

***LAB Assignment* 4**

|  |  |
| --- | --- |
| **Name :** | Noor Fatima |
| **Registration No:** | SP23-BCS-109 |
| **Course:** | Parallel and Distributed Computing |
| **Teacher:** | Sir Akhzar Nazir |
| **Date:** | 12-10-2025 |

**Part 1: CPU vs GPU Model Training:**

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import datasets, transforms

from torch.utils.data import DataLoader

import time

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

train\_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)

train\_loader = DataLoader(train\_dataset, batch\_size=64, shuffle=True)

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))

x = self.fc3(x)

return x

# Step 4: Training function

def train\_model(device):

model = SimpleNN().to(device)

criterion = nn.CrossEntropyLoss()

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

start\_time = time.time()

for epoch in range(3): # 3 epochs

epoch\_start = time.time()

for images, labels in train\_loader:

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

outputs = model(images)

loss = criterion(outputs, labels)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

print(f"Epoch {epoch+1} completed in {time.time() - epoch\_start:.2f} seconds")

total\_time = time.time() - start\_time

print(f"Total training time on {device}: {total\_time:.2f} seconds")

return total\_time

# Train on CPU

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

# CPU training

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

# GPU training

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

speedup = cpu\_time / gpu\_time

print(f"\nSpeedup = CPU Time / GPU Time = {speedup:.2f}x")

**Part 2: Effect of Batch Size:**

|  |
| --- |
| import torch  import torch.nn as nn  import torch.optim as optim  from torchvision import datasets, transforms  from torch.utils.data import DataLoader  import time  import matplotlib.pyplot as plt  transform = transforms.Compose([transforms.ToTensor()])  train\_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)  test\_dataset = datasets.MNIST(root='./data', train=False, transform=transform)  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))  x = self.fc3(x)  return x  def train\_model(batch\_size, device):  train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)  model = SimpleNN().to(device)  criterion = nn.CrossEntropyLoss()  optimizer = optim.Adam(model.parameters(), lr=0.001)  start\_time = time.time()  for epoch in range(2): # only 2 epochs to compare faster  running\_loss = 0.0  correct, total = 0, 0  for images, labels in train\_loader:  images, labels = images.to(device), labels.to(device)  outputs = model(images)  loss = criterion(outputs, labels)  optimizer.zero\_grad()  loss.backward()  optimizer.step()  running\_loss += loss.item()  \_, predicted = torch.max(outputs.data, 1)  total += labels.size(0)  correct += (predicted == labels).sum().item()  acc = 100 \* correct / total  print(f"Batch {batch\_size}, Epoch {epoch+1}, Loss: {running\_loss/len(train\_loader):.4f}, Accuracy: {acc:.2f}%")  total\_time = time.time() - start\_time  gpu\_memory = torch.cuda.memory\_allocated(device) / 1024\*\*2 # MB  return total\_time, gpu\_memory, acc  device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  batch\_sizes = [16, 64, 256, 1024]  times, memories, accuracies = [], [], []  for b in batch\_sizes:  print(f"\nTraining with batch size = {b}")  t, m, a = train\_model(b, device)  times.append(t)  memories.append(m)  accuracies.append(a)  **Plot Results:**  plt.figure(figsize=(12,5))  # Plot 1: Batch Size vs Training Time  plt.subplot(1,2,1)  plt.plot(batch\_sizes, times, marker='o')  plt.title("Batch Size vs Training Time")  plt.xlabel("Batch Size")  plt.ylabel("Time (seconds)")  # Plot 2: Batch Size vs GPU Memory  plt.subplot(1,2,2)  plt.plot(batch\_sizes, memories, marker='o', color='orange')  plt.title("Batch Size vs GPU Memory Usage")  plt.xlabel("Batch Size")  plt.ylabel("Memory (MB)")  plt.show()  **Why increasing batch size improves GPU efficiency up to a point:**  Larger batches allow the GPU to process more data in parallel, maximizing CUDA core utilization.  It reduces the overhead of transferring data between CPU and GPU.  Fewer weight updates per epoch → less synchronization overhead.  However:  After a certain batch size, you hit GPU memory limits.  The diminishing returns occur because memory bandwidth and compute throughput saturate.  Very large batches can slow down convergence (less frequent updates).  **🔹 Why accuracy sometimes drops for very large batches:**  Large batches produce smoother gradients, which can lead to:  Poor generalization.  Getting stuck in sharp minima (less robust).  Small batches add stochastic noise to updates, which helps escape local minima and improves generalization. |

**Part 3: Model Complexity and GPU Utilization:**

|  |
| --- |
| **Step 1: Setup**  import torch  import torch.nn as nn  import torch.optim as optim  from torchvision import datasets, transforms  from torch.utils.data import DataLoader  import time  import matplotlib.pyplot as plt   * **Dataset**   transform = transforms.Compose([transforms.ToTensor()])  train\_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)  train\_loader = DataLoader(train\_dataset, batch\_size=64, shuffle=True)   * **Small Model (1–2 Layers)**   class SmallNN(nn.Module):  def \_\_init\_\_(self):  super(SmallNN, self).\_\_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))  x = self.fc2(x)  return x   * **Medium Model (3–5 Layers)**   class MediumNN(nn.Module):  def \_\_init\_\_(self):  super(MediumNN, self).\_\_init\_\_()  self.fc1 = nn.Linear(28\*28, 512)  self.fc2 = nn.Linear(512, 256)  self.fc3 = nn.Linear(256, 128)  self.fc4 = 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))  x = torch.relu(self.fc3(x))  x = self.fc4(x)  return x   * **Large Model (CNN):**   class LargeCNN(nn.Module):  def \_\_init\_\_(self):  super(LargeCNN, self).\_\_init\_\_()  self.conv1 = nn.Conv2d(1, 32, kernel\_size=3, padding=1)  self.conv2 = nn.Conv2d(32, 64, kernel\_size=3, padding=1)  self.conv3 = nn.Conv2d(64, 128, kernel\_size=3, padding=1)  self.pool = nn.MaxPool2d(2, 2)  self.fc1 = nn.Linear(128\*3\*3, 256)  self.fc2 = nn.Linear(256, 10)  def forward(self, x):  x = torch.relu(self.conv1(x))  x = self.pool(torch.relu(self.conv2(x)))  x = self.pool(torch.relu(self.conv3(x)))  x = x.view(-1, 128\*3\*3)  x = torch.relu(self.fc1(x))  x = self.fc2(x)  return x   * **Training Function:**   def train\_model(model, device):  model = model.to(device)  criterion = nn.CrossEntropyLoss()  optimizer = optim.Adam(model.parameters(), lr=0.001)  start\_time = time.time()  torch.cuda.reset\_peak\_memory\_stats(device)  for epoch in range(2):  running\_loss, correct, total = 0.0, 0, 0  for images, labels in train\_loader:  images, labels = images.to(device), labels.to(device)  outputs = model(images)  loss = criterion(outputs, labels)  optimizer.zero\_grad()  loss.backward()  optimizer.step()  running\_loss += loss.item()  \_, predicted = torch.max(outputs.data, 1)  total += labels.size(0)  correct += (predicted == labels).sum().item()  acc = 100 \* correct / total  print(f"Epoch {epoch+1}: Loss={running\_loss/len(train\_loader):.4f}, Accuracy={acc:.2f}%")  total\_time = time.time() - start\_time  peak\_mem = torch.cuda.max\_memory\_allocated(device) / 1024\*\*2 # in MB  return total\_time, peak\_mem, acc   * **Run All Models on GPU**   device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  models = {  "Small": SmallNN(),  "Medium": MediumNN(),  "Large (CNN)": LargeCNN()  }  times, memories, accuracies = [], [], []  for name, model in models.items():  print(f"\n🔹 Training {name} Model")  t, m, a = train\_model(model, device)  times.append(t)  memories.append(m)  accuracies.append(a)   * **Plot the Results:**   plt.figure(figsize=(12,5))  # Time per model  plt.subplot(1,2,1)  plt.bar(models.keys(), times, color=['green','orange','red'])  plt.title("Model Complexity vs Training Time")  plt.ylabel("Time (seconds)")  plt.grid(True, axis='y')  # GPU Memory Usage  plt.subplot(1,2,2)  plt.bar(models.keys(), memories, color=['green','orange','red'])  plt.title("Model Complexity vs Peak GPU Memory Usage")  plt.ylabel("Memory (MB)")  plt.grid(True, axis='y')  plt.show()  **How model size affects GPU workload and training time:**  Small models use fewer parameters and simpler operations — GPU is underutilized, so training is fast but doesn’t fully use GPU compute power.  Medium models increase parallel computation, leading to better GPU utilization and moderate training time.  Large CNNs perform heavy convolution and matrix multiplications — GPU utilization peaks, memory usage is high, and training takes longer per epoch.  **How GPU compute and memory balance affect performance:**  GPUs perform best when both compute cores and memory bandwidth are used efficiently.  Small models → low GPU load (compute underused).  Very large models → may exceed memory capacity or cause memory bottlenecks.  Balanced models (medium complexity) often give the best speed–efficiency trade-off. |

**Part 4: Data Loading and Bottlenecks:**

|  |
| --- |
| import torch  import torch.nn as nn  import torch.optim as optim  from torchvision import datasets, transforms  from torch.utils.data import DataLoader  import time  import matplotlib.pyplot as plt  transform = transforms.Compose([transforms.ToTensor()])  train\_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)  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))  x = self.fc3(x)  return x  def train\_with\_num\_workers(num\_workers, device):  train\_loader = DataLoader(train\_dataset, batch\_size=128, shuffle=True, num\_workers=num\_workers)  model = SimpleNN().to(device)  criterion = nn.CrossEntropyLoss()  optimizer = optim.Adam(model.parameters(), lr=0.001)  total\_load\_time = 0  total\_train\_time = 0  start\_total = time.time()  for epoch in range(1):  for images, labels in train\_loader:  start\_load = time.time()  images, labels = images.to(device), labels.to(device)  end\_load = time.time()  total\_load\_time += (end\_load - start\_load)  outputs = model(images)  loss = criterion(outputs, labels)  optimizer.zero\_grad()  loss.backward()  optimizer.step()  total\_train\_time = time.time() - start\_total  avg\_load\_time = total\_load\_time / len(train\_loader)  print(f"num\_workers={num\_workers} → Total time: {total\_train\_time:.2f}s, Avg load time/batch: {avg\_load\_time:.4f}s")  return total\_train\_time, avg\_load\_time  device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  worker\_settings = [0, 2, 4, 8]  train\_times, load\_times = [], []  for n in worker\_settings:  print(f"\nTraining with num\_workers={n}")  total\_t, load\_t = train\_with\_num\_workers(n, device)  train\_times.append(total\_t)  load\_times.append(load\_t)  plt.figure(figsize=(12,5))  # Training Time  plt.subplot(1,2,1)  plt.plot(worker\_settings, train\_times, marker='o')  plt.title("num\_workers vs Total Training Time")  plt.xlabel("num\_workers")  plt.ylabel("Training Time (seconds)")  plt.grid(True)  # Data Loading Time  plt.subplot(1,2,2)  plt.plot(worker\_settings, load\_times, marker='o', color='orange')  plt.title("num\_workers vs Avg Data Loading Time per Batch")  plt.xlabel("num\_workers")  plt.ylabel("Data Loading Time (seconds)")  plt.grid(True)  plt.show() |

**Part 5: Mixed Precision Training (Optional, Bonus):**

|  |
| --- |
| import torch  import torch.nn as nn  import torch.optim as optim  from torchvision import datasets, transforms  from torch.utils.data import DataLoader  import time  import matplotlib.pyplot as plt  transform = transforms.Compose([transforms.ToTensor()])  train\_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)  test\_dataset = datasets.MNIST(root='./data', train=False, transform=transform)  train\_loader = DataLoader(train\_dataset, batch\_size=128, shuffle=True)  test\_loader = DataLoader(test\_dataset, batch\_size=128, shuffle=False)  class SimpleNN(nn.Module):  def \_\_init\_\_(self):  super(SimpleNN, self).\_\_init\_\_()  self.fc1 = nn.Linear(28\*28, 512)  self.fc2 = nn.Linear(512, 256)  self.fc3 = nn.Linear(256, 10)  def forward(self, x):  x = x.view(-1, 28\*28)  x = torch.relu(self.fc1(x))  x = torch.relu(self.fc2(x))  x = self.fc3(x)  return x  def train\_model(use\_amp=False):  device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  model = SimpleNN().to(device)  criterion = nn.CrossEntropyLoss()  optimizer = optim.Adam(model.parameters(), lr=0.001)  scaler = torch.cuda.amp.GradScaler(enabled=use\_amp)  torch.cuda.reset\_peak\_memory\_stats(device)  start\_time = time.time()  for epoch in range(2):  running\_loss, correct, total = 0.0, 0, 0  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()  running\_loss += loss.item()  \_, predicted = torch.max(outputs.data, 1)  total += labels.size(0)  correct += (predicted == labels).sum().item()  acc = 100 \* correct / total  print(f"{'AMP' if use\_amp else 'FP32'} - Epoch {epoch+1}, Loss={running\_loss/len(train\_loader):.4f}, Acc={acc:.2f}%")  total\_time = time.time() - start\_time  peak\_mem = torch.cuda.max\_memory\_allocated(device) / 1024\*\*2 # MB  return total\_time, acc, peak\_mem  print("\nTraining WITHOUT AMP (FP32)")  fp32\_time, fp32\_acc, fp32\_mem = train\_model(use\_amp=False)  print("\nTraining WITH AMP (Mixed Precision)")  amp\_time, amp\_acc, amp\_mem = train\_model(use\_amp=True)  import pandas as pd  results = pd.DataFrame({  "Mode": ["FP32 (No AMP)", "AMP (Mixed Precision)"],  "Time per 2 Epochs (s)": [fp32\_time, amp\_time],  "Accuracy (%)": [fp32\_acc, amp\_acc],  "Peak GPU Memory (MB)": [fp32\_mem, amp\_mem]  })  print(results)  plt.figure(figsize=(8,4))  plt.bar(["FP32", "AMP"], [fp32\_time, amp\_time], color=["gray", "orange"])  plt.title("Training Speed Comparison")  plt.ylabel("Total Time (s)")  plt.grid(True, axis='y')  plt.show() |

**Discussion Questions:**

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

GPU training performance depends on how efficiently the GPU is utilized. The key factors include:

* Batch Size:

Larger batches allow the GPU to process more data in parallel, improving utilization and throughput. However, extremely large batches may exceed memory limits or degrade accuracy.

* Model Size:

Bigger models (more layers/parameters) perform more matrix operations per step, keeping the GPU busier and better utilized. Small models underuse the GPU’s computational power.

* Precision (FP32 vs FP16):

Using Mixed Precision (FP16) reduces memory usage and speeds up computation by performing operations in lower precision while maintaining numerical stability.

* Data Pipeline (num\_workers, prefetching):

If data loading is slow (single-threaded or poorly optimized), the GPU will frequently sit idle waiting for new batches — decreasing performance.

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

* Small models have low computational demand — their operations can often be completed faster on the CPU without the overhead of transferring data to the GPU.
* GPU parallelism only helps when there’s enough work to distribute across thousands of GPU cores.
* Data transfer and kernel launch overheads can dominate when the model is tiny, leading to little or no speedup.

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

To ensure the GPU stays busy:

* Increase num\_workers in DataLoader to load data in parallel.
* Use pinned memory (pin\_memory=True) for faster CPU–GPU data transfer.
* Prefetch data asynchronously to overlap CPU data loading with GPU computation.
* Use larger batches if GPU memory allows, to reduce per-batch overhead.
* Optimize preprocessing (e.g., resize or augment images using GPU-based libraries like DALI).

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

|  |  |  |
| --- | --- | --- |
| **Batch Size** | **Pros** | **Cons** |
| Small | Better generalization and more gradient noise (helps escape local minima) | Slower per-epoch training, less efficient GPU use |
| Large | Faster training, stable gradient estimates | Risk of poorer generalization, needs careful learning rate tuning |

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

* The GPU must receive input data from the CPU’s main memory via the PCIe bus — which has limited bandwidth.
* If each batch is small or data is sent frequently, transfer latency becomes significant.
* Slow data augmentation or lack of pinned memory worsens the delay, causing the GPU to wait idly for input.
* Solution: Use pinned memory, async data loading, and keep as much data as possible on the GPU.