part for 3D convolution with winograd algorithm.
contrib_conv3d_winograd _without_weight_transform	3D convolution with winograd algorithm.
contrib_dense_pack	Dense operator.

算子	备注
contrib_depthwise_conv2d_nchwc	Variant of 2D depthwise convolution.
crop_and_resize	Crop input images and resize them.
cross_entropy	CrossEntropy without logits.
cross_entropy _with_logits	CrossEntropy with logits.
cumprod	Numpy style cumprod op.
cumsum	Numpy style cumsum op.
decl_tensor_intrin	Declare a tensor intrinsic function.
deformable_conv2d	Deformable 2d convolution.
dense	Dense operator.
depth_to_space	Convert channels into spatial blocks.
dilate	Dilate data with given dilation value .
dilation2d	Morphological Dilation 2D.
div	Compute a / b as in C/C++ semantics.
divide	Division with numpy-style broadcasting.
dropout	Applies the dropout operation to the input array.
dropout_raw	Applies the dropout operation to the input array.
einsum	Evaluates the Einstein summation convention on data
equal	Broadcasted elementwise test for (lhs == rhs).
erf	Take gauss error function of the input x.
exp	Take exponential of input x.
expand_dims	Insert num_newaxis axes at the position given by axis.
extern	Compute several tensors via an extern function.
extern_primfunc	Compute tensors via a schedulable TIR PrimFunc
fast_softmax	Computes softmax.
fifo_buffer	FIFO buffer to enable computation reuse in CNNs with sliding indow input
fixed_point_multiply	Fixed point multiplication between data and a fixed point constant expressed as multiplier * 2ˆ(-shift), where multiplier is a Q-number with 31 fractional bits

算子	备注
floor	Take floor of float input x.
floor_divide	Floor division with numpy-style broadcasting.
floor_mod	Floor mod with numpy-style broadcasting.
fmod	Return the remainder of x divided by y with the same sign as x.
full	Fill array with scalar value.
full_like	Return a scalar value array with the same shape and type as the input array.
gather	Gather values along given axis from given indices.
gather_nd	Gather elements or slices from data and store them to a tensor whose shape is defined by indices.
get_pad_tuple1d	Common code to get the 1 dimensional pad option :param padding: Padding size :type padding: Union[int, Tuple[int, ...]]
get_pad_tuple2d	Common code to get the pad option :param padding: Padding size :type padding: Union[int, Tuple[int, ...]]
get_pad_tuple3d	Common code to get the pad option :param padding: Padding size :type padding: Union[int, Tuple[int, ...]]
get_valid_counts	Get valid count of bounding boxes given a score threshold.
global_avg_pool1d	1D global average pooling operator.
global_avg_pool2d	2D global average pooling operator.
global_avg_pool3d	3D global average pooling operator.
global_max_pool1d	1D global maximum pooling operator.
global_max_pool2d	2D global maximum pooling operator.
global_max_pool3d	3D global maximum pooling operator.
gradient	Perform reverse-mode automatic differentiation.
greater	Broadcasted elementwise test for (lhs > rhs).
greater_equal	Broadcasted elementwise test for (lhs >= rhs).
grid_sample	Applies grid sampling to input feature map.
group_norm	Group normalization normalizes over group of channels for each training examples.

算子	备注
if_then_else	Conditional selection expression.
indexdiv	Compute floor(a / b) where a and b are non-negative.
indexmod	Compute the remainder of indexdiv.
instance_norm	Instance Normalization Applies instance normalization to the n-dimensional input array.
invert_permutation	Computes the inverse permutation of data.
isfinite	Check if input value is finite.
isinf	Check if input value is infinite.
isnan	Check if input value is Nan.
l2_normalize	Perform L2 normalization on the input data
layer_norm	Layer normalization .
layout_transform	Transform the layout of a tensor.
leaky_relu	This operator takes data as input and does Leaky version of a Rectified Linear Unit.
left_shift	Left shift with numpy-style broadcasting.
less	Broadcasted elementwise test for (lhs < rhs).
less_equal	Broadcasted elementwise test for (lhs <= rhs).
load_param_dict	Load parameter dictionary to binary bytes.
log	Take log of input x.
log_softmax	Computes log softmax.
log10	Take log10 of input x.
log2	Take log2 of input x.
logical_and	logical AND with numpy-style broadcasting.
logical_not	Compute element-wise logical not of data.
logical_or	logical OR with numpy-style broadcasting.
logical_xor	logical XOR with numpy-style broadcasting.
logsumexp	Compute the log of the sum of exponentials of input elements over given axes.
lrn	This operator takes data as input and does local response normalization.
matmul	Matmul operator.

算子	备注
max	Create a max expression over axis.
max_pool1d	1D maximum pooling operator.
max_pool2d	2D maximum pooling operator.
max _pool2d_grad	Gradient of 2D maximum pooling operator.
max_pool3d	3D maximum pooling operator.
max_value	maximum value of dtype
maximum	Maximum with numpy-style broadcasting.
mean	Computes the mean of array elements over given axes.
mean_std	Computes the mean and standard deviation of data over given axes.
mean_variance	Computes the mean and variance of data over given axes.
meshgrid	Create coordinate matrices from coordinate vectors.
min	Create a min expression over axis.
min_value	minimum value of dtype
mirror_pad	MirrorPadding
mod	Mod with numpy-style broadcasting.
multibox_prior	Generate prior(anchor) boxes from data, sizes and ratios.
multibox_t ransform_loc	Location transformation for multibox detection
multiply	Generic multiply operator.
nearbyint	Round elements of the array to the nearest integer.
negative	Compute element-wise negative of data.
nll_loss	Negative log likelihood loss.
non_max _suppression	Non-maximum suppression operator for object detection.
not_equal	Broadcasted elementwise test for (lhs != rhs).
one_hot	Returns a one-hot tensor where the locations represented by indices take value on_value, and other locations take value off_value.
ones	Fill array with ones.

算子	备注
ones_like	Returns an array of ones, with same type and shape as the input.
pad	Padding
placeholder	Construct an empty tensor object.
popcount	Count the number of set bits in input x.
power	x power y
prelu	This operator takes data as input and does Leaky version of a Rectified Linear Unit.
prod	Computes the products of array elements over given axes.
proposal	Proposal operator.
reduce_axis	Create a new IterVar for reduction.
reinterpret	Reinterpret input tensor to data type.
relu	Rectified linear unit.
repeat	Repeats elements of an array.
reshape	Reshape the input array.
reshape_like	Reshapes the input tensor by the size of another tensor.
resize1d	Image resize1d operator.
resize2d	Image resize2d operator.
resize3d	Image resize3d operator.
reverse	Reverses the order of elements along given axis while preserving array shape.
rev erse_reshape	Reshapes the input array where the special values are inferred from right to left.
reverse_sequence	Reverse the tensor for variable length slices.
right_shift	Right shift with numpy-style broadcasting.
roi_align	ROI align operator.
roi_pool	ROI pool operator.
round	Round elements of the array to the nearest integer.
rsqrt	Take reciprocal of square root of input x.
scan	Construct new tensors by scanning over axis.

算子	备注
scatter	Update data at positions defined by indices with values in updates.
scatter_add	Update data by adding values in updates at positions defined by indices.
scatter_nd	Scatter values from an array and update.
searchsorted	Find indices where elements should be inserted to maintain order.
segment_sum	Computes the sum along segment_ids along axis 0.
sequence_mask	Sets all elements outside the expected length of the sequence to a constant value.
shape_of	Get shape of a tensor.
ShapeVar	A helper which constructs a type var of which the shape kind.
sigmoid	Quick function to get sigmoid
sign	Compute element-wise absolute of data.
sin	Take sin of input x.
sinh	Take sinh of input x.
size_var	Create a new variable represents a tensor shape size, which is non-negative.
slice_like	Slice the first input with respect to the second input.
sliding_window	Slide a window over the data tensor.
softmax	Computes softmax.
sort	Performs sorting along the given axis and returns data in sorted order.
space_to_batch_nd	Divide spatial dimensions of the data into a grid of blocks and interleave them into batch dim.
space_to_depth	Convert spatial blocks into channels.
sparse_add	Computes the matrix addition of dense_mat and sparse_mat, where dense_mat is a dense matrix and sparse_mat is a sparse namedtuple with fields data, indices, and indptr.
sparse_dense	Computes the matrix multiplication of dense_mat and sparse_mat, where dense_mat is a dense matrix and sparse_mat is a sparse namedtuple with fields data, indices, and indptr.

算子	备注
sparse_fill_empty_rows	Fill rows in a sparse matrix that do not contain any values.
sparse_reshape	Reshape a sparse tensor.
sparse_to_dense	Converts a sparse representation into a dense tensor.
sparse_transpose	Computes the fast matrix transpose of x, where x is a sparse tensor in CSR format .
split	Split input tensor along axis by sections or indices.
sqrt	Take square root of input x.
squeeze	Squeeze axes in the array.
stack	Join a sequence of arrays along a new axis.
std	Computes the standard deviation of data over given axes.
stft	The STFT computes the Fourier transform of short overlapping windows of the input.
strided_set	Strided set of an array.
strided_slice	Strided slice of an array.
subtract	Generic subtract operator.
sum	Create a sum expression over axis.
tag_scope	The operator tag scope.
take	Take elements from an array along an axis.
tan	Take tan of input x.
tanh	Take hyperbolic tanh of input x.
tile	Repeats the whole array multiple times.
topk	Get the top k elements in an input tensor along the given axis.
trace	Trace tensor data at the runtime.
transpose	Permutes the dimensions of an array.
trilu	Given a 2-D matrix or batches of 2-D matrices, returns the upper or lower triangular part of the tensor.
trunc	Get truncated value of the input.
trunc_divide	Trunc division with numpy-style broadcasting.

算子	备注
trunc_mod	Trunc mod with numpy-style broadcasting.
truncdiv	Compute the truncdiv of two expressions.
truncmod	Compute the truncmod of two expressions.
unique	Find the unique elements of a 1-D tensor.
unravel_index	Convert a flat index or array of flat indices into a tuple of coordinate arrays.
upsampling	Upsampling.
upsampling3d	3D Upsampling.
var	Create a new tvm.relay.Var.
where	Selecting elements from either x or y depending on the value of the condition.
yolo_reorg	Yolo reorg operation used in darknet models.
zeros	Fill array with zeros.
zeros_like	Returns an array of zeros, with same type and shape as the input.

11.2 框架模型算子支持

IGIE 支持主流框架模型直接导入。

IGIE 支持的主流框架模型包括：

框架模型	使用接口	说明
ONNX	relay.frontend.from_onnx()	导入 ONNX 模型，转换为 IGIE relay Op
PyTorch	relay.frontend.from_pytorch()	导入 PyTorch 模型
TensorFlow	relay.frontend.from_tensorflow()	导入 TensorFlow 模型
PaddlePaddle	relay.frontend.from_paddle()	导入 Paddle 模型
TFlite	relay.frontend.from_tflite()	导入 TFlite 模型
MXNet	relay.frontend.from_mxnet()	导入 MXNet 模型
Caffe	relay.frontend.from_caffe()	导入 Caffe 模型
Keras	relay.frontend.from_keras()	导入 Keras 模型
Darknet	relay.frontend.from_darknet()	导入 Darknet 模型

Tip

relay 是 IGIE grpah 和 op 的集合，所有算子的定义都在 relay package 下面，因此 igie 的算子有时候也会称为 relay op。

11.2.1 ONNX 模型算子的支持

```
Identity
Affine
BitShift
ThresholdedRelu
ScaledTanh
ParametricSoftplus
Constant
ConstantOfShape
GivenTensorFill
FC
Scale
GRUUnit
ATen
ImageScaler
MeanVarianceNormalization
Crop
Embedding
Upsample
SpatialBN
defs/generator
Constant Implemented
RandomUniform
RandomNormal
RandomUniformLike
RandomNormalLike
defs/logical
defs/math
Add
Sub
Mul
Div
Neg
Abs
Reciprocal
Floor
Ceil
Round
IsInf
```

```
IsNaN
Sqrt
Relu
Celu
LeakyRelu
Selu
Elu
Exp
Greater
GreaterOrEqual
Less
LessOrEqual
Log
Acos
Acosh
Asin
Asinh
Atan
Atanh
Cos
Cosh
Sin
Sinh
Tan
Tanh
Pow
PRelu
Sigmoid
HardSigmoid
Max
Min
Sum
Mean
Clip
Softplus
Softmax
LogSoftmax
OneHot
Hardmax
Shrink
Softsign
Gemm
MatMul
MatMulInteger16
Mod
Xor
```

```
defs/nn
AveragePool
LpPool
GlobalLpPool
MaxPool
MaxUnpool
Conv
ConvTranspose
GlobalAveragePool
GlobalMaxPool
BatchNormalization
InstanceNormalization
LpNormalization
Dropout
Flatten
LRN
Recurrent Layers
LSTM
GRU
defs/vision
MaxRoiPool
RoiAlign
NonMaxSuppression
defs/reduction
ReduceMax
ReduceMin
ReduceSum
ReduceMean
ReduceProd
ReduceLogSumExp
ReduceLogSum
ReduceSumSquare
ReduceL1
ReduceL2
defs/sorting
ArgMax
ArgMin
TopK
defs/tensor
Cast
Reshape
Expand
Concat
Split
Slice
Transpose
```

```
DepthToSpace
SpaceToDepth
Gather
GatherElements
GatherND
Compress
Size
Scatter
ScatterElements
ScatterND
EyeLike
Squeeze
Unsqueeze
Pad
Shape
Sign
Equal
Not
And
Tile
Erf
Where
Or
Resize
NonZero
Range
CumSum
Unique
Einsum
defs/control_flow
Loop
If
Torch ATen Dispatcher.
RandomNormal
RandomNormalLike
RandomUniform
RandomUniformLike
NegativeLogLikelihoodLoss
SoftmaxCrossEntropyLoss
Adagrad
Adam
Momentum
NegativeLogLikelihoodLoss
SoftmaxCrossEntropyLoss
Adagrad
Adam
```

```
Momentum
Quantization
QuantizeLinear
DequantizeLinear
DynamicQuantizeLinear
ReverseSequence
QLinearConv
QLinearConcat
QLinearAdd
QLinearMatMul
QLinearMul
QLinearSigmoid
ConvInteger
QLinearAveragePool
QLinearGlobalAveragePool
QLinearLeakyRelu
MatMulInteger
SequenceEmpty
SequenceConstruct
SequenceInsert
SequenceLength
ConcatFromSequence
```

暂不支持以下算子：

- QLinearSoftmax
- QLinearGEMM

11.2.2 Pytorch 模型算子支持

```
aten::pixel_shuffle
aten::device
prim::device
aten::sub
aten::sub_
aten::max
aten::min
aten::mul
aten::mul_
aten::pow
aten::arange
aten::meshgrid
aten::div
aten::div_
```

```
aten::floor_divide
aten::floor_divide_
aten::true_divide
aten::addcdiv
aten::addcmul
aten::ones
aten::ones_like
aten::zeros
aten::zeros_like
aten::full
aten::full_like
aten::linspace
aten::reciprocal
aten::repeat
aten::repeat_interleave
aten::to
aten::squeeze
aten::unsqueeze
aten::unsqueeze_
aten::cat
aten::slice
aten::narrow
aten::split
aten::split_with_sizes
aten::select
aten::take
aten::where
aten::topk
aten::relu
aten::relu_
aten::prelu
aten::leaky_relu
aten::leaky_relu_
aten::elu
aten::elu_
aten::celu
aten::gelu
aten::selu
aten::silu
aten::silu_
aten::log_sigmoid
aten::adaptive_avg_pool1d
aten::adaptive_avg_pool2d
aten::adaptive_avg_pool3d
aten::adaptive_max_pool1d
aten::adaptive_max_pool2d
```

```
aten::adaptive_max_pool3d
aten::max_pool2d
aten::max_pool2d_with_indices
aten::max_pool1d
aten::max_pool3d
aten::hardtanh
aten::hardtanh_
aten::_convolution
aten::softmax
aten::threshold
aten::threshold_
aten::contiguous
aten::batch_norm
aten::instance_norm
aten::layer_norm
aten::group_norm
aten::transpose
aten::transpose_
aten::t
aten::flatten
aten::addmm
aten::size
aten::view
aten::reshape
aten::clone
aten::log_softmax
aten::sigmoid
aten::sigmoid_
aten::softplus
aten::avg_pool1d
aten::avg_pool2d
aten::avg_pool3d
aten::linear
aten::dropout
aten::dropout_
aten::feature_dropout
aten::alpha_dropout
aten::mean
aten::chunk
aten::unsafe_chunk
aten::matmul
aten::bmm
aten::expand
aten::Int
prim::NumToTensor
prim::ImplicitTensorToNum
```

```
aten::ScalarImplicit
aten::constant_pad_nd
aten::reflection_pad1d
aten::reflection_pad2d
aten::replication_pad1d
aten::replication_pad2d
aten::replication_pad3d
aten::permute
aten::sum
aten::prod
aten::argmin
aten::argmax
aten::norm
aten::frobenius_norm
aten::std
aten::var
aten::abs
aten::neg
aten::cos
aten::cosh
aten::sin
aten::sinh
aten::tan
aten::tanh
aten::tanh_
aten::acos
aten::asin
aten::atan
aten::log
aten::log2
aten::log10
aten::log1p
aten::exp
aten::erf
aten::trunc
aten::sign
aten::sqrt
aten::rsqrt
aten::ceil
aten::floor
aten::floor_
aten::round
aten::isfinite
aten::isinf
aten::isnan
aten::clamp
```

```
aten::clamp_
aten::detach
aten::upsample_bilinear2d
aten::upsample_bicubic2d
aten::upsample_nearest2d
aten::upsample_trilinear3d
aten::upsample_nearest3d
aten::expand_as
aten::lt
aten::gt
aten::le
aten::ge
aten::ne
aten::eq
aten::logical_not
aten::logical_xor
aten::bitwise_not
aten::bitwise_xor
aten::Bool
aten::Float
aten::rsub
aten::embedding
aten::one_hot
aten::mm
aten::add
aten::add_
aten::stack
aten::getitem
aten::len
aten::type_as
aten::gather
aten::index_select
aten::index
torchvision::nms
aten::logsumexp
torchvision::roi_align
torchvision::deform_conv2d
aten::unbind
aten::and
aten::logical_and
aten::_shape_as_tensor
aten::nonzero
aten::nonzero_numpy
aten::scatter
aten::index_put
aten::index_put_
```

```
aten::scalar_tensor
aten::__interpolate
aten::IntImplicit
aten::tensor
aten::numel
aten::empty
aten::bincount
aten::scatter_add
aten::not
aten::hardswish_
aten::hardswish
aten::hardsigmoid_
aten::hardsigmoid
aten::cumsum
aten::masked_fill
aten::masked_fill_
aten::masked_select
aten::argsort
aten::sort
aten::_unique2
aten::nll_loss
aten::nll_loss2d
aten::nll_loss_nd
aten::flip
aten::gru
aten::lstm
aten::all
aten::any
aten::searchsorted
aten::bucketize
aten::roll
aten::einsum
prim::Constant
prim::GetAttr
prim::ListConstruct
prim::ListUnpack
prim::TupleConstruct
prim::TupleUnpack
prim::RaiseException
prim::If
prim::Loop
aten::quantize_per_tensor
quantized::conv2d_relu
aten::dequantize
quantized::conv2d
quantized::add_relu
```

```
quantized::add
quantized::mul_relu
quantized::mul
quantized::linear
quantized::linear_relu
quantized::cat
quantized::add_scalar
quantized::mul_scalar
quantized::relu6
quantized::linear_dynamic
quantized::hardswish
quantized::conv_transpose2d
```

11.2.3 TensorFlow 模型算子支持

```
aten::pixel_shuffle
Abs
Acos
Acosh
Add
AddN
AddV2
All
Any
ArgMax
ArgMin
Asin
Asinh
Assert
Atan
Atanh
Atan2
AvgPool
AvgPool3D
BatchMatMul
BatchMatMulV2
BatchNormWithGlobalNormalization
BatchToSpaceND
BiasAdd
BroadcastTo
BroadcastArgs
Cast
Ceil
CheckNumerics
```

```
ClipByValue
Concat
ConcatV2
Conv2D
Conv2DBackpropInput
Conv3D
Conv3DBackpropInputV2
Cos
Cosh
CropAndResize
DecodeJpeg
DepthToSpace
DepthwiseConv2dNative
Dilation2D
Elu
Equal
Erf
EuclideanNorm
Exp
ExpandDims
Expm1
Fill
Floor
FloorDiv
FloorMod
FusedBatchNorm
FusedBatchNormV2
FusedBatchNormV3
Gather
GatherNd
GatherV2
Greater
GreaterEqual
Identity
IdentityN
InvertPermutation
IsFinite
IsInf
IsNan
LeakyRelu
LeftShift
Less
LessEqual
Log
Log1p
LogicalAnd
```

```
LogicalNot
LogicalOr
LogSoftmax
LRN
LSTMBlockCell
MatMul
Max
Maximum
MaxPool
MaxPool3D
Mean
Min
Minimum
MirrorPad
Mod
Mul
Neg
NonMaxSuppressionV2
NonMaxSuppressionV3
NonMaxSuppressionV4
NonMaxSuppressionV5
CombinedNonMaxSuppression
NoOp
NotEqual
OneHot
Pack
Pad
PadV2
Pow
Prod
Range
Rank
RealDiv
Relu
Relu6
Reshape
ResizeBicubic
ResizeBilinear
ResizeNearestNeighbor
ReverseV2
RightShift
Rint
Round
Rsqrt
Select
SelectV2
```

```
Selu
Shape
Sigmoid
Sign
Sin
Sinh
Size
Slice
Softmax
Softplus
Softsign
SpaceToBatchND
SpaceToDepth
SparseToDense
SparseTensorDenseMatMul
SparseFillEmptyRows
SparseReshape
SegmentSum
SparseSegmentSum
SparseSegmentSumWithNumSegments
SparseSegmentSqrtN
SparseSegmentSqrtNWithNumSegments
SparseSegmentMean
SparseSegmentMeanWithNumSegments
SparseTensorDenseAdd
Split
SplitV
Sqrt
Square
SquaredDifference
Squeeze
StopGradient
StridedSlice
Sub
Sum
Tan
Tanh
TensorArrayConcatV3
TensorArrayGatherV3
TensorArrayReadV3
TensorArrayScatterV3
TensorArraySizeV3
TensorArraySplitV3
TensorArrayV3
TensorArrayWriteV3
Tile
```

```
TopKV2
Transpose
TruncateMod
Unique
UniqueWithCounts
Unpack
UnravelIndex
Where
ZerosLike
TensorListFromTensor
TensorListGetItem
TensorListReserve
TensorListSetItem
TensorListStack
```

11.2.4 Paddle 模型算子支持

```
abs
acos
addmm
arg_max
arg_min
argsort
asin
assign
assign_value
atan
batch_norm
bicubic_interp_v2
bilinear_interp_v2
bmm
brelu
cast
ceil
concat
conv2d
conv2d_transpose
cos
cosh
cumsum
depthwise_conv2d
dot
dropout
elementwise_add
```

```
elementwise_div
elementwise_floordiv
elementwise_max
elementwise_min
elementwise_mod
elementwise_mul
elementwise_pow
elementwise_prod
elementwise_sub
equal
erf
exp
expand_v2
expand_as_v2
feed
fill_any_like
fill_constant
fill_constant_batch_size_like
flatten_contiguous_range
floor
floor_mod
gather
gather_nd
gelu
greater_equal
greater_than
group_norm
hard_shrink
hard_sigmoid
hard_swish
isfinite_v2
isinf_v2
isnan_v2
layer_norm
leaky_relu
less_equal
less_than
log
log2
log10
logical_and
logical_not
logical_or
logical_xor
lookup_table_v2
matmul
```

```
matmul_v2
mul
nearest_interp_v2
not_equal
pad1d
pad2d
pad3d
pool2d
pow
relu
relu6
reshape2
round
reciprocal
reduce_all
reduce_any
reduce_max
reduce_min
reduce_prod
reduce_sum
reduce_mean
rnn
rsqrt
scale
scatter
scatter_nd_add
selu
shape
sigmoid
sign
sin
sinh
size
slice
softmax
softplus
softsign
sqrt
square
squeeze2
swish
tan
tanh
transpose2
unsqueeze2
softplus=convert_softplus,
```

```
softsign=convert_softsign,
sqrt=convert_unary_op,
square=convert_square,
squeeze2=convert_squeeze,
swish=convert_swish,
tan=convert_unary_op,
tanh=convert_unary_op,
transpose2=convert_transpose,
unsqueeze2=convert_unsqueeze,
```

11.2.5 MXNet 模型算子支持

```
abs
log
exp
erf
sqrt
floor
ceil
round
trunc
sign
sigmoid
negative
reshape_like
zeros_like
ones_like
cos
cosh
sin
sinh
tan
tanh
where
_copy
relu
broadcast_add
broadcast_plus
broadcast_sub
broadcast_minus
broadcast_mul
broadcast_div
broadcast_mod
broadcast_maximum
```

```
broadcast_minimum
broadcast_power
arccos
arcsin
arctan
arccosh
arcsinh
arctanh
broadcast_equal
broadcast_not_equal
broadcast_greater
broadcast_greater_equal
broadcast_lesser
broadcast_lesser_equal
broadcast_logical_or
broadcast_logical_and
broadcast_logical_xor
broadcast_to
broadcast_like
logical_not
_equal
_not_equal
_greater
_greater_equal
_lesser
_lesser_equal
elemwise_add
elemwise_sub
elemwise_mul
elemwise_div
_maximum
_minimum
flatten
Flatten
square
rsqrt
cbrt
rcbrt
pow_scalar
_power_scalar
rsub_scalar
_rminus_scalar
rdiv_scalar
_rdiv_scalar
rpow_scalar
add_scalar
```

```
_plus_scalar
sub_scalar
_minus_scalar
mul_scalar
_mul_scalar
div_scalar
_div_scalar
log2
log10
log1p
expm1
_equal_scalar
_not_equal_scalar
_greater_scalar
_greater_equal_scalar
_lesser_scalar
_lesser_equal_scalar
_maximum_scalar
_minimum_scalar
mean
max
min
sum
max_axis
min_axis
sum_axis
argmax
argmin
_ones
softmax
log_softmax
Softmax
softsign
softmin
hard_sigmoid
reciprocal
Reshape
reshape
Cast
amp_cast
amp_multicast
clip
transpose
UpSampling
add_n
_zeros
```

```
FullyConnected
Activation
Convolution
Convolution_v1
Deconvolution
Pooling
Pooling_v1
Dropout
BatchNorm
BatchNorm_v1
_contrib_SyncBatchNorm
InstanceNorm
LayerNorm
GroupNorm
LRN
L2Normalization
slice
slice_like
slice_axis
SliceChannel
split
_split_v2
SwapAxis
expand_dims
Concat
concat
stack
dot
batch_dot
LeakyReLU
_arange
_full
repeat
tile
pad
Pad
take
gather_nd
reverse
SequenceReverse
squeeze
broadcast_axis
broadcast_axes
BlockGrad
shape_array
Embedding
```

```
argsort
topk
_unravel_index
SequenceMask
SoftmaxOutput
SoftmaxActivation
LinearRegressionOutput
LogisticRegressionOutput
smooth_l1
make_loss
_contrib_div_sqrt_dim
_contrib_arange_like
one_hot
depth_to_space
space_to_depth
Correlation
_contrib_BilinearResize2D
_contrib_MultiBoxPrior
_contrib_MultiBoxDetection
_contrib_ROIAlign
ROIPooling
_contrib_Proposal
_contrib_MultiProposal
_contrib_box_nms
_contrib_box_decode
_contrib_DeformableConvolution
_contrib_AdaptiveAvgPooling2D
GridGenerator
BilinearSampler
RNN
_rnn_param_concat
_contrib_interleaved_matmul_selfatt_qk
_contrib_interleaved_matmul_selfatt_valatt
_cond
Crop
ring_buffer
_contrib_quantize_v2
_contrib_quantized_concat
_contrib_quantized_ring_buffer
_sg_mkldnn_conv
_contrib_quantized_flatten
_contrib_dequantize
_contrib_quantized_act
_contrib_quantized_pooling
_contrib_quantized_batch_norm
_sg_mkldnn_fully_connected
```

```
_np_transpose
_npi_transpose
_npi_pad
_npi_concatenate
_npx_reshape
_np_copy
_npi_copy
_npi_power
_npi_power_scalar
_npi_multiply
_npi_multiply_scalar
_npi_add
_npi_add_scalar
_npi_subtract
_npi_subtract_scalar
_npi_where_rscalar
_npi_less
_npi_less_equal
_npi_tanh
_npi_true_divide_scalar
_npi_stack
```

12 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。