Tina_Linux_NPU_Lenet 模型之从训练到端侧部署

版本号: 1.0

发布日期: 2021.07.21

版本历史

版本号	日期	制/修订人	内容描述
1.0	2021.07.21	PDC	NPU Lenet 模型部署实战

目　录

插　图

1 前言

1.1 读者对象

本文档（本指南）主要适用于以下人员：

- 技术支持工程师
- 软件开发工程师
- AI 应用案客户

1.2 约定

1.2.1 符号约定

本文中可能出现的符号如下：

警告

警告

技巧

1. 技巧
2. 小常识

说明

说明

2 正文

2.1 NPU 开发简介

- 支持 int8/uint8/int16 量化精度，运算性能可达 1TOPS.
- 相较于 GPU 作为 AI 运算单元的大型芯片方案，功耗不到 GPU 所需要的 1%.
- 可直接导入 Caffe, TensorFlow, Onnx, TFLite，Keras, Darknet, pyTorch 等模型格式.
- 提供 AI 开发工具：支持模型快速转换、支持开发板端侧转换 API、支持 TensorFlow, TF Lite, Caffe, ONNX, Darknet, pyTorch 等模型.
- 提供 AI 应用开发接口：提供 NPU 跨平台 API.
- 部署工具支持离线量化 (后训练量化,PTQ) 和量化感知训练 (QAT) 两类模型的导入，对于 QAT 训练得到的.tflite,onnx 格式模型，在模型 import 阶段会根据原生模型中的量化描述生成量化表文件。

2.2 开发流程

NPU 开发完整的流程如下图所示：

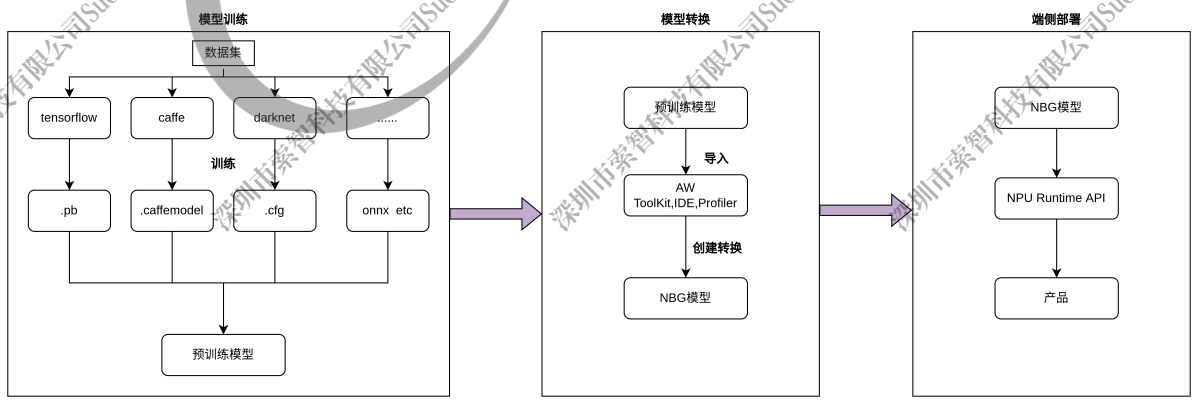

图 2-1：npu_1.png

3 Lenet 模型简介

Lenet 是一系列网络的合称，包括 Lenet1 - Lenet5，由 Yann LeCun 等人在 1990 年《Hand-written Digit Recognition with a Back-Propagation Network》中提出，是卷积神经网络的 HelloWorld。这里就以 lenet 为例介绍 AI model 在 tina 平台上部署从训练到端侧运行的全部过程。

细分环节包括，模型训练，模型导入，模型量化，模型推理，模型导出，模型仿真，模型 profile，模型端侧部署几个部分. 用一幅图表示如下：

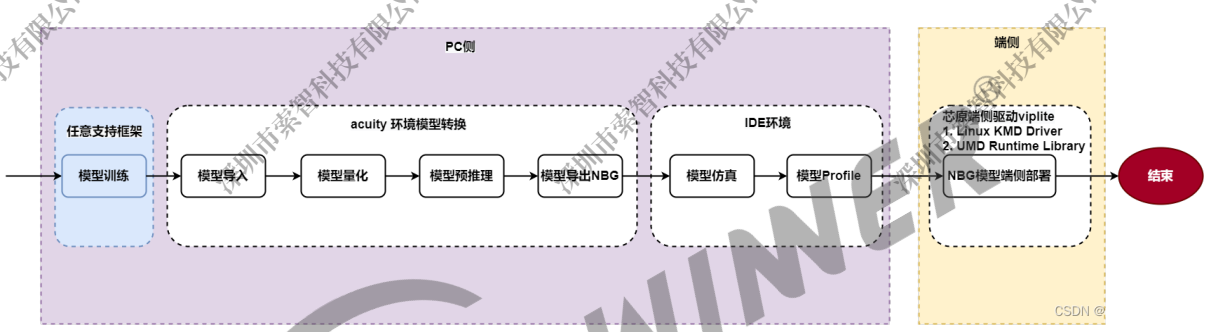

图 3-1: npu 部署流程

接下来从头开始介绍。

3.1 模型训练

本例中使用 keras 框架编写并训练 lenet5 网络，训练完成后，导出 h5 格式的模型文件,acuity tools 原生支持 H5 格式.

模型结构：

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_12 (Conv2D)           (None, 24, 24, 6)         156
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 12, 12, 6)         0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 8, 8, 16)          2416
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 4, 4, 16)          0
_________________________________________________________________
flatten_6 (Flatten)          (None, 256)               0
_________________________________________________________________
dense_18 (Dense)             (None, 120)               30840
_________________________________________________________________
dense_19 (Dense)             (None, 84)                10164
_________________________________________________________________
dense_20 (Dense)             (None, 10)                850
=================================================================
Total params: 44,426
Trainable params: 44,426
Non-trainable params: 0
```

图 3-2: lenet5 模型结构

训练完成后，观察精度是否满足训练目标要求，本例精度达到了%97, 可以拿来部署说明问题:

```
313/313 [==============================] - 1s 3ms/step - loss: 0.0784 - accuracy: 0.9759
0.07843624800443649 0.9758999943733215
```

图 3-3: lenet5 训练结果

输出保存模型为 lenet.h5,

```
(base) caozilong@caozilong-Vostro-3268:~$ ls -lh lenet*
-rw-rw-r-- 1 caozilong caozilong 568K 10月 27 22:20 lenet.h5
-rw-rw-r-- 1 caozilong caozilong 196K 10月 27 22:20 lenet_netron.h5
(base) caozilong@caozilong-Vostro-3268:~$
```


图 3-4: lenet 模型文件

使用 netron 查看模型结构:

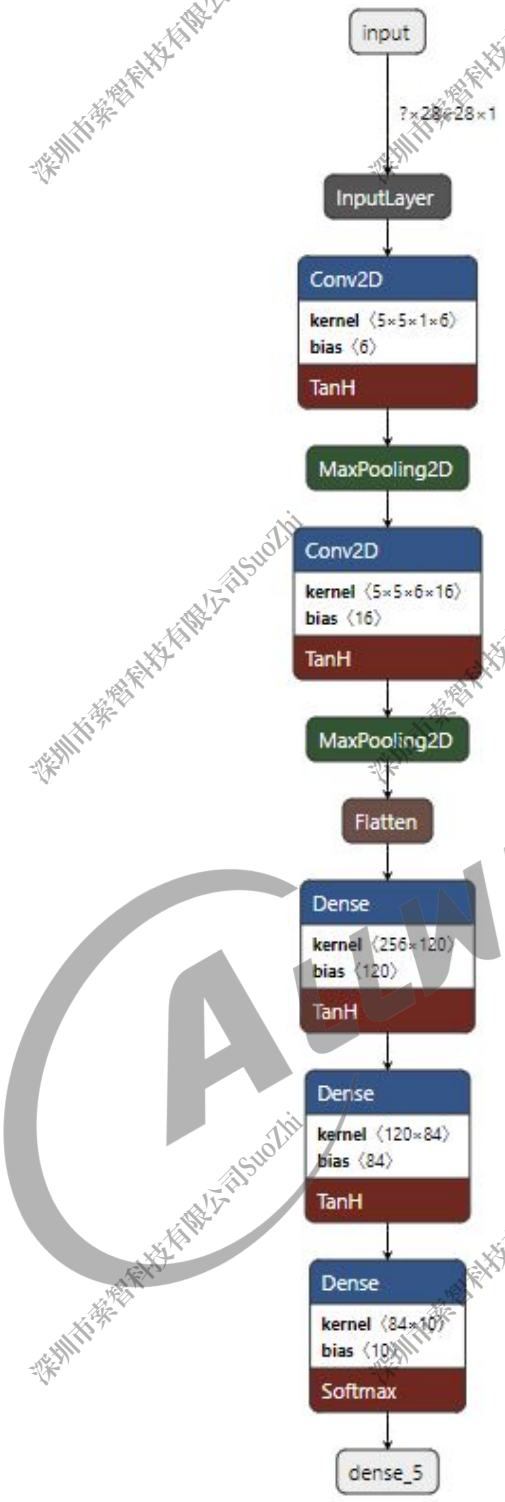

图 3-5: lenet 模型结构查看

至此，我们的原生模型已经产生，接下来就可以进行 PC 侧以及端侧的部署了，下一个环节是模型导入.

3.2 模型导入

在进行导入操作前，先看一下部署目录的结构：

如下图所示，其中 data 目录的图像来源于 mnist 数据集，作用是用来作为后训练量化的数据输入，用于给量化算法提供输入参考，从而获知实际场景的数据输入分布.dataset.txt 则是对 data 目录的引用，工具会通过 dataset.txt 文件查找到 data 目录中的每张图片。

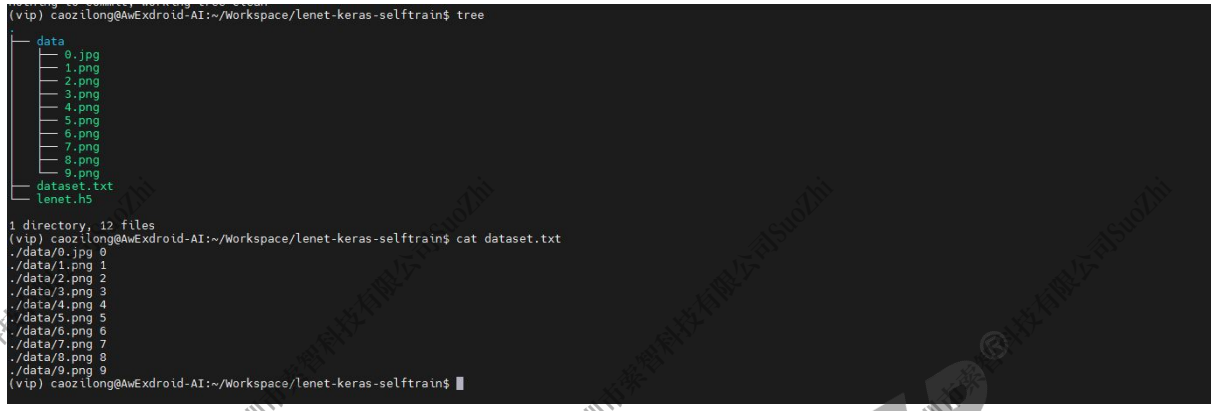

图 3-6: 部署目录结构

数据集：

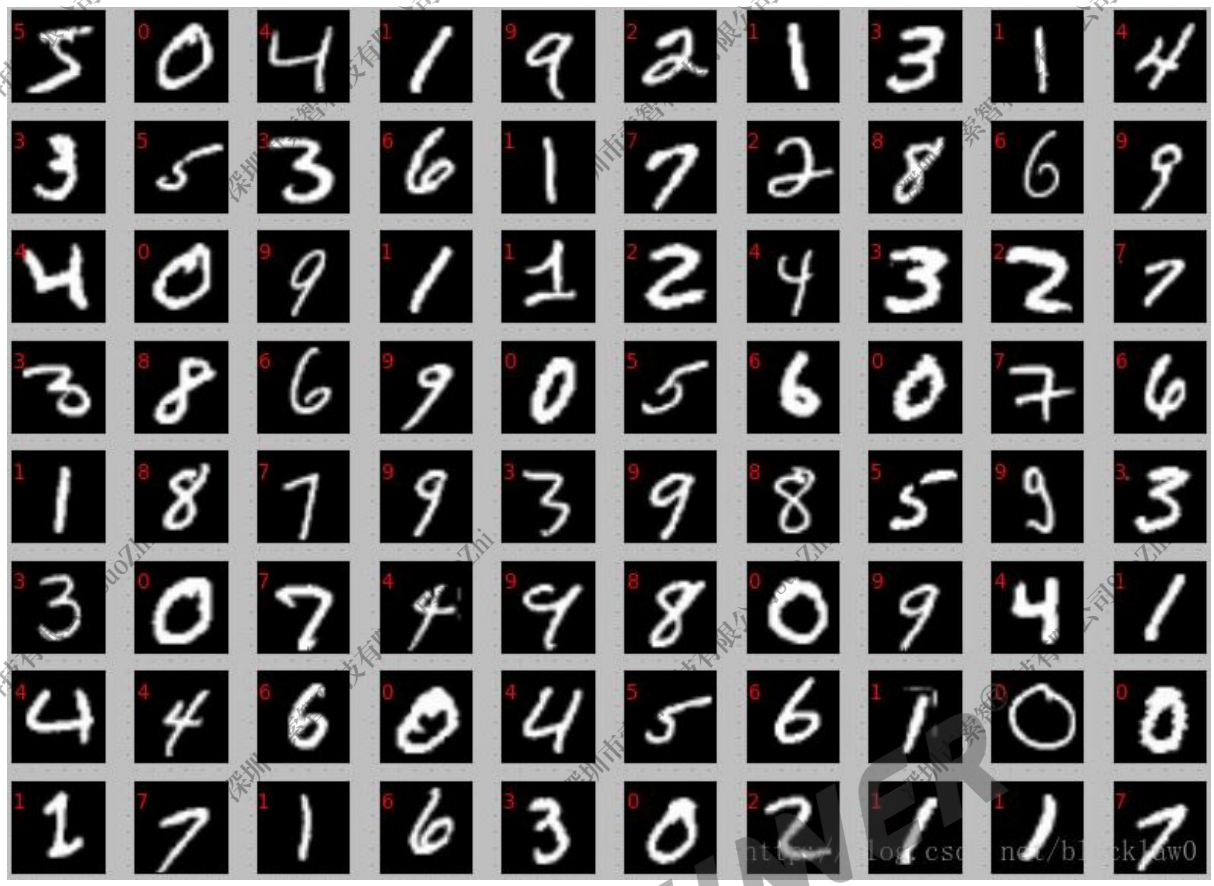

图 3-7: 量化参考图像数据集

3.3 导入模型

使用芯原提供的 acuity tool 中的 pegasus 工具进行模型的导入.

```
pegasus import onnx --model yolact-sim.onnx --output-model yolact-sim.json --output-data yolact-sim
.data
```

导入模型的目的是将开放模型转换为符合 VIP 模型网络描述文件 (.json) 和权重文件 (.data)

图 3-8: npu_import

接下来进行 lenet.h5 模型导入.

```
pegasus import keras --model lenet.h5 --output-data lenet.data --output-model lenet.json
```

执行成功后，可以看到目录中多了 lenet.json 和 lenet.data 文件，它们分别是符合芯原格式的

模型结构文件和模型权重文件.

```
D Optimizing network with force_1d_tensor, special_clip_to_relun, swapper, merge_duplicate_quantize_dequantize, batchmatmul_add_to_fc, merge_layer, resize_nearest_transfor
mer, auto_fill_multiply, compute_gather_negative, auto_fill_zero_bias, proposal_opt_import, extend_gather_to_gather_reshape, extend_stack_to_stack_reshape
D Optimizing network with auto_fill_bn, auto_fill_l2normalizescale, auto_fill_instancenormalize, merge_layer
D Optimizing network with special_add_to_conv2d
D Optimizing network with conv2d_big_kernel_size_transform
D Optimizing network with auto_fill_bn, auto_fill_l2normalizescale, auto_fill_instancenormalize, merge_layer
D Optimizing network with special_add_to_conv2d
D Optimizing network with auto_fill_zero_bias, auto_fill_tf_quantize, align_quantize, broadcast_quantize, qnt_adjust_coef, qnt_adjust_param
I End importing keras...
I Dump net to lenet.json
I Save net to lenet.data
I ---------------Error(0),Warning(0)---------------
(vip) caozilong@AwExdroid-AI:~/Workspace/lenet-keras-selftrain$ tree
.
├── data
│   ├── 0.jpg
│   ├── 1.png
│   ├── 2.png
│   ├── 3.png
│   ├── 4.png
│   ├── 5.png
│   ├── 6.png
│   ├── 7.png
│   ├── 8.png
│   └── 9.png
├── dataset.txt
├── lenet.data
├── lenet.h5
└── lenet.json

1 directory, 14 files
(vip) caozilong@AwExdroid-AI:~/Workspace/lenet-keras-selftrain$
```

图 3-9: 模型导入

导入阶段，pegasus 工具也会对模型结构进行解析并输出：

```
2022-02-24 17:31:57.310769: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): Host, Default Version
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 24, 24, 6)         156
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 6)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 16)          2416
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 4, 4, 16)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 256)               0
_________________________________________________________________
dense_3 (Dense)              (None, 120)               30840
_________________________________________________________________
dense_4 (Dense)              (None, 84)                10164
_________________________________________________________________
dense_5 (Dense)              (None, 10)                850
=================================================================
Total params: 44,426
Trainable params: 44,426
Non-trainable params: 0
_________________________________________________________________
I build output layer attach_dense_5/Softmax:out0
I Try match dense_5
I Try match dense_4
I Try match dense_3
I Try match flatten_1
I Try match max_pooling2d_3
I Try match conv2d_3
I Try match max_pooling2d_2
I Try match conv2d_2
I build input layer conv2d_2_input
```

图 3-10: 模型结构描述

导入阶段到这里还没有完，我们需要生成对网络的输入输出描述文件，这也是 acuity tool 工具要求的，输入输出描述是 YML 格式的文本文件，后面我们将通过修改 YML 文件来对模型参数，输入/输出 tensor 格式等信息进行配置.

3.3.1 创建 input/output YML 文件

YML 文件对网络的输入和输出进行描述，比如输入图像的形状，归一化系数 (均值，零点)，图像格式，输出 tensor 的输出格式，后处理方式等等，命令如下：

```
pegasus generate inputmeta --model lenet.json --input-meta-output lenet-inputmeta.yml      pegasus
generate postprocess-file --model lenet.json --postprocess-file-output lenet-postprocess-file.yml
```

命令成功执行后，目录中又多了两个文件，分别是 input 和 output YML 文件。

```
(vip) caozilong@AwExdroid-AI:~/Workspace/lenet-keras-selftrain$ tree
.
├── data
│   ├── 0.jpg
│   ├── 1.png
│   ├── 2.png
│   ├── 3.png
│   ├── 4.png
│   ├── 5.png
│   ├── 6.png
│   ├── 7.png
│   ├── 8.png
│   └── 9.png
├── dataset.txt
├── lenet.data
├── lenet.h5
├── lenet-inputmeta.yml
├── lenet.json
└── lenet-postprocess-file.yml

1 directory, 16 files
(vip) caozilong@AwExdroid-AI:~/Workspace/lenet-keras-selftrain$
(vip) caozilong@AwExdroid-AI:~/Workspace/lenet-keras-selftrain$
```

图 3-11: YML 文件生成

至此，模型导入阶段的工作才算全部完成，从这里开始，模型已经被转成了芯原格式的模型文件。后续步骤已经和原生模型没有太大关系了.

在执行下一步的模型量化前，我们需要修改 input yml 文件的 scale，mean 参数，使其和训练时的参数保持一致.

训练代码中的均值和 scale 为：

```
print(y_train)
x_train /= 255
x_test /= 255

from keras.utils import np_utils
y_train_new = np_utils.to_categorical(num_classes=10,y=y_train)
print(y_train_new)
y_test_new = np_utils.to_categorical(num_classes=10,y=y_test)
```

图 3-12: 训练代码中的均值和 scale

根据代码，均值 mean = 0, scale 为 1/255 = 0.0039.

修改 YML 为对应值：

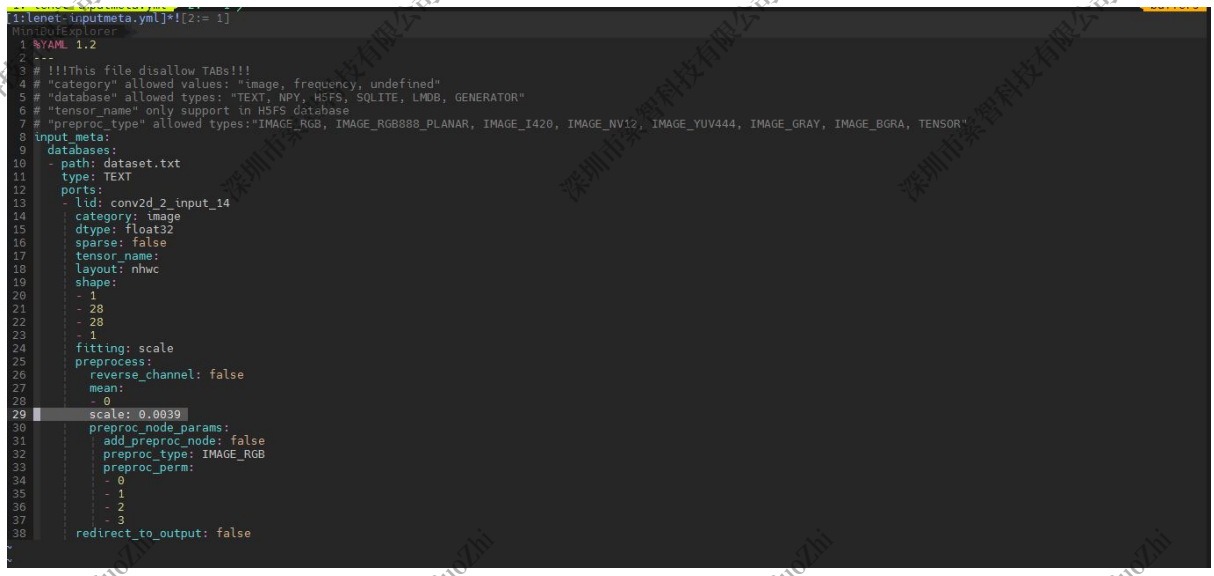

图 3-13: 修改 input YML Scale 参数

3.4 模型量化

此款 NPU 支持 uint8,int8,int16 三种量化类型，基于推理速度和精度折衷的考虑，我们用 int8，非对称量化模式，量化命令如下：

```
pegasus quantize --model lenet.json  --model-data lenet.data --batch-size 1 --device CPU --with-
input-meta lenet-inputmeta.yml --rebuild --model-quantize lenet.quantilize --quantizer asymmetric_affine
 --qtype uint8
```

命令执行后，工程目录下可以看到新创建的量化表文件文件

图 3-14: 量化表文件

3.5 模型预推理

```
pegasus inference --model lenet.json --model-data lenet.data --batch-size 1 --dtype quantized --
model-quantize lenet.quantilize --device CPU --with-input-meta lenet-inputmeta.yml --postprocess-
```

```
file lenet-postprocess-file.yml --iterations 10
```

参数中，`--batch-size 1`表示每次推理处理 1 张图像，我们 dataset 目录中包含了 10 张图像，所以要运行 10 次处理完毕，这也是后面参数`--iterations 10`的作用.

成功运行后，目录中多出了 20 个文本格式的.tensor 文件，文件中保存的是每张图像的输入和输出的 tensor 数据，默认情况下，只对输入输出 tensor 进行保存，你可以通过下面命令将每层的 tensor 都保存下来:

```
pegasus dump --model lenet.json --model-data lenet.data --with-input-meta lenet-inputmeta.yml
```

推理结束后创建的 tensor 文件：

图 3-15: tensor 输出

我们从命令的输出来看，也可以看出推理是否正确, 我们以前六个为例，可以看到 top5 输出中，每次概率最高的分别是 0,1,2,3,4 ....，和我们 dataset.txt 文件实际输入的图像数据顺序是相符的。

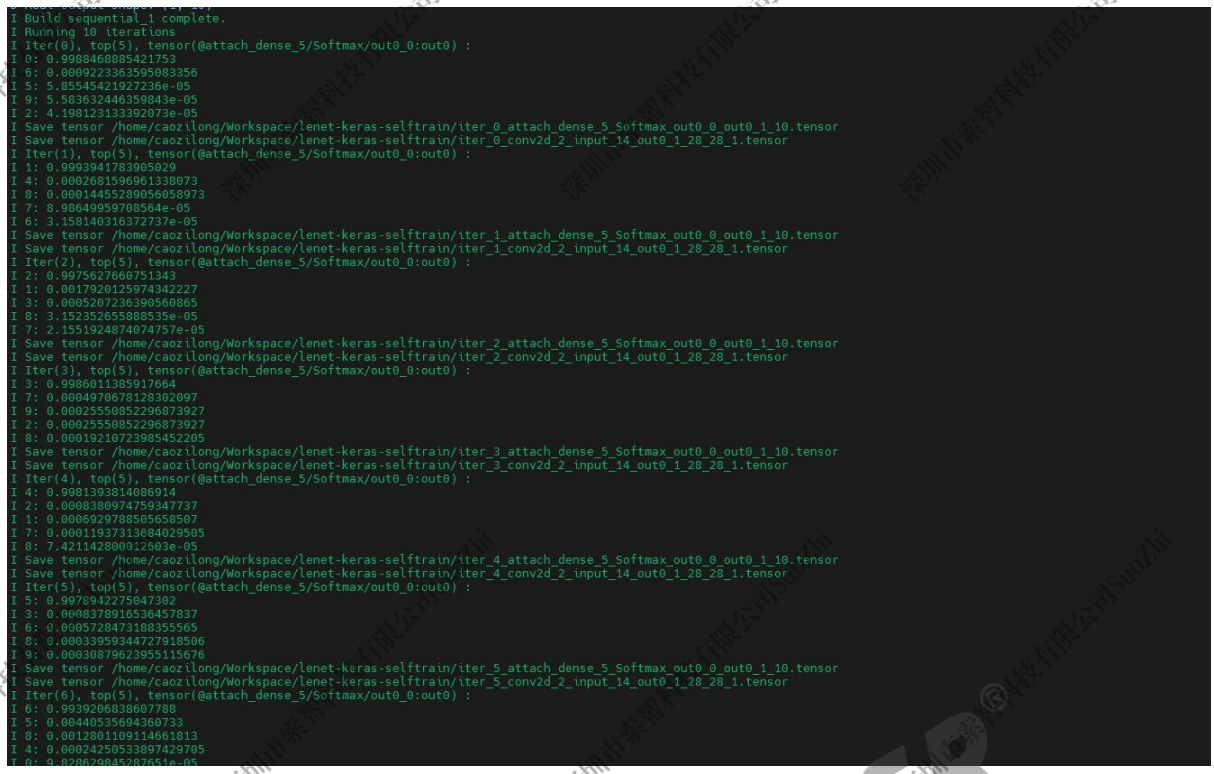

图 3-16: 部署推理过程输出

输出 tensor 则为最后一层的 softmax 输出，也就是分别为数字 0-9 的概率，以第 9 张图像的输出 tensor 为例，概率最大的是 9, 如下图：

图 3-17: softmax 输出

输入则是输入图像正则化后的浮点数据：

图 3-18: 输入 tensor

至此模型预推理阶段结束，进入下一步模型导出阶段.

3.6 导出代码和 NBG 文件

导出代码的命令如下，两次命令的区别只有选项 `--pack-nbg-unify`和`--pack-nbg-viplite`，其余完全相同。其中`--pack-nbg-unify`生成的是仿真侧的代码，而`--pack-nbg-viplite`则会生成在端侧运行部署的代码，两条命令分别执行一遍。

```
pegasus export ovxlib --model lenet.json --model-data lenet.data --dtype quantized --model-quantize
 lenet.quantilize --batch-size 1 --save-fused-graph --target-ide-project 'linux64' --with-input-
meta lenet-inputmeta.yml --postprocess-file lenet-postprocess-file.yml --output-path ovxlib/lenet/
lenet --pack-nbg-unify --optimize "VIP9000PICO_PID0XEE" --viv-sdk ${VIV_SDK}

pegasus export ovxlib --model lenet.json --model-data lenet.data --dtype quantized --model-quantize
 lenet.quantilize --batch-size 1 --save-fused-graph --target-ide-project 'linux64' --with-input-
meta lenet-inputmeta.yml --postprocess-file lenet-postprocess-file.yml --output-path ovxlib/lenet/
lenet --pack-nbg-viplite --optimize "VIP9000PICO_PID0XEE" --viv-sdk ${VIV_SDK}
```

执行结束后，工程代码和 NBG 文件都已经生成了:

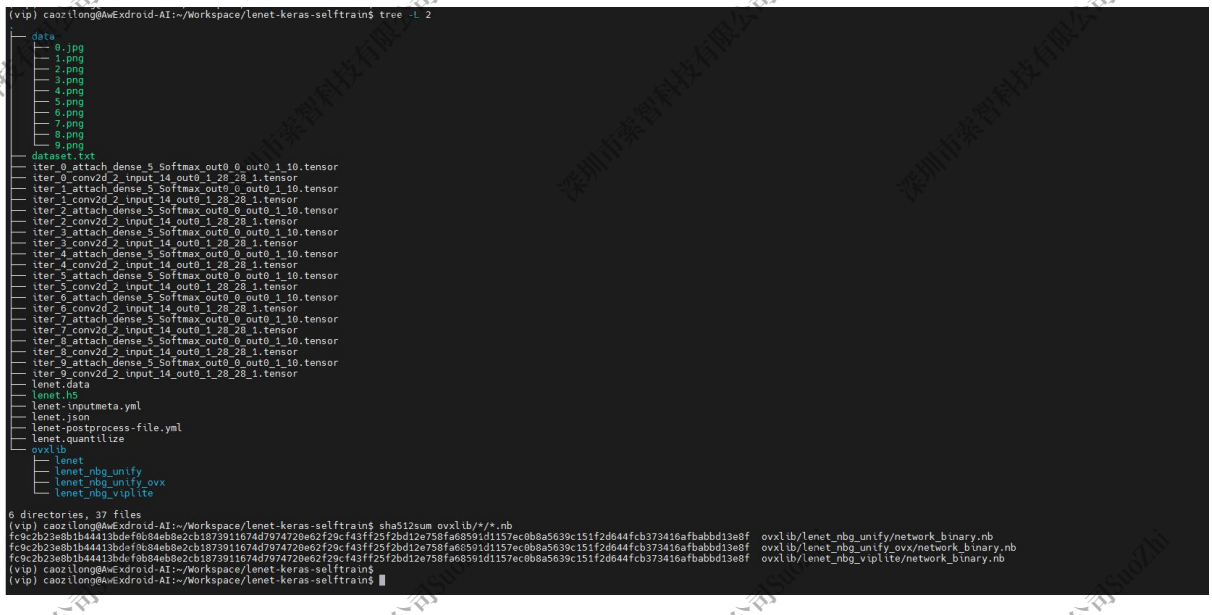

图 3-19: 导出模型和工程代码

导出的 NBG 文件可以投入到 NPU 中运行.

ovxlib/lenet/工程用于仿真和 profile.

ovxlib/lenet_nbg_viplite/可以用部署在端侧运行.

ovxlib/lenet_nbg_unify 和 ovxlib/lenet_nbg_unify_ovx 暂时可以不用理会，没有用处。

接下来，进入模型仿真环节.

3.7 模型仿真

上文说到，ovxlib/lenet/是用于仿真和 profile 分析的工程，下面我们启动 IDE 运行仿真.

3.7.1 启动 IDE

启动 IDE 的命令如下:

```
~/VeriSilicon/VivanteIDE5.5.0/ide/vivanteide5.5.0
```

启动时，首先选择一个工作目录用于保存仿真工程:

图 3-20: IDE 仿真

3.7.2 导入 ovxlib/lenet 工程

IDE 中，选择`File->Import->General选项卡->Existing Projects into Workspace`

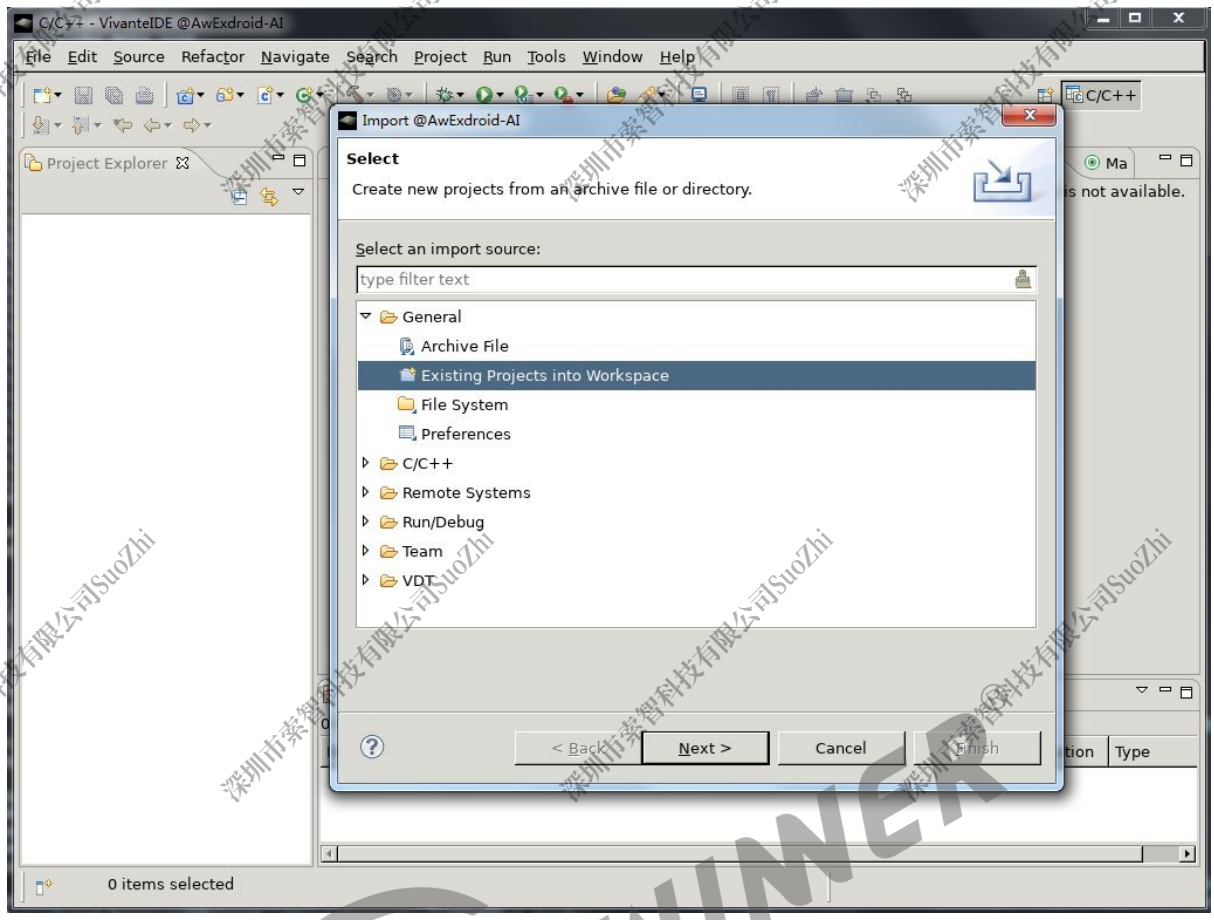

图 3-21: npu_prj_import

之后选择模型导出阶段创建的工程目录 ovxlib/lenet/

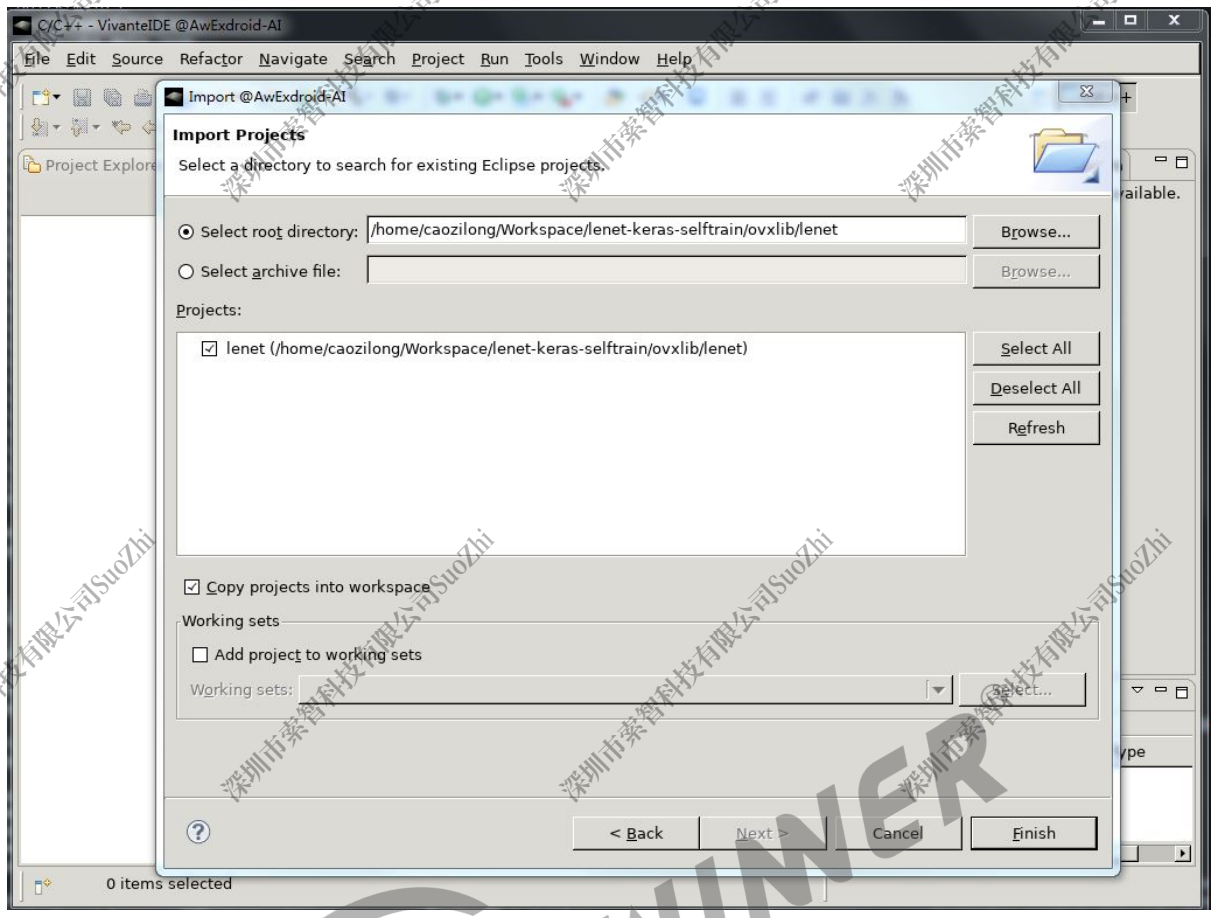

图 3-22: npu_prj_ok

这里强烈建议选中 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.

之后点击Finsih，结束导入过程. 导入后的工作空间如下所示：

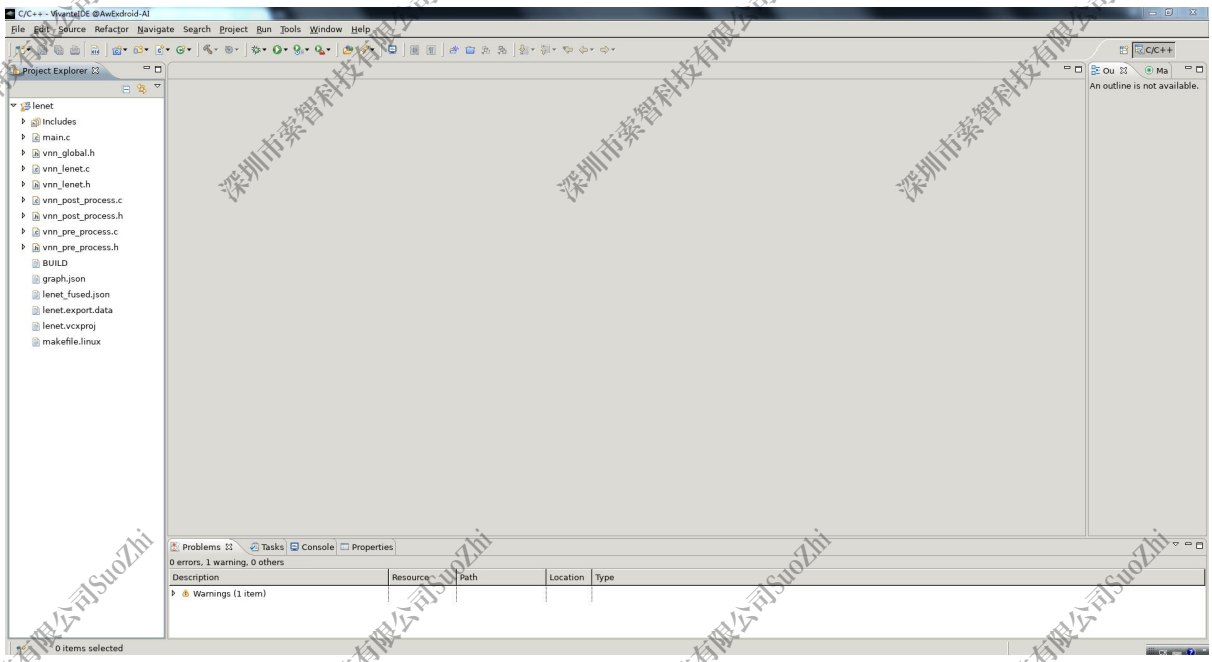

图 3-23: npu_sim_ok

3.7.3 编译工程

执行菜单命令 `Project->Build All`，先将仿真工程编译一下，看有没有犯低级错误 (比如导错工程了等等.):

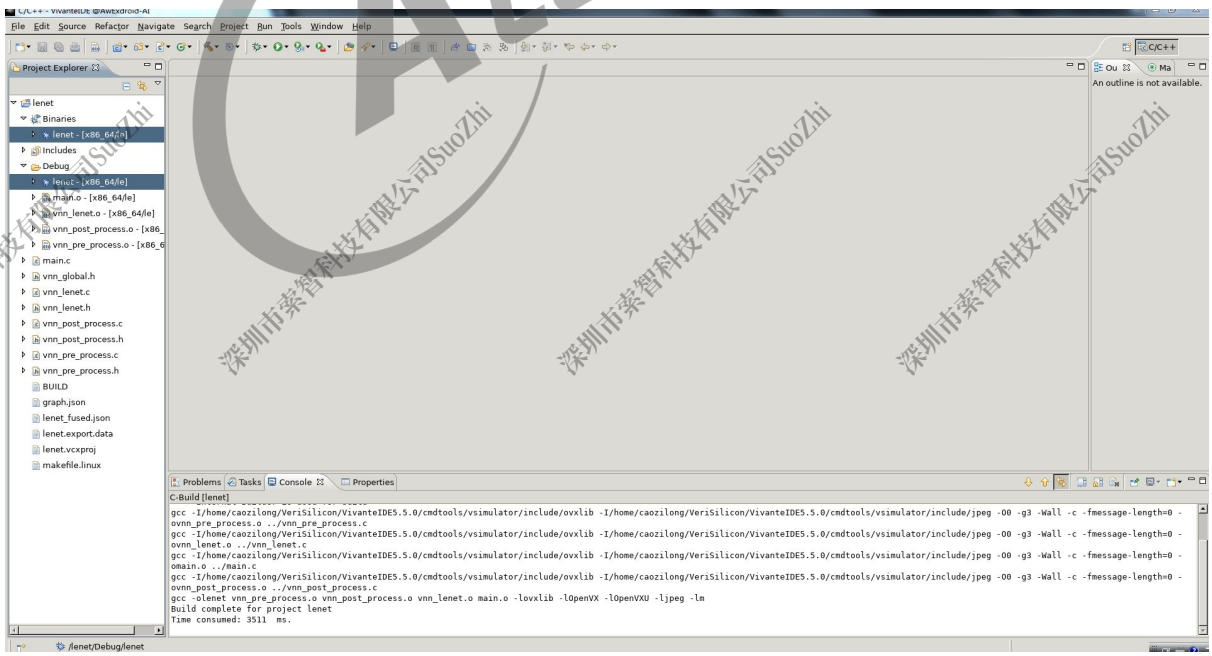

图 3-24: npu_compile_ok

编译 OK，生成了可执行 lenet 文件，我们进入下一步.

3.7.4 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。

这个选项卡最最重要的设置是 Program arguments，不同的网络根据输入输出个数的不同，输入也不尽相同，lenet 按照如下的方法设置:

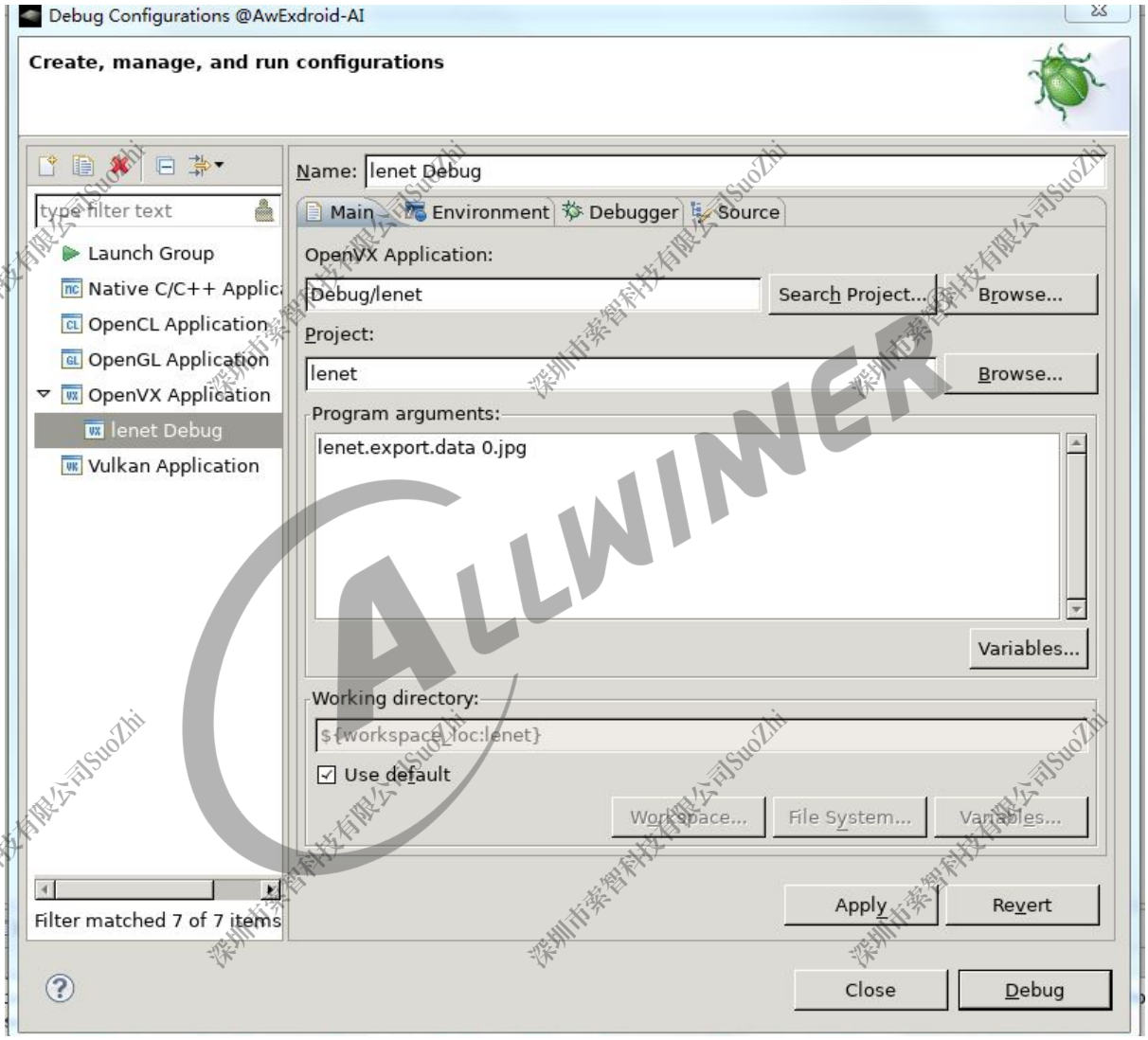

图 3-25: npu_argu

其中 lenet.export.data 是量化权重，它是在模型导出阶段生成在 ovxlib/lenet 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，而 0.jpg 则需要手动拷贝到 IDE 工程下:

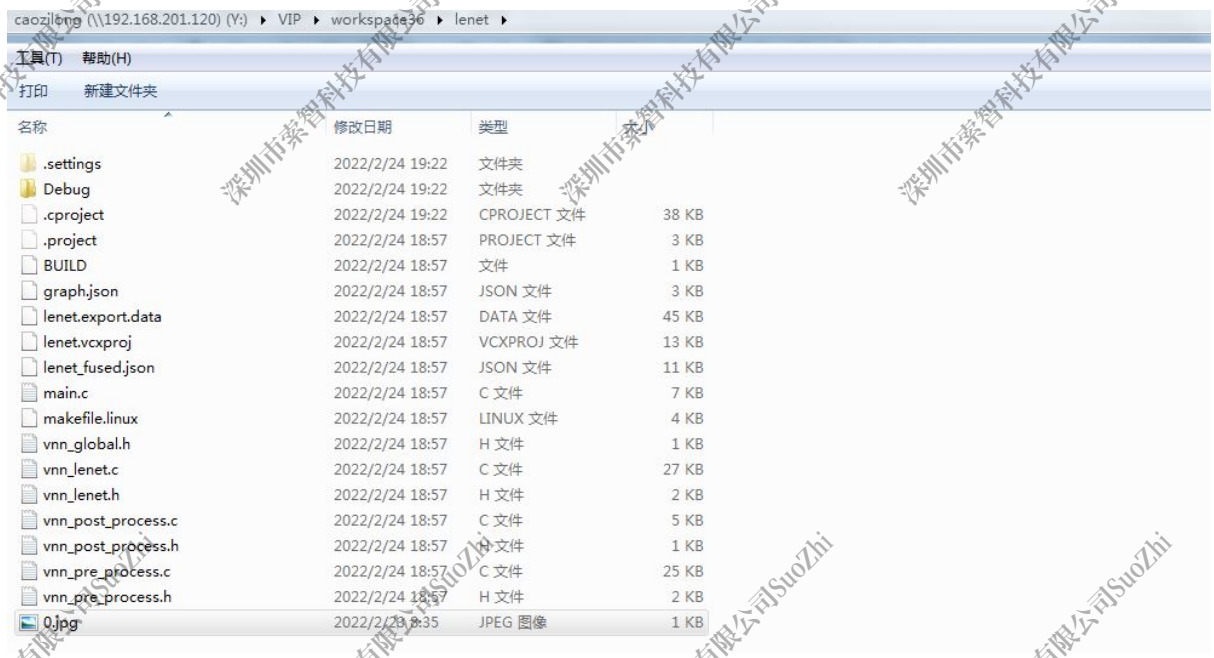

图 3-26: npu_arg_copy

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.

3.7.5 仿真

点击工具栏 Run 按钮，弹出对话框，直接点击 Run 触发仿真.

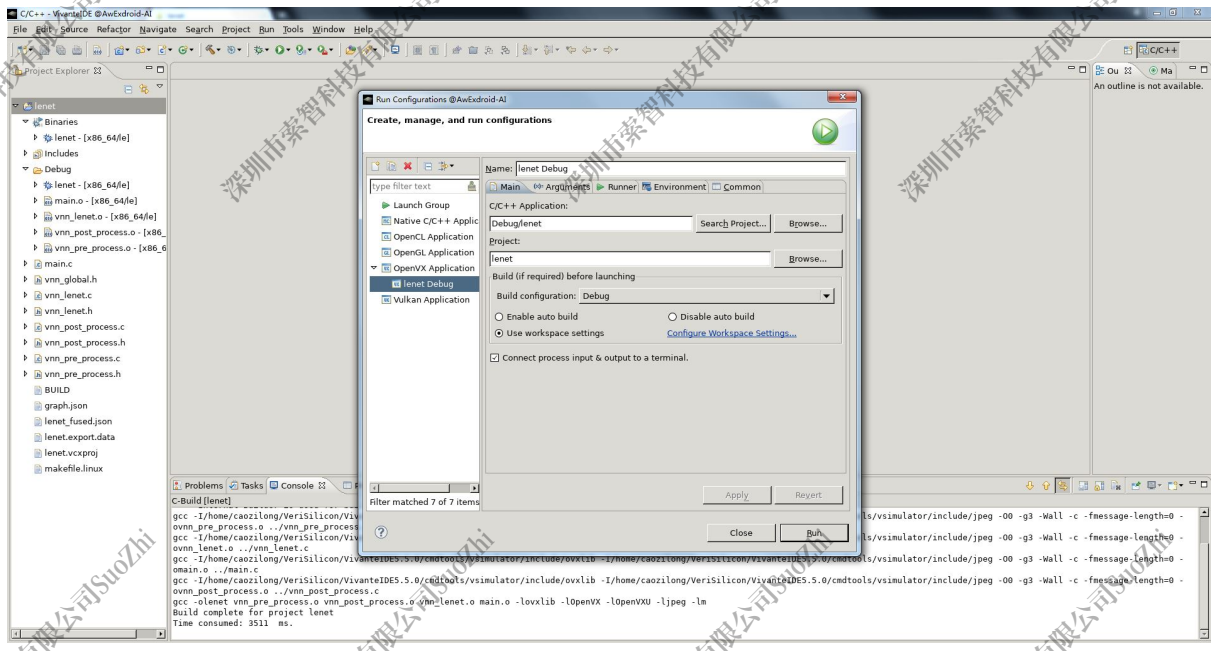

图 3-27: npu_run

lenet 是很小的网络，仿真很快结束，输出如下，根据下面控制台的输出可以看到，我们给的参数图像是 0.jpg，而仿真输出的 top5 结果表明，推理结果为 0 的概率为%99.0023，符合预期.

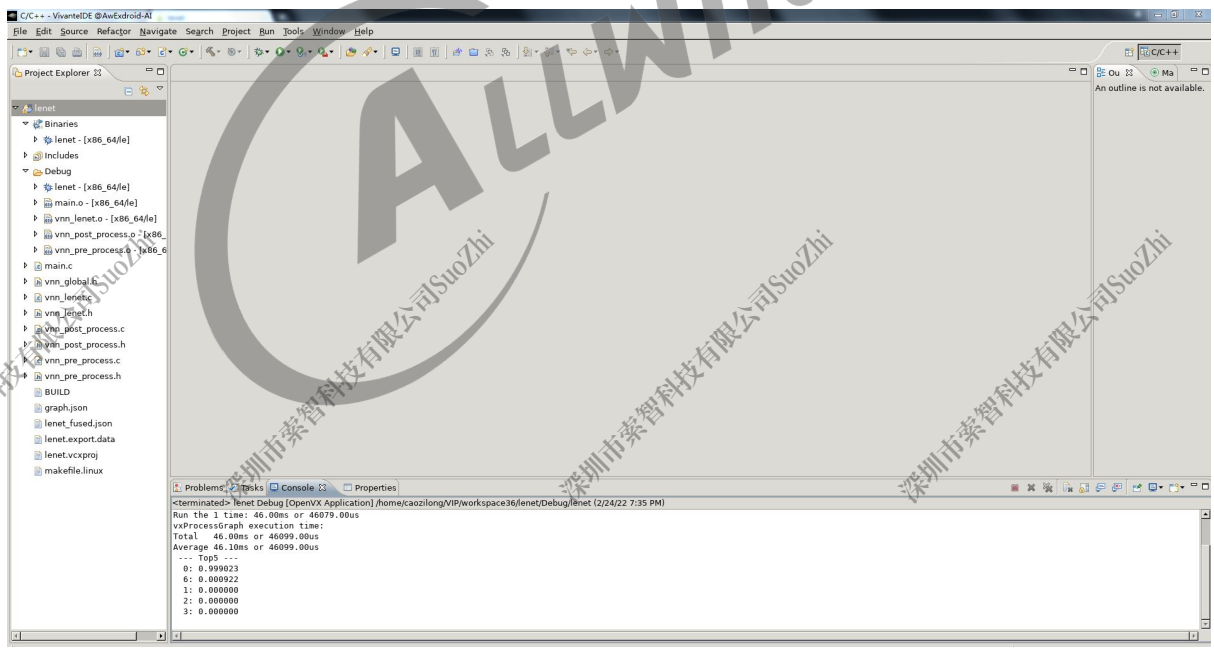

图 3-28: npu_sim_res

至此，模型仿真结束，进入模型 profile.

3.8 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。

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.

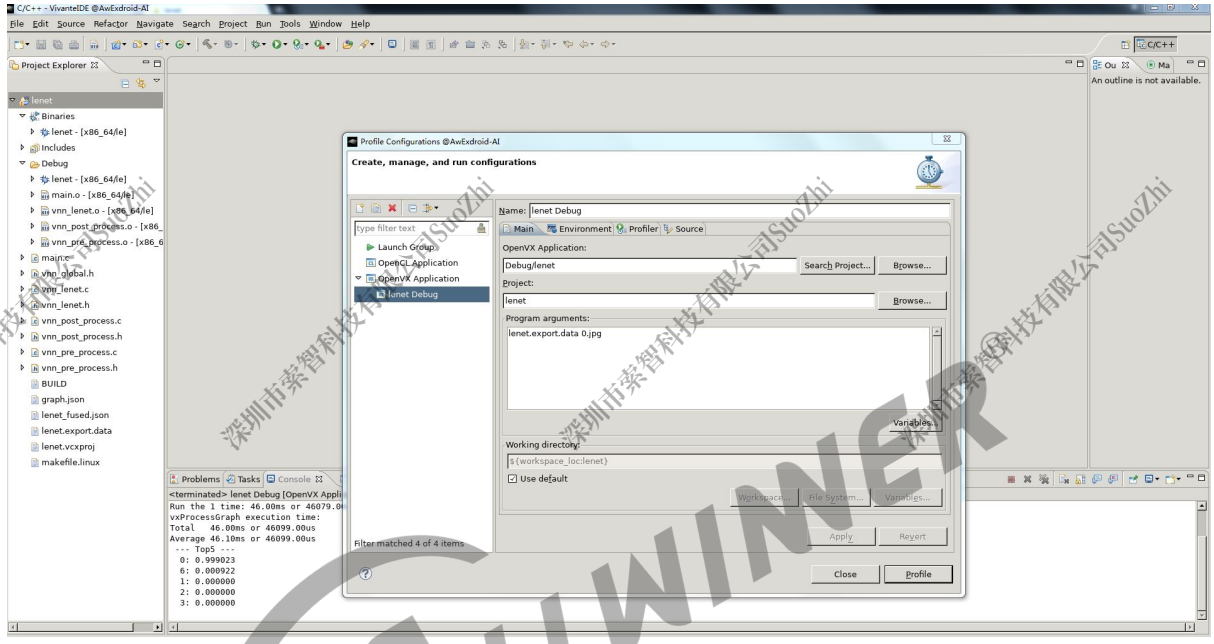

图 3-29: npu_profile_con

之后点击 Resume 继续:

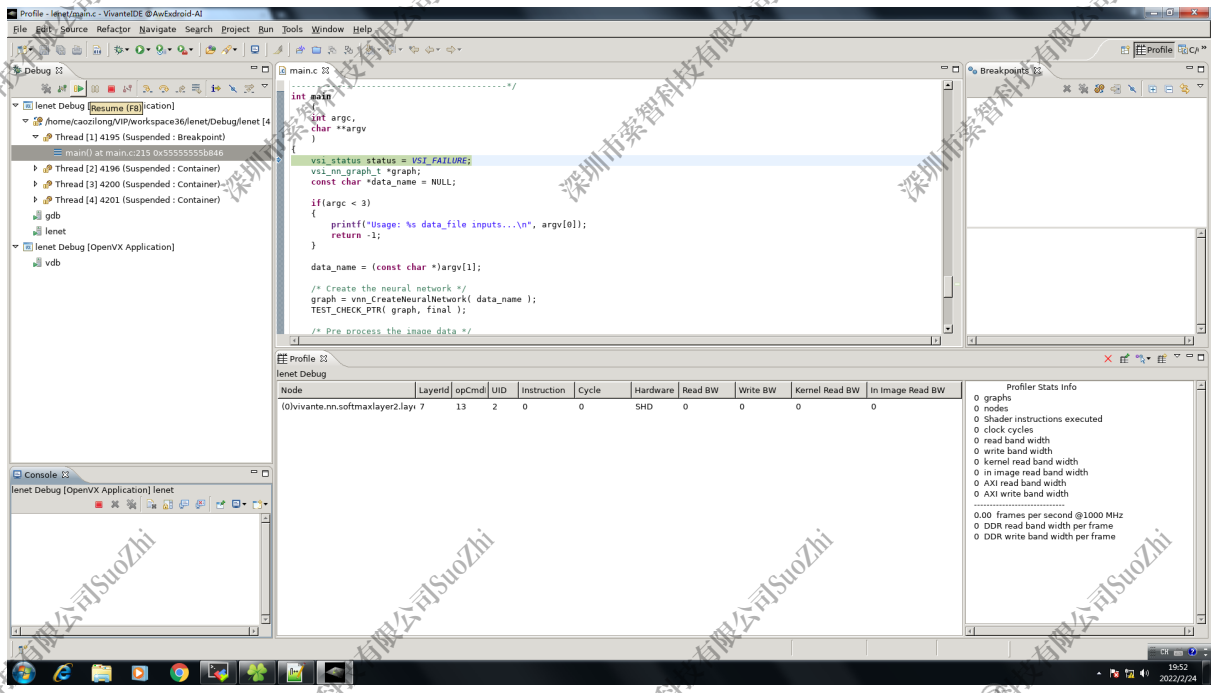

图 3-30: npu_profile_in

Profile 结束后, IDE 输出如下：

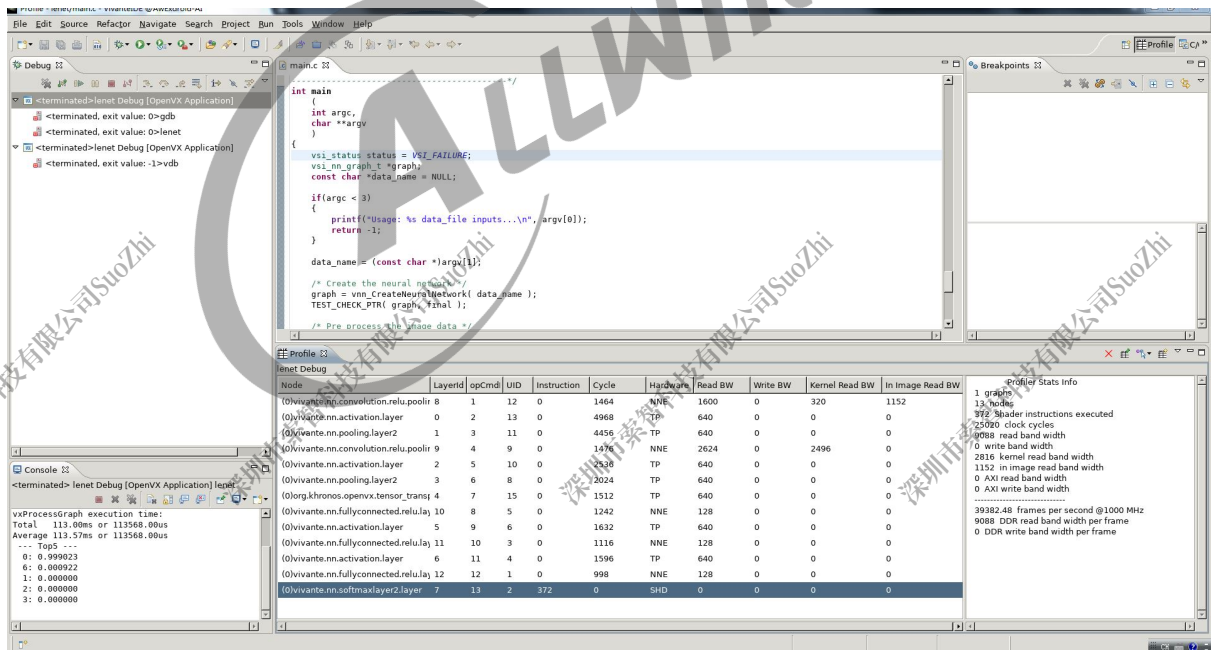

图 3-31: npu_lenet_profile

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 top5 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，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，时钟数，等等信息。更具体地分析，请参考芯原文档。

接下来，我们到了最后一步，我们前面所作地一切工作地目的，就是要将网络部署在端侧，真正地在板子上跑起来。我们开始端侧部署地介绍:

3.9 端侧部署

前面仿真阶段，仿真结束后，工程中生成了两个 bin 文件是我们部署验证要用到的，分别是 input_0.dat 和 output0_10_1.dat, 它们都是二进制格式的文件。input_0.dat 是网络第一层的输入，output0_10_1.dat 是网络最后一层的输出，由于根据仿真结果说明，这两笔数据都是正确的，可以作为 golden 数据和端侧的运行结果进行对比，如果在同样的 input 下，端侧跑出的 output tensor 和 output_0._10_1.dat 是 binary identical 的，那就说明，端侧部署是正确的。

理清了逻辑，我们开始动手操作，首先在仿真工程下认识一下这两个.dat, 下图左框中的蓝底文件:

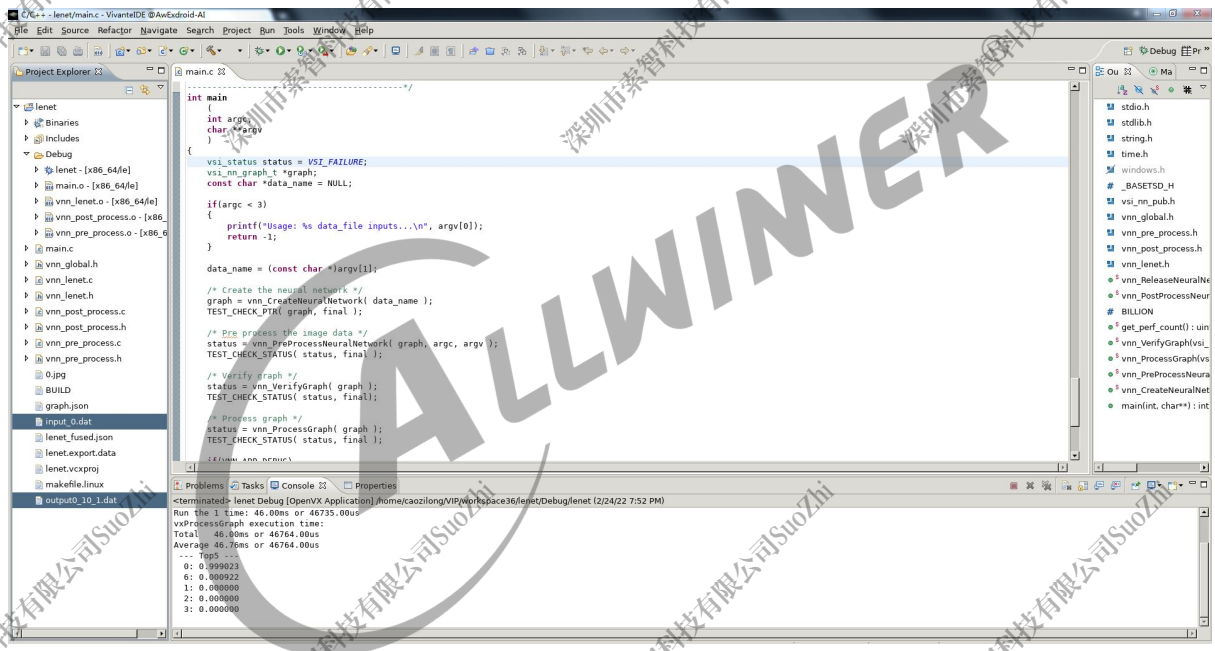

图 3-32: goldentensor

接下来，我们用到模型导出阶段生成的另一个工程，ovxlib/lenet_nbg_viplite 工程.

3.9.1 交叉编译 ovxlib/lenet_nbg_viplite 工程

我们发布的 Tina SDK 将会包含 NPU 的开发 SDK，NPU 的开发 SDK 结构如下图所示:

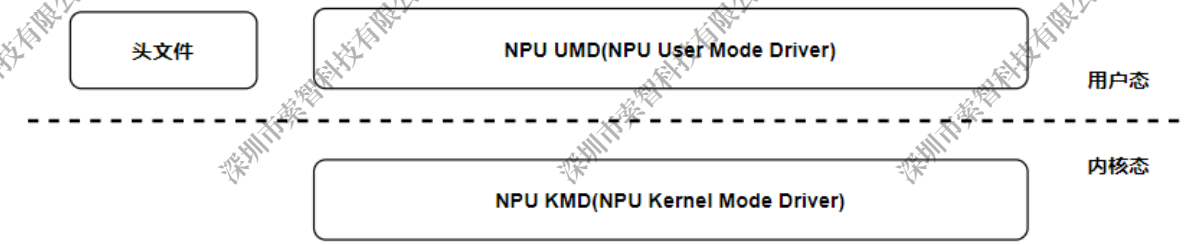

图 3-33: npu sdk

如果您拿到了 tina sdk，NPU 的 KMD 模块已经存在于 tina sdk 对应的 linux 内核代码中，而用户态 SDK, 也就是图中的 npu runtime library(UMD Driver)，则在 SDK 的 package/allwinner/目录下以库的形式存在。

接上文，我们将 ovxlib/lenet_nbg_viplite 拷贝出来，和 Tina 中 NPU runtime library 并列放在一个目录下，按照下面的内容编写 makefile 文件（SDK 中将会包含一个 demo makefile).

图 3-34: 工程目录结构

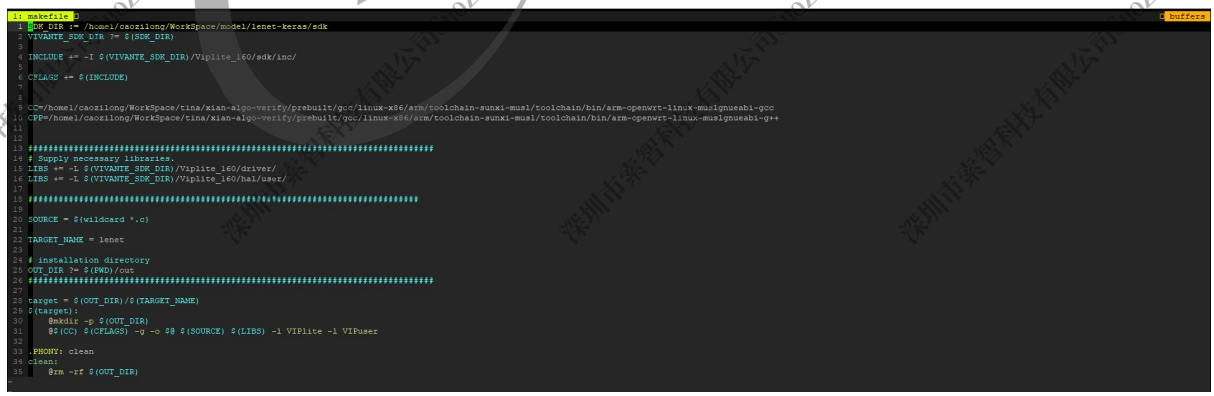

图 3-35: 工程 makefile

工程目录中，sdk 目录内容是 NPU 的用户态运行库，也就是 UMD 驱动，而 lenet_nbg_viplite 目录中的内容，则是模型部署阶段产生的 lenet_nbg_viplite 工程加上 makefile 的结果。

接下来就可以交叉编译测试工程了，在 lenet_nbg_viplite 目录，执行 make clean && make，执行结束后，在 out 目录将会生成可以在端侧跑的 lenet 测试程序。

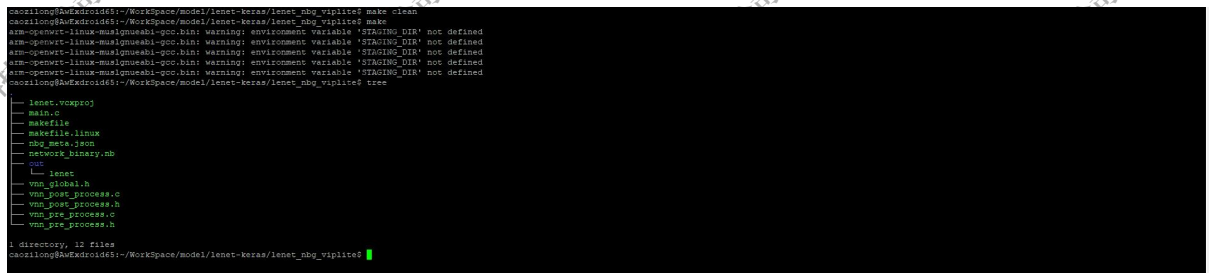

图 3-36: npu_elf_res

交叉编译到此结束，接下来准备测试工程目录。

3.9.2 准备测试工程目录

测试工程目录的内容包括，模型 NBG 文件，lenet 可执行程序，两个 UMD 动态库以及仿真阶段生成的 input_0.dat，一共 5 个文件，如下图所示：

图 3-37: npu_test_env

至此，测试目录已准备 OK，下面开始准备端侧验证平台。

3.9.3 准备端侧验证环境

首先，NPU 运行的大是你的端侧存下下面的设备节点 /dev/vipcore，否则，说明 SDK 配置是错误的，请寻求我们的协助。

图 3-38: npu_device

将前面建立的测试目录保存到 tf 卡中，我们用 TF 卡作为媒介验证, 将 TF 卡插到端侧平台，之后执行命令

```
mount -t vfat /dev/mmcblkxxx /mnt/sdcard
```

将其挂载到 /mnt/sdcard 目录，之后，进入 lenet-test 验证目录，执行命令

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/mnt/sdcard/lenet-test
```

保证运行库可以被正确的连接到。

之后就可以正式测试了，执行测试命令

```
./lenet network_binary.nb input_0.dat
```

输出如下：

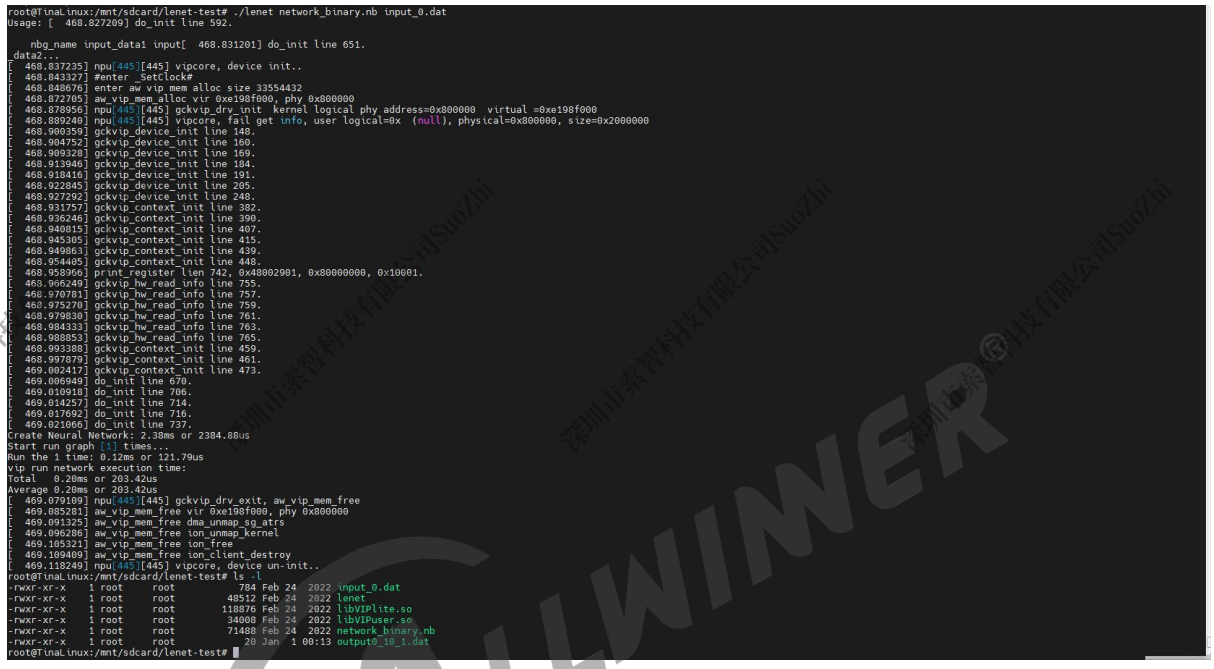

图 3-39: result

注意最后一行，可以看到测试目录下多了一个文件，output0_10_1.dat，他就是网络输出的结果。

3.10 验证

我们得到了 tensor 的端侧运行结果，将他和仿真生成的.dat 做对比，预期情况应该是 binary identical 的。

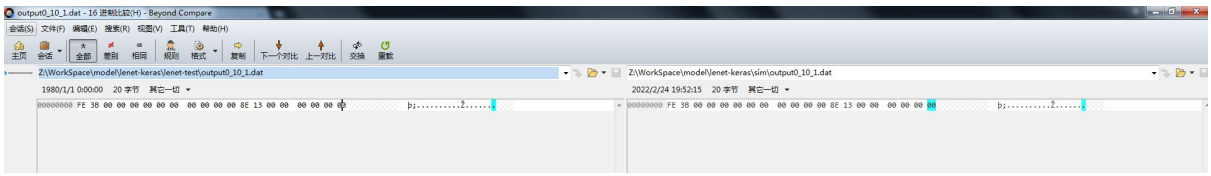

图 3-40: npu_cmp_res

结果 Binary Identical 的，和我们的预期一致.

3.10.1 验证 tensor

根据前面网络结构的描述，网络的最后一层是 softmax,softmax 层输出概率值为浮点数，在芯原的 NPU 设计中，存在三类计算单元，分别是 TP,NNE 和 PPU，这里面只有 PPU 支持浮点计算，所以 softmax 层要在 PPU 上运行的。

我们看一下 NBG 文件中的输出层信息, 如下图所示，可以看到输出为浮点 FP16，没有量化，根据上面验证 beyondcompare 对比，我们也可以看出，tensor 输出一共 20 个字节。我们来计算一下，lenet 分类网络一共识别十类目标，每一类的概率为 FP16，所以计算起来正好是 20 个字节，由此我们知道了 output tensor 的结构。

```
(vip) caozilong@AwExdroid-AI:~/Workspace/lenet-keras-selftrain/ovxlib/lenet_nbg_viplite$ nbinfo -out network_binary.nb

Verisilicon NBInfo version 1.2.1, version=0x00010201

nbg file name network_binary.nb

********************************************************************************
Output Table
********************************************************************************
Output 0
Dim Count:                                                                    2
Size of Dim[0]:                                                              10
Size of Dim[1]:                                                               1
Data Format:                                              NBG_BUFFER_FORMAT_FP16
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):                                                         64
Output name:                                              uid_2_sub_uid_1_out_0
********************************************************************************
(vip) caozilong@AwExdroid-AI:~/Workspace/lenet-keras-selftrain/ovxlib/lenet_nbg_viplite$
(vip) caozilong@AwExdroid-AI:~/Workspace/lenet-keras-selftrain/ovxlib/lenet_nbg_viplite$
```

图 3-41: nbinfo

既然知道了结构，我们就可以将运行产生的 output tensor 转换为概率打印出来，由于 32 位机上不支持 FP16 的格式，所以需要将其转换为符合 ieee754 的 float32 格式，核心转换代码如下:

```
typedef union
{
    unsigned int u;
    float f;
} _fp32_t;

static float fp16_to_fp32(const short in)
{
    const _fp32_t magic = { (254 - 15) << 23 };
    const _fp32_t infnan = { (127 + 16) << 23 };
    _fp32_t o;
    // Non-sign bits
    o.u = (in & 0x7fff ) << 13;
    o.f *= magic.f;
    if(o.f >= infnan.f)
    {
        o.u |= 255 << 23;
    }
    //Sign bit
    o.u |= (in & 0x8000 ) << 16;
    return o.f;
}
```

图 3-42: npu_fp16

输出如下:

```
caozilong@AwExdroid65:~/WorkSpace/fp16tofp32$ ./a.out output0_10_1.dat
main line 57, in
main line 75, file output0_10_1.dat len 20 byets.

dump file memory:
0xade250: 0x3bfe 0x00 0x00 0x00 0x00 0x00 0x138e 0x00 0x00 0x00 0x00 0x00 0xda1 0x02 0x00 0x00
0xade270: 0x00 0x00 0x00 0x00
class 0, prob 0.999023.
class 1, prob 0.000000.
class 2, prob 0.000000.
class 3, prob 0.000000.
class 4, prob 0.000000.
class 5, prob 0.000000.
class 6, prob 0.000922.
class 7, prob 0.000000.
class 8, prob 0.000000.
class 9, prob 0.000000.
main line 106, out
caozilong@AwExdroid65:~/WorkSpace/fp16tofp32
caozilong@AwExdroid65:~/WorkSpace/fp16tofp32
```

图 3-43: npu_fp32

对比仿真阶段的输出，得到的 TOP5 输出概率完全一样。说明我们的部署以及后处理是正确的。

至此，部署加验证过程全部结束～!

3.11 结束

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。