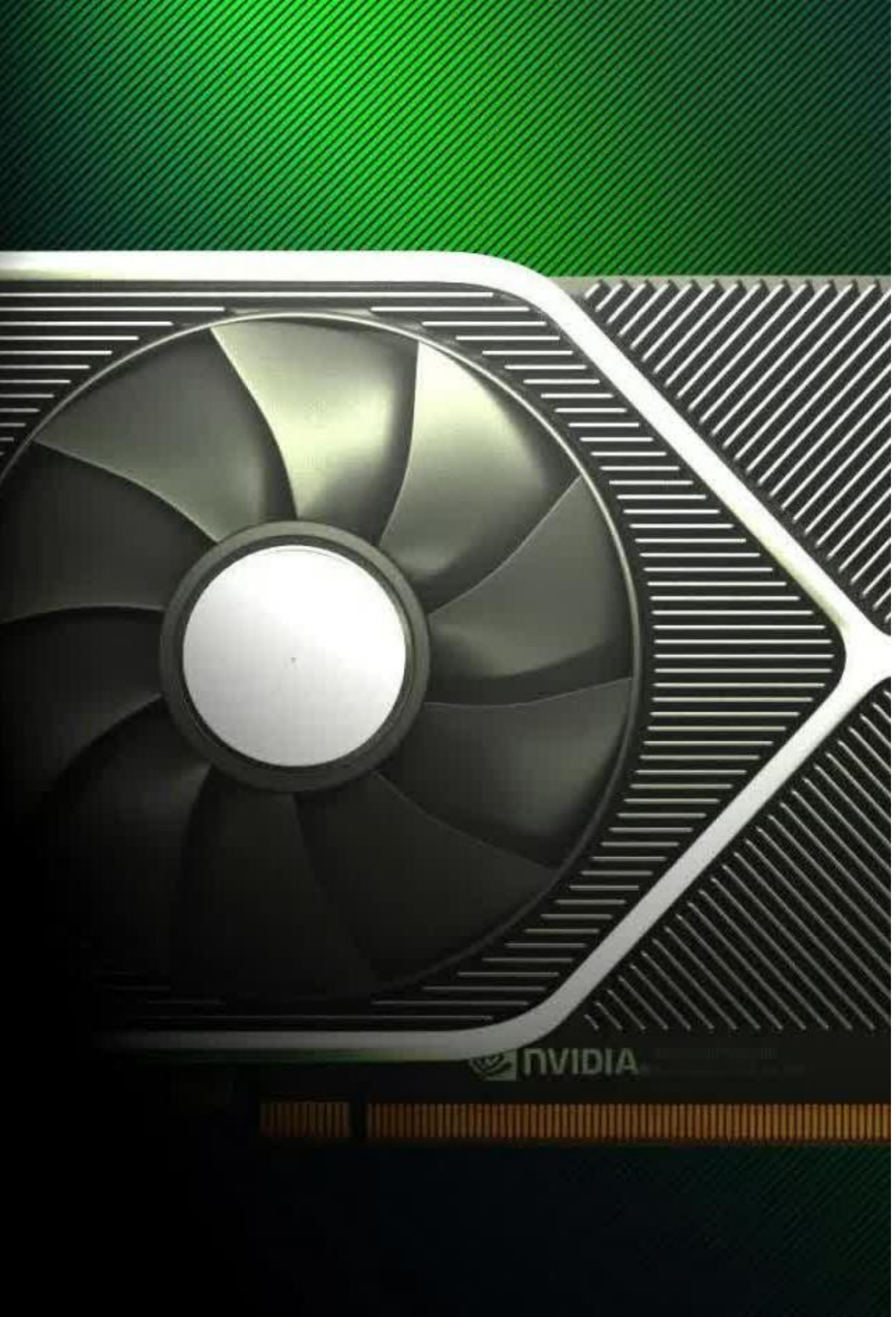
MONAI

GPU accelerated training

Nic Ma@NVIDIA



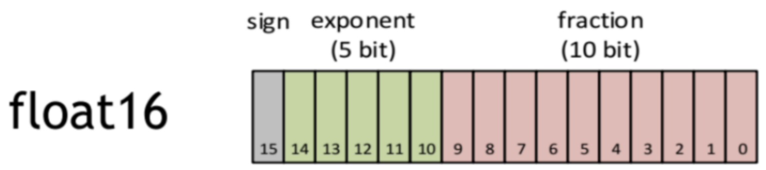
Background

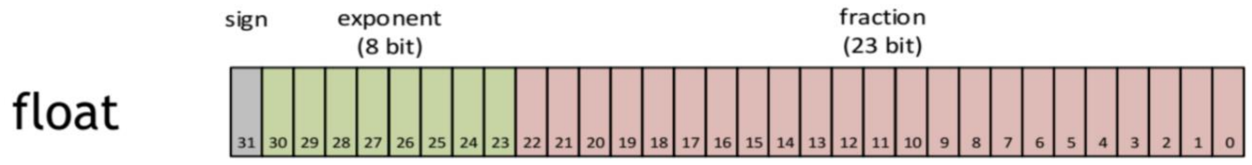
- *NVIDIA GPUs have been widely applied in many areas of deep learning training and evaluation, and the CUDA parallel computation shows obvious acceleration when comparing to traditional computation methods.*
- *To leverage advanced GPU features, popular mechanisms have been introduced, such as **automatic mixed precision** (AMP), **distributed data parallel**, etc. MONAI can support these features and provides rich examples.*
- *But actually, it's not easy to fully leverage GPU resources during training. This presentation introduces how to achieve the **best GPU performance**.*

Auto mixed precision (AMP)

FP32 vs FP16 training

What is FP16 number?



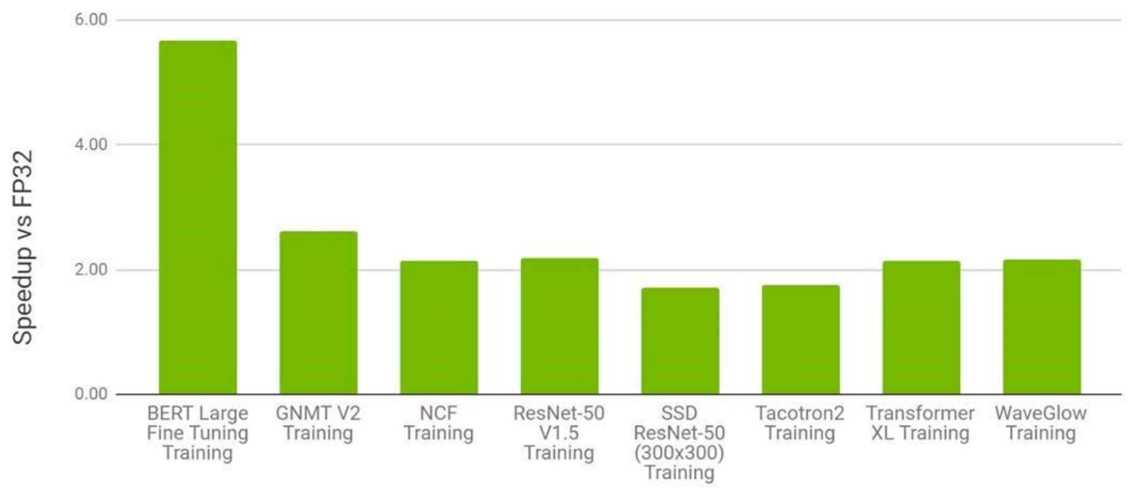


FP16 support in NVIDIA GPUs

- AMP with FP16 is the most performant option for DL training on the V100.
- For various models, AMP on V100 provides a speedup of 1.5x to 5.5x over FP32 on V100 while converging to the same final accuracy.

AMP support in PyTorch

- In 2017, NVIDIA researchers developed a methodology for mixed-precision training, which combined single-precision (FP32) with half-precision (e.g. FP16) format when training a network, and it achieved the same accuracy as FP32 training using the same hyperparameters.
- For the PyTorch 1.6 release, developers at NVIDIA and Facebook moved mixed precision functionality into PyTorch core as the AMP package, torch.cuda.amp.



How to enable AMP?

In PyTorch program vs MONAI workflows

Enable AMP in PyTorch program

- torch.cuda.amp.autocast enables autocasting for chosen regions. Autocasting automatically chooses the precision for GPU operations to improve performance while maintaining accuracy.
- torch.cuda.amp.GradScaler helps perform the steps of gradient scaling conveniently. Gradient scaling improves convergence for networks with float16 gradients by minimizing gradient underflow.
- Typical AMP example:

```python
model = Net().cuda()
optimizer = optim.SGD(model.parameters(), ...)

# Creates a GradScaler once at the beginning of training
scaler = GradScaler()

for epoch in epochs:
    for image, target in data:
        optimizer.zero_grad()

        # Runs the forward pass with autocasting
        with autocast():
            output = model(image)
            loss = loss_fn(output, target)
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()
```

Enable AMP in MONAI workflows

- Just set amp=True/False to enable/disable AMP in MONAI SupervisedTrainer or SupervisedEvaluator.
- Example code for trainer and evaluator:

```python
trainer = SupervisedTrainer(
    device=device,
    max_epochs=100,
    train_data_loader=train_loader,
    network=net,
    optimizer=opt,
    loss_function=loss,
    inferer=SimpleInferer(),
    post_transform=train_post_transforms,
    key_train_metric={"train_acc": Accuracy()},
    train_handlers=train_handlers,
    amp=True,
)
```

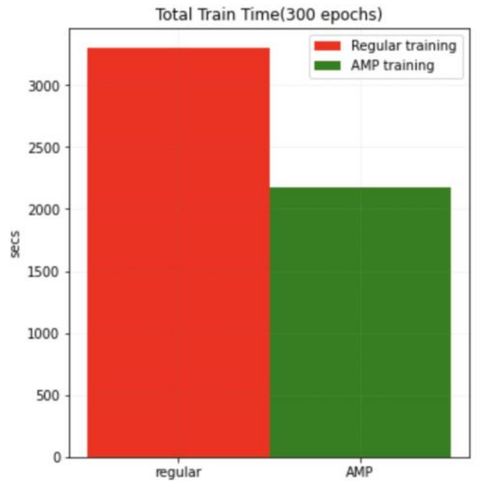
```python
evaluator = SupervisedEvaluator(
    device=device,
    val_data_loader=val_loader,
    network=net,
    inferer=SlidingWindowInferer(),
    post_transform=val_post_transforms,
    key_val_metric={"val_mean_dice": MeanDice()},
    additional_metrics={"val_acc": Accuracy()},
    val_handlers=val_handlers,
    amp=True,
)
```

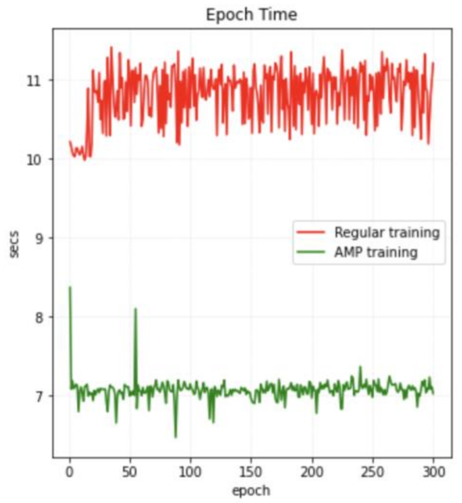
What's the performance with AMP in MONAI?

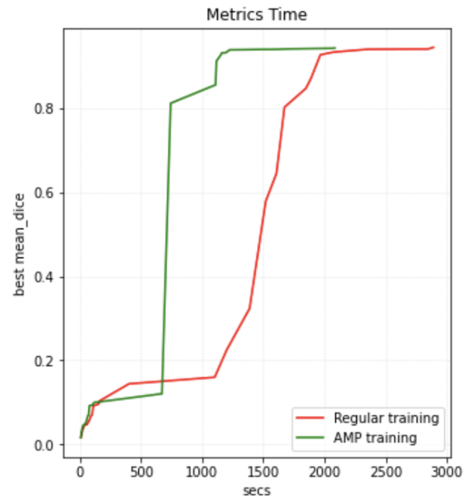
Real-world training experiments with Spleen dataset and UNet

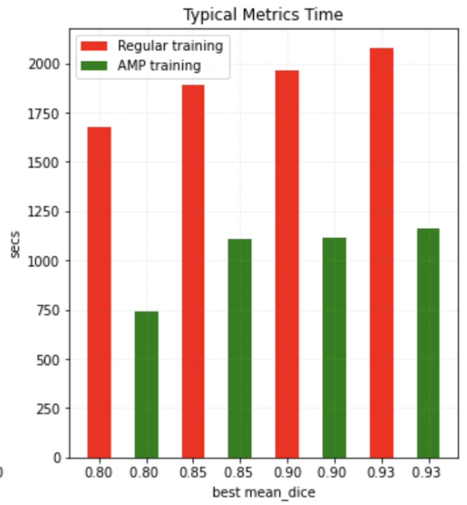
AMP training tutorial in MONAI: https://github.com/Project-MONAI/tutorials/blob/master/automatic_mixed_precision.ipynb (AMP takes fewer epochs).

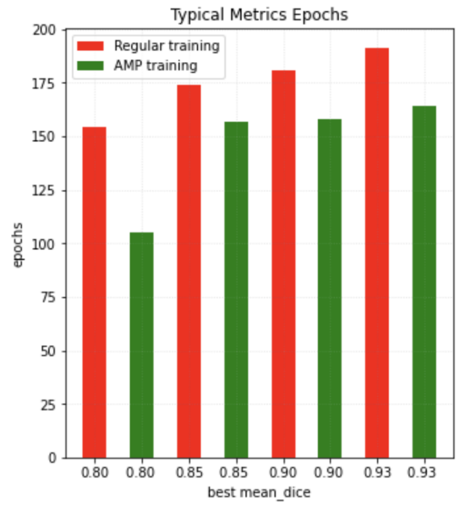
Tesla V100



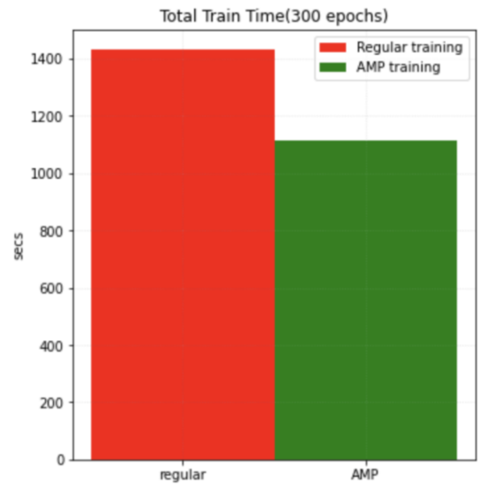


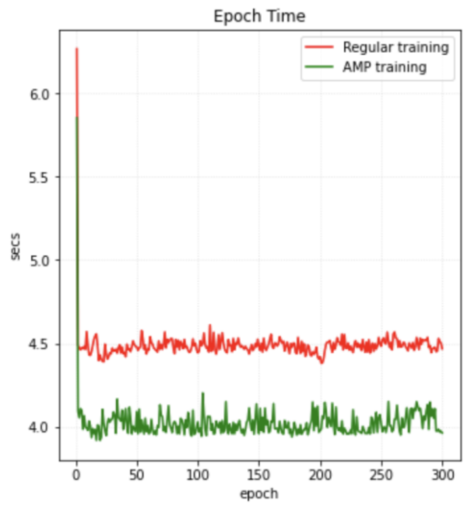


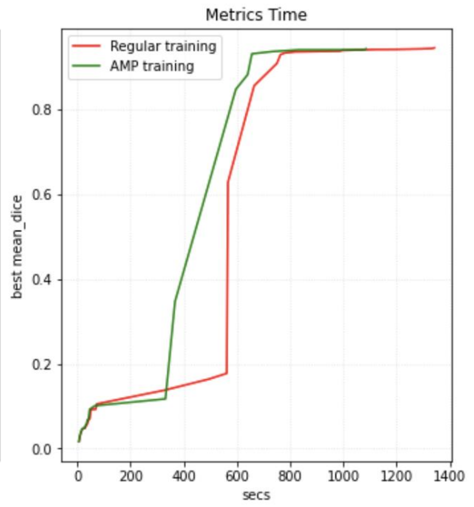


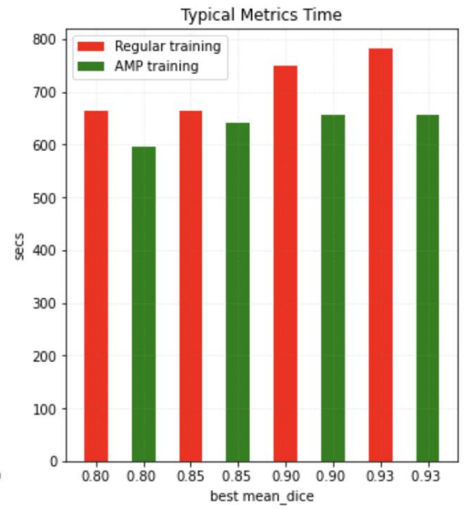


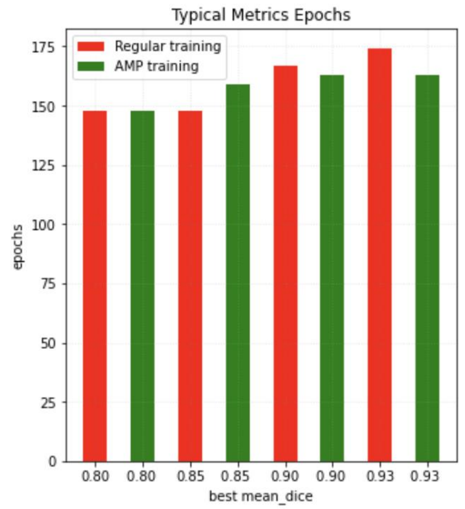
Tesla A100











GPU transforms

Experimentally support PyTorch Tensor since MONAI v0.7

Execute transforms on GPU device

- From MONAI v0.7 we introduced PyTorch Tensor based computation in transforms, many transforms already support `Tensor` data maintaining accuracy.
- Users can convert input data into GPU Tensor by ToTensor or EnsureType transform.
- Then the following transforms can execute on GPU device based on PyTorch Tensor APIs.

Cache IO and transforms data to GPU device

- Even with CacheDataset, we usually need to copy the same data to GPU memory for GPU random transforms or network computation in every epoch.
- As the memory size of new GPU devices are big enough now, an efficient approach is to cache the data to GPU memory directly.
- Then every epoch can start from GPU computation with GPU Tensor immediately.

Example transform chain

```
train_transforms = [
    LoadImaged(...),
    AddChanneld(...),
    Spacingd(...),
    Orientationd(...),
    ScaleIntensityRanged(...),
    CropForegroundd(...),
    FgBgToIndicesd(...),
    EnsureTyped(..., data_type="tensor"),
    ToDeviced(..., device="cuda:0")
    RandCropByPosNegLabeld(...)
)
dataset = CacheDataset(..., transform=train_trans)
```

- Convert to PyTorch Tensor with EnsureTyped transform, and move data to GPU with ToDeviced transform.
- CacheDataset caches the transform results until ToDeviced, so it's in GPU memory. Every epoch will fetch cache data from GPU memory and only execute the random transform RandCropByPosNegLabeld on GPU device directly.

ThreadDataLoader VS DataLoader

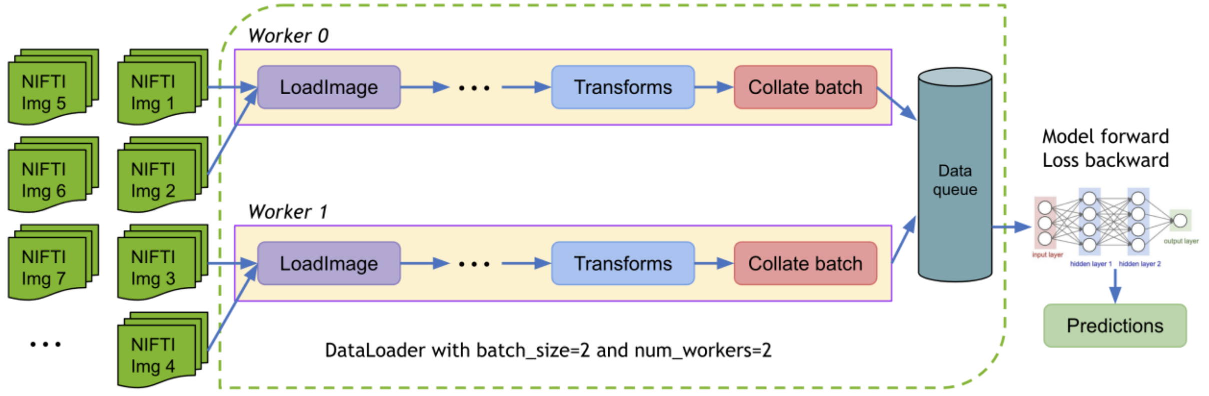
ThreadDataLoader is faster for light-weight transforms

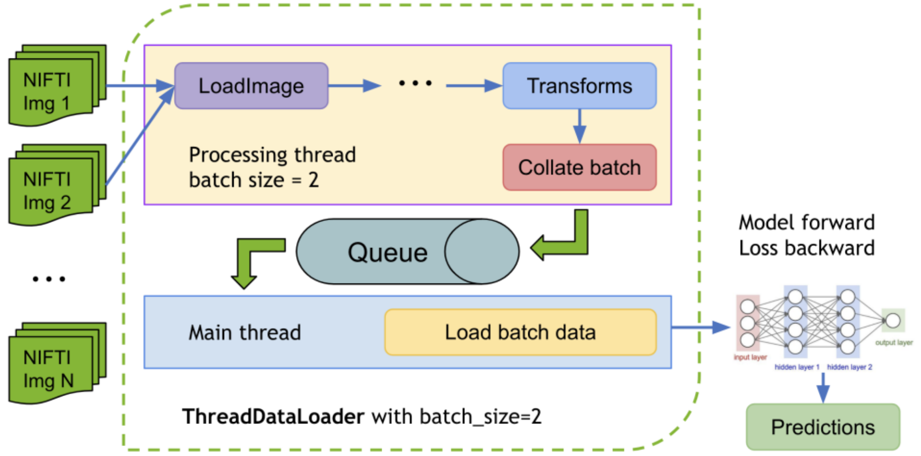
Typical PyTorch DataLoader progress

- The multi-processing execution of PyTorch DataLoader may cause unnecessary IPC time.
- Especially when we already cache all the data in memory to avoid IO operations.
- PyTorch DataLoader usually need 1~2 seconds stop after every epoch.

MONAI ThreadDataLoader progress

- Multi-threads is more efficient than multi-processing for light-weight transforms, especially when no IO operations.
- One thread processes data and put into the queue, another thread fetch data to network.
- CUDA operations of GPU transform can’t work with multi-processing environment.





Fast training for spleen segmentation task

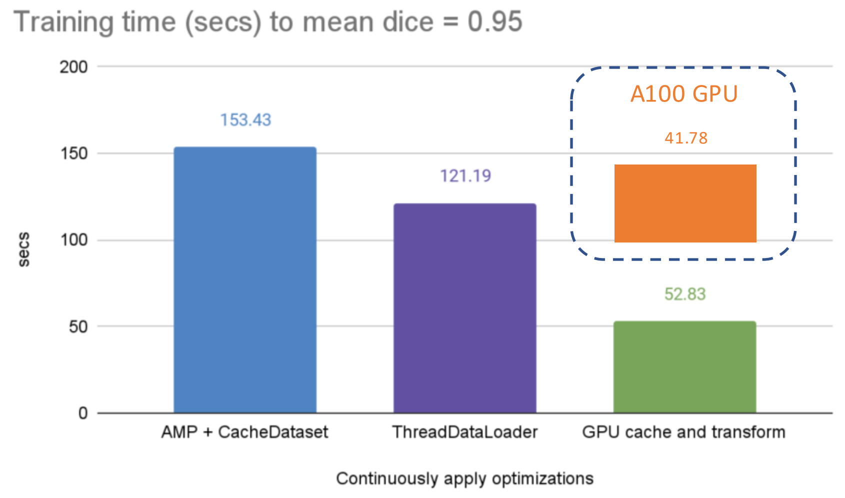
Combine AMP, GPU cache, GPU transforms, ThreadDataLoader

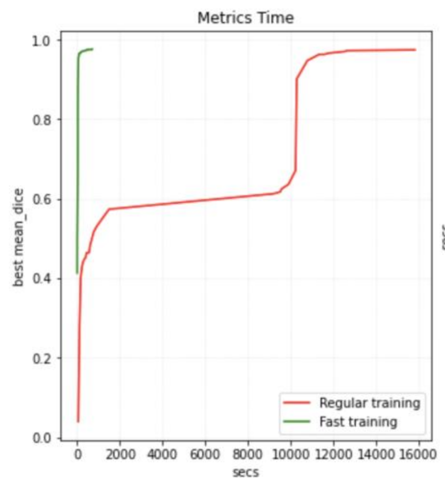
Fast training tutorial in MONAI

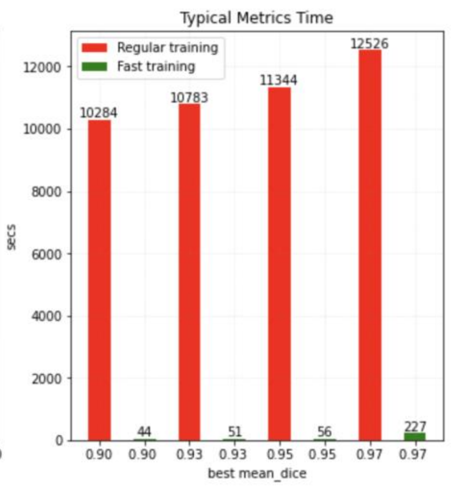
https://github.com/Project-MONAI/tutorials/blob/master/fast_training_tutorial.ipynb

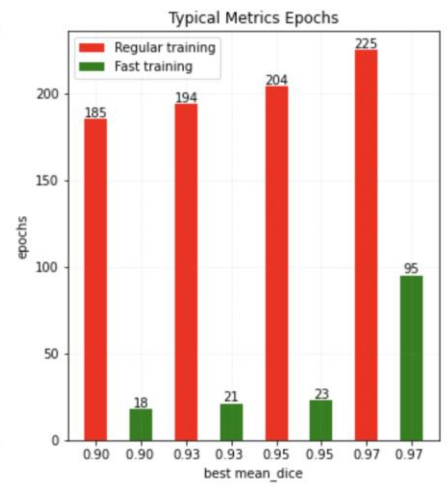
Compare the training speed on V100 GPU:

- AMP + CacheDataset.
- Also use ThreadDataLoader.
- Also cache to GPU memory and execute GPU transforms.









Compare to a regular PyTorch program

All the skills we use here to achieve 200x speed up:

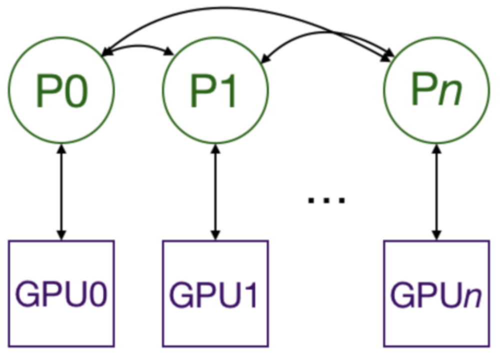
- DiceCE loss function.
- Novograd optimizer.
- AMP.
- CacheDataset.
- ThreadDataLoader
- Cache to GPU and GPU transforms.

Distributed data parallel (DDP)

PyTorch DDP, Horovod, MONAI workflows

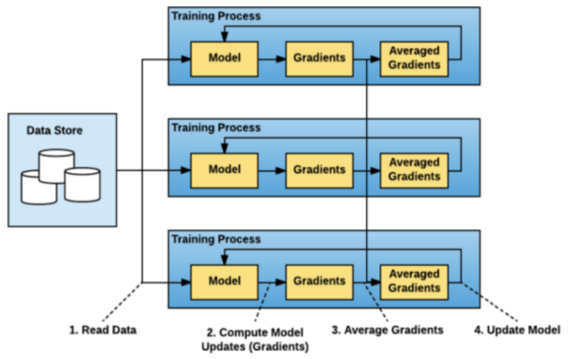
PyTorch distributed data parallel

- Pytorch has two ways to split models and data across multiple GPUs: nn.DataParallel and nn.DistributedDataParallel. nn.DataParallel is easier to use but it uses one process to compute the model weights and then distribute them to each GPU during each batch, GPU utilization is often very low. Furthermore, nn.DataParallel requires that all the GPUs be on the same node and doesn't work with PyTorch AMP training.
- Multiprocessing with DistributedDataParallel duplicates the model across multiple GPUs, each of which is controlled by one process. The GPUs can all be on the same node or spread across multiple nodes. Every process does identical tasks, and each process communicates with all the others.



Horovod distributed training framework

- Horovod is a distributed deep learning training framework for TensorFlow, Keras, PyTorch, and Apache MXNet.
- It uses SSH to remote control nodes, so no need to launch program in every node.
- Horovod open source project is at: https://github.com/horovod/horovod.



Data parallel with MONAI workflows

- It's easy to develop distributed training with MONAI workflows, nothing needs to be changed for trainer or evaluator.
- All the MONAI metrics can automatically detect distributed status and do All-Reduce during computation.

MONAI provides rich distributed training examples for *PyTorch DDP*, *Horovod* and *MONAI workflows*:

https://github.com/Project-MONAI/MONAI/tree/master/examples/distributed_training

Set up data parallel with MONAI workflows

Run on several nodes with multiple GPU devices on every node

Initialize process group

- Use init_process_group to initialize every process, every GPU runs in a separate process with unique rank ID.
- Here we use NVIDIA NCCL as the backend and must set init_method="env://" if using torch.distributed.launch.

```
# initialize the distributed training process, every GPU runs in a process
dist.init_process_group(backend="nccl", init_method="env://")
```

Prepare model and dataloader

- Wrap the model with DistributedDataParallel after moving to expected device.

```
net = DistributedDataParallel(net, device_ids=[args.local_rank])
```

- Wrap Dataset with DistributedSampler, and disable the shuffle in DataLoader. Instead, SupervisedTrainer shuffles data by train_sampler.set_epoch(epoch) before every epoch.

```
train_sampler = DistributedSampler(train_ds)
```

Set components for MONAI workflows

- Some components may only need to work in master progress (dist.get_rank() == 0), like StatsHandler and CheckpointHandler handlers, etc.
- MONAI metrics can automatically reduce metrics for distributed training.

Example command to launch on every node

- python -m torch.distributed.launch --nproc_per_node=NUM_GPUS_PER_NODE --nnodes=NUM_NODES --node_rank=INDEX_CURRENT_NODE --master_addr="192.168.1.1" --master_port=1234 unet_training_workflows.py

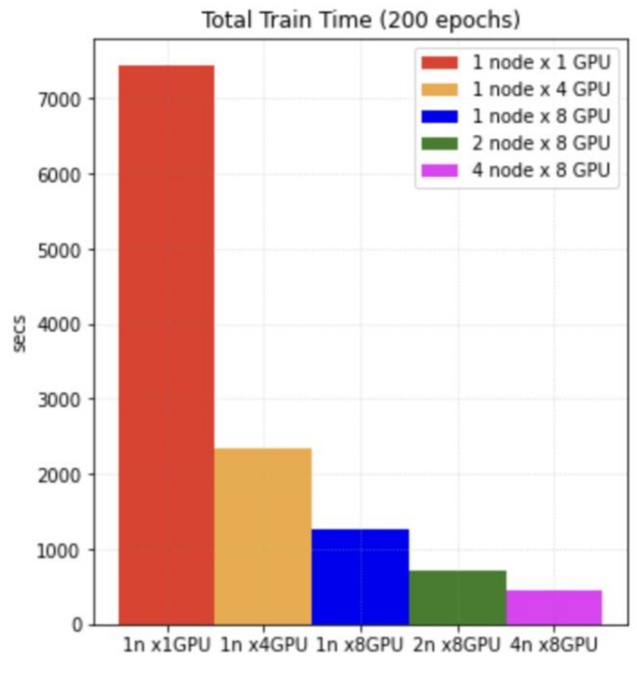
What's the performance with DDP in MONAI?

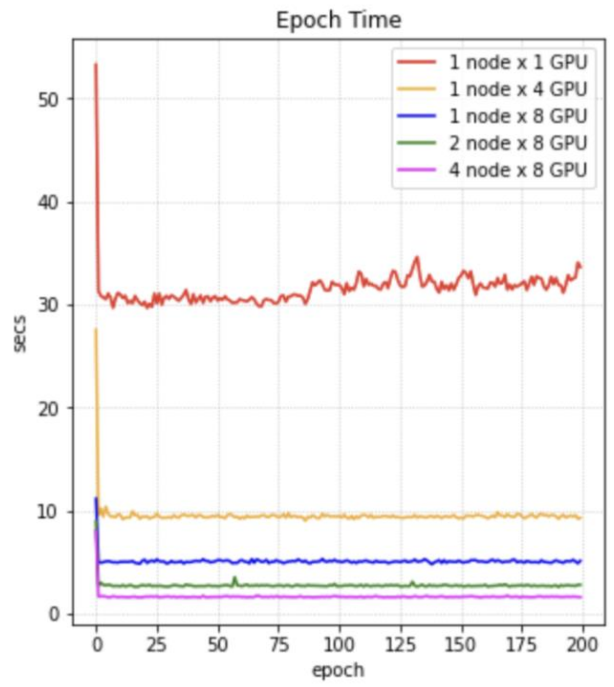
Real-world training experiments with Spleen dataset and UNet

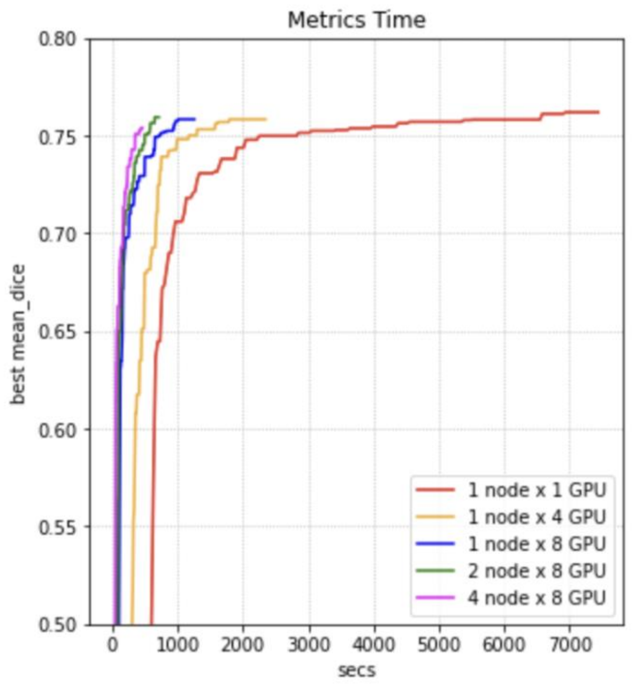
The real-world training example is based on Decathlon challenge Task01 - Brain Tumor segmentation

It contains distributed caching, training, and validation, etc.

We tried to train this example on NVIDIA NGC server, got some performance benchmarks for reference(PyTorch 1.6, CUDA 11, Tesla V100 GPUs).





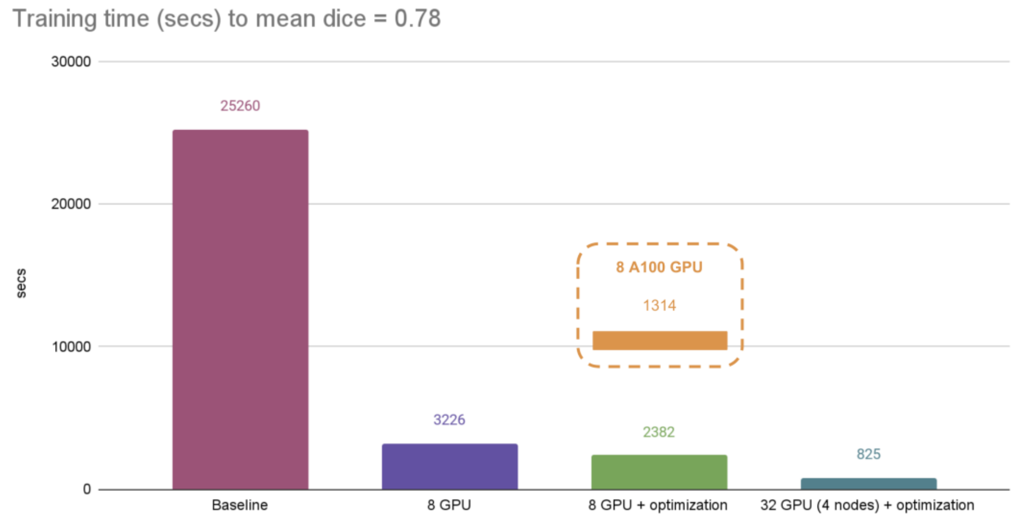


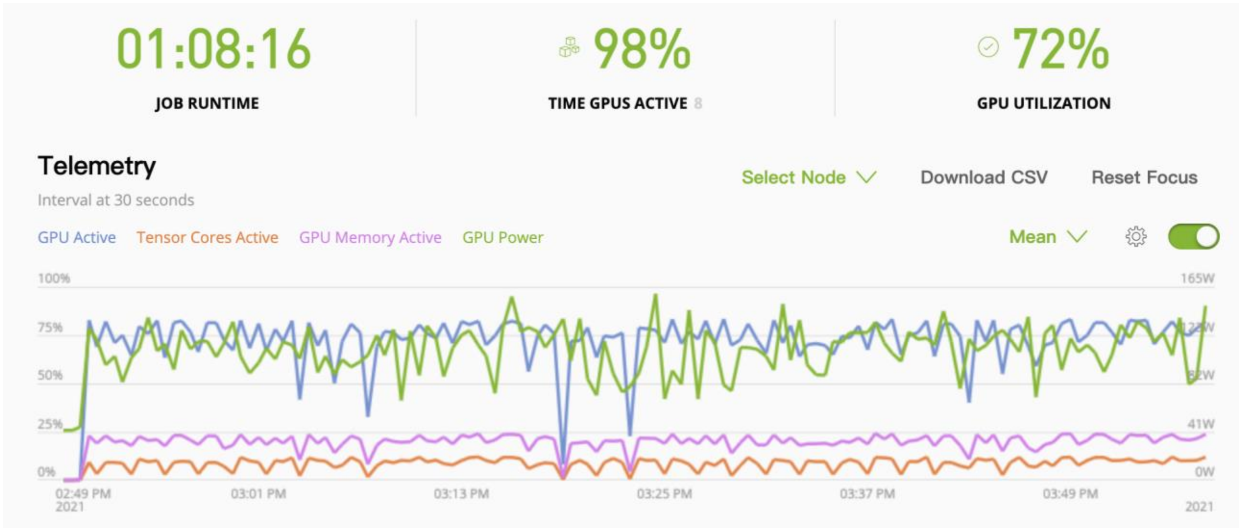
Distributed training for brain tumor segmentation task

Optimize distributed training for best performance

Partition dataset into 8 parts, every GPU progresses 1 part

1. 8 GPUs provide much more memory to cache data in GPU
2. Execute transforms on GPU
3. Use ThreadDataLoader

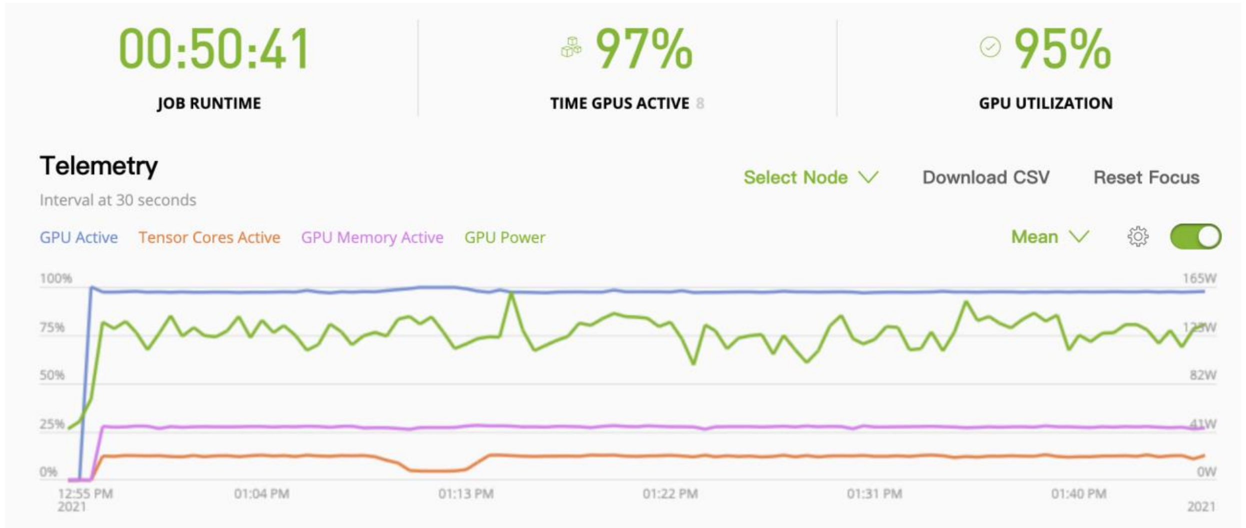




01:08:16 JOB RUNTIME

98% TIME GPUS ACTIVE 8

72% GPU UTILIZATION



00:50:41 JOB RUNTIME

97% TIME GPUS ACTIVE 8

95% GPU UTILIZATION

MONAI

QUESTIONS?