TENSAI COMPILER USER GUIDE

Tensai Flow 1.0 Beta Release

Oct 26, 2021

ETA COMPUTE
340 N Westlake Blvd. Suite 115, Westlake Village, California 91362

Table of Contents

1 Introduction

TENSAI Compiler is a tool developed at Eta Compute to aid the conversion of Neural Networks (NN) to C code. With the help of this tool a user can generate C code, from a saved NN model file generated through popular Machine Learning Frameworks, which can be compiled and executed on the target embedded hardware like Synaptics' Katana chip.

With the current release of TENSAI Compiler, the supported Machine Learning framework is limited to TensorFlow Lite, also referred to as TFLite. Figure 1 shows the workflow using the TENSAI Compiler for generation of C code.

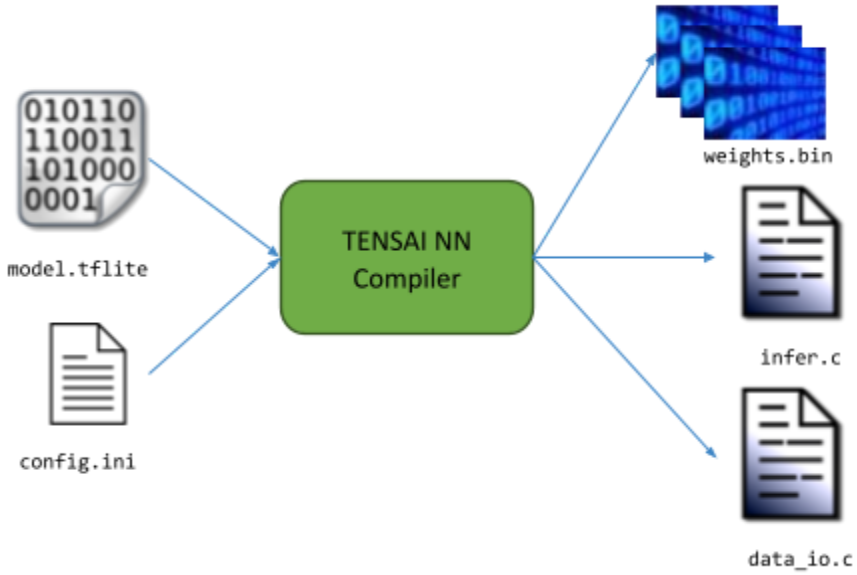


Figure 1. TENSAI Compiler Workflow

2 Prerequisites

An application developer (user) starts with the neural network file that is trained in TensorFlow (TF) framework and converted to TFLite format with full-integer quantization in Int8. This file acts as an input to the compiler. The compiler will parse this file and generate 2 sets of files, 1. Set of .c files and 2. Set of .bin files. The .c files contain the code using the executor infrastructure to run the NN and obtain the data from the sensor. The .bin files are a set of binary files containing the weights of the model. These files can be compiled using ARM GNU Toolchain and loaded on the Katana hardware.

This tool is built upon certain requirements and assumptions that are mentioned in the following sections.

2.1 Requirements

1. The TFLite file should be generated using the TF version *2.4.1*[1]
2. The input and output inference datatype of TFLite converter must be set to `tf.int8`
3. The *supported_ops* in the target spec of the TFLite converter must be set to `tf.lite.OpsSet.TFLITE_BUILTINS_INT8`
4. The current tool runs only on the Linux Ubuntu environment

2.2 Assumptions

Data acquisition from any sensors other than camera is out of the scope of this automatic conversion tool. A user is expected to write the right code for the same.

3 Operation and Core support

3.1 Supported Ops

The current version of the tool supports only a subset of TFLite ops in the conversion. Any op that is outside this list will result in a runtime exception. Below is the list of ops that are supported in the current release.

- AveragePool 2D
- Concatenation
- Convolution 2D (including Pointwise Convolution)
- Depthwise Convolution 2D
- Elementwise Add
- Elementwise Multiply
- Maxpool 2D
- Fully connected (Dense)
- Reshape
- Softmax

Additionally, the activation functions like relu and relu6 are supported if they are fused in the TFlite file.

3.2 Supported cores

The current version of the tool supports scheduling of the operations, in a NN model, to only two cores, i.e., M33 or CAPE2 and LLE/NPU. This scheduling can be supported using `config.ini` file as one of the inputs to the compiler. There are two options in the config file as described below.

```
[katana]
primary_core = lle
secondary_core = cape2
```

Snapshot 1 core_config.ini file

[1] TensorFlow operators can change between versions, and other versions may cause undesired results.

Here, primary core is the preferred choice for the user and secondary core means auxiliary core for running the operations not supported by the primary core.

The valid value for `primary_core` key is LLE, M33 or CAPE2 and `secondary_core` key is M33 or CAPE2. To schedule all the ops on M33 or CAPE2 only, supply the value M33 or CAPE2 to both the keys or just to the `primary_core` key and leave `secondary_core` null. It is highly recommended to use CAPE2 over M33 as the kernels running on CAPE2 are optimized and are ~10-15x faster than M33 generic kernels using ARM CMSIS-NN library. Scheduling all the ops on LLE is not supported as there is a limited set of ops supported on LLE core. The keys and values are case-insensitive. However, the section name `[katana]` is case sensitive. Any value provided outside this will result in a runtime exception.

On LLE, only a small subset of the above operations is supported, leaving the rest of the ops to be scheduled to run on the secondary core. Below is a list of kernels, along with any constraints that are supported on LLE.

- Convolution 2D (including Pointwise Convolution)
- Depthwise Convolution 2D
    - Kernel Size: 3x3
    - Stride h < 3, stride w < 3
    - Padding ('SAME' in TF) h < 2, w< 2
    - Input h > 2, w > 2

On CAPE2, there is a support for optimized kernels if they meet some conditions as described below:

- Convolution 2D (including Pointwise Convolution)
    - Input Channel is a multiple of 8 and Output Channel is a multiple of 2
- Depthwise Convolution 2D
    - Depth Multiplier = 1
        - Input Channel is a multiple of 8
    - Depth Multiplier is a multiple of 8
    - Depth Multiplier is a multiple of 4 and Input Channel < 32
- Fully Connected
    - Input size (Feature Len) is a multiple of 8

If any layer falls outside these criteria, then generic kernels are used.

4 Usage

Open any command line utility like terminal in Ubuntu based system and enter the directory `Tools/Tensai_compiler/bin` in the release package. As an alternative, the path to this folder can be added to the environment variable PATH in the .bashrc file of the user.

This tool takes in 7 command line arguments (other than `--help`) as shown in Snapshot 2.

`--app_dir` takes in a path to the directory where the output .c and .bin files will be saved, also the application directory. (Required)

`--model` takes in a path to the TFLite model file with the NN to be processed. (Required)

`--target_config` takes in a path to a core_config.ini file as described in Snapshot 1.

`--[no]data_io` is a boolean flag indicating the tool to write a template for data_io.c as described in ***4.1 Using data_io.c***

`--[no]memory_info_file` is a boolean flag to export the information about memory used in bytes by the weights, bias, input/output buffers, and other miscellaneous arrays needed to compute the outputs.

`--dump_buffer` is a flag that accepts an integer which corresponds to the layer number in the TFLite file. The layer numbers are identified by a field 'location' when inspected using a network viewer tool like Netron. (Check the ***4.2 Debugging Tools*** section for more info)

`--layer_profiling` is a flag that accepts one of 0, 1, 2 to print the layer profiling information for every layer in the model. (Check the ***4.2 Debugging Tools*** section for more info)

0 - means off (default)
1- means print the profiling info in a text format
2- means print the profiling info in a csv format

```
Tools/Tensai_compiler/bin>$ ./tensaicc --help
File: tensaicc
Main entry point of the compiler application.
flags:
./tensaicc:
  --app_dir: Path to the application directory in
tensaiflow_software
  --[no]data_io: Flag to indicate if data_io.c should be generated.
    (default: 'true')
  --[no]memory_info_file: Flag to indicate whether to write a CSV
    file with the name same as the model, appended by '_mem_info'
    with the information about memory used by weights per layer.
    (default: 'false')
  --model: Path to the TFLite model file.
  --target_config: Config.ini file containing the target core
    information.
    (default: 'core_config.ini')
```

```
  --dump_buffer: An integer corresponding to the location of the
    layer in the TFLite file for which the buffer values will be
    dumped over UART.
  --layer_profiling: <0|1|2>: Flag to choose if a profiling report
    per layer is to be printed on UART. Choose one of the below for
    printing:
      0-off, 1-descriptive, 2-csv
    This will also write a .json file containing layer information.
    (default: '0')

Try --helpfull to get a list of all flags.
```

Snapshot 2 tensiflow-nn-compile help options

As an alternative, flags may be loaded from text files in addition to being specified on the command line.

This means that you can throw any flags you do not want to type into a file, listing one flag per line. So, all the flags can be put in a single file and a single flag `--flagfile=flags.txt` can be passed. For example:

```
# This is a comment
# Filename: flags.txt
--model=/path/to/model.tflite
--app_dir=/path/to/application_dir
--target_config=path/to/core_config.ini
--[no]data_io
--[no]memory_info_file
```

Snapshot 3 Flag file with all the flags, each per line

4.1 Using data_io.c

To put the application in the real world, it is necessary to integrate the NN application with the proper sensor data. To ease the integration and avoid any added buffer space in the application, two interfaces, namely, `get_data()` and `post_process()` are provided. These functions will be populated with the code to read the data from the filesystem or a camera. The user of the application can feel free to modify code to get the sensor data in `get_data()` and any post processing code in the `post_process()` function by ensuring that the input parameters to these functions are kept intact.

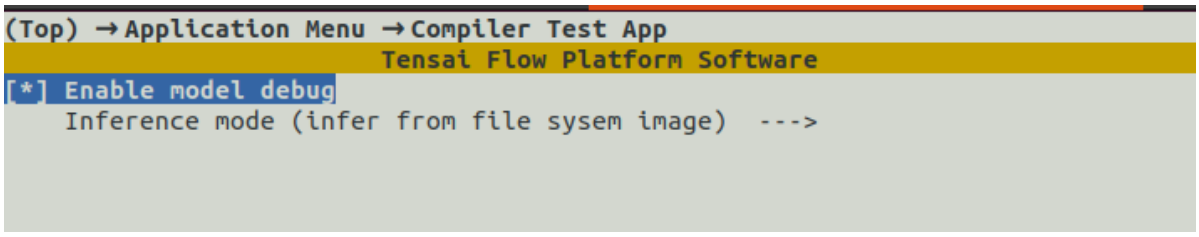
```
void get_data(int8_t *in_buf_0,
              uint16_t in_buf_0_dim_0,
              uint16_t in_buf_0_dim_1,
              uint16_t in_buf_0_dim_2) {
      // Code to get the data from sensor or filesystem
}

void post_process(int8_t *out_buf_0){
      // Post processing on the inference output buffers
      // By default, it will have the code that can be used to
      // compare the predictions to the expected output.
}
```

Snapshot 4 Sample data_io.c generated by the tool.

This functionality is useful when converting a model for different applications which may have different numbers of inputs/outputs. Moreover, the dimension information for each input will be provided as `uint16_t` variables after each input name. However, for a different type of model for the same application, this file may not have to be generated every time. Thus, use the `--nodata_io` flag to disable the generation of this file and avoid overwriting it all the time.

4.2 Debugging Tools

The model and application developers can get more insights on the performance of the inference, both in terms of the latency and correctness of the predictions by using the two command line flags `--dump_buffer` and `--layer_profiling` as described in the command line flags above. The details on how to use each of the flags is as follows. To enable the debugging tools, select the "Enable model debug" in the menuconfig as shown in Snapshot 5.



Snapshot 5 menuconfig entry enabling the debug mode for the app.

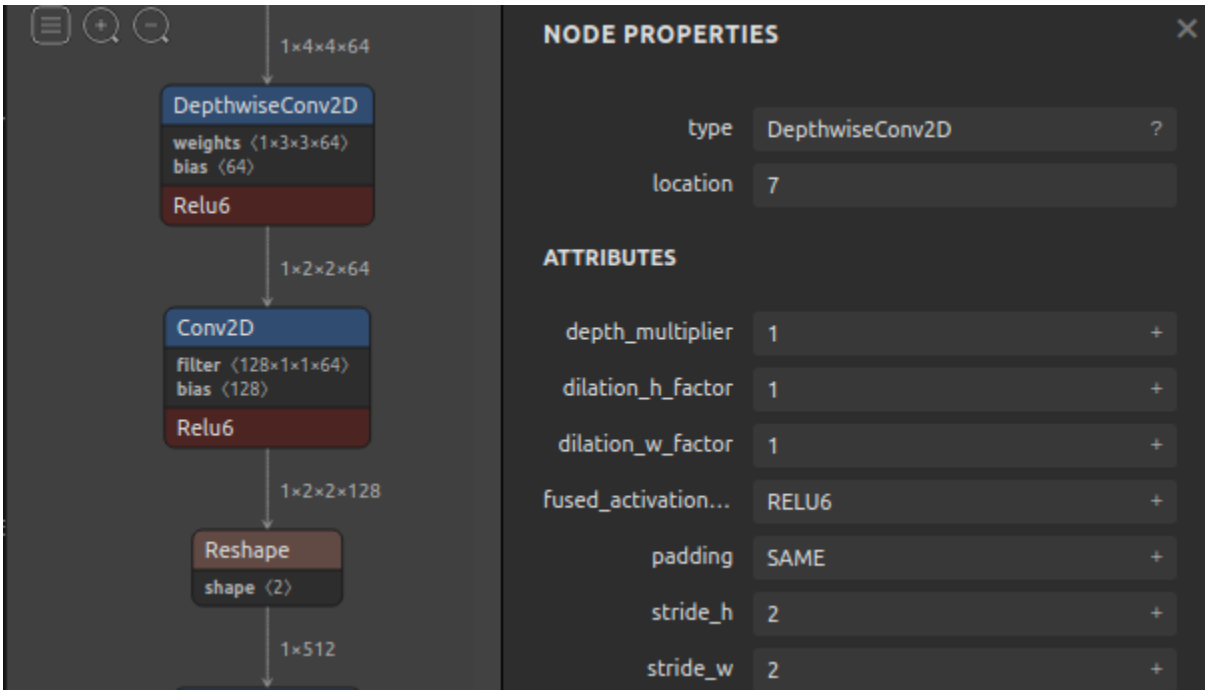
NOTE: These two flags cannot be used simultaneously as profiling a layer will give incorrect results when large buffers are dumped over the UART.

4.2.1 Buffer Dump

When using the `--dump_buffer` argument, pass in a number that corresponds to the layer location in the TFLite file. The generated code will have functions to dump the output buffer values for each layer number in this input argument. This can be utilized as a debugging tool to identify any layer which produces a high mismatch with the expected output during the

inference.

For example, see the Snapshot 6 which shows that a Depthwise Conv2D layer is at location 7. Then use the compiler flag to pass in 7 and it will produce the output as shown in Snapshot 7. Note that when printing the buffers, it will also print the output name (different from the layer name) and size of the buffer for that layer.



Snapshot 6 Mnist model snapshot from Netron tool.

```
conv2d_8_0[256] ={
-84,-116,-94,-128,-128,-95,-80,-104,-117,-82,-120,-120,-83,-128,-91,-128,
-128,-102,-119,-128,-128,-128,-51,-128,-128,-128,-53,-128,-128,-128,-128,-122,
-128,-100,-118,-101,-101,-128,-76,-90,-126,-105,-128,-128,-128,-128,-128,-79,
-128,-128,-104,-112,-76,-111,-123,-128,-78,-128,-128,-128,-128,-98,-128,-112,
-96,-128,-121,-51,-128,-128,-116,-111,-128,-87,-116,-128,-105,-128,-128,-128,
-128,-128,-128,-128,-128,-128,-128,-128,-128,-87,-92,-128,-121,-128,-128,-117,
-128,-116,-99,-127,-118,-116,-77,-127,-128,-102,-128,-101,-128,-104,-128,-127,
-128,-120,-128,-128,-89,-109,-122,-128,-115,-128,-128,-102,-94,-128,-118,-119,
-83,-113,-128,-128,-70,-128,-104,-128,-126,-105,-128,-128,-113,-128,-99,-128,
-128,-85,-128,-128,-128,-108,-100,-128,-128,-128,-111,-125,-109,-128,-128,-128,
-128,-128,-92,-126,-119,-97,-96,-87,-106,-116,-128,-88,-128,-82,-128,-96,
-128,-117,-128,-128,-107,-115,-112,-128,-86,-128,-128,-119,-128,-110,-86,-111,
-99,-119,-128,-103,-121,-128,-124,-113,-128,-122,-123,-128,-128,-128,-128,-128,
-128,-128,-115,-128,-128,-117,-120,-128,-128,-100,-124,-127,-94,-128,-128,-126,
-128,-116,-96,-116,-122,-122,-100,-106,-108,-128,-128,-108,-128,-119,-128,-122,
-128,-108,-115,-128,-128,-127,-119,-128,-97,-128,-128,-128,-128,-128,-125,-111
};
```

Snapshot 7 Mnist model output with buffer values printed.

4.2.2 Layer Profiling

When using the `--layer_profiling` argument, specify 1, or 2 to print out the per-layer timings and cycle count. Moreover, a .json file will be generated with the format as shown in Snapshot 8 below.

```
{
  "record_type": "modelLayers",
  "value": {
    "subgraphs": [
      {
        "subgraph_name": "subgraph_0_layers",
        "layers": [
          {
            "layer_num": "0",
            "layer_tfl_loc": "0",
            "layer_name": "conv2d_0",
            "layer_type": "Conv2D",
            "target_core": "NPU",
            "inputs": [
              "input_0",
              "conv2d_0_weights_1",
              "conv2d_0_bias_2",
              "conv2d_0_qinfo_3",
              "conv2d_0_qinfo_4"
            ],
            "outputs": [
              "depthwiseconv2d_1_0"
            ]
          },

          ...other layers...
        ]
      }
    ]
  }
}
```

Snapshot 8 Output format of .json file for profiling.

The output on the UART will look like the display shown in Snapshot 9 below.

```
Begin infer test
"PerLayerTiming",0,221047,8.99
"PerLayerTiming",1,96956,3.94
"PerLayerTiming",2,62020,2.52
"PerLayerTiming",3,75896,3.8
"PerLayerTiming",4,43592,1.77
"PerLayerTiming",5,48182,1.96
"PerLayerTiming",6,39342,1.60
"PerLayerTiming",7,42669,1.73
"PerLayerTiming",8,54480,2.21
"PerLayerTiming",9,20052,0.81
"PerLayerTiming",10,30143,1.22
"PerLayerTiming",11,24261,0.98
```

Snapshot 9 Model Profiling output on UART.

When printing the profiling output in CSV format, the output can be interpreted as: “PerLayerTiming”, layer order (in infer.c), cycle count, time (ms).

5 Example

Along with the compiler, two simple models with some of the layers that are mentioned in the supported ops section, are provided in the model zoo of this release. These models are trained on academic datasets, namely MNIST and Fashion MNIST dataset, to recognize 10 different classes of images. MNIST recognizes ten digits from 0-9 and Fashion MNIST recognizes the following categories: *T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot*.

The trained model files are available at:

`Tools/Tensai_compiler/model_zoo/fashion_mnist_model`

`Tools/Tensai_compiler/model_zoo/mnist_model`

To convert these models into the C code, and test the example application, use the following command.

```
Tools/Tensai_compiler/bin$ ./tensaicc \
--model=../model_zoo/fashion_mnist_model/model.tflite \
--app_dir=../../../Applications/compiler-test-app/ \
--target_config=core_config.ini \
--nodata_io
Code written successfully!
Total memory used by the weights(kB) in subgraph 0:
Weights - 26.67
NPU Weights - 2.09
Bias - 0.04
Quantization Arrays (for per-channel ops) - 0.06
Total RO Memory - 28.87
```

Snapshot 10 Command to run example Fashion MNIST model and the output.

Note the usage of `--nodata_io` flag here as the `compiler-test-app` already has the filled in `data_io.c`.

5.1 Generating input_data.h

To generate the `input_data.h` with the data from an example test image, please use the accompanying script:

`Tools/image_converter/convert_img_to_header.py`.

5.2 Compiling and Loading the App

To compile and load the app, go to the build directory of the destination app and run the below commands.

```
Applications/compiler-test-app/build$ cmake ..
Applications/compiler-test-app/build$ make app
Applications/compiler-test-app/build$ make flash
```

Snapshot 11 Commands to compile and load the app.

6 Camera demo

With the current release, we provide a working example of model inference running with the images captured from a camera integrated with the Katana chip. For this demo, we strongly recommend using the MNIST model as we have verified the pipeline with this model. To enable the inference with the camera, change the `menuconfig` of the `compiler-test-app` as shown in Snapshot 12. The explanation for the specific menu entry is provided in the Quick Start Guide provided at: `TENSAI_FLOW/Docs/Quick_Start_Guide.pdf`

```
(Top) → Application Menu → Compiler Test App
                    Tensai Flow Platform Software
[ ] Enable model debug
    Inference mode (infer from camera image)  --->
[*] Enable camera (NEW)
(324)   Camera Row count (NEW)
(324)   Camera Column count (NEW)
(72)    Signed threshold value for image processing (NEW)
[ ]     Display captured image (NEW)
[ ]     Print resized image (NEW)

[Space/Enter] Toggle/enter  [ESC] Leave menu          [S] Save
[O] Load                    [?] Symbol info           [/] Jump to symbol
[F] Toggle show-help mode   [C] Toggle show-name mode [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

Snapshot 12 Application Menu selection for inference using camera.

Generate the infer.c and related files using the compiler tool binary as shown above in Snapshot 10, but using the MNIST model from model zoo:

`Tools/Tensai_compiler/model_zoo/mnist_model.`

Build the application and load it on the device as shown in Snapshot 6 (**5.2 Compiling and Loading the App**).

To verify the pipeline, we have provided a pdf file:

`Tools/Tensai_compiler/model_zoo/mnist_model/test_images_for_camera.pdf`

Please print this file on paper and then hold the printouts in front of the camera. Make sure that no other background is visible when pointing the camera at these images, hold the pictures such that the numerical fills most of the camera's field of view. An example of the captured image to be sent for inference is shown below in Snapshot 13.

Snapshot 13 Image captured from the camera.

The model was trained on the MNIST dataset with white numerals with black backgrounds, hence the corresponding colored printouts. Moreover, room lighting may also affect the predictions so there may be a need to adjust the thresholding parameter in the application menuconfig as highlighted below in Snapshot 14.

```
(Top) → Application Menu → Compiler Test App
                      Tensai Flow Platform Software
[ ] Enable model debug
    Inference mode (infer from camera image)  --->
[*] Enable camera (NEW)
(324)   Camera Row count (NEW)
(324)   Camera Column count (NEW)
(72)    Signed threshold value for image processing (NEW)
[ ]     Display captured image (NEW)
[ ]     Print resized image (NEW)


[Space/Enter] Toggle/enter  [ESC] Leave menu           [S] Save
[O] Load                    [?] Symbol info            [/] Jump to symbol
[F] Toggle show-help mode   [C] Toggle show-name mode  [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

Snapshot 14 Application menu entry for thresholding.

The thresholding value is to be based on the range of [-128, 127] as opposed to [0, 255] because the image is expected to be shifted by the TFLite model.

When the setup is as expected, a serial utility can be used to observe the predictions as shown below in snapshot 15.

```
Inference output for mnist : FIVE
total infer time in 66 msec, in cycles 1627176
total time (capture + preprocess image + infer) in 587 msec, in cycles 14436935
Begin infer test
Respsonse recieved
Image capture done
----Outputs ----
[0]:-128
[1]:-126
[2]:-128
[3]:-119
[4]:-128
[5]:111
[6]:-128
[7]:-122
[8]:-128
[9]:-128
Inference output for mnist : FIVE
total infer time in 66 msec, in cycles 1627027
total time (capture + preprocess image + infer) in 587 msec, in cycles 14436983
```

Snapshot 15 Output with the image shown in snapshot 8.

7 List of Known Exceptions

Below is the list of exceptions that are currently thrown by the compiler tool.

- **OperationError**: Error raised when an unsupported operation is present in the input Model.
- **BatchSizeError**: Error raised when the input model has batch size > 1.
- **FileExtensionError**: Error raised when the input file is of different extension than tflite.
- **PaddingOperationError**: Error raised when the operation Pad is present such that it does unequal padding on (top, bottom) or (left, right).
- **AppDirNotFoundError**: Error raised when trying to write code to a nonexistent dir.
- **TensorDtypeError**: Error raised when a tensor other than int8 or int32 is in the model.
- **InvalidPrimaryCoreError**: Error raised when an invalid string is provided in the core_config.ini for the primary core field.
- **InvalidSecondaryCoreError**: Error raised when an invalid string is provided in the core_config.ini for the secondary core field.
- **TensorDtypeError**: Error raised when a tensor in floating point is encountered in the model.
- **ConcatOpFusedActError**: Error raised when a Concat op with fused activation is present in the input model.
- **QuantizationParamError**: Error raised when an invalid quantization parameter is found in a model. Sparse models may have a scale which is near 0 and thus cannot be supported in the current version.
- **SoftmaxInputDimsError**: Error raised when a 4D input is provided to the Softmax in a model. Softmax is only supported on a 1D array or a 2D array excluding the batch size.
- **ElementwiseOpError**: Error raised when the operations like add and mul are not elementwise, i.e.,the input1 size and input2 size are not equal.