NVIDIA AI DEVELOPER MEETUP

WHAT YOU NEED TO KNOW FOR DEEP LEARNING TRAINING

Jonghwan Lee(jonghwanl@nvidia.com), Deep Learning Data Scientist, NVIDIA

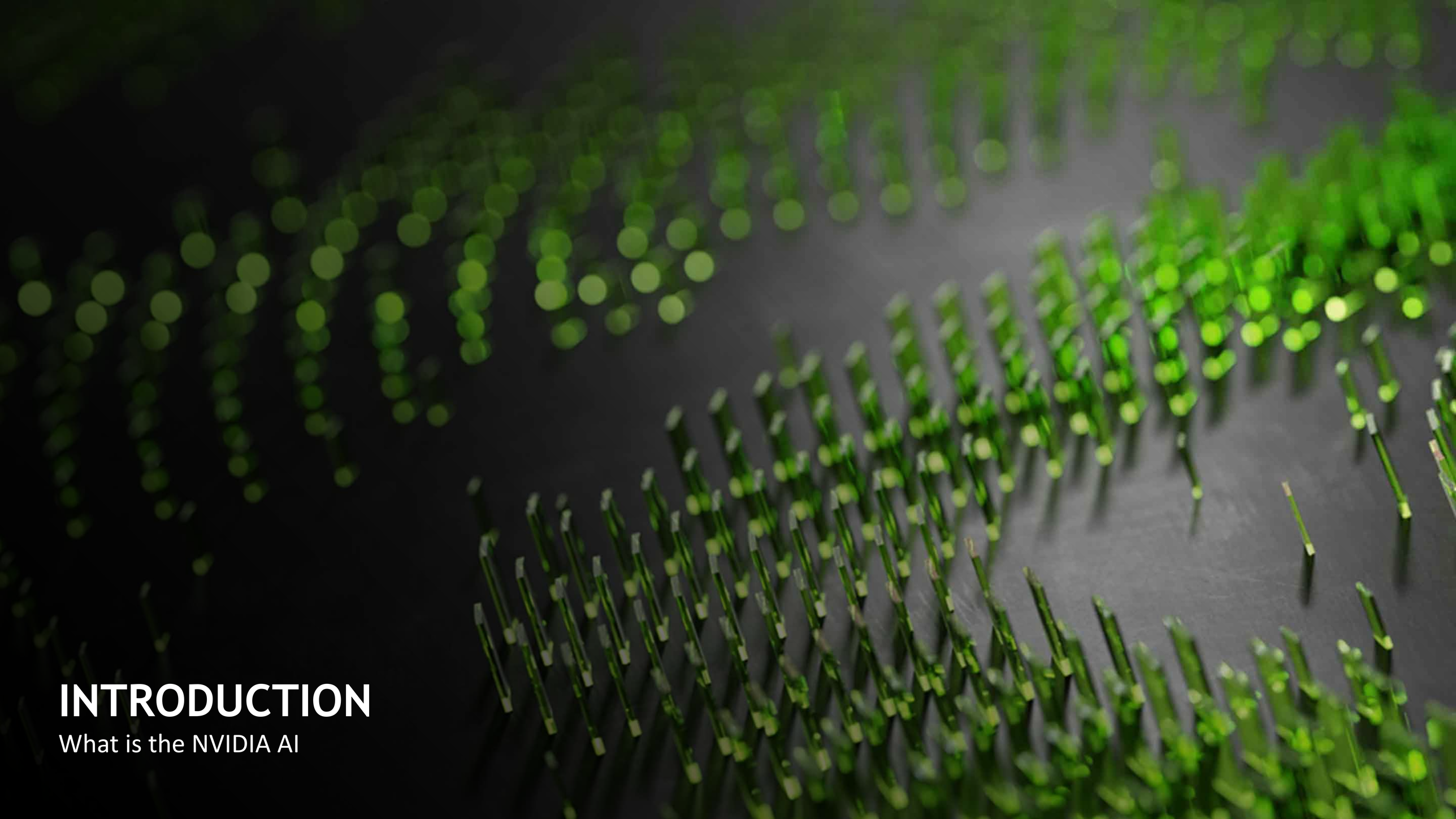
AGENDA

Introduction
- NVIDIA History & Datacenter platform
- AI/ Deep Learning Trends 2022
- NVIDIA E2E AI Software Stack & Workflow

NVIDIA AI for DL training
- APEX DDP
- APEX AMP
- APEX ASP
- Quantization (focus on QAT)

Let’s go to the Demo
- Hands-on session for:
  DDP/ AMP/ ASP/ QAT

Q & A



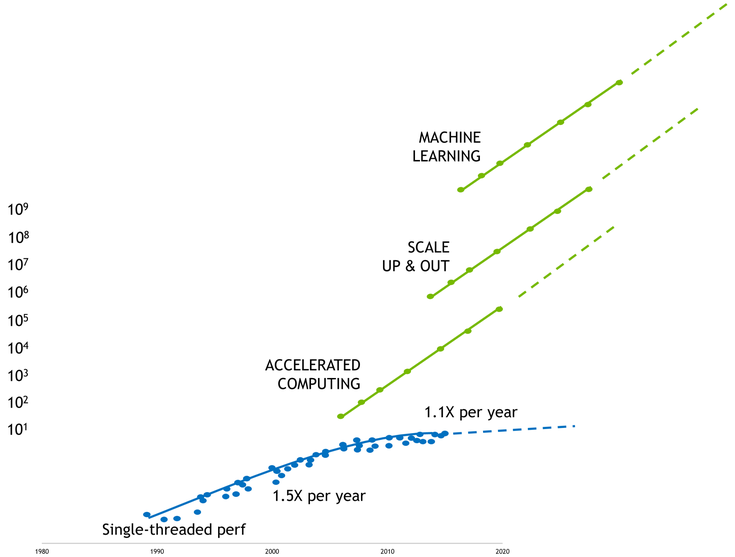
INTRODUCTION

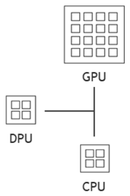
What is the NVIDIA AI

NVIDIA HISTORY

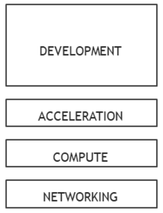
NVIDIA – A Learning Machine

Kelvin (2001) | Rankine (2003) | Curie (2004) | Tesla (2006) | Fermi (2010) | Kepler (2012) | Maxwell (2014) | Pascal (2016) | Volta (2017) | Turing (2018) | Ampere (2020) | Hopper (2022) | ???

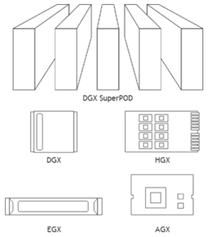




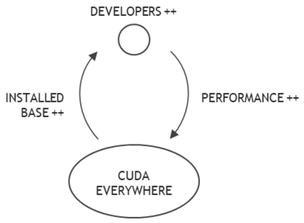
X-factor Speed-up



Full Stack



Compute Platform at scale



One Architecture

NVIDIA DATA CENTER PLATFORM

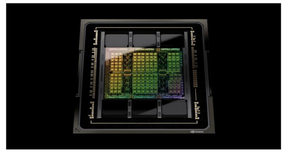
NVIDIA Hardware Platforms

Data Center	Edge Micro Data Centers	Edge Servers	Micro-Edge Servers	Devices
DGX	EGX			AGX

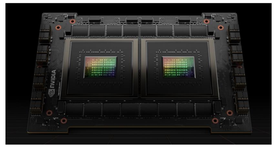
High Performance Servers	Mainstream Compute Servers	Nano-GPU
10,000 TOPS	320 TOPS	0.5 TOPS

NVIDIA Computing

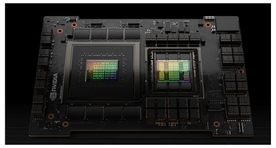
Data-center GPU



High Performance CPU



Converged CPU + GPU



NVIDIA Networking

NVLINK SWITCH SYSTEM

METROX-2 LONG-HAUL

SKYWAY GATEWAY

UFM CYBER-AI

CONNECTX ADAPTER

BLUEFIELD DPU

QUANTUM SWITCH

LINKX

NVIDIA DATA CENTER PLATFORM

NVIDIA Acceleration Libraries – CUDA, MagnumIO,DOCA and CUDA-X

CUDA-X

Math Libraries

GPU-accelerated math libraries lay the foundation for compute-intensive applications in areas such as molecular dynamics, computational fluid dynamics, computational chemistry, medical imaging, and seismic exploration.

cuBLAS
GPU-accelerated basic linear algebra (BLAS) library

cuFFT
GPU-accelerated library for Fast Fourier Transforms

CUDA Math Library
GPU-accelerated standard mathematical function library

cuRAND
GPU-accelerated random number generation (RNG)

cuSOLVER
GPU-accelerated dense and sparse direct solvers

cuSPARSE
GPU-accelerated BLAS for sparse matrices

cuTENSOR
GPU-accelerated tensor linear algebra library

AmgX
GPU-accelerated linear solvers for simulations and implicit unstructured methods

Image and Video Libraries

GPU-accelerated libraries for image and video decoding, encoding, and processing that leverage CUDA and specialized hardware components of GPUs.

nvJPEG
High performance GPU-accelerated library for JPEG decoding

lore

NVIDIA Performance Primitives
Provides GPU-accelerated image, video, and signal processing functions

NVIDIA Video Codec SDK
A complete set of APIs, samples, and documentation for hardware-accelerated video encode and decode on Windows and Linux

NVIDIA Optical Flow SDK
Exposes the latest hardware capability of NVIDIA Turing™ GPUs dedicated to computing the relative motion of pixels between images

Parallel Algorithm Libraries

GPU-accelerated libraries of highly efficient parallel algorithms for several operations in C++ and for use with graphs when studying relationships in natural sciences, logistics, travel planning, and more.

Thrust
GPU-accelerated library of C++ parallel algorithms and data structures

Communication Libraries

Performance-optimized multi-GPU and multi-node communication primitives.

NVSHMEM
OpenSHMEM standard for GPU memory, with extensions for improved performance on GPUs.

NCCL
Open-source library for fast multi-GPU, multi-node communications that maximizes bandwidth while maintaining low latency.

Partner Libraries

OpenCV
GPU-accelerated open-source library for computer vision, image processing, and machine learning, now supporting real-time operation

FFmpeg
Open-source multimedia framework with a library of plugins for audio and video processing

ArrayFire
GPU-accelerated open source library for matrix, signal, and image processing

MAGMA
GPU-accelerated linear algebra routines for heterogeneous architectures, by Magma

IMSL Fortran Numerical Library
GPU-accelerated open-source Fortran library with functions for math, signal, and image processing, statistics, by RogueWave

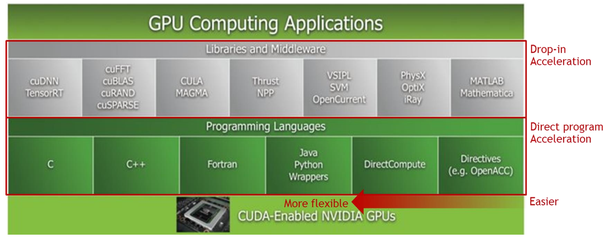
Gunrock
Library for graph-processing designed specifically for the GPU

CHOLMOD
GPU-accelerated functions for sparse direct solvers, included in the SuiteSparse linear algebra package, authored by Prof

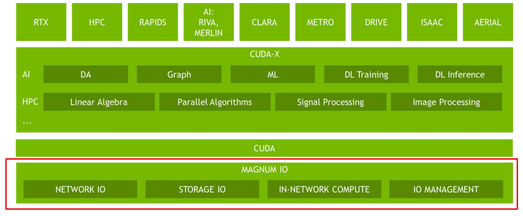
Triton Ocean SDK
Real-time visual simulation of oceans, water bodies in games, simulation, and training applications, by Triton.

CUVIlib
Primitives for accelerating imaging applications from medical, industrial, and defense domains

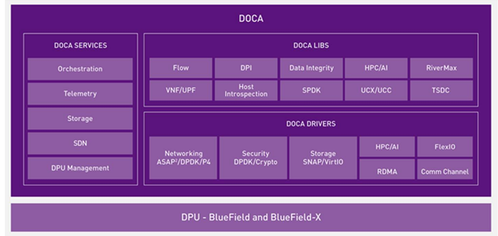
CUDA



MAGNUM IO



DOCA



NVIDIA

NVIDIA DATA CENTER PLATFORM

NVIDIA is a Full Stack Computing Platform

BUSINESS APPLICATIONS

- Customer Engagement
- Patient Diagnostics
- Fraud Detection
- Quality Assurance
- Industrial Automation
- Precision Marketing
- Molecular Simulations
- ++

NGC

SOFTWARE HUB

HELM

Certified Containers

Pre-trained Models

SDKs

NVIDIA CERTIFIED

VALIDATED SOLUTIONS

APPLICATION FRAMEWORKS

- SMART CITY — *Metropolis*
- CONVERSATIONAL AI — *Riva*
- AUTONOMOUS VEHICLES — *Drive*
- RECOMMENDATION SYSTEMS — *Merlin*
- HEALTHCARE — *Clara*
- CYBERSECURITY — *Morpheus*
- ++ …

DEVELOPER TOOLKITS

- ML & DATA ANALYTICS — RAPIDS, dmlc XGBoost
- AI TRAINING & INFERENCE — TensorFlow, TensorRT, PYTORCH, mxnet
- HIGH PERFORMANCE COMPUTING — *NVIDIA HPC SDK*
- RENDERING & VISUALIZATION — *IndeX*, *OptiX*, *CloudXR*, *MDL*

ACCELERATION LIBRARIES

- COMPUTE — *CUDA-X*
- NETWORKING, STORAGE & SECURITY — *DOCA*, *MAGNUM IO*

SERVERS & CLOUD

- DGX, HGX — Purpose Built
- EGX — Mainstream & Edge
- CSP Instances

HARDWARE TECHNOLOGIES

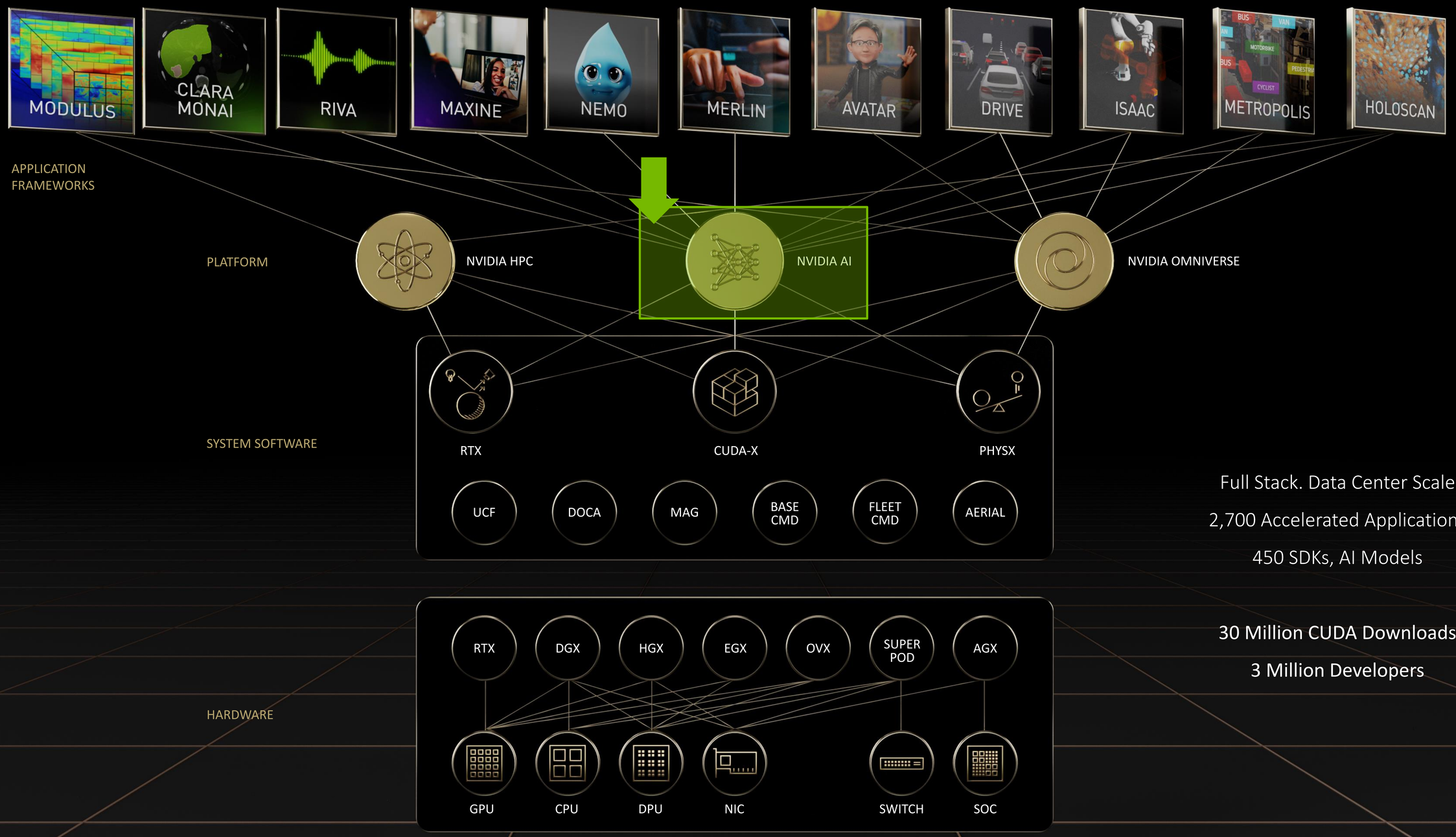
- GPU
- NVSwitch
- BF DPU
- SMART NIC
- NVIDIA Switch

OPERATIONS

- VIRTUAL GPU SW
- TRITON INFERENCE SERVER
- FLEET COMMAND
- NVIDIA GPU Operator
- Red Hat
- vmware

MONITORING

- DCGM
- Grafana
- Prometheus
- UFM



Full Stack. Data Center Scale

2,700 Accelerated Applications

450 SDKs, AI Models

30 Million CUDA Downloads

3 Million Developers

AI / DEEP LEARNING TRENDS

AI and Machine Learning trends for 2022

1. Modern Data Platform
2. Data-centric AI
3. Usage of AI and ML for Cybersecurity Applications
4. Automated Machine Learning (Hyper Automation)
5. Full-stack Deep Learning
6. NLP and AI/ML
7. Model Scale Up
8. Multi-task learning
9. Multi-modal learning
10. Graph intelligence
11. AI-enabled conceptual design
12. AI-based simulation & synthetic data
13. Digital twins
14. Self Driving vehicles and AI

…

Data	Magnum IO/ RAPIDS/ nvTabular/ cuNumeric/ Morpheus/ …
MLops	Base Command/ DeepOps/ Bright Computing/ GPU(or Network)Operator/ …
Big NLP	MegatronLM/ Nemo-Megatron/ Riva/ Merlin/ …
New approach & application	DGL with GPU/ …
Metaverse	Omniverse/ ISAAC/ Drive/ …

NVIDIA END-TO-END AI SOFTWARE STACK

Deep Learning Streamlined from Conception to Production at Scale

PRE-TRAINED MODELS

Automatic Speech Recognition
Image Classification
Object Detection
Language Modelling

Speech Synthesis
Recommender Systems
Machine Translation
Image Segmentation

DATA PREPERATION

RAPIDS

Accelerated with NVIDIA

TRAIN AT SCALE

PyTorch TensorFlow

OPTIMIZED FOR INFERENCE

TensorRT

DEPLOY AT SCALE

NVIDIA TRITON INFERENCE SERVER

NVIDIA END-TO-END AI SOFTWARE STACK

Deep Learning Streamlined from Conception to Production at Scale

PRE-TRAINED MODELS

Automatic Speech Recognition
Image Classification
Object Detection
Language Modelling

Speech Synthesis
Recommender Systems
Machine Translation
Image Segmentation

DATA PREPERATION

RAPIDS

Accelerated with NVIDIA

TRAIN AT SCALE

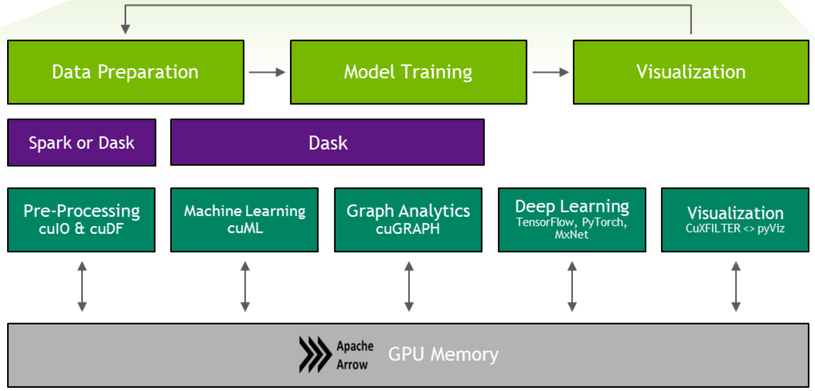
PyTorch TensorFlow

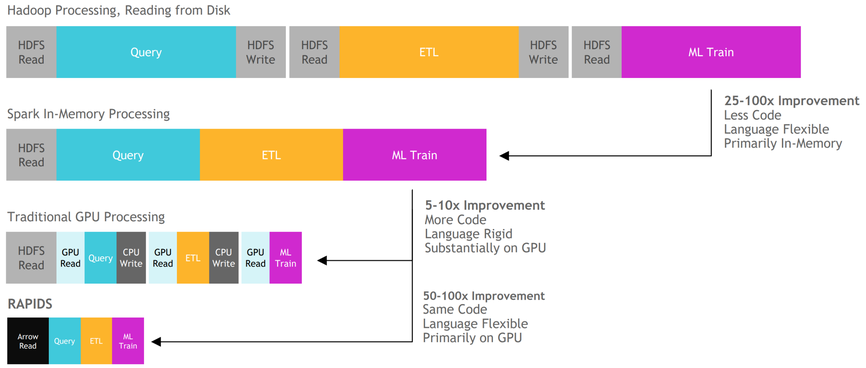
OPTIMIZED FOR INFERENCE

TensorRT

DEPLOY AT SCALE

NVIDIA TRITON INFERENCE SERVER





NVIDIA

NVIDIA END-TO-END AI SOFTWARE STACK

Deep Learning Streamlined from Conception to Production at Scale

PRE-TRAINED MODELS

Automatic Speech Recognition
Image Classification
Object Detection
Language Modelling

Speech Synthesis
Recommender Systems
Machine Translation
Image Segmentation

DATA PREPERATION

RAPIDS

Accelerated with NVIDIA

TRAIN AT SCALE

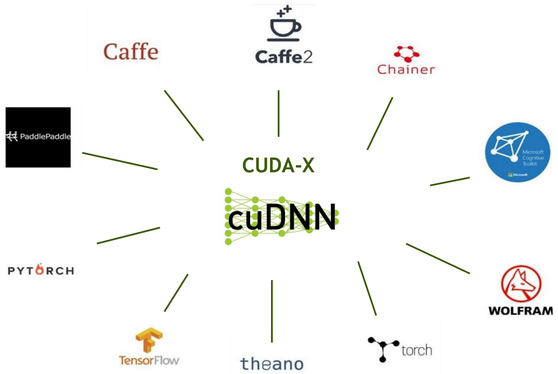
PyTorch TensorFlow

OPTIMIZED FOR INFERENCE

TensorRT

DEPLOY AT SCALE

NVIDIA TRITON INFERENCE SERVER



NVIDIA END-TO-END AI SOFTWARE STACK

Deep Learning Streamlined from Conception to Production at Scale

PRE-TRAINED MODELS

Automatic Speech Recognition
Image Classification
Object Detection
Language Modelling

Speech Synthesis
Recommender Systems
Machine Translation
Image Segmentation

DATA PREPERATION

RAPIDS

Accelerated with NVIDIA

TRAIN AT SCALE

PyTorch TensorFlow

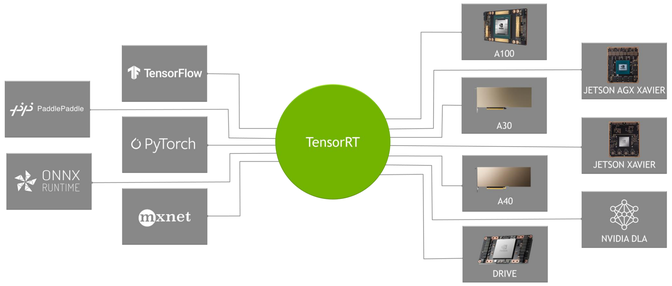
OPTIMIZED FOR INFERENCE

TensorRT

DEPLOY AT SCALE

NVIDIA TRITON INFERENCE SERVER

Trained DNN → TensorRT Optimizer → TensorRT Runtime



NVIDIA END-TO-END AI SOFTWARE STACK

Deep Learning Streamlined from Conception to Production at Scale

PRE-TRAINED MODELS

Automatic Speech Recognition
Image Classification
Object Detection
Language Modelling

Speech Synthesis
Recommender Systems
Machine Translation
Image Segmentation

DATA PREPERATION

RAPIDS

Accelerated with NVIDIA

TRAIN AT SCALE

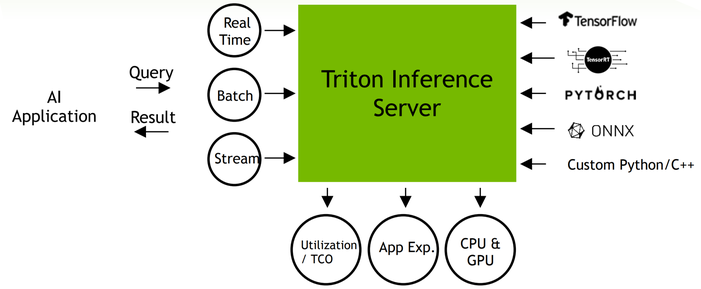
PyTorch TensorFlow

OPTIMIZED FOR INFERENCE

TensorRT

DEPLOY AT SCALE

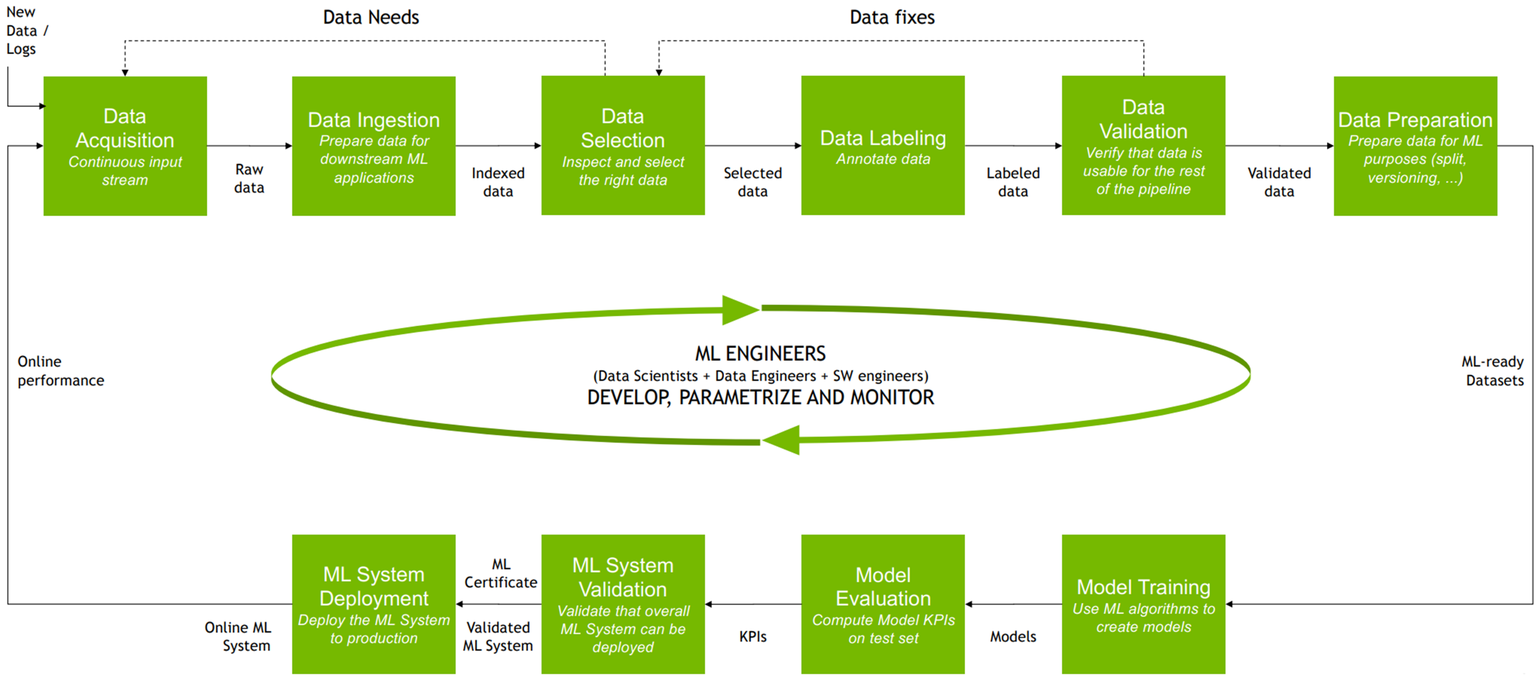
NVIDIA TRITON INFERENCE SERVER

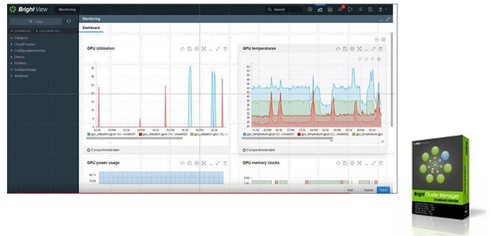


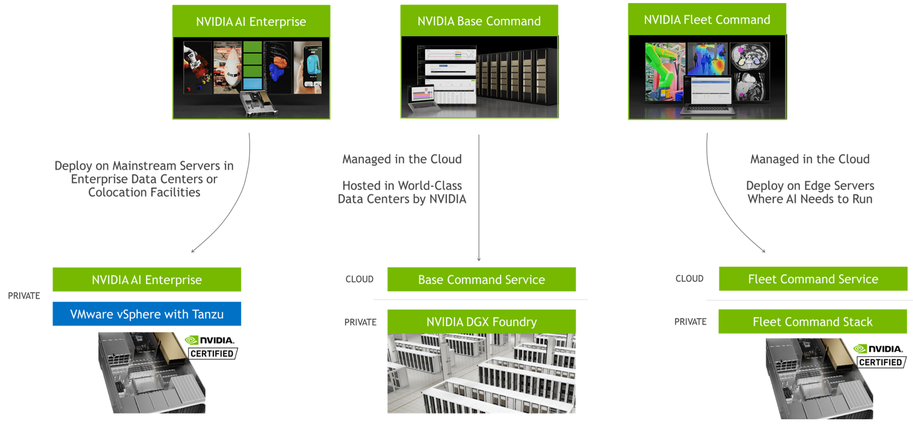
NVIDIA

NVIDIA END-TO-END AI SOFTWARE STACK

NVIDIA solutions for MLOps

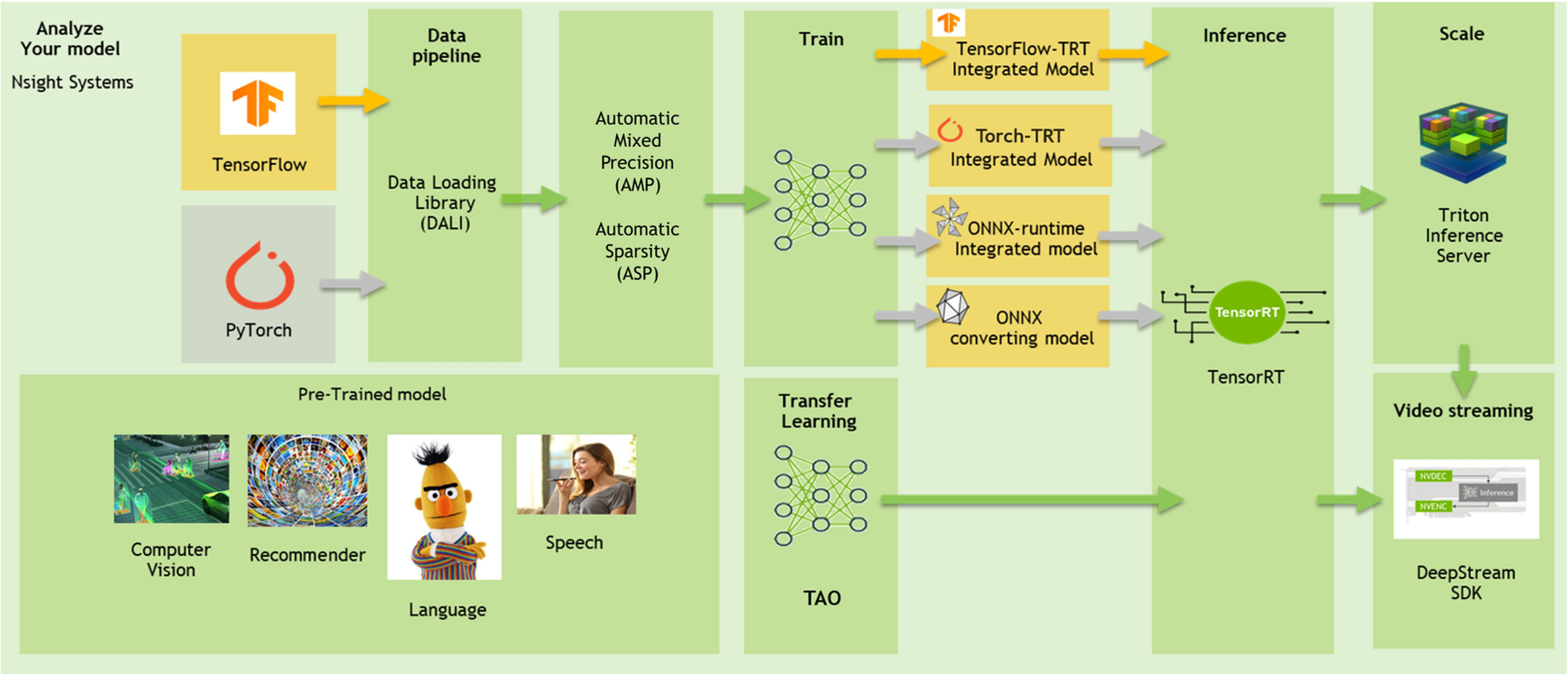






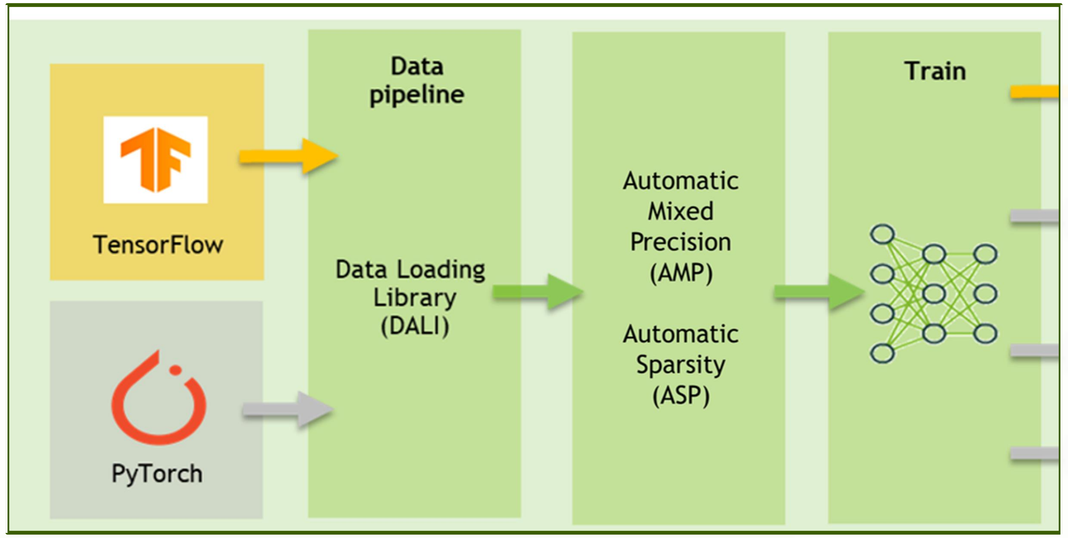
NVIDIA END-TO-END AI SOFTWARE STACK

Deep Learning Streamlined from Conception to Production at Scale



NVIDIA END-TO-END AI SOFTWARE STACK

Deep Learning Streamlined from Conception to Production at Scale



+ QAT

+ Distributed Training

NVIDIA AI FOR DL TRAINING

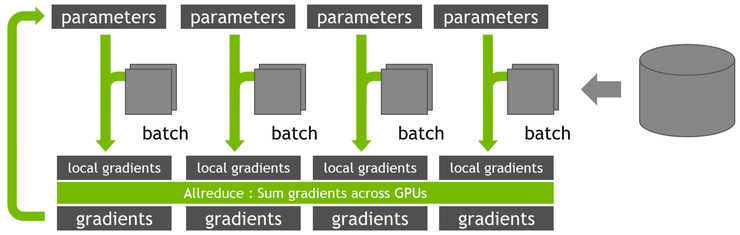
Deep Learning training optimization with NVIDIA AI

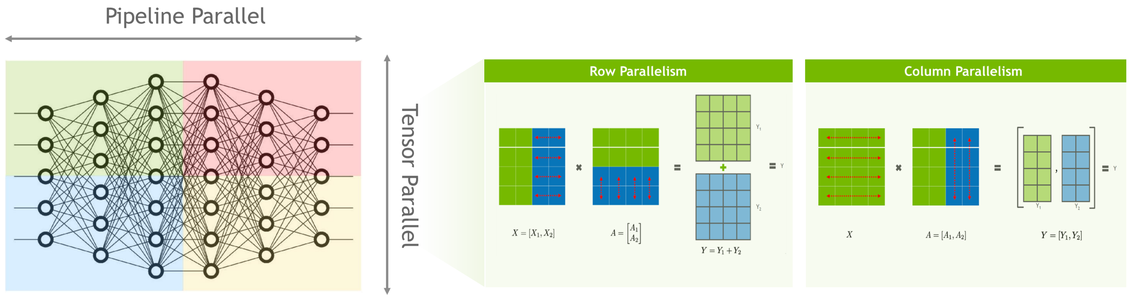
NVIDIA APEX DDP

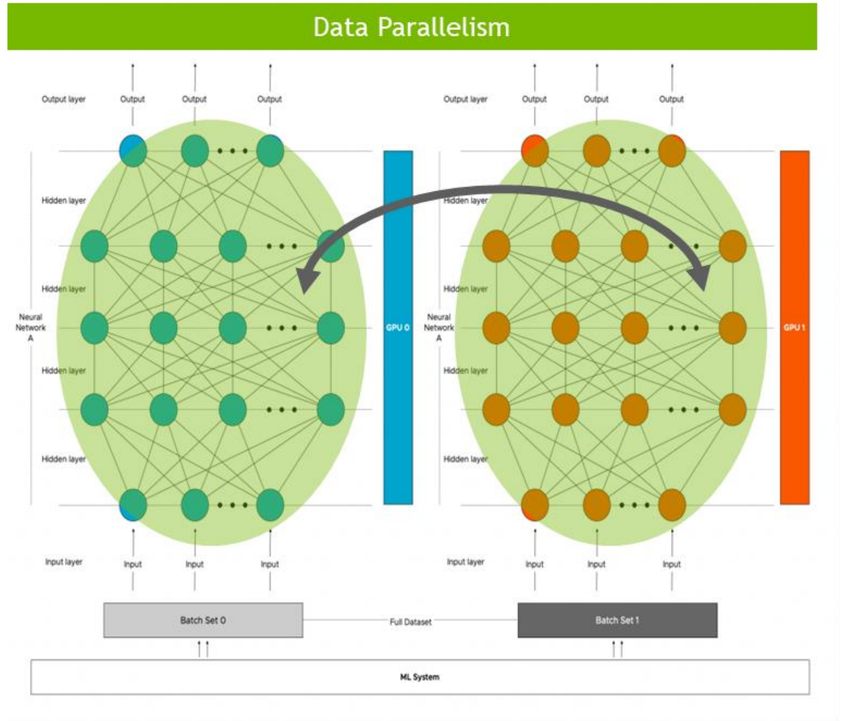
Distributed Data Parallel for Deep Learning Training

DEEP LEARINNG TRAINING WITH MULTI-GPU

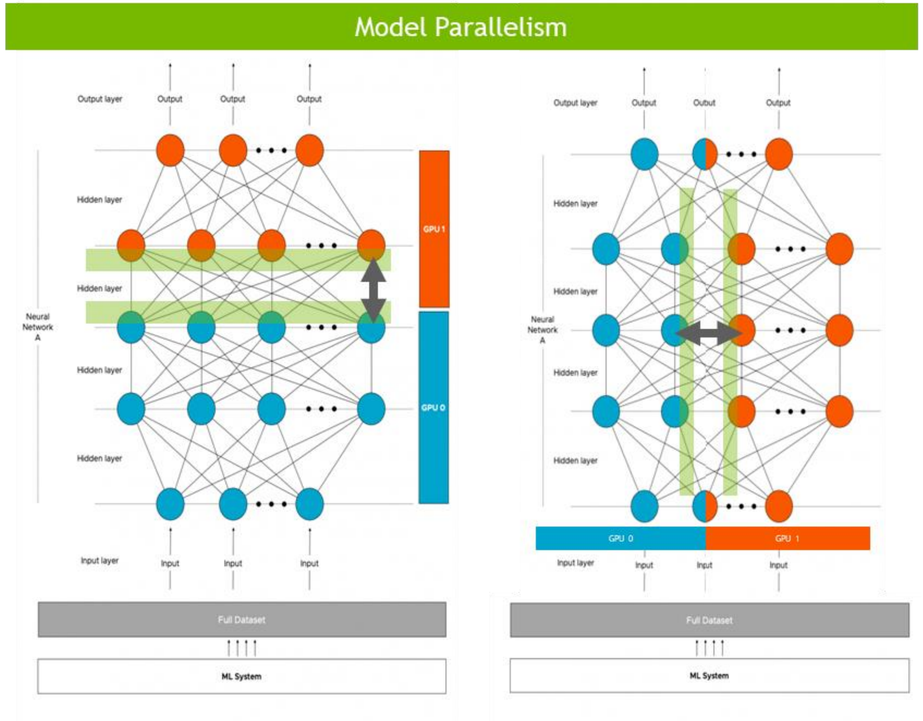
Parallelism for Deep Learning Training



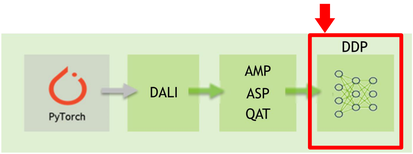




Gradients & Weight parameters



Activations & Loss information



DEEP LEARINNG TRAINING WITH MULTI-GPU

NVIDIA APEX Distributed Data Parallel

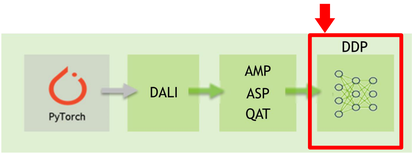
Why *APEX DDP?

- comes with all the distributed training features of the built-in PyTorch DDP
- better performance than built-in DDP
- reducing memory storage/bandwidth demands by 2x
- use larger batch sizes
- take advantage of NVIDIA Tensor Cores for matrix multiplications and convolutions
- don't need to explicitly convert your model, or the input data, to half().

APEX DDP Tips!

- Using Torch distributed launch module
  --nproc_per_node: number of GPUs on the current node, each process is bound to a single GPU
  --master_addr: IP address for the master node of your choice. type str
  --master_port: open port number on the master node. type int. if you don't know, use 8888
  * $ python -m torch.distributed.launch --nproc_per_node=NUM_GPUS
- nccl backend is currently the fastest and highly recommended backend to be used with distributed training and this applies to both single-node and multi-node distributed training.
- Torch DDP permits multiple GPUs per process, with Torch DDP you are required to manually specify the device to run on and the output device. With Apex DDP, it uses only the current device by default
- For Data loading workers, this is different from the processes that run the program on each GPU. the total # of processes = # of data loading workers + # of GPUs (one process to run each GPU)
- For per GPU batch size, it needs to be a multiple of 8 to make use of Tensor Cores.

*APEX - A PyTorch Expension



DEEP LEARINNG TRAINING WITH MULTI-GPU

NVIDIA APEX Distributed Data Parallel

How to use the APEX DDP? (Code)

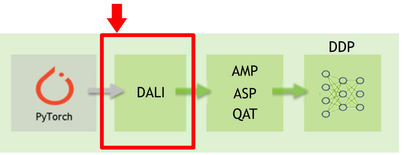
```
import torch
import torch.distributed as dist
import apex
from apex.parallel import DistributedDataParallel as DDP
…

args.gpu = args.local_rank
torch.cuda.set_device(args.gpu)
torch.distributed.init_process_group(backend='nccl', init_method='env://')
# this is the total # of GPUs across all nodes
# if using 2 nodes with 4 GPUs each, world size is 8
args.world_size = torch.distributed.get_world_size()
…

model = models.resnet50(pretrained=True)
if args.sync_bn:
    import apex
    print("using apex synced BN")
    model = apex.parallel.convert_syncbn_model(model)
model.cuda()
…
```

```
# Scale init learning rate based on global batch size
args.lr = args.lr * float(args.batch_size*args.world_size)/256.
optimizer = torch.optim.SGD(model.parameters(), args.lr,
                            momentum=args.momentum, weight_decay=args.weight_decay)
model = DDP(model, delay_allreduce=True)
criterion = nn.CrossEntropyLoss().cuda()
…

train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset)
val_sampler = torch.utils.data.distributed.DistributedSampler(val_dataset)
…
model.train()
…

while input is not None:
    output = model(input)
    loss = criterion(output, target)
    loss.backward()
    optimizer.step()
    …
    if torch.distributed.get_rank() == 0:
        is_best = val_top1 > best_prec1
        best_prec1 = max(val_top1, best_prec1)
        save_checkpoint({
            'epoch': epoch + 1, 'arch': args.arch,
            'state_dict': model.state_dict(),
            'best_prec1': best_prec1, 'optimizer': optimizer.state_dict(),
        }, is_best, writer.log_dir)
```

*APEX – A PyTorch Expension

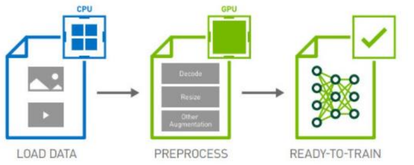
NVIDIA DALI

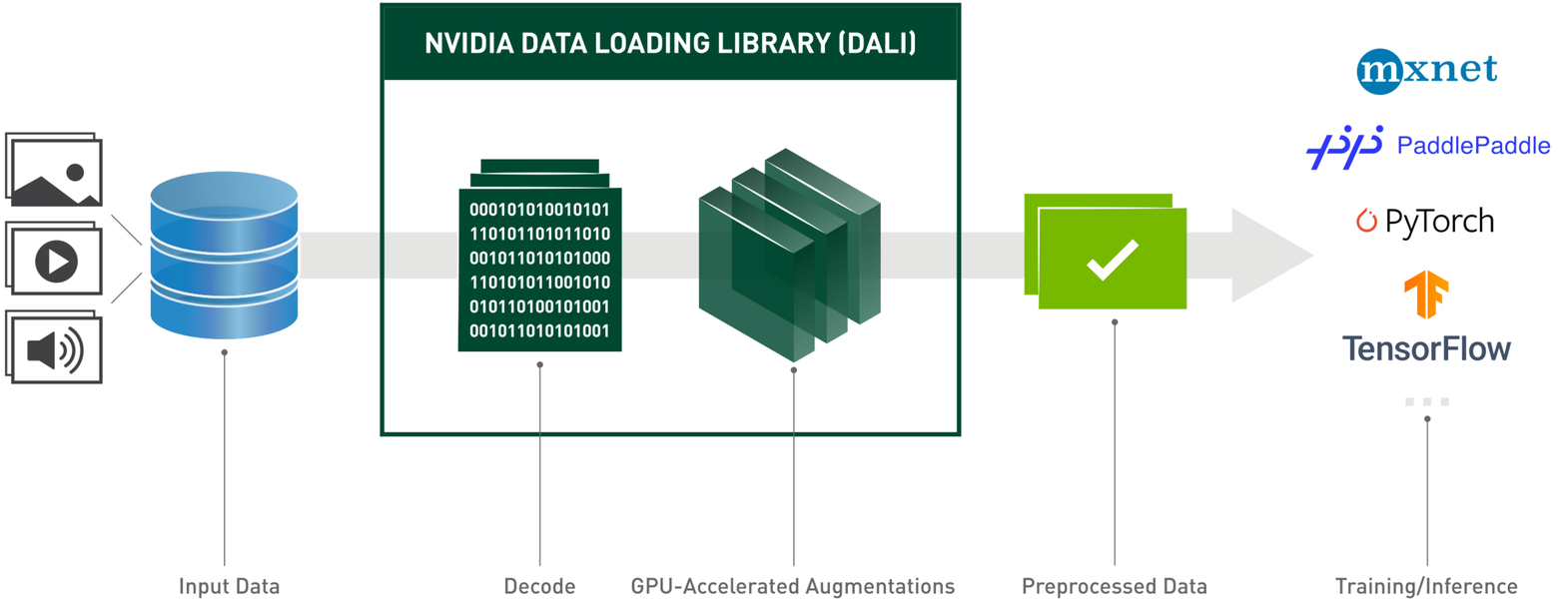
Data Loading Library for Deep Learning Training

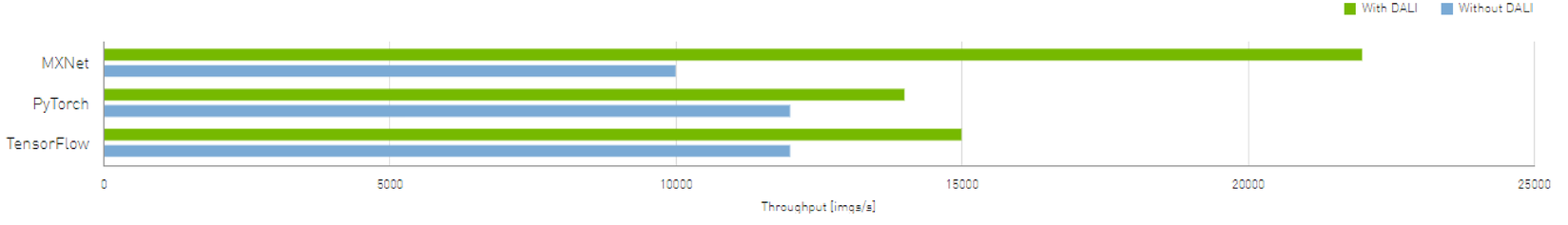


NVIDIA DALI

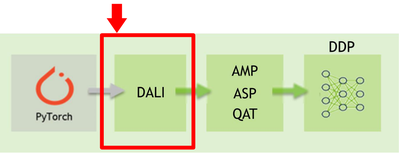
DAta Loading Library (DALI) for Fast Data loading & Pre-processing







ResNet50 training on ImageNet | NVIDIA DGX-2 | 20.01 NGC container | DALI version 0.18 | 16 × V-100 GPUs | batch size: 256



NVIDIA DALI

DAta Loading Library (DALI) for Fast Data loading & Pre-processing

	Without DALI	With DALI

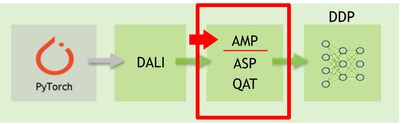
PYTORCH	`train_loader = torch.utils.data.DataLoader(...)`
	`prefetcher = data_prefetcher(train_loader)`
	
	`i = 0`
	`input, target = prefetcher.next()`
	`while input is not None:`
	`  (...)`
	`  i += 1`
	`  input, target = prefetcher.next()`
		`dali_pipe = TrainPipe(...)`
		`train_loader = DALIClassificationIterator(dali_pipe)`
		
		`for i, data in enumerate(train_loader):`
		`  input = data[0]["data"]`
		`  target = data[0]["label"].squeeze()`
		`  (...)`

TensorFlow	`def get_data():`
	`  ds = tf.data.Dataset.from_tensor_slices(files)`
	`  ds = ds.apply(transform1)`
	`  ds = ds.apply(transform2)`
	`  return ds`
	
	`classifier.train(input_fn=get_data,...)`
		`def get_data():`
		`  dali_pipe = TrainPipe(...)`
		`  with tf.device("/gpu:0"):`
		`  ds = dali_tf.DALIDataset(pipeline = dali_pipe,  device_id = 0, ...)`
		`  return ds`
		
		`classifier.train(input_fn=get_data,...)`

Operation Reference — NVIDIA DALI 1.15.0 documentation

NVIDIA APEX AMP

Automatic mixed precision for Deep Learning Training



NVIDIA APEX AMP

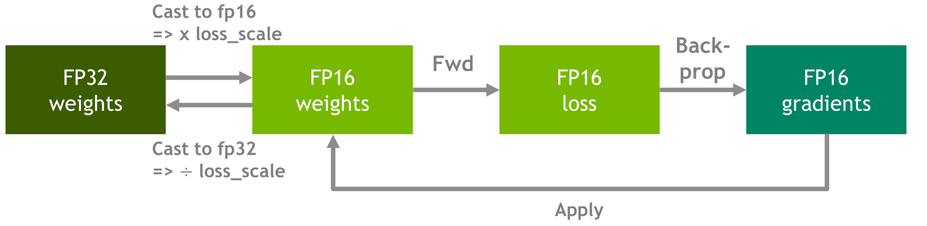
Automatic Mixed Precision for Training

Goal of AMP

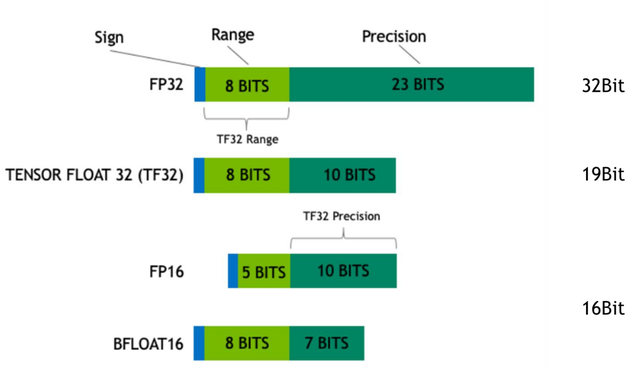
- Maximizing Model training speed with memory efficiency

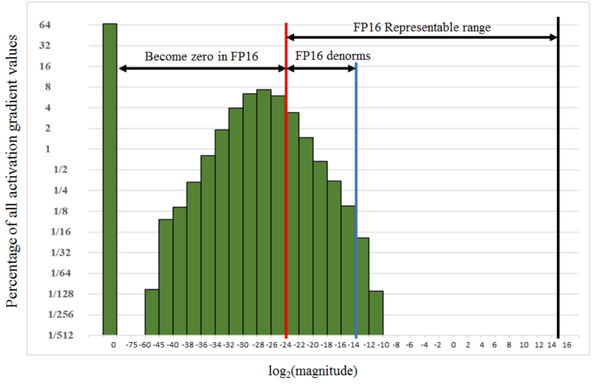
What is the AMP

- A tool to enable Tensor Core-accelerated training
- Automatically, apply FP16 precision to GEMM like layers
  and keep FP32 for Loss, norms, softmax like layers for preventing accuracy drop

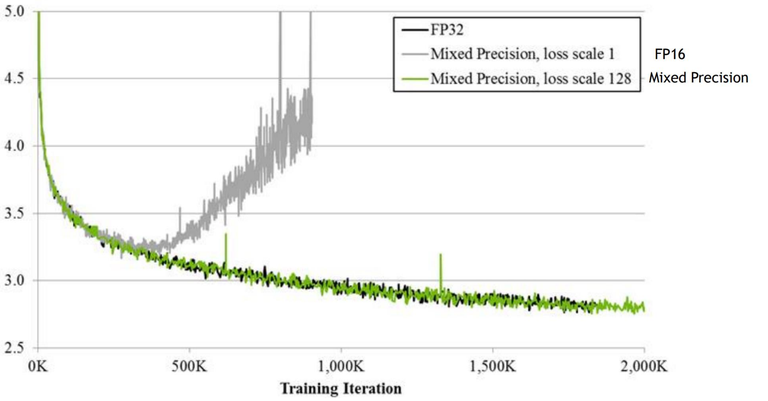


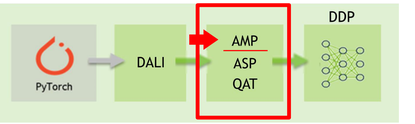
Why we use AMP rather than fp16 training





FP32		FP16
0.0000001	→	0
Near Zero		Zero





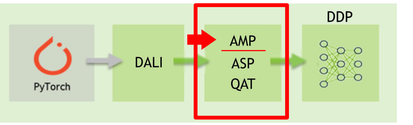
NVIDIA APEX AMP

Automatic Mixed Precision for Training

How to use the APEX AMP? (Code)

```
import torch
Import shutil
from  apex import amp
…

optimizer = torch.optim.SGD(model.parameters(), args.lr,
                    momentum=args.momentum, weight_decay=args.weight_decay)

If args.amp:
    model, optimizer = amp.initialize(model, optimizer, opt_level=args.opt_level,
                                      keep_batchnorm_fp32=args.keep_batchnorm_fp32,
                                      loss_scale=args.loss_scale
                                  )

If args.distributed:
    model = DDP(model, delay_allreduce=True)
…

while input is not None:
    output = model(input)
    loss = criterion(output, target)
    …

    If args.amp:
        with amp.scale_loss(loss, optimizer) as scaled_loss:
            scaled_loss.backward()
        torch.nn.utils.clip_grad_norm_(amp.master_params(optimizer), max_norm)
        # or torch.nn.utils.clip_grad_value_(amp.master_params(optimizer), max_)
    else:
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm)

    optimizer.step()
```

```
def save_checkpoint(model, optimizer, amp, epoch, best_prec1, is_best, …, out_dir):
    …
    checkpoint = {    'epoch': epoch + 1,
                      'arch': args.arch,
                      ‘state_dict': model.state_dict(),
                      'best_prec1': best_prec1,
                      'optimizer' : optimizer.state_dict()}
    …
    if args.amp:
        checkpoint[‘amp’] = amp.state_dict()

    torch.save(checkpoint, outdir+’/model_{}.pth’.format(epoch))

    if is_best:
        shutil.copyfile(filename, best + ‘model_best.pth’)
    …


def load_checkpoint(model, optimizer, amp, model_path, …):
    checkpoint = torch.load( model_path,
                             map_location = lambda storage,
                             loc: storage.cuda(args.gpu)
    best_prec1 = checkpoint['best_prec1’]
    start_epoch = checkpoint[‘epoch’] + 1
    model.load_state_dict(checkpoint['state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer’])
    …

    if args.amp:
        amp.load_state_dict(checkpoint[‘amp’])

    return start_epoch, best_prec1
```



NVIDIA APEX AMP

Automatic Mixed Precision for Training

How to use the APEX AMP? (Code)

```
import torch
from  apex import amp
…

optimizer = torch.optim.SGD(model.parameters(), args.lr,
                    momentum=args.momentum, weight_decay=args.weight_decay)

If args.amp:
    model, optimizer = amp.initialize(model, optimizer, opt_level=args.opt_level,
                                      keep_batchnorm_fp32=args.keep_batchnorm_fp32,
                                      loss_scale=args.loss_scale
                                     )

If args.distributed:
    model = DDP(model, delay_allreduce=True)
…

while input is not None:
    output = model(input)
    loss = criterion(output, target)
    …

    If args.amp:
        with amp.scale_loss(loss, optimizer) as scaled_loss:
            scaled_loss.backward()
    else:
        loss.backward()

        torch.nn.utils.clip_grad_norm_(amp.master_params(optimizer), max_norm)
        # or torch.nn.utils.clip_grad_value_(amp.master_params(optimizer), max_)

    optimizer.step()
```

OPT_LEVEL="O0"

FP32 training.
Your incoming model should be FP32 already, so this is likely a no-op. O0 can be useful to establish an accuracy baseline.

O1

Mixed Precision.
Patches Torch functions to internally carry out Tensor Core-friendly ops in FP16, and ops that benefit from additional precision in FP32. Also uses dynamic loss scaling. **Because casts occur in functions, model weights remain FP32.**

O2

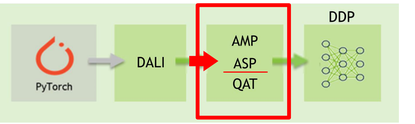
"Almost FP16" Mixed Precision.
FP16 model and data with FP32 batchnorm, FP32 master weights, and dynamic loss scaling. **Model weights, except batchnorm weights, are cast to FP16.**

O3

FP16 training.
O3 can be useful to establish the "speed of light" for your model. If your model uses batch normalization, add `keep_batchnorm_fp32=True`, which enables cudnn batchnorm.

NVIDIA APEX ASP

Automatic SParsity for Deep Learning Training



NVIDIA APEX ASP

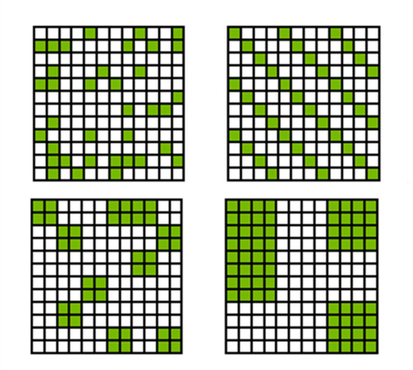
Automatic Sparsity for Training

Goal of ASP

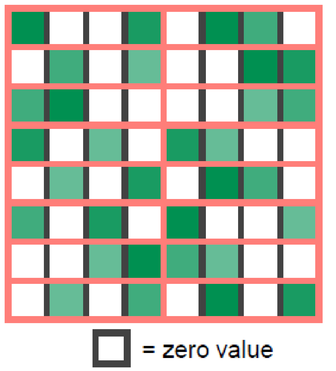
- Maximizing Model training & inference speed with compute & memory efficiency, accuracy keeping

What is the ASP

- A tool to enable sparse training and inference for PyTorch models by adding 2 lines with trying to keep accuracy
  It works because:
  - ✓ Large models are prone to capturing the noise in the training dataset.
  - ✓ Activations may have zeros because of ReLU.
- Apply 2:4 Fine grained sparsity for Ampere TensorCore
- Applicable for:
  - ✓ Conv, Linear (ConvNet, Transformer blocks, MLP, etc.)
    - Input: dense activations, sparse weights(with 2bit per-valued mask)
    - Output: dense activations



structured vs unstructured, granularity



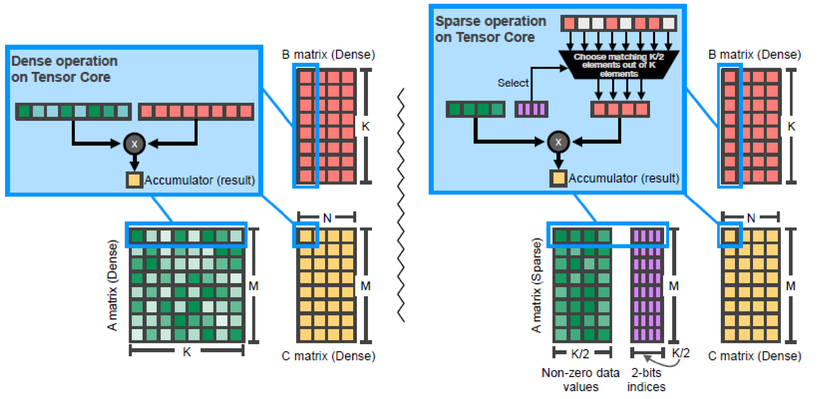
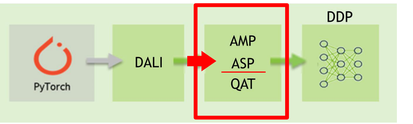


Fig. 2. Mapping a M×N×K GEMM onto a Tensor Core. Dense matrix A, of size M×K, (left side) becomes M×$\frac{K}{2}$ (right side) after pruning with 2:4 sparsity. Sparse Tensor Core hardware selects only the elements from B that correspond to the nonzero values in A, skipping the unnecessary multiplications by zero. In both dense and sparse GEMMs, B and C are dense K×N and M×N matrices, respectively.



NVIDIA APEX ASP

Automatic Sparsity for Training

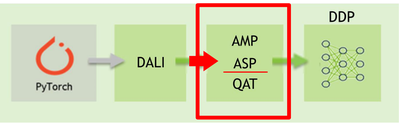
How to use the APEX ASP?

- Generate a 2:4 structured sparse model which can utilize Ampere's new features.
- Support sparse gradient backward, to study/finetune a structured sparse mode with two lines of code

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

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

optimizer = optim.SGD(model.parameters(), lr=0.01,

2 ASP.prune_trained_model(model, optimizer)

x, y = DataLoader(…) #load data samples and labels
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')
```

```
# Add mask buffers to model
ASP.init_model_for_pruning(model,
         mask_calculator="m4n2_1d",
         verbosity=2,
         whitelist=[torch.nn.Linear, torch.nn.Conv2d],
         allow_recompute_mask=False)

# Augment optimizer, p:weights
# p.grad.mul_(mask)
# p.mul_(mask)
ASP.init_optimizer_for_pruning(optimizer)


# compute masks, and fill the mask buffers
ASP.compute_sparse_masks()
```

```
bert.encoder.layer.0.attention.self.query.weight
bert.encoder.layer.0.attention.self.query.bias
bert.encoder.layer.0.attention.self.query.__weight_mma_mask
```

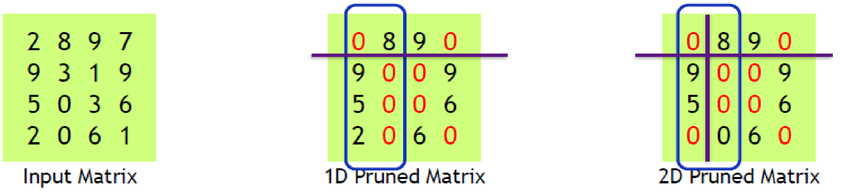


NVIDIA APEX ASP

Automatic Sparsity for Training

How to use the APEX ASP?

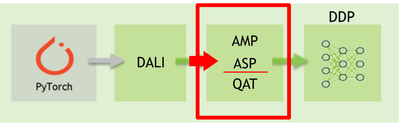
- Matrix Pruning (1D vs. 2D) in 2:4 sparse pattern.
- 1D: List all valid pruned pattern to find out max L1-norm pruned matrix.
- 2D: More stringent requirements, maybe used for sparse training acceleration



1D weights pruning is recommended!

```
# Add mask buffers to model
ASP.init_model_for_pruning(model,
         mask_calculator="m4n2_1d",
         verbosity=2,
         whitelist=[torch.nn.Linear, torch.nn.Conv2d],
         allow_recompute_mask=False)

# Augment optimizer, p:weights
# p.grad.mul_(mask)
# p.mul_(mask)
ASP.init_optimizer_for_pruning(optimizer)


# compute masks, and fill the mask buffers
ASP.compute_sparse_masks()
```

```
bert.encoder.layer.0.attention.self.query.weight
bert.encoder.layer.0.attention.self.query.bias
bert.encoder.layer.0.attention.self.query.__weight_mma_mask
```

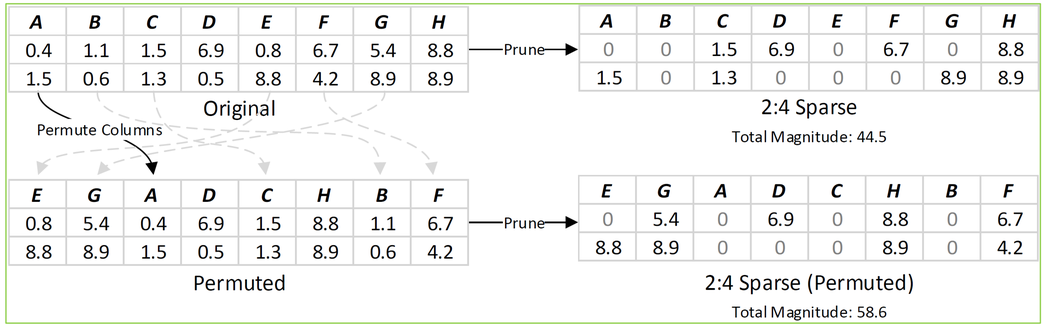


NVIDIA APEX ASP

Automatic Sparsity for Training

How to use the APEX ASP?

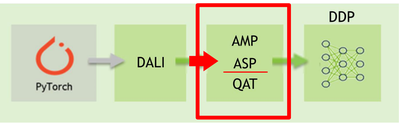
- If you are using lightweight networks,
  The weight distribution is not friendly to the 2:4 structured sparse pruning,
  especially for the so-called sparsity-sensitive layers
  => Try to find a good permutation for the sparsity-sensitive layer to reduce the magnitude loss




```
@classmethod
    def init_model_for_pruning(cls, model, mask_calculator="m4n2_1d",
            verbosity=3,
            whitelist=[torch.nn.Linear, torch.nn.Conv1d, torch.nn.Conv2d, torch.nn.Conv3d],
            allowed_layer_names=None, disallowed_layer_names=[],
            allow_recompute_mask=False, custom_layer_dict={},
            allow_permutation=True):
```

```
# Add mask buffers to model
ASP.init_model_for_pruning(model,
         mask_calculator="m4n2_1d",
         verbosity=2,
         whitelist=[torch.nn.Linear, torch.nn.Conv2d],
         allow_recompute_mask=False)

# Augment optimizer, p:weights
# p.grad.mul_(mask)
# p.mul_(mask)
ASP.init_optimizer_for_pruning(optimizer)


# compute masks, and fill the mask buffers
ASP.compute_sparse_masks()
```

```
bert.encoder.layer.0.attention.self.query.weight
bert.encoder.layer.0.attention.self.query.bias
bert.encoder.layer.0.attention.self.query.__weight_mma_mask
```

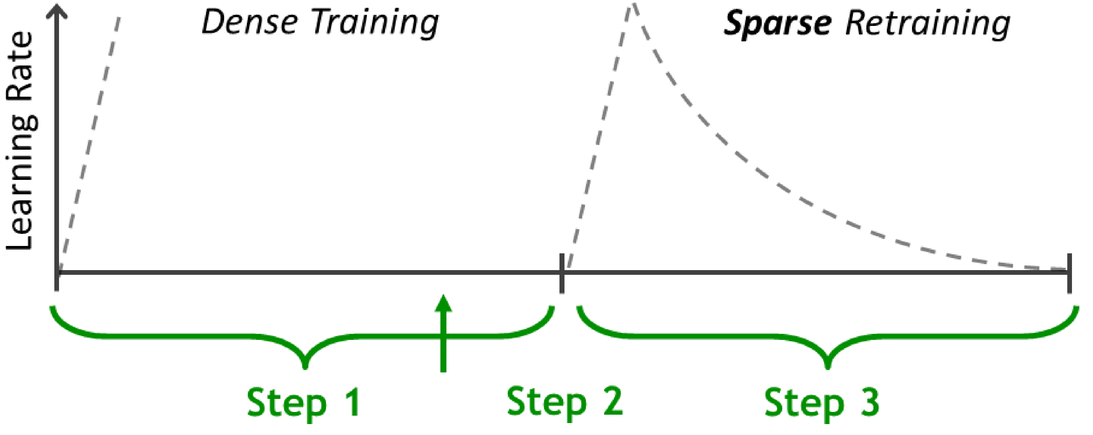


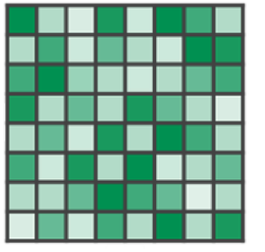
NVIDIA APEX ASP

Automatic Sparsity for Training

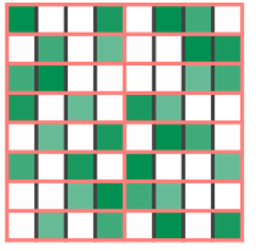
How to use the APEX ASP?

- Training recipe for ASP
  I. Train (or obtain) a dense network
  II. Prune for 2:4 sparsity
  III. 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)

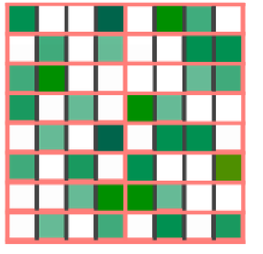




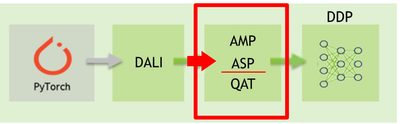
Dense weights



2:4 sparse weights



Retrained 2:4 sparse weights



NVIDIA APEX ASP

Deployment with TensorRT 8

How to use the APEX ASP?

- Deploying with TensorRT

trtexec--sparsity=enable/disable/force

✓ enable: enable sparse tactics if the weights have the right sparsity pattern

✓ force: enable sparse tactics in the builder and force-overwrite the weights to have a sparsity pattern.

TensorRT/samples/common/sampleEngines.cpp

```
const auto index = getIndex(ki, ci, rsi);
if (nz < nonzeros)
{
    sparseValues[index] = values[index];
    ++nz;
}
else
{
    sparseValues[index] = 0;
}
```

Force-overwrite: sparsity

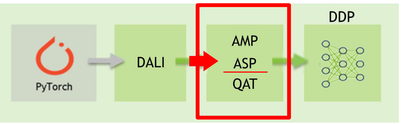
Set kSPARSE_WEIGHTS flag in the IBuilderConfig.

```
builder_config.set_flag(
    trt.BuilderFlag.SPARSE_WEIGHTS)
```

✓ Input model is a sparse model,

✓ TensorRT will profile and choose the best kernels automatically

```
if (build.sparsity != SparsityFlag::kDISABLE)
{
    config.setFlag(BuilderFlag::kSPARSE_WEIGHTS);
    if (build.sparsity == SparsityFlag::kFORCE)
    {
        sparsify(network, sparseWeights);
    }
}
```



NVIDIA APEX ASP

Deployment with TensorRT 8

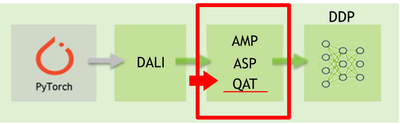
How to use the APEX ASP?

- For CV models
  - ✓ PyTorch checkpoint -> onnx -> TensorRT engine √
  - ✓ Speedup Perf:
    - ➢ trtexec --onnx=./resnet50.onnx --explicitBatch --workspace=1024 --verbose --nvtxMode=verbose --best --sparsity=force --minShapes=input_0:1024x3x224x224 --optShapes=input_0:1024x3x224x224 --maxShapes=input_0:1024x3x224x224 --buildOnly --saveEngine=./onnx_resnet50_int8_sparse_1024.engine
    - ➢ trtexec --loadEngine=./onnx_resnet50_fp16_dense_64.engine --duration=60 --noDataTransfers

    Notes: Throughput may be bound by host-to-device transfers for the inputs rather than GPU Compute and the GPU may be under-utilized. Add --noDataTransfers flag to disable data transfers.

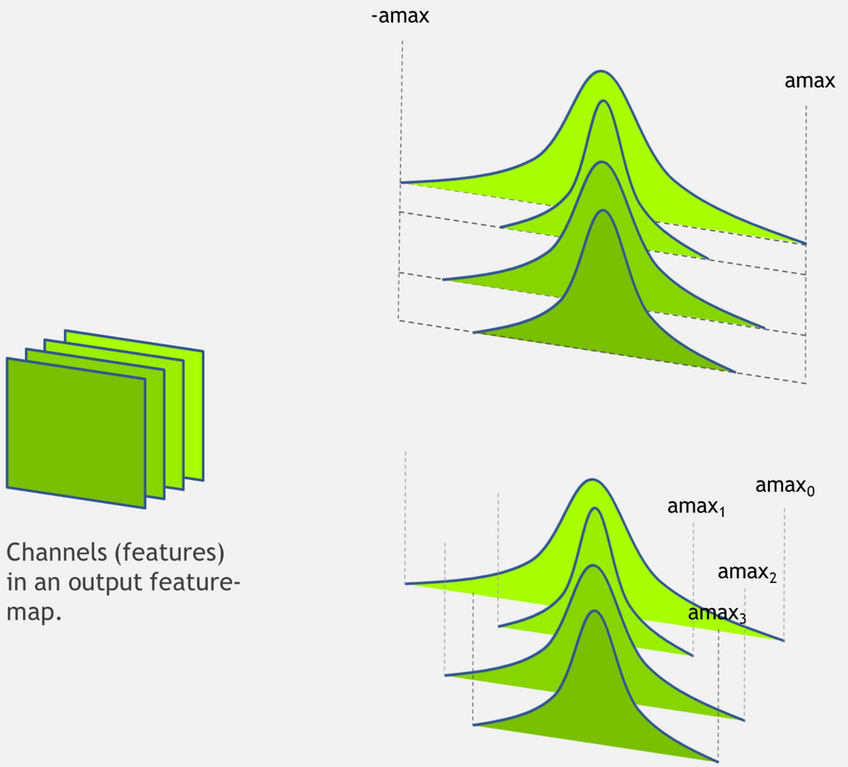
NVIDIA QUANTIZATION

Quantization Aware Training for Efficient Deep Learning inference



NVIDIA QUANTIZATION

Quantization basics



Channels (features) in an output feature-map.

Weights :

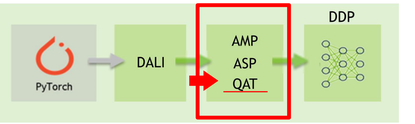
- Use scale quantization with per-column/per-channel granularity
- Use a symmetric integer range for quantization [-127, 127]) and max calibration

Activations :

- Use scale quantization with per-tensor granularity

Calibration :

- Choose calibrators
- Some of params: calib-batch-size, percentile



NVIDIA QUANTIZATION

Three methods of Quantization & Recommended workflow (for INT8 quantization)

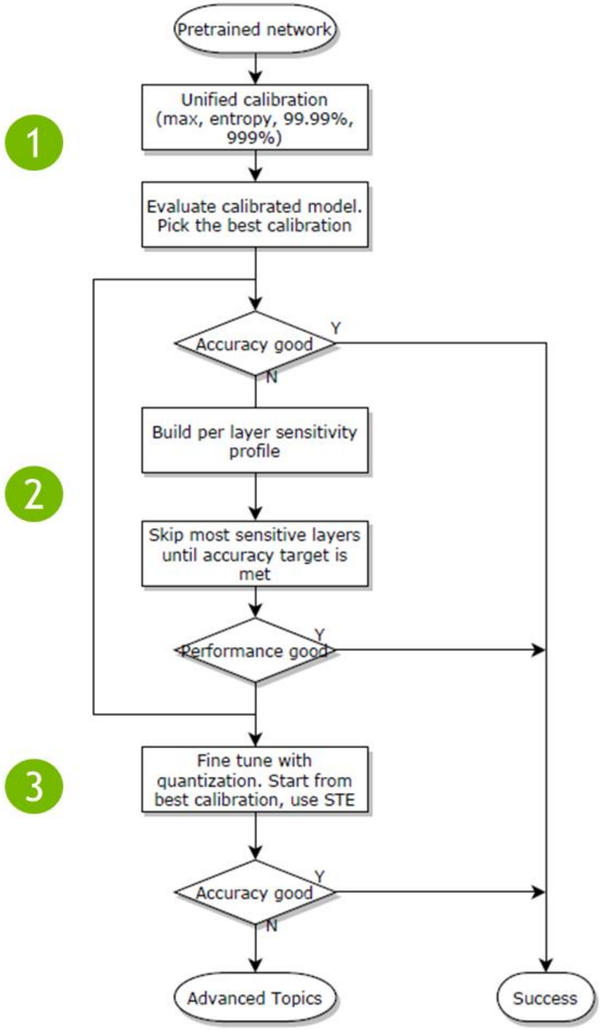


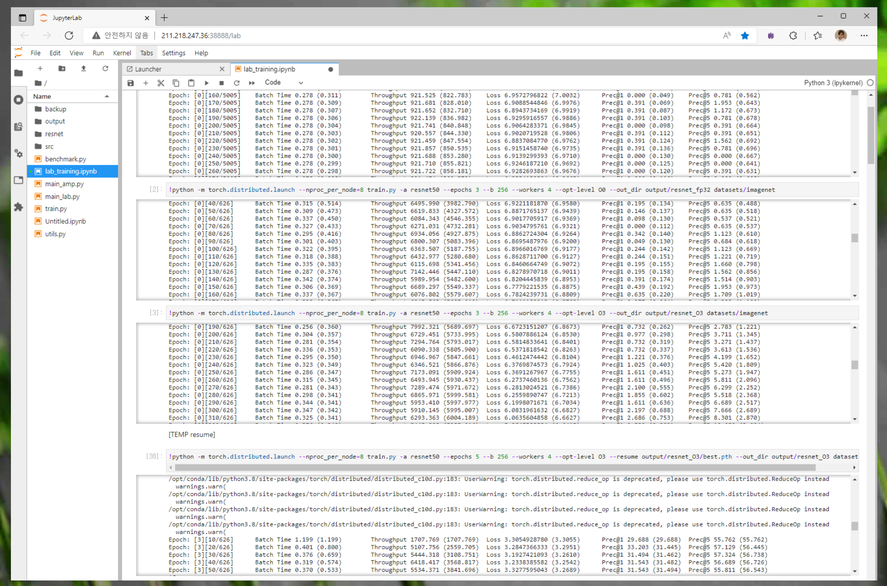
Figure 5: Flow chart of our recommended quantization workflow

1. **Post-Training Quantization(PTQ)**
   - Run activation calibration including max, entropy and 99.999% percentile
   - Try and search the best calibration setting according to the accuracy

2. **Partial quantization**
   - Perform sensitivity analysis
   - Skip sensitive layers and leave them in floating-point, check computational performance and accuracy

3. **Quantization-Aware Training(QAT)**
   - Start from the best calibrated quantized model
   - Use QAT to fine-tune for around 10% of the original training schedule with an annealing learning rate schedule

Let’s Go to the Demo!



HANDS-ON
FOR DEEP LEARNING TRAINING

All you can do for Efficient Deep Learning when Training

Q & A

THANK YOU

NVIDIA®