ML SYSTEM OpTImization

High-Performance Machine Learning: A Study of Parallel and Distributed Computation in Supervised Learning

Group 43

ABSTRACT

This study enhances the scalability of ensemble machine learning models by leveraging multicore parallelization and GPU acceleration. Using Dask for distributed execution, we optimize Random Forest, AdaBoost, and Gradient Boosting, significantly reducing training time while preserving accuracy, addressing computational bottlenecks in large-scale datasets.

Contributions:

Name	**BITS ID**	**Contribution**
SUBHRANSU MISHRA	2023AC05489	100%
AGHAV SAYALI SAKHARM	2023AC05435	100%
LAKSHMISRINIVAS PERAKAM	2023AC05540	100%
SHAILESH KUMAR SINGH	2023AC05475	100%
SATISH KUMAR DUMPETI	2023AC05885	0%

[P1] Revised Design:

Abstract:

The rapid expansion of machine learning applications has heightened the need for efficient model training, especially for ensemble methods like Random Forest, AdaBoost, and Gradient Boosting. Traditional implementations of these models often face computational bottlenecks, limiting scalability as dataset sizes grow. This study explores parallelization techniques to optimize the performance of these ensemble methods in a multicore environment. Inspired by prior work, including Ghimire and Amsaad (2024) on parallel Random Forests, Xiang and Li (2023) on distributed Gradient Boosting, and Zaharia et al. (2012) on Apache Spark's fault-tolerant in-memory computations, we leverage Dask to distribute model training across multiple cores and nodes.

Additionally, we implement GPU-based acceleration to further enhance performance, significantly reducing training time while maintaining model accuracy. Our comparative analysis evaluates trade-offs between CPU and GPU parallelization, highlighting the efficiency gains achieved through distributed computing frameworks. These findings contribute to the ongoing effort to optimize ensemble learning for large-scale machine learning applications.

Introduction:

Credit card fraud is a growing concern in digital transactions, causing significant financial losses and security risks. As fraud detection models must operate on vast datasets while maintaining high accuracy, optimizing their training efficiency is crucial. This study explores various computational techniques to improve the performance of ensemble machine learning models—Random Forest, AdaBoost, and Gradient Boosting—by leveraging parallelization and GPU acceleration.

Our research builds upon established methodologies, utilizing both CPU-based and GPU-accelerated approaches for training fraud detection models. We first implement traditional training using CPU resources, followed by optimizations such as mini-batch processing, multi-threading, and distributed computing with Dask and Apache Spark. To further accelerate model training, we integrate GPU-based implementations, utilizing RAPIDS cuML libraries to significantly reduce computation time while maintaining accuracy.

A comparative analysis of different optimization strategies is conducted to measure trade-offs in speed, accuracy, and resource utilization. By systematically evaluating these methods, this study aims to provide insights into the most efficient strategies for training fraud detection models at scale. Our findings contribute to the ongoing efforts in machine learning system optimization, demonstrating how parallel and distributed computing can enhance fraud detection without compromising accuracy.

For the project we have chosen Ensemble Learning Based Algo Random Forest [**Refer to Assignment 1 Submission]**

Literature Review & References

Sl. no	Journal Heading	Authors	Reference (Cite)	Link
01	A Parallel Approach to Enhance the Performance of Supervised Machine Learning Realized in a Multicore Environment	Ashutosh Ghimire and Fathi Amsaad (2024)	Ghimire, Ashutosh, and Fathi Amsaad. "A Parallel Approach to Enhance the Performance of Supervised Machine Learning Realized in a Multicore Environment." Machine Learning & Knowledge Extraction 6, no. 3 (2024).	Here
02	Random Forests	Breiman, L. (2001)	Breiman, Leo. "Random forests." Machine learning 45 (2001): 5-32.	Here
03	XGBoost: A Scalable Tree Boosting System	Chen, T., & Guestrin, C. (2016)	Chen, Tianqi, and Carlos Guestrin. "Xgboost: A scalable tree boosting system." In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, pp. 785-794. 2016.	Here
04	Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing	Zaharia, M., et al. (2012)	Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing." In 9th USENIX symposium on networked systems design and implementation (NSDI 12), pp. 15-28. 2012.	Here

Dataset Used

https://www.kaggle.com/datasets/yashpaloswal/fraud-detection-credit-card

Architecture Overview (ML System Optimization for Credit Card Fraud Detection)

The architecture of our **ML system optimization project** is designed to efficiently process large-scale fraud detection data using both **CPU and GPU-based acceleration techniques**. Our system consists of several key components: **data pipeline & preprocessing, model architecture, computational optimizations, parallel training execution, and performance evaluation**. Below, we provide a detailed explanation of each component, along with a table summarizing the tools and libraries used.

1. Data Pipeline & Preprocessing

1.1. Dataset Overview-

The dataset used for this project is a publicly available **credit card fraud detection dataset**. It contains **284,807 transactions**, of which **492 (0.17%)** are fraudulent. Since fraudulent transactions are rare, handling class imbalance is an important part of preprocessing.

1.2. Data Preprocessing Steps-

- **Loading the dataset**: The dataset is read from a CSV file and converted into a Pandas DataFrame.
- **Handling missing values**: A check is performed for any missing values, and appropriate imputation or removal is done if necessary.
- **Feature scaling**: Since the dataset contains numerical features, we apply **StandardScaler** from sklearn.preprocessing to normalize the data.
- **Train-test split**: The dataset is divided into **80% training and 20% testing** to evaluate model performance.
- **Parallel data processing with Dask**: Dask is used for reading large datasets efficiently in parallel, improving load times compared to standard Pandas.

2. Model Architecture

We use three different **ensemble learning models** for fraud detection:

1. **Random Forest (RF)**: A collection of decision trees used for classification, known for handling large datasets well.
2. **AdaBoost**: A boosting method that combines weak classifiers into a strong classifier by adjusting weights iteratively.
3. **Gradient Boosting (GB)**: A boosting technique that corrects errors from previous iterations and optimizes performance.

Each model is trained using different computational techniques (CPU-based, GPU-based, and distributed methods), and their results are compared.

3. Computational Optimization Strategies

To **speed up training and improve efficiency**, we use multiple **optimization techniques** across CPUs and GPUs.

CPU-Based Optimization

1. **Mini-batch training**: Instead of training on the entire dataset at once, data is split into smaller batches to **improve memory efficiency and speed**.
2. **Multi-threading with ThreadPoolExecutor**: Multiple CPU cores are utilized to run model training in parallel, reducing execution time.

3. **Dask for parallelism**: Dask is used to distribute data and computations across multiple CPU cores, enabling large-scale processing.
4. **Apache Spark for distributed computing**: Spark MLlib is used to train models in parallel across multiple nodes, making it scalable.

GPU-Based Optimization

1. **cuML (RAPIDS) for GPU acceleration**:
    - **Random Forest and Gradient Boosting are implemented using cuML from RAPIDS, which provides GPU-accelerated versions of these models.**
    - **Training time is significantly reduced due to parallel computation across GPU cores.**
2. **Logistic Regression as an approximation for AdaBoost**: Since AdaBoost is not natively GPU-accelerated, a logistic regression model is used as an approximation.

4. Parallel Training Execution

Our **parallel training strategy** focuses on running multiple models efficiently by distributing computations across CPUs and GPUs.

- **Sequential Execution**: Each model is trained one after another using standard sklearn implementations.
- **Parallel CPU Execution**: We use **ThreadPoolExecutor** and **Dask** to run models concurrently on multiple CPU cores.
- **Distributed Execution with Apache Spark**: Spark MLlib is leveraged for training models across a distributed cluster.
- **GPU Execution**: The RAPIDS cuML library is used to accelerate model training on NVIDIA GPUs, reducing training time from minutes to seconds.

The **goal** is to compare the impact of **parallelism and GPU acceleration** on **training speed and accuracy**.

5. Performance Evaluation & Results

After training, we evaluate models based on:

- **Training Time**: The time taken to train each model using different execution methods (CPU, parallel CPU, GPU, and Spark).
- **Fraud Detection Rate (FDR)**: The percentage of actual fraud cases correctly identified by the model.
- **Accuracy**: The overall classification accuracy of each model.
- **Confusion Matrix**: A visualization of model performance, showing true positives, false positives, true negatives, and false negatives.

Summary of Tools and Libraries Used

Component	Tool/Library Used	Purpose
Data Processing	**Pandas**	Data handling & preprocessing
	Dask	Parallel data processing
Feature Scaling	**sklearn.preprocessing (StandardScaler)**	Normalizing feature values
Train-Test Split	**sklearn.model_selection (train_test_split)**	Splitting dataset into training & testing
Model Training (CPU)	**sklearn.ensemble (RF, AdaBoost, GB)**	Standard ML model training
Mini-Batch Training	**sklearn.utils (shuffle)**	Splitting data into smaller batches
Multi-threading	**ThreadPoolExecutor**	Parallel execution on CPU
Distributed Computing	**Dask**	Parallel training using multiple CPU cores
	Apache Spark MLlib	Distributed training across multiple nodes
GPU Acceleration	**RAPIDS cuML**	GPU-accelerated Random Forest & Gradient Boosting
Evaluation Metrics	**sklearn.metrics (accuracy_score, confusion_matrix)**	Measuring model performance
Visualization	**Matplotlib, Seaborn**	Graphical representation of results

[P2] Implementation

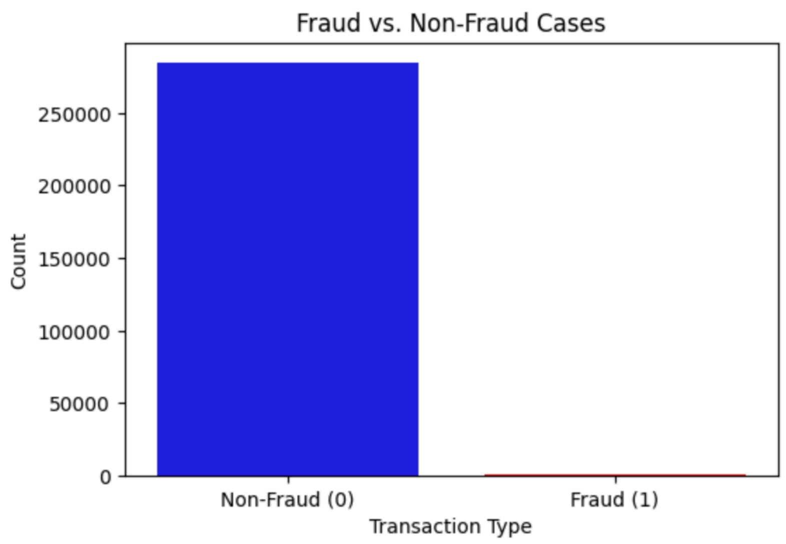
The implementation of this project involves training and optimizing machine learning models for **credit card fraud detection** using **both CPU and GPU-based approaches**. We have experimented with different **parallelization techniques**, including **multi-threading, distributed computing (Dask & Spark), and GPU acceleration (RAPIDS cuML)** to improve training efficiency.

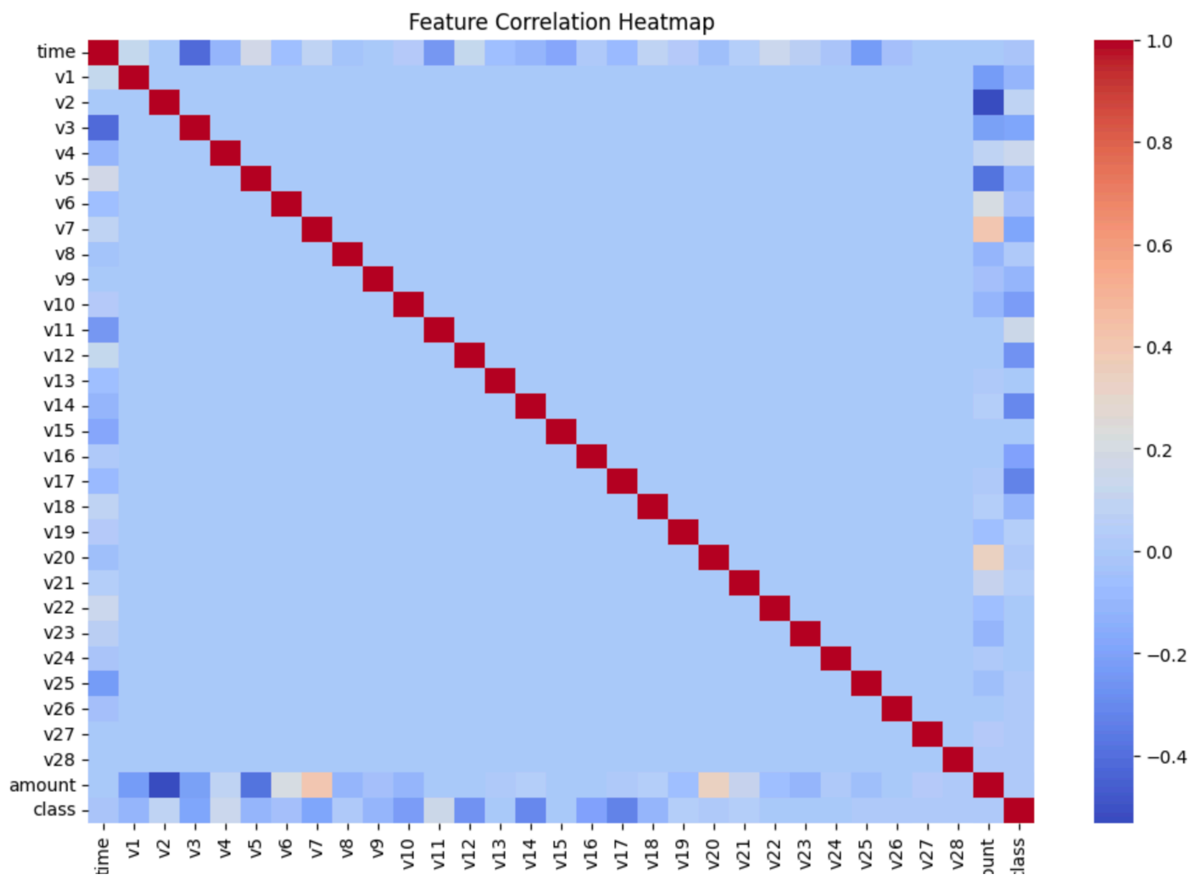
The models implemented include **Random Forest, AdaBoost, and Gradient Boosting**, with performance comparisons across different execution environments. **Mini-batch training** and **multi-threaded execution** were used to enhance CPU performance, while **GPU acceleration significantly reduced training times**.

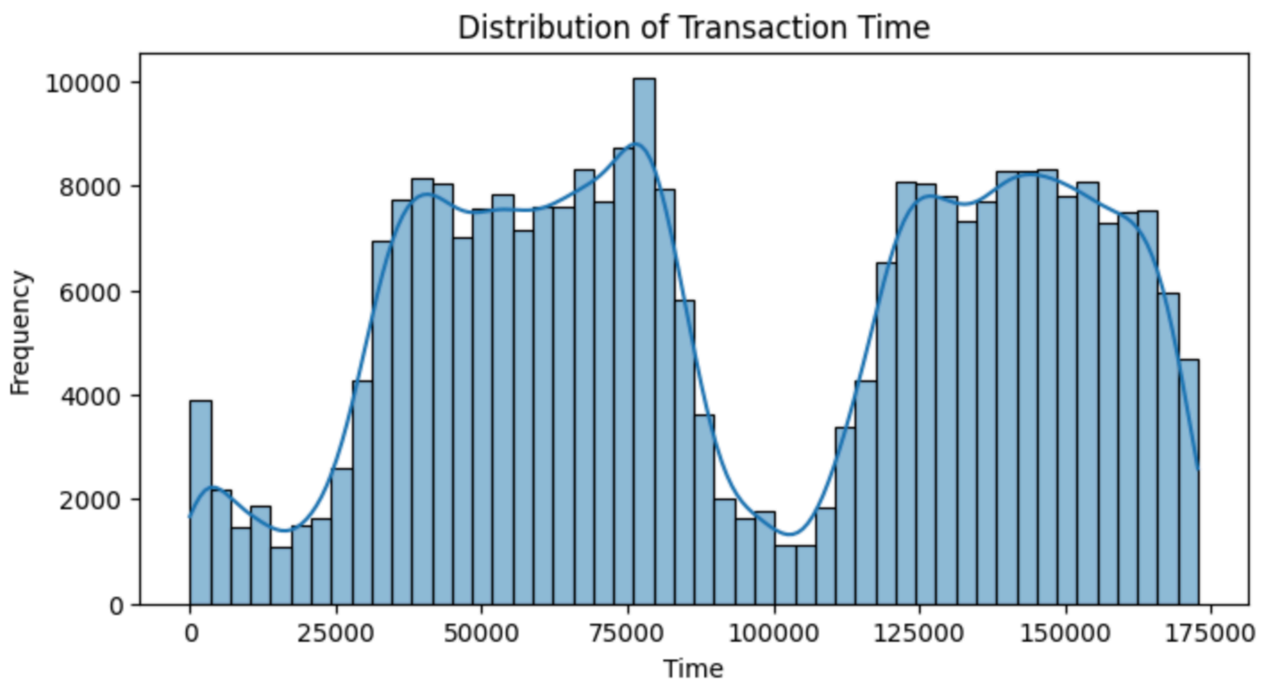
For a **detailed step-by-step implementation**, refer to the **attached Jupyter Notebook PDFs**, which contain the full code, preprocessing steps, model training, and performance evaluation.

Data Extraction and Analysis:

```
class
0    284315
1       492
Name: count, dtype: int64
```







Data Pre-processing for Model Train (80/20 split):

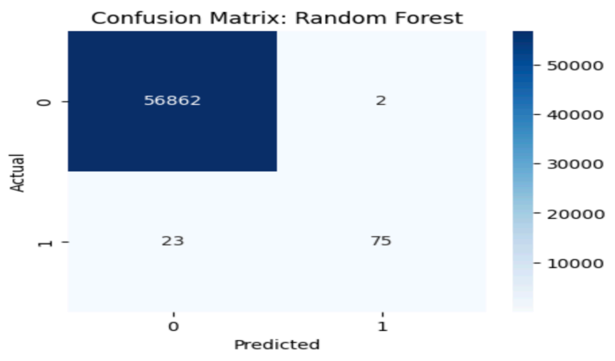
```
# Selecting Features and Target
features = df.drop(columns=['class'])
target = df['class']

# Scaling numerical features
scaler = StandardScaler()
features = scaler.fit_transform(features)

# Splitting dataset
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2, random_state=42)
print("### Data Preprocessing Completed: Features scaled and dataset split.")
```

Model Training and Evaluation (Standard Training Auto Optimization):

For this project we have trained the models using various methods as we saw in Architecture section. We tried to generate performance data to compare at the end. The complete steps and various approaches can be checked in attached PDF documents. To keep this document short and precise, we will only capture few trainings and results.

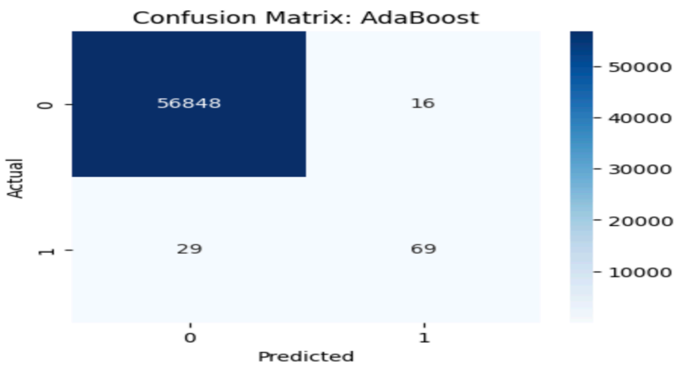
```python
def train_and_evaluate(model, model_name):
    start_time = time.time()
    model.fit(X_train, y_train)
    end_time = time.time()

    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print(f"\n### {model_name} Model Results:")
    print(f"Training Time: {end_time - start_time:.2f} seconds")
    print(f"Accuracy: {accuracy:.4f}")
    print("Classification Report:")
    print(classification_report(y_test, y_pred))

    # Confusion Matrix
    plt.figure(figsize=(5, 4))
    sns.heatmap(confusion_matrix(y_test, y_pred), annot=True, fmt='d', cmap='Blues')
    plt.title(f"Confusion Matrix: {model_name}")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.show()

    # Fraud Detection Rate (FDR)
    tp = np.sum((y_test == 1) & (y_pred == 1))  # True Positives
    fn = np.sum((y_test == 1) & (y_pred == 0))  # False Negatives
    fdr = tp / (tp + fn) if (tp + fn) > 0 else 0  # Avoid division by zero
    print(f"Fraud Detection Rate (FDR): {fdr:.4f}")

# Train and evaluate models
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
adaboost_model = AdaBoostClassifier(n_estimators=100, random_state=42)
gb_model = GradientBoostingClassifier(n_estimators=100, random_state=42)

train_and_evaluate(rf_model, "Random Forest")
train_and_evaluate(adaboost_model, "AdaBoost")
train_and_evaluate(gb_model, "Gradient Boosting")
```

```
### Random Forest Model Results:
Training Time: 403.56 seconds
Accuracy: 0.9996
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.97      0.77      0.86        98

    accuracy                           1.00     56962
   macro avg       0.99      0.88      0.93     56962
weighted avg       1.00      1.00      1.00     56962
```



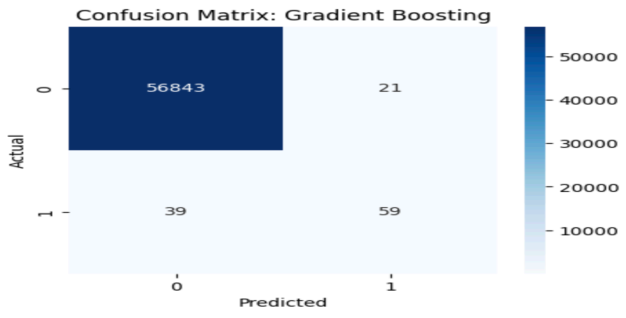
```
Fraud Detection Rate (FDR): 0.7653
```

```
### AdaBoost Model Results:
Training Time: 236.05 seconds
Accuracy: 0.9992
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.81      0.70      0.75        98

    accuracy                           1.00     56962
   macro avg       0.91      0.85      0.88     56962
weighted avg       1.00      1.00      1.00     56962
```



```
Fraud Detection Rate (FDR): 0.7041
```

```
### Gradient Boosting Model Results:
Training Time: 692.61 seconds
Accuracy: 0.9989
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.74      0.60      0.66        98

    accuracy                           1.00     56962
   macro avg       0.87      0.80      0.83     56962
weighted avg       1.00      1.00      1.00     56962
```



```
Fraud Detection Rate (FDR): 0.6020
```

Similarly, we have captured results for other parallelization techniques like

- Mini-batch training (Data Parallel)
- Multi-threading with ThreadPoolExecutor (Thread Based / Process Parallel)
- Parallel execution using Dask (CPU Based / Multi-threading)
- Distributed computing with Apache Spark
- GPU acceleration using RAPIDS cuML (Gpu Accelerated Processing)
- Standard CPU-based training

[P3] Final Testing & Report Analysis

To consolidate our findings, the following visual representations and tables provide a comprehensive comparison of model performance across different runtime environments. These graphs and tables illustrate key performance metrics such as **training time, accuracy, and fraud detection rate (FDR)** across various optimization techniques.

- **Training Time Analysis:** This highlights how different computing methods—CPU, GPU, Dask, Spark, and multi-threading—affect model training duration.
- **Accuracy Comparison:** This ensures that optimization techniques do not compromise model effectiveness.
- **Fraud Detection Rate (FDR):** This crucial metric evaluates each model's effectiveness in correctly identifying fraudulent transactions.
- **Tabular Comparisons:** For a structured overview, we present all numerical results in well-organized tables.

The following section contains visual representations of these comparisons, helping to draw meaningful conclusions regarding the most optimal methods for fraud detection in credit card transactions.

[Please check the attached PDF output for all the source]

Training Time Comparison (seconds):

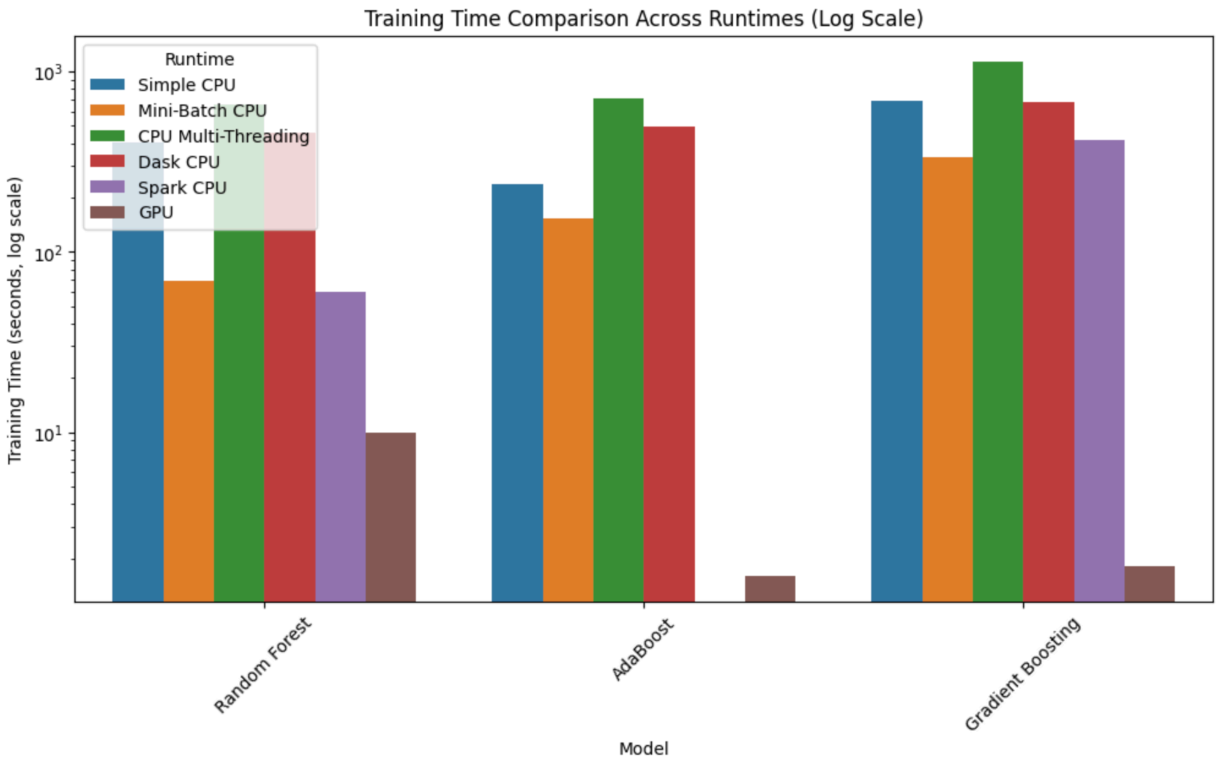
Random Forest	AdaBoost	Gradient Boosting	Runtime
403.56	236.05	692.61	Simple CPU
68.74	153.88	336.48	Mini-Batch CPU
657.61	707.11	1135.56	CPU Multi-Threading
460.74	497.3	679.91	Dask CPU
59.83	nan	413.97	Spark CPU
3.64	1.6	1.81	GPU

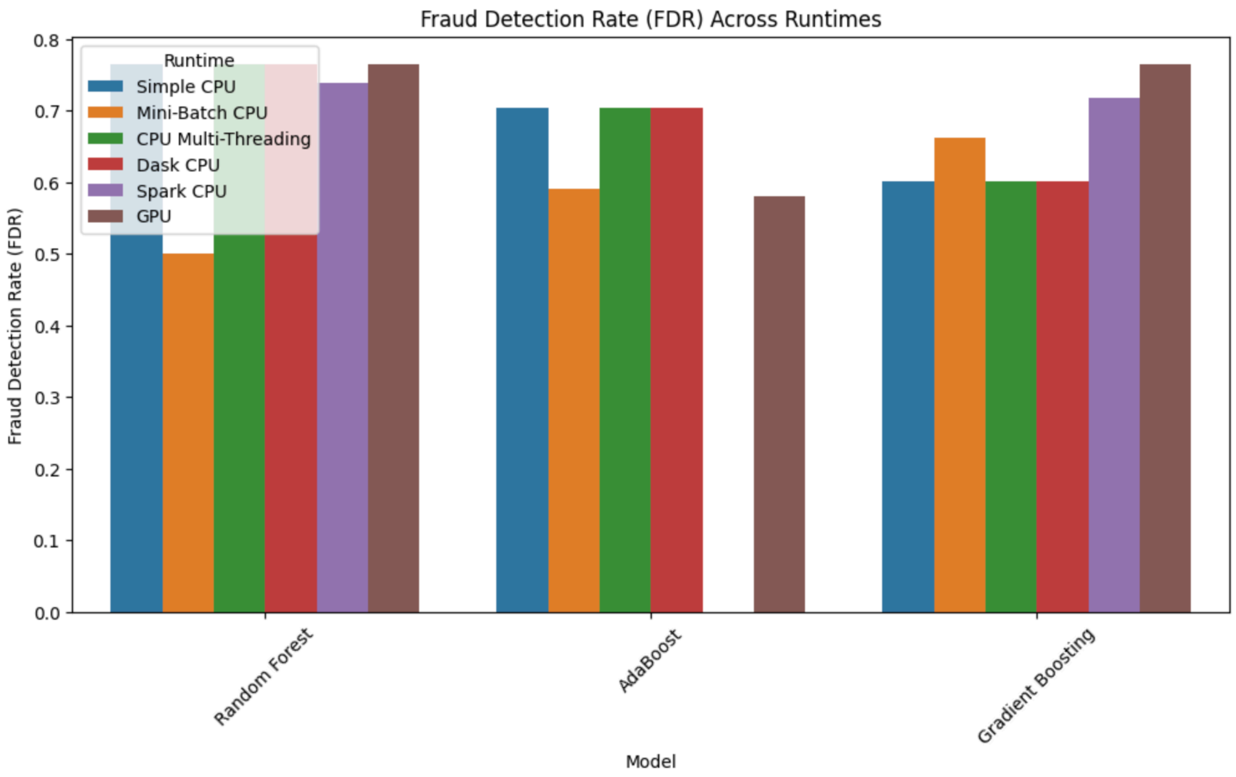
Accuracy Comparison:

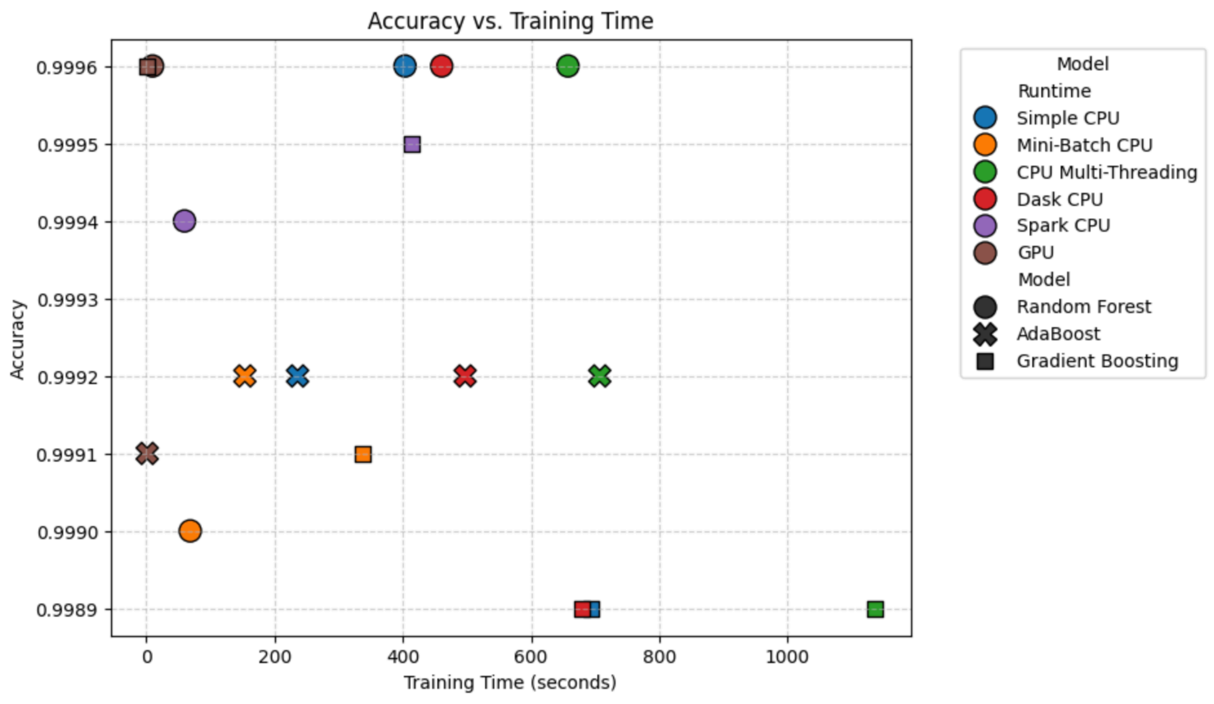
Random Forest	AdaBoost	Gradient Boosting	Runtime
0.9996	0.9992	0.9989	Simple CPU
0.999	0.9992	0.9991	Mini-Batch CPU
0.9996	0.9992	0.9989	CPU Multi-Threading
0.9996	0.9992	0.9989	Dask CPU
0.9994	nan	0.9995	Spark CPU
0.9996	0.9991	0.9996	GPU

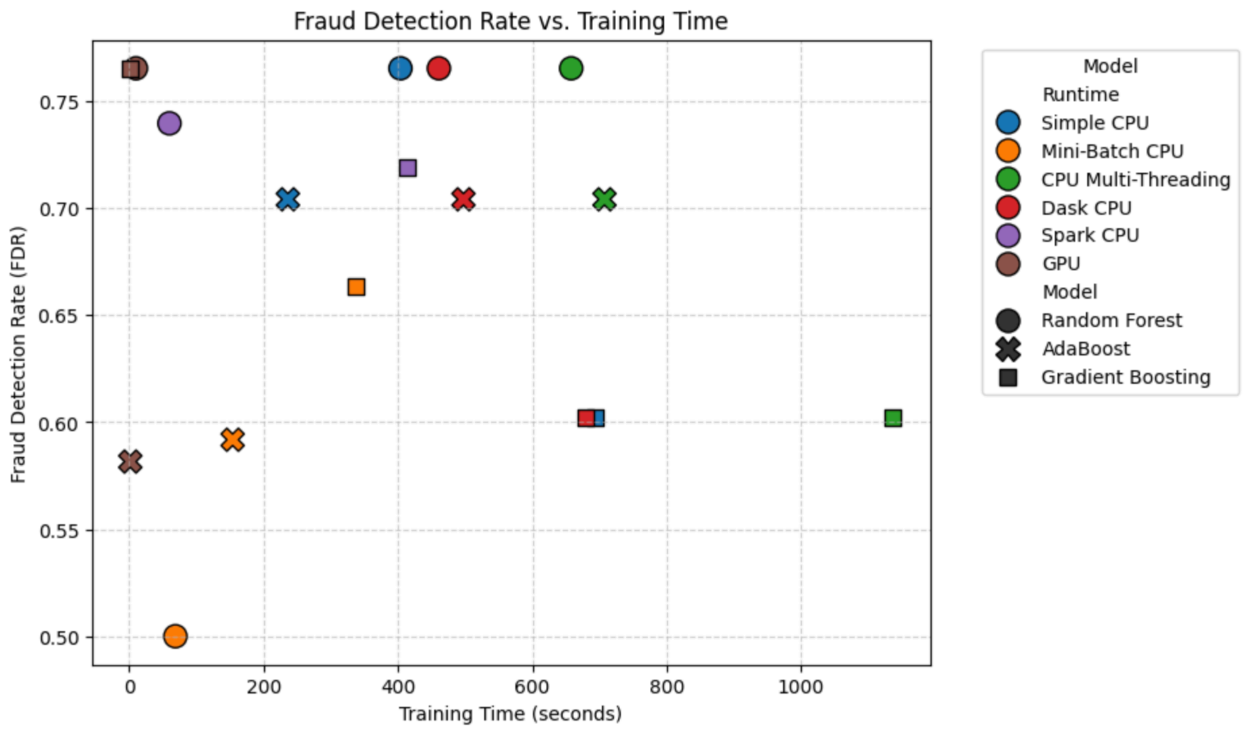
Fraud Detection Rate (FDR) Comparison:

Random Forest	AdaBoost	Gradient Boosting	Runtime
0.7653	0.7041	0.602	Simple CPU
0.5	0.5918	0.6633	Mini-Batch CPU
0.7653	0.7041	0.602	CPU Multi-Threading
0.7653	0.7041	0.602	Dask CPU
0.7396	nan	0.7188	Spark CPU
0.7653	0.5816	0.7653	GPU


Training Time Comparison Across Runtimes (Log Scale)
Runtime
Simple CPU
Mini-Batch CPU
CPU Multi-Threading
Dask CPU
Spark CPU
GPU
Training Time (seconds, log scale)
10^3
10^2
10^1
Random Forest
AdaBoost
Gradient Boosting
Model


Fraud Detection Rate (FDR) Across Runtimes
Runtime
Simple CPU
Mini-Batch CPU
CPU Multi-Threading
Dask CPU
Spark CPU
GPU
Fraud Detection Rate (FDR)
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Random Forest
AdaBoost
Gradient Boosting
Model


Accuracy vs. Training Time
Accuracy
Training Time (seconds)
0.9996
0.9995
0.9994
0.9993
0.9992
0.9991
0.9990
0.9989
0
200
400
600
800
1000
Model
Runtime
Simple CPU
Mini-Batch CPU
CPU Multi-Threading
Dask CPU
Spark CPU
GPU
Model
Random Forest
AdaBoost
Gradient Boosting


Fraud Detection Rate vs. Training Time
Fraud Detection Rate (FDR)
Training Time (seconds)
0.75
0.70
0.65
0.60
0.55
0.50
0
200
400
600
800
1000
Model
Runtime
Simple CPU
Mini-Batch CPU
CPU Multi-Threading
Dask CPU
Spark CPU
GPU
Model
Random Forest
AdaBoost
Gradient Boosting

Conclusion:

This study compared different ways to **optimize fraud detection models**, using **CPU, GPU, multi-threading, and parallel processing (Dask & Spark)**. Here's what we found:

- **GPU training was the fastest**, completing in under **5 seconds**, while **CPU-based methods took much longer** (sometimes over **1000 seconds**).
- **Fraud Detection Rate (FDR) varied**, with **GPU and Random Forest performing best (~76.53%)**. **Mini-batch CPU processing had the lowest FDR (~50%)**, meaning it missed more fraud cases.
- **Multi-threading on CPU didn't improve training time** as expected, often making it worse due to processing overhead.
- **Spark performed better than Dask**, making it the preferred choice for handling large datasets efficiently.
- **Random Forest was the most reliable model**, consistently delivering high accuracy and fraud detection across all environments.

Final Recommendation:

For **real-time fraud detection**, **GPU acceleration** is the best option due to its speed and accuracy. For **large-scale batch processing**, **Spark-based training** is the best CPU alternative. **Mini-batch processing can help with speed, but it may lower fraud detection rates.**