NPU 模型部署
开发指南

版本号: 2.5
发布日期: 2025.04.03

版本历史

版本号	日期	制/修订人	内容描述
1.0	2022.08.23	AWA1890	建立初始版本
1.1	2022.09.27	AWA0385	补充细节
1.2	2023.04.12	AWA1382	1) 第 1 章更新适用范围； 2) 新增第 2 章模型部署一般流程图。
1.3	2024.02.21	AWA1890	1) 新增第 3 章 NPU 进阶使用说明，包含多线程调用和共享权重； 2) 新增第 4 章量化精度调优。
1.4	2024.02.28	AWA0385	1) 第 3 章新增零拷贝调用、自定义算子； 2) 新增第 5 章性能优化。
2.0	2024.03.07	AWA1382	1) 新增第 2 章 NPU 开发系统介绍； 2) 新增第 3 章开发环境准备与检查； 3) 第 5 章新增三个工具使用说明和驱动说明； 4) 新增第 8 章常见问题。
2.1	2024.04.11	AWA1382	1) 增加 MR536、T536 平台说明； 2) 常见问题新增内容
2.2	2024.05.17	AWA0385	1) 第 2 章补充 v2.0.x 后，中间件的变化； 2) 第 4 章补充部分 inputmeta 及 postprcess 的说明； 3) 第 7 章补充调试库的使用说明； 4) 第 8 章补充常见问题。
2.3	2024.08.30	AWA2273	1) 第 4 章补充其它类型模型导入说明； 2) 第 7 章补充读取 NPU 频率说明； 3) 第 8 章补充常见问题。
2.4	2024.11.13	AWA1382	1) 增加 A733 平台说明； 2) 更新 T527 平台说明； 3)SDK 产品包介绍迁移到《快速入门》文档中。
2.5	2025.04.03	AWA1382	**全部章节** 增加 T736 平台说明。 **第三章 开发环境准备与检查** 更新 env.sh 使用。 **第四章 NPU 使用说明** 第 4.3.1 节补充说明命令参数。 第 4.7.2 节更新 Linux 编译命令。

目　录

插　图

1 前言

1.1 文档简介

本文档对 NPU 系统进行介绍，以及对 NPU 模型部署流程、工具使用、调试方法进行详细说明。

请按照《NPU_开发环境部署_参考指南》文档预先进行开发环境安装部署。

1.2 目标读者

本文档（本指南）主要适用于以下人员：

- 技术支持工程师
- AI 软件开发工程师
- AI 算法开发工程师

1.3 适用范围

硬件平台：V85x、R853、MR527、T527、AI985、MR536、T536、A733、T736

软件平台：Tina 系统、Android13 及以上系统

1.4 文档约定

1.4.1 名词解释

术语	解释说明
NPU	Neural Network Processing Unit，神经网络处理器。
RTOS	Real Time Operating System，实时操作系统。
PPU	Parallel Processing Unit，并行处理单元。
EVIS	Enhanced Vision Instruction Set，增强视觉指令集。

术语	解释说明
NBG	Network Binary Graph，驱动可识别的网络二进制文件。
pcq	perchannel quantize，逐通道量化。
QAT	quantization aware training，量化感知训练。

1.4.2 标志说明

本文中可能出现的符号如下：

⚠ 注意

- **提醒操作中应注意的事项。不当的操作可能会损坏器件，影响可靠性、降低性能等。**

说明

为准确理解文中指令、正确实施操作而提供的补充或强调信息。

技巧

一些容易忽视的小功能、技巧。了解这些功能或技巧能帮助解决特定问题或者节省操作时间。

2 NPU 开发系统介绍

平台内置的 NPU 具备一定的算力和一定的内部高速缓存，实现高效低功耗的 AI 运算。此 NPU 支持以下特性：

- 支持 INT8、UINT8、INT16 三种量化精度。
- 可支持导入 Onnx、TensorFlow、TFLite、Caffe、Keras、Darknet、pyTorch 等常用的深度学习模型格式。
- 提供 AI 开发工具：支持模型快速转换、优化网络、量化、仿真验证。
- 提供端侧 AI 应用开发接口：提供 NPU 跨平台 API 调用，支持 RTOS，Linux 和 Android 系统。

2.1 NPU 系统架构介绍

2.1.1 系统架构说明

NPU 的系统架构如下图所示。

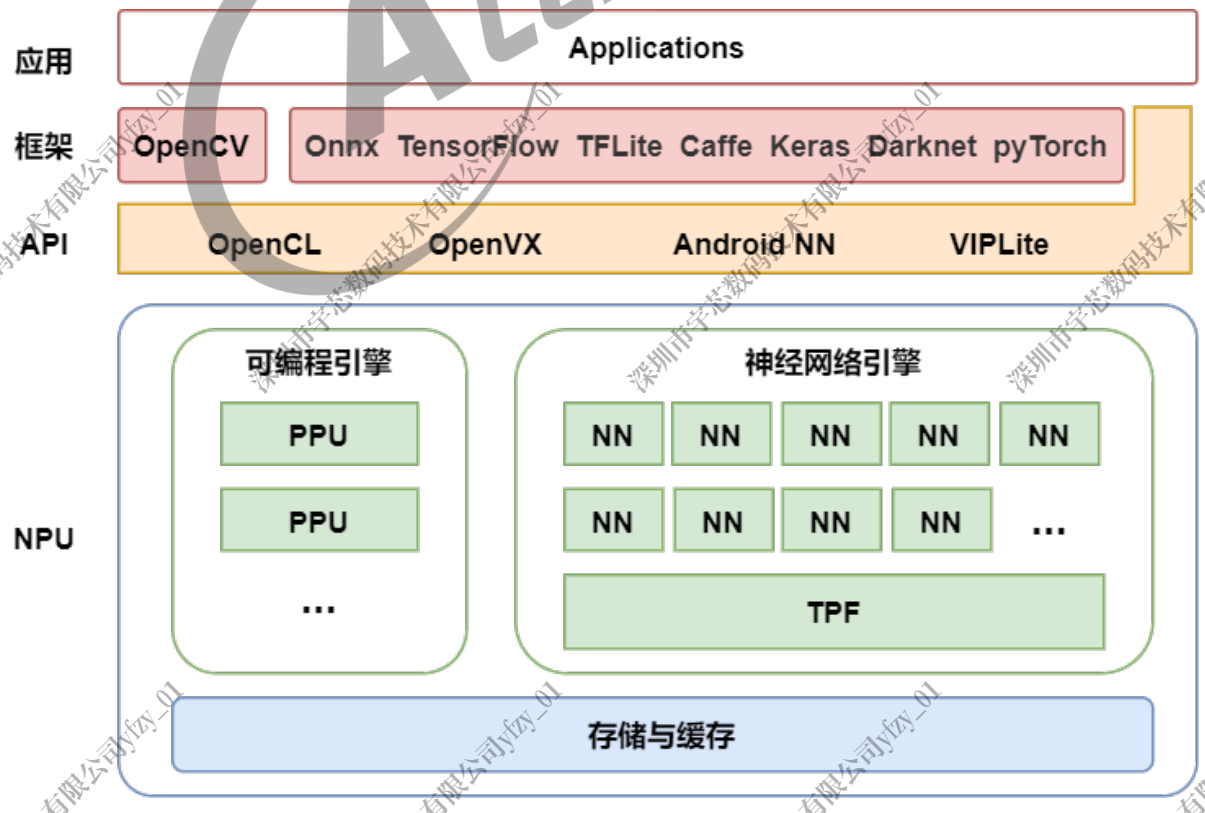

图 2-1: NPU 的系统架构图

上层的应用程序可以通过加载模型与数据到 NPU 进行计算，也可以使用 NPU 提供的软件 API 操作 NPU 执行计算。其中 VIPLite API 是此平台特定的端侧轻量级网络级调用接口。

NPU 包括三个部分：神经网络引擎（Neural Network Engine）、可编程引擎（Programmable Engines）和缓存。其中神经网络引擎包含 NN 核与张量处理器（Tensor Process Fabric，TPF）两个部分，NN 核一般进行卷积操作。而可编程引擎包含并行处理单元（Parallel Processing Unit，PPU），可以使用 EVIS 硬件加速指令与 Shader 语言进行编程，也可以实现激活函数等操作。算子是由可编程引擎与神经网络引擎共同实现的。

2.1.2 Linux 软件架构说明

运行在端侧 Linux 平台，NPU 的轻量级软件架构如下图所示，分为用户态和内核态。公开网络或私有网络和少量训练样本通过 AI 开发工具转换为 NBG（Network Binary Graph，网络二进制图）文件，AI 应用把 NBG 文件和数据通过中间库的 API（Runtime Library）送到内核驱动，最终运行在 NPU 硬件上。

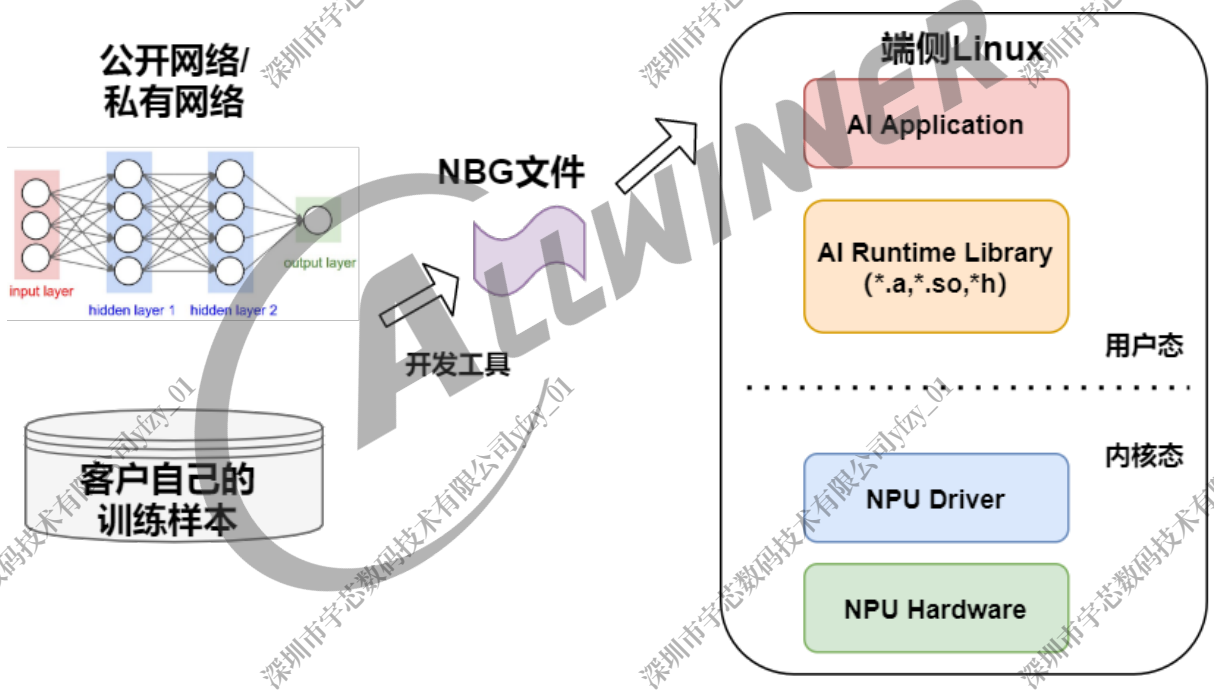

图 2-2: NPU 端侧软件架构图

2.2 NPU 开发流程介绍

NPU 开发完整的流程如下图所示。

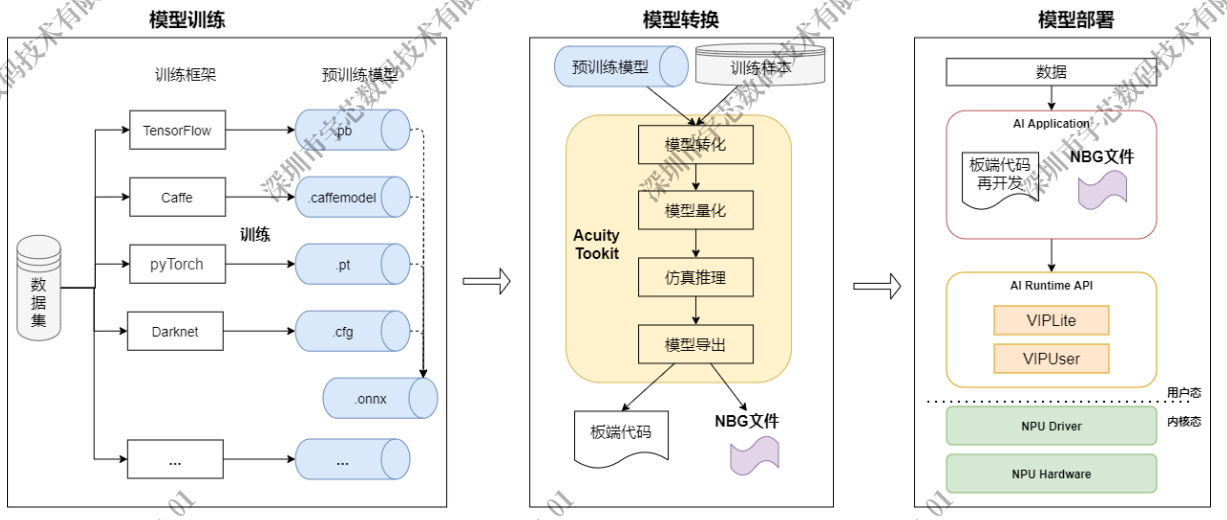

图 2-3: NPU 的开发流程示意图

2.2.1 模型训练

在模型训练阶段，用户既可以根据需求和实际情况先选择合适的框架（如 Caffe、TensorFlow 等）再使用数据集进行训练得到符合需求的模型。也可直接使用已经训练好的模型。本 NPU 支持包括分类、检测、跟踪、人脸、姿态估计、分割、深度、语音、像素处理、OCR 等各个场景 90 多个公开模型。

2.2.2 模型转换

在模型转换阶段，通过 Acuity Toolkit 把预训练模型和少量训练数据转换为 NPU 可用的模型 NBG 文件。

一般步骤如下：

1. 模型导入，生成网络结构文件、网络权重文件、输入描述文件和输出描述文件。
2. 模型量化，生成量化描述文件和熵值文件，可改用不同的量化方式。
3. 仿真推理，可逐一对比 float 和其他量化精度的仿真结果的相似度，评估量化后的精度是否满足要求。
4. 模型导出，生成端侧代码和 *.nb 文件，可编辑输出描述文件的配置，配置是否添加后处理节点等。

技巧

1. 开发工具的模型量化支持 uint8、pcq int8、int16 的量化方式，以及混合量化（如 uint8+int16 等）等多种量化方式。
2. 开发工具支持后量化，也支持前量化模型直接导入，如 TFLite、Onnx 的 QAT 模型。
3. 最后转换生成的 *.nb 文件可使用 nbinfo 工具进行查看和分析。

2.2.3 模型部署及应用开发

在模型部署阶段，就是基于 VIPLite API 开发应用程序实现业务逻辑。

可先使用 vpm_run 模型运行工具，在端侧快速验证 NBG 文件的正确性和运行结果，再利用 VIPLite API 进行开发。

说明

1. **端侧应用示例请参见ai-sdk/examples目录。ai-sdk 获取方式见《NPU_ 快速入门 _ 开发指南》的《SDK 产品包介绍》一节。**
2. **vpm_run 源码请参见ai-sdk/examples/vpm_run目录。**
3. **常用公开模型的端侧运行性能请参见《NPU_ 常见网络性能 _ 测试报告》。**

利用 Camera 输出的数据进行人形检测开发的应用案例如图所示。Camera 输出 VIPP3 通路数据导入到 NPU 应用程序中，NPU 运行结果输出给编码、预览通路进行逻辑处理。

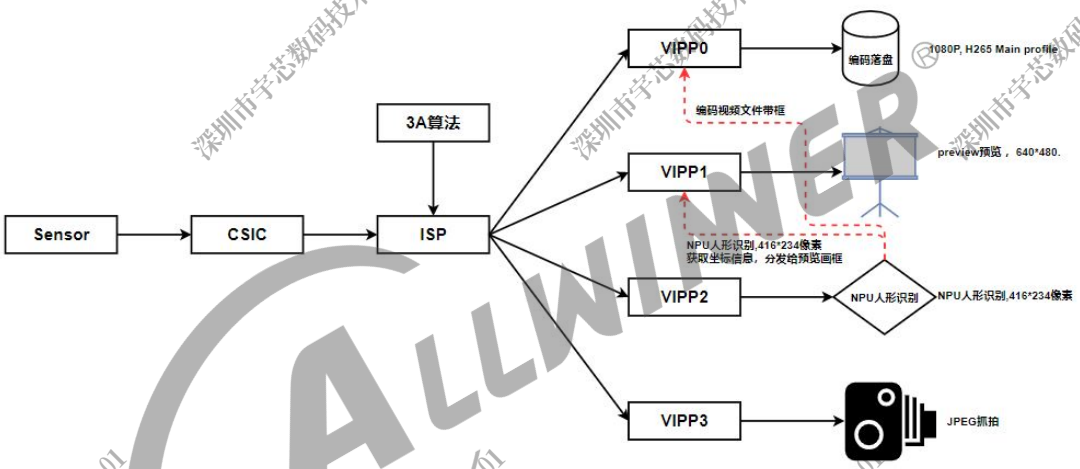

图 2-4: 人形检测的应用案例示意图

3 开发环境准备与检查

根据《NPU_开发环境部署_参考指南》，部署 PC 侧的 ubuntu 环境，使用 Docker 镜像环境或自行安装两个工具。ai-sdk 获取方式见《NPU_ 快速入门 _ 开发指南》的《SDK 产品包介绍》一节。

首先，使用pegasus --help命令检查 Acuity Toolkit 是否可用，打印关键如下：

```
Pegasus commands.

positional arguments:
 {import,export,generate,prune,inference,quantize,train,dump,measure,help}
   import          Import models.
   export          Export models.
   generate        Generate metas.
   prune           prune models.
   inference       Inference model and get result.
   quantize        Quantize model.
   train           Train model.
   dump            Dump model activations.
   measure         Get amount of calculation, parameter and activation.
   help            Print a synopsis and a list of commands.

optional arguments:
 -h, --help        show this help message and exit
```

然后，使用echo命令检查 ACUITY_PATH 环境变量是否配置，方法如下：

```
AwExdroid-AI:~/ai-sdk$ echo ${ACUITY_PATH}
/root/Verisilicon_Tool_Acuity_Toolkit/acuity-toolkit-binary-x.x.x/bin/
```

以及，使用echo命令检查 VIV_SDK 环境变量是否配置，方法如下：

```
AwExdroid-AI:~/ai-sdk$echo ${VIV_SDK}
/root/Vivante_IDE/VivanteIDEx.x.x/cmdtools
```

其次，部署过程可使用快捷脚本，请从ai-sdk/scripts目录中获取。

```
AwExdroid-AI:~/ai-sdk/scripts$ tree
├── awnet_normalize.py
├── compute_similarity.sh
├── pegasus_dump.sh
├── pegasus_export_ovx_nbg.sh
├── pegasus_export_ovx.sh
├── pegasus_import.sh
├── pegasus_inference.sh
├── pegasus_measure.sh
├── pegasus_quantize_hybird.sh
├── pegasus_quantize.sh
├── pegasus_setup.sh
└── README.md
```

建议把原始模型目录放置在ai-sdk/models下，在 models 目录下执行source env.sh v2和拷贝全部脚本，就可按照下文说明开始处理模型。而且保证模型文件的名字和模型目录名一致。

```
AwExdroid-AI:~/ai-sdk/models$ source env.sh v2
AwExdroid-AI:~/ai-sdk/models$ cp ../scripts/* .
AwExdroid-AI:~/ai-sdk/models$ ls -l
-rwxr--r-- 1 1010 1012 1143 Apr 18 08:24 awnet_normalize.py
-rwxr--r-- 1 1010 1012  502 Apr 18 08:24 compute_similarity.sh
-rwxr--r-- 1 1010 1012 1423 Sep 13 02:20 env.sh
drwxr-xr-x 3 1010 1012 4096 Nov 11 17:21 lenet
-rwxr--r-- 1 root root 2698 Nov 11 07:08 pegasus_dump.sh
-rwxr--r-- 1 root root 3086 Nov 11 07:08 pegasus_export_ovx.sh
-rwxr--r-- 1 root root 2770 Nov 11 07:08 pegasus_export_ovx_nbg.sh
-rwxr--r-- 1 root root 6170 Nov 11 07:08 pegasus_import.sh
-rwxr--r-- 1 root root 2847 Nov 11 07:08 pegasus_inference.sh
-rwxr--r-- 1 root root  546 Nov 11 07:08 pegasus_measure.sh
-rwxr--r-- 1 root root 2197 Nov 11 07:08 pegasus_quantize.sh
-rwxr--r-- 1 root root 2030 Nov 11 07:08 pegasus_quantize_hybird.sh
-rwxr--r-- 1 root root  413 Nov 11 07:08 pegasus_setup.sh
drwxr-xr-x 5 1010 1012 4096 Nov 11 07:13 yolov5s-sim
... ...
```

4 NPU 使用说明

4.1 模型转换部署一般流程

所以本平台的模型转换部署一般流程如下图所示。

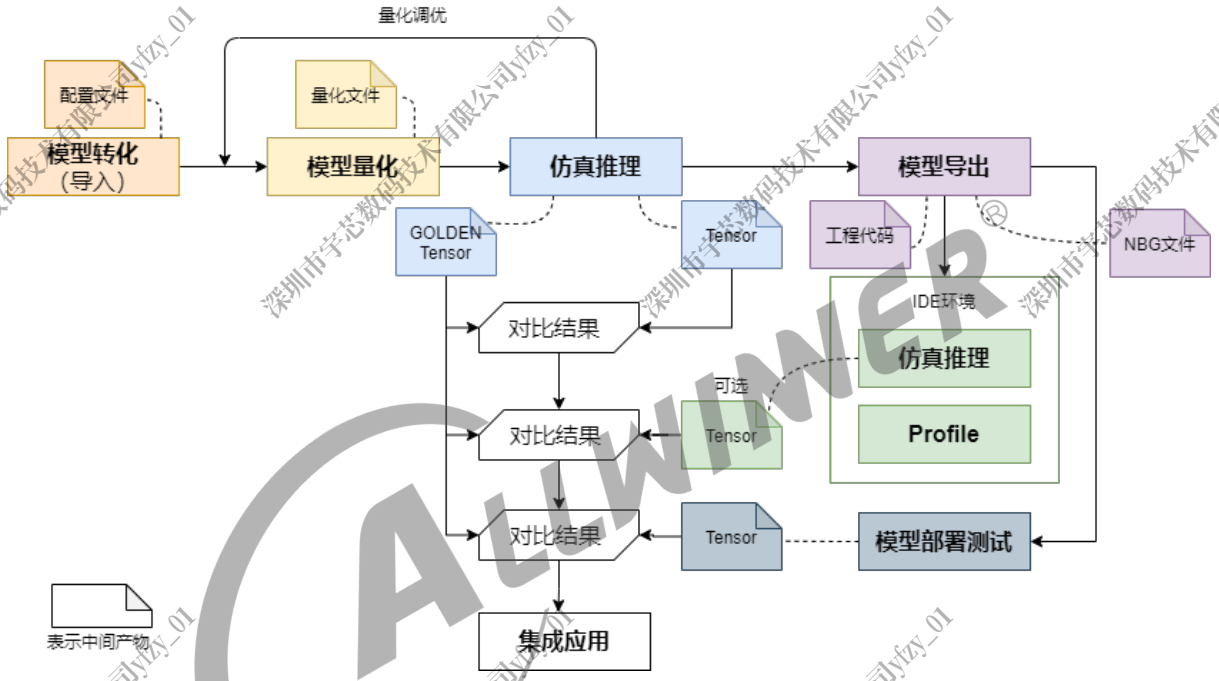

图 4-1: NPU 模型转换部署一般流程

1、模型转换

因为 NPU 运行所使用的模型是 NPU 自定义的结构，不能直接将网络训练出的模型送入 NPU 进行计算。首先把准备好模型使用 Acuity Toolkit 工具导入，转化为工具所识别的网络结构文件、权重文件和配置文件。配置文件用于对网络的输入和输出的参数进行描述，包括输入/输出 tensor 的 shape、预处理参数 (均值、scale)、tensor 的输出格式等。

2、模型量化

由于 NPU 硬件特性，需要通过量化将参数从浮点数转换为 8bit 定点数。这样做有以下优点：

（1）权重文件变小，进而可以使用容量更小的存储设备，节约存储资源；

（2）提高计算速度;

（3）减少对外部 DDR 的读写数据量。

进行模型量化后，生成量化文件。

3、仿真推理

上一步量化操作可能导致模型精度下降，可在 PC 端进行仿真推理验证，对比结果是否一致。

首先需要使用非量化情况下的模型运行生成每一层的 tensor 作为 Golden tensor。其中输入的数据可以是数据集中的任意一个数据，若多个数据则逐一产生对应的 tensor 文件。然后量化情况下使用相同的数据再次输出一次 tensor，对比这一次输出的每一层的 tensor 与 Golden tensor 的差别。符合要求则可进行下一步；若不符合要求则再次进行模型量化，返回量化调优。

技巧

量化过程不仅对参数进行了量化，也会对输入输出的数据进行量化。所以需要准备一些具有代表性的输入来参与量化，一般从训练模型的数据集里获得，例如训练数据集里的图片。选择几百张能够代表所有场景的输入数据即可。理论上说，量化数据放入得越多，量化后精度可能更好，但是到达一定阈值后效果增长将会非常缓慢甚至不再增长。

4、模型导出

导出工程代码和 NBG 文件，工程代码可在 IDE 工具上使用，而 NBG 文件在模型部署到设备端上使用。

5、IDE 仿真推理和 Profile（可选）

上文使用 Acuity Toolkit 工具进行仿真推理，也可使用 IDE 工具进行仿真推理，IDE 工具还有 Profile 功能，具体操作请参见《VivantelIDE 工具使用说明》小节。

6、模型部署测试和集成应用

利用 vpm_run 工具可直接验证 NBG 文件在设备端运行。把测试输出的 Tensor 数据与 Golden Tensor 进行对比，符合要求即可集成到基于 VIPLite API 开发的应用中。

4.2 模型准备

可直接使用ai-sdk/models/yolov5s-sim中已固化尺寸的 onnx 模型，也可参考下面步骤准备模型。

4.2.1 获取 onnx 模型

首先，下载 yolov5s.onnx 模型文件，下载链接请参见yolov5s 的 github。

可以用开源的 Netron 工具（Netron 下载链接）打开，查看模型的结构以及输入、输出节点的 shape（点击第一个节点 “images”），如下图所示，其输入是动态的。

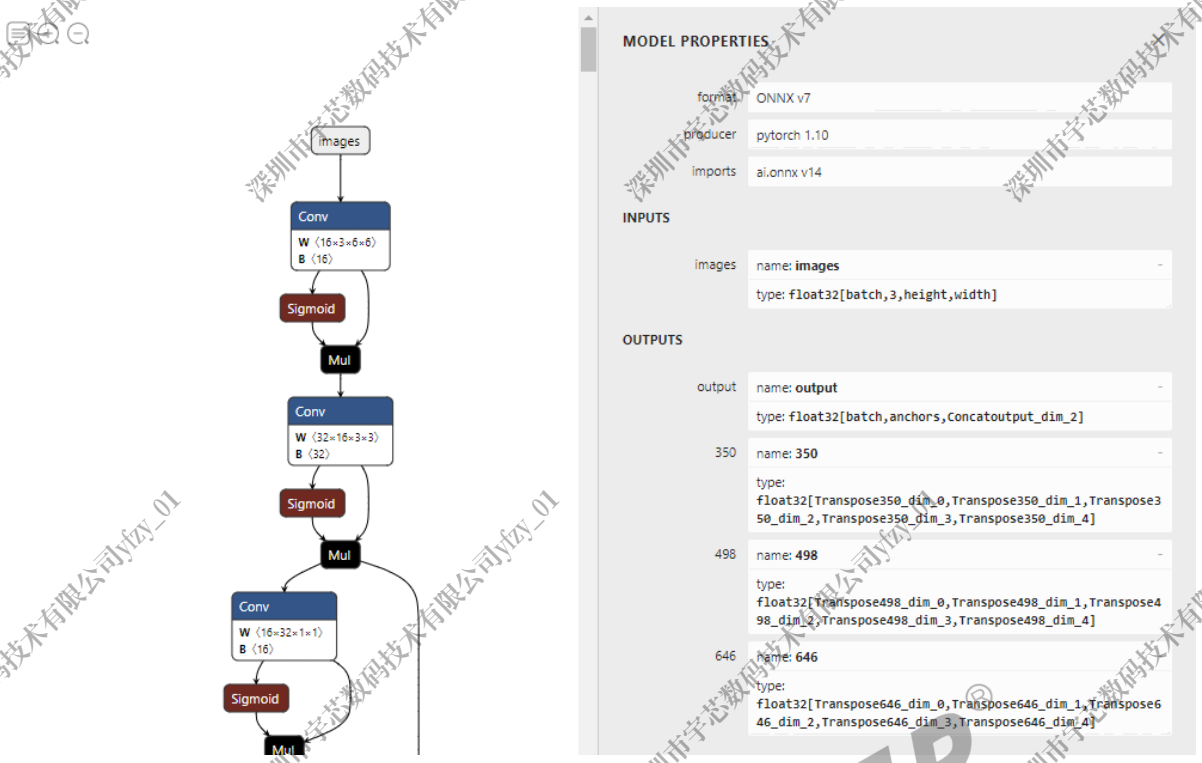

图 4-2: yolov5s 动态输入

4.2.2 固化尺寸

NPU 不支持动态输入，可以通过如下命令（需安装 onnxsim 工具），将输入固定为[1, 3, 640, 640]，同时简化模型。尺寸[1, 3, 640, 640]表示输入一张 640x640 的三通道颜色图片，也可根据自身需求进行设置，width 和 height 是 32 的倍数即可。

```
python3 -m onnxsim yolov5s.onnx yolov5s-sim.onnx --overwrite-input-shape 1,3,640,640
```

固化的输出如下：

```
Your model contains "Tile" ops or/and "ConstantOfShape" ops. Folding these ops can make the simplified model much
        larger. If it is not expected, please specify "--no-large-tensor" (which will lose some optimization chances)
Simplifying...
Finish! Here is the difference:
┏━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━┓
┃               ┃ Original Model ┃ Simplified Model ┃
┡━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━┩
│ Add           │ 10             │ 10               │
│ Cast          │ 3              │ 0                │
│ Concat        │ 41             │ 17               │
│ ConstantOfShape │ 6            │ 0                │
│ Conv          │ 60             │ 60               │
│ Equal         │ 6              │ 0                │
│ Expand        │ 12             │ 0                │
│ Gather        │ 9              │ 0                │
│ MaxPool       │ 3              │ 3                │
│ Mul           │ 75             │ 69               │
│ Pow           │ 3              │ 3                │
```

```
| Range      | 6         | 0         |
| Reshape    | 24        | 6         |
| Resize     | 2         | 2         |
| Shape      | 21        | 0         |
| Sigmoid    | 60        | 60        |
| Slice      | 9         | 9         |
| Sub        | 3         | 3         |
| Transpose  | 3         | 3         |
| Unsqueeze  | 30        | 0         |
| Where      | 6         | 0         |
| Model Size | 7.2MiB    | 7.5MiB    |
```

用 Netron 工具查看固化后的模型结构，如下图所示，输入、输出也固定为对应的尺寸。

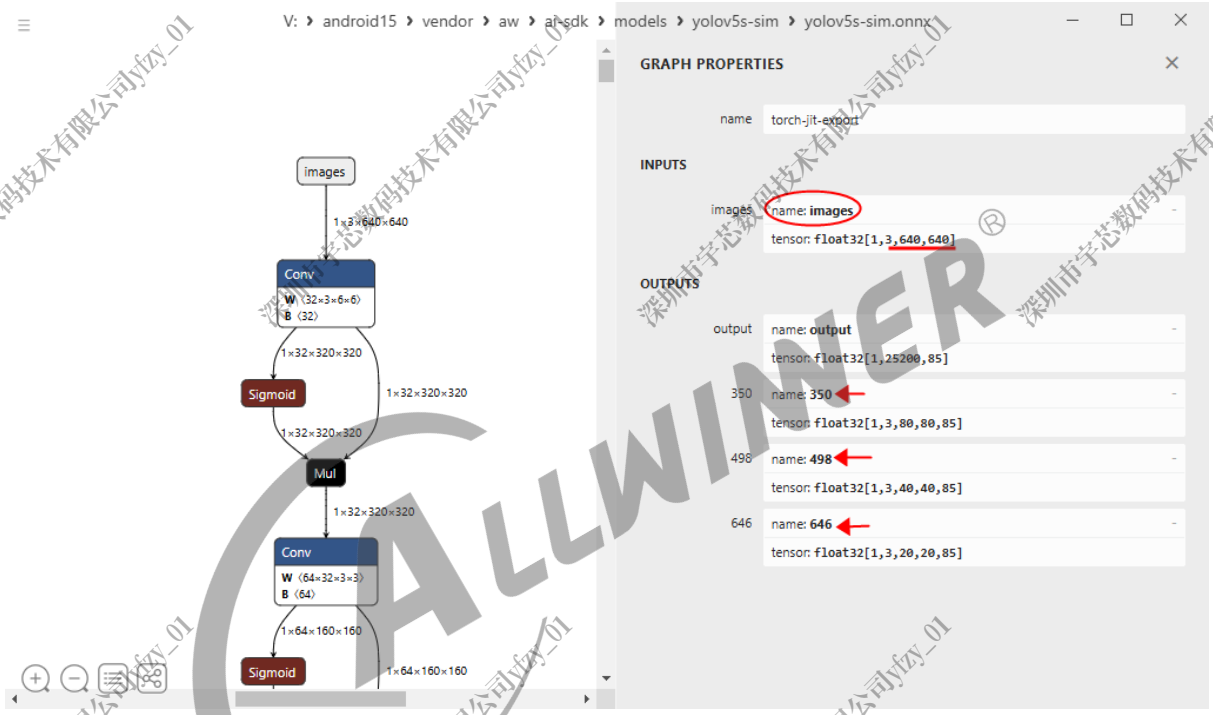

图 4-3: yolov5s-sim 固化输入 640x640

4.2.3 检测效果

下载 yolov5 仓库git clone https://github.com/ultralytics/yolov5.git，用原始代码检测生成的yolov5s-sim.onnx效果，执行以下命令：

```
python3 detect.py --weights ../yolov5s-sim.onnx --source ./data/images/bus.jpg --conf-thres 0.5
```

命令执行的输出如下：

```
detect: weights=['../yolov5s-sim.onnx'], source=./data/images/bus.jpg, data=data/coco128.yaml, imgsz=[640, 640],
        conf_thres=0.5, iou_thres=0.45, max_det=1000, device=, view_img=False, save_txt=False, save_conf=False,
        save_crop=False, nosave=False, classes=None, agnostic_nms=False, augment=False, visualize=False, update=False
        , project=runs/detect, name=exp, exist_ok=False, line_thickness=3, hide_labels=False, hide_conf=False, half=False,
        dnn=False, vid_stride=1
WARNING ⚠ Python 3.7.0 is required by YOLOv5, but Python 3.6.9 is currently installed
```

```
YOLOv5 ☒ v6.2-155-g489920a Python-3.6.9 torch-1.10.0+cu102 CPU

Loading ../yolov5s-sim.onnx for ONNX Runtime inference...
WARNING ⚠ Python 3.7.0 is required by YOLOv5, but Python 3.6.9 is currently installed
image 1/1 /home/zengshuchuan/workspace/onnx/yolov5/data/images/bus.jpg: 640x640 3 persons, 1 bus, 47.3ms
Speed: 5.0ms pre-process, 47.3ms inference, 26.7ms NMS per image at shape (1, 3, 640, 640)
Results saved to runs/detect/exp
```

效果如下图所示。

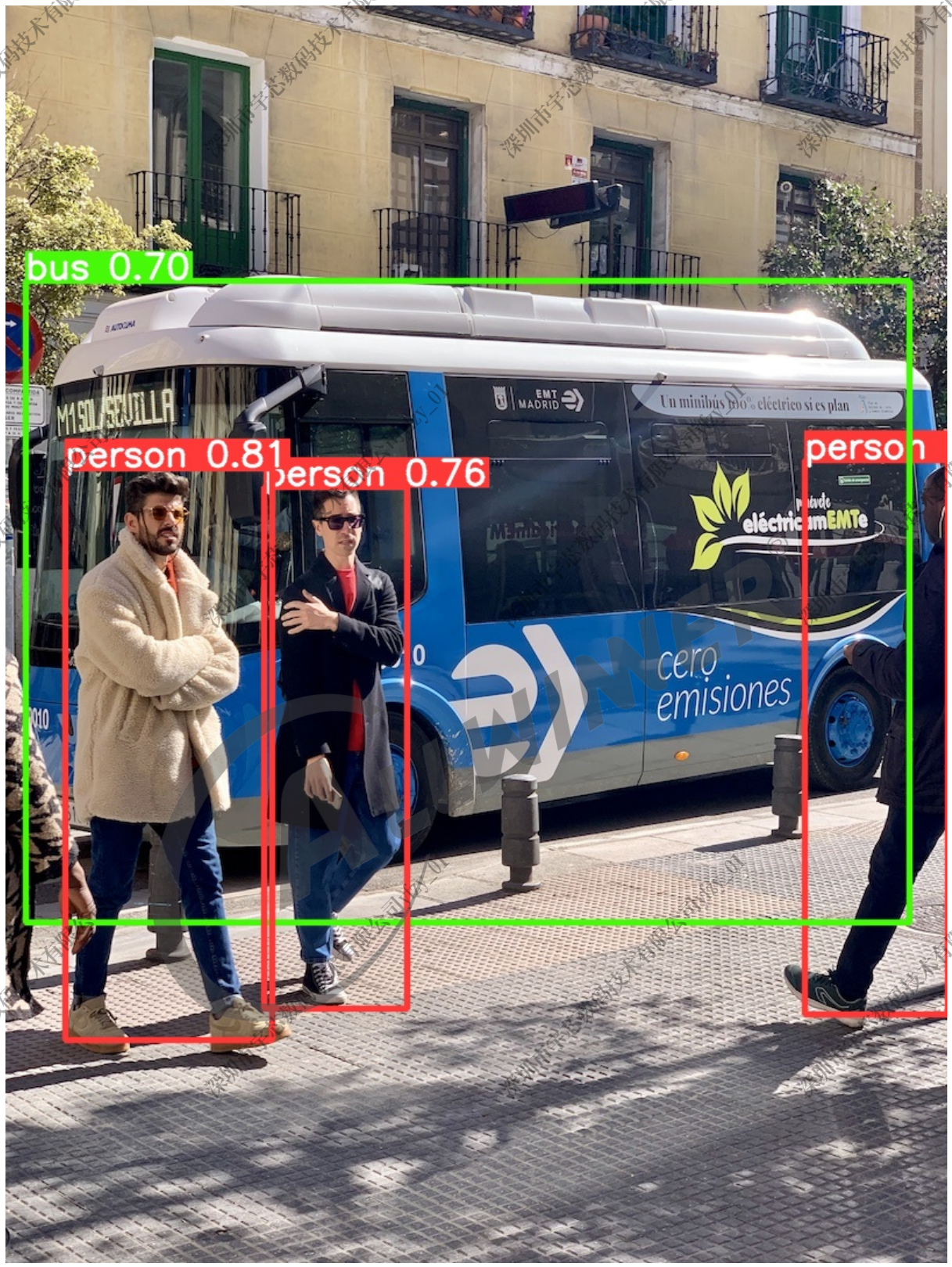

图 4-4: yolov5s 检测效果

4.2.4 确定输出尺寸

模型导入前先分析模型的输入输出节点，通过 Netron 工具查看网络输出部分，如下图所示，可看到模型有 4 个输出节点，其中 output 是其他 3 个输出（350，498，646）经过计算得出。

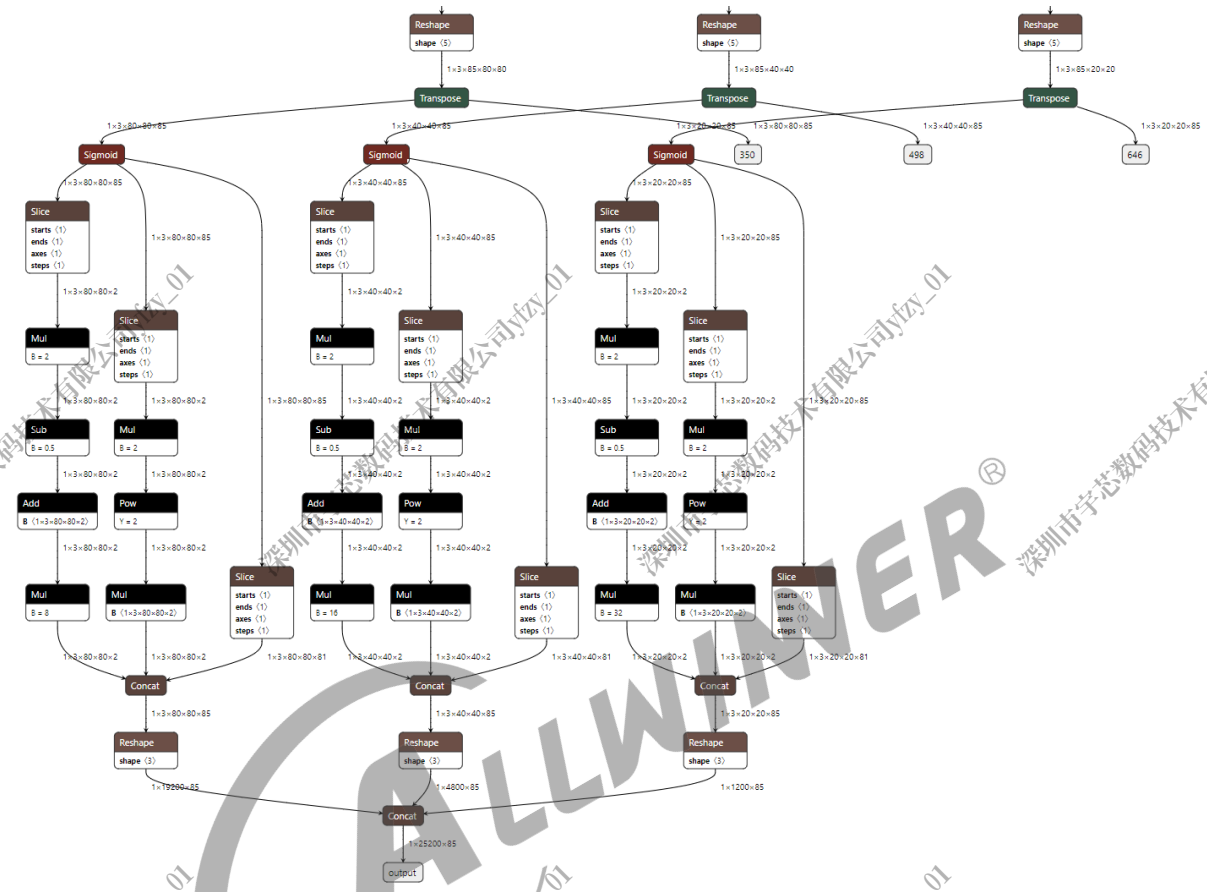

图 4-5: yolov5s-sim 输出节点

4.3 模型转换

说明

不同框架的模型转换示例请参见ai-sdk/models目录。

4.3.1 命令导入

按照模型输入输出，编写inputs_outputs.txt文件，放在与模型文件同一层目录下。inputs_outputs.txt文件内容如下：

```
--inputs images --input-size-list '3,640,640' --outputs '350 498 646'
```

说明

1. **inputs：对应上文图4-3中红色圈出的 INPUTS 的 name。**
2. **input-size-list：对应上文图4-3中红色标出的 INPUTS 的 type 中的尺寸。因为 batch 为 1，无需写，会自动不全，所以写3,640,640。详细说明见《导入命令的详细说明》。**
3. **outputs：，对应上文图4-3中红色标出的 OUTPUTS 的 name。为了精度更优，npu 只执行到 350、498、646 节点，所以需配置 outputs 为'350 498 646'，配合端侧程序，详见设备端程序运行。**

然后使用导入脚本导入模型，参数为模型名称：

```
./pegasus_import.sh yolov5s-sim
```

导入后生成如下的文件：

- yolov5s-sim.json 网络结构文件
- yolov5s-sim.data 网络权重文件
- yolov5s-sim_inputmeta.yml 输入描述文件
- yolov5s-sim_postprocess_file.yml 输出描述文件
- 如果是已量化的 onnx 模型，会生成对应的 yolov5s_uint8.quantize 文件

如不使用 ai-sdk 的脚本，输入以下命令效果同上。

```
pegasus import onnx --model yolov5s-sim.onnx --output-model yolov5s-sim.json --output-data yolov5s-sim.data $(cat
      inputs_outputs.txt)
#功能：导入模型
#onnx 模型的格式类型，还支持caffe、tensorflow、pytorch、keras、tflite、darknet
#--model 指定模型文件
#--output-model 设定网络结构文件名字
#--output-data 设定网络权重文件名字
#--inputs 输入的名字，多个用空格隔开。某些模型需配置。
#--input-size-list 输入的形状大小。某些模型需配置。
#--outputs 输出的名字，多个用空格隔开。某些模型需配置。


pegasus generate inputmeta --model yolov5s-sim.json --separated-database --input-meta-output yolov5s-
    sim_inputmeta.yml
#功能：生成输入描述文件
#--model 指定模型的网络结构文件
#--input-meta-output 设定输入描述文件名字
#--separated-database 针对多输入模型，每个输入使用独立的dataset.txt数据集描述文件

pegasus generate postprocess-file --model yolov5s-sim.json --postprocess-file-output yolov5s-sim_postprocess_file.yml
#功能：生成输出描述文件
#--model 指定模型的网络结构文件
#--postprocess-file-output 设定输出描述文件名字
```

4.3.2 导入命令的详细说明

1. –input-size-list 默认 batch 为 1，所以配置时只需要配置其他维的尺寸。若 batch 为 1，无需配置第 1 维。若 batch 不为 1，需–input-size-list 和–size-with-batch true 配合使用，例如：yolov5 中，batch 为 1，配置–input-size-list ‘3,640,640’；当 batch 为 2 时，配置–input-size-list ‘2,3,640,640’ –size-with-batch ‘True’。

2. 多个输入时，–inputs 要使用单引号括起来，且以空格为间隔，–input-size-list 和–size-with-batch 以 # 分割，例如：2 个输入的尺寸分别是（1，128）和（1，3，640，640）时，配置–inputs ‘input0 input1’ –input-size-list ‘128#3,640,640’ –size-with-batch ‘False#False’。
3. 多个输出时，–outputs 要使用单引号括起来，且以空格为间隔，例如：3 个输出时，配置–outputs ‘350 498 646’。

4.3.3 输入配置说明

通过模型导入，并自动生成基础版的输入描述文件，实际应用中需要根据需求调整文件的配置，yolov5s-sim_inputmeta.yml各项描述如下所示，

```
# !!!This file disallow TABs!!!
# "category" allowed values: "image, frequency, undefined"
# "database" allowed types: "TEXT, NPY, H5FS, SQLITE, LMDB, GENERATOR, ZIP"
# "tensor_name" only support in H5FS database
# "preproc_type" allowed types:"IMAGE_RGB, IMAGE_RGB888_PLANAR, IMAGE_RGB888_PLANAR_SEP, IMAGE_I420,
# IMAGE_NV12, IMAGE_YUV444, IMAGE_GRAY, IMAGE_BGRA, TENSOR"
input_meta:
 databases:
 - path: dataset.txt                  # 数据路径，默认使用dataset.txt数据集描述文件
  type: TEXT                        # 数据类型，dataset.txt即TEXT类型。支持格式见上方说明。
  ports:
  - lid: images_262                   # 输入节点名称
   category: image
   dtype: float32
   sparse: false
   tensor_name:
   layout: nchw                     # 数据排列，N表示Batch,C表示Channel，H表示Height,W表示Width，可配置nchw或nhwc
   shape:                           # 模型的输入shape
   - 1                            # N
   - 3                            # C
   - 640                            # H
   - 640                            # W
  fitting: scale
  preprocess:
   reverse_channel: true             # 通道排列反转
   mean:                           # 3通道各自的均值
   - 0
   - 0
   - 0
   scale:                          # 3通道的缩放值，yolov5s一般设置0.00392157
   - 1.0
   - 1.0
   - 1.0
   preproc_node_params:
    add_preproc_node: false          # 是否添加预处理节点，用于格式转化及裁剪，包含归一化及量化操作。需要设置环境变量export VSI_USE_IMAGE_PROCESS=1生效。
    preproc_type: IMAGE_RGB          # 预处理输入格式，支持格式见上方说明。
    preproc_image_size:              # 预处理图形输入尺寸
    - 640                          # W
    - 640                          # H
    preproc_crop:
     enable_preproc_crop: false      # 是否裁剪尺寸
```

```
    crop_rect:                  # 裁剪区域
      - 0                     # X
      - 0                     # Y
      - 640                     # W
      - 640                     # H
    preproc_perm:               # 用于对模型输入的维度进行置换
    - 0
    - 1
    - 2
    - 3
  redirect_to_output: false           # 把输入直接传给输出，lid需要与postprocess.yml中的lid相同，只用于分类网络。
    acuity推理时可以把分类标签输出到后处理。
```

说明

1. **首先根据模型修改均值参数，如 yolov5s 一般设置 scale: 0.00392157。**
2. **从 Acuity Toolkit 6.21.1 开始，支持 3 通道设置不同的 scale。即 6.21.1 之前的输入配置是scale:1.0，若各通道 scale 值不同，可参考《模型量化问题》。**

postprocess-file.yml 说明见下文《输出配置说明》。

4.3.4 添加预处理节点（可选）

由于 Camera 一般输出格式为 YUV，根据上述配置支持类型，我们选取 IMAGE_NV12 作为输入格式，修改步骤如下：

1. 环境变量设置export VSI_USE_IMAGE_PROCESS=1，打开预处理功能（模型转换导出全过程保持变量）。
2. 修改 inputmeta 配置文件add_preproc_node: true，打开预处理节点。
3. 修改预处理类型preproc_type: IMAGE_NV12，设置输入格式为 NV12。

⚠ 注意

1. **在模型量化和导出后，可使用ai-sdk/tools/nbinfo -in {name}.nb检查模型输入是否调整为两个输入，详见下文。nbinfo 的使用请参考《NBG 分析工具 nbinfo 使用说明》一节，docker 环境已预置 nbinfo 工具。**
2. **在模型部署时，端侧应用程序的输入要做相应的处理，请参考《带预处理节点》一节。**
3. **预处理节点yuv2rgb有两套公式，默认不是OpenCV标准转换公式，环境变量设置export VSI_NN_ENABLE_OCV_NV12=1后再导出 nbg 文件才使用 OpenCV 标准转换公式。**

从以下信息可见，模型的输入变成 y（1x1x640x640）和 uv（1x1x320x640）两个输入节点，说明添加预处理节点成功。

```
root@806a7d436164:/workspace/ai-sdk/models/yolov5s-sim# nbinfo -in yolov5s-sim_uint8.nb
********************************************************************************
Input Table
********************************************************************************
Input 0
Dim Count:                      4
Size of Dim[0]:               640
Size of Dim[1]:               640
Size of Dim[2]:                 1
```

```
Size of Dim[3]:                 1
Data Format:                    NBG_BUFFER_FORMAT_UINT8
Data Type:                      NBG_BUFFER_TYPE_TENSOR
Quantization Format:            NBG_BUFFER_QUANTIZE_NONE
Fixed Point Pos:                0
TF Scale:                       1.000000
TF Zeropoint:                   0
Memory Size (bytes):            409600
input name:                     input[0]
****************************************************************************
Input 1
Dim Count:                      4
Size of Dim[0]:                 640
Size of Dim[1]:                 320
Size of Dim[2]:                 1
Size of Dim[3]:                 1
Data Format:                    NBG_BUFFER_FORMAT_UINT8
Data Type:                      NBG_BUFFER_TYPE_TENSOR
Quantization Format:            NBG_BUFFER_QUANTIZE_NONE
Fixed Point Pos:                0
TF Scale:                       1.000000
TF Zeropoint:                   0
Memory Size (bytes):            204800
input name:                     input[1]
****************************************************************************
```

4.3.5 其它类型模型导入

除 onnx 模型外，ai-sdk/models/目录下还提供了 caffe、tensorflow、tflite、darknet、keras 和 pytorch 框架下生成的网络模型，具体信息如下：

model	platform
lenet	caffe
lstm_mnist	tensorflow
mobilenet_v1_1.0_224_quant	tflite
yolov3_tiny	darknet
MobileNetV2_Imagenet	keras
squeezenet1_0	pytorch
inception_v1	onnx
deepspeech2	tensorflow
yolov5s-sim	onnx

对于不同类型的模型，均可以使用 pegasus_import.sh 脚本，运行如下命令即可实现模型导入。部分框架下的模型导入时需要提供 inputs_outputs.txt 文件，不同模型的 inputs_outputs.txt 文件内容也会略有差异，具体参考 ai-sdk/models/目录下的示例模型。

```
./pegasus_import.sh model_name
```

使用 pegasus 命令也可以实现模型导入，下面以 ai-sdk/models/目录下的 demo 为例说明导入操

作命令。

- caffe 模型

```
# caffe  lenet
pegasus import caffe --model lenet.prototxt --weights lenet.caffemodel --output-model  lenet.json  --output-data lenet.
    data
```

- tensorflow 模型

```
# tensorflow  lstm_mnist
pegasus import tensorflow --model lstm_mnist.pb --output-model lstm_mnist.json --output-data lstm_mnist.data --
    inputs x --outputs logits --input-size-list 28,28,1
```

⚠ 注意

tensorflow 模型的导入必须设置三个参数：--inputs, --input-size-list, --outputs。这三个参数的详细说明见《导入命令的详细说明》。

- tflite 模型

```
# tflite  mobilenet_v1_1.0_224_quant
pegasus import tflite --model mobilenet_v1_1.0_224_quant.tflite --output-model mobilenet_v1_1.0_224_quant.json --
    output-data mobilenet_v1_1.0_224_quant.data
```

- darknet 模型

```
# darknet  yolov3_tiny
pegasus import darknet --model yolov3_tiny.cfg --weights yolov3_tiny.weights --output-model yolov3_tiny.json --output
    -data yolov3_tiny.data
```

- keras 模型

```
# keras  MobileNetV2_Imagenet
pegasus import keras --model MobileNetV2_Imagenet.h5 --output-model MobileNetV2_Imagenet.json --output-data
    MobileNetV2_Imagenet.data
```

- pytorch 模型，不同模型的 –input-size-list 参数值不同。

```
# pytorch  squeezenet1_0，
pegasus import pytorch --model squeezenet1_0.pt --output-model squeezenet1_0.json --output-data squeezenet1_0.
    data --input-size-list 3,227,227
```

- onnx 模型，不同模型的 –inputs、–outputs 和 –input-size-list 参数值不同，注意 –outputs 参数使用单引号括起来。

```
# onnx  yolov5s-sim
pegasus import onnx --model yolov5s-sim.onnx --output-model yolov5s-sim.json --output-data yolov5s-sim.data --
      inputs images --input-size-list 3,640,640 --outputs '350 498 646'
```

4.4 模型量化

4.4.1 准备校准数据

根据模型输入描述文件中path: dataset.txt，我们将 yolov5 github 中 data/images 选取量化所需的图片复制到模型转换目录。

通过命令find images/* > dataset.txt生成 yolov5s-sim_inputmeta.yml 中配置的数据集路径。

技巧

工具默认只会将图像拉伸缩放至网络输入尺寸，输入数据应该按模型输入尺寸给出，以达到较好量化效果。

最终文件列表如下所示：

```
├── images
│   ├── COCO_train2014_000000000081.jpg
│   ├── COCO_train2014_000000001569.jpg
│   ├── COCO_train2014_000000002849.jpg
│   ├── COCO_train2014_000000004139.jpg
│   ├── COCO_train2014_000000005933.jpg
│   └── dog.jpg
├── yolov5s-sim.onnx
└── dataset.txt
```

dataset.txt 文件内容如下所示：

```
./images/COCO_train2014_000000000081.jpg
./images/COCO_train2014_000000001569.jpg
./images/COCO_train2014_000000002849.jpg
./images/COCO_train2014_000000004139.jpg
./images/COCO_train2014_000000005933.jpg
./images/dog.jpg
```

4.4.1.1 多输入模型配置

多输入模型生成的输入描述文件x_inputmeta.yml的核心差异结构如下：

```
#以两个输入的模型进行演示
#pegasus generate inputmeta时，设置了--separated-database
input_meta:
 databases:
 - path: dataset0.txt # 输入a_2的数据集路径
   type: TEXT
   ports:
   - lid: a_2
   category: image
...
```

```
    shape:
    - 1
    - 1
    - 160
    - 3
...
 - path: dataset1.txt # 输入b_3的数据集路径
   type: TEXT
   ports:
   - lid: b_3
     category: image
...
    shape:
    - 1
    - 1
    - 3
    - 256
```

有以下三种配置方法：

技巧

batch 不为 1 时请使用第三种配置方法。

第一种配置方法，就是创建 dataset0.txt 和 dataset1.txt，分别写入数据的路径。jpg 等数据格式同理。

```
# cat dataset0.txt
a_1_1_160_3_0.npy
a_1_1_160_3_1.npy

# cat dataset1.txt
b_1_1_3_256_0.npy
b_1_1_3_256_1.npy
```

第二种配置方法，所有输入共用 dataset0.txt，dataset0.txt 中一行对应多个输入的数据文件路径，中间用空格隔开。jpg 等数据格式同理。

按照以下结构修改输入描述文件或生成输入描述文件pegasus generate inputmeta时删除–separated-database 的配置：

```
#cat x_inputmeta.yml
input_meta:
 databases:
 - path: dataset0.txt  #只有一个path
   type: TEXT
   ports:
   - lid: a_2
     category: image
...
   - lid: b_3       #注意其他输入节点的层级关系
     category: image


#cat dataset0.txt
a_1_1_160_3_0.npy b_1_1_3_256_0.npy
a_1_1_160_3_1.npy b_1_1_3_256_1.npy
```

第三种配置方法，把所有 batch 的数据放到一个 npy 文件里，只配置一个量化数据，path 中直接

配置数据文件路径。

```
#cat net/net_inputmeta.yml
input_meta:
 databases:
 - path: a_10_1_160_3.npy
   type: NPY
   ports:
   - lid: a_2
     category: image
...
 - path: b_10_1_3_256.npy
   type: NPY
   ports:
   - lid: b_3
     category: image
```

当 batch 不为 1 时，例如 shape 为 (2,1,256)，iterations 配置将为 10，把数据放到一个 npy 文件里面，shape 为（20,1,256）。

4.4.2 uint8 量化

使用量化脚本，输入模型名称，默认使用 uint8 量化，可以用 uint8，int16，pcq。

```
./pegasus_quantize.sh yolov5s-sim uint8
```

量化后生成如下的文件：

- yolov5s-sim_uint8.quantize 量化描述文件
- entropy.txt 熵值文件，每层的 output 以及 kernel（文件中的 weight）离散度，用来评估每层量化误差，熵值范围 0 到 1，值越大，精度越差

x_uint8.quantize 量化描述文件说明见下文《量化表参数说明》。

如不使用 ai-sdk 的脚本，输入以下命令效果同上。

```
pegasus quantize --model yolov5s-sim.json --model-data yolov5s-sim.data --device CPU --with-input-meta yolov5s-
      sim_inputmeta.yml --compute-entropy --rebuild --model-quantize yolov5s-sim_uint8.quantize --quantizer
      asymmetric_affine --qtype uint8

#功能：某一种精度量化
#--model    指定模型的网络结构文件
#--model-data 指定模型的网络权重文件
#--device    指定服务器的计算硬件
#--with-input-meta 指定模型的输入描述文件
#--compute-entropy 计算熵值，输出entropy.txt
#--rebuild   生成量化描述文件（与--hybrid、--rebuild-all互斥）
#--model-quantize 指定输出的量化描述文件
#--quantizer 量化类型，有asymmetric_affine，symmetric_affine，dynamic_fixed_point，
      perchannel_symmetric_affine等
#--qtype    量化数据类型，有int8，uint8，int16等
```

⚠注意

实际量化时需要配置--iterations 参数指定量化使用的数据量，否则默认只使用 dataset 中第一条数据。

4.4.3 混合量化

经过 uint8 量化后，发现精度不满足要求时，需要使用混合量化进行进一步量化调优。如果量化时打开--compute-entropy不仅输出熵值文件，Acuity 工具还会自动根据各层熵值情况，在量化描述文件中给出某些层的量化建议。用户可根据建议进行混合量化，也可根据其它方式排查，手动修改 yolov5s-sim_uint8.quantize 中的量化参数。然后将量化命令参数中--rebuild改为--hybrid，进行混合量化。

```
pegasus quantize --model yolov5s-sim.json --model-data yolov5s-sim.data --device CPU --with-input-meta yolov5s-
      sim_inputmeta.yml --compute-entropy --hybrid --model-quantize yolov5s-sim_uint8.quantize --quantizer
      asymmetric_affine --qtype uint8

#功能：混合量化
#--hybrid 混合量化（与--rebuild、--rebuild-all互斥）
```

输出文件如下，

- entropy.txt 熵值文件
- yolov5s-sim_uint8.quantize.json 增加混合量化后网络结构文件，替换 yolov5s-sim.json 进行推理和导出使用

查看 entropy.txt，发现部分层熵值过高（output 离散度 >0.5，kernel 离散度 >0.6），推荐对应 layer 用高精度的量化参数。实际场景以输出结果精度为准，请根据《量化精度调优》一章优化量化精度。

4.5 仿真推理

执行以下脚本输出 float 仿真结果，参数：模型名称，量化类型 float，int16，uint8，pcq。

```
./pegasus_inference.sh yolov5s-sim float
```

如不使用 ai-sdk 的脚本，输入以下命令效果同上。

```
pegasus inference --model yolov5s-sim.json --model-data yolov5s-sim.data --dtype float32 --iterations 1 --device CPU --
      output-dir ./inf/yolov5s-sim_non-quantized --postprocess-file yolov5s-sim_postprocess_file.yml --with-input-meta
       yolov5s-sim_inputmeta.yml

#功能：非量化仿真推理
#--model    指定模型的网络结构文件
#--model-data 指定模型的网络权重文件
#--qtype    量化数据类型，有float32,quantized
#--device    指定服务器的计算硬件
#--output-dir 输出路径
#--postprocess-file 指定模型的输出描述文件
```

```
#--with-input-meta 指定模型的输入描述文件
```

输出 float 仿真结果如下：

```
inf/yolov5s-sim_non-quantized
├─── iter_0_attach_Concat_Concat_532_out0_0_out0_1_25200_85.tensor
├─── iter_0_attach_Transpose_Transpose_214_out0_1_out0_1_3_80_80_85.tensor
├─── iter_0_attach_Transpose_Transpose_326_out0_2_out0_1_3_40_40_85.tensor
├─── iter_0_attach_Transpose_Transpose_438_out0_3_out0_1_3_20_20_85.tensor
└─── iter_0_images_262_out0_1_3_640_640.tensor
```

执行以下脚本输出 uint8 量化结果，

```
./pegasus_inference.sh yolov5s-sim uint8
```

如不使用 ai-sdk 的脚本，输入以下命令效果同上。

```
pegasus inference --model yolov5s-sim.json --model-data yolov5s-sim.data --dtype quantized --model-quantize yolov5s-
    sim_uint8.quantize --iterations 1 --device CPU --output-dir ./inf/yolov5s-sim_uint8 --postprocess-file yolov5s-
    sim_postprocess_file.yml --with-input-meta yolov5s-sim_inputmeta.yml

#功能：量化仿真推理
#--model-quantize 指定模型的量化描述文件
#--iterations 数据量
#其他参数同上
```

输出 uint8 仿真结果如下：

```
inf/yolov5s-sim_uint8
├─── iter_0_attach_Concat_Concat_532_out0_0_out0_1_25200_85.tensor
├─── iter_0_attach_Transpose_Transpose_214_out0_1_out0_1_3_80_80_85.tensor
├─── iter_0_attach_Transpose_Transpose_326_out0_2_out0_1_3_40_40_85.tensor
├─── iter_0_attach_Transpose_Transpose_438_out0_3_out0_1_3_20_20_85.tensor
└─── iter_0_images_262_out0_1_3_640_640.tensor
```

逐一对比 float 与 uint8 量化仿真结果的相似度，命令如下：

```
python3 $ACUITY_PATH/tools/compute_tensor_similarity.py a.tensor b.tensor
```

其中 a.tensor，b.tensor 分别传入 inf 目录中对应的 tensor 文件，输出如下，

```
Instructions for updating:
dim is deprecated, use axis instead
euclidean_distance 1445.037
cos_similarity 0.999855
```

根据仿真数据与相似度对比，可以初步评估模型量化后是否满足要求，基于此重新优化量化参数，或者进行下一步模型导出。

4.6 模型导出

4.6.1 输出配置说明

模型导入时，生成基础版的输出描述文件，实际应用中需要根据需求调整文件的配置，yolov5s-sim_postprocess_file.yml各项描述如下所示。

```
# "acuity_postprocs" allowed types: "classification_validate, detection_validate, dump_results, print_topn,
      classification_classic, mute_built_in_actions"
postprocess:
 acuity_postprocs:                    # acuity工具仿真使用后处理节点，具体参数配置见上方说明
  print_topn:
   topn: 5
  dump_results:
   file_type: TENSOR
  # mute_built_in_actions: true
  # python:
  #  file_path: postprocess.py
  #  parameters:
  #   output_tensors:
  #    - '@xxxx:out0'
 app_postprocs:
 - lid: attach_Concat_Concat_532/out0_0
  postproc_params:
   add_postproc_node: false          # 是否添加后处理节点
   perm:                          # 对模型输出的维度进行置换。
   - 0
   - 1
   - 2
   force_float32: true              # 强制将输出数据类型设置为 float32 的参数（如果是int会自动反量化）
 - lid: out0_1 ~ out0_3 #省略
```

依据实际场景选择，若实测使用 PPU 进行反量化效率优于软件实现且释放 CPU 资源，可将配置中输出节点都改为add_postproc_node: true。

4.6.2 导出 NBG 文件及工程代码

导出设备端工程，使用以下脚本导出，参数：模型名称，量化类型，NPU 型号，SDK 工具目录。

```
./pegasus_export_ovx_nbg.sh yolov5s-sim uint8 VIP9000PICO_PID0XEE ${VIV_SDK}
```

说明

1. 各平台型号参数

版本	平台	型号
v1	V85x、R853s	VIP9000PICO_PID0XEE
v2	MR527、AI985、T527	VIP9000NANOSI_PLUS_PID0X10000016
v3	MR536、T536、A733、T736	VIP9000NANODI_PLUS_PID0X1000003B

2. 若使用混合量化，json 文件更新了，pegasus_export_ovx_nbg 脚本设定的 json 文件名字不对，需要使用下文的完整命令。

如不使用 ai-sdk 的脚本，输入以下命令效果同上。

```
pegasus export ovxlib --model yolov5s-sim.json --model-data yolov5s-sim.data --dtype quantized --model-quantize
      yolov5s-sim_uint8.quantize --target-ide-project 'linux64' --with-input-meta yolov5s-sim_inputmeta.yml --
      postprocess-file yolov5s-sim_postprocess_file.yml --pack-nbg-unify --optimize VIP9000PICO_PID0XEE --viv-sdk ${
      VIV_SDK} --output-path ./wksp/yolov5s-sim_uint8/yolov5s-sim_uint8

#--model    指定模型的网络结构文件，注意使用混合量化后要修改此参数用新的json文件
#--model-data 指定模型的网络权重文件
#--qtype    量化数据类型，有float32,quantized
#--model-quantize 指定模型的量化描述文件
#--target-ide-project
#--with-input-meta 指定模型的输入描述文件
#--postprocess-file 指定模型的输出描述文件
#--pack-nbg-unify
#--optimize
#--viv-sdk  SDK路径，用环境变量即可
#--output-path输出路径
```

导出后生成以下文件：

```
wksp
├── yolov5s-sim_uint8
│   ├── BUILD
│   ├── graph.json
│   ├── main.c
│   ├── makefile.linux
│   ├── vnn_global.h
│   ├── vnn_post_process.c
│   ├── vnn_post_process.h
│   ├── vnn_pre_process.c
│   ├── vnn_pre_process.h
│   ├── vnn_yolov5ssimuint8.c
│   ├── vnn_yolov5ssimuint8.h
│   ├── yolov5ssimuint8.2012.vcxproj
│   ├── yolov5s-sim_uint8.export.data
│   └── yolov5ssimuint8.vcxproj
└── yolov5s-sim_uint8_nbg_unify
    ├── BUILD
    ├── main.c
    ├── makefile.linux
    ├── nbg_meta.json
    ├── network_binary.nb
    ├── vnn_global.h
    ├── vnn_post_process.c
    ├── vnn_post_process.h
    └── vnn_pre_process.c
```

```
├──── vnn_pre_process.h
├──── vnn_yolov5ssimuint8.c
├──── vnn_yolov5ssimuint8.h
├──── yolov5ssimuint8.2012.vcxproj
└──── yolov5ssimuint8.vcxproj
```

其中wksp/yolov5s-sim_uint8_nbg_unify包含设备端代码及network_binary.nb模型文件。

说明

工具生成的设备端代码用于 IDE 工具或 Unified 版本驱动，VIPLite 版本驱动不适用。

4.7 模型部署

4.7.1 设备端 vpm_run 运行

根据《模型运行工具 vpm_run 使用说明》一节在设备端运行和验证。

4.7.1.1 无预处理节点

如模型转换时无添加预处理节点，则可使用仿真生成的iter_0_images_262_out0_1_3_640_640.tensor作为输入，编辑配置文件yolov5.txt，

```
[network]
./network_binary.nb
[input]
./iter_0_images_262_out0_1_3_640_640.tensor
```

将相关文件推到设备端中，

- vpm_run
- network_binary.nb
- yolov5.txt
- iter_0_images_262_out0_1_3_640_640.tensor

使用命令执行结果，

```
./vpm_run -s yolov5.txt
```

生成输出文件，

```
output_0.txt
output_1.txt
output_2.txt
output_3.txt
```

将板端输出文件提取到 PC 工作目录，使用相似度工具比较设备端运行输出与 uint8 仿真结果，发现 output_0 差异非常大，而其它 3 个输出能达到 97% 以上相似度。因为该模型后处理节点不适

合量化，所以根据 3 个输出进一步计算所得的 output_0 精度损失严重，建议这部分运行在 CPU 或采用 float 量化运行在 PPU）。根据《量化精度调优》一章对涉及的 layer 进行量化精度调优。

4.7.1.2 带预处理节点

模型转换时，如添加了预处理节点，可支持输入格式转化。如上述例子中模型已添加 NV12 输入转化节点，可以准备对应格式数据。

1. 转化 NV12，使用 ffmpeg 工具，

```
ffmpeg -i images/bus.jpg -s 640x640 -pix_fmt nv12 input.yuv
```

2. NV12 输入是分开 Y 及 UV 两个通道，通过以下命令将 input.yuv 拆分，

```
if=input.yuv of=input0.bin bs=640 count=640
if=input.yuv of=input1.bin bs=640 count=320 skip=320
```

修改配置文件，并将相关文件推到设备端中，

```
[network]
./network_binary.nb
[input]
./input0.bin
./input1.bin
```

说明

IMAGE_I420,IMAGE_NV12,IMAGE_NV21, IMAGE_YUV444, IMAGE_YUYV422, IMAGE_UYVY422 等 YUV 格式都为 2 个输入，与上述 NV12 方法一致

IMAGE_RGB, IMAGE_RGB888_PLANAR, IMAGE_RGB888_PLANAR_SEP, IMAGE_GRAY, IMAGE_BGRA 为一个输入，只需进行 ffmpeg 格式转化即可

4.7.2 设备端程序运行

可参考 vpm_run 工具的源码，基于轻量级 VIPLite API 编写 C++ 应用程序运行在设备端，运行一个模型一般流程如图所示。

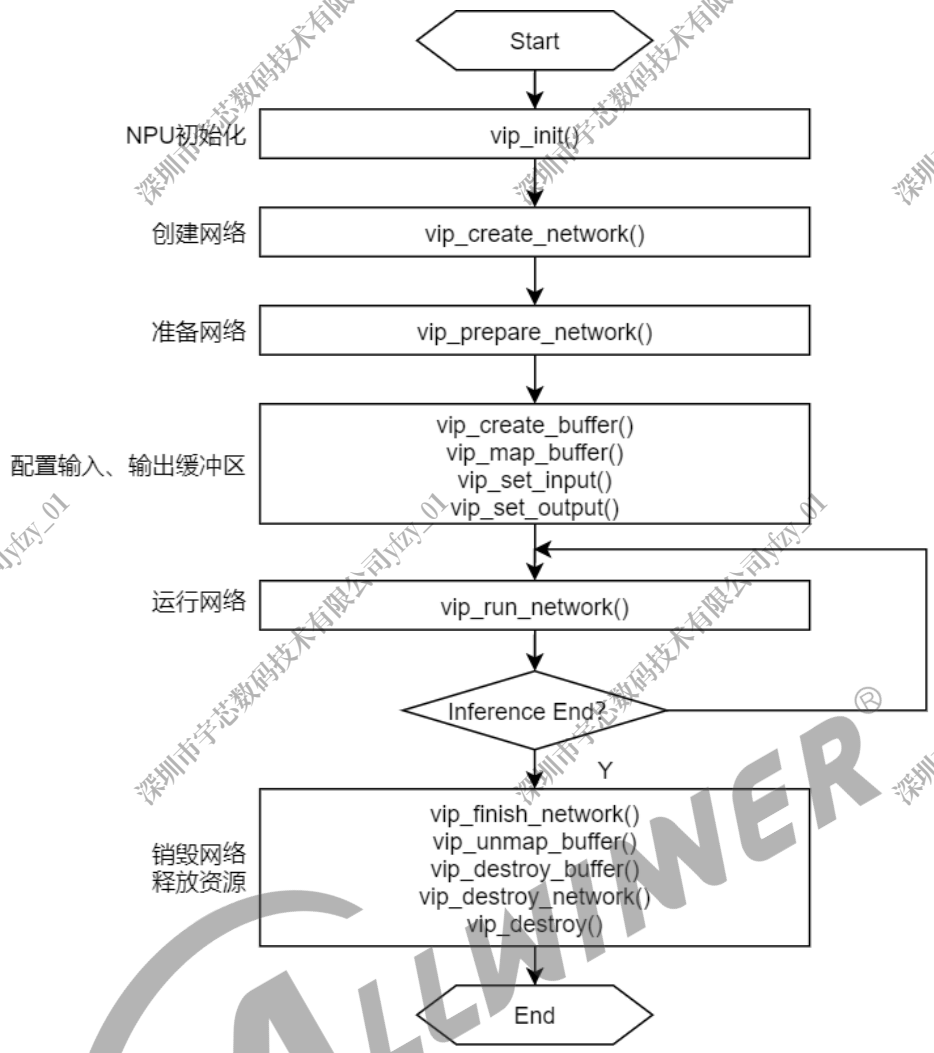

图 4-6: NPU 的轻量级网络级接口调用流程图

技巧

模型部署支持多模型运行，参见《多线程调用》。

yolov5 的设备端程序请参见ai-sdk/examples/yolov5。

⚠ 注意

yolov5 的示例程序只处理三个 output 节点，所以在《模型转换》中 import 模型时，要配置--outputs '350 498 646'。正如上文所述，vpm_run 验证发现 output_0 差异非常大，所以示例程序把 output_0 的计算放在 CPU 处理。

所以inputs_outputs.txt文件内容是--inputs images --input-size-list '3,640,640' --outputs '350 498 646'

4.7.2.1 Linux 编译

使用 Tina 预置编译

V85x、R853 平台：

```
#在SDK根目录执行以下命令
make menuconfig
Allwinner  --->
ai-sdk selection  --->
<*> yolov5..... yolov5 demo

make -j32
#输出文件在~/sdk_dir/out/xxx/compile_dir/target/ai-sdk/yolov5
```

MR527、AI985、MR536 平台：

```
#在SDK根目录执行以下命令
make menuconfig
Allwinner  --->
Vision  --->
<*> ai-sdk-viplite..... allwinner npu viplite framework  --->

make -j32
#输出文件在~/sdk_dir/out/xxx/openwrt/build_dir/target/ai-sdk/ipkg.../etc/npu/yolov5
#安装到设备在/etc/npu/yolov5
```

make 进行全局编译一次后，后续修改示例程序源码，可以进入该目录，使用mm -B命令单独编译yolov5。

⚠ 注意

1. **MR536 平台的源码在platform/allwinner/vision/ai-sdk，需在openwrt/package/allwinner/vision/ai-sdk路径进行mm。**
2. **MR527 平台部分 SDK，需要同步压缩包的文件夹（下载链接）才能编译。**

T527、T536、T736 Linux 平台：

```
#在SDK根目录执行以下命令
./build.sh buildroot_menuconfig
Target packages  --->
allwinner platform private package select  --->
allwinner  --->
vision  --->
ai-sdk  --->
[*] ai sdk
[*]   ai sdk viplite lib and vpm_run
[*]    ai sdk viplite yolov5
./build.sh buildroot_package ai-sdk-rebuild

#输出文件在~/sdk_dir/out/xxx/xxx/buildroot/buildroot/target/etc/npu/yolov5
#安装到设备在/etc/npu/yolov5
```

📖 说明

AIOT Linux v1.4 SDK 之前，按照 Target packages —>allwinner platform private package select —>ai-sdk 查找。

4.7.2.2 Android 编译

```
#在SDK根目录执行以下命令
AwExdroid88:~/Android_sdk/$ cd vendor/aw/ai-sdk/examples/yolov5
AwExdroid88:~/Android_sdk/$ mm

#输出文件分别在以下路径
#~/Android_sdk/out/target/product/xxx/vendor/etc/models/yolov5.nb
#~/Android_sdk/out/target/product/xxx/vendor/etc/input_data/dog.jpg
#~/Android_sdk/out/target/product/xxx/vendor/bin/yolov5
#~/Android_sdk/out/target/product/xxx/vendor/bin/yolov5.sh

#安装到设备在/vendor/bin/yolov5
```

4.7.2.3 运行示例程序

把二进制文件 yolov5、测试图片 dog_640_640.jpg 和模型文件 yolov5s-sim_uint8.nb（整个 test 目录）推到板端，执行以下命令：

```
./yolov5 yolov5s-sim_uint8.nb dog_640_640.jpg

#运行结果见result.png
```

若编译时进行预置，也可以直接执行 yolov5.sh 查看效果：

```
# cat /vendor/bin/yolov5.sh
#!/bin/sh

yolov5 /vendor/etc/models/yolov5.nb /vendor/etc/input_data/dog.jpg
```

yolov5s-sim_uint8.nb 文件即模型导出时生成在wksp/yolov5s-sim_uint8_nbg_unify/network_binary.nb。

Android 设备可推到/data目录，容量比较小的 Tina 设备可插入 SD 卡再推到/mnt/extsd目录中。

⚠ 注意

NBG 模型，各平台不通用，请选择正确平台的 NBG 模型。在 example 中，MR527、AI985、T527 Linux 平台使用 v2 模型，MR536、T536、A733、T736 平台使用 v3 模型。

4.7.3 数据预处理和后处理（可选）

如果模型中不包含前、后处理节点，则需应用程序自行实现，以下为参考例子。

预处理数据归一化、量化过程如下：

```
void *preprocess(unsigned char *buffer) {
  unsigned int width = 640;
  unsigned int height = 640;
  unsigned int depth = 3;
```

```
  unsigned int i, j;
  float means[] = {123.68, 116.78, 103.94};
  float stds[] = {58.40, 57.12, 57.38};
#ifdef QUAN_ASYMM
  float scale = 0.0039;
  unsigned int zero_point = 111;
  unsigned char *input = (unsigned char *)malloc(width * height * depth);
#elif defined(QUAN_INT16_FIXED_POINT)
  unsigned char fix_pos = 10000;
  short *input = (short *)malloc(width * height * depth);
#endif
  for (i = 0; i < depth; i++) {
    for (j = 0; j < width * height; j++) {
      float std = (buffer[i * width * height + j] - means[i]) / stds[i];
#ifdef QUAN_ASYMM
      input[i * width * height + j] = (unsigned char)(std / scale + zero_point);
#elif defined(QUAN_INT16_FIXED_POINT)
      if (fix_pos > 0) {
        input[i * width * height + j] = (short)(std * (float)(1 << fix_pos));
      } else {
        input[i * width * height + j] = (short)(std / (float)(1 << -fix_pos));
      }
#endif
    }
  }
  return (void *)input;
}
```

技巧

如果输入节点为 uint8 非对称量化，如果 3 通道归一化参数一致，则图像经过归一化及量化后与原始数据一致，可直接把原始数据用于输入。

后处理反量化过程如下：

```
float *postprocess(void *output, unsigned int size) {
  unsigned int i;
#ifdef QUAN_ASYMM
  float scale = 0.0039;
  unsigned int zero_point = 111;
#elif defined(QUAN_INT16_FIXED_POINT)
  unsigned char fix_pos = 10000;
#endif
  float *result = (float *)malloc(size);
  for (i = 0; i < size; i++) {
#ifdef QUAN_ASYMM
      result[i] = ((float)output[i] - zero_point) * scale;
#elif defined(QUAN_INT16_FIXED_POINT)
      if (fix_pos > 0) {
        result[i] = ((float)output[i] / (float)(1 << fix_pos));
      } else {
        result[i] = ((float)output[i] * (float)(1 << -fix_pos));
      }
#endif
  }
  return result;
}
```

4.8 Docker 中部署示例程序

4.8.1 Ubuntu 机器操作

进入 Docker 容器

```
sudo docker images
sudo docker run -it --privileged -v /home/${USER}/docker_data:/workspace ubuntu-npu:v1.x /bin/bash

#或者使用容器ID进入
sudo docker ps -a
sudo docker exec -it 容器ID /bin/bash
```

将 mobilenet_v2_ssd 例程放在 docker_data 目录（该目录已挂载到 docker 容器），路径如下：

```
.
├── board-demo
│   ├── 3rdparty
│   ├── common
│   └── mobilenet_v2_ssd_demo
└── model-convert
    └── mobilenet_v2_ssd
```

4.8.2 docker 容器内操作

4.8.2.1 模型量化转换

```
cd /workspace/model-convert/mobilenet_v2_ssd/

# 导入
./pegasus_import.sh mb2-ssd-lite-sim
# 留意生成的两个yml文件，提供的例程中已修改；
# 1.xxx_inputmeta.yml，修改校准文件的路径path，根据算法输入的预处理修改mean，scale的值；
# 2.xxx_postprocess_file.yml，add_postproc_node参数设为true，推理结果由NPU内部做反量化浮点输出；

# 量化
./pegasus_quantize.sh mb2-ssd-lite-sim uint8

# 仿真
./pegasus_inference.sh mb2-ssd-lite-sim uint8

# 导出nb模型
./pegasus_export_ovx_nbg_x527.sh mb2-ssd-lite-sim uint8
# 生成的模型文件为./wksp/mb2-ssd-lite-sim_uint8_nbg_unify/mb2-ssd-lite-sim_xxx.nb
```

4.8.2.2 板端 demo 编译

Linux 平台编译

提供的示例可以直接在 docker 环境下编译，需检查交叉编译工具链是否正确；

其中示例提供的 libVIP*.so 库作为编译链接使用，板端运行以板端的 so 库为准；

```
cd /workspace/board-demo/mobilenet_v2_ssd_demo/

cd ../0-toolchains
#解压选择硬件匹配的交叉编译工具链，下面以MR527平台使用为例
tar xvf gcc-arm-10.3-2021.07-x86_64-aarch64-none-linux-gnu.tar.xz

# 解压opencv库压缩包
cd ../3rdparty/opencv/
# armhf
unzip arm-linux-gnueabihf-aw.zip
# aarch64
unzip aarch64-linux-sunxi-glibc.zip

# 重新进入demo目录
cd /workspace/board-demo/mobilenet_v2_ssd_demo/

# 编译
 mkdir build
cd build
cmake ..
make

# 将可执行文件推到板端运行，建议推到tf卡目录，空间充足
# 将nb模型文件、输入图片push到与执行文件相同目录
# ./mbv2-ssd-demo -h 查看执行示例

# 例如
./mbv2-ssd-demo -b model/mbv2_ssd_xxx.nb -i 000012.jpg
```

Android 平台编译

下载 Android NDK，下载地址：https://developer.android.google.cn/ndk/downloads/index?hl=th

将下载的 NDK 放到编译机器目录，例如：./0-toolchains/；

请根据下载的 NDK 版本修改build_android.sh 文件；

```
#!/bin/bash

# please modify ndk path
#set -e ANDROID_NDK=/your/ndk/path/android-ndk-r25b
#if [ -z ${ANDROID_NDK} ]
#then
#  ANDROID_NDK=/your/ndk/path/android-ndk-r25b
#fi

CUR_DIR=$(cd $(dirname $0) && pwd )

# aarch64
BUILD_DIR=${CUR_DIR}/build_aarch64_android

if [[ ! -d "${BUILD_DIR}" ]]; then
  mkdir -p ${BUILD_DIR}
```

```
fi

cd ${BUILD_DIR}

cmake -DCMAKE_TOOLCHAIN_FILE="$ANDROID_NDK/build/cmake/android.toolchain.cmake"\
  -DCMAKE_MAKE_PROGRAM="$ANDROID_NDK/prebuilt/linux-x86_64/bin/make"  \
  -DANDROID_ABI="arm64-v8a" \
  -DANDROID_PLATFORM=android-24 ..

make
```

运行编译脚本./build_android.sh。

说明

ai-sdk/examples/中的示例随 SDK 发布，也可以在安卓的开发环境下 mm 即可，编译后的可执行程序存放在 ~/out/vendor/bin/。

5 NPU 进阶使用说明

5.1 零拷贝调用

5.1.1 功能介绍

NPU 在处理数据过程中，会跟其他硬件打交道，比如图像检测应用会从 Camera 模块或者解码模块中取数据，再送到 NPU 用于模型运行，最后进行输出。这个过程如果使用 CPU 进行数据搬运，会造成一次或者多次数据拷贝，不仅影响性能，还导致系统带宽增加。零拷贝技术可以通过共享同一 DDR 内存 buffer，实现 NPU 直接用上游硬件输出 buffer 中取数据。

与一般调用流程的差异在于输入缓冲区或输出缓冲区的创建方式不同，如下图的加粗部分。

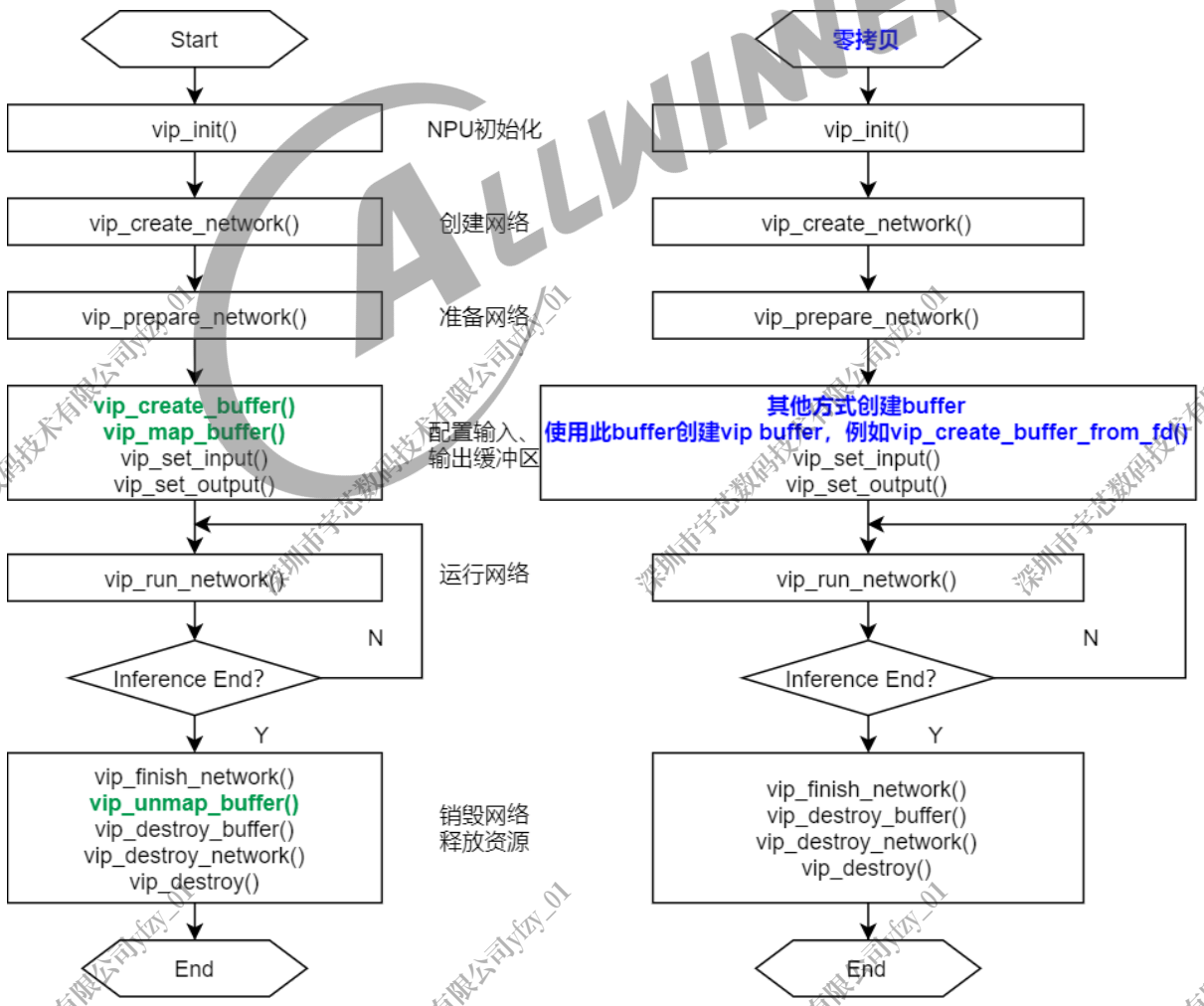

图 5-1: 零拷贝调用与一般流程调用的对比

可按需使用以下接口进行创建缓冲区：

- vip_create_buffer_from_fd() : 使用 DMA buffer 创建 NPU 的 buffer，推荐使用。
- vip_create_buffer_from_physical()：根据指定的物理内存地址创建 NPU 的 buffer。
- vip_create_buffer_from_handle()：根据句柄创建 NPU 的 buffer。V85x、R853 平台不支持。

5.1.2 代码 Demo

demo 代码放置于ai-sdk/examples/shared_buffer。

```
├── Android.bp
├── dog-90.jpg
├── lut.c
├── lut.h
├── lut.nb
├── README.md
├── shared_buffer.c
└── sunxi-g2d.h
```

在 shared_buffer.c 中，通过g2d_rotate(g2d_fd, fd1, fd2)把 dmabuf1 内容旋转 90°，存到 dmabuf2 中，然后调用lut_run，NPU 直接使用 dmabuf2 运行模型，并保存回 dmabuf2 中。在此过程中，g2d 传输到 npu 过程使用相同 buffer，从而节省拷贝的耗时及内存带宽占用。

5.2 多线程调用

目前各平台 NPU 是单核的，但可用多线程方式调用多个网络，并支持对各个网络设置优先级。NPU 硬件根据各网络任务达到时间（vip_run_network()接口）和网络优先级运行网络。

由vip_set_network()接口实现网络优先级的设置，优先级参数为 0~255，0 表示最低优先级。

参考代码如下：

```
void* network_0_demo(void* arg) {
    /* create network 0 */

    /* set priority of network. 0 ~ 255, 0 indicates the lowest priority. */
    unsigned char priority = 10;
    status = vip_set_network(network_0, VIP_NETWORK_PROP_SET_PRIORITY, &priority);
    if (status != VIP_SUCCESS) {
        printf("network 0 set priority %d fail.\n", priority);
        return -1;
    }

    /* prepare network 0 */
    /* network 0 load input */

    /* run network 0 */
    while (count < loop_count)
    {
```

```
        count++;
        status = network_0.network_input_output_set();
        status = network_0.network_run();

        usleep(200*1000);  // 200ms
    }
    return NULL;
}

void* network_1_demo(void* arg) {
    /* create network 1 */

    /* set priority of network. 0 ~ 255, 0 indicates the lowest priority. */
    unsigned char priority = 1;
    status = vip_set_network(network_1, VIP_NETWORK_PROP_SET_PRIORITY, &priority);
    if (status != VIP_SUCCESS) {
        printf("network 1 set priority %d fail.\n", priority);
        return -1;
    }

    /* prepare network 1 */
    /* network 1 load input */

    /* run network 1 */
    while (count < loop_count)
    {
        count++;
        status = network_1.network_input_output_set();
        status = network_1.network_run();
    }
    return NULL;
}

int main(int argc, char** argv) {
    /* NPU init*/

    pthread_t mythread1, mythread2;
    pthread_create(&mythread1, NULL, network_0_demo, (void*)&thread_0_param);
    pthread_create(&mythread2, NULL, network_1_demo, (void*)&thread_1_param);

    pthread_join(mythread1, &thread_result);
    pthread_join(mythread2, &thread_result);

    return ret;
}
```

具体示例，可参考 ai-sdk/examples/multi_thread。

5.3 自定义算子

5.3.1 功能介绍

当平台的算子不满足业务算法要求，可自定义算子。例如：自定义映射表算子。

5.3.2 代码 Demo

demo 代码放置于ai-sdk/examples/custom_lut。

```
├──── model   #模型自定义算子的具体实现，通过PC离线方式编译输出network_binary.nb，对应test目录的custom_lut.nb
│   ├──── build.sh
│   ├──── lut.c
│   ├──── lut.h
│   ├──── makefile.linux
│   ├──── run.sh
│   ├──── user_lut.c
│   └──── user_lut.h
├──── REARDME.md
└──── test   #板端运行资源，可通过vpm_run调用sample.txt运行
    ├──── golden.bin
    ├──── input.txt
    ├──── custom_lut.nb
    └──── sample.txt
```

第一步：基于 C 语言进行算子源码实现。

- lut.c 为主函数入口，建立输入输出 tensor，输入为 uint8 量化格式，输出为 int8 量化格式。
- user_lut.c 为自定义算子实现，通过 user_lut_node 接口调用。

lut 自定义算子实现：通过 vxCreateLUT 创建 lut 表并对其列表进行赋值，输入输出映射表通过 vxTensorTableLookupLayer 接口连接输入输出 tensor 即可。

第二步：编译模型。

① 编译模型，通过./build.sh脚本编译 custom_lut 执行文件。

② 通过./run.sh <platform>运行 custom_lut 执行文件并生成 network_binary.nb 模型文件。

第三步：测试模型。

test 目录中，

- custom_lut.nb 为生成的模型文件（对应第二步生成的 network_binary.nb）
- sample.txt 为测试描述文件，用于 vpm_run 工具测试，具体用法参考《模型运行工具 vpm_run 使用说明》一章。
- input.txt 为输入测试文件，内容为 0~255
- golden.bin 为参考输出文件（需要二进制形式），为-128~127 二进制数值

test 目录和 vpm_run 工具 push 到板端，执行vpm_run -s sample.txt进行测试，输出结果如下：

```
batch i=0, binary name: ./custom_lut.nb
input 0 dim 256 1 1 1, data_format=2, quant_format=2, name=input[0], scale=1.000000, zero_point=0
ouput 0 dim 256 1 1 1, data_format=3, name=output[0], scale=1.000000, zero_point=0
nbg name=./custom_lutt.nb
create network 1: 1067 us.
```

```
memory pool size=0byte
read golden file ./golden.bin
input 0 name: ./input.txt
prepare network 0: 1826 us.
batch: 0, loop count: 1
start to run network=./custom_lut.nb
run time for this network 0: 765 us.
run network done...
profile inference time=17us, cycle=-93
******* golden TOP5 ********
 --- Top5 ---
255: 127.000000
254: 126.000000
253: 125.000000
252: 124.000000
251: 123.000000
******* nb TOP5 ********
 --- Top5 ---
255: 127.000000
254: 126.000000
253: 125.000000
252: 124.000000
251: 123.000000
   Test output 0 passed.

destroy teset resource batch_count=1
```

结果符合预期：Test output 0 passed。

5.4 共享权重

在动态 shape 模型的场景，如 OCR 识别，让不同输入的动态 shape 模型共用一份权重数据，可节约内存，适用于 DDR 资源紧张的低内存方案。

一个动态 shape 模型，固定为几个不同 shape 的模型，这些模型都会加载到内存。这些模型共用一份权重数据，但输入、输出 shape 不同，以节约权重占用的内存空间。使用 nbstrip 工具裁剪 nb 的权重文件，减少文件的存储空间与使用时的文件读入空间，减少峰值内存。本方法适合需求多个 shape 输入且权重文件占比较大的模型。

具体使用示例可参考ai-sdk/examples/share_weight，详细说明可参考示例目录的README-CN.md。

5.5 NBG 分析工具 nbinfo 使用说明

5.5.1 功能介绍

nbinfo 是用于分析 NBG 模型文件的 PC 端命令行工具，运行于 Linux 平台。

nbinfo 针对 NBG 模型的分析内容包括：

1. 模型输入/输出信息，包含 tensor 形状，量化参数，内存等信息；
2. 模型运行时内存分析，包含总的运存大小，系统内存大小，feature map 大小统计等等；
3. 模型各层统计分析，包含总层数，各层的运算算子类型，各层的量化信息等等。

5.5.2 获取方式

二进制版本的 nbinfo 工具放在ai-sdk/tools/nbinfo（docker 环境已预置 nbinfo 工具）。

若需要 nbinfo 工具源码请联系我们，并在 Linux 环境下按照以下步骤编译：

```
#解压
tar xvf SW_NBInfo_X.X.X.tgz

#进入NBGParser目录，编译NBGParser库，生成libNBGParser.a
cd nbinfo
cd NBGParser
make STATIC_LINK=1 install

#退回到nbinfo顶层目录，生成nbinfo可执行程序
cd ..
make STATIC_LINK=1
```

5.5.3 使用说明

接下来以一个 sample 模型为例，讲解 nbinfo 的使用方式。执行指令 nbinfo --help 查看使用信息：

```
./nbinfo --help
```

输出信息如下：

```
Verisilicon NBInfo version 1.2.24, version=0x00010218

Usage: nbinfo <options> <nbfilepath>

Options:
-a                  print [a]ll info
-b                  print [b]rief summary
-n                  print [n]bg header
-l                  print [l]ayers
-o                  print [o]perations
-in                 print [in]put
-out                print [out]put
-m                  print [m]emory
-m -d               print [m]emory, shows detail memory profile
-m -f               print [m]emory, for creating network from flash project
-h                  [h]elp
```

5.5.3.1 查看模型全部信息

当需要查看模型全部信息，可执行 nbinfo -a sample_network.nb 查看：

```
./nbinfo -a ./sample_network.nb |more
```

输出结果如下：

```
nbg file name ./sample_network.nb
****************************************************************************
Overall Info
****************************************************************************
Network Name:                    ps_test_uint8_NCHW
Version:                         0x1001e
Target:                          0x10000016
Core Count:                      1
AXI SRAM Size:                   0
VIP SRAM Size:                   0x12f00
Memory Pool Size (bytes):        0
Memory Pool Alignment:           64
Layer Count:                     4
Operation Count:                 5
Input Count:                     3
Output Count                     1
CheckSum Value:                  0x4502a47e
****************************************************************************

****************************************************************************
Overall Info
****************************************************************************
device id:                       0
core count:                      1
nn command size:                 192
tp command size:                 192
.....
.....
```

5.5.3.2 查看模型输入信息

当需要查看模型输入信息时，可执行 nbinfo -in sample_network.nb 查看：

```
./nbinfo -in ./sample_network.nb |more
```

输出结果如下：

```
****************************************************************************
Input Table
****************************************************************************
Input 0
Dim Count:                       4
Size of Dim[0]:                  1920
Size of Dim[1]:                  1080
Size of Dim[2]:                  1
Size of Dim[3]:                  1
Data Format:                     NBG_BUFFER_FORMAT_UINT8
Data Type:                       NBG_BUFFER_TYPE_TENSOR
Quantization Format:             NBG_BUFFER_QUANTIZE_NONE
Fixed Point Pos:                 0
TF Scale:                        1.000000
TF Zeropoint:                    0
Memory Size (bytes):             2073600
```

```
input name:                             input[0]
********************************************************************************
Input 1
Dim Count:                                 1
Size of Dim[0]:                            1
Data Format:                    NBG_BUFFER_FORMAT_INT32
Data Type:                      NBG_BUFFER_TYPE_SCALAR
Quantization Format:                NBG_BUFFER_QUANTIZE_NONE
Fixed Point Pos:                           0
TF Scale:                           0.000000
TF Zeropoint:                              0
Memory Size (bytes):                      64
input name:                           input[1]
********************************************************************************
Input 2
Dim Count:                                 1
Size of Dim[0]:                            1
Data Format:                    NBG_BUFFER_FORMAT_INT32
Data Type:                      NBG_BUFFER_TYPE_SCALAR
Quantization Format:                NBG_BUFFER_QUANTIZE_NONE
Fixed Point Pos:                           0
TF Scale:                           0.000000
TF Zeropoint:                              0
Memory Size (bytes):                      64
input name:                           input[2]
********************************************************************************
```

可以看到模型的三个输入的 shape、量化格式、Scale、Zeropoint 等参数。

5.5.3.3 查看模型输出信息

当需要查看模型输出信息时，可执行 nbinfo -out sample_network.nb 查看：

```
./nbinfo -out ./sample_network.nb |more
```

输出结果如下：

```
********************************************************************************
Output Table
********************************************************************************
Output 0
Dim Count:                                 4
Size of Dim[0]:                          128
Size of Dim[1]:                          128
Size of Dim[2]:                            1
Size of Dim[3]:                            1
Data Format:                     NBG_BUFFER_FORMAT_UINT8
Data Type:                       NBG_BUFFER_TYPE_TENSOR
Quantization Format:             NBG_BUFFER_QUANTIZE_AFFINE_ASYMMETRIC
Fixed Point Pos:                           0
TF Scale:                           1.000000
TF Zeropoint:                              0
Memory Size (bytes):                   16384
output name:                        uid_1_out_0
********************************************************************************
```

可以看到模型输出的 shape、量化格式、Scale、Zeropoint 等参数。

5.5.3.4 查看模型内存消耗信息

当需要查看运行时内存信息时，可执行 nbinfo -d -m sample_network.nb 查看：

```
./nbinfo -d -m ./sample_network.nb |more
```

输出结果如下：

```
*******************************************************************************
Input and memory_pool overlap info
*******************************************************************************
Summary:
    memory pool size is 0, don't need the reuse optimization
*******************************************************************************

*******************************************************************************
Memory Info
*******************************************************************************
Total Video Memory (bytes):                 2113280
  -- Memory Pool (bytes):                   0
  -- Total Input Memory (bytes):            2073728
  -- Total Output Memory (bytes):             16384
  -- Total Operation Memory (bytes):           2624
    -- Total Read Only(Coeff) Memory (bytes):      576
    -- Total Command buffer (bytes):            1792
      -- Total Load States (bytes):            960
      -- Total NN, TP and SP instruction (bytes):     576
      -- Total PPU instruction (bytes):          256
    -- Dynamic Input or Output (bytes):          256
  -- Video memory heap node reserved (bytes):          20480
*******************************************************************************
Total System Memory (bytes):                31672
  -- NBG header Size (bytes):               1992
  -- Allocate memory size (bytes):            29680
*******************************************************************************
```

技巧

V85x、R853s 平台用Total Video Memory + Total System Memory作为初始化 vip_init 函数的输入参数，可起到节约内存以及减少内存申请时间的作用。可参考《模型内存问题》一节。

一个模型的主要消耗内存如下：

- memory pool size：需要分配的内存池大小。
- Total Video Memory：需要分配的 video memory 大小，统计的是当前模型运行时内存的消耗量。此内存会从最初申请的内存堆（V85x、R853s 平台在初始化时申请，其他平台在驱动挂载时申请）中分配，不足时动态申请。
- Total System Memory：需要分配的 system memory 大小，主要是用于与 CPU 的交互；

说明

各块内存的具体作用如下：

- **Memory Pool：需要分配的内存池大小，这段内存相当于 NPU 模块所需的运行内存，它的值是和 nbinfo 统计的一样准确的。**
- **Total Input Memory：Input buffer（输入缓冲区）所需的内存大小**
- **Total Output Memory：Output buffer（输出缓冲区）所需的内存大小**
- **Total Operation Memory：由 Command buffer（传递 command）、Read Only Memory（用于存储权重参数或者偏置项）和 Dynamic Input or Output 组成。**
- **Video memory heap node reserved：Video memory 的内存堆 node 节点的保存，这个值一般都是 20k。**
- **NBG header：存储 NBG 文件的头部信息。**
- **Allocate memory：分配内存空间用于存储数据，是用于存储从 NBG 文件中读取的数据或者其他需要在运行时分配的数据。**

5.5.3.5 查看模型各层信息

当需要查看端侧最终运行的模型的逐层 layers 信息时，可执行 nbinfo -l sample_network.nb 查看：

```
./nbinfo -l sample_network.nb
```

输出结果如下：

```
*******************************************************************************
Layer Table
*******************************************************************************
Layer Name:                com.vivantecorp.extension.evis.pre_process_gray_copy_U8toU8
ID:                                0
UID:                           10000
Operation Count:                   1
*******************************************************************************
Layer Name:                ConvolutionReluPoolingLayer2
ID:                                1
UID:                               5
Operation Count:                   1
*******************************************************************************
Layer Name:                TensorTranspose
ID:                                2
UID:                               3
Operation Count:                   1
*******************************************************************************
Layer Name:                ConvolutionReluPoolingLayer2
ID:                                3
UID:                               1
Operation Count:                   1
*******************************************************************************
```

可以看到，演示模型最终是 4 层，逐层信息中包括每层的名字。

5.5.3.6 查看模型各操作信息

当需要查看模型在端侧运行所需的操作和各操作所需资源情况，可执行 nbinfo -o sample_network.nb 查看：

```
./nbinfo -o sample_network.nb
```

输出结果如下：

```
********************************************************************************
Operation Table
********************************************************************************
Operation Type:                     Init
Index:                             65535
Layer ID:                          65535
Absolute OP ID:                   4294967295
Index of First Patch:                 0
Number of Patches:                    1
Command Size (bytes):                40
Instr Size (bytes):                   0
********************************************************************************
Operation Type:                   SHADER
Index:                             0
Layer ID:                          0
Absolute OP ID:                    0
Index of First Patch:             17
Number of Patches:                 7
Command Size (bytes):            688
Instr Size (bytes):               256
********************************************************************************
Operation Type:                     NN
Index:                             0
Layer ID:                          1
Absolute OP ID:                    1
Index of First Patch:              1
Number of Patches:                 6
Command Size (bytes):             24
Instr Size (bytes):               192
Coeff header Size (bytes):      0x180
********************************************************************************
Operation Type:                     TP
Index:                             0
Layer ID:                          2
Absolute OP ID:                    2
Index of First Patch:              7
Number of Patches:                 5
Command Size (bytes):             40
Instr Size (bytes):               128
********************************************************************************
Operation Type:                     NN
Index:                             1
Layer ID:                          3
Absolute OP ID:                    3
Index of First Patch:             12
Number of Patches:                 5
Command Size (bytes):             24
Instr Size (bytes):               192
Coeff header Size (bytes):       0xc0
********************************************************************************
```

可以看到，

1. 演示模型是 4 层，总有 5 步操作，其中第一个操作是初始化。

2. Operation Type类型实为算子操作在哪个硬件模块。SHADER 对应的是 PPU，PPU、NN、TP 硬件模块说明详见《系统架构说明》。
3. Command Size、 Instr Size和Coeff header Size这些参数主要用于驱动运行时 patch 地址。

5.6 模型运行工具 vpm_run 使用说明

5.6.1 功能介绍

vpm_run 是一个基于 VIPLite 驱动的端侧程序，主要用于快速验证模型文件和测试。

同时，它也是模板，源码开放。参考 vpm_run 的流程，用户可以开发自己的基于 VIPLite API 的 AI 应用程序。它的特点包括以下几点：

- vpm_run 可作为一个通用的模型运行环境，程序不需要修改，可运行任何部署正确的 NBG 模型文件。
- vpm_run 基于 VIPLite 网络层 API，程序短小精悍。
- vpm_run 具备默认的后处理程序 TOP5，如果不满足你的算法要求，可以自行扩展。

5.6.2 源码获取与编译

vpm_run 工具源码放在ai-sdk/examples/vpm_run，重点文件说明如下：

- vpm_run.c，主程序，可以通读其代码，了解 NPU runtime 的开发使用方式。
- makefile.linux，Linux 系统的编译构建 makefile 文件。
- Android.mk，Android 系统的编译构建文件。
- sample.txt，运行的参数配置文件，vpm_run 的输入输出数据均来自于此文件的描述。
- build_tst_sample.sh，Linux 系统用于编译工具链。

1）Linux 编译 vpm_run

三种方法可编译：

方法一是修改 makefile.linux 文件，直接编译

首先，修改 makefile.linux 文件。需增加 CC 变量和修改 INCLUDE 变量、LIBS 变量。CC 变量指向 SDK 中的工具链路径，INCLUDE 变量和 LIBS 变量指向 SDK 中的 VIPLite 用户态头文件和中间库的目录路径。参考如下：

```
include makefile.linux.def

#增加CC变量，表示工具链路径
#以V85x、R853平台的工具链为示例
```

```
CC=/home/xxx/tina-v85x/prebuilt/gcc/linux-x86/arm/toolchain-sunxi-musl/toolchain/bin/arm-openwrt-linux-gcc
#修改SDK_DIR、INCLUDE和LIBS变量，表示VIPLite 用户态头文件和中间库的目录路径
#以V85x、R853平台的中间件为示例
#SDK_DIR=/home/xxx/ai-sdk/viplite/
#INCLUDE += -I$(SDK_DIR)/include
#LIBS   += -L $(SDK_DIR)/arm -Wl,-rpath-link,$(SDK_DIR)/arm
SDK_DIR=/home/xxx/tina-v85x/package/allwinner/libsdk-viplite-driver/sdk_release/
INCLUDE += -I$(SDK_DIR)/sdk
LIBS   += -L $(SDK_DIR)/ -Wl,-rpath-link,$(SDK_DIR)/
OUT_DIR ?= $(PWD)/out

CFLAGS += -g -O0 -DDEBUG -D_DEBUG
CFLAGS += $(INCLUDE)


ifeq ($(SAVE_OUTPUT_TXT_FILE),1)
 CFLAGS += -DSAVE_OUTPUT_TXT_FILE
endif

target = $(OUT_DIR)/vpm_run
$(target):
  @mkdir -p $(OUT_DIR)
  @$(CC) $(CFLAGS) -g -o $@ vpm_run.c $(LIBS) -l VIPlite -l VIPuser -lm
  @cp sample.txt $(OUT_DIR)/

.PHONY: clean
clean:
  @rm -rf $(OUT_DIR)
```

然后，执行编译命令make -f makefile.linux。编译输出如下：

```
~/tina/ai-sdk/example/vpm_run$ make -f makefile.linux
arm-openwrt-linux-muslgnueabi-gcc.bin: warning: environment variable 'STAGING_DIR' not defined
arm-openwrt-linux-muslgnueabi-gcc.bin: warning: environment variable 'STAGING_DIR' not defined
arm-openwrt-linux-muslgnueabi-gcc.bin: warning: environment variable 'STAGING_DIR' not defined
```

编译后的 vpm_run 可执行程序存放在 out 目录中，信息如下：

```
~/tina/ai-sdk/example/vpm_run$ file out/vpm_run
out/vpm_run: ELF 32-bit LSB  executable, ARM, EABI5 version 1 (SYSV), dynamically linked (uses shared libs), not
      stripped
```

技巧

删除编译结果的命令为make -f makefile.linux clean。

方法二是修改 build_tst_sample.sh，脚本编译

首先，修改 build_tst_sample.sh 和 makefile.linux 文件。build_tst_sample.sh 中，修改 TOOLCHAIN 和 CROSS_COMPILE 变量，指向 SDK 中的工具链路径；makefile.linux 中修改 INCLUDE 变量、LIBS 变量，指向 SDK 中的 VIPLite 用户态头文件和中间库的目录路径。参考如下：

```
... ...
tinav85x)
  export ARCH_TYPE=arm
  export CPU_TYPE=cortex-a7
  export CPU_ARCH=armv7-a
  export FIXED_ARCH_TYPE=arm-linux-gnueabi
#修改TOOLCHAIN和CROSS_COMPILE，表示工具链路径
```

```
#以V85x、R853平台的工具链为示例
  export TOOLCHAIN=/home/xxx/tina-v85x/prebuilt/gcc/linux-x86/arm/toolchain-sunxi-musl/toolchain/
  export CROSS_COMPILE=/home/xxx/tina-v85x/prebuilt/gcc/linux-x86/arm/toolchain-sunxi-musl/toolchain/bin/arm-
      openwrt-linux-muslgnueabi-
  export LIB_DIR=$TOOLCHAIN/arm-openwrt-linux-muslgnueabi/lib
  export USE_LINUX_PLATFORM_DEVICE=1
  export USE_LINUX_RESERVE_MEM=0
  export PLATFORM_CONFIG=allwinner
;;
.....
```

然后，执行编译命令./build_tst_sample.sh <BUILD_BOARD>。

方法三是 SDK 编译

MR527/AI985/MR536 SDK 中，导入环境变量并选择方案后，按照下面的方式进行配置编译：

```
make menuconfig
Allwinner  --->
Vision  --->
  <*> ai-sdk-viplite..... allwinner npu viplite framework  --->
```

选择后保存退出，编译固件，烧录固件后，即可使用 vpm_run 命令，相关配置模型等放在设备端的 /etc/npu/ 目录下。

T527 AIOT Linux、T536 SDK 中，导入环境变量并选择方案后，按照下面的方式进行配置编译：

```
source .buildconfig
./build.sh buildroot_menuconfig
Target packages  --->
allwinner platform private package select  --->
ai-sdk  --->
 [*] ai sdk
 [*]   ai sdk viplite lib and vpm_run
 [*]    ai sdk viplite sample
cd buildroot/buildroot-xxxxxx
make ai-sdk-rebuild
#输出文件在~/sdk_dir/out/xxx/xxx/buildroot/buildroot/target/etc/npu/vpm_run
#安装到设备在/etc/npu/vpm_run
```

2）Android 编译 vpm_run

在源码目录下，mm 即可。编译后的 vpm_run 可执行程序存放在 ~/out/vendor/bin/vpm_run。

⚠ 注意

vpm_run 等应用程序编译要和中间库使用相同工具链，否则运行多次后可能出现异常。

5.6.3 参数配置文件说明

```
[network]
./network_binary.nb
[input]
#多输入模型，则配置多个输入文件；单输入模型，则配置1个输入文件。
./input_0.dat
./input_1.dat
[golden]
#可选
./int8_output_003.dat
[output]
#可选
./output_0.dat
```

sample.txt 文件的示例如上所示，主要标签介绍如下：

- [network]：NBG 文件的路径。
- [input] : 输入数据的路径。输入数据的格式支持两种：Acuity Toolkits 工具 inference 阶段生成的 input.tensor 文件；IDE 工具仿真阶段生成的 input dat 文件。
- [golden] ： 可选的。golden 数据的路径。如果存在 golden 标签，将其作为 golden 数据和 vpm_run 运行输出的 tensor 做 binary 比对，比对一致打印 pass，比对不一致将退出 vpm_run。
- [output] : 可选的。输出数据的保存路径。

⚠ 注意

1. **请确保 sample.txt 的换行符为 Unix(LF) 而不是 Windows(CR LF)，否则运行失败。**
2. **请确保 sample.txt 最后多空出一行，以免读取文件时发生错误。**

5.6.4 运行演示

sample.txt 文件按照如下修改。多组标签表示运行多个网络，下文则执行三个 NBG 网络文件。第一个模型是单输入模型；第二个模型是多输入模型，且保存输出文件；第三个模型保存输出文件，且比对 tensor。

```
[network]
./ network_binary_single.nb
[input]
./ input.bin

[network]
./ network_binary_multi.nb
[input]
./ input1.tensor
./ input2.tensor
[output]
./ output_1.dat

[network]
```

```
./ network_binary_multi.nb
[input]
./ input1.tensor
./ input2.tensor
[golden]
./ output_golden.dat
[output]
./ output_2.dat
```

端侧执行以下命令运行：

```
./vpm_run -s sample.txt
```

运行结果如下所示：

```
init vip lite, driver version=0x00010d00...
vip lite init OK.

#共三个测试模型
init test resources, task_count: 3 ...
create/prepare networks ...
#第一个模型
task i=0, binary name: ./network_binary_single.nb
input 0 dim 224 224 3 1, data_format=2, quant_format=2, name=input[0], scale=1.000000, zero_point=0
ouput 0 dim 1000 1 0 0, data_format=2, name=uid_1_out_0, scale=0.142842, zero_point=110
nbg name=./network_binary_single.nb
create network 0: 7170 us.
memory pool size=502784byte
golden file count=0
input 0 name: ./input.bin
prepare network 0: 9968 us.
#第二个模型
task i=1, binary name: ./network_binary_multi.nb
input 0 dim 3 160 1 1, data_format=2, quant_format=2, name=input[0], scale=0.996078, zero_point=0
input 1 dim 256 3 1 1, data_format=2, quant_format=2, name=input[1], scale=0.996078, zero_point=0
ouput 0 dim 256 160 1 1, data_format=1, name=uid_1_out_0, none-quant
nbg name=./network_binary_multi.nb
create network 1: 1612 us.
memory pool size=0byte
golden file count=0
input 0 name: ./input1.tensor
input 1 name: ./input2.tensor
prepare network 1: 3943 us.
#第三个模型，有一个golden文件
task i=2, binary name: ./network_binary_multi.nb
input 0 dim 3 160 1 1, data_format=2, quant_format=2, name=input[0], scale=0.996078, zero_point=0
input 1 dim 256 3 1 1, data_format=2, quant_format=2, name=input[1], scale=0.996078, zero_point=0
ouput 0 dim 256 160 1 1, data_format=1, name=uid_1_out_0, none-quant
nbg name=./network_binary_multi.nb
create network 2: 20452 us.
memory pool size=0byte
golden file count=1
0 read golden file ./output_golden.dat
input 0 name: ./input1.tensor
input 1 name: ./input2.tensor
prepare network 2: 5032 us.
task: 0, loop count: 1
#开始运行
start to run network=./network_binary_single.nb
run time for this network 0: 6663 us.
```

```
run network done...
profile inference time=5823us, cycle=3480119
******* nb TOP5 ********
 --- Top5 ---
505: 19.283667
849: 18.569458
504: 15.998302
725: 13.141462
968: 12.998620
task: 1, loop count: 1
start to run network=./network_binary_multi.nb
run time for this network 1: 927 us.
run network done...
profile inference time=72us, cycle=27624

task: 2, loop count: 1
start to run network=./network_binary_multi.nb
run time for this network 2: 1081 us.
run network done...
profile inference time=68us, cycle=27672

 #golden 输出对比通过
   Test output 0 passed.

destroy teset resource task_count=3
```

可执行多次，以下命令表示执行 100 次，每次将 sample.txt 中的所有模型运行一遍。

```
./vpm_run -s sample.txt -l 100
```

5.7 VivantelIDE 工具使用说明

5.7.1 启动 IDE

启动 IDE 的命令如下：

```
~/VivanteIDE5.7.2/ide/vivanteide5.7.2
```

启动时，首先选择一个工作目录用于保存仿真工程：

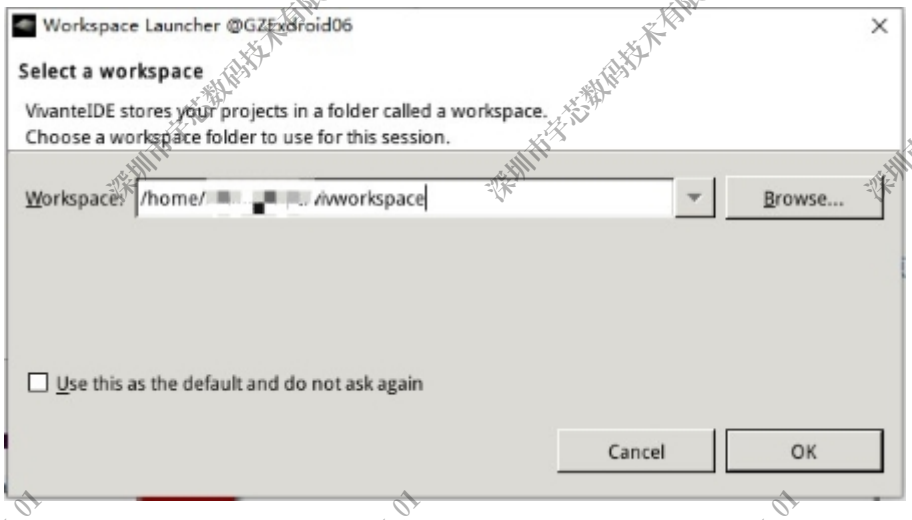

图 5-2: IDE 工具选择工作区

5.7.2 导入工程

本文以 Mobilenet_V2 为例，我们先将模型进行转换操作生成对应工程目录，参考《NPU 使用说明》一章的《模型导出》一节。

随后打开 IDE 工具，选择 File->Import->General 选项卡->Existing Projects into Workspace

说明

1. 仿真工程目录为部署目录下的 wksp/MobileNetV2_uint8/工程，而非 wksp/MobileNetV2_uint8_nbg_unify/

2. 仿真参数既可以使用图像，也可以使用 inference 阶段生成的 input tensor.

打开 Browse 选择仿真工程：

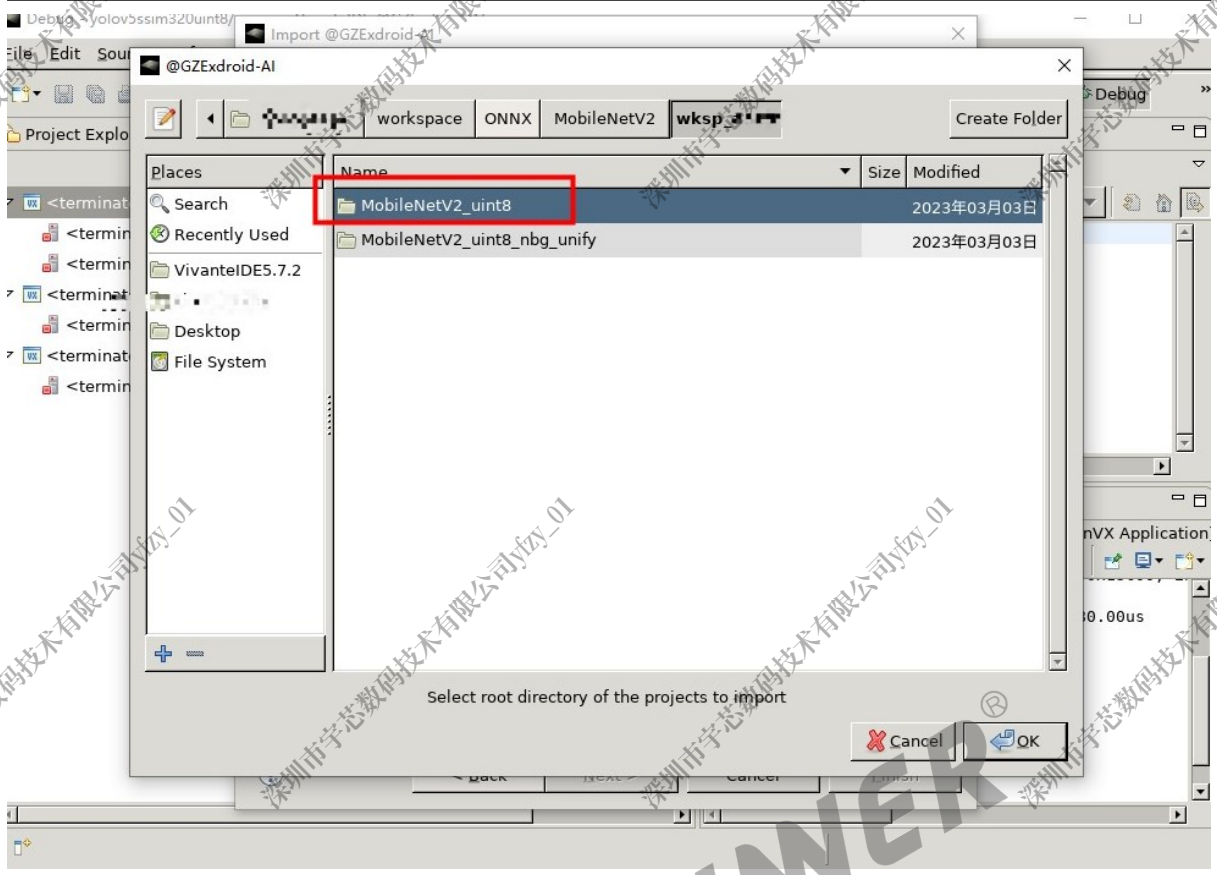

图 5-3: IDE 工具 import 项目

之后选择模型导出阶段创建的工程目录 wksp/MobileNetV2_uint8/工程：

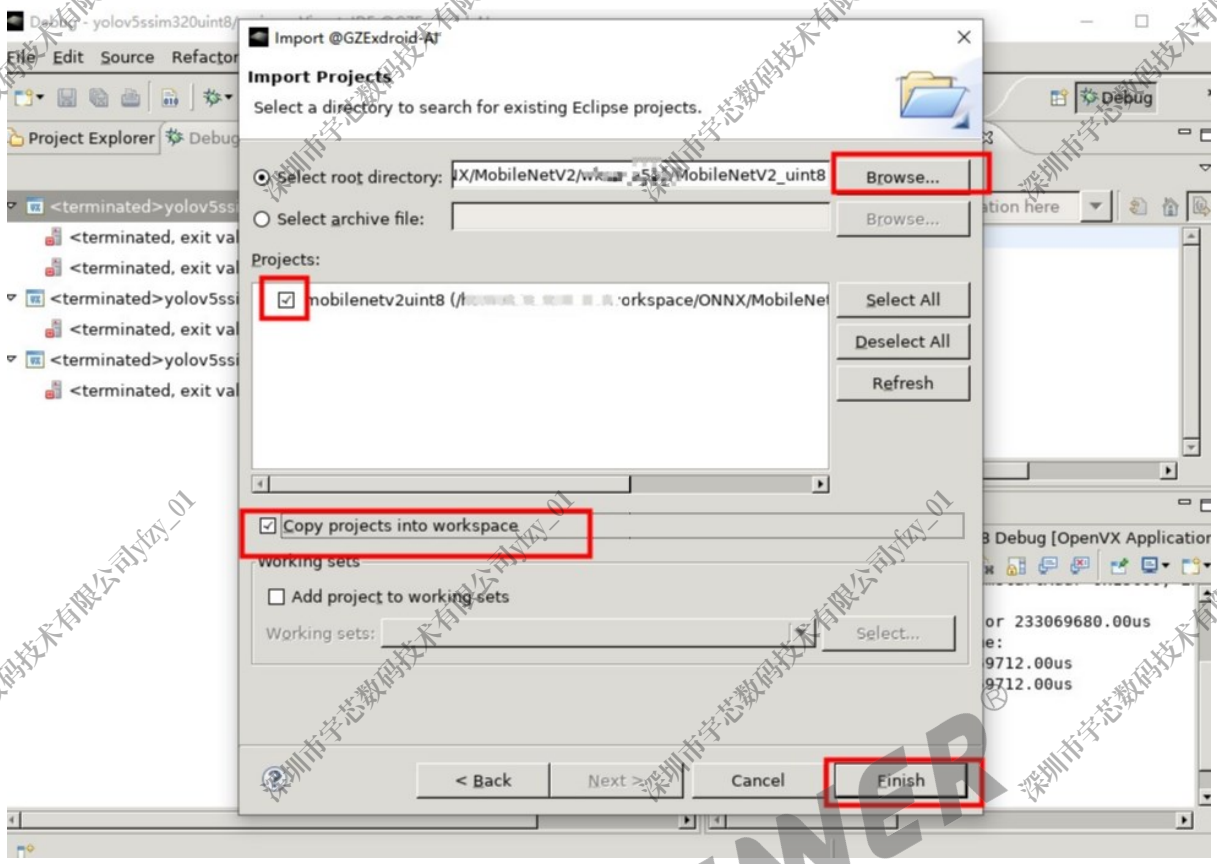

图 5-4: IDE 工具选择工程

说明

建议选中 Copy projects into workspace, 这样我们的仿真工程将会拷贝一份到 IDE 工作空间中，保证与导出空间隔离

结束导入过程，导入后的工作空间如下所示：

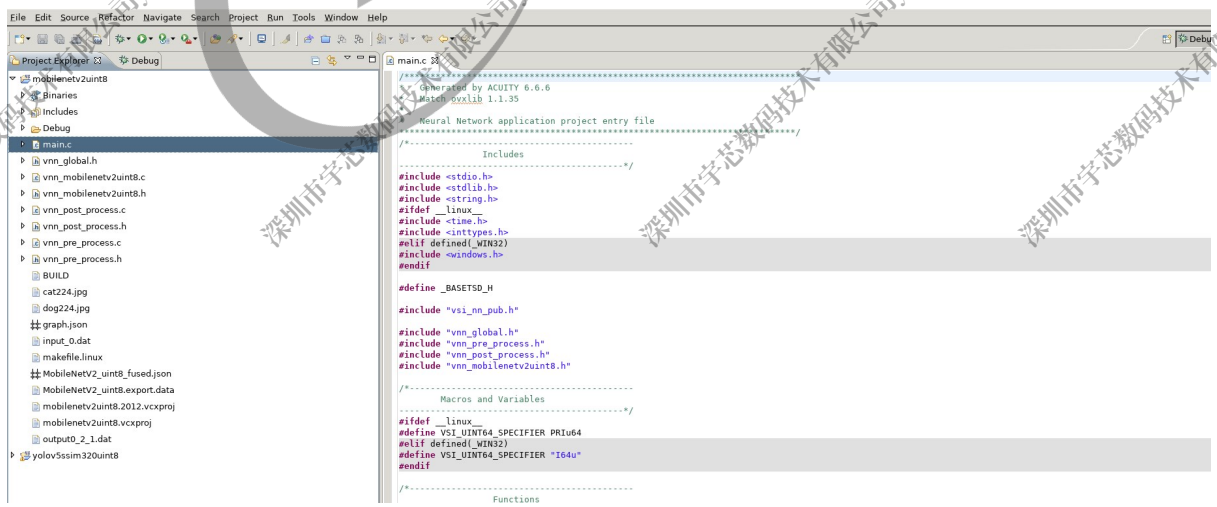

图 5-5: IDE 工具界面

5.7.3 编译工程

执行菜单命令 Project->Build All，先将仿真工程进行编译工作：

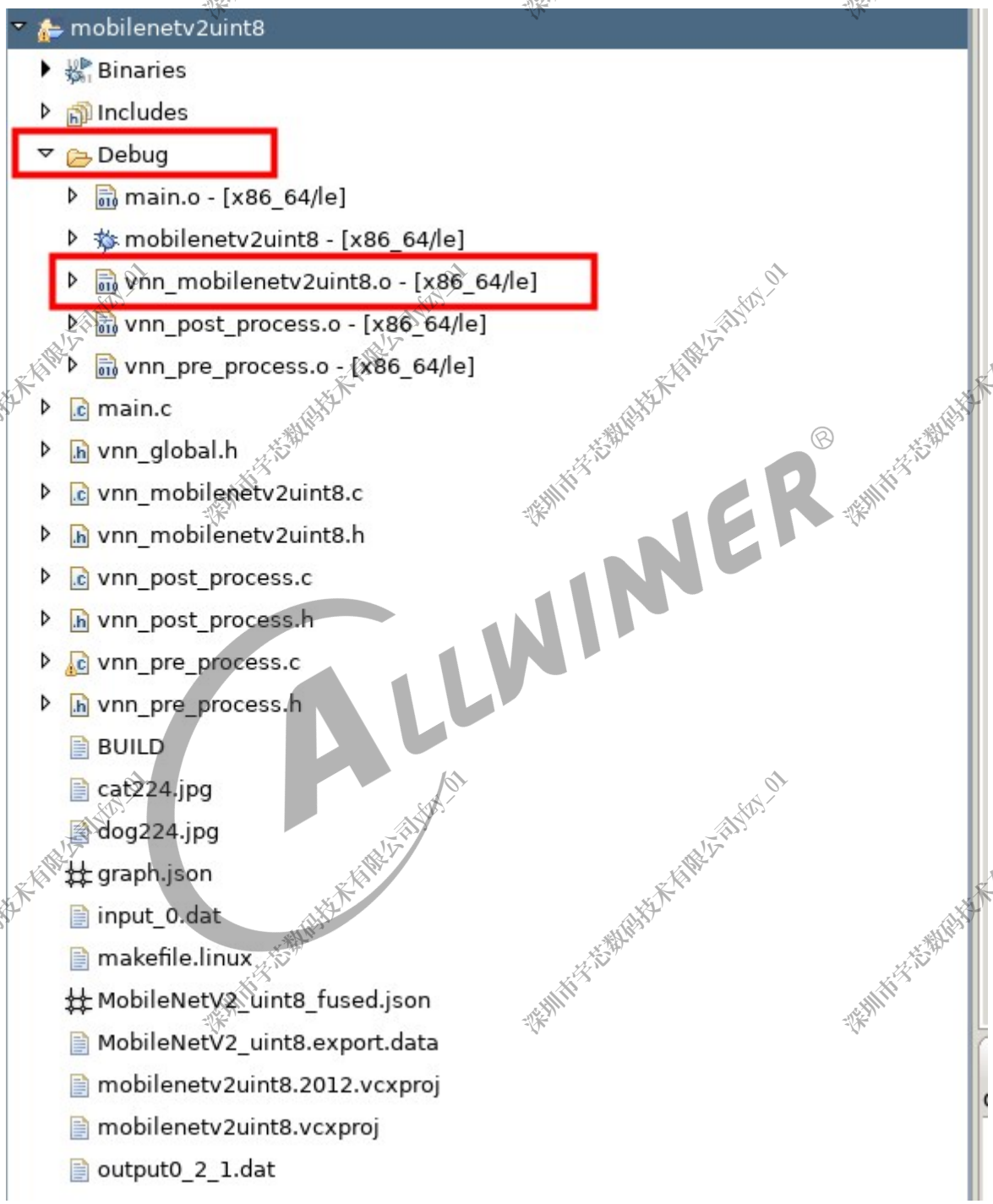

图 5-6: IDE 工具编译工程

编译成功后会生成 Debug 目录，其中有这些可执行文件。

5.7.4 模型仿真

5.7.4.1 配置仿真参数

执行菜单命令 Run->Debug Configurations...，在选项卡中，双击 OpenVX Application，即可出现下图的输出，默认情况下 Search Project 和 Browser 按钮窗口会被正确设置成如下图的样子，如果没有，请按照上面编译的结果正确选择工程和应用路径。

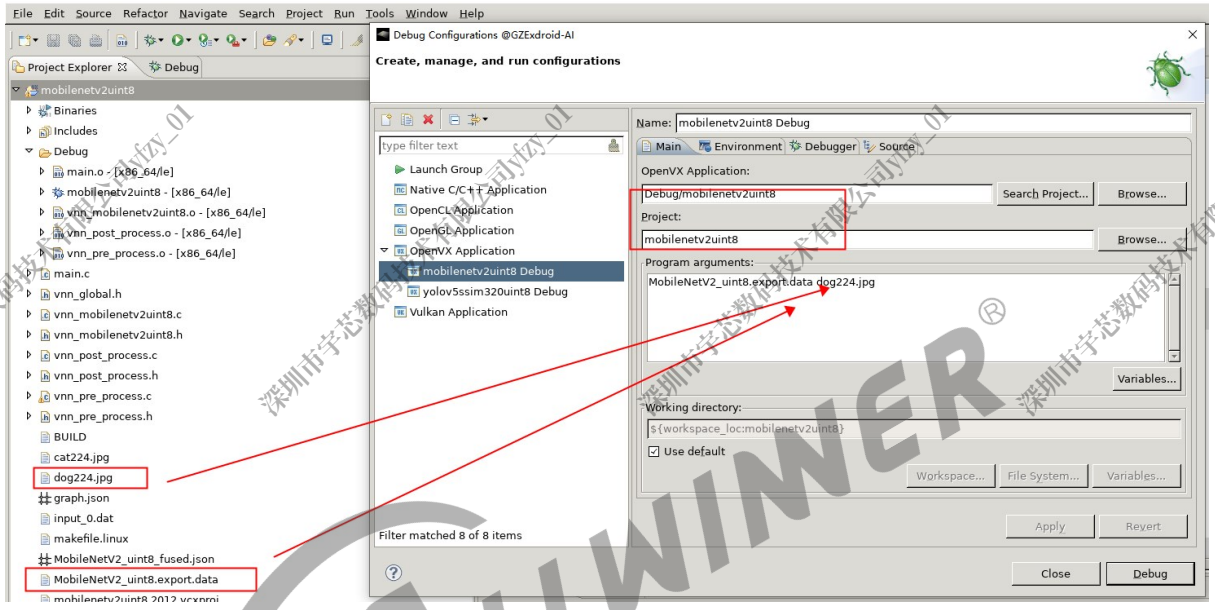

图 5-7: IDE 工具设置仿真参数 1

说明

其中 MobileNetV2_uint8.export.data 是量化权重，dog224.jpg 为 data 中文件，它们是在模型导出阶段生成在 wksp/MobileNetV2_uint8/ 工程中的，工程导入阶段已经自动拷贝到 IDE 仿真工程下，不需要手工拷贝。

然后，需要在 Debugger 中配置 Target 参数，选择 Target 为：VIP9000NANOSI_PLUS_PID0X10000016，如下图所示。

说明

V85x、R853s 平台设为 VIP9000PICO_PID0XEE；MR527、A527 平台设为 VIP9000NANOSI_PLUS_PID0X10000016。

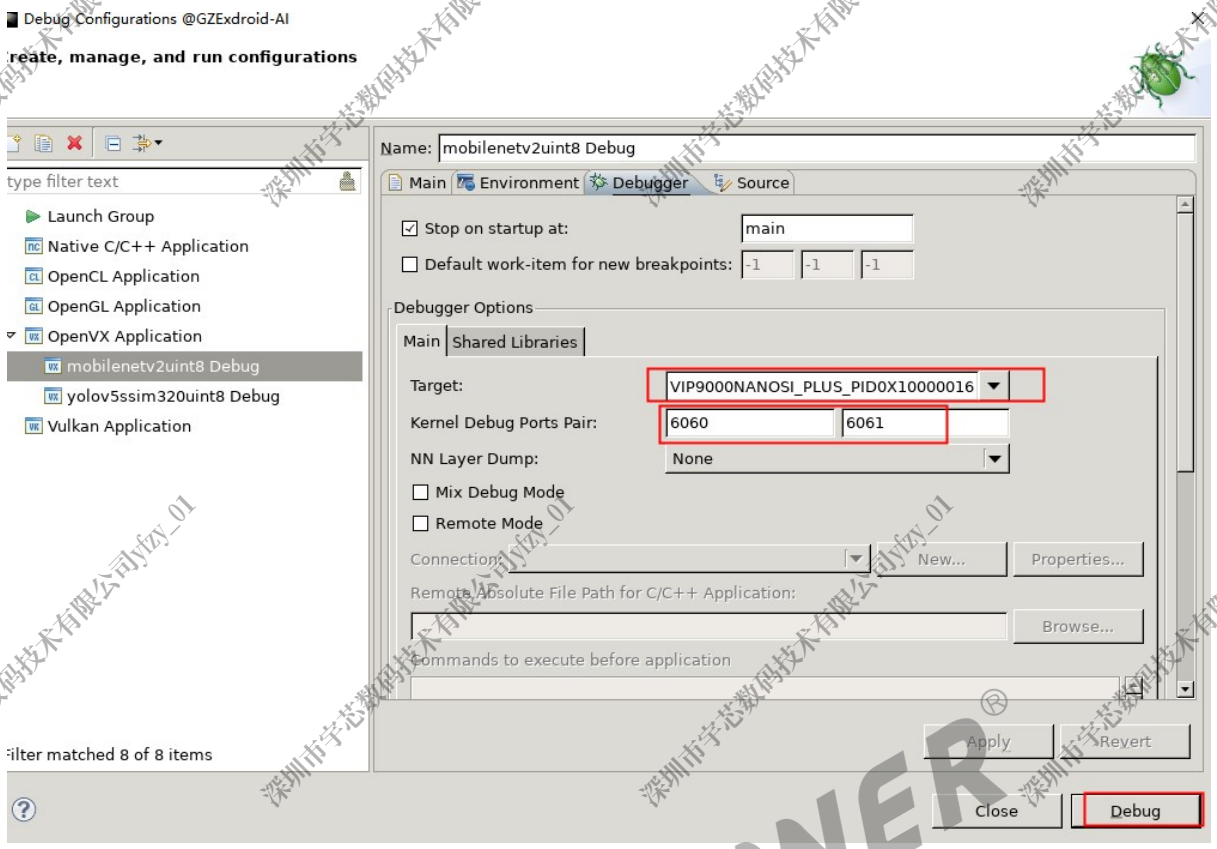

图 5-8: IDE 工具设置仿真参数 2

说明

如果运行出错可以将 Kerne lDebug Ports Pair 的值改为 7000。

输入除了图片，也可以是模型推理阶段生成的输入 tensor, 仿真程序会根据后缀名自动运行到不同的处理分支，保证处理结果都是对的。

点击 Apply, 之后就可以开始正式仿真了。

5.7.4.2 开始仿真

点击工具栏 Run 按钮，弹出对话框，直接点击 Run 触发仿真：

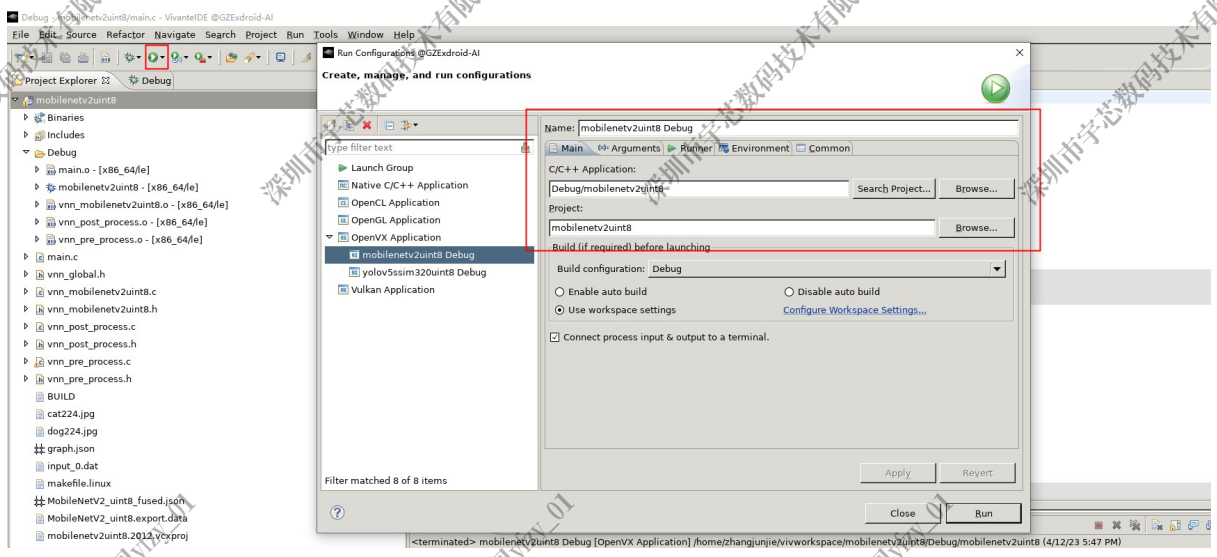

图 5-9: IDE 工具仿真配置

点击运行后将仿真执行该模型，输出如下, 根据下面控制台的输出可以看到，我们使用的参数图像是 dog224.jpg，而仿真输出的 top2 结果表明，推理结果为 dog ，符合预期。

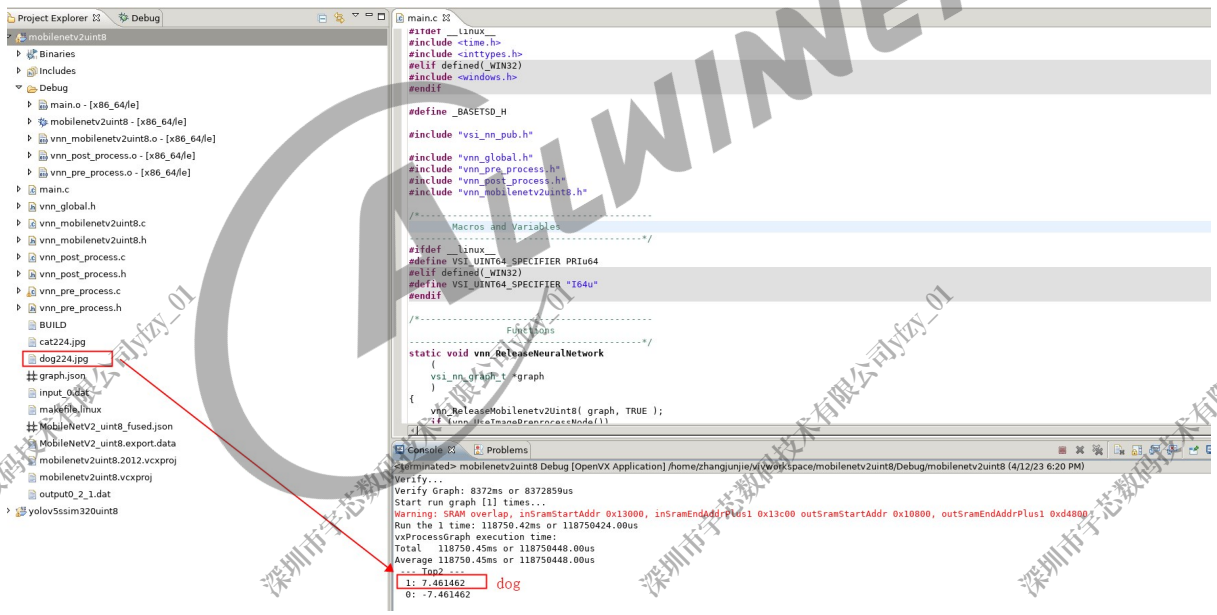

图 5-10: IDE 工具仿真 dog

当我们给的参数图像是 cat224.jpg 时，仿真输出的 top2 结果表明，推理结果为 cat ，符合预期。

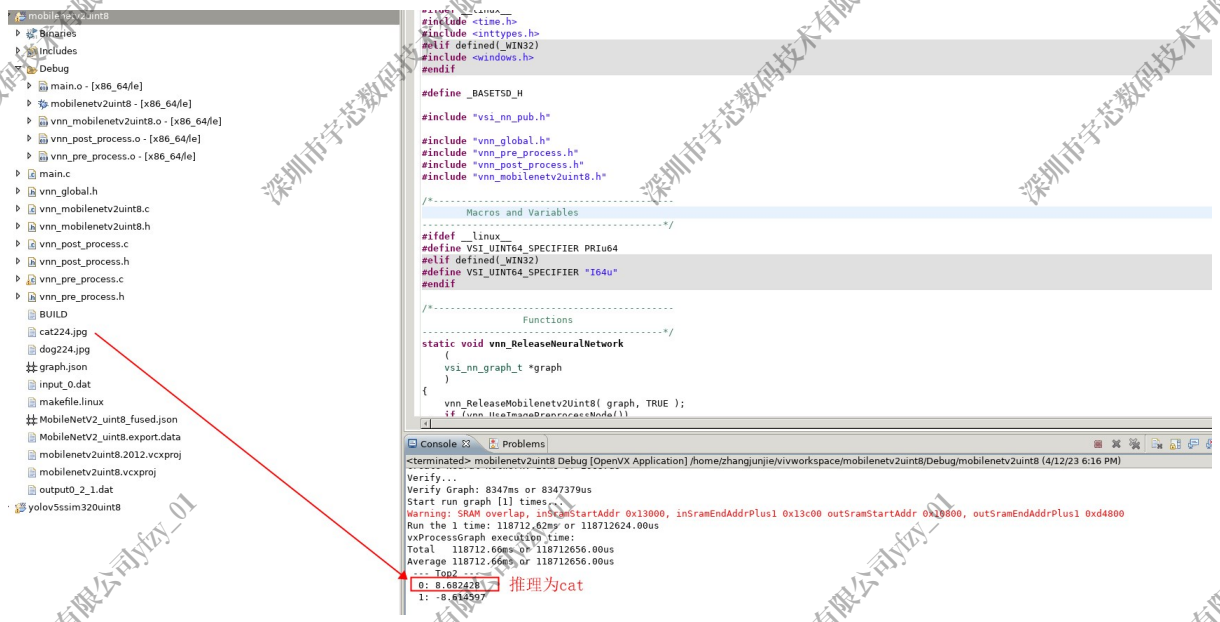

图 5-11: IDE 工具仿真 cat

查看目录可以发现，仿真过后会生成两个.dat 格式的二进制 tensor 文件，分别是输入 tensor 和输出 tensor，我们可以将 tensor 对图像做后处理，并且和 inference 阶段生成的 tensor 文件做相似度对比。

5.7.5 模型 Profile

模型 profile 可以帮助分析网络的整体运行效率，带宽，帧率以及各层的处理性能，是分析算法精度，性能瓶颈等问题的利器。

说明

模型实际性能数据请以板端部署实际推理耗时为主，IDE 工具可作为参考。

点击工具栏运行旁边的 Profile 按钮即可触发 Profile 操作，同样在选项卡中选择 Profile 按钮继续。

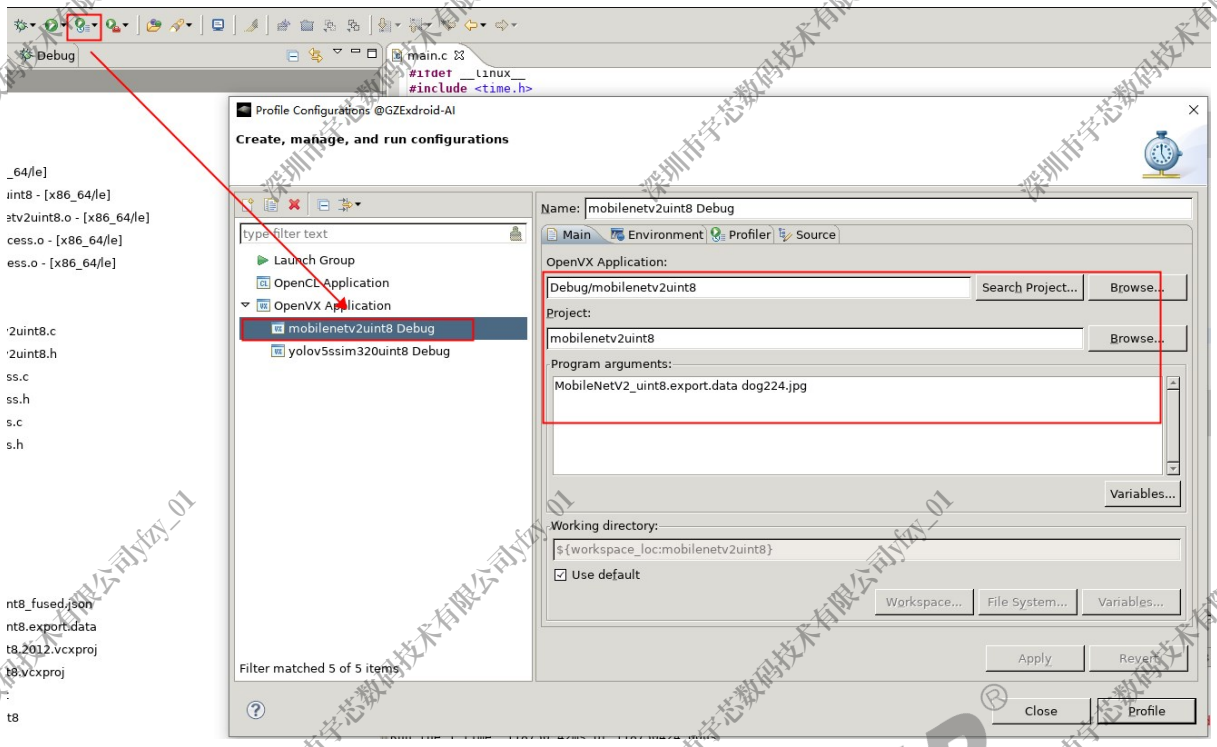

图 5-12: IDE 工具 profile 选项卡

然后进入 Profiler 选项卡继续 profile 配置：

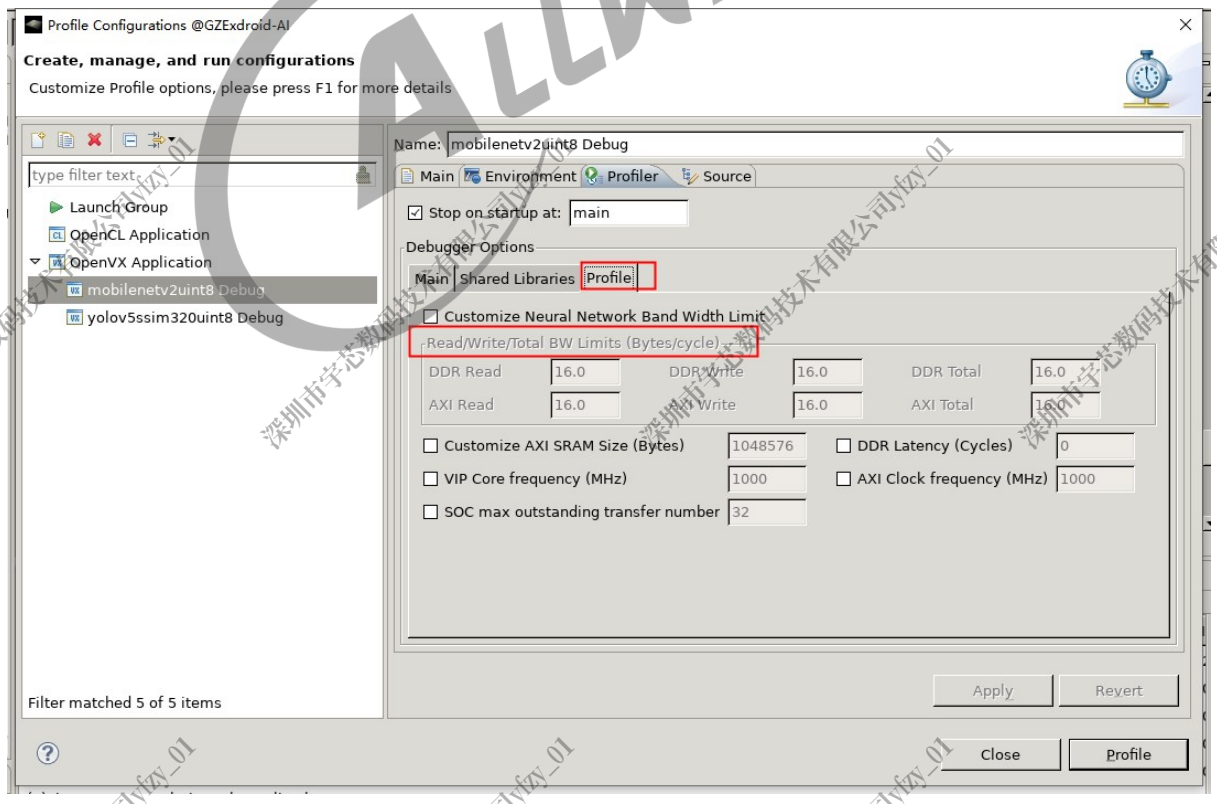

图 5-13: IDE 工具 profiler 配置

在这里我们可以进行调整带宽限制、AXI SRAM 大小、DDR Latency 值、NPU 时钟频率、AXI 时钟频率等操作。

我们可以将 VIP Core frequency（MHz）配置为当前 NPU 使用的时钟频率。然后我们再获取 dram 频率，计算总带宽：

```
#比如MR527我们获取到DDR时钟频率为1200MHz，而且使用LPDDR4，可计算带宽：
带宽 = DDR频率 *  （32/8） * 2 * 0.5 = 1200 * 4 = 4800 M/s
```

此时每 cycle NPU 分配到的带宽限制的值为总带宽/时钟频率。

之后点击 Resume 继续:

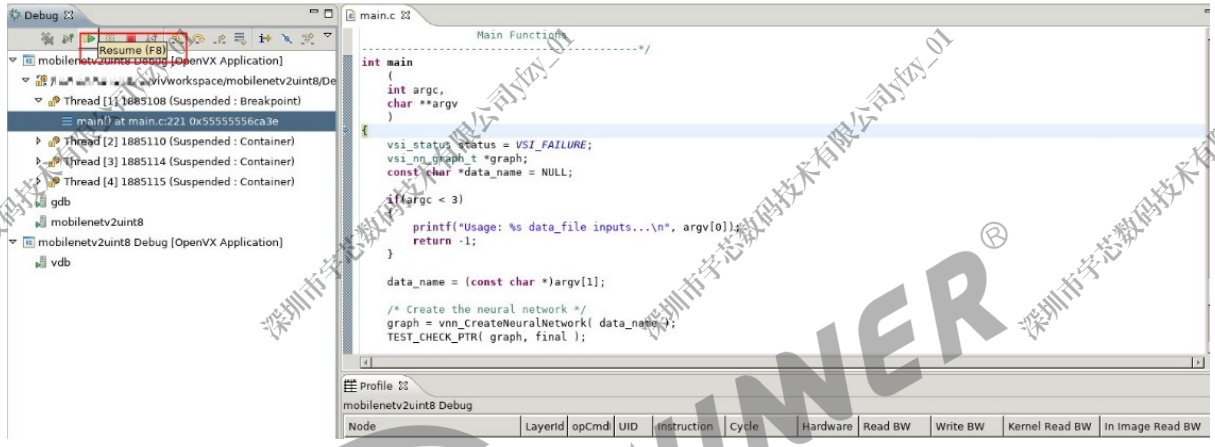

图 5-14: IDE 工具 Resume 执行

Profile 结束后, IDE 输出如下：

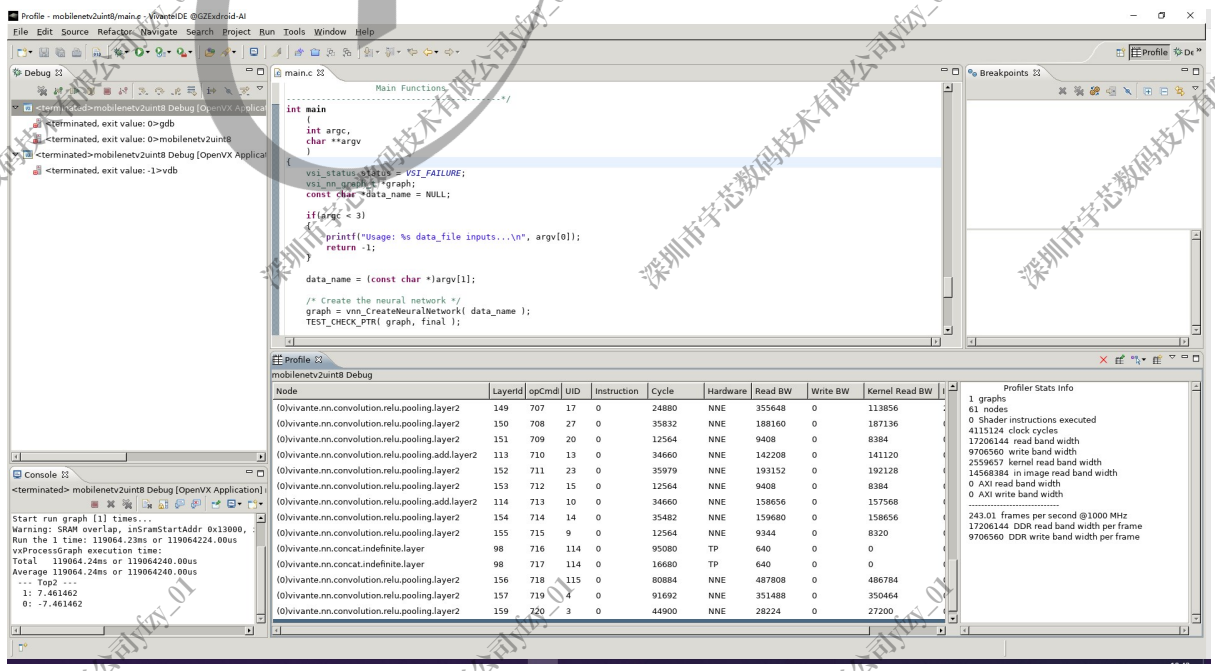

图 5-15: IDE 工具 profile 结果

简单分析一下 Profile 的信息，左下角和仿真结果输出相同，为推理 top2 的结果，中间的是各层的运行情况统计，包括每层的硬件处理单元，读写带宽，而对于由 PPU 计算的层，比如 softmax 层，还会有指令数统计等等。右下角的则是网络的整体运行性能分析，包括网络整体的读写带宽，处理帧率，时钟频率等等信息。

说明

当进行性能分析时，数据会与设备端的测试存在差异，如果有设备，建议最好在设备端进行实测。

5.8 NPU 驱动配置

5.8.1 menuconfig 配置方法

NPU 模块的 menuconfig 配置如下：

```
-> Allwinner BSP
  -> Device Drivers
    -> NPU Driver
      -><M> aw nna vip driver    ----VIPLite版本源码
      -><M> aw nna galcore driver ----Unified版本源码
```

5.8.2 设备树配置

NPU 模块的 dts 配置如下：

```
npu: npu@7122000 {
  compatible = "allwinner,npu";
  reg = <0x0 0x07122000 0x0 0x1000>;
  device_type = "npu";
  dev_name = "npu";
  interrupts = <GIC_SPI 199 IRQ_TYPE_LEVEL_HIGH>;
  /* 配置模块使用的clk */
  clocks = <&ccu CLK_NPU>, <&ccu CLK_PLL_NPU_2X>, <&dsp_ccu CLK_BUS_DSP_NPU_ACLK>, <&dsp_ccu
      CLK_BUS_DSP_NPU_HCLK>;
  clock-names = "clk_npu", "clk_parent", "npu-aclk", "npu-hclk";
  operating-points-v2 = <&npu_opp_table>;
  /* 配置模块使用的rst */
  resets = <&dsp_ccu RST_BUS_DSP_NPU>;
  reset-names = "npu_rst";
  /* 配置模块使用的中断 */
  interrupt-names = "npu";
  /* 配置NPU模块使用的频点 */
  npu-vf = <696>;
  power-domains = <&pd A523_PD_NPU>;
  status = "okay";
};
```

6 量化精度调优

6.1 量化方法和分析工具介绍

本 NPU 平台的量化特点如下：

- 支持 uint8/pcq int8/int16 的量化方式，以及混合量化（如 uint8+int16、uint8+float32 等）等多种量化方式。
- 支持后量化，也支持前量化模型直接导入，如 TFLite、ONNX 的 QAT 模型。

6.1.1 量化精度调优思路

模型量化后遇到精度下降问题时，需要调优，主要是分析找出精度损失的瓶颈层，以下提供一种思路供参考，如图所示。

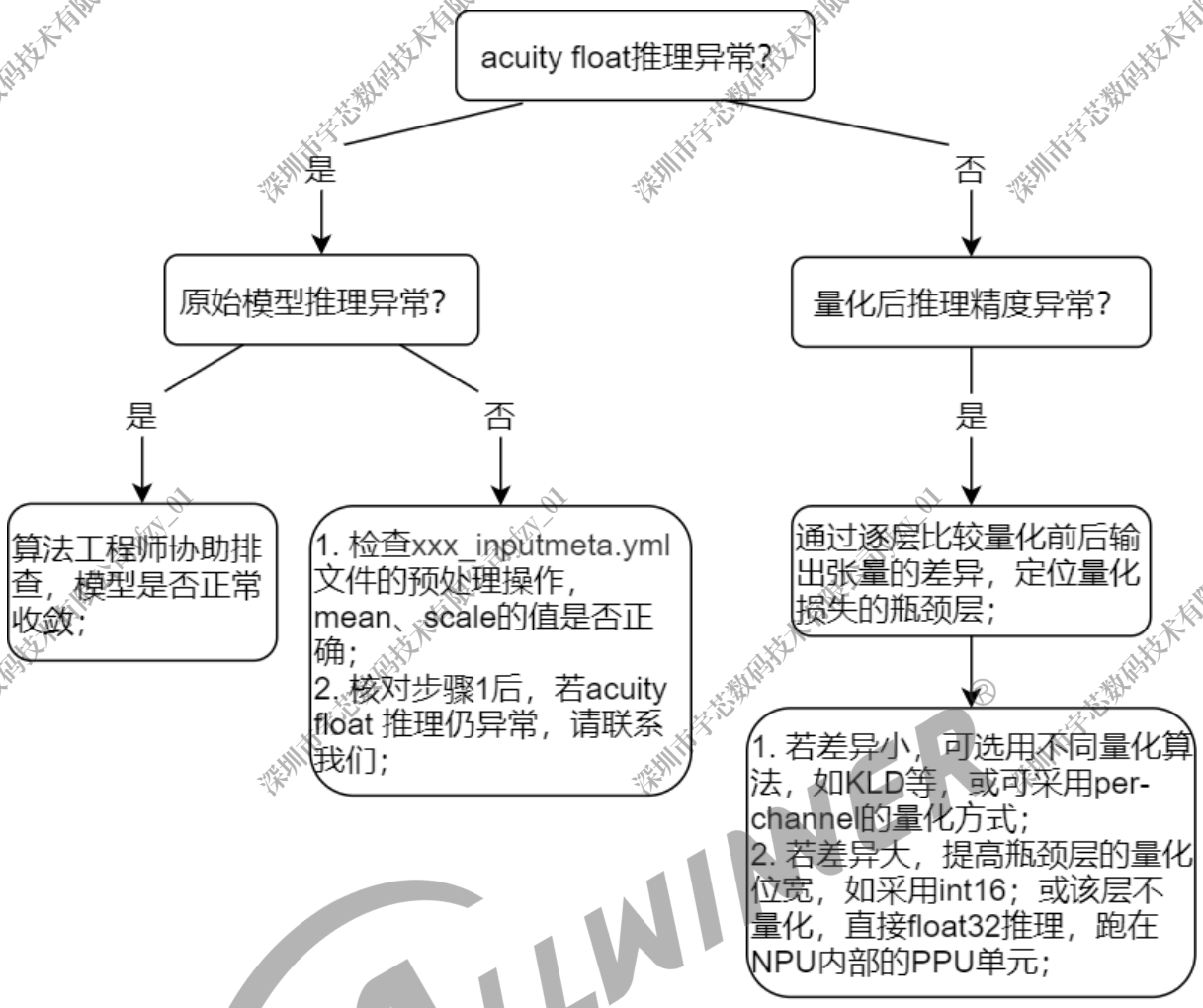

图 6-1: 量化损失问题排查

6.1.2 精度调优经验总结

- 模型自带的网络后处理节点，不适合 U8 量化，建议对应节点做 float 的混合量化；
- 若模型用了可分离卷积，若 U8 量化精度损失较大，可改用 pcq 逐通道量化的方法提高精度。

6.1.3 量化损失的分析工具

精度调优过程中，需要分析张量相似度来找出精度损失的瓶颈层，需要用到pegasus_dump.sh脚本、acuity_toolkit 工具的修改后的compute_tensor_similarity.py脚本和新增compare.py脚本。

6.1.3.1 对比张量相似度

使用./tools/compute_tensor_similarity.py 脚本（工具安装的目录 acuity-toolkit-binary-x.x.x/bin 下）对比量化前后的张量输出，命令如下：

```
# 比较两个张量
python ./tools/compute_tensor_similarity.py 1.tensor 2.tensor
```

输出有两个值，第一个值为欧氏距离，值越小，量化精度越高；第二个值为余弦相似度，值越接近 1.0，量化精度越高。输出示例如下所示：

```
euclidean_distance 489.9882
cos_similarity 0.999983
```

6.1.3.2 逐层对比张量

下面提供一个比较量化前后张量的方法，实现逐层比较功能。

- 通过脚本 pegasus_dump.sh 保存每一层的张量推理输出；

```
# 保存浮点推理张量
./ pegasus_dump.sh model_name float

# 保存uint8量化推理张量
./ pegasus_dump.sh model_name uint8
```

- 在 dump 的时候保存窗口按执行顺序打印的网络层名字为 tensor_list.txt；
- 在 compute_tensor_similarity.py 文件添加一个函数；

```
def cossim(tensor1file, tensor2file):
    tensor1, tensor2 = get_tensor(tensor1file, tensor2file)
    euclidean_distance = 0
    cos_similarity = 0
    if tensor1.shape == tensor2.shape:
        dim_now = 0
        sess = tf.Session()
        a, b = tf.placeholder(tf.float32,shape=tensor1.shape),tf.placeholder(tf.float32,shape=tensor2.shape)
        cos_similarity = compute_cos_sim(a,b,dim_now)
        euclidean_distance = compute_euclidean_dis(a,b)
        cos_similarity_result, euclidean_distance_result = sess.run(
                                                [cos_similarity, euclidean_distance],
                                                feed_dict={a:tensor1, b:tensor2})
        euclidean_distance = round(euclidean_distance_result,6)
        cos_similarity = round(cos_similarity_result,6)
    else:
        print('[INPUT ERROR]please make sure input tensors have the same shape!')

    return euclidean_distance, cos_similarity
```

- 编写 compare.py 文件；

```
#!/usr/bin/env python3
import numpy as np
```

```
import compute_tensor_similarity

folder1 = './dump_float/'
folder2 = './dump_uint8/'
tensor_list_file  = './tensor_list.txt'

fw = open('compare_result_cossim.txt', 'w', encoding='utf-8')
f  = open(tensor_list_file)
line  = f.readline().split('\n')[0]                # 调用文件的 readline()方法
while  line:
    if  (line.split('_')[0]  == 'Initializer  '):
        euclidean_dis = 0.0
        cos_sim = 1.0
    else:
        euclidean_dis, cos_sim = compute_tensor_similarity.cossim(folder1 + line, folder2 + line)

    res_str  = line  + '  eu_dis:{},  cos:{}'.format(euclidean_dis, cos_sim)
    print(res_str)
    fw.write(res_str  + '\n')

    line  = f.readline().split('\n')[0]

fw.close()
f.close()
```

执行 cpmpare.py，即可按 tensor_list.txt 中网络层名字顺序比较两个目录下张量的相似度与欧式距离。

```
python compare.py
```

说明

把 float 推理对应的 tensor 放在 dump_float 目录，把 uint8 量化对应的 tensor 放在 dump_uint8 目录。

输出示例如下图所示。

```
attach_Concat_Concat_303_out0_0_out0_nchw_1_6300_85.tensor eu_dis:5938.3876953125, cos:0.9644860029220581
attach_Transpose_Transpose_200_out0_1_out0_nchw_1_3_40_40_85.tensor eu_dis:124.1949462890625, cos:0.9995139837265015
attach_Transpose_Transpose_235_out0_2_out0_nchw_1_3_20_20_85.tensor eu_dis:66.0180435180664, cos:0.9994609951972961
attach_Transpose_Transpose_270_out0_3_out0_nchw_1_3_10_10_85.tensor eu_dis:33.8359489440918, cos:0.9994940161705017
Transpose_Transpose_270_4_out0_nchw_1_3_10_10_85.tensor eu_dis:33.8359489440918, cos:0.9994940161705017
Transpose_Transpose_235_5_out0_nchw_1_3_20_20_85.tensor eu_dis:66.0180435180664, cos:0.9994609951972961
```

图 6-2: 逐层对比的结果

6.1.4 量化表参数说明

整个量化描述文件分成两大部分：

- quantize_parameters：各层的量化配置，在仿真推理、导出模型时实际使用的配置。
- customized_quantize_layers：自定义量化，用于下一次混合量化。Acuity 工具会在这里给出建议的量化类型，也可以手动配置特定层的量化类型。

以一个 conv 节点举例说明量化表参数，其中：

- out0 为该 conv 节点输出的量化信息
- weight 为该 conv 节点权重的量化信息
- bias 为该 conv 节点 bias 数据的量化信息

```
'@Conv_/model.23/cv2/conv/Conv_28:out0':
  qtype: u8
  quantizer: asymmetric_affine
  rounding: rtne
  max_value: 9.644393920898438
  min_value: -9.644393920898438
  scale: 0.07564230263233185
  zero_point: 128
'@Conv_/model.23/cv2/conv/Conv_28:weight':
  qtype: u8
  quantizer: asymmetric_affine
  rounding: rtne
  max_value: 0.6329048275947571
  min_value: -0.9523233771324158
  scale: 0.006216580979526043
  zero_point: 153
'@Conv_/model.23/cv2/conv/Conv_28:bias':
  qtype: i32
  quantizer: asymmetric_affine
  rounding: rtne
  max_value: 436738.8435466276
  min_value: -436738.84375
  scale: 0.0002033723721979186
  zero_point: 0
```

各字段参数释义：

```
qtype           量化数据类型，例如：u8, i8, i16, i32，i64
quantizer       量化方式，asymmetric_affine 是指 非对称量化
max_value/min_value  所统计数据的最大值/最小值
scale           量化的比例因子
zero_point      量化的零点值
```

6.2 量化调优流程

6.2.1 浮点推理

在模型导入和修改输入、输出描述 yml 文件后，即可进行浮点推理。这一步是为了得到输出层的 Golden tensor 用于后面的张量对比，同时为了检验配置是否正确。

```
# 导入，注意model_name不需要后缀
./ pegasus_import.sh model_name

# 修改 xxx_inputmeta.yml
# 修改 xxx_postprocess_file.yml
# 浮点推理
./ pegasus_inference.sh model_name float
```

得到输出张量并解析结果，若结果正确则继续执行量化步骤。若 float 推理，结果出现明显错误，请及时联系我们。

说明

注意将 xxx_inputmeta.yml 文件的 mean 和 scale 参数修改为符合网络实际的训练时的参数。例如对于 yolov5s 来讲，scale 需要修改为 0.0039。

6.2.2 常规量化推理

常规量化有 uint8、pcq 和 int16。量化特性有以下几点：

- 权重 8/16bit 量化，bias 默认 32bit 量化。
- quantized 文件中 bias 支持 i32 和 i64 量化，属于硬件要求。
- min_value、max_value 参数也是 NPU 内部使用，用户可不关注。

6.2.2.1 uint8

```
# 量化
./pegasus_quantize.sh model_name uint8

# 量化后仿真推理
./pegasus_inference.sh model_name uint8
```

量化后仿真推理得到输出张量并解析结果，若结果正确则执行下一步导出模型；若结果输出有误，则进一步提高量化精度尝试。

6.2.2.2 pcq

采用 8bit 的 perchannel 量化，相较于前面的逐层量化，量化更加细致，会增加量化参数，推理耗时也会略增加。

```
# 量化
./pegasus_quantize.sh model_name pcq

# 量化后仿真推理
./pegasus_inference.sh model_name pcq
```

量化后仿真推理得到输出张量并解析结果，若结果正确则执行下一步导出模型；若结果输出有误，则进一步提高量化精度尝试。

6.2.2.3 int16

```
# 量化
./pegasus_quantize.sh model_name int16

# 量化后仿真推理
./pegasus_inference.sh model_name int16
```

量化后仿真推理得到输出张量并解析结果，若结果正确则执行下一步导出模型；若 int16 量化精度下，仍无检测输出，继续提高量化精度为 float 进行推理。

6.2.3 混合量化

6.2.3.1 详细步骤

在首次量化时会生成：

- <modelName>_<dataType>.quantize 量化文件
- entropy.txt 熵值文件，若添加参数 –compute-entropy 会自动生成该文件，计算网络每一层的量化熵值，熵值越大，精度越低。

接着进行以下步骤：

1. 打开 *.quantize 量化文件，文件末尾找到 **customized_quantize_layers** 自定义量化层部分。
2. 在自定义量化层部分，可增加、修改、删除需要进一步量化的层，例如：

```
customized_quantize_layers:
    'Sigmoid_Sigmoid_275_21': dynamic_fixed_point-i16
```

Sigmoid_Sigmoid_275_21 是需要重新量化的层名字，dynamic_fixed_point-i16 是新的量化类型。其中层名字必须从 <modelName>.json 文件中获取。

3. 修改量化脚本：

用参数 –hybrid 代替 –rebuild，可参考 pegasus_quantize-hybrid.sh 脚本，示例如下：

```
pegasus.py quantize --model yolov5s.json --model-data yolov5s.data --device CPU --with-input-meta yolov5s-inputmeta
    .yml --hybrid --model-quantize yolov5s.quantize --quantizer perchannel_symmetric_affine --qtype int8
```

4. 执行混合量化脚本。

⚠注意

customized_quantize_layers 参数书写格式要求：

1. **行首要空两格。**
2. **量化层名用单引号 '' 括起来，特别是图层名为纯数字。**
3. **冒号后面必须空一格。**

6.2.3.2 结果分析

- 更新后的量化文件 <modelName>_<dataType>.quantize；
- 会额外生成 <modelName>_<dataType>.quantize.json 文件，在导出模型阶段使用。

其中，<modelName>_<dataType>.quantize.json 与 import 阶段生成的 <modelName>.json 文件对比，这个新的网络文件包含重新量化层的数据类型转换操作，可通过文件对比工具查看。

举例下文 Sigmoid_Sigmoid_275_21 层的变化，第一次量化的情况如下：

```
"Sigmoid_Sigmoid_275_21": {
  "name": "Sigmoid_Sigmoid_275",
  "op": "sigmoid",
  "inputs": [
    "@Transpose_Transpose_214_36:out0"
    ],
  "outputs": [
    "out0"
    ]
  }
```

重新量化后的情况如下：

```
"Sigmoid_Sigmoid_275_21": {
  "name": "Sigmoid_Sigmoid_275",
  "op": "sigmoid",
  "inputs": [
    "@Sigmoid_Sigmoid_275_21_pre_asymmetric_affineu8_to_dynamic_fixed_pointi16_40:out0"
    ],
  "outputs": [
    "out0"
    ]
  }
"Sigmoid_Sigmoid_275_21_pre_asymmetric_affineu8_to_dynamic_fixed_pointi16_40": {
  "name": "Sigmoid_Sigmoid_275_21_pre_asymmetric_affineu8_to_dynamic_fixed_pointi16",
  "op": "dtype_converter",
  "inputs": [
    "@Transpose_Transpose_214_36:out0"
    ],
  "outputs": [
    "out0"
    ]
  }
```

可看到重新量化后，网络会添一个层Sigmoid_Sigmoid_275_21_pre_asymmetric_affineu8_to_dynamic_fixed_pointi16_40，op 为dtype_converter，即数据类型转换操作，把 u8 转换为 int16。

6.2.3.3 操作示例

简单介绍 uint8+float32 混合精度的量化操作。

1. uint8 量化

```
# 量化
./ pegasus_quantize.sh model_name uint8
```

2. 修改量化文件

打开量化文件，删除 customized_quantize_layers 中原有的内容，添上需要提高量化精度的层名字与量化方式。

举例如下：

```
# 如Sigmoid_Sigmoid_275_21: float32示意为：
# Sigmoid_Sigmoid_275_21为需要提高量化精度的层名字，float32为新的量化类型；

customized_quantize_layers:
  Sigmoid_Sigmoid_275_21: float32
  Slice_Slice_302_10: float32
  Slice_Slice_280_50: float32
  Slice_Slice_292_49: float32
  Mul_Mul_282_47: float32
  Sub_Sub_284_32: float32
  Add_Add_285_17: float32
  Mul_Mul_287_8: float32
  Mul_Mul_294_34: float32
  Pow_Pow_296_19: float32
  Mul_Mul_297_9: float32
  Concat_Concat_303_5: float32
  Reshape_Reshape_307_2: float32
```

技巧

1. 若层名字不正确，会提示如下图的错误。
2. 注意不同 npu 版本参数书写格式有一定差异。

```
", line 223, in apply
    if self.net.get_layer(lid).is_op('priorbox'):
AttributeError: 'NoneType' object has no attribute 'is_op'
```

图 6-3: 混合量化层名字不正确的报错提示

3. 再次执行量化

```
# 执行混合量化
./ pegasus_quantize-hybrid.sh model_name uint8
```

6.2.3.4 注意事项

- 在混合量化步骤，每次修改量化文件中对应节点的量化精度时，建议在最初生成的xx_uint8.quantize文件下修改，这样能避免节点输入类型不对的问题：

E [ops/vsi_nn_op_conv2d.c:op_check:189]Inputs/Outputs data type not support: ASYM UINT8, DFP INT16 E [vsi_nn_graph.c:setup_node:551]Check node[22] CONV2D fail

遇到上述报错，可见与 CONV2D 节点有关。

- 混合量化步骤后，用 netron 软件打开生成的 <modelName>_<dataType>.quantize.json 文件，检查是否在相应位置插入数据类型转换节点 dtype_converter，是否一一对应。示例如下，两个 dtype_converter 节点之间的网络层是 int16 量化，检查确认 u8 转 i16 与 i16 转 u8 已对应。

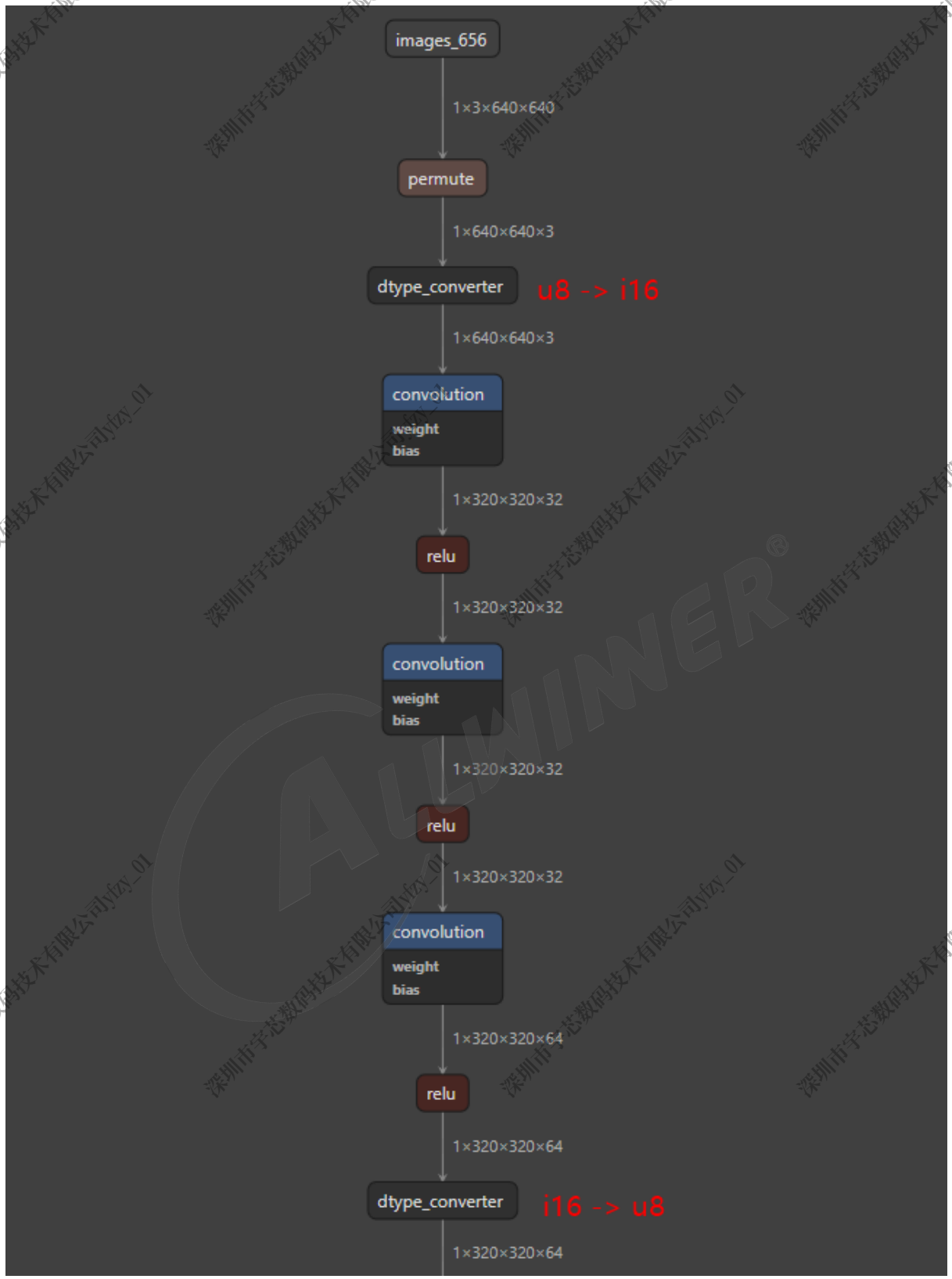

图 6-4: 混合量化检查 dtype_converter 节点

6.3 QAT 模型导入

若混合量化后的精度仍达不到要求，此时可采用 QAT 量化感知训练（TFLite、PyTorch 等框架支持 QAT 训练）的方法重新训练模型，导出 Acuity Toolkit 工具可支持的模型类型；在量化模型的导入阶段，可直接获取模型自带的量化参数，无需执行量化步骤。

目前支持的量化精度类型：int8 , uint8。

目前支持的模型类型：TFLite、ONNX。

例如在 SDK 的 models/mobilenet_v1_1.0_224_quant 目录的示例是量化模型，导入该模型后，自动生成*_uint8.quantize文件，则无需量化操作，可直接仿真推理、模型文件导出操作等。目录内容如下：

```
AwExdroid99:~/models/mobilenet_v1_1.0_224_quant$ tree
.
├── channel_mean_value.txt
├── dataset.txt
├── mobilenet_v1_1.0_224_quant.tflite
└── space_shuttle_224x224.jpg
```

量化感知训练目前已被广泛使用，一般需要额外增加训练代码，且部分开源算法的代码可能与其存在冲突。各主流框架的 QAT 训练说明可参考以下链接：

PyTorch：https://pytorch.org/blog/quantization-in-practice/。

Tensorflow：https://www.tensorflow.org/model_optimization/guide/quantization/training。

7 性能优化

7.1 分析工具介绍

7.1.1 端侧的性能和带宽 profile

获取端侧每层 Layer 的实际运行情况，需要**替换中间件的运行库以及替换打开调试的驱动 ko**。
在ai-sdk/viplite-Android或viplite-tina/中的debug目录有对应库文件：

```
#v1.13.0版本
├── debug
│   ├── BW   #可查看模型整体运行时间及带宽
│   │   ├── libVIPlite.so
│   │   └── libVIPuser.so
│   ├── CNN_BW #可查看每层运行时间及带宽
│   └── DEBUG  #可查看更多等级log
├── inc
│   ├── vip_lite_common.h
│   └── vip_lite.h
├── libVIPlite.so
└── libVIPuser.so

#v2.0.3版本
├── debug
│   ├── BW
│   │   ├── libNBGlinker.so
│   │   └── libVIPhal.so
│   ├── CNN_BW
│   └── DEBUG
├── inc
│   ├── vip_lite_common.h
│   └── vip_lite.h
├── libNBGlinker.so
└── libVIPhal.so
```

准备打开调试的驱动 ko 文件，需要修改 NPU 的头文件中对应的宏。

平台	头文件
V85x、R853	lichee/linux-4.9/include/npu/gc_vip_common.h
MR527、T527、AI985	bsp/drivers/npu/aw_nna_vip/inc/gc_vip_common.h
MR536、T536、A733、T736	bsp/drivers/npu/aw_nna_vip/vip2/inc/vip_lite_config.h

BW 需打开 vpmdENABLE_BW_PROFILING 宏为 1，参考如下：

```
+++ b/drivers/npu/aw_nna_vip/inc/gc_vip_common.h
@@ -289,7 +289,7 @@ choose driver runs on FPGA or silicon board.
            AHB mode by default, read whole VIP core performance counters in the AHB registers by CPU.
 */
 #ifndef vpmdENABLE_BW_PROFILING
-#define vpmdENABLE_BW_PROFILING         0
+#define vpmdENABLE_BW_PROFILING         1
 #endif
```

CNN_BW 需同时打开 vpmdENABLE_CNN_PROFILING 宏和 vpmdENABLE_BW_PROFILING 宏，参考如下：

```
+++ b/drivers/npu/aw_nna_vip/inc/gc_vip_common.h
@@ -279,7 +279,7 @@ choose driver runs on FPGA or silicon board.
       the rt_cnn_profile value is 1 by default.
 */
 #ifndef vpmdENABLE_CNN_PROFILING
-#define vpmdENABLE_CNN_PROFILING         0
+#define vpmdENABLE_CNN_PROFILING         2
 #endif

 /*
@@ -289,7 +289,7 @@ choose driver runs on FPGA or silicon board.
            AHB mode by default, read whole VIP core performance counters in the AHB registers by CPU.
 */
 #ifndef vpmdENABLE_BW_PROFILING
-#define vpmdENABLE_BW_PROFILING         0
+#define vpmdENABLE_BW_PROFILING         1
 #endif
```

DEBUG 可打开 vpmdENABLE_DEBUG_LOG 宏为 4：

```
--- a/drivers/npu/aw_nna_vip/inc/gc_vip_common.h
+++ b/drivers/npu/aw_nna_vip/inc/gc_vip_common.h
@@ -319,7 +319,7 @@ provide richer profiling information of the internal NN/TP. This feature needs h
     the rt_log value is 1 by default.
 */
 #ifndef vpmdENABLE_DEBUG_LOG
-#define vpmdENABLE_DEBUG_LOG         1
+#define vpmdENABLE_DEBUG_LOG         4
 #endif
```

修改完成后重新编译驱动。

将对应平台库和驱动 ko 文件推入板端，并加载驱动，和设置环境变量 LD_LIBRARY_PATH，指定库所在目录即可使用。例如：

```
adb push BW /mnt/extsd/viplite/
adb push vipcore.ko /mnt/extsd/viplite/
adb shell
rmmod vipcore
insmod /mnt/extsd/viplite/vipcore.ko
export LD_LIBRARY_PATH=/mnt/extsd/viplite/BW
cd /mnt/extsd/test
./vpm_run -s sample.txt
```

7.1.1.1 BW 库可查看模型整体运行时间及带宽

```
@dev0-core0{
Total read bandwidth (Byte): 5780864
Total write bandwidth (Byte): 1820736
Total BandWidth (Byte): 7601600
AXI read bandwidth (Byte): 0
AXI write bandwidth (Byte): 0
DDR read bandwidth (Byte): 5780864
DDR write bandwidth (Byte): 1820736
VIP total cycles: 1562296
VIP total idle cycles: 74925
VIP total busy cycles: 1487371
VIP total PWM: 95.20%
VIP NPU CLK Frequency : 900 MHz
VIP hardware profile time: 1652.6 us
VIP hardware profile FPS: 605.09
VIP hardware instantaneous bandwidth (MB/s): 4386.6
}@
```

该模型读带宽为 5780864 Byte，写带宽为 1820736 Byte，运行时钟数为 1562296，其中空闲时钟数为 74925，所以对应的运行时间为 1652.6 us。

7.1.1.2 CNN 库可查看每层运行时间及带宽

```
start to run network=./mobilenetv2-12/network_binary.nb
debugfs is already mounted!
Now NPU Frequency = 900000000
nodeId:141, core_id:0, uid:100, abs_op_id:0, layer name:ConvolutionReluPoolingLayer2, operation target:NN
execution time: 200 us
@dev0-core0{
Total read bandwidth (Byte): 459968
Total write bandwidth (Byte): 0
Total BandWidth (Byte): 459968
AXI read bandwidth (Byte): 0
AXI write bandwidth (Byte): 0
DDR read bandwidth (Byte): 459968
DDR write bandwidth (Byte): 0
VIP total cycles: 133443
VIP total idle cycles: 70483
VIP total busy cycles: 62960
VIP total PWM: 47.18%
VIP NPU CLK Frequency : 900 MHz
VIP hardware profile time: 70.0 us
VIP hardware profile FPS: 14294.79
VIP hardware instantaneous bandwidth (MB/s): 6270.5
}@
nodeId:142, core_id:0, uid:98, abs_op_id:1, layer name:ConvolutionReluPoolingLayer2, operation target:NN
execution time: 164 us
@dev0-core0{
Total read bandwidth (Byte): 1472
Total write bandwidth (Byte): 530432
Total BandWidth (Byte): 531904
AXI read bandwidth (Byte): 0
AXI write bandwidth (Byte): 0
```

```
DDR read bandwidth (Byte): 1472
DDR write bandwidth (Byte): 530432
VIP total cycles: 128227
VIP total idle cycles: 62497
VIP total busy cycles: 65730
VIP total PWM: 51.26%
VIP NPU CLK Frequency : 900 MHz
VIP hardware profile time: 73.0 us
VIP hardware profile FPS: 13692.38
VIP hardware instantaneous bandwidth (MB/s): 6945.6
}@
.....
```

可查看每层算子的硬件计算单元 operation target、运行时间 execution time、读带宽 Total read bandwidth、写带宽 Total write bandwidth。

7.1.1.3 DEBUG 库可查看更多等级 log

建议 NPU 驱动的 DEBUG 一起打开，请参见《驱动问题》一节的步骤。

7.1.2 PC 侧的算子融合情况及 Profile

设置环境变量

```
export VIV_VX_DEBUG_LEVEL=1
export VSIMULATOR_DEBUG=4
export DISABLE_IDE_DEBUG=1
export VIV_VX_ENABLE_GRAPH_TRANSFORM=-Dump
```

并使用 Pegasus export 命令导出模型，同时保存模型融合情况及打印详细模型情况

```
wksp/yolov8s_uint8
├── BUILD
├── GraphDump
│   ├── after_xxx_XXX_graph.json
│   ├── applied_optimization.txt
│   ├── before_optimization_topology.json
│   └── final_graph_topology.json
├── graph.json
├── main.c
├── makefile.linux
├── vnn_global.h
├── vnn_post_process.c
├── vnn_post_process.h
├── vnn_pre_process.c
├── vnn_pre_process.h
├── vnn_yolov8suint8.c
├── vnn_yolov8suint8.h
├── yolov8suint8.2012.vcxproj
├── yolov8s_uint8.export.data
├── yolov8s_uint8_fused.json
└── yolov8suint8.vcxproj
```

其中

- GraphDump/applied_optimization.txt 为算子融合情况
- GraphDump/before_optimization_topology.json 为模型初始结构
- GraphDump/final_graph_topology.json 为模型最终结构（即最终模型板端运行时的结构）
- GraphDump/after_xxx_XXX_graph.json 为中间每层优化过程结构

对应的输出 log 如下

```
...
================== Finial Groups Info ================
GPU Count = 1 AXI-SRAM = 0 Bytes VIP-SRAM = [479744] Bytes SWTILING_PHASE_FEATURES[1, 1, 1]
  0   NN [( 640  640    3 1,  1228800, 0x0x1fcd040(0x0x1fcd040, 0x(nil)) ->  320  320   32 1,  3276800, 0x0x205c3c0(0
        x0x205c3c0, 0x(nil))) k(3 3    3, 1536) pad(1 1) pool(1 1, 1 1)] C[  1]
  1   TP [( 320  320   32 1,  3276800, 0x0x205c3c0(0x0x205c3c0, 0x(nil)) ->  320  320   32 1,  3276800, 0x0x2065ba0(0
        x0x2065ba0, 0x(nil))) k(0 0    0, 0) pad(0 0) pool(1 1, 1 1)] P[  0] C[  2]
  2   NN [( 320  320   32 1,  3276800, 0x0x2065ba0(0x0x2065ba0, 0x(nil)) ->  160  160   64 1,  1638400, 0x0x2068300(0
        x0x2068300, 0x(nil))) k(3 3    32, 21248) pad(1 1) pool(1 1, 1 1)] P[  1] C[  3]
  3   TP [( 160  160   64 1,  1638400, 0x0x2068300(0x0x2068300, 0x(nil)) ->  160  160   64 1,  1638400, 0x0x2071a70(0
        x0x2071a70, 0x(nil))) k(0 0    0, 0) pad(0 0) pool(1 1, 1 1)] P[  2] C[  4]
  ...
155   NN [(  20   20  128 1,   51200, 0x0x23875c0(0x0x23875c0, 0x(nil)) ->   20   20   80 1,   32000, 0x0x2392d20(0x0x2392d20,
         0x(nil))) k(1 1  128, 11776) pad(0 0) pool(1 1, 1 1)] P[154] C[156]
156   TP [(  20   20   80 1,   32000, 0x0x2392d20(0x0x2392d20, 0x(nil)) ->   20   20   80 1,   32000, 0x0x1fca650(0x0x1fca650, 0
        x(nil))) k(0 0    0, 0) pad(0 0) pool(1 1, 1 1)] P[155] C[157]
157   SH [( 400   80    1 1,   32000, 0x0x1fca650(0x0x1fca650, 0x(nil)) ->  400    1    1 1,     800, 0x0x20c1e90(0x0x20c1e90, 0x(
        nil))) k(0 0    0, 0) pad(0 0) pool(0 0, 1 1)] P[156] C[158]
158   TP [(  20   20    1 1,     800, 0x0x20c1e90(0x0x20c1e90, 0x(nil)) ->   20   20    1 1,     800, 0x0x1fcb9b0(0x0x1fcb9b0, 0x(nil)
        )) k(0 0    0, 0) pad(0 0) pool(1 1, 1 1)] P[157]

Group  0 [AS = 0, VS = 479744, Workload = 159] [  0 - 158, 159], [0]
=======================================================


----------------------Begin VerifyTiling Group  0 [0 - 158]-------------------------
AXI-SRAM = 0 Bytes VIP-SRAM = 479744 Bytes SWTILING_PHASE_FEATURES[1, 1, 1]

Detected Segments
TL_VS (0 - 5)
TL_VS (6 - 10)
...
AB_VS (136 - 137)
AB_VS (140 - 158)
======================== Block [0 - 5] ==============================
  0 NN DD -> VS [( 1228800,    71680), IC(   2304), KC(   1280), NNT(     0,      0)]
  1 TP VS -> VS [(   71680,   153856), IC(      0), KC(      0), NNT(     0,      0)]
  2 NN VS -> VS [(  153856,    71680), IC(      0), KC(  12800), NNT(     0,      0)]
  3 TP VS -> VS [(   71680,    71936), IC(      0), KC(      0), NNT(     0,      0)]
  4 NN VS -> VS [(   71936,    71680), IC(      0), KC(   3840), NNT(     0,      0)]
  5 TP VS -> DD [(   71680,        0), IC(      0), KC(      0), NNT(     0,      0)]
---------------------------------------------------------------------
Segment Tiling (0 - 5)
...
LayerID operationID opCmdID UID INSTRUCTION CYCLE HARDWARE READBW WRITEBW kernelReadBW inImageReadBW
        MAC MACUtil NODE
92 0 1 226 0 727200 NNE 2677184 0 960 2670336 88473600 10.56% vivante.nn.convolution.relu.pooling.layer2
0 0 2 224 0 3361680 TP 58880 0 0 0 0 0.00% vivante.nn.activation.layer
93 0 6 222 0 766514 NNE 35904 0 12352 0 471859200 53.44% vivante.nn.convolution.relu.pooling.layer2
1 0 11 221 0 1643488 TP 29440 0 0 0 0 0.00% vivante.nn.activation.layer
...
154 0 1093 19 0 6647 NNE 10112 0 9088 0 4096000 53.49% vivante.nn.convolution.relu.pooling.layer2
```

```
81 0 1094 10 0 35024 TP 1280 64000 0 0 0 0.00% vivante.nn.activation.layer
82 0 1095 18 0 1316 SHD 32000 0 0 0 0.00% vivante.nn.tensor.reducesum
83 0 1096 9 0 2794 TP 640 832 0 0 0 0.00% vivante.nn.lut.layer

Profiler Stats Info
1 graphs
159 nodes
0 instructions executed
63219247 NN clock cycles
79745024 NN read band width
48732096 NN write band width
0 NN AXI SRAM read band width
0 NN AXI SRAM write band width
...
```

- Groups Info 显示每层算子的硬件计算单元、输入输出尺寸等信息。
- Detected Segments 说明模型拆分情况
- 最后部分是一些 Profile 信息，包括每层算子仿真时钟数、输入输出带宽；以及模型总时钟数、总输入输出带宽等信息。

7.1.3 端侧查询 DDR、NPU 等 Debug 信息

1. **挂载 debugfs**

```
mount -t debugfs none /sys/kernel/debug
```

2. **读取 DDR 频率**

```
cat /sys/kernel/debug/clk/clk_summary | grep ddr
```

如下结果

```
   pll-ddr              3     4     0  2400000000      0     0  50000      Y
```

因此 DDR 频率为 2400000000/2=1200MHz

3. **读取 NPU 频率及临时修改**

```
cat /sys/kernel/debug/clk/clk_summary | grep npu
```

如下结果

```
   npu                  0     1     0  696000000       0     0  50000      N
```

因此 NPU 频率为 696000000=696MHz。

针对 v2.0.3 及其后续版本，NPU 频率调试节点使用如下：

读取 NPU 的可用频率：

```
cat /sys/kernel/debug/viplite/clk_freq
```

如下结果

```
supported frequencies:
    328000000 (Hz)
    492000000 (Hz) <---- current freq
```

因此 NPU 的可用频率为 328Mhz、492Mhz，当前的频率是 492Mhz。

临时修改 NPU 的频率用于调试

```
echo 328000000 > /sys/kernel/debug/viplite/clk_freq
```

如下结果且串口打印success to update to frequency 328000000

```
cat /sys/kernel/debug/viplite/clk_freq
supported frequencies:
    328000000 (Hz)<---- current freq
    492000000 (Hz)
```

⚠ 注意

1. 只能设置此节点所显示的频率。设置其他频率会失败，则当前频率不变。
2. 修改的频率作为调试使用，非持久化。重启板子或卸载重装驱动会恢复到默认频率。
3. NPU 使用过程中，不可修改频率。串口有提示NPU is running, unable to change freq打印。

4. **读取 NPU loading 数据**

```
cat /sys/kernel/debug/viplite/core_loading
```

如下结果

```
NPU Loading -----> Core0: 0%
NPU Loading -----> Core0: 34%
NPU Loading -----> Core0: 96%
... ...
NPU Loading -----> Core0: 93%
NPU Loading -----> Core0: 0%
```

NPU 使用率节点以进程做统计，可以得到对应进程当前的 NPU 使用率情况。

7.2 性能分析

7.2.1 耗时分析

- 前后处理相关耗时
- 输入输出拷贝耗时
- NPU 推理耗时

7.2.2 NPU 模型运行分析

根据 profile log 分析

```
LayerID operationID opCmdID UID INSTRUCTION CYCLE HARDWARE READBW WRITEBW kernelReadBW inImageReadBW
        MAC MACUtil NODE
92 0 1 226 0 727200 NNE 2677184 0 960 2670336 88473600 10.56% vivante.nn.convolution.relu.pooling.layer2
0 0 2 224 0 3361680 TP 58880 0 0 0 0 0.00% vivante.nn.activation.layer
93 0 6 222 0 766514 NNE 35904 0 12352 0 471859200 53.44% vivante.nn.convolution.relu.pooling.layer2
1 0 11 221 0 1643488 TP 29440 0 0 0 0 0.00% vivante.nn.activation.layer
...
154 0 1093 19 0 6647 NNE 10112 0 9088 0 4096000 53.49% vivante.nn.convolution.relu.pooling.layer2
81 0 1094 10 0 35024 TP 1280 64000 0 0 0 0.00% vivante.nn.activation.layer
82 0 1095 18 0 1316 SHD 32000 0 0 0 0 0.00% vivante.nn.tensor.reducesum
83 0 1096 9 0 2794 TP 640 832 0 0 0 0.00% vivante.nn.lut.layer

Profiler Stats Info
1 graphs
159 nodes
0 instructions executed
63219247 NN clock cycles
79745024 NN read band width
48732096 NN write band width
0 NN AXI SRAM read band width
0 NN AXI SRAM write band width
```

分析逐层耗时，如上 log 所示，

- CYCLE 列为 NPU 指令数量，对应层时间为 CYCLE 除以 NPU 频率即可算出每层耗时。
- HARDWARE 为硬件计算单元，分别对应 TP、NNE、SHD 计算单元
- READBW、WRITEBW 为每层的输入输出带宽信息

根据每层算子的输入输出尺寸，以及计算耗时评估，该层耗时是否有异常，通常情况可能，

- 卷积 NNE 算力瓶颈
- 数据进行变换等操作耗时
- 带宽限制导致，一般也是数据变换等操作

7.3 优化建议

1. 根据每层运算时间，是否某些算子耗时较长，寻找可替代算子；
2. 根据每层算子的硬件计算单元，结合《NPU_ 算子直接列表》，SHD 算子（即 PPU 实现的算子）是否可以使用 NNE 算子替代；
3. 尽量使用算子融合，如 conv+relu+pool，conv+add，conv+relu+pool+add
4. 使用更低精度量化，节省算力及带宽
5. 减少某些耗时长层的尺寸/计算量，每层输入尺寸宽度尽量减少，保证相同 SRAM 下能缓存更多行数据，宽度 32bit 对比
6. 根据带宽限制设置整体模型规模

8 常见问题

8.1 模型量化问题

一、数据集配置常见报错

错误信息 1：

```
Traceback (most recent call last):
 File "pegasus.py", line 131, in <module>
...
 File "acuitylib/acuitynetbuilder.py", line 81, in build_normal_graph
 File "acuitylib/acuitynetbuilder.py", line 44, in _try_build_dataset
 File "acuitylib/dataset/file_path_dataset.py", line 32, in __init__
 File "acuitylib/dataset/base_dataset.py", line 95, in __init__
 File "acuitylib/dataset/text_dataset.py", line 9, in _build
FileNotFoundError: [Errno 2] No such file or directory: 'dataset0.txt'
[30846] Failed to execute script pegasus
```

原因：缺少 dataset0.txt 文件，可能是以下情况：

1. 未创建 dataset0.txt 文件，如多输入模型；
2. pegasus_quantize.sh 和模型目录的层级关系不对。

解决方法：调整数据集文件与 inputmeta.yml 文件的 path 参数一致。pegasus_quantize.sh 和模型目录同级（目录和模型文件同名），或 pegasus_quantize.sh 在模型目录中。

错误信息 2：

```
File "numpy/core/fromnumeric.py", line 57, in _wrapfunc
ValueError: cannot reshape array of size 2304 into shape (3,1,256)
[5009] Failed to execute script 'pegasus' due to unhandled exception!
```

原因：batch 不为 1 时，不能在 dataset.txt 中配置一个或多个（3,1,256）的 npy 文件路径。

解决方法：把所有 batch 的数据放到一个 npy 文件里，path 中直接配置这个数据文件的路径，例如配置path: data_30_1_256.npy。详见《多输入模型配置》的第三种配置方法。

错误信息 3：

```
Traceback (most recent call last):
 File "pegasus.py", line 131, in <module>
...
 File "acuitylib/dataset/text_dataset.py", line 54, in batch
IndexError: list index out of range
[34252] Failed to execute script pegasus
```

原因：量化数据数量不对，常见于多输入模型。例如：三个输入，共用一个 dataset.txt，但 dataset.txt 每一行只配置了两个数据路径。

解决方法：检查 inputmeta.yml 文件的 path 参数和 dataset.txt 中的数据路径的数量。

错误信息 4：

```
D ********** Setup database (1) **********
Traceback (most recent call last):
  File "pegasus.py", line 131, in <module>
...
  File "acuitylib/dataset/base_dataset.py", line 100, in __init__
  File "acuitylib/dataset/text_dataset.py", line 11, in _build
  File "encodings/ascii.py", line 26, in decode
UnicodeDecodeError: 'ascii' codec can't decode byte 0x93 in position 0: ordinal not in range(128)
```

原因：解析 dataset.txt 文件失败，内有非法字符，可能 path 写的是 npy 文件。

解决方法：检查 inputmeta.yml 文件中的数据集的 tpye，修正为 NPY。

```
- path: data/data.npy
  type: NPY
```

错误信息 5：

演示示例 yolov5s-sim 在板端部署的效果如图所示：

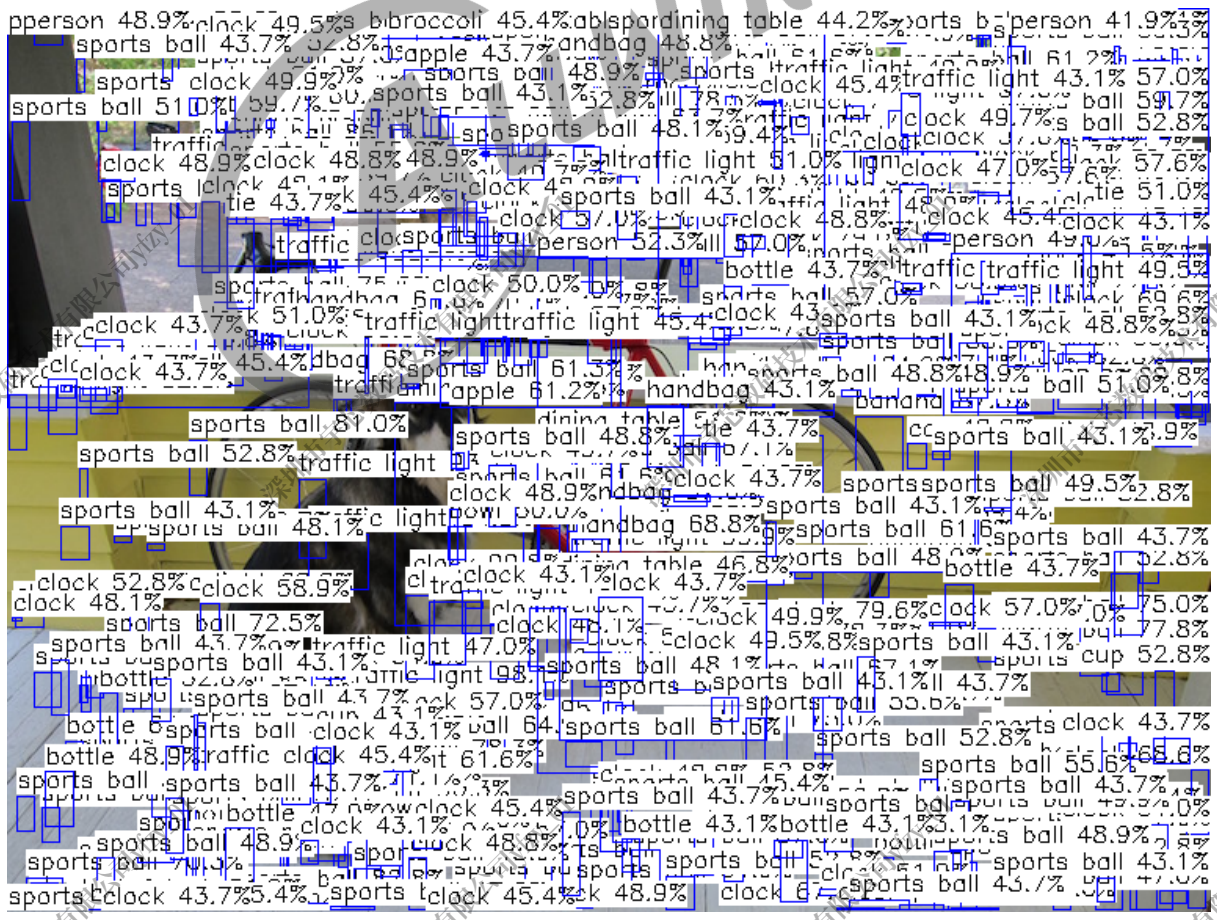

图 8-1: yolov5s 端侧检测结果异常示例

原因：xxx_inputmeta.yml 和 xxx_postprocess_file.yml 文件参数设置不正确。

解决方法：检查 xxx_inputmeta.yml 文件中的 mean，scale 参数是否设置正确；将 xxx_postprocess_file.yml 中的 add_postproc_node 参数设为 true，推理结果由 NPU 内部做反量化浮点输出。

二、多通道不同 scale 值

问题：在 Acuity Toolkit 6.21.1 之前，inputmeta.yml 的 scale 只支持设置一个值，会引入校准数据的输入偏差，影响算法量化效果，最终会影响算法量化后的推理精度。

解决方法：更新工具，或按照以下例子进行数据预处理。

例如某算法的图像 BGR 三个通道都有各自的 mean 与 scale(1/STD) 值：

```
- These are in BGR and are for ImageNet
MEANS = (103.94, 116.78, 123.68)
STD = (57.38, 57.12, 58.40)
```

第一步：参考如下 python 代码，将图片经预处理保存为 npy 文件：

```
img_path = './images'
mean = [103.94, 116.78, 123.68]
scale = [1.0/57.38,  1.0/57.12,  1.0/58.40]

img_file_list  = glob.glob(os.path.join(img_path, '*.jpg'))
for img_file in img_file_list:
    img0 = cv2.imread(img_file)  # BGR
    assert img0 is not None, 'Failed to load '  + img_path

    # resize
    new_shape = (550, 550)
    img = cv2.resize(img0, new_shape)

    pre_data = (img - mean) * scale

    npy_file_name = img_file.split('.jpg')[0]  + '.npy'
    np.save(npy_file_name, pre_data)
#上述保存的npy数据格式为nhwc，若算法需求为nchw格式，请做数据的通道变换；
```

第二步：dataset.txt 的图片路径改为 npy 文件路径，如：./images/dog.jpg 改为./images/dog.npy；

第三步：修正 inputmeta.yml 文件，mean 与 scale 值设置如下：

```
layout: nchw
shape:
- 1
- 3
- 550
- 550
fitting: scale
preprocess:
  reverse_channel: true
  mean:
  - 0
```

```
  - 0
  - 0
  scale: 1.0
```

三、混合量化参数格式问题

错误信息 1:

```
Traceback (most recent call last):
  File "/root/acuity-toolkit-whl-6.30.10/bin/pegasus.py", line 131, in <module>
    main(args)
  File "/root/acuity-toolkit-whl-6.30.10/bin/pegasus.py", line 108, in main
    ret = medusa.execute(args)
  File "acuitylib/app/medusa/commands.py", line 214, in acuitylib.app.medusa.commands.execute
  File "acuitylib/vsi_nn.py", line 589, in acuitylib.vsi_nn.VSInn.quantize
  File "acuitylib/app/medusa/quantization.py", line 153, in acuitylib.app.medusa.quantization.Quantization.run
  File "acuitylib/app/medusa/quantization.py", line 51, in acuitylib.app.medusa.quantization.Quantization.
        _run_quantization
  File "acuitylib/app/medusa/quantization.py", line 86, in acuitylib.app.medusa.quantization.Quantization.
        _quantize_net
  File "acuitylib/app/medusa/quantization.py", line 105, in acuitylib.app.medusa.quantization.Quantization.
        _generate_hybrid_table
  File "acuitylib/optimize/optimizer.py", line 299, in acuitylib.optimize.optimizer.Optimizer.apply
  File "acuitylib/optimize/rules/quantize/hybrid_insert_converter_layer.py", line 279, in acuitylib.optimize.rules.quantize
        .hybrid_insert_converter_layer.HybridInsertConverterLayer.apply
  File "acuitylib/optimize/rules/quantize/hybrid_insert_converter_layer.py", line 270, in acuitylib.optimize.rules.quantize
        .hybrid_insert_converter_layer.HybridInsertConverterLayer._insert_dc_for_hybrid_quantize
  File "acuitylib/utils.py", line 128, in acuitylib.utils.get_subgraph
TypeError: 'NoneType' object is not subscriptable
 SUCCESS
```

原因：hybrid_insert_converter_layer 时发生错误，customized_quantize_layers 参数格式不正确。

解决方法：customized_quantize_layers 参数中图层名字添加上单引号，特别是图层名字为纯数字，必须用单引号括起来。

错误信息 2:

```
ruamel.yaml.parser.ParserError: while parsing a block mapping
  in "<unicode string>", line 2, column 1:
    version: 2
    ^ (line: 2)
expected <block end>, but found '<scalar>'
  in "<unicode string>", line 2173, column 12:
     '122_193':float32
              ^ (line: 2173)

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/root/acuity-toolkit-whl-6.30.10/bin/pegasus.py", line 131, in <module>
    main(args)
  File "/root/acuity-toolkit-whl-6.30.10/bin/pegasus.py", line 108, in main
    ret = medusa.execute(args)
  File "acuitylib/app/medusa/commands.py", line 212, in acuitylib.app.medusa.commands.execute
  File "acuitylib/vsi_nn.py", line 236, in acuitylib.vsi_nn.VSInn.load_model_quantize
  File "acuitylib/acuitynet.py", line 403, in acuitylib.acuitynet.AcuityNet.load_quantize
```

```
 File "acuitylib/acuitylog.py", line 260, in acuitylib.acuitylog.AcuityLog.e
acuitylib.acuityerror.AcuityError: ('Exception:<class \'ruamel.yaml.parser.ParserError\'> while parsing a block mapping\
    n  in "<unicode string>", line 2, column 1:\n    version: 2\n    ^ (line: 2)\nexpected <block end>, but found \'<scalar
    >\'\n  in "<unicode string>", line 2173, column 12:\n     \'122_193\':float32\n           ^ (line: 2173) File:acuitynet.py
    Line:394', None)
SUCCESS
```

原因：customized_quantize_layers 参数格式不正确。

解决方法：customized_quantize_layers 参数冒号后面空一格。

错误信息 3:

```
W 09:39:40 Load file /workspace/demo_hybird/demo_hybird_pcq_float32.quantize failed.
E 09:39:40 Exception:<class 'ruamel.yaml.scanner.ScannerError> while scanning a simple key
 in "<unicode string>", line 67017, column 3:
    '1347_106':float32
    ^ (line: 67017)
could not find expected ':'
 in "<unicode string>", line 67018, column 3:
    '1791_84': float32
    ^ (line: 67018) File:acuitynet.py Line:399
```

原因：customized_quantize_layers 参数格式不正确。

解决方法：customized_quantize_layers 参数每行最前面需要空两格。

8.2 端侧部署问题

一、vpm_run 运行模型常见报错

错误信息 1:

```
failed to read sample.txt
failed batch is NULL

 Segmentation fault (core dumped)
```

原因：vmp_run 通过换行符逐行获取文件名，而该 sample.txt 的换行符为 Windows(CR LF)或\r\n。

解决方法：使用 notepad++，或dos2unix命令修改文件。

错误信息 2:

```
unsupported input file type=tenso.
error input file type
Segmentation fault (core dumped)
```

原因：vmp_run 以 [为界限，当最后一个模型没有 [时，会以\0 作为结束符。而通过换行符逐行获取文件名，而该 sample.txt 没有多空出一行。

解决方法：sample.txt 最后多空出一行。

错误信息 3：

```
Test output 0 failed: data mismatch. Output saved in file ./output_2.dat for further analysis.
```

原因：配置文件中有 golden 标签，会进行输出比对，但是不一致，就有失败信息提示。如果运行多个模型，会从此模型退出，不再执行。

解决方法：若想测试多个模型，则注释 golden 标签。

二、NPU 初始化失败

错误信息 1：

```
#用户应用程序运行vip_init(64*1024*1024)
gcvip_os_init, fail to mmap device vipcore, video memory size: 0x4000000
```

可能原因：若使用我们提供的中间库和 vpm_run 工具能正常运行，则排除硬件/驱动问题。可检查编译应用程序的工具链，若应用程序使用的工具链为 xxxx_arm-linux-gnueabihf，而 SDK 提供的中间件动态库的工具链是 toolchain-sunxi-glibc，会报此错。

解决方法：更换应用程序的编译工具链，最好采用相同的编译选项和调用约定。

错误信息 2：

```
#用户应用程序运行报错打印
[0x98a6U980]gcvip_os_call_kerne1[339], fail to ioctl vipcore, command[0]:CMD_INIT, status=-1
[0x98a6u980]gcvip_user_init[356]， fail tO call kernel for cmd init, status=-1.
[0X98a6u989]gcvip_init [4219]， failed tO user init

#内核报错打印
npu[a67][a67] DMA Address = 0x00000000
npu[a67][a67]   dmaLow   = 0x00000000
npu[a67][a67]   dmaHigh  = 0x00000000
npu[a67][a67] COMMAND BUF DUMP
npu[a67][a67] 63383000 : 0801006B FFFE0000 08010E12 00490000 08010E02 00000701 48000000 00000701
npu[a67][a67] 63383020 : 10000000 00000000
npu[a67][a67] failed to setup MMU, idleState=0x0
npu[a67][a67] core0 mmu enable fail for command mode
npu[a67][a67] fail to enable MMU, status=-1.
npu[a67][a67] fail to init core0, status=-1.
npu[a67][a67] fail to hardware initialize, status=-1.
npu[a67][a67] vipocre, failed to kernel call, command[0]: CMD_INIT, status=-1
npu[a67][a67] vipocre, failed to ioctl, command[0]: CMD_INIT
```

可能原因：此板子不支持 NPU，或者 NPU 上电异常导致 NPU 不可用，应用程序调用 vip_init() 接口失败。

解决方法：重新 insmod vipcore.ko，查看内核的 log 进行排查。

三、NPU Hang

现象：模型运行时，出现 “卡住” 状态，ctrl+c 和 ctrl+z 都无法退出。等待一段时间（约 20s）后，打印一堆信息就正常退出。

Log 关键信息：

```
failed to wait interrupt, timeout....
start device0 hang dump dump_finish=0
**********************************
******** VIPLite hang dump ********
```

NPU 运行超时，进入 Hang Debug 状态，打印超时的函数堆栈、当前 Hang 时各寄存器等的信息。此外，在当前目录还有一个新生成的viplite_hang_capture_xxxx_xxxx.log文件。

可能原因：

NPU 可能因为以下原因导致模型运行超时：

1. NBG 模型损坏（例如：OTA 升级后模型不完整）；
2. 模型本身错误（例如：PC 端仿真推理时模型都无法跑通）；
3. 应用程序错误（例如：应用程序中 API 使用错误）；
4. 资源受限（例如：与其他模块联调时资源获取失败）；
5. 驱动错误配置（例如：修改过驱动配置或代码）；
6. NPU 硬件卡死（例如：电压不足）；
7. 受其他模块影响（例如：DDR 异常导致 NPU 获取 buffer 失败）。

Hang Log 中有以下情况，表示 NPU 硬件未正常启动。

```
chip ver1    = 0x00000000 #为0
chip ver2    = 0x00000000 #为0
chip date    = 0x00000000 #为0
chip cid     = 0x00000000 #为0
```

Hang Log 中有以下情况，表示 NPU 未接受到运行的 cmd，需排查 NPU 能否正常访问 DDR 以写入/获取 CMD 内容。

```
dmaLow  = 0x00000000 #为0
dmaHigh  = 0x00000000 #为0
dmaState = 0x00000905
```

排查思路：

1. PC 端仿真推理时模型能否跑通并输出正确结果，在设备使用 vpm_run 工具测试模型能否正常跑通；
2. 切换模型（如 ai-sdk 中的 example）测试；
3. 检查 NPU 供电、其他模块是否正常。

若无法定位问题所在，请给我们提供以下文件：

1. 模型的 json 文件。
2. viplite_hang_capture_xxxx_xxxx.log文件（Hang 时在当前目录自动生成）。
3. 提高 DEBUG 级别的设备端串口 LOG（驱动源码 inc/gc_vip_common.h 修改 vpmdENABLE_DEBUG_LOG 为 4）。

解决方法：

1. 根据上文的可能原因依次排查模型/应用/驱动/硬件问题，找到对应的原因进行解决。
2. 特殊情况可调整 Hang 响应机制（请联系我们提供相应的中间件），例如：Hang 以后 recovery；Hang 以后进入 HANG DEBUG，但不生成 log，节省空间。

四、接口使用问题

NPU 应用程序崩溃，检查日志。

错误信息 1：

```
gcvip_query_input[4049], index=1 > network->fixed.header.input_count=1 error
gcvip_query_input[4066], failed to query input[1], property=7
gcvip_query_input[4049], index=2 > network->fixed.header.input_count=1 error
gcvip_query_input[4066], failed to query input[2], property=7
```

原因：模型只有 1 个输入（input_count=1），应用程序使用vip_query_input接口查询了 3 个输入的信息（index=1/index=2），因此报错。实际情况是查询输出信息时用错 vip_query_input。

解决方法：检查代码 vip_query_input/output 接口的 index 参数和接口使用，把查询输出信息的地方改用 vip_query_output。

8.3 模型精度问题

一、添加预处理节点后，板端的推理精度下降

现象：

inputmeta.yml 按照以下配置设置 RGB 输入后导出模型，板端推理输出与原始浮点模型的推理输出接近。

```
preproc_node_params:
  add_preproc_node: true
  preproc_type: IMAGE_RGB
```

inputmeta.yml 按照以下配置设置 NV12 输入后导出模型，板端推理输出与原始浮点模型的推理有偏差。

```
preproc_node_params:
  add_preproc_node: true
  preproc_type: IMAGE_NV12
```

原因：预处理节点 yuv2rgb 转换有两套公式，默认的不是 OpenCV 的标准转换公式。

解决方法：在导出 nb 文件前设置环境变量export VSI_NN_ENABLE_OCV_NV12=1，使用 OpenCV 的标准转换公式。重新导出 nb 文件即可（pegasus export 操作）。

8.4 模型内存问题

一、如何设置合适的内存大小为 vip_init 的参数

背景：vip_init 事先分配一片内存，NPU 优先使用该内存，不足时进行动态分配。避免分配过大，造成浪费；不宜过低，因为动态分配耗时。需要设置合适的初始化内存大小。

解决方法：

①、使用 nbinfo 分析模型内存占用情况，根据 Total Memory 消耗作为标准留出 2~5% 左右的冗余：

```
# ./nbinfo -m network_binary.nb
Memory Info
********************************************************************************
Total Video Memory (bytes):                16507392
Total System Memory (bytes):                  57796
********************************************************************************
```

预计消耗内存为 16507392+57796=16565188，可参考此值留出冗余配置 vip_init 的值。

②、如果需要获取精确的内存消耗值，可以事先分配足够大的内存，并在驱动内存分配处添加代码进行统计以计算模型实际所需内存，具体步骤如下：

a、修改驱动代码，可在三处添加打印监视内存堆使用情况：

每次内存分配前，打印内存堆的余量以及待分配内存大小，修改 gc_vip_video_memory.c 文件：

```
#if vpmdENABLE_VIDEO_MEMORY_HEAP
+   PRINTK("heap free size = %d , need memory = %d\n", context->video_mem_heap.free_bytes, GCVIP_ALIGN(size,
        align));
    if ((gckvip_heap_capability(&context->video_mem_heap) & alloc_flag) == alloc_flag) {
```

内存堆分配时，查看内存堆分配的内存大小以及内存堆余量，修改 gc_vip_kernel_heap.c 文件：

```
        heap->free_bytes -= pos->size;
+        PRINTK("memory heap allocate %d sizes memory,free heap update %d sizes\n", pos->size, heap->free_bytes);
```

动态分配时，查看动态分配的内存大小，修改 linux/gc_vip_kernel_allocator.c 文件：

```
    node->mem_ion_dyn.size = node->size;
+    PRINTK("use dyn allocate %d sizes memory\n", node->mem_ion_dyn.size);
    gcOnError(aw_vip_mem_alloc(kdriver->device, &node->mem_ion_dyn, "vip_dyn"));
```

b、修改例程，然后运行例程：

使用 nbinfo 工具获取一个大小，或者将 vip_init 的值设为百分百足够模型内存堆完成所有模型分配的大小【前提是板端内存足够大】，例如改为申请 30M 大小的内存，修改 vpm_run 例程：

```
-   status = vip_init(1*1024*1024);
+   status = vip_init(30*1024*1024);
```

编译后通过 adb push 方法推到板端，运行例程：

```
./vpm_run sample.txt
```

c、统计结果，计算正确内存堆大小：

查看打印 LOG，只需要关注最开始内存堆分配内存大小，以及最后一次内存堆分配时的内存堆余量即可。

```
## 内存堆分配大小，分配了一块 31457280 大小的内存堆：
[ 9546.855112] enter aw vip mem alloc size 31457280 vip_heap
.....
.....
## 内存堆分配640大小的内存时，会以4K对齐：
[ 9546.956715] npu[5e3][5e3] heap free size = 31426560 , need memory = 640
[ 9546.964234] npu[5e3][5e3] memory heap allocate 4096 sizes memory,free heap update 31422464 sizes
.....
.....
## 最后一次内存堆分配，内存堆余量为 14862336：
[ 9547.294743] npu[5e3][5e3] memory heap allocate 45056 sizes memory,free heap update 14862336 sizes
```

两者相减可得：31457280-14862336=16594944，随即调整 vpm_run 的 vip_init 的值以验证。

说明

有时调整内存堆值重新编译运行例程，会发现最后一次内存堆分配仍有余量，需要进一步根据 LOG 中的内存堆余量进行调整。

二、如何使用预留内存

背景：模型运行前的 create、prepare 操作申请资源会占用一定时间，而预留内存不释放，无需反复申请，可节省时间。或特殊情况需 NPU 在某段内存中运行。

解决方法：

NPU 驱动代码中，调用 request_mem_region 接口用于申请并访问预留内存：

```
region = request_mem_region(kdriver->cpu_physical, kdriver->vip_memsize, "vip_reserved");
```

只需配置对应的宏以及 NPU 驱动配置，并在 dts 中配置预留内存即可。

具体步骤：

①、配置 dts 文件以预留内存：

配置 dts 文件的 reserved：

```
reserved-memory {
    #address-cells = <2>;
    #size-cells = <2>;
    ranges;

    bl31 {
        reg = <0x0 0x48000000 0x0 0x01000000>;
    };
    /*配置NPU模块预留内存地址及大小*/
    vip_reserved {
        reg = <0x0 0x50000000 0x0 0x01000000>;
        no-map;
    };
};
```

②、配置 USE_LINUX_RESERVE_MEM 宏：

修改 Kbuild 添加以下代码打开预留内存配置：

```
export USE_LINUX_RESERVE_MEM=1
```

③、修改 linux/allwinner/gc_vip_platform_config.h 文件配置对上 dts 的 reserved 内存：

```
#if defined (USE_LINUX_RESERVE_MEM)
#define VIDEO_MEMORY_HEAP_SIZE       (16 << 20)
#else
#define VIDEO_MEMORY_HEAP_SIZE       (32 << 20)
#endif
/* defalut base pyhsical address of video memory heap */
#define VIDEO_MEMORY_HEAP_BASE_ADDRESS  0x50000000
```

将 Size 以及 Base 都对应上 dts 文件中的预留内存地址及大小。

④、重新编译加载驱动，查看打印：

```
[ 4383.316482] npu[55d2][55d2] contiguousSize     0x01000000
[ 4383.322628] npu[55d2][55d2] contiguousBase     0x50000000
```

然后运行例程，查看内存分配时使用的内存地址：

```
[  92.690323] npu[155][155] video memory heap base physical=0x50000000, user logical=0x0000007fa6659000, kernel
        logical=0xffffffc00c000000, size=0x4000000 bytes
.........
[  94.226776] npu[155][155] video memory heap_alloc, physical=0x50000000, virtual=0xfff00000, kernel logical=0
        xffffffc00c000000, user logical=0x0000007fa6659000, handle=0xffffff80c8a69900, size=0x280, align=0x40, allo_flag=0
        x611
```

可见，已经成功预留内存并使用对应 address 以及 size 的内存。

8.5 驱动问题

1）如何查看当前 NPU 模块的时钟频率：

```
$ mount -t debugfs none /sys/kernel/debug
$ cat sys/kernel/debug/clk/clk_summary | grep -i npu
   pll-npu-4x                2     2     0  1200000000         0     0  50000      Y
     pll-npu-1x              0     0     0   300000000         0     0  50000      Y
     pll-npu-2x              1     1     0   600000000         0     0  50000      Y
       npu                 1     2     0   600000000         0     0  50000      Y
#此时NPU模块时钟频率为600MHz
```

2）NPU 模块时钟频率的配置方法

有以下方法修改 NPU 模块时钟频率：

1. 修改 dts 设备树文件；
2. 修改驱动源码，直接写固定的时钟频率；
3. 当驱动编译进内核又不想重新烧固件时，可以通过 uboot 的方式修改；

4. CCU_DEBUG 的方式。

依次介绍上述配置方法：

①、修改 dts 设备树文件：

V85X 修改 clk-sun8iw21_tbl.c 添加支持对应频点后，修改 clock-frequency 参数；其他平台修改 dtsi 文件中的 npu_opp_table 节点，先加载驱动确认当前机子 VF 值后，修改设备树 npu_opp_table 节点对应的 VF 表中的时钟频率/电压值（并不推荐修改机子 VF 表）

②、修改驱动源码，直接写固定的时钟频率：

将时钟频率写固定后，驱动加载自动配置时钟频率为指定值，直接修改驱动源码后重新编译：

```
--- a/drivers/npu/aw_nna_vip/linux/allwinner/gc_vip_kernel_drv_platform.c
+++ b/drivers/npu/aw_nna_vip/linux/allwinner/gc_vip_kernel_drv_platform.c
@@ -537,7 +537,7 @@ vip_int32_t gckvip_drv_adjust_param(
                pr_err("vipcore: Couldn't deassert NPU RST\n");
                return -EBUSY;
        }
+        mod_clk = 696000000;
        /* Set NPU CLK */
```

③、当驱动编译进内核又不想重新烧固件时，可以通过 uboot 的方式修改：

```
reboot uboot
#等待其处于uboot中后：
fdt list /soc/npu
#V85X修改clock-frequency，其他平台修改/soc/npu_opp_table节点
fdt set /soc/npu clock-frequency <696000000>
fdt list /soc/npu
save
boot
```

④、CCU_DEBUG 的方式：

```
mount -t debugfs none /sys/kernel/debug
echo setrate > /sys/kernel/debug/ccudbg/command
echo npu > /sys/kernel/debug/ccudbg/name
#Set NPU CLK
echo 696000000 > /sys/kernel/debug/ccudbg/param
echo 1 > /sys/kernel/debug/ccudbg/start
cat /sys/kernel/debug/clk/clk_summary |grep -i npu
```

可以将 NPU 模块时钟频率配置为 696MHz。在 v2.0.3 驱动，只能使用 clk_freq 进行临时修改频率，详见《端侧查询 DDR、NPU 等 Debug 信息》。

3）VIPLite 与 Unified 驱动能否同时使用?

VIPLite/Unified 驱动分别为不同的 NPU 方案，两者不能同时挂载。【挂载 Unified 后需要重启机子才能挂载 VIPLite】

4）如何开启 DEBUG 打印

方法一：

在 Kbuild 中添加export DEBUG=1即可查看所有的内核打印信息方便调试：

```
 export AUTO_CORRECT_CONFLICTS=1
+export DEBUG=1
```

方法二：

修改 NPU 的头文件中的 vpmdENABLE_DEBUG_LOG 宏为 4：

平台	头文件
V85x、R853	lichee/linux-4.9/include/npu/gc_vip_common.h
MR527、T527、AI985	bsp/drivers/npu/aw_nna_vip/inc/gc_vip_common.h
MR536、T536、A733、T736	bsp/drivers/npu/aw_nna_vip/vip2/inc/vip_lite_config.h

```
--- a/drivers/npu/aw_nna_vip/inc/gc_vip_common.h
+++ b/drivers/npu/aw_nna_vip/inc/gc_vip_common.h
@@ -319,7 +319,7 @@ provide richer profiling information of the internal NN/TP. This feature needs h
     the rt_log value is 1 by default.
 */
 #ifndef vpmdENABLE_DEBUG_LOG
-#define vpmdENABLE_DEBUG_LOG            1
+#define vpmdENABLE_DEBUG_LOG            4
 #endif
```

修改完成后重新编译驱动并加载。

5）如何查看设备中断信息：

```
/ # cat /proc/interrupts | grep -i vipcore
108:      88      0      0      0      0      0      0      0  wakeupgen 199 Level    vipcore_0
##Unified版本将vipcore换为galcore
```

6）驱动卸载失败

```
root@TinaLinux:/# rmmod vipcore
unloading the module failed
```

原因：有进程正在使用驱动，导致卸载失败。

解决办法：使用 “ ls -l /proc//fd/ | grep vipcore ” 或者 “ lsof | grep vipcore ” 查看哪些进程占用 vipcore，并使用 “kill -9 id” 结束相关进程，最后再次卸载驱动即可。

著作权声明

本文档及内容受著作权法保护，其著作权由珠海全志科技股份有限公司（“全志”）拥有并保留一切权利。

本文档是全志的原创作品和版权财产，未经全志书面许可，任何单位和个人不得擅自摘抄、复制、修改、发表或传播本文档内容的部分或全部，且不得以任何形式传播。

商标声明





（不完全列举）均为珠海全志科技股份有限公司的商标或者注册商标。在本文档描述的产品中出现的其它商标，产品名称，和服务名称，均由其各自所有人拥有。

免责声明

您购买的产品、服务或特性应受您与珠海全志科技股份有限公司（“全志”）之间签署的商业合同和条款的约束。本文档中描述的全部或部分产品、服务或特性可能不在您所购买或使用的范围内。使用前请认真阅读合同条款和相关说明，并严格遵循本文档的使用说明。您将自行承担任何不当使用行为（包括但不限于如超压，超频，超温使用）造成的不利后果，全志概不负责。

本文档作为使用指导仅供参考。由于产品版本升级或其他原因，本文档内容有可能修改，如有变更，恕不另行通知。全志尽全力在本文档中提供准确的信息，但并不确保内容完全没有错误，因使用本文档而发生损害（包括但不限于间接的、偶然的、特殊的损失）或发生侵犯第三方权利事件，全志概不负责。本文档中的所有陈述、信息和建议并不构成任何明示或暗示的保证或承诺。

本文档未以明示或暗示或其他方式授予全志的任何专利或知识产权。在您实施方案或使用产品的过程中，可能需要获得第三方的权利许可。请您自行向第三方权利人获取相关的许可。全志不承担也不代为支付任何关于获取第三方许可的许可费或版税（专利税）。全志不对您所使用的第三方许可技术做出任何保证、赔偿或承担其他义务。