AN13699

Enabling TensorFlow Operators in TensorFlow Lite Runtime for i.MX 8 Linux Platform

Rev. 0 — 16 August 2022 Application note

Document information

Information	Content
Keywords	
Abstract	

1 Introduction

TensorFlow Lite is a popular open-source inference engine for mobile and edge devices. It comes with rich set of supported operators.

The TensorFlow Lite Operator Set counts more than a hundred of frequently used operators and layers and the majority of ML models can fit into it. The TensorFlow Lite Op Set is only a subset of the TensorFlow Operator Set, counting more than 10000 operators, layers, and algorithms. By building the ML model, you can easily come to a model with excellent performance, which goes beyond the TensorFlow Lite Op Set. In such a case, engineers are either requested to adjust the model, which can harm the performance, or to extend the TensorFlow Lite Runtime Op Set by unsupported operators.

TensorFlow Lite helps with two approaches. There is an option to implement and embed a custom operator to TensorFlow Lite or leverage the already available operator from TensorFlow Runtime.

This document presents the later approach and provide a guideline of how the TensorFlow Lite runtime with the TensorFlow operator can be built and deployed for NXP i.MX 8 devices.

The solution is demonstrated on two examples. The first example demonstrates a simple model with the TensorFlow operator inside of the TensorFlow Lite runtime. The second example shows a more complex model, which leverages the TensorFlow operators and functions for model training directly with the TensorFlow Lite runtime on an edge device.

The TensorFlow Lite is part of the NXP eIQ® ML Software Development Environment, which is available on Yocto Linux and Android platforms for i.MX 8 devices. For more information, see the ML User's Guide for Yocto Linux.

2 TensorFlow and TensorFlow Lite Operator Set

If the designed model exceeds the TensorFlow Lite Operator Set, the TensorFlow Lite converter raises an error, describing which operator is not supported in TensorFlow Lite:

```
Some operators are not supported by the native TFLite runtime,
 but you can enable the TF kernels fallback using TF Select.
 The instructions are at https://www.tensorflow.org/lite/guide/
ops_select.
TF Select ops: Roll
Details: tf.Roll(tensor<?x10xf32>, tensor<i32>, tensor<i32>) ->
 (tensor<?x10xf32>) : {device = ""}
```

To convert such a model, the Select TensorFlow Operator feature must be enabled in the converter by allowing the TensorFlow Operator Set as follows:

```
   converter.target_spec.supported_ops = [
        tf.lite.OpsSet.SELECT_TF_OPS  # enable TensorFlow ops.
]
```

The model converted with SELECT_TF_OPS enabled needs the corresponding TensorFlow Operators support in the TensorFlow Lite runtime. This is supported via the FLEX Delegate. The FLEX Delegate is the TensorFlow Lite counterpart for the Select TensorFlow Operators and bridges the TensorFlow Lite and TensorFlow Runtimes.

3 Building the TensorFlow Lite Library with the Flex Delegate for i.MX 8 platforms

The build of the TensorFlow Lite with Flex Delegate is only supported in the Bazel build system.

3.1 Building TensorFlow Lite Library with Flex Delegate support for i.MX 8

The library can be built directly by Bazel on any supported host or inside of the Docker container, which is available for TensorFlow. We recommend to use Docker, because the environment is ready for TensorFlow compilation. A compilation outside the Docker may fail for unknown reasons.

To build the library outside the Docker image, install the Bazel build system on the machine. TensorFlow requires the exact version of Bazel, which is specific to the particular TensorFlow version. We recommend to use Bazelisk to handle the Bazel version management. The Bazelisk tool is available on its GitHub space https://github.com/bazelbuild/bazelisk with pre-built executables for multiple platforms available.

To build the TensorFlow Lite library, download the following TensorFlow sources:

Clone the TensorFlow repository from https://github.com/NXPmicro/tensorflow and check out the imx-ODT-example branch:

```
$ git clone https://github.com/NXPmicro/tensorflow.git
$ cd tensorflow
$ git checkout imx-ODT-example
```

If you prefer to use the Docker image for building, setup the docker VM:

Note: *Depending on the host, the Docker may require administrative privileges to run (For example, "sudo" in Linux). Alternatively, the docker daemon can run as a non-root user (rootless mode), as described at https://docs.docker.com/engine/security/rootless/.*

1. Download the "tensorflow/tensorflow:devel" docker image. This image contains the Bazel and other required tooling for TensorFlow compilation:

   ```
   $ docker pull tensorflow/tensorflow:devel
   ```

2. Run the Docker VM. During the build process, Bazel downloads various packages from the internet. Internet access inside the Docker image is required. At least, initialize the "http_proxy and https_proxy" environmental variables inside the Docker image. The particular steps depend on your host configuration:

   ```
   $ docker run -e "http_proxy=<your-http-proxy>" \
                -e "https_proxy=<your-https-proxy>" \
                -e "no_proxy=localhost,127.0.0.1" \
                -it -w /tensorflow -v /<path-to-tensorflow-
   sources>:/tensorflow \
                -e HOST_PERMS="\\(id -u):\\)(id -g)" \
                tensorflow/tensorflow:devel bash
   ```

   At this point, we obtained the TensorFlow sources and initialized the build environment. We are ready to build the TensorFlow Lite with Flex Delegate support. NXP i.MX 8 platforms use Arm CPUs (aarch64) and we intend the build for the Linux

environment. We leverage the "elinux_aarch64" configuration, which is available for TensorFlow.

3. Configure the project using the "configure" script:

```
$ ./configure
```

4. Now we can proceed with the build. The Flex Delegate sources and Bazel build recipes are in the */tensorflow/lite/delegates/flex* folder. The following two libraries are defined:
   - "tensorflowlite_flex" – TensorFlow Lite Flex Delegate shared library (tensorflowlite_flex.so)
   - "delegate" – a special target for static linking of the TensorFlow Lite Flex Delegate. It is similar to the object library concept in CMake.
   - To build the shared library (tensorflowlite_flex.so), use the following command:

```
$ bazel --output_base=/tensorflow/docker-build/ build
 --config=monolithic --config=elinux_aarch64  -c opt //
tensorflow/lite/delegates/flex:tensorflowlite_flex
```

Note: *If the Docker is used for the build, it is useful to preserve Bazel's cache. Use the "--output_base" switch to override the default output base. For building outside of the Docker, this switch can be omitted. The directory must be available before running the Bazel build.*

The compiled library can be used with any build systems. Such approach still requires significant effort to include all the required sources and include files for the target application. The bazel target is more useful as a dependency for target application inside of the Bazel build system. We demonstrate the later approach on the well-known "benchmark_model" tool build.

To build the "benchmark model" tool with the Flex Delegate, there is a designated target in *//tensorflow/lite/tools/benchmark:benchmark_model_plus_flex*.

You can build it as follows:

```
$ bazel --output_base=/tensorflow/docker-build/ build --
config=monolithic --config=elinux_aarch64 -c opt //tensorflow/
lite/tools/benchmark:benchmark_model_plus_flex
```

The output of this is the "benchmark_model_plus_flex" binary with statically linked Flex Delegate. Use it directly on the i.MX 8 platform.

Let's review the dependencies of the following target:

```
deps = [
    ":benchmark_tflite_model_lib",
    "//tensorflow/lite/delegates/flex:delegate",
    "//tensorflow/lite/testing:init_tensorflow",
    "//tensorflow/lite/tools:logging",
],
```

The ":benchmark_tflite_model_lib" and "//tensorflow/lite/tools:logging" encapsulate the benchmark model source files and the logging tools. The "//tensorflow/lite/delegates/flex:delegate" is the Flex Delegate library and "//tensorflow/lite/testing:init_tensorflow" are additional dependencies to initialize the TensorFlow Runtime.

To create a binary with a dynamically linked Flex Delegate, create the "benchmark_model_plus_flex_dynamic" target in *tensorflow/lite/tools/benchmark/BUILD* as follows:

```
cc_import(
    name = "libtensorflowlite_flex",
    shared_library = "//tensorflow/lite/delegates/
flex:tensorflowlite_flex",
)

tf_cc_binary(
    name = "benchmark_model_plus_flex_dynamic",
    srcs = [
        "benchmark_plus_flex_main.cc",
    ],
    copts = common_copts,
    linkopts = tflite_linkopts() + select({
        "//tensorflow:android": [
            "-pie",  # Android 5.0 and later supports only PIE
            "-lm",  # some builtin ops, e.g., tanh, need -lm
        ],
        "//conditions:default": [],
    }),
    deps = [
        ":benchmark_tflite_model_lib",
        "//tensorflow/lite/testing:init_tensorflow",
        "//tensorflow/lite/tools:logging",
        ":libtensorflowlite_flex",
#       "//tensorflow/lite/delegates/flex:tensorflowlite_flex",
    ],
)
```

In the TensorFlow release 2.8.0, update the "tensorflowlite_flex" target in *//tensorflow/lite/delegates/flex/BUILD* file and mark it visible to other modules as follows:

```
tflite_flex_shared_library(
    name = "tensorflowlite_flex",
    visibility = ["//visibility:public"],
)
```

Update the "tflite_flex_shared_library" macro in the */tensorflow/lite/delegates/flex/build_def.bzl* file:

```
def tflite_flex_shared_library(
    ...
    tflite_cc_shared_object(
        name = name,
        visibility = visibility,
        linkopts = select({
    ...
```

By building the *//tensorflow/lite/tools/benchmark:benchmark_model_plus_flex_dynamic* target, note that "libtensorflowlite_flex.so" and "benchmark_model_plus_flex_dynamic" are both built.

3.2 Flex Delegate deployment on i.MX 8 Linux platform

All the build products mentioned in the previous chapter are ready for deployment on the i.MX 8 Linux platform. For the statically linked binary (in our case "benchmark_model_plus_flex") copy the binary to the target device root file system.

For the dynamically linked binary (in our case "benchmark_model_plus_flex_dynamic"), copy both the "libtensorflowlite_flex.so" and the binary to the target device "rootfs". Copy "libtensorflowlite_flex.so" to */usr/lib/* or set the LD_LIBRARY_PATH to load the library by the dynamic linker/loader.

3.3 Using hardware accelerators

The TF Operators are not part of the TFLite Operators Set, so the hardware accelerator on i.MX platforms does not support these operators. The acceleration of the TensorFlow Lite operators in this model is supported.

The hardware accelerators on the i.MX 8 Linux platforms are accessible via the VX Delegate, which is an external delegate for TensorFlow Lite. The "benchmark_model_plus_flex" already includes support for external delegates, so we can use the "–external_delegate_path" switch for inference acceleration:

```
$ ./benchmark_model_plus_flex_dynamic --
graph=/usr/bin/tensorflow-lite-2.8.0/examples/
mobilenet_v1_1.0_224_quant.tflite --enable_op_profiling=true --
external_delegate_path=/usr/lib/libvx_delegate.so
```

The support for external delegate in our "benchmark_model_plus_flex" binary is enabled by the following dependency chain in Bazel:

```
benchmark_model_plus_flex  -->
:benchmark_model_lib -->
//tensorflow/lite/tools/delegate:tflite_execution_providers --
>
//tensorflow/lite/tools/delegate:external_delegate_providers --
>
//tensorflow/lite/delegates/external:external_delegate
```

The last two are the important ones. The *//tensorflow/lite/tools/delegate:external_delegate_providers*, which registers the external delegate provider to the binary and exposes the "--external_delegate_path" command-line argument and the *//tensorflow/lite/delegates/external:external_delegate* contains the implementation of the external delegate itself.

4 Simple model with Flex Operator example

Here we demonstrate the inference with a simple model containing a TensorFlow operator. The example model is a simple CNN with the "tensorflow.roll()" operator, which is not present in the TensorFlow Lite 2.8.0 Operator Set:

```
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Conv2D,
 Softmax, Flatten
import os
```

```
import numpy as np

def make_simple_keras_model(input_shape, num_classes):
    inputs = Input(shape=input_shape, name="x")
    x = Conv2D(32,3,padding='valid', activation="relu")(inputs)
    x = Conv2D(64,3, padding='valid', activation='relu')(x)
    x = Flatten()(x)
    x = Dense(num_classes, activation='relu')(x)
    x = tf.roll(x,1,1)
    outputs = Softmax()(x)
    return Model(inputs, outputs)
```

We convert this model to TensorFlow Lite and quantize it. The usage of TensorFlow operators must be explicitly enabled by the *SELECT_TF_OPS* switch:

```
def convert_and_quantize_model_to_tflite(saved_model_dir,
 representative_data_gen):

    converter =
 tf.lite.TFLiteConverter.from_saved_model(saved_model_dir);
    converter.optimizations = [tf.lite.Optimize.DEFAULT]
    converter.representative_dataset = representative_data_gen
    converter.target_spec.supported_ops = [
        tf.lite.OpsSet.TFLITE_BUILTINS_INT8,
        tf.lite.OpsSet.SELECT_TF_OPS  # enable TensorFlow ops.
    ]
    converter.inference_input_type = tf.int8
    converter.inference_output_type = tf.int8

    converter.experimental_new_converter = True
    converter.experimental_new_quantizer = True

    tflite_model_quant_int8 = converter.convert()
    return tflite_model_quant_int8;

def save_tflite_model(tflite_model, path, name):
    name += ".tflite"
    full_path = os.path.join(path, name)
    open(full_path, "wb").write(tflite_model)
    print("Model saved to: ", full_path)

if __name__ == "__main__":
    print("TF version:", tf.__version__)
    model = make_simple_keras_model((28,28,1), 10)
    model.summary()

    mnist = tf.keras.datasets.mnist
    (train_images, train_labels), (test_images, test_labels) =
 mnist.load_data()
    train_images = train_images / 255.0
    test_images = test_images / 255.0

    train_labels = tf.keras.utils.to_categorical(train_labels)
    test_labels = tf.keras.utils.to_categorical(test_labels)

    model.compile(
        optimizer='adam',
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
```

```
    model.fit(train_images, train_labels, 32, 1)

    DIRECTORY = "SimpleFlexModel/"
    tf.saved_model.save(model, DIRECTORY)

    def representative_dataset_gen():
        counter = 0
        for i in train_images[1:100]:
            yield {"x" :
 i.astype(np.single).reshape(1,28,28,1)}

    tflite_model_int8 =
 convert_and_quantize_model_to_tflite(DIRECTORY,
 representative_dataset_gen)
    save_tflite_model(tflite_model_int8, DIRECTORY,
 "simple_flex_model_int8")
```

Now we obtained a simple TensorFlow Lite model with the TensorFlow operator. We can run this model with the "benchmark_model_plus_flex" binary, which we built in the previous chapter. Both the FlexDelegate and VX Delegate are invoked:

```
$ benchmark_model_plus_flex_dynamic --graph=./
simple_flex_model_int8.tflite --enable_op_profiling=true --
external_delegate_path=/usr/lib/libvx_delegate.so
STARTING!
Log parameter values verbosely: [0]
Graph: [/test_files/ML_ODT/simple_flex_model_int8.tflite]
Enable op profiling: [1]
External delegate path: [/usr/lib/libvx_delegate.so]
Loaded model /test_files/ML_ODT/simple_flex_model_int8.tflite
INFO: Created TensorFlow Lite delegate for select TF ops.
INFO: TfLiteFlexDelegate delegate: 1 nodes delegated out of 8
 nodes with 1 partitions.
...
Operator-wise Profiling Info for Regular Benchmark Runs:
============================== Run Order
 ==============================
    [node type]          [start]  [first]  [avg ms] [times
 called]  [Name]
    Vx Delegate          0.089    0.460    0.444         1
 [tfl.dequantize]:9
    TfLiteFlexDelegate   0.533    0.251    0.332         1
 [model/tf.roll/Roll]:8
    Vx Delegate          0.865    0.119    0.157         1
 [StatefulPartitionedCall:0]:10
...
```

5 On-device training example

In this example, we present the on-device training example available on https://www.tensorflow.org/lite/examples/on_device_training/overview and run it on the i.MX 8 Linux platform. This example demonstrates another usage of the Flex Delegate. The example pulls the operators and functions required for model training, such as "GradientTape" and loss functions, and makes them available in the TensorFlow Lite runtime on an embedded device.

As a starting point, build the model and convert it to the TFLite format using the Jupyter notebook at https://www.tensorflow.org/lite/examples/on_device_training/overview. The model can be also trained, but we will skip the model training for this demo, use a non-trained model, and perform the training procedure directly on the embedded device to make the (re-)training evident.

The model is available in "tflite_model". We must save it to a file:

```
open("./odt-model-empty.tflite", "wb").write(tflite_model)
```

The C++ code for the example is available at https://github.com/NXPmicro/tensorflow/tree/imx-ODT-example/tensorflow/lite/examples/label_image_odt.

Compile it with the Bazel for the "aarch64" platform:

```
$ bazel --output_base=/tensorflow/docker-build/ build --
config=monolithic --config=elinux_aarch64  -c opt //tensorflow/
lite/examples/label_image_odt:label_image_odt
```

For simplicity reasons, the "label_image_odt" binary is statically linked with the Flex Delegate.

To perform the training in the TFLite runtime, export the dataset to bitmap images:

```
import tensorflow as tf
import os
import pathlib
import PIL

def save_mnist(path, images, labels):
    p = pathlib.Path(path)
    p.mkdir(parents=True, exist_ok=True)
    # prep 10 dirs
    for l in range(10): (p / str(l)).mkdir(parents=True,
 exist_ok=True)
    for i, (im, l) in enumerate(zip(images, labels)):
        # print(i, im, l)
        dest = p / str(l) / f"{i}.bmp"
        im = im.reshape(28, 28)
        im = PIL.Image.fromarray(im, mode='L')
        with dest.open(mode='wb') as f: im.save(f)

PATH="./dataset/fashion_mnist/"

fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) =
 fashion_mnist.load_data()

save_mnist(os.path.join(PATH, "train"), train_images,
 train_labels)
save_mnist(os.path.join(PATH, "test"), test_images,
 test_labels)
```

Copy the required artefacts to the i.MX 8 Linux platform into the *~/odt_demo* folder:

- The "fashion_mnist" folder
- The "label_image_odt" binary
- The "odt_model_empty.tflite" model file

Run the demo:

```
$ cd odt_demo
$ ./label_image_odt --train_dataset ./fashion_mnist/train
 --test_dataset ./fashion_mnist/test/ -m ./odt-model-
empty.tflite --num_epochs 5 --batch_size 1000 --save_path ./
tmp/checkpoint.ckpt
```

Let's walk through the key part of the On-Device Training C++. The signatures allow a single model to support multiple entry points and outputs, usually connected with its own subgraph [https://www.tensorflow.org/lite/guide/signatures]. The signature can be invoked to perform the computation represented by its subgraph:

```
auto signatures = interpreter->signature_keys();
for(auto& s : signatures) {
    LOG(INFO) << "\t" << *s;
}
auto trIn = interpreter->signature_inputs("train");
for (auto& s : trIn) {
    LOG(INFO) << "\t\t"<<s.first << " : " << s.second;
}
auto trOut = interpreter->signature_outputs("train");
for (auto& s : trOut) {
    LOG(INFO) << "\t\t" << s.first << " : " << s.second;
}
```

The signatures and the subgraph's inputs and outputs are addressed by their names:

```
auto trainSig = interpreter->GetSignatureRunner("train");
trainSig->input_tensor("x")
```

To perform the learning on batches, resize the "train" subgraph to fit the batch size. This is achieved by resizing the "train" signature inputs and performing the (re)allocation of tensors. All the tensors' dimensions in the signature subgraph are adjusted:

```
trainSig->ResizeInputTensor(INPUT_X, {settings->batch_size,
 IMG_WIDTH, IMG_HEIGHT} );
trainSig->ResizeInputTensor(INPUT_Y, {settings->batch_size,
 10});
trainSig->AllocateTensors();
```

The training step (adjusting the weights based on particular batch of training data) is performed by the signature invocation:

```
trainSig->Invoke();
```

The original TFLite model does not change by the (re-)training process. The new weights are kept in the "tflite::interpreter" object in the memory. To preserve the training output, the weights must be stored to a checkpoint file. The model contains the "save" signature for these purposes:

```
auto saveSig = interpreter->GetSignatureRunner("save");
TfLiteTensor* saveSigInputTensor = saveSig-
>input_tensor(saveSig->input_names()[0]);
if (saveSig->AllocateTensors() != kTfLiteOk) {
```

```
    LOG(ERROR) << "Failed to allocate tensors for saveSig
 Runner";
    exit(-1);
}
DynamicBuffer buf;
buf.AddString(settings->save_path.c_str(), settings-
>save_path.size());
buf.WriteToTensor(saveSigInputTensor, nullptr);
saveSig->Invoke();
  LOG(INFO) << "Weights saved.";
```

6 Flex Delegate limitations

This chapter summarizes some Flex Delegate limitations.

TFLite runtime size with Flex Delegate

Reducing the TFLite binary size based on model requirements (as described at https://www.tensorflow.org/lite/guide/reduce_binary_size) is not supported for Linux platforms as of the time of writing this document (v2.8.0 release). These features are currently supported only for mobile platforms (Android and iOS).

The TensorFlow Flex Delegate for the Linux platform is built with all the supported TF operators. The stripped Flex Delegate library size is about 120 MB.

CPU-only support for TensorFlow Operators

The Flex Delegate operators are not supported on i.MX 8 hardware accelerators. The TF Operators fall back to the CPU. The acceleration of the supported TFLite Operators in the model is not impacted. The model can freely combine TFLite and TF Operators. The supported TFLite operators of the model are accelerated.

On-Device Training example is supported on CPU only

The On-Device Training is supported on CPU only for both the training and the inference task, as shown at https://blog.tensorflow.org/2021/11/on-device-training-in-tensorflow-lite.html.

Running the training on a CPU and the inference on a hardware accelerator (in floating point) is not yet supported. The following limitations apply:

1. The hardware accelerators on i.MX 8 platforms do not support models with dynamically-sized tensors. This means that the “infer” signature and the related subgraph must be composed of fixed-sized tensors so that it can be accelerated on the GPU.
2. The “train” signature and the corresponding subgraph must use dynamically-sized tensors, because the batch size can be one of the hyperparameters influencing the training process.
3. The TFLite Converter does not support combining the fixed-size and dynamically-sized tensors on the level of various signatures.

Another limitation is the quantization support, where the TFLite Converter fails to produce a valid quantized TFLite model.

7 Revision history

Table 1. Revision history

Revision number	Date	Substantive changes
0	16 August 2022	Initial release

8 Legal information

8.1 Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

8.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Evaluation products — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

8.3 Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents

For more information, please visit: http://www.nxp.com

Date of release: 16 August 2022
Document identifier: Enabling TensorFlow Ops in TensorFlow Lite Runtime