Tengine

用户手册

文档版本 1.2

发布日期 2019-06-27

OPEN AI LAB

OPEN AI LAB

变更记录

日期	版本	说明	作者
2019-03-05	0.1	初版	Rui Zhang
2019-04-15	0.2	合并之前文档	Bin Hu
2019-06-20	1.0	翻译成中文版本	Bin Hu
2019-06-25	1.1	Linux 交叉编译配置更改	Bin Hu
2019-06-27	1.2	Tengine 模型转换工具使用方法	Bin Hu

OPEN AI LAB

Menu

1 TENGINE 简介 5

2 编译 TENGINE 6

2.1 获取源代码 6
2.2 在 X86(UBUNTU)本地编译 7
2.2.1 安装依赖的库文件 7
2.2.2 安装 protobuf 7
2.2.3 安装 opencv 7
2.2.4 编辑 configuration 文件 7
2.2.5 编译 8
2.2.6 测试 9
2.3 在 ARM(LINUX)中编译源代码 10
2.3.1 获取源代码 10
2.3.2 安装 protobuf 10
2.3.3 安装 opencv 10
2.3.4 编辑 configuration 文件 10
2.3.5 编译 11
2.3.6 测试 12
2. 4 在 X86(UBUNTU)对 ARM LINUX 交叉编译 13
2.4.1 获取源代码 13
2.4.2 安装 rootfs 13
2.4.3 安装交叉编译链 13
2.4.4 从源代码交叉编译 protobuf 13
2.4.5 编辑 configuration 文件 14
2.4.6 正常编译 Tengine 15
2.4.7 测试 15
2.5 在 X86 LINUX 平台为 ANDROID 交叉编译 16
2.5.1 下载 Tengine 源代码 16
2.5.2 下载 OpenCV protobuf OpenBLAS 和 Android ndk 16
2.5.3 解压 protobuf android-ndk-r16-linux-x86_64.zip 和 openblas 16
2.5.4 设置依赖项库文件路径 17
2.5.5 编译 tengine 17
2.5.6 编译样例 18
2.5.7 在电脑上安装 adb, adb_driver 20
2.5.8 在安卓上测试 Tengine 的 Demo 20
2.6 编译 ACL 21
2.7 安装 PROTOBUF 21
2.7.1 安装 protobuf-compiler 并且确认当前版本为 3.0.0 21
2.7.2 安装 protobuf 21

OPEN AI LAB

2.7.3 安装 autoconf automake libtool 21
2.7.4 编译 protobuf 22
2.8 MAKEFILE.CONFIG 23
3 TENGINE C API 介绍 26
3.1 宏定义与结构体 26
3.2 API 介绍 31
3.2.1 Tengine 创建与释放相关 API 34
3.2.2 graph 相关操作 36
3.2.3 node 相关操作 42
3.2.4 kernel 相关操作 50
3.2.5 The tensor operation 51
3.2.6 与 graph 运行相关 API 57
3.2.7 与设备相关操作 61
3.2.8 与 context 相关部分 68
3.2.9 其他的 API 71
4 测试例程 75
4.1 安装 OPENCV 75
4.2 编译样例程序 75
4.2.1 Linux 75
4.2.2 Android 76
4.3 分类网络结构 76
4.3.1 编译例程 77
4.3.2 运行 77
4.4 检测网络 78
4.4.1 下载 model 78
4.4.2 编译样例程序 79
4.4.3 运行 79
5 一些工具与技巧 80
5.1 通用指令 80
5.1.1 设置使用的 CPU 数目 80
5.1.2 设置 kernel mode 80
5.2 使用多线程运行 TENGINE 81
5.3 支持 CPU 和 GPU 的异构计算 81
5.4 转换为 TENGINE 模型 84
5.5 量化 85
5.5.1 量化工具的使用 85
5.5.2 支持 INT8 和 FP32 的混合计算 85
5.5.3 测试量化模型的准确率 85
5.6 添加自定义算子 85
5.6.1 以 Scale Operator 为例 86

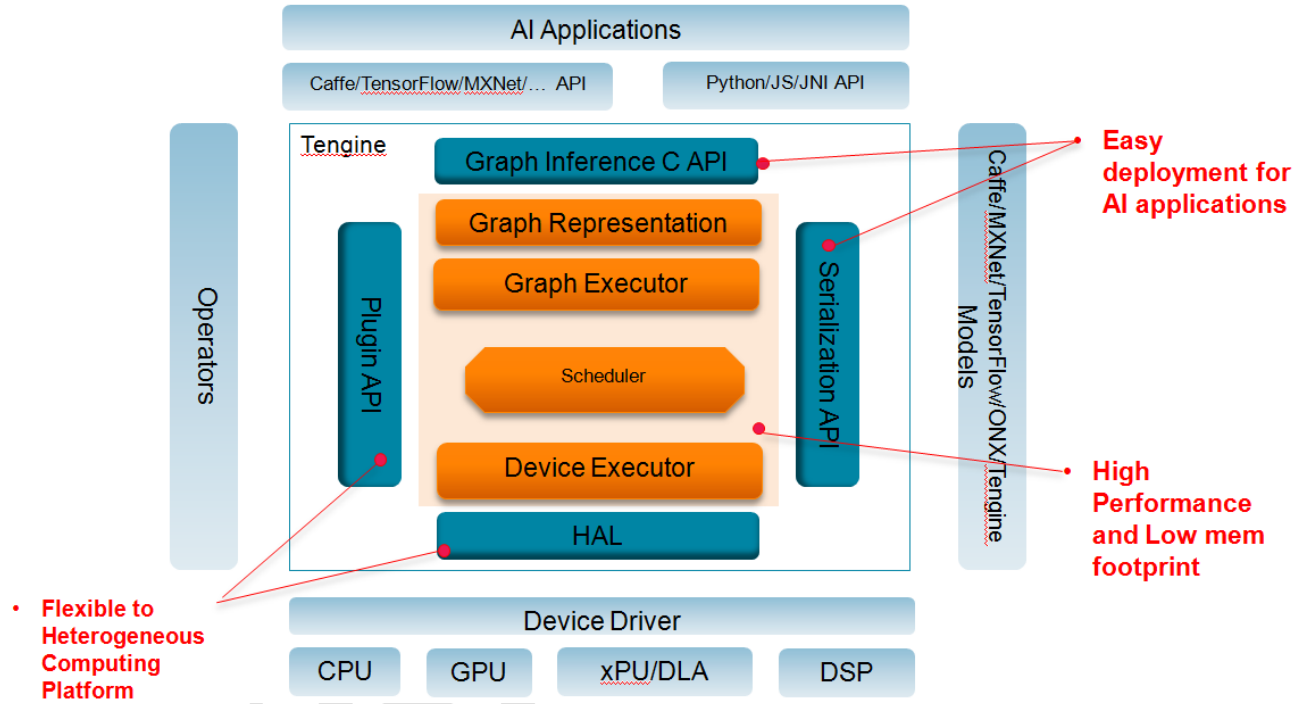
OPEN AI LAB

5.7 CAFFE WRAPPER 指令 87
5.8 TENSORFLOW WRAPPER 指令 88

OPEN AI LAB

1 Tengine 简介

Tengine 是 **OPEN AI LAB** 针对前端智能设备开发的软件开发包，核心部分是一个轻量级，模块化，高性能的 AI 推断引擎，并支持用 DLA、GPU、xPU 作为硬件加速计算资源异构加速。



Tengine 框架

Tengine 由六大模块组成: core/operator/serializer/executor/driver/wrapper。

core： 提供系统的基本组件和功能。

operator: 在此处定义了基本的 operators 框架, 比如 convolution， relu，pooling 等等。

serializer: 用来导入已保存的模型。该 serializer 框架可扩展以支持不同的框架模型，包括自定义模型格式。Tengine 可以直接导入 Caffe/ONNX/Tensorflow/MXNet 模型是或者 Tengine 自己的模型。

executor: 实现运行 graph 和 operators 的代码。

driver: 实际硬件的适配器，并通过 HAL API 向设备执行器提供服务。单个 driver 可以创建多个设备。

wrapper: 为不同的框架提供 API 封装。caffe API wrapper 和 tensorflow API wrapper 现在都可以工作了。

2 编译 Tengine

2.1 获取源代码

确认已安装 git：

git

```
miu@tengine:~$ git
bash: /usr/bin/git: No such file or directory
miu@tengine:~$
```

如果没有安装 git，你可以通过指令安装 git：

sudo apt-get install git

```
miu@tengine:~$ sudo apt install git
Reading package lists... Done
Building dependency tree
Reading state information... Done
Suggested packages:
  git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-cvs git-mediawiki git-svn
The following NEW packages will be installed:
  git
0 upgraded, 1 newly installed, 0 to remove and 197 not upgraded.
Need to get 0 B/3,907 kB of archives.
After this operation, 32.2 MB of additional disk space will be used.
Selecting previously unselected package git.
(Reading database ... 124936 files and directories currently installed.)
Preparing to unpack .../git_1%3a2.17.1-1ubuntu0.4_amd64.deb ...
Unpacking git (1:2.17.1-1ubuntu0.4) ...
Setting up git (1:2.17.1-1ubuntu0.4) ...
miu@tengine:~$
```

安装完成 git 之后，通过 git 获取 tengine 源代码，GitHub 的链接如下：

https://github.com/OAID/Tengine.git

你可以在自己选定的路径下，使用如下指令，从 GitHub 中下载 tengine。

git clone https://github.com/OAID/tengine/

```
miu@tengine:~$ git clone https://github.com/OAID/tengine
Cloning into 'tengine'...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 2457 (delta 0), reused 1 (delta 0), pack-reused 2452
Receiving objects: 100% (2457/2457), 25.59 MiB | 770.00 KiB/s, done.
Resolving deltas: 100% (1439/1439), done.
miu@tengine:~$
```

2.2 在 X86(ubuntu)本地编译

在开始编译之前，你需要确认你已经安装了 git，g++，make 和一些其他工具。如果没有，你可以通过如下指令安装：

```
sudo apt install make g++
```

如果你已经安装了这些工具，你可以跳过该步骤。

2.2.1 安装依赖的库文件

```
sudo apt install libgoogle-glog-dev libopenblas-dev
```

2.2.2 安装 protobuf

```
sudo apt install  libprotobuf-dev protobuf-compiler
```

提醒：这是安装 protobuf 的最简单方法，但要注意，Tengine 现在只支持 protobuf 3.0，如果你的环境是 Ubuntu18.04，那么你可以直接用该命令即可。

但是如果你的操作系统是 ubuntu16.04，你可以通过运行下述命令来检查 protobuf 版本。

```
sudo apt install protobuf-compiler
protoc --version
```

如果版本不是 3.0.0，你需要手动安装 protobuf3.0。你可以按照第 2.7 章所述步骤来安装 。

2.2.3 安装 opencv

```
sudo apt install libopencv-dev
```

注意：tengine 本身是不依赖于 opencv 库文件，所以如果你没有安装 libopencv，你也能生成 libtengine.so 文件，但是编译源文件之后，你可能会在编译 test 文件时产生错误提示。因此，我们建议你先安装 opencv 库。

2.2.4 编辑 configuration 文件

进入 tengine 源文件目录 (在 第 2.1 章 中安装)，复制 makefile.config.example 文件，然后将其重命名为 makefile.config。

对于 makefile.config 文件的详细编辑信息，可以参考 第 2.8 章。

```
cd ~/tengine
cp makefile.config.example makefile.config
```

OPEN AI LAB

```
miu@ubuntu:~$ cd tengine/
miu@ubuntu:~/tengine$ cp makefile.config.example makefile.config
miu@ubuntu:~/tengine$
```

使用编辑器打开 makefile.config 文件(以 vim 为例子)

```
vim makefile.config
```

你需要注释掉 CONFIG_ARCH_ARM64=y 并且启用 CONFIG_ARCH_BLAS=y，如下图：

```
# cross compile for ARM64
# CROSS_COMPILE=aarch64-linux-gnu-
# cross compile for ARM32
# CROSS_COMPILE=arm-linux-gnueabihf-

# Just to differentiate with sysroot for embedded toolchains building
# As toolchains just need a few pre-built libraries
#
# EMBEDDED_CROSS_ROOT=/opt/install/

# Set the target arch
# CONFIG_ARCH_ARM64=y
# CONFIG_ARCH_ARM32=y

# Enable Compiling float16 kernel
# CONFIG_FLOAT16=y

# Enable Compiling Optimization
CONFIG_OPT_CFLAGS = -O2

# Use BLAS as the operator implementation
 CONFIG_ARCH_BLAS=y

# Enable GPU support by Arm Computing Library
# CONFIG_ACL_GPU=y

# Set the path of ACL
# ACL_ROOT=/home/firefly/ComputeLibrary

# Enable other serializers
CONFIG_CAFFE_SERIALIZER=y
# CONFIG_MXNET_SERIALIZER=y
# CONFIG_ONNX_SERIALIZER=y
# CONFIG_TF_SERIALIZER=y
# CONFIG_TFLITE_SERIALIZER=y
```

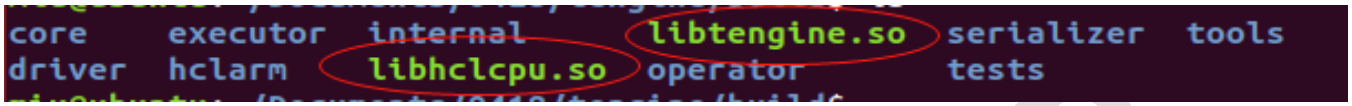
然后保存并退出 makefile.config 文件。

2.2.5 编译

进入 tengine 目录，然后使用 make 编译 tengine 项目。

```
cd ~/tengine
make -j4
```

如果在编译完成之后 build 目录下存在 libtengine.so 和 libhclcpu.so 文件，说明编译过程成功完成。



2.2.6 测试

编译完成之后，相关目录下有两个相关的测试程序：SqueezeNet 与 MobileNet。你可以使用如下指令执行相关测试：

1）运行 SqueezeNet 测试程序

```
cd ~/tengine
./build/tests/bin/bench_sqz
```

输出信息：

```
REPEAT COUNT= 100
Repeat [100] time 38050.40 us per RUN. used 3805040 us
0.2763 - "n02123045 tabby, tabby cat"
0.2673 - "n02123159 tiger cat"
0.1766 - "n02119789 kit fox, Vulpes macrotis"
0.0827 - "n02124075 Egyptian cat"
0.0777 - "n02085620 Chihuahua"
-----------------------------------
ALL TEST DONE
```

2）运行 MobileNet 测试程序

```
cd ~/tengine
./build/tests/bin/bench_mobilenet
```

输出信息：

```
REPEAT COUNT= 100
Repeat [100] time 71437.16 us per RUN. used 7143716 us
8.5976 - "n02123159 tiger cat"
7.9550 - "n02119022 red fox, Vulpes vulpes"
7.8679 - "n02119789 kit fox, Vulpes macrotis"
7.4274 - "n02113023 Pembroke, Pembroke Welsh corgi"
6.3647 - "n02123045 tabby, tabby cat"
ALL TEST DONE
```

OPEN AI LAB

2.3 在 ARM(Linux)中编译源代码

目前，tengine 开源版本 支持 ARM-v8 , tengine 商业版本支持 ARM-v7, ARM-v8 ， 更多信息可以参考《Tengine specification》。

该章节将详细描述在 arm 本地编译源代码的步骤。

2.3.1 获取源代码

你可以按照第 2.1 章 的步骤下载源代码。

2.3.2 安装 protobuf

如果你需要在 ARM SOC 板子上编译代码，只需要安装 protobuf 依赖库文件。

```
sudo apt install  libprotobuf-dev protobuf-compiler
```

注意：确认你的 Protobuf 版本是 3.0.0，如果不是 3.0.0，你需要手动安装 protobuf 3.0。请按照第 2.7 章所述步骤操作 。

2.3.3 安装 opencv

```
sudo apt install libopencv-dev
```

注意：tengine 本身是不依赖于 opencv 库文件，所以如果你没有安装 libopencv，你也能生成 libtengine.so 文件，但是编译源文件之后，你可能会在编译 test 文件时产生错误提示。因此，我们建议你先安装 opencv 库。

2.3.4 编辑 configuration 文件

对于 makefile.config 文件的详细编辑信息，可以参考 第 2.8 章。

```
cd ~/tengine
cp makefile.config.example makefile.config
```

```
miu@ubuntu:~$ cd tengine/
miu@ubuntu:~/tengine$ cp makefile.config.example makefile.config
miu@ubuntu:~/tengine$
```

使用编辑器打开 makefile.config 文件(以 vim 为例子)

```
vim makefile.config
```

- 如果你想要编译在 arm64 中：

OPEN AI LAB

请设置 CONFIG_ARCH_ARM64=y 并且注释 CONFIG_ARCH_ARM32=y

```
# Set the target arch
 CONFIG_ARCH_ARM64=y
# CONFIG_ARCH_ARM32=y

# Enable Compiling float16 kernel
# CONFIG_FLOAT16=y

# Enable Compiling Optimization
CONFIG_OPT_CFLAGS = -O2

# Use BLAS as the operator implementation
# CONFIG_ARCH_BLAS=y
```

- 如果你想要编译在 arm32 中：

请设置 CONFIG_ARCH_ARM32=y 并且编译 CONFIG_ARCH_ARM64=y

```
# Set the target arch
# CONFIG_ARCH_ARM64=y
 CONFIG_ARCH_ARM32=y

# Enable Compiling float16 kernel
# CONFIG_FLOAT16=y

# Enable Compiling Optimization
CONFIG_OPT_CFLAGS = -O2

# Use BLAS as the operator implementation
# CONFIG_ARCH_BLAS=y

# Enable GPU support by Arm Computing Library
# CONFIG_ACL_GPU=y
```

2.3.5 编译

进入 tengine 路径然后使用 make 进行编译。

```
cd ~/tengine
make -j4
```

等待一段时间，如果在编译完成之后 build 目录下存在 libtengine.so 文件，说明编译过程成功完成。

```
miu@ubuntu:~/tengine$
miu@ubuntu:~/tengine$ ls build/
core    executor  internal      libtengine.so  serializer  tools
driver  hclarm    libhclcpu.so  operator       tests
miu@ubuntu:~/tengine$
```

2.3.6 测试

Tengine 在 test 路径下提供了一些样例程序，通过运行样例程序与得到的结果，你可以判断你的 tengine 是否成功编译。

有两个测试示例来显示测试用例：

- 运行 SqueezeNet

```
cd ~/tengine./build/tests/bin/bench_sqz
```

输出信息：

```
REPEAT COUNT= 100
Repeat [100] time 38050.40 us per RUN. used 3805040 us
0.2763 - "n02123045 tabby, tabby cat"
0.2673 - "n02123159 tiger cat"
0.1766 - "n02119789 kit fox, Vulpes macrotis"
0.0827 - "n02124075 Egyptian cat"
0.0777 - "n02085620 Chihuahua"
-----------------------------------
ALL TEST DONE
```

- 运行 MobileNet

```
cd ~/tengine
./build/tests/bin/bench_mobilenet
```

输出信息：

```
REPEAT COUNT= 100
Repeat [100] time 71437.16 us per RUN. used 7143716 us
8.5976 - "n02123159 tiger cat"
7.9550 - "n02119022 red fox, Vulpes vulpes"
7.8679 - "n02119789 kit fox, Vulpes macrotis"
7.4274 - "n02113023 Pembroke, Pembroke Welsh corgi"
6.3647 - "n02123045 tabby, tabby cat"
ALL TEST DONE
```

2. 4 在 X86(Ubuntu)对 ARM Linux 交叉编译

Tengine 目前支持对 ARM64 与 ARM32 在 X86 Linux(Ubuntu)进行交叉编译，你可以按照如下步骤进行编译。

2.4.1 获取源代码

你可按照第 2.1 章 的步骤下载源代码。

2.4.2 安装 rootfs

进入 tengine 目录下的 systemroot。

```
cd ~/tengine/sysroot/
sudo apt install make
sudo apt install multistrap
```

Arm64:

```
make ubuntu
```

Arm32:

```
make ubuntu32
```

2.4.3 安装交叉编译链

Arm64:

```
sudo apt install g++-aarch64-linux-gnu
```

Arm32:

```
sudo apt install g++-arm-linux-gnueabihf
```

2.4.4 从源代码交叉编译 protobuf

2.4.4.1 编译

请按照 第 2.7 章 的步骤手动编译源代码。

2.4.4.2 将 .so 与 header 复制到 rootfs 下

- Arm64:

OPEN AI LAB

```
cd ~/tengine/
cp PROTO_PATH/lib/libprotobuf.so sysroot/ubuntu_rootfs/usr/lib/
cp PROTO_PATH/include/google sysroot/ubuntu_rootfs/usr/include -r
```

- Arm32:

```
cd ~/tengine/
cp PROTO_PATH/lib/libprotobuf.so sysroot/ubuntu_rootfs32/usr/lib/
cp PROTO_PATH/include/google sysroot/ubuntu_rootfs32/usr/include -r
```

比如: **PROTO_PATH** = opl/protobuf_3.0.0，你可以通过检查命令来确认。

./configure --with-protoc=protoc --host=aarch64-linux-gnu --prefix=**ABSOLUTE__PATH/tengine/opl/protobuf_3.0.0**

在 ARM64 中 demo 如下:

```
miu@ubuntu:~/tengine$ cp opl/protobuf_3.0.0/lib/libprotobuf.so sysroot/ubuntu_rootfs/usr/lib/
miu@ubuntu:~/tengine$ cp opl/protobuf_3.0.0/include/google/ sysroot/ubuntu_rootfs/usr/include/ -r
```

2.4.5 编辑 configuration 文件

```
cd ~/tengine/
cp makefile.config.example makefile.config
vim makefile.config
```

- Arm64:

注释 **CONFIG_ARCH_ARM64=y 并且启用 CROSS_COMPILE=aarch64-linux-gnu-**

同时把 **CONFIG_ARCH_ARM64** 打开

如下图所示:

```
#-------------------------------------------------------------------------

# cross compile for ARM64
 CROSS_COMPILE=aarch64-linux-gnu-
# cross compile for ARM32
# CROSS_COMPILE=arm-linux-gnueabihf-

# Just to differentiate with sysroot for embedded toolchains building
# As toolchains just need a few pre-built libraries
#
# EMBEDDED_CROSS_ROOT=/opt/install/

# Set the target arch
 CONFIG_ARCH_ARM64=y
# CONFIG_ARCH_ARM32=y

# Enable Compiling Optimization
CONFIG_OPT_CFLAGS = -O2
                                                              25,2
```

- Arm32:

  注释 **CONFIG_ARCH_ARM32=y** 并且使能 **CROSS_COMPILE=arm-linux-gnueabihf-**

  同时把 **CONFIG_ARCH_ARM32** 打开

  如下图：

```
# cross compile for ARM64
# CROSS_COMPILE=aarch64-linux-gnu-
# cross compile for ARM32
 CROSS_COMPILE=arm-linux-gnueabihf-

# Just to differentiate with sysroot for embedded toolchains building
# As toolchains just need a few pre-built libraries
#
# EMBEDDED_CROSS_ROOT=/opt/install/

# Set the target arch
# CONFIG_ARCH_ARM64=y
 CONFIG_ARCH_ARM32=y
```

2.4.6 正常编译 Tengine

```
make
```

等待一段时间，如果在编译完成之后 build 目录下存在 libtengine.so 文件，说明编译过程成功完成。

```
miu@ubuntu:~/tengine$
miu@ubuntu:~/tengine$ ls build/
core    executor  internal      libtengine.so  serializer  tools
driver  hclarm    libhclcpu.so  operator       tests
miu@ubuntu:~/tengine$
```

2.4.7 测试

无错误地编译完成后，通过指令打包该文件。

```
make install
```

将文件传输到目标板，用你的目标 IP 和路径替换下文的红色文字

```
scp -r ./install rk@192.168.88.52:~/test
scp -r ./models/ rk@192.168.88.52:~/test
scp -r ./tests/images/ rk@192.168.88.52:~/test
scp opl/protobuf_3.0.0/lib/libprotobuf.so.10 rk@192.168.88.52:~/test
```

登录目标板，进入交叉编译文件的测试目录。

```
cd ~/test
export LD_LIBRARY_PATH=.
mv install/lib/lib*.so .
./install/bin/bench_sqz
```

提示：你也可以仅传送应用程序。把它们放在同一个目录下。

如果你得到一个提示为 GLIBCXX 版本问题的错误，如下图显示：

```
guest@firefly:~/test$ ./bench_mobilenet
./bench_mobilenet: /usr/lib/aarch64-linux-gnu/libstdc++.so.6: version `GLIBCXX_3.4.
22' not found (required by ./libtengine.so)
guest@firefly:~/test$
```

你可以通过一下指令解决该问题：

```
sudo apt-get install libstdc++6
```

2.5 在 X86 Linux 平台为 Android 交叉编译

2.5.1 下载 Tengine 源代码

你可以按照第 2.1 章 的步骤下载源代码。

2.5.2 下载 OpenCV protobuf OpenBLAS 和 Android ndk

从链接下载以下文件 http://ftp.openailab.net.cn/Tengine_android_build/

```
android-ndk-r16-linux-x86_64.zip
openblas020_android.tgz
opencv.tgz
protobuf_lib.tgz
ComputeLibrary.tgz
```

注意： 另一个备份链接 ：

Tengine Android build (链接：https://pan.baidu.com/s/1RPHK_ji0LlL3ztjUa893Yg#list/path=%2F)密钥： *ka6a*:

2.5.3 解压 protobuf android-ndk-r16-linux-x86_64.zip 和 openblas

```
unzip android-ndk-r16-linux-x86_64.zip
tar -zxvf protobuf_lib.tgz
```

```
tar -zxvf openblas020_android.tgz
tar -xzvf  ComputeLibrary.tgz
```

你还可以在这里通过命令解包 opencv.tgz（用于测试编译）。

```
tar -xzvf opencv.tgz
```

2.5.4 设置依赖项库文件路径

我们需要更新依赖文件路径 NDK PATH，PROTOBUF_DIR，BLAS PATH 和 CONFIG_ARCH_TYPE。

```
vi  ~/tengine/android_config.txt
```

- 如果你想要编译 android for armv7:

set the CONFIG_ARCH_TYPE:**ARMv7**

- 如果你想要编译 android for armv8:

set the CONFIG_ARCH_TYPE:**ARMv8**

并且改变如下路径 ANDROID_NDK，PROTOBUF_DIR 和 BLAS_DIR 到之前存储的位置，如下所示：

```
ANDROID_NDK:/home/usr/android-ndk-r16b
PROTOBUF_DIR:/home/usr/protobuf_lib
BLAS_DIR:/home/usr/Openblas020-android
ACL_ROOT:/home/usr/ComputeLibrary
CONFIG_ARCH_TYPE:ARMv8
```

以下是一个针对 ARMv8 的配置文件样例：

```
ANDROID_NDK:/home/bhu/workshop/android/android-ndk-r16b
PROTOBUF_DIR:/home/bhu/workshop/android/protobuf_lib
BLAS_DIR:/home/bhu/workshop/android/Openblas020-android
ACL_ROOT:/home/bhu/workshop/android/ComputeLibrary
CONFIG_ARCH_TYPE:ARMv8
```

2.5.5 编译 tengine

```
cd ~/tengine
```

- 如果你想要编译 android for armv7:

```
vi android_build_armv7.sh
```

设置* DCONFIG_ARCH_ARM32=ON *

- 如果你想要编译 android for armv8:

OPEN AI LAB

```
vi android_build_armv8.sh
```

设置* DCONFIG_ARCH_ARM64=ON *

- 如果你想用 openblas 运行 tengine：

从文件 ../android_build_armv7.sh 或../android_build_armv8.sh 移出 DCONFIG_ARCH_ARM64 或者 DCONFIG_ARCH_ARM32, 设置** -DCONFIG_ARCH_BLAS=ON *.*You must set the correct** BLAS_DIR** 在文件 **android_config.txt**

```
mkdir build
cd build
```

- 如果你想要编译 android for armv7:

```
../android_build_armv7.sh
```

- 如果你想要编译 android for armv8:

```
../android_build_armv8.sh
```

```
make -j4
make install
```

2.5.6 编译样例

2.5.6.1 解压 opencv

切换到 opencv-3.4.0-android-sdk 目录，然后解包。

```
unzip opencv-3.4.0-android-sdk.zip
```

2.5.6.2 设置 *TENGINE_DIR, OpenCV_DIR, PROTOBUF_DIR.*

```
cd ~/tengine/examples
```

- 如果你想要编译 android for armv7:

```
vi android_build_armv7.sh
```

请在 example/android_build_armv7.sh 中添加正确的 blas 路径，确保安装目录位于 TENGINE_DIR 中。

- 如果你想要编译 android for armv8:

```
vi android_build_armv8.sh
```

请在 example/android_build_armv7.sh 中添加正确的 blas 路径，确保安装目录位于 TENGINE_DIR 中。

OPEN AI LAB

如下为文件截图，红色文字应该替换为实际的目录。

```
#!/bin/bash

export ANDROID_NDK=/home/usr/android-ndk-r16b

cmake -
DCMAKE_TOOLCHAIN_FILE=$ANDROID_NDK/build/cmake/android.toolchain.cmak    e \
        -DANDROID_ABI="arm64-v8a" \
        -DANDROID_PLATFORM=android-21 \
        -DANDROID_STL=c++_shared \
        -DTENGINE_DIR=/home/usr/tengine \
       -DOpenCV_DIR=/home/usr/opencv/sdk/native/jni \
       -DPROTOBUF_DIR=/home/usr/protobuf_lib \
       ..
```

- 如果你想要使用 OpenBlas 运行 Tengien：

请在 example/android_build_armv7.sh 或 example/android_build_armv8.sh.等，中添加正确的 blas 路径： -DBLAS_DIR=/home/usr/openbla020_android

2.5.6.3编译 demo 样例

```
cd ~/tengine/examples
mkdir build
cd build
```

- 如果你想要编译 android for armv7:

```
../android_build_armv7.sh
```

- 如果你想要编译 android for armv8:

```
../android_build_armv8.sh
```

```
make -j4
```

OPEN AI LAB

2.5.7 在电脑上安装 adb, adb_driver

请按照你的安卓设备的相关手册来做。

2.5.8 在安卓上测试 Tengine 的 Demo

2.5.8.1 准备文件

```
cd ~/tengine
./android_pack.sh
cp -rf ./models ./android_pack
```

其他文件:

- 测试图片(cat.jpg,bike.jpg 或者其他)

```
cp -rf ./tests/images/ ./android_pack
```

- examples/build/imagenet_classification/Classify

```
cp ./examples/build/imagenet_classification/Classify ./android_pack
```

2.5.8.2将包传输到 Android 设备

将以下文件放置到 **~/tengine/android_pack**

```
adb push ./android_pack /data/local/tmp
```

2.5.8.3运行

```
adb root
adb shell
cd /data/local/tmp/
chmod u+x Classify
export LD_LIBRARY_PATH=.
./Classify -i images/cat.jpg
```

输出信息：

```
0.2763 - "n02123045 tabby, tabby cat"
0.2673 - "n02123159 tiger cat"
```

OPEN AI LAB

```
0.1766 - "n02119789 kit fox, Vulpes macrotis"
0.0827 - "n02124075 Egyptian cat"
0.0777 - "n02085620 Chihuahua"
```

2.6 编译 ACL

目前，Tengine 支持的 ACL 版本是 18.05。

```
git clone https://github.com/ARM-software/ComputeLibrary.git
git checkout v18.05
```

你可以通过链接获得编译步骤

https://arm-software.github.io/ComputeLibrary/v18.05/

2.7 安装 Protobuf

2.7.1 安装 protobuf-compiler 并且确认当前版本为 3.0.0

```
sudo apt install protobuf-compiler
protoc --version
```

```
peter@ubuntu:~/tengine/sysroot$ protoc --version
libprotoc 3.0.0
```

2.7.2 安装 protobuf

注意：下载 protobuf 版本必须与 host 版本相同。（根据以上示例下载 3.0.0 版本）

如果确认使用的版本为 3.0.0，你可以下载源代码从以下指令。

```
cd ~
wget https://github.com/protocolbuffers/protobuf/archive/v3.0.0.tar.gz
tar -xzvf v3.0.0.tar.gz
```

2.7.3 安装 autoconf automake libtool

```
sudo apt-get install autoconf automake libtool curl
```

OPEN AI LAB

2.7.4 编译 protobuf

进入在 2.7.1 章节下载的 protobuf 目录，然后执行以下命令：

在 tengine 目录下，新建一个新的目录，命名为 opl，安装 protobuf comile 文件

```
cd ~/tengine/
mkdir opl
```

然后进入 protobuf 目录，该目录从 2.7.1.2 章下载。

```
miu@ubuntu:~$ cd protobuf-3.0.0/
miu@ubuntu:~/protobuf-3.0.0$ ls
appveyor.bat      generate_descriptor_proto.sh  protobuf-lite.pc.in
appveyor.yml      gmock.BUILD                   protobuf.pc.in
autogen.sh        java                          Protobuf.podspec
benchmarks        javanano                      protoc-artifacts
BUILD             jenkins                       python
CHANGES.txt       js                            README.md
cmake             LICENSE                       ruby
configure.ac      m4                            six.BUILD
conformance       Makefile.am                   src
CONTRIBUTORS.txt  more_tests                    tests.sh
csharp            objectivec                    update_file_lists.sh
docs              php                           util
editors           post_process_dist.sh          WORKSPACE
examples          protobuf.bzl
miu@ubuntu:~/protobuf-3.0.0$
miu@ubuntu:~/protobuf-3.0.0$
```

运行如下的指令：

```
./autogen.sh
```

你可能会遇到这样的错误报告：

```
miu@ubuntu:~/protobuf-3.0.0$ ./autogen.sh
Google Mock not present.  Fetching gmock-1.7.0 from the web...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:--  0:00:21 --:--:--     0cu
rl: (7) Failed to connect to googlemock.googlecode.com port 443: Connection refu
sed
miu@ubuntu:~/protobuf-3.0.0$
```

由于某些原因，autogen 无法通过 curl 下载模拟源代码，因此需要更新下载链接，可以在 autogen.sh 中更新下载链接：

curl $curlopts -L -o gmock-1.7.0.zip https://github.com/peter-wangxu/gmock/archive/1.7.0.zip

下图是供参考的差异。

```
miu@ubuntu:~/protobuf-3.0.0$ git diff autogen.sh_original autogen.sh
diff --git a/autogen.sh_original b/autogen.sh
index 5b4c29f..621028f 100755
--- a/autogen.sh_original
+++ b/autogen.sh
@@ -31,7 +31,7 @@ fi
 # directory is set up as an SVN external.
 if test ! -e gmock; then
   echo "Google Mock not present.  Fetching gmock-1.7.0 from the web..."
-  curl $curlopts -O https://googlemock.googlecode.com/files/gmock-1.7.0.zip
+  curl $curlopts  -L -o gmock-1.7.0.zip https://github.com/peter-wangxu/gmock/a
rchive/1.7.0.zip
   unzip -q gmock-1.7.0.zip
   rm gmock-1.7.0.zip
   mv gmock-1.7.0 gmock
miu@ubuntu:~/protobuf-3.0.0$
```

通过 autogen.sh 替换新的下载链接之后，再次运行该指令，以下是运行指令的顺序：

```
./autogen.sh
./configure --with-protoc=protoc --host=aarch64-linux-gnu --
prefix=ABSOLUTE__PATH/tengine/opl/protobuf_3.0.0
make
sudo make install
```

红色的 ABSOLUTE__PATH 表示 Tengine 的绝对目录，在本示例中 home/miu/tengine/。

所以指令为：

```
./configure --with-protoc=protoc --host=aarch64-linux-gnu --
prefix=/home/miu/tengine/opl/protobuf_3.0.0
```

2.8 makefile.config

Tengine 目录下有 makefile.config 文件（原始名称为 makefile.config.example），强烈建议你创建名为 makefile.config 的副本。文件内容如下

```
# cross compile for ARM64
# CROSS_COMPILE=aarch64-linux-gnu-
```

这是交叉编译工具设置，如果你交叉编译 ARM64，则需要使能该条件。

```
# cross compile for ARM32
# CROSS_COMPILE=arm-linux-gnueabihf-
```

这是交叉编译工具设置，如果你交叉编译 ARM32，则需要使能该条件。

ARM64 与 ARM32 一次只使用其中一个，或者不使用其中一个。

```
# 只是为了与 SysRoot 区别，以便构建嵌入式工具链
```

OPEN AI LAB

```
# 因为工具链只需要一些预构建的库
#
# EMBEDDED_CROSS_ROOT=/opt/install/
```

```
# 设置目标 arch
 CONFIG_ARCH_ARM64=y
# CONFIG_ARCH_ARM32=y
```

在本地编译代码时，可以启用目标条件

```
# 启用编译优化
CONFIG_OPT_CFLAGS = -O2
```

编译优化，我们建议你不要更改它。

```
# 使用 BLAS 作为运算符实现
# CONFIG_ARCH_BLAS=y
```

如果要在 x86 Linux 上本地编译 Tengine，则需要启用该条件。

```
# 启用 GPU 支持 ARM 计算库
# CONFIG_ACL_GPU=y
```

```
# 设置 ACL 路径
# ACL_ROOT=/home/firefly/ComputeLibrary
```

```
# 启用相应的序列化文件
CONFIG_CAFFE_SERIALIZER=y
# CONFIG_MXNET_SERIALIZER=y
# CONFIG_ONNX_SERIALIZER=y
# CONFIG_TF_SERIALIZER=y
# CONFIG_TFLITE_SERIALIZER=y
CONFIG_TENGINE_SERIALIZER=y
```

注意，你需要启用要使用的正确条件。

```
# 启用 Wrappers
```

OPEN AI LAB

```
# CONFIG_FRAMEWORK_WRAPPER=y
```

如果要使用 tensorflow 或 caffe 的 wrapper，则需要启用该条件

```
# version postfix
CONFIG_VERSION_POSTFIX=github

# support legacy API
CONFIG_LEGACY_API=y
```

This is the condition for support legacy API , we recommend you enable it.

```
# kernel configuration
CONFIG_KERNEL_FP32=y
CONFIG_KERNEL_FP16=y
CONFIG_KERNEL_INT8=y
CONFIG_KERNEL_UINT8=y
```

上面的条件用于配置 kernel 模式，我们建议你启用所有这些条件

OPEN AI LAB

3 Tengine C API 介绍

3.1 宏定义与结构体

```
#define  MAX_SHAPE_DIM_NUM                4

/* tensor 的数据类型 */
#define  TENGINE_DT_FP32                  0
#define  TENGINE_DT_FP16                  1
#define  TENGINE_DT_INT8                  2
#define  TENGINE_DT_UINT8                 3
#define  TENGINE_DT_INT32                 4
#define  TENGINE_DT_INT16                 5

/* layout 格式 */
#define  TENGINE_LAYOUT_NCHW              0
#define  TENGINE_LAYOUT_NHWC              1

/* tensor type: the content changed or not during inference */
#define  TENSOR_TYPE_UNKNOWN              0
#define  TENSOR_TYPE_VAR                  1
#define  TENSOR_TYPE_CONST                2
#define  TENSOR_TYPE_INPUT                3
#define  TENSOR_TYPE_DEP                  4

/* node 转储操作定义？？ */
#define  NODE_DUMP_ACTION_DISABLE         0
#define  NODE_DUMP_ACTION_ENABLE          1
#define  NODE_DUMP_ACTION_START           2
#define  NODE_DUMP_ACTION_STOP            3
#define  NODE_DUMP_ACTION_GET             4

/* graph 性能动作定义 */
#define  GRAPH_PERF_STAT_DISABLE          0
#define  GRAPH_PERF_STAT_ENABLE           1
#define  GRAPH_PERF_STAT_STOP             2
#define  GRAPH_PERF_STAT_START            3
#define  GRAPH_PERF_STAT_RESET            4
#define  GRAPH_PERF_STAT_GET              5

/* 遵循标准的 Unix 日志级别定义 */
enum log_level
{
```

```
        LOG_EMERG,
        LOG_ALERT,
        LOG_CRIT,
        LOG_ERR,
        LOG_WARNING,
        LOG_NOTICE,
        LOG_INFO,
        LOG_DEBUG
};

/* 注意：Android nn 仅定义一个事件？？ */
enum graph_exec_event
{
        GRAPH_EXEC_START,
        GRAPH_EXEC_SUSPEND,
        GRAPH_EXEC_RESUME,
        GRAPH_EXEC_ABORT,
        GRAPH_EXEC_DONE
};

enum graph_exec_stat
{
        GRAPH_STAT_CREATED,
        GRAPH_STAT_READY,
        GRAPH_STAT_RUNNING,
        GRAPH_STAT_DONE,
        GRAPH_STAT_ERROR
};

enum device_policy
{
        DEFAULT_POLICY,
        LATENCY_POLICY,
        LOW_POWER_POLICY
};

typedef void * context_t;
typedef void * graph_t;
typedef void * tensor_t;
typedef void * node_t;

typedef int (*event_handler_t)(graph_t, int, void* arg);

typedef void (*log_print_t)(const char*);


/* 性能分析记录 */
```

```
struct perf_info
{
    const char* name;            /* node 名称 */
    const char* dev_name;        /* 设备名称 */
    uint32_t count;
    uint32_t min;
    uint32_t max;
    uint64_t total_time;         /* us 或 cycle，取决于设备 */
    uint32_t base;               /* 1ms second time number */
};


struct custom_kernel_tensor
{
    int dim[MAX_SHAPE_DIM_NUM];  /* 形状的数组 */
    int dim_num;                 /* valid entry number */
    int element_num;
    int element_size;            /* 由数据类型(data_type)决定 */
    int data_type;
    int dev_type;                /* 表示张量属于 CPU/GPU ... */
    int layout_type;             /* NCHW 类型或者 NHWC 类型 */

    /* quant info */
    int quant_type;              /* int8, int16 or int32 */
    float* scale;
    int* zero_point;
    int* quant_number;

    void* data;                  /* pointer to host memory (virtual address) */
    void* dev_mem;               /* refers to device memory block */
    void* mapped_mem;            /* the mapped dress for device memory block */
};
//
/* 供用户添加用户定义的 Kernel */
struct custom_kernel_ops
{
    const char* kernel_name;     /* kernel 名称 */
    const char* op;              /* 要实现的 op 的名称 */
    int force;                   /* if not set, when bind() failed,
                try to use other kernel implementations*/
    void* kernel_param;          /* 用于内核实现的函数 */
    int kernel_param_size;

    /*!
    * @brief 根据输入形状生成输出形状
```

```
     *         如果未实现，请将其设置为 NULL。
     *
     * @param [in]  ops: 自定义 kernel 的指针。
     * @param [in]  inputs[]:  指向输入张量形状的指针数组。
     *                         形状为 MAX_SHAPE_DIM_NUM 元素，
     *                         值为 0 的元素表示形状的结尾。
     * @param [in]  input_num: 输入张量的数目。
     * @param [out] outputs[]: 指向已分配内存的输出张量形状的指针数组。
     * @param [in]  output_num: 输出张量的数目
     * @param [in]  layout: 图形布局为 NHWC 或 NCHW
     *
     * @return 0: success, -1: fail.
     */
    int (*infer_shape)(struct custom_kernel_ops* ops, const int* inputs[],
                  int input_num, int* outputs[],int output_num, int layout);

    /*!
     * @brief Get the inplace input tensor index for an output tensor.
     *
     * @param [in] ops: 自定义内核的指针。
     * @param [in] output_idx: 自定义内核输出的索引。
     *
     * @return the inplace input tensor index for an output tensor.
     *         if the output tensor is not an inplace one, return -1.
     */
    int (*inplace_info)(struct custom_kernel_ops* ops, int output_idx); // 可选项

    /*!
     * @brief 检查内核是否可以处理输入和输出形状。
     *
     * @param [in] ops: 自定义 kernel 的指针。
     * @param [in] inputs[]: 用于输入的自定义 kernel 的 tensor
     * @param [in] input_num: input tensors 的数目
     * @param [in] outputs[]: 自定义 kernel 的 output tensor
     * @param [in] output_num: output tensors 的数目
     *
     * @return 0 if the input and output are supported
     *         otherwise, return -1.
     *
     * notes: If not implemented, set it NULL, which means always return 0.
     */
    int (*bind)(  struct custom_kernel_ops* ops,
                  const struct custom_kernel_tensor* inputs[], int input_num,
                  const struct custom_kernel_tensor* outputs[], int output_num);
```

```
        /*!
        * @brief 预运行 graph.
        *        dynamic_shape 意味着 input_num 为零时不是异常情况。
        *
        * @param [in] ops: 自定义 kernel 的指针。
        * @param [in] inputs[]: 自定义 kernel 的 input tensor
        * @param [in] input_num: 自定义 kernel 的 input tensor 的数目
        * @param [in] outputs[]: 自定义 kernel 的 output tensor
        * @param [in] output_num: 自定义 kernel 的 output tensor 的数目
        * @param [in] dynamic_shape: It is not an abnormal case when input_num is
zero.? ?
        *
        * @return 0: success, -1: fail.
        */
        int (*prerun)(struct custom_kernel_ops* ops,
                      struct custom_kernel_tensor* inputs[], int input_num,
                      struct custom_kernel_tensor* outputs[], int output_num,
                      int dynamic_shape);
        /*!
        * @brief Reshape the graph.
        *
        * @param [in] ops: 自定义 kernel 的指针.
        * @param [in] inputs[]: 自定义 kernel 的 input tensor
        * @param [in] input_num: 自定义 kernel 的 input tensor 的数目
        * @param [in] outputs[]: 自定义 kernel 的 output tensor
        * @param [in] output_num: 自定义 kernel 的 output tensor 的数目
        *
        * @return 0: success, -1: fail.
        *
        * notes: 当输入形状更改时，将被调用它。
        *        调用 prerun()后，需要回收并重新分配运行时
        *        资源取决于输入形状
        */
        int (*reshape)(struct custom_kernel_ops* ops,
                      struct custom_kernel_tensor* inputs[], int input_num,
                      struct custom_kernel_tensor* outputs[], int output_num);

        /*!
        * @brief 运行 graph.
        *
        * @param [in] ops: 自定义 kernel 的指针.
        * @param [in] inputs[]: 自定义 kernel 的 input tensor
        * @param [in] input_num: 自定义 kernel 的 input tensor 的数目
        * @param [in] outputs[]: 自定义 kernel 的 output tensor
```

```
     * @param [in] output_num: 自定义 kernel 的 output tensor 的数目
     *
     * @return 0: success, -1: fail.
     */
     int (*run)(struct custom_kernel_ops* ops,
                  struct custom_kernel_tensor* inputs[], int input_num,
                  struct custom_kernel_tensor* outputs[], int output_num);


     /*!
     * @brief 暂停 graph 并释放 graph 运行时使用的资源。
     *
     * @param [in] ops: 自定义 kernel 的指针.
     * @param [in] inputs[]: 自定义 kernel 的 input tensor
     * @param [in] input_num: 自定义 kernel 的 input tensor 的数目
     * @param [in] outputs[]: 自定义 kernel 的 output tensor
     * @param [in] output_num: 自定义 kernel 的 output tensor 的数目
     *
     * @return 0: success, -1: fail.
     */
     int (*postrun)(struct custom_kernel_ops* ops,
                  struct custom_kernel_tensor* inputs[], int input_num,
                  struct custom_kernel_tensor* outputs[], int output_num);

     /*!
     * @brief 释放分配给此 ops 实现的资源。
     *
     * @param [in] ops: 自定义 kernel 的指针.
     *
     * @return None.
     */
     void (*release)(struct custom_kernel_ops* ops);
};
```

3.2 API 介绍

Tengine 支持的主要 APIs 是：

- init_tengine：初始化 tengine.
- release_tengine：释放 tengine 资源
- get_tengine_version：获取 tengine 版本号
- request_tengine_version：检查 tengine 版本号是否支持
- create_graph：创建 graph

- save_graph：保存 graph 到文件
- set_graph_layout：设置 graph 的布局类型(layout)
- set_graph_input_node：设置 graph 的输入节点(node)
- set_graph_output_node：设置 graph 的输出节点(node)
- merge_graph：将多个 graph 合并为一个 graph
- destroy_graph：销毁 graph
- get_graph_input_node_number：获取 graph 的输入节点(node)的个数
- get_graph_input_node：通过 ID 获取 gragh 的输入节点(node)
- get_graph_output_node_number：获取 graph 的输出节点(node)的个数
- get_graph_output_node：通过 ID 获取 gragh 的输出节点(node)
- get_graph_output_tensor：通过 ID 获取 graph 的输出节点的张量(tensor)
- get_graph_input_tensor：通过 ID 获取 graph 的输入节点的张量(tensor)
- create_graph_node：创建 graph 的节点(node)
- get_graph_node：获取节点(node)句柄
- get_node_name：获取节点(node)的名称
- get_node_op：获取节点(node)的操作(operation)
- release_graph_node：释放 graph 的节点(node)
- get_node_input_tensor：通过 ID 获取节点(node)的输入张量(tensor)
- get_node_output_tensor：通过 ID 获取节点(node)的输出张量(tensor)
- set_node_input_tensor：设置节点(node)的输入张量(tensor)
- set_node_output_tensor：设置节点(node)的输出张量(tensor)
- get_node_output_number：获取节点(node)的输出张量(tensor)个数
- get_node_input_number：获取节点(node)的输入张量(tensor)个数
- add_node_attr：添加节点(node)参数(attribute)
- get_node_attr_int：获取节点(node)的整型参数
- get_node_attr_float：获取节点(node)的浮点型参数
- get_node_attr_pointer：获取节点(node)的指针型参数
- get_node_attr_generic：获取节点(node)的通用型参数
- set_node_attr_int：设置节点(node)的整型参数
- set_node_attr_float：设置节点(node)的浮点型参数
- set_node_attr_pointer：设置节点(node)的指针型参数
- set_node_attr_generic：设置节点(node)的通用型参数
- set_custom_kernel：设置设置用户内核(kernel)
- remove_custom_kernel：移除用户内核(kernel)
- create_graph_tensor：创建 graph 的张量(tensor)

OPEN AI LAB

- get_graph_tensor：通过名称获取 graph 的张量(tensor)
- get_tensor_name：获取张量(tensor)的名称
- release_graph_tensor：释放 graph 的张量(tensor)
- get_tensor_shape：获取张量(tensor)的形状(shape)
- set_tensor_shape：设置张量(tensor)的形状(shape)
- get_tensor_buffer_size：获取张量(tensor)的缓存大小
- get_tensor_buffer：获取张量(tensor)的缓存(buffer)
- set_tensor_buffer：设置张量(tensor)的缓存(buffer)
- get_tensor_data：获取张量(tensor)的数据(data)
- set_tensor_data：设置张量(tensor)的数据(data)
- get_tensor_data_type：获取张量(tensor)的数据类型
- set_tensor_data_type：设置张量(tensor)的数据类型
- set_tensor_quant_param：设置 tensor 的量化参数
- get_tensor_quant_param：获取 tensor 的量化参数
- set_graph_attr：设置 graph 的参数
- get_graph_attr：获取 graph 的参数
- set_graph_gd_method：设置 graph 的梯度下降方法
- prerun_graph：graph 的预运行
- run_graph：运行 graph
- wait_graph：等待 graph
- postrun_graph：停止运行 graph 并释放 graph 占据的资源
- get_graph_exec_status：获取 graph 的执行状态
- set_graph_event_hook：设置 graph 的事件回调钩子
- set_default_device：设置默认设备
- set_graph_device： 设置 graph 的执行设备
- set_node_device： 设置 node 的执行设备
- get_node_device： 获取 node 的执行设备
- do_node_dump： 使能节点的 dump 功能
- get_node_dump_buffer： 获取节点的 dump 数据
- do_graph_perf_stat： 启用或禁用性能统计
- get_graph_perf_stat：获取性能统计信息
- get_device_number：获取设备编号
- get_device_name： 按 index 获取设备名称
- get_default_device：获取默认设备
- create_device：创建设备

OPEN AI LAB

- destroy_device：销毁设备
- set_device_policy：设置设备策略
- get_device_policy：获取设备策略
- get_device_attr：获取设备参数
- set_device_attr：设置设备参数
- create_context：创建 context
- destroy_context：销毁 context
- get_context_device_number：获取 context 设备数量
- get_context_device_name：获取 context 设备名称
- add_context_device：将设备添加到 context 中
- remove_context_device：将设备移除出 context
- set_context_attr：设置 context 属性
- get_context_attr：获取 context 属性
- get_tengine_errno：获取错误数量
- set_log_level：设置 log 级别
- set_log_output：设置日志打印功能
- dump_graph：dump 图的结构
- load_tengine_plugin：加载 Tengine 插件
- unload_tengine_plugin：移除 Tengine 插件
- get_tengine_plugin_number：获取 Tengine 插件数量
- get_tengine_plugin_name：通过索引获取插件的名称

3.2.1 Tengine 创建与释放相关 API

3.2.1.1 init_tengine

名称	init_tengine	
介绍	使用 tengine 库前，首先需要执行 init_tengine()，对 tengine 库进行初始化	
参数	无	
返回值	int	0 ：表示成功 -1 ：表示失败
原型	int init_tengine(void);	

OPEN AI LAB

3.2.1.2 release_tengine

名称	release_tengine	
介绍	执行 release_tengine()，对 tengine 库进行资源释放操作	
参数	无	
返回值	无	
原型	void release_tengine(void);	

3.2.1.3 get_tengine_version

名称	get_tengine_version	
介绍	获取 Tengine 库的版本信息	
参数	无	
返回值	const char *	版本的字符串指针
原型	const char * get_tengine_version(void);	

3.2.1.4 request_tengine_version

名称	request_tengine_version	
介绍	初始化 Tengine 后，通过该接口获取对版本的兼容情况	
参数	const char * version	(I) 版本号字符串的指针
返回值	int	1： 表示支持 0： 表示不支持
原型	int request_tengine_version(const char * version);	

OPEN AI LAB

3.2.2 graph 相关操作

3.2.2.1 create_graph

名称	create_graph	
介绍	创建 graph	
参数	context_t context	(I) context 的句柄
	const char * model_format	(I) 采用模型的类型，比如“tengine”，“mxnet”，“caffe”，“onnx”，“tensorflow”等
	const char * file_name	(I)文件名称
	...	(I) 其他参数
返回值	graph_t	Graph 句柄， 如果为 NULL 表示失败
原型	graph_t create_graph(context_t context, const char * model_format, const char * file_name,...);	

3.2.2.2 save_graph

名称	save_graph	
介绍	保存 graph	
参数	graph_t graph	(I) graph 的句柄
	const char * model_format	(I) 采用模型的类型，比如“tengine”，“mxnet”，“caffe”，“onnx”，“tensorflow”等
	const char * file_name	(I)文件名称
	...	(I) 其他参数
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int save_graph(graph_t graph, const char * model_format, const char *	

OPEN AI LAB

	file_name, ...)

3.2.2.3 set_graph_layout

名称	set_graph_layout	
介绍	设置 graph 的 layout	
参数	graph_t graph	(I) graph 的句柄

	int layout_type	(I) Graph 的 layout 类型，比如：
		`TENGINE_LAYOUT_NCHW`
		`TENGINE_LAYOUT_NHWC`

返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int set_graph_layout(graph_t graph, int layout_type);	

3.2.2.4 set_graph_input_node

名称	set_graph_input_node	
介绍	设置 graph 的 input node	
参数	graph_t graph	(I) graph 的句柄
	const char * input_nodes[]	(I) 输入节点数组的指针，char 类型
	int input_number	(I) 输入节点的数量
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int set_graph_input_node(graph_t graph, const char * input_nodes[], int input_number);	

OPEN AI LAB

3.2.2.5 set_graph_output_node

名称	set_graph_output_node	
介绍	设置 graph 的 output node	
参数	graph_t graph	(I) graph 的句柄
	const char * output_nodes[]	(I) 输出节点数组的指针，char 类型
	int output_number	(I) 输出节点的数量
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int set_graph_output_node(graph_t graph, const char * output_nodes[], int output_number);	

3.2.2.6 merge_graph

名称	merge_graph	
介绍	将多个 graph 合并为一个 graph	
参数	int graph_num	(I) graph 句柄的个数
	graph_t graph0	(I) graph 句柄 0
	graph_t graph1	(I) graph 句柄 1
	...	(I) 其他参数
返回值	graph_t	graph 句柄，如果为 NULL 表示失败
原型	graph_t merge_graph(int graph_num , graph_t graph0, graph_t graph1, ...);	

3.2.2.7 destroy_graph

名称	destroy_graph

OPEN AI LAB

介绍	销毁 graph	
参数	graph_t graph	(I) graph 句柄
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int destroy_graph(graph_t graph);	

3.2.2.8 get_graph_input_node_number

名称	get_graph_input_node_number	
介绍	获取 graph 的输入节点(node)数目	
参数	graph_t graph	(I) graph 句柄
返回值	int	>0 ： node 个数 -1 ： failure
原型	int get_graph_input_node_number(graph_t graph);	

3.2.2.9 get_graph_input_node

名称	get_graph_input_node	
介绍	通过 ID 获取 gragh 的输入节点(node)	
参数	graph_t graph	(I) graph 句柄
	int idx	(I) Node 下标(>=0)
返回值	node_t	node 句柄，如果为 NULL 表示失败
原型	node_t get_graph_input_node(graph_t graph, int idx);	

3.2.2.10 get_graph_output_node_number

名称	get_graph_output_node_number

OPEN AI LAB

介绍	获取 graph 的输出节点(node)个数	
参数	graph_t graph	(I) graph 句柄
返回值	int	>0 ： node 编号 -1 ： 失败
原型	int get_graph_output_node_number(graph_t graph);	

3.2.2.11 get_graph_output_node

名称	get_graph_output_node	
介绍	通过 ID 获取 gragh 的输出节点(node)	
参数	graph_t graph	(I) graph 句柄
	int idx	(I) Node 下标(>=0)
返回值	node_t	node 句柄，如果为 NULL 表示失败
原型	node_t get_graph_output_node(graph_t graph, int idx);	

3.2.2.12 get_graph_output_tensor

名称	get_graph_output_tensor	
介绍	按 ID 获取 graph 的输出张量(tensor)	
参数	graph_t graph	(I) graph 句柄
	int output_node_idx	(I) Node 下标(>=0)
	int tensor_idx	(I) tensor 下标（>=0）
返回值	tensor_t	tensor 句柄，如果为 NULL 表示失败
原型	tensor_t get_graph_output_tensor(graph_t graph, int output_node_idx, int tensor_idx);	

OPEN AI LAB

3.2.2.13 get_graph_input_tensor

名称	get_graph_input_tensor	
介绍	按 ID 获取 graph 的输入张量(tensor)	
参数	graph_t graph	(I) graph 句柄
	int input_node_idx	(I) Node 下标(>=0)
	int tensor_idx	(I) tensor 下标（>=0）
返回值	tensor_t	tensor 句柄，如果为 NULL 表示失败
原型	tensor_t get_graph_input_tensor(graph_t graph, int input_node_idx, int tensor_idx);	

3.2.2.14 get_graph_node_number

名称	get_graph_node_number	
介绍	获取 graph 的 node 数目	
参数	graph_t graph	(I) graph 句柄
返回值	int	>=0 ： 表示成功 -1 ： 表示失败
原型	int get_graph_node_number(graph_t graph);	

3.2.2.15 get_graph_node_by_idx

名称	get_graph_node_by_idx	
介绍	通过下标获取 graph 的 node	
参数	graph_t graph	(I) graph 句柄
	int node_idx	(I) node 的下标

OPEN AI LAB

返回值	node_t	node 句柄，如果为 NULL 表示失败
原型	node_t get_graph_node_by_idx(graph_t graph, int node_idx);	

3.2.3 node 相关操作

3.2.3.1 create_graph_node

名称	create_graph_node	
介绍	创建 graph 的 node	
参数	graph_t graph	(I) graph 句柄
	const char * node_name	(I) node 的名称
	const char * op_name	(I) 操作名称
返回值	node_t	node 句柄，如果为 NULL 表示失败
原型	node_t create_graph_node(graph_t graph, const char * node_name, const char * op_name);	

3.2.3.2 get_graph_node

名称	get_graph_node	
介绍	获取 graph 的 node 的句柄	
参数	graph_t graph	(I) graph 句柄
	const char * node_name	(I) node 的名称
返回值	node_t	node 句柄，如果为 NULL 表示失败
原型	node_t get_graph_node(graph_t graph, const char * node_name);	

OPEN AI LAB

3.2.3.3 get_node_name

名称	get_node_name	
介绍	获取 node 的名称	
参数	node_t node	(I) node 句柄
返回值	const char *	node 句柄名称指针，如果为 NULL 表示失败
原型	const char * get_node_name(node_t node);	

3.2.3.4 get_node_op

名称	get_node_op	
介绍	获取 node 使用的操作(op)	
参数	node_t node	(I) node 句柄
返回值	const char *	node 的 op 名称指针，如果为 NULL 表示失败
原型	const char * get_node_op(node_t node);	

3.2.3.5 release_graph_node

名称	release_graph_node	
介绍	释放 graph 的 node	
参数	node_t node	(I) node 句柄
返回值	无	
原型	void release_graph_node(node_t node);	

OPEN AI LAB

3.2.3.6 get_node_input_tensor

名称	get_node_input_tensor	
介绍	获取 node 的输入张量(tensor)	
参数	node_t node	(I) node 句柄
	int input_idx	(I) Input 的下标（>=0）
返回值	tensor_t	tensor 的句柄，如果为 NULL 表示失败
原型	tensor_t get_node_input_tensor(node_t node, int input_idx);	

3.2.3.7 get_node_output_tensor

名称	get_node_output_tensor	
介绍	获取 node 的输出张量(tensor)	
参数	node_t node	(I) node 句柄
	int output_idx	(I) Output 的下标（>=0）
返回值	tensor_t	tensor 的句柄，如果为 NULL 表示失败
原型	tensor_t get_node_output_tensor(node_t node, int output_idx);	

3.2.3.8 set_node_input_tensor

名称	set_node_input_tensor	
介绍	设置 node 的输入张量(tensor)	
参数	node_t node	(I) node 句柄
	int input_idx	(I) Input 的下标（>=0）
	tensor_t tensor	(I) tensor 的句柄

OPEN AI LAB

返回值	int	0 ： 表示设置成功 -1 ： 表示设置失败
原型	int set_node_input_tensor(node_t node, int input_idx, tensor_t tensor);	

3.2.3.9 set_node_output_tensor

名称	set_node_output_tensor	
介绍	设置 node 的输出张量(tensor)	
参数	node_t node	(I) node 句柄
	int output_idx	(I) Input 的下标（>=0）
	tensor_t tensor	(I) tensor 的句柄
	int tensor_type	(I) tensor 类型
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int set_node_output_tensor(node_t node, int output_idx, tensor_t tensor, int tensor_type);	

3.2.3.10 get_node_output_number

名称	get_node_output_number	
介绍	获取 node 的输出张量(tensor)的个数	
参数	node_t node	(I) node 句柄
返回值	int	>0 ： 输出张量(tensor)的个数 -1 ： 表示失败
原型	int get_node_output_number(node_t node);	

OPEN AI LAB

3.2.3.11 get_node_input_number

名称	get_node_input_number	
介绍	获取 node 中输入张量(tensor)的个数	
参数	node_t node	(I) node 句柄
返回值	int	>0 ： 表示输入张量(tensor)的个数 -1 ： 表示失败
原型	int get_node_input_number(node_t node);	

3.2.3.12 add_node_attr

名称	add_node_attr	
介绍	添加 node 属性	
参数	node_t node	(I) node 句柄
	const char* attr_name	(I) 属性名称
	const char* type_name	(I) 属性类型名称
	int size	(I) 属性大小
返回值	int	0 ： success -1 ： failure
原型	int add_node_attr(node_t node, const char *attr_name, const char* type_name, int size);	

3.2.3.13 get_node_attr_int

名称	get_node_attr_int	
介绍	获取 node 的 int 类型参数	
参数	node_t node	(I) node 句柄

OPEN AI LAB

	const char * attr_name	(I) 参数名称
	int * attr_val	(O) int 参数返回值指针
返回值	int	0 ： 表示获取成功 -1 ： 表示失败
原型	int get_node_attr_int(node_t node, const char * attr_name, int * attr_val);	

3.2.3.14 get_node_attr_float

名称	get_node_attr_float	
介绍	获取 node 的 float 类型参数	
参数	node_t node	(I) node 句柄
	const char * attr_name	(I) 参数名称
	float * attr_val	(O) float 参数返回值的指针
返回值	int	0 ： 表示获取成功 -1 ： 表示失败
原型	int get_node_attr_float(node_t node, const char * attr_name, float * attr_val);	

3.2.3.15 get_node_attr_pointer

名称	get_node_attr_pointer	
介绍	获取 node 的指针类型参数	
参数	node_t node	(I) node 句柄
	const char * attr_name	(I) 参数名称
	void * attr_val	(O) pointer 参数的指针

OPEN AI LAB

返回值	int	0 ： 表示获取成功 -1 ： 表示获取失败
原型	int get_node_attr_pointer(node_t node, const char * attr_name, void * attr_val);	

3.2.3.16 get_node_attr_generic

名称	get_node_attr_generic	
介绍	获取 node 的通用参数	
参数	node_t node	(I) node 句柄
	const char * attr_name	(I) 参数名称
	const char * type_name	(I) 参数类型名称
	void * buf	(O) 数据的 buffer
	int size	(I) 数据的 buffer 的尺寸
返回值	int	0 ： 表示获取成功 -1 ： 表示获取失败
原型	int get_node_attr_generic(node_t node, const char * attr_name, const char* type_name, void * buf, int size);	

3.2.3.17 set_node_attr_int

名称	set_node_attr_int	
介绍	设置 node 的 int 类型参数	
参数	node_t node	(I) node 句柄
	const char * attr_name	(I) 参数名称
	const int * attr_val	(I) int 参数返回值的指针

OPEN AI LAB

返回值	int	0 ： 表示获取成功 -1 ： 表示获取失败
原型	int set_node_attr_int(node_t node, const char * attr_name, const int *attr_val);	

3.2.3.18 set_node_attr_float

名称	set_node_attr_float	
介绍	设置 node 的 float 类型参数	
参数	node_t node	(I) node 句柄
	const char * attr_name	(I) 参数名称
	const float * attr_val	(I) float 型返回值的指针
返回值	int	0 ： 表示获取成功 -1 ： 表示获取失败
原型	int set_node_attr_float(node_t node, const char * attr_name, const float *attr_val);	

3.2.3.19 set_node_attr_pointer

名称	set_node_attr_pointer	
介绍	设置 node 的指针类型参数	
参数	node_t node	(I) node 句柄
	const char * attr_name	(I) 参数名称
	const void * attr_val	(I) 指针类型参数的指针
返回值	int	0 ： 表示获取成功 -1 ： 表示获取失败
原型	int set_node_attr_pointer(node_t node, const char * attr_name, const void	

OPEN AI LAB

	*attr_val);

3.2.3.20 set_node_attr_generic

名称	set_node_attr_generic	
介绍	设置 node 的通用参数	
参数	node_t node	(I) node 句柄
	const char * attr_name	(I) 参数名称
	const char* type_name	(I) 类型名称
	const void * buf	(I) 数据的 buffer
	int size	(I) 数据的 buffer 的尺寸
返回值	int	0 ： 表示获取成功 -1： 表示获取失败
原型	int set_node_attr_generic(node_t node, const char * attr_name, const char * type_name, const void * buf, int size);	

3.2.4 kernel 相关操作

3.2.4.1 set_custom_kernel

名称	set_custom_kernel	
介绍	设置自定义 kernel	
参数	node_t node	(I) node 句柄
	const char * dev_name	(I) 设备名称
	struct custom_kernel_ops * kernel_ops	(I) 内核操作结构体指针

OPEN AI LAB

返回值	int	0 ： 表示设置成功 -1： 表示设置失败
原型	int set_custom_kernel(node_t node, const char * dev_name, struct custom_kernel_ops * kernel_ops);	

3.2.4.2 remove_custom_kernel

名称	remove_custom_kernel	
介绍	移除自定义 kernel	
参数	node_t node	(I) node 句柄
	const char * dev_name	(I) 设备名称
返回值	int	0 ： 表示设置成功 -1： 表示设置失败
原型	int remove_custom_kernel(node_t node, const char * dev_name);	

3.2.5 The tensor operation

3.2.5.1 create_graph_tensor

名称	create_graph_tensor	
介绍	创建 graph 的 tensor	
参数	graph_t graph	(I) graph 的句柄
	const char * tensor_name	(I) tensor 的名称
	int data_type	(I) 数据类型
返回值	tensor_t	tensor 的句柄，如果为 NULL 表示失败
原型	tensor_t create_graph_tensor(graph_t graph, const char * tensor_name, int data_type);	

OPEN AI LAB

3.2.5.2 get_graph_tensor

名称	get_graph_tensor	
介绍	获取 graph 的 tensor	
参数	graph_t graph	(I) graph 的句柄
	const char * tensor_name	(I) tensor 的名称
返回值	tensor_t	tensor 的句柄，如果为 NULL 表示失败
原型	tensor_t get_graph_tensor(graph_t graph, const char * tensor_name);	

3.2.5.3 get_tensor_name

名称	get_tensor_name	
介绍	获取 tensor 的名称	
参数	tensor_t tensor	(I) tensor 的句柄
返回值	const char *	tensor 名称，如果为 NULL 表示失败
原型	const char * get_tensor_name(tensor_t tensor);	

3.2.5.4 release_graph_tensor

名称	release_graph_tensor	
介绍	释放 graph 的 tensor	
参数	tensor_t tensor	(I) tensor 的句柄
返回值	无	
原型	void release_graph_tensor(tensor_t tensor);	

OPEN AI LAB

3.2.5.5 get_tensor_shape

名称	get_tensor_shape	
介绍	获取 tensor 的 shape	
参数	tensor_t tensor	(I) tensor 的句柄
	int dims[]	(O) shape 的数组
	int dim_number	(I) 数组的尺寸
返回值	int	0 ： 表示成功 -1： 表示失败
原型	int get_tensor_shape(tensor_t tensor, int dims[], int dim_number);	

3.2.5.6 set_tensor_shape

名称	set_tensor_shape	
介绍	设置 tensor 的 shape	
参数	tensor_t tensor	(I) tensor 的句柄
	const int dims[]	(I) shape 的数组
	int dim_number	(I) 数组的尺寸
返回值	int	0 ： 表示设置成功 -1： 表示设置失败
原型	int set_tensor_shape(tensor_t tensor, const int dims[], int dim_number);	

3.2.5.7 get_tensor_buffer_size

名称	get_tensor_buffer_size
介绍	获取 tensor 的 buffer 的大小

OPEN AI LAB

参数	tensor_t tensor	(I) tensor 的句柄
返回值	Int	>0 ： buffer size -1 ： failure
原型	int get_tensor_buffer_size(tensor_t tensor);	

3.2.5.8 get_tensor_buffer

名称	get_tensor_buffer	
介绍	获取 tensor 的 buffer	
参数	tensor_t tensor	(I) tensor 的句柄
返回值	void *	Buffer 的指针，如果为 NULL 表示失败
原型	void * get_tensor_buffer(tensor_t tensor);	

3.2.5.9 set_tensor_buffer

名称	set_tensor_buffer	
介绍	设置 tensor 的 buffer	
参数	tensor_t tensor	(I) tensor 的句柄
	void * buffer	(I) Buffer 的指针
	int buffer_size	(I) 数据 buffer 的尺寸
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int set_tensor_buffer(tensor_t tensor, void * buffer, int buffer_size);	

3.2.5.10 get_tensor_data

名称	get_tensor_data

OPEN AI LAB

介绍	获取 tensor 的 data	
参数	tensor_t tensor	(I) tensor 的句柄
	void * output_data	(O) 输出 buffer 的指针
	int data_size	(I) 输出数据 buffer 的大小
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int get_tensor_data(tensor_t tensor, void * output_data, int data_size);	

3.2.5.11 set_tensor_data

名称	set_tensor_data	
介绍	设置 tensor 的数据	
参数	tensor_t tensor	(I) tensor 的句柄
	const void * input_data	(I) input 数据 buffer 的指针
	int data_size	(I) Input 数据的尺寸
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int set_tensor_data(tensor_t tensor, const void * input_data, int data_size);	

3.2.5.12 get_tensor_data_type

名称	get_tensor_data_type	
介绍	获取 tensor 的数据类型	
参数	tensor_t tensor	(I) tensor 的句柄

OPEN AI LAB

返回值	int	Tensor 类型。比如： TENGINE_DT_FP32、 TENGINE_DT_FP16、 TENGINE_DT_IN8，等等 -1 ： 失败
原型	int get_tensor_data_type(tensor_t tensor);	

3.2.5.13 set_tensor_data_type

名称	set_tensor_data_type	
介绍	设置 tensor 的数据类型	
参数	tensor_t tensor	(I) tensor 的句柄
	int data_type	(I) 数据类型，比如 ： TENGINE_DT_FP32、 TENGINE_DT_FP16、 TENGINE_DT_IN8，等等
返回值	int	0：success -1 ： failure
原型	int set_tensor_data_type(tensor_t tensor, int data_type);	

3.2.5.14 set_tensor_quant_param

名称	set_tensor_quant_param	
介绍	设置 tensor 的量化参数	
参数	tensor_t tensor	(I) tensor 的句柄
	const float *scale	(I) scale 指针
	const int *zero_point	(I) 起点指针

OPEN AI LAB

	int number	数据大小
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int set_tensor_quant_param(tensor_t tensor, const float *scale, const int *zero_point, int number);	

3.2.5.15 get_tensor_quant_param

名称	get_tensor_quant_param	
介绍	获取 tensor 的量化参数	
参数	tensor_t tensor	(I) 张量句柄
	float *scale	(I) scale 指针
	int *zero_point	(I) 起点指针
	int number	(I) 数据大小
返回值	int	0 ： 表示成功 -1 ： 表示失败

3.2.6 与 graph 运行相关 API

3.2.6.1 set_graph_attr

名称	set_graph_attr	
介绍	设置 graph 的参数	
参数	graph_t graph	(I) graph 的句柄
	const char * attr_name	(I) 要设置的参数名称
	const void * buf	(I) 设置的参数的指针
	int size	(I) 数据的尺寸

OPEN AI LAB

返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int set_graph_attr(graph_t graph, const char * attr_name, const void * buf, int size);	

3.2.6.2 get_graph_attr

名称	get_graph_attr	
介绍	获取 graph 的参数	
参数	graph_t graph	(I) graph 的句柄
	const char * attr_name	(I) 要获取的参数名称
	void * buf	(I) 要获取参数的指针
	int size	(I) 数据的大小
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int get_graph_attr(graph_t graph, const char * attr_name, void * buf, int size);	

3.2.6.3 set_graph_gd_method

名称	set_graph_gd_method	
介绍	设置图的梯度下降方法	
参数	graph_t graph	(I) graph 的句柄
	int gd_method	(I) 梯度下降函数
	...	(I) 可变参数
返回值	int	0 ： 表示成功 -1 ： 表示失败

OPEN AI LAB

原型	int set_graph_gd_method(graph_t graph, int gd_method, ...);

3.2.6.4 prerun_graph

名称	prerun_graph	
介绍	准备运行 graph，并准备资源	
参数	graph_t graph	(I) graph 的句柄
返回值	int	0 ：表示成功 -1 ：表示失败
原型	int prerun_graph(graph_t graph);	

3.2.6.5 run_graph

名称	run_graph	
介绍	运行 graph，执行推理，可以反复多次调用	
参数	graph_t graph	(I) graph 的句柄
	int block	1：阻塞. 需要配置 GRAPH_DONE 函数 0：非阻塞：需要调用 wait_graph()以获取结果
返回值	int	0 ：表示成功 -1 ：表示失败
原型	int run_graph(graph_t graph, int block);	

3.2.6.6 wait_graph

名称	wait_graph	
介绍	等待 graph 运行的结果	
参数	graph_t graph	(I) graph 的句柄

OPEN AI LAB

	int try_wait	1：检查状态，然后返回 0：立即返回
返回值	int	1 ： graph 运行完成 0 ： 未运行完成，需要再次尝试
原型	int wait_graph(graph_t graph, int try_wait);	

3.2.6.7 postrun_graph

名称	postrun_graph	
介绍	停止运行 graph 并释放 graph 占据的资源	
参数	graph_t graph	(I) graph 的句柄
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int postrun_graph(graph_t graph);	

3.2.6.8 get_graph_exec_status

名称	get_graph_exec_status	
介绍	获取 graph 的执行状态	
参数	graph_t graph	(I) graph 的句柄
返回值	int	graph 的运行状态，见 graph_exec_stat
原型	int get_graph_exec_status(graph_t graph);	

3.2.6.9 set_graph_event_hook

名称	set_graph_event_hook
介绍	设置 graph 的事件回调函数

OPEN AI LAB

参数	graph_t graph	(I) graph 的句柄
	int event	(I) 事件类型
	event_handler_t cb_func	(I) 事件回调函数指针
	void * cb_arg	(I) 回调参数
返回值	int	0 ： 表示成功 -1： 表示失败
原型	int set_graph_event_hook(graph_t graph, int event, event_handler_t cb_func, void * cb_arg);	

3.2.7 与设备相关操作

3.2.7.1 set_default_device

名称	set_default_device	
介绍	设置默认运行设备	
参数	const char * device	(I) 运行的设备名称
返回值	int	0 ： 表示成功 -1： 表示失败
原型	int set_default_device(const char * device);	

3.2.7.2 set_graph_device

名称	set_graph_device	
介绍	设置 graph 运行设备	
参数	graph_t graph	(I) graph 的句柄
	const char * dev_name	(I) 设备的名称

OPEN AI LAB

返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int set_graph_device(graph_t graph, const char * dev_name);	

3.2.7.3 set_node_device

名称	set_node_device	
介绍	设置 node 运行设备	
参数	node_t node	(I) node 的句柄
	const char * dev_name	(I) 设备的名称
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int set_node_device(node_t node, const char * dev_name);	

3.2.7.4 get_node_device

名称	get_node_device	
介绍	获取 node 运行设备	
参数	node_t node	(I) node 的句柄
返回值	const char *	设备的名称： NULL 意味着没有可利用的设备
原型	const char * get_node_device(node_t node);	

3.2.7.5 do_node_dump

名称	do_node_dump
介绍	设置节点的 dump 功能

OPEN AI LAB

参数	node_t node	(I) 节点句柄
	int action	(I) 1 使能，0 不使能
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int do_node_dump(node_t node, int action);	

3.2.7.6 get_node_dump_buffer

名称	get_node_dump_buffer	
介绍	获取节点的 dump 数据缓存区。dump 数据缓冲区的具体定义, 由产生 dump 的设备决定。	
参数	node_t node	(I) 节点句柄
	void ** buf	(O) 用于接收 dump 数据块指针的指针数组
	int buf_size	(I) 指针数组大小
返回值	int	指针数组中有效指针的个数。如果等于 buf_size，则存在有数据块没有返回的可能性
原型	int get_node_dump_buffer(node_t node, void **buf, int buf_size);	

3.2.7.7 do_graph_perf_stat

名称	do_graph_perf_stat	
介绍	启用或禁用性能统计	
参数	graph_t graph	(I) graph 的句柄
	int action	(I) 0 停止, 1 开始, 2 重新计数

OPEN AI LAB

返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int do_graph_perf_stat(graph_t graph, int action);	

3.2.7.8 get_graph_perf_stat

名称	get_graph_perf_stat	
介绍	获取性能统计信息	
参数	graph_t graph	(I) graph 的句柄
	perf_info ** buf	(O) 指向(输出性能信息的指针)的指针
	int buf_size	(I) 数据的大小
返回值	int	>=0：表示数据记录的个数 -1 ： 表示失败
原型	int get_graph_perf_stat(graph_t graph, struct perf_info ** buf, int buf_size);	

3.2.7.9 get_device_number

名称	get_device_number	
介绍	获取设备编号	
参数	无	
返回值	int	设备编号
原型	int get_device_number(void);	

3.2.7.10 get_device_name

名称	get_device_name
介绍	按 index 获取设备名称

OPEN AI LAB

参数	int idx	(I) 设备编号(>=0)
返回值	const char *	设备的名称： NULL 意味着失败
原型	const char * get_device_name(int idx);	

3.2.7.11 get_default_device

名称	get_default_device	
介绍	获取默认使用设备名称	
参数	无	
返回值	const char *	设备的名称： NULL 意味着失败
原型	const char * get_default_device(void);	

3.2.7.12 create_device

名称	create_device	
介绍	创建设备	
参数	const char * driver_name	驱动名称
	const char * dev_name	(I) 设备名称
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int create_device(const char * driver_name, const char * dev_name);	

3.2.7.13 destroy_device

名称	destroy_device
介绍	销毁设备

OPEN AI LAB

参数	const char * driver_name	(I) 驱动名称
	const char * dev_name	(I) 设备名称
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int destroy_device(const char * driver_name, const char * dev_name);	

3.2.7.14 set_device_policy

名称	set_device_policy	
介绍	设置设备策略	
参数	const char * driver_name	(I) 驱动名称
	device_policy policy	(I) 政策
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int set_device_policy(const char * device_name, device_policy policy);	

3.2.7.15 get_device_policy

名称	get_device_policy	
介绍	获取设备策略	
参数	const char * driver_name	(I) 驱动名称
返回值	int	>=0：策略值 -1 ： 表示失败
原型	int get_device_policy(const char * device_name);	

OPEN AI LAB

3.2.7.16 get_device_attr

名称	get_device_attr	
介绍	获取设备参数	
参数	const char * driver_name	(I) 驱动名称
	const char * attr_name	(I) 参数名称
	void * val	(O) 参数返回结果指针
	int size	(I) 参数返回值大小
返回值	int	0 ： 表示成功 -1： 表示失败
原型	int get_device_attr(const char * device_name, const char * attr_name, void * val, int size);	

3.2.7.17 set_device_attr

名称	set_device_attr	
介绍	设置设备参数	
参数	const char * driver_name	(I) 驱动名称
	const char * attr_name	(I) 参数名称
	void * val	(I) 参数指针
	int size	(I) 参数大小
返回值	int	0 ： 表示成功 -1： 表示失败
原型	int set_device_attr(const char * device_name, const char * attr_name, void * val, int size);	

OPEN AI LAB

3.2.8 与 context 相关部分

3.2.8.1 create_context

名称	create_context	
介绍	创建 context	
参数	const char * context_name	(I) context 的名称
	int empty_context	(I) 1：没有为该设备分配可用的设备 0：所有声明的设备都将添加到 context 中
返回值	context_t	Context 的句柄, 返回 NULL 表示创建失败
原型	context_t create_context(const char * context_name, int empty_context);	

3.2.8.2 destroy_context

名称	destroy_context	
介绍	销毁 context	
参数	context_t context	(I) context 的句柄
返回值	无	
原型	void destroy_context(context_t context);	

3.2.8.3 get_context_device_number

名称	get_context_device_number	
介绍	获取 context 设备编号	
参数	context_t context	(I) context 的句柄
返回值	int	context 使用的设备编号

OPEN AI LAB

原型	int get_context_device_number(context_t context);

3.2.8.4 get_context_device_name

名称	get_context_device_name	
介绍	获取 context 设备名称	
参数	context_t context	(I) context 的句柄
	int idx	(I) 索引
返回值	const char *	设备名称
原型	const char * get_context_device_name(context_t context, int idx);	

3.2.8.5 add_context_device

名称	add_context_device	
介绍	将设备添加到 context 中	
参数	context_t context	(I) context 的句柄
	const char * dev_name	(I) 设备名称
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int add_context_device(context_t context, const char * dev_name);	

3.2.8.6 remove_context_device

名称	remove_context_device	
介绍	将设备移除出 context	
参数	context_t context	(I) context 的句柄

OPEN AI LAB

	const char * dev_name	(I) device 名称
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int remove_context_device(context_t context, const char * dev_name);	

3.2.8.7 set_context_attr

名称	set_context_attr	
介绍	设置 context 参数	
参数	context_t context	(I) context 的句柄
	const char * attr_name	(I) 参数名称
	const void * val	(I) 设置参数值的指针
	int val_size	(I) 参数的值所占的大小
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int set_context_attr(context_t context, const char * attr_name, const void * val, int val_size);	

3.2.8.8 get_context_attr

名称	get_context_attr	
介绍	获取 context 属性	
参数	context_t context	(I) context 的句柄
	const char * attr_name	(I) 参数名称
	void * val	(O) 参数返回值的指针

OPEN AI LAB

	int val_size	(I) 参数的值所占的大小
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int get_context_attr(context_t context, const char * attr_name, void * val, int val_size);	

3.2.9 其他的 API

3.2.9.1 get_tengine_errno

名称	get_tengine_errno	
介绍	获取错误码	
参数	无	
返回值	int	tengine 错误码。定义遵循 glic
原型	int get_tengine_errno(void);	

3.2.9.2 set_log_level

名称	set_log_level	
介绍	设置 log 级别	
参数	log_level level	(I) 上下文的句柄
返回值	无	
原型	void set_log_level(enum log_level level);	

3.2.9.3 set_log_output

名称	set_log_output

OPEN AI LAB

介绍	设置 log 输出函数	
参数	log_print_t func	(I) 输出函数的指针
返回值	无	
原型	void set_log_output(log_print_t func);	

3.2.9.4 dump_graph

名称	dump_graph	
介绍	dump 图的结构	
参数	graph_t graph	(I) graph 的句柄
返回值	int	0 ： 表示成功 -1： 表示失败
原型	void dump_graph(graph_t graph);	

3.2.9.5 load_tengine_plugin

名称	load_tengine_plugin	
介绍	加载 Tengine 插件	
参数	const char * plugin_name	(I) 插件的名称
	const char * fname	(I) 文件的名称
	const char * init_func_name	(I) 初始化函数的名称
返回值	int	0 ： 表示成功 -1： 表示失败
原型	int load_tengine_plugin(const char * plugin_name, const char * fname, const char * init_func_name);	

OPEN AI LAB

3.2.9.6 unload_tengine_plugin

名称	unload_tengine_plugin	
介绍	移除 Tengine 插件	
参数	const char * plugin_name	(I) 插件的名称
	const char * rel_func_name	(I) release 函数的指针
返回值	int	0 ： 表示成功 -1 ： 表示失败
原型	int unload_tengine_plugin(const char * plugin_name, const char * rel_func_name);	

3.2.9.7 get_tengine_plugin_number

名称	get_tengine_plugin_number	
介绍	获取 Tengine 插件的编号	
参数	无	
返回值	int	插件数量
原型	int get_tengine_plugin_number(void);	

3.2.9.8 get_tengine_plugin_name

名称	get_tengine_plugin_name	
介绍	获取 Tengine 插件的名称	
参数	int idx	(I) 插件的编号(>=0)
返回值	const char *	插件名称
原型	const char * get_tengine_plugin_name(int idx);	

OPEN AI LAB

OPEN AI LAB

4 测试例程

在 Tengine 目录下有各种不同的例子。你可以按照这些步骤编译演示 demo 程序。

4.1 安装 opencv

如果已安装，请跳过这些步骤。如果没有安装，运行以下指令进行安装：

```
sudo apt-get install libopencv-dev
```

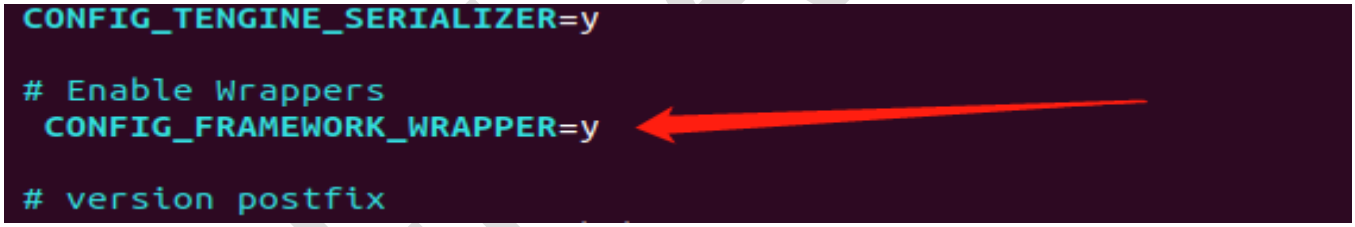
4.2 编译样例程序

4.2.1 Linux

在编译示例之前，需要确认已经安装了 cmake，如果没有，可以通过以下指令安装它。

```
sudo apt install cmake
```

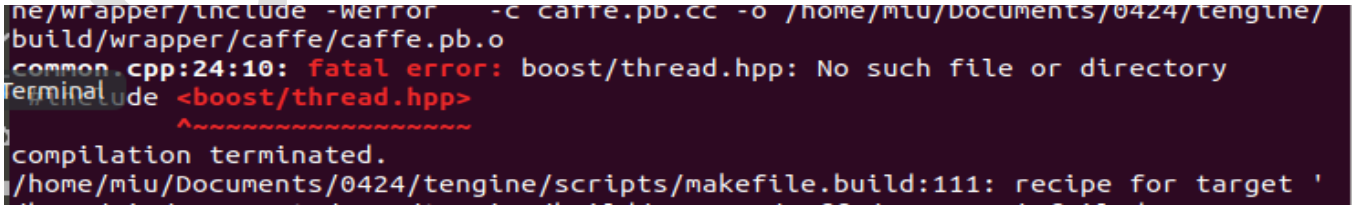
其次，你需要在 ***makefile.config*** 中启用 Wrapper。



保存 ***makefile.config*** 修改内容，然后再次运行 make 命令，按照步骤 2.2.4，你需要在 tengine 目录下运行 install 命令。

```
cd ~/tengine
make -j8
make install
```

如果有遇到错误提示，如下图所示：



你可以通过安装 boost 来修复它，请参考链接

https://blog.csdn.net/lixinneo/article/details/7679996

然后，

```
cd ~/tengine/examples
vim linux_build.sh
```

设置正确的 Tengine 路径。用你的 Tengine 的路径替换红色的单词。

```
#!/bin/bash


 cmake -DTENGINE_DIR=/home/usr/tengine ..
```

创建一个名为 build 的新目录并进入 build 目录，然后运行脚本进行编译：

```
mkdir build
cd build
../linux_build.sh
make -j4
```

4.2.2 Android

```
cd ~/tengine/examples
vim android_build_armv7.sh (or vim android_build_armv8.sh)
```

设置正确的 NDK 路径，Tengine 路径，Opencv 路径和 protobuf 路径。如果你想用 openblas 运行 Tengine，请在 android_build_armv7.sh 或者 android_build_armv8.sh 中设置正确的 blas 路径。

-DBLAS_DIR=/home/usr/openbla020_android

然后，运行以下指令：

```
mkdir build
cd build
../android_build_armv7.sh (or ../android_build_armv8.sh)
make -j4
```

编译完成后，按照 2.5.8 步骤验证 demo 程序。

4.3 分类网络结构

你可以从链接 Tengine model zoo（https://pan.baidu.com/s/1LXZ8vOdyOo50IXS0CUPp8g）(psw: 57vb).(备份链接为 : http://ftp.openailab.net.cn/Tengine_models/)中下载我们已经训练好的模型。

将这些模型文件存储在`${Tengine_ROOT}/models/`文件夹中

OPEN AI LAB

- squeezenet_v1.1(已存在于 Tengine_ROOT/models/)
- mobilenet_v1(已存在于 Tengine_ROOT/models/)
- mobilenet_v2
  - mobilenet_v2_deploy.prototxt
  - mobilenet_v2.caffemodel
- resnet50
  - resnet50.prototxt
  - resnet50.caffemodel
- googlenet
  - googlenet.prototxt
  - googlenet.caffemodel
- inception_v4
  - inception_v4.prototxt
  - inception_v4.caffemodel
- inception_v3
  - deploy_inceptionV3.prototxt
  - deploy_inceptionV3.caffemodel
- alexnet
  - alex_deploy.prototxt
  - alexnet.caffemodel
- vgg16
  - vgg16.prototxt
  - vgg16.caffemodel

4.3.1 编译例程

```
cd ${Tengine_ROOT}
make
make install
```

按照 ${TENGINE_ROOT}/examples/readme.md 中提到的步骤，进行编译，在第 4.2 章中也有详细说明。

4.3.2 运行

- 进入到可执行执行程序的目录：

```
cd ${Tengine_ROOT}/examples/build/imagenet_classification
```

- 测试图像：

OPEN AI LAB

[Usage]: ./Classify [-n model_name] [-p proto_file] [-m model_file] [-l label_file] [-i image_file][-g img_h,img_w] [-s scale] [-w mean[0],mean[1],mean[2]] [-r repeat_count] [-d device_name] [-h]

默认 model 名称： squeezenet

默认 label 文件： Tengine_ROOT/models/synset_words.txt

默认图片文件： Tengine_ROOT/tests/images/cat.jpg

默认图片尺寸： 227,227

默认缩放比例： 1.f

默认图片平均值： 104.007,116.669,122.679

- 使用示例：

```
    ./Classify -h                // 显示使用方法
    ./Classify -n squeezenet     // 使用默认图片(cat.jpg)测试 squeezenet 网络
    ./Classify -n mobilenet      // 使用默认图片(cat.jpg)测试 mobilenet 网络
    ./Classify -n mobilenet_v2   // 使用默认图片(cat.jpg)测试 mobilenet_v2 网络
    ./Classify -n resnet50       // 使用默认图片(cat.jpg)测试 resnet50 网络
    ./Classify -n alexnet        // 使用默认图片(cat.jpg)测试 alexnet 网络
    ./Classify -n googlenet      // 使用默认图片(cat.jpg)测试 googlenet 网络
    ./Classify -n inception_v3   // 使用默认图片(cat.jpg)测试 inception_v3 网络
    ./Classify -n inception_v4   // 使用默认图片(cat.jpg)测试 inception_v4 网络
    ./Classify -n vgg16 -i ~/tengine/tests/images/bike.jpg  // 使用自定义图片
(bike.jpg)测试 vgg16 网络

    // 使用自定义的参数有测试 squeezenet 网络结构
    ./Classify -n squeezenet -g 227,227 -s 1.f -w 104.007,116.669,122.679

    // 使用指定文件进行测试
    ./Classify -p ~/tengine/models/sqz.prototxt -m
~/tengine/models/squeezenet_v1.1.caffemodel
                -l ~/tengine/models/synset_words.txt -i
~/tengine/tests/images/bike.jpg
```

4.4 检测网络

你还可以验证检测网络 demo，下面是运行 caffe 模式的步骤。

4.4.1 下载 model

从 Tengine model zoo（https://pan.baidu.com/s/1LXZ8vOdyOo50IXS0CUPp8g） (密钥: 57vb) 链接中下载 caffe mode and prototxt。并将其保存到 models 目录（tengine/models）。

4.4.2 编译样例程序

按照 4.2 步骤构建样例下的 demo。

4.4.3 运行

进入 example/build 路径，选择你想要编译的 demo，这里我们以 mobilenet_ssd 为例。

```
cd  ${Tengine_ROOT}
cd examples/build/mobilenet_ssd/
./MSSD
```

等待运行完成，在显示检测结果的同一目录下保存了一张经过检测网络的图片。

OPEN AI LAB

5 一些工具与技巧

5.1 通用指令

5.1.1 设置使用的 CPU 数目

设置核心编号有两种不同的方法，一种是使用环境变量的方法设置，另一种是在代码中调用 API 的方法设置。

默认使用的 CPU 核心是 SOC 中功能最强的核。（例如，如果 SOC 为四核，一个是双核 A72，另一个是双核 A53，Tengine 默认将在双核 A72 上运行）

1) 环境变量

export TENGINE_CPU_LIST=0,1,2,…,N

例如，如果只在一个 CPU 核心上运行 demo，可以设置环境变量为（我们不建议将 tengine 设置在 0 核，容易与系统运行资源发生竞争）：

```
export TENGINE_CPU_LIST=3
```

如果要在双核上运行 demo，可以设置环境变量为：

```
export TENGINE_CPU_LIST=2,3
```

如果要在四核上运行 demo，可以设置环境变量为：

```
export TENGINE_CPU_LIST=0,1,2,3
```

2) 在代码中调用 API 方法

set_default_cpu(const int* cpu_list, int cpu_number)

注意: 该函数必须在 init_tengine()之前被调用，并且引用如下的头文件：

#include "cpu_device.h"

5.1.2 设置 kernel mode

kernel mode 将定义在线量化方法。有 3 种 kernel mode：FP32 模式、FP16 模式和 INT8 模式。默认为 FP32 模式。并且，有两种配置方法。

1) 使用环境变量进行配置

```
export KERNEL_MODE=0    //Float32 模式
export KERNEL_MODE=1    //Float16 模式
export KERNEL_MODE=2    //Int8 模式
```

OPEN AI LAB

2) 在代码中使用相关 API 进行配置

int val=1; // val 表示你想设置的 kernel 模式，0 表示 FP32，1 表示 FP16，2 表示 INT8

set_graph_attr(graph,"kernel_mode",&val,sizeof(val))

5. 设置运行 graph 的设备

set_graph_device(graph, device_name)

5.2 使用多线程运行 Tengine

你可以将文件 tests/bin/mt_ssd.cpp 作为参考。

另外，还有两个 demo 来演示如何在多线程模式下运行 Tengine 程序：

1) 线程池(pooling multi-thread)模式：

int val=1;

set_graph_attr(graph,"pooling_mt",&val,sizeof(val))

2) FC mutli-thread 模式：

int val=1;

set_graph_attr(graph,"fc_mt",&val,sizeof(val))

5.3 支持 CPU 和 GPU 的异构计算

随着深度学习的快速发展，把深度学习算法部署到嵌入式设备的需求也日益增加。目前开源的嵌入式推理框架中，有支持 ARM CPU 加速推理的，也有支持 GPU 加速的，但这些移动端推理框架中，还没有一个框架能够进行 CPU/GPU 异构调度。如今，Tengine 作为 ARM 人工智能平台的软件框架，首次发布支持 CPU/GPU 异构调度。

Tengine 的 GPU/CPU 异构调度的原理是:根据算子对计算图（Graph）进行切分，切分的子图（SubGraph）再通过调度器分配给相应的 device。由于 GPU 的编程较复杂，我们优先支持神经网络中的常用算子(例如:CONV,POOL,FC 等)，而对于某些网络中特有的算子（例如检测网络 SSD 中的 PRIORBOX 等），我们将这些算子分配给 CPU 进行计算。

OPEN AI LAB

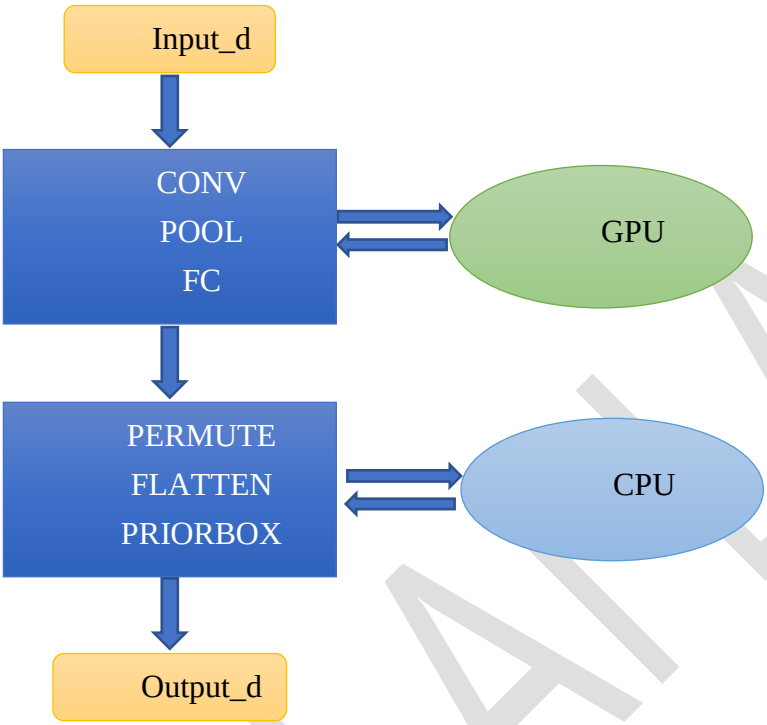


图 5.1 异构计算

下面我们将演示如何调用 Tengine 进行 GPU/CPU 异构调度,进行检测网络 MobilenetSSD 的推理加速。

我们的测试环境是 Linux, 测试平台是开发板 Firefly-RK3399：

```
GPU: Mali-T860
CPU: dual-core Cortex-A72 + quad-core Cortex-A53
```

Tengine 是通过调用 Arm Compute Library（ACL）进行 GPU 加速，我们使用的 ACL 版本为 18.05：

```
git clone https://github.com/ARM-software/ComputeLibrary.git
git checkout v18.05
scons Werror=1 -j4 debug=0 asserts=1 neon=0 opencl=1 embed_kernels=1 os=linux arch=arm64-v8a
```

为了发挥 GPU 的最高性能，我们需要设置 GPU 的频率到最高频率：

```
sudo su
echo "performance" >/sys/devices/platform/ff9a0000.gpu/devfreq/ff9a0000.gpu/governor
```

OPEN AI LAB

```
cat /sys/devices/platform/ff9a0000.gpu/devfreq/ff9a0000.gpu/cur_freq
```

显示的 GPU 频率应该是 800000000。

git clone Tengine 项目：

```
git clonehttps://github.com/OAID/Tengine.git
cp makefile.config.example makefile.config
```

在配置文件中打开开关 CONFIG_ACL_GPU=y，并指定 ACL 路径：

```
CONFIG_ACL_GPU=y
ACL_ROOT=/home/firefly/ComputeLibrary
```

编译：

```
make -j4
make install
```

下载 MobilenetSSD 模型，可以从 [Tengine model zoo](https://pan.baidu.com/s/1LXZ8vOdyOo50IXS0CUPp8g) (psw: 57vb) 下载模型~/tengine/models/ 路径下：

```
cd example/mobilenet_ssd
cmake -DTENGINE_DIR=/home/firefly/tengine .
make
```

执行时需要设置一些环境变量：

```
export GPU_CONCAT=0          # 禁用 GPU 运行 concat，避免 CPU 和 GPU 之间频繁的数据传输
export ACL_FP16=1            # 启用 GPU fp16 模式
export REPEAT_COUNT=100      # 重复运行 mssd，获取平均时间
taskset 0x1 ./MSSD -d acl_opencl   # -d acl_opencl 表示使用 GPU
```

设置 GPU_CONCAT=0，是为了避免 GPU/CPU 在 concat 这一层在 GPU/CPU 频繁来回传输数据造成性能损失；

设置 ACL_FP16=1，是支持 GPU 用 float16 的数据格式进行推理计算；

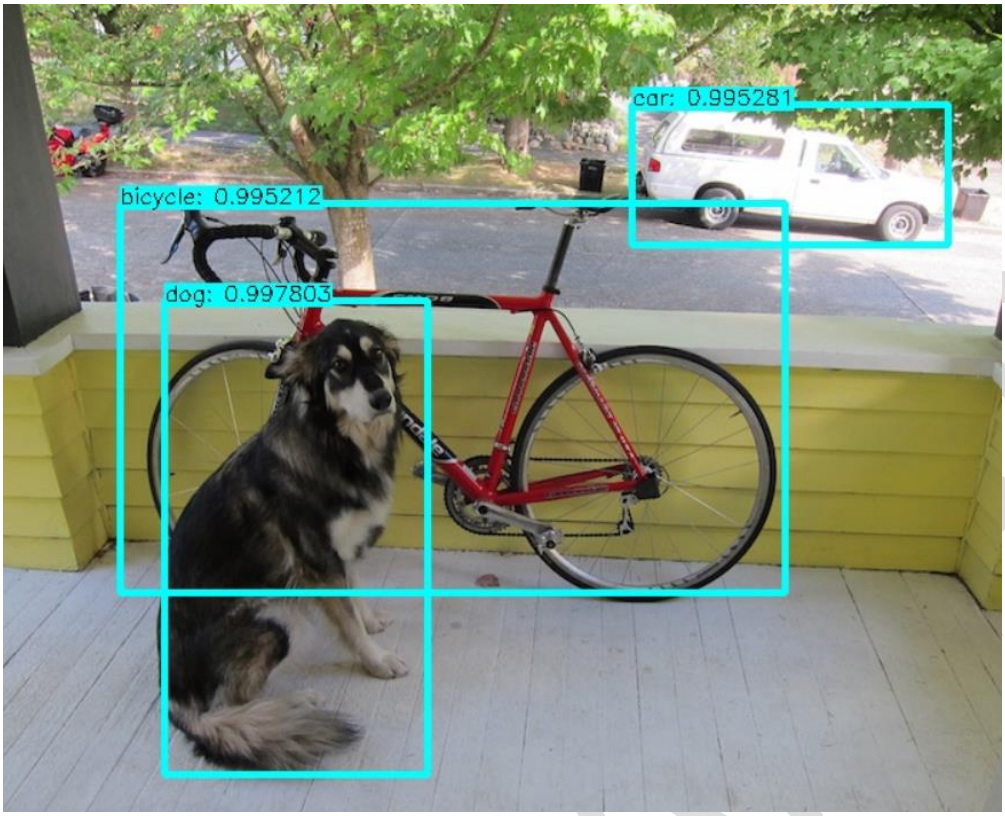
设置 REPEAT_COUNT=100，是让算法重复执行 100 次，取平均时间作为性能数据;

taskset 0x1 用于绑定 CPU0（1A53）执行程序;

执行的时候需要加 -d acl_opencl 来打开使用 gpu 的开关

从下图可以看到，GPU 用半浮点精度 float16 的检测结果是正确的。

OPEN AI LAB



5.4 转换为 Tengine 模型

按照第二章节，编译 tengine 完成后，在 tengine 根目录下执行：

```
make install
```

可以在 install/tool 下看到转换工具 convert_model_to_tm。

convert_model_to_tm 可以方便的将 Caffe、Onnx、Mxnet、 Tensorflow 或 Tf-lite 模型转换成 Tengine 模型。

命令格式为

```
Usage]: ./install/tool/convert_model_to_tm [-h] [-f file_format] [-p proto_file] [-m model_file] [-o output_tmfile]
```

file_format 可以指定：caffe、caffe_single、onnx、mxnet、tensorflow、tflite 当输入只有一个模型文件时，不需要指定“-p”选项，只指定“-m”选项即可。 “-h”是“help”选项。

下面是将 Caffe 模型和 Tensorflow 模型（squeezenet）转成 Tengine 模型的使用示例：

```
cd ~/tengine
./install/tool/convert_model_to_tm -f caffe -p models/sqz.prototxt -m models/squeezenet_v1.1.caffemodel -o models/squeezenet.tmfile
```

OPEN AI LAB

```
./install/tool/convert_model_to_tm -f tensorflow -m models/squeezenet.pb -o models/squeezenet_tf.tmfile
```

5.5 量化

Tengine 量化说明

5.5.1 量化工具的使用

你可以使用该命令运行量化工具。

[Usage]: ./install/tool/quant_tm_model [-h] [-q quant_mode] [-i input_tmfile] [-o output_tmfile]

quant_mode: 0 -- fp16, 1 -- int8

input_tmfile : the tengine FP32 mode

output_tmfile: The tengine FP16 mode or INT8 mode

5.5.2 支持 INT8 和 FP32 的混合计算

5.5.3 测试量化模型的准确率

5.6 添加自定义算子

该部分介绍如何在 Tengine 中添加新的 operator 。

要添加新的 operator，你需要：

1) 在目录 Tengine/operator/中注册新的 operator。

Operator schema 为 operator 的功能定义了一个接口，该接口独立于 operator 的实现。operator schema 需要定义：

- op 名称
- op 输入与输出
- op 参数
- infershape 函数：用于预运行(Prerun)阶段的张量形状推断。

2) 在目录 Tengine/executor/中实现 operator。

这是 operator 的具体实现。对于不同的参数（例如，conv1x1、conv3x3）或架构（例如，armv8/armv7）可以有多个实现。

OPEN AI LAB

5.6.1 以 Scale Operator 为例

在这个部分， 我们以 **Scale** operator 为例，如何一步步实现为 Tengine 添加自定义 operator。

Step 1: 创建 Operator 头文件

operators 分为两种类型：

- 没有内部参数(parameters)的 Operator： `template <typename T> OperatorNoParam`
- 有内部参数(parameters)的 Operator： `template <typename T, typename P> OperatorWithParam`

scale operator 是带有参数的运算符

- 创建 `ScaleParam` 文件 operator/include/operator/scale_param.hpp:

```
struct ScaleParam {
int   axis;
int   num_axes;
int   bias_term;

DECLARE_PARSER_STRUCTURE(ScaleParam) {
    DECLARE_PARSER_ENTRY(axis);
    DECLARE_PARSER_ENTRY(num_axes);
    DECLARE_PARSER_ENTRY(bias_term);
};
   };
```

- 创建 `Scale` 文件 operator/include/operator/scale.hpp:

```
class Scale: public OperatorWithParam<Scale,ScaleParam> {
public:

    Scale() { name_="Scale"; }
    Scale(const Scale&)= default;
    ~Scale() {}

    void SetSchema(void) override;
   };
```

Step 2: 创建 Operator Cpp 文件

在文件 operator/operator/scale.cpp 中设置 operator schema：

```
void Scale::SetSchema(void)
{
    Input({"input:float32","gamma:float32","bias:float32"})
    .Output({"output:float32"})
    .SetAttr("axis",1)
    .SetAttr("num_axes",1)
    .SetAttr("bias_term",0)
```

```
        .SetDoc(R"DOC(Scale: only caffe flavor scale)DOC");
    }
```

在文件 operator/operator/Makefile 中加入 `obj-y+=scale.o`。

Step 3: 在 Operator Plugin 中注册 Operator

在文件 operator/plugin/init.cpp 中加入对 scale operator 的注册内容：

```
    RegisterOp<Scale>("Scale");
```

Step 4: 在 Executor 文件夹中实现算法

在文件 executor/operator/common/scale.cpp 中添加 scale operator 的实现，并在函数 `RegisterScaleNodeExec` 中注册该实现：

```
namespace ScaleImpl
{
    struct ScaleOps: public NodeOps
    {
        bool Run(Node * node)
        {
            // your implementation
        }
    };
}

using namespace ScaleImpl;
void RegisterScaleNodeExec(void)
{
    ScaleOps * ops=new ScaleOps();

    NodeOpsRegistryManager::RegisterOPImplementor("common",
                "Scale",ops);
}
```

在编译文件 executor/operator/common/Makefile 中加入 `obj-y+=scale.o`。

Step 5: 在 Executor Plugin 中注册实现

在文件 executor//operator/plugin/init.cpp 中加入 `RegisterScale_NodeExec`:

```
    extern void RegisterScale_NodeExec(void);
    ...
       RegisterScale_NodeExec();
```

5.7 Caffe Wrapper 指令

你可以找到相关 demo 并按照 examples/caffe_wrapper 下的步骤进行操作：

1. 按照第 4.2 章编译 demo

注意： 在 ${TENGINE_ROOT}/examples/CMakeLists.txt 中"add_subdirectory(caffe_wrapper)" 必须被打开。

2. 运行分类 demo

- 转到执行程序目录：

```
cd ${Tengine_ROOT}/examples/build/caffe_wrapper/cpp_classification
```

- 测试一张图片：

```
./classification <deploy.prototxt> <network.caffemodel> <mean.binaryproto> <labels.txt> <img.jpg>
./classification_mobilenet <deploy.prototxt> <network.caffemodel> <mean.binaryproto> <labels.txt> <img.jpg>
```

`classification` 用来测试 squeezenet 网络。

`classification_mobilenet` 用来测试 mobilenet，并以 0.017 为标准差对图像进行预处理。

3. 运行 mtcnn demo

- 转到执行程序目录：

```
cd ${Tengine_ROOT}/examples/build/caffe_wrapper/mtcnn
```

-测试一张图片：

```
./CAFFE_MTCNN  <test.jpg>  <model_dir>  [save_result.jpg]
```

`model_dir`表示你的 mtcnn 模型路径。

5.8 Tensorflow Wrapper 指令

你可以找到 demo 并按照/examples/tensorflow_wrapper 下的步骤进行操作，

1. 按照第 4.2 章编译 demo

2. 运行程序

运行

- 转到执行程序目录

```
cd ${Tengine_ROOT}/examples/build/tensorflow_wrapper/label_image
```

- 测试一张图片：

```
./label_image_inceptionv3
./label_image_mobilenet
./label_image_resnet50
```

OPEN AI LAB

`label_image_inceptionv3`使用 inception_v3 来测试

`label_image_mobilenet` 使用 mobilenet.来测试

`label_image_resnet50`使用 resnet50.来测试

OPEN AI LAB