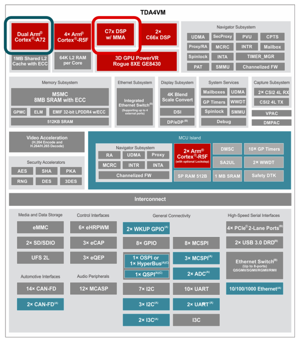
TI TVM User’s Guide

v8.6

Online HTML version available here

CONTENTS

Texas Instrument's fork of the Apache Tensor Virtual Machine (*TVM*) enables support for the TDA4 family of processors. These processors use C7x DSPs and Matrix Multiplication Accelerators (*MMA*) to accelerate *inference*-making by machine learning models. For additional information about TDA4x processors and TI's Edge AI ecosystem, refer to the Edge AI page on ti.com.



The TI Deep Learning library (*TIDL*) contains highly optimized implementations of common layers on C7x/MMA. If a model contains layers that are not implemented by TIDL, TVM can be used to run these layers on Arm (Cortex-A) cores or the C7x DSP core.

This user's guide documents the TI TVM Compiler and its usage.

CHAPTER

ONE

GETTING STARTED

Note: The TI Edge AI Cloud is a free online service used to evaluate and benchmark accelerated deep learning inference on *TDA4 family* processors. TVM is one of the frameworks supported by the TI Edge AI Cloud. If you are interested in evaluating TVM without having to install any software or purchase an evaluation board, you can use this service.

This chapter lists the steps required to install the TVM framework, compile models, and run inference on a TI evaluation board (EVM).

1.1 Installation Instructions

Follow the instructions linked below to install the TVM compiler framework on a Linux PC. The instructions also cover how to install EVM-specific dependencies required for inference.

EVM	Instructions
J721E EVM	J721E link
TDA4VM processor starter kit	TDA4VM SK link

1.2 Model Basics

Machine learning models usually take input data in tensor (multi-dimensional array) format and produce output data in tensor format. Before compiling and inferencing a model using TVM, it is recommended that you understand the basic requirements for your model:

- model semantics, including the meanings of input and output
- input name, shape
- output shape

- sample of input data
- expected output data corresponding to the sample input data

Experimenting with pre-processing input data, running inference, and post-processing input data can be accomplished even without a TVM setup and TI devices. For example, if you have an ONNX model, you can write a Python script like the following to import the `onnxruntime` package, pre-process input data, run inferencing, post-process output data, and verify the results.

```
import onnxruntime
model_file = ...
input_name = ...
input_data = ...
sess = onnxruntime.InferenceSession(model_file)
output = sess.run([], {input_name : input_data})
```

In the topics that follow, we assume you understand the basics of your model, so we focus on TVM-related aspects. Although we show data pre-processing, model input names and shapes, and data post-processing in our examples, the details about these steps are likely to be different for your model, so it is your responsibility to set them correctly.

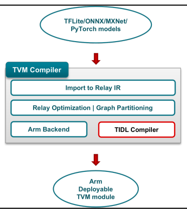
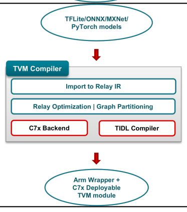
1.3 Compiling Models

TI TVM supports two options for compiling models with TIDL offload. The distinction is based on where layers that are not supported by TIDL are executed during inference:

1. Executing unsupported layers on Arm (See flow in Figure 1).
2. Executing unsupported layers on C7x (See flow in Figure 2). This option frees up the Arm device to run other aspects of the application. It can also improve overall inference performance by minimizing communication across the Arm and C7x.

The following figures show components added by TIDL and the TI TVM fork outlined in red.

Table 1.1: Model compilation with unsupported layers

Figure 1: Mapped to Arm	 TFLite/ONNX/MXNet/ PyTorch models TVM Compiler Import to Relay IR Relay Optimization | Graph Partitioning Arm Backend TIDL Compiler Arm Deployable TVM module
Figure 2: Mapped to C7x	 TFLite/ONNX/MXNet/ PyTorch models TVM Compiler Import to Relay IR Relay Optimization | Graph Partitioning C7x Backend TIDL Compiler Arm Wrapper + C7x Deployable TVM module

See *Compilation Explained* for further details about TI TVM compilation.

1.3.1 Compiling for TIDL Offload

TVM compilation is typically performed using a Python compilation script. Example scripts are provided in the TI edgeai-tidl-tools and TI TVM fork Git repositories. You can use these examples as templates to modify for your own use cases.

An overview of TI Open Source Runtime and compilation options are provided in the TI edgeai-tidl-tools overview.

The following Python functions are used in the examples provided by the TI TVM fork.

compile_model

The `compile_model` function encapsulates the steps required to compile a model with TIDL offload.

`relay.ti_tests.compile_model.`**`compile_model`**(*model_name: str*, *platform: str*, *compile_for_device: bool*, *enable_tidl_offload: bool*, *enable_c7x_codegen: bool*, *batch_size: int = 0*)

Compile a model based on the parameters specified.

Parameters

- **`model_name`** – name of the model as specified in models.py (E.g. mv2_onnx).
- **`platform`** – in ["J7", "J721S2"", "AM62A"]
- **`compile_for_device`** – True => Compile module for inference on device (aarch64). False => Compile module for inference on host (x86).
- **`enable_tidl_offload`** – Set to True to enable TIDL offload.
- **`enable_c7x_codegen`** – True => Enable c7x code generation for layers not offloaded to TIDL. i.e. entire network runs on the C7x. False => Enable Arm code generation for layers not offloaded to TIDL. i.e. Unsupported layers are run on Arm (aarch64).
- **`batch_size`** – 0: use the batch size that comes with the model otherwise: override the default batch size

Return type

True for success, False for failure.

Listing 1.1 shows the key functions called from *compile_model*.

Listing 1.1: Compiling a model with TIDL offload

```
  os.environ["TIDL_RELAY_MAX_BATCH_SIZE"] = str(batch_size)

# Obtain model and convert to Relay
mod, params = get_relay_model(model_name, batch_size)

# Get inputs to use for calibraton (required for TIDL offload)
calibration_input_list = get_calib_inputs(model_name, batch_
↪size)

# Generate a name for the artifacts folder based on the model␣
↪and other parameters
artifacts_folder = get_artifacts_folder(model_name, platform,␣
↪compile_for_device,
                                        enable_tidl_offload,␣
↪enable_c7x_codegen,
                                        batch_size)

# Compile the model using TVM and place the output in the␣
↪artifacts_folder
```

(continues on next page)

(continued from previous page)

```
  result = compile.compile_relay(mod, params, calibration_input_
↪list, platform, compile_for_device,
```

After a successful compile, the artifacts required to deploy the model are stored in the *artifacts_folder*.

compile_relay

The `compile_relay` function uses the TIDLCompiler class to partition the relay graph for offload subgraphs to TIDL.

`tvm.contrib.tidl.compile.`**`compile_relay`**(*mod: IRModule*, *params: Dict[str, NDArray]*, *calibration_input_list: List[Dict[str, NDArray]]*, *platform: str*, *compile_for_device: bool*, *enable_tidl_offload: bool*, *enable_c7x_codegen: bool*, *artifacts_folder: str*, *tidl_tensor_bits: int = 8*) → bool

Compile Relay IR module based on the parameters specified

Parameters

- **`mod`** – Input Relay IR module.
- **`params`** – The parameter dict used by Relay.
- **`platform`** – in ["J7", "J721S2", "AM62A"]
- **`calibration_input_list`** – A dictionary where the key is input name and the value is input tensor.
- **`compile_for_device`** – True => Compile module for inference on device (aarch64). False => Compile module for inference on host (x86).
- **`enable_tidl_offload`** – Set to True to enable TIDL offload.
- **`enable_c7x_codegen`** – True => Enable c7x code generation for layers not offloaded to TIDL. i.e. entire network runs on the C7x. False => Enable Arm code generation for layers not offloaded to TIDL. Unsupported layers are run on Arm (aarch64).
- **`tidl_tensor_bits`** – Number of bits used to represent TIDL tensors and weights.

Return type

True for success, False for failure.

1.4 Running Inference

TVM inference can be run using a Python script or a C/C++ application. Examples are provided in the TI edgeai-tidl-tools and TI TVM fork repositories on Github. You can use these examples as templates to modify for your own use cases.

The default runtime setup used by edgeai-tidl is DLR (Deep Learning Runtime from Amazon AWS). For the purpose of running TVM compiled models with the DLR runtime, DLR is simply a wrapper around the TVM runtime.

See *Inference Explained* for further details about TVM inference.

The following Python functions and C++ code are used in the examples provided in the TI TVM fork.

1.4.1 Python

The `run_model` function shows how to run inference with the DLR or the TVM Runtime.

`relay.ti_tests.infer_model.`**`run_model`**(*artifacts_folder: str*, *input_dict*, *use_dlr: bool*)

Run model with given input using DLR or TVM Runtime.

Parameters

- **`artifacts_folder`** – Folder containing compilation artifacts.
- **`input_dict`** – Dictionary of input name (str) to input data (numpy.ndarray).
- **`use_dlr`** – If True, use DLR. If False, use the TVM runtime directly.

Returns

results

Return type

List of result tensors.

1.4.2 C++

Listing 1.2 shows how to use the C++ and TVM Runtime APIs to (relay_mul example in TI TVM repo)

- Load compilation artifacts (shared library, parameter file, and JSON representation of the network graph)
- Create a TVM Graph Executor

- Set up inputs to the Graph Executor
- Run inference on the Graph Executor
- Extract outputs from the Graph Executor

Listing 1.2: Running inference using C++ and the TVM Runtime

```cpp
void DeployGraphExecutor() {
  const std::string artifacts_folder("artifacts_relay_mul_c7x_target/");

  // load in the library
  DLDevice dev{kDLCPU, 0};
  tvm::runtime::Module loaded_lib = tvm::runtime::Module::LoadFromFile(artifacts_folder + "deploy_lib.so");

    // Load JSON
  std::ifstream loaded_json(artifacts_folder + "deploy_graph.json");
  std::string json_data((std::istreambuf_iterator<char>(loaded_json)), std::istreambuf_iterator<char>());
  loaded_json.close();

  // Load params from file
  std::ifstream loaded_params(artifacts_folder + "deploy_param.params", std::ios::binary);
  std::string params_data((std::istreambuf_iterator<char>(loaded_params)), std::istreambuf_iterator<char>());
  loaded_params.close();
  TVMByteArray params_arr;
  params_arr.data = params_data.c_str();
  params_arr.size = params_data.length();

  LOG(INFO) << "Creating graph executor...";
  // Create the graph executor module
  int device_type = dev.device_type; // Need an int, the DLDeviceType enum
                                     // results in an ambiguity for TVMArgsSetter.

  tvm::runtime::Module mod =
    (*tvm::runtime::Registry::Get("tvm.graph_executor.create
```

(continues on next page)

(continued from previous page)

```
↪"))(json_data,
                                                                                    ␣
↪loaded_lib,
                                                                                    ␣
↪device_type,
                                                                                    ␣
↪dev.device_id);

  // Load params into Graph Executor
  LOG(INFO) << "Loading params ...";
  tvm::runtime::PackedFunc load_params = mod.GetFunction("load_
↪params");
  load_params(params_arr);

  tvm::runtime::PackedFunc set_input           = mod.GetFunction(
↪"set_input");
  tvm::runtime::PackedFunc get_output          = mod.GetFunction(
↪"get_output");
  tvm::runtime::PackedFunc run                 = mod.GetFunction(
↪"run");

  LOG(INFO) << "Initializing inputs ...";
  auto f32 = tvm::runtime::DataType::Float(32);
  tvm::runtime::NDArray a = tvm::runtime::NDArray::Empty({672,␣
↪14, 14}, f32, dev);
  tvm::runtime::NDArray b = tvm::runtime::NDArray::Empty({672, 1,
↪ 1},   f32, dev);
  tvm::runtime::NDArray c = tvm::runtime::NDArray::Empty({672,␣
↪14, 14}, f32, dev);

  for (int i = 0; i < 672; ++i)
    static_cast<float*>(b->data)[i] = 4;

  for (int i = 0; i < 672*14*14; ++i)
    static_cast<float*>(a->data)[i] = i;

  set_input("a", a);
  set_input("b", b);

  // run the code
  LOG(INFO) << "Running ...";
  run();
```

(continues on next page)

(continued from previous page)

```

  // get the output
  get_output(0, c);

  for (int i = 0; i < 672*14*14; ++i)
    ICHECK_EQ(static_cast<float*>(c->data)[i], i * 4);

  LOG(INFO) << "Pass";
}
```

Listing 1.3 shows how to use the C++ and DLR Runtime APIs to run inference (test_dlr_cpp example in TI TVM repo).

Listing 1.3: Running inference using C++ and the DLR Runtime

```
  // Step 1: Create DLR model from compiled model artifacts
  DLRModelHandle model;
  const char *model_path = "../artifacts/mv2_onnx_J7_target_tidl_
↪c7x";
  if (argc > 1)  model_path = argv[1];
  status = CreateDLRModel(&model, model_path, 1, 0);
  check_status(status, &model, "CreateDLRModel");

  // Step 2: Set input tensor
  const char *model_input_name = "data";
  int64_t shape[4] = {1, 3, 224, 224};
  DLTensor in_tensor = { (void*) airshow,
                         {kDLCPU, 0},
                         4,
                         {kDLFloat, 32, 1},
                         shape,
                         NULL,
                         0
                       };
  status = SetDLRInputTensorZeroCopy(&model, model_input_name, &
↪in_tensor);
  check_status(status, &model, "SetDLRInputTensorZeroCopy");

  // Step 3: Run inference
  status = RunDLRModel(&model);
  check_status(status, &model, "RunDLRModel");

```

(continues on next page)

(continued from previous page)

```
  // Step 4: Get output
  float *probs;
  status = GetDLROutputPtr(&model, 0, (const void**) &probs);
  check_status(status, &model, "GetDLROutputPtr");

  int64_t size;
  int dim;
  int64_t out_shape[8];
  char* type_name;
  status = GetDLROutputSizeDim(&model, 0, &size, &dim);
  check_status(status, &model, "GetDLROutputSizeDim");
  status = GetDLROutputShape(&model, 0, out_shape);
  check_status(status, &model, "GetDLROutputShape");
  status = GetDLROutputType(&model, 0, (const char**) &type_
↪name);
  check_status(status, &model, "GetDLROutputType");

  printf("\nModel output 0 size=%" PRId64 ", dim=%d\n", size,␣
↪dim);
  printf("Model output 0 shape: ");
  for (int i = 0; i < dim; i++)  printf("%" PRId64 "x", out_
↪shape[i]);
  printf("\n");
  printf("Model output 0 type: %s\n", type_name);

  // Step 5: Interpret results
  int imax = 0;
  for (int i = 0; i < size; i++)
    if (probs[i] > probs[imax])
      imax = i;
  printf("Top 1 index = %d, probability = %f\n\n", imax,␣
↪probs[imax]);

  // Step 6: Tear down
  status = DeleteDLRModel(&model);
  check_status(status, NULL, "DeleteDLRModel");
```

CHAPTER

TWO

COMPILATION EXPLAINED

This section explains TI TVM compilation in more detail. An introduction to this topic is provided in *Compiling Models*.

2.1 Environment Setup

If they have not already been set up by Edgeai, the following three environment variables are required before running the compilation script.

- `TIDL_TOOLS_PATH`: Point to installed /processor_sdk_rtos/tidl_release/tidl_tools.
- `ARM64_GCC_PATH`: Point to installed /gcc-arm-9.2-2019.12-x86_64-aarch64-none-linux-gnu.
- `CGT7X_ROOT`: Point to installed TI C7x C/C++ compiler 3.1.0.LTS in the Processor SDK package or from ti.com.

2.2 Frontends

TVM can accept machine learning models in many formats, including Tensorflow/TFLite, Keras, Core ML, MXNet, ONNX, and PyTorch. As the first step of compilation, these formats are all imported into TVM's internal common representation, *Relay IR*, using different frontends in TVM.

TI TVM provides additional examples beyond those provided by *Apache TVM* in `tests/python/relay/ti_tests/models.py` to show how to import machine learning models in different network formats into Relay IR.

2.3 Calibration Data

After partitioning layers into *subgraphs* that can be offloaded to TIDL, these subgraphs need to be imported to TIDL. Because TIDL runs inference with quantized fixed-point values, the TIDL import process requires calibration data so that each layer's dynamic range can be estimated and the scaling factor for converting between floating point and fixed point can be computed.

You only need to provide calibration data for the whole model. The TVM+TIDL compilation flow automatically obtains the corresponding tensor values at the TIDL subgraph boundaries and feeds those values into the TIDL import process for calibration. The calibration data you provide should represent typical input for the model.

2.4 Artifacts

2.4.1 Deployable module

After a successful TVM+TIDL compilation, a deployable module consists of 3 files that are saved in the `<artifacts_folder>`.

- **.json:** A JSON file describing the compiled graph through information about nodes, allocation, etc.
- **.so:** The shared library containing code to run nodes in the compiled graph. This is a fat binary. Imported TIDL subgraph artifacts and generated C7x code are embedded in this fat binary.
- **.params:** Contains weights associated with the nodes in the compiled graph.

At inference time, DLR/TVM runtime read these 3 files and create a runtime instance to run inference.

Hint: During development, you may export the x86_64 Linux filesystem where you run compilation, and mount the filesystem on your EVM so that you do not need to copy the deployable module.

For deploying onto your EVM, the deployable module is all that is needed. During compilation, TI TVM saves intermediate results in the `<artifacts_folder>/tempDir` directory. These results may help you understand and debug the compilation. The following subsections describe some of the intermediate artifacts.

2.4.2 Relay graphs

- **relay_graph.orig.txt:** The original relay graph from the TVM frontend.
- **relay_graph.prepared.txt:** The relay graph after transformations that prepare for TIDL offload.
- **relay_graph.annotated.txt:** The relay graph annotated for TIDL offload.
- **relay_graph.partitioned.txt:** The relay graph partitioned for TIDL offload.
- **relay_graph.import.txt:** The relay graph used to import into TIDL.
- **relay_graph.boundary.txt:** The relay graph used to obtain calibration data at TIDL subgraph boundaries.
- **relay_graph.optimized.txt:** The optimized relay graph for code generation.
- **relay_graph.wrapper.txt:** The wrapper relay graph on the Arm side for dispatching the graph to C7x.

For example, you may examine `relay_graph.import.txt` to see how many TIDL subgraphs are created and which layers are not offloaded to TIDL.

2.4.3 Imported TIDL artifacts

TIDL subgraphs are imported into TIDL artifacts in TIDL-specific formats. These artifacts are embedded into the `.so` fat binary in the deployable module. The DLR/TVM runtime retrieves TIDL artifacts and invokes the TIDL runtime at inference time.

- **relay.gv.svg:** A graphical view of the whole network and where the TIDL subgraphs are located.
- **subgraph<n>_net.bin.svg:** A graphical view of TIDL subgraphs.

2.4.4 Generated C7x code

When `c7x_codegen` is set to 1 in the compilation script, TI TVM generates C7x code for layers not offloaded to TIDL. This C7x code is compiled and embedded into the `.so` fat binary in the deployable module. The DLR/TVM runtime retrieves the C7x code and dispatches it to the C7x for execution.

- **model_<n>.c:** Contains generated code either to run a TIDL subgraph or non-TIDL layers.

2.5 Debugging Compilation

TI TVM uses the TIDL_RELAY_IMPORT_DEBUG environment variable to help debug the TVM+TIDL compilation flow. The available settings are as follows.

2.5.1 TIDL_RELAY_IMPORT_DEBUG=1

When set to 1, verbose output at the terminal provides more information about the TIDL import. This includes whether a node is supported by TIDL, relay node to TIDL node conversion, imported TIDL subgraphs, optimized TIDL subgraphs, and calibration processes. For example:

```
RelayImportDebug: In TIDL_relayAllowNode:
RelayImportDebug:   name: nn.conv2d
RelayImportDebug: In TIDL_relayAllowNode:
RelayImportDebug:   name: nn.batch_norm
```

2.5.2 TIDL_RELAY_IMPORT_DEBUG=2, 3

When set to 2 or 3, verbose information about importing TIDL subgraphs is provided in addition to the information provided when the setting is 1.

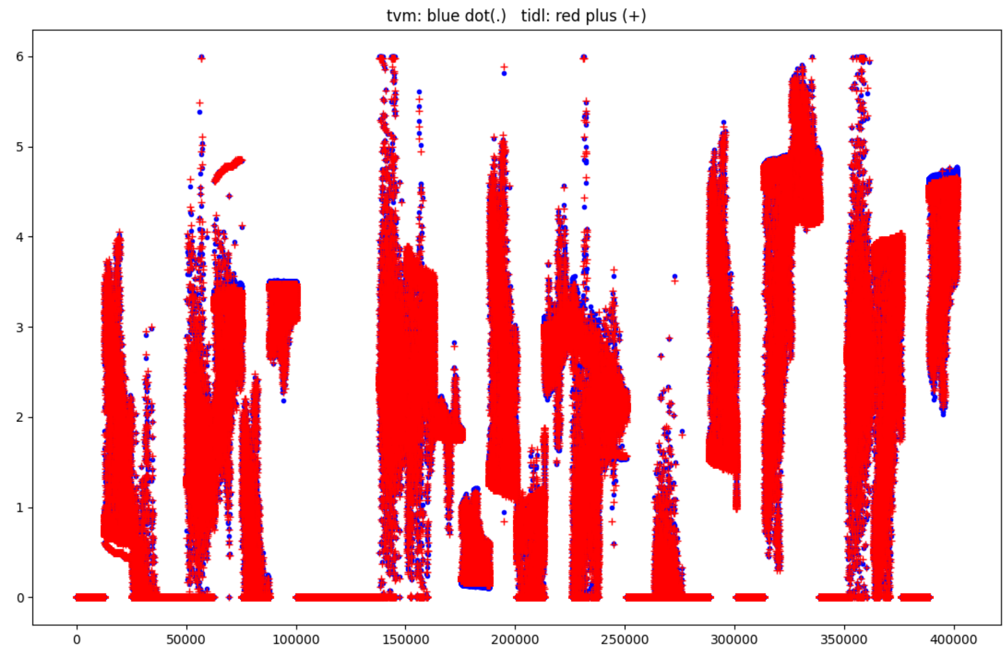
2.5.3 TIDL_RELAY_IMPORT_DEBUG=4

When set to 4, the TIDL import generates output for each TIDL layer in the imported TIDL subgraph using calibration inputs. This output is stored in the auto-generated `tempDir/tidl_import_subgraph<subgraph_id>.txt<layer_id><dimensions>_float.bin` files.

The compilation also generates corresponding output from running the original model on x86_64 hosts using TVM code generation for x86_64. This is stored in the `tempDir/tidl_<subgraph_id>_layer<layer_id>.npy` files.

A script, `python/tvm/contrib/tidl/compare_tensors.py` is provided to compare the two results with a graphical view. You can run the script as follows:

```
# in tests/python/relay/ti_tests/
TIDL_RELAY_IMPORT_DEBUG=4 python3 ./compile_model.py mv1_tf --
↪target --tidl --c7x
# compare_tensors.py <artifacts_folder> <subgraph_id> <layer_id>
python3 $TVM_HOME/python/tvm/contrib/tidl/compare_tensors.py␣
↪artifacts/mv1_tf_J7_target_tidl_c7x 0 2
```


tvm: blue dot(.) tidl: red plus (+)
0
1
2
3
4
5
6
0
50000
100000
150000
200000
250000
300000
350000
400000

CHAPTER

THREE

INFERENCE EXPLAINED

This section explains more details about running and debugging TVM inference. An introduction to this topic is provided in *Running Inference*.

There are two inference scenarios:

- Running TIDL unsupported layers on Arm
- Running TIDL unsupported layers on C7x

These correspond to compiling the model with `c7x_codegen=0` or `c7x_codegen=1`. The same inference script or application can be used for both scenarios.

3.1 Running Unsupported Layers on Arm

When running unsupported layers on Arm, TI TVM graph runtime on Arm looks at each node in the graph (.json).

- If it is a TIDL subgraph, it is dispatched to C7x for execution (via OpenVX).
- If it is a non-TIDL node, it is executed with TVM-generated Arm code.

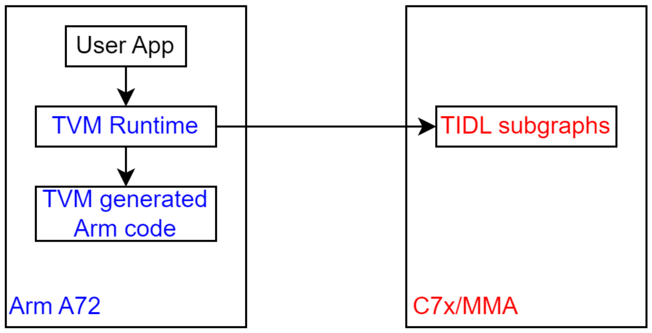


Fig. 3.1: Model inference with unsupported layers mapped to Arm

3.2 Running Unsupported Layers on C7x

When running unsupported layers on C7x, TI TVM graph runtime on Arm looks at the single node in the wrapper graph (.json), 'tidl_tvm_0', and dispatches the whole graph to C7x (via OpenVX).

TI TVM graph runtime on C7x looks at each node in the graph.

- If it is a TIDL subgraph, it is executed with the TIDL library.
- If it is a non-TIDL node, it is executed with TVM-generated C7x code.

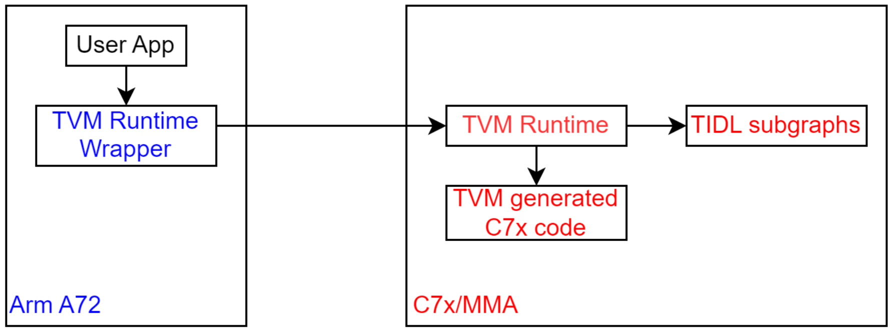


Fig. 3.2: Model inference with unsupported layers mapped to C7x

3.3 Debugging Inference

TIDL uses the TIDL_RT_DEBUG environment variable to help debug the TVM+TIDL inference flow.

3.3.1 Vision apps "printf" terminal

First, you need to open a "printf" terminal, where debug "printf" output from C7x core is shown.

Open a terminal on your EVM, and run the following commands.

```
# if using EdgeAI Start Kit SDK
root@tda4vm-sk:~# /opt/vx_app_arm_remote_log.out

# if using J721E SDK
root@j7-evm:~# cd /opt/vision_apps/
root@j7-evm:/opt/vision_apps# source ./vision_apps_init.sh
```

Then, run the inference in a different terminal.

3.3.2 Debugging TIDL subgraphs

The available settings for the TIDL_RT_DEBUG environment variable are as follows.

TIDL_RT_DEBUG=1

When set to 1, TIDL subgraph performance information is printed out during inference, either on the "printf" terminal or on the terminal where inference is running. For example:

```
[C7x_1 ] 1851913.287814 s:  Layer,   Layer Cycles,
↪kernelOnlyCycles, coreLoopCycles,LayerSetupCycles,
↪dmaPipeupCycles, dmaPipeDownCycles, PrefetchCycles,
↪copyKerCoeffCycles,LayerDeinitCycles,LastBlockCycles,␣
↪paddingTrigger,    paddingWait,LayerWithoutPad,LayerHandleCopy,
↪   BackupCycles,  RestoreCycles,
[C7x_1 ] 1851913.287889 s:      0,          201247,               ␣
↪171496,          173177,            1021,            9371,         ␣
↪        20,                0,                  0,             375,␣
↪         41487,            6392,              51,          191300,
↪               0,               0,                0,
[C7x_1 ] 1851913.287956 s:      1,           44208,                ␣
↪17603,          18221,             4170,           2679,           ␣
↪       18,                0,                  0,             662, ␣
↪        17603,            7720,             454,           33530,␣
↪          2096,               0,                0,
... ... ...
```

TIDL_RT_DEBUG=2, 3

When set to 2 or 3, more verbose information about TIDL subgraphs is provided in addition to the information provided when the setting is 1. For example:

```
[C7x_1 ] 1852081.872134 s: Alg Alloc for Layer # -    0
[C7x_1 ] 1852081.872160 s: Alg Alloc for Layer # -    1
... ... ...
[C7x_1 ] 1852081.873059 s: TIDL Memory requirement
[C7x_1 ] 1852081.873087 s: MemRecNum , Space     , Attribute ,  ␣
↪ SizeinBytes
[C7x_1 ] 1852081.873117 s:  0         , DDR       , Persistent, ␣
↪  15208
[C7x_1 ] 1852081.873145 s:  1         , DDR       , Persistent, ␣
↪  136
```

(continues on next page)

(continued from previous page)

```
... ... ...
[C7x_1 ] 1852081.874400 s: Alg Init for Layer # -    2 out of ␣
↪32
[C7x_1 ] 1852081.874470 s: Alg Init for Layer # -    3 out of ␣
↪32
... ... ...
[C7x_1 ] 1852081.911120 s: Starting Layer # -    1
[C7x_1 ] 1852081.911145 s: Processing Layer # -    1
[C7x_1 ] 1852081.911375 s: End of Layer # -    1 with outPtrs[0]␣
↪= 7002001e
[C7x_1 ] 1852081.911400 s: Starting Layer # -    2
[C7x_1 ] 1852081.911422 s: Processing Layer # -    2
[C7x_1 ] 1852081.911493 s: End of Layer # -    2 with outPtrs[0]␣
↪= 7004550e
... ...
```

TIDL_RT_DEBUG=4, 5

Setting this environment variable to 4 or 5 is supported only when running TIDL unsupported layers on Arm. When set to 4 or 5, tensor output from each layer in the TIDL subgraph is dumped into the Arm Linux file system, with filenames `tidl_trace_subgraph_<subgraph_id>_<layer_id>_<tensor_shape>.y` for raw data and `tidl_trace_subgraph_<subgraph_id>_<layer_id>_<tensor_shape>_float.bin` for converted float data.

3.3.3 Debugging TVM nodes

The TVM_RT_DEBUG, TVM_RT_TRACE_NODE, TVM_RT_TRACE_SIZE, and TVM_TRACE_NODE environment variables can be used as follows to aid in debugging TVM nodes.

TVM_RT_DEBUG=1

When set to 1, TVM runtime on Arm collects performance statistics for each node in the graph and saves into the Arm Linux file system, with the filename `tvm_arm.trace`. You can use the `python/tvm/contrib/tidl/dump_tvm_trace.py` script to dump the details.

When TIDL unsupported layers are run on Arm, the `tvm_arm.trace` file includes the execution time for TIDL nodes and layers running on Arm. For example:

```
# python3 $TVM_HOME/python/tvm/contrib/tidl/dump_tvm_trace.py␣
↪tvm_arm.trace
Trace size: 633 version: 0x20220728 device: J7 core: Arm
node 1: tidl_8  1520.9 microseconds
node 2: tvmgen_default_fused_multiply  436.81 microseconds
node 3: tidl_7  993.62 microseconds
node 4: tvmgen_default_fused_multiply_1  149.11 microseconds
node 5: tidl_6  1162.075 microseconds
node 6: tvmgen_default_fused_multiply_11  176.68 microseconds
node 7: tidl_5  2233.29 microseconds
node 8: tvmgen_default_fused_multiply_2  120.095 microseconds
node 9: tidl_4  1503.91 microseconds
node 10: tvmgen_default_fused_multiply_3  163.83 microseconds
node 11: tidl_3  1381.615 microseconds
node 12: tvmgen_default_fused_multiply_4  61.94 microseconds
node 13: tidl_2  1195.17 microseconds
node 14: tvmgen_default_fused_multiply_5  70.53 microseconds
node 15: tidl_1  1311.9 microseconds
node 16: tvmgen_default_fused_multiply_51  71.32 microseconds
node 17: tidl_0  1279.285 microseconds
node 4294967295: Graph  13856.93 microseconds
```

When TIDL unsupported layers are running on C7x, `tvm_arm.trace` includes only the execution time for a single node representing the whole graph. For example:

```
# python3 $TVM_HOME/python/tvm/contrib/tidl/dump_tvm_trace.py␣
↪tvm_arm.trace
Trace size: 69 version: 0x20220728 device: J7 core: Arm
node 1: tidl_tvm_0  9126.275 microseconds
node 4294967295: Graph  9129.92 microseconds
```

TVM_RT_DEBUG=2

When set to 2, in addition to the information provided when the TVM_RT_DEBUG setting is 1, TVM runtime on C7x also collects performance statistics for each node in the graph on the C7x and saves them to the Arm Linux file system, with the filename `tvm_c7x.trace`. You can use the `python/tvm/contrib/tidl/dump_tvm_trace.py` script to dump the details. For example:

```
# python3 $TVM_HOME/python/tvm/contrib/tidl/dump_tvm_trace.py␣
↪tvm_c7x.trace
Trace size: 631 version: 0x20220728 device: J7 core: C7x
node 1: tidl_8  909.002 microseconds
```

(continues on next page)

(continued from previous page)

```
node 2: tvmgen_default_fused_multiply  62.201 microseconds
node 3: tidl_7  378.24 microseconds
node 4: tvmgen_default_fused_multiply_1  83.055 microseconds
node 5: tidl_6  445.359 microseconds
node 6: tvmgen_default_fused_multiply_1  83.433 microseconds
node 7: tidl_5  1590.525 microseconds
node 8: tvmgen_default_fused_multiply_2  82.979 microseconds
node 9: tidl_4  809.138 microseconds
node 10: tvmgen_default_fused_multiply_3  110.202 microseconds
node 11: tidl_3  802.037 microseconds
node 12: tvmgen_default_fused_multiply_4  46.95 microseconds
node 13: tidl_2  848.143 microseconds
node 14: tvmgen_default_fused_multiply_5  55.662 microseconds
node 15: tidl_1  908.129 microseconds
node 16: tvmgen_default_fused_multiply_5  54.628 microseconds
node 17: tidl_0  990.953 microseconds
node 4294967295: Graph  8283.967 microseconds
```

The `tvm_c7x.trace` output is available only if the model was compiled with `c7x_codegen=1`.

TVM_RT_DEBUG=3

When set to 3, in addition to the information provided when the setting is 1 or 2, TVM runtime on Arm and C7x collects output tensor statistics for each layer and saves it to the trace files. Collected statistics include minimum, maximum, sum, and the sum of the first half of the tensor.

Please ignore performance numbers obtained in this mode; there is overhead when collecting tensor statistics.

```
# python3 $TVM_HOME/python/tvm/contrib/tidl/dump_tvm_trace.py␣
↪tvm_arm.trace
Trace size: 1833 version: 0x20220728 device: J7 core: Arm
node 1: tidl_8  1527.845 microseconds
  output 0: ndim=4 type_code=2 elem_bytes=4 num_elements=56448
           min=0.0 max=21.715360641479492 sum=234286.71875 fh_
↪sum=110520.9296875
  output 1: ndim=4 type_code=2 elem_bytes=4 num_elements=72
           min=0.73150634765625 max=0.999969482421875 sum=68.
↪35482788085938 fh_sum=34.87158203125
node 2: tvmgen_default_fused_multiply  416.19 microseconds
  output 0: ndim=4 type_code=2 elem_bytes=4 num_elements=56448
           min=0.0 max=21.439014434814453 sum=226814.515625 fh_
```

(continues on next page)

(continued from previous page)

```
↪sum=106811.2890625
... ... ...
```

```
# python3 $TVM_HOME/python/tvm/contrib/tidl/dump_tvm_trace.py␣
↪tvm_c7x.trace
Trace size: 1831 version: 0x20220728 device: J7 core: C7x
node 1: tidl_8  934.738 microseconds
  output 0: ndim=4 type_code=2 elem_bytes=4 num_elements=56448
            min=0.0 max=21.715360641479492 sum=234286.71875 fh_
↪sum=110520.9296875
  output 1: ndim=4 type_code=2 elem_bytes=4 num_elements=72
            min=0.73150634765625 max=0.999969482421875 sum=68.
↪35482788085938 fh_sum=34.87158203125
node 2: tvmgen_default_fused_multiply  64.704 microseconds
  output 0: ndim=4 type_code=2 elem_bytes=4 num_elements=56448
            min=0.0 max=21.439014434814453 sum=226814.515625 fh_
↪sum=106811.2890625
... ... ...
```

TVM_RT_DEBUG=4

When set to 4, TVM runtime on Arm and C7x prints out information about each node when each node is being executed, either on the terminal where inference is run or the "printf" terminal. This mode can be helpful for debugging the model's execution.

TVM_RT_TRACE_NODE=<node_id> TVM_RT_DEBUG=3, 4

When these settings are used, TVM runtime on Arm and C7x also saves the tensor outputs of the specified node into the trace file. The `dump_trace.py` script saves the tensor outputs into the Arm Linux file system as numpy files, with the filename `n<node_id>_o<output_id>.npy`. For example, see the `tensor values saved in` messages in the following example.

Tensor outputs are saved as float values in the trace.

```
# TVM_RT_TRACE_NODE=1 TVM_RT_DEBUG=4 python3 ./infer_model.py ...
↪ ...
... ...
# python3 $TVM_HOME/python/tvm/contrib/tidl/dump_tvm_trace.py␣
↪tvm_c7x.trace
Trace size: 227911 version: 0x20220728 device: J7 core: C7x
node 1: tidl_8  912.196 microseconds
```

(continues on next page)

(continued from previous page)

```
  output 0: ndim=4 type_code=2 elem_bytes=4 num_elements=56448
           min=0.0 max=21.715360641479492 sum=234286.71875 fh_
↪sum=110520.9296875
           tensor values saved in n1_o0.npy
  output 1: ndim=4 type_code=2 elem_bytes=4 num_elements=72
           min=0.73150634765625 max=0.999969482421875 sum=68.
↪35482788085938 fh_sum=34.87158203125
           tensor values saved in n1_o1.npy
node 2: tvmgen_default_fused_multiply  65.955 microseconds
  output 0: ndim=4 type_code=2 elem_bytes=4 num_elements=56448
           min=0.0 max=21.439014434814453 sum=226814.515625 fh_
↪sum=106811.2890625
node 3: tidl_7  388.9 microseconds
```

TVM_RT_TRACE_SIZE=<new_size> TVM_TRACE_NODE=<node_id> TVM_RT_DEBUG=3, 4

If you see a `Not enough trace memory for dumping output <node_id>` message in the verbose output for a node, you can use `TVM_RT_TRACE_SIZE` to increase the size of the trace buffer that stores the trace. The default is 2*1024*1024 (2MB) bytes.

CHAPTER

FOUR

RECOMMENDED DEVELOPMENT FLOW

This section describes the recommended development flow for using TVM to compile and infer a model.

4.1 Step 1: Model Selection

You may have already developed and trained a model. But if you are using a model downloaded from public domain, we recommend you look at TI EdgeAI ModelZoo first. TI EdgeAI ModelZoo contains models that have been tweaked and optimized for inference speed on TI SoCs.

4.2 Step 2: Compile with c7x_codegen=0

Adapt the example compilation scripts in TI edgeai-tidl-tools for use with your model. Additional examples are provided in the TVM Git repository.

See the *Compilation Explained* section for more about the compilation process and compiled artifacts.

When troubleshooting and optimizing, check the following:

- Have all layers been offloaded to TIDL?
- If not, which layers are not offloaded?
- How many TIDL subgraphs are there?

4.3 Step 3: Inference and Performance Profiling

Adapt the example inference scripts in TI edgeai-tidl-tools for your model. Additional examples are provided in the TVM Git repository.

First, get the compiled model artifacts (TVM deployable module) to run on the EVM. Then check to make sure the inference results match the expected outputs for the given inputs.

After the model is running correctly on the EVM, use the performance profiling method described in the *Inference Explained* section to see if performance matches expectations.

4.4 Step 4: Compile with c7x_codegen=1

If there are TIDL unsupported layers in the model, you may also try running them on the C7x. Running layers on the C7x can help save the overhead between C7x TIDL subgraphs and layers on Arm. This can also lead to better performance with either TVM auto-generated C7x code or user-written C7x code for the TIDL unsupported layers.

4.5 Step 5: Inference and Performance Profiling

Once the model is compiled successfully with c7x_codegen=1, run it on the EVM and check to make sure the inference results still match the expected outputs for the given inputs.

After the model is running correctly on the EVM, use the performance profiling method described in the *Inference Explained* section to see if performance matches expectations.

4.6 Step 6: Performance Tuning

If the performance of TIDL unsupported layers does not match expectations, try the following:

- Work around issues by rewriting Relay IR code.
- Optimize the C7x code (either TVM-generated or user-written)

See the *Extending TVM* section for examples. Feedback on TI E2E forum is welcome (see the *Support* section).

CHAPTER

FIVE

EXTENDING TVM

This section explains how to extend TVM to help improve the inference performance of a model. These extensions should not require a rebuild of TVM.

5.1 Transforming Relay IR

After a model is converted into TVM's Relay IR format, it is possible to rewrite Relay IR into another equivalent Relay IR for various reasons such as: simplifying arithmetic, removing identity ops, or maximizing TIDL offload. The `python/tvm/relay/backend/contrib/tidl/prepare.py` script contains the transformations run before TIDL partitioning and C7x code generation. These transformations are Python based, straightforward to understand, and easy to write.

The following example shows how transforming Relay IR can help maximize TIDL offload. The example is available in the `tests/python/relay/ti_tests/unit_tests/conv2d_1x2_stride.py` script. The input for this example has a convolution layer with 3x3 kernel size and a 1x2 stride, as shown in the below Relay IR.

```
%0 = nn.conv2d(%i0, %w1, strides=[1, 2], padding=[1, 1, 1, 1],␣
↪kernel_size=[3, 3]);
```

However, TIDL does not support such a kernel size and stride combination. TIDL does support the combination of 3x3 kernel size and 1x1 stride. Using the `ConvertConvStride` transformation, you can rewrite the original convolution as a convolution with 3x3 kernel size and 1x1 stride, followed by maxpooling with 1x1 pool size and 1x2 stride, as shown in the below Relay IR.

```
%0 = nn.conv2d(%i0, meta[relay.Constant][0] /* ty=Tensor[(16, 8,␣
↪3, 3), float32] */,
     Tensor[(1, 8, 224, 224), float32], Tensor[(16, 8, 3, 3),␣
↪float32],
     padding=[1, 1, 1, 1], kernel_size=[3, 3]) /* ty=Tensor[(1,␣
↪16, 224, 224), float32] */;
```

(continues on next page)

(continued from previous page)

```
%1 = nn.max_pool2d(%0, Tensor[(1, 16, 224, 224), float32], pool_
↪size=[1, 1], strides=[1, 2],
    padding=[0, 0, 0, 0]) /* ty=Tensor[(1, 16, 224, 112),␣
↪float32] */;
```

After the transformation, the convolution layer can be offloaded to TIDL and can benefit from the performance boost offered by TIDL on C7x/MMA. The max pooling layer with a 1x1 pool size and 1x2 stride is still not supported by TIDL, but the computation is much simpler and we can use TVM C7x code generation to generate code for this layer. Although this transformation doubles the amount of computation from what was originally required, the overall performance still is better than running the original convolution layer on Arm or C7x.

5.2 Customizing Compute and Schedule

TVM uses strategies to turn each operator into code. An operator can have many strategies associated with it. A strategy consists of two parts, the `compute` and the `schedule`.

- The *compute* specifies what this operator does and is defined as a math formula.
- The *schedule* specifies how to realize the computation via loop nests and data movement.

For each operator, you can customize the compute, the schedule, or both in order to generate better performing code on C7x. The following subsection provide some examples.

5.2.1 Customizing compute

The `python/tvm/topi/c7x/resize.py` script makes a copy of the default `tvm.relay.op.image._image.compute_resize2d` and simplifies the index computation so that the generated code can be software pipelined on C7x devices.

The `python/tvm/relay/op/strategy/c7x.py` script overrides the default strategy for resize2d on C7x with the updated compute and schedule.

5.2.2 Customizing schedule

The `python/tvm/topi/c7x/injective.py` script uses a customized schedule for C7x injective operators. It transforms loop nests, applies DMA with double buffering, and vectorizes the innermost loop for C7x.

5.2.3 Customizing compute and schedule only for a special case

Significantly simplified computation is possible if *all* of the following are true:

- Pool size is 1x1.
- Data layout is "NCHW".
- Dilation is 1x1.
- No padding is used.

The `python/tvm/relay/op/strategy/c7x.py` script customizes the strategy for `max_pool2d` only in this special case. The simplified computation is defined in the `compute_max_pool2d_1x1_pool_size` function in the `python/tvm/topi/c7x/pooling.py`, which treats the operator as an injective operator to use the C7x injective schedule.

For all other cases, the default strategy is used.

5.2.4 Customizing compute and schedule in compilation script

It is also possible to customize the strategy for a relay operator in your compilation script. The `tests/python/relay/ti_tests/unit_tests/resize_nchw_1x2.py` script shows how to overwrite the default resize2d strategy for C7x in the `add_c7x_resize_strategy` function.

5.2.5 Customizing a relay operator to call into an external library

If the resize2d operator has a 1x2 upscaling factor on NCHW float data, the same script used in the previous section, `tests/python/relay/ti_tests/unit_tests/resize_nchw_1x2.py` customizes the compute to call an external function, `resize_nchw_1x2`, in an external library.

The `resize_nchw_1x2.cpp` C++ file contains the function definition. Tensors are passed in a `DLTensor` data structure from TVM runtime to the C/C++ function. The definition of `DLTensor` can be found in the 3rdparty/dlpack/include/dlpack/dlpack.h header file.

The `V1` version of the code shows a naive implementation of resize2d with a 1x2 upscaling factor. The `V2` version of the code optimizes the implementation using the C7x streaming engine (SE) feature.

The C++ file is compiled into a library and linked into the C7x deployable module for the model. As shown in the example, `build_and_set_ext_lib` function in the `unit_utils.py` script, the `CGT7X_EXT_LIBS` environment variable specifies additional libraries that can be linked into the C7x deployable module. For example:

```
CGT7X_EXT_LIBS="-l /path/to/lib1 -l /path/to/lib2"
```

Note: `resize_nchw_1x2.cpp` is not a generic implementation for all resize2d cases with an upscaling factor 1x2. This file handles only float data in "NCHW" layout.

CHAPTER

SIX

BUILDING PACKAGES

6.1 Building TVM

Note: These steps described here are required only if you intend to modify the TI TVM package. Refer to *Getting Started* for instructions on using the prebuilt packages from TI.

The TVM Install from Source page provides instructions on installing the dependencies required for building TVM from source.

The sections below specify additional dependencies required to build TI's tidl-j7 branch.

Note: The TI TVM package builds on Linux only. MacOS and Windows builds are not currently supported.

TI's TVM releases are synchronized with TI's Processor SDK RTOS releases. The following table lists the PSDK RTOS releases and the corresponding TVM tag that is compatible with each one.

PSDK release	TVM release tag
8.6	TIDL_PSDK_8.6.0
8.5	TIDL_PSDK_8.5.0, TIDL_PSDK_8.5.1
8.4	TIDL_PSDK_8.4
8.2	TIDL_PSDK_8.2
8.1	TIDL_PSDK_8.1
8.0	TIDL_PSDK_8.0
7.3	TIDL_PSDK_7.3

6.1.1 Prerequisites

PSDK RTOS

- Download and install the Processor SDK RTOS release corresponding to the TVM release tag you plan to use.
- Set the PSDKR_PATH environment variable to point to the installation. For example:

```
export PSDKR_PATH=/path/to/ti-processor-sdk-rtos-j721e-evm-
↪08_06_zz_ww
```

Clang/LLVM

- Download and install clang+llvm-10.0.0-x86_64-linux-gnu-ubuntu-18.04 from the LLVM Git repository.

Arm GCC

- If you are building the aarch64 TVM runtime and DLR packages, download and install the x86_64 Linux hosted cross compiler for AArch64 GNU/Linux from the Arm GNU Toolchain download page.
- Set the ARM64_GCC_PATH environment variable to point to the installation directory. For example:

```
export ARM64_GCC_PATH=/path/to/gcc-arm-9.2-2019.12-x86_64-
↪aarch64-none-linux-gnu
```

6.1.2 TVM compiler and runtime for x86_64

The steps below outline building the TVM compiler and creating the Python package for x86_64.

```
git clone https://github.com/TexasInstruments/tvm.git; cd tvm
git checkout <corresponding_tag>
git submodule update --init --recursive

mkdir build_x86; cd build_x86
cmake -DUSE_MICRO=ON -DUSE_SORT=ON -DUSE_TIDL=ON -DUSE_LLVM="/
↪path/to/clang+llvm-10.0.0-x86_64-linux-gnu-ubuntu-18.04/bin/
↪llvm-config --link-static" -DHIDE_PRIVATE_SYMBOLS=ON -DUSE_
↪TIDL_RT_PATH=$(ls -d ${PSDKR_PATH}/tidl_j7*/arm-tidl/rt) -DUSE_
```

(continues on next page)

(continued from previous page)

```
↪TIDL_PSDKR_PATH=${PSDKR_PATH} ..
make clean; make

# Build python package in $TVM_HOME/python/dist
cd ..; rm -fr build; ln -s build_x86 build
cd python; python3 ./setup.py bdist_wheel; ls dist
```

Note: Building the TVM compiler for AArch64 is not supported.

6.1.3 TVM runtime for AArch64

The TVM Runtime is an alternative to using the DLR for running inference. It provides C and Python APIs to load and run models compiled by TVM. The steps below outline building just the TVM runtime for AArch64.

```
mkdir build_aarch64; cd build_aarch64
cmake -DUSE_SORT=ON -DUSE_TIDL=ON -DUSE_TIDL_RT_PATH=$(ls -d $
↪{PSDKR_PATH}/tidl_j7*/arm-tidl/rt) -DUSE_TIDL_PSDKR_PATH=$
↪{PSDKR_PATH} -DCMAKE_TOOLCHAIN_FILE=../cmake/modules/contrib/
↪ti-aarch64-linux-gcc-toolchain.cmake ..
make clean; make runtime
```

6.2 Building DLR

The Neo-AI Deep Learning Runtime (*DLR*) is used for inference, that is, to load and run models compiled by TVM. DLR can be built for x86_64 to enable host emulation, that is, to run a model with TIDL offload on a x86_64 PC. DLR can also be built AArch64 and used for inference on the device.

6.2.1 x86_64 package

```
git clone https://github.com/TexasInstruments/neo-ai-dlr.git; cd␣
↪neo-ai-dlr
git checkout <corresponding_tag>
git submodule update --init --recursive

mkdir build_x86; cd build_x86
```

(continues on next page)

(continued from previous page)

```
cmake -DUSE_TIDL=ON -DUSE_TIDL_RT_PATH=$(ls -d ${PSDKR_PATH}/
↪tidl_j7*/arm-tidl/rt) -DDLR_BUILD_TESTS=OFF ..
make clean; make

# Build python package in $DLR_HOME/python/dist
cd ..; rm -f build; ln -s build_x86 build
cd python; python3 ./setup.py bdist_wheel; ls dist
```

6.2.2 AArch64 package

```
git clone <this_repo>; cd neo-ai-dlr
git checkout <corresponding_tag>
git submodule update --init --recursive

mkdir build_aarch64; cd build_aarch64
cmake -DUSE_TIDL=ON -DUSE_TIDL_RT_PATH=$(ls -d ${PSDKR_PATH}/
↪tidl_j7*/arm-tidl/rt) -DDLR_BUILD_TESTS=OFF -DCMAKE_TOOLCHAIN_
↪FILE=../cmake/ti-aarch64-linux-gcc-toolchain.cmake ..
make clean; make -j$(nproc)

# build python package in $DLR_HOME/python/dist
cd ..; rm -f build; ln -s build_aarch64 build
cd python; python3 ./setup.py bdist_wheel; ls dist
```

6.3 Rebuilding C7x Firmware

In some cases, you may want to rebuild the C7x firmware when using the TVM+TIDL flow. For example to increase the number of subgraphs or the heap size as described below. Please refer to the official user guide on rebuilding firmware for more details.

6.3.1 Increase allowed number of TIDL subgraphs

When using C7x to run the TIDL unsupported layers, the maximum number of TIDL subgraphs is 16 by default. If you have a model that has more than 16 TIDL subgraphs, this limit can be increased by rebuilding the TIDL library and the C7x firmware. Information on the number of TIDL subgraphs are available in the output messages shown during compilation. For example, "TIDL import of 41 Relay IR subgraphs succeeded."

First, set up the PSDK RTOS build.

```
$ export PSDK_INSTALL_PATH=/path/to/ti-processor-sdk-rtos-<SOC>-
↪evm-<xx>_<yy>_<zz>_<ww>
$ cd ${PSDK_INSTALL_PATH}
$ ./psdk_rtos/scripts/setup_psdk_rtos.sh
```

Next, edit *${PSDK_INSTALL_PATH}/tidl_<SOC>_<xx>_<yy>_<zz>_<ww>/ti_dl/inc/itidl_ti.h* header file. Increase *TIDL_MAX_OBJECTS_PER_LEVEL* to match the number of TIDL subgraphs in your model. Then rebuild the *tidl_algo.lib* library.

```
$ cd ${PSDK_INSTALL_PATH}/tidl_<SOC>_<xx>_<yy>_<zz>_<ww>
$ vi ti_dl/inc/itidl_ti.h
$ make tidl_algo
```

Finally, rebuild the C7x firmware and copy it to your EVM.

```
$ cd ${PSDK_INSTALL_PATH}/vision_apps
$ make firmware
$ scp out/<SOC>/C7<XYZ>/FREERTOS/release/vx_app_rtos_linux_c7x_1.
↪out root@evm:/lib/firmware/vision_apps_evm (or vision_apps_
↪eaik)
```

Note: Rebuilding C7x firmware alone does not work if layers unsupported by TIDL are executed on the Arm, because the maximum number of TIDL subgraphs is also subject to the OpenVX environment setup.

6.3.2 Increase allocated C7x DDR Heap Size

The default C7x firmware comes with a default allocation for C7x local heap, which is used by TVM runtime to run TIDL subgraphs and unsupported layers. If your model requires more memory, you can increase the allocated C7x local heap size and rebuild the C7x firmware. Please refer to the official developer notes on updating SDK memory map for details.

CHAPTER

SEVEN

ADDITIONAL DOCUMENTATION AND LINKS

See the following documentation for related information.

- Apache TVM documentation
- TI Deep Learning library (TIDL) User Guide
- Edge AI page on ti.com
- TI Edgeai TIDL Tools and Examples Git repository
- TI TVM fork Git repository
- C7000 Code Generation Tools technical documentation
- Arm Code Generation Tools technical documentation

CHAPTER

EIGHT

GLOSSARY

Apache TVM
An open source machine learning compiler framework for CPUs, GPUs, and other devices. It enables running and optimization of machine learning computations on such hardware.

compute
A mathematical formula that specifies what an operator does.

DLR
Deep Learning Runtime from Amazon AWS. For the purpose of running TVM compiled models with the DLR runtime, DLR is simply a wrapper around the TVM runtime.

inference
Inferencing is the act of using a trained deep learning network to output a prediction about incoming data.

MMA
The Matrix Multiplication Accelerator (MMA) is a key hardware accelerator on TDA4 processors. The MMA provides highly parallel deep learning instructions. It is architected to optimize data flow management for deep learning, while minimizing power and external memory devices. The MMA is accessed as an extension of the C71x instruction set, and leverages the same highly parallel data path as the C71x.

Relay IR
TVM's internal common representation for machine learning models.

schedule
Learning rate schedules allow you to optimize the training of a machine learning network. Such optimization is possible because increased performance and faster training can be achieved using a learning rate that changes during training. Common learning rate schedules include time-based decay, step decay, and exponential decay.

subgraphs
In machine learning, a graph is a data structure that consists of nodes (also called vertices) and edges that connect these nodes. A subgraph is a subset of these nodes and edges. Subgraphs can be useful in machine learning for purposes such as identifying patterns or relationships within the data or for simplifying a complex graph.

TDA4 family
: The TDA4VM family includes dual Arm Cortex-A72 SoC and C7x DSPs. It is designed for applications in deep-learning, vision and multimedia.

TIDL
: TI Deep Learning library is TI's software ecosystem for deep learning algorithm (CNN) acceleration. It contains highly optimized implementations of common layers on C7x/MMA. TI TVM can offload some of its computations to TIDL.

TVM
: Tensor Virtual Machine (TVM) is a compiler stack used to compile various deep learning models from different frameworks to specialized CPU, GPU or other accelerator architectures.

CHAPTER

NINE

RELEASE NOTES

9.1 TIDL_PSDK_8.6

- New devices
  - Support AM62A - To compile model for AM62A, set `platform` argument to "AM62A", see *Compiling Models* for details.
- Bug fixes in C7x code generation
  - Fix DMAPass src on/off chip check
  - Fix scatter_nd code generation

9.2 TIDL_PSDK_8.5

- Bug fixes in C7x code generation
  - Support non-contiguous access pattern in DMA
  - Support strided access pattern in SE
  - Support odd number of DMA blocks and iterations
  - Simplify resize2d index computation
  - Optimize maxpool2d with 1x1 pool size
- TIDL offload
  - Rewrite broadcast 1D "add" as "bias_add" and offload to TIDL
  - Do not offload resize2d with asymmetric or non-power-of-2 scaling factor
  - Enabled TIDL batch processing
  - Rewrite conv2d with 1x2 strides as conv2d with 1x1 strides followed by maxpool2d with 1x2 strides

  - Offload sigmoid to TIDL
- TVM extension
  - Support calling external function with DLTensor args
  - Demonstrate overwriting default TVM strategy for an operator
  - Demonstrate overwriting default TVM strategy for an operator for a special case

9.3 TIDL_PSDK_8.4

- Bug fixes in C7x code generation
  - Fix streaming engine pass for loops containing multiple accesses to same tensor
  - Add nop() function to support Reshape op in TVM C runtime
  - Avoid overriding generic op strategy in "hls.py" (back-ported from upstream TVM)
  - Add C7x strategy for concatenate instead of using generic strategy
  - Do not vectorize inner loop with iterations less than vector length
  - Do not vectorize if loop body contains call
  - Fix vectorization factor to use largest data type in computation
  - Add 64-bit integer support in streaming engine config
  - Fix TVM C runtime to support more than 255 functions/layers
  - Return tvm runtime create failure to OpenVX node
  - Apply tiling and dma schedule only to broadcast ops in injective.py
- J721S2 support
- Merge with upstream neo-ai tvm 1.11.2
- Tensor debug support for TVM Arm runtime and TVM C7x runtime

9.4 TIDL_PSDK_8.2

- Initial release of C7x code generation support
- Merge with neo-ai-tvm 1.10.0

CHAPTER TEN

SUPPORT

Post questions about TI TVM to the TI E2E™ design community forum and select the TI device being used.

CHAPTER

ELEVEN

IMPORTANT NOTICE AND DISCLAIMER

INDEX