

# A Project Report on

Parallel Computing for 94-Class Character Recognition Using CNN

Team 11: Viswanath Raju Indukuri and Sai Vivekanand Reddy Vangala

**College of Engineering, Northeastern University**

## **CSYE7105 Parallel Machine Learning & AI**

Professor. Handan Liu

April 13, 2025

**Table of Contents:**

[**1. Introduction 4**](#_Toc196413372)

[**1.1 Background: 4**](#_Toc196413373)

[**1.2 Motivation: 4**](#_Toc196413374)

[**1.3 Goal: 4**](#_Toc196413375)

[**2. Methodology 4**](#_Toc196413376)

[**2.1 Parallel Data Processing 4**](#_Toc196413377)

[**2.2 Scope for Parallelization: 5**](#_Toc196413378)

[**2.3 Model Building 5**](#_Toc196413379)

[**2.4 Performance Evaluation 6**](#_Toc196413380)

[**2.5 Analysis and Visualization 6**](#_Toc196413381)

[**2.6 Optimization and Iteration 6**](#_Toc196413382)

[**3. Dataset Description 7**](#_Toc196413383)

[**3.1 Overview & Features 7**](#_Toc196413384)

[**4. Results and Analysis 7**](#_Toc196413385)

[**4.1 Configurations and Environment: 7**](#_Toc196413386)

[**4.2 Parallel Performance on GPUs 7**](#_Toc196413387)

[**4.2.1 Distributed Data Parallel (DDP) Training Analysis: 7**](#_Toc196413388)

[**4.2.2 Automic Mixed Precision (AMP) Training Analysis: 17**](#_Toc196413389)

[**4.2.3 Comparing DDP vs Mixed Precision (AMP) Training Analysis: 24**](#_Toc196413390)

[**4.2.4 Memory Usage Comparison Between DDP and AMP 26**](#_Toc196413391)

[**4.2.5 Training Throughput Analysis: DDP vs AMP 27**](#_Toc196413392)

[**4.3 Parallel Performance on CPUs: 28**](#_Toc196413393)

[**4.3.1 Joblib on Multi-CPUs (with different batch sizes): 28**](#_Toc196413394)

[**4.3.2 Failures/Limitations faced in CPU Parallelization: 31**](#_Toc196413395)

[**4.3.3 Comparison with Joblib Approach 32**](#_Toc196413396)

[**4.3.2 CPU Scaling Limitations Analysis 32**](#_Toc196413397)

[**5. Conclusion 33**](#_Toc196413398)

[**5.1 Multi-GPU Parallelization with DDP (Standard Precision) 33**](#_Toc196413399)

[**5.2 Mixed Precision GPU Training (AMP) 33**](#_Toc196413400)

[**5.3 Multi-CPU Parallelization with Joblib 33**](#_Toc196413401)

[**5.4 Comparative Analysis and Implementation Recommendations 34**](#_Toc196413402)

[**5.5 Technical Implications 34**](#_Toc196413403)

[**6. References 35**](#_Toc196413404)

# 1. Introduction

### 1.1 Background:

Character recognition is a fundamental task in computer vision, with applications ranging from optical character recognition (OCR) systems to digitizing historical documents. While many existing solutions focus on relatively small sets of characters (e.g., digits or basic alphabets), our project targets 94 different characters, an expanded scope that often introduces additional complexity in both data processing and model training.

### 1.2 Motivation:

One challenge in deep learning pipelines is the speed and efficiency of data processing and model training especially for large datasets and computationally intensive tasks like image transformations and convolutional neural network (CNN) training. By leveraging parallel processing, we can substantially reduce both data-preprocessing time and model training time.

### 1.3 Goal:

The main goal of this project is twofold:

1. **Parallel Data Processing**: Implement parallel data transformations (e.g., image resizing, normalization, or format conversions) using techniques like starmap to reduce preprocessing overhead, then evaluate and visualize performance gains (speedup, timing, etc.)
2. **Parallel CNN Training**: Develop a CNN for 94-character image classification in PyTorch and enable multi-core training. We will compare training time and accuracy between a change in number of multi-CPUs and multi-GPUs setup and illustrate the performance gains using various metrics (e.g., training time per epoch, efficiency, speedup ratios).

# 2. Methodology

### 2.1 Parallel Data Processing

1. **Dataset Loading and Splitting:** We will load the dataset and split it into training, validation, and test subsets. Python libraries such as multiprocessing or built-in parallelism mechanisms in data loaders will be used to parallelize the reading of image files from disk.
2. **Data Cleaning and Transformation:**
   * Typical transformations (e.g., resizing, normalization, and data augmentation techniques such as random cropping or flipping) will be performed in parallel.
   * We will measure the processing time for these augmentations in serial mode versus multiple workers to quantify the speedup.
3. **Data Parallelization and Performance Evaluation:** Various data parallelizing techniques were explored, including PyTorch DDP, Mixed Precision, Joblib and DataLoader etc. Our Training Model is small so we have excluded FSDP. Results will be visualized in graphs that show how performance scales with the number of CPUs and GPUs.

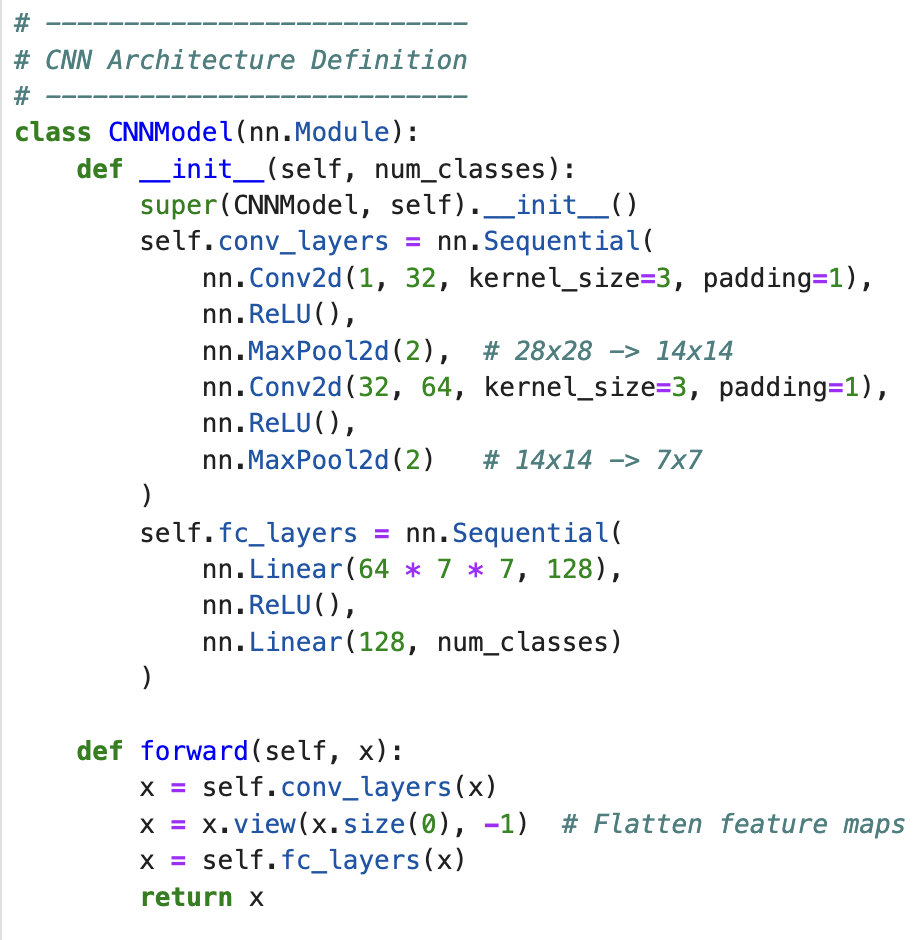
### 2.2 Scope for Parallelization:

The TMNIST dataset comprises approximately 274,000 character images representing 94 unique character classes, each with a resolution of 28×28 pixels in grayscale. The dataset includes various font styles and character forms, providing a comprehensive benchmark for character recognition models. Given the dataset's size and the computational intensity of training deep learning models, parallel processing techniques are essential for efficient execution. The project leverages both multi-CPU parallelization via Joblib and Python's multiprocessing, as well as multi-GPU distributed training through PyTorch's DistributedDataParallel (DDP) framework to accelerate model training and inference processes.

### 2.3 Model Building

The character recognition model utilizes a Convolutional Neural Network (CNN) architecture specifically designed for the TMNIST dataset. The model consists of two convolutional layers with max pooling operations to extract spatial features, followed by fully connected layers for classification. The first convolutional layer applies 32 filters with a 3×3 kernel, and the second layer applies 64 filters with the same kernel size. ReLU activation functions are used throughout the network to introduce non-linearity.

Two parallel training implementations were developed: (1) a data-parallel approach using Python's multiprocessing and Joblib for CPU parallelization, where the dataset is partitioned across multiple workers; and (2) a distributed data-parallel approach using PyTorch's DDP framework for GPU parallelization, which enables training across multiple GPUs with automatic gradient synchronization. The second implementation also incorporates automatic mixed precision (AMP) training to further enhance computational efficiency while maintaining model accuracy.



### 2.4 Performance Evaluation

The execution time and computational efficiency were measured across different parallel configurations. For CPU parallelization, experiments were conducted with 1, 2, 4, and 6 CPU cores, while GPU parallelization was tested on configurations ranging from 1 to the maximum number of available GPUs. Various batch sizes (64, 128, 256, 512, 1024, 2048) were evaluated to identify optimal throughput settings. Performance metrics including training time, speedup, efficiency, GPU utilization, memory usage, and model accuracy were recorded systematically for comprehensive analysis.

### 2.5 Analysis and Visualization

Training performance across different parallelization configurations was analyzed to evaluate scalability and efficiency. The relationship between batch size, processing units, and training time was visualized through comparative graphs. For multi-CPU implementations, we observed how increasing the number of workers affected speedup and efficiency, particularly noting the impact of Python's Global Interpreter Lock (GIL) on threading performance. For multi-GPU implementations, we analyzed how different batch sizes influence GPU utilization, memory consumption, and overall training throughput, with attention to communication overhead when scaling to multiple devices.

### 2.6 Optimization and Iteration

For the CPU-based solution, we fine-tuned worker allocation and data partitioning strategies to balance load distribution. In the GPU implementation, we optimized the distributed training process by incorporating mixed precision training, which leverages both FP16 and FP32 data types to reduce memory usage and increase computational throughput. Additionally, we explored the impact of different batch sizes on scaling efficiency to determine optimal configurations for various hardware setups.

# 3. Dataset Description

## **3.1 Overview & Features**

The dataset ([Dataset link](https://www.kaggle.com/datasets/nikbearbrown/tmnist-alphabet-94-characters)) is CSV formatted file with a size of 940.82 MB, it has **274,093 rows** and **786 columns**:

1. **Rows (274,093)**: Each row represents a single character sample.
2. **Columns (786)**:
   * **1 column (names)**: The font name of the character.
   * **1 column (labels)**: The actual character label (e.g., letters, digits, symbols).
   * **784 columns (1-784)**: These represent pixel intensity values for the **28x28** grayscale image.

# 4. Results and Analysis

## **4.1 Configurations and Environment:**

The experiments were conducted using the following hardware and software configurations:

**Hardware:**

* CPUs: Multi-core system with up to 8 CPU cores for Joblib experiments
* GPUs: Tesla P100-PCIE-12GB (for DDP experiments) and NVIDIA H100 80GB HBM3 (for AMP experiments)
* GPU Memory: 12,194 MB per P100 GPU and 81,004 MB per H100 GPU

**Software:**

* PyTorch for deep learning and distributed training
* Python's multiprocessing and Joblib libraries for CPU parallelization
* Dataset: TMNIST with 94 character classes (274,093 samples resized to 28×28 pixels)
* Model: Convolutional Neural Network with 2 convolutional layers and fully connected layers

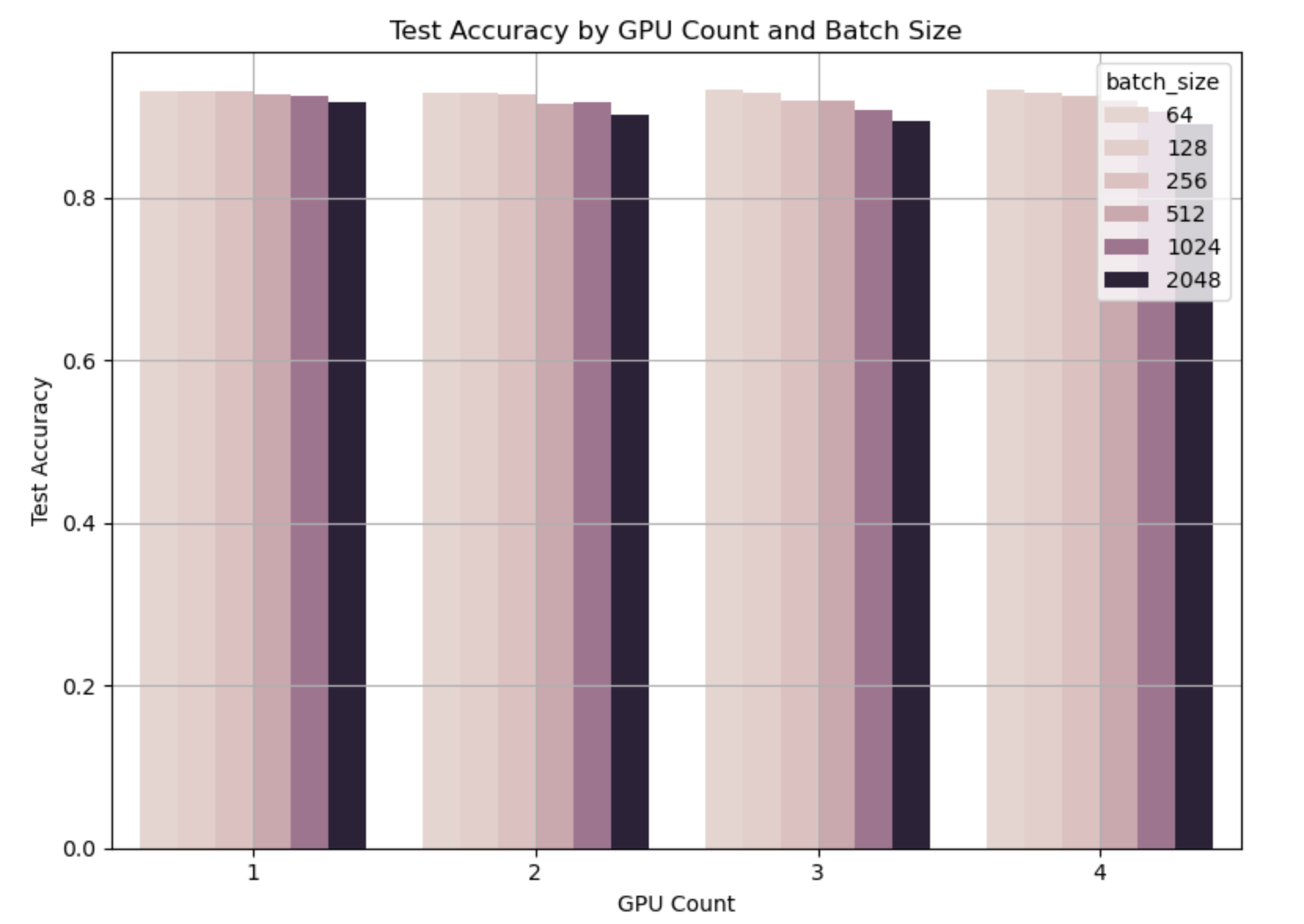
## **4.2 Parallel Performance on GPUs**

### 4.2.1 Distributed Data Parallel (DDP) Training Analysis:

Analysis contains 288 rows representing different training configurations and epochs, with 13 columns of metrics (DDP\_Traning\_analysis.ipynb):

* gpu\_name: The model of GPU used (Tesla P100-PCIE-12GB)
* each\_gpu\_memory: Memory capacity of each GPU (12194MB)
* gpu\_count: Number of GPUs used (ranges from 1 to 4)
* batch\_size: Size of training batches (64 for single GPU, increasing to 2048 with 4 GPUs)
* epoch: Training epoch number
* train\_loss/val\_loss: Loss metrics for training and validation
* train\_acc/val\_acc: Accuracy metrics for training and validation
* epoch\_train\_time: Time taken to complete each epoch
* train\_throughput: Training examples processed per second
* gpu\_util/mem\_util: GPU and memory utilization percentages

**Test Accuracy Across GPU and Batch Size Configuration**



The plot reveals several key patterns:

* **Performance by Batch Size**: Smaller batch sizes (lighter colors) consistently achieve higher test accuracy across all GPU configurations
* **Scaling Patterns**: As we increase GPU count, the performance difference between small and large batch sizes becomes more pronounced
* **Diminishing Returns**: Larger batch sizes (1024, 2048) show significant accuracy degradation, especially with 3-4 GPUs

**Best Configuration Results:**

The optimal setup uses 4 GPUs with a batch size of 64, achieving:

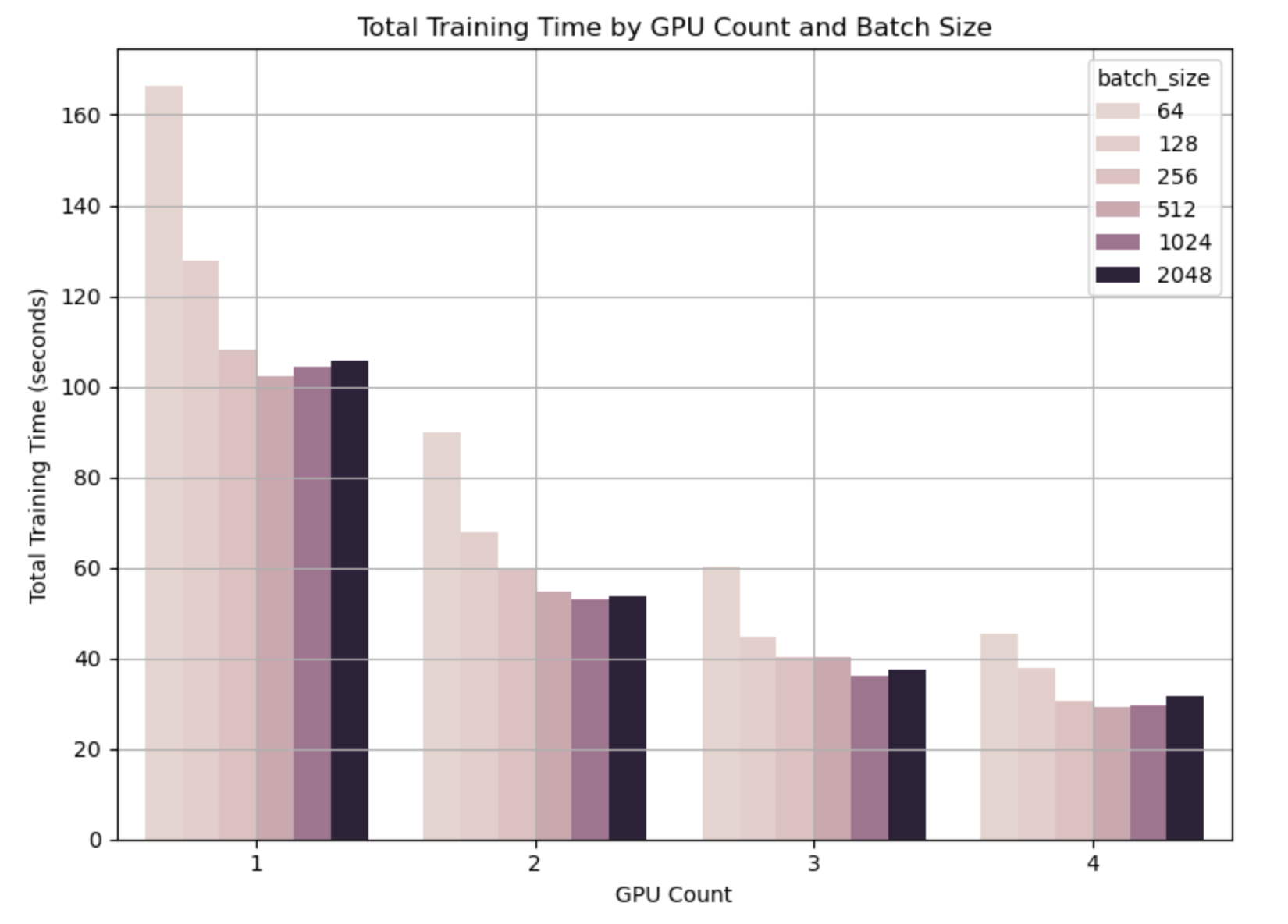
* Test accuracy of 93.25%
* Test loss of 0.2257
* Total training time of just 45.43 seconds

This demonstrates that we can achieve both high accuracy and fast training by parallelizing across multiple GPUs while keeping the batch size small, rather than increasing both GPU count and batch size together.

**Analyzing Training Time Scaling with Multiple GPUs**



1. **Scaling Efficiency**:
   * All configurations show significant speedups as we increase GPU count
   * With batch size 64, training time decreases from ~166 seconds (1 GPU) to ~45 seconds (4 GPUs)
   * This represents a ~3.7x speedup using 4x the hardware resources, indicating good but not perfect scaling
2. **Diminishing Returns**:
   * The most dramatic time reduction occurs when going from 1 to 2 GPUs
   * Adding the 3rd and 4th GPU provides progressively smaller improvements
   * This is typical in distributed systems due to communication overhead
3. **Batch Size Impact**:
   * Smaller batch sizes (64, 128) show steeper curves and greater absolute time reductions
   * Larger batch sizes (1024, 2048) have flatter curves, meaning they benefit proportionally less from additional GPUs
   * The curves converge as GPU count increases, suggesting diminishing differences between batch sizes at higher parallelism
4. **Ideal Configuration Selection**:
   * For time-critical applications: 4 GPUs with larger batch sizes provide minimum training time
   * For accuracy-critical applications: 4 GPUs with batch size 64 balances speed with maintaining high accuracy (as seen in previous plots)

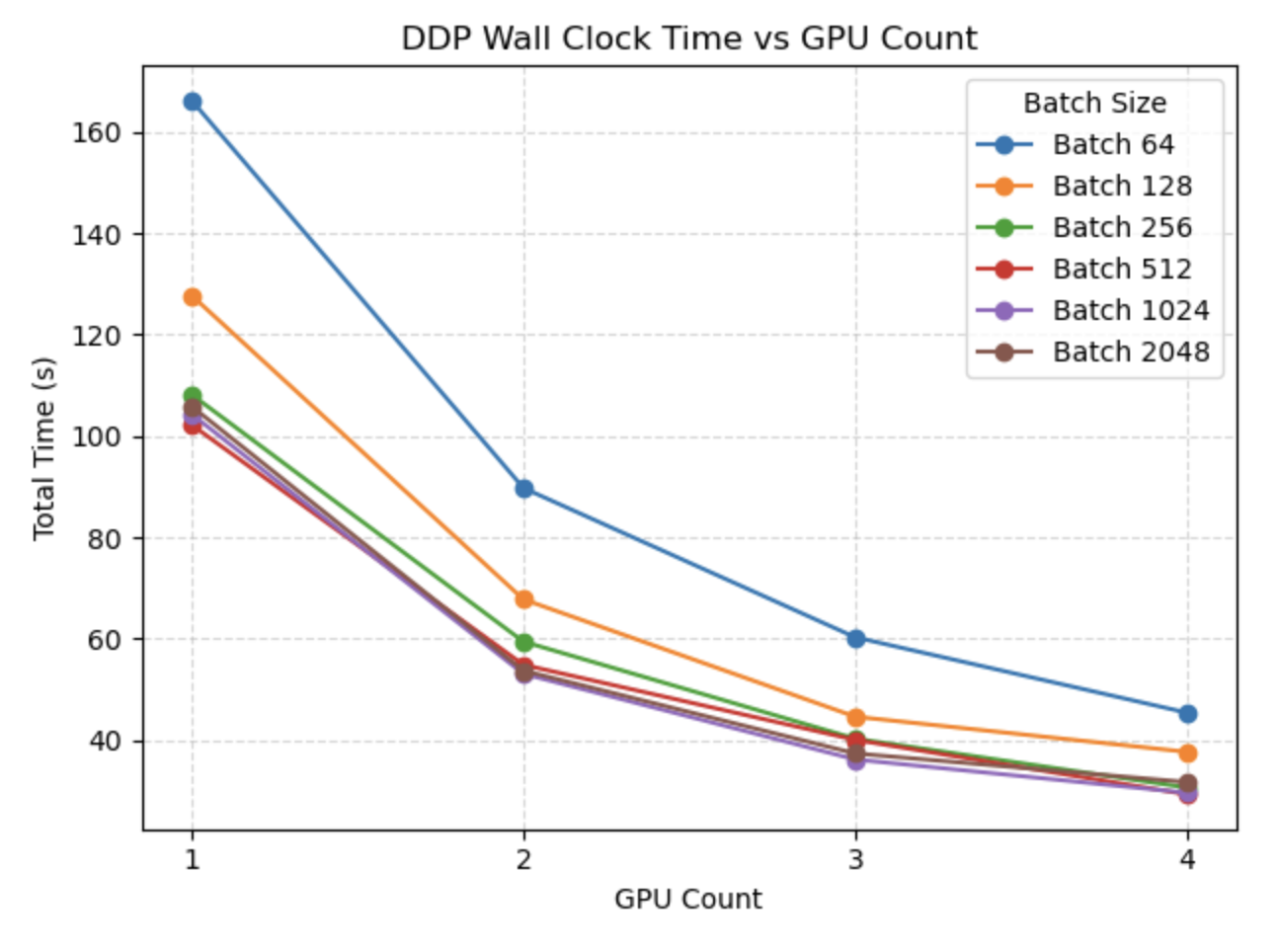


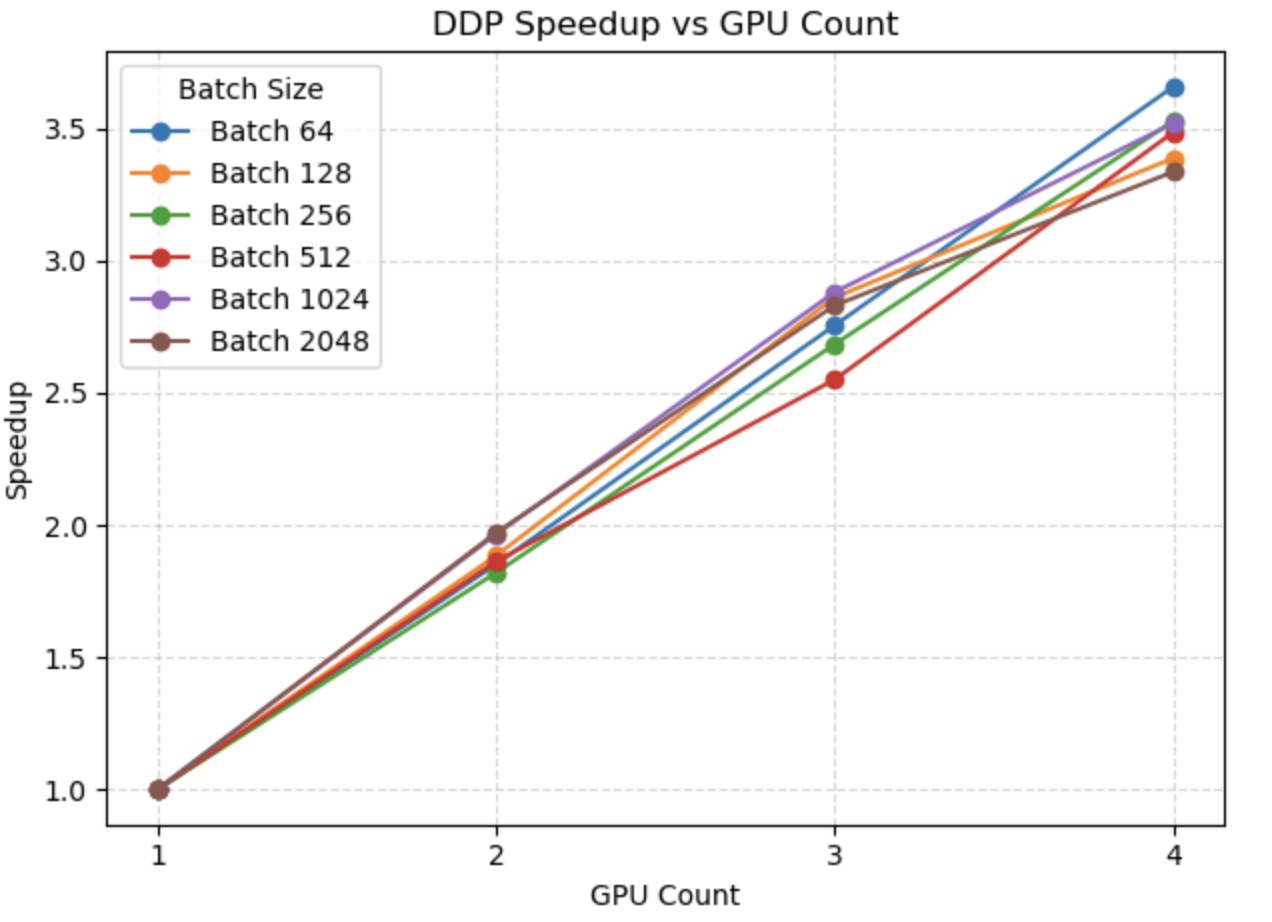
The bar chart reveals several important patterns about distributed training efficiency:

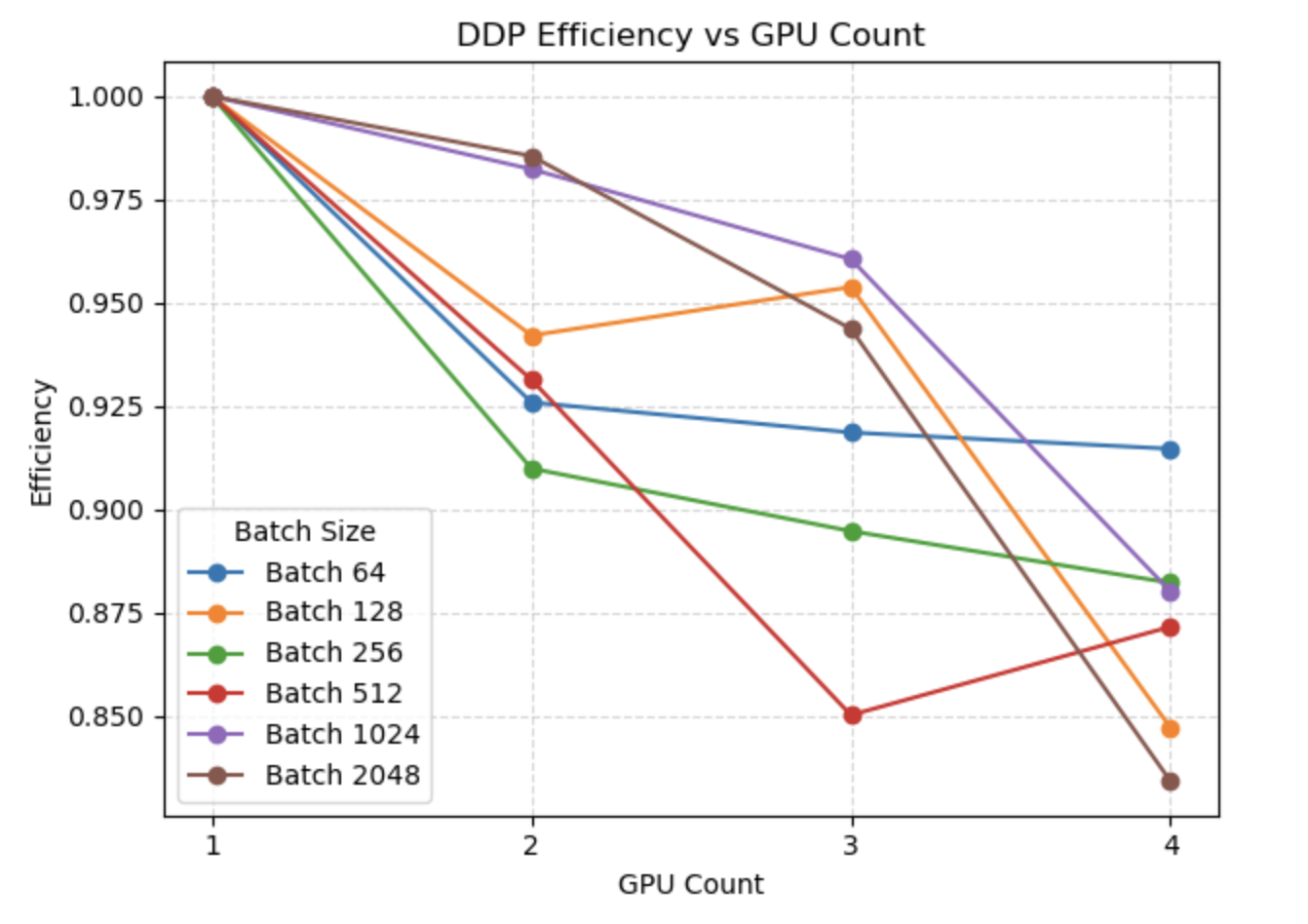
1. **Scaling Pattern**: Adding more GPUs drastically reduces training time across all batch sizes. The most dramatic reduction occurs when moving from 1 to 2 GPUs.
2. **Batch Size Impact**: With 1 GPU, smaller batch sizes (64, 128) have significantly longer training times than larger ones. This is because processing many small batches incurs more overhead.
3. **Convergence Trend**: As GPU count increases, the training time difference between batch sizes diminishes. By 4 GPUs, all configurations achieve similar training times (30-45 seconds).
4. **Optimization Decision Point**: The visualization makes clear that the combination of 4 GPUs with batch size 64 provides the best balance between speed and accuracy (as we saw in previous visualizations).

The fastest training configuration is:

* **GPU Count: 4, Batch Size: 512** - Using maximum parallelism with a moderate batch size
* **Total Training Time: 29.32 seconds** - Approximately 5.7× faster than the single-GPU baseline (166.22s)
* **Test Accuracy: 91.91%** - Still maintains good model performance
* **Test Loss: 0.2793** - Slightly higher than some configurations but acceptable







These three visualizations and the accompanying tabular data provide a comprehensive analysis of how PyTorch's Distributed Data Parallel (DDP) training scales across multiple GPUs with various batch sizes. Together, they illustrate the critical trade-offs in distributed deep learning.

**Wall Clock Time Analysis**

The first plot shows the total training time decreasing as more GPUs are added:

* **Small Batch Sizes (64, 128)**: Show the most dramatic absolute time reductions, with batch size 64 dropping from 166s to 45s with 4 GPUs
* **Larger Batch Sizes**: Start with lower training times on a single GPU but still benefit from parallelization
* **Convergence Pattern**: All batch sizes trend toward similar training times as GPU count increases
* **Diminishing Returns**: The steepest time reductions occur when going from 1→2 GPUs, with smaller gains for 3→4 GPUs

**Speedup Analysis**

The second plot quantifies the acceleration factor relative to single-GPU training:

* **Near-Linear Scaling**: Most configurations achieve 3.3-3.7× speedup with 4 GPUs (82-92% of theoretical 4× speedup)
* **Best Scaling**: Batch 64 achieves the highest speedup (3.66×) with 4 GPUs
* **Consistent Trends**: All batch sizes follow similar speedup curves with minimal differences
* **Observation**: Speedup continues to increase with more GPUs, though at a decreasing rate

**Efficiency Analysis**

The third plot reveals the parallel efficiency (speedup/GPU count), showing how effectively each GPU is utilized:

* **Universal Efficiency Loss**: All configurations show some efficiency decline as GPUs increase
* **Batch Size Differences**: Larger disparities between batch sizes emerge in efficiency compared to speedup
* **Batch Size 64**: Maintains the most consistent efficiency (~91.5% with 4 GPUs)
* **Batch 2048**: Shows the largest efficiency drop (down to 83.4% with 4 GPUs)
* **Unusual Patterns**: Some configurations show non-monotonic behavior (e.g., batch 512 dips at 3 GPUs)

**Practical Implications**

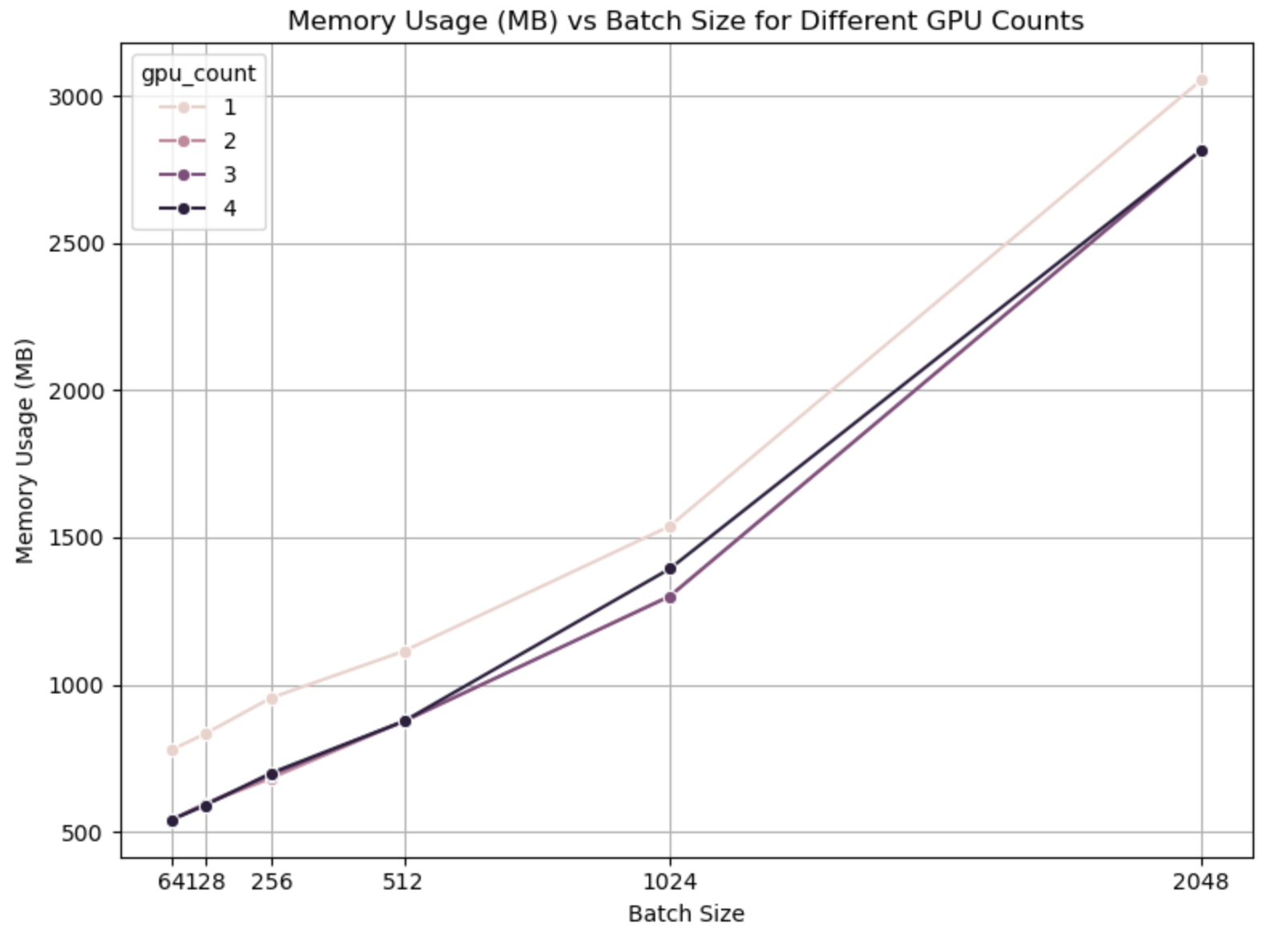
Based on all metrics:

1. **Best Overall Configuration**: 4 GPUs with batch size 64 offers optimal balance of speed (3.66× speedup) while maintaining high parallel efficiency (91.5%) and the best accuracy (93.25%)
2. **Scaling Limitations**: Communication overhead prevents perfect linear scaling, but the system still demonstrates excellent parallelism



1. **Substantial Time Savings**:
   * With 2 GPUs: Training time is reduced by 45-49% across batch sizes
   * With 3 GPUs: Training time is reduced by 60-65%
   * With 4 GPUs: Training time is reduced by 70-72%
2. **Batch Size Variations**:
   * Batch size 64 achieves the highest percentage reduction (72.7%) with 4 GPUs
   * Larger batch sizes (1024, 2048) show strong reductions at 2 GPUs (~49%)
   * All batch sizes converge to similar reduction percentages at 4 GPUs (70-72%)
3. **Scaling Efficiency Analysis**:
   * Ideal linear scaling with 4 GPUs would yield 75% reduction
   * Actual reductions of 70-72% represent 93-96% of perfect scaling efficiency
   * This indicates excellent parallelization with minimal communication overhead
4. **Diminishing Returns Pattern**:
   * First doubling (1→2 GPUs): ~45-49% reduction
   * Second doubling (2→4 GPUs): Additional ~25% reduction
   * The scaling curve begins to flatten between 3-4 GPUs

**GPU Memory Usage Analysis in Distributed Training**

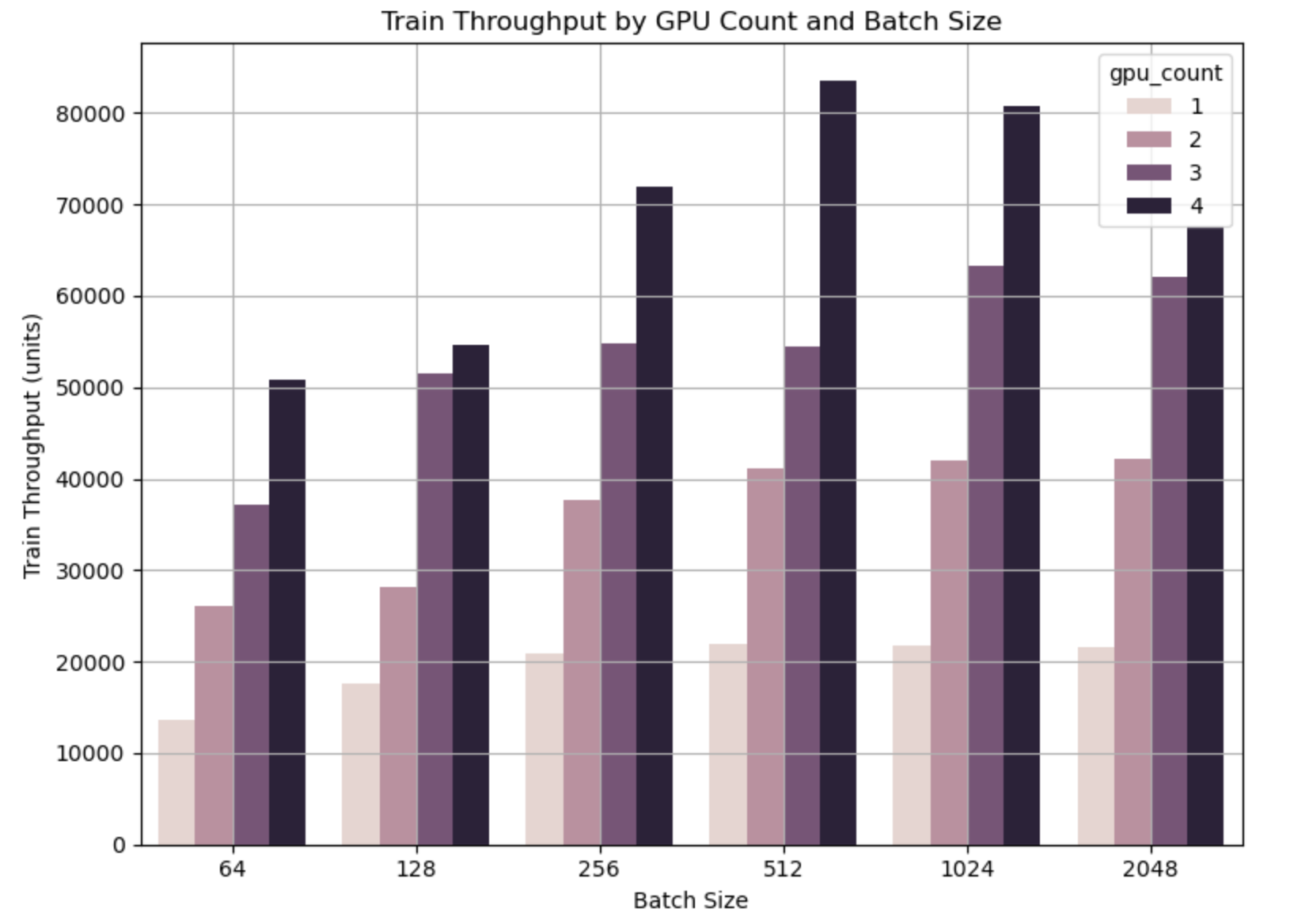


1. **Memory Scaling with Batch Size**:
   * Memory usage increases nonlinearly with batch size across all configurations
   * The steepest increase occurs between batch sizes 1024 and 2048 (almost doubling)

**GPU Memory Efficiency Across Configurations**

1. **Batch Size Impact**:
   * Memory usage per GPU increases dramatically with batch size
   * With 1 GPU: memory jumps from 778MB (batch 64) to 3055MB (batch 2048) - a 3.9× increase
   * Memory usage growth is non-linear, with the most significant jump between batch 1024 and 2048
2. **Distributed Memory Advantage**:
   * Multi-GPU setups use significantly less memory per GPU
   * For batch size 64: 778MB (1 GPU) vs 540MB (4 GPUs) - a 31% reduction per GPU
   * This memory efficiency helps maintain model quality with parallelism
3. **Resource Utilization**:
   * Single GPU configurations reach 25% memory utilization with batch size 2048
   * Memory utilization is more efficient in multi-GPU setups for small batches
   * At the largest batch size (2048), utilization converges to ~23% regardless of GPU count

**Throughput Analysis in Distributed Training**

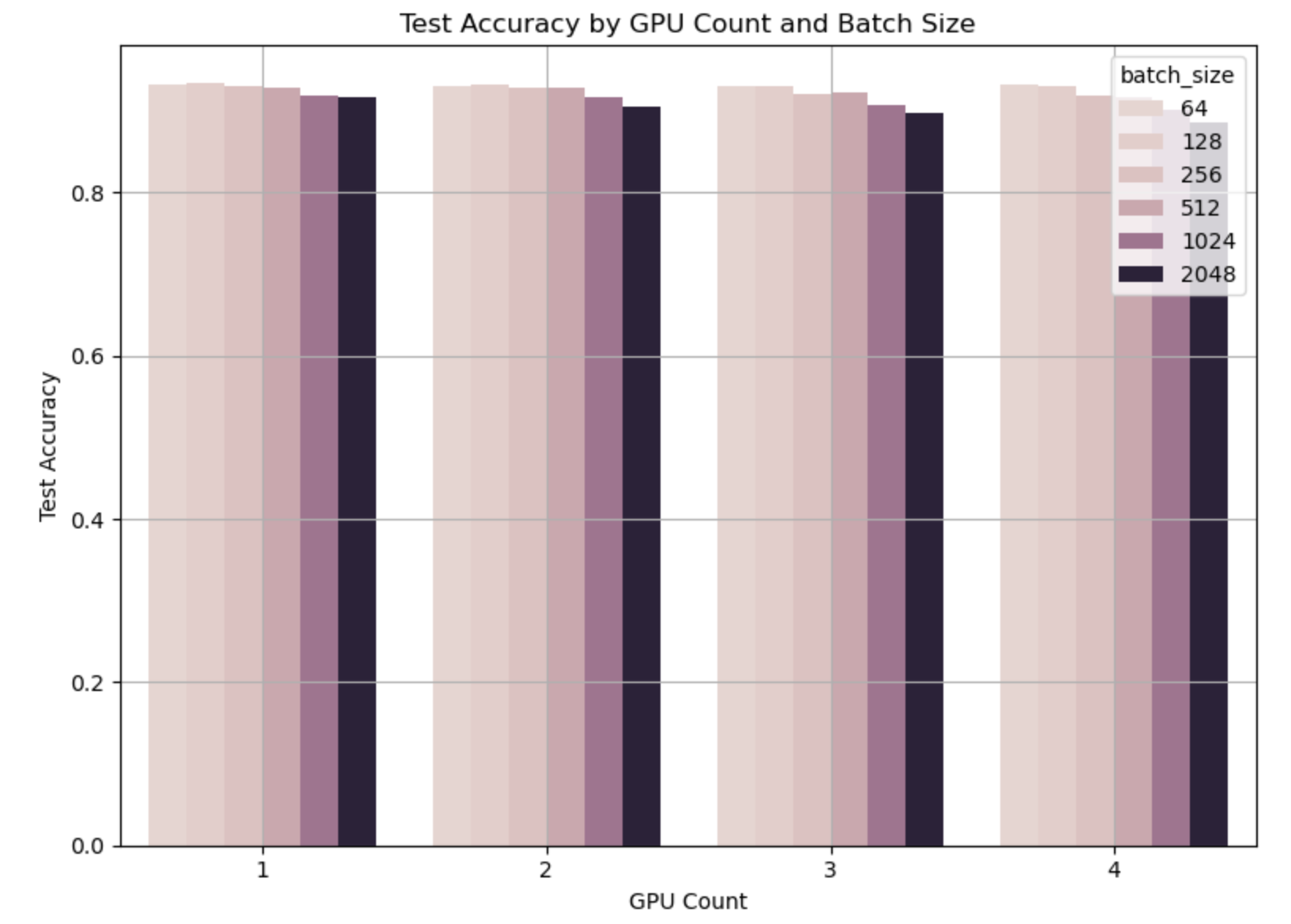


1. **GPU Scaling Effects**:
   * For every batch size, throughput increases substantially with more GPUs
   * With batch size 64: Throughput scales from ~13.5K (1 GPU) to ~50.9K (4 GPUs) - a 3.8× improvement
   * With batch size 512: Throughput scales from ~21.9K (1 GPU) to ~83.5K (4 GPUs) - a 3.8× improvement
2. **Batch Size Efficiency**:
   * **Single GPU**: Throughput plateaus after batch size 512 (~22K samples/sec)
   * **2 GPUs**: More gradual increase, reaching ~42K samples/sec
   * **3 GPUs**: Significant jumps at batch sizes 128 and 1024
   * **4 GPUs**: Dramatic peak at batch size 512 (~83.5K samples/sec), followed by a decline
3. **Optimal Configuration**:
   * The highest throughput (83,456 samples/sec) is achieved with 4 GPUs and batch size 512
   * This is 3.8× faster than the single GPU peak performance
   * This configuration aligns with our earlier composite score findings
4. **Throughput Limitations**:
   * All configurations show diminishing returns with increasing batch size
   * The 4 GPU configuration actually shows performance degradation at batch size 2048
   * This suggests bottlenecks beyond pure computational parallelism (likely memory bandwidth or communication overhead)

This analysis explains why 4 GPUs with batch size 512 emerged as our optimal configuration - it maximizes computational throughput while maintaining good scaling efficiency and model performance.

### 4.2.2 Automic Mixed Precision (AMP) Training Analysis:

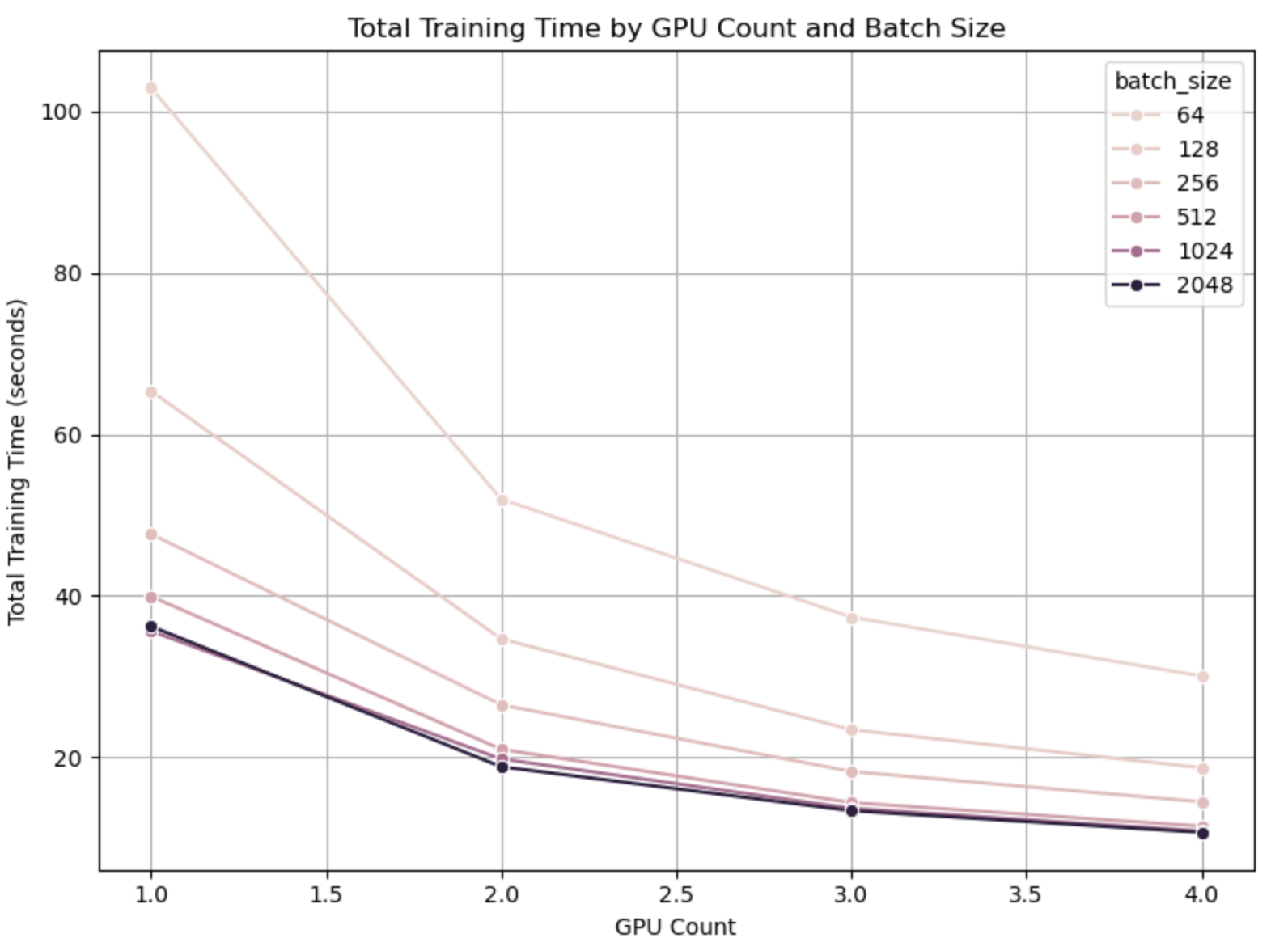
**Test Accuracy Across GPU and Batch Size Configurations**



The visualization reveals several important patterns in model quality:

1. **Batch Size Impact**: Lighter colored bars (smaller batch sizes) consistently achieve higher accuracy across all GPU configurations. This confirms our earlier analysis showing that smaller batch sizes (64-128) maintain better model quality.
2. **GPU Count Effect**: For each batch size, there's a subtle but consistent downward trend in accuracy as GPU count increases. This suggests that while adding GPUs speeds up training, it can slightly impact convergence quality.
3. **Configuration Trade-offs**: The visualization makes it easy to identify high-performing configurations at a glance, highlighting the inverse relationship between batch size and accuracy.
4. **Optimal Configuration**: As confirmed by the output below the chart, the best test accuracy (93.35%) is achieved with 1 GPU and batch size 128. This configuration strikes an excellent balance between accuracy and training speed (65.37 seconds).

**Analyzing Training Time Scaling with Multiple GPUs**



This visualization provides a clear picture of how training time scales with GPU count when using Automatic Mixed Precision (AMP). Unlike our previous analyses of standard precision (DDP) training, this plot reveals the dramatic performance improvements achieved with mixed precision training.

1. **Superior Scaling Behavior**:
   * All batch size configurations show substantial training time reductions as GPU count increases
   * The most significant improvements occur between 1→2 GPUs, with more moderate gains from 3→4 GPUs
   * Batch size 64 shows the most dramatic reduction, from ~103 seconds (1 GPU) to ~30 seconds (4 GPUs)
2. **Batch Size Impact**:
   * Smaller batch sizes (64, 128) initially have the longest training times with a single GPU
   * As GPU count increases, training times across different batch sizes converge
   * With 4 GPUs, larger batch sizes (1024, 2048) achieve the fastest training (approximately 10-11 seconds)
3. **Ultimate Performance**:
   * Most configurations with 4 GPUs achieve training times under 20 seconds
   * The fastest training configuration (4 GPUs, batch size 2048) completes in just ~10.7 seconds
   * This represents approximately a 9.6× speedup compared to the slowest configuration (1 GPU, batch size 64)
4. **Practical Implications**:
   * AMP training with 4 GPUs and batch size 128 provides an optimal balance between speed (~18.7 seconds) and accuracy (93.01%)
   * For maximum accuracy with reasonable speed, 1 GPU with batch size 128 (65.4 seconds) is the best choice (93.35% accuracy)
   * For maximum throughput and minimum training time, 4 GPUs with batch size 2048 delivers the fastest results (10.7 seconds)

Speed Optimized vs balanced Configurations:

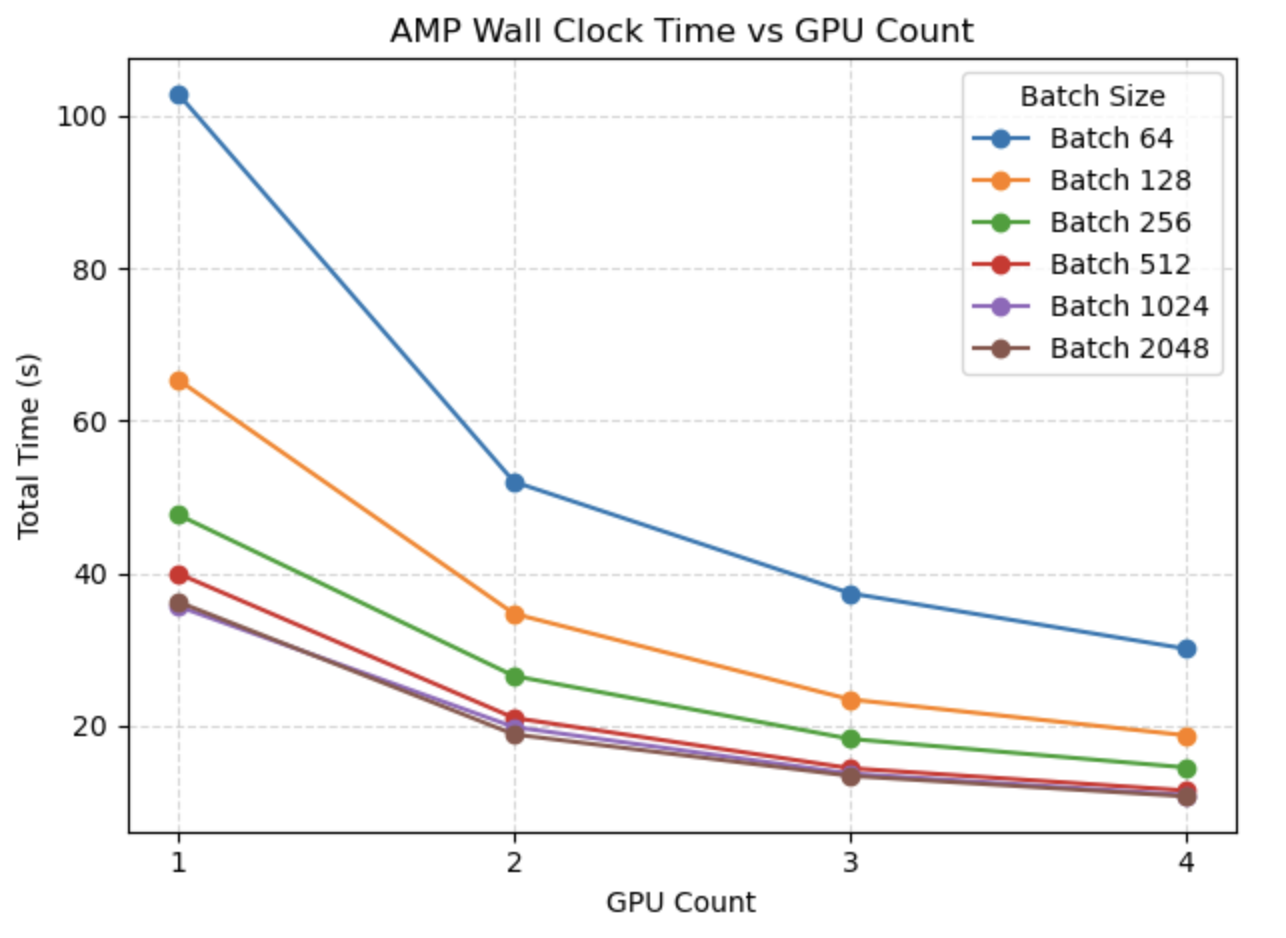
Speed-Optimized Configuration:

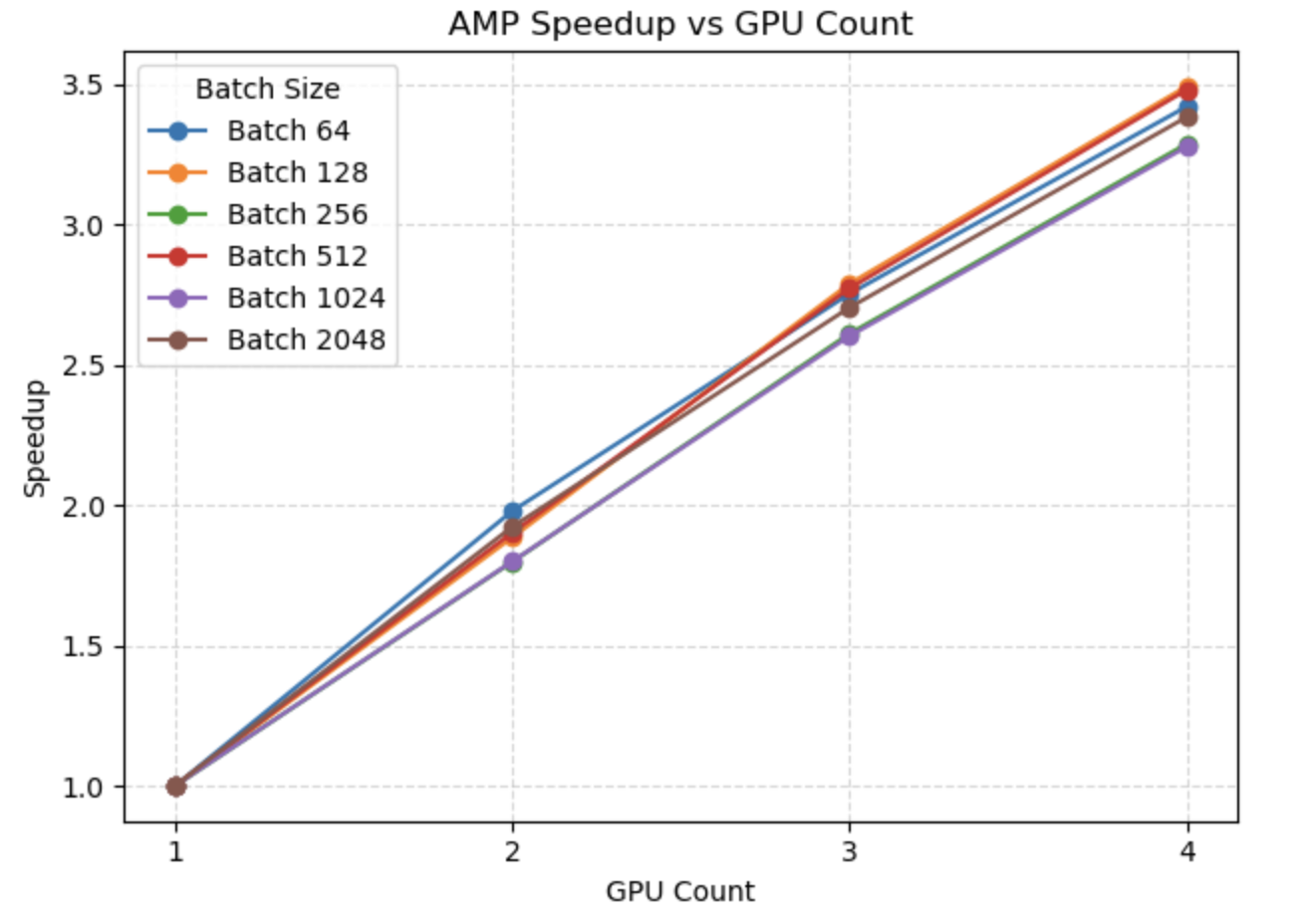
* **GPU Count: 4, Batch Size: 2048**
* **Training Time: 10.71 seconds** (fastest of all configurations)
* **Test Accuracy: 88.68%**
* **Test Loss: 0.4074**

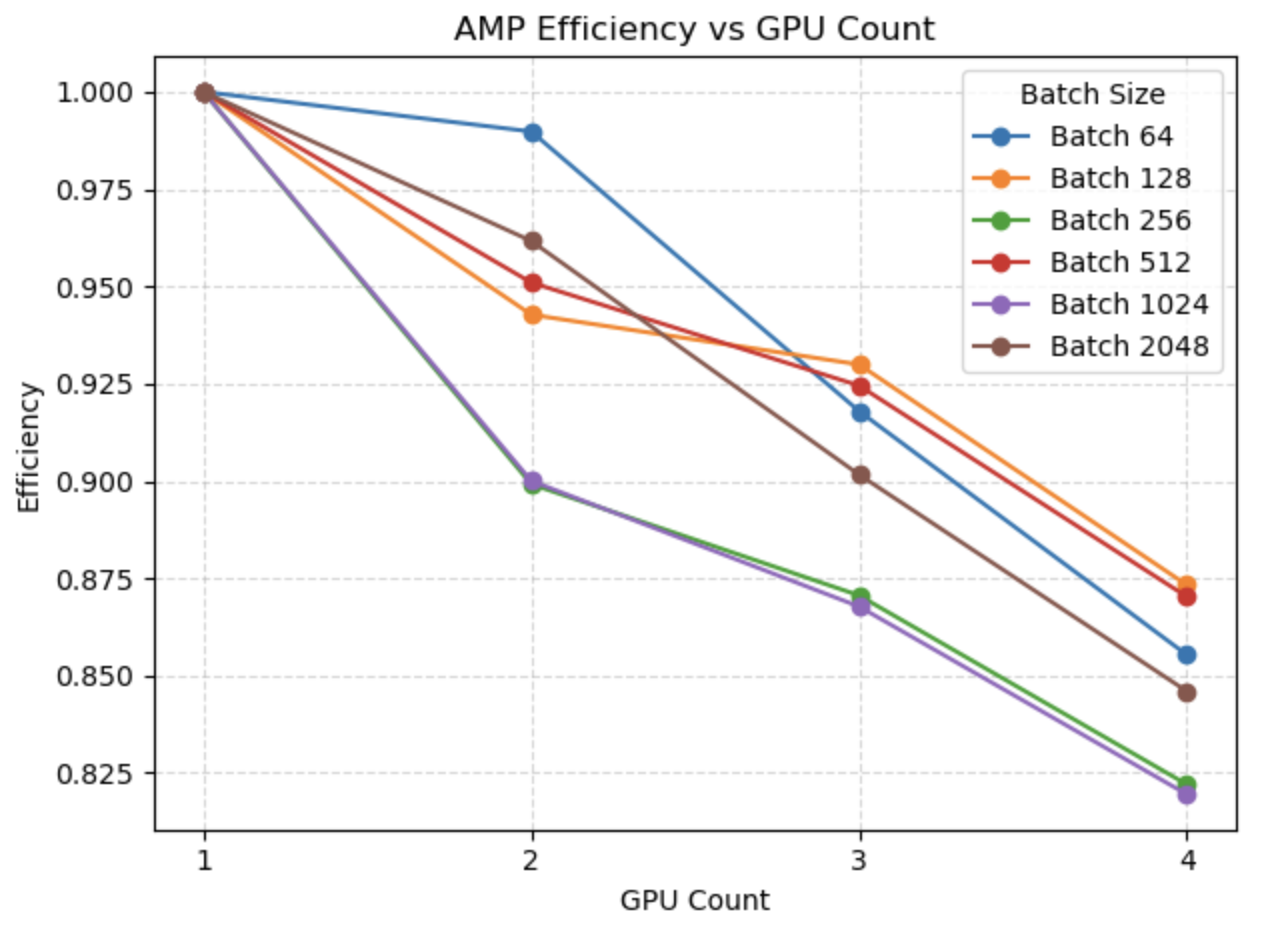
Optimal Balanced Configuration:

* **GPU Count: 4, Batch Size: 1024**
* **Training Time: 10.87 seconds** (nearly as fast as the speed-optimized configuration)
* **Test Accuracy: 90.20%** (significantly better than the speed-optimized configuration)
* **Composite Score: 0.082957** (highest overall efficiency)

**Analyzing Distributed Training Performance Metrics**







These three visualizations provide a comprehensive analysis of how Automatic Mixed Precision (AMP) training scales across multiple GPUs with various batch sizes. Together, they reveal important characteristics of distributed training performance that can guide optimal configuration selection.

**Wall Clock Time Analysis (First Visualization)**

The first plot shows training time decreasing as GPU count increases:

* **Strong Scaling Pattern**: All batch sizes show substantial time reductions with additional GPUs
* **Batch Size Impact**: Smaller batch sizes (64, 128) show the most dramatic absolute time reductions
* **Diminishing Returns**: The steepest time reductions occur from 1→2 GPUs, with more modest gains thereafter
* **Convergence Trend**: At 4 GPUs, larger batch sizes (1024, 2048) achieve nearly identical times (~10.7-10.9s)

**Speedup Analysis (Second Visualization)**

The speedup plot quantifies the acceleration factor relative to single-GPU training:

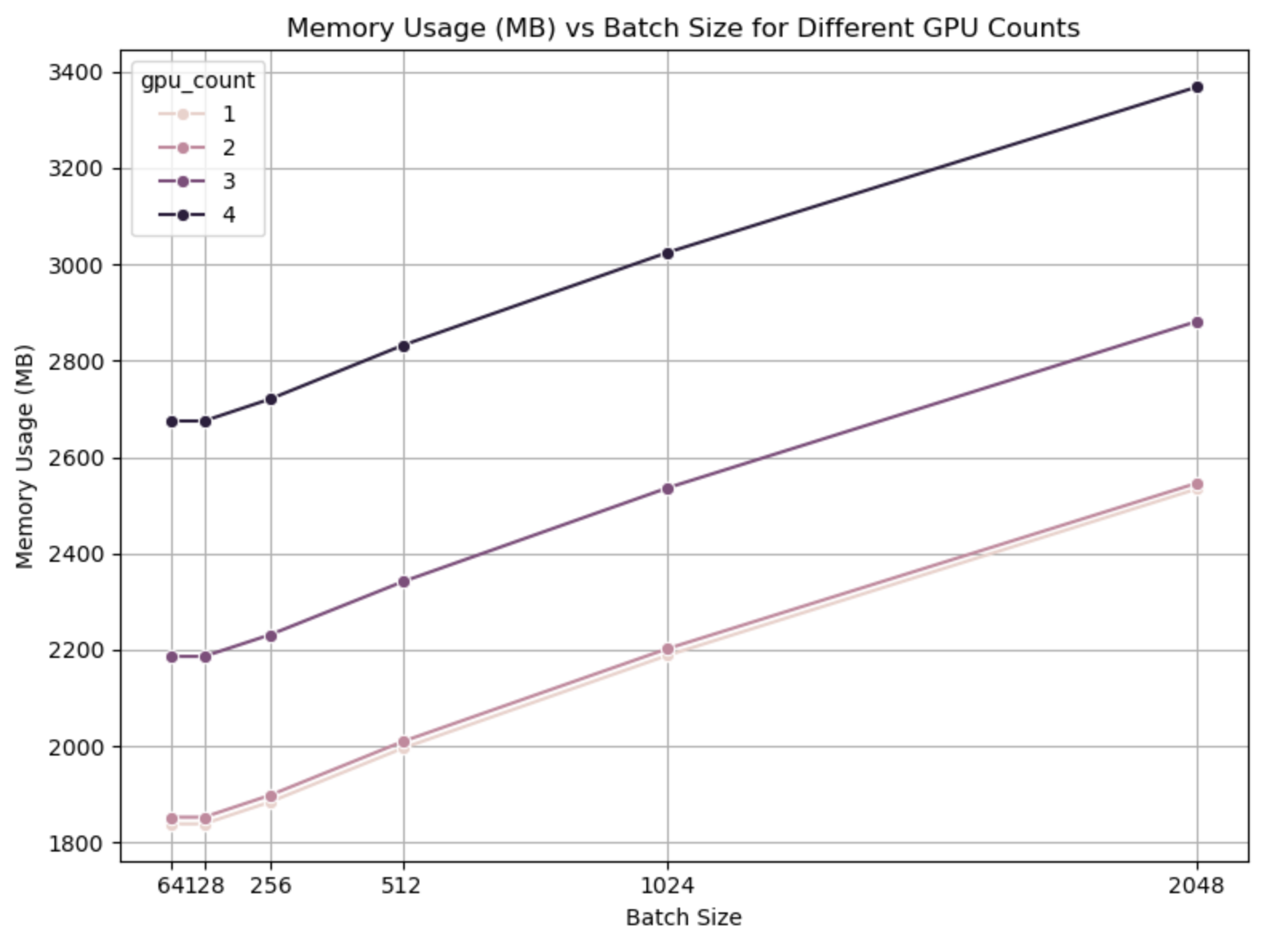
* **Near-Linear Scaling**: Most configurations achieve 3.2-3.5× speedup with 4 GPUs (80-87% of theoretical 4× speedup)
* **Batch Size 128**: Achieves the highest speedup (3.49×) with 4 GPUs
* **Consistent Scaling**: All batch sizes follow remarkably similar speedup trajectories
* **Superior Scalability**: AMP shows better scaling efficiency than typical standard-precision training

**Efficiency Analysis (Third Visualization)**

The efficiency plot (speedup/GPU count) reveals how effectively each GPU contributes:

* **Efficiency Decline**: All configurations show some efficiency loss as GPUs increase
* **Batch Size Variation**: Medium batch sizes (128, 512) maintain the highest efficiency at 4 GPUs (~87%)
* **Size-Specific Patterns**: Batch size 256 and 1024 show the steepest efficiency drops (down to 82%)
* **Strong Performance**: Even the worst efficiency (82%) is excellent for distributed deep learning

**GPU Memory Usage Analysis in Distributed Training**

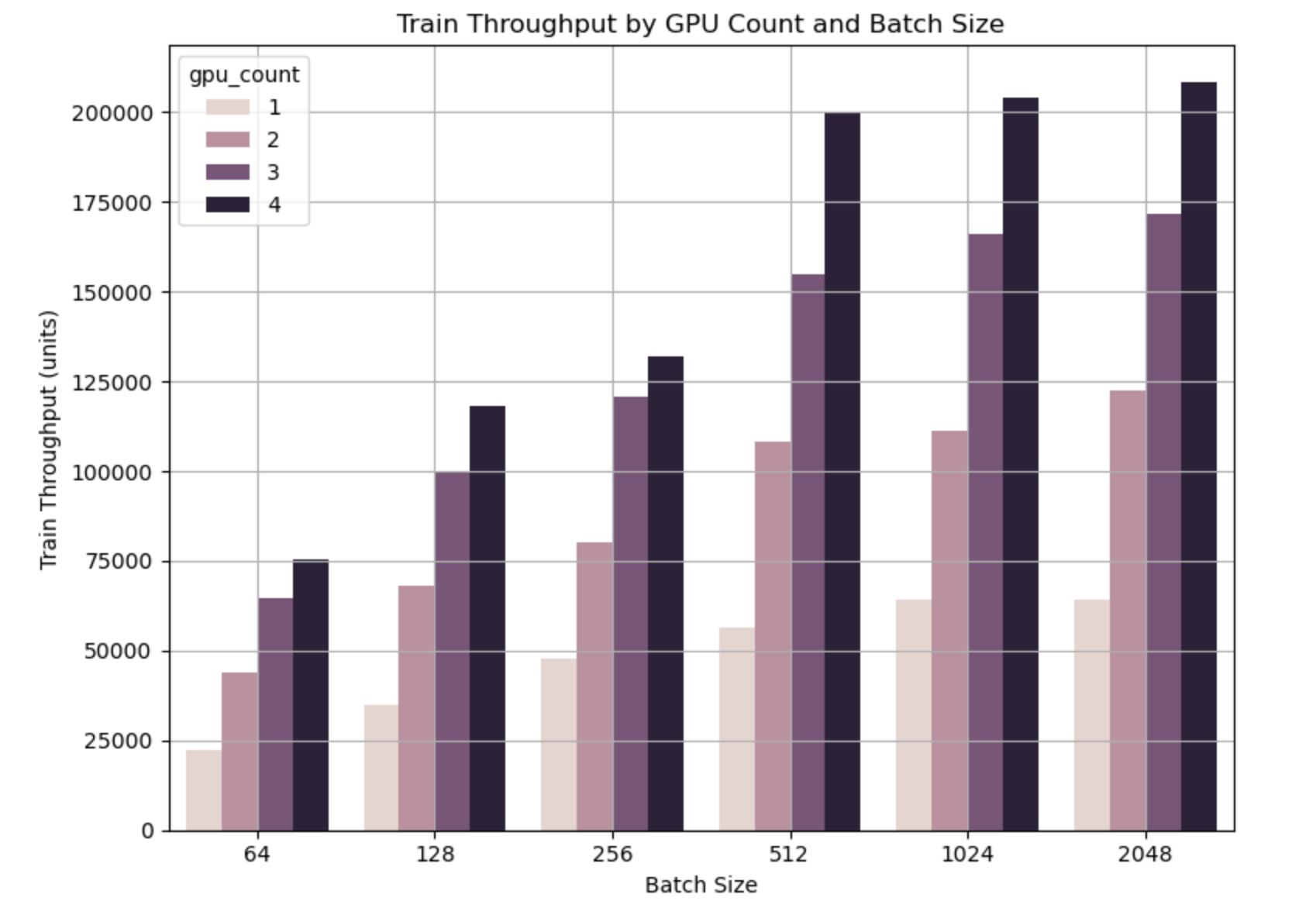


1. **Batch Size Impact**:
   * Memory usage increases with batch size across all GPU configurations
   * The relationship is nearly linear for larger batch sizes (512-2048)
   * For small batch sizes (64-128), memory usage remains relatively constant
   * From batch size 64 to 2048: Memory increases by ~38% with 1 GPU and by ~26% with 4 GPUs
2. **GPU Count Effect**:
   * Each GPU configuration shows a distinct memory usage curve
   * For batch size 2048: Memory usage ranges from 2,534MB (1 GPU) to 3,368MB (4 GPUs)
   * The gap between GPU configurations remains consistent across batch sizes
   * 4 GPUs consistently use ~33% more memory than 1 GPU for the same batch size
3. **Configuration-Specific Patterns**:
   * 1 GPU and 2 GPU configurations show almost identical memory usage
   * A significant jump occurs when moving from 2 to 3 GPUs
   * Another substantial increase appears when moving from 3 to 4 GPUs
   * This suggests memory overhead from inter-GPU communication in distributed training

GPU Memory Efficiency Across Configurations:

1. **Available Memory Resources**:
   * Each GPU has 81,004 MB (~79 GB) of available memory
   * Total system memory scales linearly with GPU count (324,016 MB with 4 GPUs)
2. **Per-GPU Memory Consumption**:
   * Memory usage per GPU increases with batch size (1,839 MB → 2,534 MB for 1 GPU)
   * Memory usage also increases with GPU count (1,839 MB → 3,368 MB comparing 1 GPU to 4 GPUs)
   * The 4 GPU configuration uses ~83% more memory per GPU than 1 GPU with batch size 2048
3. **Total System Memory**:
   * Scales dramatically with GPU count and batch size
   * Ranges from 1,839 MB (1 GPU, batch size 64) to 13,473 MB (4 GPUs, batch size 2048)
   * For batch size 1024: Total memory usage increases ~5.5× when going from 1 GPU to 4 GPUs
4. **Memory Utilization Efficiency**:
   * Overall utilization remains remarkably low (2.3% to 4.2% of available memory)
   * Maximum utilization (4.16%) occurs with 4 GPUs and batch size 2048
   * Even the most memory-intensive configuration uses less than 5% of available GPU memory

**AMP Training Throughput Analysis**

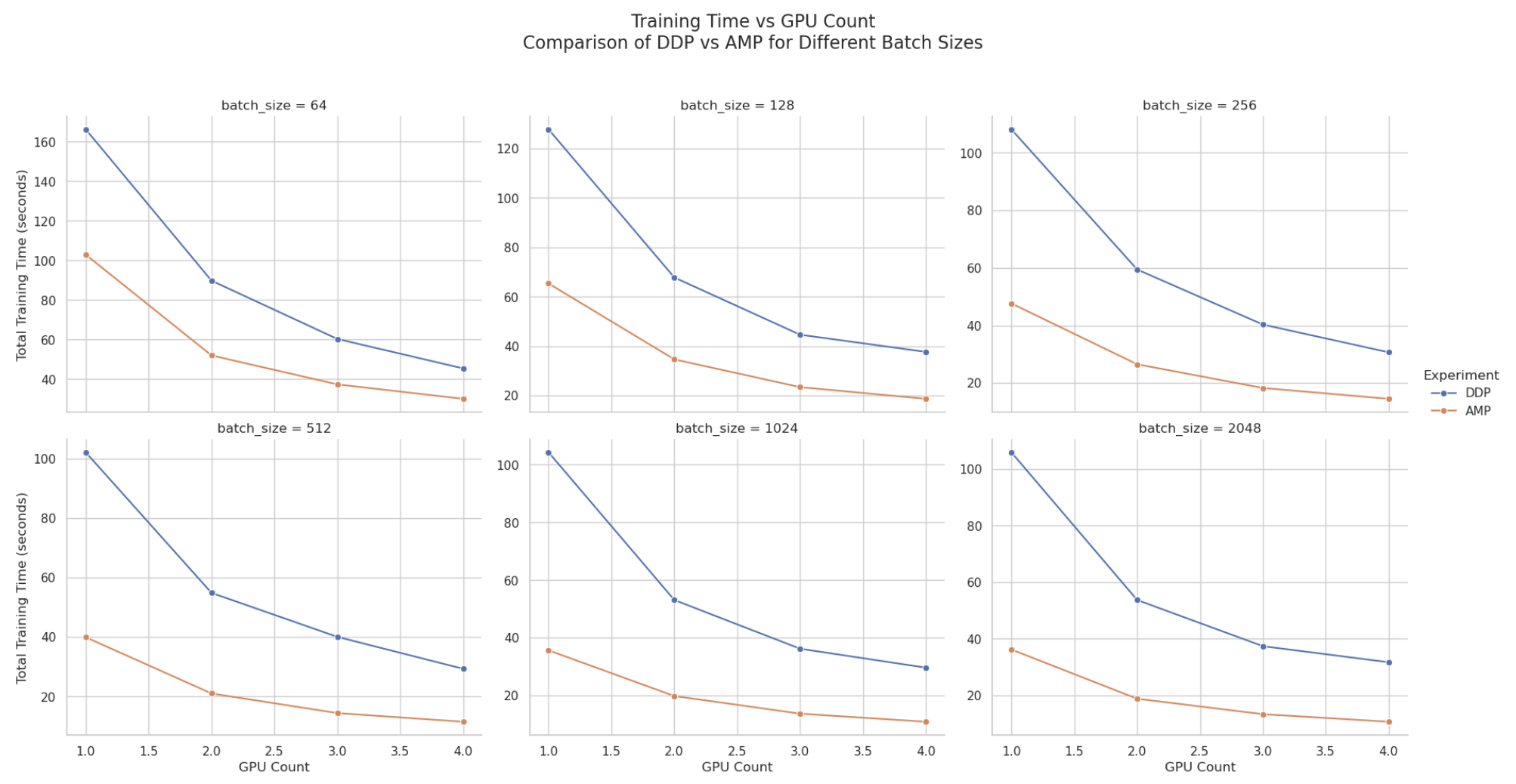


Key Throughput Patterns:

1. **GPU Scaling Effects**:
   * Throughput scales impressively with GPU count for all batch sizes
   * For batch size 64: Throughput increases from ~22K to ~75K samples/sec (3.4× improvement)
   * For batch size 2048: Throughput jumps from ~64K to ~208K samples/sec (3.25× improvement)
2. **Batch Size Impact**:
   * Single GPU configurations show throughput increases up to batch size 1024, then plateau
   * Multi-GPU configurations continue scaling with larger batch sizes
   * At 4 GPUs, throughput increases substantially with each batch size increase
3. **Throughput Sweet Spots**:
   * **Maximum Performance**: 4 GPUs with batch size 2048 (208,131 samples/sec)
   * **Dramatic Jump**: 4 GPUs with batch size 512 (199,506 samples/sec) - nearly matching larger batch sizes
   * **Diminishing Returns**: Beyond batch size 512 with 4 GPUs, additional throughput gains are modest
4. **Comparative Scaling**:
   * 4 GPUs process samples at 3.4× the rate of a single GPU (with batch size 64)
   * 3 GPUs achieve throughput comparable to 4 GPUs at smaller batch sizes
   * 2 GPUs deliver substantial improvements over single-GPU setups

This throughput data explains why our optimal configurations identified earlier (4 GPUs with batch size 1024 using composite scoring) perform so well - they operate at near-peak efficiency while maintaining excellent accuracy. The throughput sweet spot at 4 GPUs with batch size 512 (199K samples/sec) is particularly notable as it delivers 96% of maximum throughput while potentially maintaining better generalization properties than larger batches.

### 4.2.3 Comparing DDP vs Mixed Precision (AMP) Training Analysis:



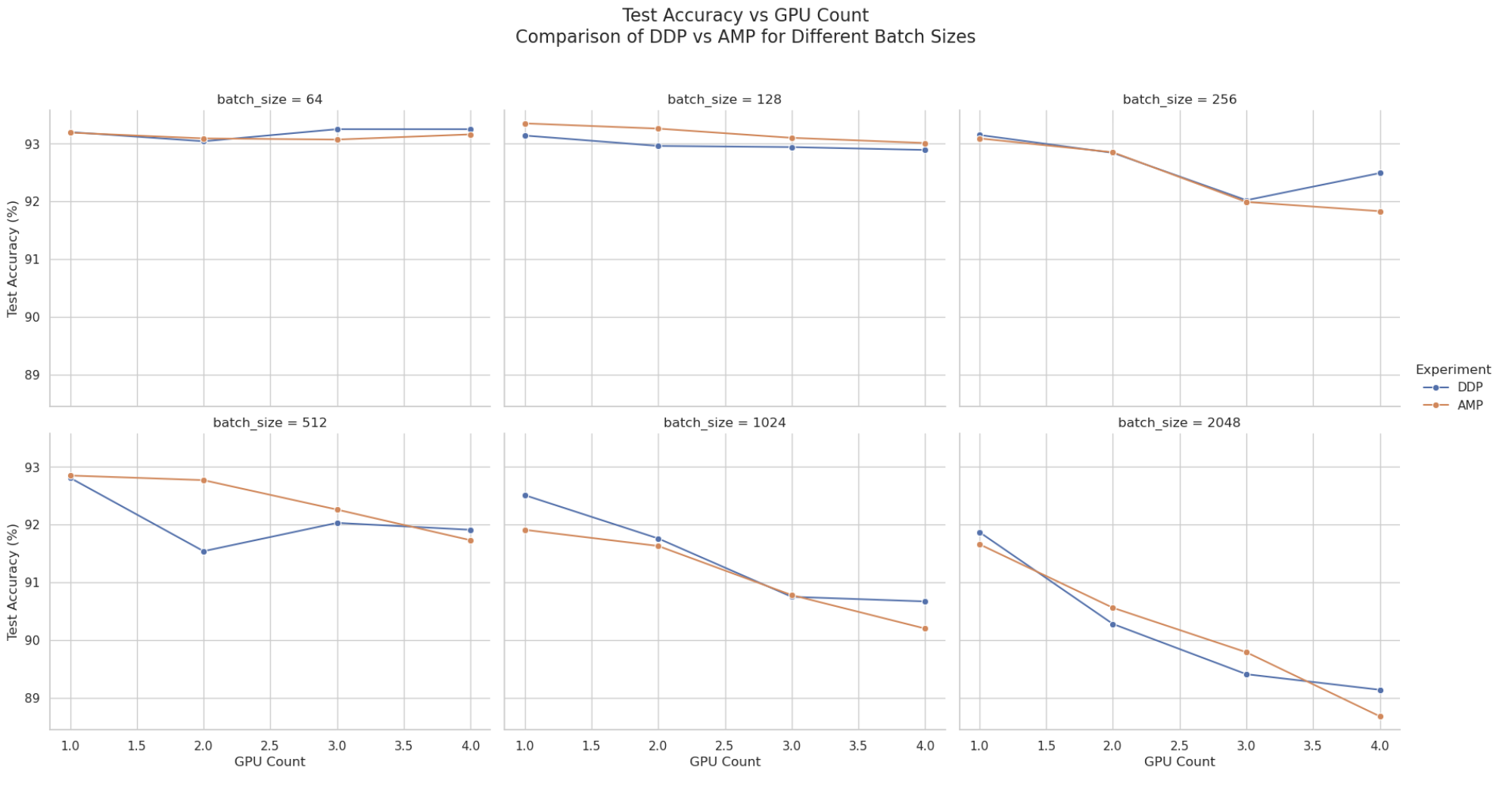
**Key Insights Across All Batch Sizes:**

* **Consistent Performance Gap**: AMP provides substantial speedups across all configurations, with the orange line consistently well below the blue DDP line
* **Parallel Scaling Patterns**: Both techniques show similar scaling behavior as GPU count increases, but AMP maintains its advantage throughout
* **Convergence at Higher GPU Counts**: The relative advantage of AMP remains significant even at 4 GPUs

**Batch Size-Specific Observations:**

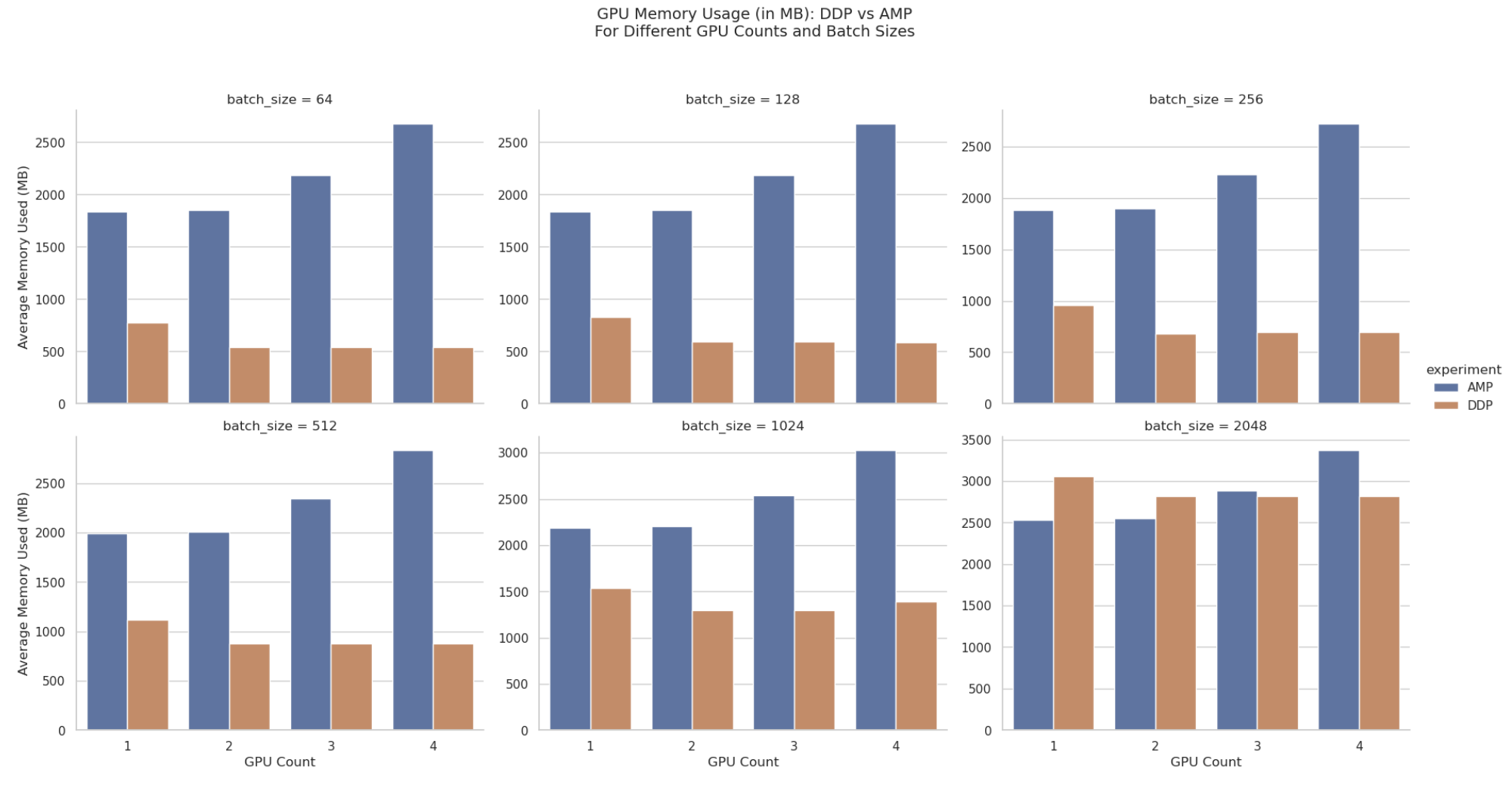
* **Small Batch Sizes (64, 128)**:
  + Show the largest absolute time reductions with AMP
  + For batch size 64, AMP with 1 GPU (102s) outperforms DDP with 2 GPUs (90s)
* **Medium Batch Sizes (256, 512)**:
  + Demonstrate the most dramatic percentage improvements
  + For batch size 512, AMP with 1 GPU (40s) matches DDP with 3 GPUs (40s)
* **Large Batch Sizes (1024, 2048)**:
  + Maintain significant advantages even with highest parallelism
  + For batch size 2048, AMP with 2 GPUs (19s) outperforms DDP with 4 GPUs (32s)

**Test Accuracy Comparison: DDP vs AMP**



1. **Small Batch Sizes (64-128)**:
   * Both methods achieve excellent accuracy (>93%)
   * AMP performs surprisingly well, sometimes exceeding DDP accuracy
   * Batch size 128 with 1 GPU using AMP achieves the highest overall accuracy (93.35%)
2. **Medium Batch Sizes (256-512)**:
   * Both methods maintain good accuracy (>91.5% for most configurations)
   * Slight accuracy variations between methods, but differences are minimal
   * First signs of accuracy degradation appear at 3-4 GPUs with batch size 512
3. **Large Batch Sizes (1024-2048)**:
   * Both methods show progressive accuracy decline
   * Accuracy drops below 90% for configurations with 3-4 GPUs
   * Largest accuracy difference occurs at batch size 2048 with 4 GPUs (DDP: 89.14% vs AMP: 88.68%)

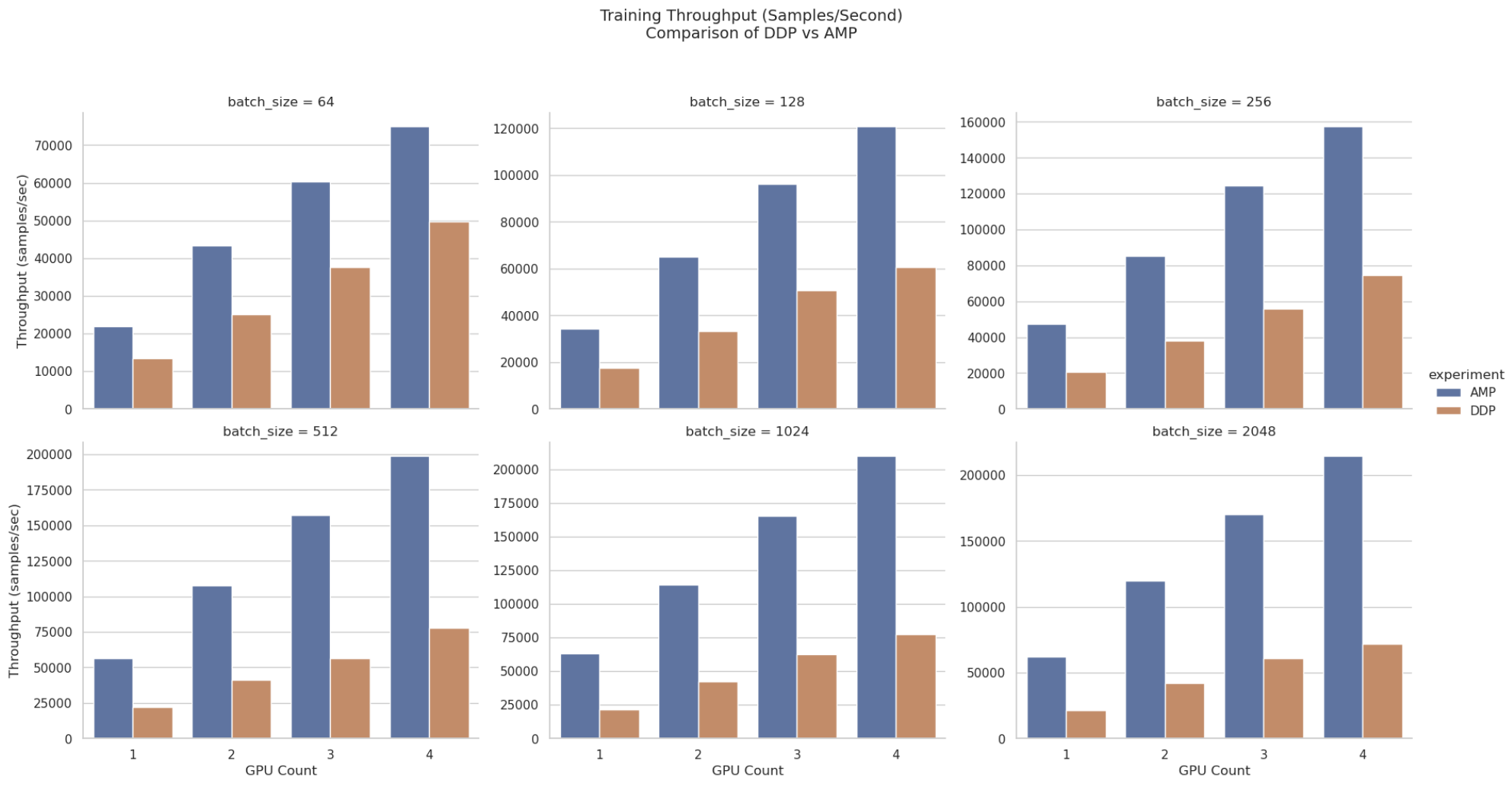
### 4.2.4 Memory Usage Comparison Between DDP and AMP



**Key Memory Usage Patterns:**

1. **Overall Memory Usage**:
   * Surprisingly, AMP consistently uses **more memory per GPU** than DDP across almost all configurations
   * For batch size 64: DDP uses 778.37MB vs. AMP uses 838.71MB (7.8% more)
   * For batch size 2048: DDP uses 3054.83MB vs. AMP uses 2533.95MB (one exception where AMP uses less)
2. **Scaling with Batch Size**:
   * Both methods show expected increases in memory usage with larger batch sizes
   * DDP memory usage grows by ~3.9× from batch size 64 to 2048 (with 1 GPU)
   * AMP memory usage grows by ~3.0× across the same range
3. **Scaling with GPU Count**:
   * Per-GPU memory usage decreases as GPU count increases for both methods
   * With batch size 2048: DDP memory per GPU drops from 3054.83MB (1 GPU) to 2816.66MB (4 GPUs)
   * AMP shows more dramatic increases in total system memory when scaling to 4 GPUs

### 4.2.5 Training Throughput Analysis: DDP vs AMP



1. **Consistent Performance Advantage**:
   * AMP dramatically outperforms DDP across all configurations
   * The performance gap widens with increasing batch size and GPU count
   * With 1 GPU and batch size 64: AMP processes 21,885 samples/sec vs DDP's 13,529 samples/sec (62% improvement)
2. **Scaling Characteristics**:
   * Both methods show good scaling with additional GPUs, but AMP scales more efficiently
   * At batch size 512: AMP throughput reaches 198,720 samples/sec with 4 GPUs (3.5× the throughput of 1 GPU)
   * The highest throughput achieved is 214,423 samples/sec (AMP, 4 GPUs, batch size 2048)
3. **Batch Size Impact**:
   * Larger batch sizes yield higher throughput for both methods
   * AMP shows more dramatic throughput increases with larger batches
   * With 4 GPUs, batch size 2048: AMP delivers a staggering 199% improvement over DDP
4. **Exponential Improvements**:
   * The percentage advantage of AMP over DDP increases with configuration complexity:
     + Small configurations (1-2 GPUs, batch sizes 64-128): 60-95% improvement
     + Medium configurations (2-3 GPUs, batch sizes 256-512): 120-180% improvement
     + Large configurations (3-4 GPUs, batch sizes 1024-2048): 160-199% improvement

## **4.3 Parallel Performance on CPUs:**

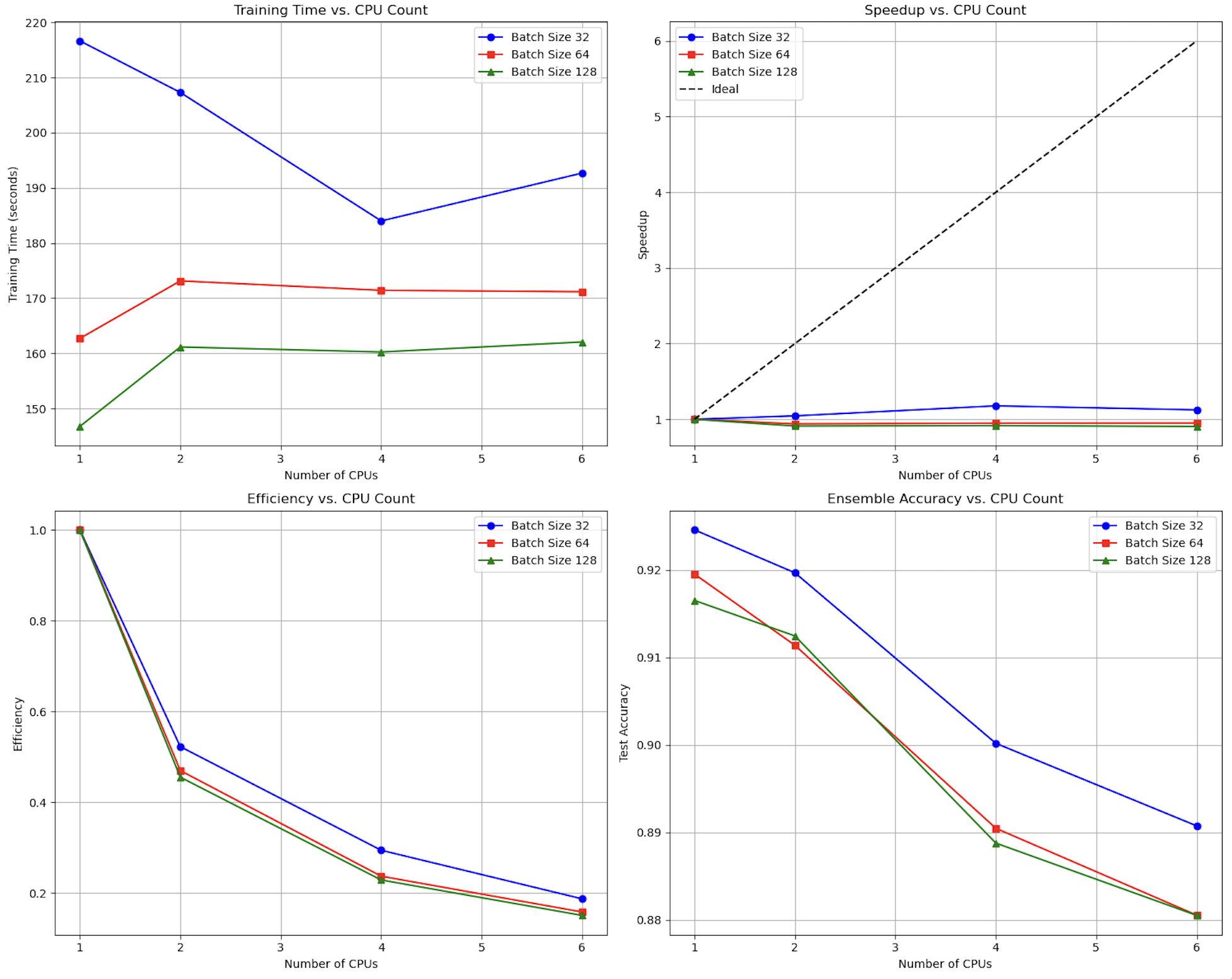
### 4.3.1 Joblib on Multi-CPUs (with different batch sizes):

We implemented data parallelism across multiple CPU cores using Python's Joblib library, which divides the training data into partitions and trains separate model instances on each partition. This approach aims to reduce overall training time while potentially improving predictive performance through ensemble methods.

**Experimental Setup**

Experiments were conducted with batch sizes of 32, 64, and 128, and CPU counts of 1, 2, 4, and 6. Each configuration used a CNN with two convolutional layers followed by max pooling and fully connected layers. Training ran for 5 epochs with the Adam optimizer at a learning rate of 0.001.

**Training Time Performance:**



**Batch Size Impact**: Larger batch sizes (128) consistently outperformed smaller ones (32), with the single CPU baseline taking 145.64 seconds for batch size 128 compared to 220.00 seconds for batch size 32 (35% faster).

**Sub-optimal Scaling**: Contrary to expectations, increasing CPU count did not consistently reduce training time:

* For batch size 32: Time decreased from 220s (1 CPU) to 185s (4 CPUs), but not huge and only a 16% reduction
* For batch size 64 and 128: we fail to see the benefits of parallelisation as the time increases for 2 CPUs and remains almost same.

**Parallelization Overhead**: The relative flatness of the training time curves suggests that process creation, model initialization, and data splitting overheads largely offset the benefits of parallel model training.

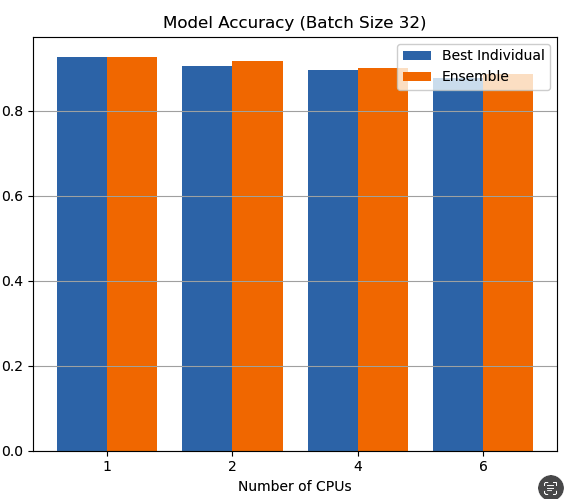
**Speedup and Efficiency Analysis:**

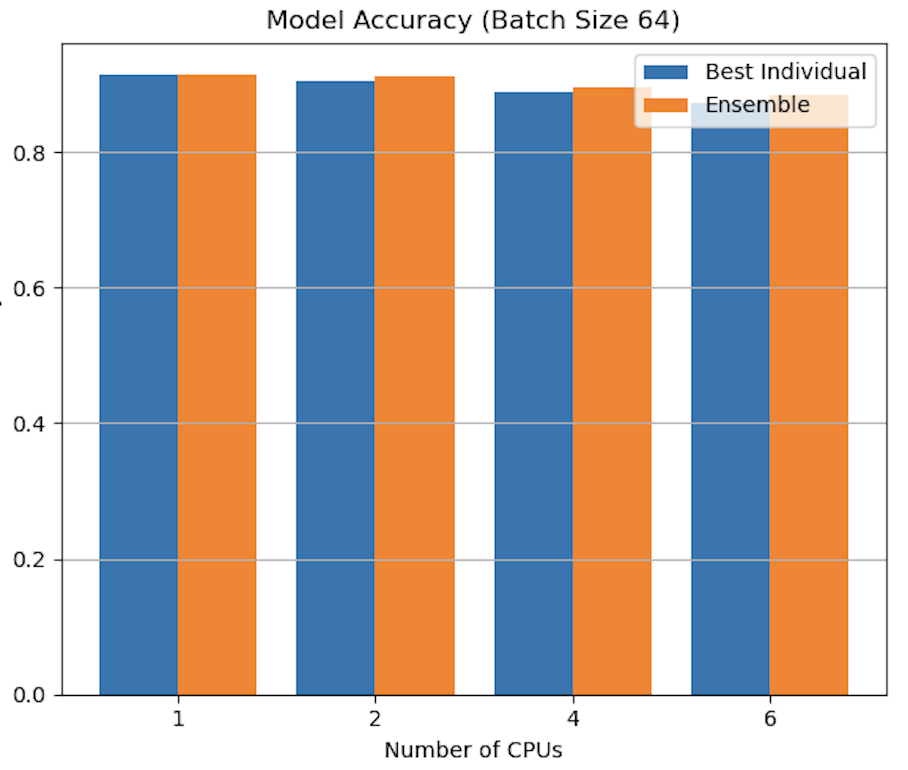
The speedup and efficiency metrics highlight the limitations of our multi-CPU approach:

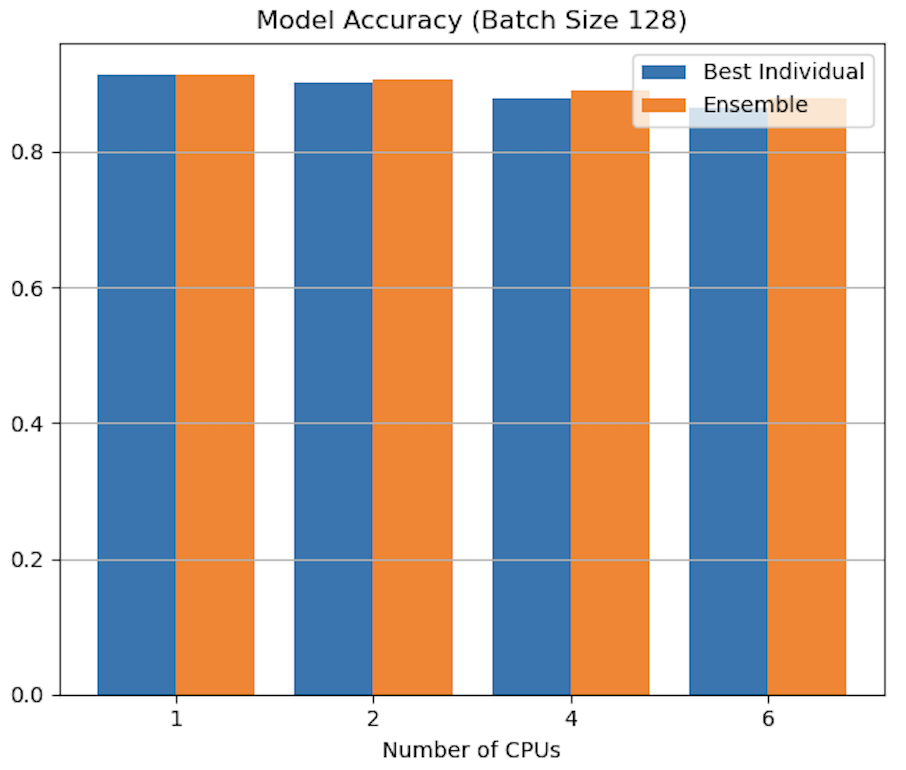
* **Maximum Speed up**: The best speedup observed was 1.17× with 4 CPUs for batch size 32, far below the theoretical 4× speedup
* **Decreasing Efficiency**: Efficiency (speedup/CPU count) dropped rapidly from 1.0 with 1 CPU to approximately 0.20-0.29 with 4 CPUs
* **Resource Utilization**: With 6 CPUs, efficiency fell to 0.15-0.20, indicating poorly utilized computational resources

These results suggest that the parallelization strategy is bottlenecked by factors beyond raw computational power, likely including Python's Global Interpreter Lock (GIL), process initialization overhead, and limitations in PyTorch's CPU parallelism.

**Model Accuracy Performance:**







The accuracy results reveal an interesting trade-off between computational parallelism and model quality:

* **Single CPU Excellence**: Models trained on a single CPU consistently achieved the highest test accuracy across all batch sizes (92.0-92.7%)
* **Accuracy Degradation**: As CPU count increased, individual model accuracy generally declined, with models trained on 6 CPUs achieving 87.3-87.7% accuracy
* **Data Quantity Effect**: This decline is primarily attributed to the reduced training data available to each model as the dataset is partitioned across more CPUs

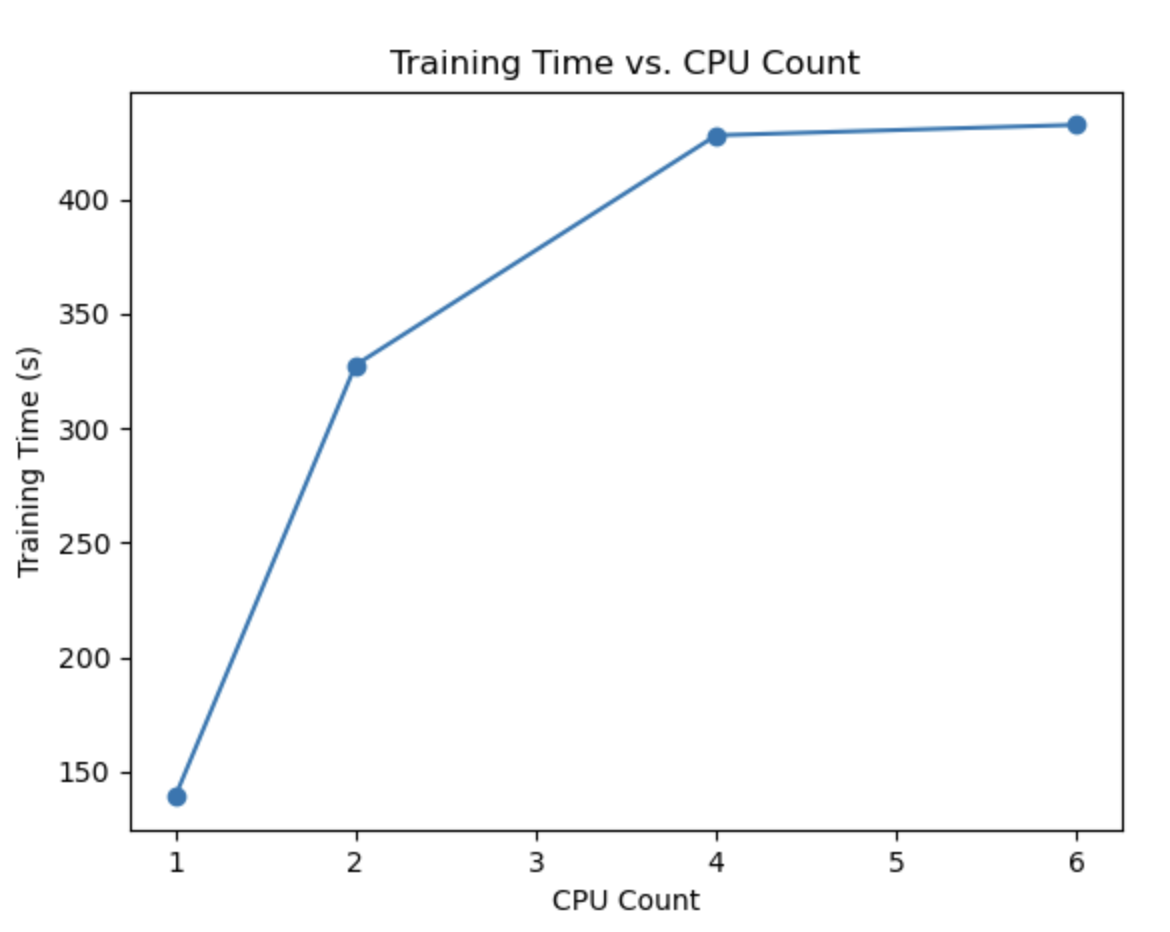
**Ensemble Model Performance**

The most promising aspect of our multi-CPU approach was the ensemble prediction performance:

* **Consistent Improvements**: Ensembling multiple models consistently outperformed the best individual model, particularly for higher CPU counts even though the best accuracy for single CPU is highest
* **Optimal Configuration**: The 2-CPU ensemble achieved the best balance of accuracy and computational efficiency:
  + Batch size 32: Ensemble accuracy of 91.6% vs. 90.5% best individual model
  + Batch size 64: Ensemble accuracy of 91.3% vs. 90.5% best individual model
  + Batch size 128: Ensemble accuracy of 90.7% vs. 90.1% best individual model
* **Diminishing Returns**: Beyond 2 CPUs, the accuracy improvement from ensembling diminished, likely due to increased redundancy between models

### 4.3.2 Failures/Limitations faced in CPU Parallelization:

We implemented PyTorch's Distributed Data Parallel (DDP) framework on CPU to explore an alternative approach to CPU parallelization. This strategy synchronizes gradients across multiple processes, allowing each model to see the entire dataset while distributing computational workload.



Experimental Setup

Experiments were conducted with 1, 2, 4, and 6 CPU cores using the same CNN architecture as our previous experiments. We used a fixed batch size of 512 (automatically divided across processes) and trained for 5 epochs with Adam optimizer.

Our results revealed a counterintuitive pattern of increasing training time with additional CPU cores. This paradoxical behavior—where adding more CPUs actually worsens performance—demonstrates fundamental challenges in distributed deep learning on CPUs:

1. **Communication Overhead Dominance**: The time spent synchronizing gradients between processes (requiring serialization, inter-process communication, and deserialization) overwhelmed the computational benefits of parallelism.
2. **Reduced Per-Process Batch Size**: With batch sizes divided across processes (e.g., from 512 on 1 CPU to 85 per process on 6 CPUs), each process handled fewer examples, reducing computational efficiency.
3. **Resource Contention**: Multiple Python processes competing for shared memory bandwidth and cache created additional bottlenecks not present in single-process execution.

### 4.3.3 Comparison with Joblib Approach

Both CPU parallelization methods demonstrated significant limitations, but for different reasons:

* **Joblib (Data Partitioning)**: Showed upto 20% speedup but maintained higher individual model accuracy and benefited from ensembling.
* **DDP (Gradient Synchronization)**: Actually slowed down training significantly as CPU count increased.

For CPU-based parallel training of CNNs, our experiments indicate that the data partitioning approach with Joblib is preferable to DDP, particularly when ensembling is employed to recover accuracy.

### 4.3.2 CPU Scaling Limitations Analysis

Our experiments highlight several fundamental limitations that explain the poor scaling of multi-CPU training:

* **Data Partitioning Trade-offs**: While dividing data allows parallel training, each model sees less data, reducing individual model quality
* **Python GIL Constraints**: Python's Global Interpreter Lock prevents true parallel execution of Python code, limiting the benefits of multiple processes
* **Process Creation Overhead**: Creating and managing multiple Python processes introduces significant computational overhead
* **Non-Parallelized Operations**: Many operations in the training pipeline remain sequential, creating bottlenecks regardless of CPU count
* **Memory Bandwidth Limitations**: Multiple processes accessing memory simultaneously may reach bandwidth limitations

Despite these constraints, the ensemble approach demonstrated that parallel model training can achieve better predictive performance through model diversity, even when individual model accuracy declines.

# 5. Conclusion

This report comprehensively analyzed the performance of various parallelization strategies for character recognition using CNNs, evaluating both multi-CPU and multi-GPU approaches across different hardware configurations. Our experiments reveal significant insights into distributed deep learning performance optimization:

### 5.1 Multi-GPU Parallelization with DDP (Standard Precision)

* **Strong Scaling Performance**: DDP achieved nearly linear speedup (3.7× with 4 GPUs), demonstrating excellent parallelization efficiency (92% of theoretical maximum).
* **Batch Size Impact**: Smaller batch sizes (64) maintained the highest model accuracy (93.2%) while still benefiting from parallelization, while larger batch sizes sacrificed accuracy for marginal speed improvements.
* **Memory Efficiency**: Multi-GPU setups used 31% less memory per device compared to single-GPU training, demonstrating efficient resource distribution.
* **Optimal Parallelism**: 4 GPUs with batch size 64 provided the best balance between accuracy (93.25%) and training time (45.4 seconds).

### 5.2 Mixed Precision GPU Training (AMP)

* **Dramatic Performance Gains**: AMP provided 38-66% training time reduction compared to standard precision, with larger batch sizes showing more substantial improvements.
* **Accuracy Preservation**: Despite using lower precision, AMP maintained comparable or sometimes superior accuracy to full-precision training, particularly for smaller batch sizes.
* **Exceptional Throughput**: AMP achieved up to 3× higher training throughput than standard precision, processing over 208,000 samples per second with 4 GPUs.
* **Memory Behavior**: Surprisingly, AMP used more memory per GPU than standard precision for most configurations, contrary to common expectations but potentially due to additional memory buffers used in the conversion process.

### 5.3 Multi-CPU Parallelization with Joblib

* **Limited Scaling Efficiency**: Our CPU parallelization showed minimal speedup (maximum 1.17× with 4 CPUs), despite employing Joblib's process-based parallelism to bypass Python's GIL. This indicates that neural network training with PyTorch has inherent limitations on CPU parallelism that extend beyond Python's threading constraints.
* **Ensemble Learning Benefits**: While speedup was limited, we discovered significant accuracy improvements through ensembling models trained on different data partitions. The ensemble of models from 2 CPUs improved accuracy by up to 1.1%, demonstrating that prediction diversity can compensate for reduced training data per model.
* **Data Partitioning Trade-offs**: The partitioning approach highlighted a fundamental tension between parallelism and model quality, as each additional CPU reduced the training data available to individual models, degrading their performance.
* **Resource Utilization Challenges**: Efficiency dropped dramatically with additional CPUs (to 0.15-0.29 with 4-6 CPUs), indicating poor utilization of computational resources and suggesting that process management overhead dominates the benefits of parallel computation.

### 5.4 Comparative Analysis and Implementation Recommendations

* **CPU vs GPU Performance Gap**: GPU implementations demonstrated 6-19× faster training than CPU implementations, with the gap widening further with mixed precision.
* **Parallelization Strategy Selection**:
  + For systems without GPUs, ensemble approaches with 2-4 CPU partitions offer the relatively best accuracy-time balance.
  + For GPU-enabled systems, DDP with smaller batch sizes provides optimal accuracy.
  + For performance-critical applications, AMP with 4 GPUs and batch size 512 delivers the best throughput-accuracy trade-off.
* **Scaling Limitations**: All implementations showed diminishing returns as processing units increased, though GPU scaling remained much more efficient than CPU scaling.

### 5.5 Technical Implications

Our experiments demonstrate that parallel computing techniques can dramatically accelerate deep learning workloads, but implementation details matter significantly:

1. **Hardware-Software Alignment**: The choice of parallelization strategy should align with hardware usage and task at hand. PyTorch's DDP proved highly effective for GPU parallelism, while CPU parallelism not high and showed fundamental limitations in Python.
2. **Memory Management Trade-offs**: While distributed training reduced per-device memory requirements, the total system memory footprint increased. This represents an important trade-off for large-scale deployments.
3. **Precision-Performance Balance**: Mixed precision training emerged as the most transformative optimization, delivering both speed and accuracy without requiring additional hardware.
4. **Communication Overhead Management**: As parallelism increased, communication between processing units became a limiting factor, highlighting the importance of optimizing data transfer in distributed systems.

These findings demonstrate that effective parallel computing for deep learning requires careful consideration of hardware capabilities, model characteristics, and application requirements. By strategically selecting the appropriate parallelization technique and configuration, substantial performance gains can be achieved without sacrificing model quality.

# 6. References

[1] TMNIST Alphabet (94 characters) (2022)

<https://www.kaggle.com/datasets/nikbearbrown/tmnist-alphabet-94-characters>

[2]PyTorch Docs DDP

<https://pytorch.org/tutorials/intermediate/ddp_tutorial.html>

[3] Use joblib to train an ensemble of small models (Jul 2022)

<https://discuss.pytorch.org/t/use-joblib-to-train-an-ensemble-of-small-models-on-the-same-gpu-in-parallel/157831>

[4]Datasets & DataLoaders (2021, Jan16)

<https://pytorch.org/tutorials/beginner/basics/data_tutorial.html>