NVIDIA

A21261: GET THE HIGHEST INFERENCE PERFORMANCE USING TENSORRT

Joohoon Lee – TensorRT Product Manager

October, 2020

AGENDA

Introduction

Optimizations

Workflow with an example

Useful tools and resources

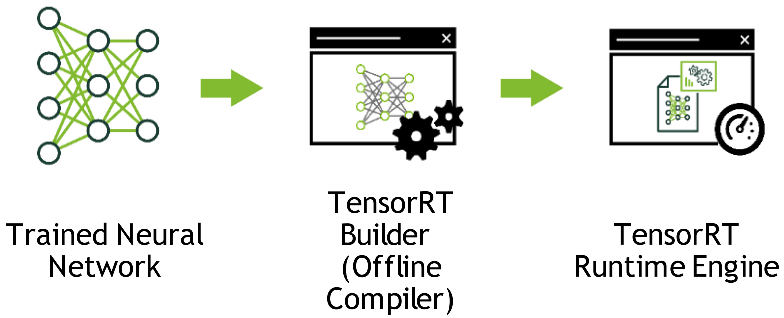
INTRODUCTION

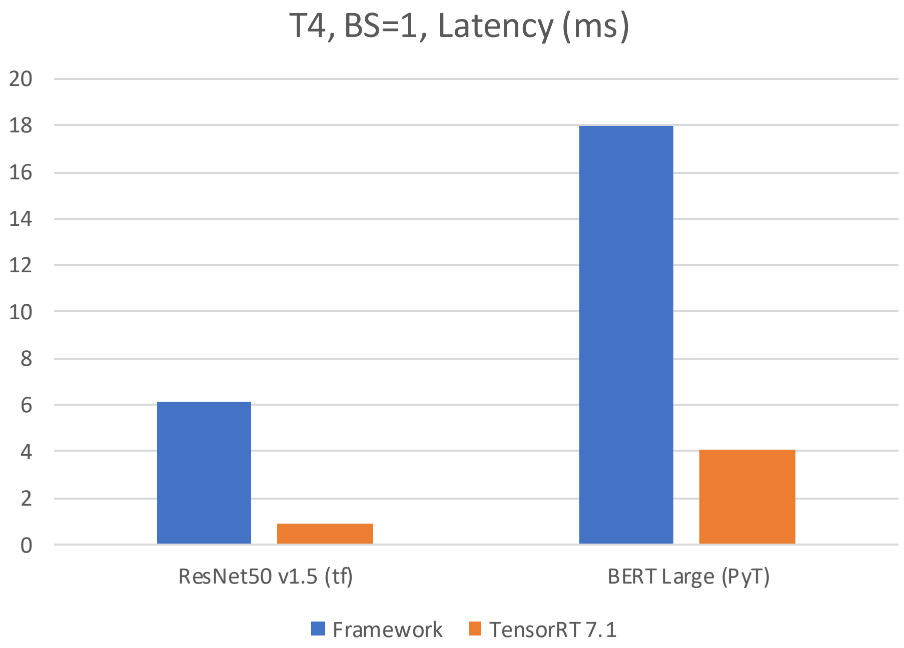
NVIDIA TensorRT

SDK for High-Performance Deep Learning Inference

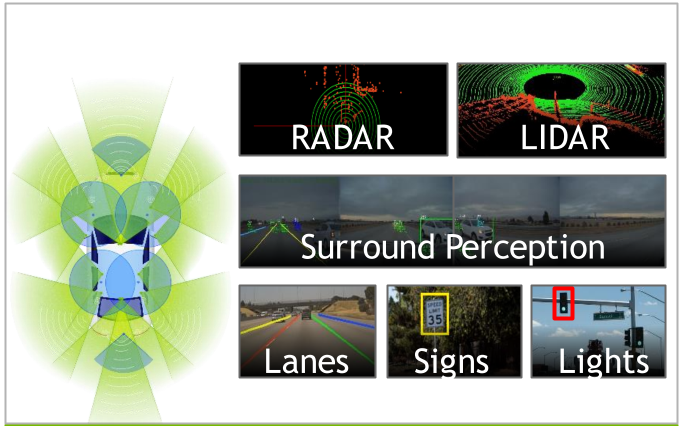
Optimize and Deploy neural networks in production environments

- Maximize throughput for latency-critical apps with compiler and runtime
- Deploy responsive and memory efficient apps with INT8 & FP16 optimizations
- Optimize every network including CNN, RNN, MLP and Transformers
- Deploy to hyperscale data centers, edge and embedded platforms
- C++ and Python APIs

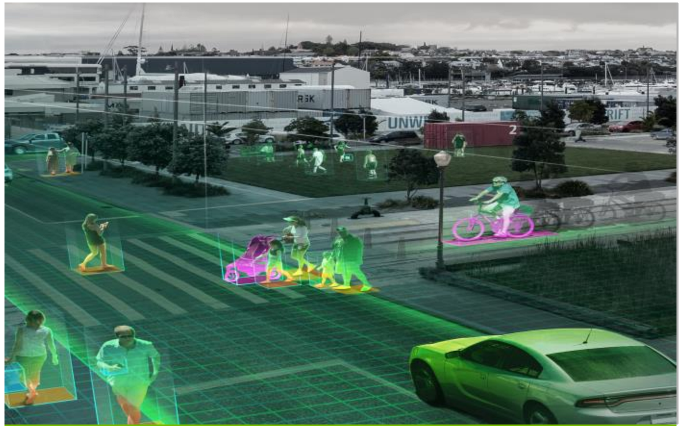




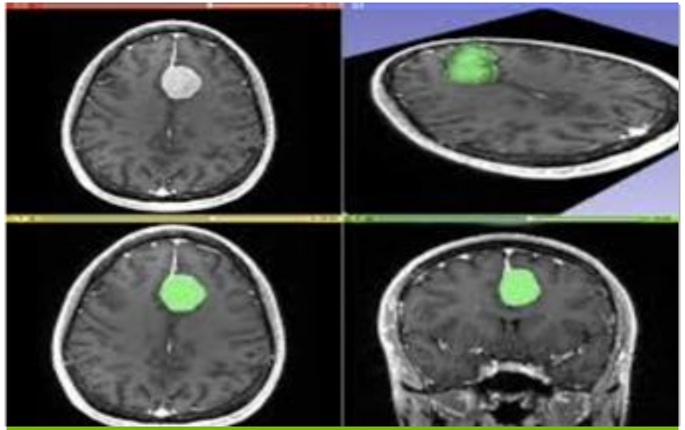
TensorRT USE CASES



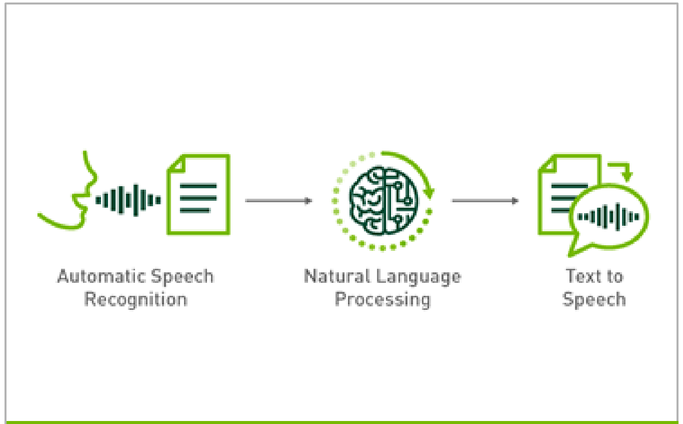
Autonomous Driving - DRIVE


Metropolis - DeepStream


Robotics - Isaac SDK


Medical - Clara

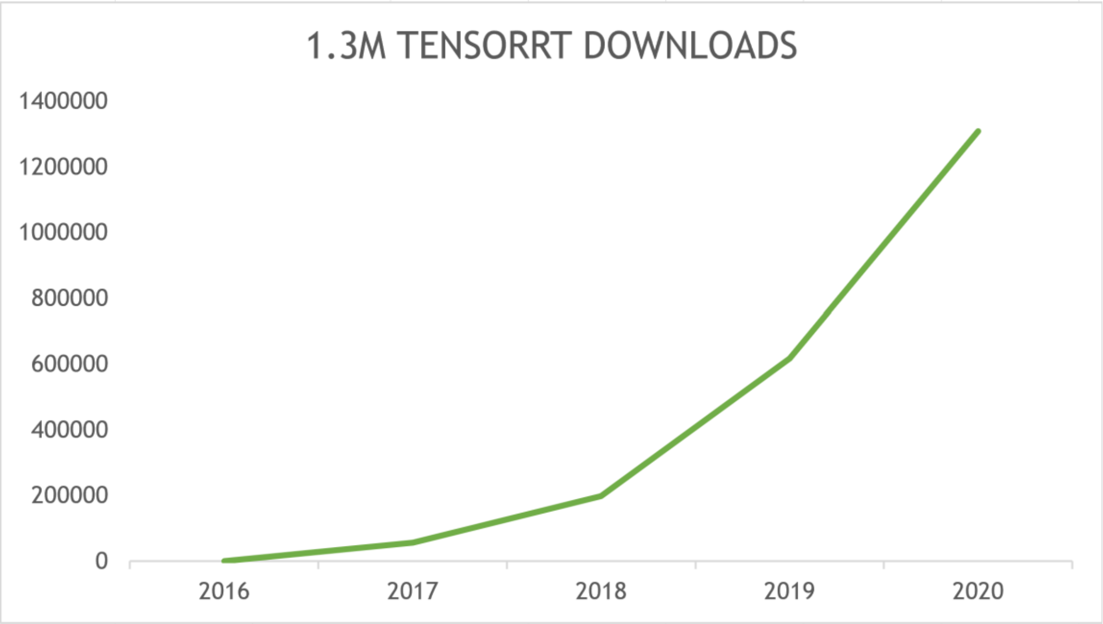
Broadcast Engine



Conversational AI – Jarvis/NeMo

200K DEVELOPERS USE TensorRT

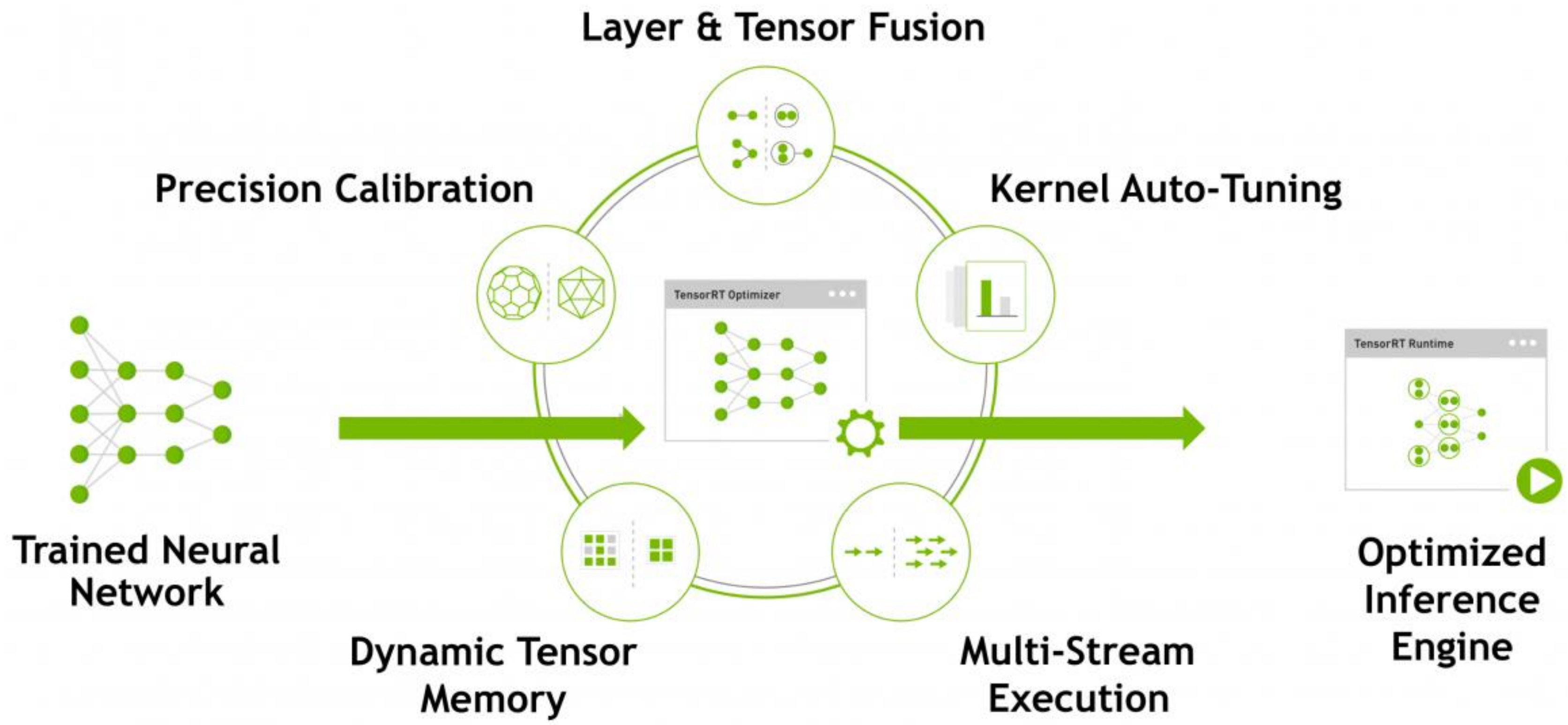
500K+ Data center Downloads
800K+ Edge Downloads



developer.nvidia.com/tensorrt

OPTIMIZATIONS

NVIDIA TensorRT OPTIMIZATIONS



PRECISION CALIBRATION

Leverage reduced precision TensorCore :

- FP16 - Volta and newer GPUs
- INT8 – Turing and newer GPUs

Precision	Dynamic Range	
FP32/TF32	$-3.4 \times 10^{38} \sim +3.4 \times 10^{38}$	← Training precision
FP16	-65504 ~ +65504	← No calibration required
INT8	-128 ~ +127	← Requires calibration

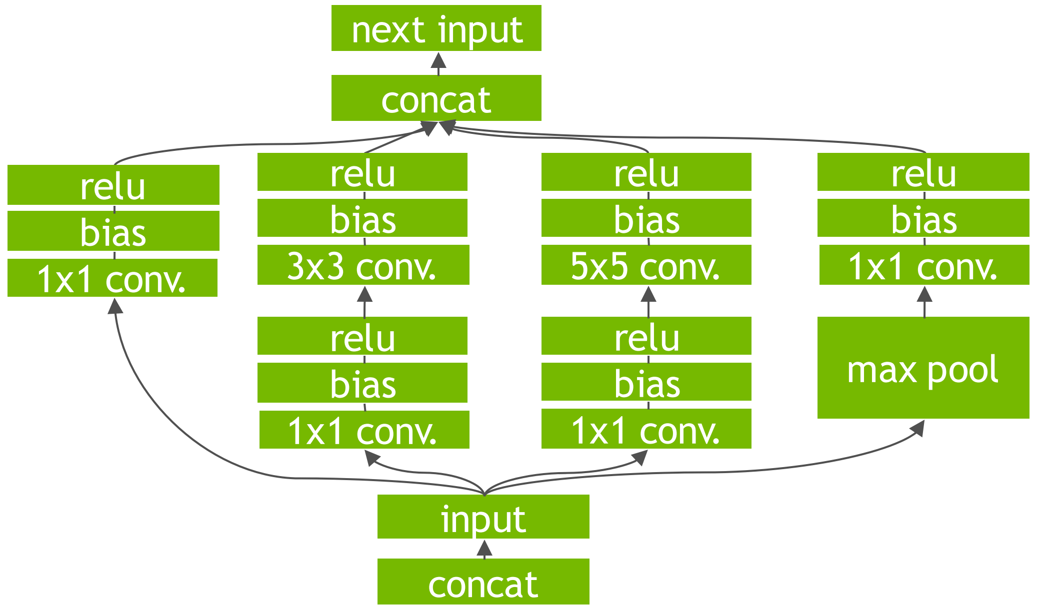
Reduced precision calibration for INT8 inference:

- Minimizes accuracy loss between FP32 and INT8 inference on a calibration dataset

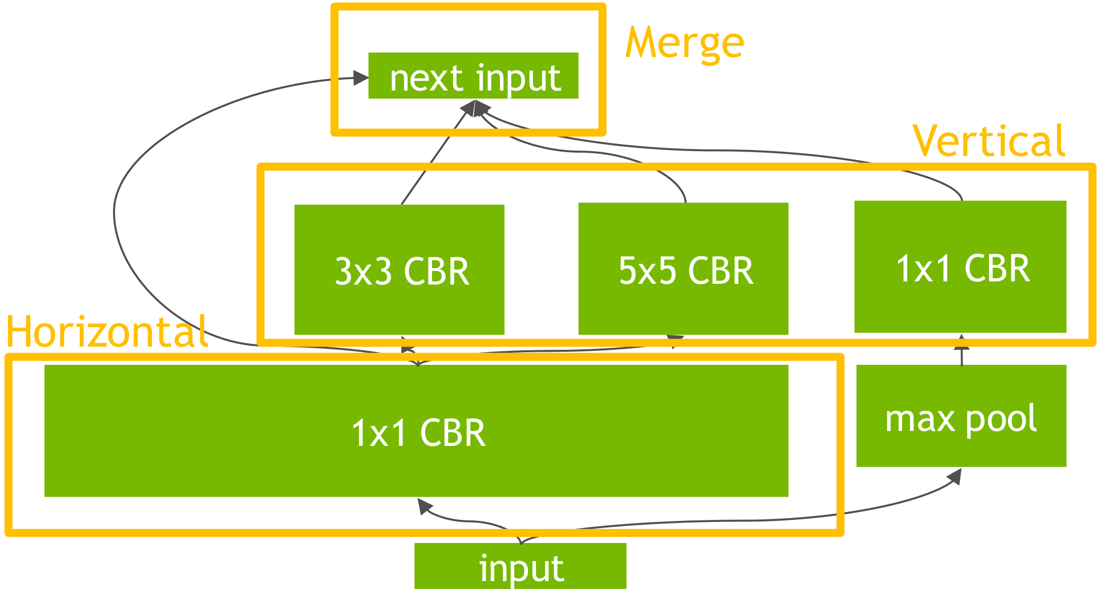
	FP32 Top 1	INT8 Top 1	Difference
Googlenet	68.87%	68.49%	0.38%
VGG	68.56%	68.45%	0.11%
Resnet-50	73.11%	72.54%	0.57%
Resnet-152	75.18%	74.56%	0.61%

LAYER & TENSOR FUSION

Un-Optimized Network



TensorRT Optimized Network



LAYER & TENSOR FUSION

- Vertical Fusion
- Horizonal Fusion
- Concat Elision
- … …

Network	Layers before	Layers after
VGG19	43	27
Inception V3	309	113
ResNet-152	670	159

Supported Layer Fusions

Convolution and ReLU Activation

FullyConnected and ReLU Activation

Scale and Activation

Convolution And ElementWise Sum

Shuffle and Reduce

Shuffle and Shuffle

Scale(add 0, multiply by 1)

Convolution and Scale

Reduce

… …

KERNEL AUTO-TUNING

Kernel Auto-Tuning

100s for specialized kernels
Optimized for every GPU platform

A100

Jetson AGX

DRIVE AGX

Multiple factors:

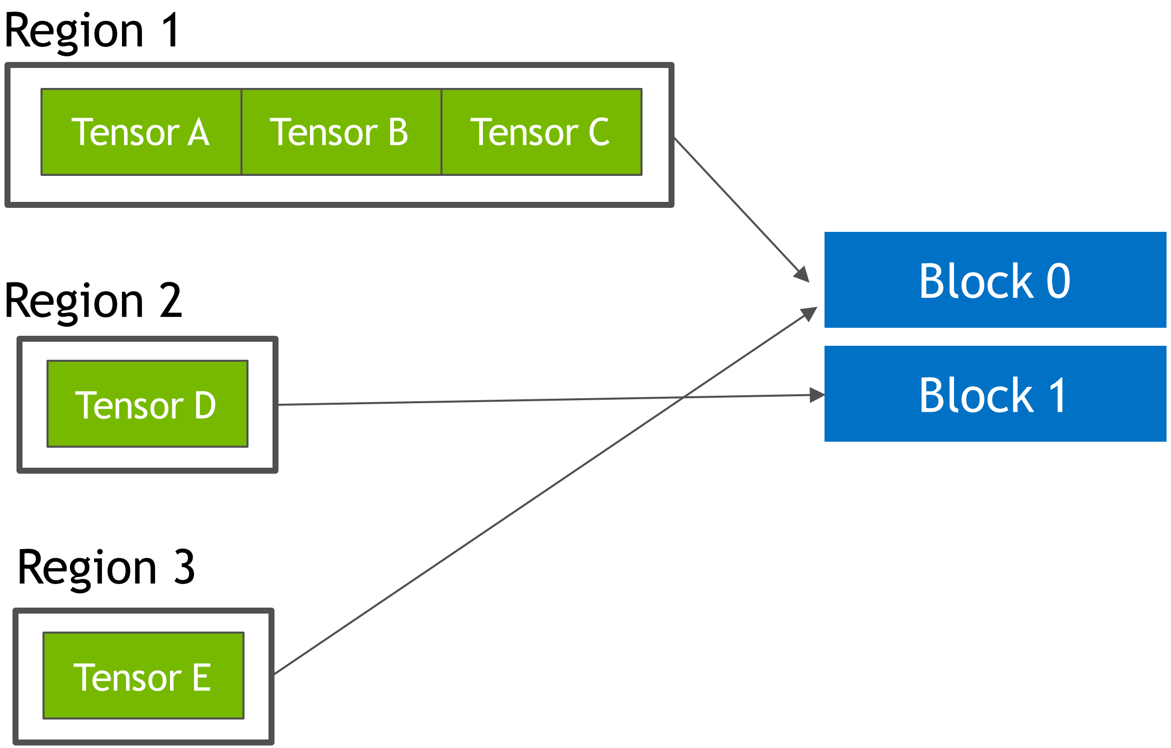
- Target platform
- Batch size
- Input dimensions
- Filter dimensions
- Tensor layout

Choice:

- Implementation of specific algorithm
- Kernels
- Tensor layouts

```
683 [03/17/2020-21:23:14] [V] [TRT] --------------- Timing Runner: (Unnamed Layer* 2) [Convolution] (C
684 [03/17/2020-21:23:14] [V] [TRT] (Unnamed Layer* 2) [Convolution] (scudnn) Set Tactic Name:
volta_scudnn_128x128_relu_medium_nn_v1
                                                                          ime 0.572736
                                                                  on] (scudnn_winograd) Set Tactic Name
volta_scudnn_winograd_128x128_ldg1_ldg4_relu_tile148t_nt_v1
                                                                          ime 0.235456
                                                                  on] (scudnn) Set Tactic Name:
volta_scudnn_128x64_relu_xregs_large_nn_v1
```

DYNAMIC TENSOR MEMORY

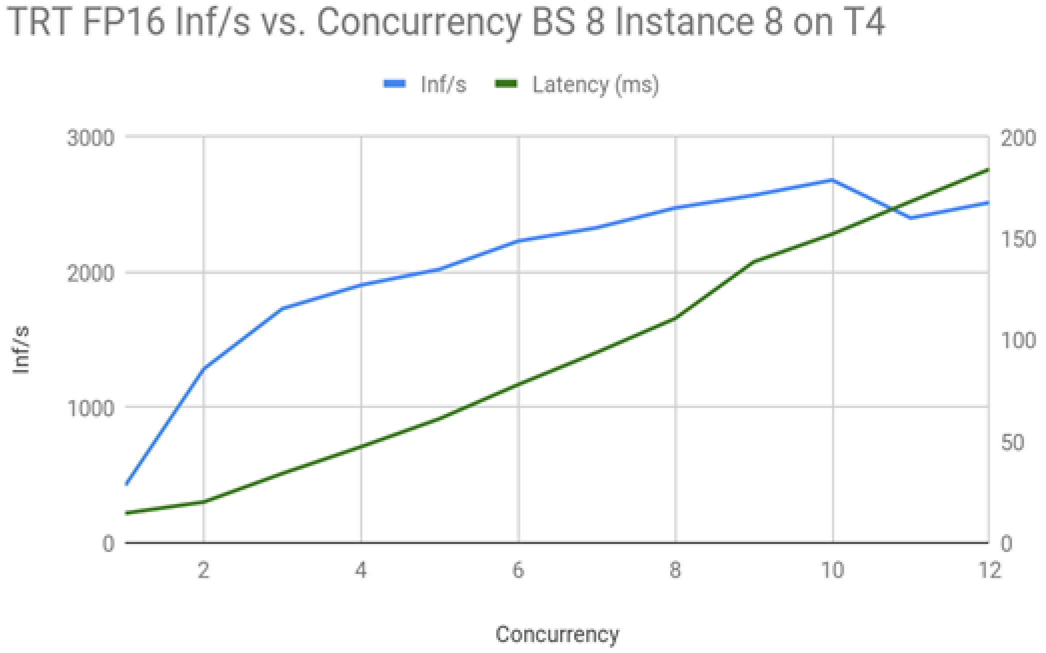


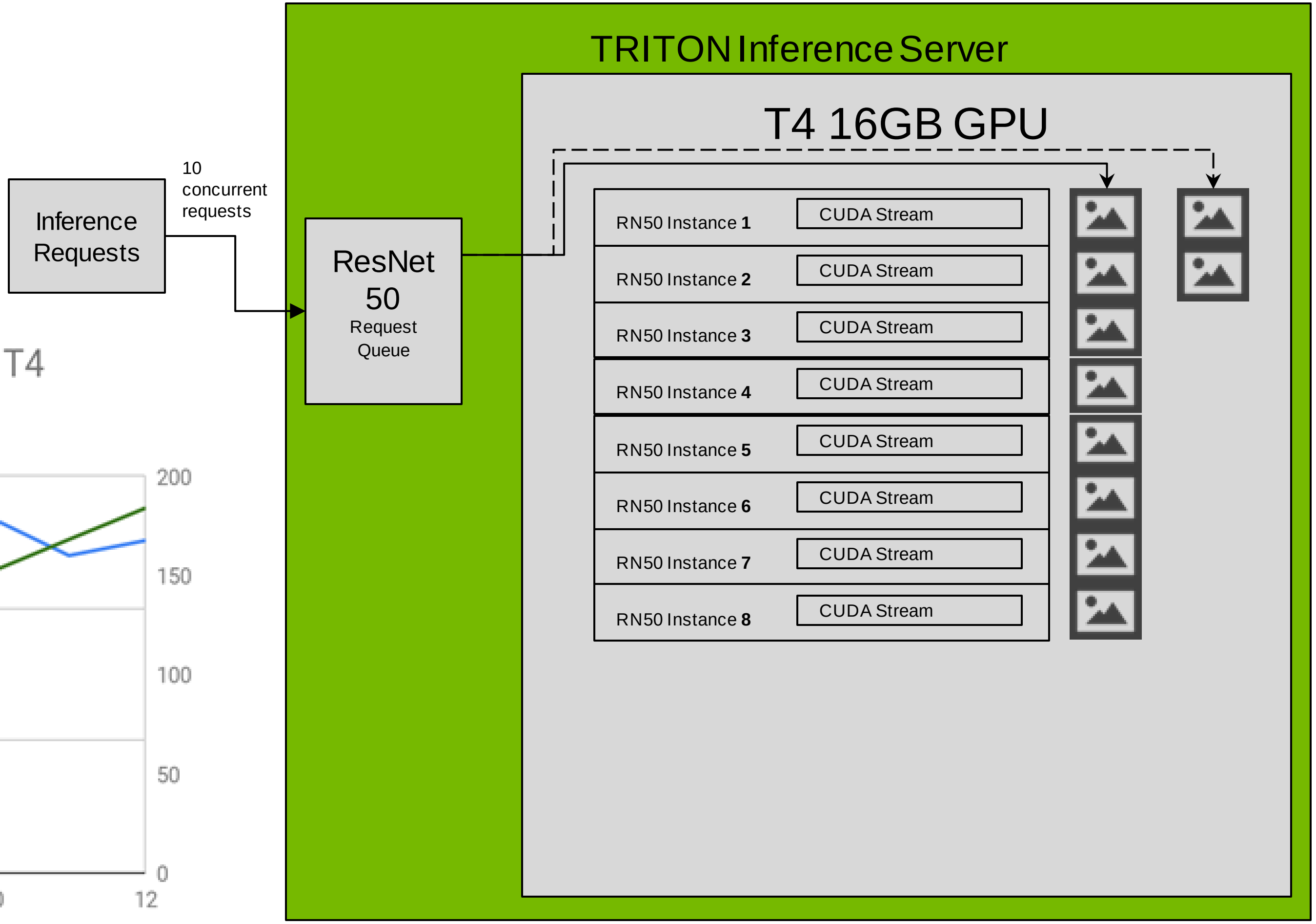
- Reduces memory footprint and improves memory re-use
- Graph Optimizer combines tensors into regions
- Region lifetime is a section of network execution time
- Memory Optimizer assigns regions to blocks; regions assigned to a block have disjoint lifetimes
- Just like register allocation

MULTI-STREAM CONCURRENT EXECUTION

Resnet50 serving on T4

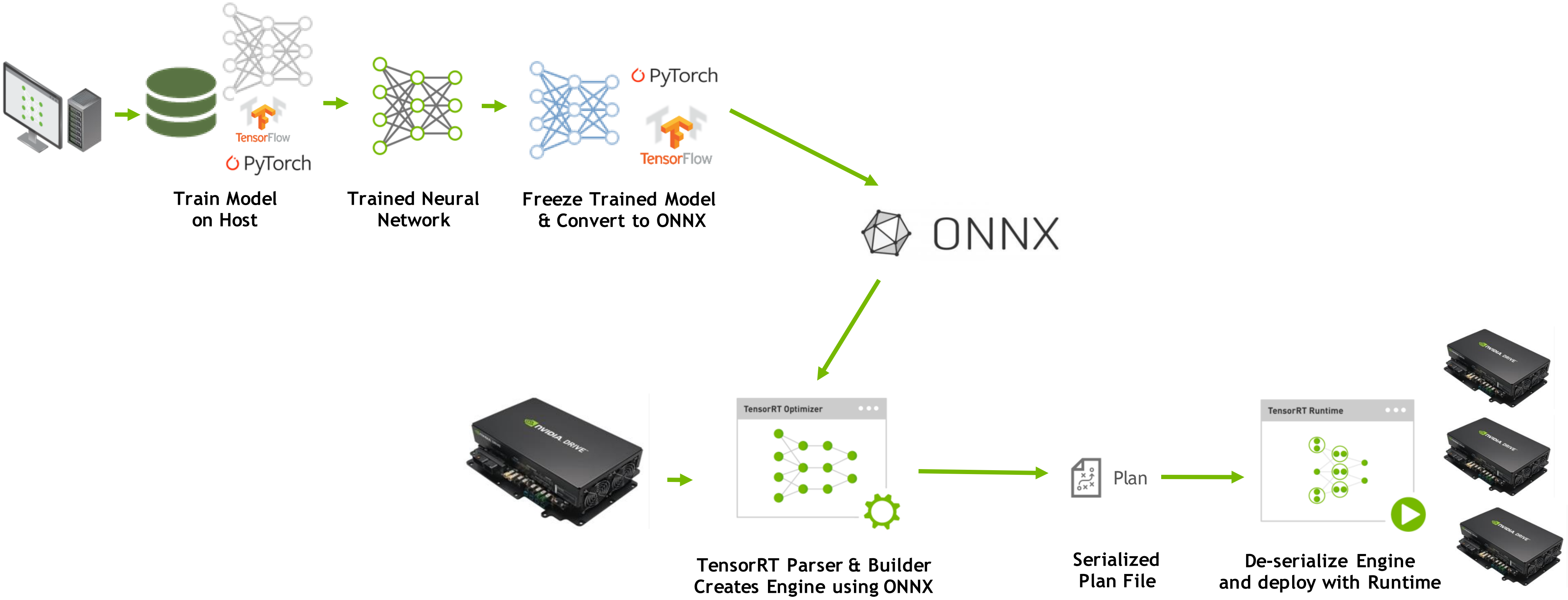
6x Better Performance and Improved GPU Utilization Through Multiple Stream Concurrency



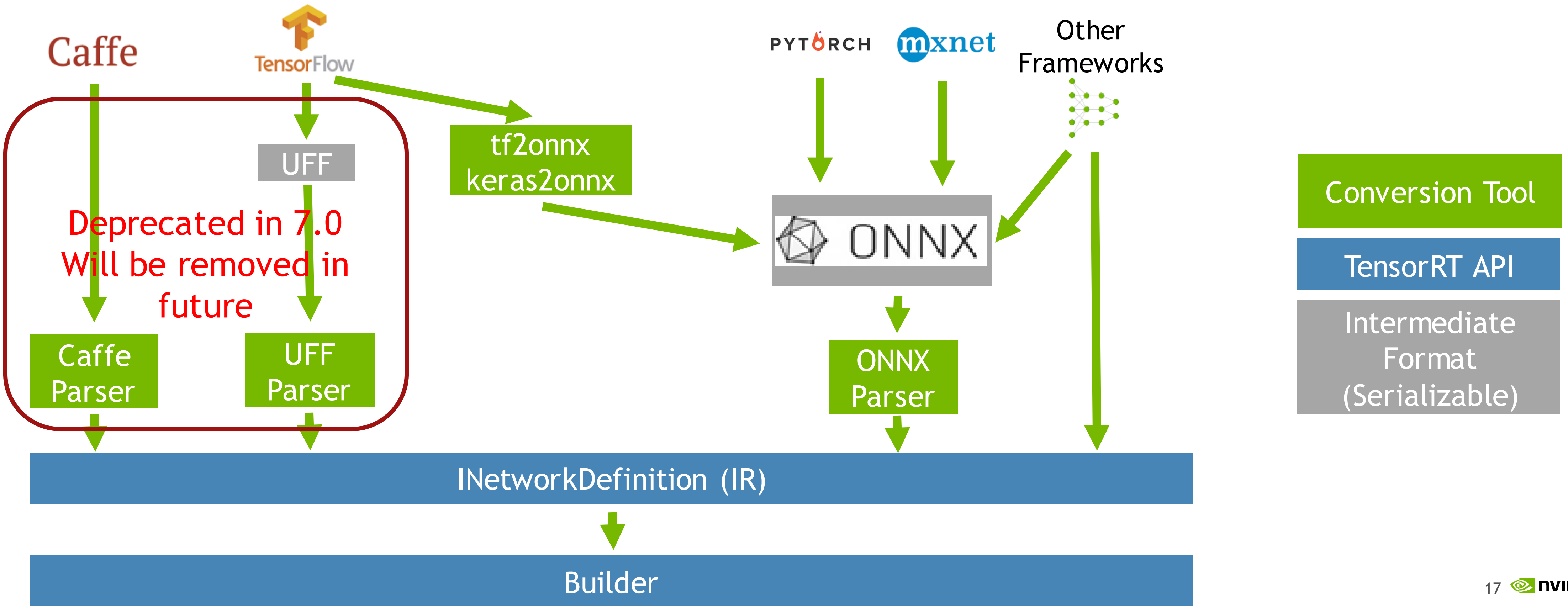


WORKFLOW WITH AN EXAMPLE

WORKFLOW FROM DL FRAMEWORK TO TENSORRT

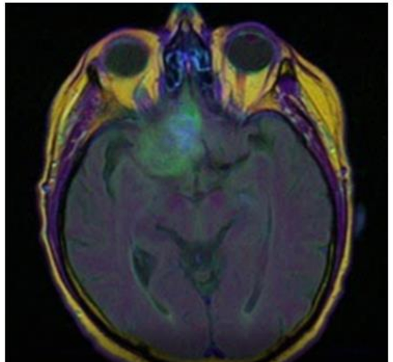


WORKFLOW – MODEL IMPORT



UNET EXAMPLE : PYTORCH TO ONNX

```
import torch
from torch.autograd import Variable
import torch.onnx as torch_onnx
import onnx
def main():
    input_shape = (3, 256, 256)
    model_onnx_path = "unet.onnx"
    dummy_input = Variable(torch.randn(1, *input_shape))
    model = torch.hub.load('mateuszbuda/brain-segmentation-pytorch', 'unet',
      in_channels=3, out_channels=1, init_features=32, pretrained=True)
    model.train(False)

    inputs = ['input.1']
    outputs = ['186']
    dynamic_axes = {'input.1': {0: 'batch'}, '186':{0:'batch'}}
    out = torch.onnx.export(model, dummy_input, model_onnx_path, input_names=inputs,
     output_names=outputs, dynamic_axes=dynamic_axes)

if __name__=='__main__':
    main()
```



(2a): Original MRI input image

(2b): Segmented ground truth from test dataset

UNET EXAMPLE : ONNX PARSER AND BUILD

```
ICudaEngine* createCudaEngine(string const& onnxModelPath )
{
    const auto explicitBatch = 1U << static_cast<uint32_t>(nvinfer1::NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
    unique_ptr<nvinfer1::IBuilder, Destroy<nvinfer1::IBuilder>> builder{nvinfer1::createInferBuilder(gLogger)};
    unique_ptr<nvinfer1::INetworkDefinition, Destroy<nvinfer1::INetworkDefinition>> network{builder->createNetworkV2(explicitBatch)};
    unique_ptr<nvonnxparser::IParser, Destroy<nvonnxparser::IParser>> parser{nvonnxparser::createParser(*network, gLogger)};
    unique_ptr<nvinfer1::IBuilderConfig,Destroy<nvinfer1::IBuilderConfig>> config{builder->createBuilderConfig()};

    if (!parser->parseFromFile(onnxModelPath.c_str(), static_cast<int>(ILogger::Severity::kINFO)))
    {
        cout << "ERROR: could not parse input engine." << endl;
        return nullptr;
    }

    config->setMaxWorkspaceSize(MAX_WORKSPACE_SIZE);
    builder->setFp16Mode(builder->platformHasFastFp16());


    auto profile = builder->createOptimizationProfile();
    profile->setDimensions(network->getInput(0)->getName(), OptProfileSelector::kMIN, Dims4{1, 3, 256 , 256});
    profile->setDimensions(network->getInput(0)->getName(), OptProfileSelector::kOPT, Dims4{1, 3, 256 , 256});
    profile->setDimensions(network->getInput(0)->getName(), OptProfileSelector::kMAX, Dims4{32, 3, 256 , 256});
    config->addOptimizationProfile(profile);

    return builder->buildEngineWithConfig(*network, *config);
}
```

UNET EXAMPLE : SERIALIZE ENGINE

```
void createAndSerializeEngine(string const& onnxModelPath)
{
    string enginePath{getBasename(onnxModelPath) + ".engine"};
    ICudaEngine* engine{nullptr};

    engine = createCudaEngine(onnxModelPath);

    if (engine)
    {
        unique_ptr<IHostMemory, Destroy<IHostMemory>> engine_plan{engine->serialize()};
        // Save engine for future uses.
        writeBuffer(engine_plan->data(), engine_plan->size(), enginePath);
    }

}
```

UNET EXAMPLE : DESERIALIZE ENGINE

```
ICudaEngine* deserializeEngine(string const& enginePath)
{
    string enginePath{getBasename(enginePath) + ".engine"};
    ICudaEngine* engine{nullptr};

    string buffer = readBuffer(enginePath);

    if (buffer.size())
    {
        // Try to deserialize engine.
        unique_ptr<IRuntime, Destroy<IRuntime>> runtime{createInferRuntime(gLogger)};
        engine = runtime->deserializeCudaEngine(buffer.data(), buffer.size(), nullptr);

    }

    return engine;
}
```

UNET EXAMPLE : INFERENCE

```
void launchInference(IExecutionContext* context, cudaStream_t stream, vector<float> const&
inputTensor, vector<float>& outputTensor, void** bindings, int batchSize)
{
    int inputId = getBindingInputIndex(context);

    cudaMemcpyAsync(bindings[inputId], inputTensor.data(), inputTensor.size() * sizeof(float),
    cudaMemcpyHostToDevice, stream);
    context->enqueueV2(bindings, stream, nullptr);
    cudaMemcpyAsync(outputTensor.data(), bindings[1 - inputId], outputTensor.size() * sizeof(float),
    cudaMemcpyDeviceToHost, stream);
    cudaStreamSynchronize(stream);
}
```

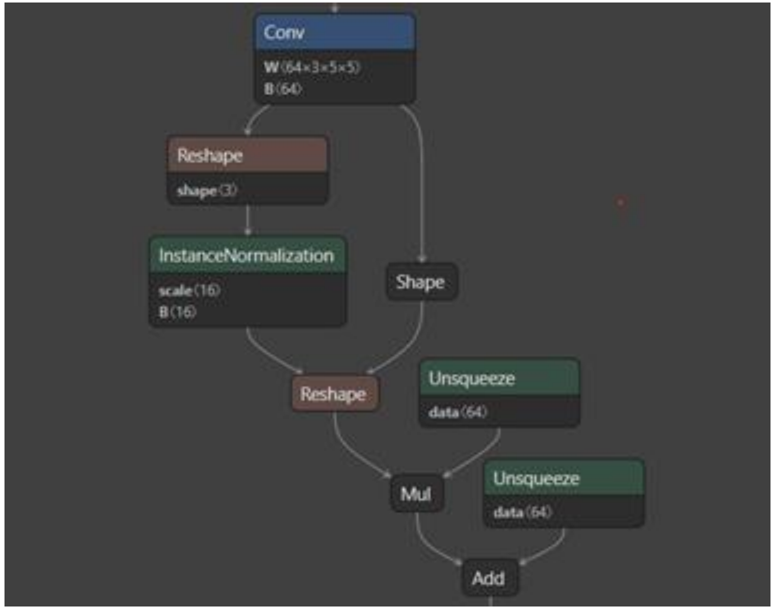
UNET EXAMPLE : APPLICATION

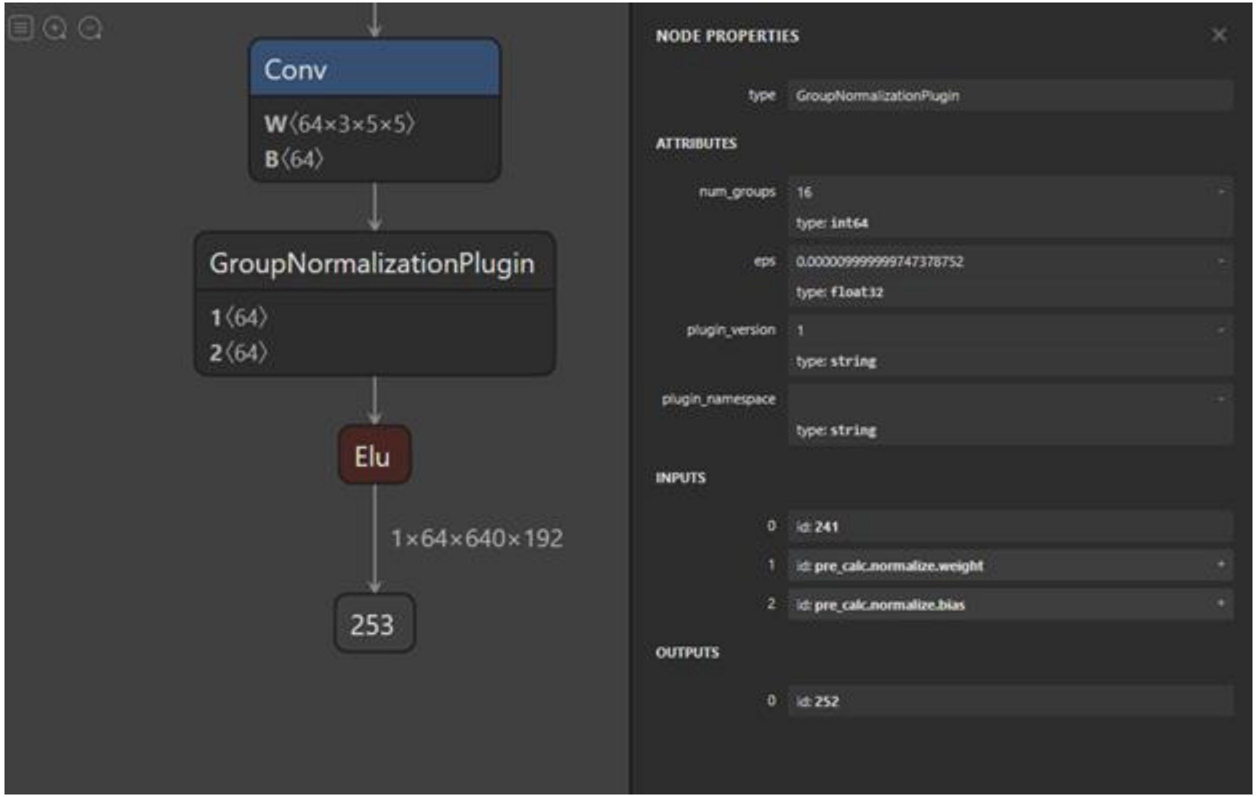
```
void main(int argc, char* argv[])
{
    unique_ptr<ICudaEngine, Destroy<ICudaEngine>> engine{nullptr};
    unique_ptr<IExecutionContext, Destroy<IExecutionContext>> context{nullptr};
    vector<float> inputTensor, outputTensor, referenceTensor;
    void* bindings[2]{0};
    CudaStream stream;
    int batchSize = 4;

    engine.reset(deserializeEngine("/tmp/unet.engine"));

    for (int i = 0; i < engine->getNbBindings(); ++i)
    {
        Dims dims{engine->getBindingDimensions(i)};
        size_t size = std::accumulate(dims.d+1, dims.d + dims.nbDims, batchSize, multiplies<size_t>());
        cudaMalloc(&bindings[i], batchSize * size * sizeof(float));
        if (engine->bindingIsInput(i))
            inputTensor.resize(size);
        else
            outputTensor.resize(size);
    }

    // Create Execution Context.
    context.reset(engine->createExecutionContext());

    Dims dims_i{engine->getBindingDimensions(0)};
    Dims4 inputDims{batchSize, dims_i.d[1], dims_i.d[2], dims_i.d[3]};
    context->setBindingDimensions(0, inputDims);

    launchInference(context.get(), stream, inputTensor, outputTensor, bindings, batchSize);
}
```

USEFUL TOOLS AND RESOURCES

USING PLUGIN WITH ONNX

ONNX GraphSurgeon





ONNX-GS : Available in open source starting from TensorRT 7.1

Allows the user to write a custom layer plugin and substitute in the ONNX graph to be parsed by TensorRT ONNX parser

BENCHMARK TOOL

trtexec

trtexec is a command line tool for performance benchmark

Source code available in open source

Pre-built binary is provided in the package and container

Sample commands:

./trtexec –onnx=unet.onnx --shapes=input:1x3x256x256 --fp16

./trtexec –onnx=unet.onnx --shapes=input:1x3x256x256 --best

Run ./trtexec --help for more advanced settings

```
root@home:/workspace/tensorrt/bin# ./trtexec --deploy=/home/ResNet50.prototxt --output=fc1000
&&&& RUNNING TensorRT.trtexec # ./trtexec --deploy=/home/ResNet50.prototxt --output=fc1000
[09/08/2020-22:16:21] [I] === Model Options ===
[09/08/2020-22:16:21] [I] Format: Caffe
[09/08/2020-22:16:21] [I] Model:
[09/08/2020-22:16:21] [I] Prototxt: /home/ResNet50.prototxt
[09/08/2020-22:16:21] [I] Output: fc1000
[09/08/2020-22:16:21] [I] === Build Options ===
[09/08/2020-22:16:21] [I] Max batch: 1
[09/08/2020-22:16:21] [I] Workspace: 16 MB
[09/08/2020-22:16:21] [I] minTiming: 1
[09/08/2020-22:16:21] [I] avgTiming: 8
[09/08/2020-22:16:21] [I] Precision: FP32
[09/08/2020-22:16:21] [I] Calibration:
[09/08/2020-22:16:21] [I] Safe mode: Disabled
[09/08/2020-22:16:21] [I] Save engine:
[09/08/2020-22:16:21] [I] Load engine:
[09/08/2020-22:16:21] [I] Builder Cache: Enabled
[09/08/2020-22:16:59] [I] Host Latency
[09/08/2020-22:16:59] [I] min: 2.9292 ms (end to end 2.94873 ms)
[09/08/2020-22:16:59] [I] max: 5.97357 ms (end to end 8.49121 ms)
[09/08/2020-22:16:59] [I] mean: 3.03532 ms (end to end 5.31506 ms)
[09/08/2020-22:16:59] [I] median: 2.99298 ms (end to end 5.49902 ms)
[09/08/2020-22:16:59] [I] percentile: 3.42749 ms at 99% (end to end 6.19971 ms at 99%)
[09/08/2020-22:16:59] [I] throughput: 323.412 qps
[09/08/2020-22:16:59] [I] walltime: 3.00546 s
[09/08/2020-22:16:59] [I] Enqueue Time
[09/08/2020-22:16:59] [I] min: 0.213867 ms
[09/08/2020-22:16:59] [I] max: 1.729 ms
[09/08/2020-22:16:59] [I] median: 0.855591 ms
[09/08/2020-22:16:59] [I] GPU Compute
[09/08/2020-22:16:59] [I] min: 2.8252 ms
[09/08/2020-22:16:59] [I] max: 5.80713 ms
[09/08/2020-22:16:59] [I] mean: 2.90018 ms
[09/08/2020-22:16:59] [I] median: 2.86328 ms
[09/08/2020-22:16:59] [I] percentile: 3.29834 ms at 99%
[09/08/2020-22:16:59] [I] total compute time: 2.81897 s
```

ACCURACY VALIDATION TOOL

Poligraphy (Coming Soon)

Polygraphy is a Python based toolkit designed to assist in running and debugging deep learning models in various frameworks

Source code will be available in open source

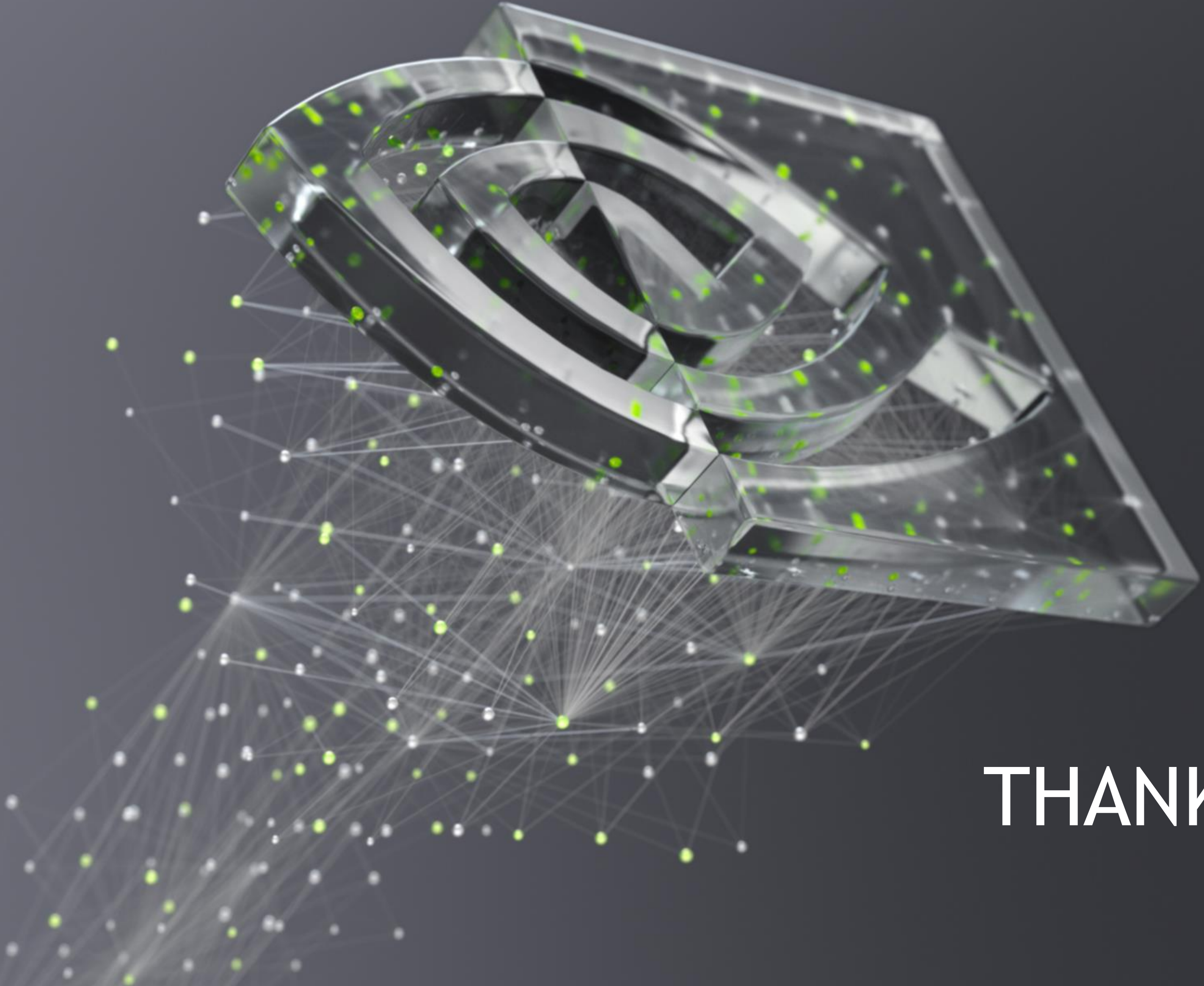
Polygraphy will be shipped inside TensorRT container

```python
from polygraphy.backend.onnx import OnnxFromTfGraph, BytesFromOnnx
from polygraphy.backend.onnxrt import OnnxrtRunner, SessionFromOnnxBytes
from polygraphy.backend.tf import TfRunner, GraphFromFrozen, SessionFromGraph
from polygraphy.backend.trt import TrtRunner, EngineFromNetwork, NetworkFromOnnxBytes
from polygraphy.comparator import Comparator, DataLoader

# Convert the model into the various formats we care about.
load_frozen = GraphFromFrozen("/path/to/frozen/model.pb")
build_tf_session = SessionFromGraph(load_frozen)
export_serialized_onnx = BytesFromOnnx(OnnxFromTfGraph(load_frozen))
build_onnxrt_session = SessionFromOnnxBytes(export_serialized_onnx)
build_engine = EngineFromNetwork(NetworkFromOnnxBytes(export_serialized_onnx))

# We want to run the model with TensorFlow, ONNX Runtime, and TensorRT.
runners = [
    TfRunner(build_tf_session),
    OnnxrtRunner(build_onnxrt_session),
    TrtRunner(build_engine),
]

# For this model, assume inputs need to be bounded.
data_loader = DataLoader(int_range=(0, 2), float_range=(0.0, 2.0))

# Finally, run and check accuracy.
run_results = Comparator.run(runners, data_loader=data_loader)
assert bool(Comparator.compare_accuracy(run_results))
```

DOWNLOAD TensorRT TODAY!

TensorRT DOCUMENTATION	*GETTING STARTED RESOURCES*	*FRAMEWORK INTEGRATIONS*
Installation Guide	GTC WEBINARS	PyTorch
Programming Guide	NVIDIA Developer Blog	Caffe2
API Reference	GitHub	Cognitive Toolkit
Samples		Chainer
		PaddlePaddle
		mxnet
		MATLAB

Free download to members of NVIDIA Developer Program at
developer.nvidia.com/tensorrt



THANK YOU!