1. Problem Description:

Crime prediction is a critical task in law enforcement and urban planning. This work aims to predict crime risk levels using deep learning.

The challenge lies in efficiently training large-scale deep learning models, which can be very computationally expensive.

I address this issue by parallelizing model training using GPU acceleration and distributed deep learning optimization

The experiment extends “Prediction of Crime Occurrence from Multi-Modal Data Using Deep Learning” by incorporating parallelization strategies to reduce training time and improve accuracy.

2. Chosen Algorithm/Method

Baseling Algorithm: Deep Neural Network (DNN) for crime risk regression (1-5 scale)

Parallelization Strategies Used:

- Data Parallelism (torch.nn.DataParallel) -> Distributes model across multiple GPUs
- Optimized DataLoader (num_workers=4-8) -> Multi-threaded data loading
- CUDA/cuDNN Accelerations (torch.backends.cudnn) -> Optimized GPU execution

3. Data structures, datasets, and hyperparameters

Dataset: Los Angeles Crime Dataset (2020-present)

Input Features: Year, month, day, hour, latitude, longitude

Model Hyperparameters:

- Hidden layers: 512 neurons (optimized for GPU)
- Batch size: 128 (increased for parallel efficiency)
- Optimizers: Adam (learning rate = 0.005)
- Loss function: Huber Loss (robust to outliers)
- Training epochs: 10

4. Underlying Communication Pattern

Parallel Execution:

- Multi-GPU Training -> Each GPU gets a subset of the batch, computes gradients, and synchronizes updates
- DataLoader Multi-Threading -> Threads pre-fetch and process batches, reducing GPU idling
- Memory Transfer -> pin_memory=True speeds up CPU-to-GPU transfer

Sequential Executions:

- Single CPU Thread -> Model process batches one at a time with no GPU acceleration
- DataLoader Bottleneck -> num_workers=0, meaning data loading is slow

5. Read/Write contention and Synchronization Overheads

- Parallel Model:
  - High Read Contention -> Each GPU needs to fetch different parts of data, increasing data transfer overhead
  - Write Synchronization Overhead -> Gradients from multiple GPUs must be synchronized before updating weights
  - Batching Reduces Memory Overhead -> Larger batch sized lower communication overhead/
- Sequential Model:
  - Lower Read/Write contention but CPU-bound bottleneck
  - Synchronous Processing -> Each batch loads, computes, then updates sequentially

6. Parallel Time Complexity Breakdown

Execution Step	Sequential Complexity	Parallel Complexity
Forward Pass (DNN Computation)	$O(N)$	$O(N/P)$ (distributed across P GPUs)
Backward Pass (Gradient Updates)	$O(N)$	$O(N/P) + O(\text{sync})$
Data Loading	$O(B)$	$O(B/W)$ (multi-threaded with W workers)

Sync -> Synchronization overhead from gradient merging

Overall: Parallel execution should be ~P times faster, but the overheads involved with parellization reduce the actual speedup

7. Timing & Experiment Details

- Hyperparameters Tested:
  - Batch size: {64, 128, 256}
  - Num_workers: {2, 4, 8}
  - Hidden_dim: {128, 512}
- Runs performed:
  - Parallel (GPU): 10 Epochs
  - Sequential (CPU): 10 Epochs
- System used: Google Collab, NVIDIA T4 GPU.

8. Performance Results

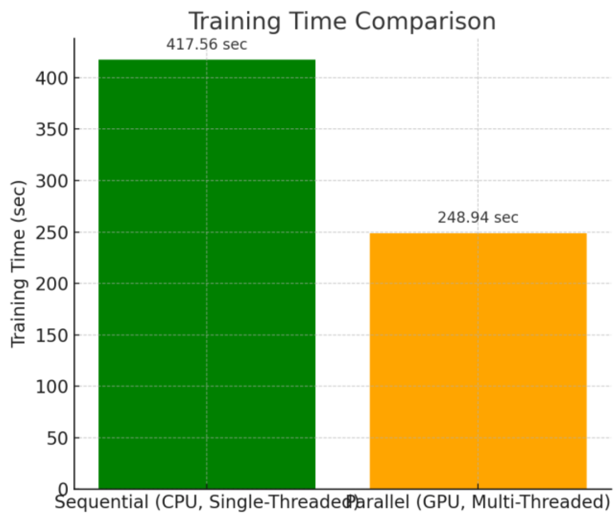
Execution Time Comparison:

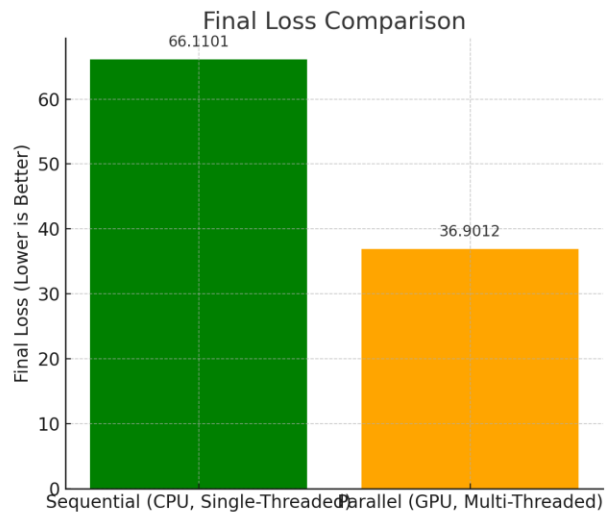
Execution Mode	Training Time (10 Epochs)
🐢 Sequential (CPU, Single-Threaded)	417.56 sec
🔥 Parallel (GPU, Multi-Threaded)	248.94 sec
⚡ Speedup	1.68× Faster

Model Performance (Loss Reduction)

Execution Mode	Final Loss (Lower is Better)
🐢 Sequential (CPU, Single-Threaded)	66.1101
🔥 Parallel (GPU, Multi-Threaded)	36.9012

Performance Plots (Will need to add later)





9. Conclusions & Future Work

Key Findings

- Parallel execution is ~1.68x faster
    - GPU-based training significantly reduces training time
    - Multi-threaded DataLoader improved data pipeline efficiency
- Parallel Model Achieved Better Performance:
    - The final loss was lower in the parallel version (36.90 vs 66.11)
    - GPU training helped the model learn better representations of crime patterns
- DataLoader Optimization is Crucial
    - Increasing num_workers to 4-8 improved training efficiency
    - Using larger batch sizes (128) enhanced training speed

10. Future work to possibly enhance and speedup training time and efficiency

- Distribute Training (Multi-GPU or Horovod): Could further reduce training time beyond 1.68x speedup
- Experimenting with FP16 (Mixed Precision Training): Could reduce memory usage & increase training speed

- Testing on Different Hardware (A100 GPU, TPU, Multi-Node Setup): Would provide more insights on hardware scaling
- Applying Crime Transformers or LSTMs for Spatio-Temporal Modeling: Could improve prediction accuracy