DEVELOPING REAL-TIME NEURAL NETWORKS FOR JETSON

John Welsh, 3/31/2020

OVERVIEW

What is Jetson?

Hardware, software, and ecosystem. Enabling AI at the edge.

Optimizing with TensorRT

From PyTorch to TensorFlow. Worfklows for optimizing existing models. The benefits on Jetson.

Designing for real-time

Pragmatic tips to consider when creating neural networks for a real-time task.

WHAT IS NVIDIA JETSON?

JETSON AI COMPUTER LINEUP

AI Platform for Entry, Mainstream, and Fully Autonomous Edge Devices

JETSON NANO

0.5 TFLOPS (FP16)
5-10W
45mm x 70mm
$129

JETSON TX2 series
(TX2, TX2 4GB, TX2i*)

1.3 TFLOPS (FP16)
7.5-15W*
50mm x 87mm
Starting at $249

JETSON XAVIER NX

6 TFLOPS (FP16) | 21 TOPS (INT8)
10-15W
45mm x 70mm
$399

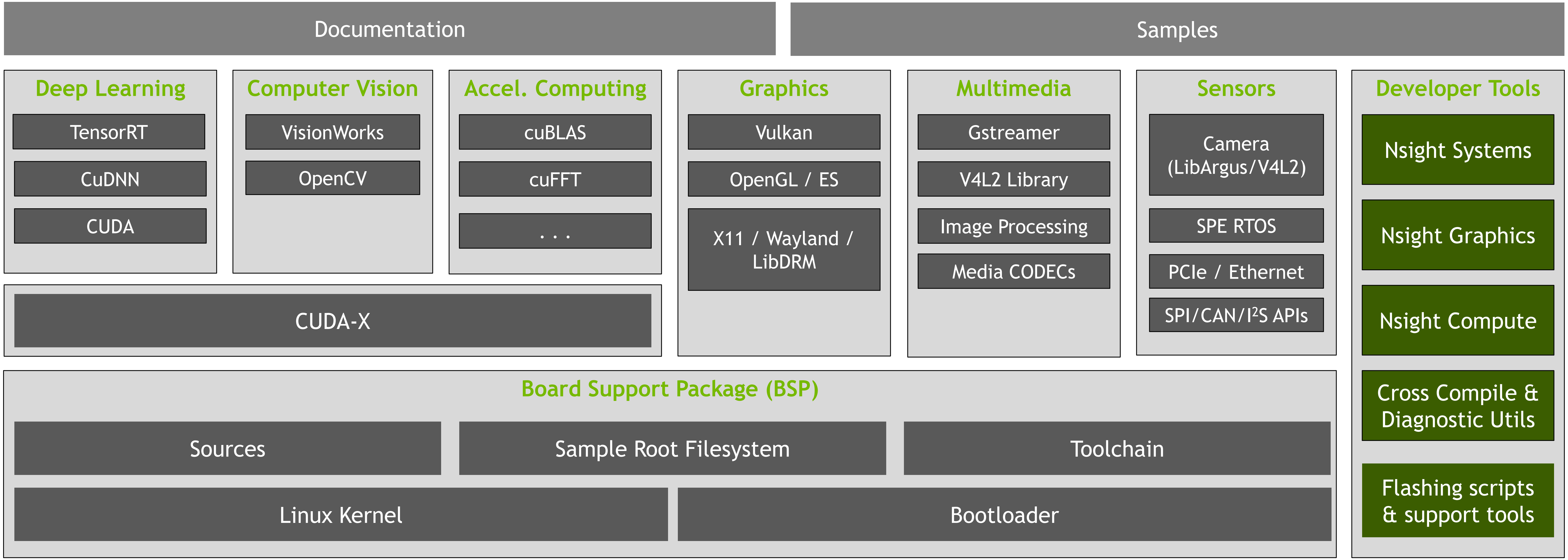
JETSON AGX XAVIER series
(AGX Xavier, Xavier ind.)

20-32 TOPS (INT8)
5.5-11 TFLOPS (FP16)
10-30W*
100mm x 87mm
Starting at $899

One Software Architecture

JETPACK

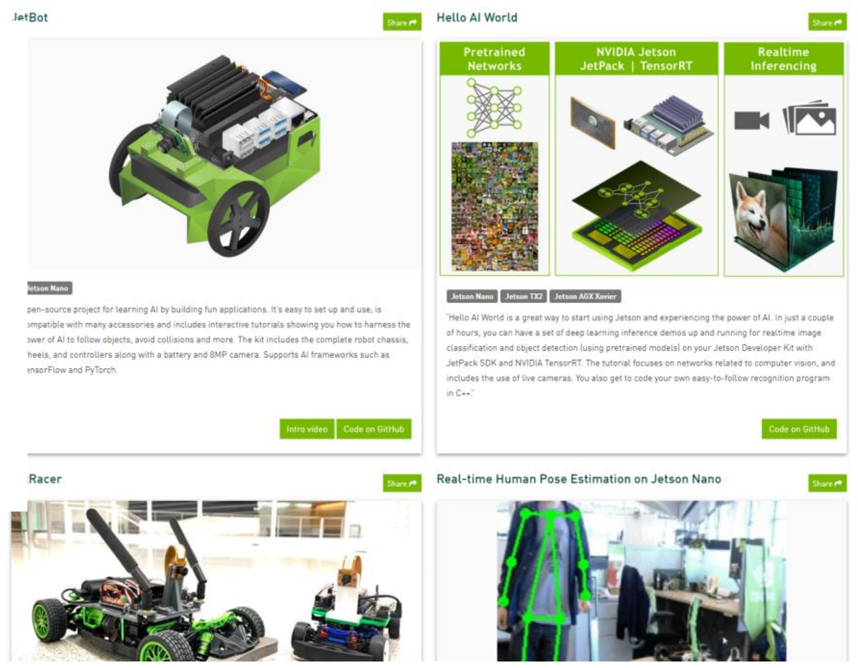
ALLin-One Software Development Kit

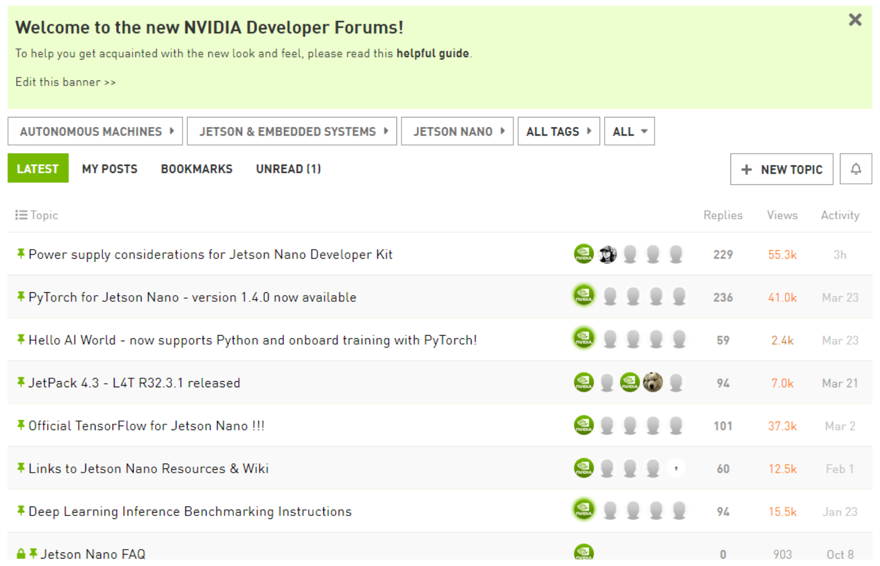


JETSON ECOSYSTEM

Strong and growing

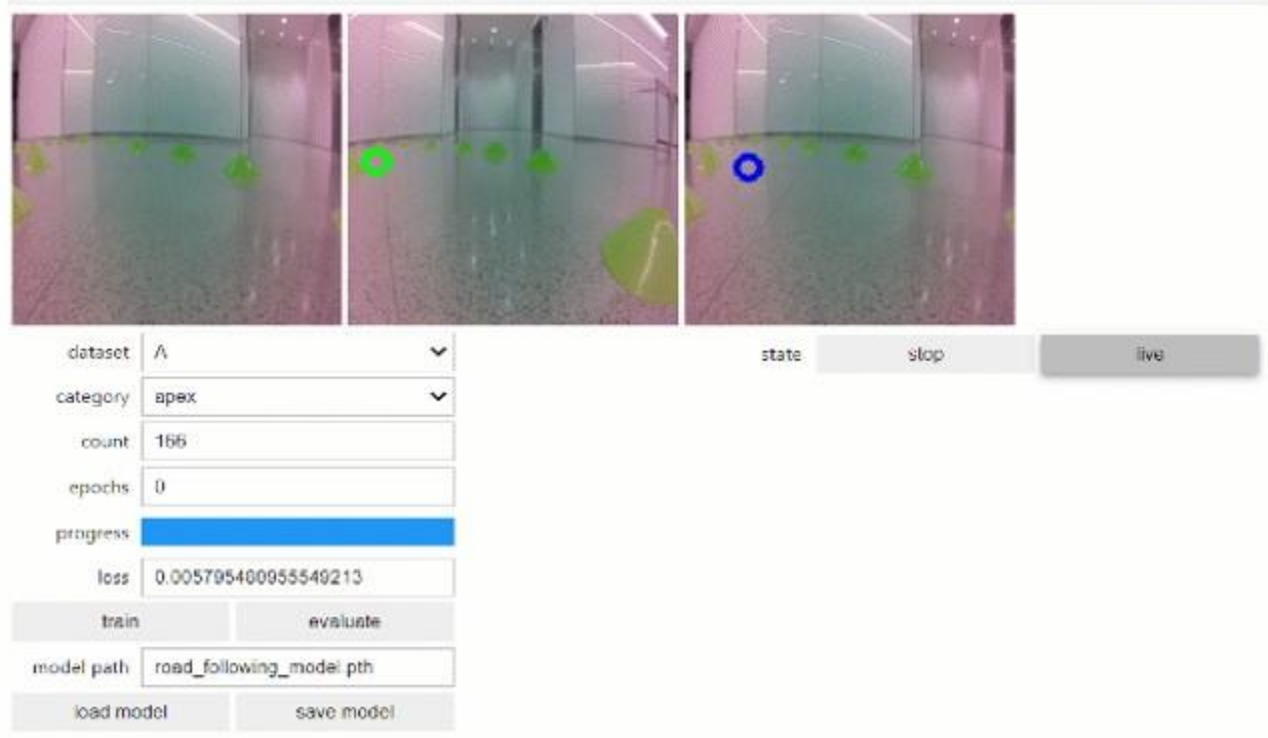
- Open source projects
  - Jetson Projects
  - NVIDIA-AI-IOT GitHub (github.com/NVIDIA-AI-IOT)
- Developer forums
- Developer kits and third party carrier boards
- Camera ecosystem partners

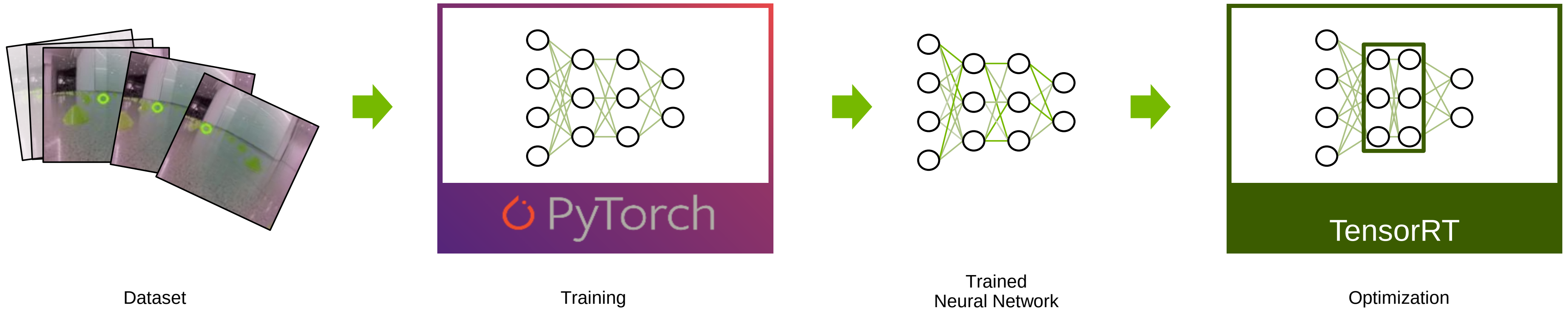




JETRACER: EXAMPLE APPLICATION / WORKFLOW

github.com/NVIDIA-AI-IOT/jetracer





PYTORCH TO TENSORRT

TENSORRT
GoogLeNet

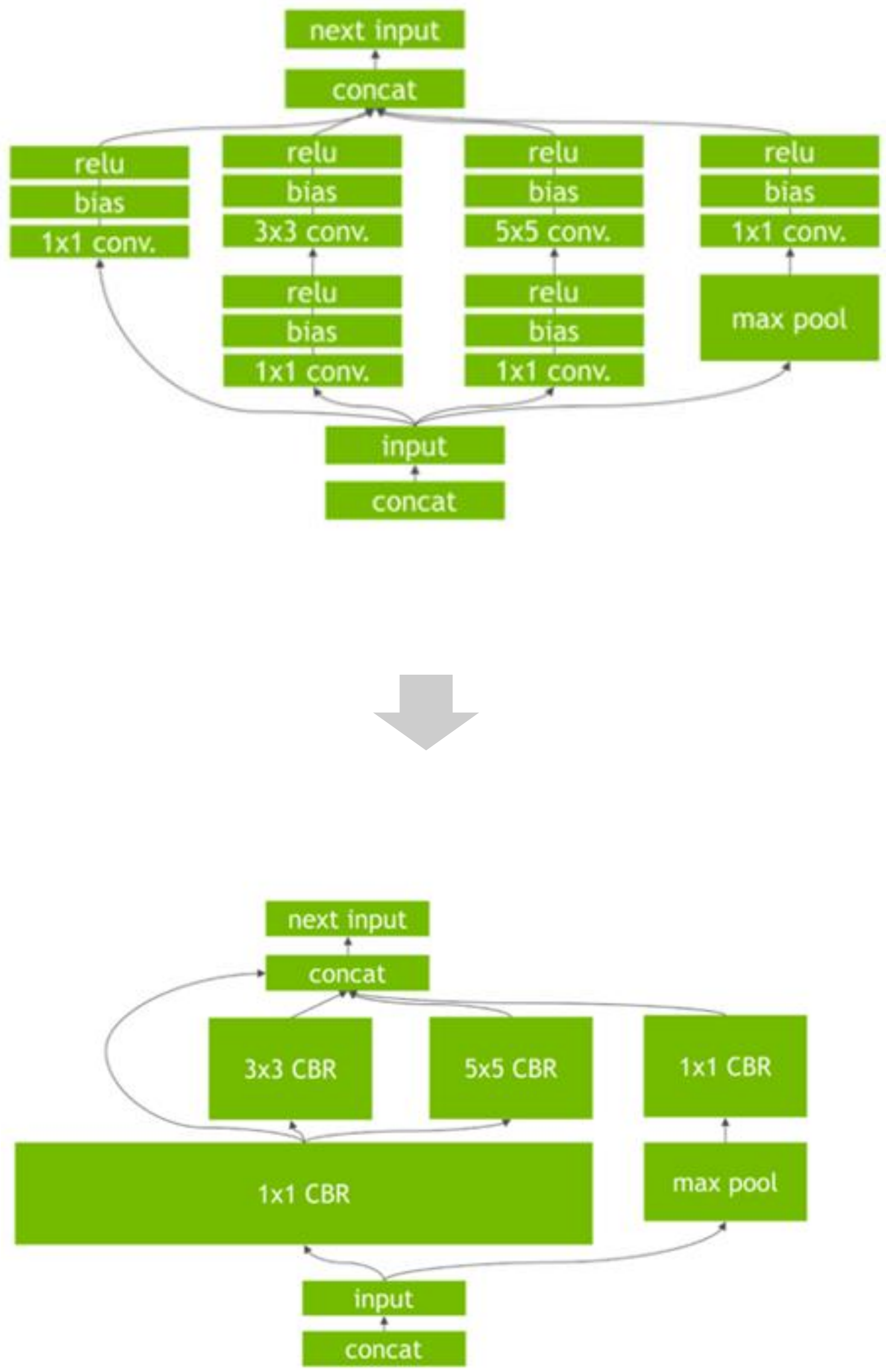
Vertical fusions

Horizontal Fusions

- Multiple Conv2d with single outputs -> Single Conv2d multiple outputs

Platform specific optimizations

- Reduced precision
- Auto kernel selection



TORCHVISION PACKAGE

github.com/pytorch/vision

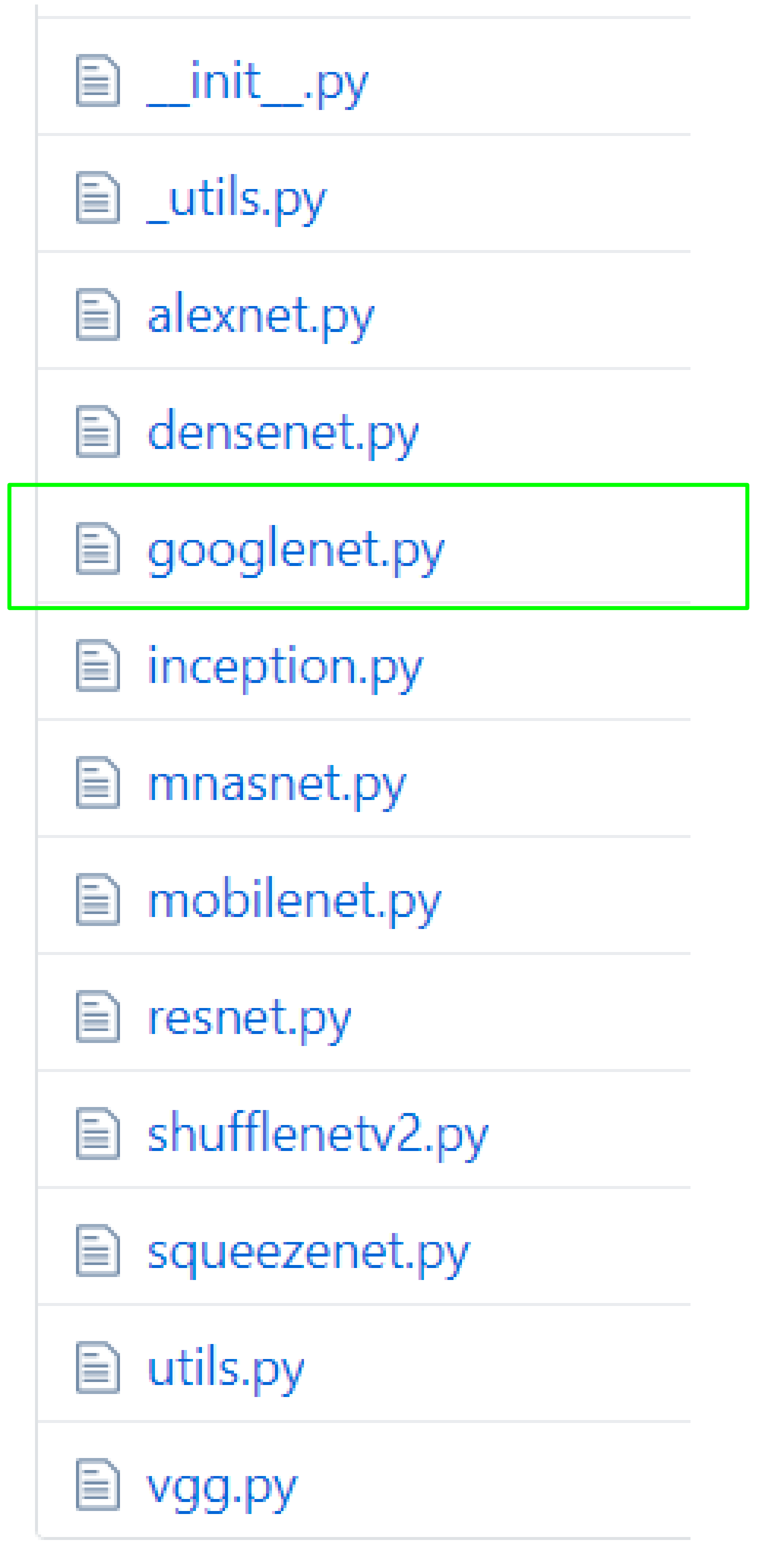
Many models pre-trained on ImageNet

AlexNet, ResNet, DenseNet, MobileNet V2, to name a few

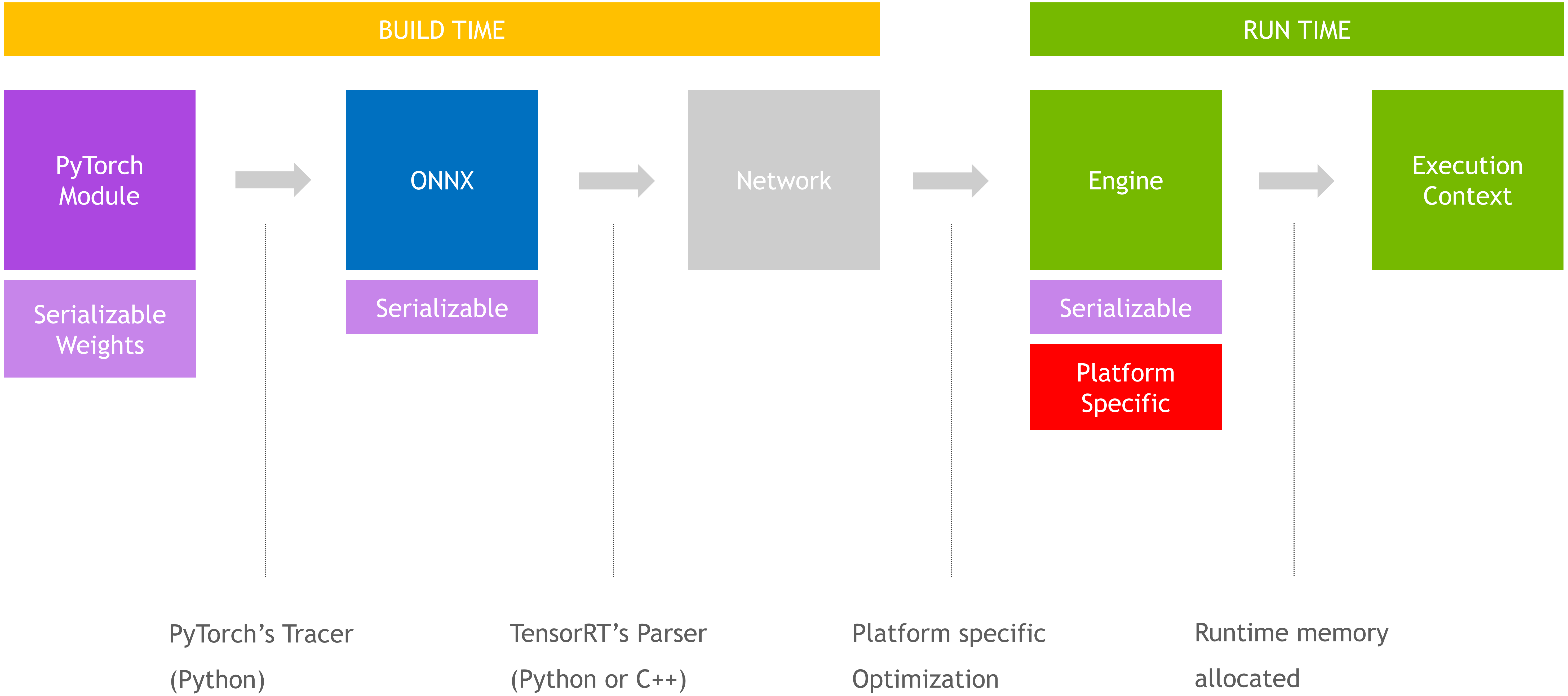
Many datasets, transformations, and utilities for vision tasks

Easy to extend and modify models for new tasks

Models largely supported by torch2trt



PYTORCH -> ONNX ->TENSORRT



Export to ONNX

using PyTorch

Uses PyTorch's tracer to export to convert program to Graph

Graph is converted to ONNX format, serialized, and saved

github.com/onnx/onnx-tensorrt

```
import torch
from torchvision.models import googlenet

model = googlenet(…).cuda().eval()

x = torch.ones(1, 3, 224, 224).cuda()

torch.onnx.export(model, x, 'googlenet.onnx')
```

Onnx-tensorrt

deserialize, optimize, run

Parses ONNX file

Builds optimized TensorRT engine

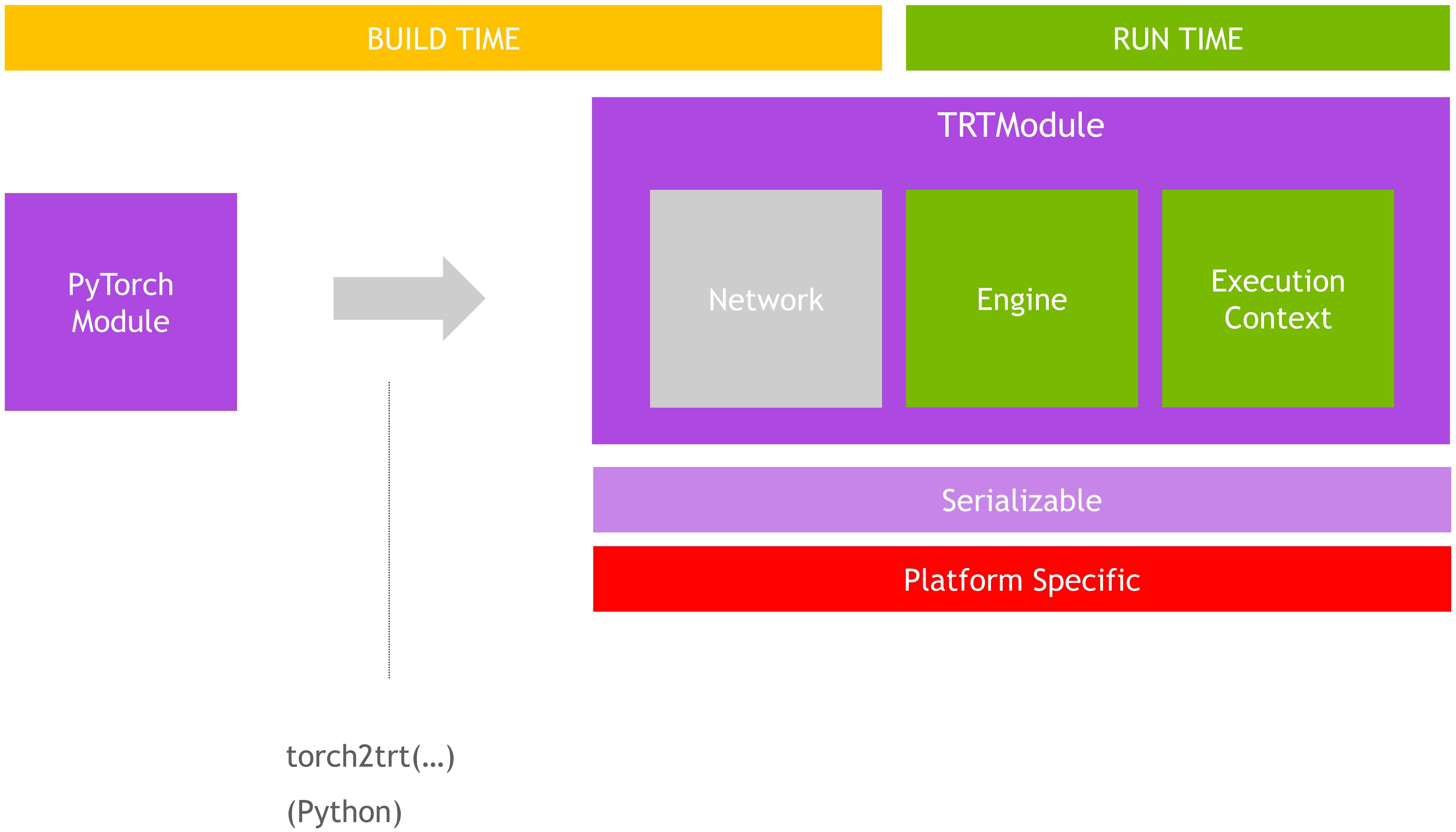
Implements TensorRT ONNX backend using engine

Allows simple execution on numpy arrays

```
import onnx
import onnx_tensorrt.backend as backend
import numpy as np

model = onnx.load('googlenet.onnx')

engine = backend.prepare(model, device='CUDA:1')

x = np.zeros(32, 3, 224, 224)

y = engine.run(x)[0]
```

TORCH2TRT

github.com/NVIDIA-AI-IOT/torch2trt



torch2trt

conversion

Data is executed through network

"Conversion Hooks" construct network using TensorRT Python API

Engine is built using optimization parameters passed to torch2trt

TRTModule is returned, which is functionally equivalent to original PyTorch Module

```
import torch
from torch2trt import torch2trt
from torchvision.models import googlenet

model = googlenet(…).cuda().eval()

x = torch.ones(1, 3, 224, 224).cuda()

model_trt = torch2trt(model, [x])
```

torch2trt

execution

Nearly same as PyTorch module

Currently, dimensions must match those provided during conversion

Batch size must not exceed max_batch_size

```
x = torch.randn(1, 3, 224, 224).cuda()

y = model(x)
y_trt = model_trt(x)

torch.max(torch.abs(y - y_trt))
```

PyTorch/torch2trt

basic timing

Can easily profile using time library

Be careful! PyTorch GPU calls are asynchronous.

TRTModule is bound to PyTorch stream, use PyTorch to synchronous

```
import time


# benchmark throughput
t0 = time.time()
torch.cuda.current_stream().synchronize()

for i in range(100):
    y = model_trt(x)

torch.cuda.current_stream().synchronize()
t0 = time.time()

print(100.0 / (t1 - t0))
```

JETSON THROUGHPUT (FPS) BY FRAMEWORK

Platform (Precision)	Nano (FP32)		AGX Xavier (FP32)	
Framework	PyTorch	TensorRT	PyTorch	TensorRT
googlenet	22.2	54.3	49.7	316
resnet18	30.5	52.5	164	339
resnet50	11.2	18.7	66.3	117
densenet121	10	21.9	26.4	114

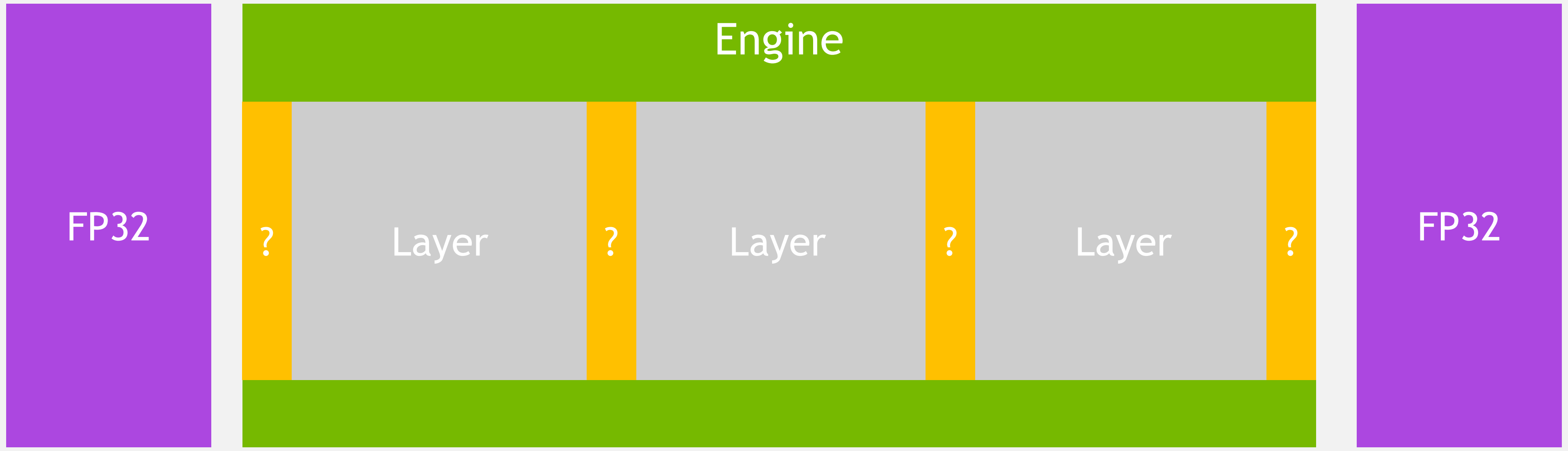
torch2trt

reduced precision

Supports FP16 / INT8 depending on platform

Input and output binding data types remain the same (match input data type)

Internal precision of layers determined by TensorRT builder




```
# fp16 (internally)
x = torch.zeros(1, 3, 224, 224).cuda()

model_trt = torch2trt(model, [x], fp16_mode=True)

# fp16 (bindings also)
x = torch.zeros(1, 3, 224, 224).cuda().half()

model_trt = torch2trt(
    model.half(), [x], fp16_mode=True)

# int8
model_trt = torch2trt(
    model,[x], int8_mode=True)
```

JETSON THROUGHPUT (FPS) BY PRECISION

Platform	Nano		AGX Xavier		
Precision	FP32	FP16	FP32	FP16	INT8
googlenet	54.3	90.3	316	527	672
resnet18	52.5	91	339	717	1030
resnet50	18.7	37.2	117	321	458
densenet121	21.9	42	114	164	230

JETSON SUPPORT MATRIX

	Nano	TX2	Xavier NX	AGX Xavier
Memory	4GB	8GB	8GB	8-32GB
Fp16 Support	YES	YES	YES	YES
Int8 Support	NO	YES	YES	YES
Deep Learning Accelerators	NONE	NONE	2	2

torch2trt

batch size

Batching reduces relative overhead, improves throughput

Specified by parameter, not input data

Runtime batch size must not exceed value

```
x = torch.zeros(1, 3, 224, 224).cuda()

model_trt = torch2trt(
    model, [x], max_batch_size=8
)

y = torch.zeros(8, 3, 224, 224).cuda()

z = model_trt(y)
```

JETSON THROUGHPUT (FPS) / LATENCY (MS) BY BATCH SIZE

Platform (Precision)	Nano (FP16)				AGX Xavier (FP16)			
Batch Size	1	2	4	8	1	2	4	8
googlenet	90.4 - 11.5	96.9 - 21.2	102 - 39.8	105 - 77.4	523 - 2.21	789 - 2.85	1030 - 4.23	1230 - 6.88
resnet18	90.8 - 11.5	98.4 - 20.8	99.9 - 40.6	101 - 80.3	718 - 1.66	1070 - 2.09	1420 - 3.08	1570 - 5.42
resnet50	34.7 - 29.5	39.4 - 51.6	40.7 - 98.6	41 - 192	318 - 3.4	470 - 4.6	562 - 7.44	620 - 13.2
densenet121	42.1 - 24.7	44.9 - 46	47.2 - 86.8	47.2 - 169	164 - 6.49	239 - 8.82	312 - 13.3	366 - 22.4

torch2trt

rename bindings

By default, inputs are named input_0, input_1, … in order

Outputs are named output_0, output_1, …

Can re-map input / output names if needed

```
x = torch.zeros(1, 3, 224, 224).cuda()

model_trt = torch2trt(model, [x],
    input_names=['image'],
    output_names=['logits']
)
```

torch2trt
int8 calibration

By default, calibrates in input data

Tracing ignores batch, but calibration uses *all* data in batch

Can override default calibration algorithm (see TensorRT Python API for options)

```
# calibrate on random data
x = torch.randn(32, 3, 224, 224).cuda()

model_trt = torch2trt(model,[x], int8_mode=True)

# specify calibration algorithm
model_trt = torch2trt(model, [x], int8_mode=True,
int8_calib_algorithm=…)
```

torch2trt

int8 calibration (more data)

Dynamically loads input data to support larger datasets

Calibration dataset provides *only* inputs, excluding batch

```
# define input dataset class
class ImageFolderDataset():
    def __init__(self, folder):
        self.paths = glob.glob(…)

    def __len__(self):
        return len(self.paths)

    def __getitem__(self, idx):
        path = self.paths[idx]
        # load image to CxHxW tensor
        return [ image ]


# calibrate on image folder
calib_dataset = ImageFolderDataset('images')

x = torch.zeros(1, 3, 224, 224).cuda()

model_trt = torch2trt(model,[x], int8_mode=True,
    int8_calib_dataset=calib_dataset)
```

torch2trt

int8 calibration (multiple inputs)

Some modules require multiple inputs

Specified in order they are fed to module

(albeit not GoogLeNet)

```
class StereoImageDataset():
    …
    def __getitem__(self, idx):
        ...
        return [ left_image, right_image ]


# calibrate multiple input model
calib_dataset = StereoDataset(…)

left = torch.zeros(1, 3, 224, 224).cuda()
right = torch.zeros(1, 3, 224, 224).cuda()

model_trt = torch2trt(model,[left, right],
int8_mode=True,
    int8_calib_dataset=calib_dataset)
```

torch2trt

save / load

Same as PyTorch module

Allows TRTModule to replace PyTorch submodule

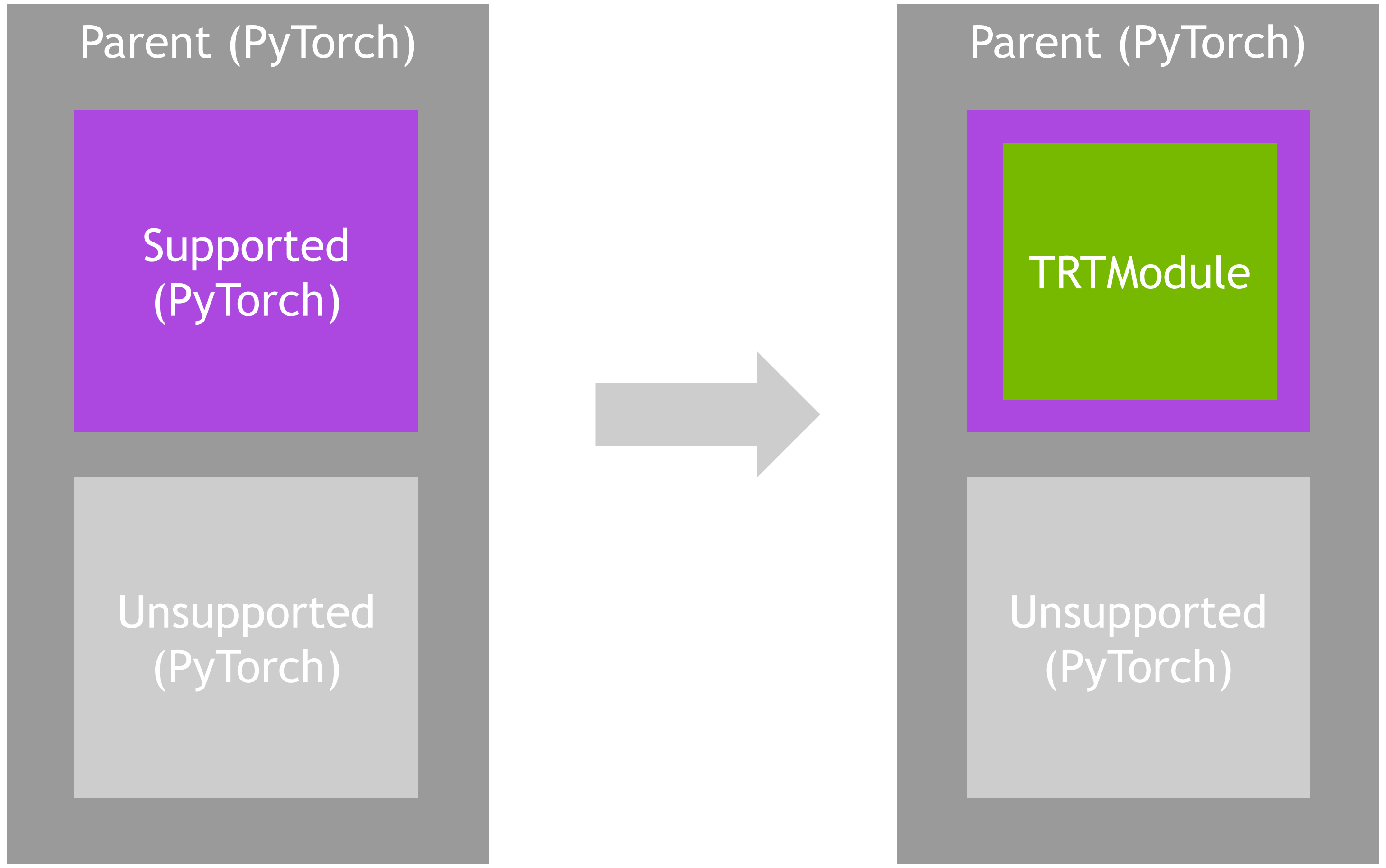
Network is dropped when saved (since it is not serializable)

```
# save state dict
torch.save(model_trt.state_dict(),'model_trt.pth')


# load state dict
From torch2trt import TRTModule()

model_trt = TRTModule()

model_trt.load_State_dict(
  torch.load('model_trt.pth')
)
```

EXECUTION AND STORAGE PARITY

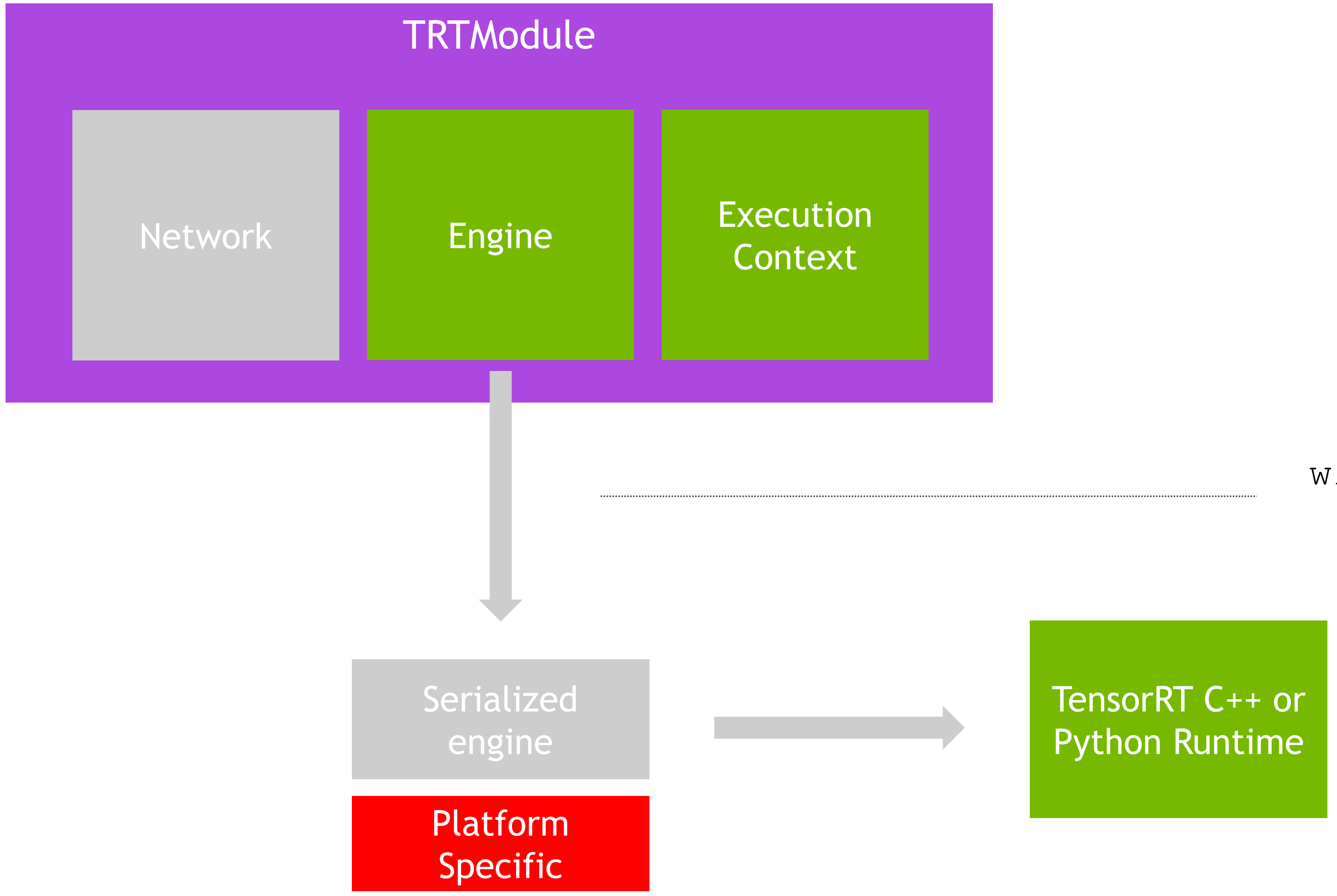
Allows partial conversion of modules

```
# replace and save
parent.supported = torch2trt(parent.supported, …)
torch.save(parent.state_dict(), …)

# replace and load
parent.supported = TRTModule()
torch.load_state_dict(…)
```



SAVING FOR C++

Same as TensorRT Python API!




```
with open('model.engine', 'wb') as f:
    f.write(model_trt.engine.serialize())
```

torch2trt

custom converter

torch2trt is easy to modify

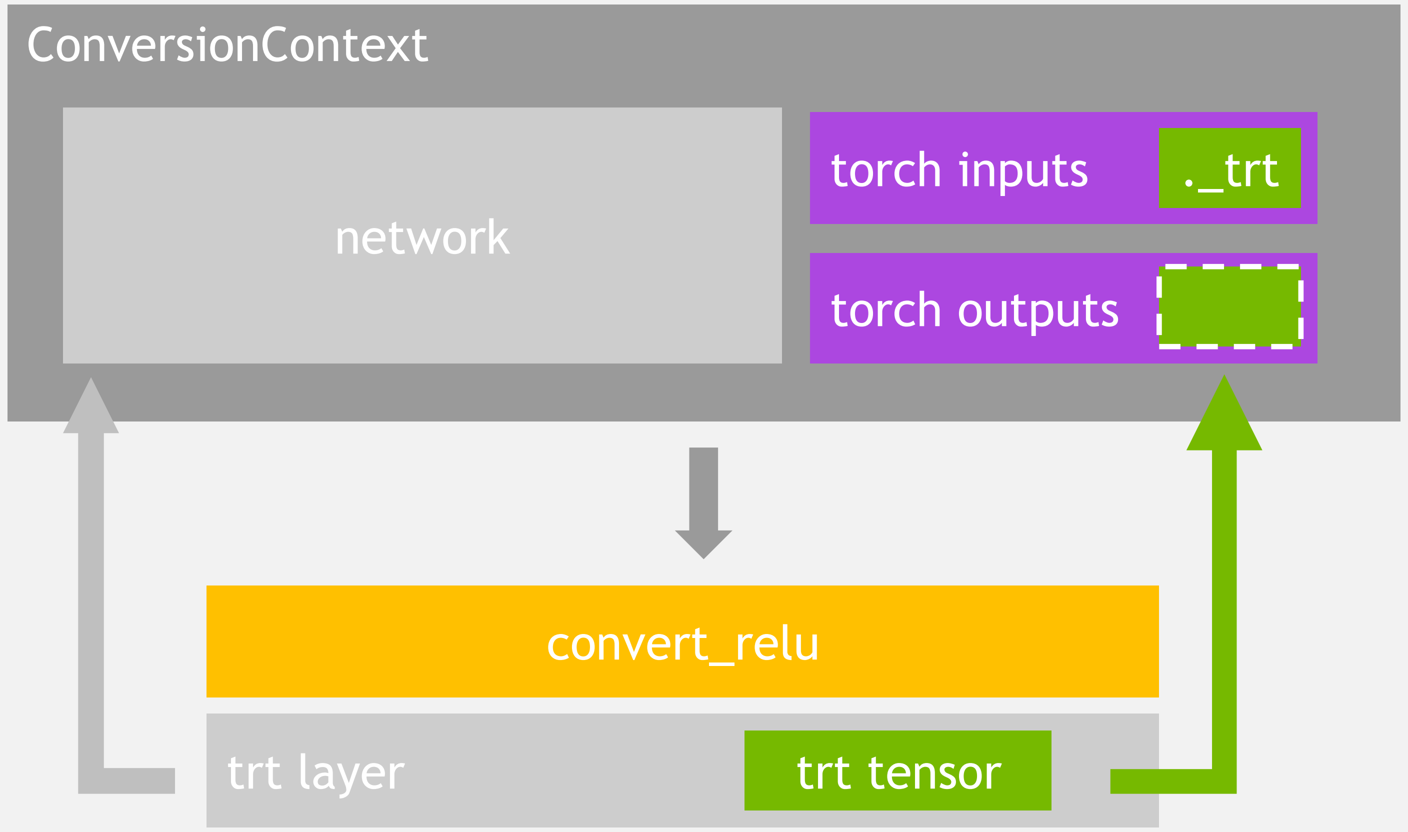
Define converter with @tensorrt_converter

Converter takes a “ConversionContext”

- ctx.network - TensorRT network being constructed
- ctx.method_args - Arguments to PyTorch method
- ctx.method_kwargs - Keyword args to PyTorch method
- ctx.method_return - Return value of PyTorch method

Converter uses TensorRT Python API to extend network

Converter must set _trt attribute of relevant torch outputs




```
@tensorrt_converter('torch.relu')
def convert_relu(ctx):
    input = ctx.method_args[0]
    output = ctx.method_return
    trt_layer = ctx.network.add_activation(
        input=input._trt,
        type=trt.ActivationType.RELU
    )
    output._trt = trt_layer.get_output(0)
```

NETWORK VISUALIZATION

What layers were added to the network?

Convert network to GraphViz “Dot” format

Useful for debugging

Easily render as PDF

```
from torch2trt.utils import trt_network_to_dot_graph

dot = trt_network_to_dot_graph(model_trt.network)

dot.render(‘googlenet.gv’, view=True)
```

NETWORK VISUALIZATION

How are layers mapped?

Use TensorRT profiler! (next slide)

Each line in output is single layer

Horizontal 1x1 convolutions fused

Batch Norms (Scale) fused

Activations fused


```
(Unnamed Layer* 10) [Pooling]: 0.036384ms
(Unnamed Layer* 11) [Convolution] + (Unnamed Layer* 13) [Activation] || (Unnamed Layer* 14) [Convolution] + (Unnamed Layer* 16) [Activation] || (Unnamed Layer* 20) [Convolution] + (Unnamed Layer* 22) [Activation]: 0.072224ms
(Unnamed Layer* 17) [Convolution] + (Unnamed Layer* 19) [Activation]: 0.09824ms
(Unnamed Layer* 23) [Convolution] + (Unnamed Layer* 25) [Activation]: 0.012672ms
(Unnamed Layer* 26) [Pooling]: 0.019296ms
(Unnamed Layer* 27) [Convolution] + (Unnamed Layer* 29) [Activation]: 0.033024ms
(Unnamed Layer* 13) [Activation]_output copy: 6.25901ms
```

torch2trt

TensorRT profiling

Adds fine-grained profiling of internal TensorRT layers

Prints to stdout

Model execution becomes synchronous

```
x = torch.zeros(1, 3, 224, 224).cuda()

model_trt.enable_profiling()

y = model_trt(x)
```

PyTorch/torch2trt

CUDA profiling

Capture all CUDA runtime calls in region

Dump files for NVIDIA Visual Profiler

```
torch.cuda.profiler.init('googlenet.nvvp',
output_mode='csv')


# collect region using context manager
torch.cuda.current_stream().synchronize()
with torch.cuda.profiler.profile():
    y = model_trt(x)
    torch.cuda.current_stream().synchronize()


# collect region using start/stop
torch.cuda.current_stream().synchronize()
torch.cuda.profiler.start()
y = model_trt(x)
torch.cuda.current_stream().synchronize()
torch.cuda.profiler.stop()
```

PYTORCH VISUAL PROFILE

Low GPU Utilization

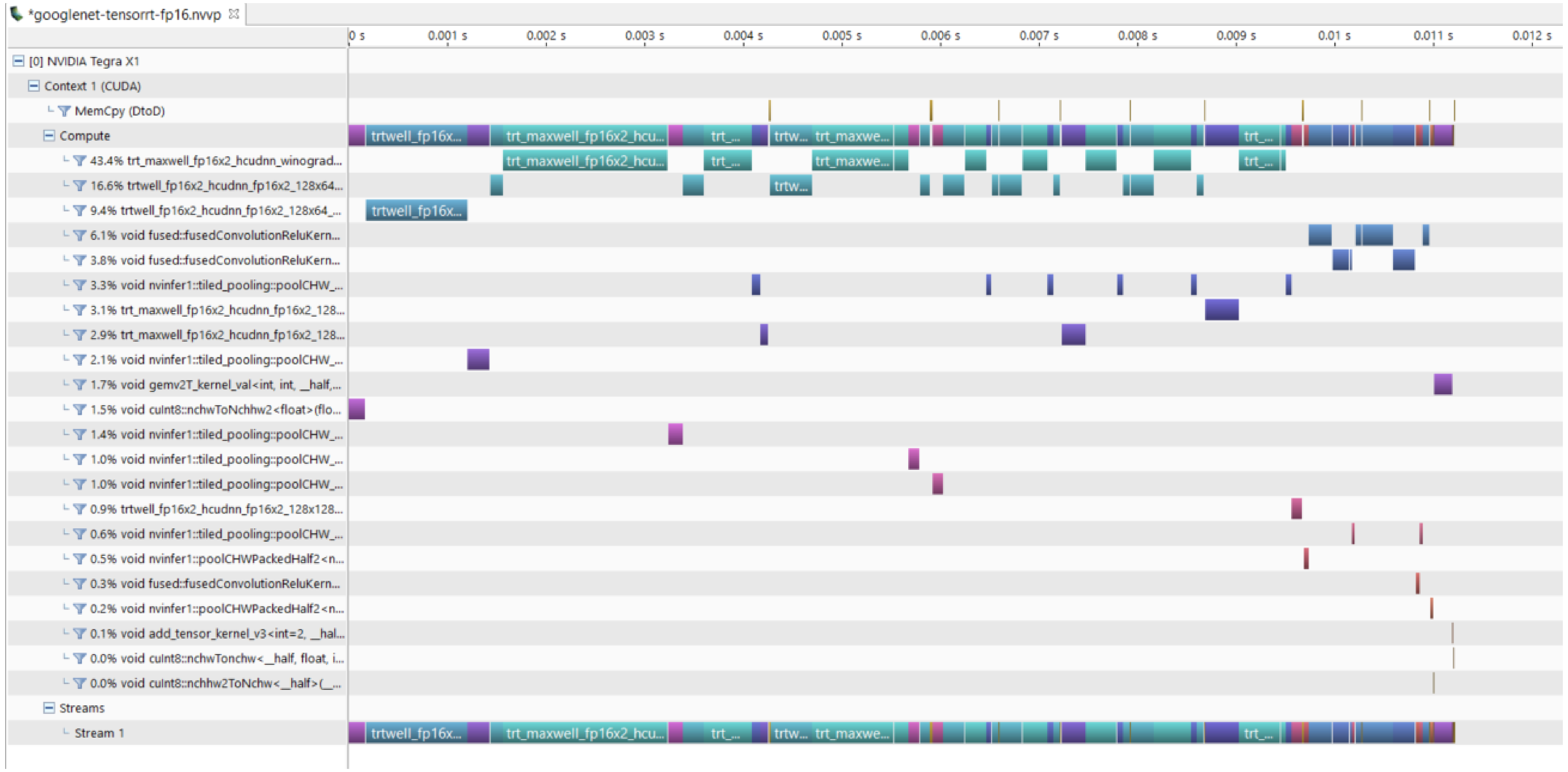
Many kernel Invocations

Analysis | GPU Details (Summary) | CPU Details | OpenACC Details

Name	Invocations	Avg. Duration	Re
cudnn::maxwell::gemm::computeOffsetsKer...	39	3.11 μs	n/
_ZN2at6native18elementwise_kernelILi512...	57	29.699 μs	n/
void cudnn::detail::bn_fw_inf_1C11_kernel_...	57	36.779 μs	n/
void cudnn::detail::explicit_convolve_sgem...	2	65.807 μs	n/
void at::native::reduce_kernel<int=512, at::n...	1	77.292 μs	n/
void im2col4d_kernel<float, int>(im2col4d...	2	78.75 μs	n/
void CatArrayBatchedCopy<float, unsigne...	9	113.634 μs	n/
void cudnn::winograd::generateWinogradTi...	16	141.627 μs	n/
maxwell_scudnn_128x32_relu_interior_nn	11	156.86 μs	n/
maxwell_scudnn_128x64_relu_interior_nn	14	236.544 μs	n/
void gemv2T_kernel_val<int, int, float, float...	1	247.084 μs	n/
maxwell_scudnn_128x128_relu_interior_nn	12	264.861 μs	n/
maxwell_scudnn_128x32_relu_small_nn	1	363.803 μs	n/
void at::native::_GLOBAL__N__62_tmpxft_00...	13	373.758 μs	n/
maxwell_scudnn_winograd_128x128_ldg1_l...	16	580.322 μs	n/
maxwell_scudnn_128x64_relu_medium_nn	1	1.56318 ms	n/

TENSORRT VISUAL PROFILE

High GPU Utilization



Few Kernel Invocations

Analysis | GPU Details (Summary) | CPU Details | OpenACC

Name	Invocations	Avg. Duration
void culnt8::nchwTonchw<_half, float, ...	1	5.417 μs
void add_tensor_kernel_v3<int=2, _hal...	1	7.448 μs
void nvinfer1::tiled_pooling::poolCHW_...	2	31.406 μs
void fused::fusedConvolutionReluKerne...	1	37.552 μs
void nvinfer1::poolCHWPackedHalf2<n...	2	39.088 μs
void nvinfer1::tiled_pooling::poolCHW_...	6	59.774 μs
trtwell_fp16x2_hcudnn_fp16x2_128x12...	1	102.344 μs
void nvinfer1::tiled_pooling::poolCHW_...	1	105.625 μs
void nvinfer1::tiled_pooling::poolCHW_...	1	108.646 μs
void fused::fusedConvolutionReluKerne...	3	138.142 μs
void nvinfer1::tiled_pooling::poolCHW_...	1	149.843 μs
trt_maxwell_fp16x2_hcudnn_fp16x2_12...	2	159.947 μs
trtwell_fp16x2_hcudnn_fp16x2_128x64_...	11	164.763 μs
void fused::fusedConvolutionReluKerne...	4	166.94 μs
void culnt8::nchwToNchhw2<float>(flo...	1	169.115 μs
void gemv2T_kernel_val<int, int, _half,...	1	187.5 μs
void nvinfer1::tiled_pooling::poolCHW_...	1	226.614 μs
trt_maxwell_fp16x2_hcudnn_winograd_...	15	316.496 μs
trt_maxwell_fp16x2_hcudnn_fp16x2_12...	1	339.844 μs
trtwell_fp16x2_hcudnn_fp16x2_128x64_...	1	1.03276 ms

TENSORFLOW TO TENSORRT

OPTIMIZING TENSORFLOW

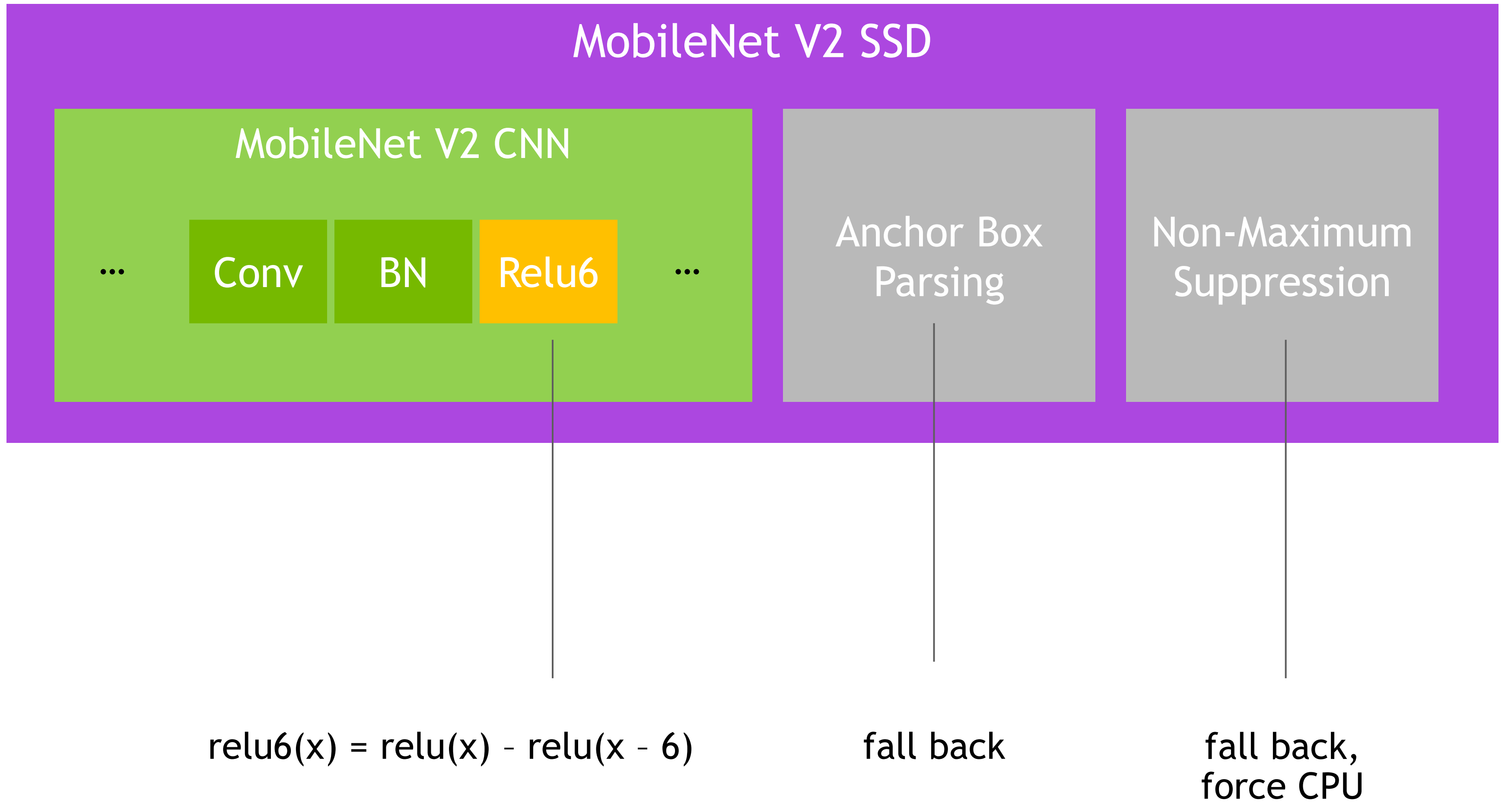
What are our options?

- TF-TRT (TensorRT integration in TensorFlow)
  - Runs like a normal TensorFlow graph
  - Unsupported operations fall-back to TensorFlow
- TensorFlow -> UFF -> TensorRT
  - Convert TensorFlow graph to UFF format
  - Parse UFF file and optimize with TensorRT
  - Requires TensorRT Plugins for unsupported parts

SINGLE SHOT DETECTOR

Case study (TF-TRT)

- Sourced from TensorFlow object detection API
- CNN Backbone
  - Supported, except ReLU 6 (at the time)
- Anchor box parsing
  - Fall back to TF
- Non-maximum suppression
  - Fall back to TF
  - Native TF was slow… repetitive unnecessary CPU/GPU copies



TensorFlow/TF-TRT

TensorFlow profiler

Execute TensorFlow graph enabling tracing

Export metadata in chrome trace format

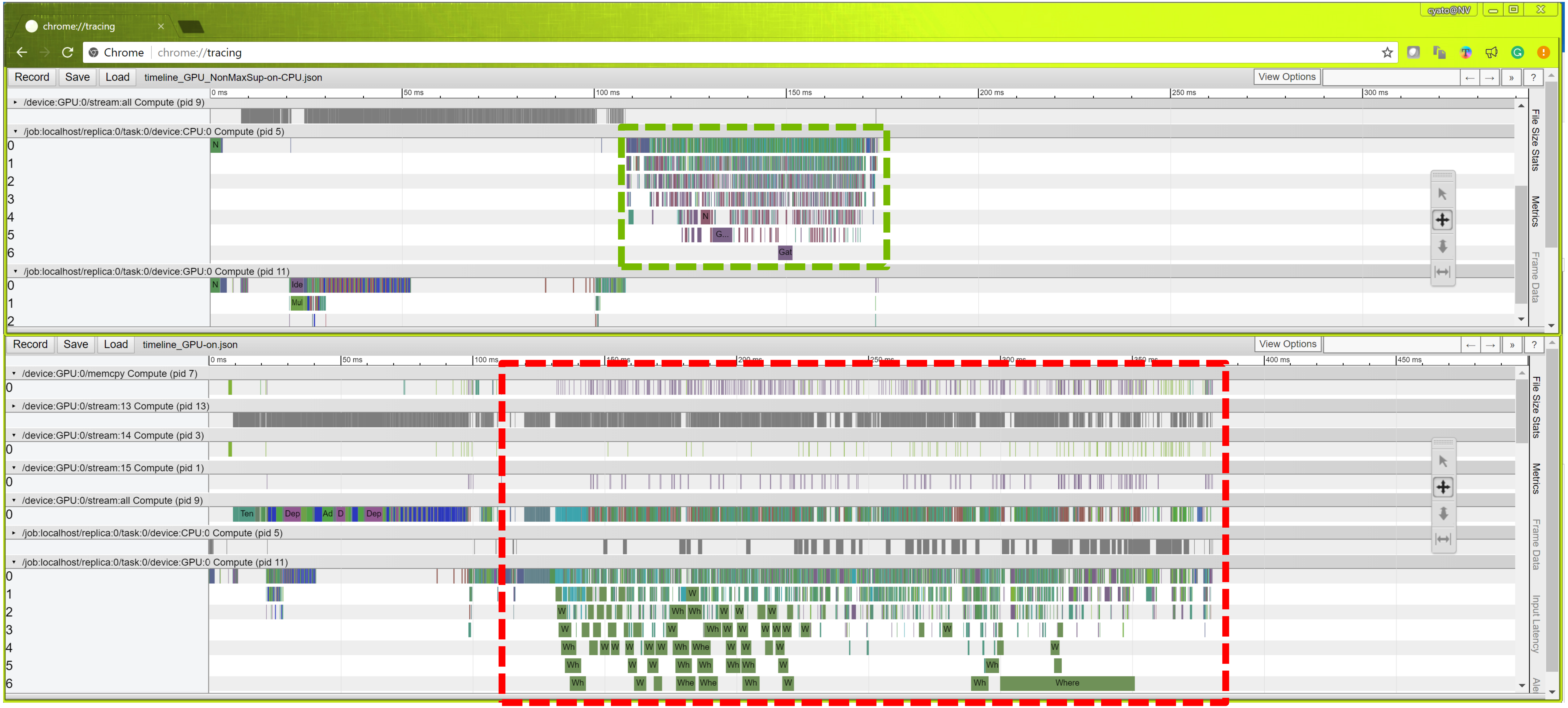
Visualize with chrome browser

- Easily spot data copies, layer calls, layer devices

We used this to find a CPU->GPU copy bottleneck

```
options =
tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)

run_metadata = tf.RunMetadata()

Sess.run(…, options=options, run_metadata=run_metadata)

run_timeline = timeline.Timeline(run_metadata.step_stats)
Chrome_trace = run_timeline.generate_chrome_trace_format()
```

TENSORFLOW PROFILE TRACE



TF-TRT

Optimize frozen graph

One call: “create_inference_graph”

Input is frozen graph, with all TensorFlow layers

Output is frozen graph, with sub-graphs as TensorRT blocks

Minimum segment size is used to control granularity

- prevent “small” engines with non-negligible overhead

```
frozen_graph = tf.GraphDef()
with open(‘frozen_inference_graph.pb’, ‘rb’) as f:
    frozen_graph.ParseFromString(f.read())


trt_graph = trt.create_inference_graph(
    input_graph_def=frozen_graph,
    outputs=[‘detection_boxes’,
‘detection_classes’, ‘detection_scores’,
‘num_detections’],
    max_batch_size=1,
    max_workspace_size=1 << 25,
    precision_mode=‘FP16’,
    minimum_segment_size=50
)
```

TF-TRT

Execute graph

Set allow_growth to prevent TensorFlow from hogging Jetson memory

Model	Input Size	TF-TRT TX2	TF TX2
ssd_mobilenet_v1_coco	300x300	50.5ms	72.9ms
ssd_inception_v2_coco	300x300	54.4ms	132ms

```
# configure session to allow growth for memory
tf_config = tf.ConfigProto()
tf_config.gpu_options.allow_growth = True
tf_sess = tf.Session(config=tf_config)

# load optimized grpah
tf.import_graph_def(trt_graph, name='')

# execute graph as normal tensorflow ...
```

DESIGNING FOR REAL-TIME

PRAGMATIC CONSTRAINTS

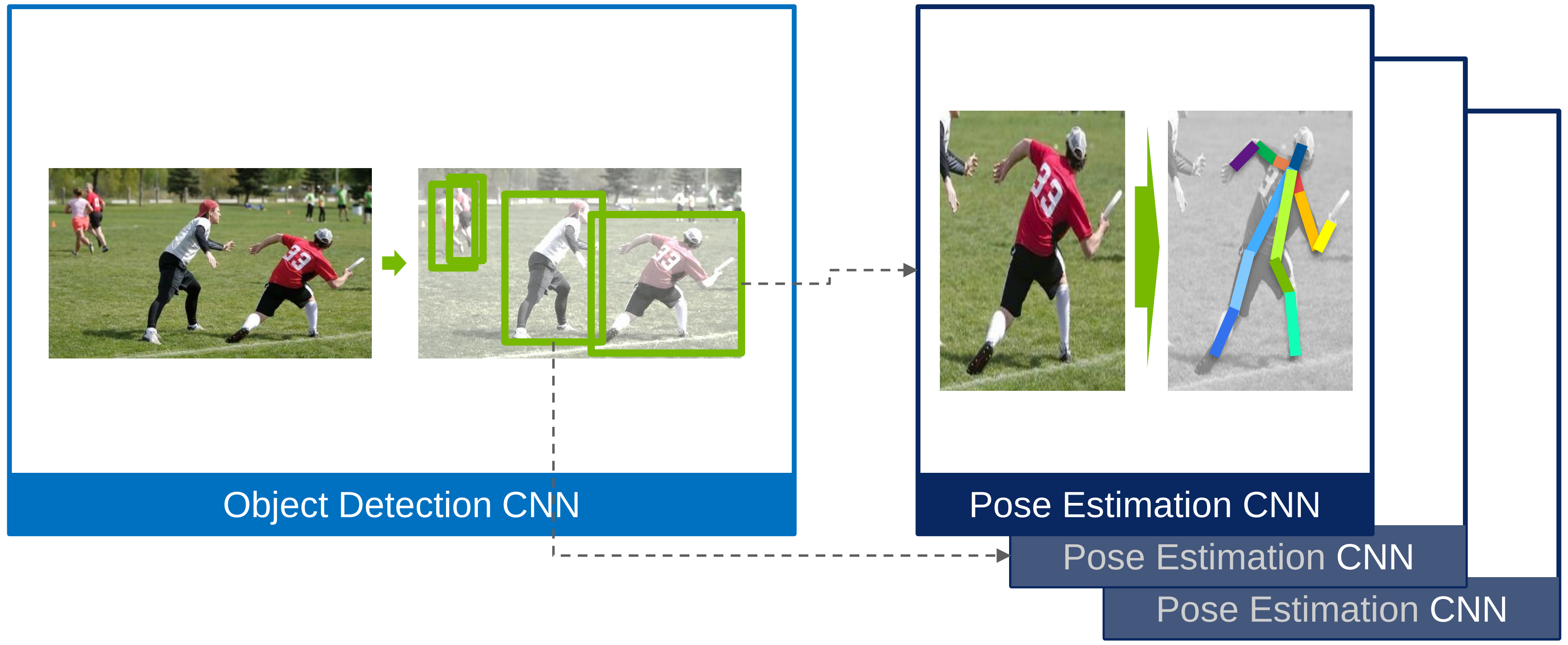
For real-time deployment on Jetson

- Avoid data dependent CNN execution like two-stage detectors (when appropriate)
    - Typically, this will keep runtime and memory nearly static
- Use TensorRT supported layers when possible
    - Using just one framework can reduce memory consumption
    - More possible fusions, fewer unnecessary type casting / reformatting
- Lightweight post-processing / parsing
    - Similar to (1), to ensure near-constant runtime

POSE DETECTION

Case study

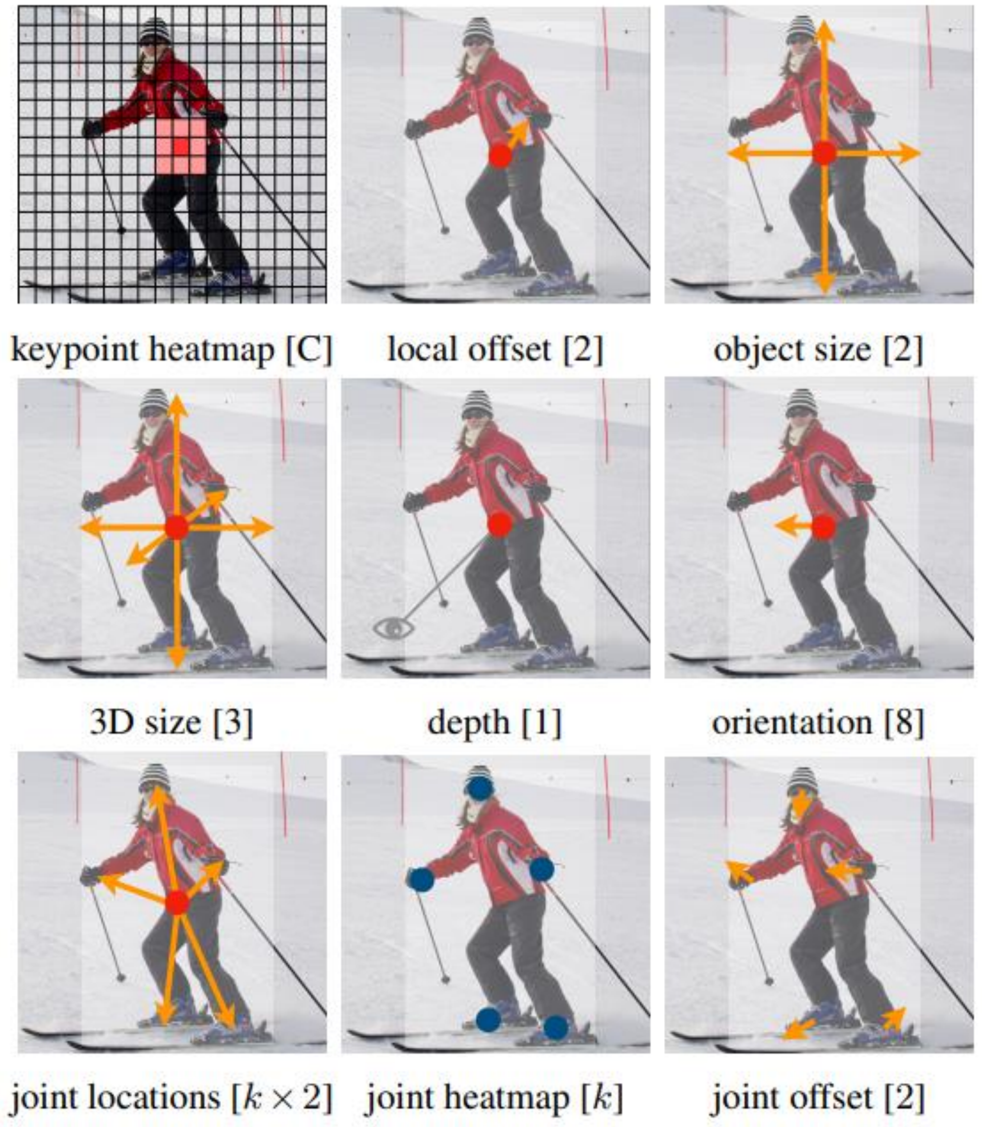
- Top performing methods commonly include
  - Two stage detectors
  - Ensemble networks
- These methods are usually computationally expensive
  - Two Stage scale's with number of objects in image



CENTERNET

Near static runtime

- Single CNN produces feature maps
- Objects parse by finding peak of heatmap
- Other semantics then parsed
- No second large CNN execution

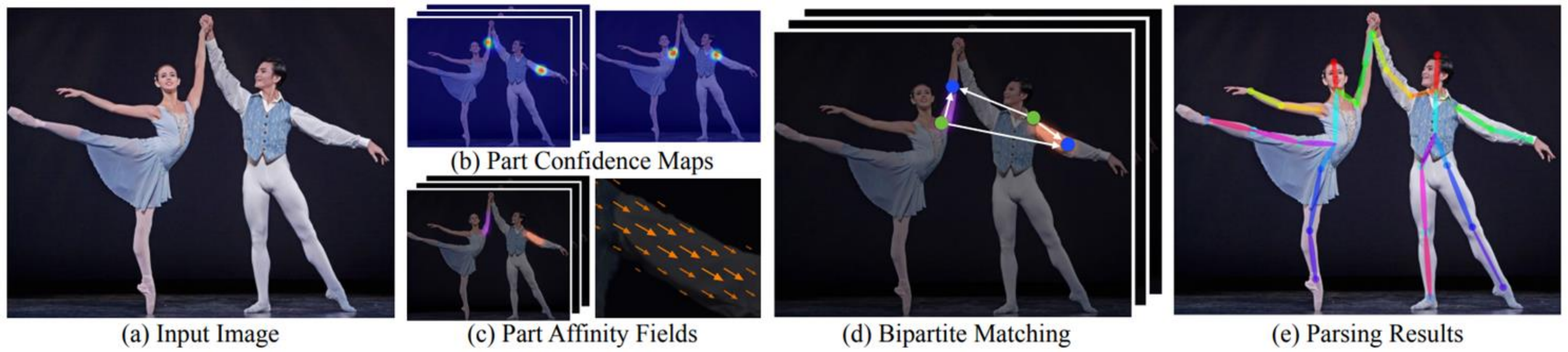


Zhou, Xingyi, Dequan Wang, and Philipp Krähenbühl. "Objects as Points." *arXiv preprint arXiv:1904.07850* (2019).

PART AFFINITY FIELDS

Near static runtime

- Single CNN produces two feature maps
  - Confidence Map
  - Part affinity field
- Part x,y coordinates proposed from local maxima of confidence maps
- Part associate scores produced by integrating between parts
- Assignment algorithm applied to associate parts
  - <1ms on CPU typically

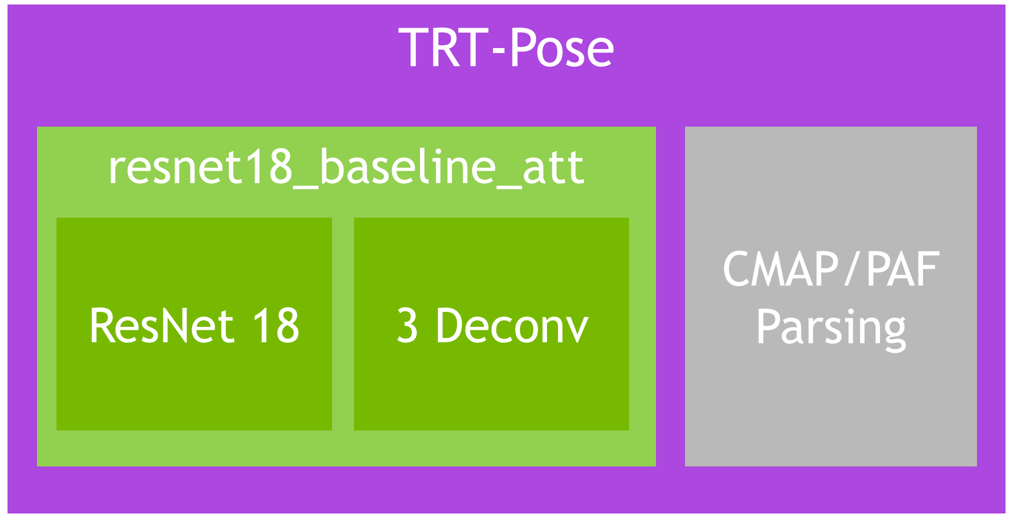


Cao, Zhe, et al. "Realtime multi-person 2d pose estimation using part affinity fields." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017.

TRT-POSE: REAL-TIME POSE DETECTION

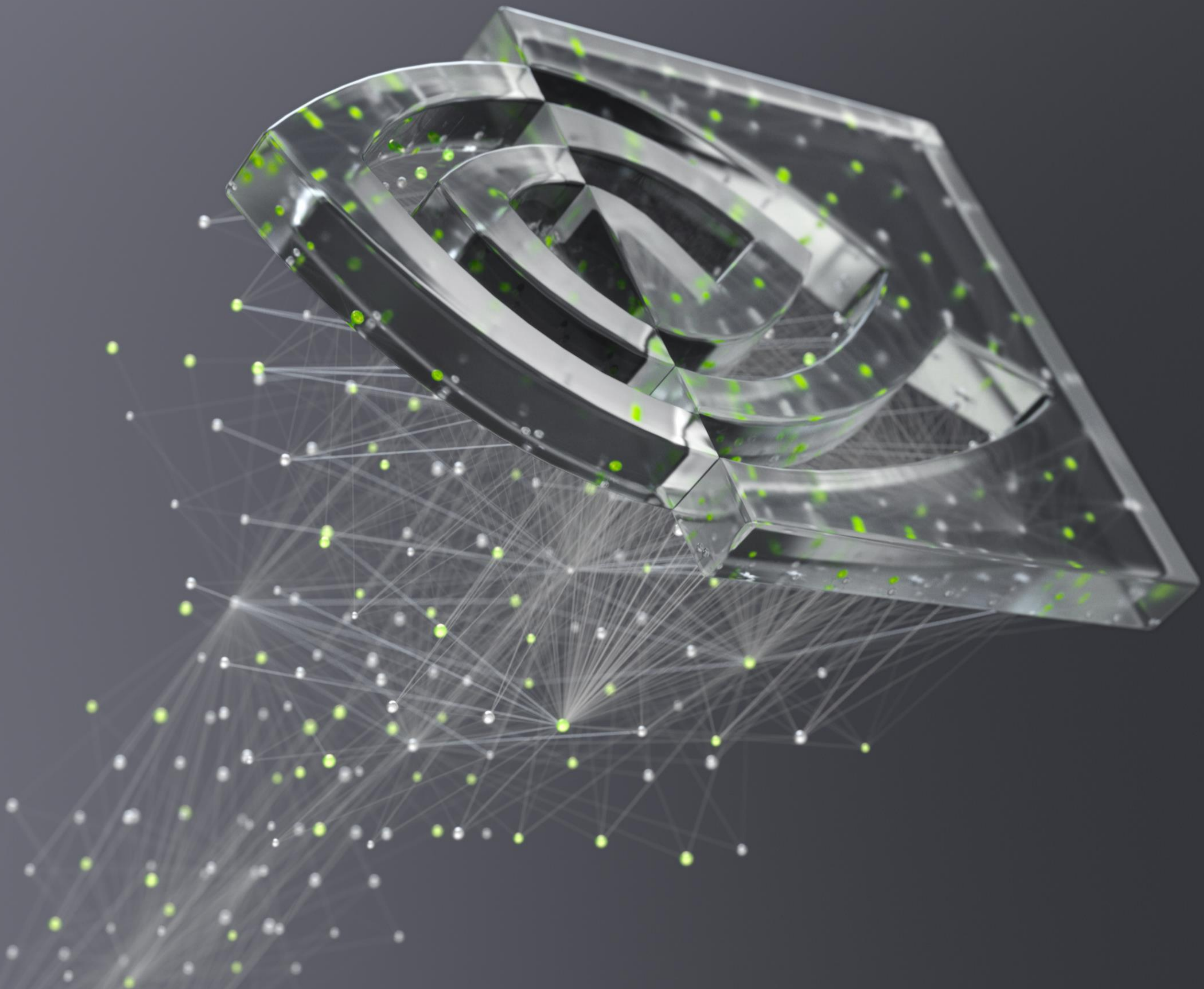
github.com/NVIDIA-AI-IOT/trt_pose

- Resnet18_baseline_att
  - Resnet18 well optimized by TensorRT for Jetson
  - 3x Deconvolution at 4x4 pixel natively supported by TRT
- CMAP / PAF post processing
  - Low post-processing runtime
- ~22 FPS Jetson Nano



USEFUL EXTRAS

- Torchvision package
  - Many TensorRT ready pre-trained backbone architectures
  - Easy to use /extend
  - github.com/pytorch/vision
- Segmentation_models.pytorch
  - Many TensorRT ready *multi-scale* pre-trained backbone architectures
  - Easy to use / extend
  - github.com/qubvel/segmentation_models.pytorch
- Jetson Benchmarks
  - Various reproducible benchmarks for tasks like Object Detection with TensorRT. Including DLA.
  - github.com/NVIDIA-AI-IOT/jetson_benchmarks



NVIDIA