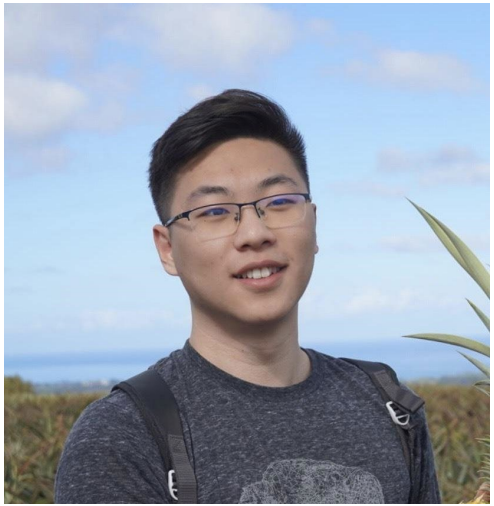
DATA+AI SUMMIT BY databricks

Efficient Stable Diffusion Pre-Training on Billions of Images with Ray

Yunxuan Xiao, Hao Chen (Anyscale Inc.)
[Date]

Speakers



Yunxuan Xiao

Software Engineer @ Anyscale

- Maintainer of Ray Train and Ray Tune.

- Building large-scale distributed training infrastructure.

Hao Chen

Staff Software Engineer @ Anyscale

- Tech lead of Ray Data.
- Early Ray committer.
- Previously led Ant Group's Ray team that built world's largest Ray production workloads.

Overview

- We pre-trained the Stable Diffusion v2 model on ~2 billion images for under $40,000.
- Utilized Ray Data to efficiently process large datasets with heterogeneous resources and mitigate preprocessing bottlenecks.
- Conducted scalable, fault-tolerant training with Ray Train, accelerating training throughput by 3x with infrastructure and algorithm optimizations.

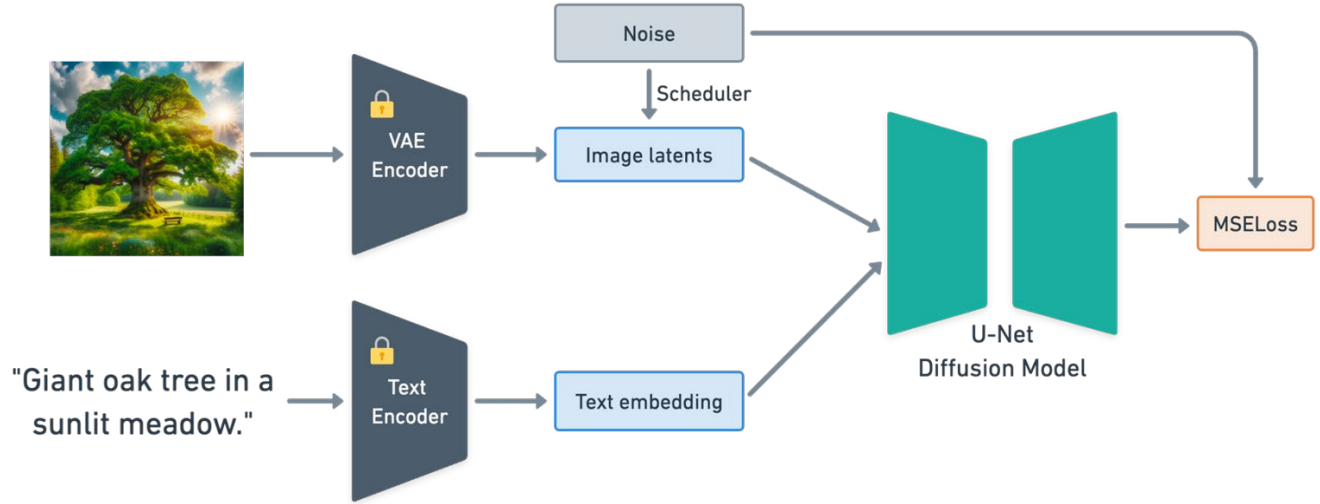
Content

- Stable Diffusion Pre-training and Challenges
- Scalable Data Processing with Ray Data
- Efficient Distributed Training with Ray Train

Content

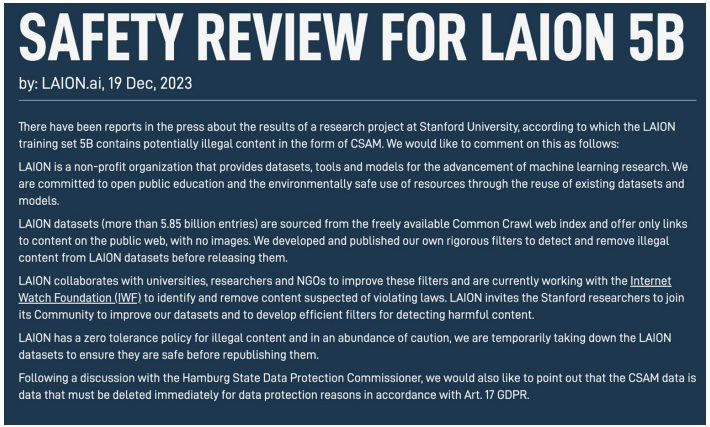
- Stable Diffusion Pre-training and Challenges
- Scalable Data Processing with Ray Data
- Efficient Distributed Training with Ray Train

Stable Diffusion Model Architecture



- A pre-trained VAE and a text encoder(OpenCLIP-ViT/H) encodes the input images and text prompts.
- A trainable U-Net model learns the diffusion process with the image latents and text embeddings.

Why pre-train from scratch?



SAFETY REVIEW FOR LAION 5B

by: LAION.ai, 19 Dec, 2023

There have been reports in the press about the results of a research project at Stanford University, according to which the LAION training set 5B contains potentially illegal content in the form of CSAM. We would like to comment on this as follows:

LAION is a non-profit organization that provides datasets, tools and models for the advancement of machine learning research. We are committed to open public education and the environmentally safe use of resources through the reuse of existing datasets and models.

LAION datasets (more than 5.85 billion entries) are sourced from the freely available Common Crawl web index and offer only links to content on the public web, with no images. We developed and published our own rigorous filters to detect and remove illegal content from LAION datasets before releasing them.

LAION collaborates with universities, researchers and NGOs to improve these filters and are currently working with the Internet Watch Foundation (IWF) to identify and remove content suspected of violating laws. LAION invites the Stanford researchers to join its Community to improve our datasets and to develop efficient filters for detecting harmful content.

LAION has a zero tolerance policy for illegal content and in an abundance of caution, we are temporarily taking down the LAION datasets to ensure they are safe before republishing them.

Following a discussion with the Hamburg State Data Protection Commissioner, we would also like to point out that the CSAM data is data that must be deleted immediately for data protection reasons in accordance with Art. 17 GDPR.

Illegal CSAM content in LAION-5B dataset. [link]

Mickey Mouse in front of a McDonalds sign. [link]

- Avoid generating illegal or copyrighted contents.
- Train Proprietary Model for better performance.
- Reduce reliance on third party libraries and licenses.

Challenges of SD Pretraining

- **Scalable and Performant Data Preprocessing**
  - Large Scale Dataset: 2B Images
  - Complex and heavy Preprocessing logics
  - Includes both CPU and GPU-intensive workloads

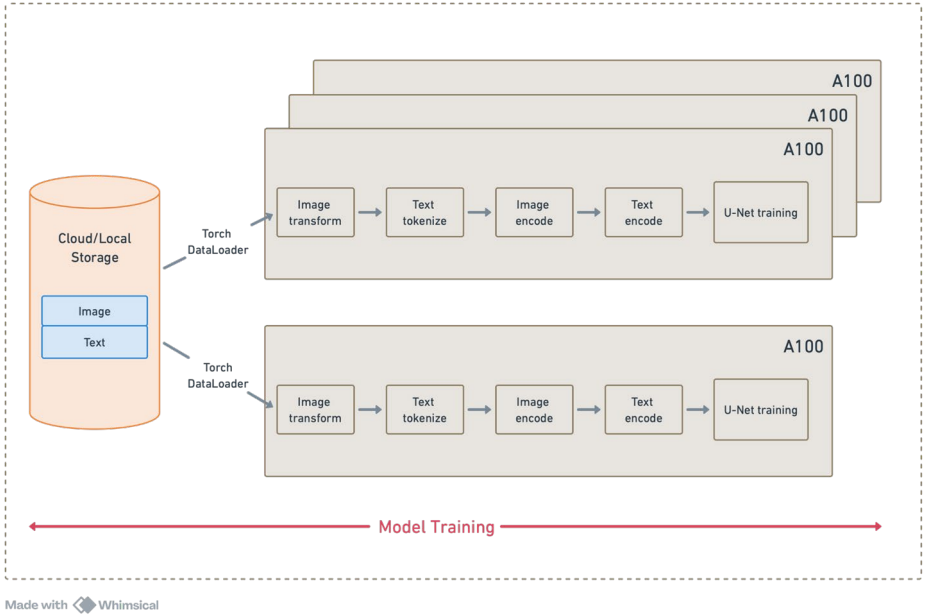
- **Efficient Distributed Large-scale Training**
  - Long-running Job: 13,000+ A100 Hours
  - Fault-tolerant training and maximize GPU utilization

Content

- Challenges in Stable Diffusion Pretraining
- Scalable Data Processing with Ray Data
- Efficient Distributed Training with Ray Train

Traditional Data Pipeline w/ Torch DataLoader

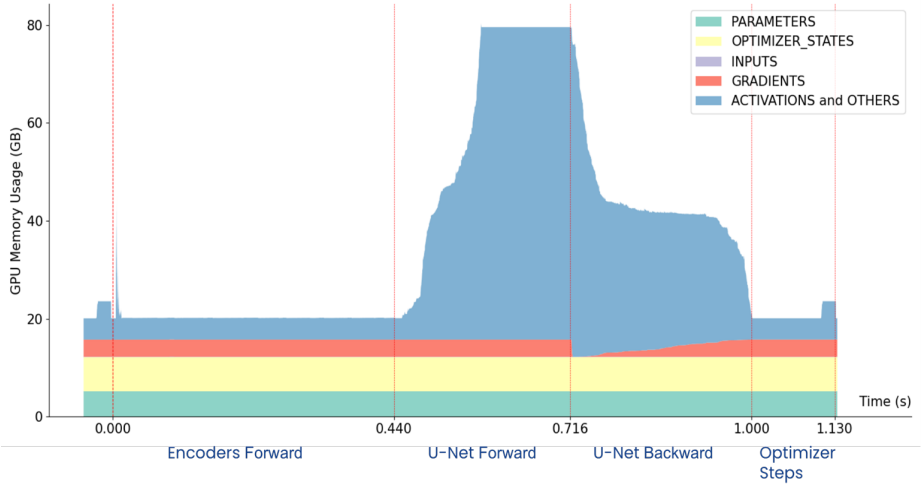
- Ingest data from S3 using Torch DataLoader
- Sequentially execute the following preprocessing functions:
  - Image transformation
  - Text tokenization
  - Image encoding
  - Text encoding
- Feed data into the U-net model for training
- Everything running on the same A100 nodes.



Data preprocessing blocks GPU training

- Image transformation and text tokenization only use CPUs.
- Encoding doesn't need A100s.

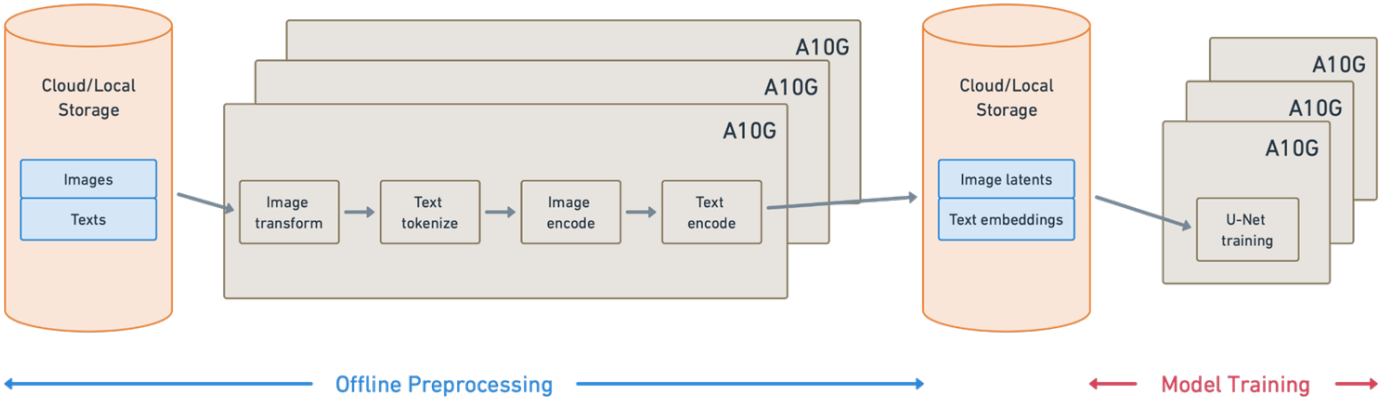
- Low GPU utilization
  - 39% time spent on encoding w/ only 25% GRAM utilization.



GPU Memory footprint over time in an iteration on A100-80G. The red dashed lines represent the start and end times of each step.

Offline Data Preprocessing

- Offline data preprocessing job:
  - Load data from S3
  - Transform images and tokenize captions
  - Infer images and texts with encoders on A10G GPUs
  - Save results to back to S3
- Training job:
  - Ingest preprocessed data
  - Feed directly into A100



Ray Data

- Ray Data: a scalable data processing library for ML workloads built on top of Ray. Particularly optimized for 2 scenarios:
- Offline batch inference
  - Image/video/audio processing + inference, embedding generation, LLM batch inference, etc.
- Training ingestion
  - Scalable training data loading and preprocessing.

Implementing Offline Preprocessing w/ Ray Data

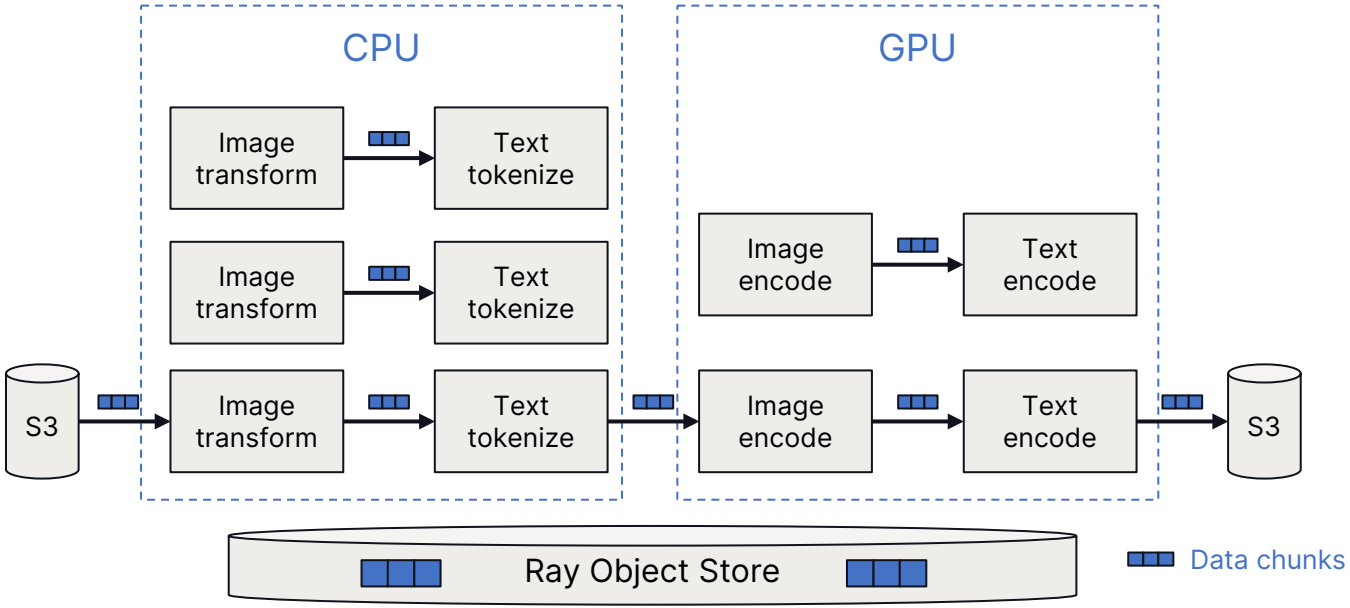
```python
# 1. Load data from S3.
ds = ray.data.read_parquet("s3://input_path")


# Define a function that transforms images and tokenizes
# captions for each row.
def transform_and_tokenize(row):
    row["image"] = crop_and_normalize(row["image"])
    row["caption_ids"] = tokenize(row["caption_ids"])
    return row


# 2. Apply transform
ds = ds.map(transform_and_tokenzie)
```

```python
# Define a callable class that conducts inference with
# image and text encoders.
class SDLatentEncoder:

    def __init__(self):
        # Load and cache image and text encoders.
        self.vae = AutoencoderKL.from_pretrained("...").to("cuda")
        self.text_encoder = CLIPTextModel.from_pretrained("...").to("cuda")

    def __call__(self, batch):
        encoded_batch = {}
        with torch.no_grad():
            # Encode images and texts for each batch.
            encoded_batch["image_latents}"] = self.vae.encode(batch["image"])[
                "latent_dist"
            ]
            encoded_batch["caption_embeddings"] = self.text_encoder(
                batch["caption_ids"]
            )
        return encoded_batch


# 3. Apply encoder inference.
ds = ds.map_batches(SDLatentEncoder, batch_size=100, num_gpus=1)
# 4. Write results to S3.
ds.write_parquet("s3://output_path")
```

Why Ray Data

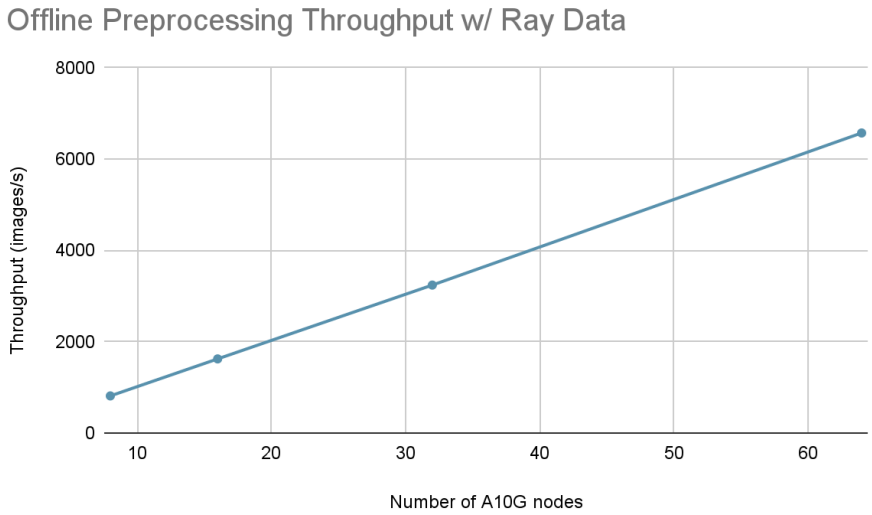
- Streaming execution, scalable to petabyte-scale data
- Support heterogeneous resource requirements
- Automatic failure recovery
- Support a large variety of data sources and formats
  - S3, GCS
  - Parquet, images, JSON, text, CSV, etc.
- Python native & seamless integration with other ML libraries

Why Ray Data (cont'd)

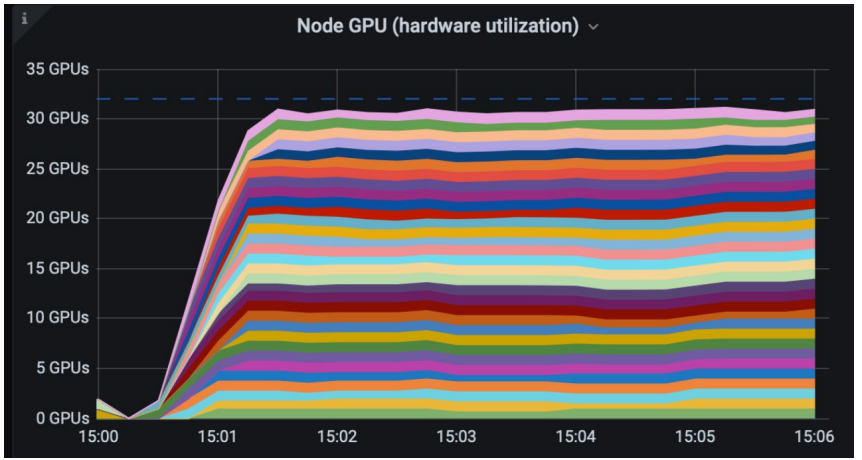


- Streaming data loading, processing, writing.
- Intermediate data buffered in Ray Object Store.
- Schedule tasks with heterogeneous resource requirements.
- Operators adjust parallelism dynamically and independently.

Benchmark Results

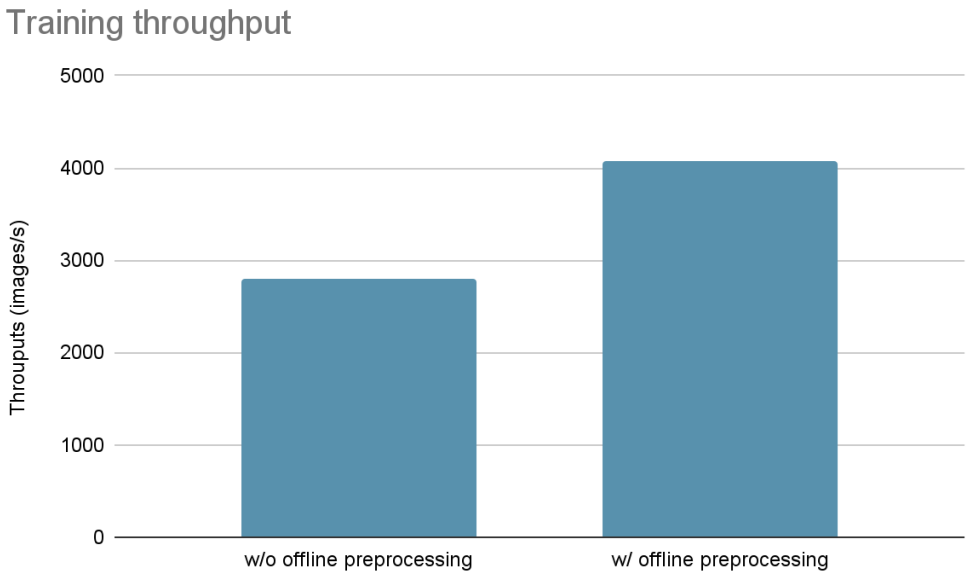


Throughput scales linearly while adding more A10Gs.



Constant 95+% GPU utilization

Benchmark Results (cont'd)



+45% training throughput with preprocessed data

Limitations of Offline Preprocessing

- Development velocity
  - When experimenting with different preprocessing logics, need to wait for days to preprocess the entire dataset.
- Flexibility
  - Doesn’t support dynamic preprocessing logics
    - e.g. random crop, multi-aspect

Online preprocessing without blocking A100 GPUs?

Ray Data For Online Preprocessing

```python
# 1. Reuse the same code as the offline job to create the datasets.
def create_dataset(input_uri):
    ds = ray.data.read_parquet(input_uri)
    ds = ds.map(transform_and_tokenzie)
    ds = ds.map_batches(
        SDLatentEncoder,
        batch_size=100,
        num_gpus=1,
        accelerator_type=NVIDIA_TESLA_A10G,
    )
    return ds


# 2. Create 2 datasets and pass them to the Trainer.
datasets = {
    "train": create_dataset("s3://train_data"),
    "validation": create_dataset("s3://validation_data"),
}
```

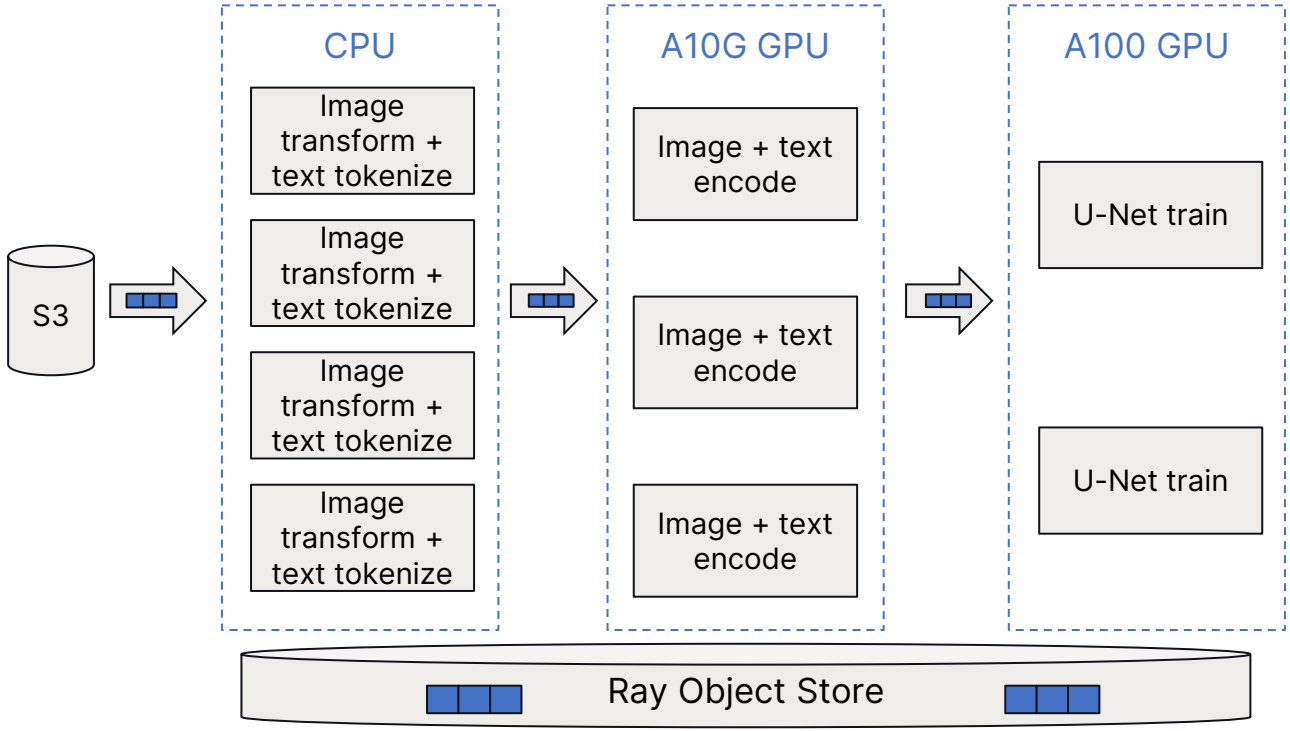
- Reuse the same preprocessing pipeline code

```python
def train_func():
    # 3. In the training function, get the data shard for the
    # current training worker. And feed data batches to the model.
    train_ds = ray.train.get_dataset_shard("train")
    train_dataloader = train_ds.iter_torch_batches(batch_size=50)

    for batch in train_dataloader:
        ...

# Internally Ray Data dynamically splits the datasets across
# distributed training workers.
trainer = TorchTrainer(
    train_func,
    datasets=datasets,
    scaling_config=ScalingConfig(
        num_workers=8,
        use_gpu=True,
        resources_per_worker={
            "num_gpus": 1,
            "accelerator_type": NVIDIA_A100,
        },
    ),
)
trainer.fit()
```

Heterogeneous GPU requirements

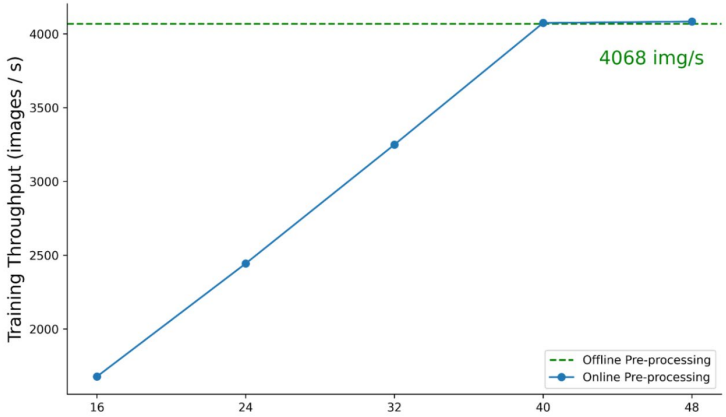
- Dynamically split data across distributed workers; no need for manual sharding

Architecture w/ Ray Data



- Schedule different stages on different hardwares separately.
- Each stage scales up individually, making GPUs always saturated.
- Streaming across the entire pipeline.

Benchmark Results



Throughput scales linearly while adding more A10Gs until reaching the same number as offline processing.

	Online Preprocessing w/ Torch DataLoader	Online Preprocessing w/ Ray Data
Cluster	4 x p4de.24xlarge	4 x p4de.24xlarge 40 x g5.2xlarge
Training throughput (images/s)	2811	4075 (+45%)
Total training time on Anyscale	111.3h	76.8h (-31.0%)
Cost per epoch	$18,192	$16,275 (-10.5%)

Ray Data vs Torch DataLoader for online preprocessing

Ray Data for SD Data Preprocessing

- Offline Data Preprocessing
  - Streaming and scalable.
  - Fault tolerant.
  - Maximizes GPU utilization w/ heterogeneous resource scheduling.

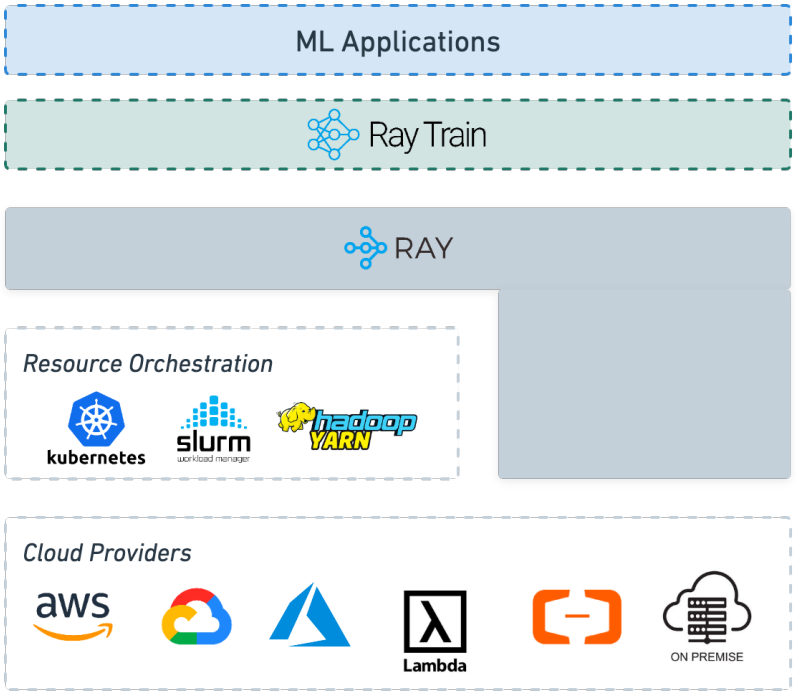
- Online Data Preprocessing
  - Unified data pipeline code.
  - Splits data for distributed training dynamically.
  - Improves perf by scaling out data preprocessing on CPUs and lower-end GPUs.
  - Makes A100s dedicated for U-Net training, improving A100 availability.

Content

- Challenges in Stable Diffusion Pretraining
- Scalable Data Processing with Ray Data
- Distributed Training with Ray Train

Distributed Training with Ray Train

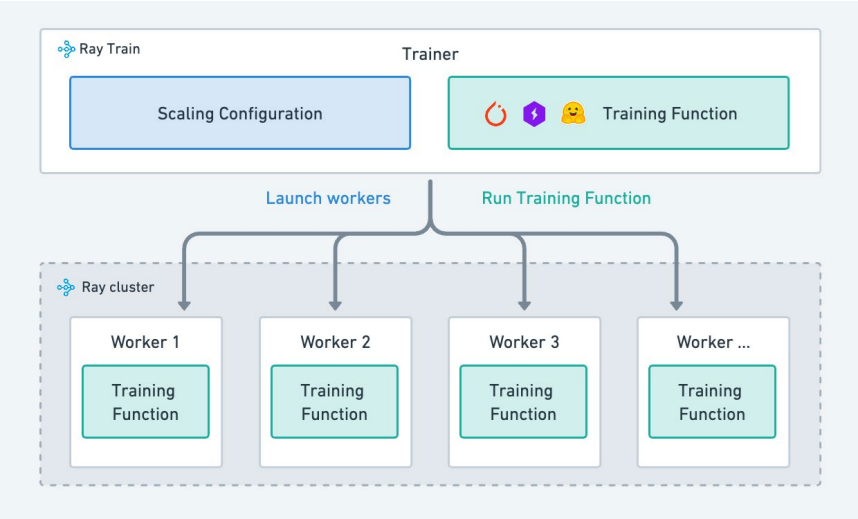
Ray Train Ecosystem



Ecosystem Integrations

- ML Frameworks
  - PyTorch, HuggingFace, Lightning, TensorFlow, …
- Cloud Providers:
  - AWS, GCP, Azure, Aliyun, vSphere, …
- Cluster Managers:
  - K8s, Yarn, Slurm, LSF, …

Distributed Training with Ray Train




```python
from ray.train.torch import TorchTrainer
from ray.train import ScalingConfig

def train_func(config):
    # Your PyTorch/Lightning/Hugging Face training code here.

scaling_config = ScalingConfig(num_workers=4, use_gpu=True)
trainer = TorchTrainer(train_func, scaling_config=scaling_config)
result = trainer.fit()
```

Distributed Training with Ray Train

PyTorch Integration

Setup distributed env

Setup DDP model

Setup distributed sampler

Move batches to GPU

```python
def train_func(config):
  dist.init_process_group("nccl")
  rank = os.environ["LOCAL_RANK"]
  device = torch.device(f"cuda:{rank}")

  model = MyTorchModel(...)
  model = model.to(device)
  model = DDP(model, device_ids=[device])


  sampler = DistributedSampler(
      dataset,
      rank=os.environ["RANK"],
      num_replicas=os.environ["WORLD_SIZE"],
      shuffle=True,
  )
  dataloader = DataLoader(
    ...,
    sampler=sampler
  )

  # Training
  for epoch in range(num_epochs):
    for inputs, labels in dataloader:
      # train batch
      inputs = inputs.to(device)
      labels = labels.to(device)
      ...
```

```python
from ray.train.torch import prepare_model, prepare_data_loader

def train_func(config):
  model = MyTorchModel(...)
  model = prepare_model(model)

  dataloader = DataLoader(...)
  dataloader = preapre_data_loader(dataloader)

  # Training
  for epoch in range(num_epochs):
    for inputs, labels in dataloader:
      ...
```

Ray Train

```python
from ray.train.torch import TorchTrainer
from ray.train import ScalingConfig

trainer = TorchTrainer(
  train_func,
  scaling_config=ScalingConfig(num_workers=16, use_gpu=True)
)
trainer.fit()
```

Distributed Training with Ray Train

PyTorch Lightning Integration

```python
import pytorch_lightning as pl
from pytorch_lightning.strategies import DDPStrategy
from pytorch_lightning.plugins.environments import LightningEnvironment

def train_func(config):
  model = CustomLightningModule(...)
  dataloader = CustomDataLoader(...)

  trainer = pl.Trainer(
    ...,
    device="auto",
    strategy=DDPStrategy(),
    plugins=[LightningEnvironment()]
  )

  trainer.fit(model, train_dataloaders=dataloader)
```

```python
import pytorch_lightning as pl
from ray.train.lightning import RayDDPStrategy, RayLightningEnvironment, prepare_trainer

def train_func(config):
  model = CustomLightningModule(...)
  dataloader = CustomDataLoader(...)

  trainer = pl.Trainer(
    ...,
    device="auto",
    strategy=RayDDPStrategy(),
    plugins=[RayLightningEnvironment()]
  )
  trainer = prepare_trainer(trainer)
  trainer.fit(model, train_dataloaders=dataloader)
```

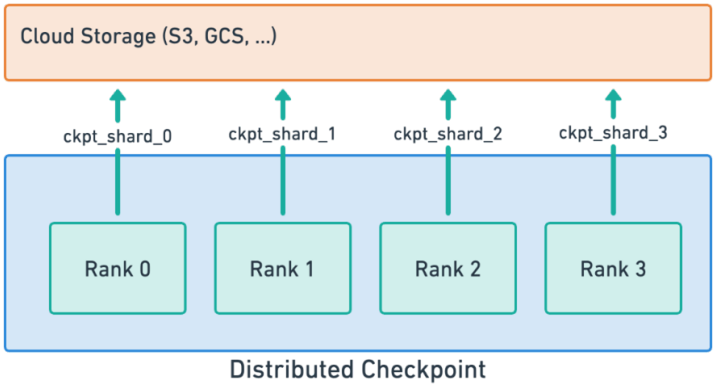
Ray Train

```python
from ray.train.torch import TorchTrainer
from ray.train import ScalingConfig

trainer = TorchTrainer(
  train_func,
  scaling_config=ScalingConfig(num_workers=16, use_gpu=True)
)
trainer.fit()
```

Key features of Ray Train

- Efficient Distributed Checkpoint
- Fault-Tolerant Training
- Easily integrate with training acceleration techniques

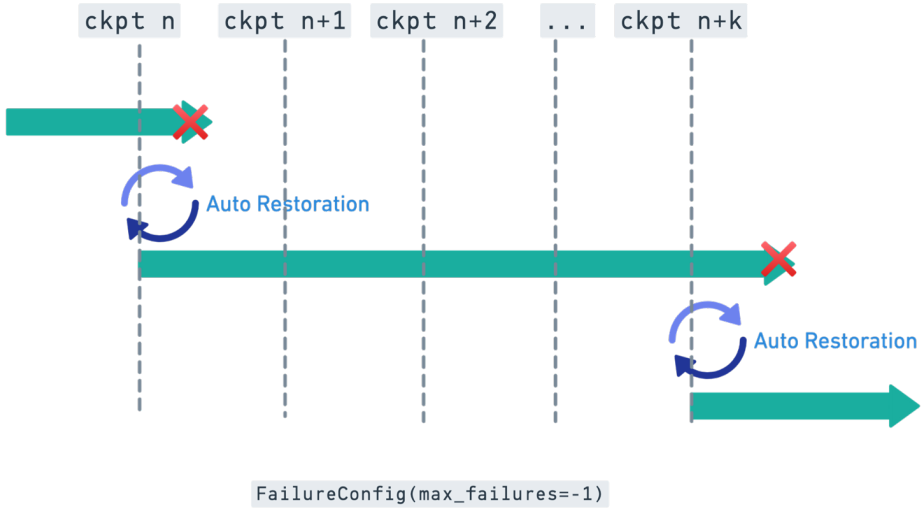
Efficient Distributed Checkpoint




```python
def train_func():
    context = ray.train.get_context()
    world_rank = context.get_world_rank()

    # Save ckpt to */ckpt_dir/shard_{world_rank}
    ckpt = Checkpoint.from_directoy("*/ckpt_dir")
    ray.train.report(..., ckpt=ckpt)

trainer = TorchTrainer(
  train_func,
  ...,
  run_config=RunConfig(
     storage_path="s3://ray-train-experiments/"
  ),
)
```

- Each worker independently syncs its checkpoint(shard) to cloud storage (e.g. S3, GCS).
- Support flexible checkpointing logics (World Rank 0, Local Rank 0, All ranks)

Fault-tolerant Training

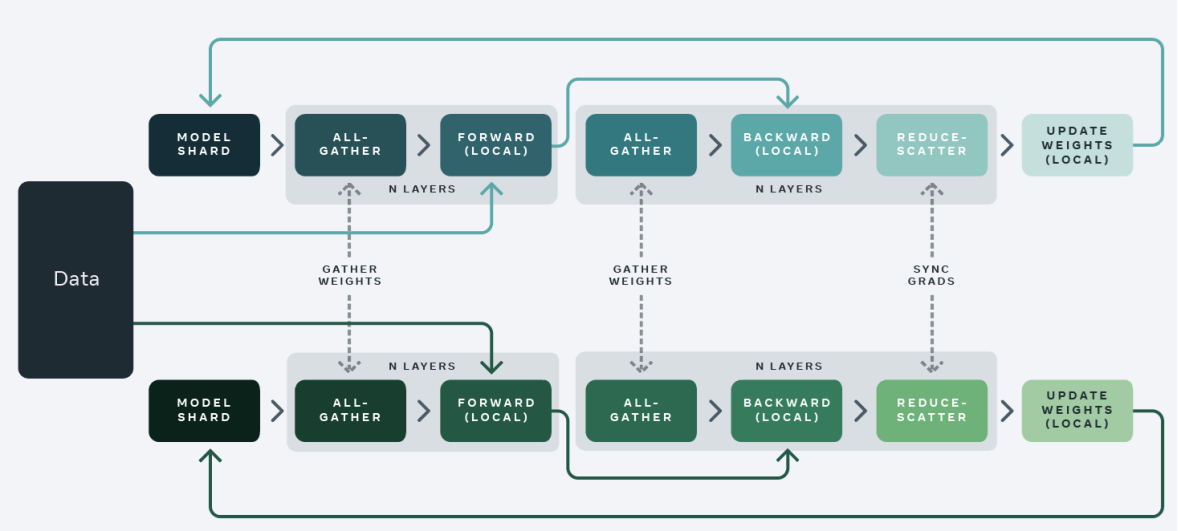



```python
def train_func():
    model = ...

    ckpt = ray.train.get_checkpoint()
    if ckpt:
      with ckpt.as_directory() as ckpt_dir:
        model.load_checkpoint(ckpt_dir)
    ...

trainer = TorchTrainer(
  train_func,
  run_config=RunConfig(
     ...,
     failure_config=FailureConfig(max_failure=-1)
  ),
)
```

- Ray Cluster automatic scales up on node failure
- Restore training from the latest checkpoint

Training Acceleration

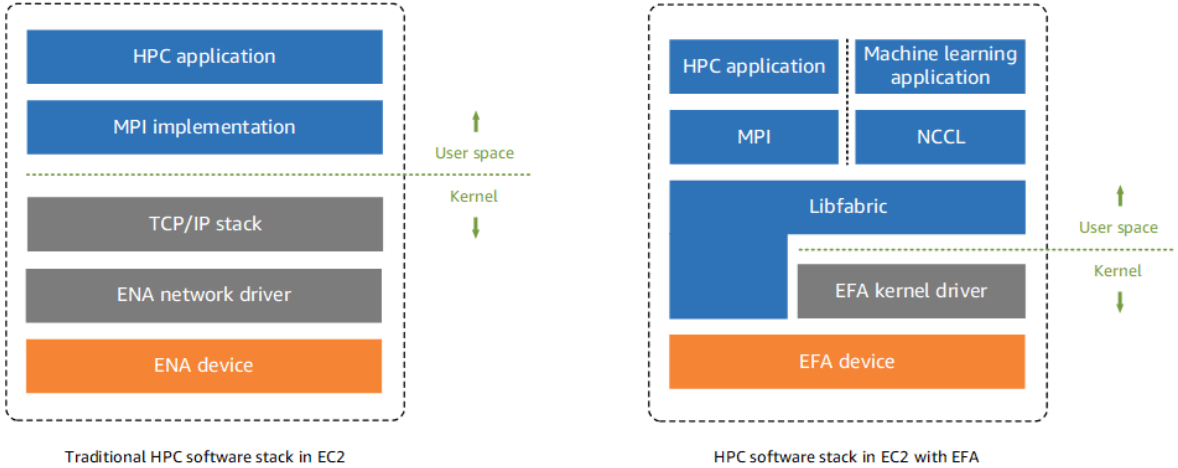
FSDP



- FSDP [8] is designed to reduce communication overhead by sharding model state. With SHARD_GRAD_OP mode, which partition the gradient and optimizer states among all workers.
- Compared to DDP, it reduces communication overhead of full gradient synchronization and also reduces peak GRAM usage, allowing for larger batch sizes and higher throughput.

Training Acceleration

EFA



- Elastic Fabric Adapter (EFA) provides lower and more consistent latency and higher throughput than the TCP transport traditionally used in cloud-based HPC systems.
- Significantly reduces the communication overhead and speeds up distributed training.

Training Acceleration

Ablation Study - Training Speedup

	Baseline (DDP)	+ EFA	+ FSDP	+ torch.compile
Resolution @ 256x256	1075	1269 (1.18x)	1925 (1.79x)	2910 (2.71x)
Resolution @ 512x512	264	474 (1.86x)	667 (2.52x)	805 (3.05x)

Table 4.a: Training throughput (images/s) and speedup on 16 x A100-40G.

	Baseline (DDP)	+ EFA	+ FSDP	+ torch.compile
Resolution @ 256x256	1573	4068 (2.59x)	5014 (3.18x)	5908 (3.75x)
Resolution @ 512x512	389	1029 (2.64x)	1168 (3.00x)	1349 (3.46x)

Table 4.b: Training throughput (images/s) and speedup on 32 x A100-80G.

Results

Instance Type	p4de.24xlarge
Cloud Provider	AWS (us-west-2)
GPU Type	A100-80G
Global Batch Size	4096
Training Procedure	Phase 1: 1,126,400,000 samples at resolution 256x256 Phase 2: 1,740,800,000 samples at resolution 512x512
Total A100 Hours	13,165
Total Training Cost	$39,511 (1-yr reservation instances) $67,405 (on-demand instances)

- Ray Data helps resolved preprocessing bottlenecks, boosting training throughput by 30%.
- System and training optimizations further reduce training costs by 3x over baseline.
- We pre-trained the Stable Diffusion v2 model from scratch for under $40,000.

Takeaways

- Decouple encoder forward pass from Unet training to resolve preprocessing bottleneck.
- Leverage heterogeneous resources to reduce overall training costs.
- Implement fault-tolerant training and efficient checkpointing to minimize training progress loss.
- Apply infrastructure and algorithmic optimizations to accelerate training speed.

Thank you