NVIDIA | DEEP LEARNING INSTITUTE

PRODUCTION DEPLOYMENT

Building Transformer-Based Natural Language Processing Applications
(Part 3)

FULL COURSE AGENDA

Part 1: Machine Learning in NLP

Lecture: NLP background and the role of DNNs leading to the Transformer architecture

Lab: Tutorial-style exploration of a ***translation task*** using the Transformer architecture

---

Part 2: Self-Supervision, BERT, and Beyond

Lecture: Discussion of how language models with self-supervision have moved beyond the basic Transformer to BERT and ever larger models

Lab: Practical hands-on guide to the NVIDIA NeMo API and exercises to build a ***text classification task*** and a ***named entity recognition task*** using BERT-based language models

---

Part 3: Production Deployment

Lecture: Discussion of production deployment considerations and NVIDIA Triton Inference Server

Lab: Hands-on deployment of an example ***question answering task*** to NVIDIA Triton

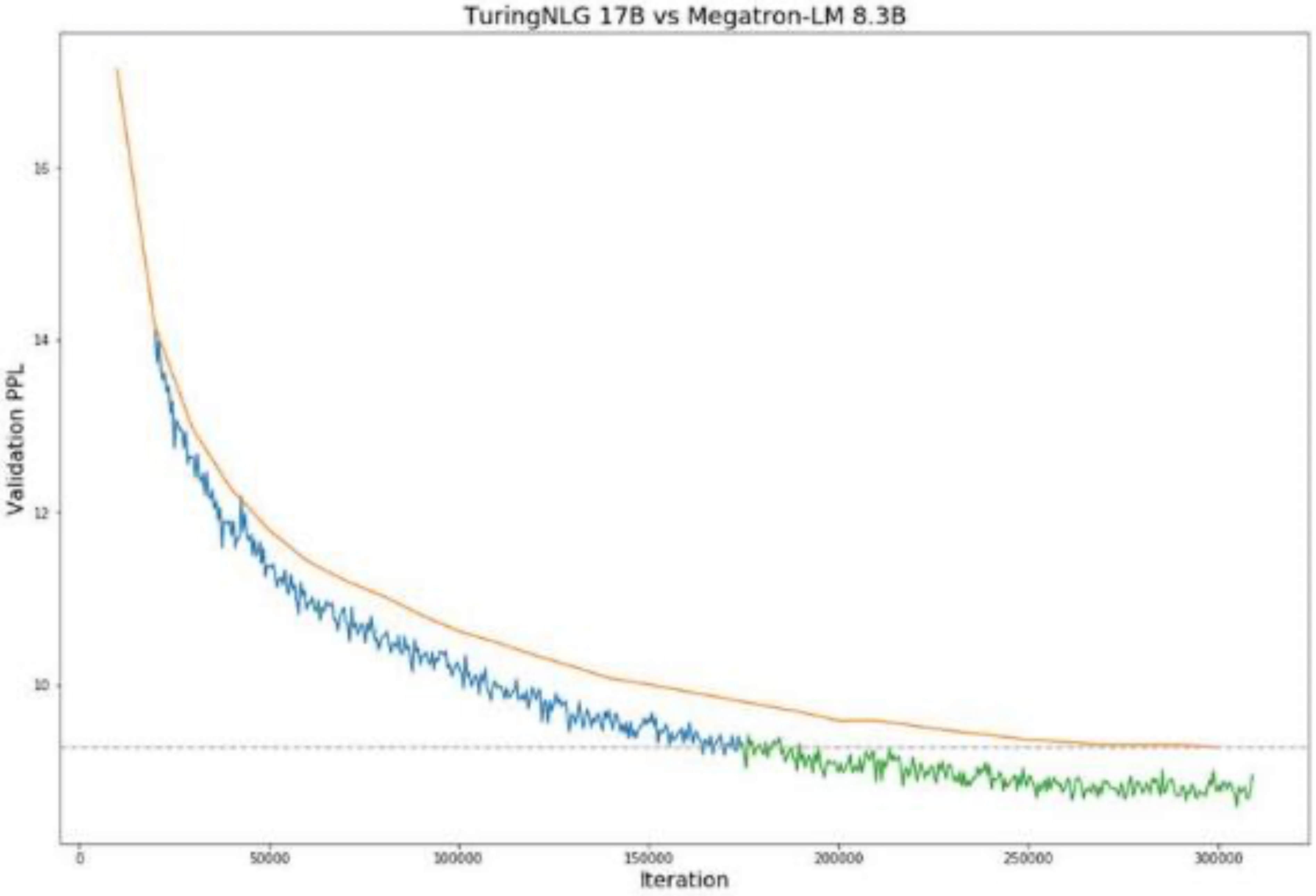
Part 3: Production Deployment

- Lecture
  - Model Selection
  - Post-Training Optimization
  - Product Quantization
  - Knowledge Distillation
  - Model Code Efficiency
  - Model Serving
  - Building the Application
- Lab
  - Exporting the Model
  - Hosting the Model
  - Server Performance
  - Using the Model

YOUR NETWORK IS TRAINED

YOUR NETWORK IS TRAINED

Now what?

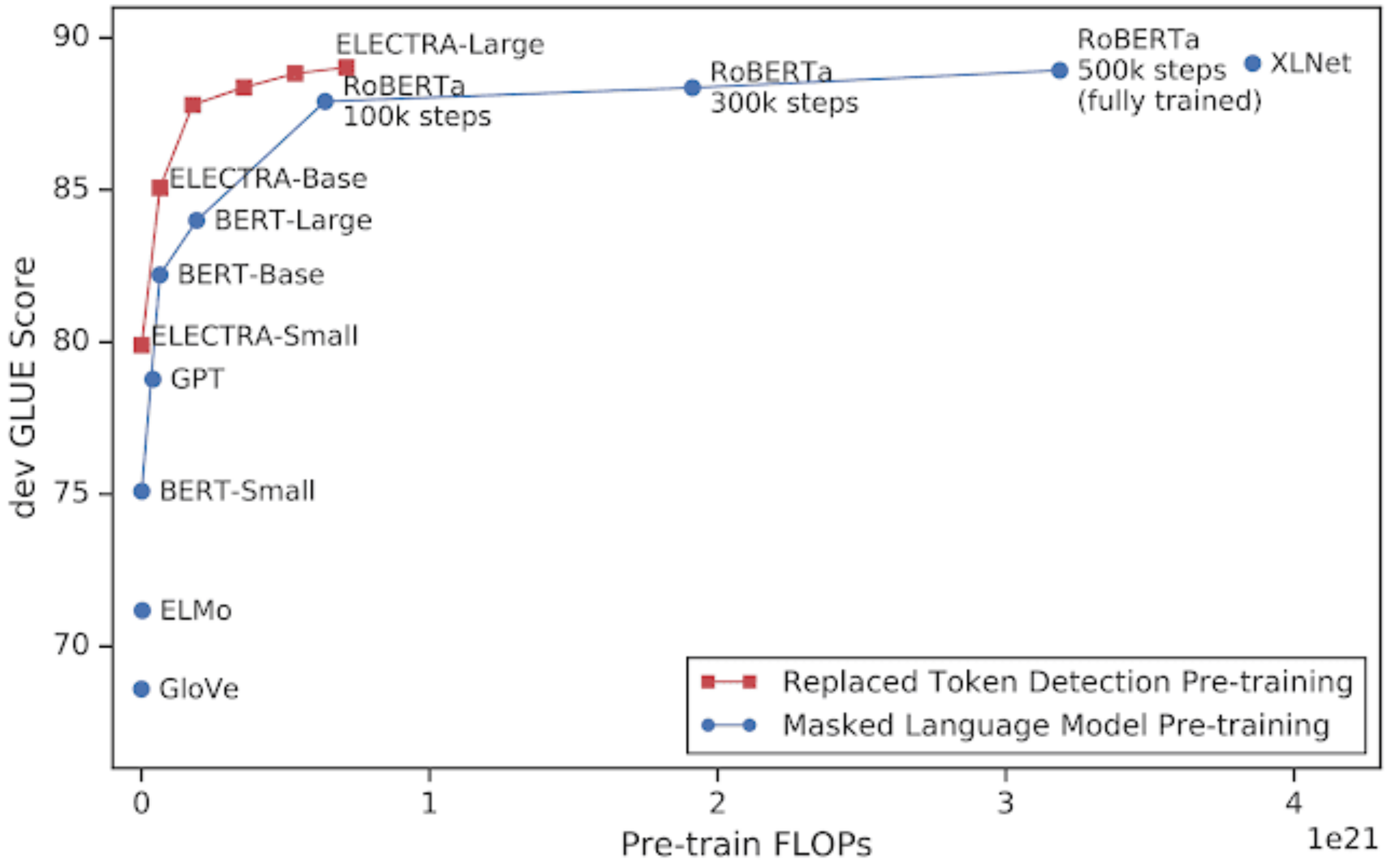


https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/

MEETING REQUIREMENTS OF YOUR BUSINESS

NLP MODELS ARE LARGE

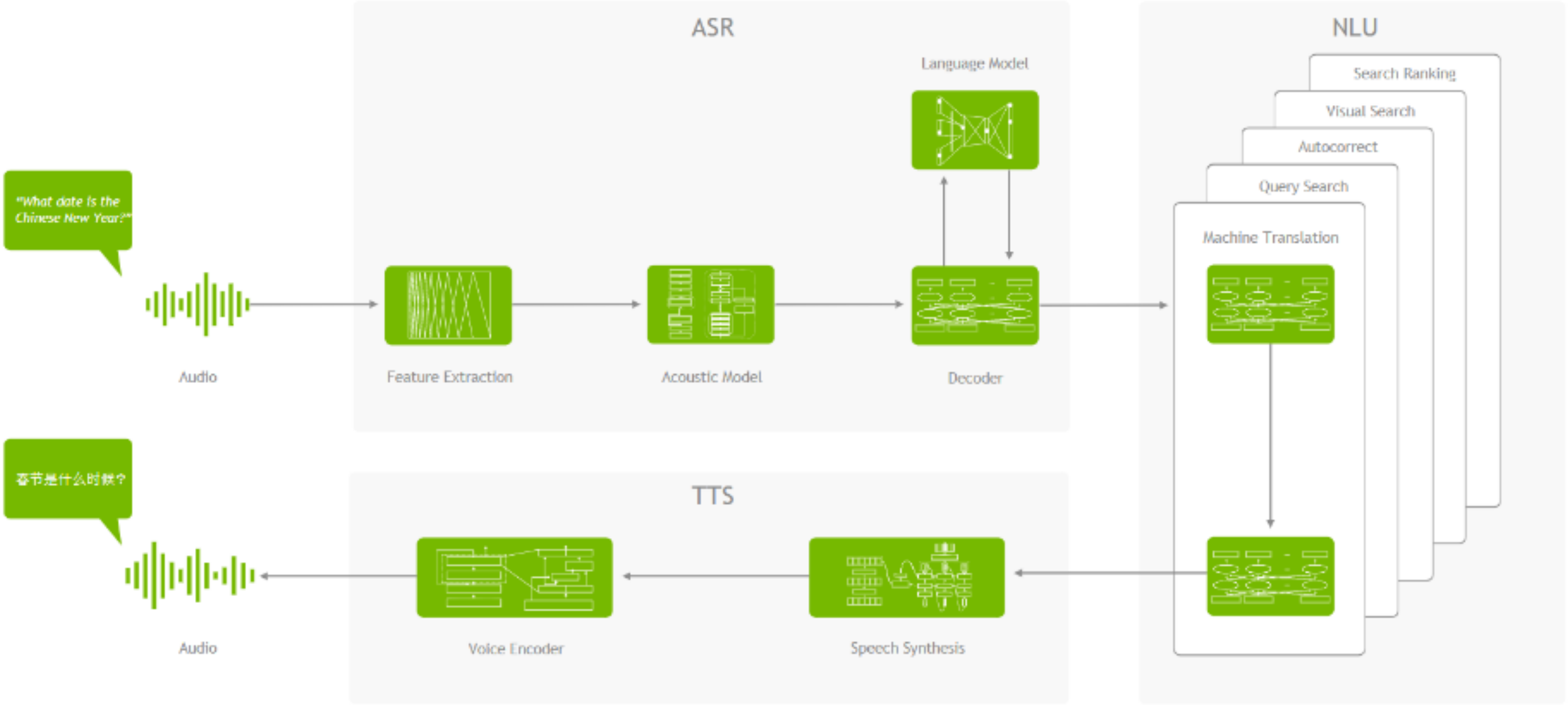
The Inference cost is high



https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/

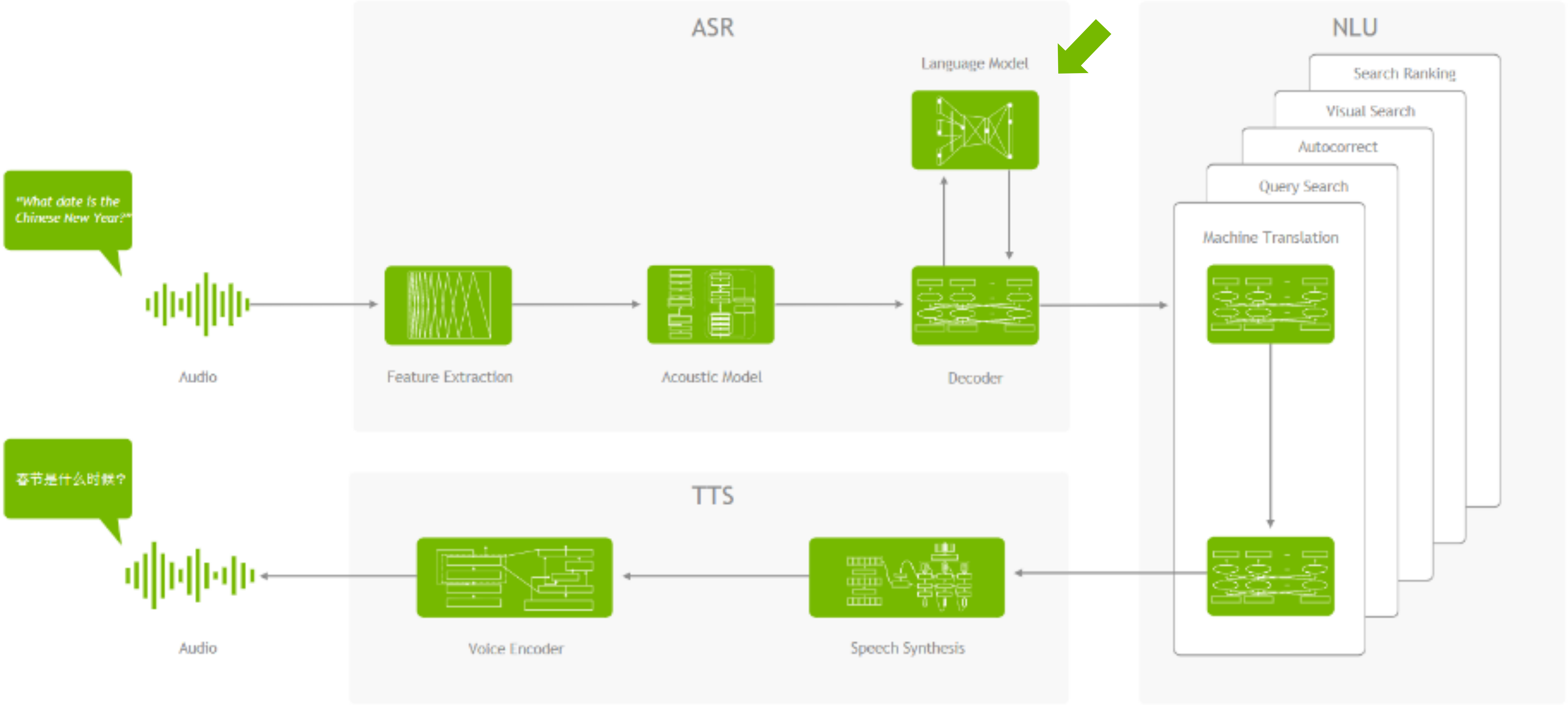
THEY DO NOT LIVE IN ISOLATION

Example of a conversational AI application



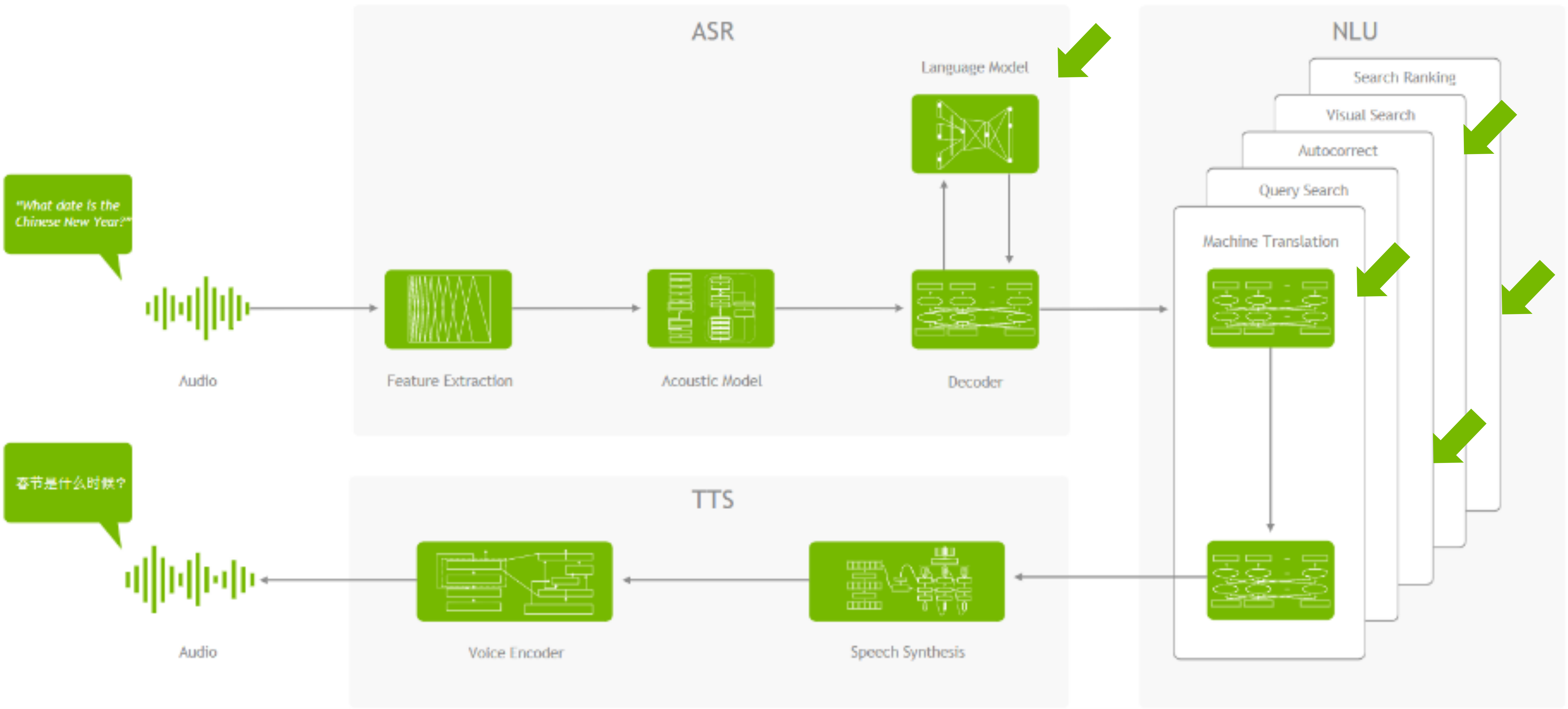
THEY DO NOT LIVE IN ISOLATION

Real Time Applications Need to Deliver Latency <300 ms



THEY DO NOT LIVE IN ISOLATION

Real Time Applications Need to Deliver Latency <300 ms



THEY DO NOT LIVE IN ISOLATION

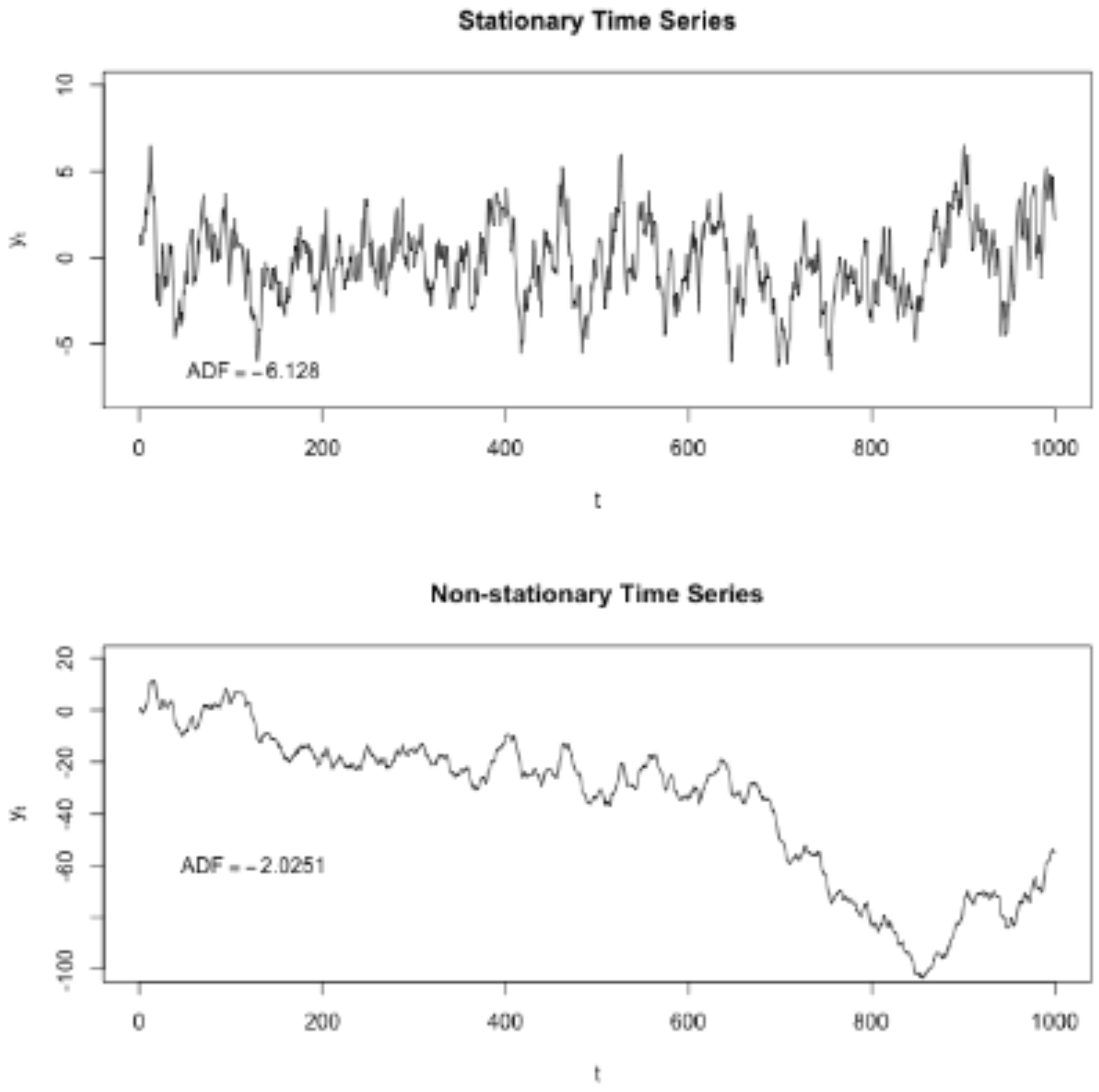
Application bandwidth = Cost

		Batch size	Inference on	Throughput (Query per second)	Latency (milliseconds)
CPU	Original 3-layer BERT	1	Azure Standard F16s_v2 (CPU)	6	157
	ONNX Model	1	Azure Standard F16s_v2 (CPU) **with ONNX Runtime**	111	9
GPU	Original 3-layer BERT	4	Azure NV6 GPU VM	200	20
	ONNX Model	4	Azure NV6 GPU VM **with ONNX Runtime**	500	8
	ONNX Model	64	Azure NC6S_v3 GPU VM **with ONNX Runtime + System Optimization** (Tensor Core with mixed precision, Same Accuracy)	10667	6

https://cloudblogs.microsoft.com/opensource/2020/01/21/microsoft-onnx-open-source-optimizations-transformer-inference-gpu-cpu/

AND THEY NEED TO EVOLVE OVER TIME

A lot of processes are not stationary



https://en.wikipedia.org/wiki/Stationary_process

THERE'S MORE TO AN APPLICATION THAN JUST THE MODEL

Nonfunctional requirements

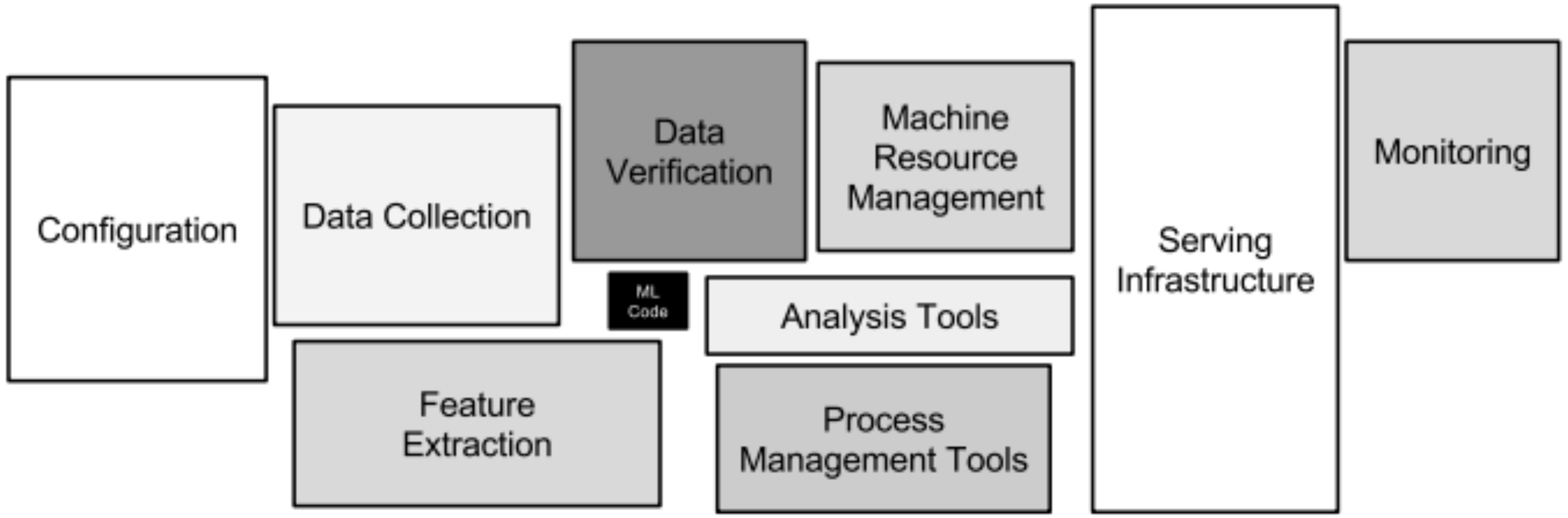


Figure 1: Only a small fraction of real-world ML systems is composed of the ML code, as shown by the small black box in the middle. The required surrounding infrastructure is vast and complex.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., ... & Dennison, D. (2015). Hidden technical debt in machine learning systems. In Advances in neural information processing systems (pp. 2503-2511).

THERE'S MORE TO AN APPLICATION THAN JUST THE MODEL

Nonfunctional requirements

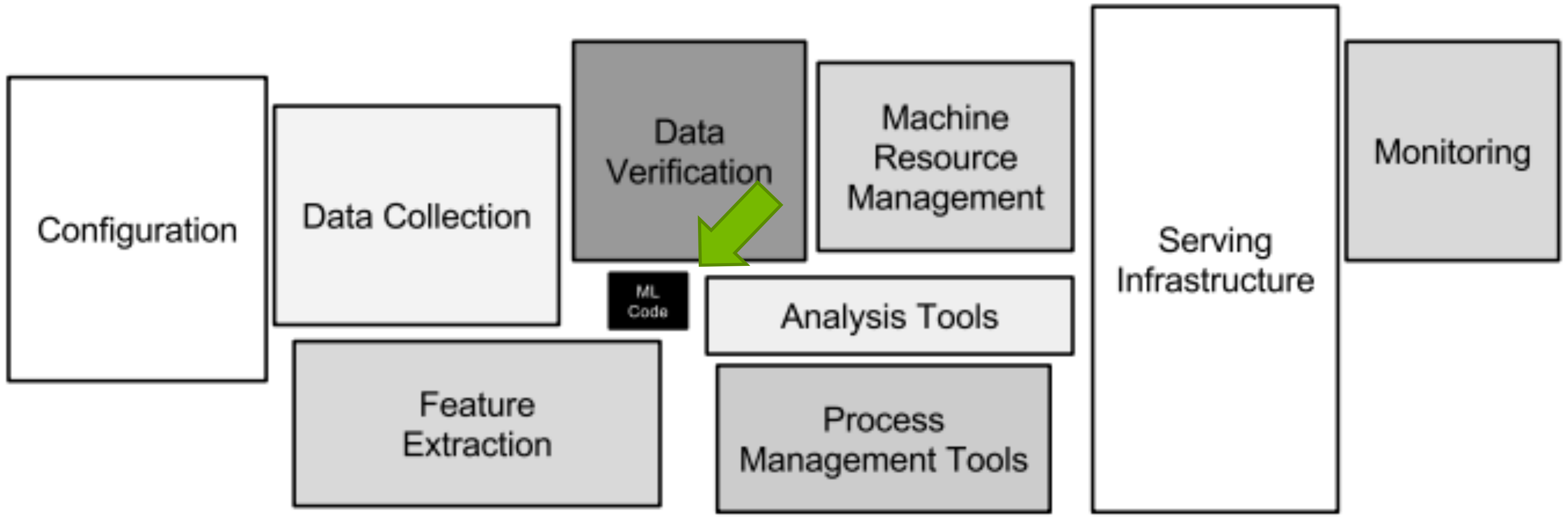


Figure 1: Only a small fraction of real-world ML systems is composed of the ML code, as shown by the small black box in the middle. The required surrounding infrastructure is vast and complex.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., ... & Dennison, D. (2015). Hidden technical debt in machine learning systems. In Advances in neural information processing systems (pp. 2503-2511).

Part 3: Production Deployment

- Lecture
  - Model Selection
  - Post-Training Optimization
  - Product Quantization
  - Knowledge Distillation
  - Model Code Efficiency
  - Model Serving
  - Building the Application
- Lab
  - Exporting the Model
  - Hosting the Model
  - Server Performance
  - Using the Model

MODEL SELECTION

Not all models are created equally

NLP

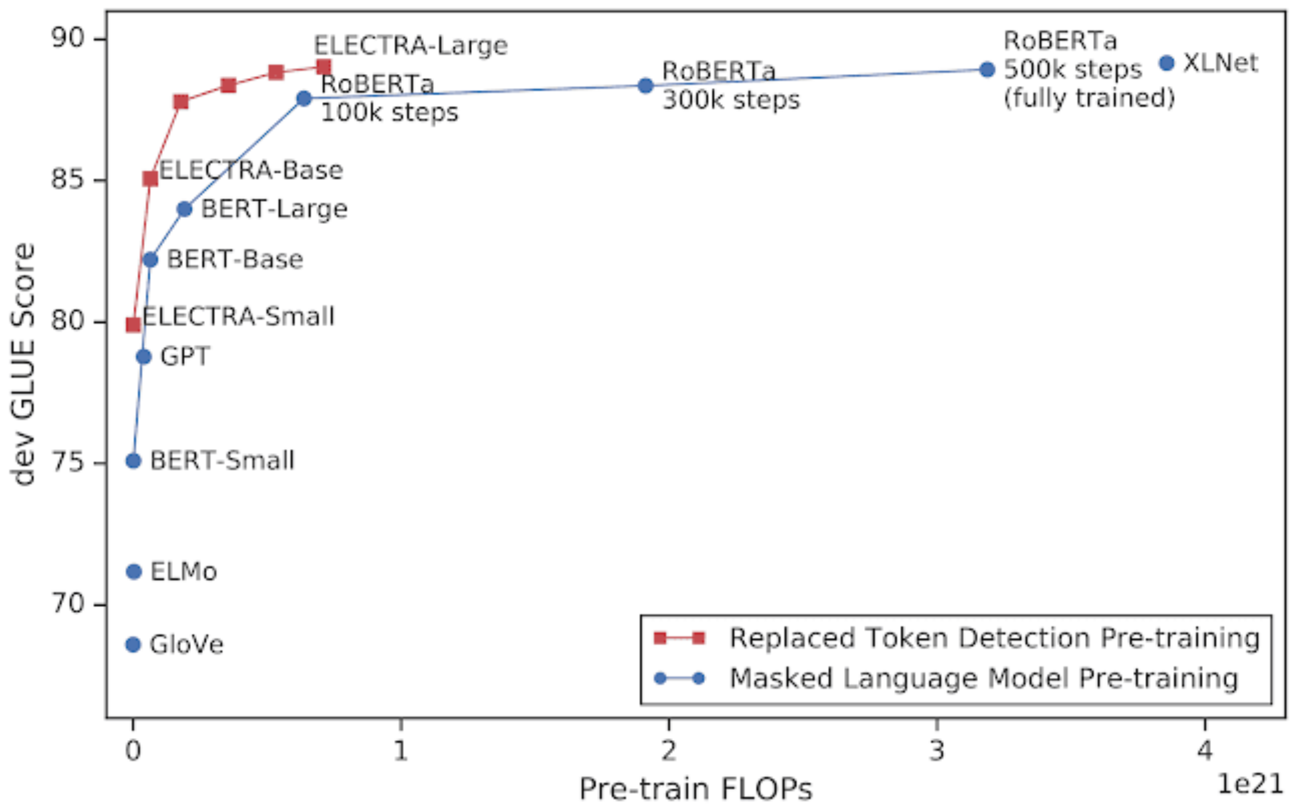
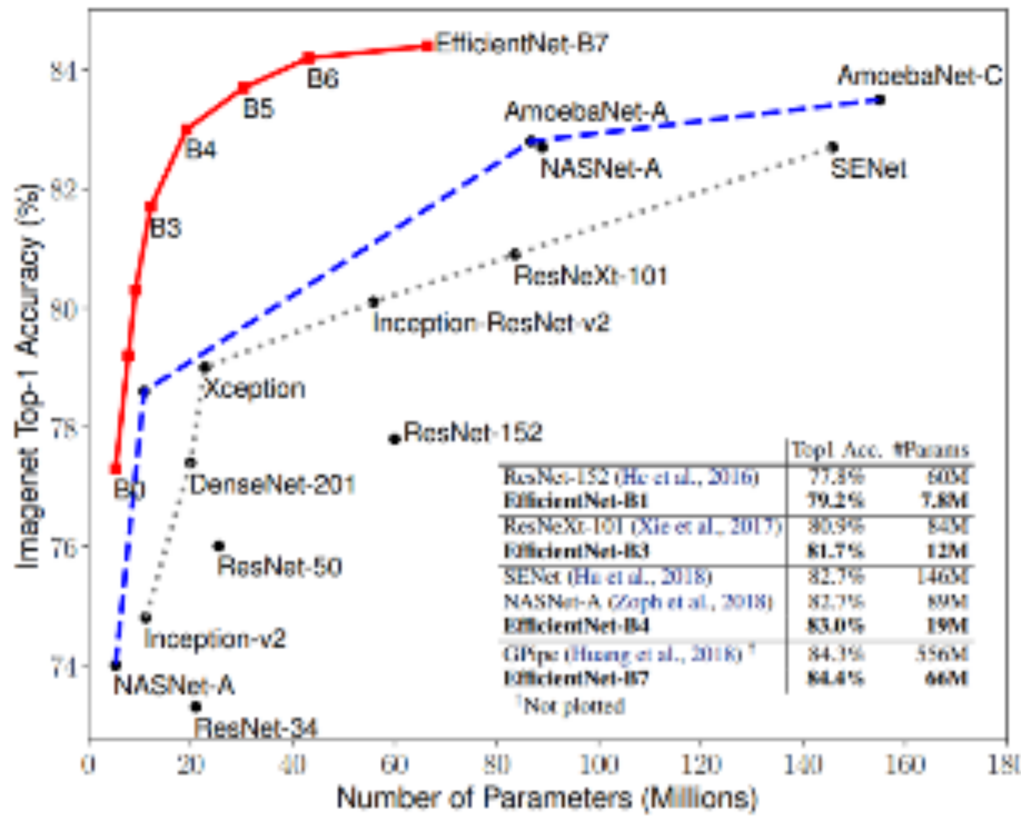
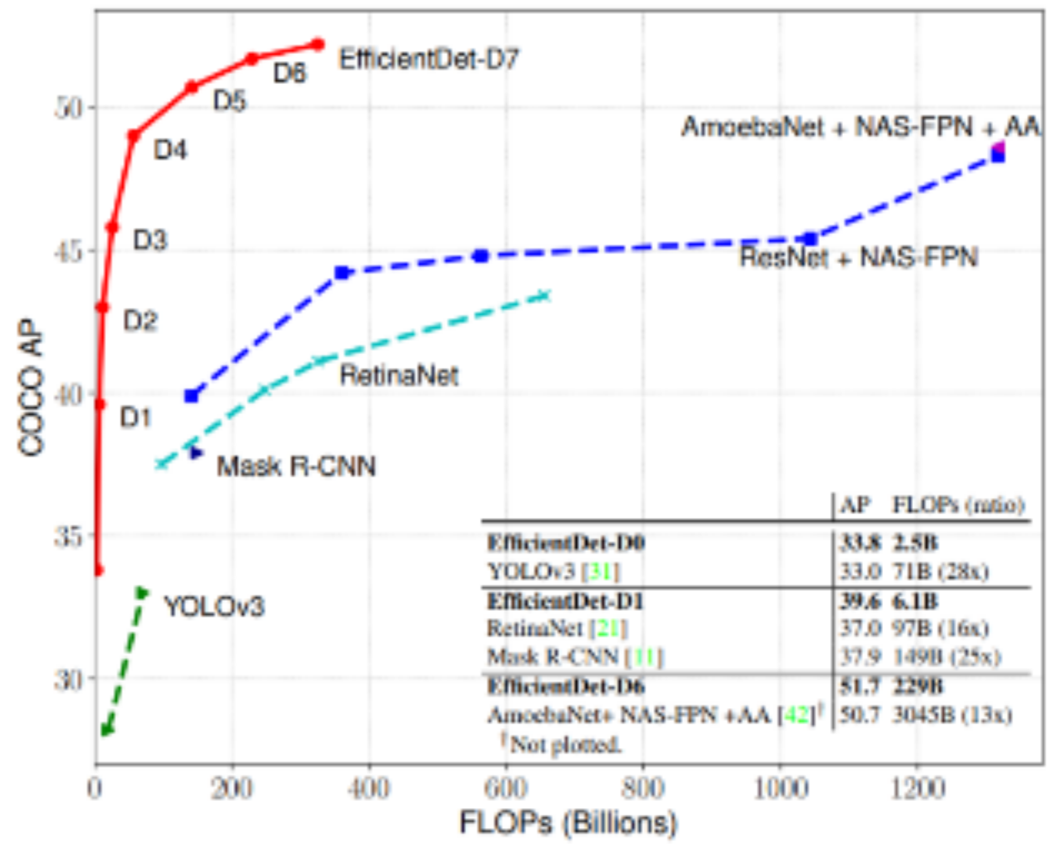


Image Classification



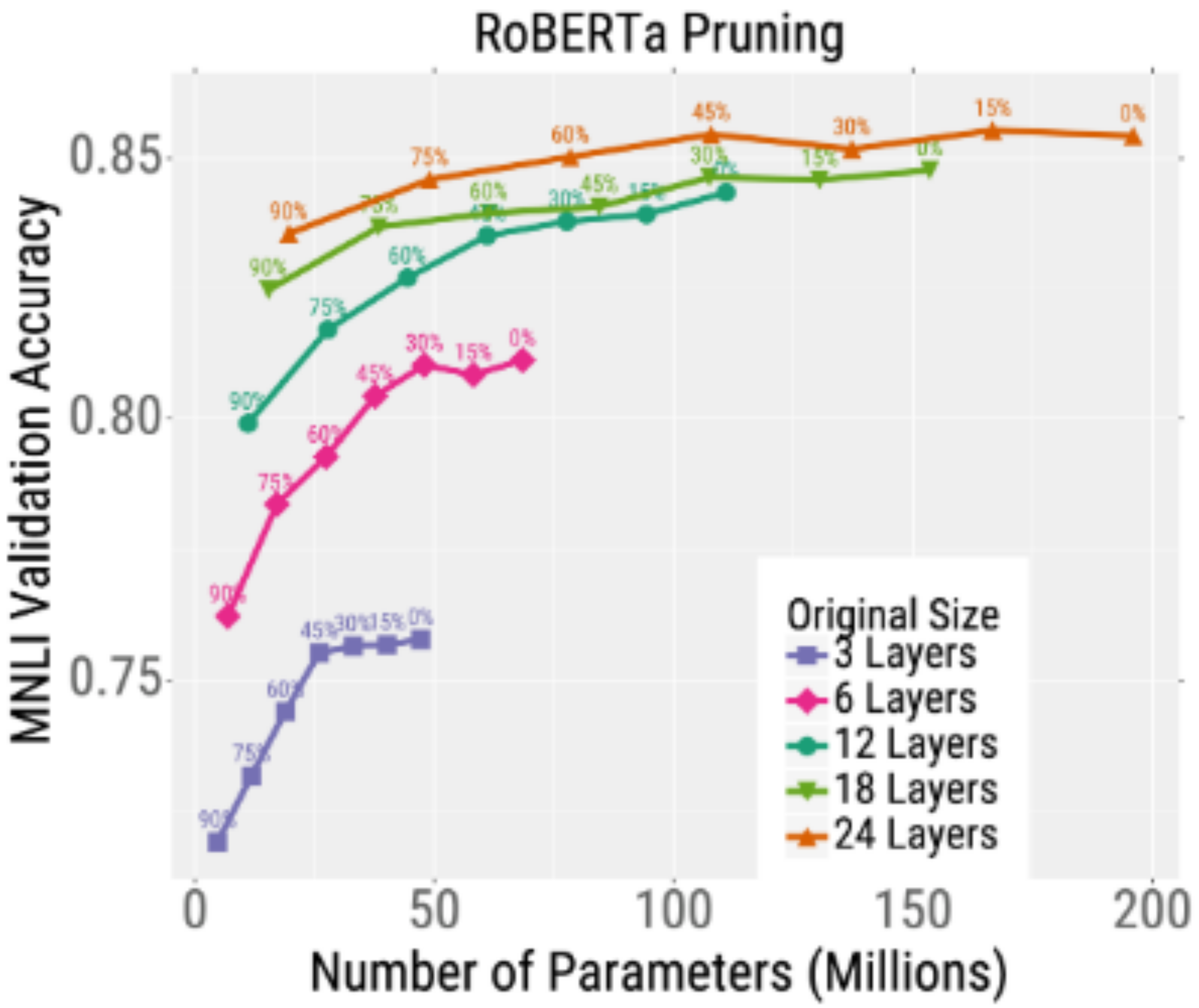
Object detection

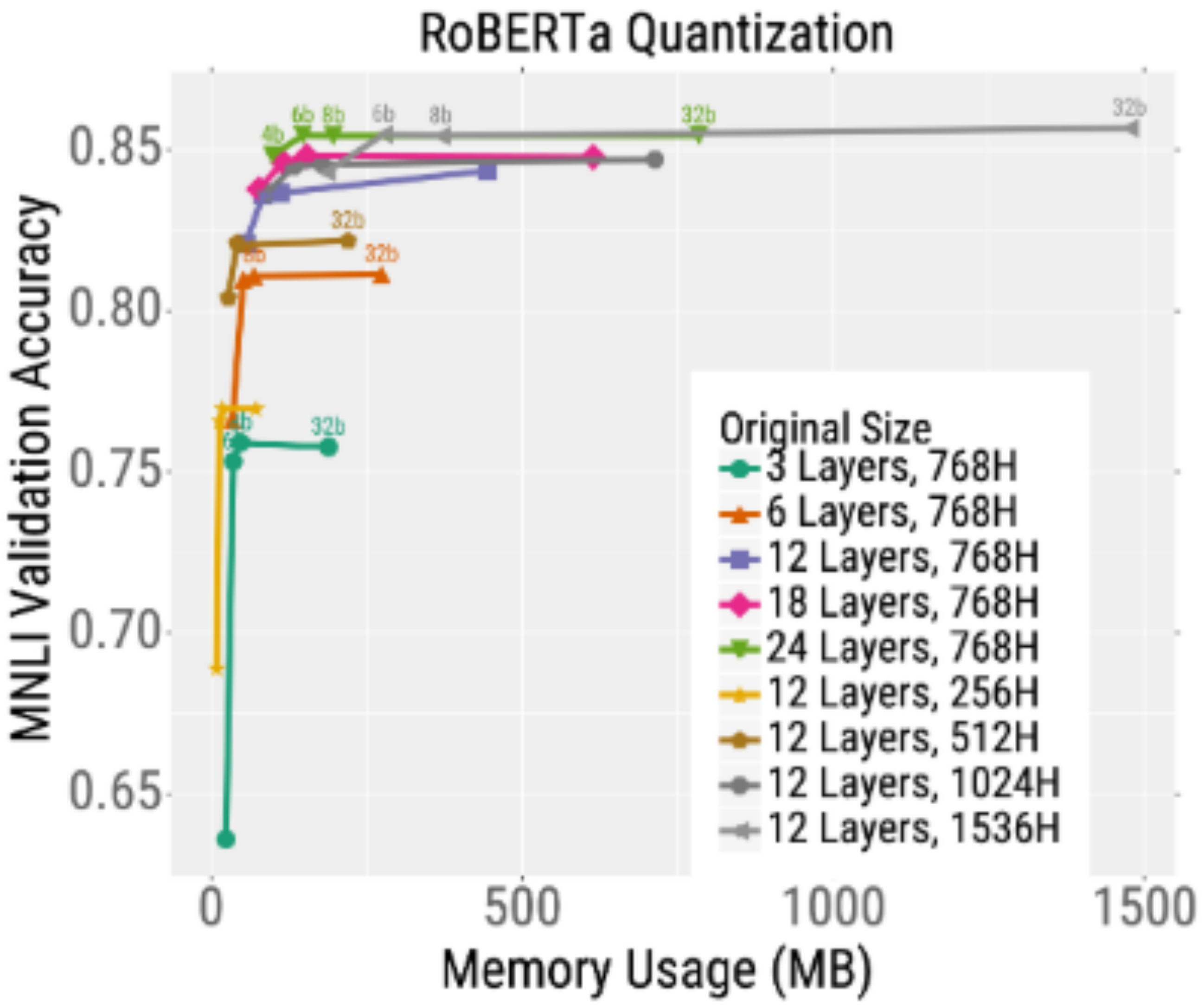


MODEL SELECTION

Not all models respond in the same way to knowledge distillation, pruning and quantization

Common Practice	Train Small Model	Stop Training When Converged	Lightly Compress
Optimal	Train Large Model	Stop Training Early	Heavily Compress





https://bair.berkeley.edu/blog/2020/03/05/compress/

Li, Z., Wallace, E., Shen, S., Lin, K., Keutzer, K., Klein, D., & Gonzalez, J. E. (2020). Train large, then compress: Rethinking model size for efficient training and inference of transformers. arXiv preprint arXiv:2002.11794.

MODEL SELECTION

And very large models are and will continue to be prevalent in NLP

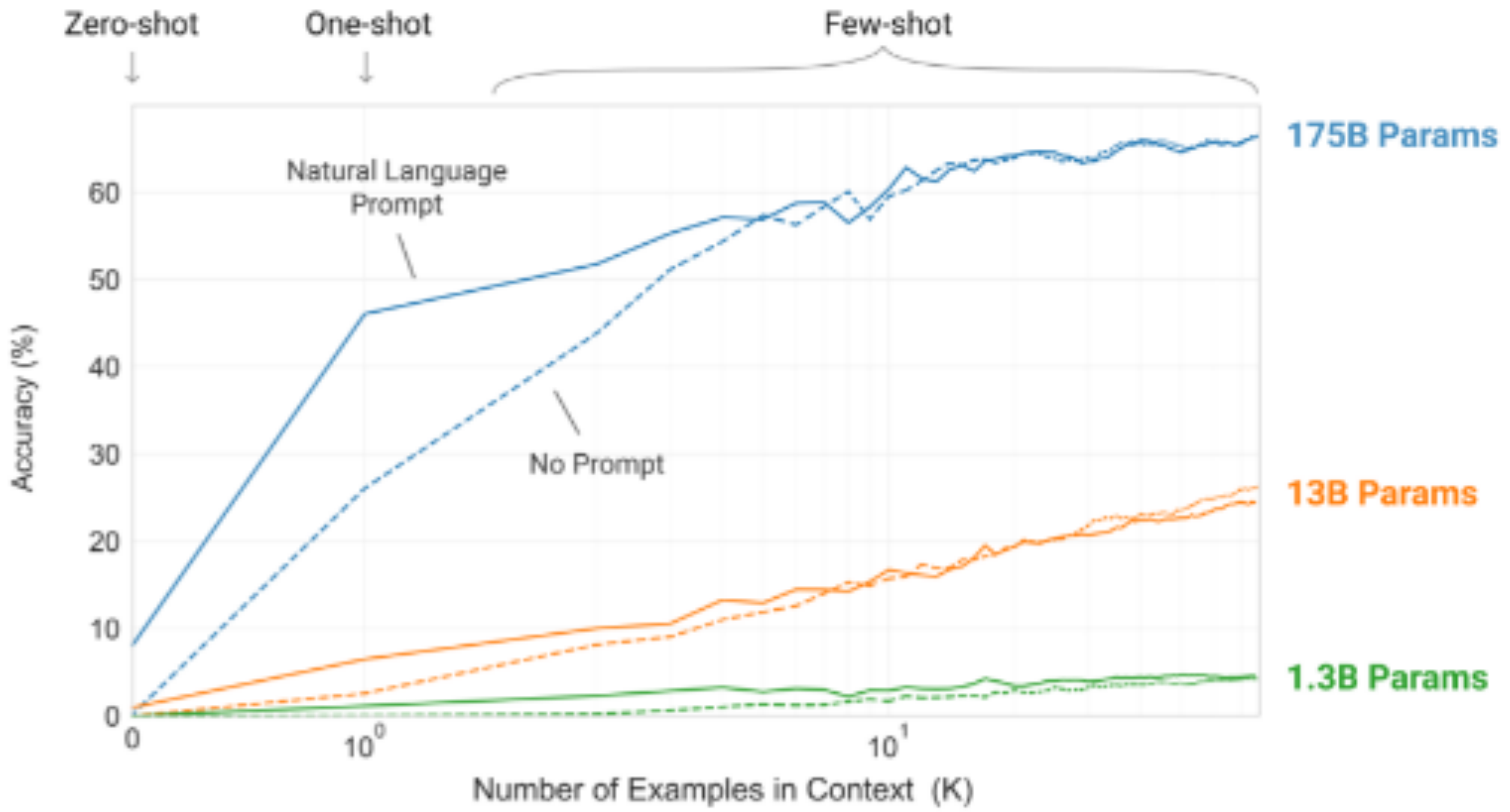


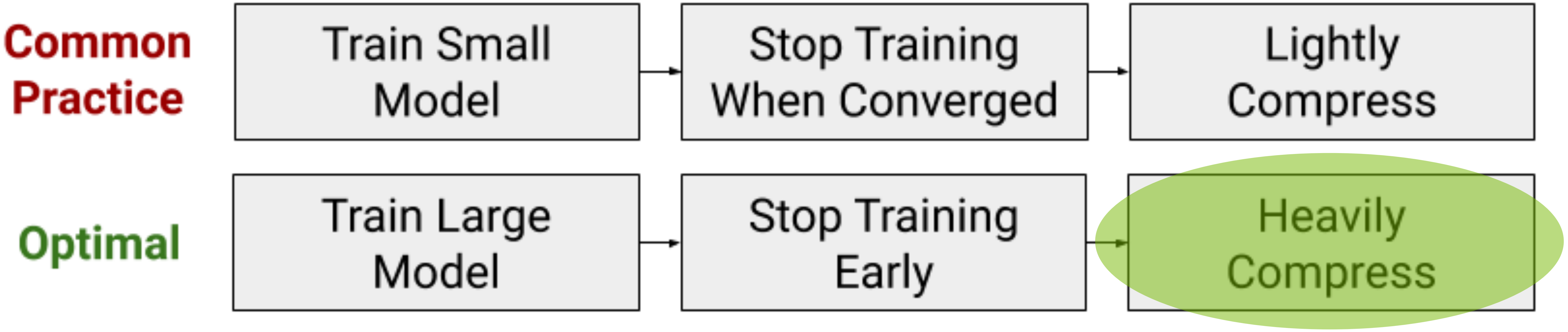
Figure 1.2: Larger models make increasingly efficient use of in-context information. We show in-context learning performance on a simple task requiring the model to remove random symbols from a word, both with and without a natural language task description (see Sec. 3.9.2). The steeper "in-context learning curves" for large models demonstrate improved ability to learn a task from contextual information. We see qualitatively similar behavior across a wide range of tasks.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Agarwal, S. (2020). Language Models are Few-Shot Learners. arXiv preprint arXiv:2005.14165.

DIRECT IMPLICATIONS

INCREASING IMPORTANCE OF PRUNING AND QUANTIZATION

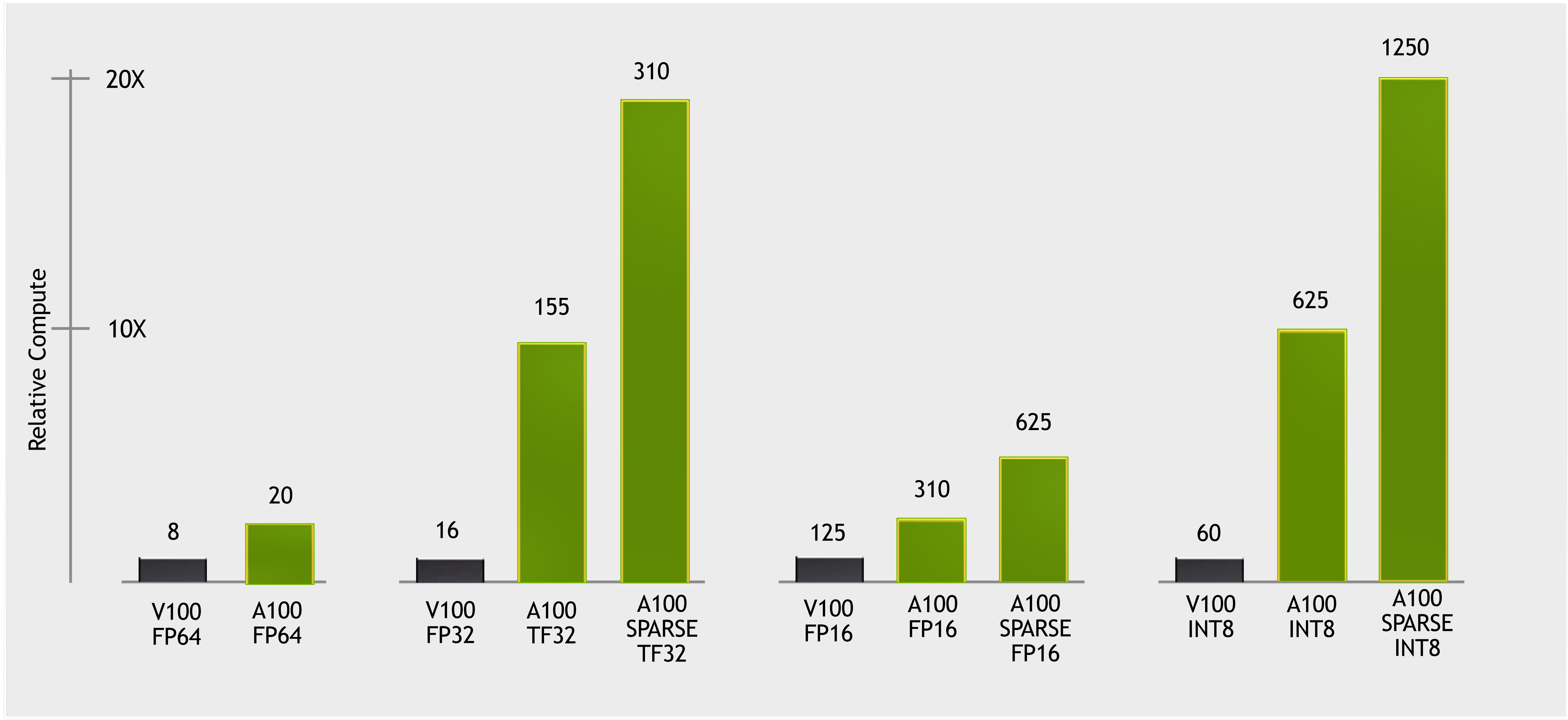
E.g. Train Large then compress



https://bair.berkeley.edu/blog/2020/03/05/compress/
Li, Z., Wallace, E., Shen, S., Lin, K., Keutzer, K., Klein, D., & Gonzalez, J. E. (2020). Train large, then compress: Rethinking model size for efficient training and inference of transformers. arXiv preprint arXiv:2002.11794.

INCREASING IMPORTANCE OF PRUNING AND QUANTIZATION

Hardware acceleration for reduced precision arithmetic and sparsity

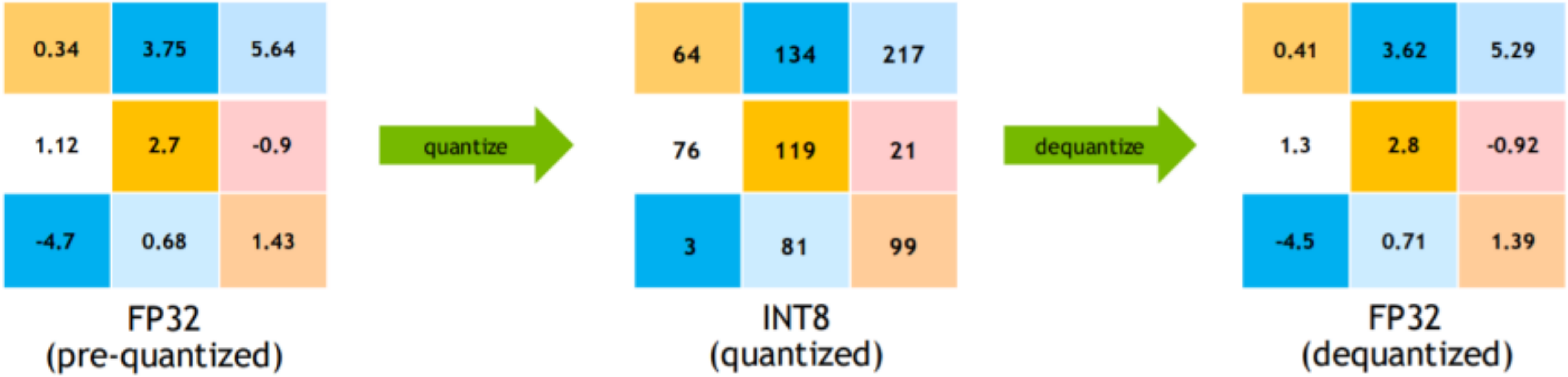


Part 3: Production Deployment

- Lecture
  - Model Selection
  - Post-Training Optimization
  - Product Quantization
  - Knowledge Distillation
  - Model Code Efficiency
  - Model Serving
  - Building the Application
- Lab
  - Exporting the Model
  - Hosting the Model
  - Server Performance
  - Using the Model

QUANTIZATION

The idea



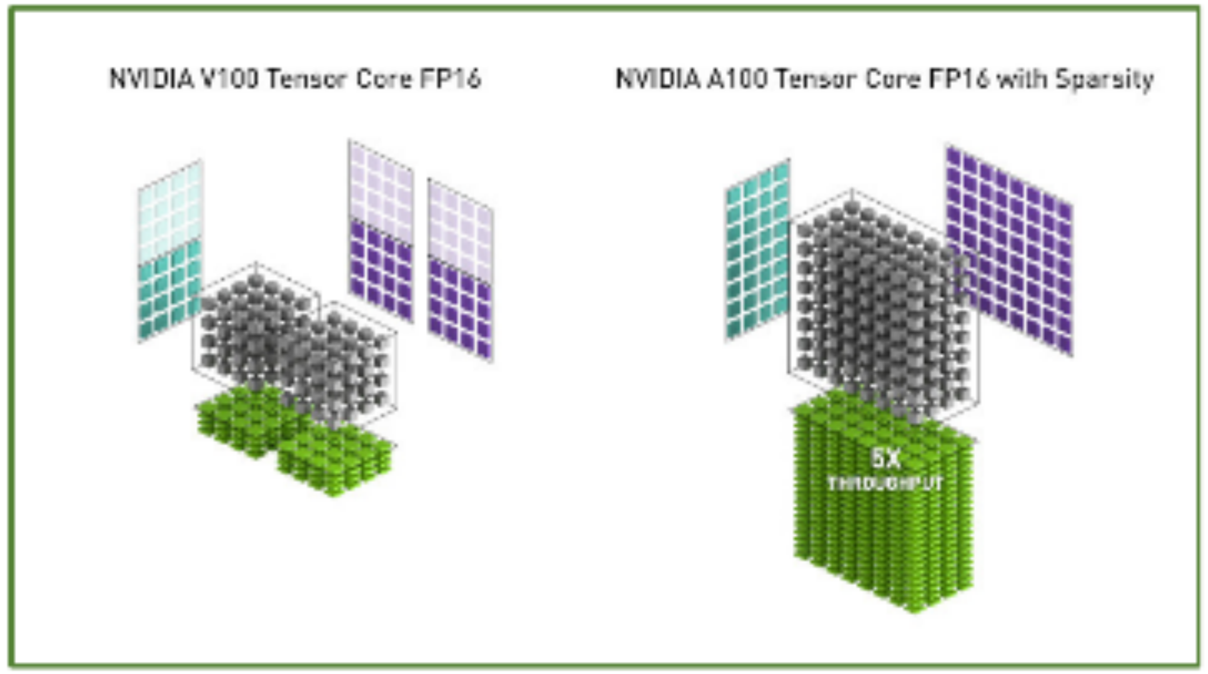
QUANTIZATION

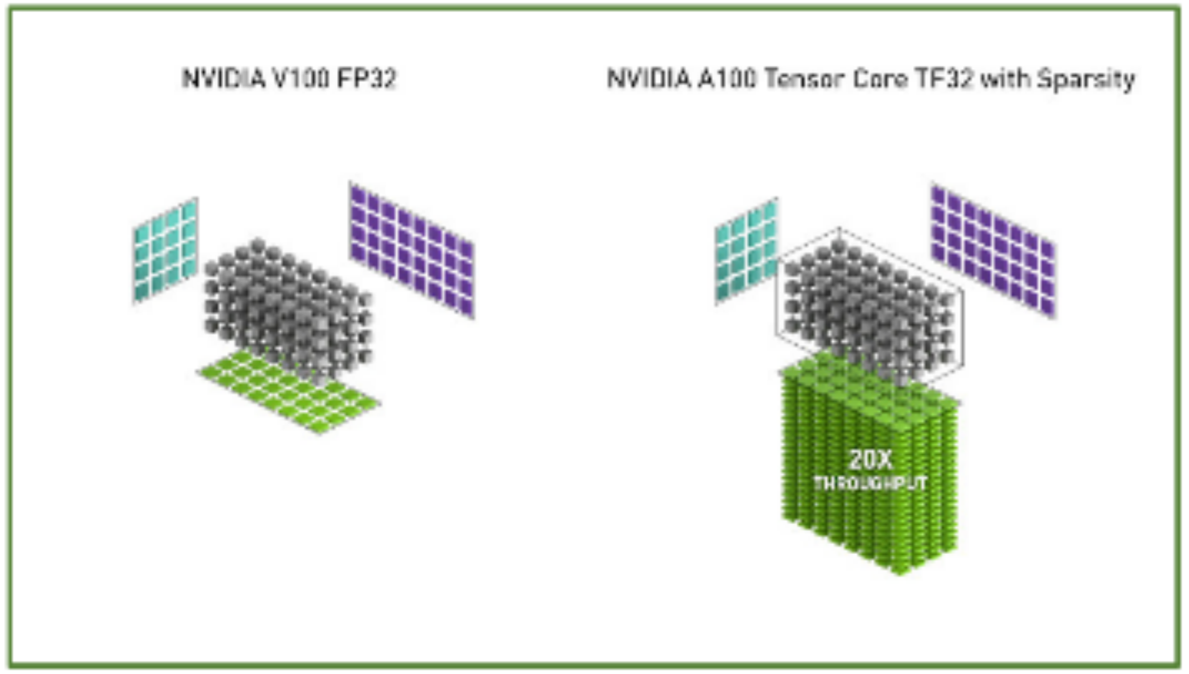
The rationale

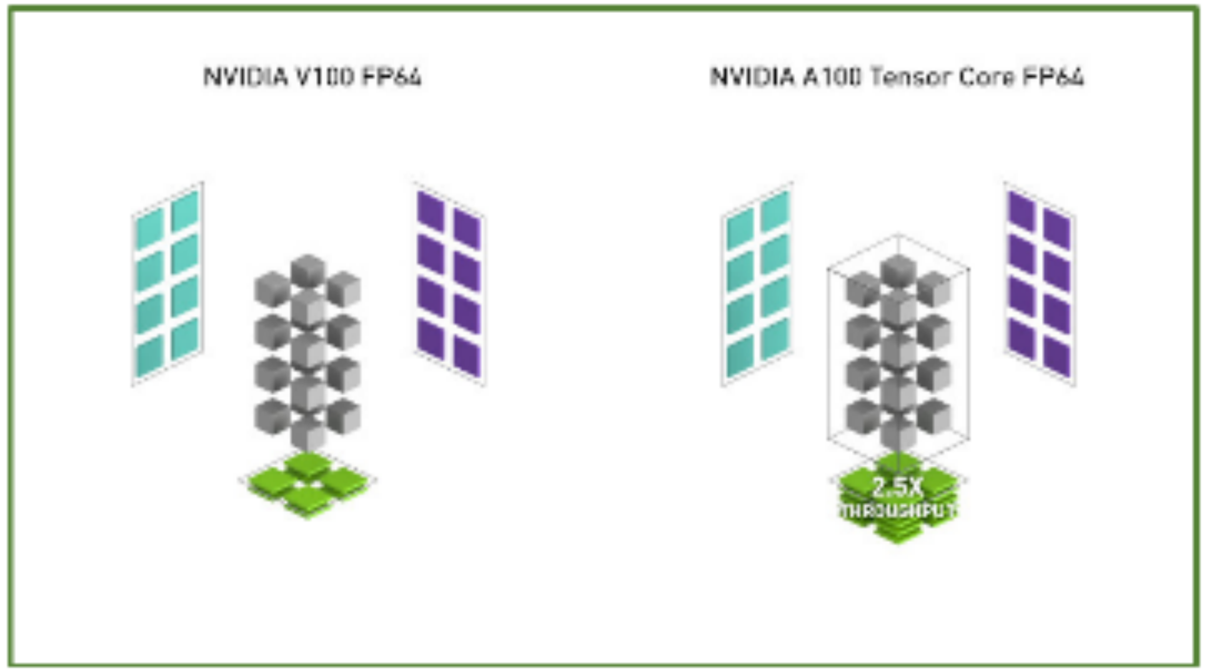
Input Datatype	Accumulation Datatype	Math Throughput	Bandwidth Reduction
FP32	FP32	1x	1x
FP16	FP16	8x	2x
INT8	INT32	16x	4x
INT4	INT32	32x	8x
INT1	INT32	128x	32x

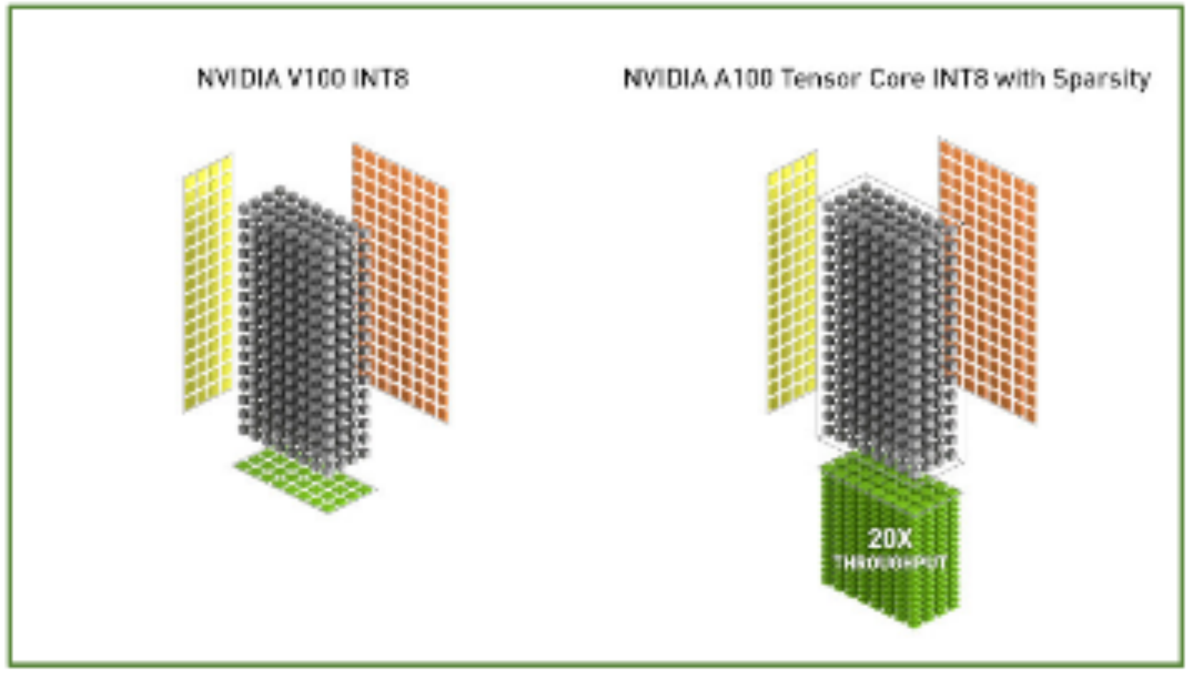
QUANTIZATION

The rationale









QUANTIZATION

The results (speedup and throughput)

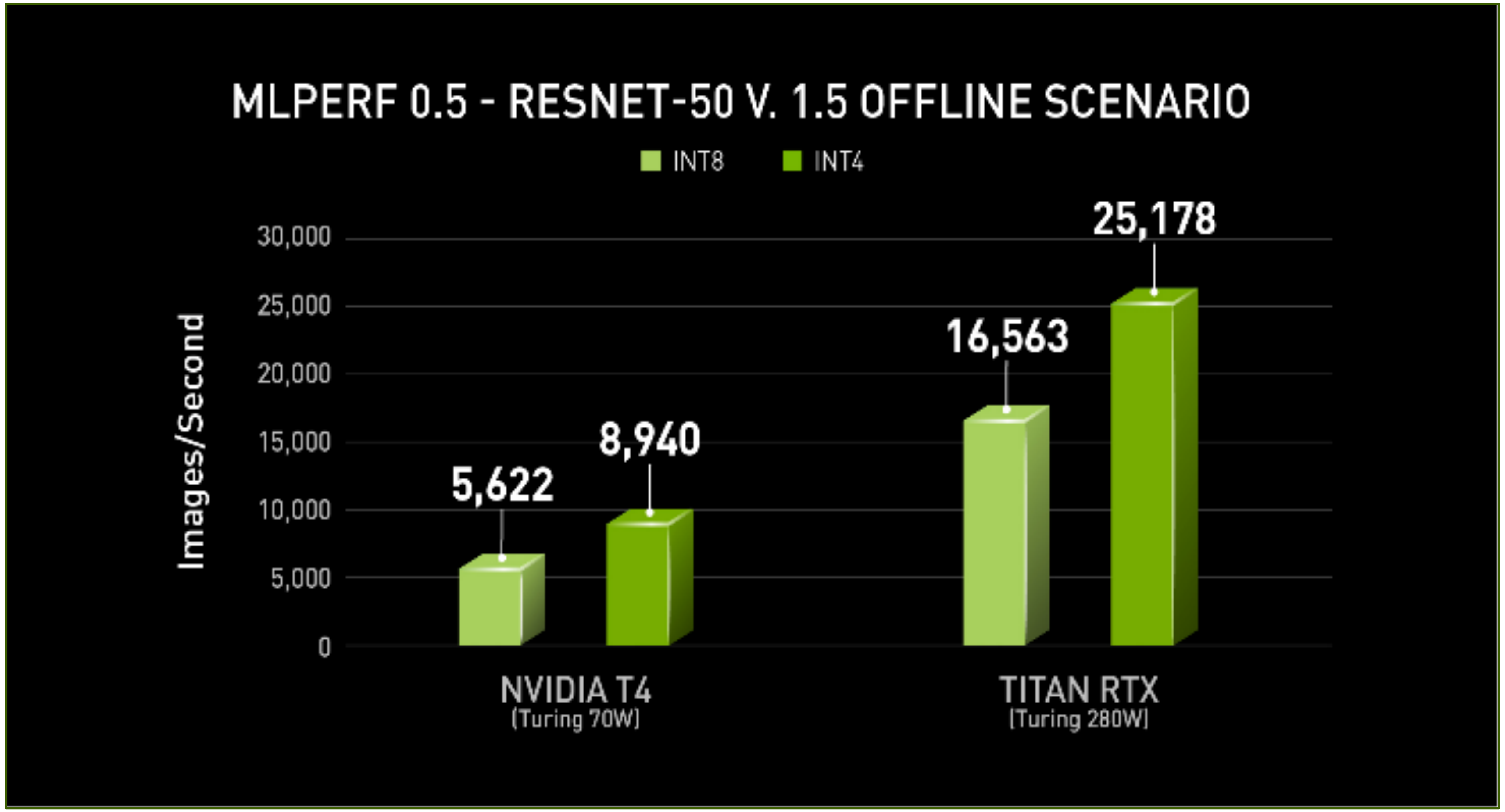
	Batch size 1			Batch size 8			Batch size 128		
	FP32	FP16	Int8	FP32	FP16	Int8	FP32	FP16	Int8
MobileNet v1	1	1.91	2.49	1	3.03	5.50	1	3.03	6.21
MobileNet v2	1	1.50	1.90	1	2.34	3.98	1	2.33	4.58
ResNet50 (v1.5)	1	2.07	3.52	1	4.09	7.25	1	4.27	7.95
VGG-16	1	2.63	2.71	1	4.14	6.44	1	3.88	8.00
VGG-19	1	2.88	3.09	1	4.25	6.95	1	4.01	8.30
Inception v3	1	2.38	3.95	1	3.76	6.36	1	3.91	6.65
Inception v4	1	2.99	4.42	1	4.44	7.05	1	4.59	7.20
ResNext101	1	2.49	3.55	1	3.58	6.26	1	3.85	7.39

Image/s	Batch size 1			Batch size 8			Batch size 128		
	FP32	FP16	Int8	FP32	FP16	Int8	FP32	FP16	Int8
MobileNet v1	1509	2889	3762	2455	7430	13493	2718	8247	16885
MobileNet v2	1082	1618	2060	2267	5307	9016	2761	6431	12652
ResNet50 (v1.5)	298	617	1051	500	2045	3625	580	2475	4609
VGG-16	153	403	415	197	816	1269	236	915	1889
VGG-19	124	358	384	158	673	1101	187	749	1552
Inception v3	156	371	616	350	1318	2228	385	1507	2560
Inception v4	76	226	335	173	768	1219	186	853	1339
ResNext101	84	208	297	200	716	1253	233	899	1724

TensorRT optimized models executed on Tesla T4, input size 224x224 for all apart from the Inception networks for which the input size was 299x299

QUANTIZATION

Beyond INT8



INT4 quantization for resnet50
"Int4 Precision for AI Inference"

IMPACT ON ACCURACY

In a wide range of cases minimal

Model	FP32	Int8 (max)	Int8 (entropy)	Rel Err (entropy)
MobileNet v1	71.01	69.43	69.46	2.18%
MobileNet v2	74.08	73.96	73.85	0.31%
NASNet (large)	82.72	82.09	82.66	0.07%
NASNet (mobile)	73.97	12.95	73.4	0.77%
ResNet50 (v1.5)	76.51	76.11	76.28	0.30%
ResNet50 (v2)	76.37	75.73	76.22	0.20%
ResNet152 (v1.5)	78.22	5.29	77.95	0.35%
ResNet152 (v2)	78.45	78.05	78.15	0.38%
VGG-16	70.89	70.75	70.82	0.10%
VGG-19	71.01	70.91	70.85	0.23%
Inception v3	77.99	77.7	77.85	0.18%
Inception v4	80.19	1.68	80.16	0.04%

COCO

Model	Backbone	FP32	INT8	Rel Err
SSD-300	MobileNet v1	26	25.8	0.77%
SSD-300	MobileNet v2	27.4	26.8	2.19%
Faster RCNN	ResNet-101	33.7	33.4	0.89%

All results COCO mAP on COCO 2017 validation, higher is better

Pascal VOC

Model	Backbone	FP32	INT8	Rel Err
SSD-300	VGG-16	77.7	77.6	0.13%
SSD-512	VGG-16	79.9	79.9	0.0%

All results VOC mAP on VOC 07 test, higher is better

IMPACT OF MODEL DESIGN

Not all neural network mechanisms quantize well

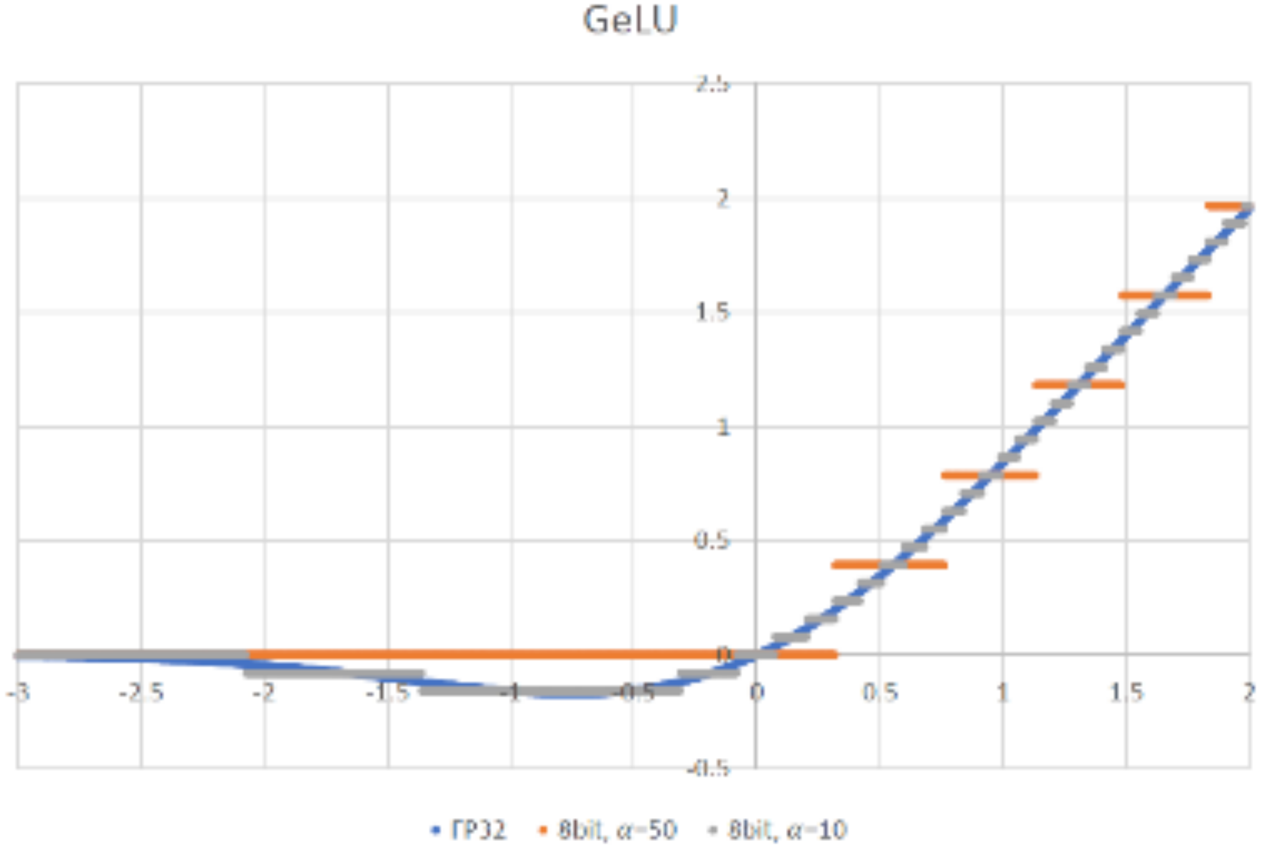
Bert large uncased	FP32	Int8	Rel Err %
MRPC	0.855	0.823	3.74%
SQuAD 1.1 (F1)	91.01	85.16	6.43%

IMPACT OF MODEL DESIGN

Model alterations required

Bert large uncased	FP32	Int8	Rel Err %
MRPC	0.855	0.823	3.74%
SQuAD 1.1 (F1)	91.01	85.16	6.43%

Bert large uncased	FP32	Int8 (GeLU10)	Rel Err %
MRPC	0.855	0.843	0.70%
SQuAD 1.1 (F1)	91.01	90.40	0.67%



$$f(x) = \frac{x}{2}(1 + erf(\frac{x}{\sqrt{2}}))$$

- GeLU produces highly asymmetric range
- Negative values between [-0.17,0]
- All negative values clipped to 0
- GeLU10 allows to maintain negative values

LOSS OF ACCURACY

Reasons

Outlier in the tensor:

- Example: BERT, Inception V4
- Solution: Clip. Tighten the range, use bits more efficiently

Not enough precision in quantized representation

- Example: Int8 for MobileNet V1
- Example: Int4 for Resnet50
- Solution: Train/fine tune for quantization

LEARN MORE

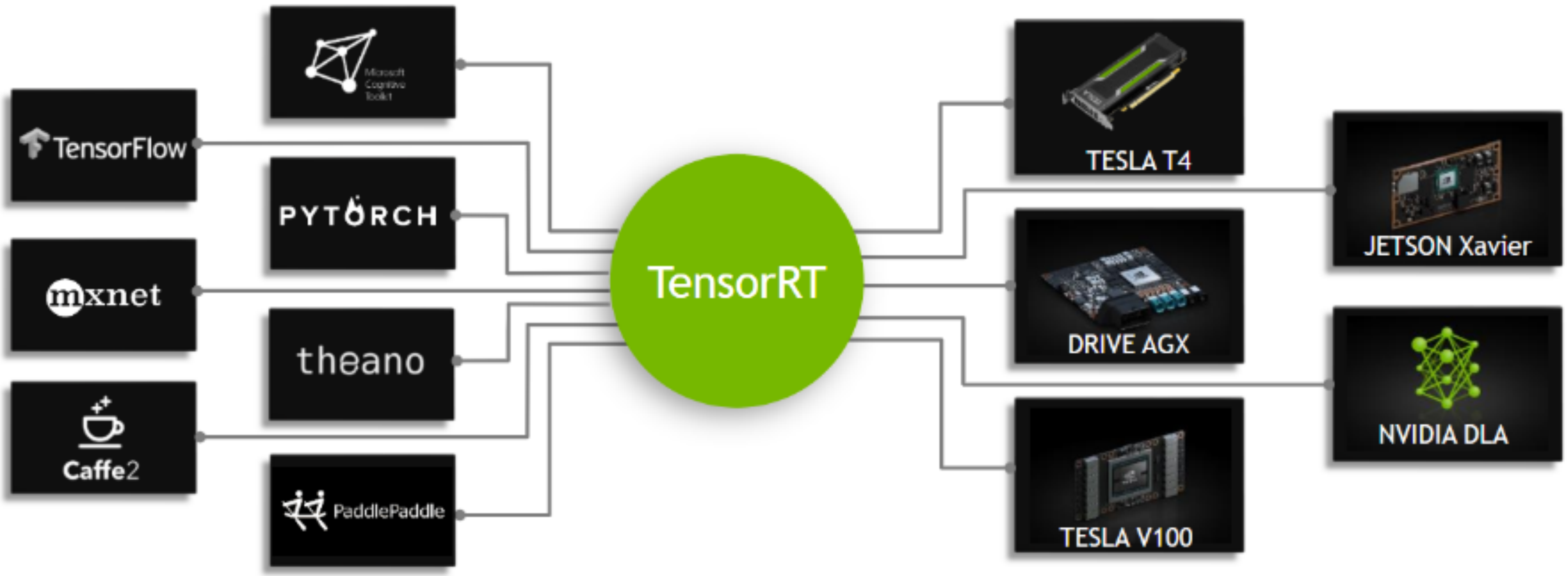
GTC Talks

- S9659: Inference at Reduced Precision on GPUs
- S21664: Toward INT8 Inference: Deploying Quantization-Aware Trained Networks using TensorRT

QUANTIZATION TOOLS

NVIDIA TENSORRT

From Every Framework, Optimized For Each Target Platform



INT8 QUANTIZATION EXAMPLE

TF-TRT

```
Step 1  Obtain the TF frozen graph (trained in FP32)
…

Step 2  Create the calibration graph -> Execute it with calibration data -> Convert it to the INT8
optimized graph
# create a TRT inference graph, the output is a frozen graph ready for calibration
calib_graph = trt.create_inference_graph(input_graph_def=frozen_graph, outputs=outputs,
              max_batch_size=1, max_workspace_size_bytes=1<<30,
              precision_mode="INT8", minimum_segment_size=5)

# Run calibration (inference) in FP32 on calibration data (no conversion)
f_score, f_geo = tf.import_graph_def(calib_graph, input_map={"input_images":inputs},
              return_elements=outputs, name="")
Loop img: score, geometry = sess.run([f_score, f_geo], feed_dict={inputs: [img]})

# apply TRT optimizations to the calibration graph, replace each TF subgraph with a TRT node
optimized for INT8
trt_graph = trt.calib_graph_to_infer_graph(calib_graph)

Step 3  Import the TRT graph and run
…
```

https://docs.nvidia.com/deeplearning/dgx/tf-trt-user-guide/index.html

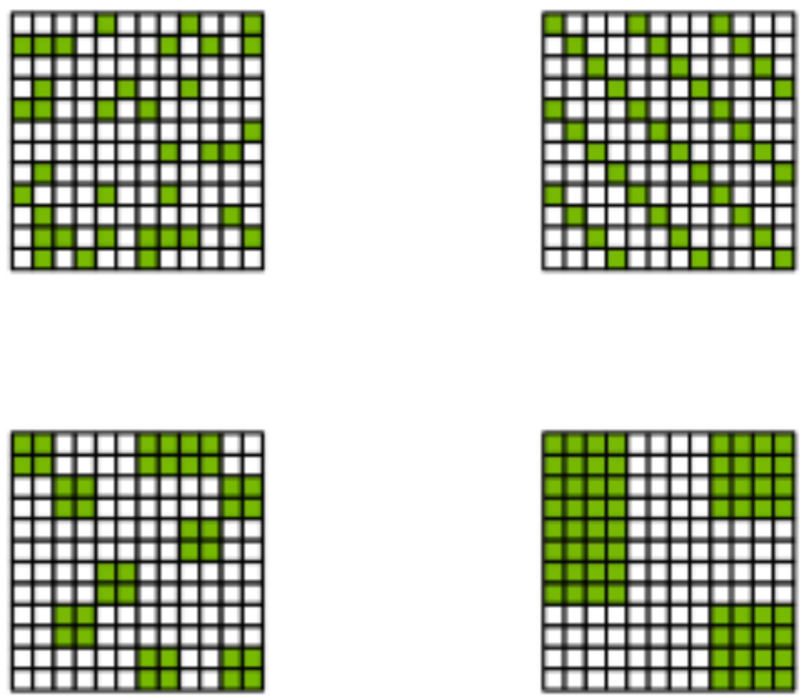
PRUNING

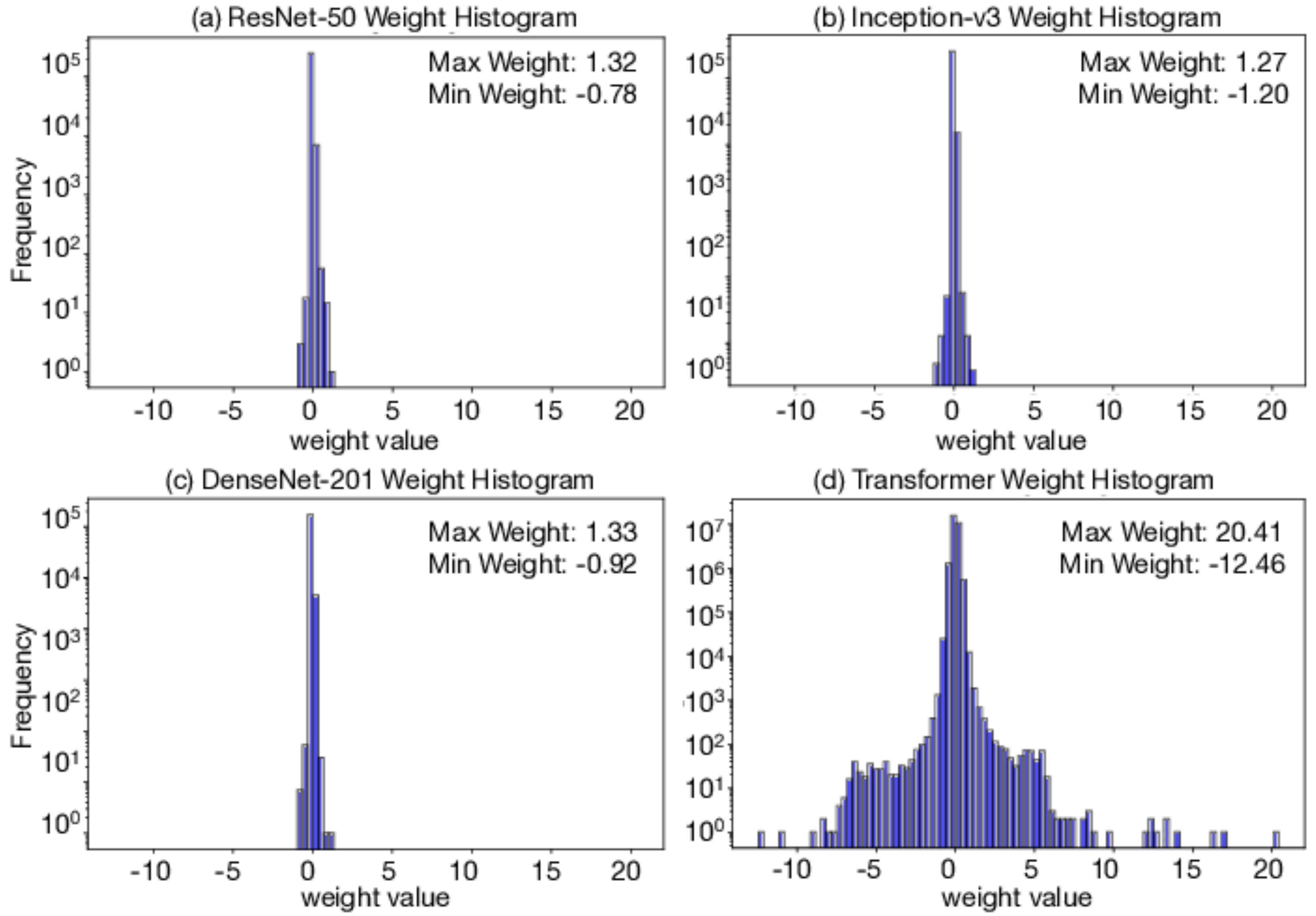
PRUNING

The idea

The opportunity:

- Reduced memory bandwidth
- Reduced memory footprint
- Acceleration (especially in presence of hardware acceleration)





Tambe, T., Yang, E. Y., Wan, Z., Deng, Y., Reddi, V. J., Rush, A., ... & Wei, G. Y. (2019). AdaptivFloat: A Floating-point based Data Type for Resilient Deep Learning Inference. *arXiv preprint arXiv:1909.13271*.

DIFFICULT TO GET TO WORK RELIABLY

STRUCTURED SPARSITY

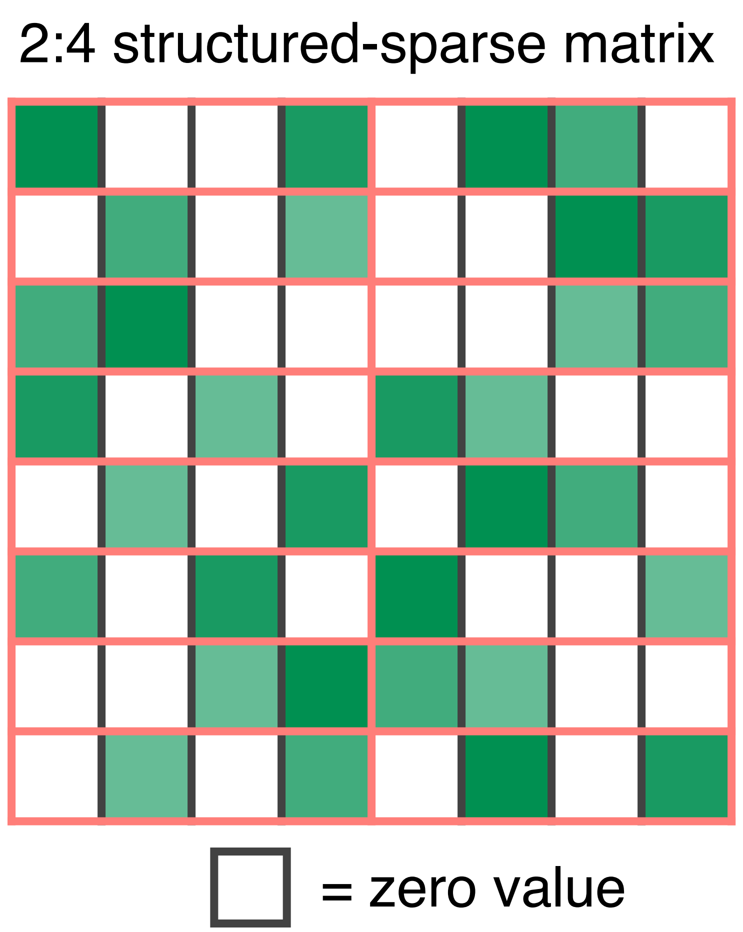
SPARSITY IN A100 GPU

Fine-grained structured sparsity for Tensor Cores

- 50% fine-grained sparsity
- 2:4 pattern: 2 values out of each contiguous block of 4 must be 0

Addresses the 3 challenges:

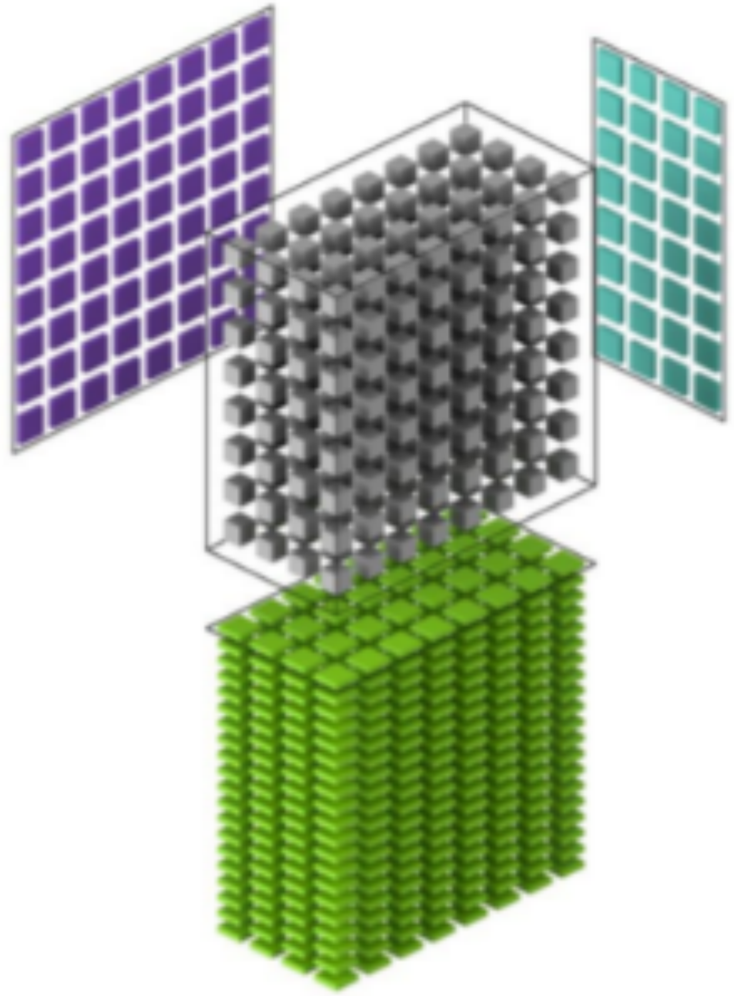
- Accuracy: maintains accuracy of the original, unpruned network
  - Medium sparsity level (50%), fine-grained
- Training: a recipe shown to work across tasks and networks
- Speedup:
  - Specialized Tensor Core support for sparse math
  - Structured: lends itself to efficient memory utilization



PRUNING

Structured sparsity

INPUT OPERANDS	ACCUMULATOR	TOPS	Dense vs. FFMA	Sparse Vs. FFMA
FP32	FP32	19.5	-	-
TF32	FP32	156	8X	16X
FP16	FP32	312	16X	32X
BF16	FP32	312	16X	32X
FP16	FP16	312	16X	32X
INT8	INT32	624	32X	64X
INT4	INT32	1248	64X	128X
BINARY	INT32	4992	256X	-



RELIABLE APPROACH

PRUNING

Model performance

Network	Accuracy				
	Dense FP16	Sparse FP16		Sparse INT8	
ResNet-34	73.7	73.9	0.2	73.7	-
ResNet-50	76.6	76.8	0.2	76.8	0.2
ResNet-101	77.7	78.0	0.3	77.9	-
ResNeXt-50-32x4d	77.6	77.7	0.1	77.7	-
ResNeXt-101-32x16d	79.7	79.9	0.2	79.9	0.2
DenseNet-121	75.5	75.3	-0.2	75.3	-0.2
DenseNet-161	78.8	78.8	-	78.9	0.1
Wide ResNet-50	78.5	78.6	0.1	78.5	-
Wide ResNet-101	78.9	79.2	0.3	79.1	0.2
Inception v3	77.1	77.1	-	77.1	-
Xception	79.2	79.2	-	79.2	-
VGG-16	74.0	74.1	0.1	74.1	0.1
VGG-19	75.0	75.0	-	75.0	-

PRUNING

Model performance

Network	Accuracy				
	Dense FP16	Sparse FP16		Sparse INT8	
ResNet-50 (SWSL)	81.1	80.9	-0.2	80.9	-0.2
ResNeXt-101-32x8d (SWSL)	84.3	84.1	-0.2	83.9	-0.4
ResNeXt-101-32x16d (WSL)	84.2	84.0	-0.2	84.2	-
SUNet-7-128	76.4	76.5	0.1	76.3	-0.1
DRN-105	79.4	79.5	0.1	79.4	-

PRUNING

Model performance

Network	Dense FP16	Sparse FP16		Sparse INT8	
	Accuracy				
MaskRCNN-RN50	37.9	37.9	-	37.8	-0.1
SSD-RN50	24.8	24.8	-	24.9	0.1
FasterRCNN-RN50-FPN-1x	37.6	38.6	1.0	38.4	0.8
FasterRCNN-RN50-FPN-3x	39.8	39.9	-0.1	39.4	-0.4
FasterRCNN-RN101-FPN-3x	41.9	42.0	0.1	41.8	-0.1
MaskRCNN-RN50-FPN-1x	39.9	40.3	0.4	40.0	0.1
MaskRCNN-RN50-FPN-3x	40.6	40.7	0.1	40.4	0.2
MaskRCNN-RN101-FPN-3x	42.9	43.2	0.3	42.8	0.1
RetinaNet-RN50-FPN-1x	36.4	37.4	1.0	37.2	0.8
RPN-RN50-FPN-1x	45.8	45.6	-0.2	45.5	0.3

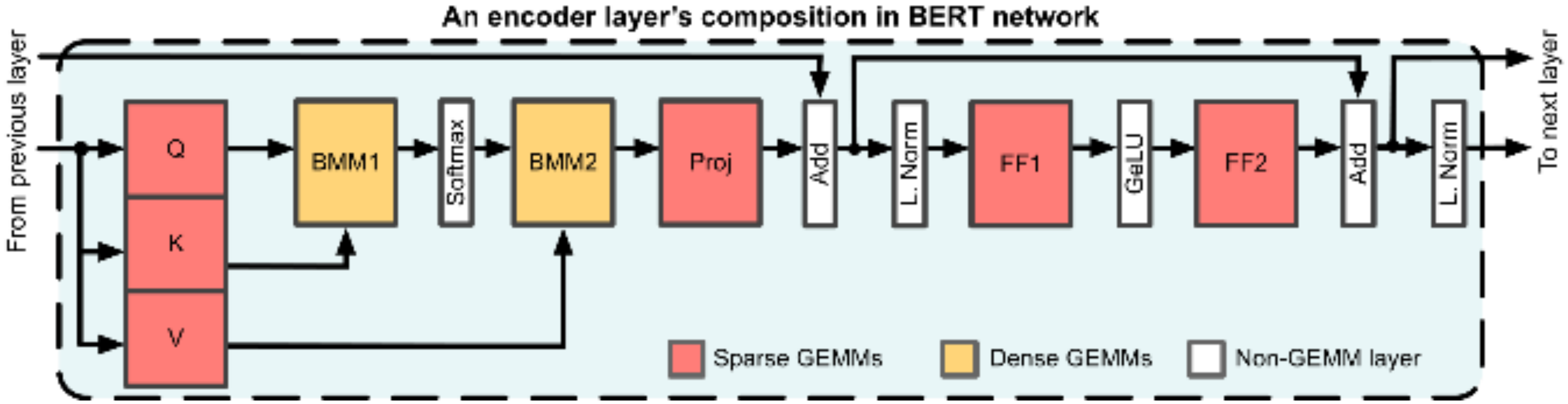
RN = ResNet Backbone
FPN = Feature Pyramid Network
RPN = Region Proposal Network

IMPACT ON NLP

NETWORK PERFORMANCE

BERT-Large

1.8x GEMM Performance -> 1.5x Network Performance
Some operations remain dense:
Non-GEMM layers (Softmax, Residual add, Normalization, Activation functions, …)
GEMMs without weights to be pruned – Attention Batched Matrix Multiplies



TRAINING RECIPE

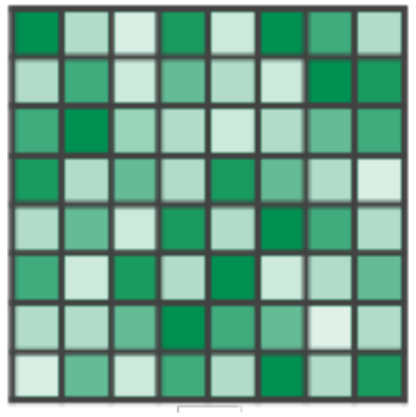
RECIPE FOR 2:4 SPARSE NETWORK TRAINING

1) Train (or obtain) a dense network

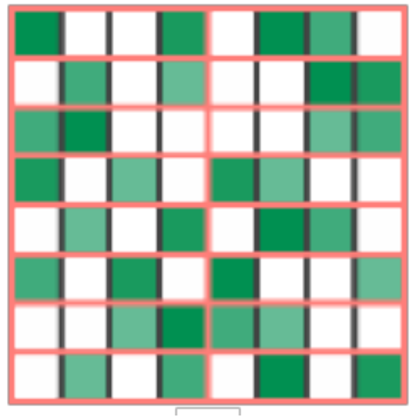
2) Prune for 2:4 sparsity

3) Repeat the original training procedure

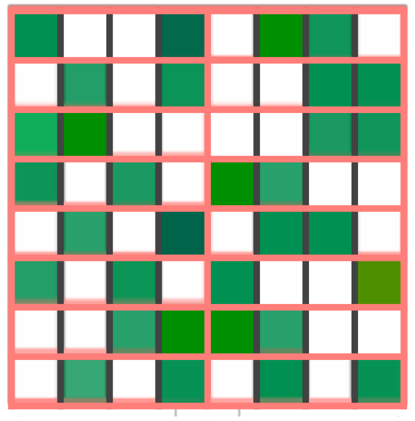
- Same hyper-parameters as in step-1
- Initialize to weights from step-2
- Maintain the 0 pattern from step-2: no need to recompute the mask



Dense weights

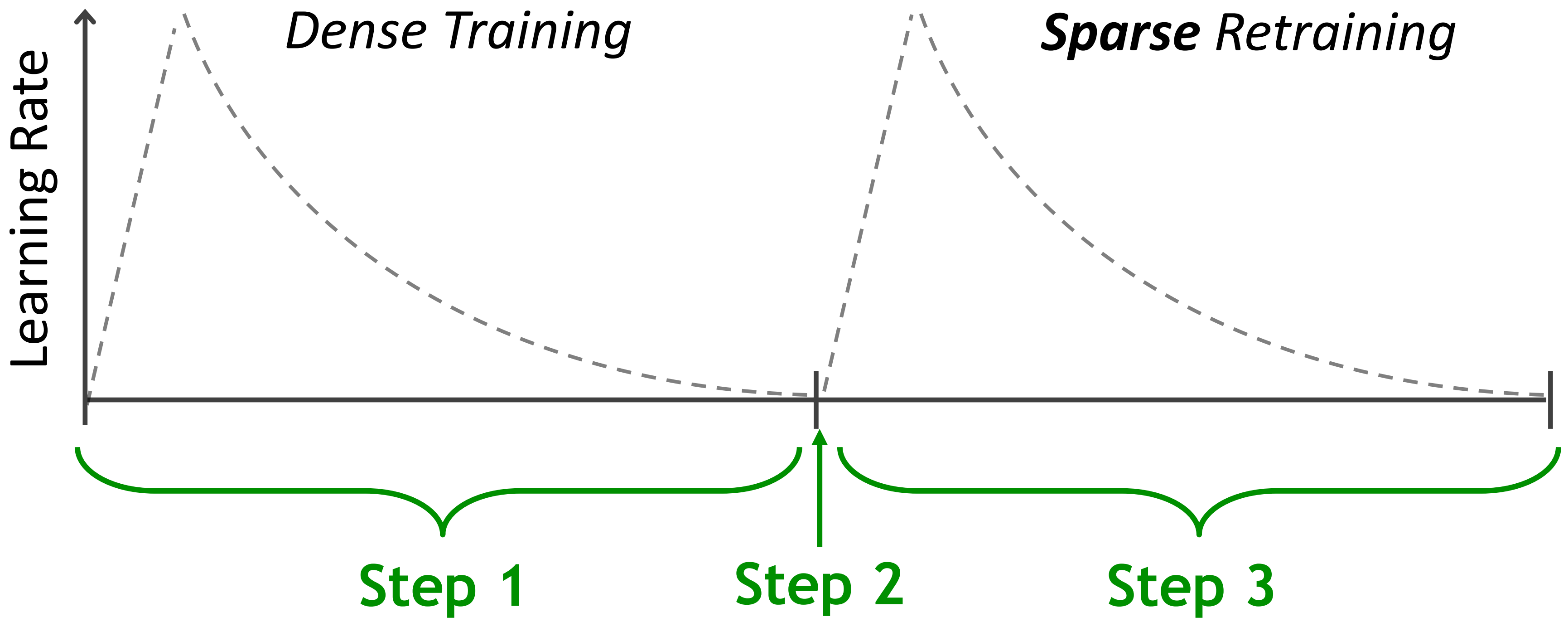


2:4 sparse weights



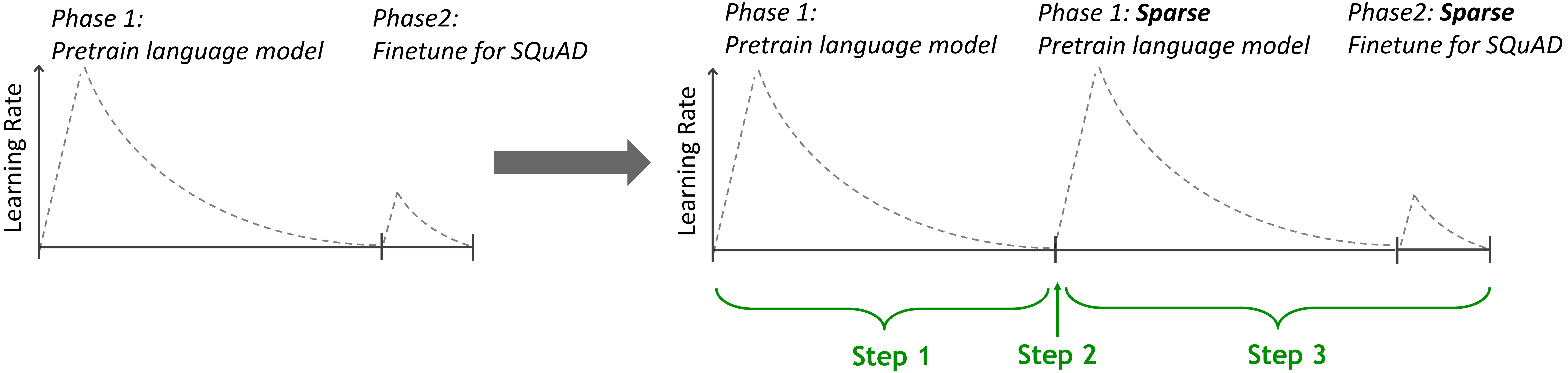
Retrained 2:4 sparse weights

EXAMPLE LEARNING RATE SCHEDULE



BERT SQUAD EXAMPLE

SQuAD Dataset and fine-tuning is too small to compensate for pruning on its own



APEX: AUTOMATIC SPARSITY

TAKING ADVANTAGE OF STRUCTURED SPARSITY

APEX's Automatic SParsity: ASP

PyTorch sparse fine-tuning loop

```
import torch
from apex.contrib.sparsity import ASP
device = torch.device('cuda')

model = TheModelClass(*args, **kwargs) # Define model structure
model.load_state_dict(torch.load('dense_model.pth'))

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9) # Define optimizer

ASP.prune_trained_model(model, optimizer)

x, y = DataLoader(…) #load data samples and labels to train the model
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

torch.save(model.state_dict(), 'pruned_model.pth') # checkpoint has weights and masks
```

Init mask buffers, tell optimizer to mask weights and gradients, compute sparse masks: Universal Fine Tuning

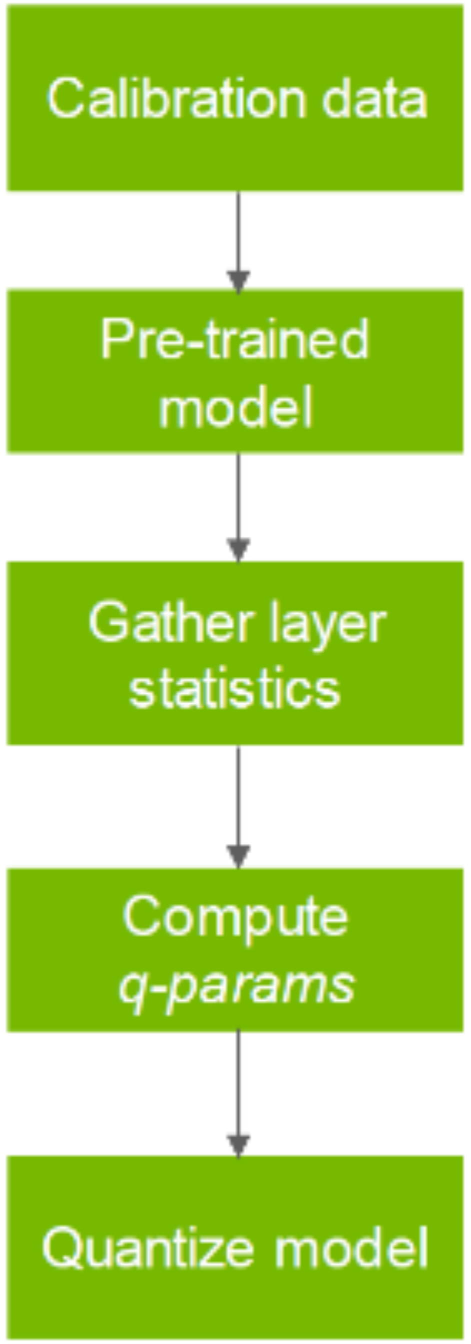
Part 3: Production Deployment

- Lecture
  - Model Selection
  - Post-Training Optimization
  - Product Quantization
  - Knowledge Distillation
  - Model Code Efficiency
  - Model Serving
  - Building the Application
- Lab
  - Exporting the Model
  - Hosting the Model
  - Server Performance
  - Using the Model

QUANTIZATION

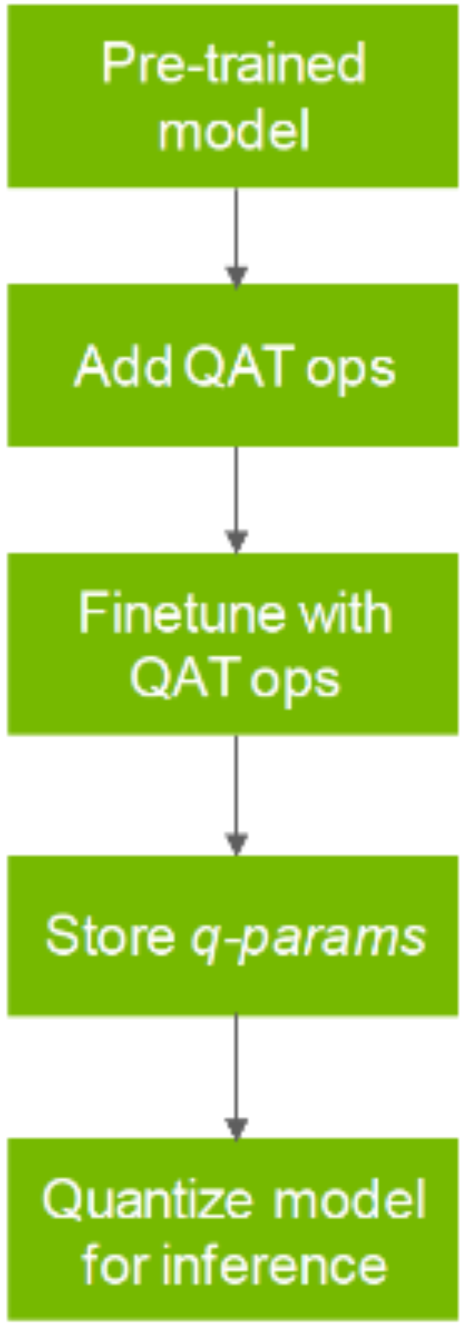
Approaches

Post-training quantization(PTQ)



PTQ	QAT
Usually fast	Slow
No re-training of the model	Model needs to be trained/finetuned
Plug and play of quantization schemes	Plug and play of quantization schemes (requires re-training)
Less control over final accuracy of the model	More control over final accuracy since *q-params* are learned during training.

Quantization-aware training (QAT)



EXTREME MODEL COMPRESSION

Training with quantization noise

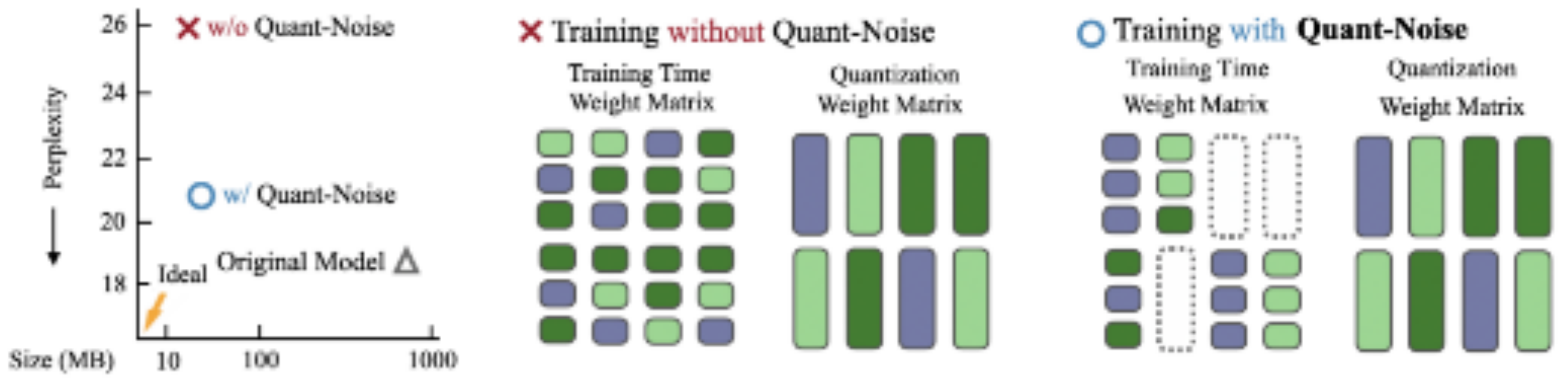


Figure 1: **Quant-Noise** trains models to be resilient to inference-time quantization by mimicking the effect of the quantization method during training time. This allows for extreme compression rates without much loss in accuracy on a variety of tasks and benchmarks.

Quantization Scheme	Language Modeling 16-layer Transformer Wikitext-103			Image Classification EfficientNet-B3 ImageNet-1k		
	Size	Compression	PPL	Size	Compression	Top-1
Uncompressed model	942	× 1	18.3	46.7	× 1	81.5
int4 quantization	118	× 8	39.4	5.8	× 8	45.3
- trained with QAT	118	× 8	34.1	5.8	× 8	59.4
- trained with Quant-Noise	118	× 8	**21.8**	5.8	× 8	**67.8**
int8 quantization	236	× 4	19.6	11.7	× 4	80.7
- trained with QAT	236	× 4	21.0	11.7	× 4	80.8
- trained with Quant-Noise	236	× 4	**18.7**	11.7	× 4	**80.9**
iPQ	38	× 25	25.2	3.3	× 14	79.0
- trained with QAT	38	× 25	41.2	3.3	× 14	55.7
- trained with Quant-Noise	38	× 25	**20.7**	3.3	× 14	**80.0**
iPQ & int8 + Quant-Noise	38	× 25	21.1	3.1	× 15	79.8

Table 1: **Comparison of different quantization schemes with and without Quant-Noise** on language modeling and image classification. For language modeling, we train a Transformer on the Wikitext-103 benchmark and report perplexity (PPL) on test. For image classification, we train a EfficientNet-B3 on the ImageNet-1k benchmark and report top-1 accuracy on validation and use our re-implementation of EfficientNet-B3. The original implementation of Tan *et al.* [4] achieves an uncompressed Top-1 accuracy of 81.9%. For both settings, we report model size in megabyte (MB) and the compression ratio compared to the original model.

Polino, A., Pascanu, R., & Alistarh, D. (2018). Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668*.

“We used Quant-Noise to compress Facebook AI’s state-of-the-art RoBERTa Base model from 480 MB to 14 MB while achieving 82.5 percent on MNLI, compared with 84.8 percent for the original model.”

Part 3: Production Deployment

- Lecture
  - Model Selection
  - Post-Training Optimization
  - Product Quantization
  - Knowledge Distillation
  - Model Code Efficiency
  - Model Serving
  - Building the Application
- Lab
  - Exporting the Model
  - Hosting the Model
  - Server Performance
  - Using the Model

KNOWLEDGE DISTILLATION

The idea

Distilling the Knowledge in a Neural Network

Geoffrey Hinton[*] [†]
Google Inc.
Mountain View
geoffhinton@google.com

Oriol Vinyals[†]
Google Inc.
Mountain View
vinyals@google.com

Jeff Dean
Google Inc.
Mountain View
jeff@google.com

Abstract

A very simple way to improve the performance of almost any machine learning algorithm is to train many different models on the same data and then to average their predictions [3]. Unfortunately, making predictions using a whole ensemble of models is cumbersome and may be too computationally expensive to allow deployment to a large number of users, especially if the individual models are large neural nets. Caruana and his collaborators [1] have shown that it is possible to compress the knowledge in an ensemble into a single model which is much easier to deploy and we develop this approach further using a different compression technique. We achieve some surprising results on MNIST and we show that we can significantly improve the acoustic model of a heavily used commercial system by distilling the knowledge in an ensemble of models into a single model. We also introduce a new type of ensemble composed of one or more full models and many specialist models which learn to distinguish fine-grained classes that the full models confuse. Unlike a mixture of experts, these specialist models can be trained rapidly and in parallel.

KNOWLEDGE DISTILLATION

DistillBERT

Table 1: **DistilBERT retains 97% of BERT performance.** Comparison on the dev sets of the GLUE benchmark. ELMo results as reported by the authors. BERT and DistilBERT results are the medians of 5 runs with different seeds.

Model	**Score**	CoLA	MNLI	MRPC	QNLI	QQP	RTE	SST-2	STS-B	WNLI
ELMo	68.7	44.1	68.6	76.6	71.1	86.2	53.4	91.5	70.4	56.3
BERT-base	79.5	56.3	86.7	88.6	91.8	89.6	69.3	92.7	89.0	53.5
DistilBERT	77.0	51.3	82.2	87.5	89.2	88.5	59.9	91.3	86.9	56.3

Table 2: **DistilBERT yields to comparable performance on downstream tasks.** Comparison on downstream tasks: IMDb (test accuracy) and SQuAD 1.1 (EM/F1 on dev set). D: with a second step of distillation during fine-tuning.

Model	IMDb (acc.)	SQuAD (EM/F1)
BERT-base	93.46	81.2/88.5
DistilBERT	92.82	77.7/85.8
DistilBERT (D)	-	79.1/86.9

Table 3: **DistilBERT is significantly smaller while being constantly faster.** Inference time of a full pass of GLUE task STS-B (sentiment analysis) on CPU with a batch size of 1.

Model	# param. (Millions)	Inf. time (seconds)
ELMo	180	895
BERT-base	110	668
DistilBERT	66	410

Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. arXiv preprint arXiv:1910.01108.

Part 3: Production Deployment

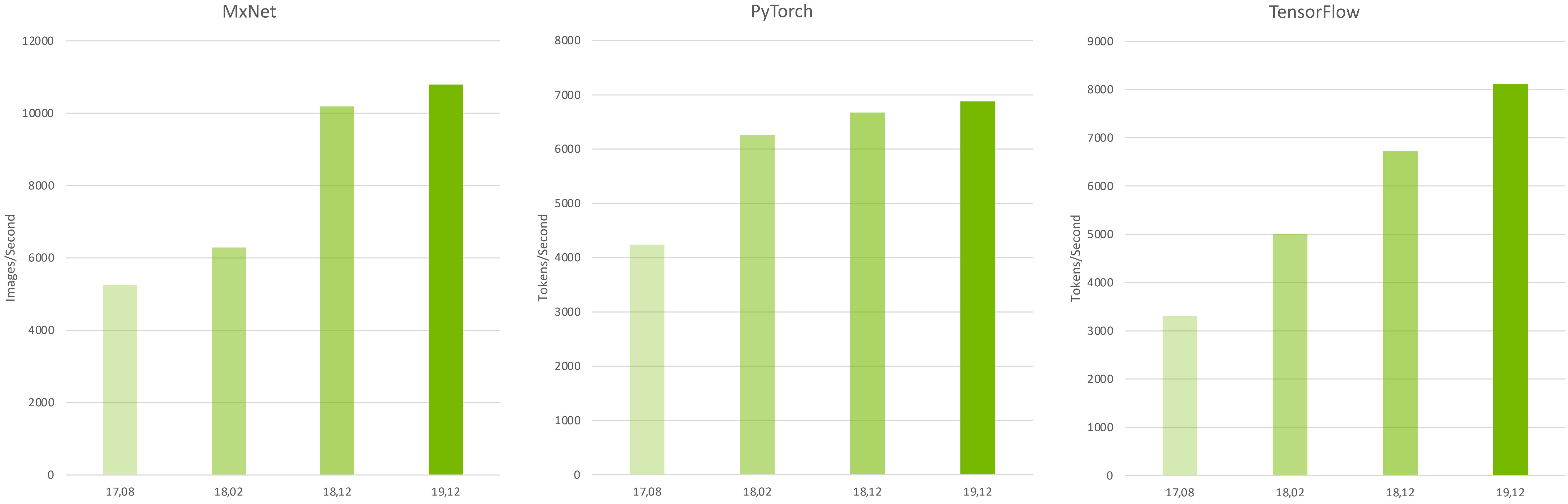
- Lecture
  - Model Selection
  - Post-Training Optimization
  - Product Quantization
  - Knowledge Distillation
  - Model Code Efficiency
  - Model Serving
  - Building the Application
- Lab
  - Exporting the Model
  - Hosting the Model
  - Server Performance
  - Using the Model

NOT ALL MODELS HAVE THE SAME CODE QUALITY

COMPUTE MATTERS

But so does code quality

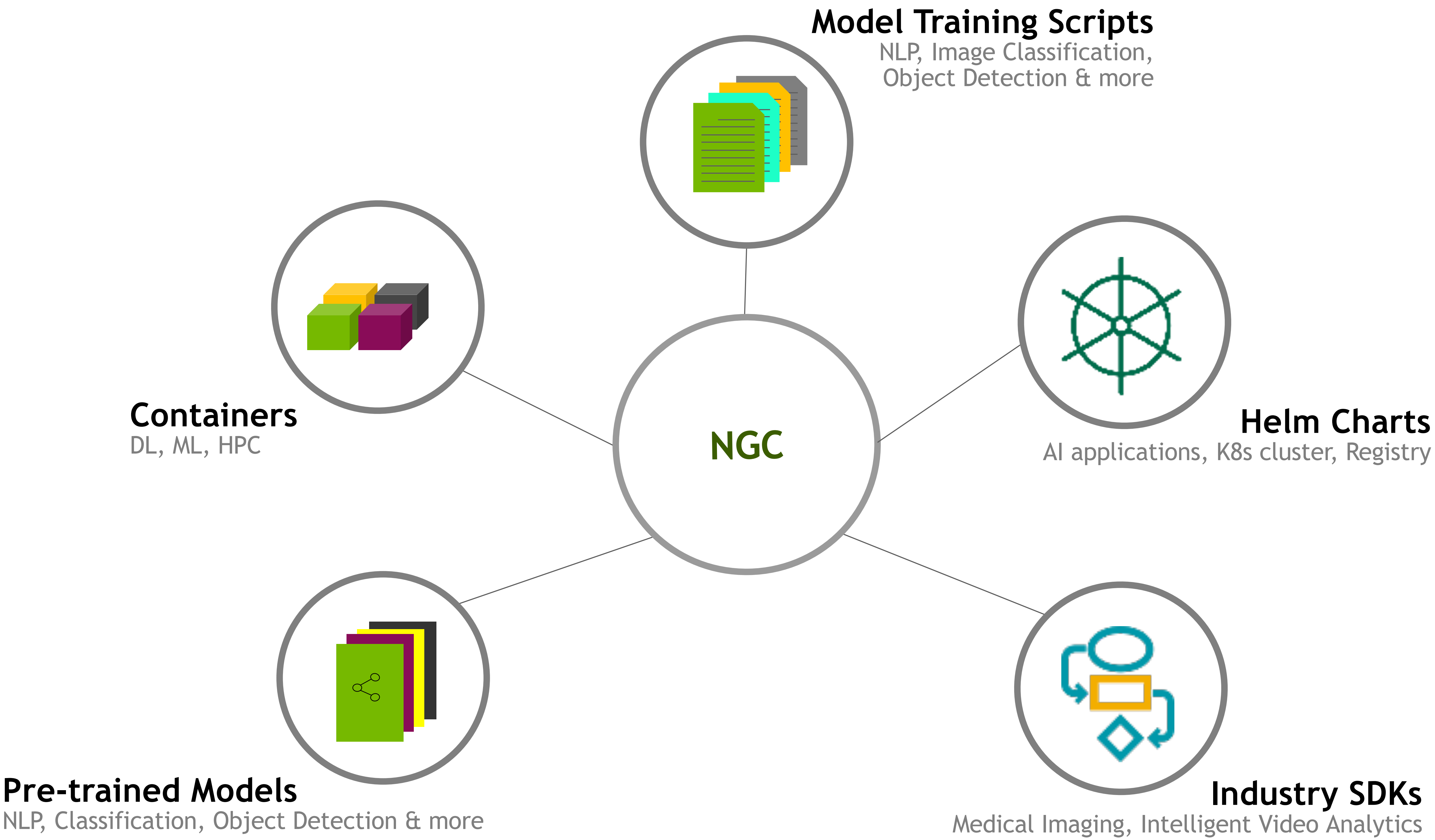
Monthly DL Framework Updates & Optimizations Drive Performance



ResNet-50 v1.5 Training | 8x V100 | DGX-1

NGC: GPU-OPTIMIZED SOFTWARE HUB

Simplifying DL, ML and HPC Workflows



PRETRAINED MODELS & MODEL SCRIPTS

Build AI Solutions Faster

PRE-TRAINED MODELS

- Deploy AI quickly with models for industry specific use cases
- Covers everything from speech to object detection
- Integrate into existing workflows with code samples
- Easily use transfer learning to adapt to your bespoke use case

MODEL SCRIPTS

- Reference neural network architectures across all domains and popular frameworks with latest SOTA
- Jupyter notebook starter kits

Healthcare (~30 models)	BioBERT (NLP), Clara (Computer Vision)
Manufacturing (~25 Models)	Object Detection, Image Classification
Retail (~25 models)	BERT, Transformer
70 TensorRT Plans	Classification/Segmentation for v5, v6, v7
Natural Language Processing	25 Bert Configurations
Recommendation Engines	Neural Collaborative Filtering, VAE
Speech	Jasper, Tacotron, WaveGlow
Translation	GNMT

THIS APPLIES NOT ONLY TO TRAINING BUT INFERENCE AS WELL

CODE QUALITY IS KEY

Dramatic differences in model performance

3-layer BERT with 128 sequence length

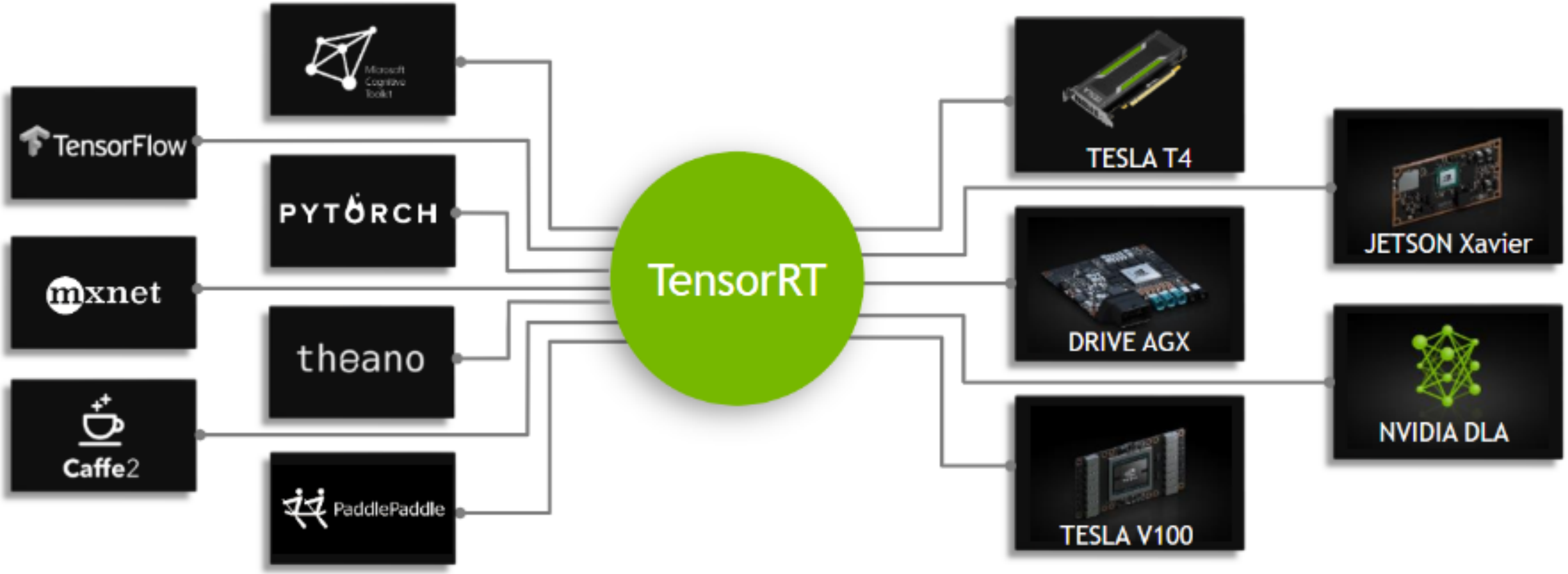
		Batch size	Inference on	Throughput (Query per second)	Latency (milliseconds)
CPU	Original 3-layer BERT	1	Azure Standard F16s_v2 (CPU)	6	157
	ONNX Model	1	Azure Standard F16s_v2 (CPU) **with ONNX Runtime**	111	9
GPU	Original 3-layer BERT	4	Azure NV6 GPU VM	200	20
	ONNX Model	4	Azure NV6 GPU VM **with ONNX Runtime**	500	8
	ONNX Model	64	Azure NC6S_v3 GPU VM **with ONNX Runtime + System Optimization** (Tensor Core with mixed precision, Same Accuracy)	10667	6

https://cloudblogs.microsoft.com/opensource/2020/01/21/microsoft-onnx-open-source-optimizations-transformer-inference-gpu-cpu

OPTIMIZING INFERENCE WITH TENSORRT

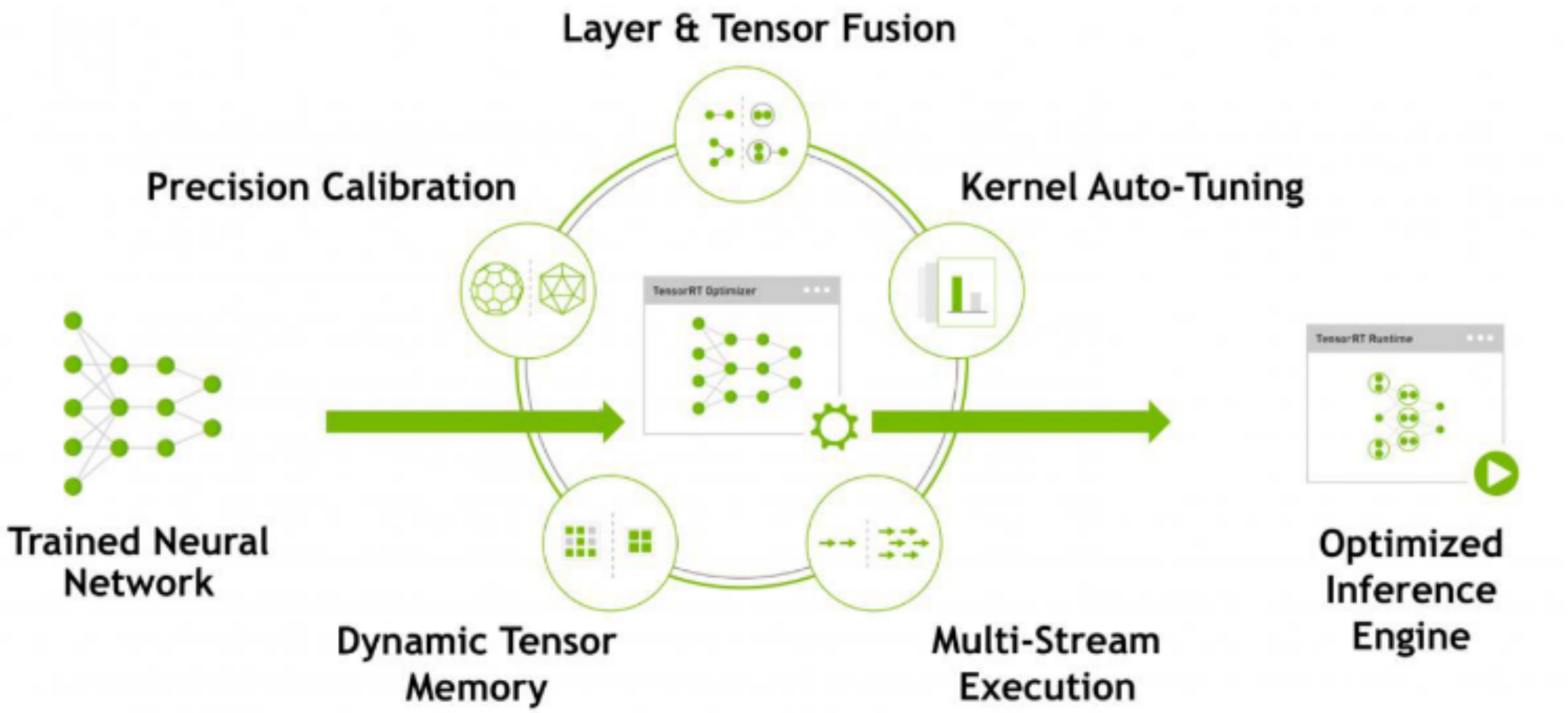
NVIDIA TENSORRT

From Every Framework, Optimized For Each Target Platform



TENSORRT

Optimizations



developer.nvidia.com/tensorrt

TensorRT ONNX PARSER

High-Performance Inference for ONNX Models

Optimize and deploy models from ONNX-supported frameworks to production

Apply TensorRT optimizations to any ONNX framework (Caffe 2, Microsoft Cognitive Toolkit, MxNet & PyTorch)

Import TensorFlow and Keras through converters (tf2onnx, keras2onnx)

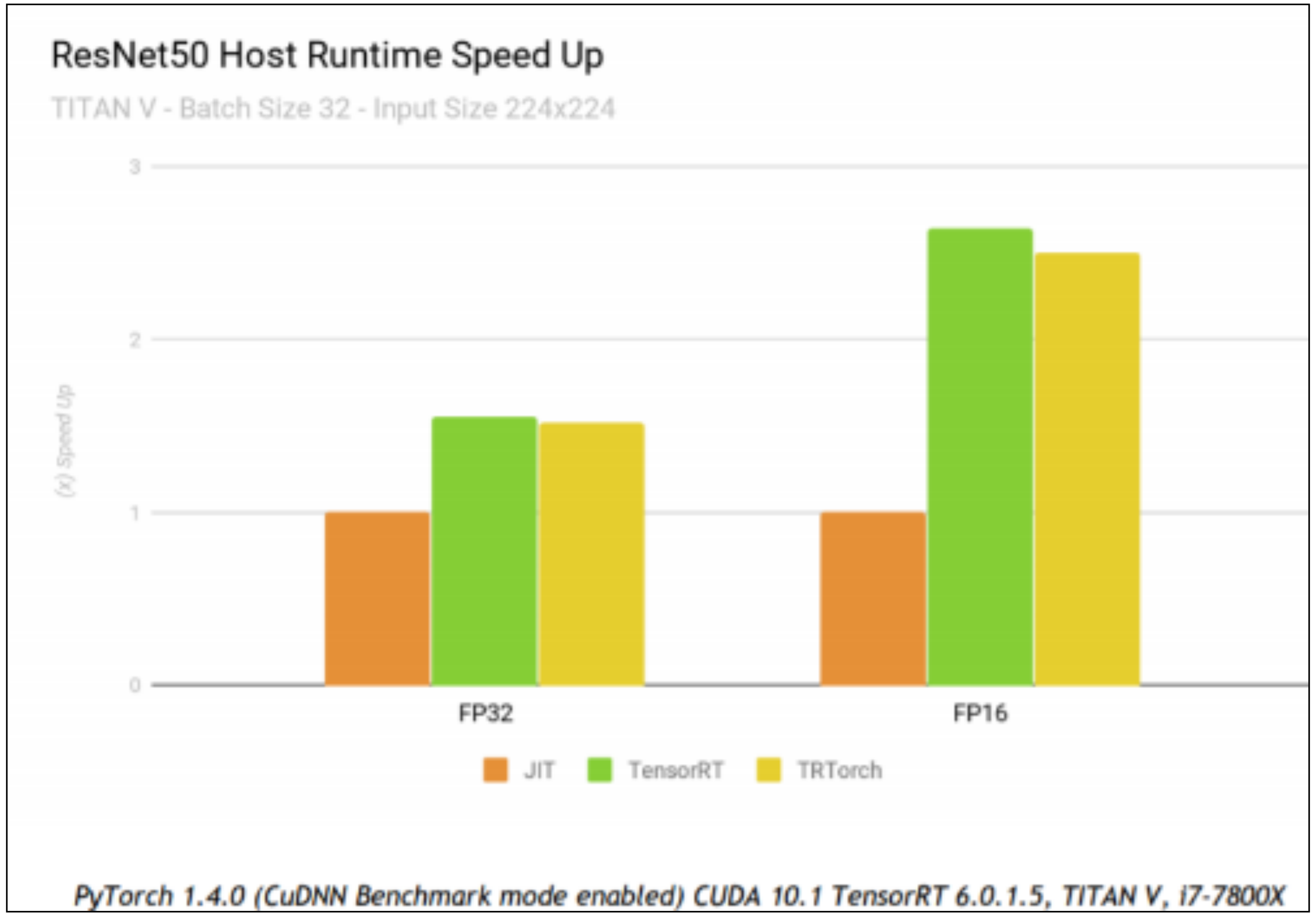
Use with C++ and Python apps

20+ New Ops in TensorRT 7

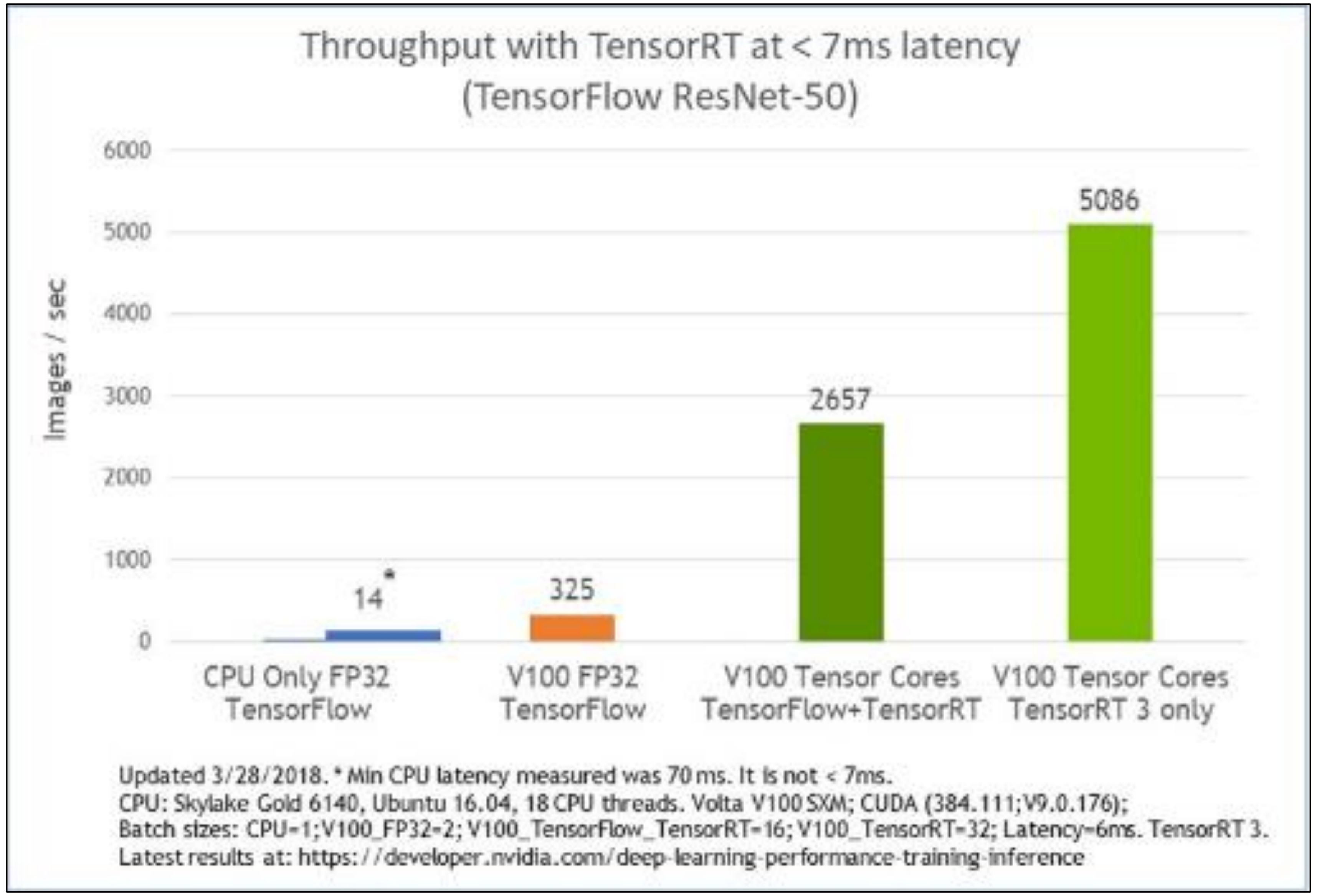
Support for Opset 11 (See List of Supported Ops)

TENSORRT

Tight integration with DL frameworks



Pytorch -> TRTorch



TensorFlow -> TF-TRT

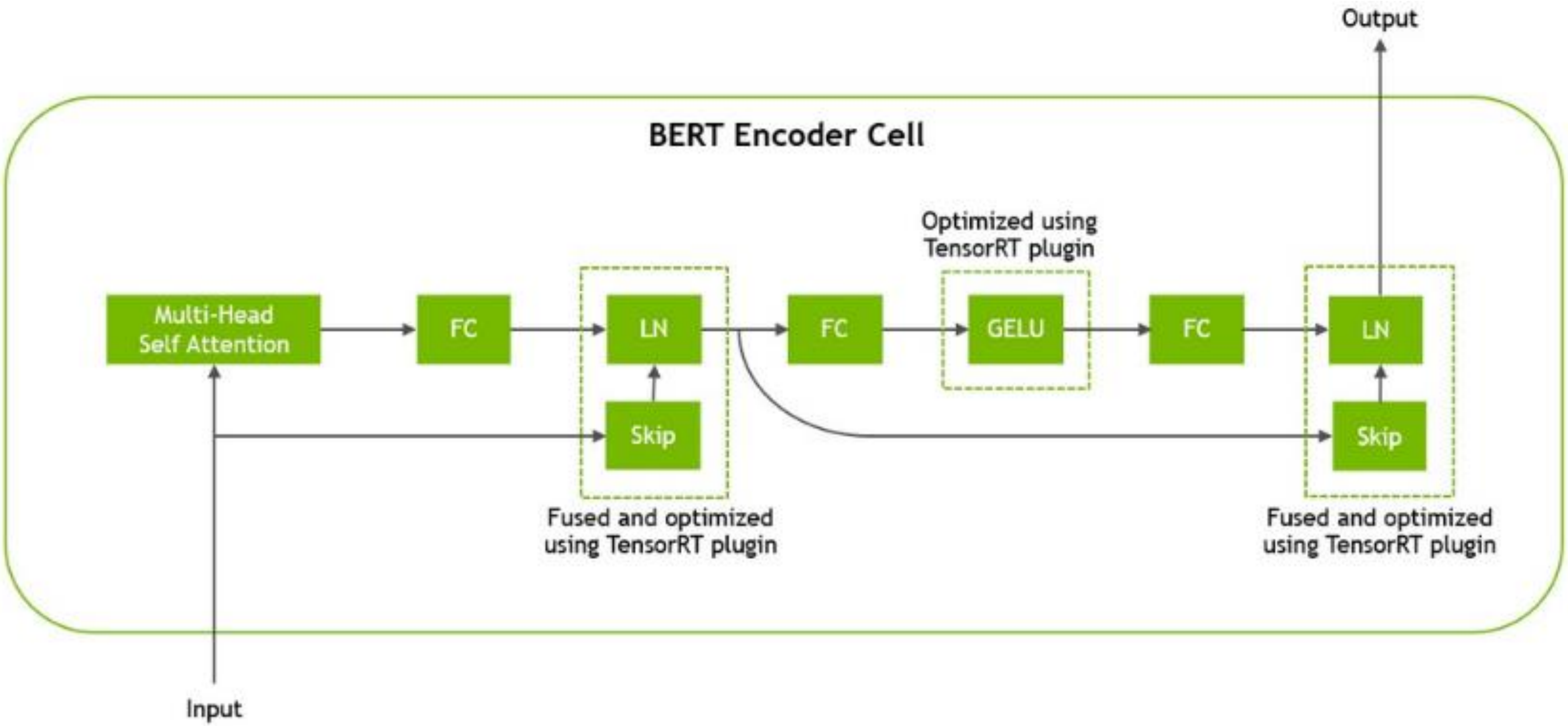
WIDELY ADOPTED

Accelerating most demanding applications

IMPACT ON NLP

TENSORRT

BERT Encoder optimizations



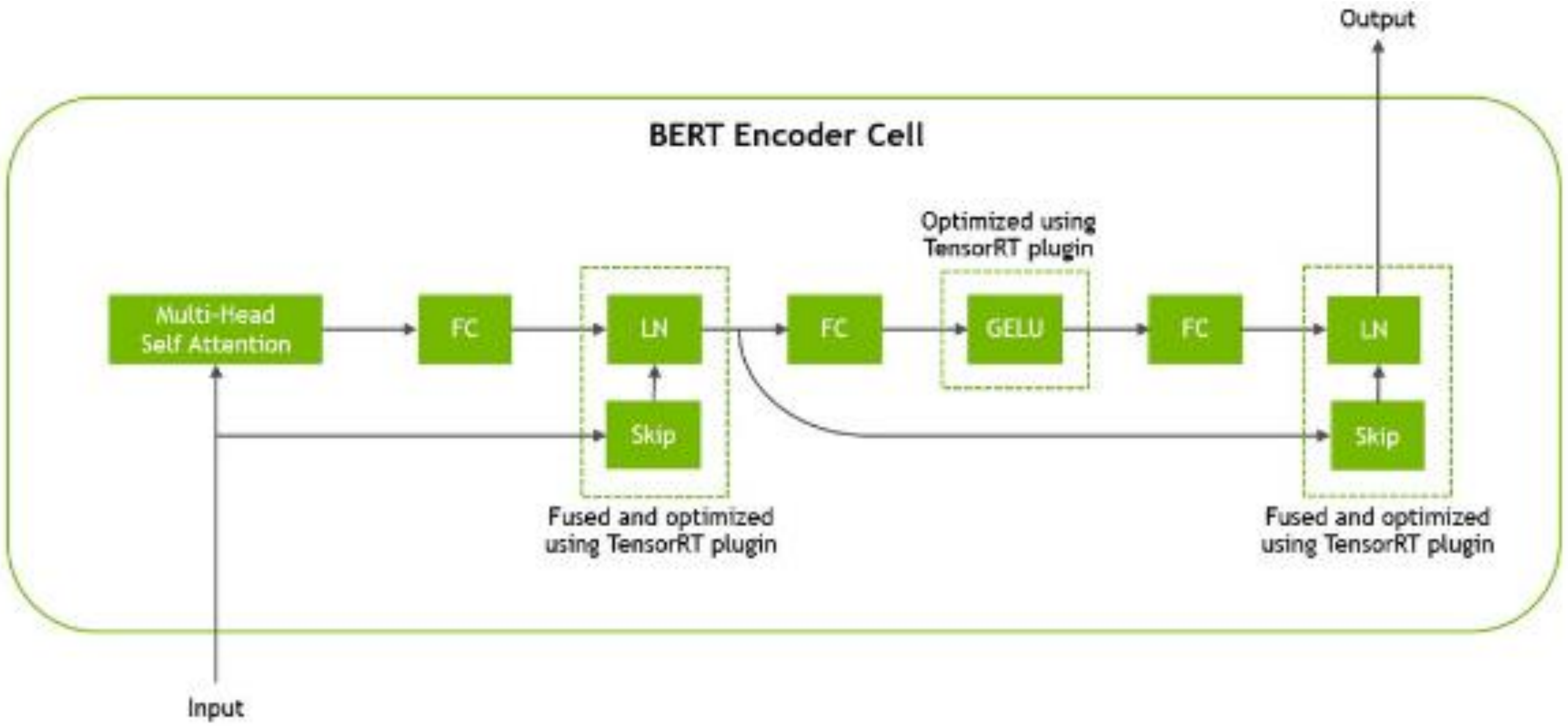
CUSTOM PLUGINS

Optimized GeLU as well as skip and layer-normalization operations

- Naïve implementation would require a large number of TensorRT elementary layers
- For k layers, the naïve implementation would require k-1 memory roundtrips
- The skip and layer-normalization(LN) layers occur twice per Transformer layer and are fused in a single kernel

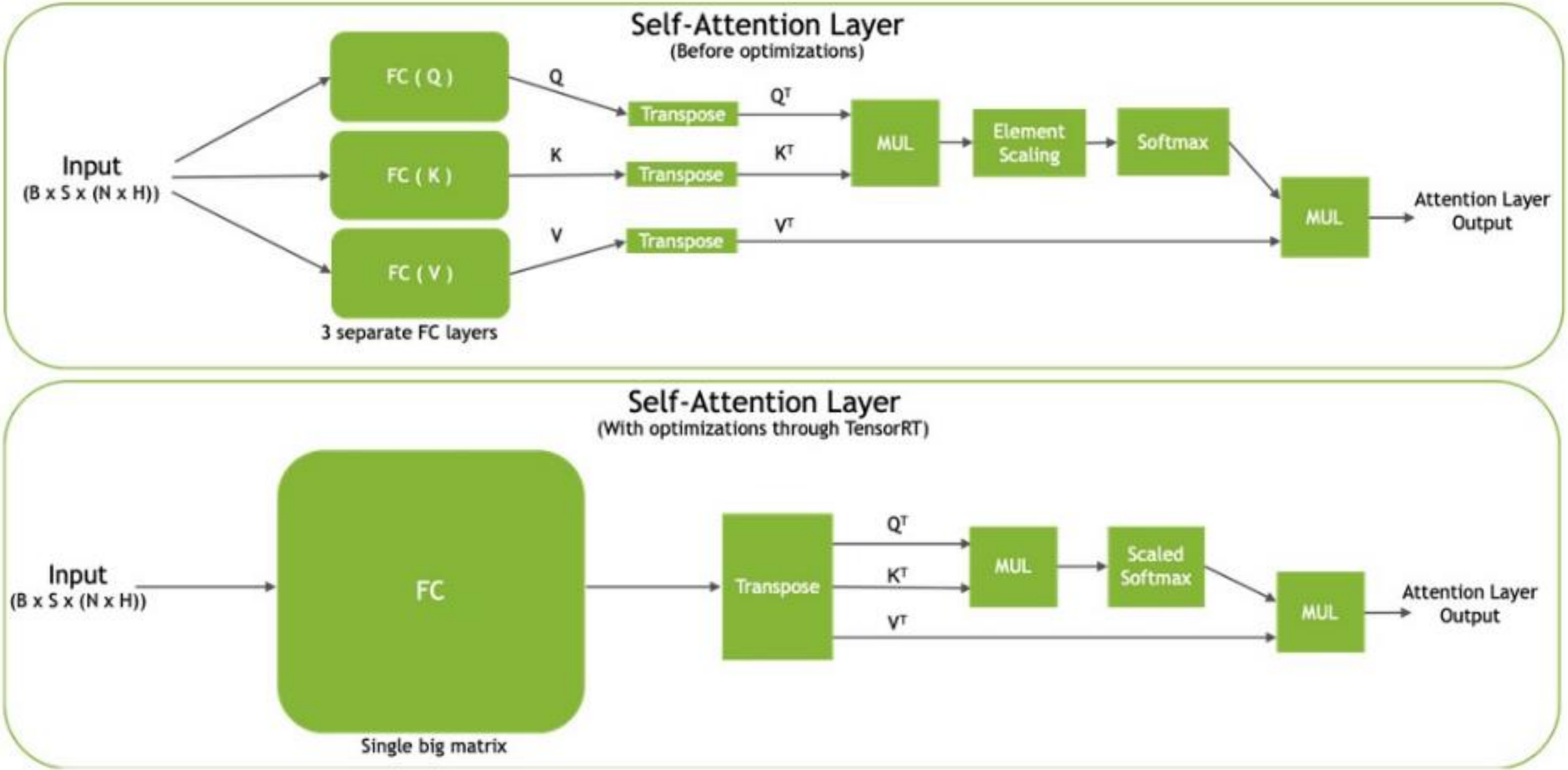
gelu(x) = a * x * (1 + tanh(b * (x + c * x^3)))

```
Result = x^3
Result = c * Result
Result = x + Result
Result = b * Result
Result = tanh(Result)
Result = x * Result
Result = a * Result
```



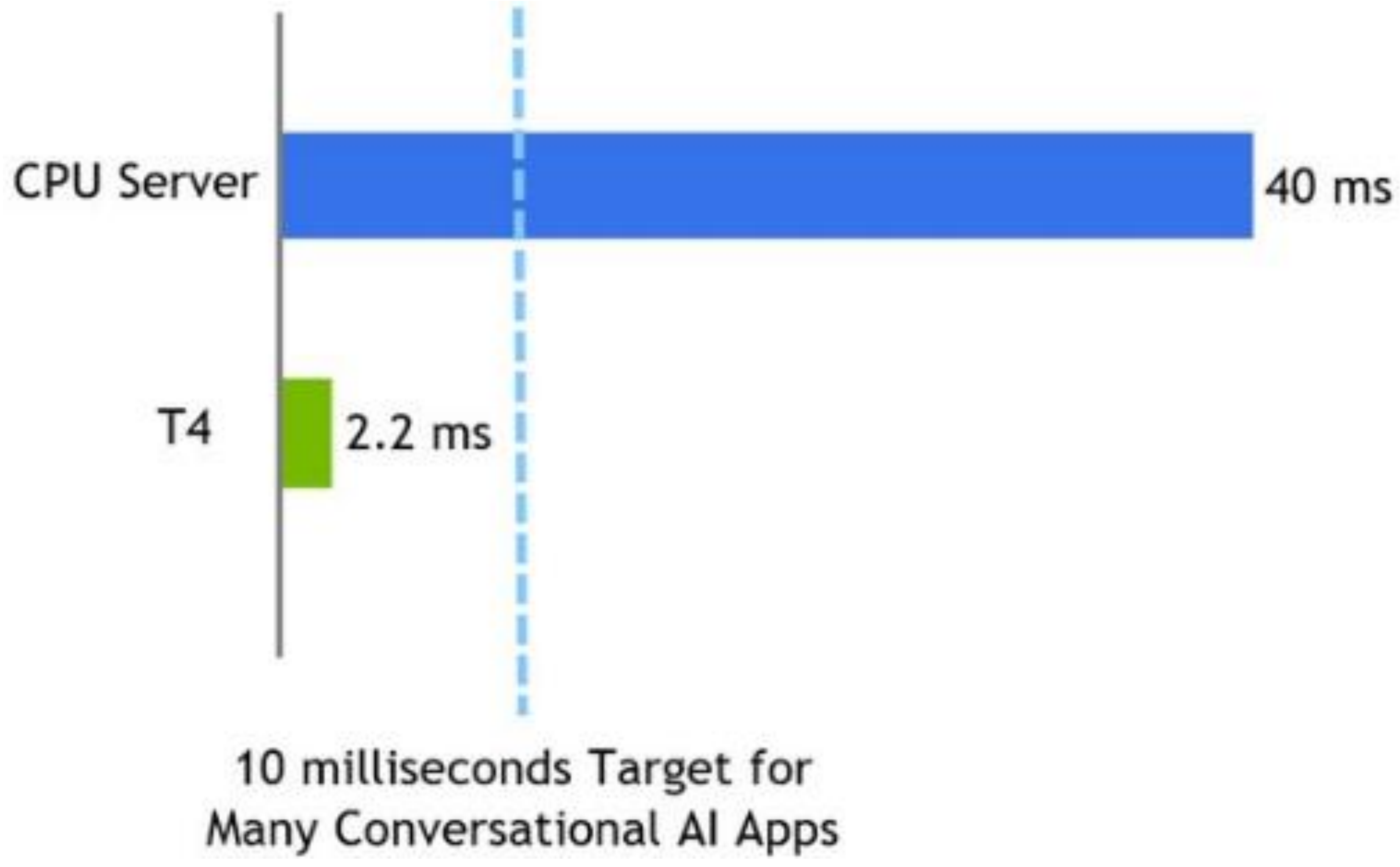
CUSTOM PLUGINS

Self-attention layer



IMPLICATIONS

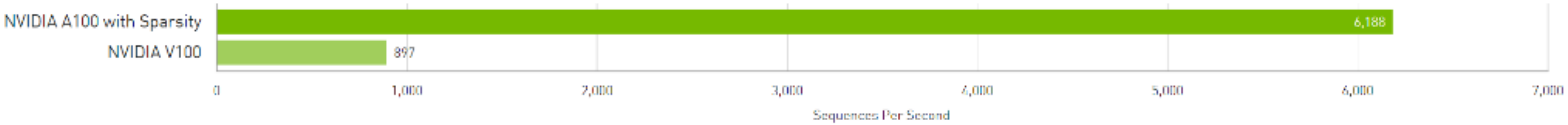
Significant impact on latency and throughput (batch 1)



Using a Tesla T4 GPU, BERT optimized with TensorRT can perform inference in 2.2 ms for a QA task similar to available in SQuAD with batch size =1 and sequence length = 128.

IMPLICATIONS

Significant impact on latency and throughput



DGX A100 server w/ 1x NVIDIA A100 with 7 MIG instances of 1g.5gb | Batch Size = 94 | Precision: INT8 | Sequence Length = 128
DGX-1 server w/ 1x NVIDIA V100 | TensorRT 7.1 | Batch Size = 256 | Precision: Mixed | Sequence Length = 128

BEYOND BERT

FASTER TRANSFORMER

Designed for training and inference speed

- Encoder:
  - 1.5x compare to TensorFlow with XLA on FP16
- Decoder on NVIDIA Tesla T4
  - 2.5x speedup for batch size 1 (online translating scheme)
  - 2x speedup for large batch size in FP16
- Decoding on NVIDIA Tesla T4
  - 7x speedup for batch size 1 and beam width 4 (online translating scheme)
  - 2x speedup for large batch size in FP16.
- Decoding on NVIDIA Tesla V100
  - 6x speedup for batch size 1 and beam width 4 (online translating scheme)
  - 3x speedup for large batch size in FP16.

https://github.com/NVIDIA/DeepLearningExamples/tree/master/FasterTransformer#feature-support-matrix

CONSIDER USING TENSORRT

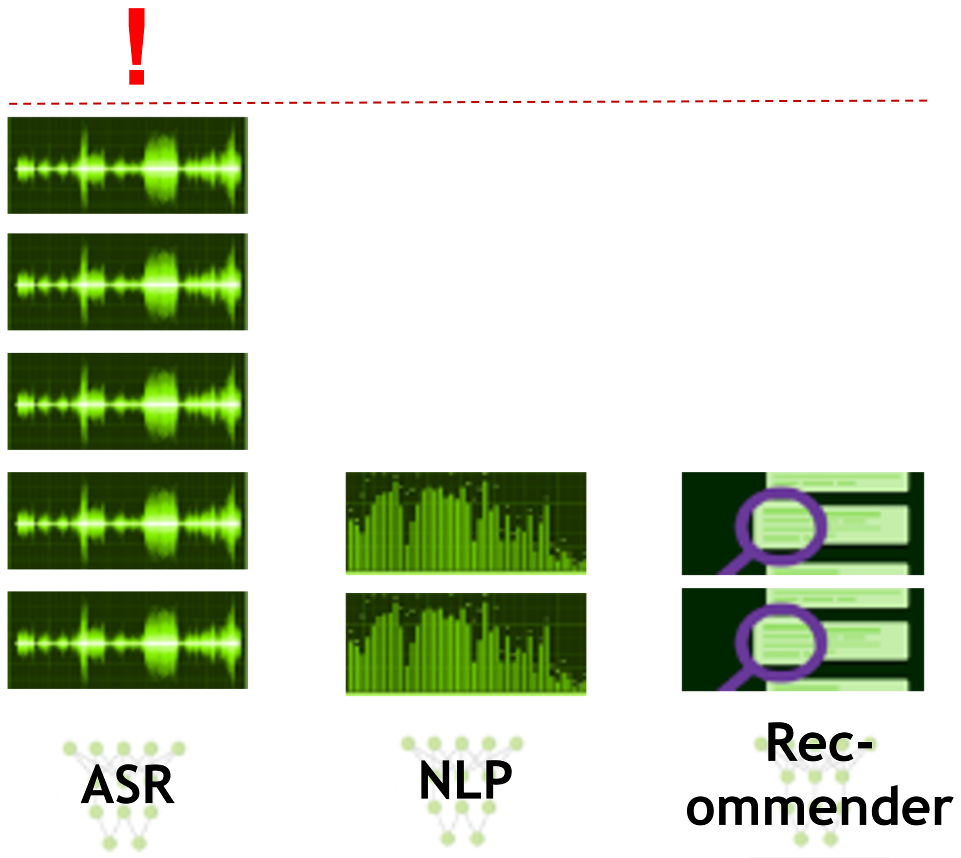
Part 3: Production Deployment

- Lecture
  - Model Selection
  - Post-Training Optimization
  - Product Quantization
  - Knowledge Distillation
  - Model Code Efficiency
  - Model Serving
  - Building the Application
- Lab
  - Exporting the Model
  - Hosting the Model
  - Server Performance
  - Using the Model

INEFFICIENCY LIMITS INNOVATION

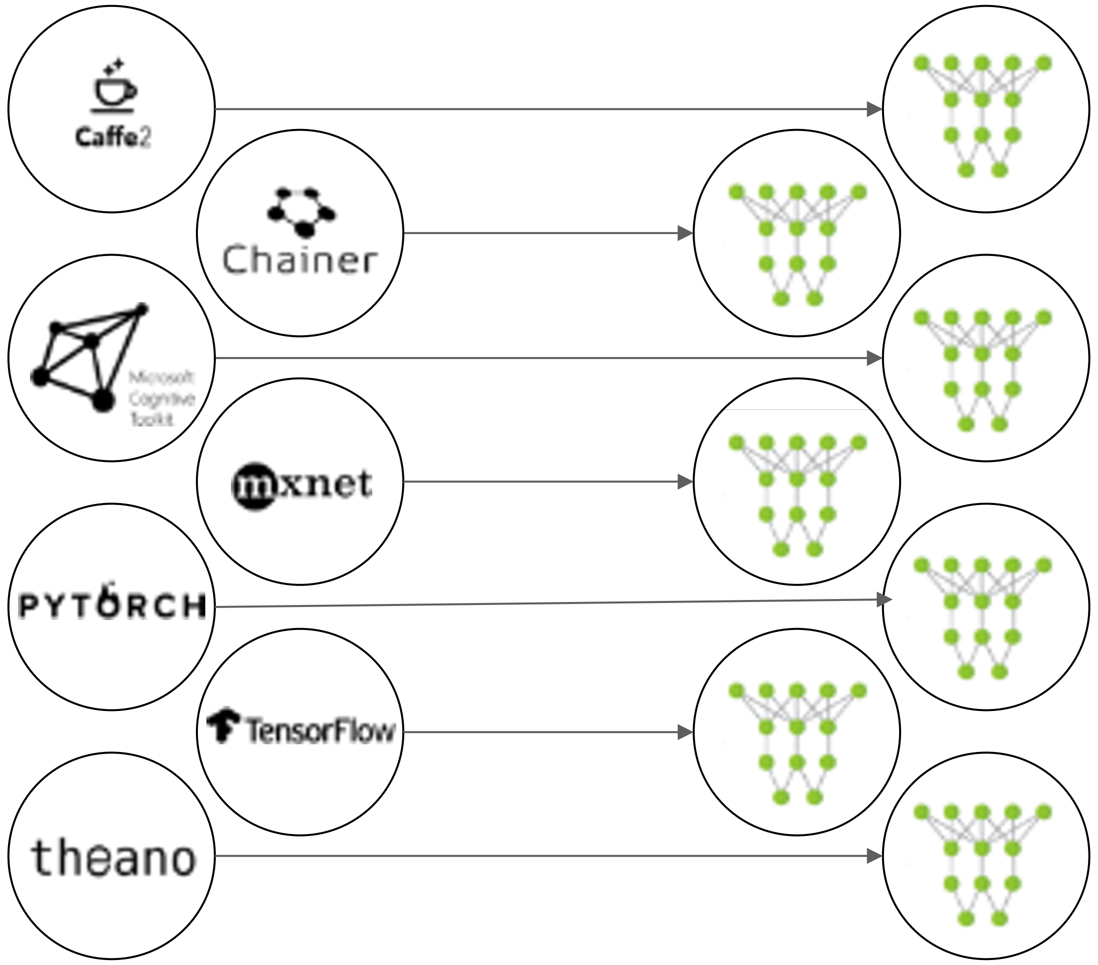
Difficulties with deploying data center inference

Single Model Only



Some systems are overused while others are underutilized

Single Framework Only



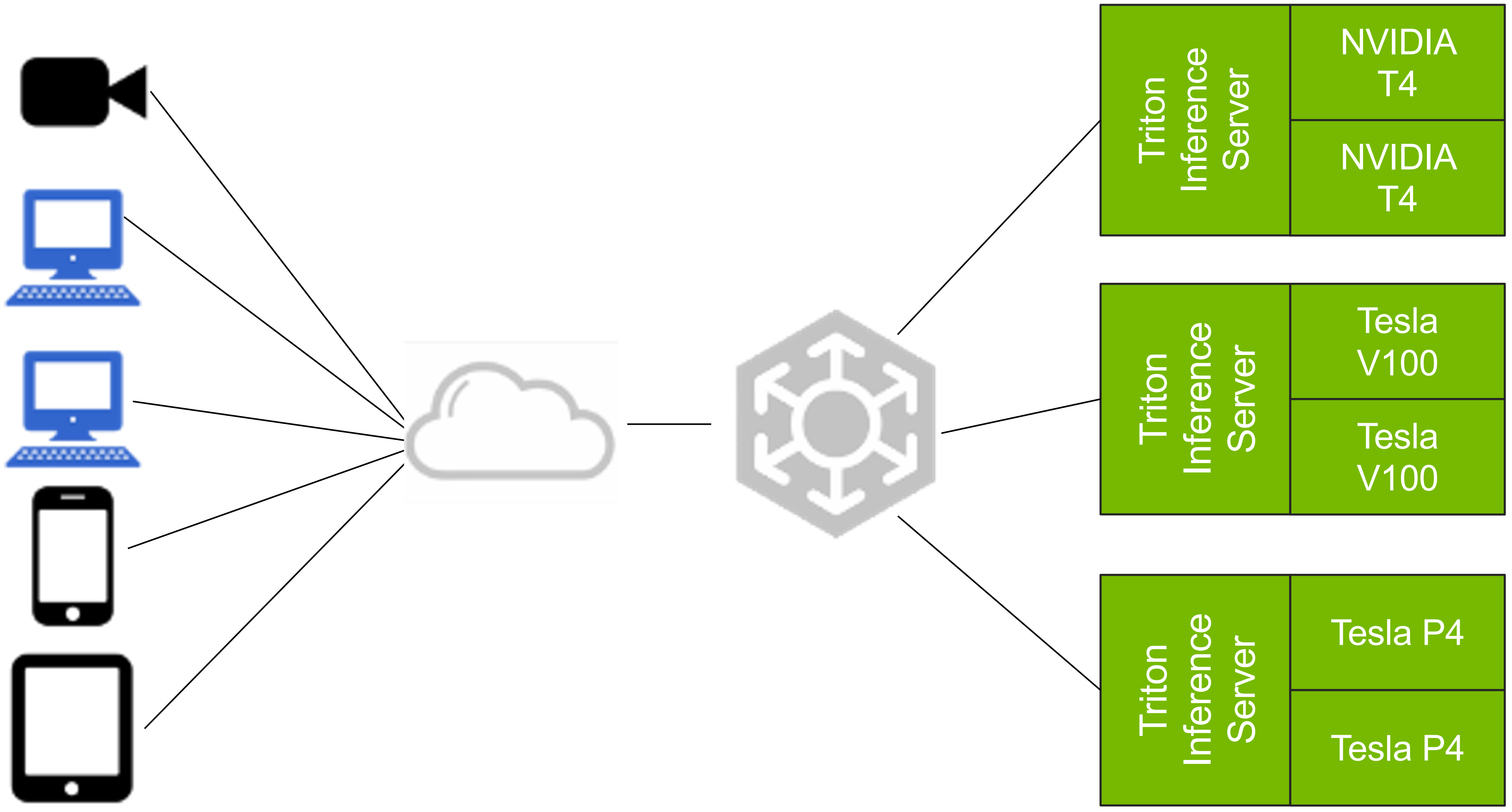
Solutions can only support models from one framework

Custom Development

Developers need to reinvent the plumbing for every application

NVIDIA TRITON INFERENCE SERVER

Production data center inference server



Maximize real-time inference performance of GPUs

Quickly deploy and manage multiple models per GPU per node

Easily scale to heterogeneous GPUs and multi GPU nodes

Integrates with orchestration systems and auto-scalers via latency and health metrics

Now open source for thorough customization and integration

FEATURES

Concurrent Model Execution
Multiple models (or multiple instances of same model) may execute on GPU simultaneously

CPU Model Inference Execution
Framework native models can execute inference requests on the CPU

Metrics
Utilization, count, memory, and latency

Custom Backend
Custom backend allows the user more flexibility by providing their own implementation of an execution engine through the use of a shared library

Model Ensemble
Pipeline of one or more models and the connection of input and output tensors between those models (can be used with custom backend)

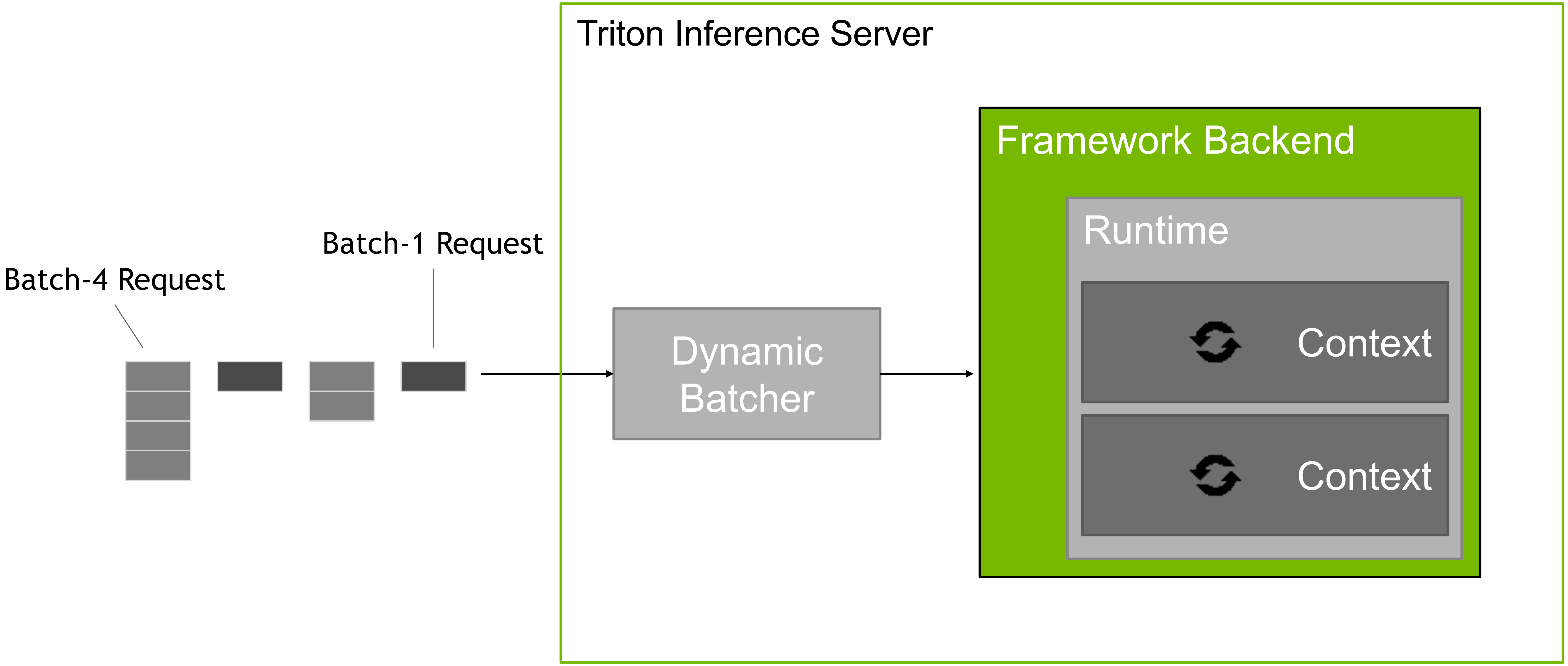
Dynamic Batching
Inference requests can be batched up by the inference server to 1) the model-allowed maximum or 2) the user-defined latency SLA

Multiple Model Format Support
PyTorch JIT (.pt)
TensorFlow GraphDef/SavedModel
TensorFlow and TensorRT GraphDef
ONNX graph (ONNX Runtime)
TensorRT Plans
Caffe2 NetDef (ONNX import path)

CMake build
Build the inference server from source making it more portable to multiple OSes and removing the build dependency on Docker

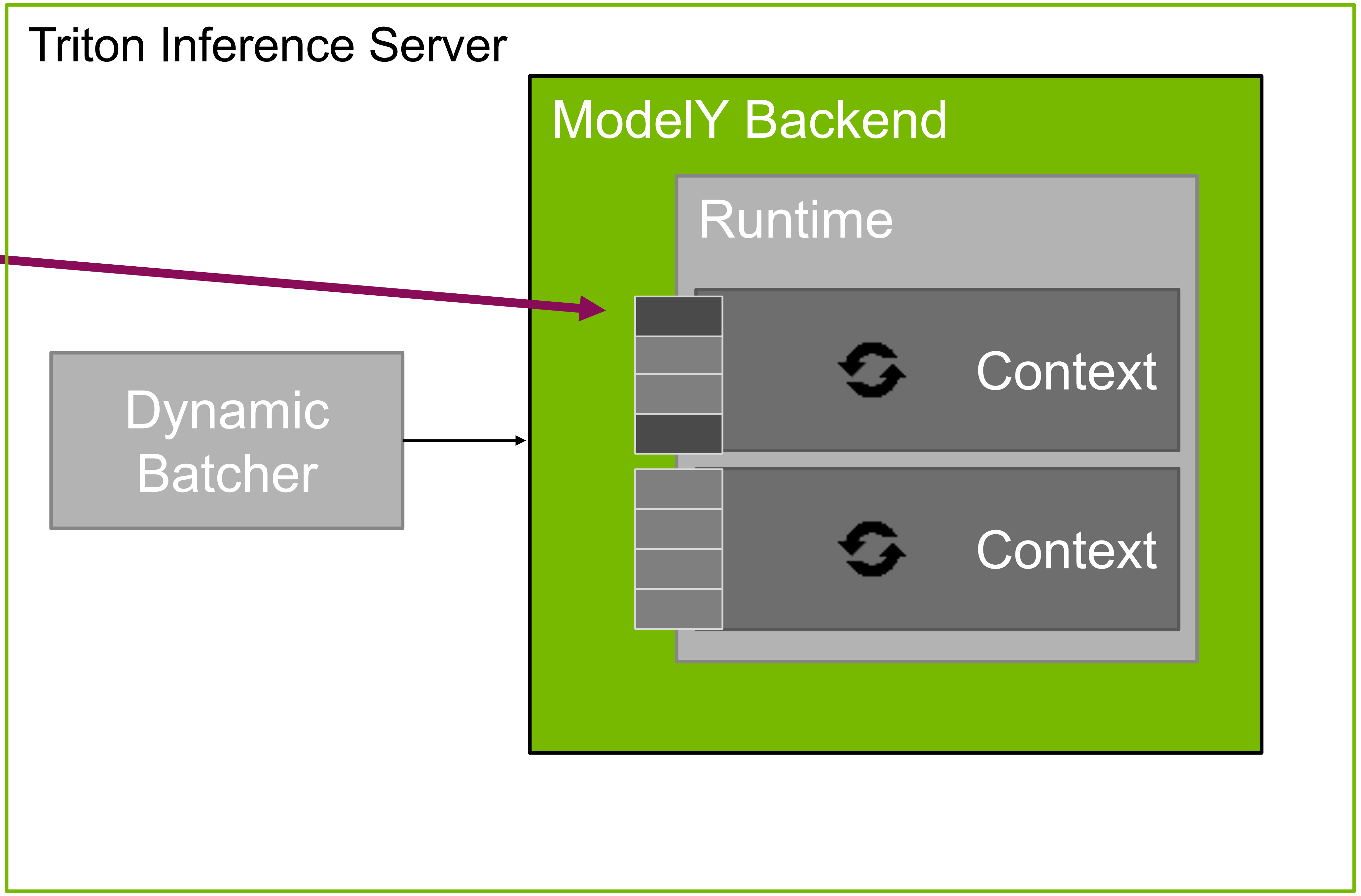
Streaming API
Built-in support for audio streaming input e.g. for speech recognition

DYNAMIC BATCHING SCHEDULER



DYNAMIC BATCHING SCHEDULER

Grouping requests into a single "batch" increases overall GPU throughput



Preferred batch size and wait time are configuration options.

Assume 4 gives best utilization in this example.

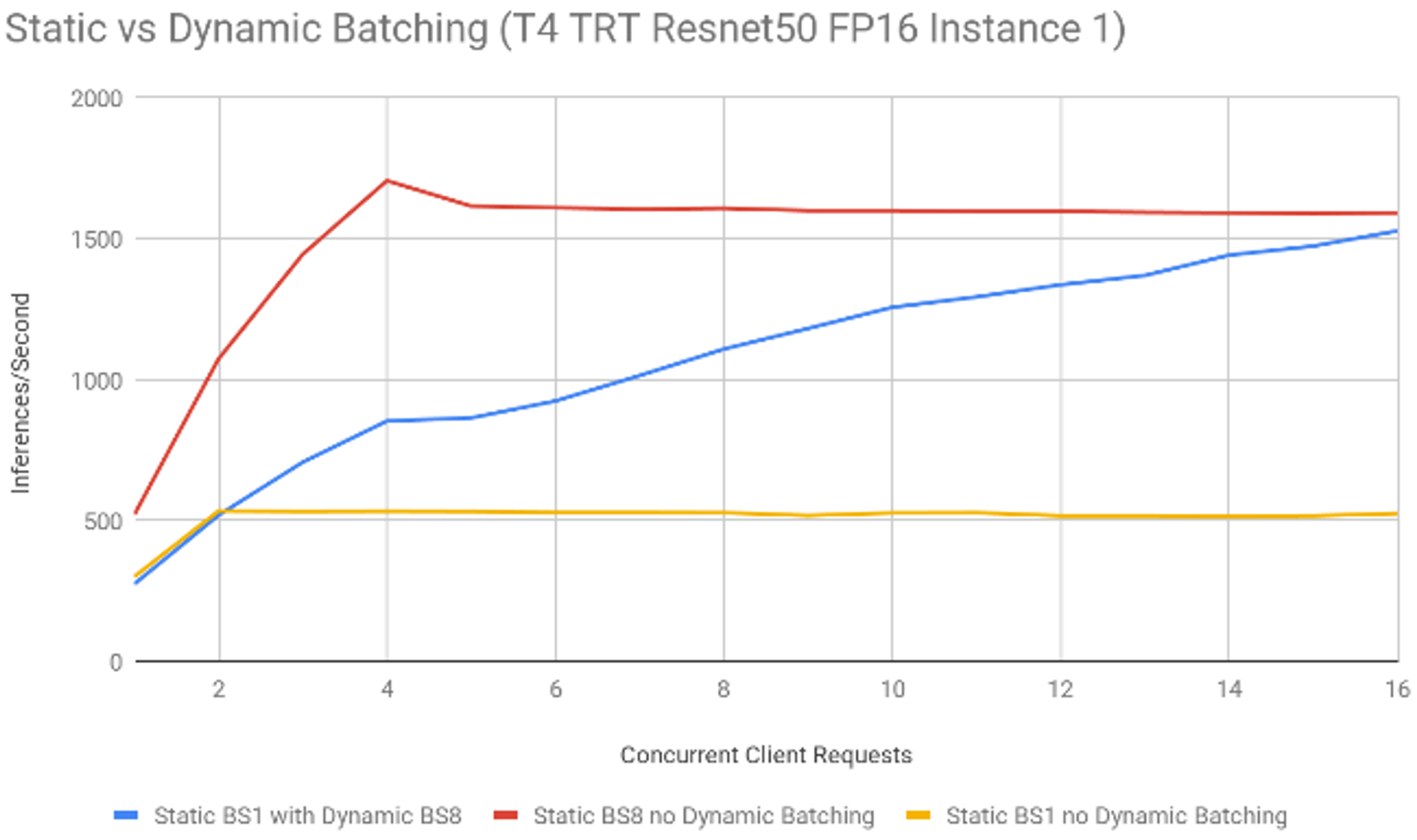
DYNAMIC BATCHING

2.5X Faster Inferences/Second at a 50ms End-to-End Server Latency Threshold

Triton Inference Server groups inference requests based on customer defined metrics for optimal performance

Customer defines 1) batch size (required) and 2) latency requirements (optional)

Example: No dynamic batching (batch size 1 & 8) vs dynamic batching



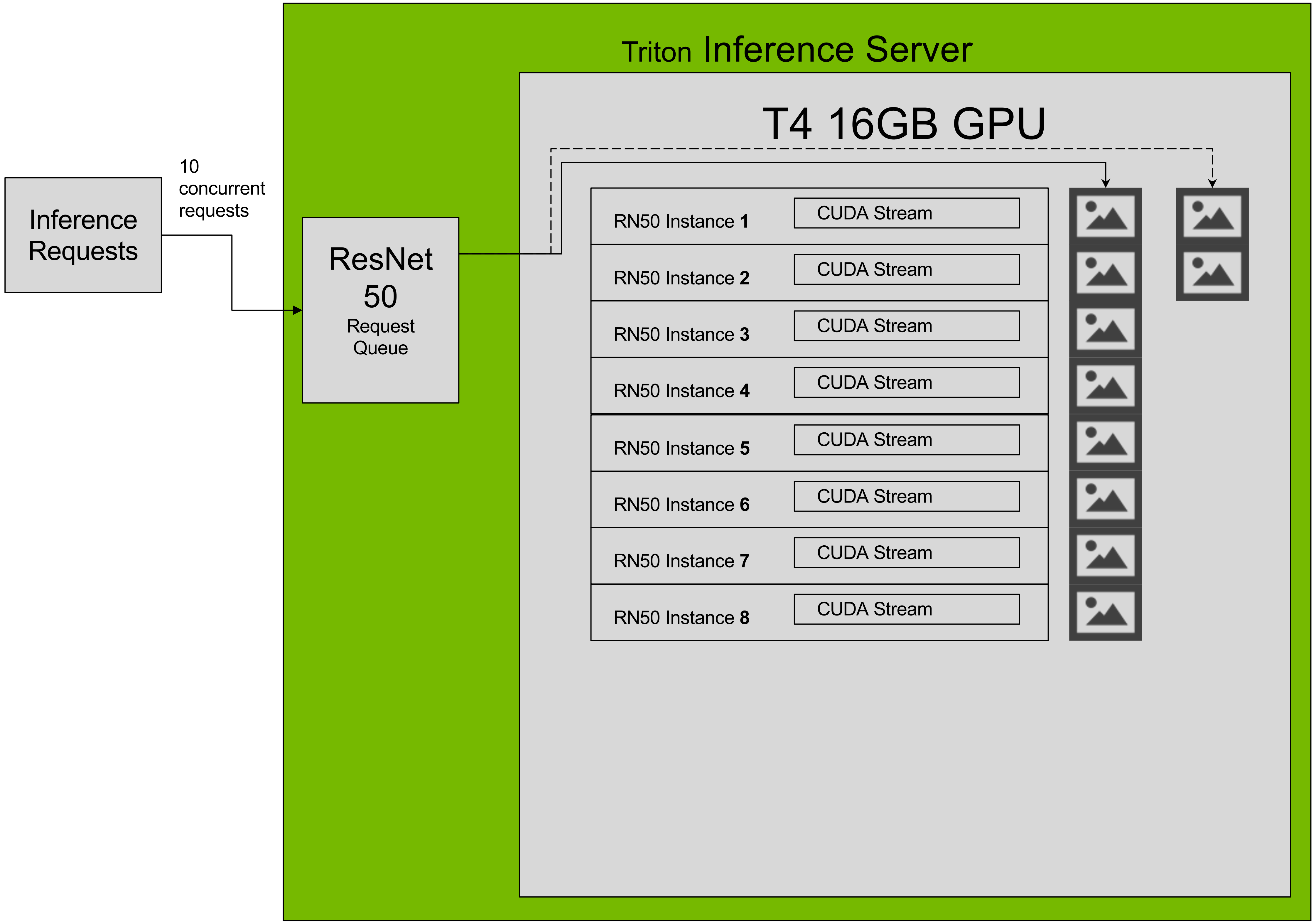
CONCURRENT MODEL EXECUTION - RESNET 50

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

<u>Common Scenario 1</u>

One API using <u>multiple</u> copies of the <u>same</u> model on a GPU

Example: 8 instances of TRT FP16 ResNet50 (each model takes 2 GB GPU memory) are loaded onto the GPU and can run concurrently on a 16GB T4 GPU.
10 concurrent inference requests happen: each model instance fulfills one request simultaneously and 2 are queued in the per-model scheduler queues in Triton Inference Server to execute after the 8 requests finish. With this configuration, 2680 inferences per second at 152 ms with batch size 8 on each inference server instance is achieved.



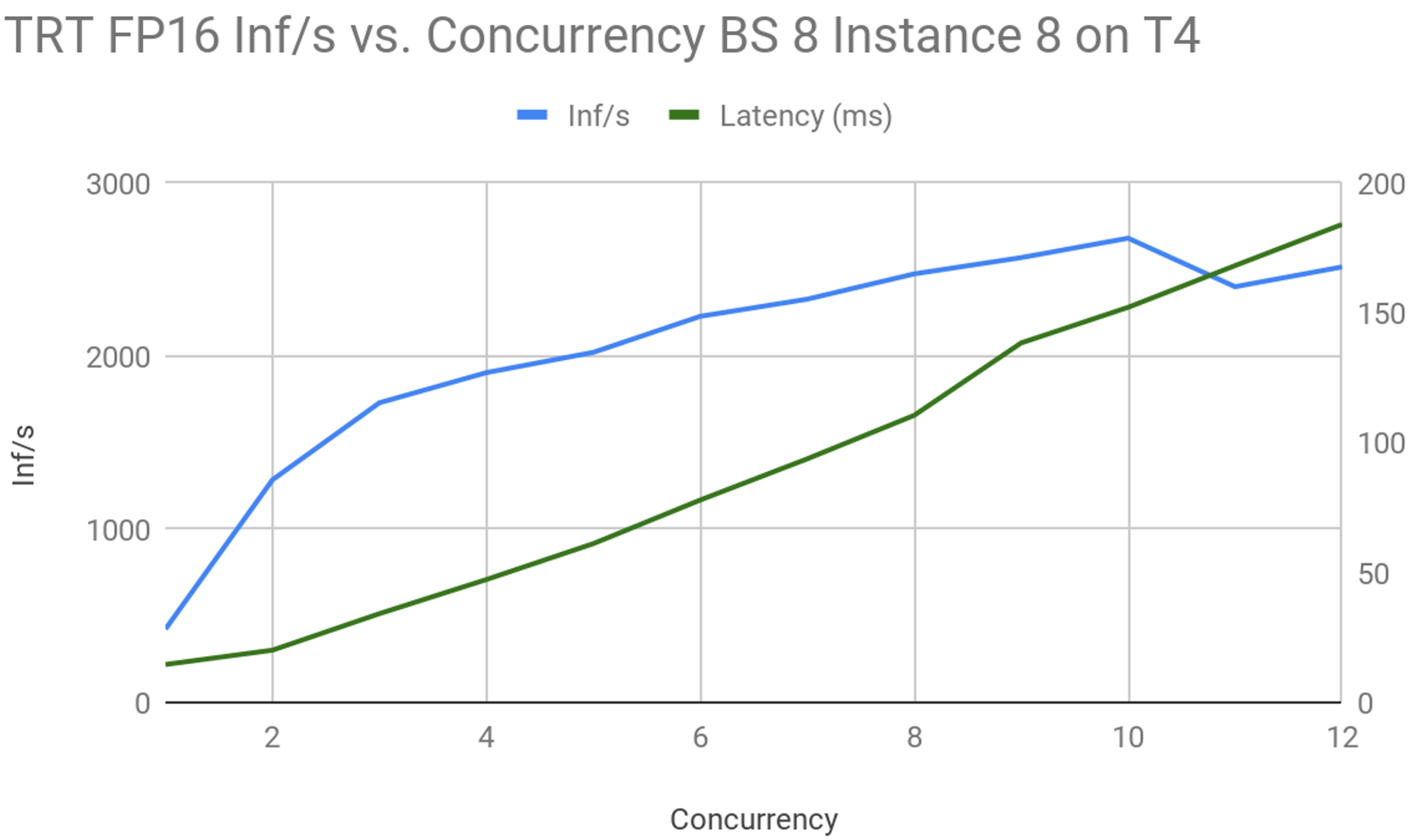
CONCURRENT MODEL EXECUTION - RESNET 50

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

Common Scenario 1

One API using multiple copies of the same model on a GPU

Example: 8 instances of TRT FP16 ResNet50 (each model takes 2 GB GPU memory) are loaded onto the GPU and can run concurrently on a 16GB T4 GPU.
10 concurrent inference requests happen: each model instance fulfills one request simultaneously and 2 are queued in the per-model scheduler queues in Triton Inference Server to execute after the 8 requests finish. With this configuration, 2680 inferences per second at 152 ms with batch size 8 on each inference server instance is achieved.

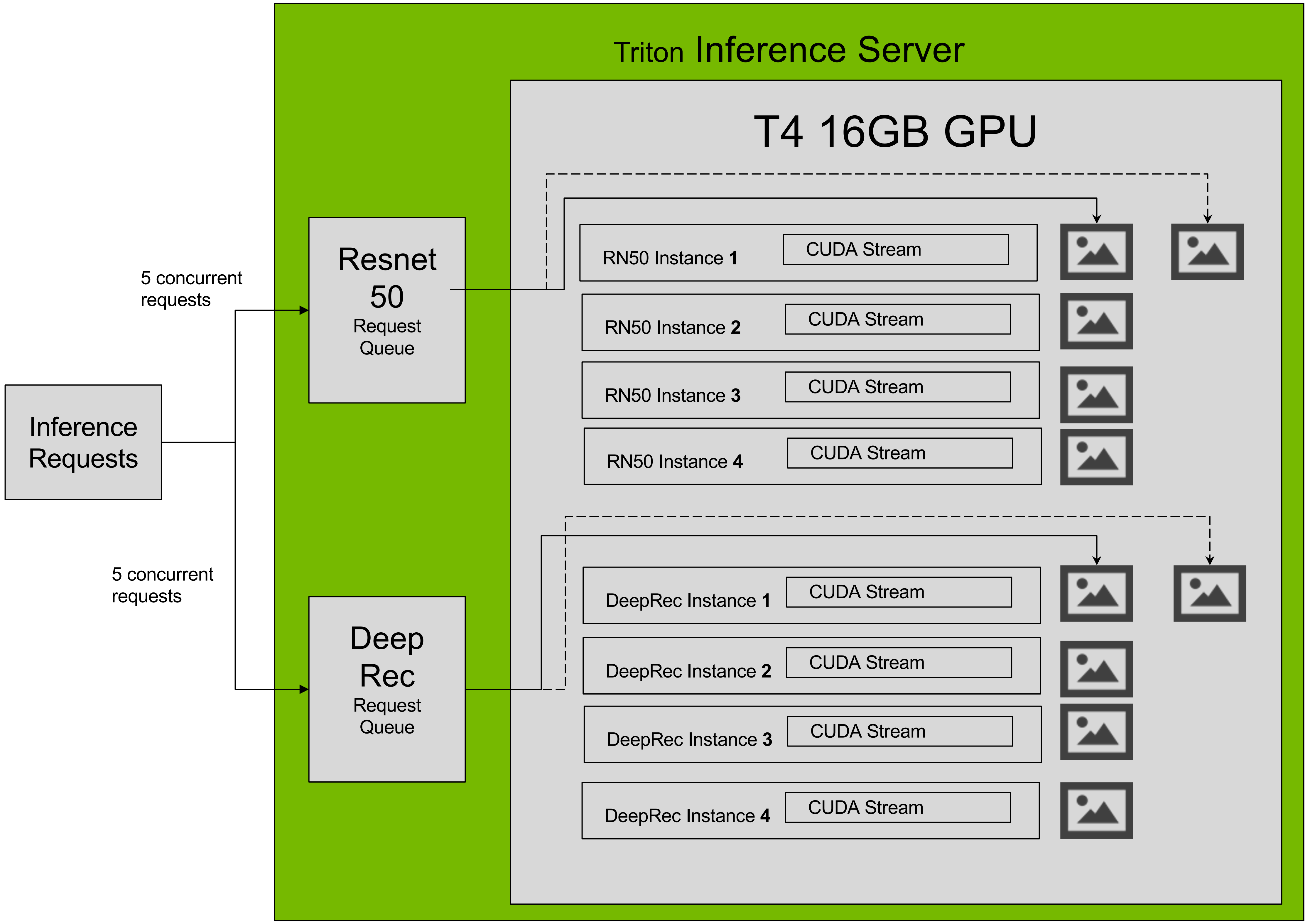


CONCURRENT MODEL EXECUTION
RESNET 50 & DEEP RECOMMENDER

Common Scenario 2

Many APIs using multiple different models on a GPU

Example: 4 instances of TRT FP16 ResNet50 and 4 instances of TRT FP16 Deep Recommender are running concurrently on one GPU. Ten requests come in for both models at the same time (5 for each model) and fed to the appropriate model for inference. The requests are fulfilled concurrently and sent back to the user. One request is queued for each model. With this configuration, 5778 inferences per second at 80 ms with batch size 8 on each inference server instance is achieved.

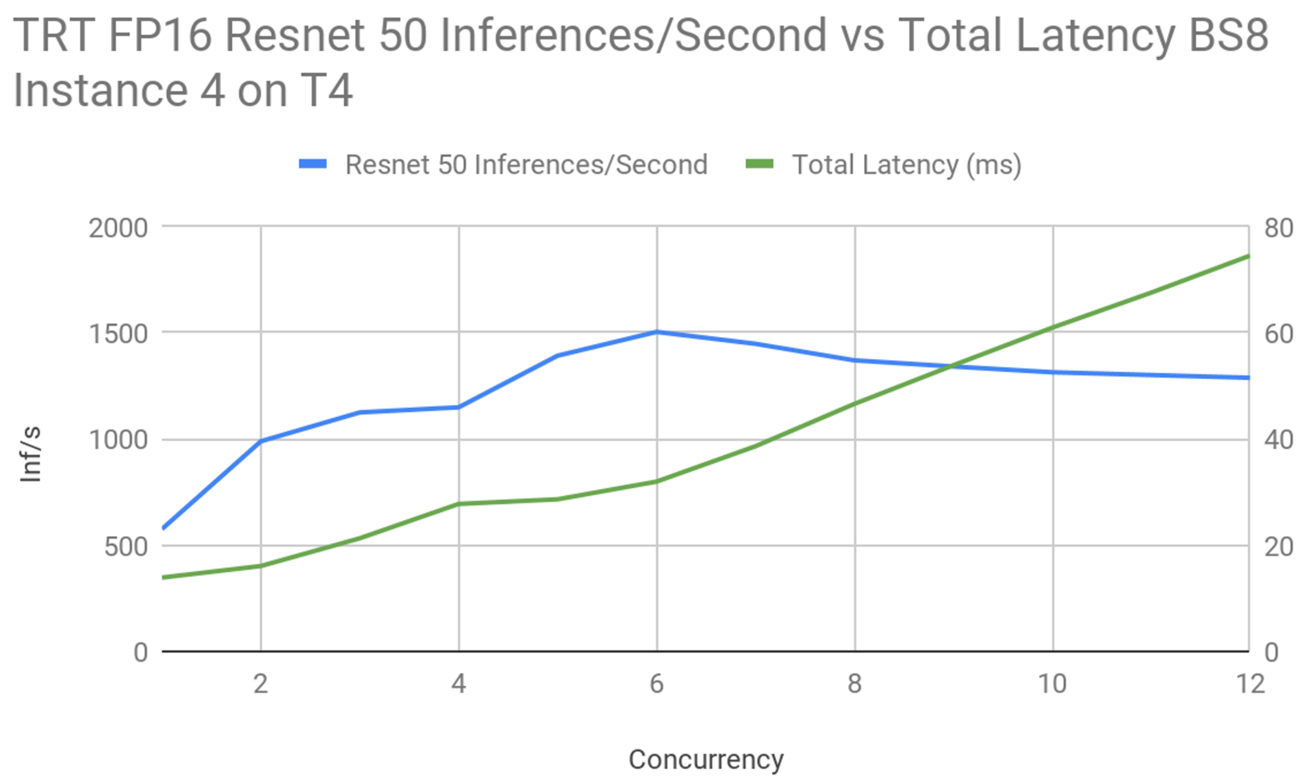


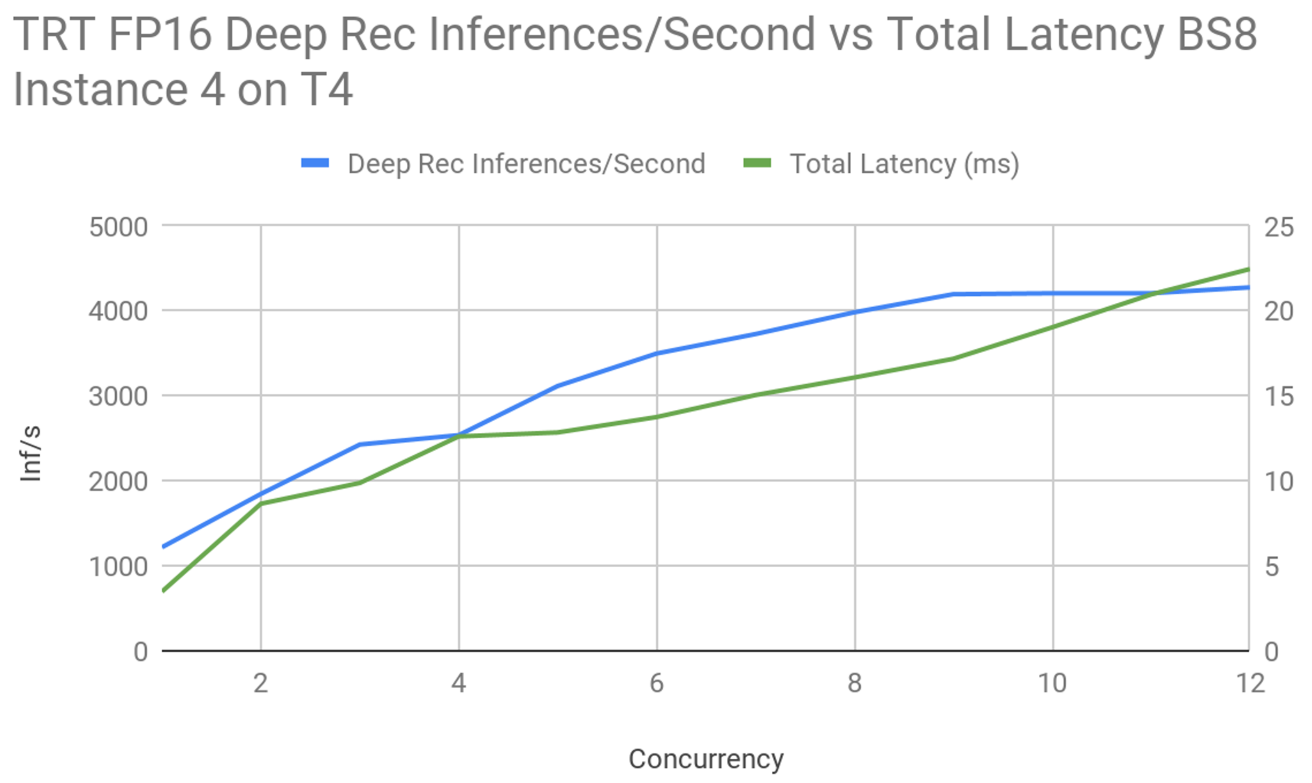
CONCURRENT MODEL EXECUTION
RESNET 50 & DEEP RECOMMENDER

<u>Common Scenario 2</u>

<u>Many</u> APIs using multiple <u>different</u> models on a GPU

Example: 4 instances of TRT FP16 ResNet50 and 4 instances of TRT FP16 Deep Recommender are running concurrently on one GPU. Ten requests come in for both models at the same time (5 for each model) and fed to the appropriate model for inference. The requests are fulfilled concurrently and sent back to the user. One request is queued for each model. With this configuration, 5778 inferences per second at 80 ms with batch size 8 on each inference server instance is achieved.





TRITON INFERENCE SERVER METRICS FOR AUTOSCALING

Before Triton Inference Server - 800 FPS

- One model per GPU
- Requests are steady across all models
- Utilization is low on all GPUs

Before Triton Inference Server - 5,000 FPS

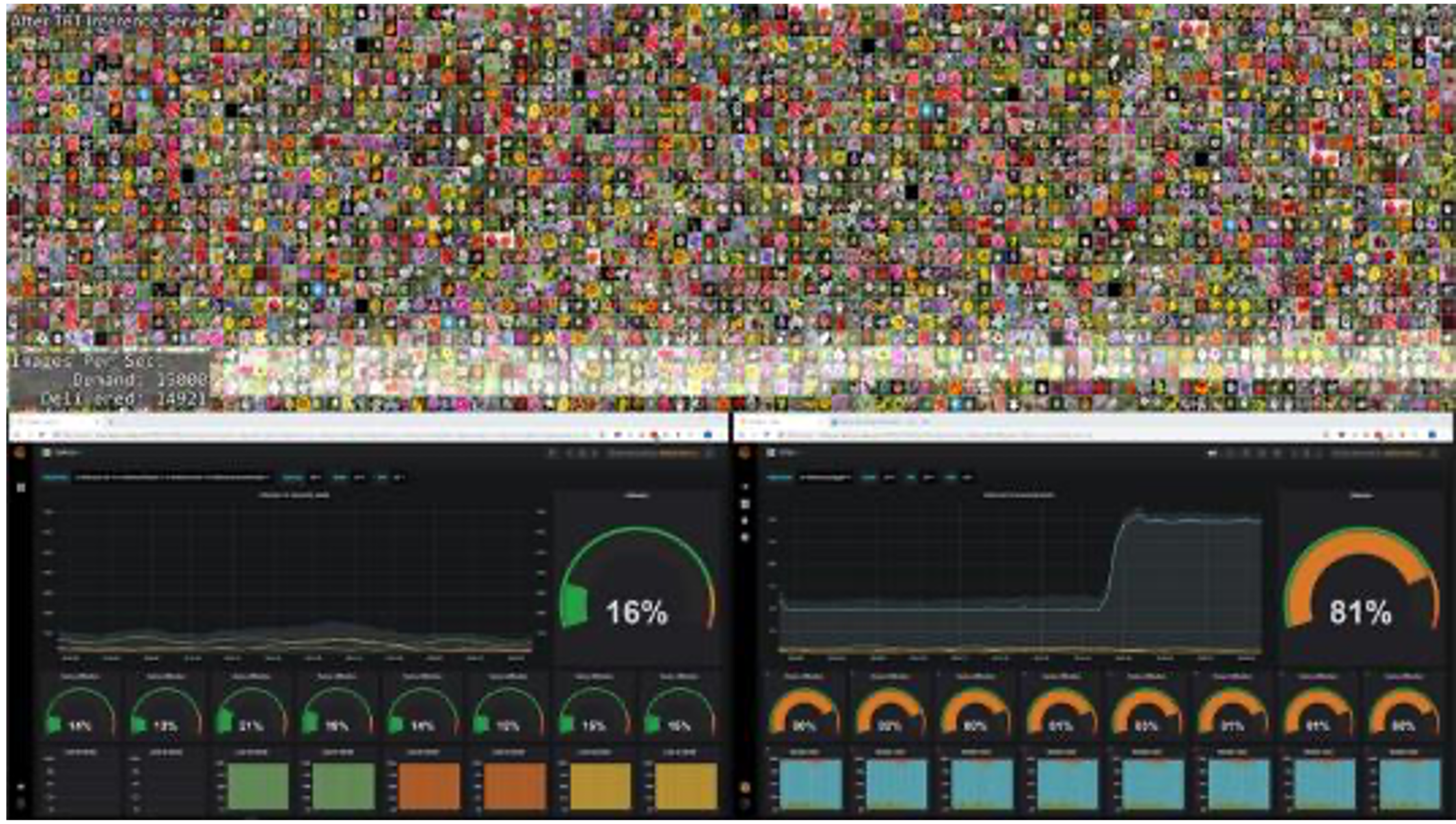
- Spike in requests for blue model
- GPUs running blue model are being fully utilized
- Other GPUs remain underutilized

TRITON INFERENCE SERVER METRICS FOR AUTOSCALING

After Triton Inference Server - 5,000 FPS

- Load multiple models on every GPU
- Load is evenly distributed between all GPUs

After Triton Inference Server - 15,000 FPS

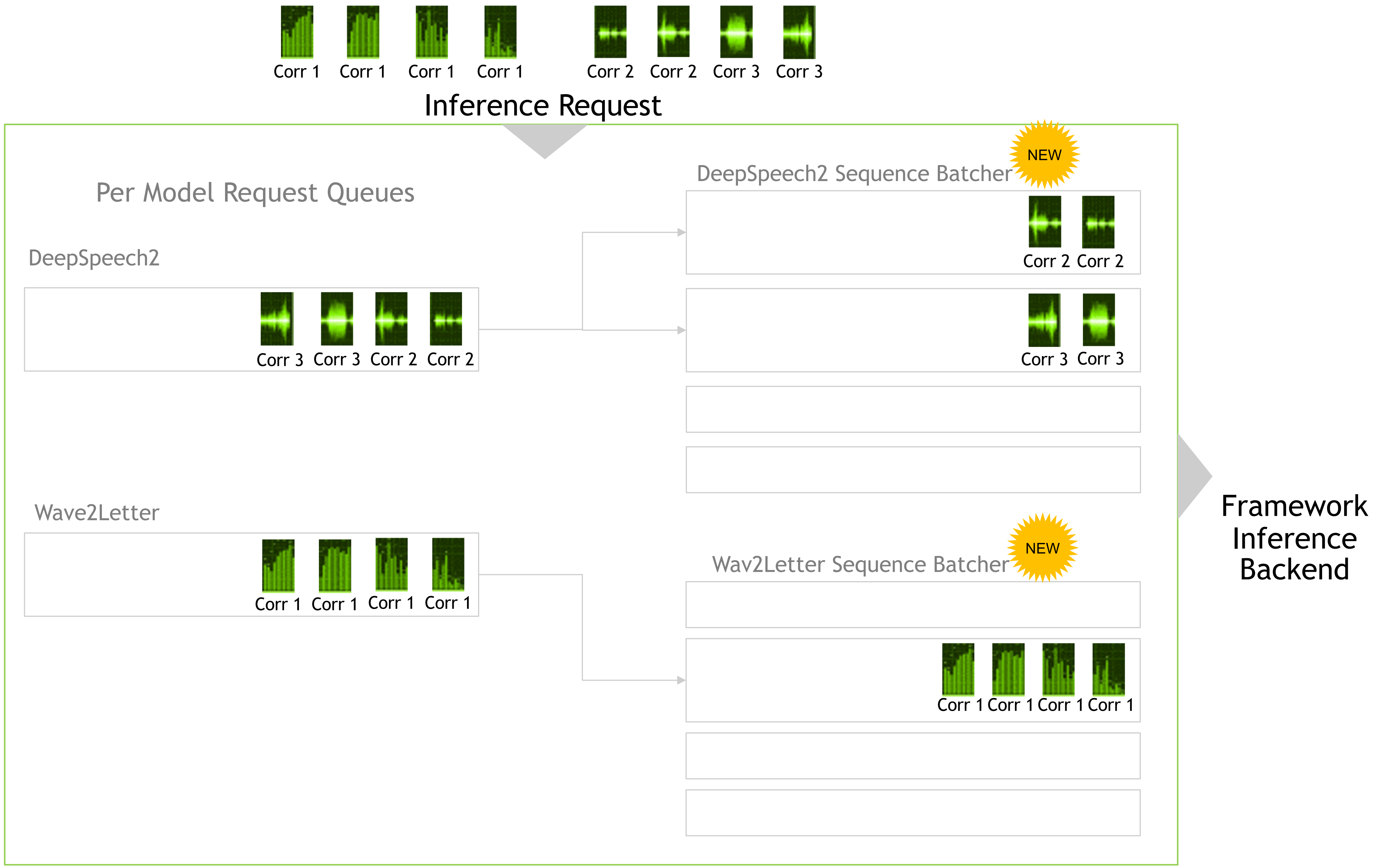


- Spike in requests for blue model
- Each GPU can run the blue model concurrently
- Metrics to indicate time to scale up
    - GPU utilization
    - Power usage
    - Inference count
    - Queue time
    - Number of requests/sec

STREAMING INFERENCE REQUESTS

New Streaming API

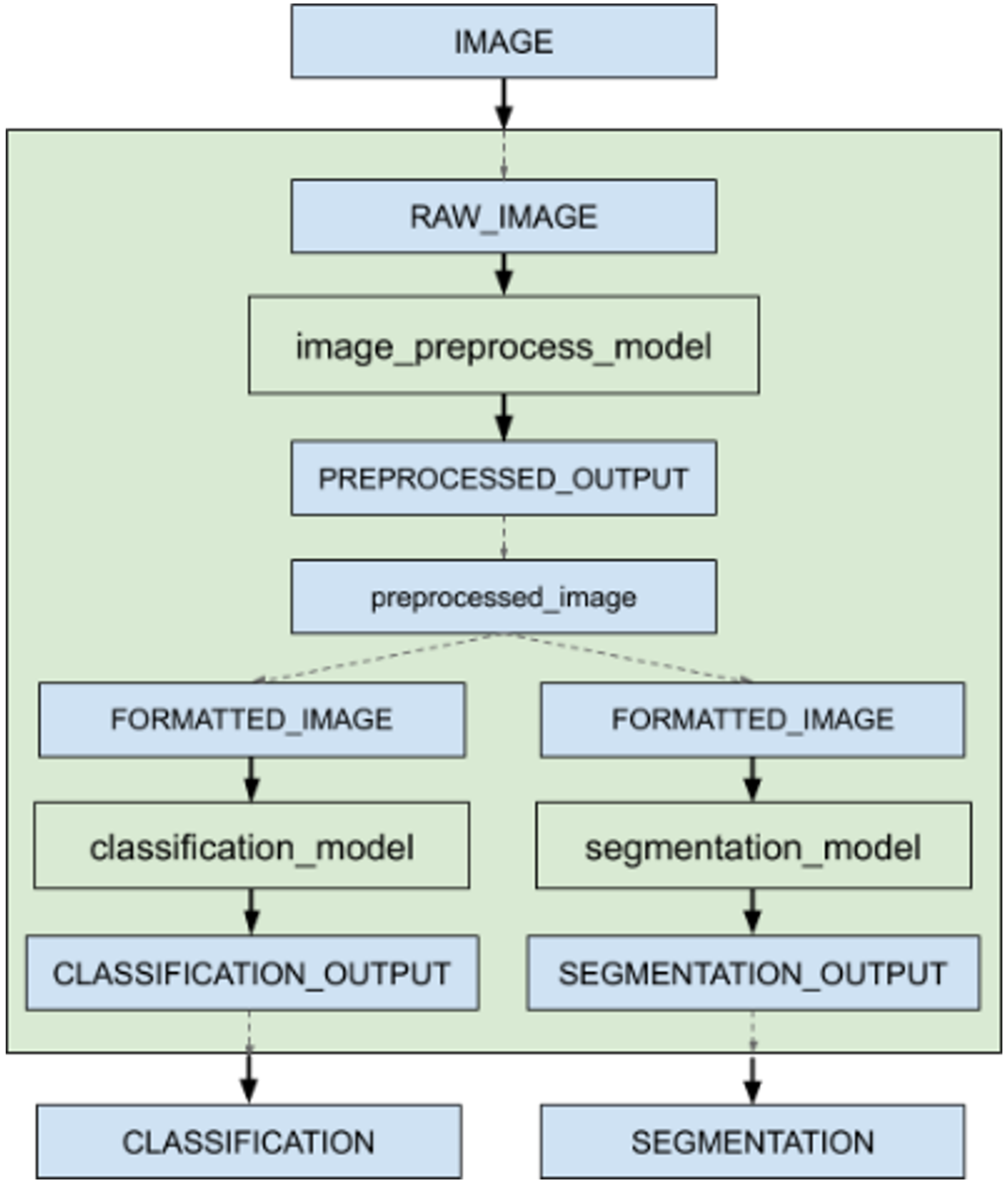
Based on the correlation ID, the audio requests are sent to the appropriate batch slot in the sequence batcher*



*Correct order of requests is assumed at entry into the endpoint
Note: Corr = Correlation ID

MODEL ENSEMBLING

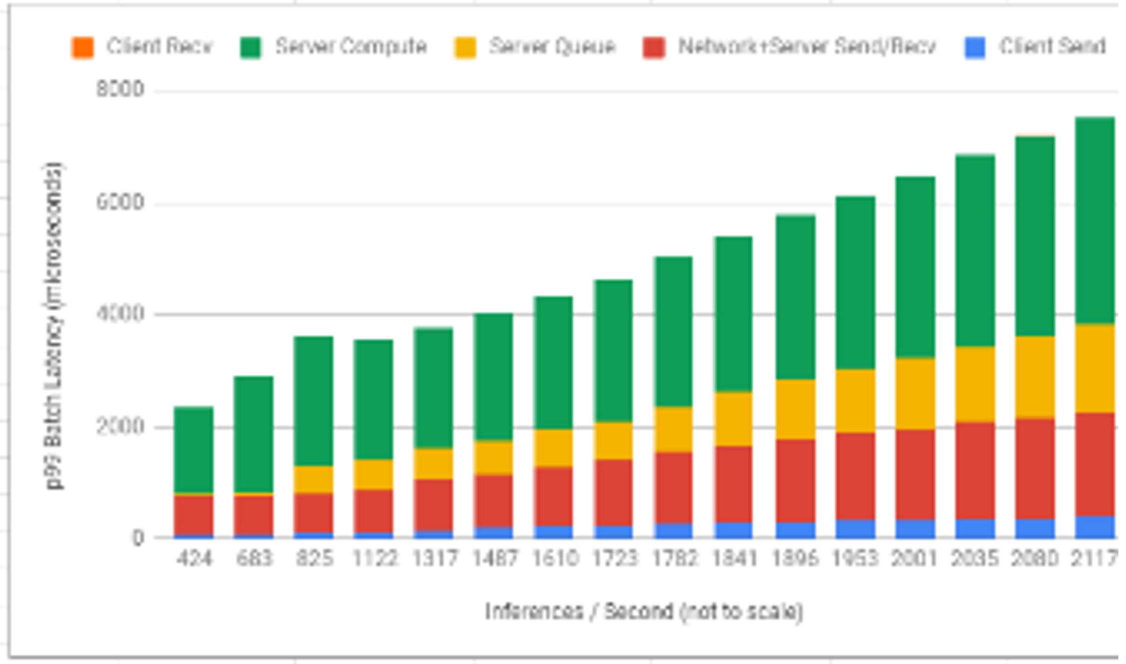
- Pipeline of one or more models and the connection of input and output tensors between those models
- Use for model stitching or data flow of multiple models such as data preprocessing → inference → data post-processing
- Collects the output tensors in each step, provides them as input tensors for other steps according to the specification
- Ensemble models will inherit the characteristics of the models involved, so the meta-data in the request header must comply with the models within the ensemble



`perf_client` TOOL

- Measures throughput (inf/s) and latency under varying client loads
- **`perf_client`** Modes
  1. Specify how many concurrent outstanding requests and it will find a stable latency and throughput for that level
  2. Generate throughput vs latency curve by increasing the request concurrency until a specific latency or concurrency limit is reached
- Generates a file containing CSV output of the results
- Easy steps to help visualize the throughput vs latency tradeoffs

	p99 Batch Latency (microseconds)					
	Client Send	Network+Server Send/Recv	Server Queu	Server Compute	Client Recv	Total
24	75	689	51	1522	6	2343
83	91	696	42	2076	7	2912
25	104	706	508	2293	7	3618
22	126	755	522	2140	7	3550
17	156	909	548	2158	7	3778
87	194	969	601	2247	7	4018
10	224	1060	690	2357	7	4328
23	248	1141	723	2505	7	4624
82	272	1290	797	2668	7	5034
41	289	1352	987	2781	7	5416
96	302	1467	1093	2922	7	5791
53	327	1588	1135	3073	8	6131
01	334	1619	1271	3252	8	6484
35	362	1723	1350	3419	8	6862
80	374	1782	1461	3565	8	7190
17	383	1874	1560	3710	8	7535

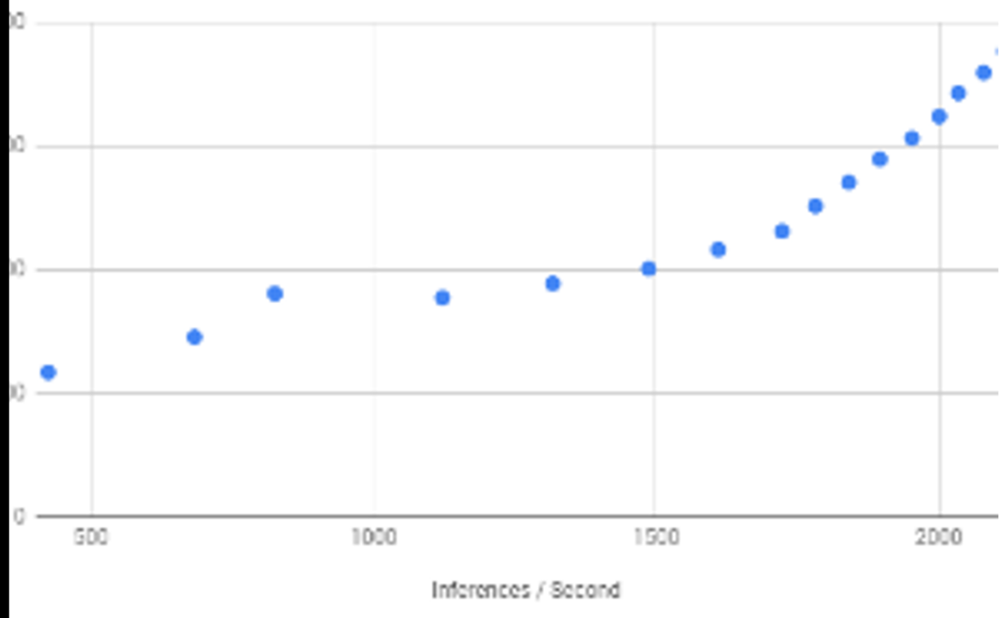



```
    Request count: 2187
    Throughput: 729 infer/sec
    Avg latency: 2728 usec (standard deviation 162 usec)
    Avg gRPC time: 2687 usec (marshal 89 usec + response wait 2591 usec + unmarshal 7 usec)
  Server:
    Request count: 2623
    Avg request latency: 1970 usec (overhead 18 usec + queue 38 usec + compute 1914 usec)

Request concurrency: 3
  Pass [1] throughput: 861 infer/sec. Avg latency: 3471 usec (std 1429 usec)
  Pass [2] throughput: 861 infer/sec. Avg latency: 3467 usec (std 1312 usec)
  Pass [3] throughput: 861 infer/sec. Avg latency: 3468 usec (std 1446 usec)
  Client:
    Request count: 2585
    Throughput: 861 infer/sec
    Avg latency: 3468 usec (standard deviation 1446 usec)
    Avg gRPC time: 3440 usec (marshal 98 usec + response wait 3335 usec + unmarshal 7 usec)
  Server:
    Request count: 3093
    Avg request latency: 2701 usec (overhead 16 usec + queue 484 usec + compute 2201 usec)

Request concurrency: 4
  Pass [1] throughput: 918 infer/sec. Avg latency: 4342 usec (std 1251 usec)
  Pass [2] throughput: 894 infer/sec. Avg latency: 4459 usec (std 1392 usec)
  Pass [3] throughput: 909 infer/sec. Avg latency: 4384 usec (std 1271 usec)
  Client:
    Request count: 2728
    Throughput: 909 infer/sec
    Avg latency: 4383 usec (standard deviation 1271 usec)
    Avg gRPC time: 4355 usec (marshal 118 usec + response wait 4230 usec + unmarshal 7 usec)
  Server:
    Request count: 3267
    Avg request latency: 3587 usec (overhead 15 usec + queue 1376 usec + compute 2196 usec)

Inferences/Second vs. Client Average Batch Latency
Concurrency: 1, 418 infer/sec, latency 2376 usec
Concurrency: 2, 729 infer/sec, latency 2728 usec
Concurrency: 3, 861 infer/sec, latency 3468 usec
Concurrency: 4, 909 infer/sec, latency 4383 usec
```



ALL CPU WORKLOADS SUPPORTED

Deploy the CPU workloads used today and benefit from Triton Inference Server features (TRT not required)

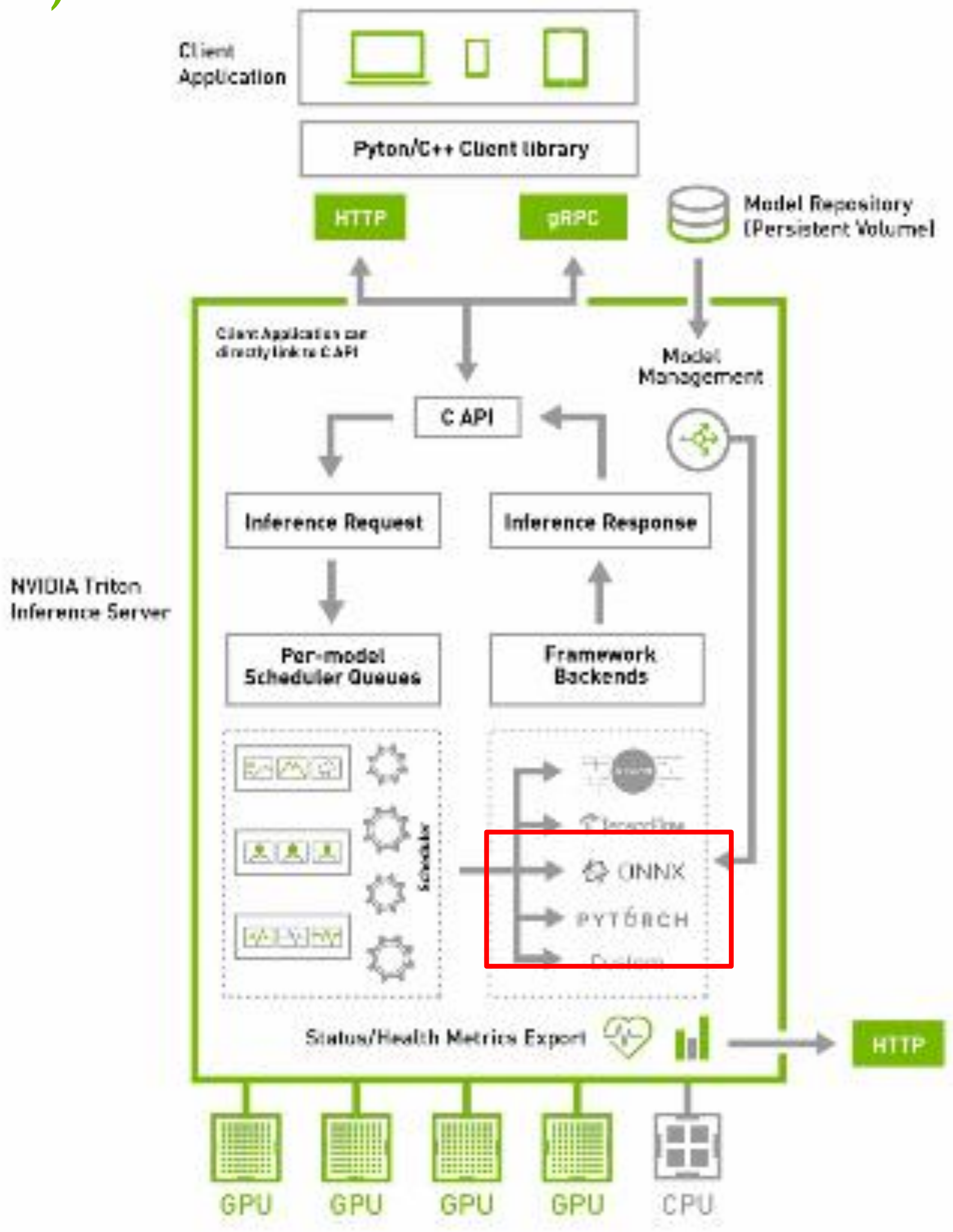
Triton relies on framework backends (Tensorflow, Caffe2, PyTorch) to execute the inference request on CPU

Support for Tensorflow and Caffe2 CPU optimizations using Intel MKL-DNN library

Allows frameworks backends to make use of multiple CPUs and cores

Benefit from features:

- Multiple Model Framework Support
- Dynamic batching
- Custom backend
- Model Ensembling
- Audio Streaming API



TRITON INFERENCE SERVER COLLABORATION WITH KUBEFLOW

What is Kubeflow?

- Open-source project to make ML workflows on Kubernetes simple, portable, and scalable
- Customizable scripts and configuration files to deploy containers on their chosen environment

Problems it solves

- Easily set up an ML stack/pipeline that can fit into the majority of enterprise datacenter and multi-cloud environments

How it helps Triton Inference Server

- Triton Inference Server is deployed as a component inside of a production workflow to
    - Optimize GPU performance
    - Enable auto-scaling, traffic load balancing, and redundancy/failover via metrics

Kubeflow

TensorRT

For a more detailed explanation and step-by-step guidance for this collaboration, refer to this GitHub repo.

TRITON INFERENCE SERVER HELM CHART

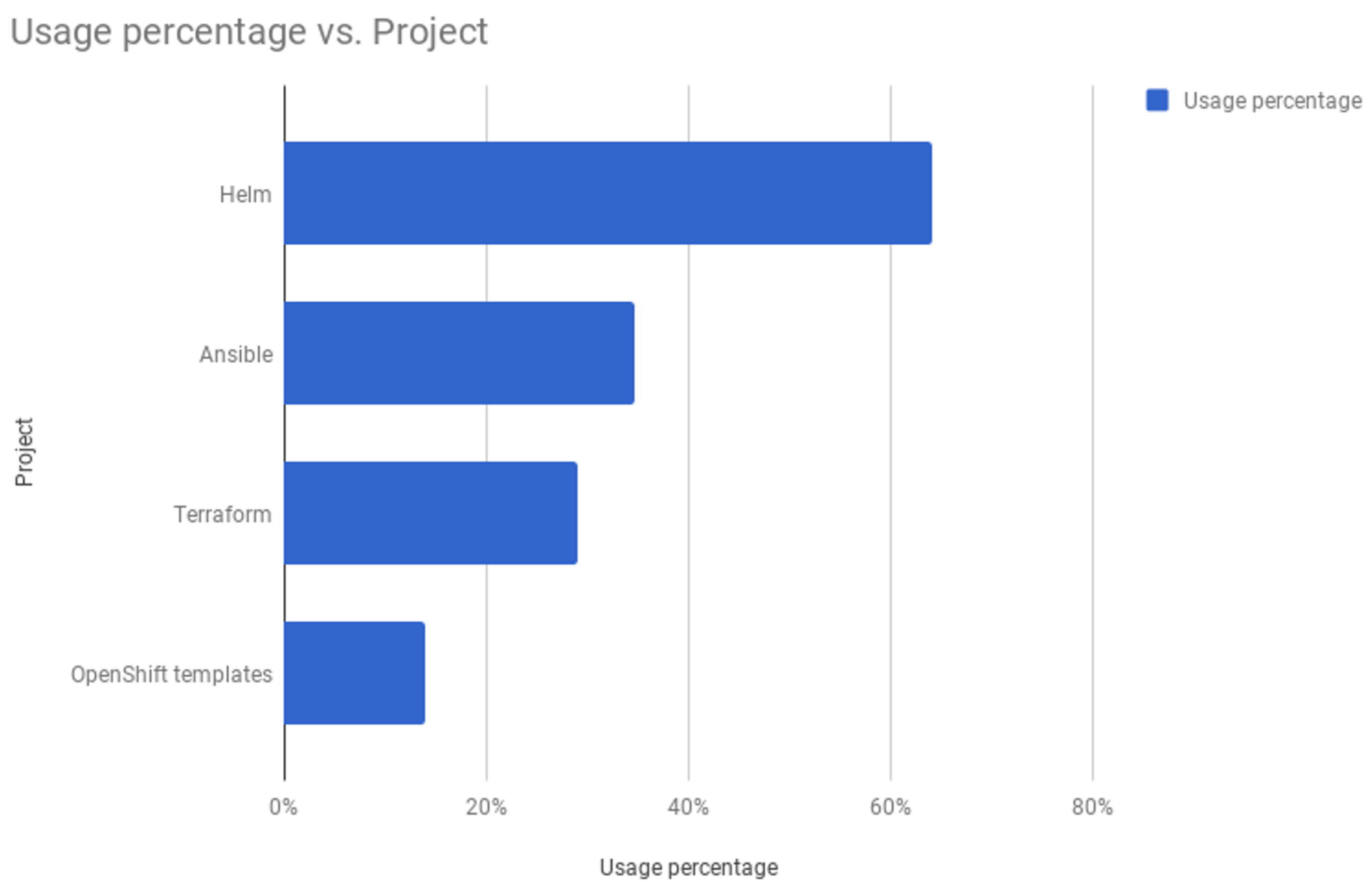
Simple helm chart for installing a single instance of the NVIDIA Triton Inference Server

Helm: Most used “package manager” for Kubernetes

We built a simple chart (“package”) for the Triton Inference Server.

You can use it to easily deploy an instance of the server. It can also be easily configured to point to a different image, model store, …

https://github.com/NVIDIA/tensorrt-inference-server/tree/b6b45ead074d57e3d18703b7c0273672c5e92893/deploy/single_server



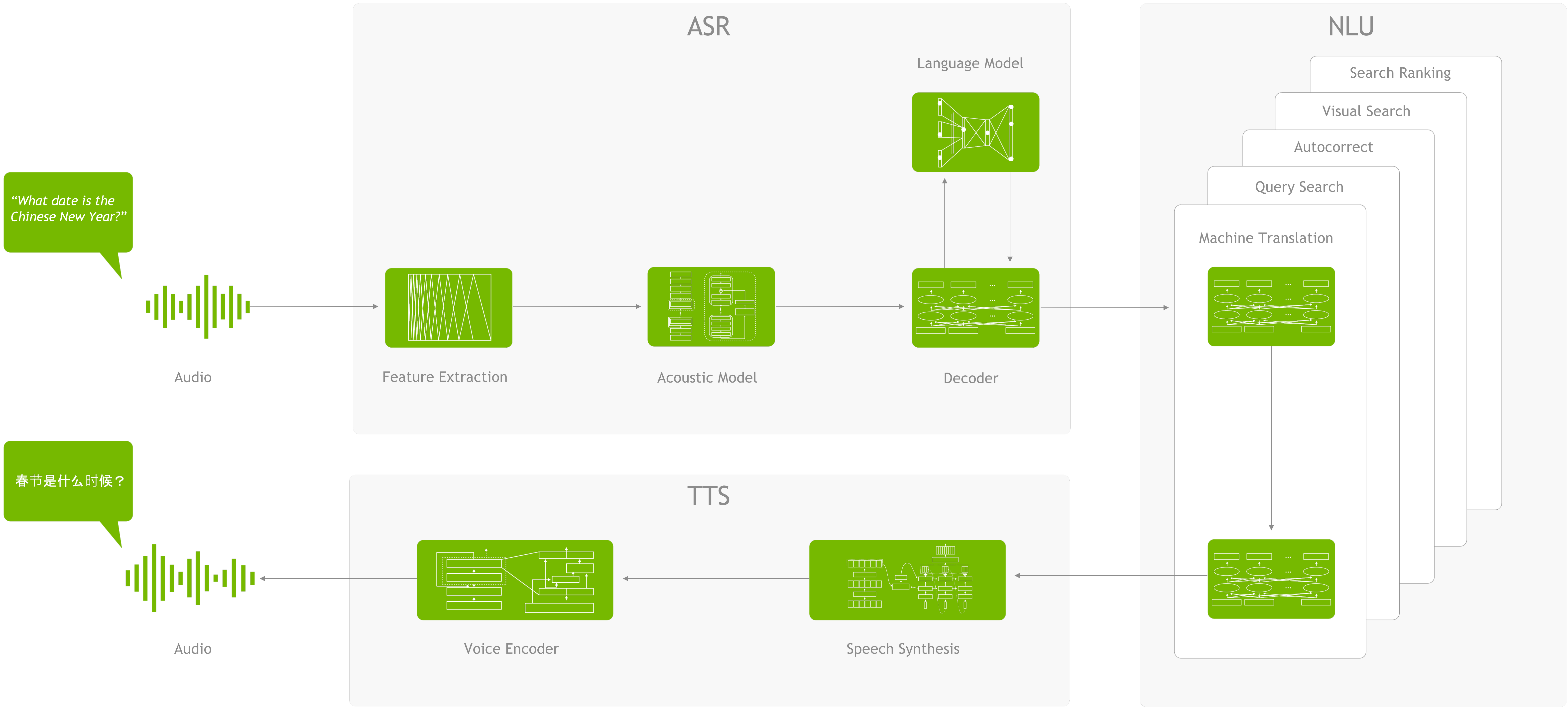
Part 3: Production Deployment

- Lecture
  - Model Selection
  - Post-Training Optimization
  - Product Quantization
  - Knowledge Distillation
  - Model Code Efficiency
  - Model Serving
  - Building the Application
- Lab
  - Exporting the Model
  - Hosting the Model
  - Server Performance
  - Using the Model

APPLICATION != SINGLE MODEL

THE APPLICATION

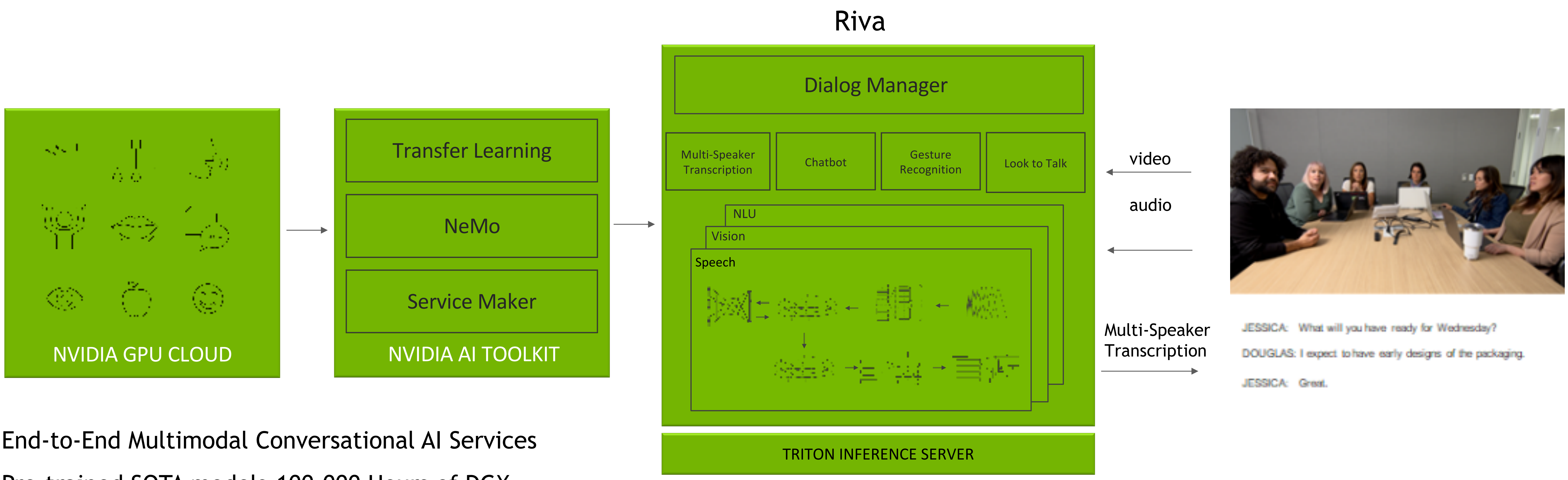
Typically composed of many components



RIVA

NVIDIA RIVA

Fully Accelerated Framework for Multimodal Conversational AI Services



End-to-End Multimodal Conversational AI Services

Pre-trained SOTA models-100,000 Hours of DGX

Retrain with NeMo

Interactive Response – 150ms on A100 versus 25sec on CPU

Deploy Services with One Line of Code

PRETRAINED MODELS AND AI TOOLKIT

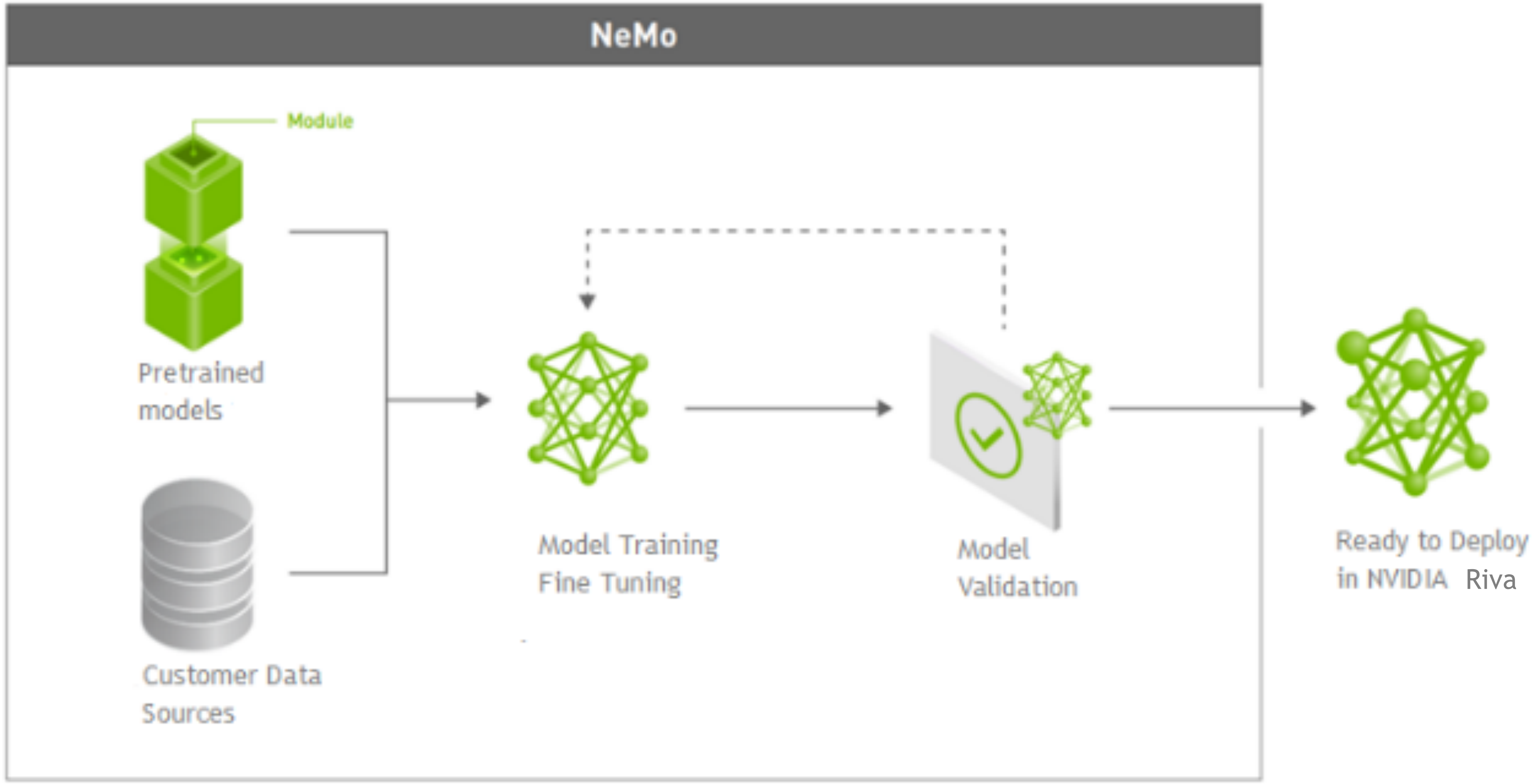
Train SOTA Models on Your Data to Understand your Domain and Jargon

100+ pretrained models in NGC

SOTA models trained over 100,000 hours on NVIDIA DGX™

Retrain for your domain using NeMo & TAO Toolkit

Deploy trained models to real-time services using Helm charts



MULTIMODAL SKILLS

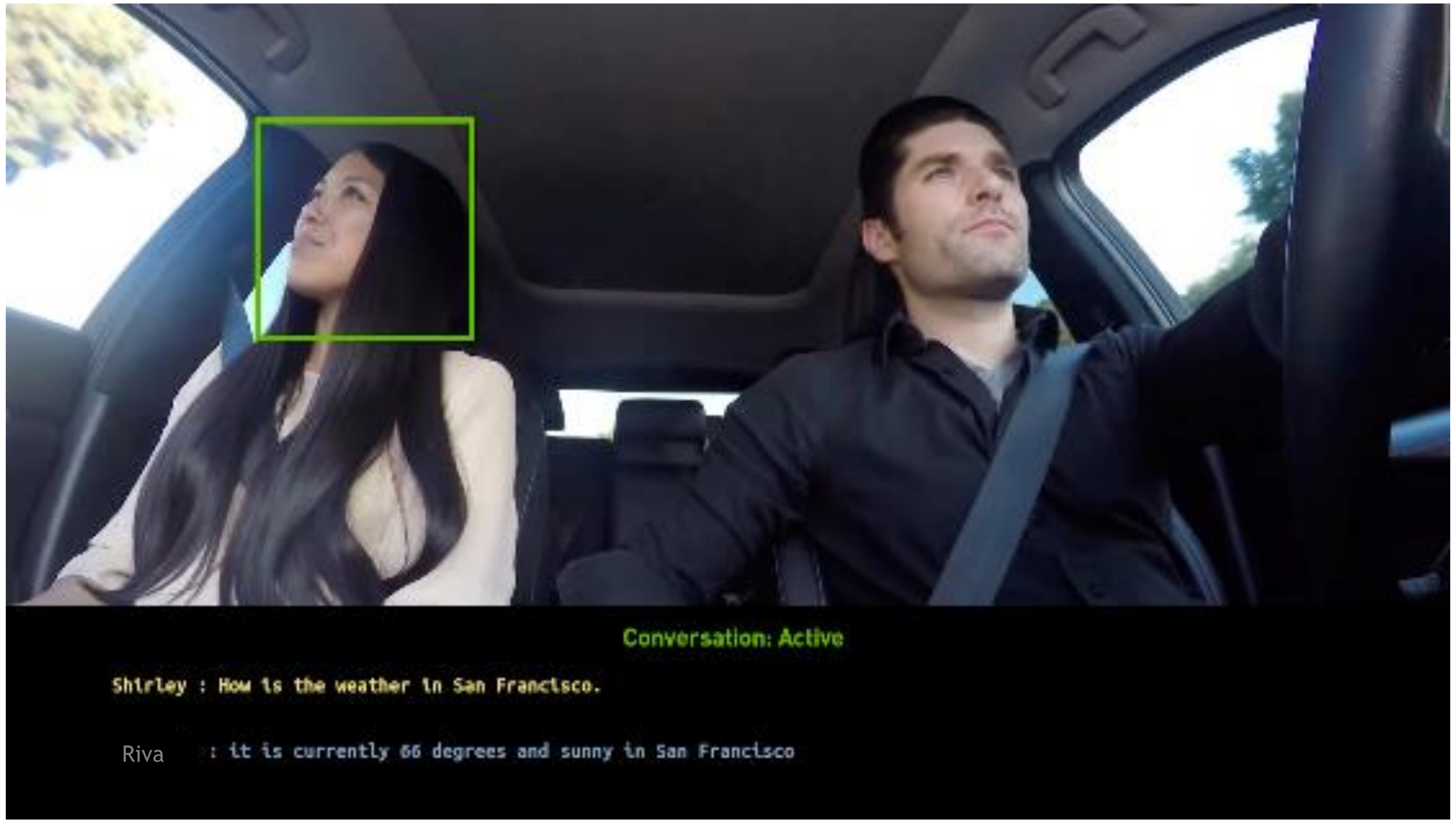
Use speech and vision for natural interaction

Build new skills by fusing services for ASR, NLU, TTS, and CV

Reference skills include:

- Multi-speaker transcription
- Chatbot
- Look-to-talk

Dialog manager manages multi-user and multi-context scenarios



Multimodal application with multiple users and contexts

BUILD CONVERSATIONAL AI SERVICES

Optimized Services for Real Time Applications

Build applications easily by connecting performance tuned services

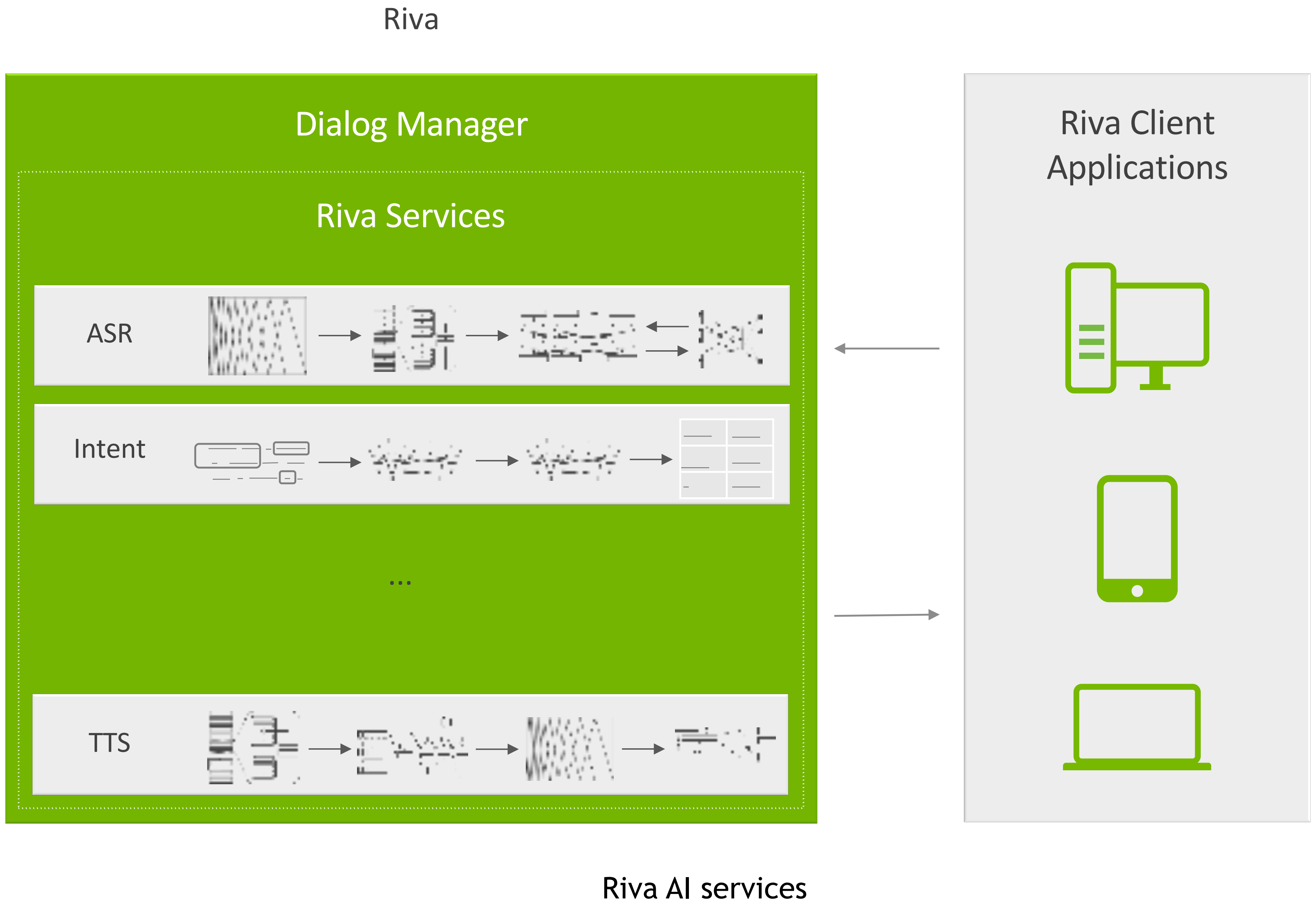
Task specific services include:

- ASR
- Intent Classification
- Slot Filling
- Pose Estimation
- Facial Landmark Detection

Services for streaming & batch usage

Build new services from any model in ONNX format

Access services for gRPC and HTTP endpoints



https://ngc.nvidia.com/catalog/model-scripts/nvidia:jasper_for_trtis

DEPLOY MODELS AS REAL-TIME SERVICES

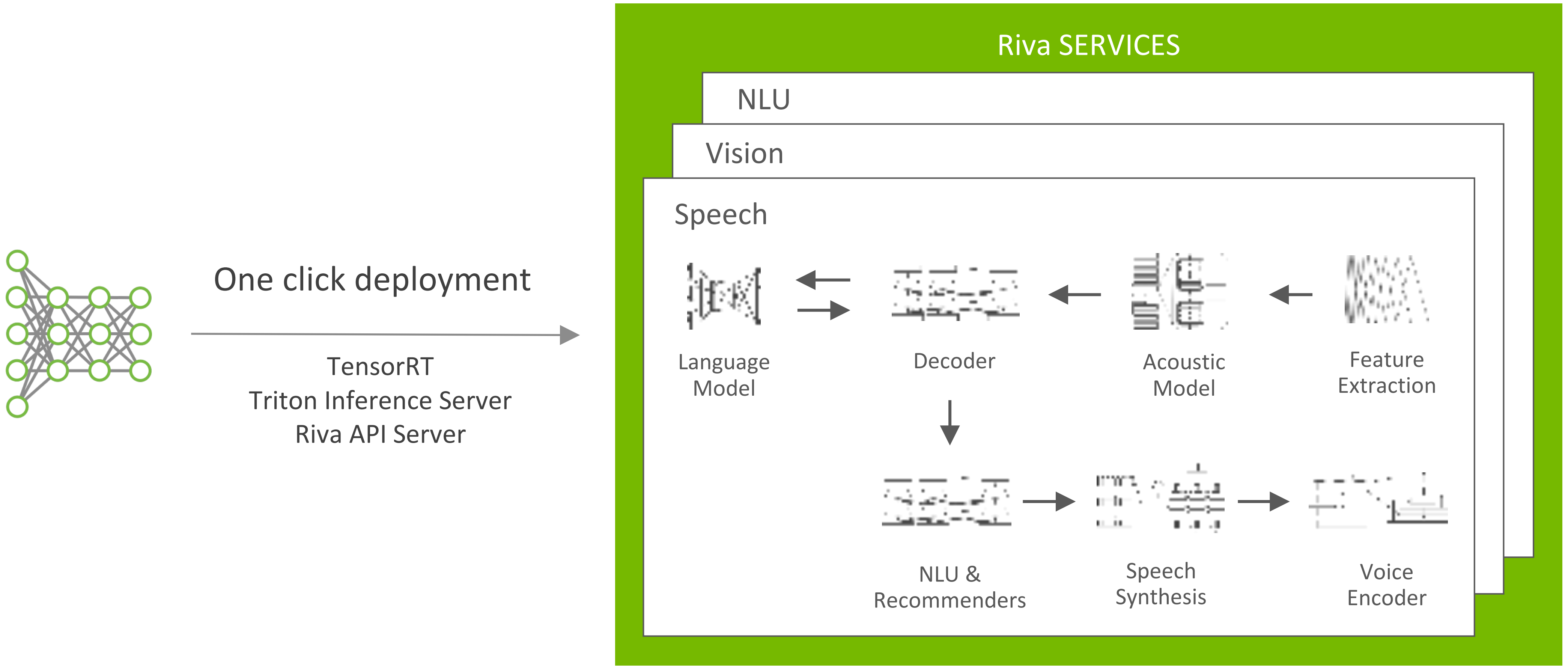
One Click to Create High-Performance Services from SOTA Models

Deploy models to services in the cloud, data center, and at the edge

Single command to set up and run the entire Riva application

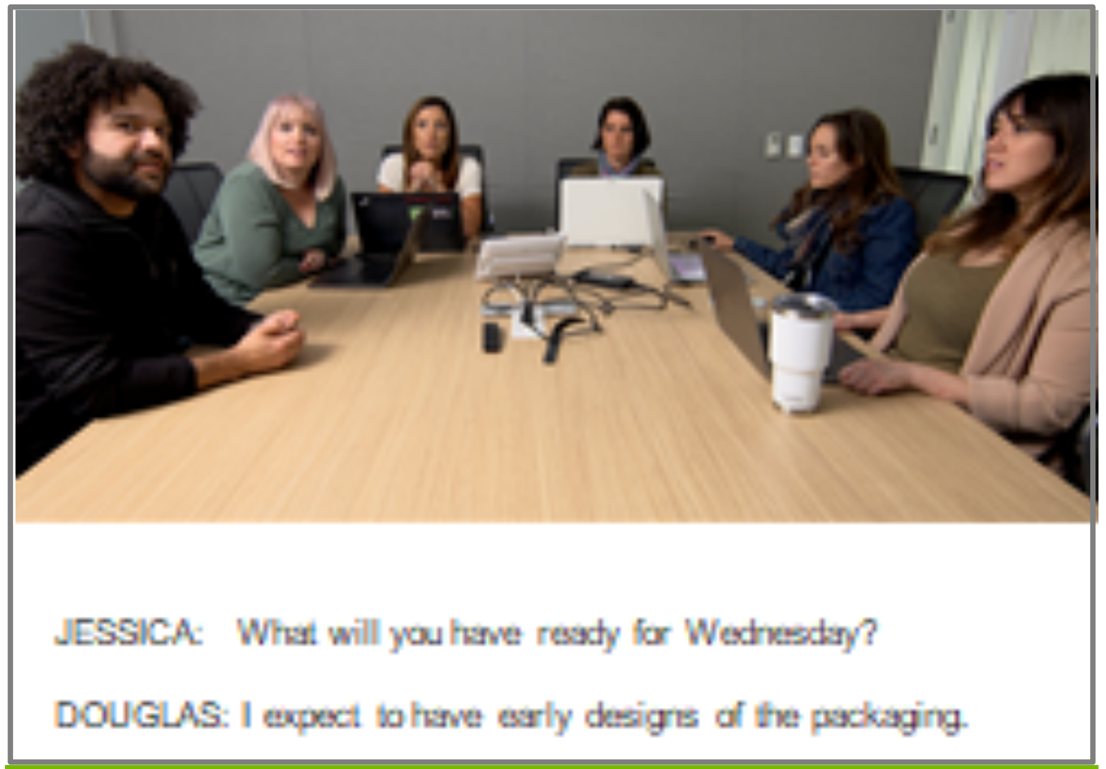
through Helm charts on Kubernetes cluster

Customization of Helm charts for your setup and use case.



Helm command to deploy models to production

RIVA SAMPLES

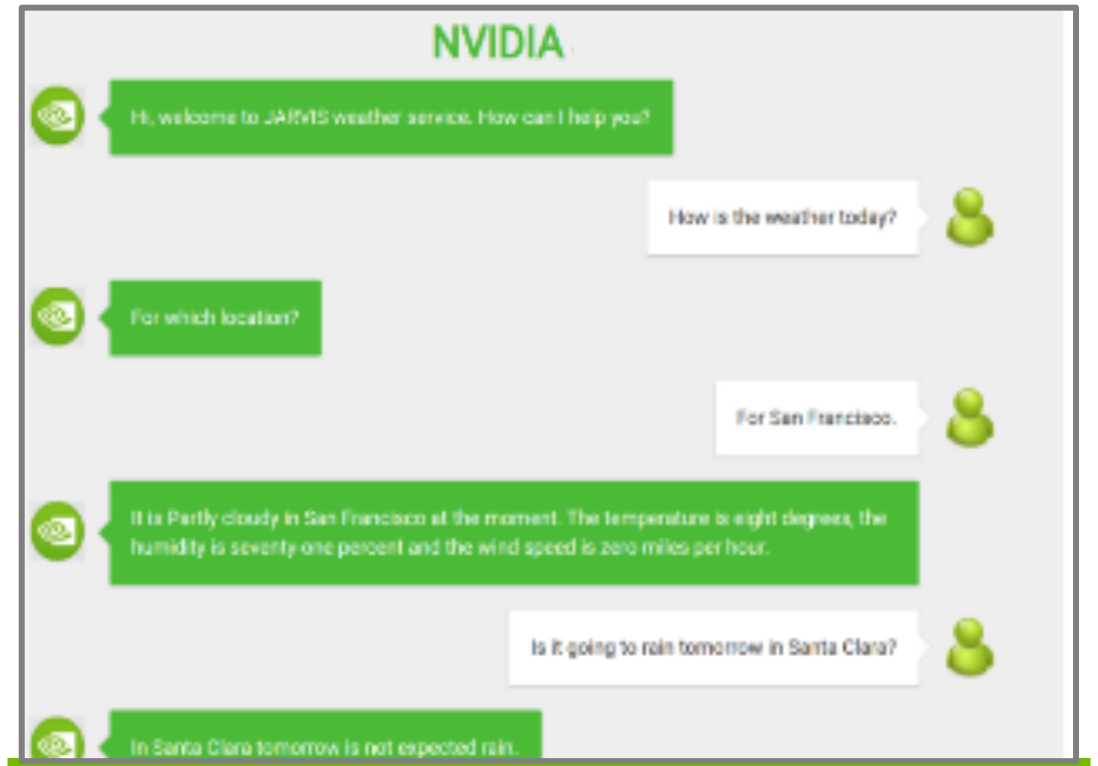


Visual Diarization

Transcribe multi-user multi-context conversations

Look To Talk

Wait for gaze before triggering AI assistant

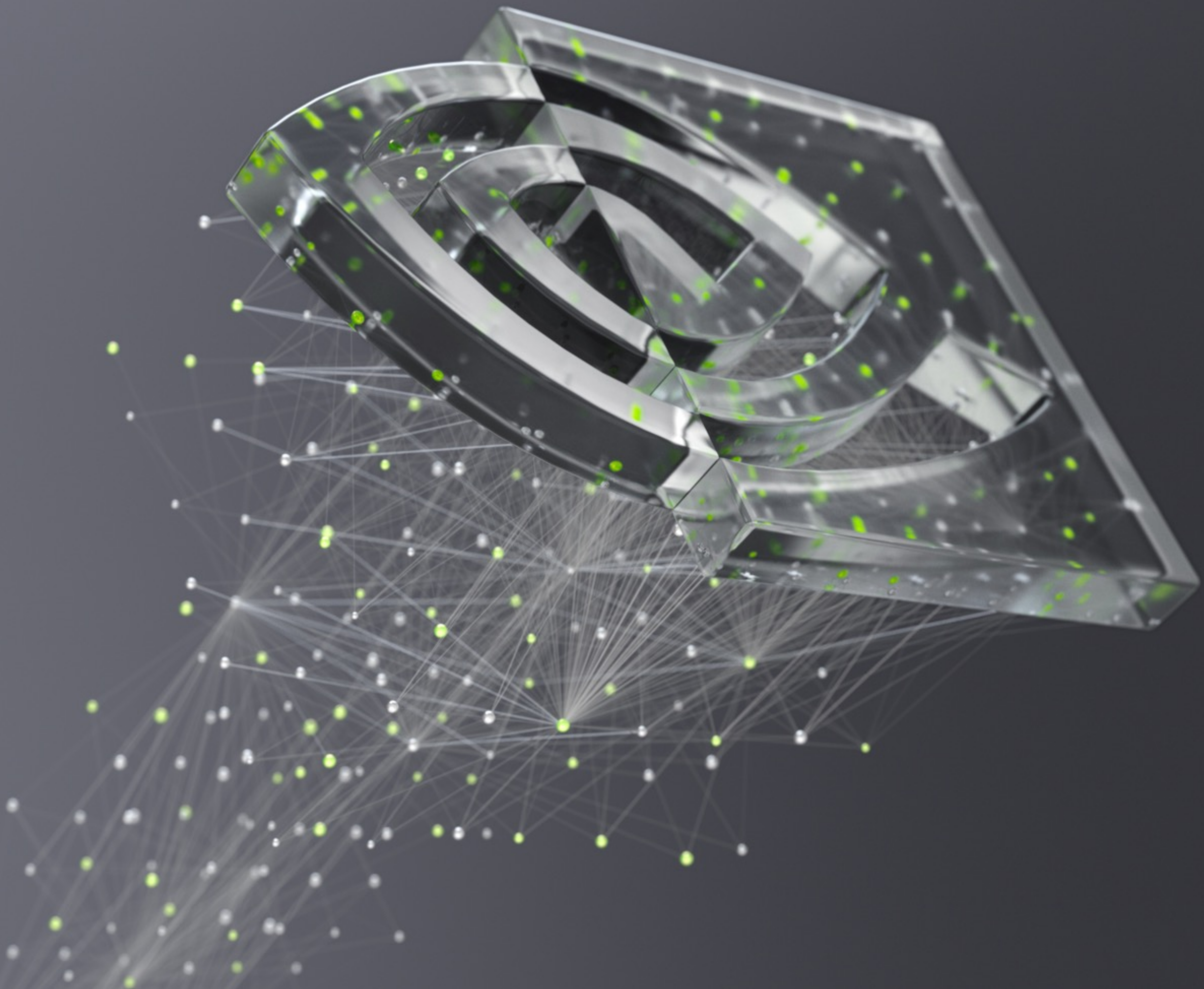


Virtual Assistant

End-to-end conversational AI system

Part 3: Production Deployment

- Lecture
  - Model Selection
  - Post-Training Optimization
  - Product Quantization
  - Knowledge Distillation
  - Model Code Efficiency
  - Model Serving
  - Building the Application
- Lab
  - Exporting the Model
  - Hosting the Model
  - Server Performance
  - Using the Model



NVIDIA

DEEP
LEARNING
INSTITUTE