



NVIDIA DL INFERENCE TECHNOLOGY

Jonghwan Lee, DL Solutions Architect, jonghwani@nvidia.com

JUL. 2nd, 2019 @ AI conference Korea 2019



AGENDA

Introduction

- HW for efficient Inference
- Algorithm for efficient inference

About TensorRT

- What is TensorRT?
- How to use TensorRT?

Learn More

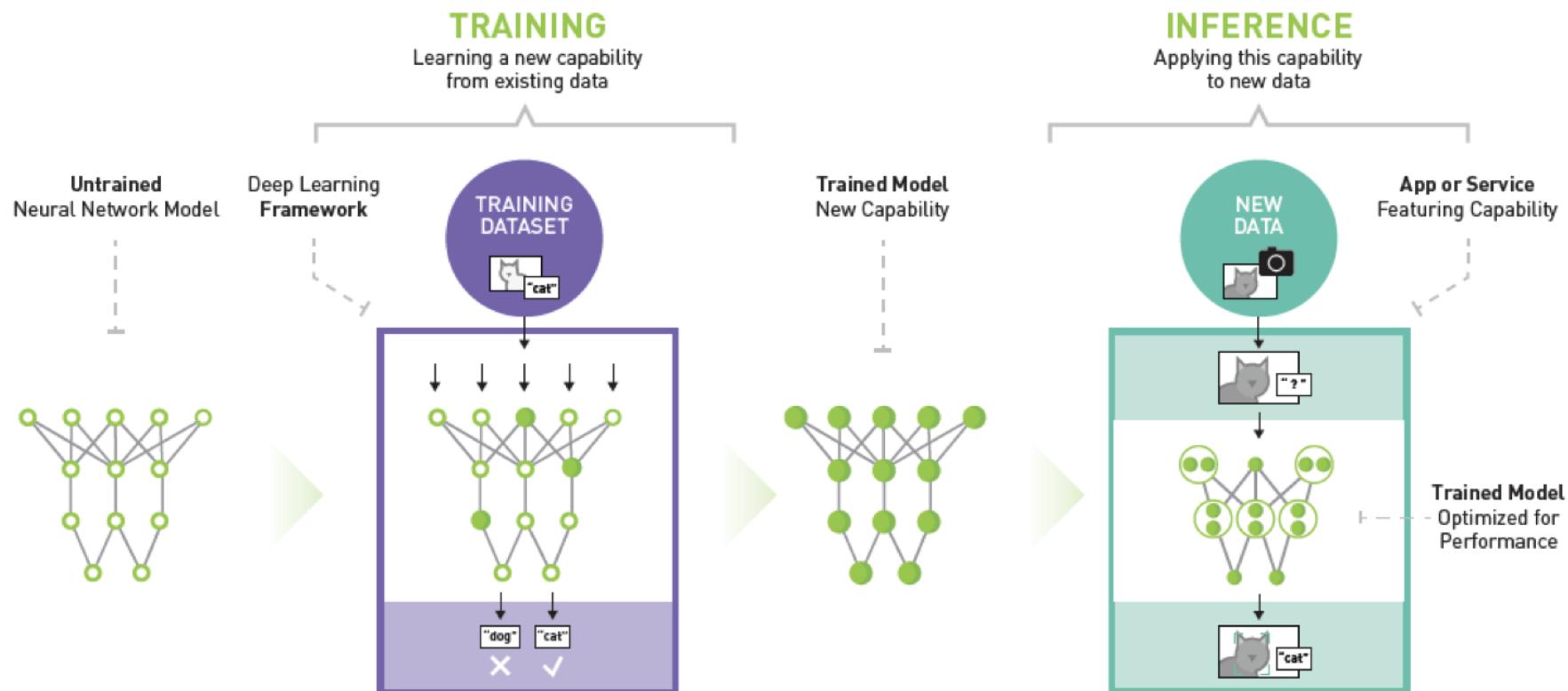
- Resources
- Related Sessions



INTRODUCTION

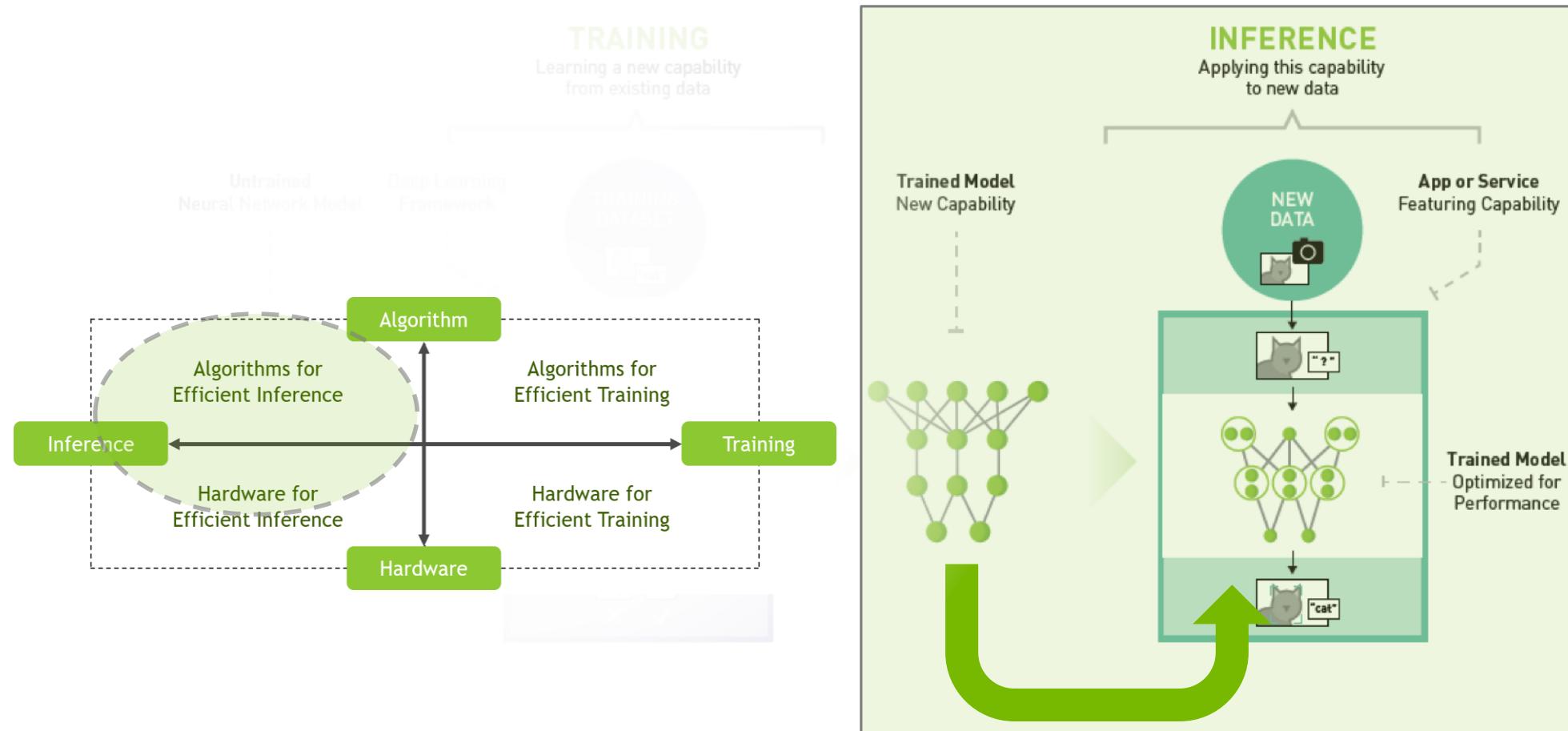
DEEP LEARNING WORKFLOW

Train Neural network and Deploy it



DEEP LEARNING WORKFLOW

How to deploy Neural Network efficiently

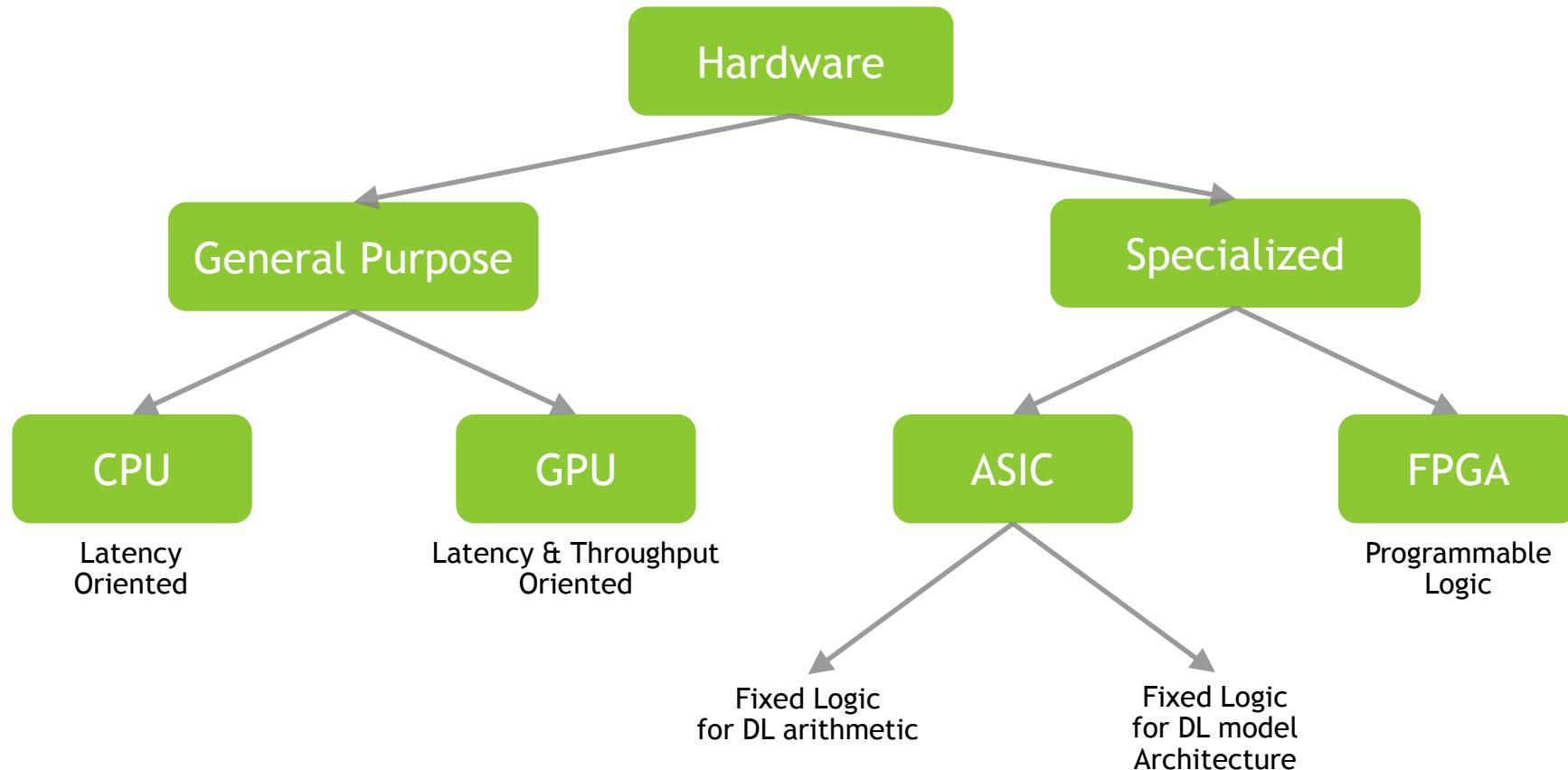




HARDWARE FOR EFFICIENT INFERENCE

DEEP LEARNING PLATFORM

Various platform to deploy Deep Learning



DEEP LEARNING PLATFORM

Considerations for DL inference Hardware

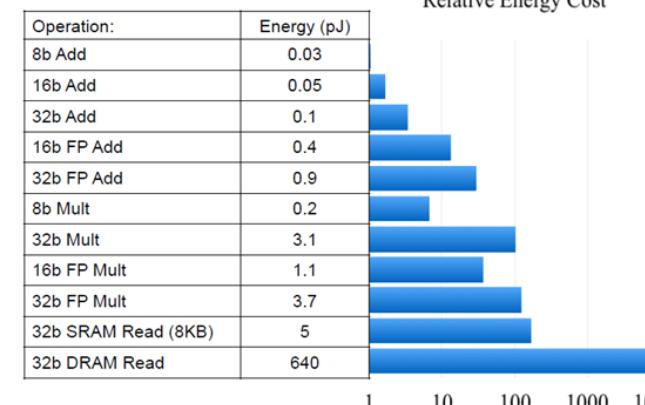
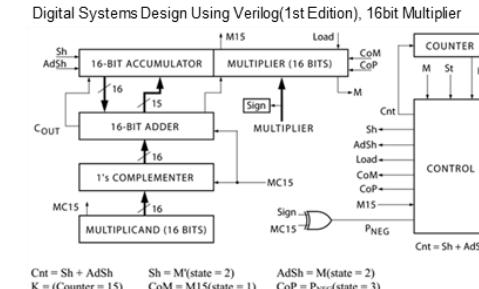
- Number Representation

		Range	Accuracy
FP32	S E M	$10^{-38} - 10^{38}$.000006%
FP16	S E M	$6 \times 10^{-5} - 6 \times 10^4$.05%
Int32	S M	$0 - 2 \times 10^9$	$\frac{1}{2}$
Int16	S M	$0 - 6 \times 10^4$	$\frac{1}{2}$
Int8	S M	$0 - 127$	$\frac{1}{2}$
Fixed point	S I F	-	-

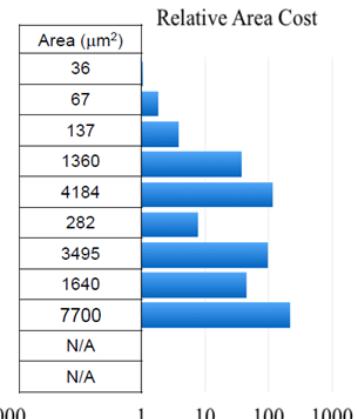
Dally, High Performance Hardware for Machine Learning, NIPS'2015



Lower-bit & Non-floating data type
gives efficiency to both Memory and Logic-size



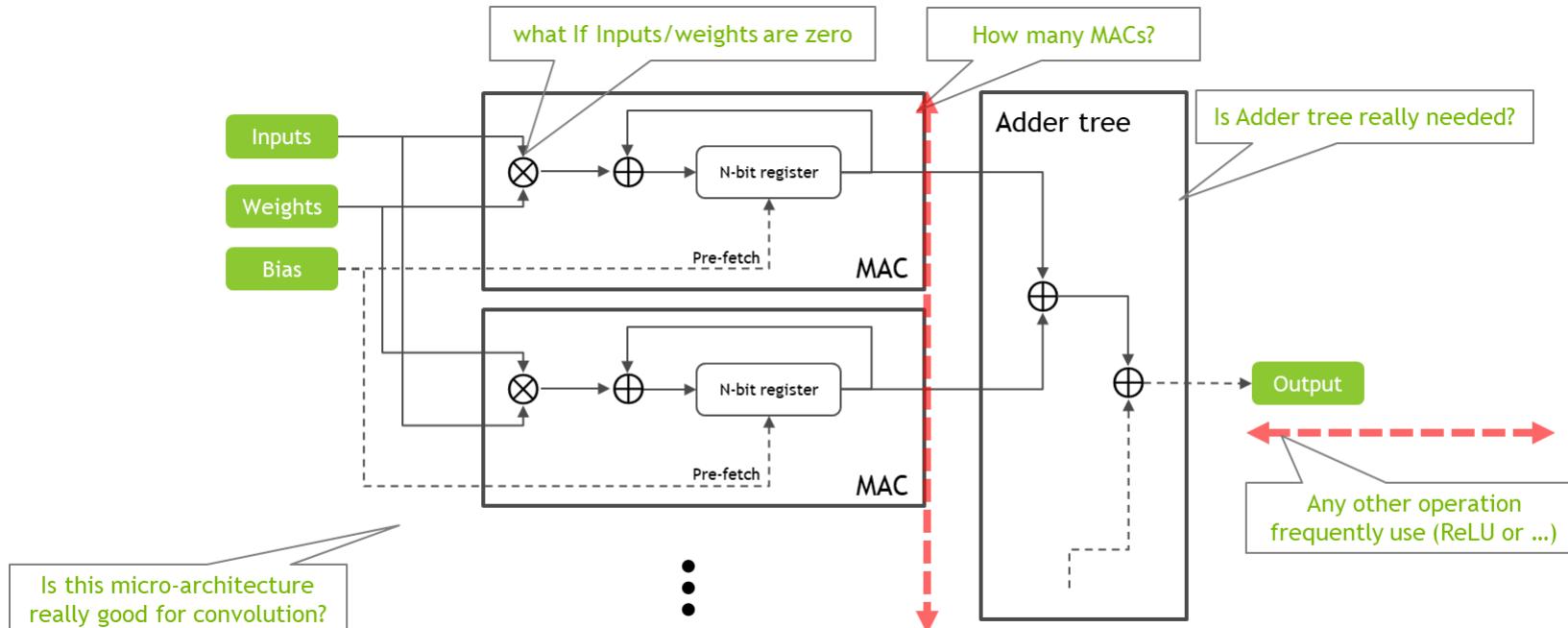
Energy numbers are from Mark Horowitz "Computing's Energy Problem", ISSCC 2014
Area numbers are from synthesized result using Design Compiler
under TSMC 45nm tech node. FP units used DesignWareLibrary.



DEEP LEARNING PLATFORM

Considerations for DL inference Hardware

- Arithmetic pipeline (specialized DL HW case)

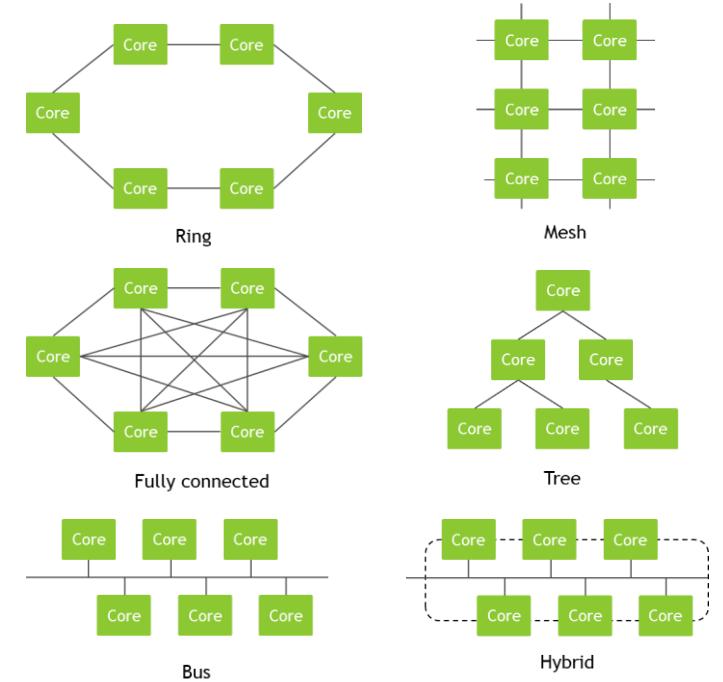
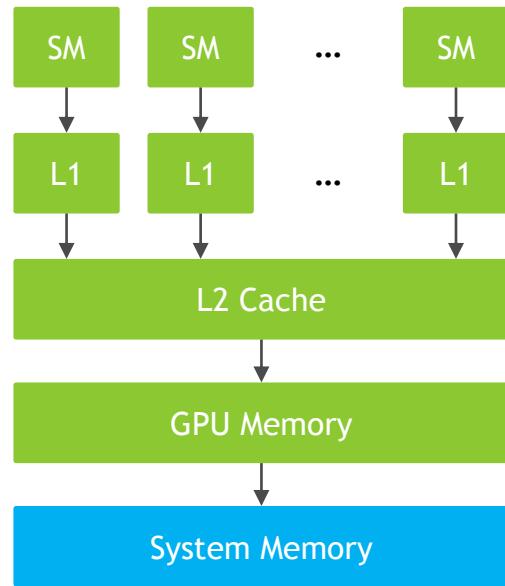
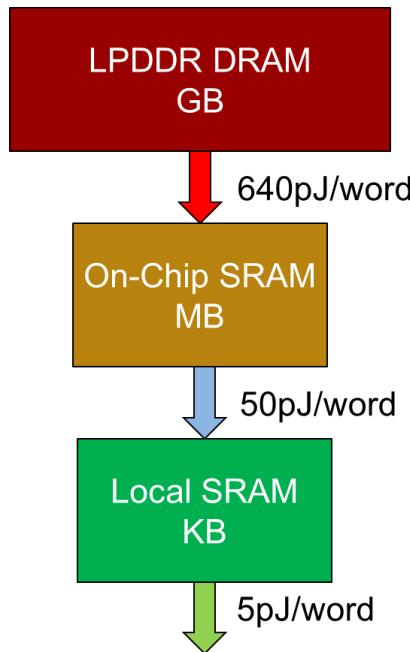


Simple Convolution engine in ALU

DEEP LEARNING PLATFORM

Considerations for DL inference Hardware

- Memory & Bus architecture



POWERING THE AI REVOLUTION

NVIDIA is advancing GPU computing for deep learning and AI at the speed of light. We create the entire stack. It starts with the most advanced GPUs and the systems and software we build on top of them. We integrate and optimize every deep learning framework. We work with the major systems companies and every major cloud service provider to make GPUs available in datacenters and in the cloud. And we create computers and software to bring AI to the edge, from self-driving cars to autonomous robots to medical devices.



ONE ARCHITECTURE





ALGORITHM FOR EFFICIENT INFERENCE

ALGORITHMS FOR EFFICIENT INFERENCE

Optimization Techniques

- Network Optimization
 - ✓ Layer/Tensor Fusion
 - ✓ Winograd Transformation
- Network Compression
 - ✓ Network Pruning
 - ✓ Weight Sharing
 - ✓ Quantization & Calibration
 - ✓ Low Rank Approximation
 - ✓ Huffman Encoding

ALGORITHMS FOR EFFICIENT INFERENCE

Network Optimization Techniques

- Network Optimization

- ✓ Layer Fusion

Ex) Merge(Convolution, BN) = Convolution'
(convolution) $Y = W * X + b$
(BN) $Y = a * X + b'$
(Merged) $Y = (aW) * X + (ab + b')$

- ✓ Winograd Transformation

Ex) 3x3 Convolution

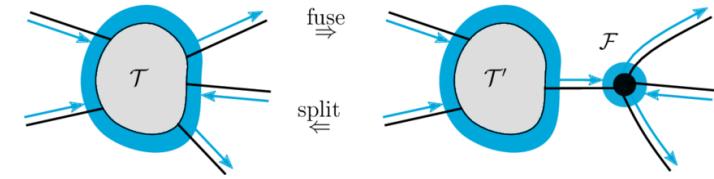
- Input transform

$$\text{Input Feature Map} = \left(\begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \right) \cdot \left(\begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \right)^T$$

- Kernel transform

$$\text{Kernel} = \left(\begin{bmatrix} \frac{1}{4} & \frac{1}{4} & 0 \\ 0 & \frac{1}{4} & 0 \\ \frac{1}{4} & 0 & 0 \end{bmatrix} \right) \cdot \left(\begin{bmatrix} \frac{1}{4} & \frac{1}{4} & 0 \\ 0 & \frac{1}{4} & 0 \\ \frac{1}{4} & 0 & 0 \end{bmatrix} \right)^T$$

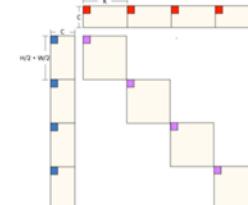
- ✓ Tensor Fusion



- output transform

$$\text{Output Feature Map} = \left(\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \right) \cdot \left(\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \right)^T$$

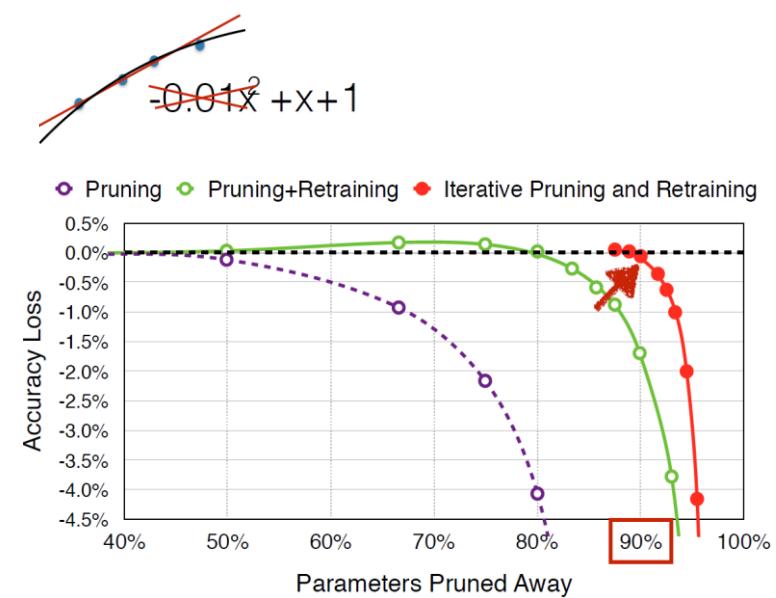
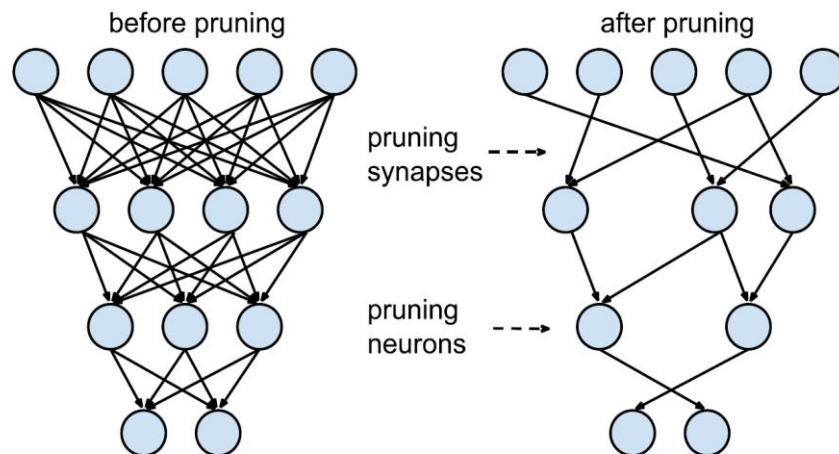
Batched-Gemm



ALGORITHMS FOR EFFICIENT INFERENCE

Network Compression Techniques

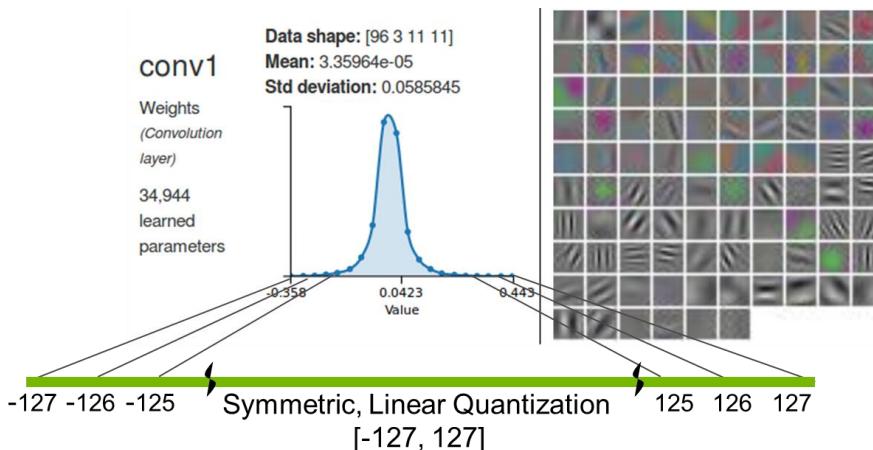
- Network Compression
 - ✓ Network Pruning



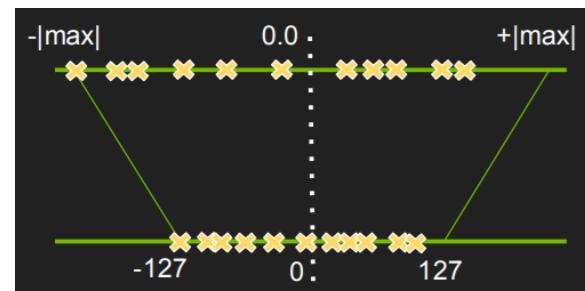
ALGORITHMS FOR EFFICIENT INFERENCE

Network Compression Techniques

- Network Compression
 - ✓ Quantization & Calibration



```
Int8_weight = Round_to_nearest_int( scaling_factor * F32_weight )
scaling_factor = 127.0f / max( abs( all_F32_weights_in_the_filter ) )
```



In general,
Low-bit quantization occurs
Significant accuracy loss.



Saturate above |T| to 127

Use calibration to get proper |T|
(To minimize information loss,
find value which shows min-entropy
on quantization)

$$D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

P: Reference probability distribution
Q: Quantized probability distribution

ALGORITHMS FOR EFFICIENT INFERENCE

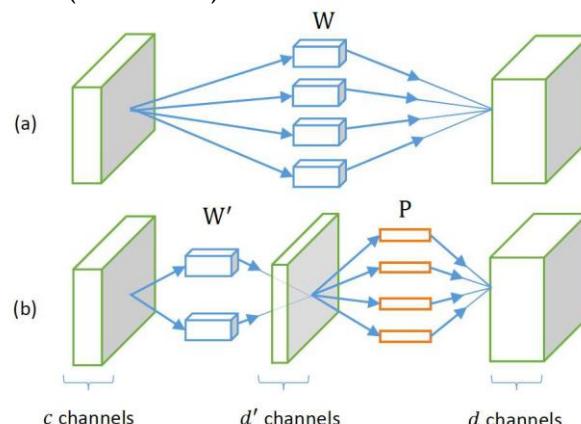
Network Compression Techniques

- Network Compression
 - ✓ Low Rank Approximation (SVD)

For Convolution layer,

Decompose a convolution layer with d filters with filter-size $k \times k \times c$ to

- Layer with d' filters ($k \times k \times c$)
- Layer with d filters ($1 \times 1 \times d'$)



For Fully-connected layer,

Build a mapping from row / column indices of matrix $\mathbf{W} = [W(x, y)]$ to vectors i and j : $x \leftrightarrow i = (i_1, \dots, i_d)$ and $y \leftrightarrow j = (j_1, \dots, j_d)$.

TT-format for matrix \mathbf{W} :

$$\mathbf{W}(i_1, \dots, i_d; j_1, \dots, j_d) = \mathbf{W}(x(i), y(j)) = \underbrace{\mathbf{G}_1[i_1, j_1]}_{1 \times r} \underbrace{\mathbf{G}_2[i_2, j_2]}_{r \times r} \dots \underbrace{\mathbf{G}_d[i_d, j_d]}_{r \times 1}$$

Notation & terminology:

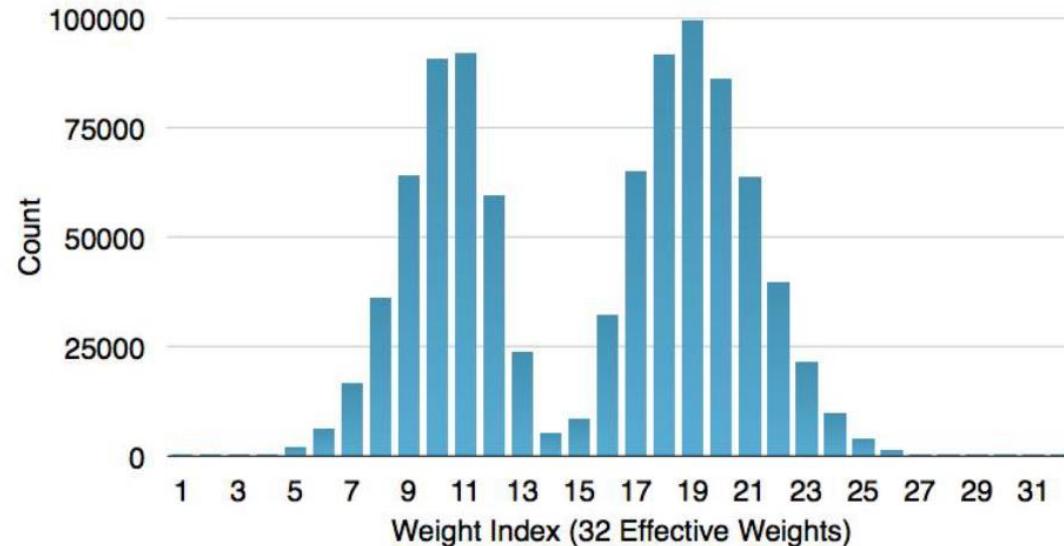
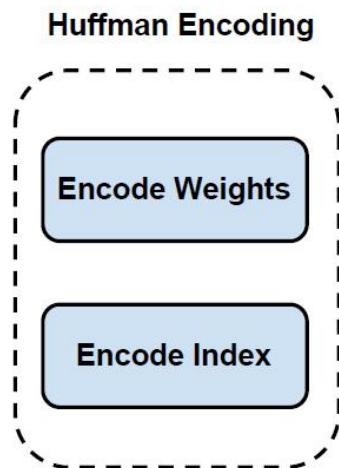
- $\mathbf{W} \in \mathbb{R}^{M \times N}$, $M = m^d$, $N = n^d$;
- $i_k \in \{1, \dots, m\}$, $j_k \in \{1, \dots, n\}$;
- \mathbf{G}_k — TT-cores;
- r — TT-rank;

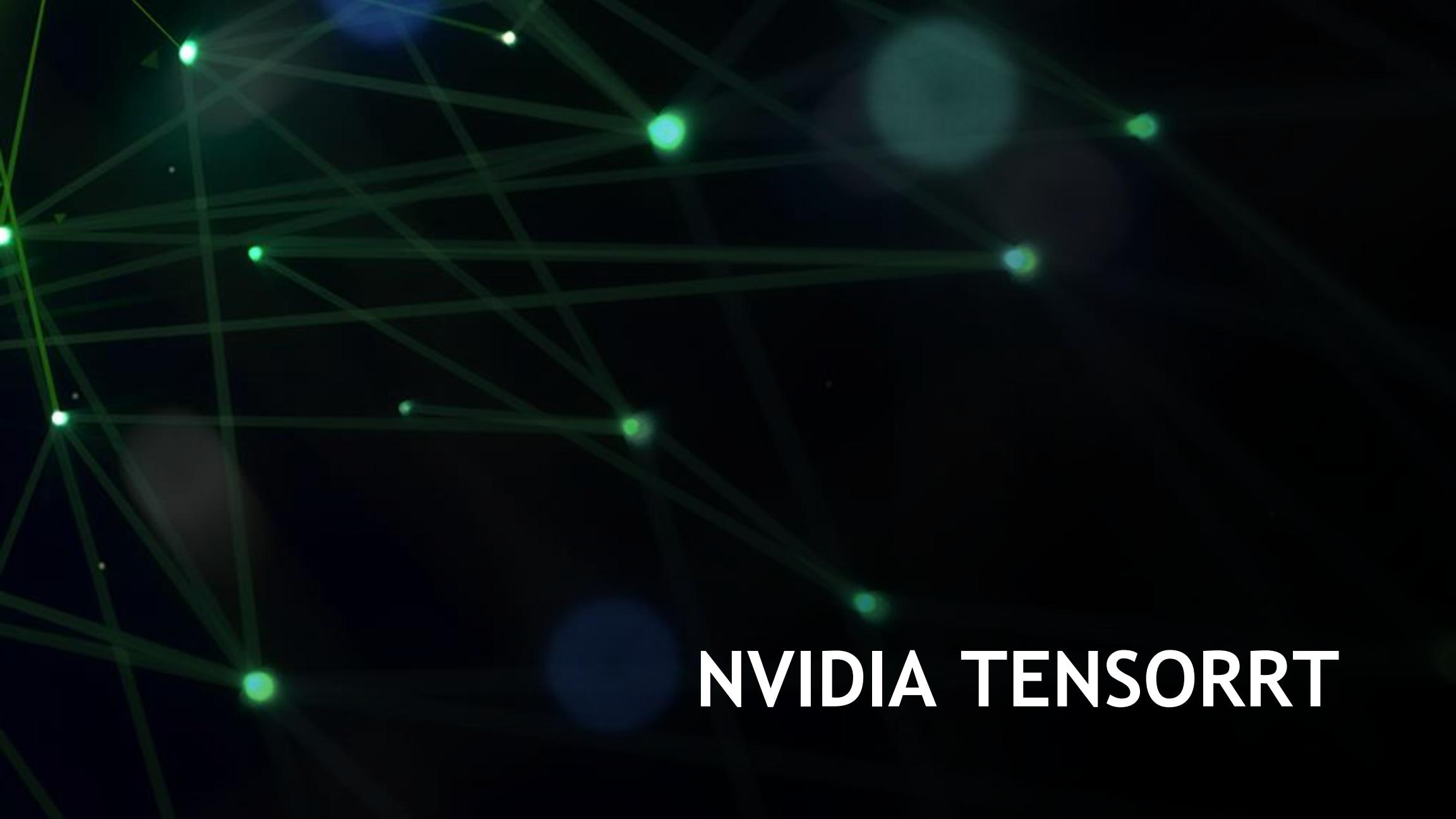
TT-format exists for any matrix \mathbf{W} and uses $O(dmnr^2)$ memory to store $O(m^d n^d)$ elements. Efficient only if TT-rank is small.

ALGORITHMS FOR EFFICIENT INFERENCE

Network Compression Techniques

- Network Compression
 - ✓ Huffman Encoding
 - In-frequent weights : use more bit to represent
 - Frequent weights : use less bit to represent

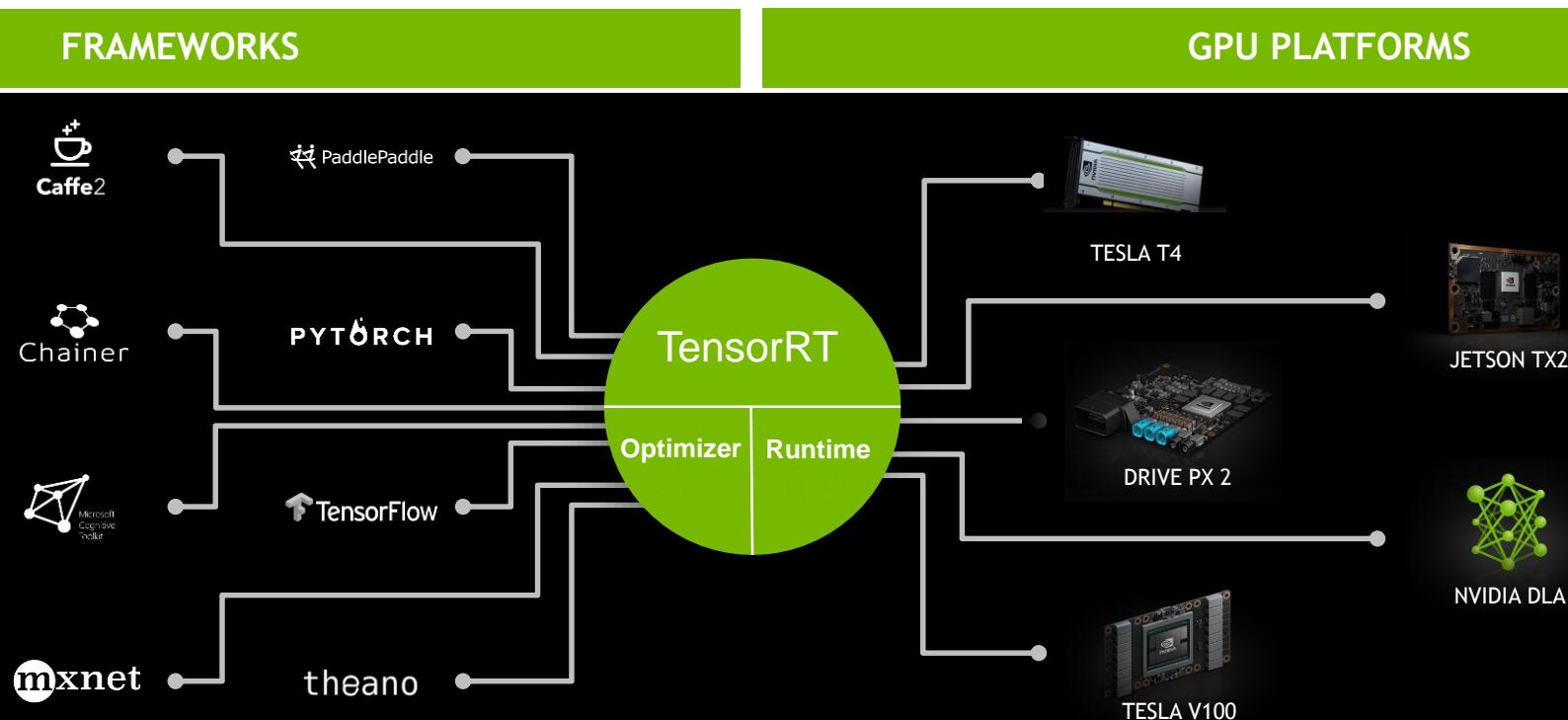




NVIDIA TENSORRT

NVIDIA TensorRT

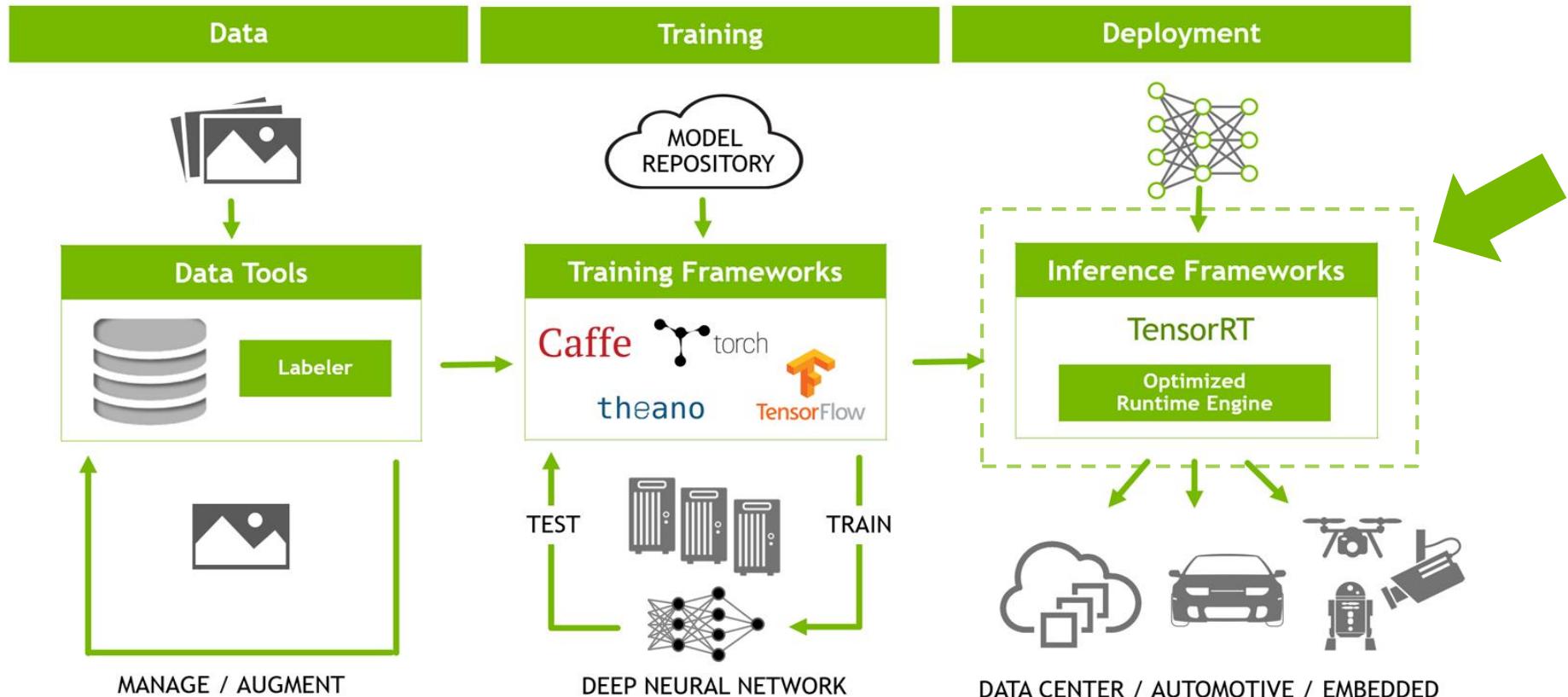
From Every Framework, Optimized For Each Target Platform



developer.nvidia.com/tensorrt

TensorRT OVERVIEW

High-performance DL Inference Engine for Production Deployment



TensorRT OVERVIEW

Platform for High-performance Deep Learning Inference

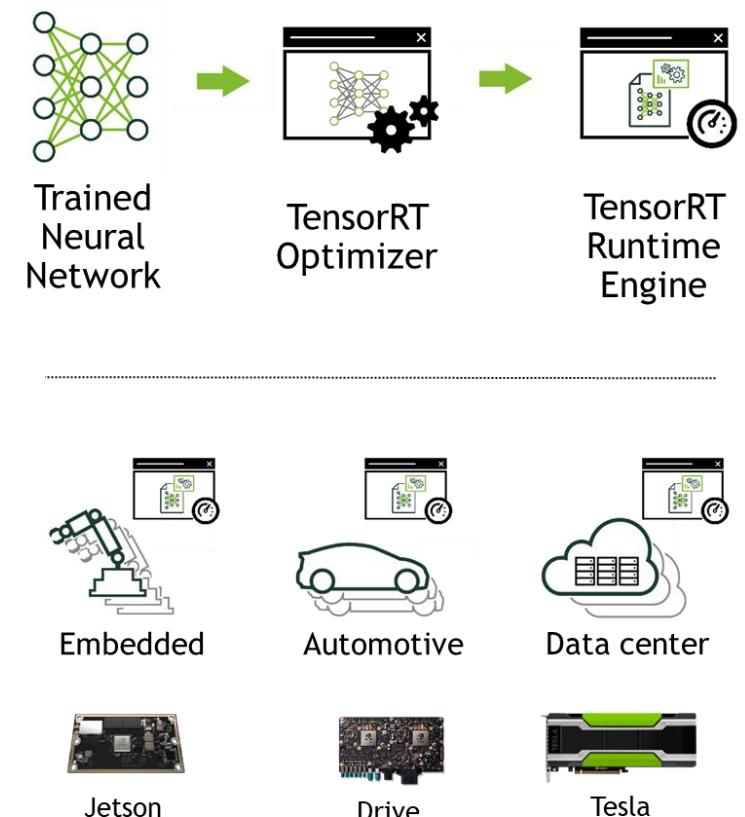
Optimize and Deploy neural networks in production environments

Maximize throughput for latency-critical apps with optimizer and runtime

Deploy responsive and memory efficient apps with INT8 & FP16 optimizations

Accelerate every framework with TensorFlow integration and ONNX support

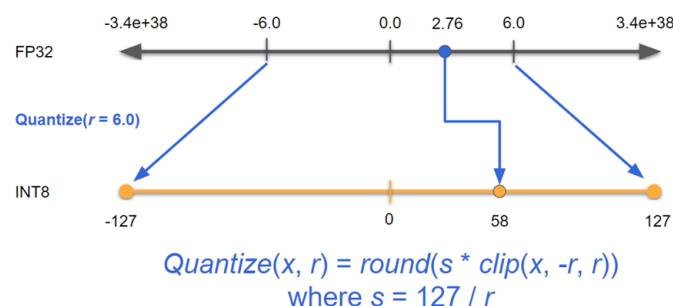
Run multiple models on a node with containerized inference server



TensorRT OPTIMIZER

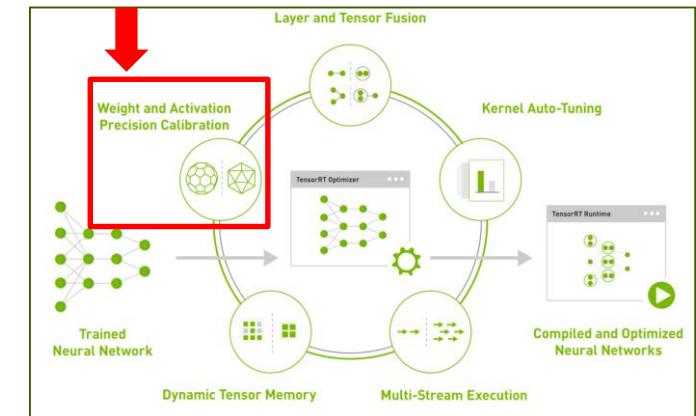
Lower-bit Quantization & Precision Calibration

Quantization



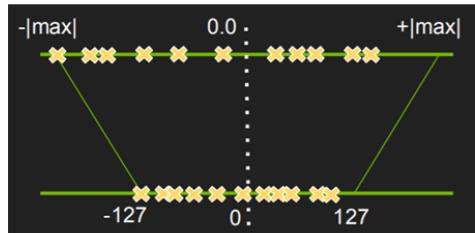
Symmetric linear quantization

x : Input
r : Floating point range
s : Scaling factor



Calibration

Saturate above $|T|$ to 127



In general,
Low-bit quantization occurs
Significant accuracy loss.



Use calibration to get proper $|T|$
(To minimize information loss,
find value which shows min-entropy
on quantization)

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

TensorRT OPTIMIZER

Lower-bit Quantization & Precision Calibration

	FP32		INT8					
			Calibration using 5 batches		Calibration using 10 batches		Calibration using 50 batches	
NETWORK	Top1	Top5	Top1	Top5	Top1	Top5	Top1	Top5
Resnet-50	73.23%	91.18%	73.03%	91.15%	73.02%	91.06%	73.10%	91.06%
Resnet-101	74.39%	91.78%	74.52%	91.64%	74.38%	91.70%	74.40%	91.73%
Resnet-152	74.78%	91.82%	74.62%	91.82%	74.66%	91.82%	74.70%	91.78%
VGG-19	68.41%	88.78%	68.42%	88.69%	68.42%	88.67%	68.38%	88.70%
Googlenet	68.57%	88.83%	68.21%	88.67%	68.10%	88.58%	68.12%	88.64%
Alexnet	57.08%	80.06%	57.00%	79.98%	57.00%	79.98%	57.05%	80.06%
NETWORK	Top1	Top5	Diff Top1	Diff Top5	Diff Top1	Diff Top5	Diff Top1	Diff Top5
Resnet-50	73.23%	91.18%	0.20%	0.03%	0.22%	0.13%	0.13%	0.12%
Resnet-101	74.39%	91.78%	-0.13%	0.14%	0.01%	0.09%	-0.01%	0.06%
Resnet-152	74.78%	91.82%	0.15%	0.01%	0.11%	0.01%	0.08%	0.05%
VGG-19	68.41%	88.78%	-0.02%	0.09%	-0.01%	0.10%	0.03%	0.07%
Googlenet	68.57%	88.83%	0.36%	0.16%	0.46%	0.25%	0.45%	0.19%
Alexnet	57.08%	80.06%	0.08%	0.08%	0.08%	0.07%	0.03%	-0.01%

In general,
Low-bit quantization occurs
Significant accuracy loss.

Use calibration to get proper |T|
(To minimize information loss,
find value which shows min-entropy
on quantization)

auto-Tuning

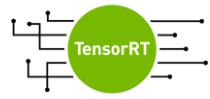


Compiled and Optimized
Neural Networks

precision

ration required

s calibration



INT8 APIs And Optimizations

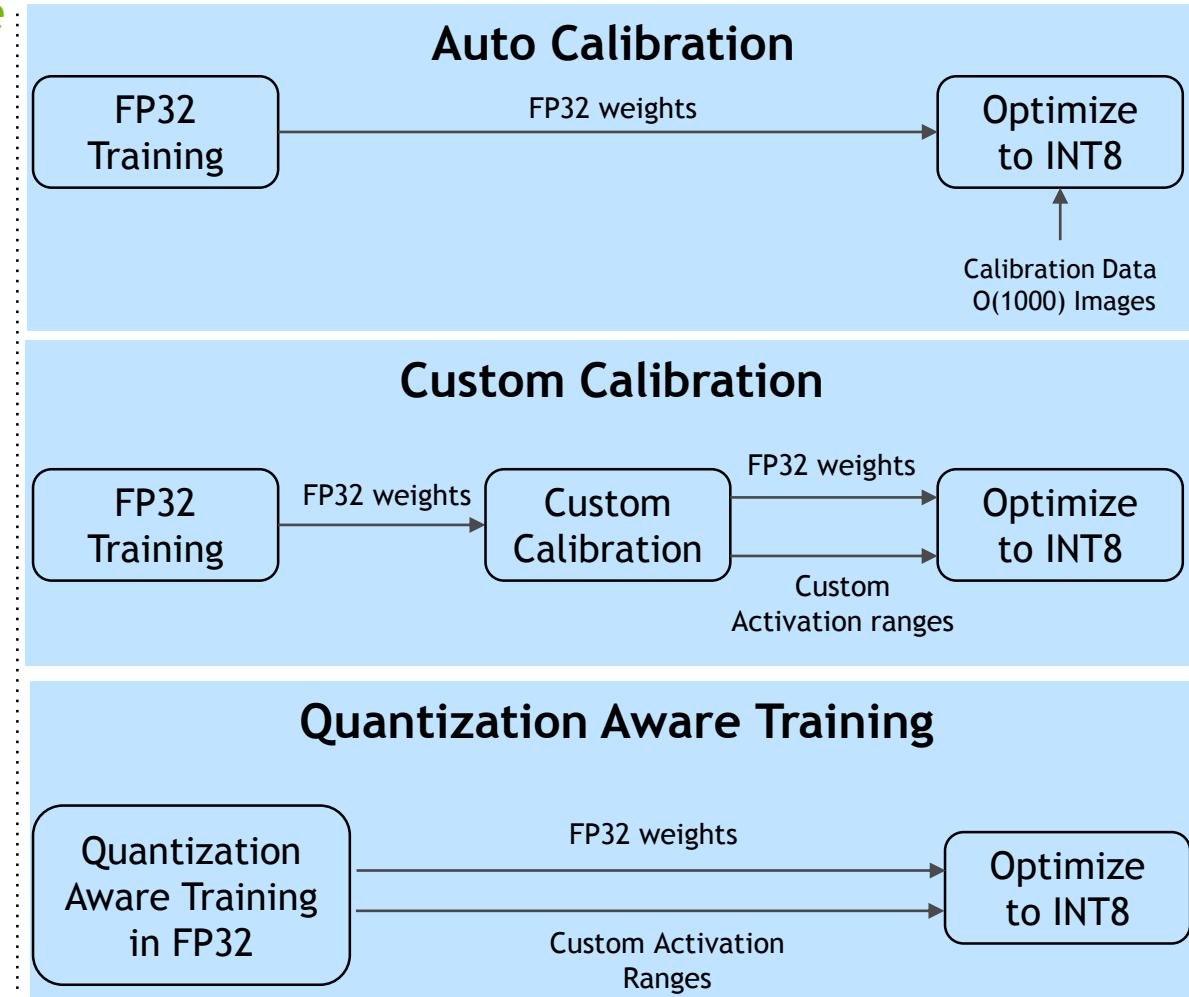
High-performance Optimizations and Flexible APIs For Mixed Precision Inference

Maximize throughput at low latency with mixed precision compute in production

Apply INT8 quantization aware training or custom calibration algorithms with new APIs

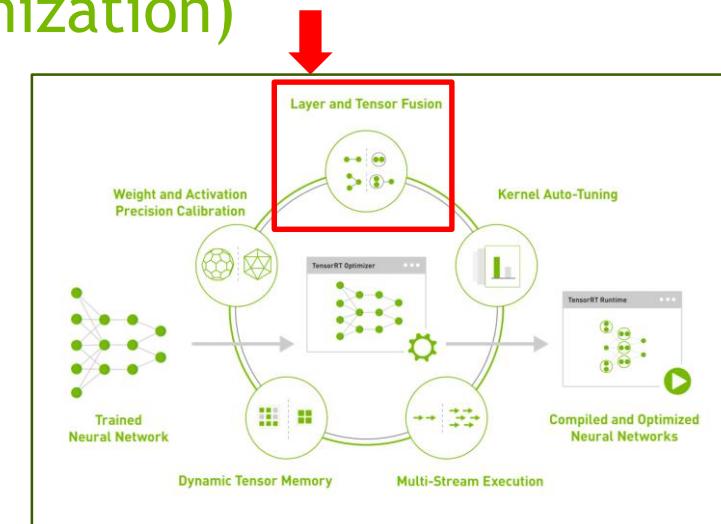
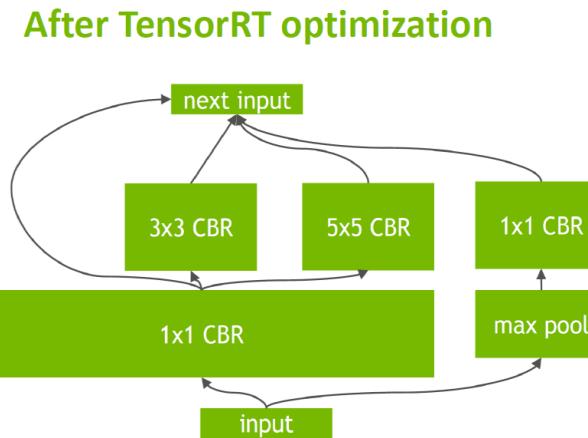
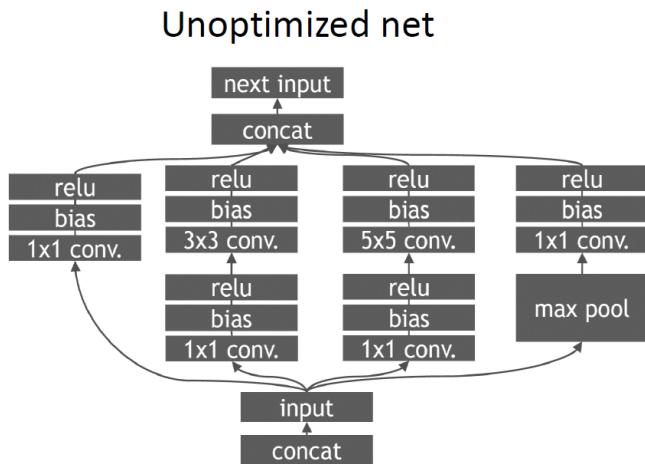
Control precision per-layer with new APIs

Optimizations for depth wise convolution operation



TensorRT OPTIMIZER

Layer & Tensor Fusion (Graph optimization)



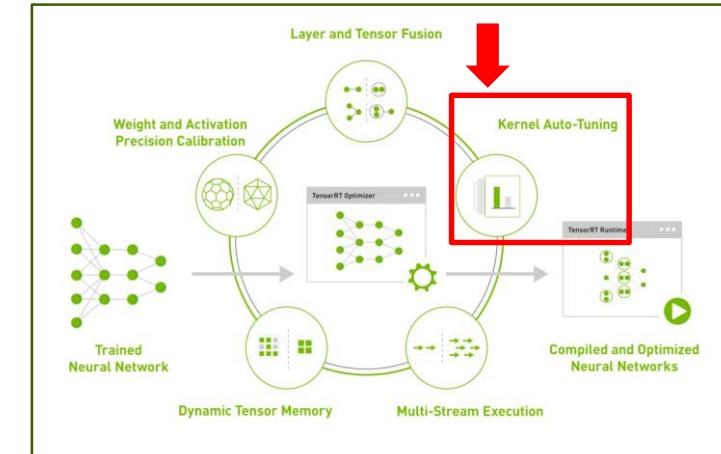
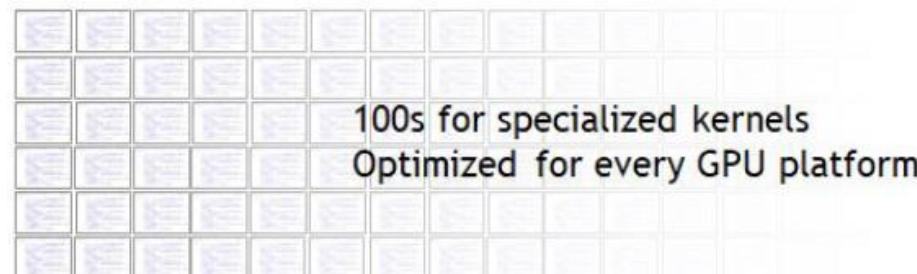
- Vertical Fusion (e.g. Conv/Bias/Relu)
- Horizontal Fusion (e.g. 1x1 CBR layers)
- Eliminate concatenation layers

Networks	Number of layers (Before)	Number of layers (After)
VGG19	43	27
Inception v3	309	113
ResNet-152	670	159

TensorRT OPTIMIZER

Kernel Auto-tuning

- Maximize Kernel performance
- Select the best performance for target GPU
- Parameters
 - Input data size
 - Batch
 - Tensor layout
 - Input dimension
 - Memory
 - Etc.



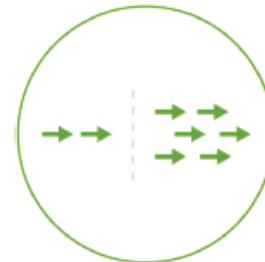
Automatic selection of best matching kernel

TensorRT OPTIMIZER

Dynamic Tensor Memory & Multi-Stream execution



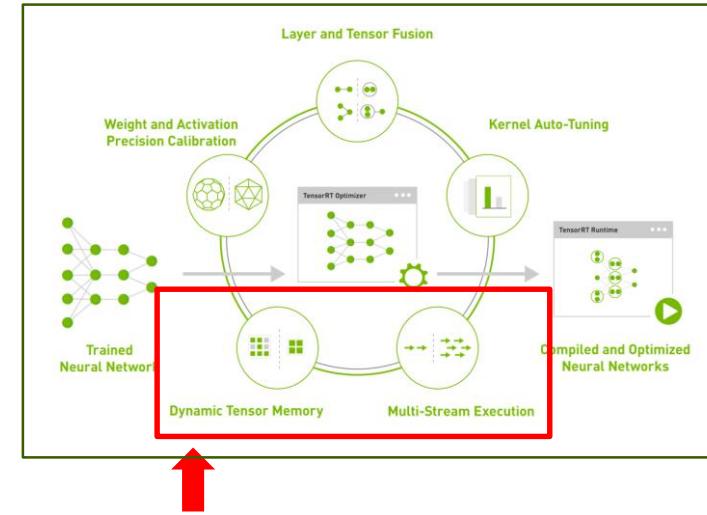
Dynamic Tensor Memory



Multi-Stream Execution

- Reduces memory footprint and improves memory re-use
- Manages memory allocation for each tensor only for the duration of its usage

- Scalable design to process multiple input streams in parallel

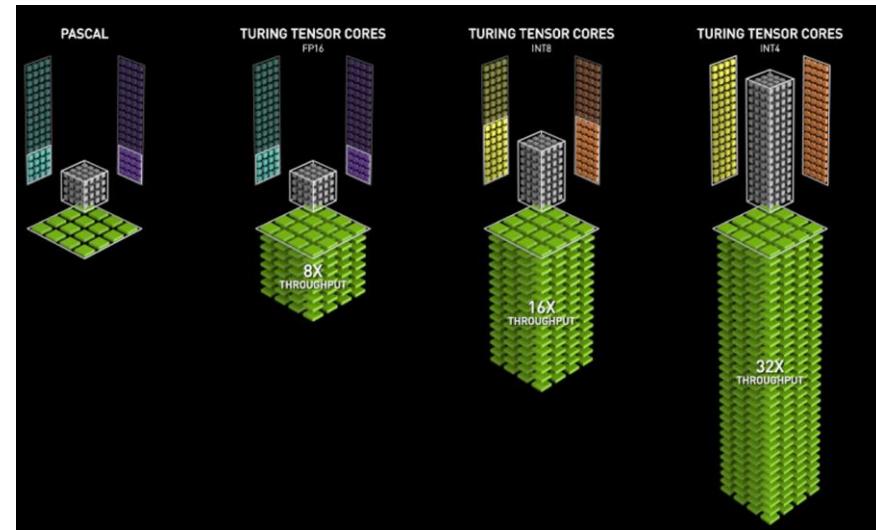


NVIDIA TENSOR CORE

Mixed Precision Matrix Math 4x4 matrices

- ▶ Available in Volta and Turing architecture GPUs
- ▶ 125 Tflops in FP16 vs 15.7 Tflops in FP32 (**8x speed-up**)
- ▶ Optimized **4x4x4** dot operation (GEMM)

$$\mathbf{D} = \left(\begin{array}{cccc} \mathbf{A}_{0,0} & \mathbf{A}_{0,1} & \mathbf{A}_{0,2} & \mathbf{A}_{0,3} \\ \mathbf{A}_{1,0} & \mathbf{A}_{1,1} & \mathbf{A}_{1,2} & \mathbf{A}_{1,3} \\ \mathbf{A}_{2,0} & \mathbf{A}_{2,1} & \mathbf{A}_{2,2} & \mathbf{A}_{2,3} \\ \mathbf{A}_{3,0} & \mathbf{A}_{3,1} & \mathbf{A}_{3,2} & \mathbf{A}_{3,3} \end{array} \right) \text{FP16 or FP32} \times \left(\begin{array}{cccc} \mathbf{B}_{0,0} & \mathbf{B}_{0,1} & \mathbf{B}_{0,2} & \mathbf{B}_{0,3} \\ \mathbf{B}_{1,0} & \mathbf{B}_{1,1} & \mathbf{B}_{1,2} & \mathbf{B}_{1,3} \\ \mathbf{B}_{2,0} & \mathbf{B}_{2,1} & \mathbf{B}_{2,2} & \mathbf{B}_{2,3} \\ \mathbf{B}_{3,0} & \mathbf{B}_{3,1} & \mathbf{B}_{3,2} & \mathbf{B}_{3,3} \end{array} \right) \text{FP16} + \left(\begin{array}{cccc} \mathbf{C}_{0,0} & \mathbf{C}_{0,1} & \mathbf{C}_{0,2} & \mathbf{C}_{0,3} \\ \mathbf{C}_{1,0} & \mathbf{C}_{1,1} & \mathbf{C}_{1,2} & \mathbf{C}_{1,3} \\ \mathbf{C}_{2,0} & \mathbf{C}_{2,1} & \mathbf{C}_{2,2} & \mathbf{C}_{2,3} \\ \mathbf{C}_{3,0} & \mathbf{C}_{3,1} & \mathbf{C}_{3,2} & \mathbf{C}_{3,3} \end{array} \right) \text{FP16 or FP32}$$



TensorRT OVERVIEW

Plug-in for custom operations

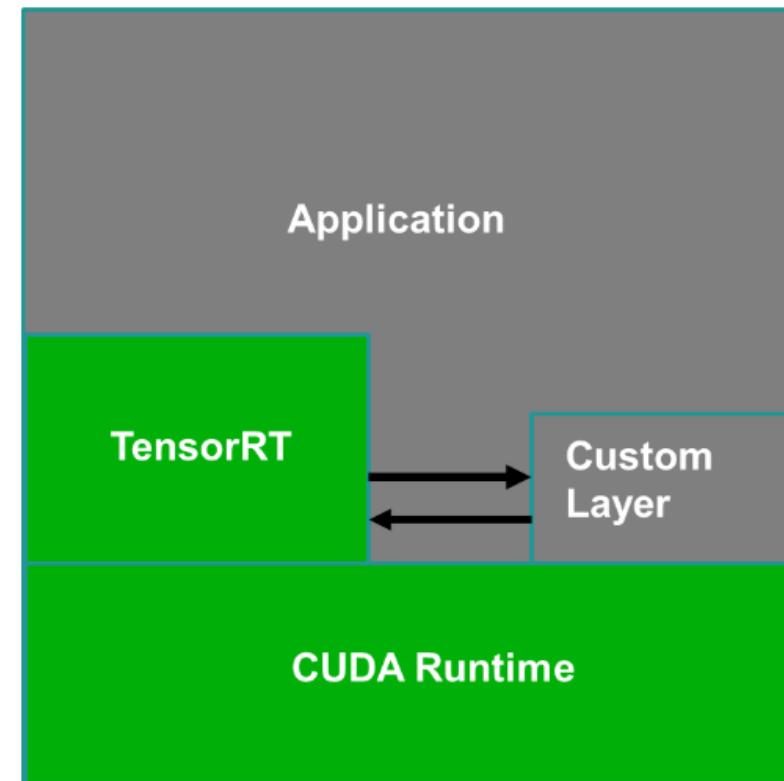
- Allow users to express and provide implementations of novel layers

TensorRT provides APIs and dedicated implementations for most common layers

Use the Custom layer API for infrequent or more innovative layers or your own confident implementation

Register custom implementations via a callback mechanism

Can be used in conjunction with reduced precision optimization

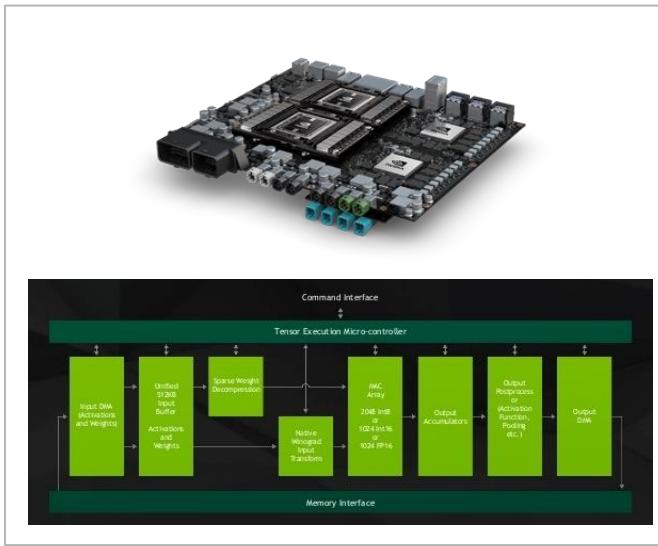


TensorRT 5.1 & FRAMEWORK INTEGRATIONS

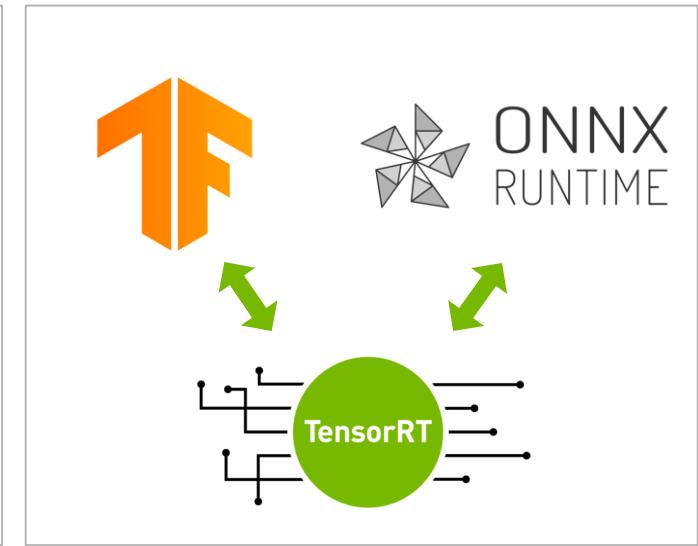
New Models Supported • INT8 for DLA • TF 2.0 & ONNX Runtime



20+ New Ops for Image Applications,
Deploy Weight Updates Faster



Deploy INT8 Precision on DLA
Accelerator

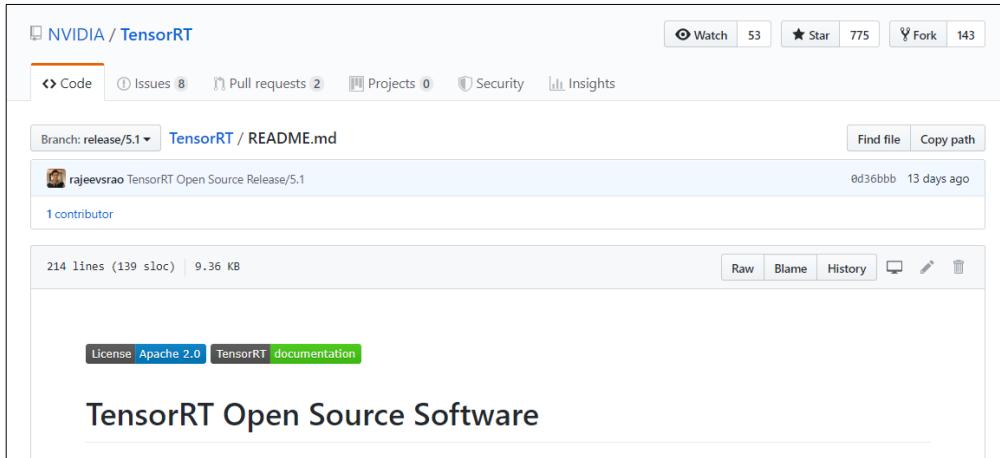


Integrated with Tensorflow 2.0 &
ONNX Runtime

TensorRT 5.1 Available as free download to members of NVIDIA Developer Program
developer.nvidia.com/tensorrt

TensorRT OPEN SOURCE SOFTWARE

TensorRT Plugins, Parsers (Caffe and ONNX) and Samples



TensorRT Open Source Software

This repository contains the Open Source Software (OSS) components of NVIDIA TensorRT. Included are the TensorRT plugins and parsers (Caffe and ONNX), as well as sample applications demonstrating usage and capabilities of the TensorRT platform.

Prerequisites

To build the TensorRT OSS components, ensure you meet the following package requirements:

System Packages

- CUDA
 - Recommended versions:
 - cuda-10.1 + cuDNN-7.5
 - cuda-10.0 + cuDNN-7.5
 - cuda-9.0 + cuDNN 7.3

Sources for TensorRT plugins and parsers (Caffe and ONNX), as well as sample applications demonstrating usage and capabilities of the TensorRT platform

<https://github.com/NVIDIA/TensorRT>

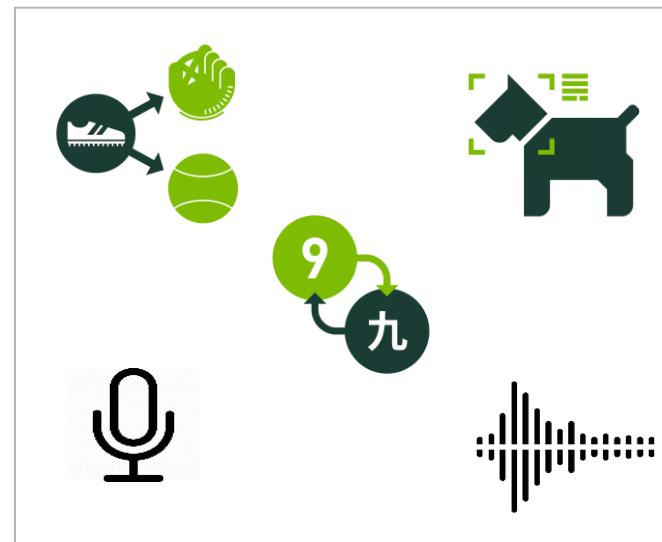
TensorRT 5 & TensorRT inference server

Turing Support • Optimizations & APIs • Inference Server



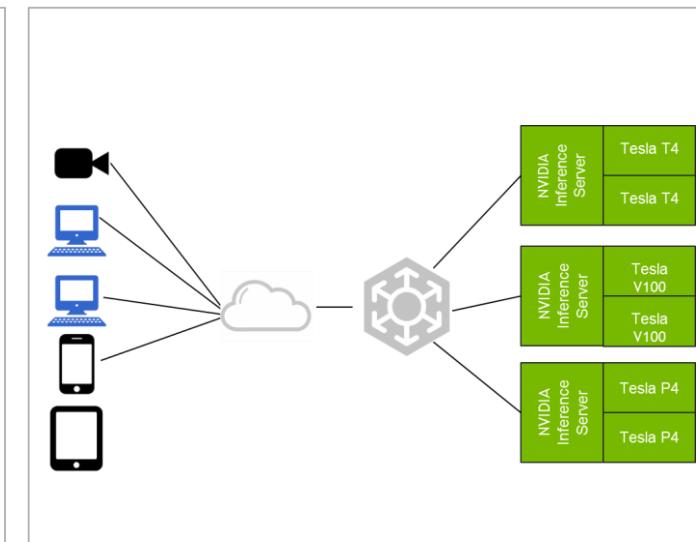
World's Most Advanced Inference Accelerator

Up to 40x faster perf. on Turing Tensor Cores



New optimizations & flexible INT8 APIs

New INT8 workflows, Win & CentOS support



TensorRT inference server

Maximize GPU utilization, run multiple models on a node

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



TensorRT 5 Supports Turing GPUs

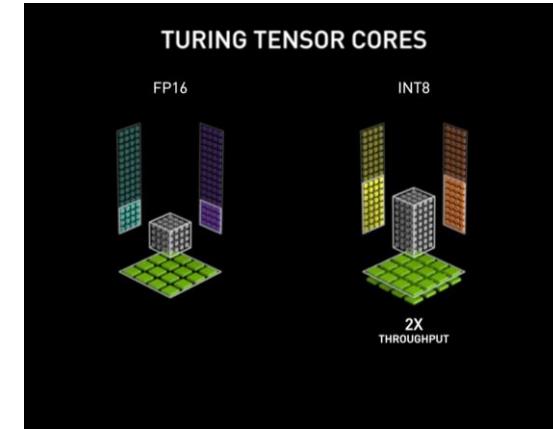
Fastest Inference Using Mixed Precision (FP32, FP16, INT8) and Turing Tensor Cores

Speed up recommender, speech, video and translation in production

Optimized kernels for mixed precision (FP32, FP16, INT8) workloads on Turing GPUs

Up to 40x faster inference for apps vs CPU-only platforms

MPS maximizes utilization with multiple separate inference processes

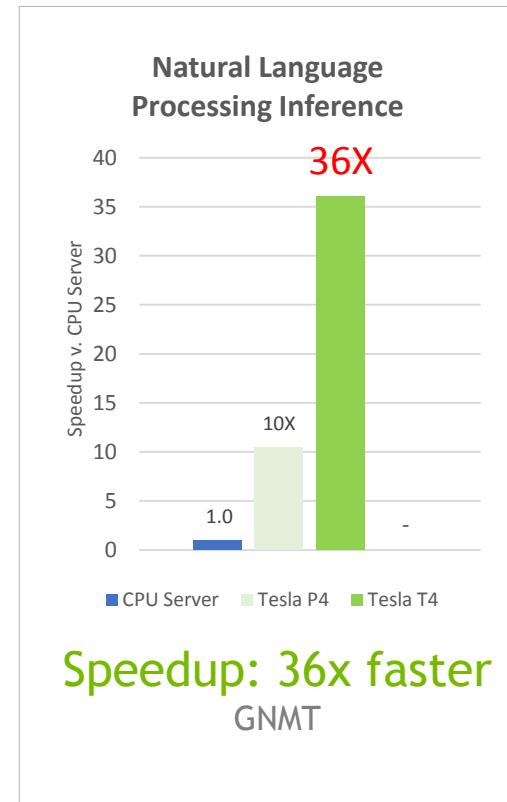
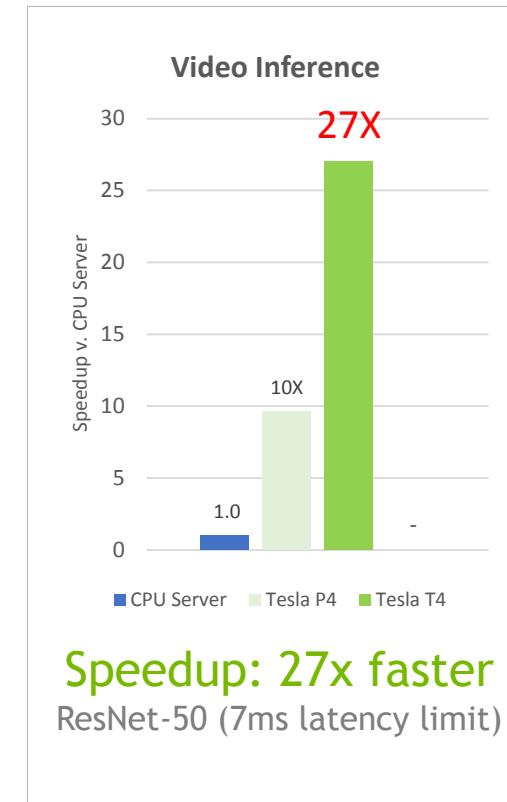
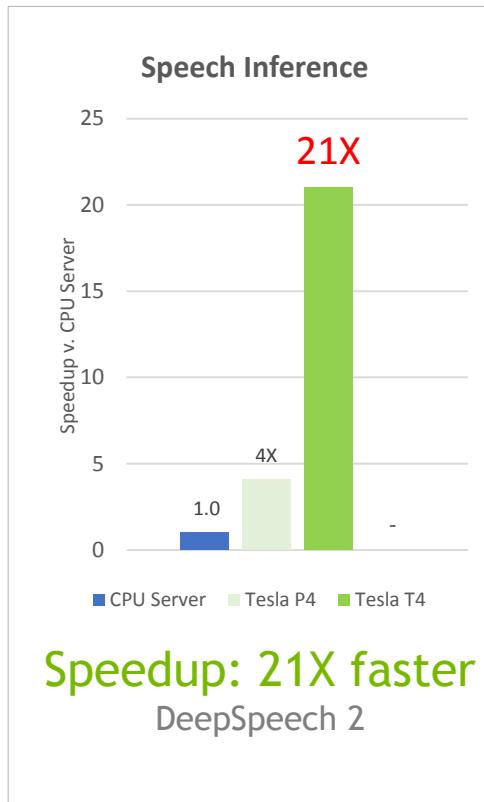
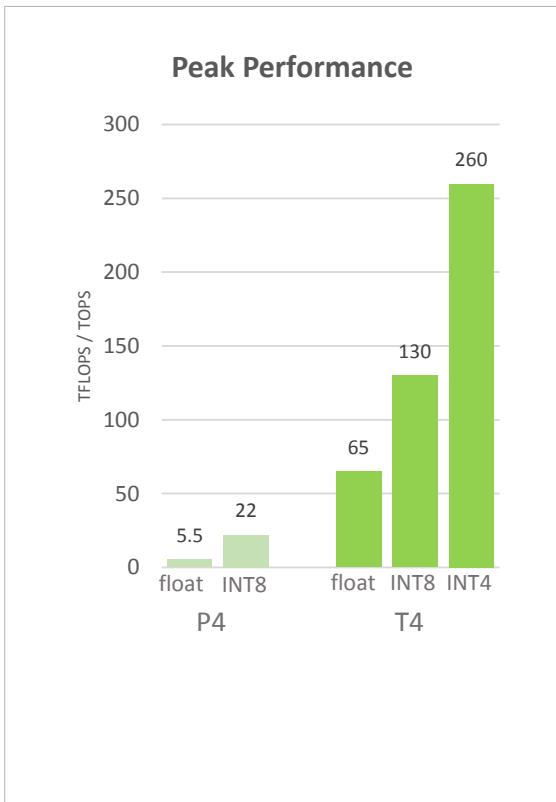




BENCHMARK

TensorRT INFERENCE PERFORMANCE

CPU vs. P4 vs. T4



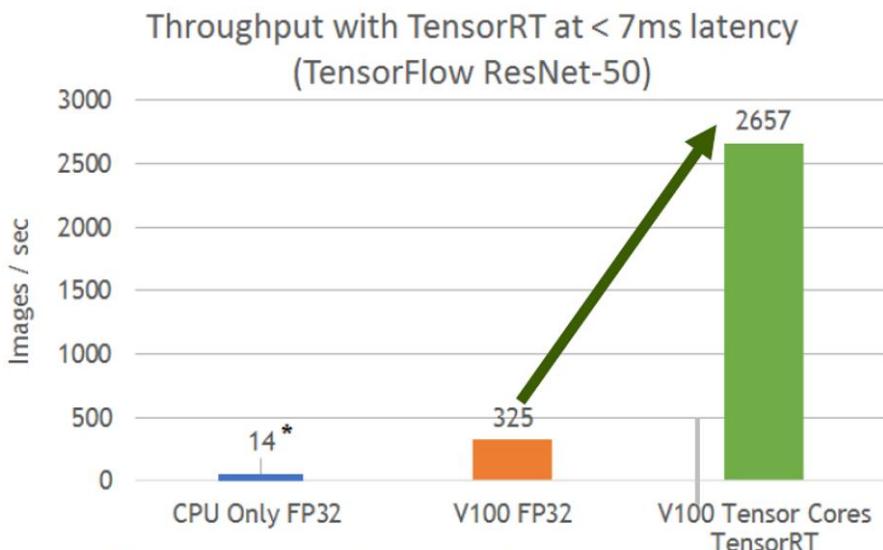
For all three graphs:

Dual-Socket Xeon Gold 6140 @ 3.6GHz with single GPU as shown 18.11-py3 | TensorRT 5.0 | CPU FP32, P4 & T4: INT8 | Batch Size = 128

TensorRT INFERENCE PERFORMANCE

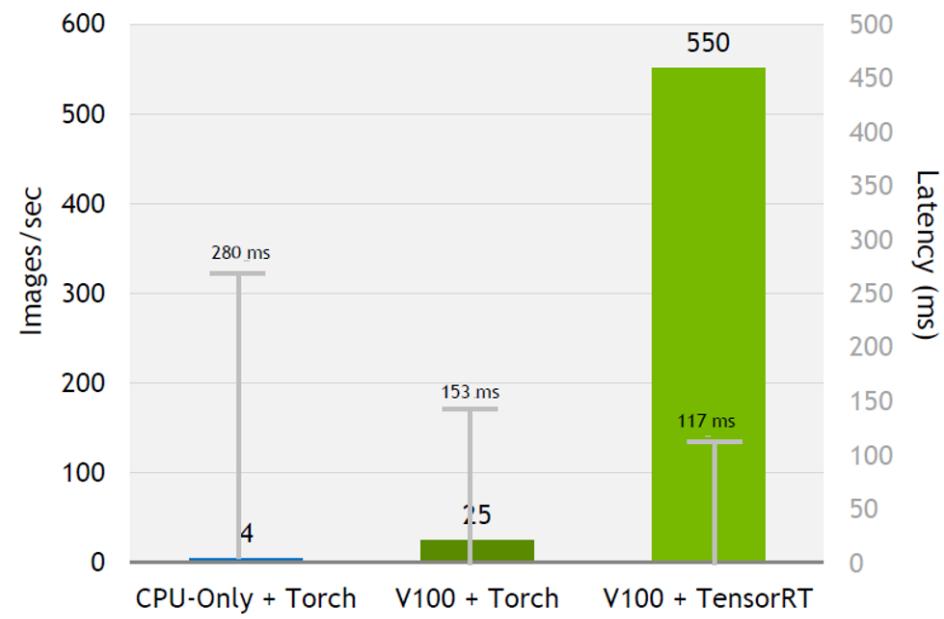
CPU vs. V100

40x Faster CNNs on V100 vs. CPU-Only
Under 7ms Latency (ResNet50)



* Min CPU latency measured was 70 ms. It is not < 7 ms.
CPU: Skylake Gold 6140, 2.5GHz, Ubuntu 16.04; 18 CPU threads. Volta V100 SXM; CUDA (384.111; v9.0.176);
Batch sizes: CPU=1, V100_FP32=2, V100_TensorFlow_TensorRT=16 w/ latency=6ms

140x Faster Language Translation RNNs on
V100 vs. CPU-Only Inference (OpenNMT)



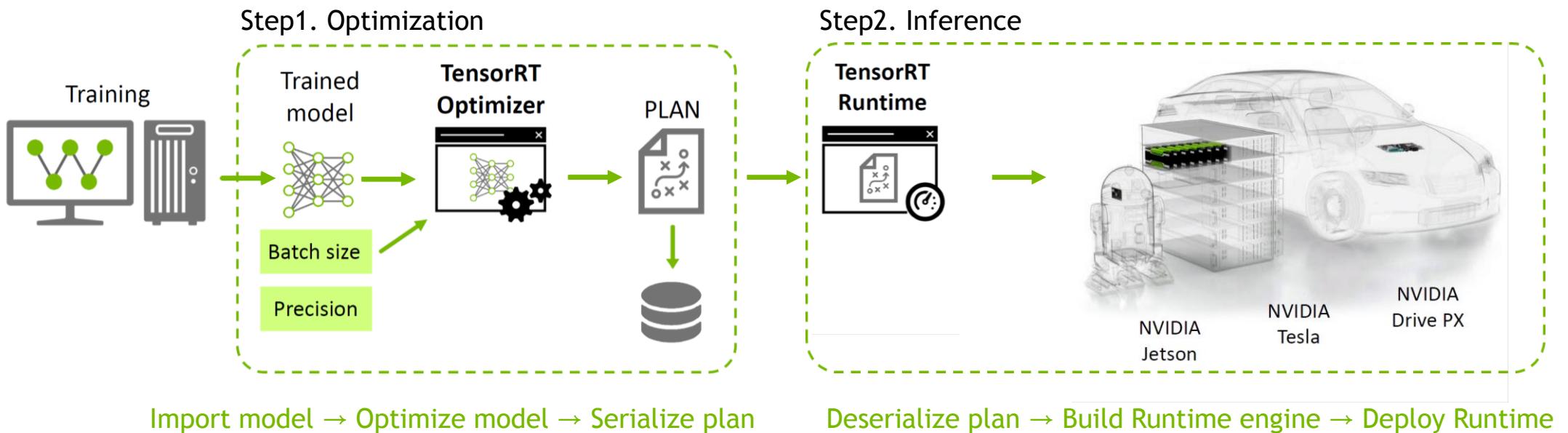
Inference throughput (sentences/sec) on OpenNMT 692M. V100 + TensorRT: NVIDIA TensorRT (FP32), batch size 64, Tesla V100-PCIE-16GB, E5-2690 v4@2.80GHz 3.5GHz Turbo (Broadwell) HT On. V100 + Torch: Torch (FP32), batch size 4, Tesla V100-PCIE-16GB, E5-2690 v4@2.80GHz 3.5GHz Turbo (Broadwell) HT On. CPU-Only: Torch (FP32), batch size 1, Intel E5-2690 v4@2.80GHz 3.5GHz Turbo (Broadwell) HT On



TENSORRT WORKFLOW

TensorRT WORKFLOW

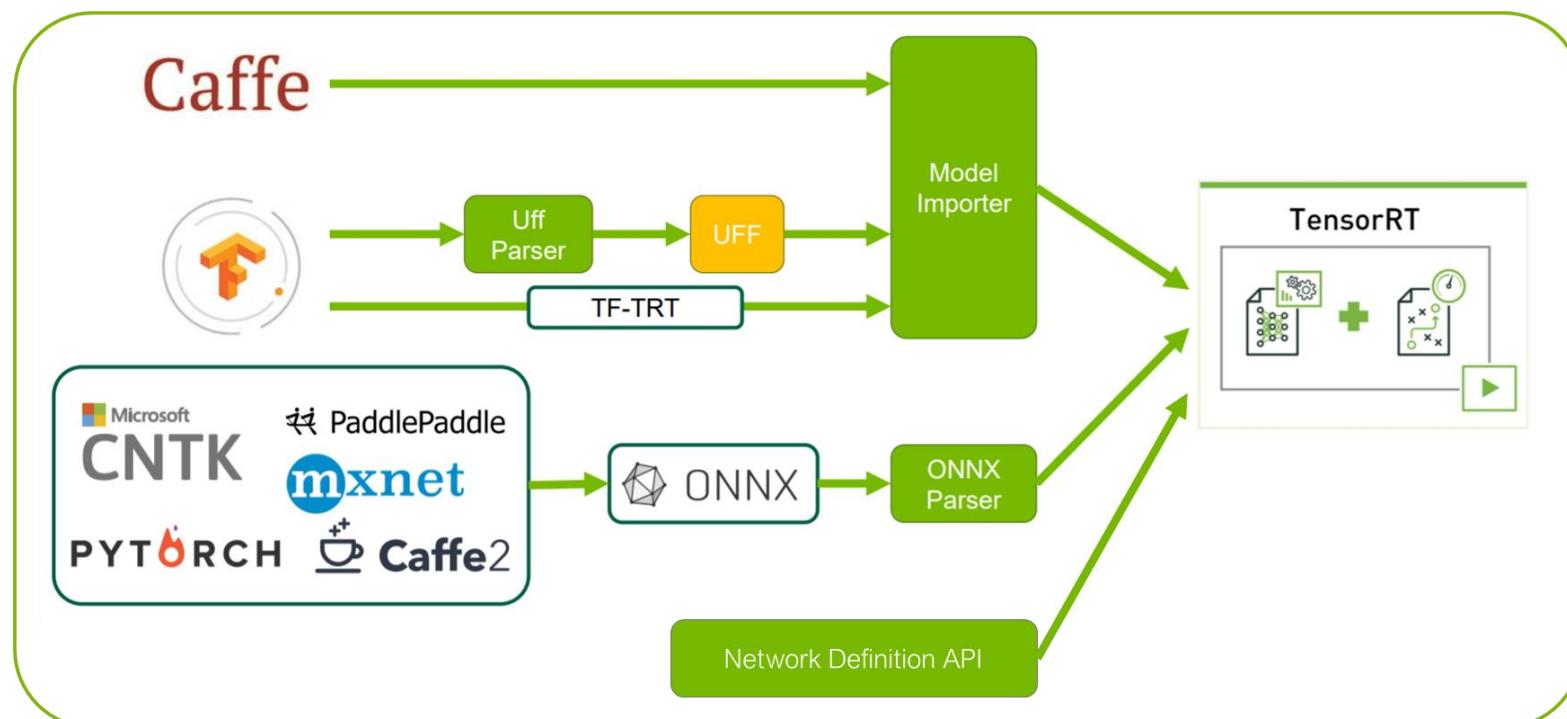
How to accelerate DNN inference with TensorRT



TensorRT WORKFLOW

Step1. Optimize model on TensorRT

1. Import model



TensorRT WORKFLOW

Step1. Optimize model on TensorRT

1. Import model (Caffe)

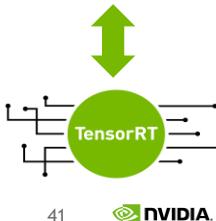
```
Import tensorrt as trt  
  
#Reports errors, warning, and messages  
TRT_LOGGER = trt.Logger(trt.Logger.WARNING)  
  
#create the build and network  
builder = trt.Builder(TRT_LOGGER)  
  
network = builder.create_network()  
  
#Create parser and parse the network  
parser = trt.CaffeParser()  
  
model_tensors = parser.parse(deploy = DEPLOY_FILE,  
                           model = MODEL_FILE,  
                           network = network,  
                           dtype = trt.float32)
```

Reports errors, warnings, and messages

Create the builder/network and parser

Parse the network

Caffe



TensorRT WORKFLOW

Step1. Optimize model on TensorRT

1. Import model (ONNX)

```
Import tensorrt as trt  
  
#Reports errors, warning, and messages  
TRT_LOGGER = trt.Logger(trt.Logger.WARNING)  
  
#create the build and network  
builder = trt.Builder(TRT_LOGGER)  
  
network = builder.create_network()  
  
#Create parser and parse the network  
parser = trt.OnnxParser(network)  
  
model_file = open(model_path, 'rb')  
  
model_tensors = parser.parse(model_file)
```

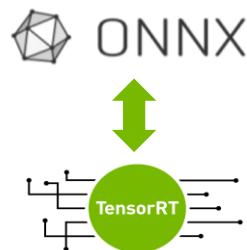
Export ONNX

```
torch.onnx.export(model,\n                  dummy_input,\n                  model_path)
```

Reports errors, warnings, and messages

Create the builder/network and parser

Parse the network



TensorRT WORKFLOW

Step1. Optimize model on TensorRT

1. Import model (TensorFlow)

- Save UFF model from TensorFlow

```
python freeze_graph.py \
    --input_graph=GRAPH \
    --input_checkpoint=CHECKPOINTS \
    --output_graph=FROZEN_MODEL \
    --output_node_names=OUTPUT_LAYERS
```

```
frozen_model = \
    tf.graph_util.convert_variables_to_constants( \
        sess, tf.get_default_graph().as_graph_def(), \
        output_node_names = OUTPUT_LAYERS )
```

```
uff_model = \
    uff.from_tensorflow_frozen_model( \
        frozen_model, \
        OUTPUT_LAYERS )
```

GraphDef(.pb) Checkpoint(.ckpt)

Freeze

Frozen model

UFF

TensorRT WORKFLOW

Step1. Optimize model on TensorRT

1. Import model (TensorFlow)

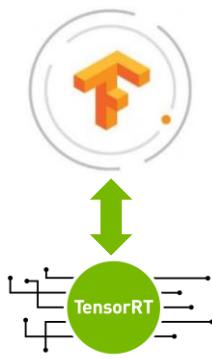
- Importing UFF model

```
Import tensorrt as trt  
  
#Reports errors, warning, and messages  
TRT_LOGGER = trt.Logger(trt.Logger.WARNING)  
  
#create the build and network  
builder = trt.Builder(TRT_LOGGER)  
  
network = builder.create_network()  
  
#Create parser and parse the network  
parser = trt.UffParser()  
parser.register_input(INPUT_LAYER_NAME, (1,28,28))  
parser.register_output(OUTPUT_LAYER_NAME)  
  
Parser.parse(uff_model, network)
```

Reports errors, warnings, and messages

Create the builder/network and parser

Parse the network



TensorRT WORKFLOW

Step1. Optimize model on TensorRT

2. Build an engine

```
builder.max_batch_size = max_batch_size  
builder.max_workspace_size = 1 << 20  
...  
engine = builder.build_cuda_engine(network)
```

- Set batch size
- Set Memory size to be used
- Build an engine

3. Optimization configuration

```
builder.fp16_mode(or int8_mode) = True  
batchstream = ImageBatchStream( \\\n    NUM_IMAGES_PER_BATCH, \\ \n    calibration_files )  
Int8_calibrator = EntropyCalibrator( \\\n    ["input_node_name"], \\ \n    batchstream )  
builder.int8_calibrator = Int8_calibrator
```

- Set Bit precision
- Set dataset stream for calibration
- Create Calibrator
- Set Calibrator

int8_mode

TensorRT WORKFLOW

Step1. Optimize model on TensorRT

4. (Optional) Add custom operations into TensorRT

- Unsupported layer “ReLU6”

```
def build_model():
    # Create the keras model
    model = tf.keras.models.Sequential()
    model.add(tf.keras.layers.InputLayer(input_shape=[1, 28, 28], name="InputLayer"))
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(512))                                Custom layer
    model.add(tf.keras.layers.Activation(activation=tf.nn.relu6, name="ReLU6"))
    model.add(tf.keras.layers.Dense(10, activation=tf.nn.softmax, name="OutputLayer"))
    return model
```

- `tf.nn.relu6`
 - ✓ Computes Rectified Linear 6: $\min(\max(\text{features}, 0), 6)$

TensorRT WORKFLOW

Step1. Optimize model on TensorRT

4. (Optional) Add custom operations into TensorRT

```
> Unsupported layer "ReLU6"
    • Build the network
      <tf> builder = trt.Builder()
      <tf> builder.addLayer("ReLU6")
      <tf> builder.addInput("input", input_shape=[1, 28, 28])
      <tf> builder.addOutput("output", output_shape=[1, 28, 28])
      <tf> builder.configure()
      <tf> builder.getNbOutputs()
      <tf> builder.getOutputDimensions()
      <tf> builder.configure()
      <tf> builder.getNbOutputs()
      <tf> builder.getOutputDimensions()
      <tf> builder.getWorkspaceSize()
      <tf> builder.serialize()
      <tf> builder.getSerializationSize()

      <tf> model = builder.build()
      <tf> model.add(tf.keras.layers.ReLU())
      <tf> model.add(tf.keras.layers.Dense(10))
      <tf> model.add(tf.keras.layers.Dense(1))
      <tf> return model
```

The code snippet shows the process of building a TensorRT engine from a TensorFlow model. It includes the following steps:

- Unsupported layer "ReLU6": A note indicating that the ReLU6 layer is not supported by TensorRT.
- Build the network: The code uses a TensorRT builder to build the network. It adds a ReLU6 layer, sets the input shape to [1, 28, 28], and adds an output layer with the same shape. It then configures the builder, gets the number of outputs, gets the output dimensions, and gets the workspace size. Finally, it serializes the builder to create a TensorRT model.
- Do inference: The code adds a Dense layer with 10 units and another Dense layer with 1 unit to the model. It then returns the model.

The code is divided into two phases:

- Build Phase**: The first part of the code, which includes the unsupported layer and the configuration of the builder.
- Execution Phase**: The second part of the code, which adds the inference layers to the model.

- Use C++ API for custom layers, due to easily accessing libraries like [CUDA and cuDNN](#)
- package the layer using pybind11 in Python, and [load the plugin into a Python application](#)

TensorRT WORKFLOW

Step1. Optimize model on TensorRT

4. (Optional) Add custom operations into TensorRT

- Custom operation with CUDA

```
int ClipPlugin::enqueue(const void* const* inputs, void** outputs, cudaStream_t stream);
{
    int status = -1;

    // Our plugin outputs only one tensor
    void *output = outputs[0];

    // Launch CUDA kernel wrapper and save its return value
    status = clipInference(stream, mInputVolume * batchSize, mClipMax, inputs[0], output);

    return status;
}
```

TensorRT WORKFLOW

Step1. Optimize model on TensorRT

4. (Optional) Add custom operations into TensorRT

- Custom operation with CUDA

```
int clipInference(
    cudaStream_t stream,
    int n,
    float clipMax,
    const void* input,
    void* output)
{
    const int blocksize = 512;
    const int gridSize = (n + blocksize - 1)/blockSize;
    clipKernel<float, blocksize><<<gridSize, blockSize, 0, stream>>>(
        n, clipMax,
        static_cast<const float*>(input),
        static_cast<float*>(output));
    return 0;
}
```

```
template <typename T, unsigned nthdsPerCTA>
__launch_bounds__(nthdsPerCTA)
__global__ void clipKernel(
    int n,
    const T clipMax,
    const T* input,
    T* output)
{
    for (int i = blockIdx.x * nthdsPerCTA + threadIdx.x; i < n; i += blockDim.x * nthdsPerCTA)
    {
        output[i] = min<T>(max<T>(input[i], 0), clipMax);
    }
}
```

```
template <typename T>
__device__ __forceinline__ const T& min(const T& a, const T& b)
{
    return (a > b) ? b : a;
}

template <typename T>
__device__ __forceinline__ const T& max(const T& a, const T& b)
{
    return (a > b) ? a : b;
}
```

TensorRT WORKFLOW

Step1. Optimize model on TensorRT

4. (Optional) Add custom operations into TensorRT

- Custom operation with CuDNN

```
int ClipPlugin::initialize()
{
    // Initialize cudnn
    cudnnCreate(&mCudnn);

    // Create Tensor Descriptors for ClipReLU
    checkCUDNN(cudnnCreateTensorDescriptor(&mInput_descriptor));
    checkCUDNN(cudnnSetTensor4dDescriptor(mInput_descriptor,
                                         /*format=*/
                                         CUDNN_TENSOR_NHWC, /*dataType=*/
                                         CUDNN_DATA_FLOAT,
                                         /*batch_size=*/
                                         batch_size, /*channels=*/
                                         channel,
                                         /*image_height=*/
                                         height, /*image_width=*/
                                         width));
    checkCUDNN(cudnnCreateTensorDescriptor(&mOutput_descriptor));
    checkCUDNN(cudnnSetTensor4dDescriptor(mOutput_descriptor,
                                         /*format=*/
                                         CUDNN_TENSOR_NHWC, /*dataType=*/
                                         CUDNN_DATA_FLOAT,
                                         /*batch_size=*/
                                         batch_size, /*channels=*/
                                         channel,
                                         /*image_height=*/
                                         height, /*image_width=*/
                                         width));

    // Create Activation Descriptors for ClipReLU
    checkCUDNN(cudnnCreateActivationDescriptor(&mActivation_descriptor));
    checkCUDNN(cudnnSetActivationDescriptor(mActivation_descriptor,
                                         /*mode=*/
                                         CUDNN_ACTIVATION_CLIPPED_RELU, /*reluNanOpt=*/
                                         CUDNN_PROPAGATE_NAN,
                                         /*relu_coef=*/
                                         mClipMax));
}
```

TensorRT WORKFLOW

Step1. Optimize model on TensorRT

4. (Optional) Add custom operations into TensorRT

- Custom operation with CuDNN

```
int ClipPlugin::enqueue(const void* const* inputs, void** outputs, cudaStream_t stream)
{
    cudnnSetStream(mCudnn, stream);

    const float alpha = 1;
    const float beta = 0;

    checkCUDNN(cudnnActivationForward(mCudnn,
                                       mActivation_descriptor,
                                       &alpha,
                                       mInput_descriptor,
                                       inputs[0],
                                       &beta,
                                       mOutput_descriptor,
                                       outputs[0]));

};

    return 0;
}
```

TensorRT WORKFLOW

Step1. Optimize model on TensorRT

4. (Optional) Add custom operations into TensorRT

- API for Serialization (common)

```
void ClipPlugin::serialize(void* buffer) const
{
    char *d = static_cast<char*>(buffer);
    const char *a = d;

    writeToBuffer(d, mClipMax);

    assert(d == a + getSerializationSize());
}

size_t ClipPlugin::getSerializationSize() const
{
    return sizeof(float);
}
```

TensorRT WORKFLOW

Step1. Optimize model on TensorRT

4. (Optional) Add custom operations into TensorRT

- API for Builder (common)

```
int ClipPlugin::getNbOutputs() const
{
    return 1;
}

Dims ClipPlugin::getOutputDimensions(int index, const Dims* inputs, int nbInputDims)
{
    // Validate input arguments
    assert(nbInputDims == 1);
    assert(index == 0);

    // Clipping doesn't change input dimension, so output Dims will be the same as input Dims
    return *inputs;
}
```

TensorRT WORKFLOW

Step1. Optimize model on TensorRT

4. (Optional) Add custom operations into TensorRT

- API for Plugin Creator (common)

```
IPluginV2* ClipPluginCreator::createPlugin(const char* name, const PluginFieldCollection* fc)
{
    float clipMax;
    const PluginField* fields = fc->fields;

    // Parse fields from PluginFieldCollection
    assert(fc->nbFields == 1);

    for (int i = 0; i < fc->nbFields; i++){
        if (strcmp(fields[i].name, "clipMax") == 0 ){
            assert(fields[i].type == PluginFieldType::kFLOAT32);
            clipMax = *(static_cast<const float*>(fields[i].data));
        }
    }
    return new ClipPlugin(name, clipMax);
}

IPluginV2* ClipPluginCreator::deserializePlugin(const char* name, const void* serialData, size_t serialLength)
{
    // This object will be deleted when the network is destroyed, which will
    // call ClipPlugin::destroy()
    return new ClipPlugin(name, serialData, serialLength);
}
```

TensorRT WORKFLOW

Step1. Optimize model on TensorRT

4. (Optional) Add custom operations into TensorRT

- Graph surgeon with created plugin

```
import graphsurgeon as gs
trt_relu6 = gs.create_plugin_node(name="trt_relu6", op="ClipPlugin",
clipMax=6.0)
namespace_plugin_map = {"ReLU6": trt_relu6}

dynamic_graph = gs.DynamicGraph(model_path)
dynamic_graph.collapse_namespaces(namespace_plugin_map)
# Save resulting graph to UFF file
output_uff_path = model_path_to_uff_path(model_path)
uff.from_tensorflow(
    dynamic_graph.as_graph_def(),
    ["OutputLayer/Softmax"],
    output_filename=output_uff_path,
    text=True
)
```

TensorRT WORKFLOW

Step1. Optimize model on TensorRT

5. Serialize network model

```
Serialized_engine = engine.serialize()  
  
with open(engine_path, 'wb') as f:  
    f.write(serialized_engine)
```

TensorRT WORKFLOW

Step2. Inference model on TensorRT

1. Create Runtime and Deserialize network model

```
runtime = trt.runtime(TRT_LOGGER)

with open(engine_path, 'rb') as f:
    engine = runtime.deserialize_cuda_engine(f.read())
```

2. Initialize performing inference

```
import pycuda.driver as cuda

# Determine dimensions and create page-locked memory buffers
# (i.e. won't be swapped to disk) to hold host inputs/outputs.
h_input = cuda.pagelocked_empty(engine.get_binding_shape(0).volume(), dtype=np.float32)
h_output = cuda.pagelocked_empty(engine.get_binding_shape(1).volume(), dtype=np.float32)

# Allocate device memory for inputs and outputs.
d_input = cuda.mem_alloc(h_input.nbytes)
d_output = cuda.mem_alloc(h_output.nbytes)
# Create a stream in which to copy inputs/outputs and run inference.
stream = cuda.Stream()
```

TensorRT WORKFLOW

Step2. Inference model on TensorRT

3. Run Inference

```
with engine.create_execution_context() as context:  
    cuda.memcpy_htod_async(d_input, h_input, stream)  
  
    context.execute_async(bindings=[int(d_input), int(d_output)],  
                         stream_handle=stream.handle)  
  
    cuda.memcpy_dtoh_async(h_output, d_output, stream)  
  
    stream.synchronize()
```

- Transfer input to GPU
- Run Inference
- Transfer output from GPU
- Synchronize the stream





Yes!!!

Now, I can inference
my custom model with
NVIDIA TensorRT.



LEARN MORE ABOUT
TENSORRT

LEARN MORE ABOUT TENSORRT

Helpful Links & Examples on NGC

- TensorRT Developer Guide
 - : <https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html>
- Best Practice for TensorRT Performance
 - : <https://docs.nvidia.com/deeplearning/sdk/tensorrt-best-practices/index.html>
- TF-TRT User Guide
 - : <https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html>
- TensorRT backend for ONNX
 - : <https://github.com/onnx/onnx-tensorrt>
- NVIDIA GPU CLOUD(NGC) example
 - : /usr/src/tensorrt/samples/python

Sample name	Description
Int8_caffe_mnist	Performs INT8 calibration and inference. Calibrates a network for execution in INT8.
uff_custom_plugin	Implements a clip layer (as a CUDA kernel), wraps the implementation in a TensorRT plugin
yolov3_onnx	Implements a full ONNX-based pipeline for performing inference with the YOLOv3-608 network

RELATED SESSIONS IN AI CONFERENCE

Learn more about TensorRT and Its usecases

- **TensorRT usecase**
 - : **Kakao OCR inference** 성능 최적화
Track2, Today 2:40pm - 3:20pm
이현수 (Kakao)
- **Best Practice for TensorRT Performance**
 - : **IVA를 위한 NVIDIA Deepstream SDK 및 Transfer Learning Toolkit 소개**
Track3, Today 2:40pm - 3:20pm
홍광수 과장 (NVIDIA)
 - : **Adding a custom CUDA C++ Operations in Tensorflow for boosting BERT Inference**
Track1, Today 2:40pm - 3:20pm
이민석 과장 (NVIDIA)
- **TensorRT inference server**
 - : **효율적인 Deep Learning 서비스 구축을 위한 핵심 애플리케이션**
Track2, Today 5:20pm - 6:00pm
정소영 상무 (NVIDIA)

Q & A



NVIDIA®

