Embedded Deep Learning Deployment. Demystified.

TI model Zoo - Ready-to-use, Easy-to-use and Efficient AI models

Jacinto™ AI monthly webinar series

2022 Sep

Code examples used in the webinar are below.

https://e2e.ti.com/support/processors-group/processors/f/processors-forum/1014084/tda4vm-jacinto-ti-edge-ai-monthly-webinar-jul-2021-embedded-deep-learning-deployment-demystified

Why is “deployment” important in deep learning?

Deep learning has two parts: Training and Inference

Training is done once or a few times : Off-line operation

Inference happens during the life of the product : Real-time operation

How many inference queries happen a day across all AI applications?

200 trillion queries/day[1]

Growing exponentially as more products are including AI

Training is off-line, Inference is real-time!

- ✓ Inference needs to be efficient – faster and low power
- ✓ That is what we will cover in this webinar
- ✓ TI Jacinto’s Edge AI processors make the model deployment as efficient and simple as it can be!

Source: K. Lee, V. Rao, and W. C. Arnold, “Accelerating facebooks infrastructure with application-specific hardware,” https://engineering.fb.com/data-center-engineering/accelerating-infrastructure/, Facebook, 3 2019.

Webinar | Agenda

Recap from the previous webinar:

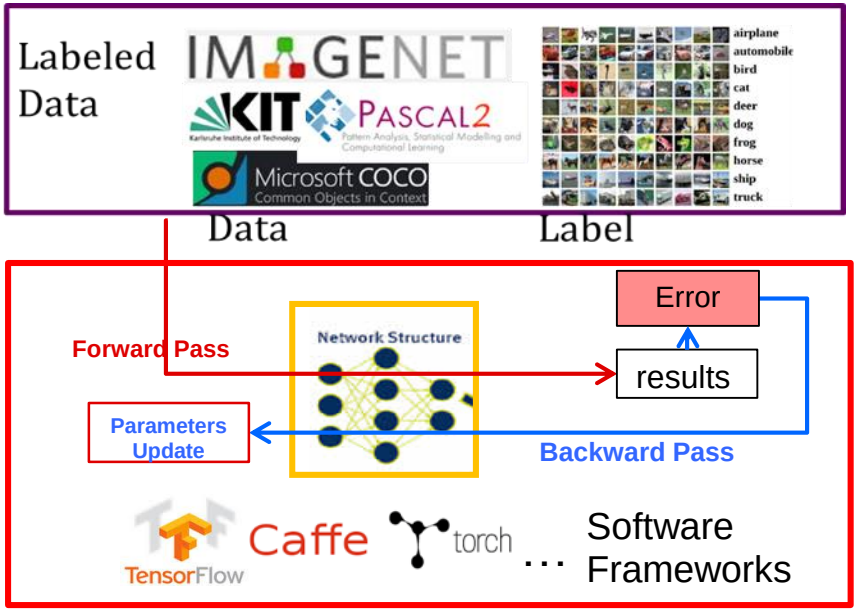
Hello world program: Step1: PC Step2: Embedded on ARM. Step3: **Embedded with Deep learning acceleration**

- Model Deployment in the edgeAI processor
  - What is in the model?
  - Model compilation procedure
  - Open-source run time options
  - Hands-on examples
- Model Zoo
  - Performance benchmarking
- $249 Edge AI EVM

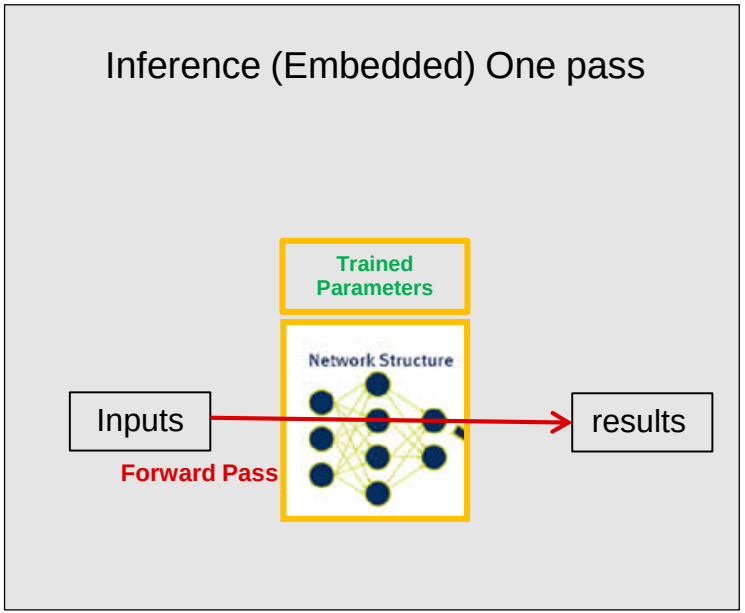
$187.5 BeagleBone AI-64

- Call to action

Deep learning | Training and Inference



Computing Platform (Training)



Embedded Hardware (Inference)

AI Deployment in your system | Three steps

DL Tools & software to reduces model development time

Accelerated inference using open-source industry standard RunTime Engines
Out of box optimized inference support for 60+ models

Train anywhere, Develop anywhere

1

PYTORCH mxnet Keras Caffe TensorFlowLite

1. TI Model Zoo (60+ models)
   - Model Selection tool
   - New weights
2. Own model

Optional QAT (Quantization artifacts tool) from TI

https://github.com/TexasInstruments/jacinto-ai-devkit

Compile & Optimize for TI SoC
Using industry standard Compilers/RTs

2

TVM Compiler/ TFLite / ONNX-RT

- Common representation
- Post Training Quantization
- Calibration
- Optimization
- Compilation

Deploy on TI SoC
Using industry standard APIs

3

TFLite RT /ONNX-RT/Neo-AI-DLR

TI Edge AI Processor

TIDL RunTime ↔ TIDL Library

Linux | RTOS

Cortex-A | DLA

Embedded AI “Hello World“ in 3 steps

Recap from previous webinar

1. Run the program first on the PC (using industry standard tools)
2. Port the ”same program” to embedded platform: J7 EVM device
3. Run with deep learning acceleration

Hello world | on Jacinto EVM in the cloud (DL acceleration)

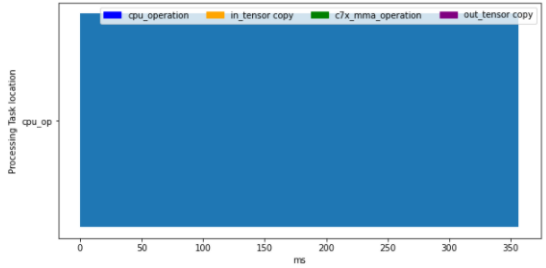
```
n [4]: calib_images = [
'../sample-images/elephant.bmp',
'../sample-images/bus.bmp',
'../sample-images/bicycle.bmp',
'../sample-images/zebra.bmp',
]

# Make a dictionary of labels and class values
label_dict = make_label_dict('TFL-OD-200-ssd-mobV1-coco-mlperf-300x300/labels.txt')

output_dir = 'custom-artifacts'
tflite_model_path ='TFL-OD-200-ssd-mobV1-coco-mlperf-300x300/ssd_mobilenet_v1_coco_2018_01_28.tflite'


compile_options = {
    'tidl_tools_path' : os.environ['TIDL_TOOLS_PATH'],
    'artifacts_folder' : output_dir,
    'tensor_bits' : 8,
    'accuracy_level' : 1,
    'advanced_options:calibration_frames' : len(calib_images),
    'advanced_options:calibration_iterations' : 3,
}

# create the output dir if not preset
# clear the directory
if not os.path.exists(output_dir):
    os.makedirs(output_dir, exist_ok=True)

for root, dirs, files in os.walk(output_dir, topdown=False):
    [os.remove(os.path.join(root, f)) for f in files]
    [os.rmdir(os.path.join(root, d)) for d in dirs]

tidl_delegate = [tflite.load_delegate(os.path.join(os.environ['TIDL_TOOLS_PATH'], 'tidl_model_import_tflite.so'), c
interpreter = tflite.Interpreter(model_path=tflite_model_path, experimental_delegates=tidl_delegate)
interpreter.allocate_tensors()

input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

import tqdm

mean = [128.0, 128.0, 128.0]
scale = [0.0078125, 0.0078125, 0.0078125]
layout = 'NHWC'
reverse_channels = False

for num in tqdm.trange(len(calib_images)):
    #interpreter.set_tensor(input_details[0]['index'], preprocess_for_tflite_mobilenetv1(calib_images[num]))
    interpreter.set_tensor(input_details[0]['index'], preprocess(calib_images[num] , size, mean, scale, layout, rev
    interpreter.invoke()

100%|          | 4/4 [01:25<00:00, 21.49s/it]
```

Two things are new in this step

1) Compile the model to the TI's DL accelerators
2) Use the compiled output for inference

- Rest of the steps are similar to previous steps
- A delegate option in the Interpreter is a way to "**offload**" parts of the network to **hardware accelerator**
- Model compilation uses the same Tensorflowlite APIs
- Acceleration happens automatically by just adding one line of code- tflite.load_delegate (APIs coming from TFLite)

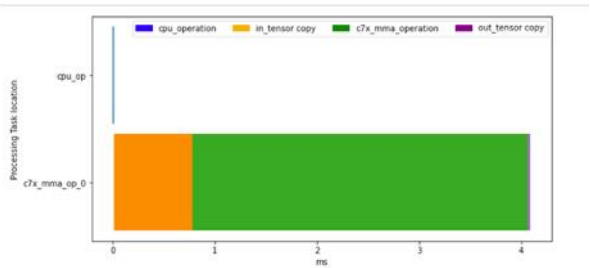
Hello world! | PC to TI Edge AI Processor Same Code



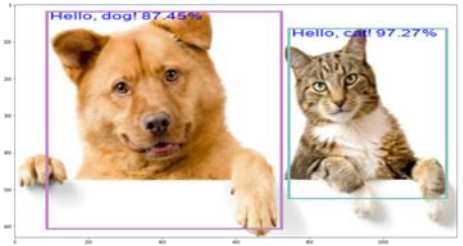

```
SoC: J721E/DRA829/TDA4VM
 OPP:
  Cortex-A72 @2GHZ
  DSP C7x-MMA @1GHZ
  DDR @4266 MT/s

 Inferences Per Second    :    2.81 fps
 Inference Time Per Image :  356.39 ms
 DDR usage Per Image      :    0.00 MB
```

- ❑ 2.81 fps AI processing (ONLY ARM)

CPU is involved only in the beginning for small amount of time (Blue)




```
SoC: J721E/DRA829/TDA4VM
 OPP:
  Cortex-A72 @2GHZ
  DSP C7x-MMA @1GHZ
  DDR @4266 MT/s

 Inferences Per Second    :  303.06 fps
 Inference Time Per Image :    3.30 ms
 DDR usage Per Image      :   13.51 MB
```



- ❑ 303.6 fps AI processing (TIDL acceleration)

CPU is involved only in the beginning for small amount of time (Blue)

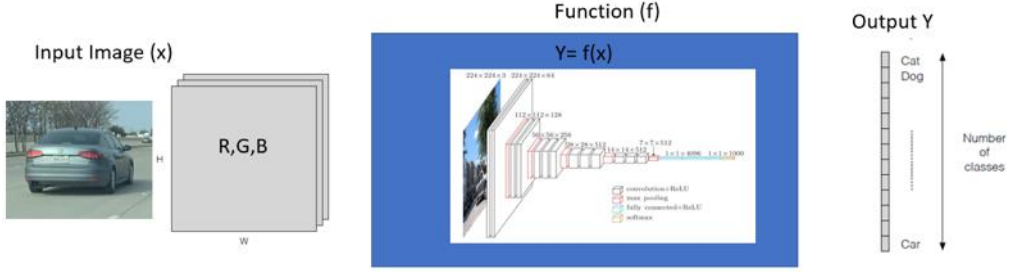
Previous webinar results: HelloWorld is working fine and faster. But, is there any difference?

Confidence in “Hello Dog” went down! 94.4% to 87.45%. We will address this in this webinar.

Deep dive into the deep learning model

Deep learning model | What is inside?

- Focusing on **vision analytics** in this webinar – More computationally intensive
- Convolution Neural Networks (CNN's) are universal functional approximators



f(.) ∈ [MobileNets, VGG16,Resnets,EfficientNets]

InceptionNet was named after movie "inception"

- Three primary functions

1. Classification

Image → CNN f (x) → Single Image label (car, person,…)

2. Object Detection (Hello World Example)

Image → CNN f (x) → Multiple ROI labels (car, person,…)

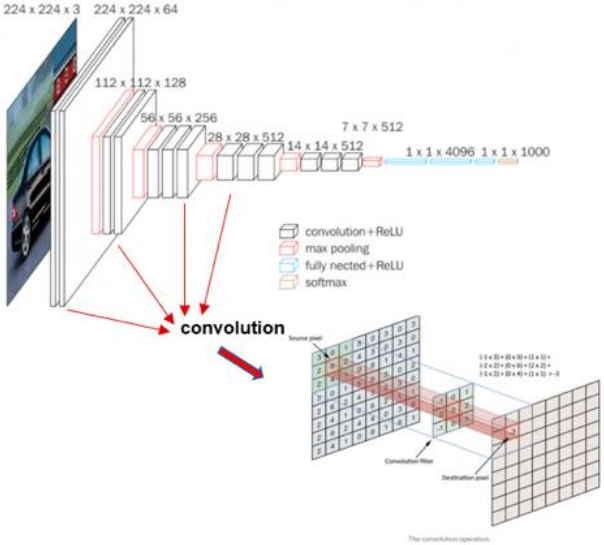
BoundingBox locations $[(bx_1, by_1, H_1, W_1), \ldots (bx_n, by_n, H_n, W_n)]$

3. Semantic Segmentation

Image → CNN f (x) → pixel label (car, person,…)

Deep Learning Model Metrics | TOPS, FPS and FPS/TOPS

- Basic kernel for a CNN is a convolution
- Convolutions can be efficiently done via matrix multiplication!
- Complexity of any network can be measured in MACs
- Scales linearly with input image resolution



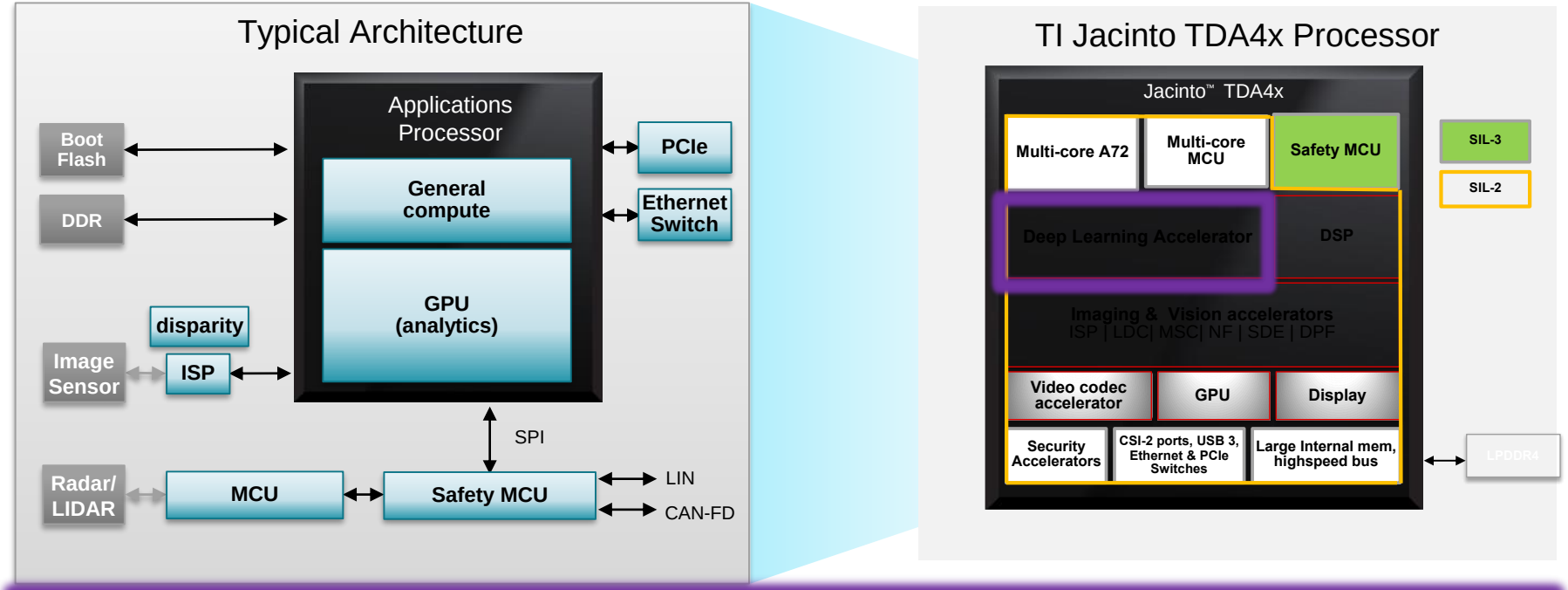
TDA4V Embedded EdgeAI processor

- Can compute 64x64 =4096 MACs/cycle
- 1 MAC = 2 Ops
- TDA4VMID @ 1GHz Frequency

  4096x2x1e9 = 8 Tera Ops (TOPs)

- More TOPs => more processing capability
  - However, not all TOPs are created equal
  - Not reflective of DL accelerators capability to run CNN's
- Better Performance Metric
  - Given a CNN and input resolution (pixels), how many Frames per Second (FPS) can be processed?
- Even better Metric
  - How many FPS per TOPS? Indicates architecture, energy efficiency

Model deployment | with purpose-built solutions



TIDL (TI Deep learning accelerator): AI in low power, AI with low complexity
Simple Linux based programming using popular software frameworks

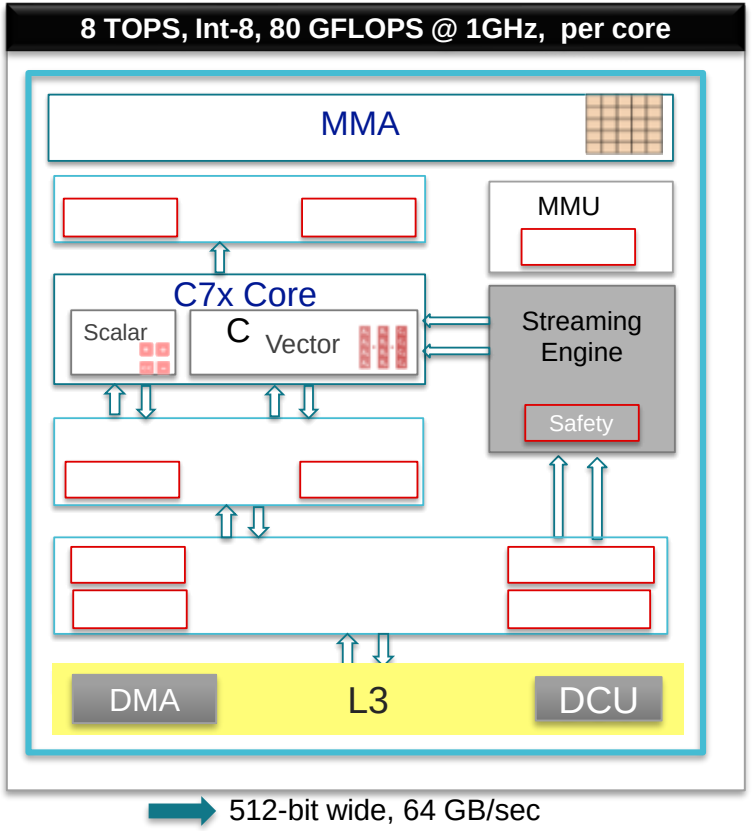
C7x + MMA | Industry's most efficient Deep Learning Accelerator

High FPS/TOPS
Designed for Lower power
Enables Fan-less design

Lowest #of DDR interfaces & bandwidth
SUPER-TILING
Using TI Proprietary Technology

Designed for Functional Safety
SIL 2
ECC on data memory

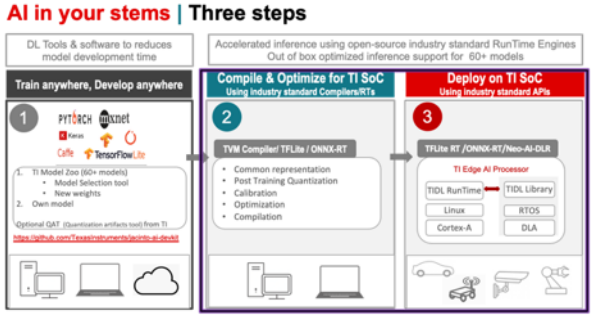
- C7x DSP + Matrix Multiply Accelerator (MMA)
  - Programable accelerator for tensor, vector and scalar processing
- Self sustained for DL work-loads
  - No dependency on host ARM, GPU, has its own DMA engine and memory sub-system
- **Smart memory architecture** results in up to 90% utilization of the accelerator and DDR BW savings
  - High bandwidth interconnect, Large internal memory, 4D programmable DMA, Data forward engine

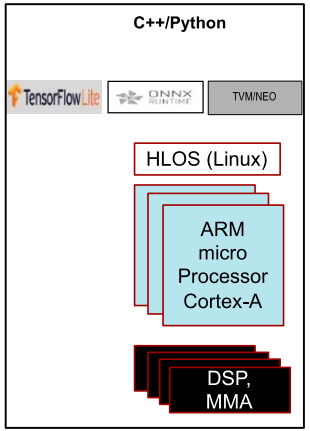


Model Compilation and Deployment

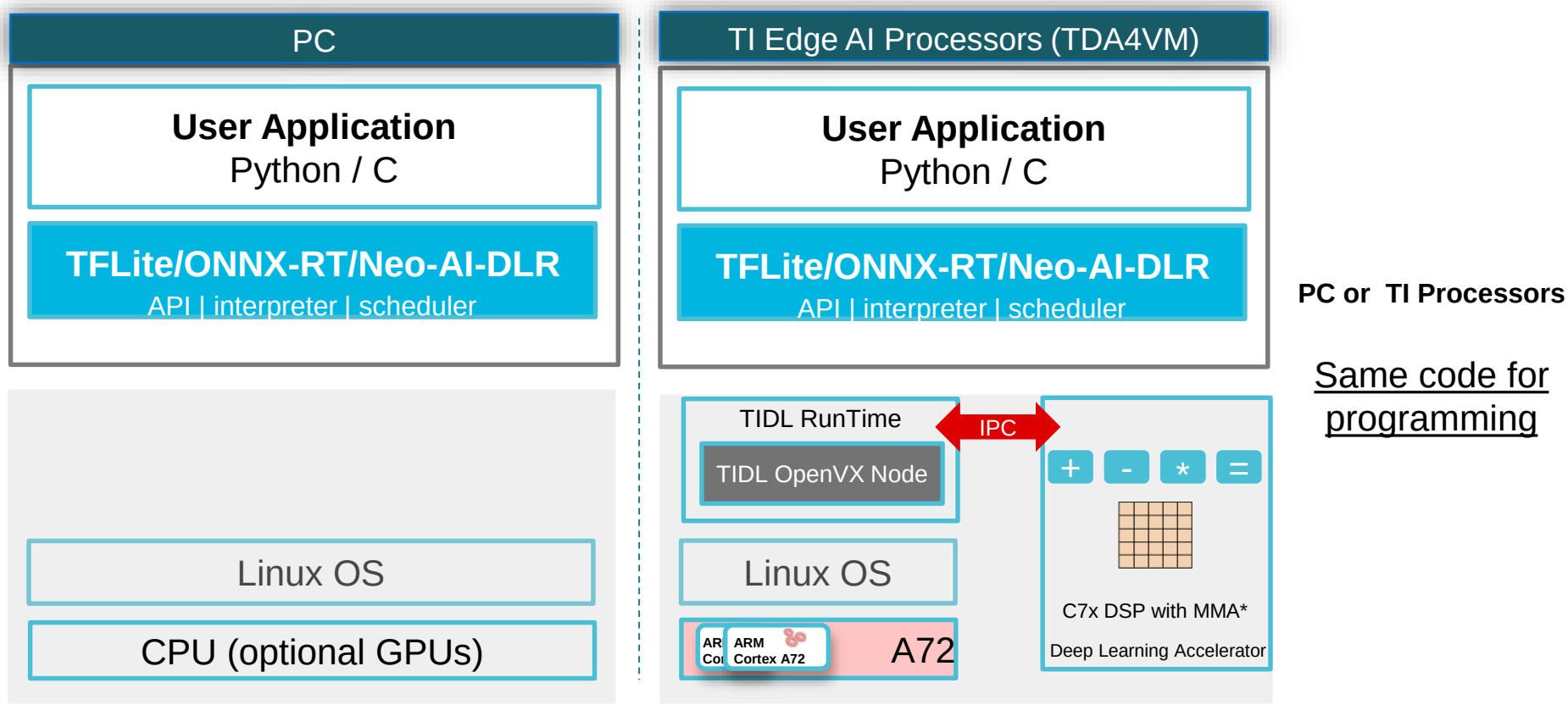
Model compilation | TI Deep Learning

- Application level programming (Python or C++)
- No need to learn any special CPU programming
- Flexibility to tweak compilation
- Support for popular deep learning run times
  - Tensorflow lite, ONNX, TVM





Common Development Environment | PC and TI Edge AI Processor



PC or TI Processors

Same code for programming

*MMA: Matrix Multiplication Accelerator (Tensor Processing Unit)

Development environment | Industry standard options

Training framework	Optimizer / Compiler / Graph format	RunTime Engine
TensorFlow	TensorFlowLite™ format	TFLite RunTime:: TIDL
	ONNX™ format	ONNX-RunTime:: TIDL
TensorFlowLite	TVM / SageMaker Neo™ compiler	Neo-AI-DLR::TIDL

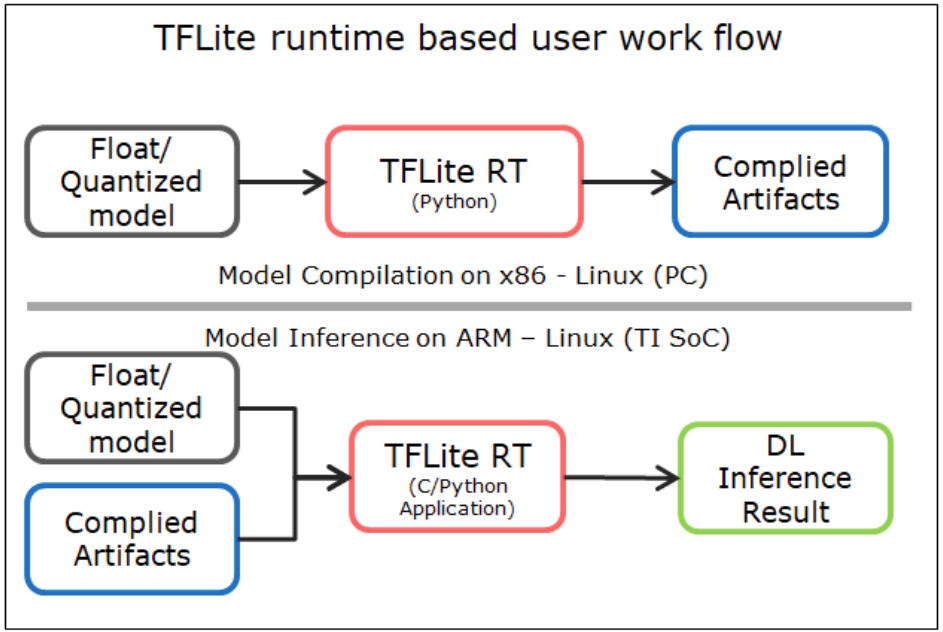
Easy Development Environment

- Linux Callable APIs
- Support of Python and C/C++ API
- Support of Multiple exchange formats
- Open source/Community run time interfaces
  - Tensor flow, TVM, ONNX Run time

Easy hardware deployment: cloud or EVM

Jupyter notebooks enable seamless access to EVM farm in a python environment

Model deployment | PC to embedded device



TFLite runtime based user work flow

Model compilation in TFlite runtime (Offline process)

- TFlite runtime as the top level inference API for user
- Offloading subgraphs to C7x/MMA for accelerated execution
- Multiple options to optimize accuracy and performance

Model Inferencing using TFlite runtime (Real-time)

- TFlite runtime as the top level inference API for user applications
- Uses “artifacts” from compilation to run the code on C7x/MMA

Very similar flow for other run-times (ONNX, TVM)

Open Source Run-Time Compilation Details

- All the details so far are just to get you more comfortable with the model deployment process.
- TI did the work for you to jump start your application
- TI's model zoon are already pre-compiled and ready to use

Model Compilation Options

Name	Description	Default values
platform	"J7"	"J7"
version	TIDL version - open source runtimes supported from version 7.2 onwards	(7,3)
tensor_bits	Number of bits for TIDL tensor and weights - 8/16	8
debug_level	0 - no debug, 1 - rt debug prints, >=2 - increasing levels of debug and trace dump	0
max_num_subgraphs	offload up to <num> tidl subgraphs	16
deny_list	force disable offload of a particular operator to TIDL	"" - Empty list
accuracy_level	0 - basic calibration, 1 - higher accuracy(advanced bias calibration), 9 - user defined [^3]	1
ti_internal_nc_flag	internal use only	-
advanced_options:calibration_frames	Number of frames to be used for calibration - min 10 frames recommended	20
advanced_options:calibration_iterations	Number of bias calibration iterations	50
advanced_options:output_feature_16bit_names_list	List of names of the layers (comma separated string) as in the original model whose feature/activation output user wants to be in 16 bit	""
advanced_options:params_16bit_names_list	List of names of the output layers (separated by comma or space or tab) as in the original model whose parameters user wants to be in 16 bit [^1] [^5]	""
advanced_options:quantization_scale_type	0 for non-power-of-2, 1 for power-of-2	0
advanced_options:high_resolution_optimization	0 for disable, 1 for enable	0
advanced_options:pre_batchnorm_fold	Fold batchnorm layer into following convolution layer, 0 for disable, 1 for enable	1

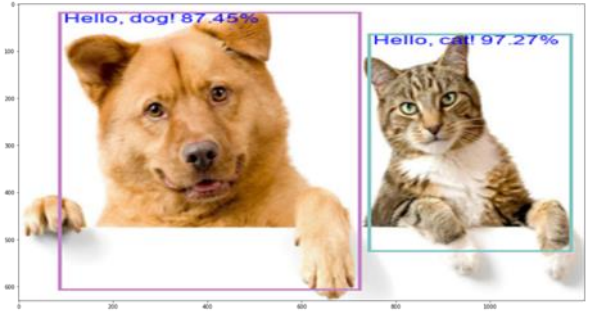
- All these options can be adjusted directly in Python or C- APIs
- We will look at couple of examples

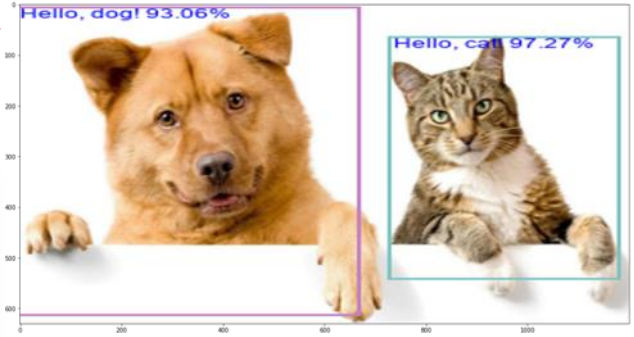
Hello world | advanced calibration

```
In [*]: calib_images = [
    '../sample-images/elephant.bmp',
    '../sample-images/bus.bmp',
    '../sample-images/bicycle.bmp',
    '../sample-images/zebra.bmp',
    ]

    # Make a dictionary of labels and class values
    label_dict = make_label_dict('TFL-OD-200-ssd-mobV1-coco-mlperf-300x300/labels.txt')

    output_dir = 'custom-artifacts_adv_calib'
    tflite_model_path = 'TFL-OD-200-ssd-mobV1-coco-mlperf-300x300/ssd_mobilenet_v1_coco_2018_01_28.tflite'

    compile_options = {
        'tidl_tools_path' : os.environ['TIDL_TOOLS_PATH'],
        'artifacts_folder' : output_dir,
        'tensor_bits' : 8,
        'accuracy_level' : 1,
        'advanced_options:calibration_frames' : len(calib_images),
        'advanced_options:calibration_iterations' : 20, #3,
    }

    # create the output dir if not present
```

Compile options: high_accuracy and increase calibration iterations

(In python example) **Just 1 field to change**





Confidence % now is higher!

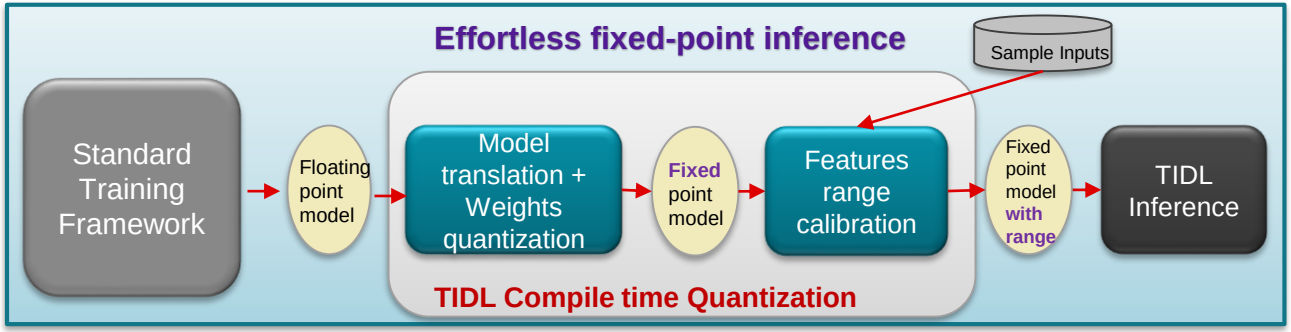
- Typically, you run this on large number of inputs to estimate accuracy.
- What is causing this? Quantization effects
- More calibration iterations minimizes the quantization error

Model Compilation | Quantization

- Floating-point inference are not cost and power-efficient so we need to quantize the model to Fixed-point
- Two options

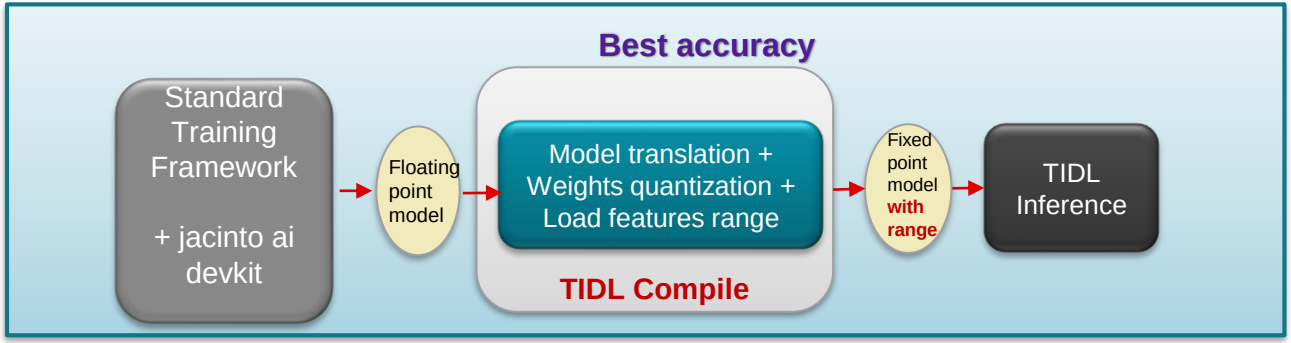
1. Post training Quantization

- 8-bit, 16-bit or mixed (layer-level)
- Histogram based activation range
- Adv parameter calibration



2. Quantization aware training

- Near lossless accuracy
- Just a few lines of code changes
- Jacinto_ai_dev_kit



Model Compilation | Output files

The compilation step generates multiple output files to examine the process

- It is recommended to see the log file to ensure all the layers are mapped
- Assess the performance requirements
- Review quantization effects
- Layer names
- Visualize graphs

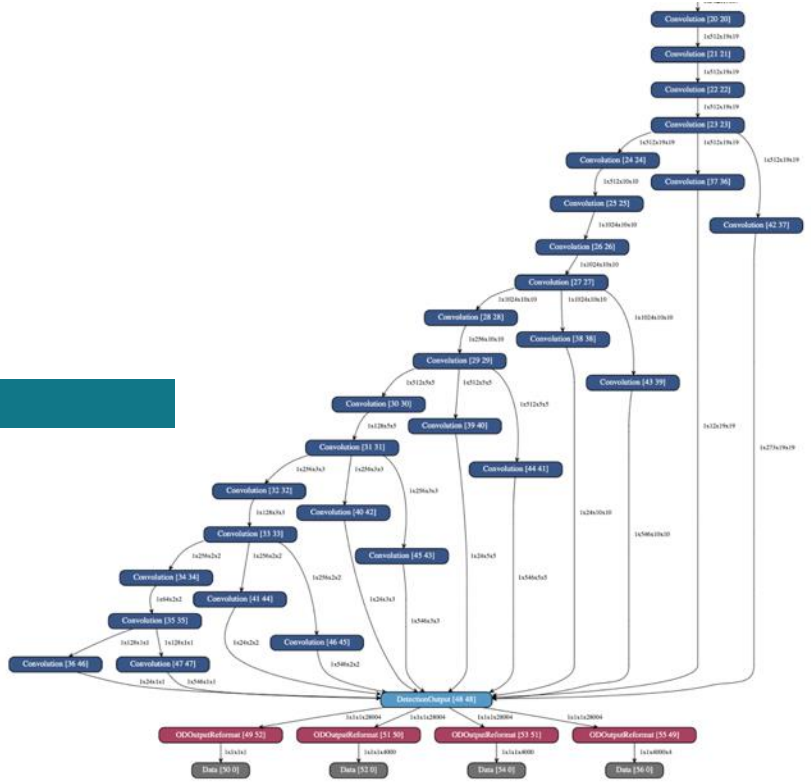
- **Compile output files**
  - 170_tidl_io_1.bin
  - 170_tidl_net.bin
  - allowedNode.txt
  - tempDir
    - 170_tidl_net
      - bufinfolog.csv
      - bufinfolog.txt
      - perSiminfo.bin
    - 170_tidl_io__LayerPerChannelMean.bin
    - 170_tidl_io__stats_tool_out.bin
    - 170_tidl_io_1.bin
    - 170_tidl_net.bin
    - 170_tidl_net.bin_netLog.txt
    - 170_tidl_net.bin_paramDebug.csv
    - 170_tidl_net.bin_layer_info.txt
    - 170_tidl_net.bin.svg
    - calib_raw_data_170.bin

Shows layer mapping
Total GMACs: 1.2371

Model compilation Graphical Output

• • •

170_tidl_net.bin.svg



Model compilation Layer Mapping Output

170_tidl_net.bin_netLog.txt

Num of Layer Detected : 57

Num	TIDL Layer Name	Out Data Name	Group	#Ins	#Outs	Inbuf Ids	Outbuf Id	In NCHW	Out NCHW	MACS
0	TIDL_DataLayer	normalized_input_image_tensor	0	-1	1	x x x x x x x x	0	0 0 0 0	1 3 300 300	0
1	TIDL_ConvolutionLayer	reExtractor/MobilenetV1/MobilenetV1/Conv2d_0/Relu6	0	1	1	0 x x x x x x x	1	1 3 300 300	1 32 150 150	19440000
2	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_1_depthwise/Relu6	0	1	1	1 x x x x x x x	2	1 32 150 150	1 32 150 150	6480000
3	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_1_pointwise/Relu6	0	1	1	2 x x x x x x x	3	1 32 150 150	1 64 150 150	46080000
4	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_2_depthwise/Relu6	0	1	1	3 x x x x x x x	4	1 64 150 150	1 64 75 75	3240000
5	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_2_pointwise/Relu6	0	1	1	4 x x x x x x x	5	1 64 75 75	1 128 75 75	46080000
6	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_3_depthwise/Relu6	0	1	1	5 x x x x x x x	6	1 128 75 75	1 128 75 75	6480000
7	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_3_pointwise/Relu6	0	1	1	6 x x x x x x x	7	1 128 75 75	1 128 75 75	92160000
8	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_4_depthwise/Relu6	0	1	1	7 x x x x x x x	8	1 128 75 75	1 128 38 38	1663488
9	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_4_pointwise/Relu6	0	1	1	8 x x x x x x x	9	1 128 38 38	1 256 38 38	47316992
10	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_5_depthwise/Relu6	0	1	1	9 x x x x x x x	10	1 256 38 38	1 256 38 38	3326976
11	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_5_pointwise/Relu6	0	1	1	10 x x x x x x x	11	1 256 38 38	1 256 38 38	94633984
12	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_6_depthwise/Relu6	0	1	1	11 x x x x x x x	12	1 256 38 38	1 256 19 19	831744
13	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_6_pointwise/Relu6	0	1	1	12 x x x x x x x	13	1 256 19 19	1 512 19 19	47316992
14	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_7_depthwise/Relu6	0	1	1	13 x x x x x x x	14	1 512 19 19	1 512 19 19	1663488
15	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_7_pointwise/Relu6	0	1	1	14 x x x x x x x	15	1 512 19 19	1 512 19 19	94633984
16	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_8_depthwise/Relu6	0	1	1	15 x x x x x x x	16	1 512 19 19	1 512 19 19	1663488
17	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_8_pointwise/Relu6	0	1	1	16 x x x x x x x	17	1 512 19 19	1 512 19 19	94633984
18	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_9_depthwise/Relu6	0	1	1	17 x x x x x x x	18	1 512 19 19	1 512 19 19	1663488
19	TIDL_ConvolutionLayer	r/MobilenetV1/MobilenetV1/Conv2d_9_pointwise/Relu6	0	1	1	18 x x x x x x x	19	1 512 19 19	1 512 19 19	94633984
20	TIDL_ConvolutionLayer	/MobilenetV1/MobilenetV1/Conv2d_10_depthwise/Relu6	0	1	1	19 x x x x x x x	20	1 512 19 19	1 512 19 19	1663488
21	TIDL_ConvolutionLayer	/MobilenetV1/MobilenetV1/Conv2d_10_pointwise/Relu6	0	1	1	20 x x x x x x x	21	1 512 19 19	1 512 19 19	94633984
22	TIDL_ConvolutionLayer	/MobilenetV1/MobilenetV1/Conv2d_11_depthwise/Relu6	0	1	1	21 x x x x x x x	22	1 512 19 19	1 512 19 19	1663488
23	TIDL_ConvolutionLayer	/MobilenetV1/MobilenetV1/Conv2d_11_pointwise/Relu6	0	1	1	22 x x x x x x x	23	1 512 19 19	1 512 19 19	94633984
24	TIDL_ConvolutionLayer	/MobilenetV1/MobilenetV1/Conv2d_12_depthwise/Relu6	0	1	1	23 x x x x x x x	24	1 512 19 19	1 512 10 10	460800
25	TIDL_ConvolutionLayer	/MobilenetV1/MobilenetV1/Conv2d_12_pointwise/Relu6	0	1	1	24 x x x x x x x	25	1 512 10 10	1 1024 10 10	52428800
26	TIDL_ConvolutionLayer	/MobilenetV1/MobilenetV1/Conv2d_13_depthwise/Relu6	0	1	1	25 x x x x x x x	26	1 1024 10 10	1 1024 10 10	921600
27	TIDL_ConvolutionLayer	/MobilenetV1/MobilenetV1/Conv2d_13_pointwise/Relu6	0	1	1	26 x x x x x x x	27	1 1024 10 10	1 1024 10 10	104857600
28	TIDL_ConvolutionLayer	netV1/Conv2d_13_pointwise_1_Conv2d_2_1x1_256/Relu6	0	1	1	27 x x x x x x x	28	1 1024 10 10	1 256 10 10	26214400
29	TIDL_ConvolutionLayer	V1/Conv2d_13_pointwise_2_Conv2d_2_3x3_s2_512/Relu6	0	1	1	28 x x x x x x x	29	1 256 10 10	1 512 5 5	29491200
30	TIDL_ConvolutionLayer	netV1/Conv2d_13_pointwise_1_Conv2d_3_1x1_128/Relu6	0	1	1	29 x x x x x x x	30	1 512 5 5	1 128 5 5	1638400
31	TIDL_ConvolutionLayer	V1/Conv2d_13_pointwise_2_Conv2d_3_3x3_s2_256/Relu6	0	1	1	30 x x x x x x x	31	1 128 5 5	1 256 3 3	2654208
32	TIDL_ConvolutionLayer	netV1/Conv2d_13_pointwise_1_Conv2d_4_1x1_128/Relu6	0	1	1	31 x x x x x x x	32	1 256 3 3	1 128 3 3	294912
33	TIDL_ConvolutionLayer	V1/Conv2d_13_pointwise_2_Conv2d_4_3x3_s2_256/Relu6	0	1	1	32 x x x x x x x	33	1 128 3 3	1 256 2 2	1179648
34	TIDL_ConvolutionLayer	enetV1/Conv2d_13_pointwise_1_Conv2d_5_1x1_64/Relu6	0	1	1	33 x x x x x x x	34	1 256 2 2	1 64 2 2	65536
35	TIDL_ConvolutionLayer	V1/Conv2d_13_pointwise_2_Conv2d_5_3x3_s2_128/Relu6	0	1	1	34 x x x x x x x	35	1 64 2 2	1 128 1 1	73728
36	TIDL_ConvolutionLayer	BoxPredictor_0/BoxEncodingPredictor/BiasAdd	0	1	1	23 x x x x x x x	36	1 512 19 19	1 12 19 19	2217984
37	TIDL_ConvolutionLayer	BoxPredictor_0/ClassPredictor/BiasAdd	0	1	1	23 x x x x x x x	37	1 512 19 19	1 273 19 19	50459136
38	TIDL_ConvolutionLayer	BoxPredictor_1/BoxEncodingPredictor/BiasAdd	0	1	1	27 x x x x x x x	38	1 1024 10 10	1 24 10 10	2457600
39	TIDL_ConvolutionLayer	BoxPredictor_1/ClassPredictor/BiasAdd	0	1	1	27 x x x x x x x	39	1 1024 10 10	1 546 10 10	55910400
40	TIDL_ConvolutionLayer	BoxPredictor_2/BoxEncodingPredictor/BiasAdd	0	1	1	29 x x x x x x x	40	1 512 5 5	1 24 5 5	307200
41	TIDL_ConvolutionLayer	BoxPredictor_2/ClassPredictor/BiasAdd	0	1	1	29 x x x x x x x	41	1 512 5 5	1 546 5 5	6988800
42	TIDL_ConvolutionLayer	BoxPredictor_3/BoxEncodingPredictor/BiasAdd	0	1	1	31 x x x x x x x	42	1 256 3 3	1 24 3 3	55296
43	TIDL_ConvolutionLayer	BoxPredictor_3/ClassPredictor/BiasAdd	0	1	1	31 x x x x x x x	43	1 256 3 3	1 546 3 3	1257984
44	TIDL_ConvolutionLayer	BoxPredictor_4/BoxEncodingPredictor/BiasAdd	0	1	1	33 x x x x x x x	44	1 256 2 2	1 24 2 2	24576
45	TIDL_ConvolutionLayer	BoxPredictor_4/ClassPredictor/BiasAdd	0	1	1	33 x x x x x x x	45	1 256 2 2	1 546 2 2	559104
46	TIDL_ConvolutionLayer	BoxPredictor_5/BoxEncodingPredictor/BiasAdd	0	1	1	35 x x x x x x x	46	1 128 1 1	1 24 1 1	3072
47	TIDL_ConvolutionLayer	BoxPredictor_5/ClassPredictor/BiasAdd	0	1	1	35 x x x x x x x	47	1 128 1 1	1 546 1 1	69888
48	TIDL_DetectionOutputLayer	TfLite_Detection_Process_Intermediate	0	12	1	36 38 40 42 44 46 37 39 41 43 45 47	48	1 12 19 19	1 1 1 28004	0
49	TIDL_TfODOutputLayer	TFLite_Detection_PostProcess	0	1	1	48 x x x x x x x	49	1 1 1 28004	1 1 4000 4	0
50	TIDL_TfODOutputLayer	TFLite_Detection_PostProcess:1	0	1	1	48 x x x x x x x	50	1 1 1 28004	1 1 1 4000	0
51	TIDL_TfODOutputLayer	TFLite_Detection_PostProcess:2	0	1	1	48 x x x x x x x	51	1 1 1 28004	1 1 1 4000	0
52	TIDL_TfODOutputLayer	TFLite_Detection_PostProcess:3	0	1	1	48 x x x x x x x	52	1 1 1 28004	1 1 1 1	0
53	TIDL_DataLayer	TFLite_Detection_PostProcess	0	1	-1	49 x x x x x x x	0	1 1 4000 4	0 0 0 0	0
54	TIDL_DataLayer	TFLite_Detection_PostProcess:1	0	1	-1	50 x x x x x x x	0	1 1 1 4000	0 0 0 0	0
55	TIDL_DataLayer	TFLite_Detection_PostProcess:2	0	1	-1	51 x x x x x x x	0	1 1 1 4000	0 0 0 0	0
56	TIDL_DataLayer	TFLite_Detection_PostProcess:3	0	1	-1	52 x x x x x x x	0	1 1 1 1	0 0 0 0	0

170_tidl_net.bin_layer_info.txt

Total Giga Macs : 1.2371

Debug Accuracy Differences | Weights Quantization statistic Analysis

- The import tool generates parameter quantization statistics.
  - This information is saved as "*_paramDebug.csv" in the same location as output TIDL model files.
  - This information calculated using the float weights and quantized weights
- The important information is mean and max of all the absolute float parameters and quantized numbers
  - User can compare this file with 16-bit and 8-bit parameters

8-bit

LayerId	meanDifference	maxDifference	meanOrigFloat	meanRelDifference	orgmax	quantizedMax	orgAtmaxDiff	quantizedAtMaxI	maxRelDifference	Scale
1	0.009079	0.030883	1.983426	1.058606	3.952945	3.922062	0.107803	0.092647	14.058746	32.380924
1	0.01045	0.057536	0.480229	3.202275	7.364561	7.307025	0.145063	0.172607	18.987646	17.380535
2	0.01229	0.045691	2.266234	1.13127	5.848518	5.802827	-0.292469	-0.274149	6.263741	21.885885
2	0.011207	0.030509	0.540566	4.234289	8.048687	8.048687	0.160545	0.188641	17.500296	15.903215

170_tidl_net.bin_paramDebug.csv

16-bit

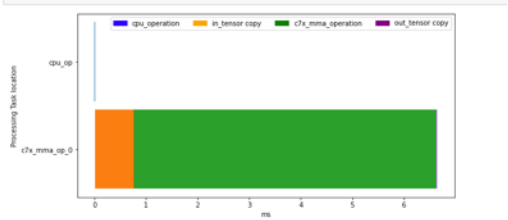
LayerId	meanDifference	maxDifference	meanOrigFloat	meanRelDifferen	orgmax	quantizedMax	orgAtmaxDiff	quantizedAtMaxI	maxRelDifference	Scale
1	0.000033	0.000118	1.970974	0.007077	3.882989	3.882871	0.018391	0.018367	0.130669	8438.85938
1	0.000041	0.000225	0.480229	0.103985	7.364561	7.364336	-0.001202	-0.001124	6.532568	4449.41699
2	0.000049	0.000178	2.181959	0.027657	5.836805	5.836627	0.014495	0.014428	0.46165	5614.02979
2	0.000059	0.000123	0.540566	1.037649	8.048687	8.048687	-0.000859	-0.000737	14.230615	4071.22315

Hello world | 16-bit quantization

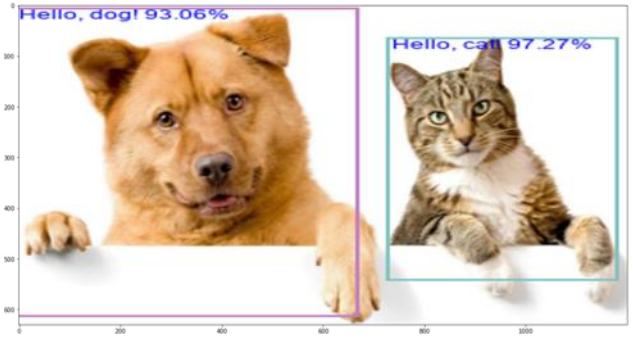
```
compile_options = {
    'tidl_tools_path' : os.environ['TIDL_TOOLS_PATH'],
    'artifacts_folder' : output_dir,
    'tensor_bits' : 16,
    'accuracy_level' : 1,
    'debug_level':2,
    'advanced_options:calibration_frames' : len(calib_images),
    'advanced_options:calibration_iterations' : 20, #50, #3,
    'advanced_options:high_resolution_optimization' : 1,
}
```

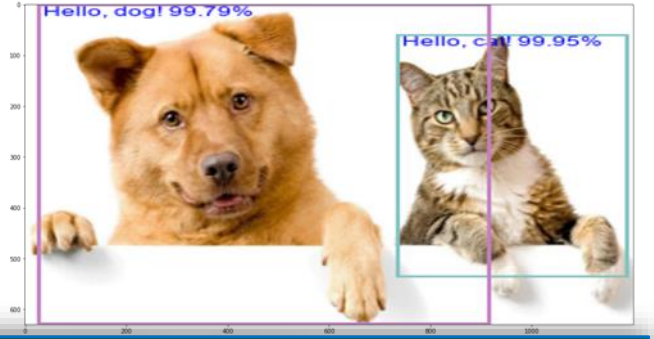
Change compile options to 16-bit tensor bits

(In python example)




```
SoC: J721E/DRA829/TDA4VM
 OPP:
  Cortex-A72 @2GHZ
  DSP C7x-MMA @1GHZ
  DDR @4266 MT/s

 Inferences Per Second    :  169.95 fps
 Inference Time Per Image :    5.88 ms
 DDR usage Per Image      :   19.97 MB
```





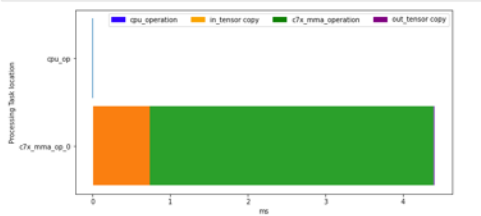
- Obviously, now the confidence % is much higher . But, it runs slower compared to 8-bit mode (169.95 fps vs 300 fps)
- In general, 8-bit computation is good enough for most practical applications.
- It's also very easy to select which layers can be 8-bit and which can be 16-bit

Hello world | Mixed 8-bit and 16-bit quantization

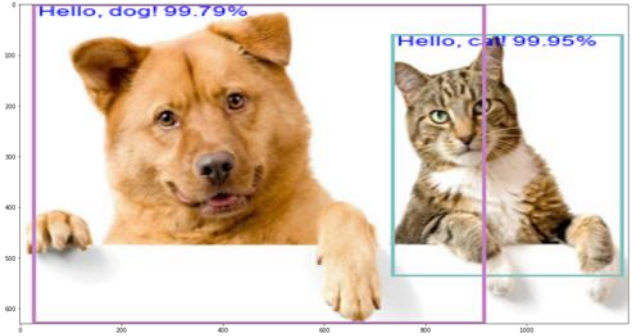
```
compile_options = {
    'tidl_tools_path' : os.environ['TIDL_TOOLS_PATH'],
    'artifacts_folder' : output_dir,
    'tensor_bits' : 8,
    'accuracy_level' : 1,
    'advanced_options:calibration_frames' : len(calib_images),
    'advanced_options:calibration_iterations' : 20,
    'advanced_options:params_16bit_names_list': 'FeatureExtractor/MobilenetV1/MobilenetV1/Conv2d_0/Relu6',
    'advanced_options:output_feature_16bit_names_list': 'FeatureExtractor/MobilenetV1/MobilenetV1/Conv2d_1_depthwise/Re
}
```

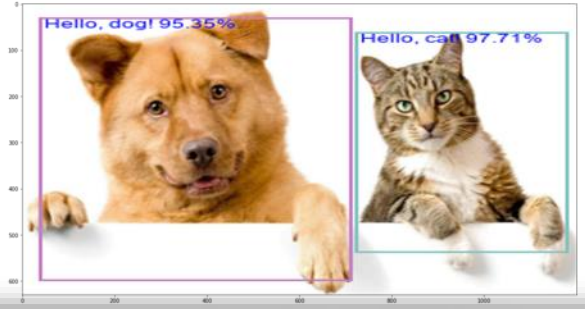
Change compile options to 16-bit tensor bits only for 1st layer

(In python example)




```
SoC: J721E/DRA829/TDA4VM
 OPP:
  Cortex-A72 @2GHz
  DSP C7x-MMA @1GHz
  DDR @4266 MT/s

 Inferences Per Second    :  273.13 fps
 Inference Time Per Image :    3.66 ms
 DDR usage Per Image      :   12.64 MB
```

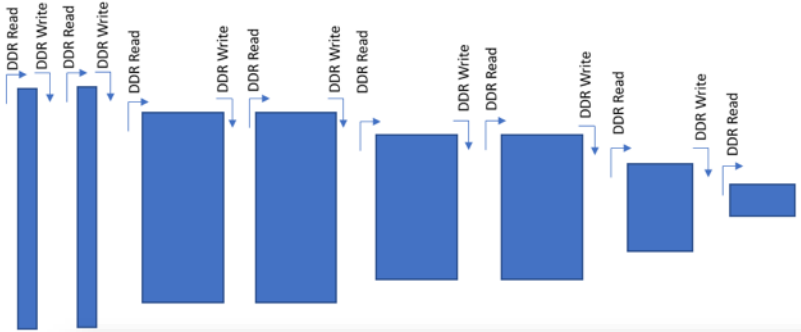




- Obviously, now the confidence % is slightly lower. But, it runs faster compared to 8-bit mode (273.13 fps vs 169.95 fps)
- It's very easy to select which layers can be 8-bit and which can be 16-bit
- And, all this is done in Python with just a few lines of code and examining the model compilation output files

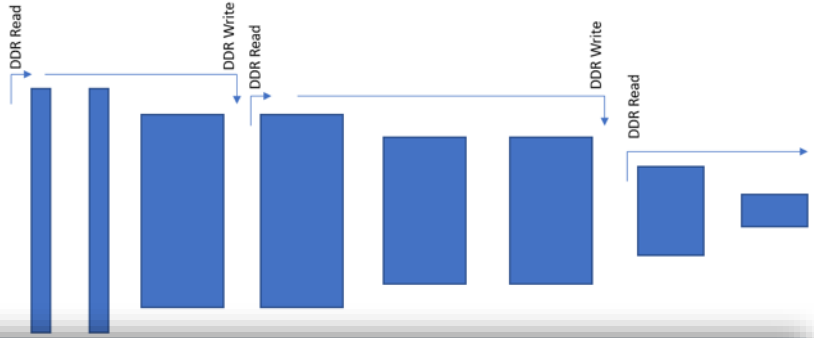
Parameters affecting Performance

- Convention DL engines
  - Process CNN's on a per layer basis
  - Results in a large DDR footprint (MB/Frame)
  - More DDR interfaces required to support DDR bandwidth consumption



- Super-Tiling is a TI proprietary technology
  - Optimizes memory management
  - Minimizes the number of DDR transactions
  - Enables SoC's with fewer DDR interfaces resulting in higher performance and better energy efficiency

Enabled by setting
advanced_options:high_resolution_optimization



Recommended for medium and large sized networks

Deep learnig model deployment | Supported Layers

1. Convolution Layer
2. Spatial Pooling Layer
   - Average and Max Pooling
3. Global Pooling Layer
   - Average and Max Pooling
4. Element Wise Layer
   - Add, Product and Max
5. Inner-Product (FC/Dense/Matmul) Layer
6. Soft-Max Layer
7. Bias Add Layer
8. Concatenate layer
9. Scale Layer
10. Batch Normalization layer
11. Re-size Layer
    - Bi-leaner/Nearest Neighbor Up-sample
12. Arg-max layer
13. ReLU Layer
14. RelU6 layer
15. PReLU (One Parameter per channel)
16. Slice layer
17. Crop layer
18. Flatten layer
19. Shuffle Channel Layer
20. Detection output Layer (**SSD - Post Processing As defined in caffe-Jacinto and TF Object detection API**)
21. Deconvolution/Transpose convolution
22. Custom/ User Defined Layer (Call Back)

Note : Please refer to TIDL users Guide for up to date information
https://software-dl.ti.com/jacinto7/esd/processor-sdk-rtos-jacinto7/07_03_00_07/exports/docs/tidl_j7_02_00_00_07/ti_dl/docs/user_guide_html/index.html

Model Compilation Advanced scenarios

What if a layer is not supported?

MobileNetV3 | Evolution

- This is a popular network from GoogleAI/GoogleBrain
  - The Table1 shows various layers in this network
- Mobilenet v1: Introduced depth-wise plus point-wise convolution reducing the computation cost significantly [Used this in Hello World example]
- Mobilenet v2: Added residual connections for faster training and better accuracy. [similar idea as in Resnet]
- Mobilenet v3: Add a few more tricks including:
  - Redesign of expensive layers
  - SE (Squeeze-and-Excitation): to capture interactions between the channels
  - HS (HardSwish, a non-linear activation function) in the deeper layers vs ReLU
- HardSwish and SE are not supported by TIDL

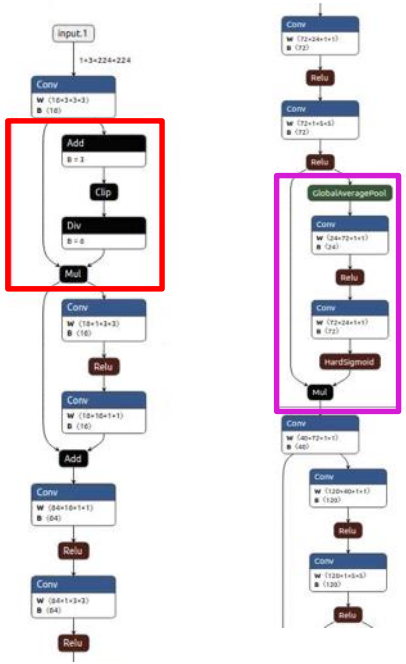
Input	Operator	exp size	#out	SE	NL	s
$224^2 \times 3$	conv2d	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	-	RE	1
$112^2 \times 16$	bneck, 3x3	64	24	-	RE	2
$56^2 \times 24$	bneck, 3x3	72	24	-	RE	1
$56^2 \times 24$	bneck, 5x5	72	40	✓	RE	2
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 3x3	240	80	-	HS	2
$14^2 \times 80$	bneck, 3x3	200	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	480	112	✓	HS	1
$14^2 \times 112$	bneck, 3x3	672	112	✓	HS	1
$14^2 \times 112$	bneck, 5x5	672	160	✓	HS	2
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	conv2d, 1x1	-	960	-	HS	1
$7^2 \times 960$	pool, 7x7	-	-	-	-	1
$1^2 \times 960$	conv2d 1x1, NBN	-	1280	-	HS	1
$1^2 \times 1280$	conv2d 1x1, NBN	-	k	-	-	1

Table 1. Specification for MobileNetV3-Large. SE denotes whether there is a Squeeze-And-Excite in that block. NL denotes the type of nonlinearity used. Here, HS denotes h-swish and RE denotes ReLU. NBN denotes no batch normalization. s denotes stride.

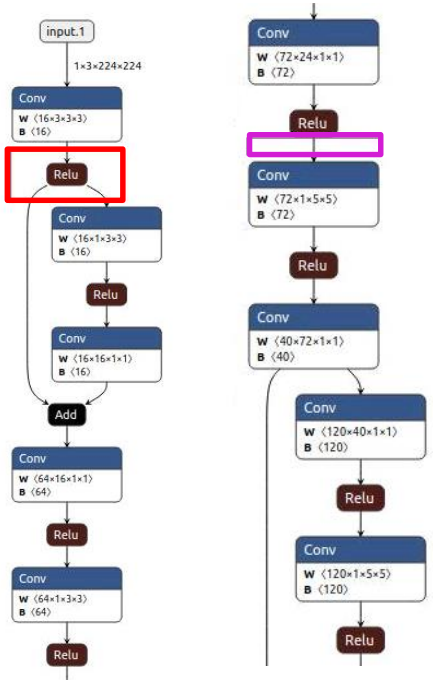
https://arxiv.org/pdf/1905.02244.pdf

It is very easy to make small changes to deploy this model

MobileNetV3 | Model analysis and changes



1. Red box: HardSwish is replaced by ReLU
2. Purple box: SE has been removed



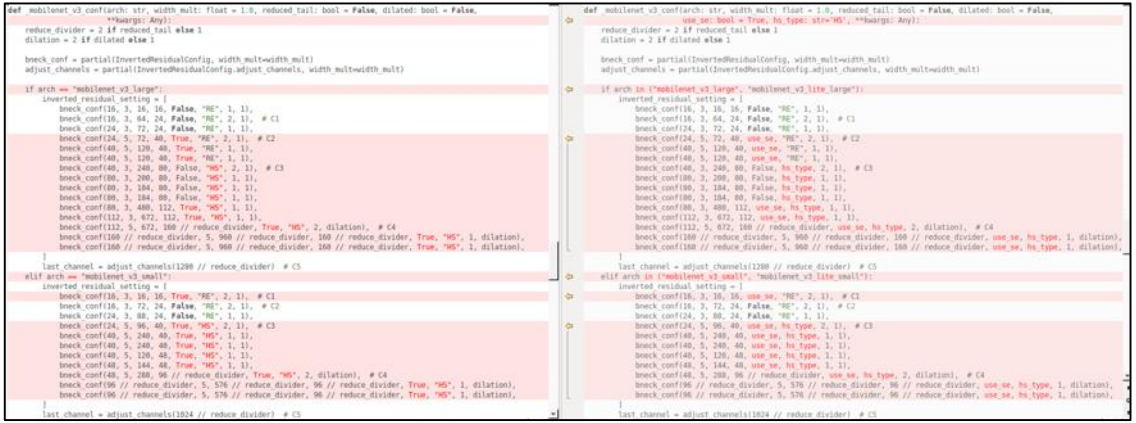
- Use Netron app to visualize this
- https://www.electronjs.org/apps/netron

- MobileNetV3 is implemented in torchvision
- https://github.com/pytorch/vision/blob/master/torchvision/models/mobilenetv3.py

Simple Model changes | MobileNetV3 Lite

- The pictures show the changes that was done to:
  - Replace HardSwish by ReLU
  - Remove SE

- Source code is also available as a reference



Accuracy Check | After MobileNetV3 changes

- Training Scripts for ImageNet Classification are given in torchvision
  - https://github.com/pytorch/vision/tree/master/references/classification
  - By default, we can train the model mobilenet_v3_large using those scripts.
    - The resulting accuracy is obtained in this page: https://pytorch.org/vision/stable/models.html

- Now, redo using the modified model
  - The accuracy drop due to lite version is only 2%
  - This is a reasonable accuracy hit to take since the lite version will run optimally on the embedded devices

Model Name	ImageNet Accuracy % (Top-1)
mobilenet_v3_large	74.042
mobilenet_v3_lite_large (Compiled)	72.122

The model is compiled, optimized and ready to be deployed!
Similarly, we have 100+ models to choose

- All the details so far are just to get you more comfortable with the model deployment process.
- For your application, you typically do not need to experiment with this a lot
- TI did the work for you to jump start your application

Model Zoo Get to market faster

Performance and Accuracy Results

TI model zoo | Jump start your AI development

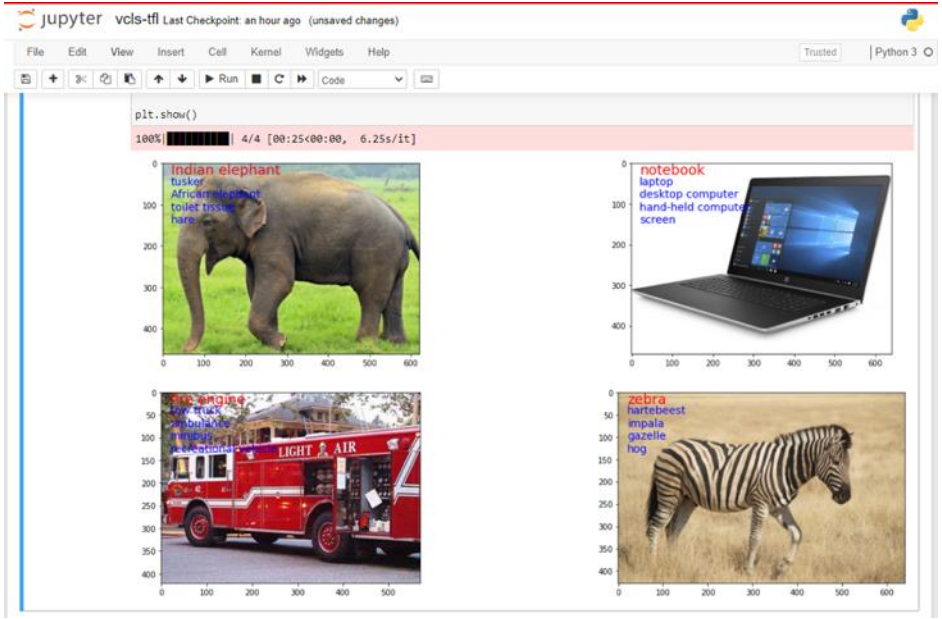
- Deep learning community is pretty extensive. It is common practice to:
  - Use architectures of networks published in literature
  - Use open-source implementations
    - Popular runtimes; Tensorflow, ONNX and TVM
  - Use pretrained models and fine-tune on your datasets (Transfer Learning)

- How TI is making it easy?
  - Pre-compiled models (100+ now, continuously adding)
  - Provide all the scripts to benchmark under top popular runtimes
  - Provide the scripts to do transfer learning
  - Documentation: **link** git clone/pull URLs: link

In most cases, you can pick a model that compiles without changes.
Let us know if any specific model is of interest

Classification | Accuracy

- ImageNet Database(ILSVRC-2012)
  - 1000 Classes (zebras, elephant truck,…)
  - 14,000,000 labelled Images
- Classification accuracy
  - Top 1 (most popular)
  - Top 5 (one of the top 5 is correct)
- You can run this today on EdgeAI Cloud tool

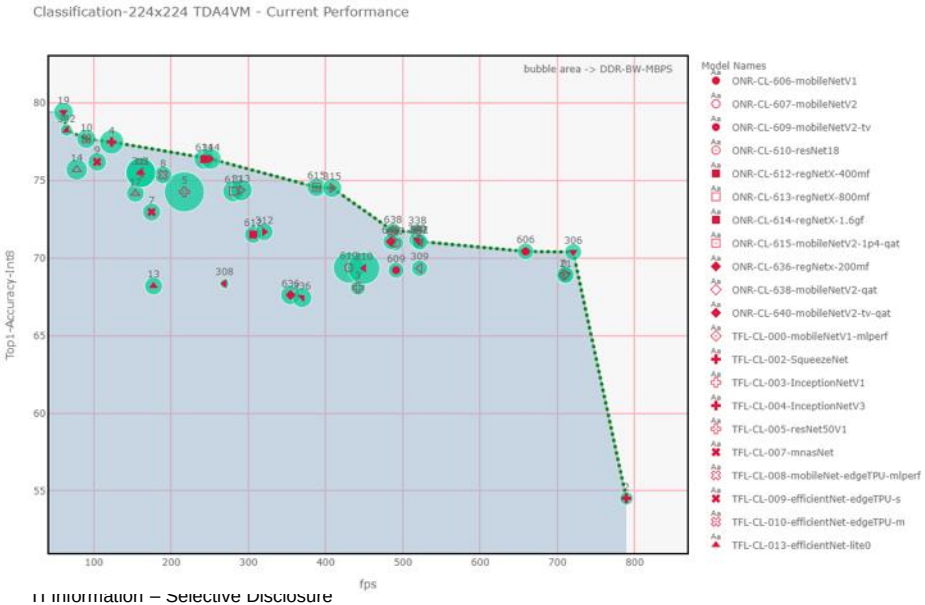


Classification performance of MobileNetV1 on sample ImageNet pictures

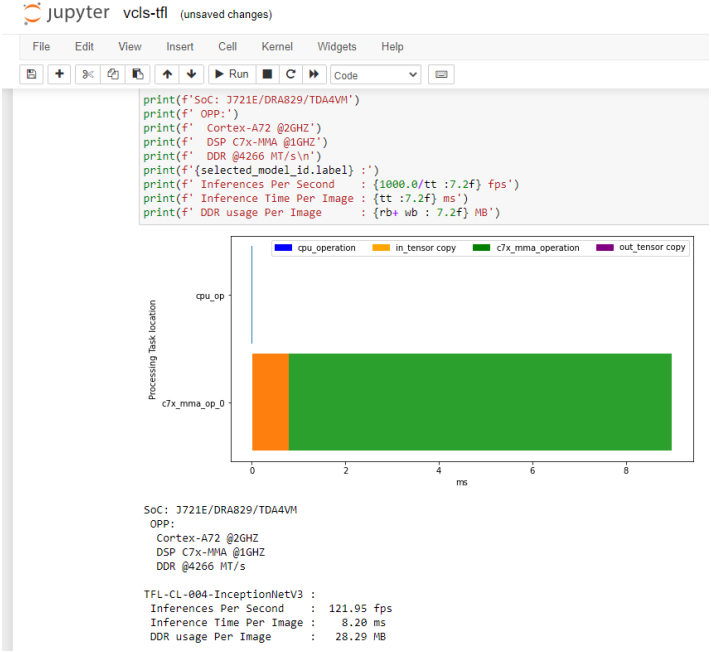
https://dev.ti.com/edgeaisession/

Classification | Performance

- Model Selection Tool enables quick comparison between CNN accuracy vs fps



- Obtain FPS and DDR Bandwidth utilization!



Object Detection | Accuracy

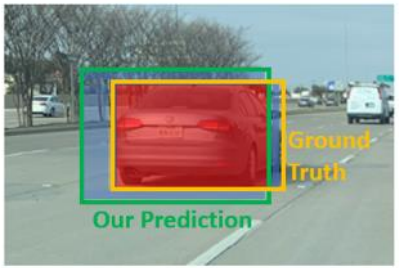
- OD networks output:
  - Bounding Boxes
  - Class: (Car, Sign…)

- True Positive (TP): Object Identified and IOU > IOU Threshold
- False Positive(FP): Object Identified and IOU < IOU Threshold
- False Negative(FN): Object missed
- Precision= $\frac{TP}{TP+FP}$; Recall= $\frac{TP}{TP+FN}$

- Bounding Box Performance:

Intersection over Union (IOU)

IOU= $\frac{Area\ of\ Intersection}{Area\ of\ Union}$



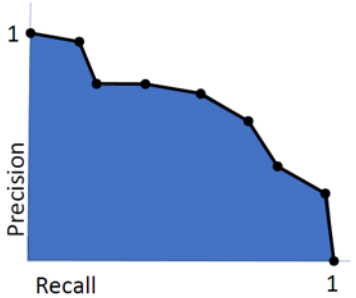
IOU > 0.5 average

IOU > 0.7 good

IOU > 0.9 excellent

- mean Average Precision (mAP 0.5:0.95)
  - For IOU thresholds 0.5:0.95
  - Compute Precision/ Recall

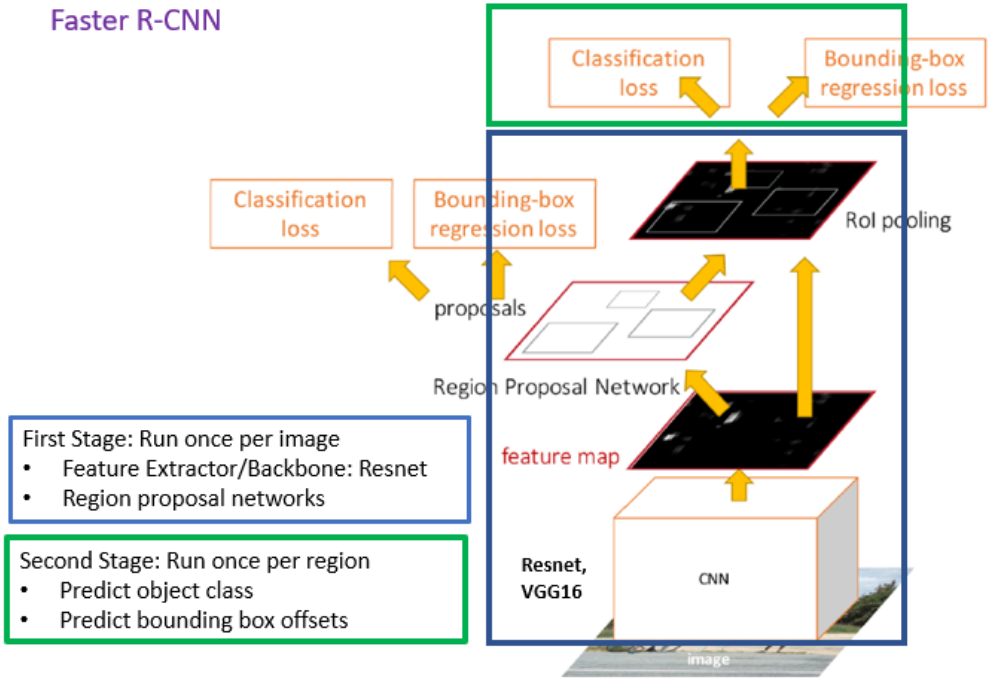
mAP is area under the Precision/Recall Curve



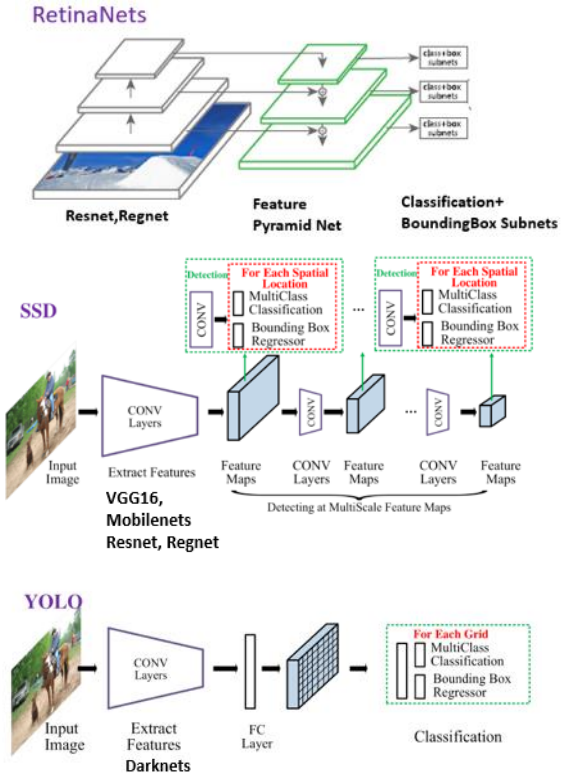
- Main Data set is the MS COCO dataset

Object Detection | Types of Networks

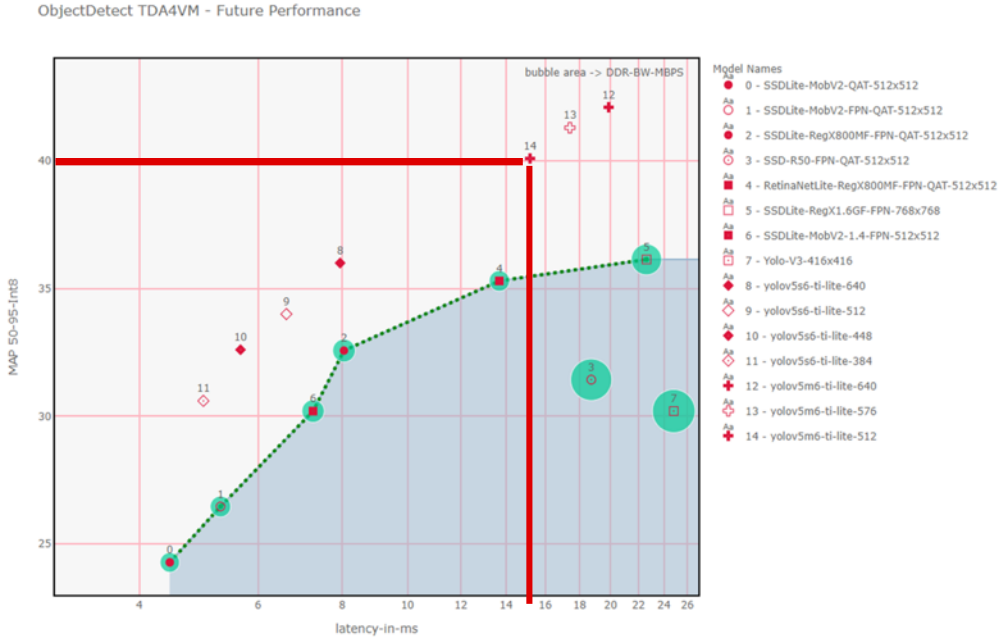
Two Stage Networks



One Stage Networks



Object Detection | Accuracy Metrics



Example: Yolo5m6-ti-lite (512x512)
MAP(50-95 Int8): 40
FPS = (1000/15ms)~ 66fps

Semantic Segmentation | Accuracy

- Label every pixel with a category label

- Important for
  - Drivable space estimation and occupancy grid
  - Assist object detection

- Feature extractor networks
  - MobileNets, Resnets, Regnets

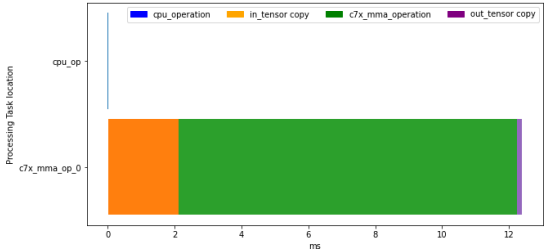
- True Positive (TP) : Number of correctly classified pixel
- False Positive (FP) : Number of pixels of class Y incorrectly assigned to class X (ground truth)
- False Negative (FN): The number of pixels of class X (ground truth) incorrectly assigned to class Y

$$\text{IOU}_{\text{Class}} = \frac{TP}{TP+FP+FN}$$

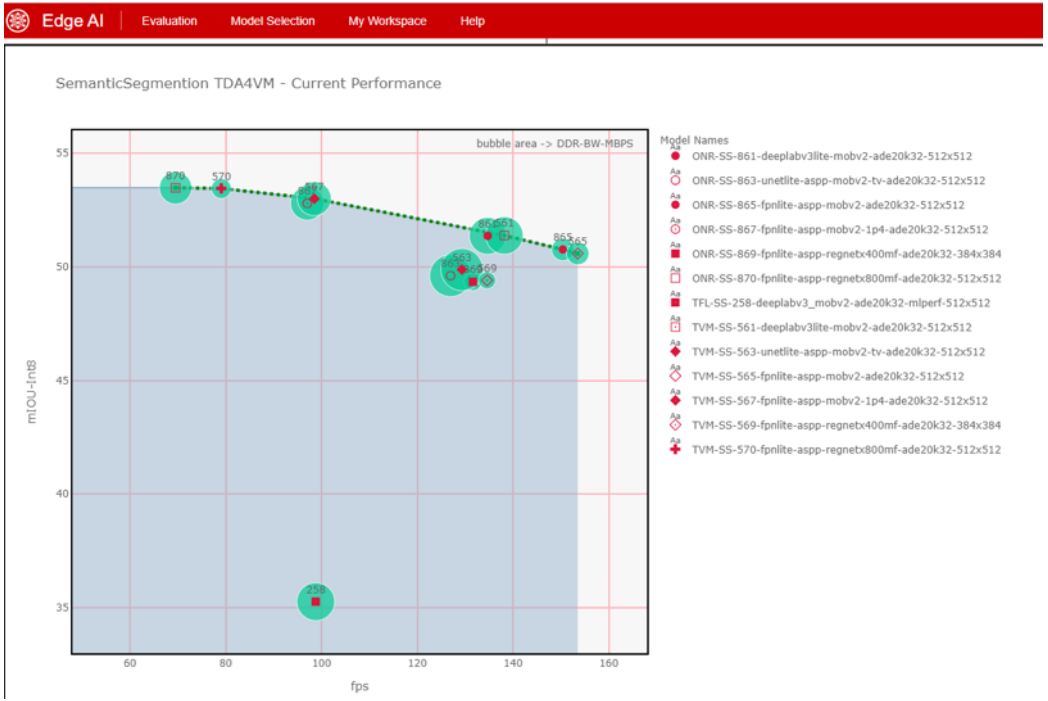
- mean IOU is average IOU across classes

Semantic Segmentation | Performance

- FPS and DDR BW utilization




```
SoC: J721E/DRA829/TDA4VM
 OPP:
  Cortex-A72 @2GHZ
  DSP C7x-MMA @1GHZ
  DDR @4266 MT/s

TFL-SS-258-deeplabv3_mobv2-ade20k32-mlperf-512x512 :
 Inferences Per Second    :   98.84 fps
 Inference Time Per Image :   10.12 ms
 DDR usage Per Image      :   48.81 MB
```



Model deployment Metrics | Summary

1. Accuracy
   - Different metrics for different tasks
   - Classification Networks : Top 1 or Top 5 metric
   - Object Detection Networks: mean Average Precision (mAP)
   - Semantic Segmentation Networks: mean Intersection over Union (mIOU)

2. Frames Per Second / Latency in ms
   - Higher FPS /Lower latency is better

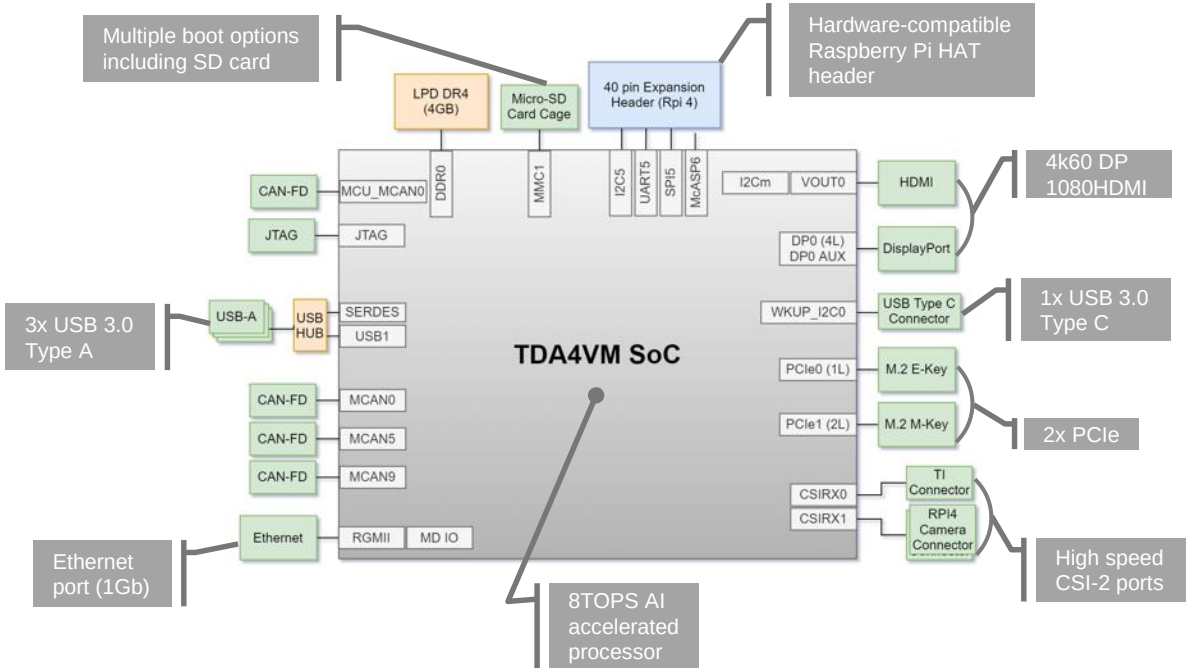
3. DDR BW utilization
   - Lower DDR BW/ Frame utilization is better

All this can be evaluated in the Cloud Evaluation Tool – Now!

Cloud Development to Your own hardware

Low-cost development tool from TI

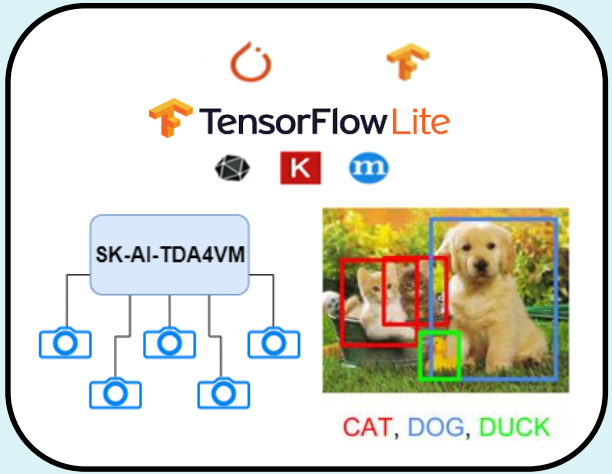
Edge AI Starter Kit | Jacinto™ TDA4VM processor



Fast out-of-box Edge AI demo:

1. Insert programmed SD card*
2. Plug-in all peripherals
3. Run demo in under an hour!

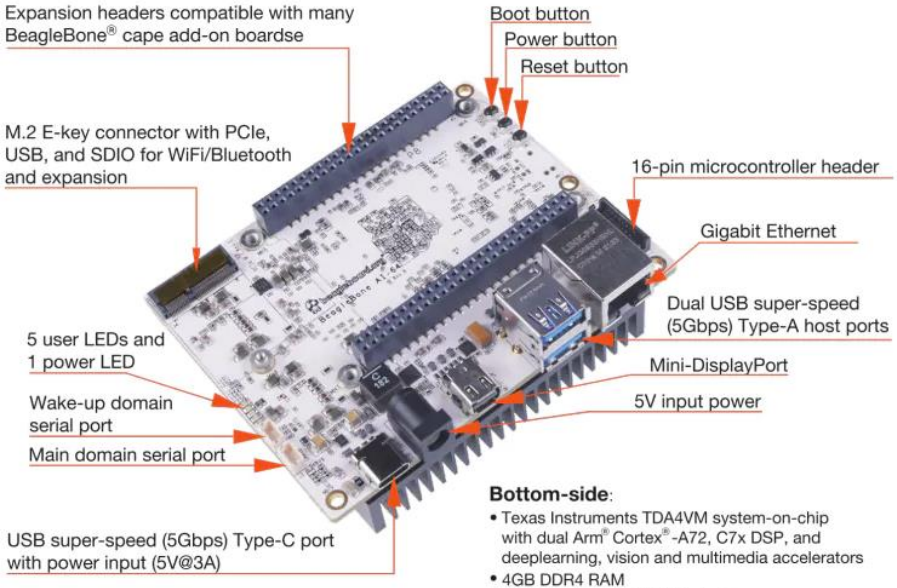
** Follow instructions in Edge AI Devkit to program SD card*



Part number: SK-AI-TDA4VM | **Price:** $249 | **Order:** Now

Quick start guide: Download | **Processor SDK:** Download

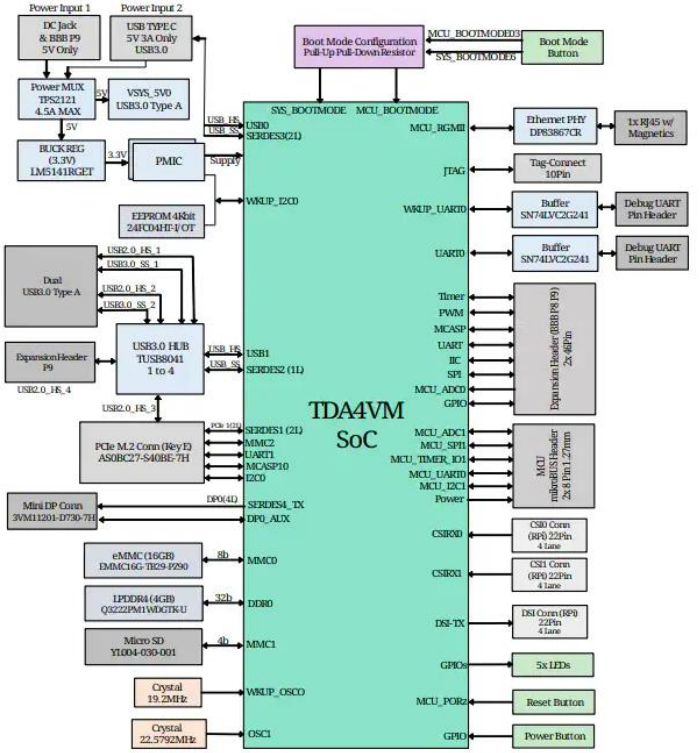
Beaglebone AI-64 | Jacinto™ TDA4VM processor



Bottom-side:

- Texas Instruments TDA4VM system-on-chip with dual Arm® Cortex®-A72, C7x DSP, and deeplearning, vision and multimedia accelerators
- 4GB DDR4 RAM
- 16GB on-board eMMC flash storage
- Micro-SD slot
- Dual CSI-2 camera connectors

BLOCK DIAGRAM



Price: $187.5 | https://beagleboard.org/

Summary | Deep Learning Deployment

Deep learning has two parts: Training and Inference

Training is done once or a few times : Off-line operation

Inference happens during the life of the product : Real-time operation

TI's Jacinto TDA4 EdgeAI Embedded Solutions Offer:

- ✓ **Efficiency:** Most efficient (FPS/TOPS, FPS/Watt) real-time inferencing in Edge Devices
- ✓ **Ready to use:** An extensive library of compiled models that are production redy
- ✓ **Easy to use:** Open-source runtime enabling simple Python and C based inferencing

Call to action

- Recommendations for further development
  - Compile different models with different options and see the effect on the inference results
  - Review the published results with your own results
  - Pick the models that would be relevant for your use case.
- Reimagine “what’s possible” for your application with embedded edge AI
  - Cloud Tool: https://dev.ti.com/edgeai
  - Product Folder: https://www.ti.com/product/TDA4VM
  - TDA4 EVM: http://www.ti.com/tool/TDA4VMXEVM
  - TDA4 SK EVM : https://www.ti.com/tool/SK-TDA4VM
- Contact TI for support (e2e.ti.com)
  - Please also let us know any specific topics you want us to cover in the future webinars

Code examples used in the webinar are below.

https://e2e.ti.com/support/processors-group/processors/f/processors-forum/1014084/tda4vm-jacinto-ti-edge-ai-monthly-webinar-jul-2021-embedded-deep-learning-deployment-demystified