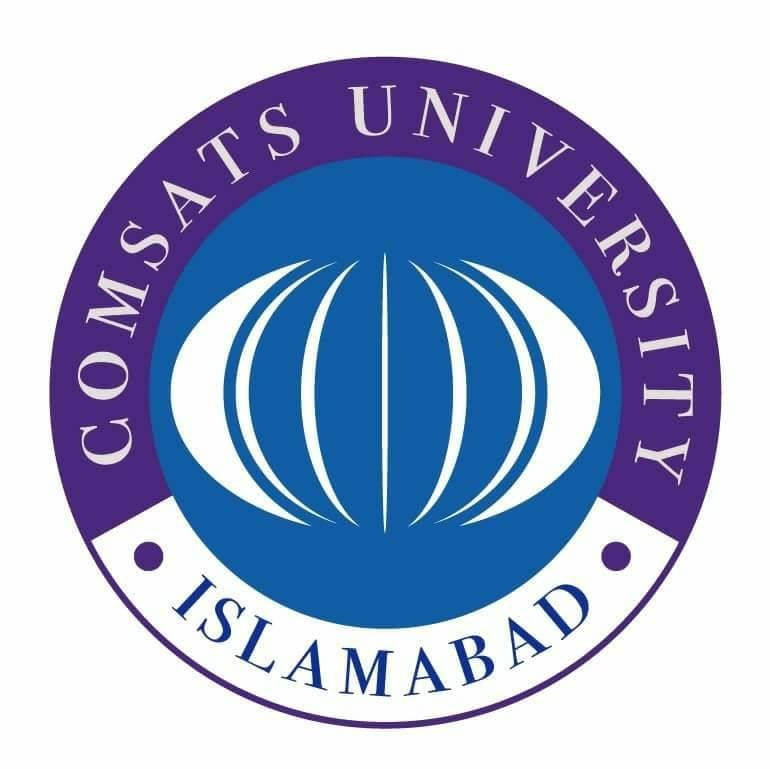
**COMSATS UNIVERSITY ISLAMABAD**

LAHORE CAMPUS



Lab Assignment 04

Submitted by : Shamama Aslam

Submitted to : Sir Akhzar Nazir

Reg no : SP23-BCS-143

Section : C

Course Title : Parallel and Distributing Computing

Date : 9 Oct. 2025

**Part 01**

import torch, torch.nn as nn, torch.optim as optim, torchvision, torchvision.transforms as transforms, time

# Dataset

trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True,

transform=torchvision.transforms.ToTensor())

trainloader = torch.utils.data.DataLoader(trainset, batch\_size=64, shuffle=True)

# Simple fully connected model

class SimpleNN(nn.Module):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.fc1 = nn.Linear(28\*28, 128)

self.fc2 = nn.Linear(128, 10)

def forward(self, x):

x = x.view(-1, 28\*28)

x = torch.relu(self.fc1(x))

return self.fc2(x)

# Training function

def train\_model(device):

model, criterion, optimizer = SimpleNN().to(device), nn.CrossEntropyLoss(), optim.Adam(SimpleNN().parameters(), 0.001)

start = time.time()

for images, labels in trainloader:

images, labels = images.to(device), labels.to(device)

optimizer.zero\_grad()

loss = criterion(model(images), labels)

loss.backward()

optimizer.step()

total = time.time() - start

print(f"Training time on {device}: {total:.2f} sec")

return total

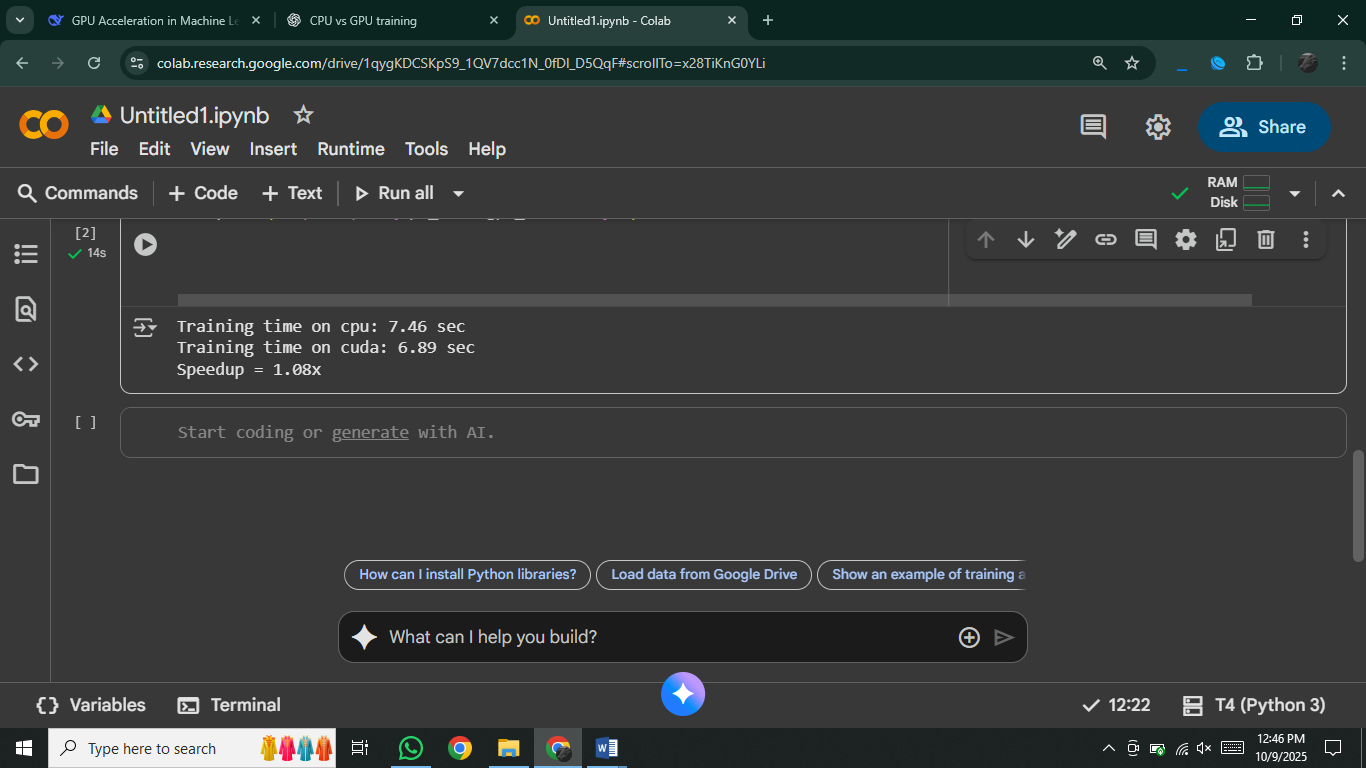
# Run CPU vs GPU

cpu\_time = train\_model(torch.device("cpu"))

if torch.cuda.is\_available():

gpu\_time = train\_model(torch.device("cuda"))

print(f"Speedup = {cpu\_time/gpu\_time:.2f}x")



### **7. Discussion**

Why GPU is faster (or not):

The **GPU is faster** because it can perform many computations in parallel, while the CPU works mostly sequentially. The actual speedup depends on dataset size, model complexity, and memory transfers. For small datasets like MNIST and simple models, the GPU advantage is smaller since data transfer overhead reduces the gain. For larger datasets and deeper models, GPUs provide much higher speedup as their parallel cores are fully utilized.

**Part 2:**

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms

import time

import matplotlib.pyplot as plt

# Dataset

transform = transforms.Compose([transforms.ToTensor()])

trainset = torchvision.datasets.MNIST(root='./data', train=True,

download=True, transform=transform)

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

# Simple model

class SimpleNN(nn.Module):

def \_\_init\_\_(self):

super(SimpleNN, self).\_\_init\_\_()

self.fc1 = nn.Linear(28\*28, 128)

self.fc2 = nn.Linear(128, 10)

self.relu = nn.ReLU()

def forward(self, x):

x = x.view(-1, 28\*28)

return self.fc2(self.relu(self.fc1(x)))

# Training function

def train\_model(batch\_size):

loader = torch.utils.data.DataLoader(trainset, batch\_size=batch\_size,

shuffle=True, num\_workers=2)

model = SimpleNN().to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

start = time.time()

correct, total = 0, 0

for images, labels in loader:

images, labels = images.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(images)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

\_, predicted = outputs.max(1)

total += labels.size(0)

correct += predicted.eq(labels).sum().item()

return time.time()-start, 100\*correct/total

# Run with different batch sizes

batch\_sizes = [16, 64, 256, 1024]

times, accs = [], []

for b in batch\_sizes:

print(f"\nBatch size {b}")

t, a = train\_model(b)

times.append(t); accs.append(a)

print(f"Time: {t:.2f} sec, Accuracy: {a:.2f}%")

# Plots

plt.plot(batch\_sizes, times, marker='o')

plt.xlabel("Batch Size"); plt.ylabel("Time per Epoch (s)")

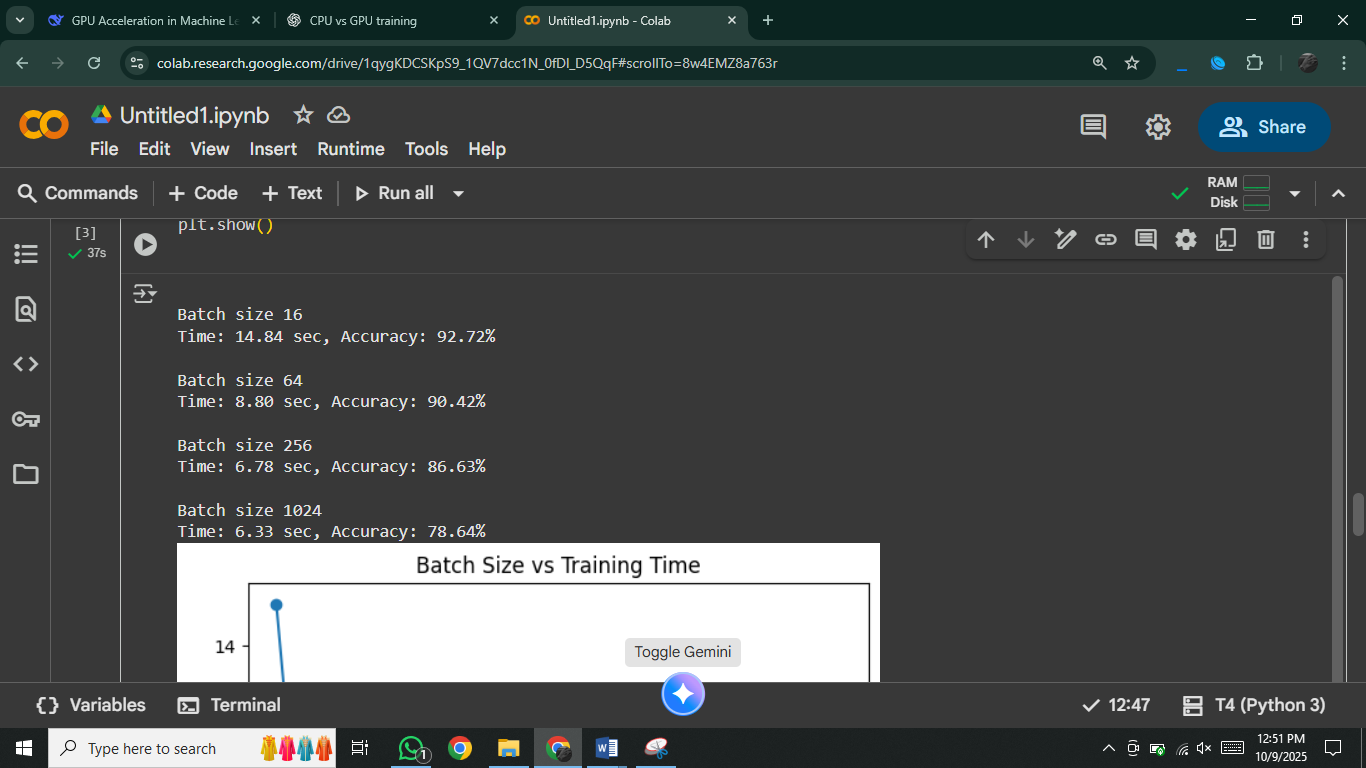
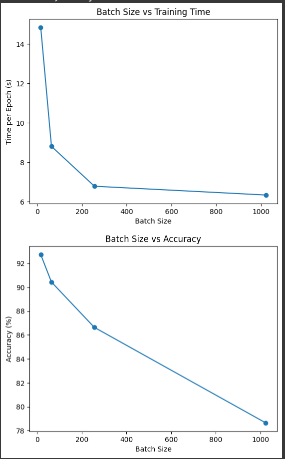
plt.title("Batch Size vs Training Time")

plt.show()

plt.plot(batch\_sizes, accs, marker='o')

plt.xlabel("Batch Size"); plt.ylabel("Accuracy (%)")

plt.title("Batch Size vs Accuracy") plt.show()



### **Why does increasing batch size improve GPU efficiency up to a point?**

**Answer:**  
Larger batch sizes allow the GPU to process more data in parallel, leading to better utilization of GPU cores and faster training per epoch. However, after a certain point, the GPU memory becomes a bottleneck, and extra-large batches don’t bring much additional speedup.

**Q2: Why does accuracy sometimes drop for very large batches?**

**Answer:**  
Very large batches reduce the randomness in gradient updates, making the model converge to sharp minima instead of flatter ones. This can lead to poorer generalization and slightly lower accuracy on unseen data, even if training loss decreases.

**Part 3:**

# Define models

class SmallNN(nn.Module):

def \_\_init\_\_(self):

super(SmallNN, self).\_\_init\_\_()

self.fc = nn.Linear(28\*28, 10)

def forward(self, x): return self.fc(x.view(-1, 28\*28))

class MediumNN(nn.Module):

def \_\_init\_\_(self):

super(MediumNN, self).\_\_init\_\_()

self.fc1 = nn.Linear(28\*28, 256)

self.fc2 = nn.Linear(256, 128)

self.fc3 = nn.Linear(128, 10)

self.relu = nn.ReLU()

def forward(self, x):

x = x.view(-1, 28\*28)

return self.fc3(self.relu(self.fc2(self.relu(self.fc1(x)))))

class LargeCNN(nn.Module):

def \_\_init\_\_(self):

super(LargeCNN, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(1, 32, 3, padding=1)

self.conv2 = nn.Conv2d(32, 64, 3, padding=1)

self.pool = nn.MaxPool2d(2, 2)

self.fc1 = nn.Linear(64\*7\*7, 256)

self.fc2 = nn.Linear(256, 10)

self.relu = nn.ReLU()

def forward(self, x):

x = self.pool(self.relu(self.conv1(x)))

x = self.pool(self.relu(self.conv2(x)))

x = x.view(-1, 64\*7\*7)

x = self.relu(self.fc1(x))

return self.fc2(x)

# Train function

def train\_once(model):

loader = torch.utils.data.DataLoader(trainset, batch\_size=64, shuffle=True, num\_workers=2)

model = model.to(device)

optimizer = optim.Adam(model.parameters(), lr=0.001)

criterion = nn.CrossEntropyLoss()

start = time.time()

for images, labels in loader:

images, labels = images.to(device), labels.to(device)

optimizer.zero\_grad()

loss = criterion(model(images), labels)

loss.backward()

optimizer.step()

break # just 1 mini-batch for demo

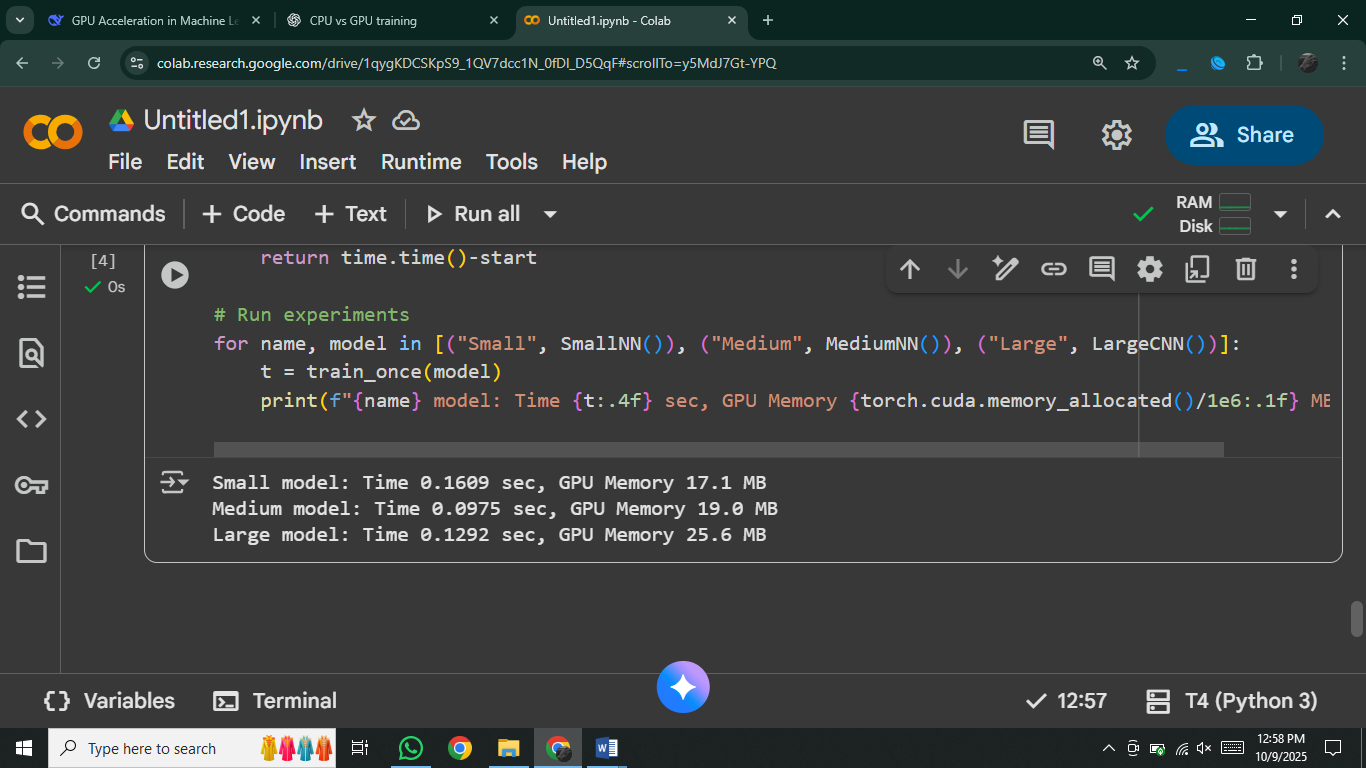
return time.time()-start

# Run experiments

for name, model in [("Small", SmallNN()), ("Medium", MediumNN()), ("Large", LargeCNN())]:

t = train\_once(model)

print(f"{name} model: Time {t:.4f} sec, GPU Memory {torch.cuda.memory\_allocated()/1e6:.1f} MB")



**How model size affects GPU workload and training time**

* Larger models (more layers/parameters) increase the workload because more matrix multiplications and gradient updates are required.
* On CPU, this makes training much slower. On GPU, the workload can be parallelized, so while training time increases, the GPU still handles it much more efficiently than a CPU.
* Very small models don’t fully use the GPU cores, so the speedup is minimal.

**How GPU compute and memory balance affect performance**

* If a model has high compute demand but fits well into GPU memory, performance is excellent because GPU cores stay fully utilized.
* If the model is too large for GPU memory, training slows down (due to memory swapping or smaller batch sizes).
* Best performance happens when compute demand and available GPU memory are balanced so that the GPU can process large enough batches without running out of memory.

**Part 4:**

import torch

import torchvision

import torchvision.transforms as transforms

import time

transform = transforms.Compose([

transforms.ToTensor(),

transforms.Normalize((0.5,), (0.5,))

])

train\_set = torchvision.datasets.MNIST(root="./data", train=True, download=True, transform=transform)

def measure\_loader\_speed(num\_workers, batches=200):

loader = torch.utils.data.DataLoader(

train\_set,

batch\_size=128,

shuffle=True,

num\_workers=num\_workers

)

start = time.time()

for idx, (images, labels) in enumerate(loader):

if idx >= batches: # only measure limited batches

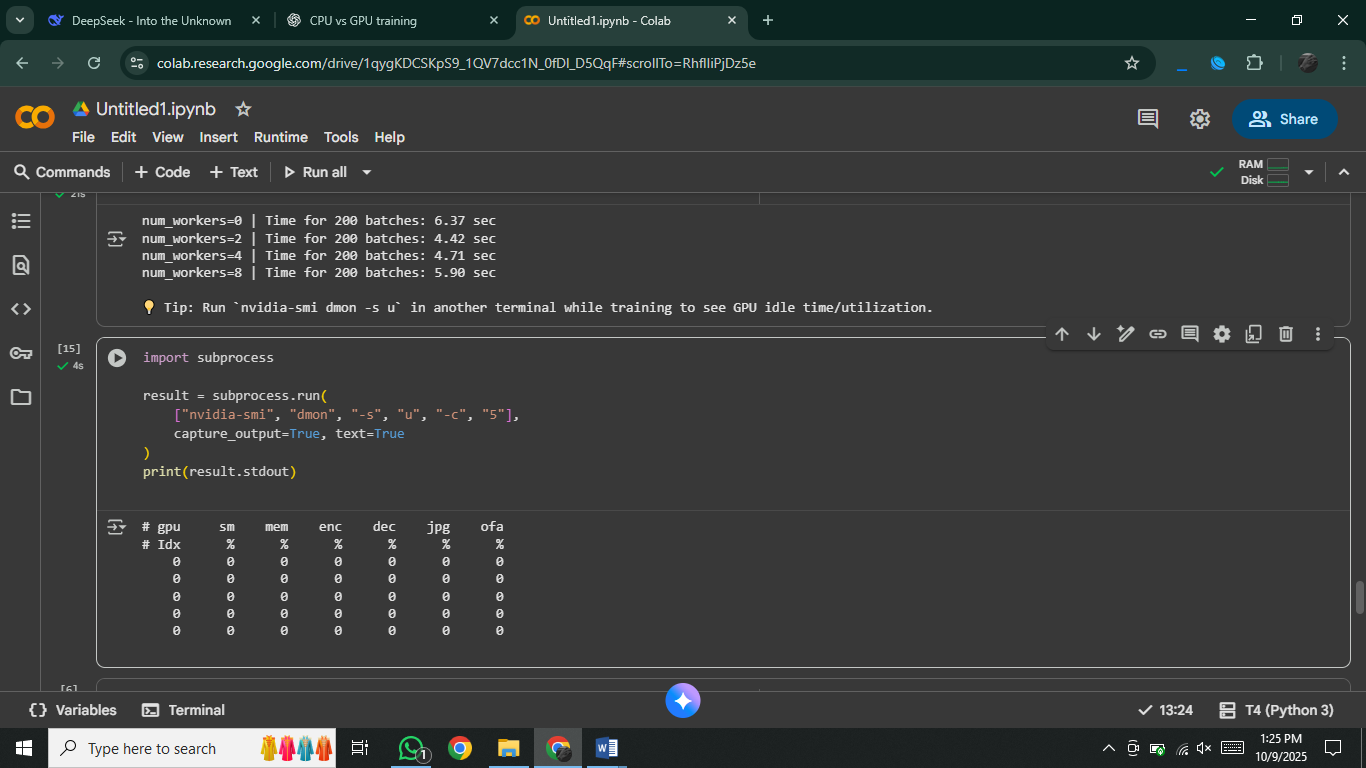
break

end = time.time()

print(f"num\_workers={num\_workers} | Time for {batches} batches: {end-start:.2f} sec")

for workers in [0, 2, 4, 8]:

measure\_loader\_speed(workers)

**Output:**

### **Why inefficient data pipelines cause low GPU utilization?**

* The **CPU is responsible for loading and preprocessing data** (from disk → RAM → GPU).
* If data loading is **too slow** (e.g., num\_workers=0, single-threaded), the **GPU finishes training on one batch and then waits idle** until the CPU provides the next batch.
* This creates a **bottleneck**: the GPU (which is much faster) is under-utilized, because the CPU/data pipeline cannot keep up.
* Result: **low GPU utilization** (you’ll often see GPU at 0–20% usage in nvidia-smi).

### **How overlapping CPU data loading and GPU training improves performance?**

* With **multi-threaded data loading** (num\_workers=2,4,8), the CPU prepares multiple batches in parallel.
* While the GPU is training on the current batch, the CPU is **already preparing the next batch** in the background.
* When the GPU is ready for more data, the next batch is instantly available → **no waiting time**.
* This overlapping ensures the GPU is **continuously busy** (higher utilization, closer to 80–100%).
* Result: **faster training, less idle time, better hardware efficiency**.

**Part 5:**

import torch

import torchvision

import torchvision.transforms as transforms

import torch.nn as nn

import torch.optim as optim

import time

transform = transforms.Compose([

transforms.ToTensor(),

transforms.Normalize((0.5,), (0.5,))

])

train\_set = torchvision.datasets.MNIST(root="./data", train=True, download=True, transform=transform)

train\_loader = torch.utils.data.DataLoader(train\_set, batch\_size=256, shuffle=True, num\_workers=4)

test\_set = torchvision.datasets.MNIST(root="./data", train=False, download=True, transform=transform)

test\_loader = torch.utils.data.DataLoader(test\_set, batch\_size=256, shuffle=False, num\_workers=4)

class SimpleNN(nn.Module):

def \_\_init\_\_(self):

super(SimpleNN, self).\_\_init\_\_()

self.fc1 = nn.Linear(28\*28, 256)

self.fc2 = nn.Linear(256, 128)

self.fc3 = nn.Linear(128, 10)

def forward(self, x):

x = x.view(-1, 28\*28)

x = torch.relu(self.fc1(x))

x = torch.relu(self.fc2(x))

return self.fc3(x)

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

print("Using device:", device)

def train\_model(use\_amp=False, epochs=1):

model = SimpleNN().to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

scaler = torch.cuda.amp.GradScaler(enabled=use\_amp) # AMP scaler

start = time.time()

for epoch in range(epochs):

model.train()

for images, labels in train\_loader:

images, labels = images.to(device), labels.to(device)

optimizer.zero\_grad()

with torch.cuda.amp.autocast(enabled=use\_amp):

outputs = model(images)

loss = criterion(outputs, labels)

scaler.scale(loss).backward()

scaler.step(optimizer)

scaler.update()

end = time.time()

# Evaluate accuracy

correct, total = 0, 0

model.eval()

with torch.no\_grad():

for images, labels in test\_loader:

images, labels = images.to(device), labels.to(device)

outputs = model(images)

\_, predicted = torch.max(outputs, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

accuracy = 100 \* correct / total

mem\_used = torch.cuda.max\_memory\_allocated(device) / (1024\*\*2) if torch.cuda.is\_available() else 0

print(f"{'AMP' if use\_amp else 'FP32'} | Time: {end-start:.2f}s | Accuracy: {accuracy:.2f}% | Max GPU Mem: {mem\_used:.2f} MB")

return end-start, accuracy, mem\_used

time\_fp32, acc\_fp32, mem\_fp32 = train\_model(use\_amp=False, epochs=1)

### time\_amp, acc\_amp, mem\_amp = train\_model(use\_amp=True, epochs=1)

### **How FP16 training makes training faster**

* FP16 numbers are **smaller than FP32**, so they take **less memory**.
* Because they are smaller, the GPU can do **more calculations in the same time**.
* This also lets us use **bigger batch sizes**, which makes training even faster.

Result: Training runs **faster** and uses **less GPU memory**.

### **When FP16 can cause problems**

* FP16 cannot store very **tiny numbers** → they may become **0** (called underflow).
* FP16 cannot store very **big numbers** → they may become **infinity (Inf)** (called overflow).
* Some models (like RNNs or models with very small losses) are more likely to face this issue.

Solution: PyTorch AMP automatically **mixes FP16 and FP32** to avoid most problems.

# **Discussion Questions & Answers**

**1. What factors most affect GPU training performance (batch size, model size, precision, data pipeline)?**

* **Batch size**: Larger batches improve parallelism but too large causes memory issues and accuracy drop.
* **Model size**: Bigger models use more GPU compute and memory, increasing training time.
* **Precision**: Mixed precision (FP16) speeds up training and saves memory but may cause instability.
* **Data pipeline**: If data loading is slow, GPU sits idle, reducing efficiency.

**2. Why might small models not benefit much from GPU acceleration?**

* Small models have few computations → CPU can handle them quickly.
* GPU overhead (data transfer, kernel launch) can dominate → no big speedup.

**3. How can you minimize GPU idle time during training?**

* Use num\_workers > 0 in DataLoader (parallel data loading).
* Prefetch and pin memory (pin\_memory=True).
* Use larger batch sizes to better utilize GPU.
* Overlap data loading with training.

**4. What are the trade-offs between higher batch size and model accuracy?**

* **Higher batch size** → faster per epoch, better GPU utilization.
* **Too large batch size** → fewer weight updates per epoch, risk of poor generalization, lower accuracy.
* Small batches add "gradient noise" which often improves generalization.

**5. Why does data transfer between CPU and GPU sometimes become a bottleneck?**

* Each batch must be copied from CPU RAM → GPU VRAM.
* If CPU loading/preprocessing is slow, GPU waits idle.
* Solutions: pin\_memory=True, larger batches, efficient preprocessing, using GPU-accelerated data pipelines.