TEXAS INSTRUMENTS

TI TVM User's Guide

v11.2.0

Copyright © 2026, Texas Instruments Incorporated

Online HTML version available here

CONTENTS

1 Getting Started 2

2 Compiling Models 7

3 Running Inference 12

4 Supported Operators 19

5 Troubleshooting 24

6 Additional Documentation 28

7 Glossary 29

8 Release Notes 31

9 Support 33

10 IMPORTANT NOTICE AND DISCLAIMER 34

Index 35

Texas Instruments' fork of the Apache Tensor Virtual Machine (*TVM*) enables support for the Jacinto/Sitara family of processors. These processors include a C7™ NPU designed to accelerate machine learning *inference*. For additional information about TDA4x processors and TI's Edge AI ecosystem, refer to the Edge AI page on ti.com.

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

TI TVM provides a complete compilation and inference runtime for deploying deep neural networks on TI's embedded processors. Key benefits:

- **Unified Runtime**: Single framework for compilation and inference
- **Hardware Acceleration**: Automatic offload of supported layers to the C7™ NPU for maximum performance
- **Flexible Fallback**: Unsupported layers execute on Arm or C7™ NPU via TVM code generation
- **Standard Model Support**: Import models from ONNX
- **Production Ready**: Integrated with TI's Edge AI ecosystem

TIDL (TI Deep Learning) is TI's software product for accelerating deep neural networks on TI's embedded devices. It provides highly optimized implementations of common DNN layers running on the C7™ NPU, and is released as part of TI's Processor SDK. TI TVM integrates TIDL as an accelerator backend — the compiler automatically partitions the model so that TIDL-supported layers execute with maximum hardware efficiency while TVM generates C7™ NPU code for any remaining layers.

Key Insight

- *TIDL* handles supported operators with highly optimized C7™ NPU performance
- TVM handles operators outside TIDL coverage via C7™ NPU code generation
- The result: the **entire model runs on C7™ NPU**

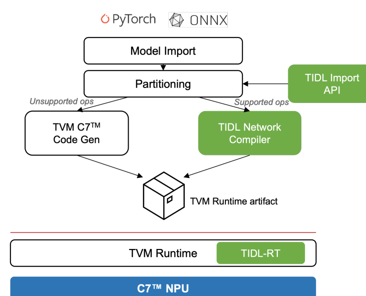

Fig. 1: TI TVM Architecture: Models are imported, partitioned between TIDL-accelerated and TVM-generated code, and compiled into a unified runtime artifact. Green components are handled by TIDL; white components are handled by TVM.

New to TI TVM? Start with the *Getting Started* chapter.

CHAPTER

ONE

GETTING STARTED

This chapter walks you through the end-to-end workflow for deploying a deep learning model on a TI evaluation board (EVM) using TI TVM. By the end you will have a compiled model artifact running inference on the target device.

The workflow consists of four steps:

1. **Install** — Set up TI TVM and the required toolchain on your host machine.
2. **Understand your model** — Identify input/output names, shapes, and verify expected results before bringing TVM into the picture.
3. **Compile** — Use the TI TVM compiler to partition the model between *TIDL*-accelerated and TVM-generated code, and produce a deployable artifact.
4. **Run inference** — Load the compiled artifact on the EVM and run inference using the TVM runtime.

Follow the *Quick Start* to get something running end-to-end, then refer to *Compiling Models* and *Running Inference* for full reference documentation.

1.1 Installation Instructions

This section describes how to set up TI TVM on a host machine for model compilation and on a target EVM for inference.

1.1.1 Prerequisites

Host machine (x86_64):

- Ubuntu 22.04 LTS
- Python 3.10 (virtual environment recommended)
- At least 16 GB RAM and 20 GB free disk space
- System packages:

```
sudo apt-get install libyaml-cpp-dev libglib2.0-dev python-setuptools
```

Target EVM (AArch64):

- SD card with Linux file system from appropriate TI Processor Linux SDK installed

1.1.2 Supported Devices

Device Family	SOC Variable	C7™ NPU Acceleration
J721S2 / TDA4VL	J721S2 or TDA4VL	Yes
J784S4 / TDA4VH	J784S4 or TDA4VH	Yes
J722S / TDA4AEN / AM67A	J722S or TDA4AEN or AM67A	Yes
AM62A	AM62A	Yes
AM62	AM62	No

1.1.3 Processor SDK Versions

TI TVM 0.18.0 is released as part of the following Processor SDK versions:

Device Family	Processor SDK RTOS	Processor SDK Linux
J721E / TDA4VM	11.02.00.06	11.02.00.04
J721S2 / AM68A	11.02.00.06	11.02.00.04
J784S4 / AM69A	11.02.00.06	11.02.00.04
J722S / AM62A / AM62	N/A	N/A

1.1.4 Host Installation (Compilation)

The recommended way to install TI TVM is via the edgeai-tidl-tools repository. The setup script installs TVM, TIDL tools, and all required Python dependencies in a single step.

Clone edgeai-tidl-tools

Clone the repository and check out the tag that matches your SDK version:

```
git clone https://github.com/TexasInstruments/edgeai-tidl-tools.git
cd edgeai-tidl-tools
git checkout <tag>   # e.g. 11_02_04_00 -- see repository tags for your SDK␣
↪version
```

Run the setup script

Note

A Python 3.10 virtual environment is recommended before running the setup script.

```
./scripts/setup/setup.sh
```

The setup script installs the following:

Component	Version / Notes
TVM Python wheel	0.18.0 (x86_64, Ubuntu 22.04, Python 3.10)
ONNX Runtime Python wheel	1.15.0 (TI build with TIDL execution provider)
TFLite Runtime Python wheel	2.12.0 (TI build)
TIDL Runtime Python wheel	0.1.0
TIDL tools	Per-SOC libraries downloaded from `software-dl.ti.com`
AArch64 GCC toolchain	arm-gnu-toolchain 13.2.Rel1 (for cross-compilation)
C7x compiler (CGT)	ti-cgt-c7000 5.0.0.LTS (required for `c7x_codegen=1`)
C++ runtime dependencies	ONNX Runtime, TFLite, and TVM C++ headers/libraries
Example models and inputs	Out-of-box data for edgeai-tidl-tools examples

Set environment variables

Source the environment setup script with your target SoC:

```
source ./scripts/setup/setup_env.sh <SOC>    # e.g. J721E, J784S4, AM62A
```

This sets `SOC`, `TIDL_TOOLS_PATH`, `ARM64_GCC_PATH`, and `CGT7X_ROOT` automatically. See the table below for the full list of variables.

Variable	Description
`SOC`	Target SoC identifier (see Supported Devices above)
`TIDL_TOOLS_PATH`	Path to TIDL libraries (required for TIDL offload)
`ARM64_GCC_PATH`	Path to AArch64 GCC toolchain (required when `compile_for_device=1`)
`CGT7X_ROOT`	Path to C7000 Code Generation Tools (required when `c7x_codegen=1`)

Verify the installation

```
python3 -c "import tvm; print(tvm.__version__)"
```

Expected output: `0.18.0`

1.1.5 Target Installation (Inference)

All dependencies required to run TVM inference are included in the Linux filesystem packaged with the Processor SDK.

Note

Processor SDK 11.2 only: Two Python packages required by the TVM runtime (`psutil` and `typing_extensions`) are missing from this release due to a known issue. Install them manually on the EVM:

```
pip3 install psutil typing_extensions
```

This will be resolved in the next SDK release.

1.2 Preparing Your Model

TI TVM compiles models in **ONNX** format. If your model is in a different framework, export it to ONNX first.

If you do not have a model yet, the TI Edge AI Model Zoo provides models that have been validated and optimized for inference on TI SoCs.

Before compiling with TVM, make sure you know:

- **Input name and shape** — required by the TVM compiler (e.g. `"input"` with shape `[1, 3, 224, 224]`)
- **Expected outputs** — needed to verify inference results on the EVM match the original model

1.2.1 Verify the model before compilation

It is strongly recommended to run inference with your model on the host using `onnxruntime` before compiling with TVM. This confirms the model is well-formed and establishes a reference output to compare against after deployment.

```
import numpy as np
import onnxruntime as rt

sess = rt.InferenceSession("model.onnx")
input_name = sess.get_inputs()[0].name
input_shape = sess.get_inputs()[0].shape   # e.g. [1, 3, 224, 224]

# Replace with real pre-processed input data
input_data = np.random.rand(*input_shape).astype(np.float32)
output = sess.run(None, {input_name: input_data})
print(output)
```

Save the output — you will use it to validate inference results after deploying the compiled model on the EVM.

1.3 Quick Start

This page gets you from a trained model to a running inference on the EVM in four steps. For a full explanation of each step, see *Compiling Models* and *Running Inference*.

1.3.1 Prepare calibration data

```
import numpy as np

# Replace "input" with your model's actual input name, and preprocess()
# and calib_images with your own data loading and preprocessing logic.
calib = np.stack([preprocess(img) for img in calib_images])
np.savez_compressed("calibration.npz", input=calib)  # key must match model␣
↪input name
```

1.3.2 Compile the model

```
python -m tvm.driver.tvmc compile model.onnx \
  --target tidl \
  --c7x-codegen 1 \
  --tidl-calibration-input calibration.npz \
  --output ./artifacts/
```

This produces three files in `./artifacts/`: `deploy_lib.so`, `deploy_graph.json`, and `deploy_param.params`.

1.3.3 Copy artifacts to the EVM

```
scp -r ./artifacts/ root@<evm-ip>:~/model/
```

1.3.4 Run inference on the EVM

```
import tvm
from tvm.contrib import graph_executor
import numpy as np

# Load the compiled module
lib    = tvm.runtime.load_module("model/deploy_lib.so")
graph  = open("model/deploy_graph.json").read()
params = bytearray(open("model/deploy_param.params", "rb").read())

dev = tvm.cpu(0)
module = graph_executor.create(graph, lib, dev)
module.load_params(params)

# Run inference
module.set_input("input", input_data)
module.run()
output = module.get_output(0).numpy()
print(output)
```

Note

The first inference on the EVM includes TIDL initialization overhead. Run a warm-up inference before measuring performance.

Next steps:

- *Compiling Models* — full TVMC and Python API reference, compilation artifacts, debugging
- *Running Inference* — inference runtimes, performance profiling, debugging

Once inference results match expectations, use the performance profiling tools in *Running Inference* to measure execution time and identify bottlenecks.

CHAPTER

TWO

COMPILING MODELS

TI TVM compiles a model into a deployable artifact through three stages:

```
ONNX model
    → Import into TVM
    → Partition
          TIDL-supported layers  →  TIDL artifacts
          Unsupported layers     →  C7x™ NPU generated code
    → deploy_lib.so + deploy_graph.json + deploy_param.params
```

The same model is compiled twice in sequence — first for x86 (host emulation) and then for AArch64 (EVM). The TIDL partitioning and import steps run only once; the results are reused for the EVM build via the `REUSE_TIDL_ARTIFACTS` environment variable.

2.1 Supported Model Formats

TI TVM accepts models in **ONNX** format. ONNX has been validated by TI. If your model is in a different framework, export it to ONNX first.

2.2 Calibration Data

TIDL runs inference with quantized fixed-point values. During compilation, calibration data is used to estimate each layer's dynamic range and compute scaling factors. You only need to provide calibration data for the whole model; TI TVM automatically extracts the tensor values at TIDL subgraph boundaries.

Provide a small representative set of input frames (typically 2–10) as a `.npz` file:

```
import numpy as np

# Each key is an input name; value is an array of N frames with shape [N, ...]
calib_dict = {
    "input": np.stack([preprocess(img) for img in calib_images])
}
np.savez_compressed("calibration.npz", **calib_dict)
```

Choose calibration inputs that represent the distribution of real inference inputs — poor calibration data leads to accuracy loss after quantization.

2.3 Compilation Interfaces

TI TVM provides two equivalent interfaces for compilation. Both produce identical artifacts and accept the same options — the choice is a matter of preference:

- **TVMC** — command-line interface; options passed via a YAML config file
- **Python API** — `compile_model` function; options passed as a Python dict

2.3.1 Compiling with TVMC

Prepare a YAML config file with the TIDL compile options (see *Compilation Options* for the full list):

```
# config.yaml
compile_options:
  "tensor_bits": 8
  "advanced_options:calibration_frames": 5
  "advanced_options:calibration_iterations": 5
  "advanced_options:c7x_codegen": 1
```

Then compile the model:

```
python -m tvm.driver.tvmc compile model.onnx \
  --target tidl \
  --tidl-config config.yaml \
  --tidl-calibration-input calibration.npz \
  --output ./artifacts/
```

TVMC options

Option	Default	Description
`--target`	—	Must be set to `tidl` to enable TI TVM compilation.
`--tidl-config`	—	Path to a YAML file containing TIDL compile options.
`--tidl-calibration-inpu`	—	Path to a `.npz` file containing calibration frames. Required when TIDL offload is enabled.
`--c7x-codegen`	0	Set to `1` to compile TVM-generated code for C7™ NPU execution. Unsupported layers run on C7™ NPU instead of Arm. Overrides the value in `--tidl-config` if both are set.
`--enable-tidl-offload`	1	Set to `0` to disable TIDL offload and run the entire model via TVM code generation only.
`--compile-for-device`	1	Set to `1` to cross-compile for AArch64 (EVM). Set to `0` to compile for x86 host execution only.
`--output`	`./ model-a`	Directory where compiled artifacts are written.

2.3.2 Compiling with the Python API

The `compile_model` function accepts the same options as the TVMC YAML config file, passed as a Python dict via `delegate_options`:

Example — compile for both host and EVM:

```
import os
import numpy as np
from tvm.contrib.tidl.compile import compile_model

delegate_options = {
    "artifacts_folder":                      "./artifacts",
    "tensor_bits":                           8,
    "advanced_options:c7x_codegen":          1,
    "advanced_options:calibration_frames": 5,
}

calibration_input = [{"input": frame} for frame in calib_frames]

# Compile for x86 host first
compile_model(
    platform=os.environ["SOC"],
    compile_for_device=False,
    enable_tidl_offload=True,
    delegate_options=delegate_options,
    calibration_input_list=calibration_input,
    model_path="model.onnx",
    input_shape_dict=[{"input": (1, 3, 224, 224)}],
)

# Reuse TIDL artifacts from the host build for the EVM build
os.environ["REUSE_TIDL_ARTIFACTS"] = "1"

compile_model(
    platform=os.environ["SOC"],
    compile_for_device=True,
    enable_tidl_offload=True,
    delegate_options=delegate_options,
    calibration_input_list=calibration_input,
    model_path="model.onnx",
    input_shape_dict=[{"input": (1, 3, 224, 224)}],
)
```

2.4 Compilation Options

The following options control TIDL compilation behaviour. They are passed via the `compile_options` section in the TVMC YAML config file, or as keys in the `delegate_options` dict when using the Python API.

Option	Default	Description
`tensor_bits`	8	Quantization precision: `8`, `16`, or `32`.
`advanced_options:c7x_codegen`	0	Set to `1` to compile TVM-generated code for C7™ NPU execution. Unsupported layers run on C7™ NPU instead of Arm.
`advanced_options:calibration_f:`	2	Number of calibration frames used for quantization.
`advanced_options:calibration_i:`	5	Number of calibration iterations per frame.

Note

For a complete, production-ready example of both compilation interfaces see tvmrt_wrapper.py in the edgeai-tidl-tools repository.

2.5 Compilation Artifacts

After a successful compilation, three files are written to `artifacts_folder`:

- **deploy_lib.so** — fat binary containing TIDL subgraphs and C7™ NPU code
- **deploy_graph.json** — execution graph describing nodes and data flow
- **deploy_param.params** — model weights

These three files are everything needed to run inference on the EVM.

Hint

During development you can NFS-mount the host compilation directory on the EVM, avoiding the need to copy artifacts after each recompile.

2.5.1 Intermediate artifacts

TI TVM also writes intermediate files to `artifacts_folder/tempDir/` that are useful for understanding and debugging compilation.

Relay graphs — snapshots of the model at each compilation stage:

- `relay_graph.orig.txt` — original graph from the TVM frontend
- `relay_graph.prepared.txt` — after pre-TIDL transformations
- `relay_graph.annotated.txt` — annotated for TIDL offload
- `relay_graph.partitioned.txt` — after partitioning; check this to see which layers are offloaded to TIDL and which are not
- `relay_graph.import.txt` — used for TIDL import
- `relay_graph.optimized.txt` — final optimized graph for code generation
- `relay_graph.wrapper.txt` — Arm-side wrapper graph for C7™ NPU dispatch

TIDL artifacts:

- `relay.gv.svg` — graphical view of the whole network with TIDL subgraph boundaries highlighted
- `subgraph<n>_net.bin.html` — detailed view of each TIDL subgraph

Generated C7|tm| NPU code (when `c7x_codegen=1`):

- `model_<n>.c` — TVM-generated C7™ NPU code for each subgraph and non-TIDL layer

2.6 Running Unsupported Layers on Arm

Setting `--c7x-codegen 0` maps TIDL-unsupported layers to the Arm core instead of C7™ NPU. This is useful when the C7000 CGT toolchain is not available or for quick bring-up without the full toolchain.

```
python -m tvm.driver.tvmc compile model.onnx \
  --target tidl \
  --c7x-codegen 0 \
  --tidl-calibration-input calibration.npz \
  --output ./artifacts/
```

Note

Arm fallback requires `ARM64_GCC_PATH` to be set and increases inference latency due to data transfers between Arm and C7™ NPU. Use `c7x_codegen=1` for production deployments.

2.7 Debugging Compilation

Set `TIDL_RELAY_IMPORT_DEBUG` to get verbose output during compilation.

Value	Output
`1`	Per-node TIDL support status, Relay-to-TIDL node conversion, subgraph summary, and calibration progress.
`2`, `3`	All output from level 1, plus detailed TIDL subgraph import information.

Example output at level 1:

```
export TIDL_RELAY_IMPORT_DEBUG=1
python -m tvm.driver.tvmc compile model.onnx --target tidl ...
```

```
RelayImportDebug: In TIDL_relayImportNode:
RelayImportDebug: node name: 185, op name: tidl.conv2d, num_args: 2
RelayImportDebug:   args[0] dims: [1, 512, 7, 7]
RelayImportDebug:   args[1] dims: [512, 512, 3, 3]
```

CHAPTER THREE

RUNNING INFERENCE

TVM inference requires the three artifact files produced by compilation: `deploy_lib.so`, `deploy_graph.json`, and `deploy_param.params`. The runtime behaviour is determined at compile time: with `c7x_codegen=1` the Arm core dispatches the entire graph to C7™ NPU via OpenVX; with `c7x_codegen=0` the Arm core executes non-TIDL layers directly alongside TIDL subgraph dispatch.

The same inference code works for both cases.

3.1 Running Inference with Python

Load the compiled module and run inference using TVM's `graph_executor`:

```
import tvm
from tvm.contrib import graph_executor
import numpy as np

artifacts_folder = "./artifacts"

# Load compiled artifacts
graph  = open(f"{artifacts_folder}/deploy_graph.json").read()
lib    = tvm.runtime.load_module(f"{artifacts_folder}/deploy_lib.so")
params = bytearray(open(f"{artifacts_folder}/deploy_param.params", "rb").
↪read())

# Create runtime session
sess = graph_executor.create(graph, lib, tvm.cpu())
sess.load_params(params)

# Run inference
sess.set_input("input", input_data)   # replace "input" with your input name
sess.run()

output = sess.get_output(0).asnumpy()
```

Note

The first inference call includes TIDL initialisation overhead. Run a warm-up inference before measuring performance.

The `tvmrt_wrapper.py` in edgeai-tidl-tools wraps this pattern and automatically selects the correct artifact suffix (`.pc` or `.evm`) based on the host architecture. See tvmrt_wrapper.py for a complete reference implementation.

3.2 Performance Profiling

After running inference, retrieve performance metrics via `get_TI_benchmark_data()`:

```
benchmark = sess.get_TI_benchmark_data()
```

The `tvmrt_wrapper.py` `get_performance()` method processes this data into the following metrics:

Metric	Unit	Description
`total_time`	ms	Total inference time from run start to run end
`core_time`	ms	Total time excluding I/O copy overhead
`subgraph_time`	ms	Time spent executing TIDL subgraphs on C7™ NPU
`read_total`	bytes	DDR read bytes during inference (not available on x86)
`write_total`	bytes	DDR write bytes during inference (not available on x86)
`num_subgraphs`	—	Number of TIDL subgraphs detected in the compiled model

For trace-based performance profiling using `TVM_RT_DEBUG`, see *Debugging Inference*.

3.3 Running edgeai-tidl-tools Examples

The edgeai-tidl-tools repository provides ready-to-run examples for both Python and C++.

3.3.1 On the host (x86)

Ensure environment variables are set by sourcing `setup_env.sh` before running any examples.

```
cd ./runtimes/examples/python/basic_example/

# Compile and run inference with TVM runtime
python3 basic_example.py --config ./config.yaml -r tvmrt --compile
python3 basic_example.py --config ./config.yaml -r tvmrt --infer

cd -
```

Model artifacts are saved to `./runtimes/examples/model-artifacts/`. Inference outputs are saved to `./runtimes/examples/python/basic_example/outputs/{model_name}/offload/frame_{frame_num}/`.

C++ inference examples are also provided (compilation not supported from C++):

```
cd ./runtimes/examples/cpp/basic_example
../bin/Release/basic_example --config ./config.yaml
cd -
```

3.3.2 On the EVM

Model compilation must be performed on the host first. Transfer artifacts to the EVM (or use NFS):

```
scp -r ./runtimes/examples/model-artifacts root@<evm-ip>:~/
```

Then run inference on the EVM:

```
export SOC=<SOC>
cd ./runtimes/examples/python/basic_example/
python3 basic_example.py --config ./config.yaml -r tvmrt --infer
cd -
```

> **Note**
>
> Processor SDK 11.2 is missing two TVM runtime dependencies. Install them before running TVM examples on the EVM:
>
> ```
> pip3 install psutil typing_extensions
> ```
>
> See *Installation Instructions* for details.

3.4 Debugging Inference

TIDL uses the `TIDL_RT_DEBUG` environment variable to control debug output during inference.

3.4.1 Vision apps "printf" terminal

Open a terminal on the EVM and run the following command to see debug output from the C7™ NPU core:

```
# 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 inference in a separate terminal.

3.4.2 Debugging TIDL subgraphs

TIDL_RT_DEBUG=1

When set to 1, TIDL subgraph performance information is printed 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, detailed TIDL subgraph import and layer execution information is provided in addition to the level 1 output. 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
... ... ...
[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

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 to the Arm Linux filesystem 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.

3.4.3 Debugging TVM nodes

The `TVM_RT_DEBUG`, `TVM_RT_TRACE_NODE`, `TVM_RT_TRACE_SIZE`, and `TVM_TRACE_NODE` environment variables control TVM node debugging.

TVM_RT_DEBUG=1

When set to 1, TVM runtime on Arm collects performance statistics for each node in the graph and saves them to the Arm Linux filesystem as `tvm_arm.trace`. Use the `dump_tvm_trace.py` script to read the file:

```
# 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
```

(continues on next page)

(continued from previous page)

```
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 C7™ NPU, `tvm_arm.trace` shows only a single node representing the whole graph:

```
# 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, TVM runtime on C7™ NPU also collects per-node performance statistics and saves them to `tvm_c7x.trace` in addition to the level 1 output. 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
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
```

`tvm_c7x.trace` is only available when the model was compiled with `c7x_codegen=1`.

TVM_RT_DEBUG=3

When set to 3, TVM runtime on Arm and C7™ NPU also collects output tensor statistics (min, max, sum, and sum of the first half) for each layer and saves them to the trace files.

Note

Performance numbers collected at level 3 include instrumentation overhead and should not be used for benchmarking.

```
# 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_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 C7™ NPU prints information about each node as it executes, either on the inference terminal or the "printf" terminal. Useful for tracing execution order and identifying hangs.

TVM_RT_TRACE_NODE=<node_id> TVM_RT_DEBUG=3, 4

When set, TVM runtime saves the tensor outputs of the specified node to the trace file. The `dump_tvm_trace.py` script then saves them as numpy files named `n<node_id>_o<output_id>.npy`. For example:

```
# 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
  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.
```

(continues on next page)

(continued from previous page)

```
↪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, use `TVM_RT_TRACE_SIZE` to increase the trace buffer size. The default is 2 MB (2*1024*1024 bytes).

CHAPTER

FOUR

SUPPORTED OPERATORS

TIDL accelerates the operators listed in this chapter on the C7™ NPU. Operators not listed here fall back to TVM-generated C7™ NPU code (`c7x_codegen=1`) or Arm (`c7x_codegen=0`).

ONNX 1.14.0 / ONNX Runtime 1.15.0 (OPSET-19, IR-8)

For detailed per-operator test reports including attribute and shape coverage, see the edgeai-tidl-tools operator reports.

4.1 Convolution

ONNX Operator	TIDL Layer	Key Constraints
`Conv`	TIDL_ConvolutionLayer	Stride must match horizontally and vertically. 3x3 kernel with stride 3 not supported on AM62A/AM67A. Kernel >7 with stride 2 not supported. Depthwise supported for 1x3s1, 3x3s1/s2, 5x5s1/s2, 7x7s1/s2. Stride 4 only with 11x11 kernel.
`ConvTranspose`	TIDL_Deconv2DLayer	Supported kernels: 4x4, 3x3, 2x2 with 2x2 stride. 16-bit not supported on AM62A/AM67A. Default dilation (1,1) only.
`DeformConv`	TIDL_DeformableConvLayer	3x3 stride-1 filter only.

4.2 Pooling

ONNX Operator	TIDL Layer	Key Constraints
`AveragePool` / `GlobalAveragePool` / `MaxPool`	TIDL_PoolingLayer	GlobalAveragePool: plane size (H×W) must be $\leq$ 1024. Validated kernel sizes: 3x3, 2x2, 1x1 with stride 1 or 2 (except 2x2 stride 1). Default dilation (1,1) only.

4.3 Normalization

ONNX Operator	TIDL Layer	Key Constraints
`BatchNormalization`	TIDL_BatchNormLayer	training_mode=1 not supported. Scale, bias, mean, and variance must be constant 1-D tensors.
`InstanceNormalization`	TIDL_InstanceNormLayer	Input must be $\geq$ 3D. Scale and bias must be 1-D channel vectors.
`LayerNormalization`	TIDL_LayerNormLayer	Width axis only. Scale and bias must be [1,N] or [N].

4.4 Activations

ONNX Operator	TIDL Layer	Key Constraints
`Relu`	TIDL_ReLULayer	
`PRelu`	TIDL_PReLULayer	Variable slope not supported. Slope must be broadcast-able to the channel dimension.
`LeakyRelu`	TIDL_LeakyReluLayer	
`Sigmoid` / `Logistic`	TIDL_SigmoidLayer	
`Tanh`	TIDL_TanhLayer	
`HardSigmoid`	TIDL_HardSigmoidLayer	
`HardSwish`	TIDL_HardSwishLayer	
`Elu`	TIDL_ELULayer	
`Mish`	TIDL_MishLayer	
`Clip`	TIDL_ClipLayer	min $\leq$ 0 and max > 0 only.
`Softmax`	TIDL_SoftMaxLayer	Width and height axes only.

4.5 Element-wise Arithmetic

ONNX Operator	TIDL Layer	Key Constraints
`Add` / `Sub` / `Mul` / `Div` / `Max` / `Min` / `Sum`	TIDL_EltWiseLayer	Two inputs only. Inputs must have matching dimensions or be broadcast-able.
`Abs`	TIDL_AbsLayer	
`Neg`	TIDL_NegLayer	
`Sqrt`	TIDL_SqrtLayer	
`Pow`	TIDL_PowLayer	Exponent must be a constant scalar tensor.
`Exp`	TIDL_ExpLayer	
`Log`	TIDL_LogLayer	
`Floor`	TIDL_FloorLayer	

4.6 Trigonometric

ONNX Operator	TIDL Layer
`Sin`	TIDL_SinLayer
`Cos`	TIDL_CosLayer
`Tan`	TIDL_TanLayer
`Asin`	TIDL_AsinLayer
`Acos`	TIDL_AcosLayer
`Atan`	TIDL_AtanLayer
`Sinh`	TIDL_SinhLayer
`Cosh`	TIDL_CoshLayer
`Asinh`	TIDL_AsinhLayer

4.7 Linear and Matrix

ONNX Operator	TIDL Layer	Key Constraints
`Gemm` / `MatMul`	TIDL_InnerProductLayer	Gemm: transA=0, alpha=1.0, beta=1.0. Bias must be [1,N] or [N]. MatMul with signed/unsigned mixed inputs not supported on TDA4VM.
`TopK`	TIDL_TopKLayer	sorted=0 not supported. K must be a model initializer. Width and height axes only.

4.8 Shape Manipulation

ONNX Operator	TIDL Layer	Key Constraints
`Reshape`	TIDL_ReshapeLayer	Variable shape not supported. Input and output volume must match.
`Flatten`	TIDL_FlattenLayer	
`Squeeze`	TIDL_SqueezeLayer	
`Unsqueeze`	TIDL_UnsqueezeLayer	Output dimensions must be ≤ 6.
`Transpose`	TIDL_TransposeLayer	For >4D inputs, permutations mapping the width dimension to output position 0 or 1 are not supported.
`Concat`	TIDL_ConcatLayer	Axis values -3, -2, -1 only. Batch dimension not supported.
`Slice` / `Split`	TIDL_SliceLayer	4D input only. Batch size=1. Non-singular strides only supported as part of the Patch Merging fusion pattern.
`Pad`	TIDL_PadLayer	Constant pad mode with constant_value=0 only. Width/height axes only.
`Expand`	TIDL_ExpandLayer	Shape tensor must be constant.

4.9 Spatial and Image

ONNX Operator	TIDL Layer	Key Constraints
`Upsample` / `Resize`	TIDL_ResizeLayer	Modes: nearest or linear. Width and height axes only. Scales < 1 not supported. Width and height scales must match. Power-of-2 scales only.
`DepthToSpace`	TIDL_DepthToSpaceLayer	4D input. Channel depth must be multiple of blocksize^2. DCR and CRD modes supported.
`SpaceToDepth`	TIDL_SpaceToDepthLayer	4D input. H and W must be multiples of blocksize.
`GridSample`	TIDL_GridSampleLayer	Nearest and bilinear modes. Zero padding only. 2D grid only. Nearest-neighbour rounding may differ from ONNX reference.

4.10 Reduction

ONNX Operator	TIDL Layer	Key Constraints
`ReduceMean`	TIDL_ReduceMeanLayer	
`ReduceSum`	TIDL_ReduceSumLayer	
`ReduceMin` / `ReduceMax`	TIDL_ReduceLayer	Height axis only. keepdims=1 only.
`ArgMin` / `ArgMax`	TIDL_ArgOpLayer	keepdims=1 only. axis=-3 only.

4.11 Gather and Scatter

ONNX Operator	TIDL Layer	Key Constraints
`Gather`	TIDL_GatherLayer	1D indices only. Input rank must be > 1. Data cannot be constant; only indices can be constant.
`ScatterND` / `ScatterElements`	TIDL_ScatterElementsLayer	ScatterElements: 'none' reduction only. ScatterND: 'mul' reduction not supported. Scatter along width axis only. Input data must be a zero tensor.

4.12 Quantization

ONNX Operator	TIDL Layer	Key Constraints
`Cast`	TIDL_CastLayer	Only at terminal nodes (network input/output).
`DequantizeLinear`	TIDL_DequantizeLayer	axis=1 only. QDQ models only.
`QuantizeLinear`	TIDL_QuantizeLayer	axis=1 only. QDQ models only.

4.13 Fusion Patterns Only

The following operators are not supported in isolation but are accelerated when they appear as part of a recognised fusion pattern.

ONNX Operator	TIDL Layer	Fusion Pattern
`Erf` / `Identity`	TIDL_IdentityLayer	GELU activation
`DropOut`	TIDL_DropOutLayer	Not supported standalone

CHAPTER FIVE

TROUBLESHOOTING

This chapter covers common errors and their solutions.

5.1 Environment Setup Errors

5.1.1 TIDL_TOOLS_PATH not set

Error:

```
Environment variable TIDL_TOOLS_PATH is not set!
```

Solution:

Set the environment variable to point to your TIDL tools installation:

```
export TIDL_TOOLS_PATH=/path/to/tidl_tools
```

Or run the setup script:

```
source /path/to/edgeai-tidl-tools/scripts/setup/setup_env.sh
```

5.1.2 ARM64_GCC_PATH not set

Error:

```
Environment variable ARM64_GCC_PATH is not set!
```

Solution:

This is required when compiling for device (`compile_for_device=1`). Set it to point to your ARM64 GCC toolchain:

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

5.1.3 CGT7X_ROOT not set

Error:

```
Set environment variable CGT7X_ROOT to location of C7000 Code Generation Tools
```

Solution:

Required when `c7x_codegen=1`. Set it to point to your C7000 CGT installation:

```
export CGT7X_ROOT=/path/to/ti-cgt-c7000_x.x.x
```

5.2 Compilation Errors

5.2.1 artifacts_folder is not empty

Error:

```
'artifacts_folder' is not empty - please clear the folder and re-run!
```

Solution:

Clear the artifacts folder before recompiling:

```
rm -rf ./artifacts/*
```

5.2.2 Dynamic shapes not supported

Error:

```
Dynamic shape/network not supported by TVM+TIDL yet!!!
```

Solution:

TVM+TIDL requires static shapes. Ensure your model has fixed input dimensions. If the model has dynamic axes, fix them before compilation using the ONNX opset tools or by re-exporting from the training framework with explicit input shapes.

5.2.3 TIDL import failed

Error:

```
TIDL import of Relay IR graph failed.
```

Possible causes:

1. Unsupported operator encountered
2. Operator parameter constraints violated
3. Calibration data issue

Solution:

1. Set `TIDL_RELAY_IMPORT_DEBUG=1` to see detailed import logs
2. Check `relay_graph.import.txt` in artifacts folder
3. Verify calibration data matches model input requirements

5.2.4 Unsupported operator

Error in debug output:

```
Layer type not supported by TIDL --- layer type - X, Node name - Y
```

Solution:

The operator will fall back to Arm/C7x execution. Options:

1. Use `c7x_codegen=1` to run unsupported ops on C7™ NPU
2. Check if an equivalent supported operator exists
3. Use the deny_list option to explicitly exclude problematic layers

5.2.5 Only ONNX models supported

Error:

```
ERROR: Only ONNX models can be converted to Relay IR internally.
```

Solution:

Convert your model to ONNX format first, or convert to Relay IR manually:

```
# For PyTorch models
torch.onnx.export(model, dummy_input, "model.onnx")
```

5.3 Inference Errors

5.3.1 Vision apps not initialized

Symptom: Inference hangs or fails silently

Solution:

Ensure vision_apps is initialized on the EVM:

```
cd /opt/vision_apps
source ./vision_apps_init.sh
```

5.3.2 Trace file shows slow performance

Symptom: `tvm_arm.trace` shows unexpectedly high execution times

Possible causes:

1. First inference includes initialization overhead
2. TIDL subgraphs not being used
3. Too many Arm fallback operations

Solution:

1. Run multiple inferences and ignore the first one
2. Check that TIDL artifacts are being loaded
3. Review `relay_graph.partitioned.txt` to see TIDL offload coverage

5.4 Debugging Tips

For detailed debug output during compilation, see *Debugging Compilation* in the Compiling Models chapter.

For inference debugging using `TIDL_RT_DEBUG` and `TVM_RT_DEBUG`, see the Debugging Inference section in the *Running Inference* chapter.

5.5 FAQ

Q: Why are some operators not offloaded to TIDL?

A: An operator may not be offloaded if:

- It's not in the supported operators list (see *Supported Operators*)
- Its parameters violate TIDL constraints (e.g., kernel size, stride)
- It's explicitly in the deny_list

Q: Should I use c7x_codegen=0 or c7x_codegen=1?

A: Use `c7x_codegen=1` for production deployments — it runs TIDL-unsupported layers on the C7™ NPU and eliminates Arm-C7™ NPU data transfer overhead. Use `c7x_codegen=0` only if the C7000 CGT toolchain is not available or as a temporary fallback during initial bring-up.

Q: How do I know which layers are running on TIDL vs Arm/C7x?

A: Check `relay_graph.partitioned.txt` - nodes prefixed with `tidl_` run on TIDL.

Q: My model accuracy is different from the original framework?

A: This is usually due to quantization. Try:

- Increase `calibration_frames` and `calibration_iterations`
- Use `tensor_bits=16` for higher precision
- Use `accuracy_level=9` with custom calibration options

Q: How many TIDL subgraphs should my model have?

A: Fewer subgraphs generally means better performance due to reduced data transfer overhead. If you see many small subgraphs, check if unsupported operators are fragmenting the graph. Consider using `c7x_codegen=1` to consolidate execution on the C7™ NPU.

Q: What calibration data should I use?

A: Use representative data from your actual use case. The calibration data helps TIDL determine the dynamic range for quantization. Using non-representative data (e.g., all zeros or random noise) can lead to poor accuracy.

CHAPTER

SIX

ADDITIONAL DOCUMENTATION

TI edgeai-tidl-tools
The primary repository for getting started with TI's Edge AI runtimes. Provides setup scripts, Python and C++ wrapper APIs, and ready-to-run examples for model compilation and inference using TVM, ONNX Runtime, and TFLite on TI SoCs.

TI TVM source
TI's fork of Apache TVM. Contains TI-specific extensions including the TIDL BYOC integration, C7™ NPU code generation, and the C7x runtime port.

TI Edge AI
Overview of TI's complete Edge AI ecosystem, including supported devices, development tools, model zoo, and training and benchmark frameworks.

Apache TVM documentation
Upstream TVM documentation covering Relay IR, operator strategies, autotuning, and the runtime. Useful reference for understanding TVM internals and features beyond TI-specific extensions.

C7000 Code Generation Tools
Technical documentation for the TI C7000 compiler, assembler, and linker. Relevant for users writing or debugging custom C7x NPU code when `c7x_codegen=1` is enabled.

CHAPTER

SEVEN

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.

C7™ NPU
The C7™ Neural Processing Unit combines TI's C7x DSP with the Matrix Multiplication Accelerator (MMA). It provides highly parallel deep learning instructions optimized for neural network inference. Also referred to as C7x+MMA in older documentation.

calibration
The process of determining quantization parameters (scale and zero-point) for converting floating-point tensors to fixed-point representation. Calibration data should be representative of actual inference inputs.

compute
A mathematical formula that specifies what an operator does.

DMA
Direct Memory Access. Used by the C7™ NPU to efficiently transfer data between memory and processing units without CPU involvement.

fat binary
A deployable module (.so file) that contains both TVM-generated code and embedded TIDL artifacts, allowing a single file to run inference.

inference
The act of using a trained deep learning model to produce a prediction from input data.

MMA
The Matrix Multiplication Accelerator (MMA) is a key hardware accelerator on TDA4/AM6x 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 C7x instruction set. Together with the C7x DSP, it forms the C7™ NPU.

OpenVX
A cross-platform API for computer vision applications. TI TVM uses OpenVX to dispatch TIDL subgraphs to the C7™ NPU during inference.

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

schedule
In TVM, a schedule specifies how to realize a computation via loop nests and data movement. Schedules can be tuned to optimize performance for specific hardware.

Streaming Engine
A C7x hardware feature that prefetches data in predictable patterns, improving memory bandwidth utilization

for vector operations.

subgraphs
A subset of nodes in a model graph. In TI TVM, the model is partitioned into subgraphs: those that can be accelerated by TIDL run on the C7™ NPU, and the remainder execute via TVM code generation.

TDA4 and AM6x families
The TDA4 and AM6x families include dual/quad Arm Cortex-A72/Cortex-A53 SoCs with C7™ NPU. They are 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 C7™ NPU. TI TVM can offload supported 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

EIGHT

RELEASE NOTES

8.1 Software Manifest

The Software Manifest lists all open-source components included in this release, along with their versions and license information.

8.1.1 TIDL_PSDK_11.2

- New features
  - Updated TI TVM fork to upstream TVM version 0.18.0 (from 0.12.0)
  - Replaced neo-ai-dlr with TVM runtime for inference across all interfaces and example applications
  - Added TVMC command-line interface for simplified model compilation on C7™ NPU
  - Added TIDL offload support for object detection post-processing (ODPostProc) via TIDL Meta Architecture
  - Added TIDL offload support for Transformer layer patterns (Layer Normalization, GeLU)
- Bug fixes
  - Revamped TIDL offload compilation mechanism for improved reliability

8.1.2 TIDL_PSDK_8.6

- New devices
  - Added AM62A support. To compile a model for AM62A, set the `platform` argument to `"AM62A"`; see *Compiling Models* for details.
- Bug fixes in C7™ NPU code generation
  - Fixed DMAPass on/off-chip source check
  - Fixed scatter_nd code generation

8.1.3 TIDL_PSDK_8.5

- Bug fixes in C7™ NPU code generation
  - Added support for non-contiguous access patterns in DMA
  - Added support for strided access patterns in the Streaming Engine
  - Added support for odd numbers of DMA blocks and iterations
  - Simplified resize2d index computation

    - Optimized maxpool2d with 1x1 pool size
- TIDL offload improvements
    - Rewrote broadcast 1D `add` as `bias_add` and offloaded to TIDL
    - Disabled TIDL offload for resize2d with asymmetric or non-power-of-2 scaling factors
    - Enabled TIDL batch processing
    - Rewrote conv2d with 1x2 strides as conv2d with 1x1 strides followed by maxpool2d with 1x2 strides
    - Added TIDL offload support for sigmoid
- TVM extensions
    - Added support for calling external functions with DLTensor arguments
    - Added example demonstrating how to override the default TVM strategy for an operator
    - Added example demonstrating how to override the default TVM strategy for a specific case

8.1.4 TIDL_PSDK_8.4

- Bug fixes in C7™ NPU code generation
    - Fixed Streaming Engine pass for loops with multiple accesses to the same tensor
    - Added `nop()` function to support the Reshape operator in TVM C runtime
    - Avoided overriding generic operator strategy in `hls.py` (back-ported from upstream TVM)
    - Added C7™ NPU strategy for concatenate instead of using the generic strategy
    - Disabled vectorization of inner loops with fewer iterations than the vector length
    - Disabled vectorization when the loop body contains a function call
    - Fixed vectorization factor to use the largest data type in the computation
    - Added 64-bit integer support in Streaming Engine configuration
    - Fixed TVM C runtime to support more than 255 functions/layers
    - Returned TVM runtime creation failures to the OpenVX node
    - Applied tiling and DMA schedule only to broadcast ops in `injective.py`
- Added J721S2 device support
- Merged with upstream neo-ai-tvm 1.11.2
- Added tensor debug support for TVM Arm runtime and TVM C7™ NPU runtime

8.1.5 TIDL_PSDK_8.2

- Initial release of C7™ NPU code generation support
- Merged with neo-ai-tvm 1.10.0

CHAPTER

NINE

SUPPORT

Post questions and report issues about TI TVM to the TI E2E™ design community forum. Select the processor family you are using — for example, Processors > Jacinto™ Automotive Processors for TDA4/J7 devices, or Processors > Sitara™ Processors for AM6x devices.

CHAPTER TEN

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to TI's Terms of Sale (https://www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265

INDEX

A

Apache TVM, **29**

C

C7tm NPU, **29**
calibration, **29**
compute, **29**

D

DMA, **29**

F

fat binary, **29**

I

inference, **29**

M

MMA, **29**

O

OpenVX, **29**

R

Relay IR, **29**

S

schedule, **29**
Streaming Engine, **29**
subgraphs, **30**

T

TDA4 and AM6x families, **30**
TIDL, **30**
TVM, **30**