ONE(On-device Neural Engine)

Release 1.31.0

Samsung Research & contributors

Jul 10, 2026

CONTENTS

1 Overview **1**
- 1.1 Background . . . 1
- 1.2 Roadmap . . . 1
- 1.3 Overall Architecture . . . 3
- 1.4 Supported Operations . . . 3
- 1.5 Workgroup . . . 3

2 How To **5**
- 2.1 How to Build Compiler . . . 5
- 2.2 How to Build Package . . . 9
- 2.3 How to Build Runtime . . . 12
- 2.4 How to Build Runtime with GBS for Tizen/RPi4 . . . 17
- 2.5 How to Build Using Prebuilt Docker Image . . . 22
- 2.6 How to Cross-build Runtime for AARCH64 . . . 22
- 2.7 How to Cross-build Runtime for Android . . . 24
- 2.8 How to Cross-build Runtime for ARM . . . 25
- 2.9 How to Run Package . . . 27
- 2.10 How to Make an Application with Runtime . . . 27
- 2.11 How to Use API . . . 27
- 2.12 How to Use NNAPI Binding . . . 27
- 2.13 How to Use NNFW API . . . 28
- 2.14 How to Use Specific Backend during Inference . . . 29
- 2.15 How to Contribute . . . 30
- 2.16 How to Remote Debugging with Visual Studio Code . . . 31
- 2.17 How to Add a New Operation . . . 33
- 2.18 How To Introduce a New Operation Into Compiler . . . 33
- 2.19 How To Introduce a New Operation Into Runtime . . . 33

3 Runtime **43**
- 3.1 API . . . 43
- 3.2 Core . . . 44
- 3.3 Controlflow Operations . . . 46
- 3.4 Executors . . . 47
- 3.5 Heterogeneous Execution . . . 48
- 3.6 Backend API . . . 50
- 3.7 Compute . . . 52
- 3.8 Supported Operations and backend . . . 52

4 Compiler **53**
- 4.1 Frontend . . . 53

4.2 Middleend . . . 53
4.3 Backend . . . 53
4.4 IR . . . 53
4.5 Interpreter . . . 53
4.6 Libraries . . . 53
4.7 Tools . . . 53

5 Common IR **55**
5.1 Introduction to circle . . . 55
5.2 What is Common IR . . . 55

6 Package **57**
6.1 Introduction to Package . . . 57
6.2 Design of Package . . . 57

7 Platform **59**
7.1 Ubuntu . . . 59
7.2 Tizen . . . 59
7.3 Android . . . 59

8 Devices **61**
8.1 ODROID-XU3 . . . 61
8.2 ODROID-XU4 . . . 61
8.3 Raspberry Pi 3 . . . 61

9 Test & Benchmarks **63**
9.1 Scripts . . . 63
9.2 Benchmarks . . . 63

10 Release **65**
10.1 1.0 . . . 65
10.2 1.1 . . . 65
10.3 1.2 . . . 65
10.4 1.3 . . . 65
10.5 1.4 . . . 65
10.6 1.5 . . . 65
10.7 1.6 . . . 66
10.8 1.7 . . . 67
10.9 1.8 . . . 68
10.10 1.9 . . . 69
10.11 1.10 . . . 70
10.12 1.11 . . . 70
10.13 1.12 . . . 71
10.14 1.13 . . . 72
10.15 1.14 . . . 72
10.16 1.15 . . . 72
10.17 1.16 . . . 73

11 Indices and tables **75**

CHAPTER

ONE

OVERVIEW

1.1 Background

Artificial intelligence (AI) techniques are getting popular and utilized in various products and services. While the cloud-based AI techniques have been used to perform compute/memory intensive inferences because of the powerful servers on cloud, on-device AI technologies are recently drawing attention from the mobile industry for response time reduction, privacy protection, and connection-less AI service. Big mobile players are investing their research effort in the on-device AI technologies and have already announced hardware and software on-device AI solutions. We are not leading this trend currently, but since on-device AI area has just started and remains in the initial stage, there are still opportunities and possibilities to reduce the gap between pioneers and us. We believe that on-device AI will become a key differentiator for mobile phones, TV, and other home appliances. Therefore, developing on-device AI software stack is of paramount importance in order to take leadership in the on-device AI technology.

Although the vision of on-device AI is promising, enabling on-device AI involves unique technical challenges compared to traditional cloud-based approach. This is because on-device AI tries to conduct inference tasks solely on device without connecting to cloud resources. Specifically, hardware resources on device, such as processor performance, memory capacity and power budget are very scarce and limit the compute capability, which is typically required to execute complex neural network (NN) models. For example, in one product requirement, a mobile device should consume less than 1.2W and could use at most 2W only for 10 minutes due to thermal constraints. On-device AI software stack needs to support diverse device environments, since embedded platforms may consist of heterogeneous compute devices, such as CPU, GPU, DSP, or neural processing unit (NPU), and use different OS platforms, such as Tizen, Android, or various Linux systems.

To tackle the challenges above and to have the leadership in on-device AI technology, this project, as the first step, aims at developing a neural network inference framework specialized and optimized for on-device AI.

1.2 Roadmap

This document describes the roadmap of the **ONE** project.

Project **ONE** aims at providing a high-performance, on-device neural network (NN) inference framework that performs inference of a given NN model on processors, such as CPU, GPU, DSP, or NPU, on the target platform, such as Tizen, Android and Ubuntu.

1.2.1 Progress

Until last year, we already saw significant gains in accelerating with a single CPU or GPU backend. We have seen better performance improvements, not only when using a single backend, but even when mixing CPUs or GPUs, considering the characteristics of individual operations. It could give us an opportunity to have a high degree of freedom in terms of operator coverage and possibly provide better performance compared to single backend acceleration.

On the other hand, we introduced the compiler as a front-end. This will support a variety of deep learning frameworks in relatively spacious host PC environments, while the runtime running on the target device is intended to take a smaller

burden. In this approach, the compiler and the runtime will effectively share information among themselves by the Common IR, named *circle*, and a container format which is referred to as the *NN Package*.

1.2.2 Goal

In the meantime, we have been working on improving the acceleration performance based on the vision model. From this year, now we start working on the voice model. The runtime requirements for the voice model will be different from those of the vision model. There will be new requirements that we do not recognize yet, along with some already recognized elements such as control flow and dynamic tensor. In addition, recent studies on voice models require efficient support for specific architectures such as attention, transformer and BERT. Also, depending on the characteristics of most voice models with large memory bandwidth, we will have to put more effort into optimizing the memory bandwidth at runtime.

1.2.3 Deliverables

- Runtime
  - Control Flow support (IF/WHILE)
  - Dynamic Tensor support
  - High quality kernel development for UINT8 quantized model
  - End-to-end performance optimization for voice models
- Compiler
  - More than 100 operations support
  - Standalone *circle* interpreter
  - Completion and application of *circle2circle* pass
    - *circle-quantizer* for UINT8 and INT16
    - *circle-optimizer*
  - Graphical *circle* model viewer

1.2.4 Milestones

- 2020 Project Milestones

1.2.5 Workgroups (WGs)

- We organize WGs for major topics and each WG will be working on its own major topic by breaking it into small tasks/issues, performing them inside WG and collaborating between WGs.
- The WG information can be found here.

1.3 Overall Architecture

1.3.1 Compiler

1.3.2 Package

1.3.3 Runtime

1.4 Supported Operations

As of 2020-06-26

- TensorFlow commit e5023a1738cce7efcdf9d87863b85c80ab2f8c9e
- This commit is Tensorflow nightly build after v2.2.0 release

1.4.1 circle additional operators

1.5 Workgroup

1.5.1 Runtime WG1

1.5.2 Runtime WG2

1.5.3 Runtime WG3

1.5.4 Compiler Frontend WG

1.5.5 Compiler Backend WG

1.5.6 How to Join?

CHAPTER

TWO

HOW TO

2.1 How to Build Compiler

This document is based on the system where Ubuntu Desktop Linux 22.04 LTS is installed with default settings, and can be applied in other environments without much difference.

2.1.1 Build Requires

If you are going to build this project, the following modules must be installed on your system:

- CMake
- GNU C/C++ compiler (gcc, g++) - the recommended version is GCC 11 or higher. Please check the *known issues* section for the explanation.

In the Ubuntu, you can easily install it with the following command.

```
$ sudo apt-get install cmake gcc g++
```

If your linux system does not have the basic development configuration, you will need to install more packages. A list of all packages needed to configure the development environment can be found in the https://github.com/Samsung/ONE/blob/master/infra/docker/focal/Dockerfile file.

Here is a summary of it

```
$ sudo apt-get install \
build-essential \
cmake \
git \
libboost-all-dev \
libgflags-dev \
libgoogle-glog-dev \
libatlas-base-dev \
libhdf5-dev \
libprotobuf-dev \
protobuf-compiler \
wget \
zip \
unzip \
python3 \
python3-pip \
python3-venv \
python3-dev \
```

(continues on next page)

(continued from previous page)

```
hdf5-tools \
curl
$ pip install numpy flatbuffers
```

Supported platforms:

- Ubuntu 20.04
- Ubuntu 22.04 (recommended)
- Ubuntu 24.04 (experimental)

Supported Python versions:

- Python 3.10 for Ubuntu 20.04 and Ubuntu 22.04
- Python 3.12 for Ubuntu 24.04

  [!NOTE] Python 3.10 needs to be installed manually on Ubuntu 20.04, as it is not available in the default repositories.

  The `ppa:circletools/onepython-focal` repository is a temporary channel provided only for building this compiler.

The following steps are required only for Ubuntu 20.04:

```
sudo apt-get install software-properties-common
sudo -E add-apt-repository ppa:circletools/onepython-focal -y
sudo apt-get -y install python3.10 python3.10-venv python3.10-dev
python3.10 -m ensurepip
```

2.1.2 Build for Ubuntu

In a typical linux development environment, including Ubuntu, you can build the compiler with a simple command like this:

```
$ git clone https://github.com/Samsung/ONE.git one
$ cd one
$ ./nncc configure
$ ./nncc build
```

Build artifacts will be placed in `build` folder.

To run unit tests:

```
$ ./nncc test
```

Above steps will build all the modules in the compiler folder. There are modules that are currently not active. To build only as of now active modules of the compiler, we provide a preset of modules to build with below command:

```
$ ./nnas create-package --prefix $HOME/.local
```

With this command, `~/.local` folder will contain all files in release. If you have added `~/.local/bin` in PATH, then you will now have latest compiler binaries.

Build for debug and release separately

Build target folder can be customized by `NNCC_WORKSPACE` environment, as we may want to separate debug and release builds.

```
$ NNCC_WORKSPACE=build/debug ./nncc configure
$ NNCC_WORKSPACE=build/debug ./nncc build
```

will build debug version in `build/debug` folder, and

```
$ NNCC_WORKSPACE=build/release ./nncc configure -DCMAKE_BUILD_TYPE=Release
$ NNCC_WORKSPACE=build/release ./nncc build
```

will build release version in `build/release` folder.

2.1.3 Build for Windows

To build for Windows, we use MinGW(Minimalist GNU for Windows). Here you can download a tool that includes it.

```
$ git clone https://github.com/Samsung/ONE.git one
$ cd one
$ NNAS_BUILD_PREFIX=build ./nnas create-package --preset 20200731_windows --prefix␣
↪install
```

- `NNAS_BUILD_PREFIX` is the path to directory where compiler-build-artifacts will be stored.
- `--preset` is the one that specifies a version you will install. You can see `infra/packaging/preset/` directory for more details and getting latest version.
- `--prefix` is the install directory.

2.1.4 Cross build for Ubuntu/ARM32 (experimental)

Some modules are availble to run in Ubuntu/ARM32 through cross building.

While configuring the build, some modules need to execute tools for generating test materials and they need to execute in the host(x86-64). So some modules are needed to build the tools for host before cross building.

Cross build overall steps are like, (1) configure for host (2) build tools for host (3) configure for ARM32 target (4) and then build for ARM32 target.

Unit tests can also run in target device. But value test needs to run TensorFlow lite to get expected results, and it would be a task to do this so the data files from host execution are used instead.

Thus to run the unit tests in the target, running in host is needed in prior.

Prepare root file system

You should prepare Ubuntu/ARM32 root file system for cross compilation. Please refer how-to-cross-build-runtime-for-arm.md for preparation.

You can set `ROOTFS_ARM` environment variable if you have in alternative folder.

Clean existing external source for patches

Some external projects from source are not "cross compile ready with CMake" projects. This experimental project prepared some patches for this. Just remove the source and stamp file like below and the `make` will prepare patch applied source codes.

```
rm -rf externals/HDF5
rm -rf externals/PROTOBUF
rm externals/HDF5.stamp
rm externals/PROTOBUF.stamp
```

Build

To cross build, `infra/nncc/Makefile.arm32` file is provided as an example to work with `make` command.

```
make -f infra/nncc/Makefile.arm32 cfg
make -f infra/nncc/Makefile.arm32 debug
```

First `make` will run above steps (1), (2) and (3). Second `make` will run (4).

Test

Preprequisite for testing in ARM32 device.

```
# numpy is required for value match in ARM32 target device
sudo apt-get install python3-pip
python3 -m pip install numpy
```

You can also run unit tests in ARM32 Ubuntu device with cross build results. First you need to run the test in host to prepare files that are currently complicated in target device. For value test with python, separate venv is requried. make target `test_venv` will prepare this.

```
# run this in x86-64 host
make -f infra/nncc/Makefile.arm32 test_prep

# run this in ARM32 target device
make -f infra/nncc/Makefile.arm32 test_venv
make -f infra/nncc/Makefile.arm32 test
```

NOTE: this assumes

- host and target have same directoy structure
- should copy `build` folder to target or
- mounting `ONE` folder with NFS on the target would be simple

2.1.5 Known issues

There's a potential known build error when attempting to cross-compile for ARM32 using GCC 10.5. You might encounter an error: `comparison of unsigned expression in '< 0' is always false [-Werror=type-limits]` reported from the `CircleNodeMixins.h` file. This is likely GCC's bug in this specific version. There's a workaround for it though - you can apply the following changes to both for loops in the `CircleNodeMixins.h`:

```
FixedArityNode()
{
  if constexpr (N > 0)
  {
    _args.resize(N);
    for (uint32_t n = 0; n < N; ++n)
```

(continues on next page)

(continued from previous page)

```
    {
    _args[n] = std::make_unique<loco::Use>(this);
    }
  }
}

void drop(void) final
{
  if constexpr (N > 0)
  {
    for (uint32_t n = 0; n < N; ++n)
    {
      _args.at(n)->node(nullptr);
    }
  }
}
```

2.2 How to Build Package

2.2.1 Overview

This document describes how to build a Package to run the model in our runtime *onert* that consists of model and additional file(s). Users can build a package through command line tools.

Steps of building a Package:

1. Import model and convert to circle
2. Optionally, optimize and quantize circle
3. Create package from circle

NOTE: Examples and options of each command shown below are from the version of writing this document. They may differ from latest version of commands, 1.9.0. Please fire an issue or post a PR to correct them if anything needs update.

2.2.2 Import model

Currently TensorFlow and TensorFlow lite models are supported as of writing this document.

To import a model, use `one-import` with a model framework key and arguments.

```
$ one-import FRAMEWORK [arguments]
```

Execute `one-import` without any key will show the list of supported frameworks.

Example of `one-import` command:

```
$ one-import
Usage: one-import [FRAMEWORK] ...
Available FRAMEWORK drivers:
  bcq
  tf
  tflite
```

Example for TensorFlow

This is an example to import TensorFlow model:

```
$ one-import tf --input_path mymodel.pb --output_path mymodel.circle \
--input_arrays input1,input2 --input_shapes "1,224,224,3:1000" \
--output_arrays output
```

Running with `--help` will show current required/optional arguments:

```
$ one-import tf --help
Convert TensorFlow model to circle.
Usage: one-import-tf
    --version Show version information and exit
    --input_path <path/to/tfmodel>
    --output_path <path/to/circle>
    --input_arrays <names of the input arrays, comma-separated>
    --input_shapes <input shapes, colon-separated>
    --output_arrays <names of the output arrays, comma-separated>
    --v2 Use TensorFlow 2.x interface (default is 1.x interface)
```

Example for TensorFlow lite

This is an example to import TensorFlow lite model:

```
$ one-import tflite --input_path mymodel.tflite --output_path mymodel.circle
```

Likewise, running with `--help` will show current required/optional arguments:

```
$ one-import tflite --help
Convert TensorFlow lite model to circle.
Usage: one-import-tflite
    --version Show version information and exit
    --input_path <path/to/tflitemodel>
    --output_path <path/to/circle>
```

Example for TensorFlow Model Including BCQ Information

This is an example to import TensorFlow model which includes BCQ information. As a result of this command, BCQ information nodes will be preserved.

```
$ one-import bcq --input_path bcqmodel.pb --output_path bcqmodel.circle
```

Likewise, running with `--help` will show current required/optional arguments:

```
$ one-import bcq --help
Convert TensorFlow model with BCQ to circle.
Usage: one-import-bcq
    --version Show version information and exit
    --input_path <path/to/tfmodel/with/BCQ>
    --output_path <path/to/circle>
    --input_arrays <names of the input arrays, comma-separated>
    --input_shapes <input shapes, colon-separated>
    --output_arrays <names of the output arrays, comma-separated>
    --v2 Use TensorFlow 2.x interface (default is 1.x interface)
```

2.2.3 Optimize circle model

circle model can be optimized for better performance and smaller size. Typical optimization algorithm for this is to fuse some patterns of operators to one fused operator.

This is an example to optimize circle model:

```
$ one-optimize --all --input_path mymodel.circle --output_path optmodel.circle
```

Run with `--help` will show current optimization options:

```
$ one-optimize --help
Optimize circle model.
Usage: one-optimize
    --version       Show version information and exit
    --all           Enable all optimization algorithms
    --fuse_bcq      Enable FuseBCQ Pass
    --fuse_instnorm Enable FuseInstanceNormalization Pass
    --resolve_customop_add
                    Enable ResolveCustomOpAddPass Pass
    --resolve_customop_batchmatmul
                    Enable ResolveCustomOpBatchMatMulPass Pass
    --resolve_customop_matmul
                    Enable ResolveCustomOpMatMulPass Pass
    --input_path <path/to/input/circle>
    --output_path <path/to/output/circle>
```

2.2.4 Quantize circle model

Floating-point circle model can be quantized to lower-precision format (e.g., uint8 or int16) for faster inference speed and smaller model size, by reducing the number of bits that represent weights and activations.

This is an example to quantize circle model:

```
$ one-quantize --input_path mymodel.circle --output_path quantmodel.circle
```

Like wise, `--help` will show current quantization options:

```
$ one-quantize --help
Quantize circle model.
Usage: one-quantize
    --version         Show version information and exit
    --input_dtype     Input data type (supported: float32, default=float32)
    --quantized_dtype Output quantized data type (supported: uint8, default=uint8)
    --granularity     Quantize granularity (supported: layer, channel, default=layer)
    --min_percentile  Minimum percentile (0.0~100.0, default=1.0)
    --max_percentile  Maximum percentile (0.0~100.0, default=99.0)
    --mode            Record mode (supported: percentile/moving_average,␣
↪default=percentile)
    --input_path <path/to/input/circle>
    --input_data <path/to/input/data>
    --output_path <path/to/output/circle>
```

2.2.5 Pack circle model

Use `one-pack` command to create package.

```
$ one-pack -i mymodel.circle -o nnpackage
```

`nnpackage` is a folder containing circle model and addition file(s)

```
$ tree nnpackage
nnpackage
└── mymodel
    ├── metadata
    │   └── MANIFEST
    └── mymodel.circle
```

Likewise, `--help` will show current package options:

```
$ one-pack --help
Package circle to nnpkg
Usage: one-pack
    -v, --version Show version information and exit
    -i <path/to/circle>
    -o <path/to/nnpackage/folder>
```

2.3 How to Build Runtime

This document is based on the system where Ubuntu Desktop Linux 22.04 LTS is installed with default settings, and can be applied in other environments without much difference. For reference, the development of our project started in the Ubuntu Desktop Linux 16.04 LTS environment.

2.3.1 Build requirements

If you are going to build this project, the following modules must be installed on your system:

- C & C++ compiler
- CMake

In the Ubuntu, you can easily install it with the following command.

```
$ sudo apt-get install cmake
```

If your linux system does not have the basic development configuration, you will need to install more packages. A list of all packages needed to configure the development environment can be found in `infra/docker/<ubuntu_code>/Dockerfile` of our repository.

Here is a summary of it for runtime and related tools

```
$ sudo apt install \
build-essential \
cmake \
git \
graphviz \
lcov \
libhdf5-dev \
python3 \
```

(continues on next page)

(continued from previous page)

```
python3-pip \
python3-dev \
scons \
software-properties-common \
unzip \
wget \
curl
```

2.3.2 Get ONE source code

Getting code is as simple as

```
$ git clone https://github.com/Samsung/ONE.git one
$ cd one
```

2.3.3 Install python packages

Python package `numpy` is needed for build. It's recommended to always install python packages in virtualenv. To do this, at first you need to install python venv package:

```
$ sudo apt-get install python3-venv
```

Next, create virtualenv in `.venv` directory (make sure that you are in ONE directory):

```
$ python3 -m venv .venv
```

and activate it (please keep in mind that you will have to activate it always before build)

```
$ source .venv/bin/activate
```

Next, just in case upgrade pip package:

```
(.venv)$ pip install -U pip
```

and finally install `numpy`:

```
(.venv)$ pip install numpy
```

Additionally, package `yapf` is needed in specific version:

```
(.venv)$ pip install yapf==0.40.2
```

2.3.4 Build from source code, for Ubuntu

In a typical linux development environment, including Ubuntu, you can build the runtime with a simple command like this (make sure that you are in ONE directory):

```
$ source .venv/bin/activate
(.venv)$ ./nnfw configure
(.venv)$ ./nnfw build
(.venv)$ ./nnfw install
```

For easy build process, we provides `Makefile.template` makefile.

```
(.venv)$ make -f Makefile.template
```

To release build the runtime, add the environment variable `BUILD_TYPE=release` to the build command as follows.

```
(.venv)$ export BUILD_TYPE=release
(.venv)$ make -f Makefile.template
```

Or you can simply do something like this:

```
(.venv)$ BUILD_TYPE=release make -f Makefile.template
```

The build method described here is a `native build` in which the build environment and execution environment are same. So, this command creates a runtime binary targeting the current build architecture, probably x86_64, as the execution environment. You can find the build output in the ./Product folder as follows:

```
$ tree -L 2 ./Product
./Product
├── out -> /home/sjlee/star/one/Product/x86_64-linux.release/out
└── x86_64-linux.release
    ├── obj
    └── out

5 directories, 3 files

$ tree -L 3 ./Product/out
./Product/out
├── bin
│   ├── onert_run
│   ├── tflite_comparator
│   └── tflite_run
├── include
│   ├── nnfw
│   │   ├── NeuralNetworksEx.h
│   │   ├── NeuralNetworksExtensions.h
│   │   ├── NeuralNetworks.h
│   │   ├── nnfw_experimental.h
│   │   └── nnfw.h
│   └── onert
│       ├── backend
│       ├── compiler
│       ├── exec
│       ├── ir
│       └── util
├── lib
│   ├── libbackend_cpu.so
│   ├── libbackend_ruy.so
│   ├── libneuralnetworks.so
│   ├── libnnfw-dev.so
│   └── libonert_core.so
├── nnapi-gtest
│   ├── nnapi_gtest
│   ├── nnapi_gtest.skip
│   └── nnapi_gtest.skip.x86_64-linux.cpu
```

(continues on next page)

(continued from previous page)

```
├── test
│   ├── command
│   │   ├── nnpkg-test
│   │   ├── prepare-model
│   │   ├── unittest
│   │   └── verify-tflite
│   ├── FillFrom_runner
│   ├── list
│   │   ├── benchmark_nnpkg_model_list.txt
│   │   ├── nnpkg_test_list.armv7l-linux.acl_cl
│   │   ├── nnpkg_test_list.armv7l-linux.acl_neon
│   │   ├── nnpkg_test_list.armv7l-linux.cpu
│   │   ├── tflite_comparator.aarch64.acl_cl.list
│   │   ├── tflite_comparator.aarch64.acl_neon.list
│   │   ├── tflite_comparator.aarch64.cpu.list
│   │   ├── tflite_comparator.armv7l.acl_cl.list
│   │   ├── tflite_comparator.armv7l.acl_neon.list
│   │   ├── tflite_comparator.armv7l.cpu.list
│   │   └── tflite_comparator.x86_64.cpu.list
│   ├── models
│   │   ├── run_test.sh
│   │   └── tflite
│   ├── nnpkgs
│   │   └── FillFrom
│   └── onert-test
└── unittest
    ├── ndarray_test
    ├── nnfw_api_gtest
    ├── nnfw_api_gtest_models
    │   ├── add
    │   ├── add_invalid_manifest
    │   ├── add_no_manifest
    │   ├── if_dynamic
    │   ├── mobilenet_v1_1.0_224
    │   └── while_dynamic
    ├── nnfw_lib_misc_test
    ├── test_cker
    ├── test_onert_core
    ├── test_onert_frontend_nnapi
    └── tflite_test

26 directories, 42 files
```

Here, let's recall that the main target of our project is the arm architecture. If you have a development environment running Linux for arm on a device made of an arm CPU, such as Odroid-XU4, you will get a runtime binary that can be run on the arm architecture with the same command above. This is the simplest way to get a binary for an arm device. However, in most cases, native builds on arm devices are too impractical as they require too long. Therefore, we will create an executable binary of an architecture other than x86_64 through a `cross build`. For cross-build method for each architecture, please refer to the corresponding document in the following section, *How to cross-build runtime for different architecture*.

Run test

The simple way to check whether the build was successful is to perform inference of the NN model using the runtime. The model to be used for the test can be obtained as follows.

```
$ wget https://storage.googleapis.com/download.tensorflow.org/models/tflite/model_zoo/
↪upload_20180427/inception_v3_2018_04_27.tgz
$ tar zxvf inception_v3_2018_04_27.tgz ./inception_v3.tflite
$ ls *.tflite
inception_v3.tflite
```

The result of running the inception_v3 model using runtime is as follows. Please consider that this is a test that simply checks execution latency without considering the accuracy of the model.

```
$ ./Product/out/bin/onert_run ./inception_v3.tflite
Model Filename ./inception_v3.tflite
===================================
MODEL_LOAD   takes 1.108 ms
PREPARE      takes 0.190 ms
EXECUTE      takes 183.895 ms
- MEAN     :  183.895 ms
- MAX      :  183.895 ms
- MIN      :  183.895 ms
- GEOMEAN  :  183.895 ms
===================================
```

If you use `tflite_run` instead of `onert_run`, the model will be executed using Tensorflow lite, the basic framework for verification. From the previous build result, you can see that it is the path to the directory where `tflite_run` and `onert_run` are located.

If you come here without any problems, you have all of the basic environments for runtime development.

2.3.5 Build for Tizen

(Will be written)

2.3.6 Build using docker image

If your development system is not a linux environment like Ubuntu, but you can use docker on your system, you can build a runtime using a pre-configured docker image. Of course, you can also build a runtime using a docker image in a ubuntu environment, without setting up a complicated development environment. For more information, please refer to the following document.

- Build using prebuilt docker image

2.3.7 How to cross-build runtime for different architecture

Please refer to the following document for the build method for architecture other than x86_64, which is the basic development environment.

- Cross building for ARM
- Cross building for AARCH64
- Cross building for Android

2.4 How to Build Runtime with GBS for Tizen/RPi4

This document describes how to build runtime with GBS for Tizen AARCH64. As a real example, we'll also describe how to prepare Tizen on Raspberry Pi 4 and show you how to run our test package runner `onert_run`.

For ARM32, there would be not much difference with some changes.

Host PC is Ubuntu 18.04 but other versions or distro may work with a little adjustments.

Detailed technical informations are not described here so please read referecnce pages while you go on.

2.4.1 Setting up build environment

(1) Add Tizen build tools repo

```
$ sudo vim /etc/apt/sources.list
```

Add this at the end

```
deb [trusted=yes] http://download.tizen.org/tools/latest-release/Ubuntu_18.04/ /
```

Note: There's a slash('/') at the end.

For other versions of Ubuntu, please refer http://download.tizen.org/tools/latest-release/ lists.

(2) Update package informations and upgrade to latest

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

(3) Install GBS tools

```
$ sudo apt-get install gbs mic
```

To get more informations, please refer HERE

2.4.2 Build ONERT

(1) Set `python2` as default python

Some tools of GBS run in `python2` and won't run with `python3`. Please check `python` version and set it to 2.x.

(2) set `TIZEN_BUILD_ROOT`

You may set `GBS-ROOT` to any place you like. Ususally we use home folder.

```
$ export TIZEN_BUILD_ROOT=$HOME/GBS-ROOT/
```

Adding to `$HOME/.profile` file would be a good thing.

(3) clone ONE repo

```
git clone https://github.com/Samsung/ONE.git
```

(4) Build

```
$ cd ONE

$ gbs -c runtime/infra/gbs/gbs.conf build --include-all -A aarch64 --define 'test_build 1
↪'
```

- `-A aarch64` is to set architecture to AARCH64. Use `arm32` for ARM32 target.
- `--define 'test_build 1'` is to enable test build so that we can use `onert_run`

Now take a cup of coffee.

(5) Build result RPM packages

```
$ ls ~/GBS-ROOT/local/repos/tizen/aarch64/RPMS
nnfw-1.10.0-1.aarch64.rpm
nnfw-debuginfo-1.10.0-1.aarch64.rpm
nnfw-debugsource-1.10.0-1.aarch64.rpm
nnfw-devel-1.10.0-1.aarch64.rpm
nnfw-minimal-app-1.10.0-1.aarch64.rpm
nnfw-minimal-app-debuginfo-1.10.0-1.aarch64.rpm
nnfw-plugin-devel-1.10.0-1.aarch64.rpm
nnfw-test-1.10.0-1.aarch64.rpm
nnfw-test-debuginfo-1.10.0-1.aarch64.rpm
```

`-1.10.0-1` may differ as this document was written with under `1.10.0` development.

2.4.3 Prepare Tizen on Raspberry Pi 4

Please refer https://wiki.tizen.org/Quick_guide_for_RPI4 for detailed descriptions.

(1) Download flashing tool

```
$ wget \
https://git.tizen.org/cgit/platform/kernel/u-boot/plain/scripts/tizen/sd_fusing_rpi3.sh?
↪h=tizen \
--output-document=sd_fusing_rpi3.sh

$ chmod 755 sd_fusing_rpi3.sh
```

(2) Prepare Micro-SD memory card.

You first need to find out device name. This document will skip how to find this. Suppose it's `/dev/sdj`:

```
$ sudo ./sd_fusing_rpi3.sh -d /dev/sdj --format
```

You need to change `/dev/sdj` to your configuration.

Partition table may look like this

```
Device     Boot    Start      End  Sectors  Size Id Type
/dev/sdj1  *        8192   139263   131072   64M  e W95 FAT16 (LBA)
/dev/sdj2         139264  6430719  6291456    3G 83 Linux
/dev/sdj3        6430720  9183231  2752512  1.3G 83 Linux
/dev/sdj4        9183232 62521343 53338112 25.4G  5 Extended
/dev/sdj5        9185280 61958143 52772864 25.2G 83 Linux
/dev/sdj6       61960192 62025727    65536   32M 83 Linux
/dev/sdj7       62027776 62044159    16384    8M 83 Linux
/dev/sdj8       62046208 62111743    65536   32M 83 Linux
/dev/sdj9       62113792 62130175    16384    8M 83 Linux
/dev/sdj10      62132224 62263295   131072   64M 83 Linux
/dev/sdj11      62265344 62521343   256000  125M 83 Linux
```

(3) Download images

Please visit http://download.tizen.org/snapshots/tizen/unified/latest/images/standard/iot-boot-arm64-rpi4/ and http://download.tizen.org/snapshots/tizen/unified/latest/images/standard/iot-headed-3parts-aarch64-rpi.

Please visit `iot-boot-armv7l-rpi4` folder for ARM32 images.

Get latest file. As of writing this document, name has `20200908.3`.

```
$ wget  http://download.tizen.org/snapshots/tizen/unified/latest/images/standard/iot-
↪boot-arm64-rpi4/tizen-unified_20200908.3_iot-boot-arm64-rpi4.tar.gz

$ wget http://download.tizen.org/snapshots/tizen/unified/latest/images/standard/iot-
↪headed-3parts-aarch64-rpi/tizen-unified_20200908.3_iot-headed-3parts-aarch64-rpi.tar.gz
```

(4) Flash images to memory card

As like above, suppose memory card is at `/dev/sdj`

```
$ sudo ./sd_fusing_rpi3.sh -d /dev/sdj \
-b tizen-unified_20200908.3_iot-boot-arm64-rpi4.tar.gz \
tizen-unified_20200908.3_iot-headed-3parts-aarch64-rpi.tar.gz
```

You need to change `/dev/sdj` to your configuration and also `tizen-unified_...` file to your latest download file name.

(5) Assign IP address for `sdb` connection

Here, we provide a way to connect `sdb` tool through TCP/IP.

Below steps will modify root image and set fixed IP address.

(5-1) Mount image to host

```
$ mkdir j2
$ sudo mount /dev/sdj2 j2
```

As like above, please update `/dev/sdj2` to your configuration.

(5-2) Add a new file

```
$ vi j2/etc/systemd/system/ip.service
```

and set as like:

```
[Service]
Restart=always
RestartSec=1
User=root
ExecStart=/bin/sh -c "ifconfig eth0 192.168.x.y netmask 255.255.255.0 up"

[Install]
WantedBy=multi-user.target
```

Replace 192.168.x.y to your actual ip address.

(5-3) Add a symbolic link

```
$ sudo mkdir -p j2/etc/systemd/system/multi-user.target.wants/
$ pushd j2/etc/systemd/system/multi-user.target.wants/
$ sudo ln -s ../../system/ip.service .
$ popd
```

(5-4) Now that every thing is ready, unmount and unplug your memory card and plug into RPi4, turn on the power.

```
$ sync
$ sudo umount j2
```

2.4.4 sdb connect to Tizen/RPi4

You may need to install Tizen Studio to use `sdb` command. Please visit https://developer.tizen.org/ if you don't have this.

We assume `sdb` command is in the PATH.

(1) Connect

```
$ sdb connect 192.168.x.y
connecting to 192.168.x.y:26101 ...
connected to 192.168.x.y:26101
```

Please update `192.168.x.y` part to your actual IP address.

Check with `devices` command: you should see `rpi3` or alike.

```
$ sdb devices
List of devices attached
192.168.x.y:26101     device          rpi3
```

(2) Remount filesystem with R/W

You need to remount file system with Read/Write so that you can install packages.

```
$ sdb root on
$ sdb shell
```

Inside your Tizen/RPi4:

```
sh-3.2# mount -o rw,remount /
```

(3) Download dependent packages

In your host, maybe with another terminal, download packages from http://download.tizen.org/releases/daily/tizen/unified/latest/repos/star

```
$ wget http://download.tizen.org/releases/daily/tizen/unified/latest/repos/standard/
↪packages/aarch64/libarmcl-v21.02-17.5.aarch64.rpm

$ wget http://download.tizen.org/releases/daily/tizen/unified/latest/repos/standard/
↪packages/aarch64/libhdf5-101-1.10.1-3.85.aarch64.rpm

$ wget http://download.tizen.org/releases/daily/tizen/unified/latest/repos/standard/
↪packages/aarch64/libhdf5_cpp101-1.10.1-3.85.aarch64.rpm
```

(4) Copy to device

```
$ sdb push libarmcl-v21.02-17.5.aarch64.rpm /opt/usr/home/owner/share/tmp/
$ sdb push libhdf5-101-1.10.1-3.85.aarch64.rpm /opt/usr/home/owner/share/tmp/
$ sdb push libhdf5_cpp101-1.10.1-3.85.aarch64.rpm /opt/usr/home/owner/share/tmp/
```

And our runtime packages

```
$ cd ~/GBS-ROOT/local/repos/tizen/aarch64/RPMS
$ sdb push nnfw-1.10.0-1.aarch64.rpm /opt/usr/home/owner/share/tmp/
$ sdb push nnfw-test-1.10.0-1.aarch64.rpm /opt/usr/home/owner/share/tmp/
```

(5) Install dependent packages

Within Tizen/RPi4 shell

```
sh-3.2# cd /opt/usr/home/owner/share/tmp/

sh-3.2# rpm -i libarmcl-v21.02-17.5.aarch64.rpm
sh-3.2# rpm -i libhdf5-101-1.10.1-3.85.aarch64.rpm
sh-3.2# rpm -i libhdf5_cpp101-1.10.1-3.85.aarch64.rpm
```

There may be message like this but it seems OK:

```
/sbin/ldconfig: Cannot lstat /lib64/libhdf5.so.101.0.0: Permission denied
```

Continue install

```
sh-3.2# rpm -i nnfw-1.10.0-1.aarch64.rpm
sh-3.2# rpm -i nnfw-test-1.10.0-1.aarch64.rpm
```

Our `Product` binary folder is installed at `/opt/usr/nnfw-test`.

```
sh-3.2# cd /opt/usr/nnfw-test
sh-3.2# ls -al
total 16
drwxr-xr-x  4 root root 4096 Jan  1 09:05 .
drwxr-xr-x 14 root root 4096 Jan  1 09:05 ..
drwxr-xr-x  3 root root 4096 Jan  1 09:05 Product
drwxr-xr-x  3 root root 4096 Jan  1 09:05 infra
```

(6) Run nnpackage

Refer `how-to-build-package.md` document to produce nnpackage from a model.

Assume `mobilenet_v2_1.4_224` nnpackage is already copied to `/opt/usr/home/owner/media/models` folder with `sdb` command.

```
sh-3.2# BACKENDS="cpu" Product/out/bin/onert_run \
/opt/usr/home/owner/media/models/mobilenet_v2_1.4_224

Package Filename /opt/usr/home/owner/media/models/mobilenet_v2_1.4_224
===================================
MODEL_LOAD   takes 65.403 ms
PREPARE      takes 158.716 ms
EXECUTE      takes 373.447 ms
- MEAN     :  373.447 ms
```

(continues on next page)

(continued from previous page)

```
- MAX      :  373.447 ms
- MIN      :  373.447 ms
- GEOMEAN  :  373.447 ms
===================================
```

2.5 How to Build Using Prebuilt Docker Image

You can pull a prebuilt image from the `DockerHub` and use it to build your project.

We are supporting docker images for the following environments:

- Ubuntu 22.04 LTS (x86_64, arm64): `nnfw/one-devtools:jammy`
- Ubuntu 24.04 LTS (x86_64, arm64): `nnfw/one-devtools:noble`
  - Built arm64 runtime on Ubuntu 24.04 is not tested yet

We are supporting docker images for cross building as well:

- Ubuntu 22.04 LTS (x86_64 host, arm32/arm64 target): `nnfw/one-devtools:jammy`
- Ubuntu 24.04 LTS (x86_64 host, arm32/arm64 target): `nnfw/one-devtools:noble`
  - Built arm64 runtime on Ubuntu 24.04 is not tested yet

We are supporting docker images for android building as well:

- Ubuntu host, android target: `nnfw/one-devtools:android-sdk`

2.6 How to Cross-build Runtime for AARCH64

In **ONE**, we use `AARCH64` on build files such as Makefile, CMakeLists.txt and so on.

2.6.1 Prepare AARCH64 Ubuntu RootFS

Install required packages

```
$ sudo apt-get install qemu qemu-user-static binfmt-support debootstrap
```

Use `install_rootfs.sh` script to prepare Root File System. You should have `sudo`

```
$ sudo ./tools/cross/install_rootfs.sh aarch64
```

- supports `arm`(default) and `aarch64` architecutre for now
- supports `jammy`, and `noble` release

To see the options,

```
$ ./tools/cross/install_rootfs.sh -h
```

RootFS will be prepared at `tools/cross/rootfs/aarch64` folder.

**** CAUTION: The OS version of rootfs must match the OS version of execution target device. On the other hand, you need to match the Ubuntu version of the development PC with the Ubuntu version of rootfs to be used for cross-build. Otherwise, unexpected build errors may occur.***

If you are using Ubuntu 22.04 LTS, select `jammy`, if you are using Ubuntu 24.04 LTS, select `noble`. You can check your Ubuntu code name in the following way.

```
$ cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=22.04
DISTRIB_CODENAME=jammy
DISTRIB_DESCRIPTION="Ubuntu 22.04.3 LTS"
```

`install_rootfs.sh` will select development system code name as default.

If a build error occurs because the version of the development system and the target system do not match, and if you can't replace your development system for any reason, you can consider cross-build using the docker image.

Prepare RootFS at alternative folder

Use `ROOTFS_DIR` to a full path to prepare at alternative path.

```
$ ROOTFS_DIR=/home/user/rootfs/aarch64-bionic sudo -E ./tools/cross/install_rootfs.sh␣
↪aarch64
```

Using proxy

If you need to use proxy server while building the rootfs, use `--setproxy` option.

```
# for example,
$ sudo ./tools/cross/install_rootfs.sh aarch64 --setproxy="1.2.3.4:8080"
# or
$ sudo ./tools/cross/install_rootfs.sh aarch64 --setproxy="proxy.server.com:8888"
```

This will put `apt` proxy settings in `rootfs/etc/apt/apt.conf.d/90proxy` file for `http`, `https` and `ftp` protocol.

2.6.2 Cross build for AARCH64

Install cross compilers

```
$ sudo apt-get install gcc-aarch64-linux-gnu g++-aarch64-linux-gnu
```

Give `TARGET_ARCH` variable to set the target architecture

```
$ CROSS_BUILD=1 TARGET_ARCH=aarch64 make
$ CROSS_BUILD=1 TARGET_ARCH=aarch64 make install
```

- supports `armv7l` and `aarch64` for now

If you used `ROOTFS_DIR` to prepare in alternative folder, you should also give this to makefile template.

```
$ CROSS_BUILD=1 ROOTFS_DIR=/home/user/rootfs/aarch64-jammy TARGET_ARCH=aarch64 make -f␣
↪Makefile.template
```

Run test

To run and test the cross-compiled runtime, you need to copy the compiled output to the target device of the architecture in which it is executable. Please refer to the following document for details on the test procedure. In the guide, `armv7l-linux.debug` in path should be replaced by referring to your build result.

- Testing neural network model inference with a cross-build runtime

2.7 How to Cross-build Runtime for Android

Note: To set up a basic build environment on your development PC, please read the how-to-build-runtime.md document first. The cross build covered in this document assumes that you have an environment in which the native build operates normally without problems.

Supported Architecture : AARCH64 only

2.7.1 Prepare Android NDK

There are two ways to get Android NDK: install Android NDK on your PC or use docker image which already has Android NDK installed.

Install Android NDK

You can use `tools/cross/install_android_sdk.sh` script to prepare Android SDK including NDK.

```
$ tools/cross/install_android_sdk.sh --install-dir /path/to/android-sdk
```

If `--install-dir` is not specified, Android SDK will be installed in `tools/cross/android_sdk`.

You can find NDK in `{android-sdk-dir}/ndk/{ndk-version}` directory.

You can check default install NDK version by running `tools/cross/install_android_sdk.sh -h` command.

Installed NDK version can be changed by specifying `--ndk-version` option like below:

```
$ tools/cross/install_android_sdk.sh --ndk-version {ndk-version} --install-dir /path/to/
↪android-sdk
```

If you want to install NDK version in existing android sdk directory, you can use `--update` option like below:

```
$ tools/cross/install_android_sdk.sh --update --install-dir /path/to/android-sdk
```

CMake 3.6.0 or later is required for Android NDK r27d CMake support.

Host Environment Requirements

If you want to use docker image, please use `infra/docker/android-sdk/Dockerfile` of our repository or pull the image `nnfw/one-devtools:android-sdk` from `DockerHub`. It includes Android SDK and NDK.

You can build the docker image with following command:

```
$ docker build -t nnfw/one-devtools:android-sdk -f infra/docker/android-sdk/Dockerfile .
```

2.7.2 Build

Build and install the runtime

Some tools/libs are still not supported and those are not built by default - mostly due to dependency on HDF5 library.

Different from cross build for linux,

- Wrapper toolchain file for Android NDK: `runtime/infra/cmake/buildtool/cross/toolchain_aarch64-android.cmake`
- `ANDROID_NDK` environment variable or `CMAKE_ANDROID_NDK` cmake variable is required

If you are using docker image `nnfw/one-devtools:android-sdk`, you don't need to specify above variable because environment variable is already set in the image.

Here is an example of using Makefile.

```
TARGET_OS=android \
CROSS_BUILD=1 \
ANDROID_NDK=/path/android-sdk/ndk/{ndk-version}/ \
make -f Makefile.template install
```

Otherwise, you can use command

```
$ source .venv/bin/activate
(.venv)$ ./nnfw configure \
  -DCMAKE_TOOLCHAIN_FILE=infra/cmake/buildtool/cross/toolchain_aarch64-android.cmake \
  -DCMAKE_ANDROID_NDK=/path/android-sdk/ndk/{ndk-version}
(.venv)$ ./nnfw build
(.venv)$ ./nnfw install --prefix build/onert/out
```

2.8 How to Cross-build Runtime for ARM

2.8.1 Prepare ARM Ubuntu RootFS

Install required packages

```
$ sudo apt-get install qemu-user-static binfmt-support debootstrap
```

Use `install_rootfs.sh` script to prepare Root File System. You should have `sudo`

```
$ sudo ./tools/cross/install_rootfs.sh arm
```

- supports `arm`(default) and `aarch64` architecutre for now
- supports `jammy`, and `noble` release

To see the options,

```
$ ./tools/cross/install_rootfs.sh -h
```

RootFS will be prepared at `tools/cross/rootfs/arm` or `tools/cross/rootfs/aarch64` folder.

**** CAUTION: The OS version of rootfs must match the OS version of execution target device. On the other hand, you need to match the Ubuntu version of the development PC with the Ubuntu version of rootfs to be used for cross-build. Otherwise, unexpected build errors may occur.***

If you are using Ubuntu 22.04 LTS, select `jammy`, if you are using Ubuntu 24.04 LTS, select `noble`. You can check your Ubuntu code name in the following way.

```
$ cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=22.04
DISTRIB_CODENAME=jammy
DISTRIB_DESCRIPTION="Ubuntu 22.04.3 LTS"
```

`install_rootfs.sh` will select development system code name as default.

If a build error occurs because the version of the development system and the target system do not match, and if you can't replace your development system for any reason, you can consider cross-build using the docker image.

Prepare RootFS at alternative folder

Use ROOTFS_DIR to a full path to prepare at alternative path.

```
$ ROOTFS_DIR=/home/user/rootfs/arm-jammy sudo -E ./tools/cross/install_rootfs.sh arm
```

Using proxy

If you need to use proxy server while building the rootfs, use --setproxy option.

```
# for example,
$ sudo ./tools/cross/install_rootfs.sh arm --setproxy="1.2.3.4:8080"
# or
$ sudo ./tools/cross/install_rootfs.sh arm --setproxy="proxy.server.com:8888"
```

This will put apt proxy settings in rootfs/etc/apt/apt.conf.d/90proxy file for http, https and ftp protocol.

2.8.2 Install ARM Cross Toolchain

We recommend you have g++ >= 9 installed on your system for c++17.

2.8.3 Build and install ARM Compute Library

Mostly you only need once of ACL (ARM Compute Library) build.

To build ACL, you need to install scons

```
$ sudo apt-get install scons
```

ACL source will be automatically installed in externals/ARMCOMPUTE when you build runtime without any changes.

You can check ACL source information in infra/cmake/packages/ARMComputeSourceConfig.cmake

2.8.4 Install numpy

Python package numpy is needed for build. Please use this guide to install it.

2.8.5 Cross build for ARM by using Makefile.template

Give TARGET_ARCH variable to set the target architecture.

If you used ROOTFS_DIR to prepare in alternative folder, you should also give this to makefile.

Remember to activate venv if you didn't activate it already:

```
$ source .venv/bin/activate
```

```
(.venv)$ CROSS_BUILD=1 TARGET_ARCH=armv7l make -f Makefile.template

# If ROOTFS_DIR is in alternative folder
(.venv)$ ROOTFS_DIR=/path/to/your/rootfs/arm \
CROSS_BUILD=1 TARGET_ARCH=armv7l make -f Makefile.template
```

Makefile.template will pass crossbuild toolchain setting to cmake automatically by parsing variables.

Run test

To run and test the cross-compiled runtime, you need to install library packages and copy the compiled output to the target device of the architecture in which it is executable.

1. Install hdf5 library package

```
$ sudo apt install libhdf5-dev
```

1. Copy all artifacts under the `./Product/armv7l-linux.<BUILD_TYPE>` folder to the target device, Odroid-XU4 for example, as a whole.

```
$ ssh odroid mkdir -p one/Product
sjlee@odroid's password:
$ scp -rp ./Product/armv7l-linux.debug odroid:one/Product
sjlee@odroid's password:
FillFrom_runner                                                                 ␣
↪       100%  224KB 223.6KB/s   00:00
benchmark_nnapi.sh                                                              ␣
↪       100% 7464     7.3KB/s   00:00
common.sh                                                                       ␣
↪       100% 2084     2.0KB/s   00:00
test_framework.sh                                                               ␣
↪       100% 3154     3.1KB/s   00:00
test-driver.sh
...
```

1. Log in to the target device, go to the copied path, and reestore the symbolic link settings of the `Product` directory.

```
$ ssh odroid
sjlee@odroid's password:
...
$ cd ~/one/Product
$ ln ${PWD}/armv7l-linux.debug/out out
$ cd ..
$ ls -la Product
drwxrwxr-x  5 sjlee sjlee 4096 Jun  4 20:55 armv7l-linux.debug
lrwxrwxrwx  1 sjlee sjlee   51 Jun  4 20:55 out -> /home/sjlee/one/Product/armv7l-linux.
↪debug/out
```

Now you can test the compilation result in the same way as the native build. Please refer to the following document for details on the test procedure.

- Testing neural network model inference using the runtime

2.9 How to Run Package

2.10 How to Make an Application with Runtime

2.11 How to Use API

2.12 How to Use NNAPI Binding

Runtime supports Android Neural Networks API as a frontend. We provide the whole `NeuralNetworks.h` implementation. Its source code is located at `runtime/onert/api/nnapi`.

So users can just use NN API in the same way as the official guide, but should watch the version of `NeuralNetworks.h`. It may not be the latest. There is a copy of the file is located at `runtime/nnapi-header/include/NeuralNetworks.h`.

2.13 How to Use NNFW API

2.13.1 Prepare nnpackage

Convert tensorflow pb file to nnpackage

Follow the compiler guide to generate nnpackge from tensorflow pb file

Convert tflite file to nnpackage

Please see model2nnpkg for converting from tflite model file.

2.13.2 Build app with NNFW API

Here are basic steps to build app with NNFW C API

1. Initialize nnfw_session

```
nnfw_session *session = nullptr;
nnfw_create_session(&session);
```

1. Load nnpackage

```
nnfw_load_model_from_file(session, nnpackage_path);
```

1. (Optional) Assign a specific backend

```
  // Use 'acl_neon' backend only
  // Note that defalut backend is 'cpu'.
  nnfw_set_available_backends(session, "acl_neon");
```

1. Compilation

```
  // Compile model
  nnfw_prepare(session);
```

1. Prepare Input/Output

```
  // Prepare input. Here we just allocate dummy input arrays.
  std::vector<float> input;
  nnfw_tensorinfo ti;
  nnfw_input_tensorinfo(session, 0, &ti); // get first input's info
  uint32_t input_elements = num_elems(&ti);
  input.resize(input_elements);
  // TODO: Please add initialization for your input.
  nnfw_set_input(session, 0, ti.dtype, input.data(), sizeof(float) * input_elements);

  // Prepare output
  std::vector<float> output;
  nnfw_output_tensorinfo(session, 0, &ti); // get first output's info
  uint32_t output_elements = num_elems(&ti);
```

(continues on next page)

(continued from previous page)

```
  output.resize(output_elements);
  nnfw_set_output(session, 0, ti.dtype, output.data(), sizeof(float) * output_elements);
```

1. Inference

```
  // Do inference
  nnfw_run(session);
```

2.13.3 Run Inference with app on the target devices

reference app : minimal app

```
$ ./minimal path_to_nnpackage_directory
```

2.14 How to Use Specific Backend during Inference

ONE runtime has many ways to use specific backend during inference

2.14.1 Using NNFW API

nnfw_set_available_backends

- Multiple backends can be set and they must be separated by a semicolon (ex: "acl_cl;cpu").
- For each backend string, `libbackend_{backend}.so` will be dynamically loaded during nnfw_prepare.
- Among the multiple backends, the 1st element is used as the default backend.

2.14.2 Using Environment Variable

1. BACKENDS

- Same as `nnfw_set_available_backends`
- Example

```
BACKENDS=cpu ./Product/out/bin/onert_run ...
```

2. OP_BACKEND_[OP_TYPE]

- Set backend for specific operator type
- Example
  - Execute `Conv2D` operator on ruy backend and others on cpu backend

```
OP_BACKEND_Conv2D=ruy BACKENDS="cpu;ruy" ./Product/out/bin/onert_run ...
```

3. OP_BACKEND_MAP

- Set backend for specific operator by its index
- Format : `<op_id>=<backend>;<op_id>=<backend>...`
- Example
  - Execute `operator 10` on `acl_cl` backend and others on `acl_neon` backend

```
OP_BACKEND_MAP="10=acl_cl" BACKENDS="acl_neon;acl_cl" ./Product/out/bin/onert_run ...
```

2.15 How to Contribute

ONE always welcomes your contribution, but there are basic guidelines that you should follow in order to make your contribution be accepted.

This document explains such guidelines for beginners.

2.15.1 General contribution guidelines

If you are not familiar with git or github, please visit here for basic understanding of git and github.

2.15.2 How to create a Pull Request

This section explains the steps to create a pull request (PR).

1. Create an issue

   Maintainers will accept your contribution only when it is well aligned with the *roadmap* and design principles of **ONE**. So, it is optional, but recommended for contributors to create an issue and have a discussion with maintainers before writing code.

2. Create a draft PR

   Maintainers will accept your pull request only when it is **reasonably small** and **focused**. Sometimes, your contribution may require huge and loosely-coupled changes. You **should** split your contribution into multiple small, but focused pull requests in this case. Unfortunately, it is possible that maintainers reject your pull request as it is hard for them to understand the intuition behind these changes. So, it is optional, but recommended for contributors to present the full draft of your contribution and have a discussion with maintainers before creating PR(s).

3. Create a commit

   It is time to create a commit for submission once you are convinced that your contribution is ready to go. Please include signed-off message at the end of commit message. If not, your pull request will be **rejected** by CI.

4. Check code format locally

   ONE has its code formatting rules, and any pull request that violates these rules will be **rejected** by CI. So, it is optional, but recommended for contributor to check code format locally before submission. Please refer this guide for details to run format checker locally.

5. Create a PR

   It is time to send a pull request. Please explain your intention via description. Maintainers will review your pull request based on that description. Each pull request needs approval from at least two reviewers to be accepted. Note that **description should include at least four words**. If not, your pull request will be **rejected** by CI.

6. Request review

   It is recommended to assign reviewers yourself. Maintainers will honor your review request, and accept your pull request only when

   - Approved by 1+ reviewers
   - 0 rejection(Request Changes)
   - 0 pending review request
   - All the reviewers in the list must approve your pull request

You can add/remove pending review requests in the middle of the review process. Maintainers (or reviewers) could review your pull request even without explicit review request.

7. Update per feedback

   Sometimes, maintainers (or reviewers) will request changes on your pull request. Please update your pull request upon such feedbacks. These update commits will be squashed into the first commit of your pull request later. Please do **NOT** include a sign-off message or write a full description for update commits.

2.16 How to Remote Debugging with Visual Studio Code

This document describes how to debug ONE runtime on arm devices using visual studio code.

2.16.1 Setup build host

Install gdb-multiarch

1. Install `gdb-multiarch`

```
$ sudo apt install gdb-multiarch
```

Configure VS code

1. Install Native Debug extension on VS code
2. Setup GDB environment on VS code

- Debug -> Add configuration -> GDB: Connect to gdbserver
- Change configuration as below
  - Change <TARGET_IP> to IP of your target
  - The default port number for gdbserver is 2345. You can change this number.
  - You can change `executable` configuration from `tflite_run` to other binaries you want to debug.

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "gdb",
            "request": "attach",
            "name": "Attach to gdbserver",
            "gdbpath": "/usr/bin/gdb-multiarch",
            "executable": "./Product/armv7l-linux.debug/out/bin/tflite_run",
            "target": "<TARGET_IP>:2345",
            "remote": true,
            "printCalls": true,
            "cwd": "${workspaceRoot}",
            "valuesFormatting": "parseText"
        }
    ]
}
```

2.16.2 Setup target device

Install gdbserver and debugging symbols

You need to setup a target device for remote debugging.

1. Install `gdbserver`

```
$ sudo apt install gdbserver
```

1. Install `libc6-dbg` and copy debugging symbols

```
$ sudo apt install libc6-dbg
$ sudo mkdir -p /lib/.debug
$ sudo ln -s /usr/lib/debug/lib/arm-linux-gnueabihf/ld-2.27.so /lib/.debug
```

2.16.3 Run remote debugging

1. Start gdbserver on target

```
gdbserver --multi :<PORT> <BINARY_PATH> <EXECUTION_ARGUMENTS>
```

Example

```
gdbserver --multi :2345 Product/armv7l-linux.debug/out/bin/tflite_run ../models/slice_
↪test.tflite
```

1. Connect to gdbserver using VS code
   - Setup breakpoints on any code you want.
   - Click F5 to start remote debugging.
   - Program will execute and exit if no breakpoint exists.

2.16.4 Optional: Setup rootfs on build host

When debugging starts, `gdb` downloads shared libraries that one runtime uses from the target device. This process makes `gdb` to wait for shared library download to finish for every debugging start.

To reduce shared library loading, you can setup an arm root file system on your build host and use it.

1. Create arm root file system

Following CrossBuildForArm to create an arm root file system.

You can use an arm root file system created for arm cross-compile.

1. Install `libc6-dbg` on arm root file system

<ROOTF_DIR> should point ARM root file system.

Default path is `tools/cross/rootfs/arm` folder.

```
$ sudo chroot <ROOTFS_DIR>
$ apt install libc6-dbg
$ exit
```

1. Create symbolic link of one runtime on arm rootfs

`gdb` will use source code folder at sysroot.

```
$ ln -s <ONE_DIR> <ROOTFS_DIR>/<ONE_DIR>
```

Example

```
$ ln -s /home/user/one /home/user/one/tools/cross/rootfs/arm/home/user/one/
```

1. Setup `.gdbinit` file on one folder

gdb will use <ROOTFS_DIR> to find arm related symbols.

```
set sysroot <ROOTFS_DIR>
set debug-file-directory <ROOTFS_DIR>/usr/lib/debug
```

2.16.5 Troubleshooting

Unable to open 'unordered_map.h'

If you are using docker to build one runtime, you should download and decompress gcc-linaro at `/opt` folder

```
wget https://releases.linaro.org/components/toolchain/binaries/6.3-2017.02/arm-linux-
↪gnueabihf/gcc-linaro-6.3.1-2017.02-x86_64_arm-linux-gnueabihf.tar.xz -O gcc-hardfp.tar.
↪xz
sudo tar -xf gcc-hardfp.tar.xz -C /opt/ && sudo rm -rf gcc-hardfp.tar.xz
```

Skip STL files

Step into (F11) will debug STL files such as `unordered_map` or `vector`.

To skip those files from debugging, you can add below line to `.gdbinit` file.

This function is supported on gdb versions >= 7.12.

```
skip -gfile /opt/gcc-linaro-6.3.1-2017.02-x86_64_arm-linux-gnueabihf/arm-linux-gnueabihf/
↪include/c++/6.3.1/bits/*
```

2.17 How to Add a New Operation

2.17.1 Compiler

- How to introduce a new operation into compiler

2.17.2 Runtime

- How to introduce a new operation into runtime

2.18 How To Introduce a New Operation Into Compiler

2.19 How To Introduce a New Operation Into Runtime

ONE's runtime has three main modules: **core**, **frontend** and **backend**. This document provides some lightweight guidance about how to introduce a new operation into these modules to make onert support the operation.

2.19.1 Index

- *How To Introduce a New Operation Into Runtime*
  - *Index*
  - *Core*
  - *Frontend*
    - *Loaders*
      - *Base Loader*
      - *TFLite Loader*
      - *Circle Loader*
    - *NNAPI*
  - *Backend*
    - *ShapeFixer*
      - *acl_cl*
      - *acl_neon*
      - *cpu*
    - *KernelGenerator*
      - *acl_cl*
      - *acl_neon*
      - *cpu*
    - *ConstantInitializer (in some cases)*
      - *cpu*
  - *Samples (to be updated)*

2.19.2 Core

This module has graph-based IR(intermediate representation). You have to add IR for the new operation.

1. Add name of new operation at Operations.lst

```
OP(Select)
```

1. Create a class for node of new operation in here

```
#include "ir/Operation.h"

namespace onert
{
namespace ir
{
namespace operation
{

class Select : public Operation
{
```

(continues on next page)

(continued from previous page)

```
public:
  enum Input
  {
    COND = 0,
    INPUT1 = 1,
    INPUT2 = 2
  };

  enum Output
  {
    OUTPUT = 0,
  };

public:
  Select(const OperandIndexSequence &inputs, const OperandIndexSequence &outputs);

public:
  void accept(OperationVisitor &v) const override;
  OpCode opcode() const final { return OpCode::Select; }
};

} // namespace operation
} // namespace ir
} // namespace onert
```

You can also define the class in other source file like below

```
#include "ir/operation/Select.h"

#include "ir/OperationVisitor.h"

namespace onert
{
namespace ir
{
namespace operation
{

void Select::accept(OperationVisitor &v) const { v.visit(*this); }

Select::Select(const OperandIndexSequence &inputs, const OperandIndexSequence &outputs)
    : Operation{OperandConstraint::createExact(3u), inputs, outputs}
{
}
```

- Operations.Include.h

```
#include "ir/operation/Select.h"
```

1. Add to the OperationValidator to check if the node is valid.

- OperationValidator.h

```
void visit(const operation::Select &node) override;
```

- OperationValidator.cc

```
void OperationValidator::visit(const ir::operation::Select &node)
{
  const auto output_index{node.getOutputs().at(ir::operation::Select::Output::OUTPUT)};
  const auto cond_index{node.getInputs().at(ir::operation::Select::Input::COND)};
  const auto input1_index{node.getInputs().at(ir::operation::Select::Input::INPUT1)};
  const auto input2_index{node.getInputs().at(ir::operation::Select::Input::INPUT2)};

  UNUSED_RELEASE(output_index);
  UNUSED_RELEASE(cond_index);
  UNUSED_RELEASE(input1_index);
  UNUSED_RELEASE(input2_index);

  const auto output_type = _ctx.at(output_index).typeInfo();
  const auto cond_type = _ctx.at(cond_index).typeInfo();
  const auto input1_type = _ctx.at(input1_index).typeInfo();
  const auto input2_type = _ctx.at(input2_index).typeInfo();

  UNUSED_RELEASE(output_type);
  UNUSED_RELEASE(cond_type);
  UNUSED_RELEASE(input1_type);
  UNUSED_RELEASE(input2_type);

  assert(cond_type.type() == ir::DataType::BOOL8);
  assert(output_type.type() == ir::DataType::FLOAT32 || output_type.type() ==␣
↪ir::DataType::INT32 ||
         output_type.type() == ir::DataType::QUANT8_ASYMM);
  assert(output_type.type() == input1_type.type());
  assert(output_type.type() == input2_type.type());

  const auto output_shape = _ctx.at(output_index).shape();
  const auto cond_shape = _ctx.at(cond_index).shape();
  const auto input1_shape = _ctx.at(input1_index).shape();
  const auto input2_shape = _ctx.at(input2_index).shape();

  UNUSED_RELEASE(output_shape);
  UNUSED_RELEASE(cond_shape);
  UNUSED_RELEASE(input1_shape);
  UNUSED_RELEASE(input2_shape);

  assert(output_shape == input1_shape);
  assert(cond_shape == input1_shape);
  assert(input2_shape == input1_shape);
}
```

1. Add to the Dumper to dump IR information of new operation.

- Dumper.cc

```
void Dumper::visit(const Select &node)
{
```

(continues on next page)

(continued from previous page)

```
  VERBOSE(LIR) << "* Select" << std::endl;
  VERBOSE(LIR) << "  - Inputs : Cond(" << node.getInputs().at(Select::Input::COND).
↪value()
               << ") Input1" << node.getInputs().at(Select::Input::INPUT1).value() << ")␣
↪Input2"
               << node.getInputs().at(Select::Input::INPUT2).value() << ")" << std::endl;
  VERBOSE(LIR) << "  - Output : Output(" << node.getOutputs().at(Select::Output::OUTPUT).
↪value()
               << ")" << std::endl;
}
```

1. Add code for shape inference

- ONE runtime tries to calculate shapes and allocate memory during compilation time. For some calculations of output shapes that cannot be done during compilation time, ONE runtime will calculate shapes and allocate memory during execution time.
- Calculation of shapes during compilation time is called *static shape inference* and calculation of shapes during execution time is called *dynamic shape inference*.
- *StaticShapeInferer.h*

```
  void visit(const ir::operation::Select &op) override;
```

- StaticShapeInferer.cc

```
void StaticShapeInferer::visit(const ir::operation::Select &op)
{
  const auto input_cond_idx{op.getInputs().at(ir::operation::Select::Input::CONDITION)};
  const auto &input_cond = _operands.at(input_cond_idx);

  const auto &input_true = ...
  const auto &input_false = ...
  ir::Operand &output = ...

  // Select output shpae
  ir::Shape new_shape = shape_inference::inferSelectShape(
      input_cond.info().shape(), input_true.info().shape(), input_false.info().shape());
  output.info().shape(new_shape);
}
```

- DynamicShapeInference.h

```
  void visit(const ir::operation::Select &op) override;
```

- DynamicShapeInference.cc

```
void DynamicShapeInferer::visit(const ir::operation::Select &op)
{
  const auto input_cond_idx = op.getInputs().at(ir::operation::Select::Input::CONDITION);
  const auto &input_cond = _tensor_registry->getITensor(input_cond_idx);

  const auto &input_true = ...
  const auto &input_false = ...
```

(continues on next page)

(continued from previous page)

```
  auto output = ...

  if ((!input_cond->is_dynamic()) && (!input_true->is_dynamic()) && (!input_false->is_
↪dynamic()))
  {
    return;
  }

  auto input_cond_shape = input_cond->getShape();
  auto input_true_shape = input_true->getShape();
  auto input_false_shape = input_false->getShape();

  // Select output shpae
  ir::Shape new_shape =
      shape_inference::inferSelectShape(input_cond_shape, input_true_shape, input_false_
↪shape);

  output->applyShape(new_shape);
}
```

2.19.3 Frontend

This module generates IR from a model. There are two kinds of frontend: Loader and NNAPI. First, Loader loads a model file and generates IR from it. Second, NNAPI generates IR from a model set via Neural Networks API of android

Loaders

Base Loader

This is where the common parts of loaders are implemented.

1. Add to base_loader to load new operation and to generate IR from it

- base_loader

```
    case BuiltinOperator::BuiltinOperator_SELECT:
      loadSelect(op);
      return;
```

```
template <typename LoaderDomain, typename SpecificLoader>
void BaseLoader<LoaderDomain, SpecificLoader>::loadSelect(const Operator *op)
{
  ir::OperandIndexSequence inputs;
  ir::OperandIndexSequence outputs;

  loadOperationIO(op, inputs, outputs);

  std::unique_ptr<ir::Operation> new_op{new ir::operation::Select{inputs, outputs}};
  _graph.addOperation(std::move(new_op));
}
```

TFLite Loader

This loads a tflite file. If you want new operation to be loaded on only TFLite Loader, you only need to implement loading the operation here.

Circle Loader

This loads a circle file generated by the compiler. If you want new operation to be loaded on only Circle Loader, you only need to implement loading the operation here.

NNAPI

1. Add to the OperationFactory to generate IR of new operation

- OperationFactory

```
  _map[ANEURALNETWORKS_SELECT] = [](const OperationFactory::Param &init_param, Operands &
↪) {
    assert(init_param.input_count == 3 && init_param.output_count == 1);

    OperandIndexSequence outputs{init_param.outputs[0]};

    // Each input should be interpreted as follows:
    //
    //  0 -> Cond Tensor Index
    //  1 -> Input1 Tensor Index
    //  2 -> Input2 Tensor Index
    OperandIndexSequence inputs;
    for (uint32_t n = 0; n < init_param.input_count; ++n)
    {
      inputs.append(OperandIndex{init_param.inputs[n]});
    }

    return new operation::Select{inputs, outputs};
  };
```

1. If you want that NNAPI supports new operation of TFLite's model, you need to update the things related to the operation in nnapi_delegate like below

```
      case tflite::BuiltinOperator_SELECT:
        nnapi_version = 12;  // require NNAPI 1.2
        nn_op_type = ANEURALNETWORKS_SELECT;
        break;
```

2.19.4 Backend

This module generates kernels and tensors of backend such as ComputeLibrary from generated graph-based IR. For this, the runtime fairly works on it internally. But this is not enough because of dependence on backend. So, there are several components that require additional implementation on each backend.

ShapeFixer

Even for tensors of the same operation, the shape required for each backend can be different. Therefore, this component modifies and fixes shape of tensors of the backend.

acl_cl

The kernel of the ACL for the Add operation needs to match the same rank to support the broadcast.

- ShapeFixer.h

```
void visit(const ir::operation::Add &) override;
```

- ShapeFixer.cc

```
void ShapeFixer::visit(const ir::operation::Add &node)
{
  const auto lhs_index{node.getInputs().at(ir::operation::Add::Input::LHS)};
  const auto rhs_index{node.getInputs().at(ir::operation::Add::Input::RHS)};

  if (!(_ctx.at(lhs_index).shape() == _ctx.at(rhs_index).shape()))
  {
    const auto broadcast_rank =
        std::max(_ctx.at(lhs_index).shape().rank(), _ctx.at(rhs_index).shape().rank());
    const_cast<ir::Shape &>(_ctx.at(lhs_index).shape()).extendRank(broadcast_rank);
    const_cast<ir::Shape &>(_ctx.at(rhs_index).shape()).extendRank(broadcast_rank);
  }
}
```

acl_neon

Same implementation as acl_cl is required.

cpu

This backend doesn't usually require a change of shape.

- ShapeFixer.h

```
void visit(const ir::operation::Select &) override;
```

- ShapeFixer.cc

```
void ShapeFixer::visit(const ir::operation::Select &) { /* DO NOTHING */}
```

KernelGenerator

This component generates kernels of backend. You have to generate kernel of new operation. And then append it to execution builder. You can obtain information of the node from IR and necessary tensors from tensor builder.

acl_cl

- KernelGenerator.h

```
void visit(const ir::operation::Select &) override;
```

- KernelGenerator.cc

```
void KernelGenerator::visit(const ir::operation::Select &node)
{
  const auto output_index{node.getOutputs().at(ir::operation::Select::Output::OUTPUT)};
  const auto cond_index{node.getInputs().at(ir::operation::Select::Input::COND)};
  const auto input1_index{node.getInputs().at(ir::operation::Select::Input::INPUT1)};
  const auto input2_index{node.getInputs().at(ir::operation::Select::Input::INPUT2)};

  auto output_alloc = _tensor_builder->at(output_index).get();
  auto cond_alloc = _tensor_builder->at(cond_index).get();
  auto input1_alloc = _tensor_builder->at(input1_index).get();
  auto input2_alloc = _tensor_builder->at(input2_index).get();

  auto fn = std::make_unique<::arm_compute::CLSelect>();

  fn->configure(cond_alloc->handle(), input1_alloc->handle(), input2_alloc->handle(),
                output_alloc->handle());

  auto acl_fn = asAclFunction(std::move(fn));

  _execution_builder->append(std::move(acl_fn));
}
```

acl_neon

Similar implementation as acl_cl is required.

cpu

- KernelGenerator.h

```
void visit(const ir::operation::Select &) override;
```

- KernelGenerator.cc

```
void KernelGenerator::visit(const ir::operation::Select &node)
{
  const auto output_index{node.getOutputs().at(0)};
  const auto condition_index{node.getInputs().
↪at(ir::operation::Select::Input::CONDITION)};
  const auto true_index{node.getInputs().at(ir::operation::Select::Input::INPUT_TRUE)};
  const auto false_index{node.getInputs().at(ir::operation::Select::Input::INPUT_FALSE)};

  auto output_tensor = _tensor_reg->getPortableTensor(output_index);
  auto condition_tensor = _tensor_reg->getPortableTensor(condition_index);
  auto true_tensor = _tensor_reg->getPortableTensor(true_index);
  auto false_tensor = _tensor_reg->getPortableTensor(false_index);

  auto fn = std::make_unique<ops::SelectLayer>();

  fn->configure(condition_tensor, true_tensor, false_tensor, output_tensor);
```

(continues on next page)

(continued from previous page)

```
  _return_fn = std::move(fn);
}
```

ConstantInitializer (in some cases)

This component registers function initializing constant tensors and initialize constant tensor layer. Most tensors will be automatically registered internally. And there are some exceptions.

cpu

- ConstantInitializer.h

```
void visit(const ir::operation::Conv2D &) override;
```

- ConstantInitializer.cc

```
void ConstantInitializer::visit(const ir::operation::Conv2D &node)
{
  const auto &kernel_index = node.getInputs().at(ir::operation::Conv2D::KERNEL);
  const auto &kernel_obj = _operands.at(kernel_index);
  registerCopyInitializer(kernel_index, kernel_obj);

  const auto &bias_index = node.getInputs().at(ir::operation::Conv2D::BIAS);
  const auto &bias_obj = _operands.at(bias_index);
  registerCopyInitializer(bias_index, bias_obj);
}
```

2.19.5 Samples (to be updated)

- `Select` operation
  - Simple explanation : `Output[i] = Condition[i] ? input1[i] :  input2[i]`
  - PR : https://github.com/Samsung/ONE/pull/XXX

CHAPTER

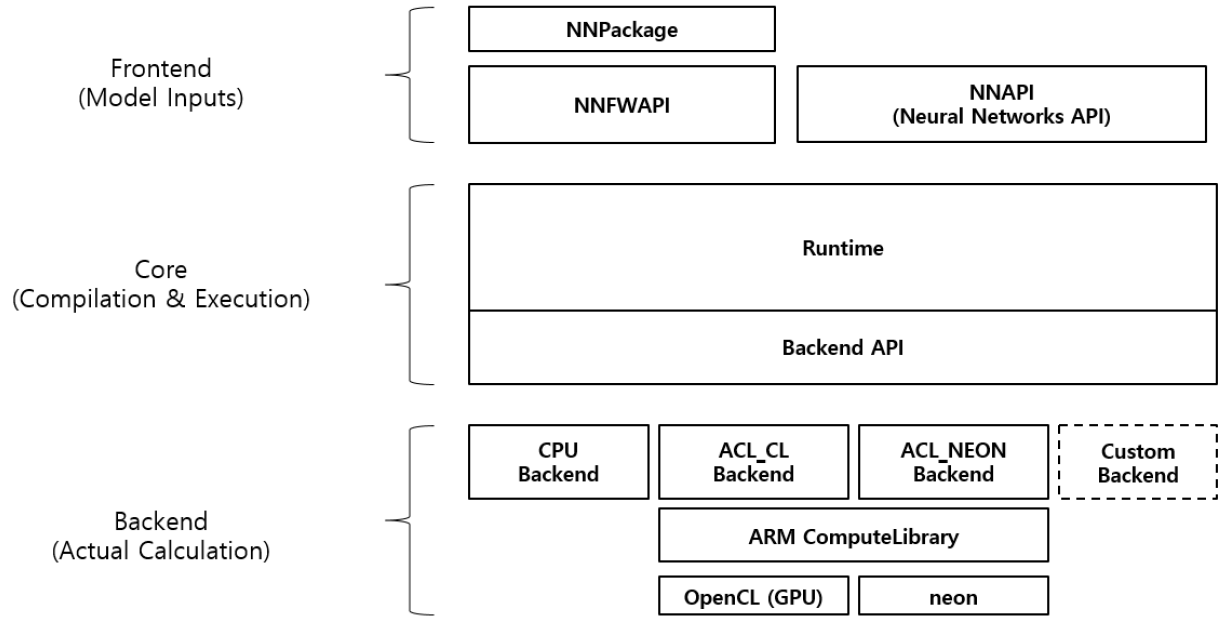
THREE

RUNTIME

3.1 API

3.1.1 Runtime Layered Architecture

Here is a figure of runtime layered architecture.



There are three parts - Frontend, Core and Backend. Core works with Frontend and Backend API. Frontend gets user inputs(neural networks models) and Backend does the actual computation.

3.1.2 Frontend API

Frontend API is about from creation/loading the model and

Runtime supports two (frontend) APIs - NN API and NNFW API.

NN API

NN API stands for Android Neural Networks API. It is part of Android Open Source Project and we provide a binding between NN API and One Runtime.

For usage, refer to *Howto : NN API*.

NNFW API

NNFW API is ONE's own API. It supports loading models from NN Packages. As it is our own API, It can do most of functionalities that One Runtime offers. Representatively, it provides functions for execution with multiple backends.

For usage, refer to *Howto : NNFW API*.

3.1.3 Backend API

Backend API enables anyone to extend the runtime in terms of operation computation and memory management.

For detailed descriptions, refer to *Backend API*.

3.2 Core

Runtime Core is a compilation/execution engine for neural network models.

3.2.1 Modules

Runtime Core has four modules. These are namespaces as well as directory names in `/runtime/onert/core/src/`.

- `ir` stands for Intermediate Representation which contains Neural Network Graph data structures
- `compiler` converts IR to an executable format
- `exec` is an execution module which is the result of a compilation
- `backend` is an interface for memory management for operands and actual calculation of operations

Module `ir`

This module contains data structures of pure Neural Network models. The models from NN Packages or NN API are converted to these structures.

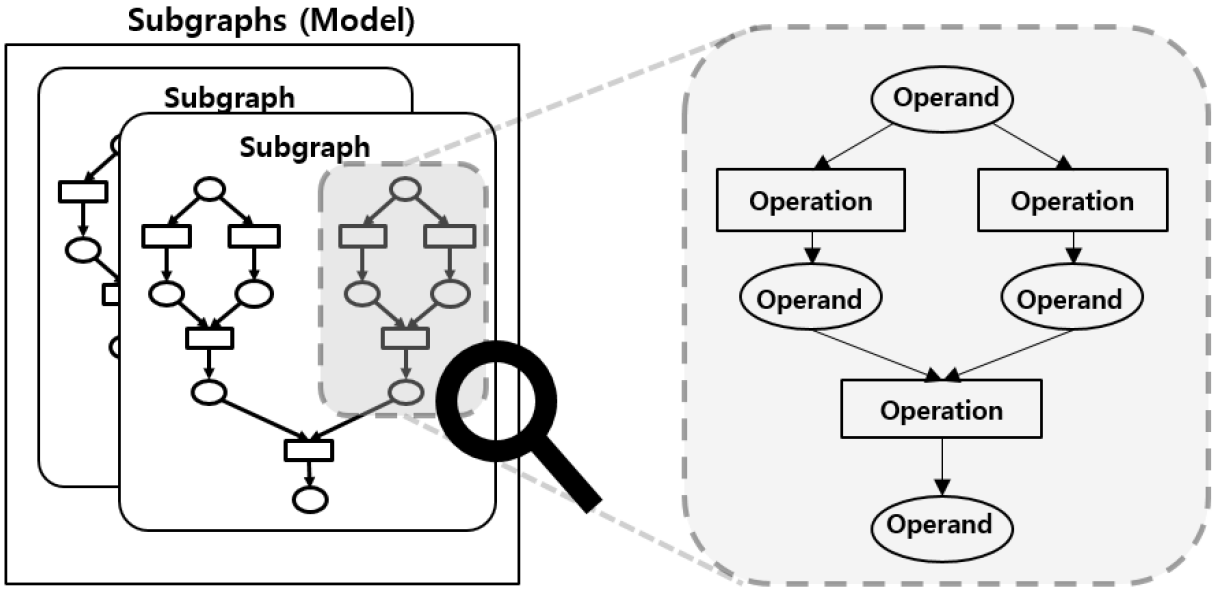
- `Subgraphs` is the entire neural network model which is a set of subgraphs
- `Subgraph` consists of operands and operations
- `Operand` (a.k.a. Tensor) has a shape, data type, data and references to operations
- `Operation` (a.k.a. Operator) has operation type, params, and references to operands

`Operand` and `Operation` are nodes of the graph, and the reference relationship between them is the edges of the graph.

`Subgraphs` represents the whole model. A model can have more than one `Subgraph` to support control flow operations. Those operations make calls to another subgraph and when the execution on another subgraph is done it gets back to previous subgraph execution with returned operands.

All graphs are a DAG so once model inputs are given we can run it in topological order.

Here's a figure of how those data structures are organized.



Module `compiler`

`Compiler` is the main class of this module. Everything starts from it.

What it does is making models executable. It schedules execution order and assigns a backend for each operation. Here are major phases of compilation.

1. Lowering

In Lowering, `Compiler` assigns a *backend* for each operation. It means that the operation will be run with the assigned backend's kernel.

There is a scheduler that allows the user to manually specify backends via compiler options. There is another scheduler that automatically assigns backends based on profile information measured in advance and saved.

2. Tensor Registration

Each backend manages its tensors. In this phase, operand informations get registered to the corresponding backend. This will be used in generating tensor objects.

Q. What are the differences between 'operand' and 'tensor'?

In **ONE** runtime, 'operand' refers to an operand in a neural network model. While 'tensor' includes all 'operand' info plus actual execution info like actual buffer pointer. In short, 'operand' is for `ir`, 'tensor' is for `backend`.

3. Linearization (Linear Executor Only)

Linearization means sorting operations in topological order. It saves execution time since it is not needed to resolve the next available operations after every operation at execution time.

It also makes plans for tensor memory. It can save some memory space by reusing other operands' space that does not overlap lifetime. All allocations are done at compile time (after *4. Kernel Generation*) which saves execution time too.

4. Kernel Generation

'kernel' here means an implementation of the actual calculation of an operation.

A backend is assigned for each operation. In this phase, a kernel for each operation is generated.

Let's say we have some functions written in a certain programming language. Then its compiler compiles each function into a chunk of assembly. Here 'function' is like 'operation' and 'assembly' is like 'kernel'.

5. Create Executor

With generated tensors and kernels, the compiler creates executor objects. There are 3 types of executors: Linear, Dataflow, and Parallel. Linear executor is the default executor and Dataflow Executor and Parallel Executor are experimental.

For more about executors, please refer to the Executors document.

Module `exec`

`exec` stands for 'execution'. As a result of the compilation, `Execution` class is created. This class manages the actual execution of the model inference. Here is a typical usage of this class.

1. Resize input size if needed
2. Provide input and output buffers
3. Run the inference in either synchronous or asynchronous mode
4. Check out the results which are stored in output buffers provided earlier

Module `backend`

Backends are plugins and they are loaded dynamically (via `dlopen`). So this module is a set of interface classes for backend implementation. `compiler` can compile with a variety of backends without knowing specific backend implementation.

Backend interface classes are mostly about memory management and kernel generation. For more, please refer to the Backend API document.

3.3 Controlflow Operations

We call the `If` and `While` operations "Controlflow operations". These operations are special. Instead of computing data, they are used to invoke another subgraph and return back which constitutes conditional/iterations work in dataflow models.

3.3.1 Defining controlflow operations

As we use Tensorflow Lite schema (or Circle which is based on TF Lite), the runtime follows the way TF Lite does. The details are stated in the Control Flow in TensorFlow Lite RFC document.

Controlflow operations from NN API are not yet supported. But we expect that they can be enabled in a similar way.

3.3.2 Implementation

Graph representation

`onert` internally has its representation for controlflow operations and subgraphs. It is straightforward as it is pretty much isomorphic with the schema. The `onert`'s in-memory model contains multiple subgraphs and the controlflow operations have same parameters (subgraph indices), just like TF Lite schema has.

Execution

The `controlflow` backend is a built-in backend to support these controlflow operations. This backend is special as it has access to `onert` core's executor manager (`ExecutorMap`) so it can invoke/return a subgraph. This backend has implementations for `If` and `While` operations and they make use of the access to executor manager.

An `Executor` has two different ways to execute depending on if it is the initial execution or invoking a subgraph from a controlflow operation.

- Executing the primary subgraph
  - Pass user-given tensors as the subgraph inputs and outputs
- Executing a subgraph for controlflow operations
  - Pass controlflow operation inputs tensors as the subgraph inputs
  - Pass the subgraph outputs as controlflow operation outputs

Kernel Implementation

Here is a brief explanation what the kernels do, which is quoted from Control Flow in TensorFlow Lite.

- `If` : Check the condition input and invoke one of the 2 subgraphs.
- `While` :
  - Invoke the condition subgraph. Break out the loop if the result is false.
  - Invoke the body subgraph, use the output as the input of the next iteration.

Invoking a subgraph needs to pass the operation's inputs to the subgraph inputs. And Returning back needs to pass the subgraph outputs to the operation outputs.

When invoking a subgraph and returning back, the current kernel implementation makes a copy of all the subgraph inputs and outputs. This is going to be optimized to minimize redundant copies.

3.4 Executors

Executor (`IExecutor`) is an execution engine of a subgraph that can execute inference for the subgraph. It is the result of a `Subgraph` compilation. Compared to common programming language tools, it is like an interpreter with code to execute.

3.4.1 Understanding models

We can think of an NNPackage model as a set of tasks with dependencies. In other words, it is a form of DAG (more precisely, it is a set of DAGs, as we need multiple subgraphs to support control flow operations). And that is exactly the same concept with Dataflow programming.

That is, there are some input tensors that must be ready to run a operation. And the execution must be done in topological order. Here's the workflow for execution.

1. User gives model input tensors
2. Start execution
3. Perform the operations that are ready
4. Mark the tensors as ready that are made from the operations that was just performed
5. Check if there are some operations ready
    1. If yes, Go to 3

    2. Otherwise, Finish execution

6. User consumes data of model output tensors

We have 3 different types of executors in our codebase and they all are based on the above explanation. However, only `LinearExecutor` is official and the other two are experimental.

3.4.2 Linear Executor

`LinearExecutor` is the main executor. As we know the model to run and the model graph does not change at runtime, we do not need to do the above steps 3-5 at runtime. During the compilation for Linear Executor, it sorts operations in topological order so we can just execute in that fixed order which means that it cannot perform the operations in parallel.

If the tensors are static, it also can analyze the lifetimes of the tensors and pre-allocate tensor memory with reusing memory between the tensors whose lifetimes do not overlap.

3.4.3 Dataflow Executor (experimental)

Unlike `LinearExecutor`, `DataflowExecutor` does steps 3-5 at runtime. By doing it we can know which operations are available at a specific point. However this executor still executes the operations one at a time. Just choose any operation that is ready then execute, wait for it to finish then repeat. So there may be no advantage compared to `LinearExecutor` but `DataflowExecutor` is the parent class of `ParallelExecutor`. And `DataflowExecutor` can be used for profiling executions for the heterogeneous scheduler.

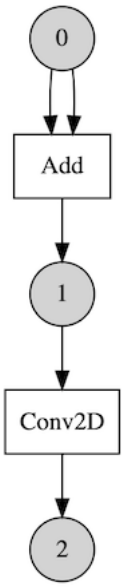
3.4.4 Parallel Executor (experimental)

Just like `DataflowExecutor`, `ParallelExecutor` does steps 3-5 at runtime. One big difference is that it creates a `ThreadPool` for each backend for parallel execution (`ThreadPool` is supposed to have multiple threads, however for now, it can have only one thread). Multiple operations ready to execute can be executed in different backends at the same time, which could lead to some performance gain.

3.5 Heterogeneous Execution

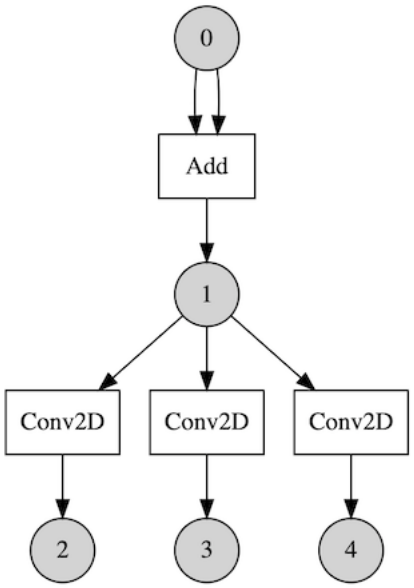
A major feature of ONE Runtime is that it can perform inference for a neural network model with different backends (computation libraries). This is called “Heterogenous Execution”.

A neural network model consists of operations and each operation can be computed by a variety of backends. Then we can choose the faster one for a given computation. Let’s say we have a model that has two operations in sequence.



And say we have two backends A and B. A has a faster kernel for Add and B has a faster kernel for Conv2D. Then it might be the best choice to run Add with A and run Conv2D with B.

Here is another case. Let's say we have a model that is not sequential so there are multiple operations can be run at the same time.

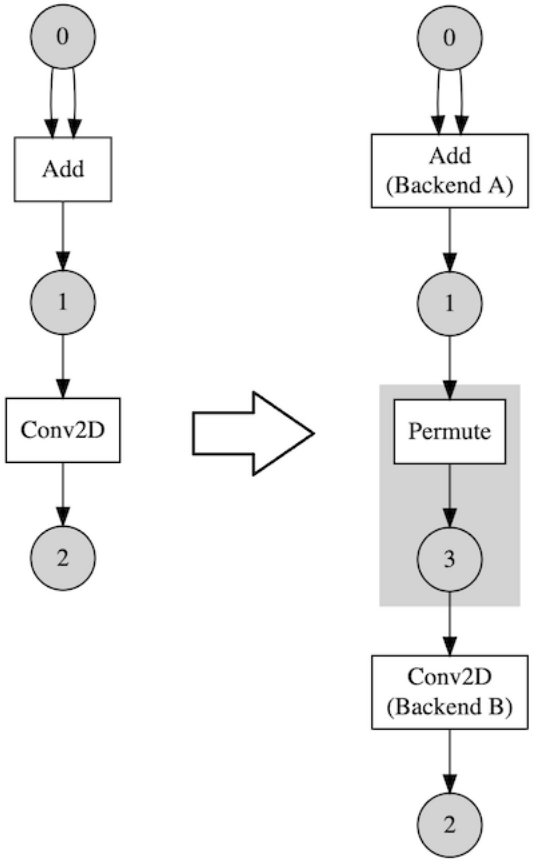


Say we have 3 backends that are based on CPU, GPU and NPU (Neural Processing Unit) respectively. After executing Add, 3 Conv2D operations are ready to run. We may utilize those backends with Parallel Executor (experimental). For this case we may get performance gain regardless of kernels' speed as those are run in parallel independently.

3.5.1 Graph Transformation

Unfortunately, it is not that simple to get a performance gain. As each backend has its own memory management module, a copy must be done between backend boundaries. Plus, it may require layout changes so "Permute" operations are added from `PermutationInsertionPass`. This process is done from the Lowering phase of compilation.

Here is an example of that. Let's say we have assigned different backends for Add and Conv2D. So a Permute operation is inserted between them.



This means that choosing a backend for an operation should be careful. We must take Permute(copy) overhead into account. That cost could dominate actual computation's so choosing the fastest one is not always the best choice.

Sometime later, we are going to introduce a concept of compatible backends and they can share their tensors freely. So for those cases, Permute operations are not inserted.

3.5.2 Automatic Scheduling

Now we know that it could be useful to have a lot of backends, but one might wonder how we can schedule backends effectively. When a model is large, it is not trivial to do it manually by a user.

There is a profiling-based automatic scheduler (experimental). It first runs every operation in sequence with every backend it has. And it records the duration of the execution for each. Then that data can be used to schedule. The scheduler assigns backends according to a certain algorithm with the given data. And the user can execute it with Parallel Scheduler.

There were actually some performance gain with the model Inception V3 and two backends - acl_neon(GPU) and acl_cl(GPU). Note that this model contains operations that can be run at the same time.

3.6 Backend API

Backend API is defined by One Runtime. It is about actual computation of operations and memory management for operands. In order to allow different kinds of computation units or libraries, Backend API is exposed to support user defined operation kernels and memory manager. It contains several C++ interface classes which are **subject to change**.

3.6.1 How backends are loaded

When a backend ID is given to a session, the compiler module tries to load `libbackend_{BACKEND_ID}.so`. If it is successful, the runtime looks up for C API functions in it and makes use of those.

3.6.2 C and C++ API

C API

We have two C API functions which are used as the entrypoint and the exitpoint. Here are the definitions of those.

```
onert::backend::Backend *onert_backend_create();
void onert_backend_destroy(onert::backend::Backend *backend);
```

What they do is creating a C++ object and destroying it, respectively. These two functions are the only ones that are dynamically resolved at runtime.

C++ API

> **NOTE** C++ API is subject to change so it may change in every release

C API above is just an entrypoint and it delegates core stuff to the C++ API.

Major classes are described below. One must implement these classes (and some more classes) to create a backend.

- `Backend` : Responsible for creating a backend context which is a set of backend components
- `BackendContext` : Holds data for the current session and also responsible for creation of tensor objects and kernels
  - `BackendContext::genTensors` : Creates tensor objects
  - `BackendContext::genKernels` : Creates kernels
- `IConfig` : Configurations and miscellaneous stuff (global, not session based)
- `ITensorRegistry` : A set of tensor (`ITensor`) objects that are used by the current backend

Please refer to each class document for details. You may refer to *Bundle Backends* for actual implementation samples.

3.6.3 Provided Backend Implementations

We provide some backends along with the runtime. There is the special backend `builtin` which is part of runtime core, and some bundle backends which are baseline backends and samples of backend implementation.

3.6.4 `builtin` Backend

`builtin` is a special backend that is always loaded (statically linked, part of runtime core). It is implemented just like other backends, but there are some things that it does exclusively.

- Has kernels for If, While and Permute operations (Kernels from other backends are never used)
- The runtime core directly creates `builtin`'s tensor objects to accept user-given input and output buffers
- The runtime core gives the executor a context to `builtin` backend which lets control flow ops properly change the execution flow

3.6.5 Bundle Backends

Without actual implementation of backends, we cannot run any model. So we provide 3 bundle backends which support dozens of operations.

cpu

This backend is written in C++ and all the computation is done exclusively on a CPU.

acl_neon

`acl_neon` is a backend that is an adaptation layer of ARM ComputeLibrary NE (NEON) part. So it's CPU-only and restricted to ARM.

acl_cl

`acl_cl` is a backend that is an adaptation layer of ARM ComputeLibrary CL (OpenCL) part. OpenCL support (`libOpenCL.so`) is also necessary in the running environment to be able to use this backend. Also, it works only on ARM.

3.7 Compute

`compute` directory is for the libraries for actual computation of neural network operations. These libraries are used by backends. Currently we have two libraries.

3.7.1 ARMComputeEx

It is an extension of ARM ComputeLibrary, in order to support some operations that are not yet supported by ComputeLibrary. It is used by `acl_cl` and `acl_neon` backends.

The code structure looks just like ComputeLibrary's. Some of the code could be copied from the latest version of ComputeLibrary to support some operations quickly when those are not included in the latest version yet.

3.7.2 cker

"cker" stands for Cpu KERnel. It is a port of Tensorflow lite's operation kernels with some additions. It is used by the `cpu` backend.

3.8 Supported Operations and backend

As of 2021-03-08

3.8.1 Raw-data format (float32, int32, boolean, etc)

3.8.2 Quantization format (uint8 asymmetric)

3.8.3 Quantization format (int8)

CHAPTER

FOUR

COMPILER

4.1 Frontend

4.2 Middleend

4.3 Backend

4.4 IR

4.5 Interpreter

4.6 Libraries

4.7 Tools

CHAPTER

FIVE

COMMON IR

5.1 Introduction to circle

5.2 What is Common IR

CHAPTER

SIX

PACKAGE

6.1 Introduction to Package

6.2 Design of Package

6.2.1 Manifest

6.2.2 Supported Models

6.2.3 User Defined Operation

CHAPTER

SEVEN

PLATFORM

7.1 Ubuntu

7.2 Tizen

7.3 Android

CHAPTER

EIGHT

DEVICES

8.1 ODROID-XU3

8.2 ODROID-XU4

8.3 Raspberry Pi 3

CHAPTER

NINE

TEST & BENCHMARKS

9.1 Scripts

9.2 Benchmarks

CHAPTER

TEN

RELEASE

10.1 1.0

10.1.1 Release Note 1.0.0

10.2 1.1

10.2.1 Release Note 1.1.0

10.3 1.2

10.3.1 Release Note 1.2.0

10.4 1.3

10.4.1 Release Note 1.3.0

10.5 1.4

10.5.1 Release Note 1.4.0

10.6 1.5

10.6.1 Release Note 1.5.0

Feature Highlights

- **ONE** Compiler
  - Compiler supports more operations
- **ONE** Runtime
  - CPU backend supports more operations

ONE Compiler

Compiler supports more operations

The following operations are supported :

- Abs, Add, ArgMax, AvgPool2D, BatchToSpaceND, Cast, Concat, Const, Conv2D, Cos, Custom, DepthwiseConv2D, Div, Elu, Equal, Exp, ExpandDims, Fill, FloorDiv, FloorMod, FullyConnected, Gather, GatherNd,

Greater, GreaterEqual, If, LeakyRelu, Less, LocalResponseNormalize, LogicalAnd, LogicalNot, LogicalOr, Logistic, Maximum, MaxPool2D, Mean, Minimum, MirrorPad, Mul, Neg, NotEqual, OneHot, Pack, Pad, Pow, Range, ReduceAny(Any), ReduceMax(Max), ReduceProd, ReduceSum(Sum), ReLU, RELU_N1_TO_1, Reshape, ResizeNearestneighbor, Rsqrt, Select, Shape, Sin, Slice, Softmax, SpaceToBatchND, SpaceToDepth, Split, SplitV, Square, SquaredDifference, Squeeze, StridedSlice, Sub, Tanh, Tile, TopKV2, Transpose, Unpack(Unstack), While, ZerosLike

ONE Runtime

CPU backend supports more operations

The following operations are supported on CPU backend :

- ArgMax, Cos, ExpandDims, Fill, Log, LogicalNot, LogicalOr, Mean, Neg, Pow, ReLU, ReduceAny, ReduceProd, Reverse, Round, Select, SquaredDifference, Tile, ZerosLike

10.7 1.6

10.7.1 Release Note 1.6.0

Feature Highlights

- **ONE** Compiler
  - Compiler supports 22 more operations including 1 custom operation.
- **ONE** Runtime
  - CPU backend supports 8 more operations
  - Support dynamically shaped tensors
  - Support Control Flow operations
  - API updates
- Add **ONE** Runtime package for `x86_64` linux (Experimental)

ONE Compiler

Compiler supports more operations

- AddN, ArgMin, Custom(BatchMatmul), Ceil, DepthToSpace, Floor, InstanceNormalize, L2Normalization, L2Pool, LessEqual, Log, LogSoftmax, PReLU, Rank, ReduceMin(Min), ResizeBilinear, Round, ScatterND, Sqrt, TransposeConv, BCQGather, BCQFullyConnected

ONE Runtime

CPU backend supports more operations

- BatchMatMul, BroadcastTo, Einsum, FusedBatchNorm, MatrixBandPart, Range, ReduceAll, Add(quant8)

Support dynamically shaped tensors

- Support static shape inference (input resizing)
- Support dynamic shape inference (general resizing)

Support Control Flow operations

- IF and WHILE
- Fully support static and dynamic tensors

API updates

- Introduce `nnfw_set_input_tensorinfo` for input resizing
- `nnfw_input_tensorinfo` and `nnfw_output_tensorinfo` behavior have changed to return tensorinfo according to the session state

10.8 1.7

10.8.1 Feature Highlights

- **ONE** Compiler
  - Compiler supports more operations
  - New command line interface for user interface consistancy
- **ONE** Runtime
  - Runtime CPU backend supports more operations
  - Runtime CPU backend supports more quant8 operations
  - API changes
  - New optimization

10.8.2 ONE Compiler

Compiler supports more operations

- MatrixDiag, MatrixSetDiag, ReverseSequence, ReverseV2, SegmentSum, SelectV2, SparseToDense, Where

New command line interface for user interface consistancy

- one-import: imports conventional model files to circle
  - one-import-tf: imports TensorFlow model to circle
  - one-import-tflite: imports TensorFlow lite model to circle
- one-optimize: circle optimize command
- one-quantize: circle quantize command
  - supports float32 to uint8, layer wise (for Conv series)
- one-pack: package command
- one-prepare-venv: prepares python virtual environment for importing TensorFlow model
- one-codegen: backend(if available) code generator

10.8.3 ONE Runtime

Runtime CPU backend supports more operations

- LogSoftmax, SpaceToBatchND

Runtime CPU backend supports more quant8 operations

- Logistic, Mul, Tanh, SpaceToBatchND, Transpose, Sub, Max, Min, Less, Greater, GreaterEqual, LessEqual, Equal, NotEqual

API changes

- Introduce basic asynchronous execution API

New optimization

- Remove dynamic tensor overhead from static models

10.9 1.8

10.9.1 Release Note 1.8.0

Feature Highlights

- **ONE** Compiler
  - Support new command line interface
- **ONE** Runtime
  - CPU backend supports 7 more operations
  - CPU backend supports 9 more quant8 operations

ONE Compiler

New command line interface for user interface consistancy

- `one-import-bcq` : import BCQ(Binary coding quantized) TensorFlow model
- Commands now support `--version` option to show version number

Changes

- Experimental support for TensorFlow 2.x has updated to 2.3.0 (TensorFlow 1.3.2 is our official support version)
- Support more operators in luci-interpreter
- Enhancing one-quantizer

ONE Runtime

Rename headers

- Rename `nnfw_dev.h` to `nnfw_experimental.h`

Optimization

- Remove copies for model input/outputs whenever possible

Support CPU backend operation

- BatchToSpaceND, L2Normalization, ReLU6, ResizeBilinear, SpaceToDepth, SplitV, StatelessRandomUniform

Support CPU backend quant8 operation

- BatchToSpaceND, L2Normalization, Pad, PadV2, ResizeBilinear, Slice, Quantize, SpaceToDepth, Sum

10.10 1.9

10.10.1 Release Note 1.9.0

ONE Compiler

Compiler supports more operations

- NonMaxSuppressionV4, NonMaxSuppressionV5, PadV2, Unique

Changes

- Quantization enhancements: channel wise UINT8 quantization(Conv2D, DepwiseConv, TransposeConv, FullyConnected)
- Experimental requantization from INT8 to UINT8
- Adding more operator value tests
- tf2tfliteV2 supports conversion from Keras model, saved model
- Refactoring for better maintenance long Class codes using visitor patterns
- Introducing optimization pass that fuses batch normalization with Transposed Convolution.

ONE Runtime

Runtime backend operation support

- CPU backend: RANK
- CPU backend qasymm uint8: LOG_SOFTMAX
- ACL-CL backend: LEAKY_RELU, RESIZE_NEAREST_NEIGHBOR

Optimization

- Copy Elimination between compatible backends

Operation Implementation

- Operations with same parameters are unified

Change

- CPU backend qasymm uint8 performance enhancement: arithmetic operations

10.10.2 Release Note 1.9.1

ONE Compiler

Changes

- `tf2nnpkg` now supports to import TensorFlow model which includes BCQ information.
- Minor change for preserving BCQ information.
- Fix invalid input arguments and add more error handles for one-cmds

10.11 1.10

10.11.1 Release Note 1.10.0

ONE Compiler

Compiler supports more operations

- Dequantize, UnidirectionalSequenceLSTM

Changes

- New `--fold_dequantize` option in `one-optimize`
- New `--fuse_add_with_tconv` option in `one-optimize`
- Support `int16` quantization in `one-quantize`
- Test scripts are added for basic testing of one-cmds command line tools
- Bug fixes for one-cmds command line tools

ONE Runtime

Runtime backend operation support

- ACL-CL backend: OneHot
- CPU backend: FullyConnected for Float32 16x1 Block Sparsity

Optimization

- Speed up for ReduceSum, StrideSlice and BinaryArithmetic in CPU backend

10.12 1.11

10.12.1 Release Note 1.11.0

ONE Compiler

Compiler supports more operations

- MaxPoolWithArgMax by CustomOp

Changes

- `one-build` command added as representative command
- one-cmds are now revised to python script and supports configuration file as input parameters
- added `rawdata2hdf5` tool to help creating input datasets for calibration
- added more optimization passes in `one-optimize`; `fuse_preactivation_batchnorm`, `make_batchnorm_gamma_positive` and `fuse_activation_function`

ONE Runtime

Runtime backend operation supports more operations and types

- CPU backend
  - float: AddN, Floor, UniDirectionalSequenceLSTM
  - uint8: Dequantize, Rank
  - int8: Dequantize, Rank, Shape

10.13 1.12

10.13.1 Release Note 1.12.0

ONE Compiler

Compiler Frontend

- Add optimization pass: ReplaceMulAddWithDepthwiseConvPass, SubstitutePackToReshape, RemoveRedundantTranspose, ShuffleWeightTo16x1Float32Pass
- Add quantization for InstanceNorm.
- Fix bug of `one-import-bcq` command for `--v1`, `--v2` arguments.
- Fix FuseBCQPass to work with inter-subgraphs in the model file and minor BCQ related optimizations.

ONE Runtime

Runtime backend operation supports more operations and types

- CPU backend
  - Concat: int8
  - DepthToSpace: float, uint8, int8
  - LeakyRelu: float
- ACL-CL backend
  - ArgMin: float, uint8, int8
- ACL-NEON backend
  - ArgMax: int8
  - ArgMin: float, uint8, int8

nnpackage defines configuration file

- Allow users to set configuration variable via conf file. For more information, See nnpackage spec

10.14 1.13

10.14.1 Release Note 1.13.0

ONE Compiler

Compiler Frontend

- Add optimization pass: ConvertNCHWToNHWC, FoldSparseToDensePass, FuseBatchNormWithConvPass, ForwardReshapeToUnaryOpPass, RemoveUnnecessarySlicePass, RemoveUnnecessarySplitPass, RemoveUnnecessaryReshapePass, RemoveRedundantReshape, SubstituteTransposeToReshapePass, SubstituteSqueezeToReshapePass,
- Support more operators: FAKE_QUANT
- Enhancements: Support auto generated random input for record-minmax (for better quantization testing)
- Changes: `--all` option to `--O1` in circle2circle(and one-optimize)
- Fixes: `tf2tfliteV2` accept input shapes `--v2` option, lots of fixes for increase test coverage
- Experimental: Compile ONNX models to circle

10.15 1.14

10.15.1 Release Note 1.14.0

ONE Compiler

Compiler Frontend

- `one-codegen` interface now distinguishes own arguments from backend's.
- Adds `RemoveUnnecessaryStridedSlice` optimization pass.
- Introduces experimental support for generating profile data.
  - Adds `--generate_profile_data` option to `one-optimize`, `one-quantize`.

10.16 1.15

10.16.1 Release Note 1.15.0

ONE Compiler

Compiler Frontend

- Support more Ops for quantization
- Fix `record-minmax` tool for bool type, NaN values
- Fix `one-cmds` test scripts
- Remove `stdex` module
- `arser` supports short option

ONE Runtime

Runtime backend supports more operations and types

- CPU backend
  - Add: int8
  - AvgPool2d: int8
  - Conv2D: int8
  - DepthwiseConv2D: int8
  - Div: uint8
  - Elu: float
  - ExpandDims: int8
  - LogicalAnd: boolean
  - Maximum: uint8
  - MaxPool2D: int8
  - Minimum: uint8
  - Mul: int8
  - Pad: int8
  - PadV2: int8
  - Quantize: uint8, int8
  - Reshape: int8
  - Resizebiliear: int8
  - Softmax: int8
  - Squeeze: int8
  - Sub: int8

ARM Compute Library Update

- ONERT uses Compute Library v21.02

10.17 1.16

10.17.1 Release Note 1.16.0

ONE Compiler

Compiler Frontend

- Enable `PadV2` in luci-interpreter and quantization
- Provide `circle-tensordump`, `circledump` as a development tool
- Provide `luci-eval-driver` as test tool
- Enable `STRING` type as constant values in CircleConst
- Fix CircleCustom may have 0 input, 0 output

- Enable debian package generation
- More optimization pass
  - Min(6)+ReLU to ReLU6
  - Remove FakeQuant Op
- Experimental support of ONNX upgraded to version 1.8.0 with additional patch
- Fix bugs where one-cmds' config file didn't evaluate boolean properly

10.17.2 Release Note 1.16.1

ONE Compiler

Compiler Frontend

- Extends the point where `one-codegen` finds backends.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search