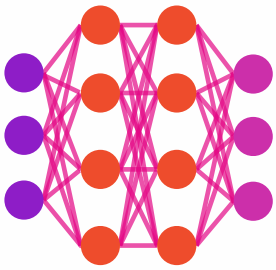
MODERN DATA PLATFORM EXCELLENCE SERIES

ALLUXIO

PyTorch Model Training Performance Tuning



PyTorch

A Comprehensive Guide

INTRODUCTION

IIn the fast-paced, tech-driven world, Artificial Intelligence (AI) continues to grow in popularity and disrupt a wide range of domains. PyTorch, an open-source machine learning framework, has become the de facto choice for many organizations to develop and deploy deep learning models.

Model training is the most compute-intensive phase of the machine learning pipeline. It requires continuous performance optimization. It can be challenging and time-consuming to fine-tune the training performance because many factors, such as I/O, data operations, GPU, and CPU processing, can make training slow.

This eBook is your go-to guide for pinpointing the root cause of low performance and efficiency issues in PyTorch training. It is tailored for AI/ML platform engineers, data platform engineers, backend software engineers, MLOps engineers, site reliability engineers, architects, machine learning engineers, and anyone who wants to implement advanced performance tuning techniques with PyTorch. Basic Python programming skills and knowledge of Linux command-line usage are assumed.

The techniques introduced in this guide are for tuning the infrastructure of PyTorch and the resources it uses. The tuning tips are generic and apply to any model algorithm, including CNNs, RNNs, GANs, transformers (GPT, BERT), and many more; and, in any domains, computer vision (CV), natural language processing (NLP), and so on. Note that this guide does not serve the purpose of tuning model parameters, such as picking the right set of hyperparameters.

You will learn the following:

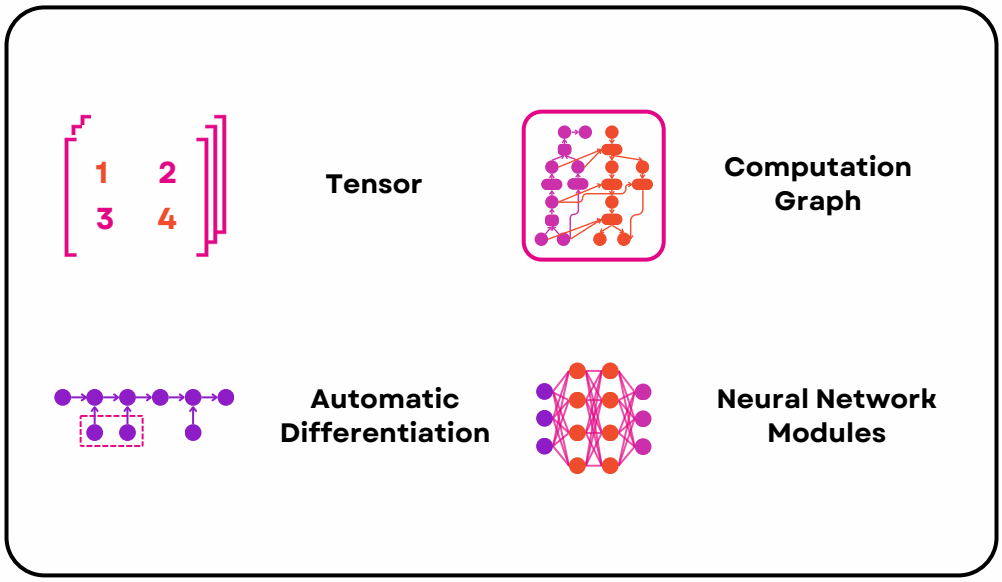
- The basics of PyTorch, including how tensors, computation graphs, automatic differentiation, and neural network modules work
- What can impact the performance of model training in the ML pipeline
- The process of optimizing PyTorch model training step-by-step
- Best tuning tips in data loading, data operations, GPU processing, and CPU processing, with lines of code. With these tips, the average training epoch time can be reduced by 5-10x
- Case studies using Alluxio as a data access layer to power model training in real-world production

Table of Contents

CHAPTER

01 Understand the Basics of PyTorch in Model Training

Before you start tuning PyTorch, you need to understand how it works at a fundamental level. This chapter helps you understand how tensors, computation graphs, automatic differentiation, and neural network modules work. Once familiar with these fundamental concepts, you can start optimizing your PyTorch code to improve its performance.



1.1 Tensors

In PyTorch, tensors are fundamental data structures that resemble arrays or matrices, similar to NumPy's ndarrays. However, what sets tensors apart is their ability to utilize GPU resources for accelerated computation seamlessly. This makes PyTorch an ideal choice for training deep learning models that require immense computational power.

When working with tensors in PyTorch, several essential attributes and operations play a crucial role in training models. Firstly, tensors possess a shape attribute, providing valuable information about their dimensions. Understanding the size and structure of tensors is fundamental for designing and debugging models effectively.

Furthermore, PyTorch offers an extensive range of tensor operations, including element-wise operations (e.g., addition, multiplication), matrix operations (e.g., matrix multiplication, matrix inversion), and reduction operations (e.g., sum, mean). These operations empower researchers and practitioners to perform computations and manipulate tensors during the training process efficiently.

1.2 Computation Graph

PyTorch's computation graph is a fundamental component that captures the sequence of operations performed on tensors in a deep learning model. It automatically tracks and records these operations during the forward pass, creating a dynamic computational graph. This graph serves as the backbone for efficient automatic differentiation, allowing gradients to be computed efficiently during the backward pass. By leveraging the computation graph, PyTorch enables efficient backpropagation, making it easier to optimize model parameters and train deep learning models effectively.

One of the key benefits of PyTorch's computation graph is its flexibility in handling dynamic model architectures. The graph enables researchers and practitioners to create models with varying computational flows, including recurrent connections, conditional branching, and even adaptive network structures. This flexibility is especially valuable for advanced research and experimentation, where models need to be modified on-the-fly. By dynamically constructing the computation graph, PyTorch empowers users to explore complex network designs and push the boundaries of deep learning.

1.3 Automatic Differentiation

PyTorch's automatic differentiation capability is a key concept that allows for efficient gradient computation during the training of neural networks. By setting the requires_grad attribute on tensors, PyTorch automatically tracks operations on tensors and computes gradients using the backpropagation algorithm. This enables efficient optimization of model parameters using gradient-based optimization algorithms like stochastic gradient descent.

1.4 Neural Network Modules

PyTorch provides a module-based approach for building neural networks. Neural network modules are classes that inherit from the nn.Module base class and encapsulate layers, activation functions, and other components of a neural network. This modular design simplifies defining, organizing, and training complex neural network architectures in PyTorch. Neural network modules offer flexibility and reusability, making constructing and experimenting with different network structures easier.

CHAPTER

02 PyTorch Training Performance Tuning Tips

Now that you understand how PyTorch works, you can start to identify potential areas for optimization. In this chapter, you will learn performance tuning recommendations, including data loading, data processing, GPU processing, and CPU processing.

2.1 Identify Bottlenecks Using Monitoring Tools

We recommend identifying the bottleneck in a system before optimizing it. This is because the bottleneck can vary depending on a number of factors, such as the size of the dataset, the complexity of the model, and the hardware that is being used. By identifying the bottleneck, you can focus your optimization efforts on the areas with the biggest impact on performance.

For example, if the dataset is large, the bottleneck may be the data loading step. The bottleneck may be the model training step if the model is very complex. In PyTorch, the bottleneck can also vary depending on the specific code used. For example, if the code is not using GPU acceleration, the bottleneck may be the CPU. However, if the code uses GPU acceleration, the bottleneck may be the GPU memory or the bandwidth between the CPU and the GPU.

By identifying the bottleneck in a system, you can focus your optimization efforts on the areas with the biggest impact on performance. This can lead to significant performance improvements, saving time and money.

In this section, we will explore how to identify bottlenecks in PyTorch training using monitoring tools and then we will discuss the tuning tips for different types of bottlenecks in the next sections.

2.1.1 Traditional Command Line Tools

Command-line tools are useful for monitoring PyTorch training and identifying the bottleneck. They are easy to use, can be accessed from any terminal, and can be used to monitor a variety of metrics, including CPU usage, GPU usage, memory usage, and I/O traffic. However, command-line tools can only provide a limited view of the PyTorch training process and cannot provide information about the specific operations causing the bottleneck. They can also be difficult to use for users who need to become more familiar with them.

Here is a list of the most commonly used command-line tools for monitoring resource usage:

- nvidia-smi: This tool provides information about GPU utilization, memory usage, and other metrics related to the NVIDIA GPU.
- htop: It is a command-line tool that hierarchically displays system processes and provides insights into CPU and memory usage.
- iotop: With this tool, you can monitor I/O usage by displaying I/O statistics of processes running on your system.
- gpustat: It is a Python-based user-friendly command-line tool for monitoring NVIDIA GPU status.
- nvtop: Similar to nvidia-smi, nvtop displays real-time GPU usage and other metrics in a user-friendly interface.
- py-spy: It is a sampling profiler for Python that helps identify performance bottlenecks in your code.
- strace: This tool allows you to trace system calls made by a program, providing insights into its behavior and resource usage.

2.1.2 TensorBoard

TensorBoard is a visualization tool that can monitor PyTorch training and identify the bottleneck. It is easy to use and can be accessed from any web browser. TensorBoard can monitor various metrics, including data loading time, memory copy time, CPU usage, and GPU usage.

Here are step-by-step instructions on how you can integrate TensorBoard with your PyTorch.

- Step 1: Installation: Install TensorBoard by running the following command:

```
pip install torch torchvision torch_tb_profiler tensorboard
```

- Step 2: Import: In your Python script or Jupyter Notebook, import the necessary modules:

```
from torch.utils.tensorboard import SummaryWriter
```

- Step 3: Initialize SummaryWriter: Create an instance of the SummaryWriter class in your Python script:

```
writer = SummaryWriter()
```

- Step 4: Create a profiler to record execution events. Here's an example:

```
with torch.profiler.profile(
        schedule=torch.profiler.schedule(wait=1, warmup=2, active=3,
repeat=1),

 on_trace_ready=torch.profiler.tensorboard_trace_handler('./log/resnet50'),
        record_shapes=True,
        profile_memory=True,
        with_stack=True
) as prof:
    for step, data in enumerate(train_loader):
        …
        train(data)
        prof.step()
```

- Step 5: Launch the TensorBoard

```
tensorboard --logdir=./log
```

- Step 6: Open the TensorBoard profile URL in a web browser http://localhost:6006/#PyTorch_profiler

You will see a dashboard as below:

In the following sections, we will discuss how to read the information from this dashboard.

2.1.3 Visdom

For PyTorch 2.0, TensorBoard will not work unless you implement your own SummaryWriter, but you can use Visdom instead.

Here are step-by-step instructions on how you can integrate TensorBoard with your PyTorch.

- Step 1: Installation: Install Visdom by running the following command:

```
pip install visdom
```

- Step 2: Import: In your Python script or Jupyter Notebook, import the necessary modules:

```
import visdom
```

- Step 3: Start Visdom Server: Start the Visdom server to serve the visualizations by running the following command in your terminal:

```
python -m visdom.server
```

- Step 4: Initialize Visdom: Create an instance of the visdom.Visdom class in your Python script:

```
viz = visdom.Visdom()
```

Step 5: Create a Function to Monitor GPU Utilization: Define a function that monitors the GPU utilization and updates the visualization using Visdom. Here's an example:

```
def monitor_gpu_utilization():
    while True:
        # Get GPU utilization using PyTorch
        gpu_utilization = torch.cuda.max_memory_allocated() / (1024**3) # 
Convert bytes to GB

        # Update the Visdom plot
        viz.line(
            X=[i],
            Y=[gpu_utilization],
            win='gpu_util',
            opts={'title': 'GPU Utilization', 'xlabel': 'Time', 'ylabel': 
'Utilization (GB)'},
            update='append' if i > 0 else None
        )

        # Sleep for a certain period (e.g., 1 second) before updating the 
plot again
        time.sleep(1)
```

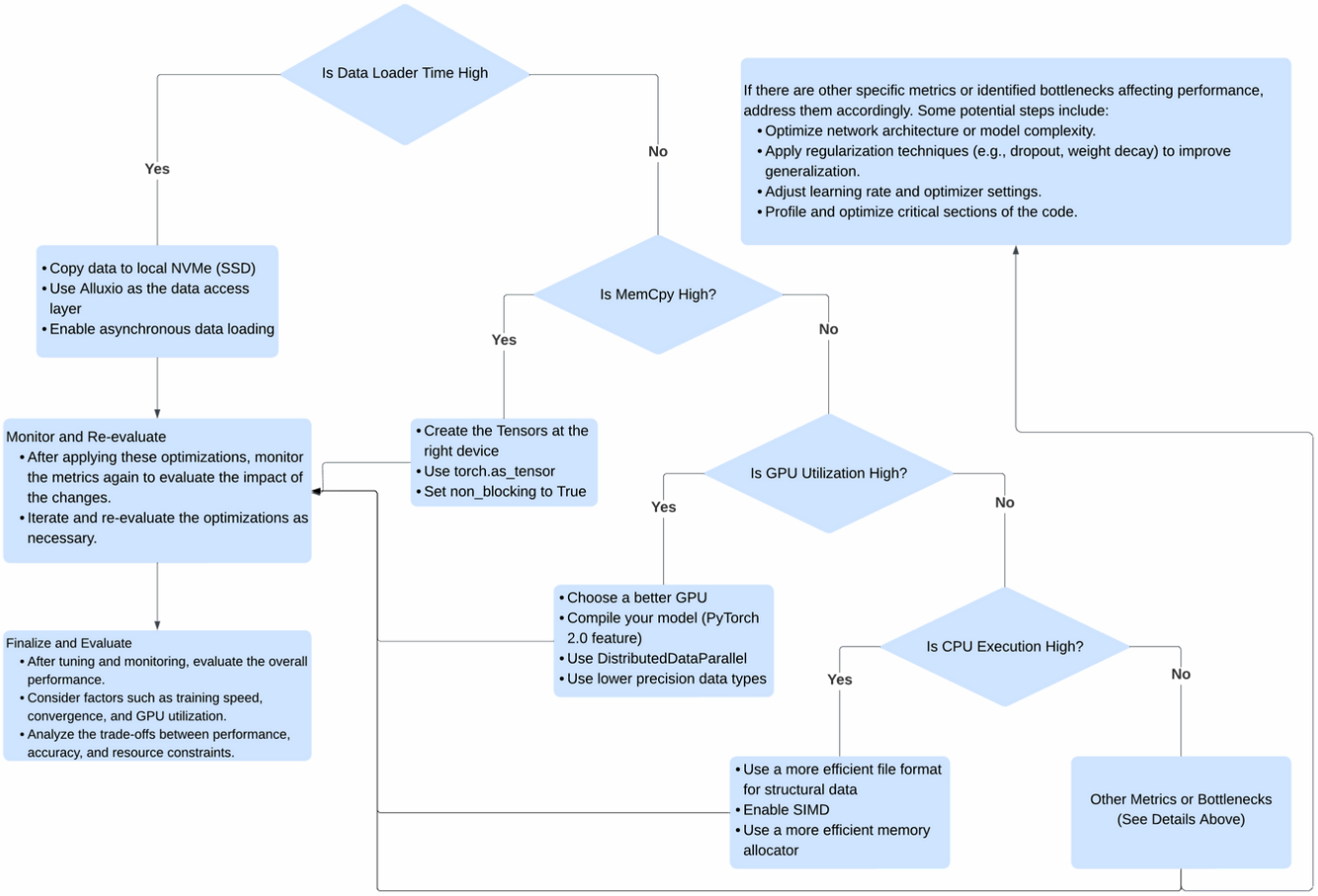
- Step 6: Start Monitoring: Start the monitoring function to continuously update the GPU utilization plot.

```
monitor_gpu_utilization()
```

- Step 7: View the Plot: Open your web browser and go to the Visdom server's URL (usually http://localhost:8097) to view the GPU utilization plot. The plot will update in real time, displaying the GPU utilization over time.

2.2 The Performance Tuning Process

Here's a process for tuning PyTorch training based on key metrics such as GPU utilization, CPU execution, data loader time, and memory copy (memcpy). This flow is presented in a decision tree:



This tuning process provides a structured approach to address performance bottlenecks in PyTorch based on key metrics. By following this decision tree, you can systematically identify the relevant bottlenecks and apply appropriate optimization techniques to improve the efficiency and performance of your PyTorch training pipeline.

2.3 Optimize I/O Performance

I/O is a major bottleneck in slowing the training process because data must be loaded from storage to GPU for processing. This process can be very time-consuming, especially when your datasets are large or remote. In our experience working with top tech companies, such as Uber, Shopee, and AliPay, data loading can account for nearly 80% of end-to-end training time. You can typically see performance improvements up to 10x by just optimizing I/O.

Moreover, slow I/O leads to underutilized GPUs. Given the scarcity and high cost of GPUs, you want to fully utilize them during training. You need to ensure that the data is fed to the GPU at the rate that matches its computations. If I/O is slow, the GPUs remain idle until the data is ready, wasting expensive resources. We typically observe GPU utilization of less than 50% because of slow I/O.

You can optimize I/O in the following ways:

- Copy your data to a faster local storage device before loading, such as a solid-state drive (SSD).
- Deploy a high-performance data access layer with a cache on top of your storage.
- Parallelize the data loading process, such as using multiple PyTorch workers to load the data.

2.3.1 Copy Data to Local NVMe (SSD)

A single NVMe (SSD) drive can offer up to 7 Gbps input/output (I/O) bandwidth. So preloading the data utilized by the active training job onto the NVMe disk in advance will greatly speed up the data loading process.

Limitations of this approach:

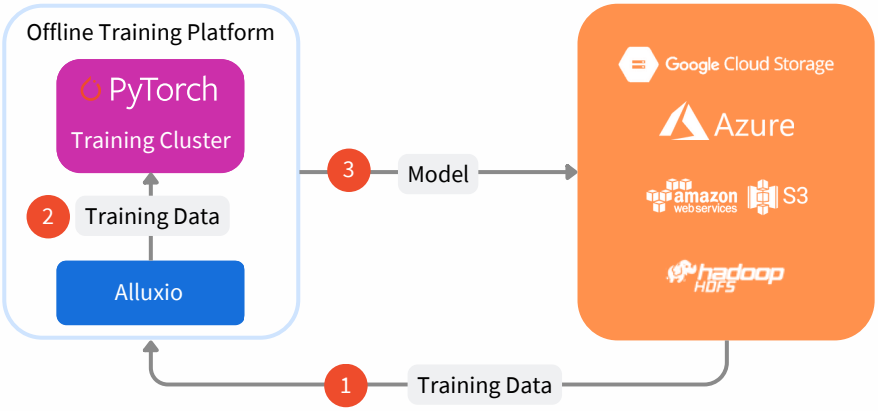
- The local disk has limited storage capacity and may not be sufficient to accommodate the entire training dataset.
- Storing sensitive data on the local disk may violate GDPR regulations.
- It is burdensome to manage copying and removing inactive data from the disk promptly.

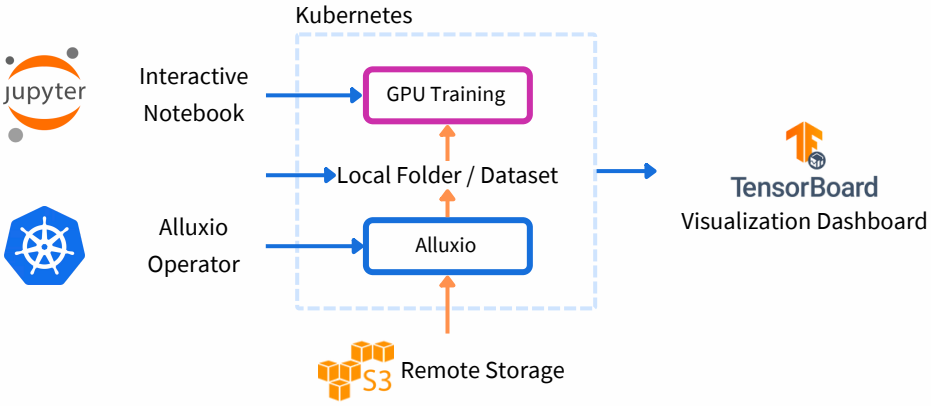
2.3.2 Use Alluxio as the High-performance Data Access Layer

Since copying data to local SSDs has limitations, there is a better way to optimize I/O – using Alluxio as a high-performance data access layer. Alluxio provides the following values:

- Automatically load / unload / update data from your existing data lake.
- Faster access to training data with optimized I/O.
- Increase the productivity of the data engineering team by eliminating the need to manage data copies.
- Reduce cloud storage API and egress costs, such as the cost of S3 GET requests, data transfer costs, etc.
- Maintain an optimal I/O with high data throughput to keep GPU fully utilized.

Below is an architecture diagram of Alluxio with PyTorch. You can deploy Alluxio co-located with the training cluster to get the maximum performance. Alluxio's distributed caching leverages the local NVMe (SSD) disk to cache remote data. On-demand loading retrieves data from remote mass storage, and inactive data is evicted using LRU or FIFO algorithms.

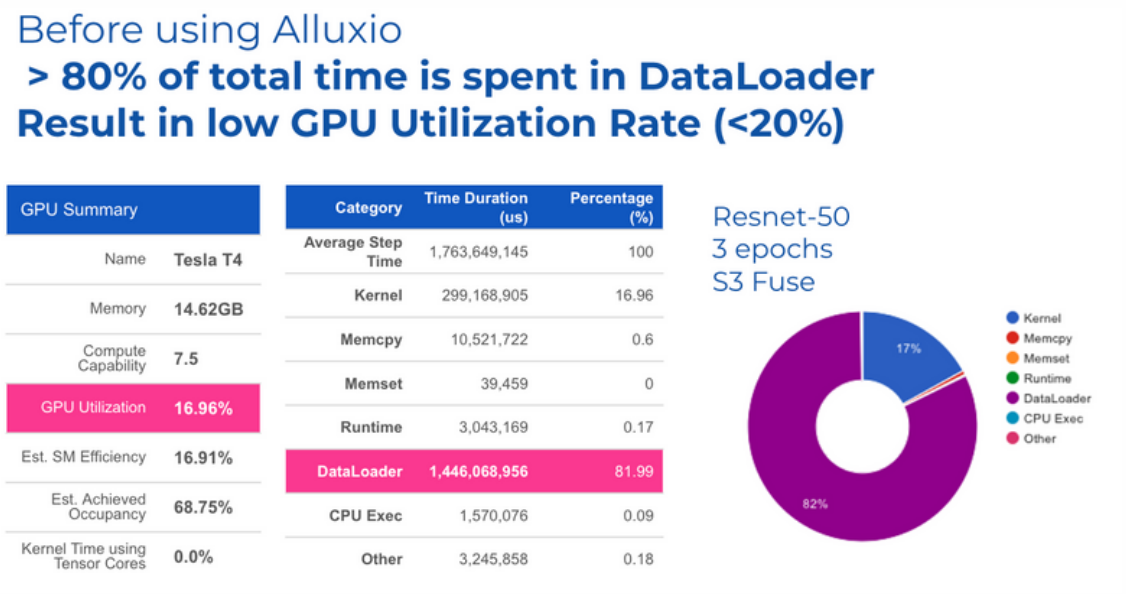


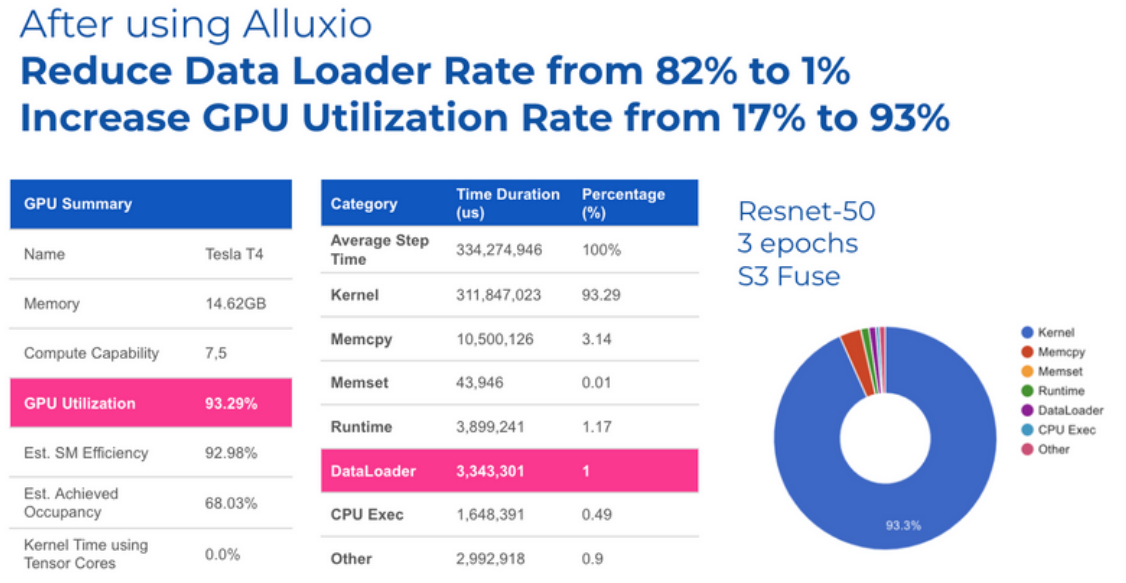


As shown, remote data can be mounted into a local folder, eliminating the need to modify existing code that relies on local datasets for training. The entire dataset can be mounted, but only the actively used data is cached or stored in the local disk. Alluxio provides near NVMe I/O bandwidth for hot reads or network I/O bandwidth for code reads, enhancing training data accessibility.

With optimized I/O and high data throughput, the GPUs are kept busy without waiting for network I/O as data is cached locally on compute instances across different steps of the training pipeline. You will achieve better performance at a significantly lower cost with full utilization of GPU resources.

Here are the results we obtained for our experiment on ResNet50 (in comparison to S3-fuse). For more details, check out the video here: https://www.youtube.com/watch?v=v3iPfnJV2ZM.



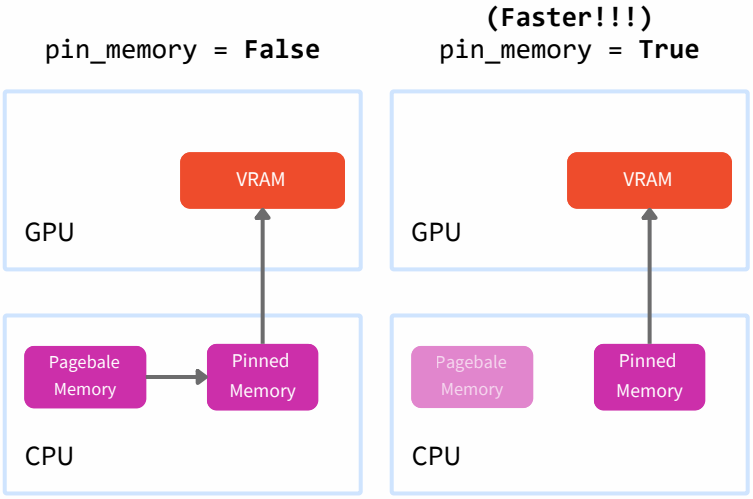


2.3.3 Enable Asynchronous Data Loading

By default, the data loading in PyTorch is synchronous, which means the training process has to wait until the training data is available.

Set the value of num_workers to be greater than 0 could enable the asynchronous data loading.

Furthermore, set pin_memory=True could enable the asynchronous memory copy from the host to the GPU by using pinned memory.



Step 5: Create a Function to Monitor GPU Utilization: Define a function that monitors the GPU utilization and updates the visualization using Visdom. Here's an example:

```
loader=DataLoader(
    data_set,
    …
    num_workers=2,
    pin_memory=True)
```

Note that it only helps the loading of Tensors or the maps and iterables of Tensors.

For the customized type, You can enable the pin_memory by defining a pin_memory() method on your custom type.

```
class SimpleCustomBatch:
    def __init__(self, some_data):
        …
        self.some_tensor = …

    # custom memory pinning method on custom type
    def pin_memory(self):
        self.some_tensor.pin_memory()
        return self
```

2.4 Create the Tensors at the Right Device

2.4.1 CREATE THE TENSORS AT THE RIGHT DEVICE

We highly recommend directly creating tensors on the device where you intend to perform operations. Consider the code below, where a tensor is initially created in CPU memory and then moved to GPU memory. This approach typically involves allocating CPU RAM and using the cudaMemcpy function to transfer data between the GPU and CPU.

```
import torch

tensor_x = torch.randn([10, 5] #Tenser created on CPU's memory
device = torch.device("cuda")
tensor_x = x.to(device) #Tensor moved to GPU's memory
```

If you notice significant time spent on memory copying (memcpy) in the performance dashboard, and if you intend to use tensors on the GPU, it is advisable to create them directly on the GPU, as shown in the code below:

```
tensor_cuda = torch.randn((), device=device=torch.device('cuda'),
dtype=dtype)
```

By creating tensors directly on the GPU, unnecessary memory transfers can be avoided, leading to improved performance and reduced overhead. This approach is particularly beneficial when working extensively with GPU operations in deep learning tasks.

2.4.2 Use torch.as_tensor (others)

Using torch.as_tensor() provides several benefits. Unlike torch.tensor(), it avoids unnecessary data copying. It converts data into a tensor while sharing the underlying data and preserving the autograd history, if possible.

Suppose the data is already a tensor with the requested data type and device, torch.as_tensor() simply returns the data itself. However, if the data is a tensor with a different data type or device, it creates a new tensor by copying the data using data.to(dtype=dtype, device=device).

When the data is a NumPy array (ndarray) with the same data type and device, torch.as_tensor() constructs a tensor using torch.from_numpy(). This method, which is invoked internally by as_tensor(), is significantly faster than using torch.tensor().

In summary, torch.as_tensor() efficiently converts data into a tensor, allowing for memory sharing and preserving the autograd history. It provides a faster alternative to torch.tensor() when working with NumPy arrays. By using torch.as_tensor(), unnecessary data copying can be avoided, resulting in improved performance and efficient tensor creation.

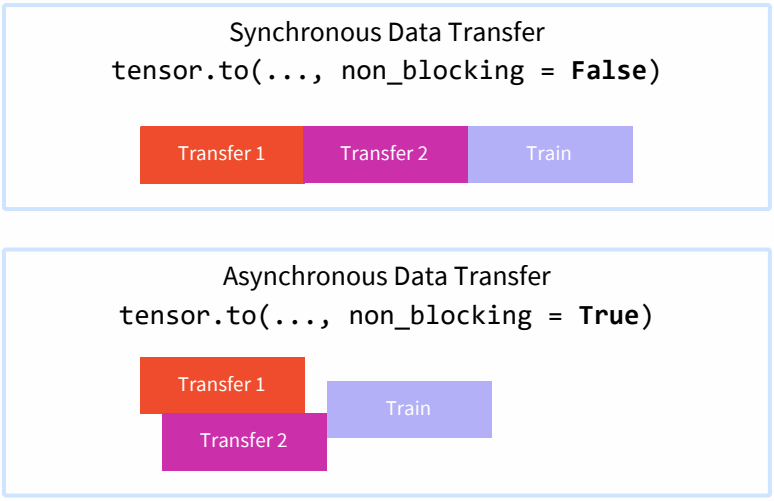
2.4.3 Set non_blocking to True

The following examples show how we can load data to a GPU device memory in parallel with non_blocking=True.

```
data1 = data1.to('cuda:0', non_blocking=True) #transfer1
data2 = data2.to('cuda:0', non_blocking=True) #transfer2

output = model(data) # synchronous point
```

The transfer1 and transfer2 could be processed in parallel, as the diagram shows below. The model training has to wait for data to be pushed onto the device (synchronous point).



Overlap the data transfers to reduce the run time.

Here is another example to show we do the asynchronous CPU to GPU Copy:

```
# Create CPU and GPU tensors
cpu_tensor = torch.randn(32, 3, 224, 224)
gpu_tensor = torch.empty(32, 3, 224, 224).cuda()

# Perform asynchronous copy from CPU to GPU with non_blocking=True
gpu_tensor.copy_(cpu_tensor, non_blocking=True)
```

Using non_blocking=True is particularly beneficial when there is a need for overlapping data transfer and computation. It helps to avoid unnecessary synchronization between CPU and GPU, improving the overall performance and efficiency of PyTorch computations.

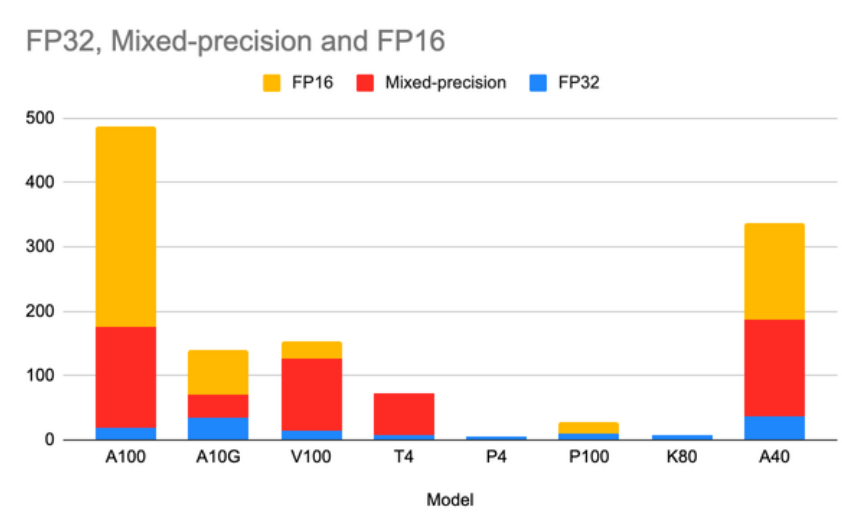
2.5 GPU-Specific Optimization

2.5.1 HOW TO CHOOSE THE RIGHT GPU

The choice of GPU is crucial in the quality of your deep learning experience. Here are some tips for selecting the most suitable GPUs for your needs:

- Compute Capability and Memory: Take into account the compute capability and memory capacity of the GPUs. Higher compute capabilities enable advanced deep learning features and algorithms, while larger memory capacities handle bigger models and datasets. Ensure that the chosen GPUs meet the requirements of your deep learning tasks.
- Performance vs. Cost: Assess the performance-to-cost ratio of the GPUs. Seek GPUs that strike a balance between performance and cost based on your budget and specific deep learning requirements. Factors such as the number of CUDA cores, memory bandwidth, and pricing should be considered to make an informed decision.
- Compatibility with Deep Learning Frameworks: Verify that the chosen GPUs are compatible with popular deep learning frameworks like PyTorch or TensorFlow. Consult the frameworks' documentation or official resources for a list of supported GPUs and recommended configurations.

Here is a diagram illustrating the compute capabilities of different GPU models.



To accelerate PyTorch code with the greatest speedup, you should consider modern NVIDIA GPUs such as A100, H100, and V100. These GPUs offer exceptional performance, particularly for the "torch.compile" feature we will discuss in the upcoming section.

You can use the code below to determine if your GPU can get the best performance with the most modern feature in PyTorch.

```
//code from PyTorch official site
import torch
import warnings

gpu_ok = False
if torch.cuda.is_available():
    device_cap = torch.cuda.get_device_capability()
    if device_cap in ((7, 0), (8, 0), (9, 0)):
        gpu_ok = True

if not gpu_ok:
    warnings.warn(
        "GPU is not NVIDIA V100, A100, or H100. Speedup numbers may be
lower "
        "than expected."
    )
```

By considering these tips and selecting GPUs that align with your specific deep learning requirements, you can ensure a fast and efficient deep learning training.

2.5.2 Compile Your Model

You can compile your model using PyTorch 2.0 and modern GPUs like A100 using the code below.

```
//code from PyTorch official site
def foo(x, y):
    a = torch.sin(x)
    b = torch.cos(y)
    return a + b
opt_foo1 = torch.compile(foo)
print(opt_foo1(torch.randn(10, 10), torch.randn(10, 10)))
```

You can pass whatever Python function you want to optimize to torch.compile. Then you can use the return value from torch.compile as the optimized function to replace the original function.

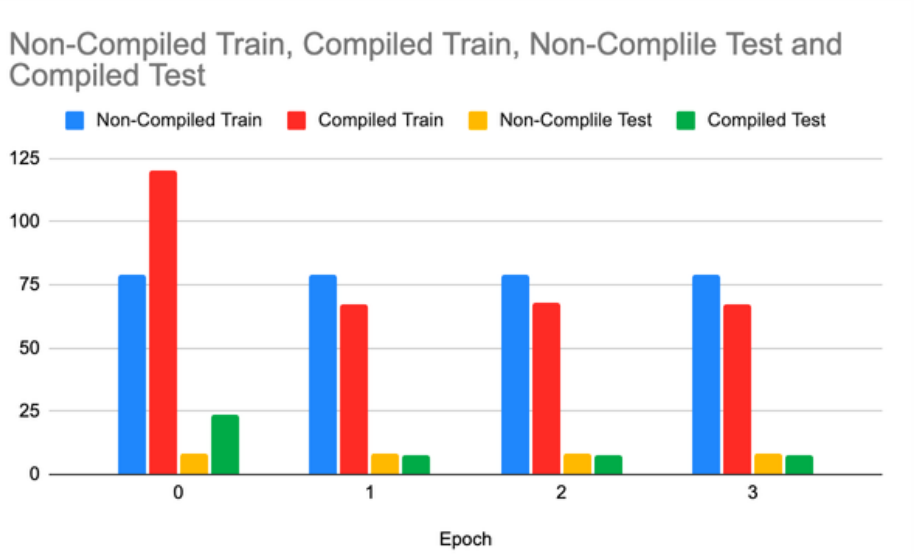
If you do not want to do the replacement, you can also decorate the existing function by "@torch.compile". Then you can still use the original function name in your existing code.

```
//code from PyTorch official site
@torch.compile
def foo(x, y):
    a = torch.sin(x)
    b = torch.cos(y)
    return a + b
print(foo(torch.randn(10, 10), torch.randn(10, 10)))
```

You can also compile a model, it benefits both model training and testing

```
mod = MyModule()
opt_mod = torch.compile(mod)
print(opt_mod(torch.randn(10, 100)))
```

Note that the functions and models optimized by torch.compile might take much longer in the first iteration, as it must compile the model, but in subsequent iterations, we see significant speedups compared to original ones.



The torch.compile feature in PyTorch makes functions faster through operation fusion and graph capture. Operation fusion condenses multiple operations into fewer ones, reducing overhead. Graph capture enables dynamic graph analysis and optimizations during the forward pass, further improving performance. Together, they streamline execution, eliminate redundant computations, and enhance overall efficiency in PyTorch training.

2.5.3 Use DistributedDataParallel (DDP)

Using DistributedDataParallel (DDP) in PyTorch enables distributed training across multiple GPUs or machines, significantly boosting compute power and reducing training time. When your model exceeds the capacity of a single GPU, DDP becomes essential.

To implement DDP, import the necessary modules.

```
import torch.distributed as dist

import torch.multiprocessing as mp

from torch.nn.parallel import DistributedDataParallel as DDP
```

And then initialize the process group.

```
os.environ['MASTER_ADDR'] = 'localhost'
os.environ['MASTER_PORT'] = '12355'
dist.init_process_group("gloo", rank=rank, world_size=world_size)
```

Then, wrap your existing model with DDP for parallel training.

```
model = nn.Linear(10, 10).to(rank) # your existing model
ddp_model = DDP(model, device_ids=[rank]) # wrap the model as a DDP model
```

For basic use cases, DDP only requires a few more codes to set up the process group. Compared with the well-known DataParallel, the pros of using DDP are

- Model is replicated only once
- Supports scaling to multiple machines
- Faster because it uses multiprocessing

Overall, DistributedDataParallel (DDP) in PyTorch provides efficient multi-GPU training, improved scalability, and a simplified implementation process. However, it also introduces communication overhead, increased memory consumption, complex debugging scenarios, and dependencies on hardware and network configurations. Carefully consider these factors when deciding to use DDP for distributed training in PyTorch.

2.5.4 Use Lower Precision Data Types

Reduced Precision Training (RPT) in PyTorch involves using lower-precision numerical formats, such as half-precision floating-point, for model parameters and computations. There are several benefits to using reduced precision:

- Memory Efficiency: Lower-precision formats require less memory, allowing for training and deployment of larger neural networks within available GPU memory constraints.
- Faster Data Transfer: Reduced precision requires less memory bandwidth, leading to faster data transfer operations between GPU memory and other components, improving overall performance.
- Accelerated Math Operations: Math operations run faster with reduced precision, especially on GPUs with Tensor Core support. For example, float16 matrix multiplication and convolution can be up to 16 times faster than float32 on A100 GPUs.

PyTorch's Automatic Mixed Precision (AMP) package is recommended for achieving the benefits of reduced precision while maintaining accuracy. AMP identifies steps that require full precision and uses 32-bit floating point for those specific steps, while using lower precision (e.g., float16 or bfloat16) elsewhere.

The autocast context manager or decorator in AMP allows specific regions of the script to run in mixed precision. By wrapping the forward pass(es) of the network, including loss computations, in autocast, operations run in an op-specific dtype chosen by autocast to improve performance while preserving accuracy.

```
with autocast(device_type='cuda', dtype=torch.float16):
output = model(input)
 loss = loss_fn(output, target)
```

It is important to note that backward passes under autocast are not recommended, as backward ops should run in the same type that autocast used for the corresponding forward ops.

Compatibility and support for reduced precision training may vary across hardware architectures and software libraries. Hardware with FP16 support, specifically Tensor Cores, is recommended. Volta architecture or later, such as V100 or A100 GPUs, are commonly recommended for reduced precision training.

By leveraging reduced precision training and PyTorch's AMP package, significant memory savings, faster data transfer, and accelerated math operations can be achieved, enhancing the efficiency and performance of deep learning models.

2.6 CPU-Specific Optimization

When your PyTorch training job is CPU-bound, optimizing the CPU performance and reducing computational bottlenecks can significantly improve the training speed and efficiency. Besides using CPUs with higher core counts or CPUs specifically designed for high-performance computing (HPC) tasks, here are several strategies to optimize a CPU-bound PyTorch training jobs.

2.6.1 Use More Efficient File Format For Structural Data

Parquet and Arrow are complementary technologies with different design tradeoffs. Parquet is a storage format focused on maximizing space efficiency, while Arrow is an in-memory format optimized for vectorized computational kernels.

The main distinction between the two is that Arrow enables O(1) random access lookups to any array index, whereas Parquet does not. Parquet achieves space efficiency through techniques like dremel record shredding, variable length encoding schemes, and block compression. However, these techniques come at the cost of performant random access lookups.

A common approach that leverages both technologies' strengths is to stream data in thousand-row batches from a compressed representation like Parquet into the Arrow format. These batches can then be processed individually, allowing for efficient computations on the Arrow data while managing memory requirements. The results can be accumulated in a more compressed representation. This approach ensures that the computation kernels are agnostic to the encodings of the source and destination data.

Based on the identified bottleneck in section 2.1, if the CPU is the bottleneck, it is recommended to use Arrow to save CPU resources. Arrow's efficient computation capabilities make it a suitable choice in this scenario. On the other hand, if the bottleneck is IO, using Parquet can help reduce the number of bytes loaded from the disk or transmitted over the network, optimizing IO performance.

By considering the specific bottleneck and leveraging the strengths of Parquet and Arrow accordingly, one can make informed decisions to improve CPU or IO performance in data processing workflows.

2.6.2 Enable SIMD

SIMD stands for Single Instruction, Multiple Data. It is a type of parallel processing that allows a single instruction to be executed on multiple data points simultaneously. This can speed up computation significantly, reducing the number of instructions that need to be executed.

No code modifications are needed if you want to enable the SIMD in PyTorch. Simply replace Pillow with Pillow-SIMD. Pillow-SIMD is a fork of Pillow that leverages the CPU's SIMD (Single Instruction, Multiple Data) instructions to accelerate image preprocessing operations.

Alternatively, the NVIDIA DALI library can offload preprocessing operations to the GPU, completely freeing up the CPU. However, for computer vision job, DALI's API is significantly different from torchvision.transforms, making it less user-friendly. Therefore, this section focuses mainly on introducing Pillow-SIMD.

Pillow, as a drop-in replacement for PIL, is already known for its speed advantage over other image processing libraries. However, Pillow-SIMD was created to enhance the performance further. According to benchmarks, Pillow-SIMD is 4-6 times faster than Pillow and other libraries like ImageMagick, OpenCV, and IPP on the same hardware/platform. It achieves this speed improvement by optimizing common image manipulation instructions using the SIMD approach, which enables parallel processing of multiple data points simultaneously.

Note that when installing Pillow-SIMD in a conda virtual environment, other package installations may update Pillow-SIMD to the latest version of Pillow due to conda's automatic dependency handling. In that case, Pillow-SIMD needs to be installed again.

Firstly, we should remove pil, pillow, jpeg, libtiff and libjpeg-tubopackages:

```
conda uninstall -y --force pillow pil jpeg libtiff libjpeg-turbo
pip uninstall -y    pillow pil jpeg libtiff libjpeg-turbo
```

Then, install pillow-simd with SIMD enabled.

```
conda install -yc conda-forge libjpeg-turbo
CFLAGS="${CFLAGS} -mavx2" pip install --upgrade --no-cache-dir --force-
reinstall --no-binary :all: --compile pillow-simd
```

2.6.3 Use More Efficient Memory Allocator

Jemalloc and TCMalloc are memory allocators that outperform the default malloc in deep learning. They reduce memory fragmentation, optimize allocation and deallocation, improve thread scalability, and handle large memory footprints efficiently. With improved memory management, they minimize overhead, enhance memory utilization, and boost overall system performance for model training tasks.

Switch the memory allocator by updating the environment variable LD_PRELOAD.

```
export LD_PRELOAD=jemalloc.so:$LD_PRELOAD
```

Or

```
export LD_PRELOAD=tcmalloc.so:$LD_PRELOAD
```

CHAPTER

03

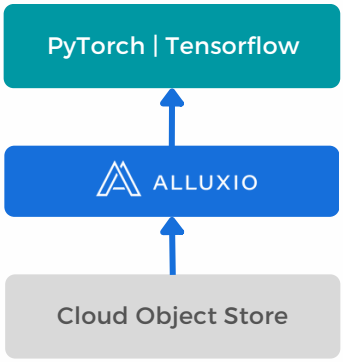
Chapter 3: Real-world Case Studies Using Alluxio as the Data Access Layer

3.1 AliPay: Speed Up Large-Scale Computer Vision Model Training on Billions of Files

Alipay is one of the world's largest mobile payment platforms, serving over 1.3 billion users and 80 million merchants. In order to provide its users with the best possible experience, Alipay relies on machine learning models to power a variety of features, such as fraud detection, risk assessment, and personalized recommendations.

However, as Alipay's user base and transaction volume grew, the company began to experience challenges with model training. The disparity between computation and storage performance was causing model training to be slow and inefficient. Additionally, the high cost of specialized hardware was putting a strain on Alipay's budget.

To address these challenges, Alipay began using Alluxio, a unified data access layer that can accelerate machine learning workloads. Alluxio provides a high-performance cache that sits between the compute and storage layers, reducing latency and improving throughput. This allows Alipay to train models on commodity hardware, which is much more cost-effective than specialized hardware.



In addition to improving performance, Alluxio also simplifies data management for Alipay. Alluxio eliminates the need to maintain data copies by providing on-demand data access. This frees up data engineers to focus on other tasks, such as optimizing model performance.

As a result of using Alluxio, Alipay has seen significant improvements in the speed and efficiency of its model training. The company has also reduced its infrastructure costs and freed up data engineers to focus on more strategic tasks. Learn more here >>.

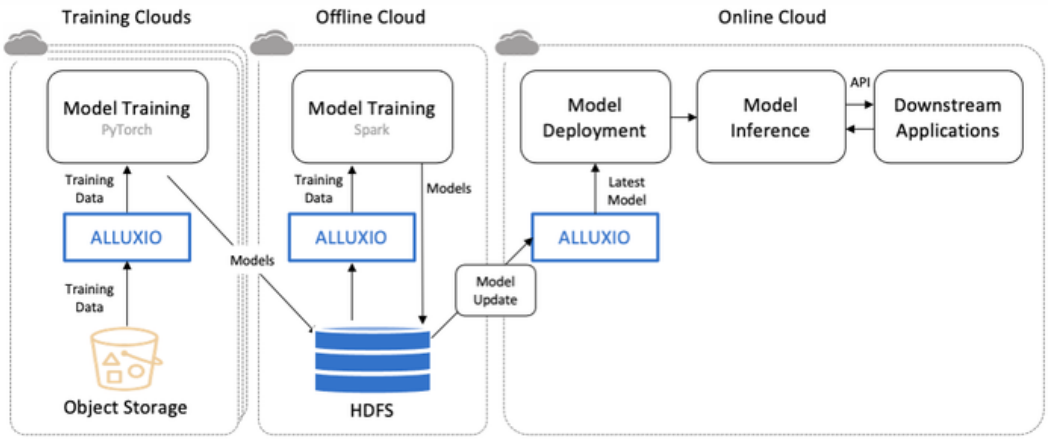
> "After attempting various methods to address our challenges, only Alluxio is able to meet our requirements for large-scale AI training. Alluxio has significantly enhanced our AI training jobs for our businesses in various domains."
>
> *— Chuanying Chen, Senior Software Engineer at AliPay*

3.2 Zhihu: Accelerated LLM Model Training with 90% GPU Utilization

Zhihu (NYSE: ZH) is a leading online content community in China with 400 million users, 100 million MAU, and 54 billion monthly views currently. Zhihu trains custom large language models (LLMs) to power its search and recommendation features. To develop LLMs, Zhihu needed a high-performance data access layer to access data from multiple clouds efficiently.

The Zhihu team faced several challenges building a high-performance data access layer for LLMs. First, they needed to find a way to access data from multiple clouds efficiently. Second, they needed to ensure that the data access layer was scalable to meet the growing demands of LLM training and deployment. Third, they needed to ensure the data access layer was reliable and could withstand unexpected failures.

The Zhihu team chose to use Alluxio as the high-performance data access layer for LLMs. Alluxio provides an acceleration service for large-scale data access. Alluxio acts as a unified acceleration solution for large-scale data access to model training and deployment.



After adopting Alluxio, Zhihu saw significant performance, scalability, and reliability improvements. They could train LLMs 2-3 times faster and deploy updated models every minute instead of hours or days. They also saw a 50% reduction in infrastructure costs. Read the full story here >>.

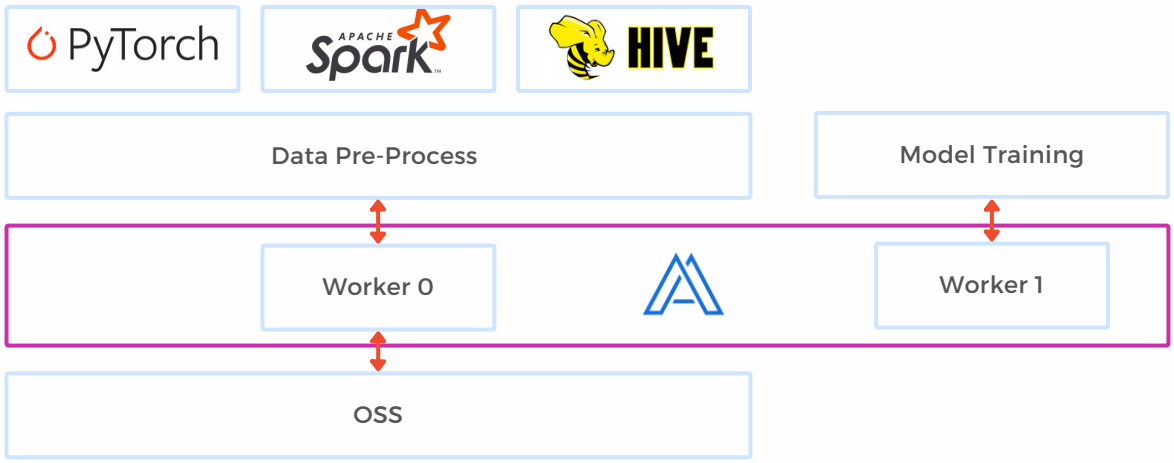
> We choose Alluxio as the high-performance data access layer to tackle our technical challenges. As a result, we've achieved a 90% GPU utilization, 50% reduced infrastructure and operations costs, and accelerated model deployment and update times from several hours to minutes.
>
> *— Mengyu Hu, Software Engineer in the data platform team at Zhihuy*

3.3 Bilbili: 3x Training Performance and Data Sharing Between Preprocessing and Training

Bilibili (NASDAQ: BILI) is a leading video community with 230 million monthly active users. The company uses machine learning to power a variety of features, such as video recommendations, content moderation, and ad targeting.

Bilibili needed a high-performance data access layer to support its growing machine learning workloads. The company's previous data access layer was based on HDFS, but it could not keep up with the demands of machine learning.

Bilibili began using Alluxio as its data access layer for machine learning. Alluxio is a distributed filesystem that provides a unified view of data from multiple storage systems. This allows Bilibili to access data from HDFS, OSS, and other storage systems without having to move the data.



Alluxio provides a high-performance cache between the compute and storage layers. This cache reduces latency and improves throughput, which can significantly improve the performance of machine learning workloads.

As a result of using Alluxio, Bilibili has seen 3x improvements in the performance of its machine learning workloads. The company has also reduced its infrastructure costs and improved model quality. Read more here >>.

> Alluxio has powered our AI platform in both data pre-processing and model training stages. Using Alluxio as a data layer between compute and storage, our AI platform has improved the training efficiency and model accuracy with data access simplified from the user side.
>
> *— Lei Li, AI Platform Lead at Bilibili*

CHAPTER

04 Summary and Additional Resources

4.1 Key Takeaways

In this comprehensive guide, you have learned various practices and techniques to optimize PyTorch model training performance.

As we wrap up, let us reflect on the key takeaways:

- Identify Performance Bottlenecks: Using command line tools and TensorBoard, you can analyze your model performance and identify any bottlenecks. Addressing these bottlenecks is the first step toward optimizing your PyTorch model training.
- Optimize I/O: Techniques such as copying data to a local NVMe (SSD), using Alluxio as a data access layer, and enabling asynchronous data loading can significantly speed up data loading, reducing the time for model training.
- Enhance Data Operations: Create tensors on the right device using the torch.as_tensor function, and setting non_blocking to True can help optimize data operations, improving overall performance.
- GPU-Specific Optimization: Selecting the right GPU, compiling your model, using DistributedDataParallel (DDP), and employing lower precision data types are critical to harnessing the full potential of GPU processing.
- CPU-Specific Optimization: Techniques like using more efficient file formats for structural data, enabling SIMD, and using a more efficient memory allocator can help optimize CPU processing, aiding in quicker model training.
- Real-world case studies from AliPay, Zhihu, and Bilibili demonstrate the benefits of using Alluxio as the data access layer to accelerate end-to-end training speed and increase GPU utilization.

Implementing these optimization techniques will ensure efficient use of resources and faster model training, enabling a faster time-to-market for your AI-powered business.

4.2 Additional Resources

To further enhance your understanding and implementation of the techniques provided by this eBook, you can refer to the following resources:

- PyTorch performance tuning guide: https://PyTorch.org/tutorials/recipes/recipes/tuning_guide.html
- NVIDIA deep learning performance: https://docs.nvidia.com/deeplearning/performance/index.html
- Unlock the full potential of Alluxio in your AI infrastructure:
  - Learn more about Alluxio's architecture, deployment, and configuration best practices: https://docs.alluxio.io/
  - Attend online events to learn from experts and share your experiences with the community: https://www.alluxio.io/product-school/
  - Join 11k+ members in the Alluxio community slack channel to ask any questions and provide your feedback: https://alluxio.io/slack

References

[1] PyTorch: An Imperative Style, High-Performance Deep Learning Library: https://arxiv.org/pdf/1912.01703.pdf
[2] PyTorch website: https://PyTorch.org/
[3] PyTorch Github repository: https://github.com/PyTorch/PyTorch
[4] PyTorch performance tuning guide: https://PyTorch.org/tutorials/recipes/recipes/tuning_guide.html
[5] PyTorch distributed and parallel training tutorial: https://pytorch.org/tutorials/distributed/home.html
[6] Towards Data Science | Optimize PyTorch Performance for Speed and Memory Efficiency (2022): https://towardsdatascience.com/optimize-pytorch-performance-for-speed-and-memory-efficiency-2022-84f453916ea6
[7] NVIDIA GTC 2021 | PyTorch Performance Tuning Guide: https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s31831/
[8] AWS blog | Optimizing I/O for GPU performance tuning of deep learning training in Amazon SageMaker: https://aws.amazon.com/blogs/machine-learning/optimizing-i-o-for-gpu-performance-tuning-of-deep-learning-training-in-amazon-sagemaker/
[9] PyTorch TensorBoard support: https://PyTorch.org/tutorials/beginner/introyt/tensorboardyt_tutorial.html
[10] Alluxio | Alipay: Optimizing Alluxio for Efficient Large-Scale Training on Billions of Files: https://www.alluxio.io/blog/optimizing-alluxio-for-efficient-large-scale-training-on-billions-of-files/
[11] Alluxio | Building High-performance Data Access Layer for Model Training and Model Serving for LLM: https://www.alluxio.io/blog/building-high-performance-data-access-layer-for-model-training-and-model-serving-for-llm/
[12] Alluxio | When AI Meets Alluxio at Bilibili | Building an Efficient AI Platform for Data Preprocessing and Model Training: https://www.alluxio.io/blog/when-ai-meets-alluxio-at-bilibili-building-an-efficient-ai-platform-for-data-preprocessing-and-model-training/

Book Authors

Dr. Beinan Wang is a Senior Staff Software Engineer at Alluxio and a committer and TSC of PrestoDB. Prior to Alluxio, he was the Tech Lead of the Presto team at Twitter and he built large-scale distributed SQL systems for Twitter's data platform. He has twelve-year of experience working on performance optimization, distributed caching, and volume data processing. He received his Ph.D. in computer engineering from Syracuse University on the symbolic model checking and runtime verification of distributed systems.

Hope Wang has a decade of experience in Data, AI, and Cloud. An open-source contributor to Alluxio, Trino, and PrestoDB. She also holds AWS Certified Solutions Architect – Professional status. She is a Developer Advocate at Alluxio and previously worked in venture capital and as a Data Architect. She earned a BS in Computer Science, a BA in Economics, and an MEng in Software Engineering from Peking University, as well as an MBA from USC.

Dr.Chunxu Tang is a Research Scientist at Alluxio and a committer of PrestoDB. Prior to Alluxio, he served as a Senior Software Engineer in Twitter's data platform team, where he gained extensive experience with a wide range of data systems, including Presto, Zeppelin, BigQuery, and Druid. He received his Ph.D. in computer engineering from Syracuse University, where he conducted research on distributed collaboration systems and machine learning applications.

About Alluxio

Alluxio, the developer of the open source data platform, makes it easy to manage your data and serve it from any storage to any compute engine in any environment — on premise, in the cloud, or across clouds. By removing complexities and toil from managing and accessing data infrastructure, Alluxio accelerates and future-proofs your data strategy, delivering performant, accessible, cost-effective, resilient, and secure data applications that power improved outcomes, at any scale. To learn more, contact info@alluxio.com or follow us on Linkedin, or Twitter.