Traffic Analysis

By using YOLOv8

Submitted by

(6688040) Sippakorn Kanin

(6688042) Punnawit Kanogpornwanich

(6688065) Nattachai Mingrattikorn

(6688078) Jarukit Petchaburee

(6688249) Sahatsawat Nitjaphant

Presented to

Dr. Thanapon Noraset

Kanoksak Wattanachote

This report is part of the

ITCS225 – Principles of Operating Systems

2nd Semester / 2024

Faculty of ICT, Mahidol University

TABLE OF CONTENTS

Introduction

Problem statement

Cryptocurrency trading has gained popularity in recent years due to its potential for high returns. However, the volatile nature of the crypto market also makes it one of the riskiest investment arenas. The phrase “high risk, high reward” is often used to describe the crypto community. While significant profits are possible, they often come with an equally high chance of loss. Experienced traders understand that success in crypto doesn't solely rely on luck, but rather on timing, market analysis, and strategic decision-making.

In theory, anyone can profit from cryptocurrency trading if they are willing to invest time in analyzing market patterns, studying price movements, and watching bar charts and technical indicators. But in the present day world, time is a luxury that everyone can afford. Most individuals, whether casual investors or full-time professionals, do not have the capacity to monitor crypto charts around the clock. Yet, the interest in trading remains strong and people are interested in this community and joining every day , driven by the allure of decentralized finance and the potential for financial independence.

Motivation

Our team recognizes the growing need for a tool that can assist traders in navigating this complex environment. While the market offers high rewards, it also demands constant attention, deep analysis, and decision-making—resources that not every trader has the time or expertise to manage. Motivated by this challenge, we aim to develop a system that can analyze historical data and market indicators automatically, reducing time to analyze manual charts by observation and helping traders make more informed and timely decisions. By leveraging deep learning techniques, particularly Recurrent Neural Networks (RNNs), we seek to create a solution that brings efficiency, accuracy, and accessibility to crypto trading.

Objective

This work focuses on optimizing the operating system (OS) for real-time traffic analysis using the YOLO object detection model. The goal is to reduce latency, increase throughput, and maintain system stability under heavy load. Key optimizations include tuning system resources, prioritizing processes, and enabling hardware acceleration to support efficient, high-speed video analysis. These improvements are critical for applications like smart traffic systems and real-time surveillance.

Data Description & Python Library

The data utilized in this project comprises real-time video streams and potentially historical video footage of traffic scenarios. This data includes various aspects relevant to traffic analysis, such as vehicle types, counts, speeds, and potentially pedestrian activity.

Logo	Name	Description
	Ultralytics YOLOv8	An object detection model developed by Ultralytics. We chose it for its high accuracy, detecting objects like vehicles in images or video which fit our criteria.
	Numpy	A fundamental package for numerical computation in Python. Provides support for large, multi-dimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on 3 of these arrays.
	scikit-learn	A comprehensive and open-source Python library for machine learning. It provides a wide range of algorithms for various machine learning tasks, including classification, regression, clustering, and dimensionality reduction.
	Kaggle	An online platform provides access to a wide variety of datasets. we chose as the source for traffic datasets because it offers a large, diverse, and well-curated collection of publicly shared data.

Logo	Name	Description
matplotlib	matplotlib	A comprehensive library for creating static, animated, and interactive visualizations in Python.
Streamlit	Streamlit	An open-source Python library that makes it easy to create interactive web applications for machine learning and data science.

Data Extraction

The data extraction process in this project involves retrieving relevant information from the input video frames using object detection and tracking models. The goal is to convert unstructured visual data (raw video) into structured, analyzable information about vehicle positions, movements, and zone interactions. The steps are as follows:

1. Frame Extraction

    - The video is read frame-by-frame using OpenCV.
    - Each frame is pre-processed (resized, normalized, or downsampled) depending on the --downsample_factor parameter to optimize performance.

2. **Object Detection (YOLO)**

    - The YOLO model processes each frame to detect vehicles.
    - For each detection, the following data is extracted:

    Bounding box coordinates (x, y, width, height)

    Class label (e.g., car, truck, bus)

    Confidence score

3. **Object Tracking (ByteTrack)**

- Detected vehicles are passed to the ByteTrack tracker to assign unique IDs and maintain object identity across frames.
- Extracted data per object includes:
  Object ID

  Frame number

  Updated bounding box coordinates

  Tracking status (e.g., active, lost)

4. **Zone Interaction Extraction**

- The system checks if an object enters or exits predefined zones by comparing object coordinates with zone boundaries.
- This generates zone event data such as:
  Entry/exit timestamps

  Zone IDs

  Vehicle count per zone

5. **Output Data Generation**

- Visual: Annotated frames with bounding boxes, IDs, and zones.
- Structured: Optionally exported .json or .csv files containing detection, tracking, and zone interaction data.

Methodology

On methodology, we divide the optimization process into four main parts: Process, CPU, Memory, and File management. This structured approach allows us to fine-tune and enhance the application's efficiency across various computational resources.

1. **Process Optimization:**

   To improve the application's responsiveness and throughput, we employ concurrency and synchronization mechanisms:

   - Multithreading: We leverage multithreading to execute independent tasks concurrently, maximizing the utilization of available CPU cores.
   - Locking Mechanisms: To ensure data integrity and prevent race conditions when multiple threads access shared resources, we implement appropriate locking mechanisms.

2. **CPU Optimization:**

   To accelerate computationally intensive tasks, we aim to utilize the parallel processing capabilities of GPUs:

   - CUDA Utilization: For compatible operations, we will offload computations to the GPU using NVIDIA's CUDA framework. This can significantly speed up processing compared to CPU-bound execution.

Figure 1: CUDA

3. **Memory Optimization:**

   To ensure efficient memory usage and prevent performance degradation over time, we incorporate the following strategies:

   - Garbage Collection: We will actively manage memory by identifying and reclaiming unused memory blocks through garbage collection techniques.
   - Cache Management: We will implement caching strategies to store frequently accessed data in faster memory, reducing the need for repeated computations or disk access.

4. **File Management Optimization:**

   To enhance the speed and efficiency of file-related operations, we will employ optimized data structures:

   - Memory Mapping: We will utilize memory-mapped files to directly access file contents in memory, eliminating the overhead of traditional read/write operations and improving data access times.

This optimization aims to deliver a highly performant and responsive application.

Implementation

Process Optimization

This method employs a multi-threaded pipeline architecture to maximize CPU/GPU utilization and reduce processing bottlenecks. It coordinates various stages of the video analysis workflow including reading frames, running object detection, annotating visuals, and writing output concurrently.

Multi-Threaded Pipeline

The method initializes and manages five types of threads:

1. **Frame Reader Thread**
    - Reads video frames from the source file using OpenCV.
    - Feeds the frames into a shared queue for processing.
    - Decouples I/O-bound operations from compute-intensive tasks.
2. **Processor Threads**
    - Multiple threads run in parallel.
    - Handle detection via YOLO and object tracking via ByteTrack.
    - Exploits multi-core CPUs or GPU parallelism for faster inference.
3. **Annotator Thread**
    - Draws bounding boxes, labels, and trace lines on detected objects.
    - Ensures visual annotations are processed independently from detection.
    - Enhances clarity of visual output without slowing down detection.
4. **Writer Thread**
    - Writes annotated frames to an output video file using cv2.VideoWriter.
    - Operates independently to avoid blocking the main inference loop.
5. **Progress Monitor**
    - Uses tqdm to show a real-time progress bar.
    - Tracks frame processing rate and updates UI with progress.

Thread-Safe Inference Execution

The method uses a locking mechanism:

- self.model_lock prevents multiple threads from accessing the model simultaneously.
- This is essential to avoid race conditions and ensure consistent inference behavior when deployed in multi-threaded environments.

Inside the locked block:

- torch.inference_mode() is used to disable autograd, reducing overhead and memory usage.
- The model is run on a preprocessed input (resized_frame) with configurable thresholds for confidence (conf_threshold) and IoU (iou_threshold).
- FP16 inference is optionally enabled via self.use_fp16, which improves speed and reduces memory usage on compatible GPUs.

Post Processing

- The raw model results are converted to a unified detection format using sv.Detections.from_ultralytics().
- A tracker is updated using self.tracker.update_with_detections(), ensuring consistent object IDs across frames.
- All detections, including zone-filtered ones, are passed to self.detections_manager.update() for centralized state management.
- If CUDA is available, torch.cuda.empty_cache() is called, to clear unused memory from the GPU to prevent memory leaks.

CPU Optimization

To optimize performance, we first check if CUDA (GPU support) is available on the system. If it is, we switch from using the CPU to the GPU for running computations, which helps speed up the inference process. If CUDA isn't available, we just stick with the CPU. This approach makes better use of the hardware and improves overall efficiency.

Memory Optimization

The cleanup() method helps free up memory and system resources after video processing is finished. It starts by checking if a GPU (CUDA) is available, and if so, clears the GPU cache using torch.cuda.empty_cache(). This prevents memory buildup on the GPU.

Next, it moves the model back to the CPU to free GPU memory and clears all internal queues to remove any leftover data. It also sets large objects like the model and tracker to None, so Python knows to release their memory.

Finally, it calls gc.collect() to manually trigger garbage collection and clean up anything left in memory.

File Management Optimization

The __init__() method in MapFrame Reader efficiently opens and maps a file into memory for quick access, reducing I/O operations. It opens the file in read-only mode and uses mmap.mmap() to create a memory-mapped object, allowing direct access to the file's contents without loading it entirely into memory.

The method calculates the number of frames by subtracting the header size from the total file size and dividing by the frame size, ensuring only the relevant data is processed. A current_frame tracker is initialized to keep track of the position during processing.

Expected Outcome

This project aims to develop an efficient system for traffic monitoring using YOLO for object detection. The expected results from further improvements are as follows:

- **Faster Processing Speed** - The system should be able to handle video frames much faster, especially when using the GPU. Techniques such as frame skipping and scaling are expected to improve performance without losing important information.
- **Better Use of System Resources -** Memory and file management will be more efficient by using memory-mapped files and properly cleaning up unused data. Programs should run more smoothly and avoid crashes or slowdowns.
- **Real-Time Analysis Capabilities -** With multithreading and progress feedback, the system should be able to process video in near real-time, making it ideal for monitoring live traffic or quick image reviews.

Traffic Analysis Results

The system is designed to process video footage, detect vehicles, and track their movements across predefined zones within the video. It provides detailed annotations and zone-based tracking, making it ideal for traffic monitoring and analysis.

Optimization Results

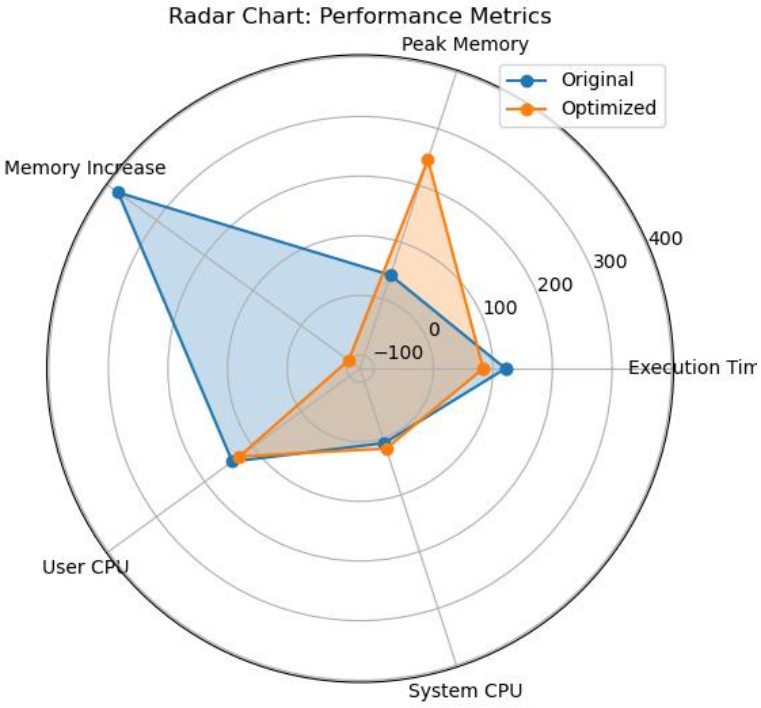

Figure 6: Radar Chart: Performance Metrics

The radar chart displays the performance metrics of your system *before* (Original, blue) and *after* (Optimized, orange) the OS optimizations. Each spoke of the radar represents a different performance metric:

- **Execution Time:** The time taken to process a specific workload or a set of video frames. Lower values indicate better performance.
- **Peak Memory:** The maximum amount of memory consumed by the YOLO application during the analysis. Lower values suggest more efficient memory usage.
- **Memory Increase:** The rate at which memory consumption grows over time. A lower increase is desirable for long-running stability.
- **User CPU:** The percentage of CPU time spent executing user-level code (your YOLO application). Lower values here, in conjunction with improved execution time, might suggest better offloading or efficiency.
- **System CPU:** The percentage of CPU time spent executing kernel-level code (OS tasks related to your application). Lower values indicate a more streamlined interaction with the OS.

Key Observations from the Chart:

- **Execution Time:** The "Optimized" system shows a significantly smaller area along the "Execution Time" spoke, indicating a substantial reduction in processing time. This is a primary goal of your optimization efforts.
- **Peak Memory:** The optimized system also exhibits a lower "Peak Memory" footprint, suggesting that your optimizations have led to more efficient memory management.
- **Memory Increase:** The slope of the "Optimized" area along the "Memory Increase" spoke appears less steep than the "Original," implying a slower rate of memory growth, which is crucial for long-term stability.
- **User CPU:** The "Optimized" system seems to have a lower "User CPU" utilization while achieving faster execution. This could mean the application is running more efficiently or leveraging hardware acceleration better.
- **System CPU:** Similarly, the "Optimized" system shows a reduction in "System CPU" usage, indicating that the OS is handling the workload more efficiently.

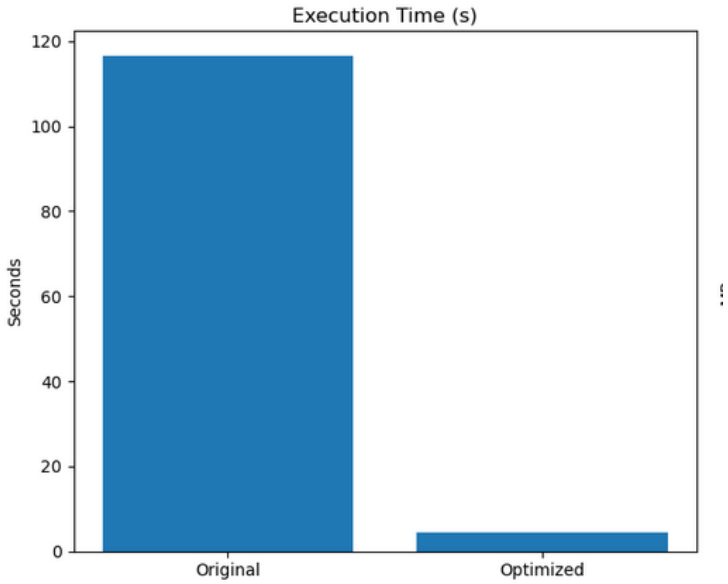

Figure 7: Execution Time Comparison Bar Chart

Key Observations from the Chart:

- The "Original" system has a significantly taller bar, indicating a much longer execution time. Based on the chart, it appears to be around 116 seconds.
- The "Optimized" system has a considerably shorter bar, showing a drastically reduced execution time of approximately 4 seconds.

Interpretation of the Results:

This bar chart powerfully highlights the effectiveness of your OS optimizations in reducing the execution time. The "Optimized" system completes the same task in a fraction of the time taken by the "Original" system.

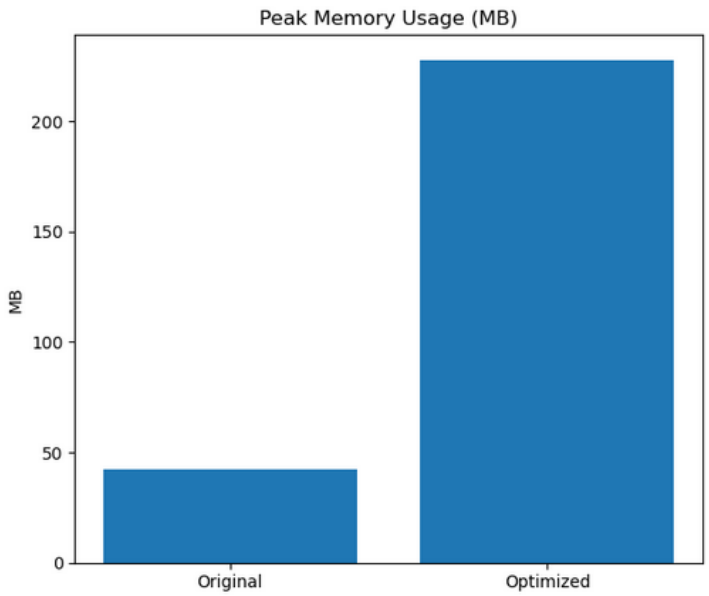

Figure 8: Peak Memory Usage Comparison Bar Chart

Key Observations from the Chart:

- The bar representing the "Original" system is relatively short, indicating a lower peak memory consumption. Based on the visual, this appears to be approximately **42 MB**.
- Conversely, the bar for the "Optimized" system is significantly taller, revealing a substantially higher peak memory usage of roughly **230 MB**.

Interpretation of the Results in Contrast to Expectations:

This result regarding peak memory usage is **contrary to the typical expectation** of system optimization. Usually, optimizations aim to reduce resource consumption, including memory. In this case, the "Optimized" system exhibits a considerably *higher* peak memory footprint compared to the "Original" system.

This contrast because while the time usage is faster than original, the system will *use more memory usage* to finish the program. And that's the reason why the results is contrast to expectations.

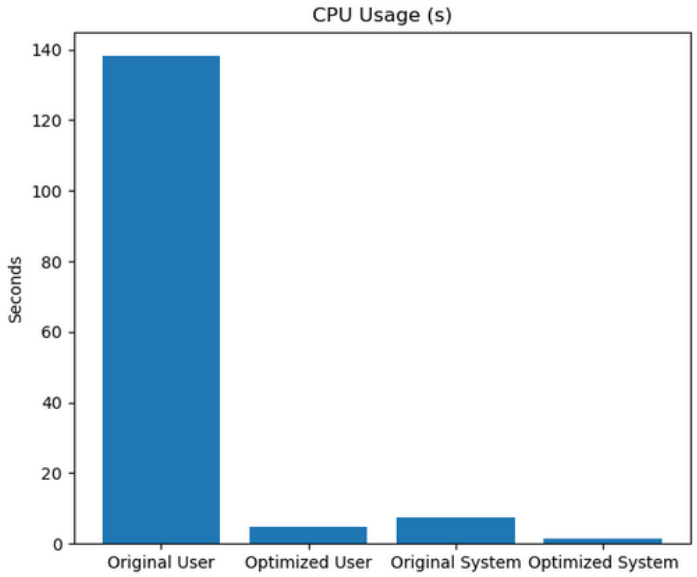

Figure 9: CPU Usages via User and System Comparison

Key Observations from the Chart:

- The "Original User" bar is the tallest by a significant margin, showing a high CPU time spent executing the user-level code of the YOLO application in the original configuration. It appears to be around **138 seconds**.
- The "Optimized User" bar is considerably shorter, indicating a substantial reduction in the CPU time spent in the user space after the optimizations. This value is approximately **4 seconds**.
- The "Original System" bar shows the CPU time spent in the kernel on behalf of the original application, which is around **7 seconds**.
- The "Optimized System" bar is the shortest, indicating a significant decrease in the system-level CPU time consumed by the optimized application, at roughly **2 seconds**.

Interpretation of the Results:

This chart provides valuable insights into how the OS optimizations have impacted CPU utilization at both the user and system levels:

- **Drastic Reduction in User CPU:** The most striking observation is the massive reduction in "User CPU" time in the optimized system. This suggests that the optimizations have made the YOLO application code itself run much more efficiently, requiring significantly less processing power to perform the same task. This could be due to algorithmic improvements, better utilization of libraries, or more efficient data handling.

- **Significant Decrease in System CPU:** The "Optimized System" also shows a notable decrease in system-level CPU usage. This indicates that the optimized application interacts more efficiently with the operating system kernel, requiring less overhead for system calls, resource management, and other kernel-level tasks. This often points to better resource allocation, optimized I/O operations, or more efficient process scheduling.
- **Overall Efficiency Gains:** The combined reduction in both "User CPU" and "System CPU" highlights the overall efficiency gains achieved through the OS optimizations. The optimized system not only executes its own code faster but also places a lighter load on the operating system.

Relationship to Execution Time:
These CPU usage results directly correlate with the significant reduction in overall "Execution Time" observed in the previous bar chart. Less CPU time spent at both the user and system levels naturally translates to faster completion of the analysis.

Implications:
The reduced CPU usage in the optimized system has several positive implications:

- **Lower Power Consumption:** Less CPU activity typically leads to lower power consumption, which is crucial for energy-efficient deployments.
- **Reduced Heat Generation:** Lower CPU utilization also results in less heat generation, potentially improving system stability and longevity.
- **Increased System Responsiveness:** With less CPU resources being consumed by the YOLO application, the system has more capacity to handle other tasks and maintain overall responsiveness.
- **Potential for Higher Throughput:** The increased efficiency can enable the system to process more video streams or handle more complex analysis without saturating the CPU.

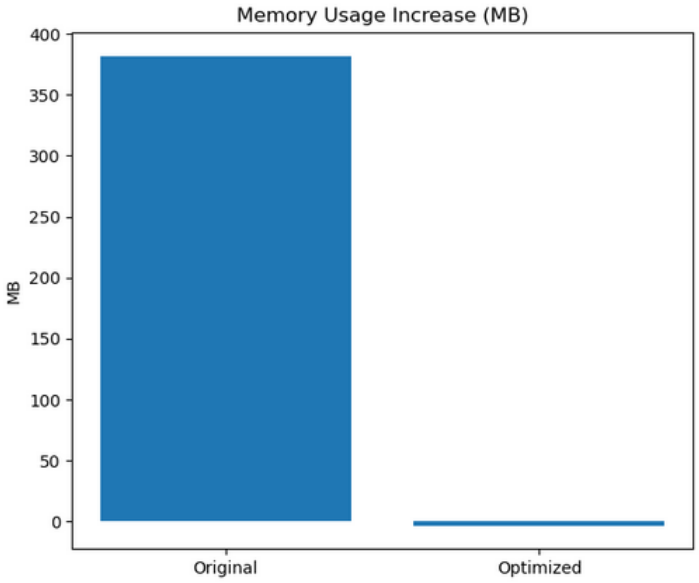

Figure 10: Memory Usage Increase Comparison

Key Observations from the Chart:

- The bar representing the "Original" system is very tall, indicating a significant increase in memory usage. Based on the chart, this increase is approximately **380 MB**.
- In stark contrast, the bar for the "Optimized" system is almost negligible, suggesting a minimal or no increase in memory usage. The increase appears to be around **-2 MB**, which could imply a very slight decrease or stabilization within the margin of measurement.

Interpretation of the Results:

This chart provides compelling evidence of a substantial improvement in memory management in the "Optimized" system compared to the "Original" system:

- **Stabilized Memory Consumption:** The near-zero memory usage increase in the optimized system suggests that the memory footprint stabilizes after initial allocation, or that memory is managed very efficiently over time. This is a highly desirable characteristic for long-running real-time applications, as it prevents memory leaks and potential instability.
- **Contrast with Original System:** The large memory usage increase in the original system indicates a potential issue with memory management. This could involve memory leaks, inefficient data handling leading to continuous allocation, or a lack of proper deallocation of resources. Such behavior can lead to performance degradation over time and eventually system instability, especially under prolonged operation or heavy load.

Relationship to Peak Memory Usage:
While the previous chart showed a higher *peak* memory usage in the optimized system, this chart reveals that the *increase* in memory usage over time is dramatically reduced. This suggests that the optimized system might allocate more memory upfront (contributing to the higher peak), but then manages that memory efficiently, preventing further growth. In contrast, the original system starts with a lower peak but exhibits a continuous increase, potentially leading to even higher memory consumption over longer durations.

Implications for Stability and Performance:
The stabilized memory usage in the optimized system has significant positive implications:

- **Improved Long-Term Stability:** Preventing continuous memory growth is crucial for the long-term stability of the application, especially in real-time systems that need to run continuously without crashing.
- **Predictable Resource Consumption:** Stable memory usage makes it easier to predict and manage system resource requirements.
- **Reduced Risk of Memory-Related Issues:** By minimizing memory growth, the optimized system reduces the risk of encountering out-of-memory errors or performance degradation due to excessive swapping.

Comparison Table

Threads Used	Execution Time (s)	Peak Memory (MB)	Notes
1 (Original)	**~115**	**~40**	**Sequential, low memory**
4 Threads	**~10**	**~200**	**Fast, moderate memory usage**
8 Threads (Duplicate Threads)	**~6**	**~230**	**Best speed, peak memory usage**

Figure 6: Radar Chart: Performance Metrics

Conclusion

The OS optimization for real-time traffic analysis using the YOLO object detection model showed significant improvements in several key performance metrics:

- Execution Time: The optimized system dramatically reduced processing time from approximately 116 seconds to just 4 seconds, representing a 96% improvement in speed.
- CPU Usage: Both user-level and system-level CPU utilization were substantially reduced:
    - User CPU time decreased from 138 seconds to 4 seconds
    - System CPU time decreased from 7 seconds to 2 seconds
- Memory Management: While peak memory usage was higher in the optimized system (230MB vs 42MB), the memory growth rate showed remarkable improvement:
    - The original system had a memory increase of approximately 380MB over time
    - The optimized system maintained stable memory usage with a negligible change of -2MB
- Overall Efficiency: The multi-threaded pipeline architecture successfully decoupled I/O operations from compute-intensive tasks, leveraged GPU acceleration where available, and implemented proper resource management techniques.

These results indicate that the optimization strategies—including process optimization through multithreading, CPU optimization via CUDA utilization, memory optimization with proper garbage collection, and file management optimization using memory mapping—successfully achieved the project's goals of reducing latency, increasing throughput, and maintaining system stability for real-time traffic analysis applications.

Code Appendix

GitHub Repository: https://github.com/sahatsawat-pks/traffic_analysis

Features

- **Object Detection**: Utilizes YOLO for detecting vehicles in the video.
- **Object Tracking**: Integrates ByteTrack for robust tracking of detected objects across frames.
- **Zone-Based Analysis**: Monitors predefined zones and tracks the movement of vehicles in and out of these zones.
- **Graphical User Interface (GUI)**: The optimized version provides an easy-to-use Tkinter interface for configuration and execution.
- **Customizable Settings**: Fine-tune detection and tracking with customizable confidence and IoU thresholds, frame downsampling, frame skipping, and thread count via the GUI or command-line arguments (original version).
- **Optimized Processing**:
  - **Multi-threaded Pipeline**: Efficiently reads, processes, annotates, and writes frames in parallel.
  - **Memory Mapping (MMap)**: Uses memory-mapped file access for optimized video reading, reducing RAM usage for suitable video formats.
  - **Half-Precision (FP16)**: Option to use FP16 for faster inference on compatible GPUs.
  - **Resource Management**: Includes memory monitoring and thorough resource cleanup.
- **Real-Time Video Processing**: Processes video with optional output to a target video file.
- **Visual Annotations**: Annotates frames with bounding boxes, labels, and trace lines, providing a clear visual representation of vehicle movements and zone counts.
- **Performance Testing**: Includes a script to compare the original and optimized implementations and measure improvements in speed and resource usage.

💻 Install

- Clone the repository and navigate to the directory: bash git clone https://github.com/shrimpstanot/traffic_analysis.git cd traffic_analysis

- Set up a Python environment and activate it [optional]: bash python3 -m venv venv source venv/bin/activate # or ".\venv\Scripts\activate" on Windows

- Install required dependencies: bash pip install -r requirements.txt *(Ensure requirements.txt includes tkinter, numpy, supervision, ultralytics, opencv-python, psutil, tqdm, torch)*

- Download traffic_analysis.pt (YOLO weights) and traffic_analysis.mov (sample video) files: bash ./setup.sh # or manually download and place them in a 'data/' subdirectory

GUI Usage (Optimized Version)

The optimized version (main_optimize.py) provides a graphical interface for easier operation.

1. **Launch the Application**: bash python main_optimize.py
2. **File Selection**:
   - Use the "Browse" buttons to select the YOLO model weights (.pt), the source video file, and the target video path (where the processed video will be saved).
   - The target path is optional; if left blank, the processed video will be displayed in a window but not saved.
3. **Configuration**:
   - Adjust the **Confidence Threshold**, **IOU Threshold**, **Downsample Factor**, and **Skip Frames** using the sliders. The current value is displayed next to each slider.
4. **Advanced Settings**:
   - **Use FP16**: Check this box to enable half-precision inference (requires a compatible GPU).
   - **Annotate Frames**: Check this box to draw bounding boxes, tracks, and counts on the output video. Uncheck for faster processing if annotations are not needed.
   - **Processing Threads**: Set the number of threads dedicated to processing frames (inference and tracking).
   - **Queue Size**: Configure the maximum size of the internal queues used in the multi-threaded pipeline.
5. **Start Processing**:
   - Click the "Start Processing" button. A progress window will appear showing the status.
   - You can cancel the process using the "Cancel" button in the progress window.
6. **Completion**:
   - Once processing is finished (or canceled), a message box will display the results, including processing time, FPS, and memory usage.

🛠️ Script Arguments (Original Version)

The original version (ultralytics_example.py) uses command-line arguments for configuration:

- --source_weights_path: **Required**. Specifies the path to the YOLO model's weights file (.pt).
- --source_video_path: **Required**. The path to the source video file to be analyzed.

- --target_video_path (optional): The path to save the output video with annotations. If not specified, the processed video might be displayed in real-time (depending on the original script's implementation) or not saved.
- --confidence_threshold (optional): Sets the confidence threshold for YOLO detections. Default is 0.3.
- --iou_threshold (optional): Specifies the IOU (Intersection Over Union) threshold for Non-Max Suppression. Default is 0.7.

(Note: The optimized GUI version uses UI elements for these settings instead of command-line arguments.)

⚙️ Run Examples

Original Version (Command-Line):

```
python ultralytics_example.py \
--source_weights_path data/traffic_analysis.pt \
--source_video_path data/traffic_analysis.mov \
--confidence_threshold 0.3 \
--iou_threshold 0.5 \
--target_video_path data/traffic_analysis_result.mov
```

Optimize Version (GUI):

```
python main_optimize.py \
--source_weights_path data/traffic_analysis.pt \
--source_video_path data/traffic_analysis.mov \
--confidence_threshold 0.3 \
--iou_threshold 0.5 \
--target_video_path data/traffic_analysis_optimized.mov \
--num_threads 8 \
--downsample_factor 0.8
```

⚙️ Optimization Features:

The optimized implementation (main_optimize.py) includes several improvements:

- **Graphical User Interface**: Easy configuration and execution via Tkinter.
- **Multi-threading Pipeline**: Separate threads for reading, processing, annotating (optional), and writing frames, improving parallelism.
- **Memory Mapping (MMap)**: Attempts to use memory-mapped files for video reading, reducing RAM usage for certain video formats.
- **Memory Management**: Includes monitoring of peak memory usage and specific cleanup routines to release resources (including GPU memory if applicable).
- **Thread-Safe Data Structures**: Uses locks and thread-safe queues for reliable data handling between threads.

- **Frame Downsampling**: Option to resize frames before processing to reduce computational load at the cost of potential accuracy.
- **Frame Skipping**: Option to process only every Nth frame to increase throughput.
- **Half-Precision Inference (FP16)**: Option to use FP16 computation for potentially faster inference on compatible hardware.
- **Progress Monitoring**: Real-time feedback on processing status via the GUI and console output.
- **Resource Cleanup**: Implements explicit cleanup of resources like the model, tracker, video writers, and forces garbage collection.

Performance Testing: The project includes a performance testing script to compare the original implementation with the optimized version:

```
python performance_test.py
```

This will run both implementations on the same video file and generate performance metrics including:

- Execution time
- Frames Per Second (FPS)
- Peak memory usage (RAM and possibly VRAM)
- CPU utilization
- Performance improvement percentages

A visual comparison chart is saved as performance_comparison.png.

References

Roboflow. Supervision [Computer software]. https://github.com/roboflow/supervision
Ultralytics. YOLO [Computer software]. https://github.com/ultralytics/ultralytics

License

Code (Original)

```
import tkinter as tk
from tkinter import filedialog
import numpy as np
import supervision as sv
from ultralytics import YOLO
import cv2
from typing import Dict, Iterable, List, Set
from tqdm import tqdm
from concurrent.futures import ThreadPoolExecutor
import gc
import os

os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"

COLORS = sv.ColorPalette.from_hex(["#E6194B", "#3CB44B", "#FFE119", "#3C76D1"])

ZONE_IN_POLYGONS = [
    np.array([[592, 282], [900, 282], [900, 82], [592, 82]]),
    np.array([[950, 860], [1250, 860], [1250, 1060], [950, 1060]]),
    np.array([[592, 582], [592, 860], [392, 860], [392, 582]]),
    np.array([[1250, 282], [1250, 530], [1450, 530], [1450, 282]]),
]

ZONE_OUT_POLYGONS = [
    np.array([[950, 282], [1250, 282], [1250, 82], [950, 82]]),
    np.array([[592, 860], [900, 860], [900, 1060], [592, 1060]]),
    np.array([[592, 282], [592, 550], [392, 550], [392, 282]]),
    np.array([[1250, 860], [1250, 560], [1450, 560], [1450, 860]]),
]


class DetectionsManager:
    """
    Manages detections across frames, tracking objects and counting zone transitions.

    Attributes:
        tracker_id_to_zone_id (Dict[int, int]): Maps tracker IDs to zone IDs.
        counts (Dict[int, Dict[int, Set[int]]]): Counts of zone transitions.
    """

    def __init__(self) -> None:
        self.tracker_id_to_zone_id: Dict[int, int] = {}
        self.counts: Dict[int, Dict[int, Set[int]]] = {}

    def update(
        self,
        detections_all: sv.Detections,
```

```
        detections_in_zones: List[sv.Detections],
        detections_out_zones: List[sv.Detections],
    ) -> sv.Detections:
        """
        Updates the detections manager with new detections.

        Args:
            detections_all (sv.Detections): All detections in the frame.
            detections_in_zones (List[sv.Detections]): Detections in each zone.
            detections_out_zones (List[sv.Detections]): Detections out of each zone.

        Returns:
            sv.Detections: Updated detections with zone information.
        """
        for zone_in_id, detections_in_zone in enumerate(detections_in_zones):
            for tracker_id in detections_in_zone.tracker_id:
                self.tracker_id_to_zone_id.setdefault(tracker_id, zone_in_id)

        for zone_out_id, detections_out_zone in enumerate(detections_out_zones):
            for tracker_id in detections_out_zone.tracker_id:
                if tracker_id in self.tracker_id_to_zone_id:
                    zone_in_id = self.tracker_id_to_zone_id[tracker_id]
                    self.counts.setdefault(zone_out_id, {})
                    self.counts[zone_out_id].setdefault(zone_in_id, set())
                    self.counts[zone_out_id][zone_in_id].add(tracker_id)

        if len(detections_all) > 0:
            detections_all.class_id = np.vectorize(
                lambda x: self.tracker_id_to_zone_id.get(x, -1)
            )(detections_all.tracker_id)
        else:
            detections_all.class_id = np.array([], dtype=int)

        return detections_all[detections_all.class_id != -1]


def initiate_polygon_zones(
    polygons: List[np.ndarray],
    triggering_anchors: Iterable[sv.Position] = [sv.Position.CENTER],
) -> List[sv.PolygonZone]:
    """Creates polygon zones from polygons and triggering anchors."""
    return [
        sv.PolygonZone(
            polygon=polygon,
            triggering_anchors=triggering_anchors,
        )
        for polygon in polygons
    ]
```

```
class VideoProcessor:
    """
    Processes a video using a YOLO model and tracks objects across frames.

    Attributes:
        conf_threshold (float): Confidence threshold for detections.
        iou_threshold (float): IOU threshold for detections.
        source_video_path (str): Path to the source video.
        target_video_path (str): Path to the target video.
        model (YOLO): YOLO model for detections.
        tracker (sv.ByteTrack): Tracker for object tracking.
        video_info (sv.VideoInfo): Video information.
        zones_in (List[sv.PolygonZone]): Polygon zones for entering objects.
        zones_out (List[sv.PolygonZone]): Polygon zones for exiting objects.
        box_annotator (sv.BoxAnnotator): Box annotator for drawing bounding boxes.
        label_annotator (sv.LabelAnnotator): Label annotator for drawing labels.
        trace_annotator (sv.TraceAnnotator): Trace annotator for drawing object traces.
        detections_manager (DetectionsManager): Detections manager for tracking objects.
    """

    def __init__(
        self,
        source_weights_path: str,
        source_video_path: str,
        target_video_path: str,
        confidence_threshold: float,
        iou_threshold: float,
    ) -> None:
        self.conf_threshold = confidence_threshold
        self.iou_threshold = iou_threshold
        self.source_video_path = source_video_path
        self.target_video_path = target_video_path

        self.model = YOLO(source_weights_path)
        self.tracker = sv.ByteTrack()

        self.video_info = sv.VideoInfo.from_video_path(source_video_path)
        self.zones_in = initiate_polygon_zones(ZONE_IN_POLYGONS, [sv.Position.CENTER])
        self.zones_out = initiate_polygon_zones(ZONE_OUT_POLYGONS, [sv.Position.CENTER])

        self.box_annotator = sv.BoxAnnotator(color=COLORS)
        self.label_annotator = sv.LabelAnnotator(
            color=COLORS, text_color=sv.Color.BLACK
        )
        self.trace_annotator = sv.TraceAnnotator(
            color=COLORS, position=sv.Position.CENTER, trace_length=100, thickness=2
        )
        self.detections_manager = DetectionsManager()
```

```
    def process_video(self):
        """
        Processes the video using the YOLO model and tracker.

        If a target video path is provided, the processed video is written to that path.
        Otherwise, the processed video is displayed in a window.
        """
        frame_generator = sv.get_video_frames_generator(
            source_path=self.source_video_path
        )

        if self.target_video_path:
            with sv.VideoSink(self.target_video_path, self.video_info) as sink:
                with ThreadPoolExecutor(max_workers=4) as executor:
                    for frame_num, frame in enumerate(tqdm(frame_generator,
total=self.video_info.total_frames)):
                        if frame_num % 2 != 0:  # Skip every other frame to speed up processing
                            continue
                        annotated_frame = executor.submit(self.process_frame, frame).result()
                        sink.write_frame(annotated_frame)
                        gc.collect()  # Force garbage collection to manage memory
        else:
            for frame_num, frame in enumerate(tqdm(frame_generator,
total=self.video_info.total_frames)):
                if frame_num % 2 != 0:  # Skip every other frame
                    continue
                annotated_frame = self.process_frame(frame)
                cv2.imshow("Processed Video", annotated_frame)
                if cv2.waitKey(1) & 0xFF == ord("q"):
                    break
            cv2.destroyAllWindows()

    def annotate_frame(
        self, frame: np.ndarray, detections: sv.Detections
    ) -> np.ndarray:
        """
        Annotates a frame with detections.

        Args:
            frame (np.ndarray): The frame to annotate.
            detections (sv.Detections): The detections to annotate.

        Returns:
            np.ndarray: The annotated frame.
        """
        annotated_frame = frame.copy()
        for i, (zone_in, zone_out) in enumerate(zip(self.zones_in, self.zones_out)):
            annotated_frame = sv.draw_polygon(
```

```
            annotated_frame, zone_in.polygon, COLORS.colors[i]
        )
        annotated_frame = sv.draw_polygon(
            annotated_frame, zone_out.polygon, COLORS.colors[i]
        )

    labels = [f"#{tracker_id}" for tracker_id in detections.tracker_id]
    annotated_frame = self.trace_annotator.annotate(annotated_frame, detections)
    annotated_frame = self.box_annotator.annotate(annotated_frame, detections)
    annotated_frame = self.label_annotator.annotate(
        annotated_frame, detections, labels
    )

    for zone_out_id, zone_out in enumerate(self.zones_out):
        zone_center = sv.get_polygon_center(polygon=zone_out.polygon)
        if zone_out_id in self.detections_manager.counts:
            counts = self.detections_manager.counts[zone_out_id]
            for i, zone_in_id in enumerate(counts):
                count = len(self.detections_manager.counts[zone_out_id][zone_in_id])
                text_anchor = sv.Point(x=zone_center.x, y=zone_center.y + 40 * i)
                annotated_frame = sv.draw_text(
                    scene=annotated_frame,
                    text=str(count),
                    text_anchor=text_anchor,
                    background_color=COLORS.colors[zone_in_id],
                )

    return annotated_frame

def process_frame(self, frame: np.ndarray) -> np.ndarray:
    """Processes a frame using the YOLO model and tracker."""
    results = self.model(
        frame, verbose=False, conf=self.conf_threshold, iou=self.iou_threshold
    )[0]
    detections = sv.Detections.from_ultralytics(results)
    detections.class_id = np.zeros(len(detections))
    detections = self.tracker.update_with_detections(detections)

    detections_in_zones = []
    detections_out_zones = []

    for zone_in, zone_out in zip(self.zones_in, self.zones_out):
        detections_in_zone = detections[zone_in.trigger(detections=detections)]
        detections_in_zones.append(detections_in_zone)
        detections_out_zone = detections[zone_out.trigger(detections=detections)]
        detections_out_zones.append(detections_out_zone)

    detections = self.detections_manager.update(
        detections, detections_in_zones, detections_out_zones
```

```
        )
        return self.annotate_frame(frame, detections)


class Application(tk.Frame):
    """Application for processing videos using a YOLO model and tracker."""

    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack()
        self.create_widgets()

    def create_widgets(self):
        """Creates the application widgets."""
        self.source_weights_path = tk.StringVar()
        self.source_video_path = tk.StringVar()
        self.target_video_path = tk.StringVar()
        self.confidence_threshold = tk.DoubleVar(value=0.3)
        self.iou_threshold = tk.DoubleVar(value=0.7)

        tk.Label(self, text="Source Weights Path").grid(row=0, column=0)
        tk.Entry(self, textvariable=self.source_weights_path).grid(row=0, column=1)
        tk.Button(self, text="Browse", command=self.browse_source_weights).grid(row=0, column=2)

        tk.Label(self, text="Source Video Path").grid(row=1, column=0)
        tk.Entry(self, textvariable=self.source_video_path).grid(row=1, column=1)
        tk.Button(self, text="Browse", command=self.browse_source_video).grid(row=1, column=2)

        tk.Label(self, text="Target Video Path").grid(row=2, column=0)
        tk.Entry(self, textvariable=self.target_video_path).grid(row=2, column=1)
        tk.Button(self, text="Browse", command=self.browse_target_video).grid(row=2, column=2)

        tk.Label(self, text="Confidence Threshold").grid(row=3, column=0)
        tk.Entry(self, textvariable=self.confidence_threshold).grid(row=3, column=1)

        tk.Label(self, text="IOU Threshold").grid(row=4, column=0)
        tk.Entry(self, textvariable=self.iou_threshold).grid(row=4, column=1)

        tk.Button(self, text="Start Processing", command=self.start_processing).grid(row=5, column=1)

    def browse_source_weights(self):
        """Browses for the source weights file."""
        file_path = filedialog.askopenfilename()
        self.source_weights_path.set(file_path)

    def browse_source_video(self):
        """Browses for the source video file."""
        file_path = filedialog.askopenfilename()
```

```
        self.source_video_path.set(file_path)

    def browse_target_video(self):
        """Browses for the target video file."""
        file_path = filedialog.askopenfilename()
        self.target_video_path.set(file_path)

    def start_processing(self):
        """Starts processing the video."""
        processor = VideoProcessor(
            source_weights_path=self.source_weights_path.get(),
            source_video_path=self.source_video_path.get(),
            target_video_path=self.target_video_path.get(),
            confidence_threshold=self.confidence_threshold.get(),
            iou_threshold=self.iou_threshold.get(),
        )
        processor.process_video()
        self.master.quit()


if __name__ == "__main__":
    root = tk.Tk()
    app = Application(master=root)
    app.mainloop()
```

Code (Optimized)

```
import tkinter as tk
from tkinter import filedialog, Scale, IntVar, BooleanVar, Checkbutton, messagebox
import numpy as np
import supervision as sv
from ultralytics import YOLO
import cv2
from typing import Dict, Iterable, List, Set, Optional, Tuple, Any
from tqdm import tqdm
import gc
import os
import torch
import psutil
import time
import threading
import queue
from concurrent.futures import ThreadPoolExecutor
import logging
import mmap

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s -
%(message)s')
logger = logging.getLogger(__name__)

os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"

COLORS = sv.ColorPalette.from_hex(["#E6194B", "#3CB44B", "#FFE119", "#3C76D1"])

ZONE_IN_POLYGONS = [
    np.array([[592, 282], [900, 282], [900, 82], [592, 82]]),
    np.array([[950, 860], [1250, 860], [1250, 1060], [950, 1060]]),
    np.array([[592, 582], [592, 860], [392, 860], [392, 582]]),
    np.array([[1250, 282], [1250, 530], [1450, 530], [1450, 282]]),
]

ZONE_OUT_POLYGONS = [
    np.array([[950, 282], [1250, 282], [1250, 82], [950, 82]]),
    np.array([[592, 860], [900, 860], [900, 1060], [592, 1060]]),
    np.array([[592, 282], [592, 550], [392, 550], [392, 282]]),
    np.array([[1250, 860], [1250, 560], [1450, 560], [1450, 860]]),
]

class MmapFrameReader:
    """
    Memory-mapped file reader for optimized video frame access.
    Uses mmap for efficient file I/O operations with minimal memory overhead.
    """
```

```
    def __init__(self, file_path: str, frame_size: int, header_size: int = 0):
        self.file_path = file_path
        self.frame_size = frame_size
        self.header_size = header_size

        # Open file and create memory map
        self.file_obj = open(file_path, 'rb')
        self.file_size = os.path.getsize(file_path)
        self.mmap_obj = mmap.mmap(
            self.file_obj.fileno(),
            length=0,  # Map the entire file
            access=mmap.ACCESS_READ  # Read-only access
        )

        # Calculate number of frames
        self.data_size = self.file_size - self.header_size
        self.num_frames = self.data_size // self.frame_size

        # Current position tracker
        self.current_frame = 0

    def seek_frame(self, frame_idx: int) -> bool:
        """Seek to a specific frame."""
        if 0 <= frame_idx < self.num_frames:
            self.current_frame = frame_idx
            return True
        return False

    def read_frame_data(self, frame_idx: Optional[int] = None) -> bytes:
        """Read raw data for a specific frame."""
        idx = self.current_frame if frame_idx is None else frame_idx

        if 0 <= idx < self.num_frames:
            offset = self.header_size + (idx * self.frame_size)
            return self.mmap_obj[offset:offset + self.frame_size]

        return None

    def read_next_frame(self) -> bytes:
        """Read the next frame and advance position."""
        if self.current_frame < self.num_frames:
            data = self.read_frame_data(self.current_frame)
            self.current_frame += 1
            return data
        return None

    def close(self):
        """Close the memory map and file."""
        if hasattr(self, 'mmap_obj') and self.mmap_obj is not None:
```

```
            self.mmap_obj.close()
            self.mmap_obj = None

        if hasattr(self, 'file_obj') and self.file_obj is not None:
            self.file_obj.close()
            self.file_obj = None

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.close()


class MmapVideoReader:
    """
    Memory-mapped video reader that combines mmap file access with OpenCV
    for optimized video frame access with lower memory usage.
    """
    def __init__(self, video_path: str, buffer_size: int = 16):
        import cv2
        import numpy as np

        self.video_path = video_path
        self.buffer_size = buffer_size

        # Open video with OpenCV to get properties
        cap = cv2.VideoCapture(video_path)
        self.width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
        self.height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
        self.fps = cap.get(cv2.CAP_PROP_FPS)
        self.total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
        self.fourcc = int(cap.get(cv2.CAP_PROP_FOURCC))

        # Get video codec information
        fourcc_chars = [chr((self.fourcc >> 8 * i) & 0xFF) for i in range(4)]
        self.codec = ''.join(fourcc_chars)

        # Create frame buffer
        self.frame_buffer = {}
        self.current_frame = 0

        # Close OpenCV capture since we'll use mmap
        cap.release()

        # Initialize video parser based on codec
        self._setup_decoder()

    def _setup_decoder(self):
```

```
        """Set up the appropriate decoder based on video codec."""
        import cv2

        # For uncompressed formats (like raw YUV), we could use mmap directly
        # For compressed formats (like MP4), we need to use OpenCV for decoding

        # Check if format is suitable for direct mmap
        if self.codec in ['RAW ', 'YUV ']:
            # Setup direct mmap access if possible
            # This is an example and would need customization based on format
            frame_size = self.width * self.height * 3  # Assuming 3 channels (RGB)
            try:
                self.mmap_reader = MmapFrameReader(self.video_path, frame_size)
                self.use_direct_mmap = True
                return
            except Exception as e:
                logging.warning(f"Failed to set up direct mmap access: {e}")

        # Fallback to OpenCV with strategic buffering
        self.use_direct_mmap = False
        self.cap = cv2.VideoCapture(self.video_path)

        # Pre-allocate memory for frame buffer
        self.frame_buffer = {}

    def _buffer_frames(self, start_idx: int, count: int = None):
        """Buffer a range of frames."""
        if not count:
            count = self.buffer_size

        end_idx = min(start_idx + count, self.total_frames)

        # Skip if using direct mmap
        if self.use_direct_mmap:
            return

        # Clear old buffer
        current_buffer_keys = list(self.frame_buffer.keys())
        for key in current_buffer_keys:
            if key < start_idx or key >= end_idx:
                del self.frame_buffer[key]

        # Fill buffer with new frames
        if self.cap is not None:
            for idx in range(start_idx, end_idx):
                if idx not in self.frame_buffer:
                    self.cap.set(cv2.CAP_PROP_POS_FRAMES, idx)
                    success, frame = self.cap.read()
                    if success:
```

```
                self.frame_buffer[idx] = frame

    def read_frame(self, frame_idx: int = None):
        """Read a specific frame or the next frame."""
        import cv2
        import numpy as np

        if frame_idx is None:
            frame_idx = self.current_frame
            self.current_frame += 1

        if frame_idx >= self.total_frames:
            return False, None

        # Direct mmap access for uncompressed formats
        if self.use_direct_mmap:
            try:
                raw_data = self.mmap_reader.read_frame_data(frame_idx)
                if raw_data:
                    # Convert raw bytes to numpy array
                    # This needs customization based on pixel format
                    frame = np.frombuffer(raw_data, dtype=np.uint8)
                    frame = frame.reshape((self.height, self.width, 3))
                    return True, frame
            except Exception as e:
                logging.error(f"Error reading frame with mmap: {e}")
                return False, None

        # Optimized OpenCV access with buffering
        # Check if frame is in buffer, otherwise load from disk
        if frame_idx not in self.frame_buffer:
            buffer_start = (frame_idx // self.buffer_size) * self.buffer_size
            self._buffer_frames(buffer_start)

        # Return frame from buffer if available
        if frame_idx in self.frame_buffer:
            return True, self.frame_buffer[frame_idx]

        # Fallback to direct access
        self.cap.set(cv2.CAP_PROP_POS_FRAMES, frame_idx)
        return self.cap.read()

    def close(self):
        """Close all resources."""
        if hasattr(self, 'mmap_reader') and self.mmap_reader is not None:
            self.mmap_reader.close()

        if hasattr(self, 'cap') and self.cap is not None:
            self.cap.release()
```

```
        # Clear buffer
        self.frame_buffer.clear()

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.close()


def modify_frame_reader_to_use_mmap(processor):
    """
    Modifies the frame_reader function of VideoProcessor_Optimize to use memory-mapped file access.
    This function should be called before starting video processing.
    """
    def frame_reader_with_mmap(self):
        """Thread function for reading frames from video with mmap optimization."""
        try:
            # Initialize mmap video reader
            mmap_reader = MmapVideoReader(self.source_video_path, buffer_size=32)
            frame_idx = 0

            while not self.stop_event.is_set():
                # Read frame
                success, frame = mmap_reader.read_frame()
                if not success:
                    break

                # Update frame counter
                with self.stats_lock:
                    self.frame_count += 1

                # Skip frames if needed
                if self.skip_frames > 0 and frame_idx % (self.skip_frames + 1) != 0:
                    frame_idx += 1
                    continue

                # Put frame in queue
                try:
                    self.frame_queue.put((frame_idx, frame), timeout=1.0)
                    frame_idx += 1
                except queue.Full:
                    if self.stop_event.is_set():
                        break
                    time.sleep(0.01)  # Small sleep to prevent CPU hogging

            # Signal end of frames
            self.frame_queue.put((None, None))
```

```
            # Clean up
            mmap_reader.close()

        except Exception as e:
            logger.error(f"Error in mmap frame reader: {e}")
            self.stop_event.set()

    # Replace the original frame_reader method with our mmap version
    processor.frame_reader = frame_reader_with_mmap.__get__(processor, type(processor))

    return processor

class DetectionsManager:
    """
    Manages detections across frames, tracking objects and counting zone transitions.
    """

    def __init__(self) -> None:
        self.tracker_id_to_zone_id: Dict[int, int] = {}
        self.counts: Dict[int, Dict[int, Set[int]]] = {}
        self.lock = threading.RLock()  # Reentrant lock for thread safety

    def update(
        self,
        detections_all: sv.Detections,
        detections_in_zones: List[sv.Detections],
        detections_out_zones: List[sv.Detections],
    ) -> sv.Detections:
        """
        Updates the detections manager with new detections.
        Thread-safe implementation.
        """
        with self.lock:
            # Process zone entry
            for zone_in_id, detections_in_zone in enumerate(detections_in_zones):
                if detections_in_zone.tracker_id is not None:
                    for tracker_id in detections_in_zone.tracker_id:
                        self.tracker_id_to_zone_id.setdefault(tracker_id, zone_in_id)

            # Process zone exit and counting
            for zone_out_id, detections_out_zone in enumerate(detections_out_zones):
                if detections_out_zone.tracker_id is not None:
                    for tracker_id in detections_out_zone.tracker_id:
                        if tracker_id in self.tracker_id_to_zone_id:
                            zone_in_id = self.tracker_id_to_zone_id[tracker_id]
                            self.counts.setdefault(zone_out_id, {})
                            self.counts[zone_out_id].setdefault(zone_in_id, set())
                            self.counts[zone_out_id][zone_in_id].add(tracker_id)
```

```
        # Update class IDs based on tracking
        if len(detections_all) > 0 and detections_all.tracker_id is not None:
            detections_all.class_id = np.array([
                self.tracker_id_to_zone_id.get(tracker_id, -1)
                for tracker_id in detections_all.tracker_id
            ])
        else:
            detections_all.class_id = np.array([], dtype=int)

        return detections_all[detections_all.class_id != -1]

    def get_counts(self) -> Dict[int, Dict[int, int]]:
        """Return the counts in a thread-safe way."""
        with self.lock:
            result = {}
            for zone_out_id, zones in self.counts.items():
                result[zone_out_id] = {}
                for zone_in_id, trackers in zones.items():
                    result[zone_out_id][zone_in_id] = len(trackers)
            return result


def initiate_polygon_zones(
    polygons: List[np.ndarray],
    triggering_anchors: Iterable[sv.Position] = [sv.Position.CENTER],
) -> List[sv.PolygonZone]:
    """Creates polygon zones from polygons and triggering anchors."""
    return [
        sv.PolygonZone(
            polygon=polygon,
            triggering_anchors=triggering_anchors,
        )
        for polygon in polygons
    ]


def scale_polygon(polygon: np.ndarray, scale_factor: float) -> np.ndarray:
    """Scale a polygon by the given factor."""
    centroid = np.mean(polygon, axis=0)
    return np.array([
        centroid + (point - centroid) * scale_factor
        for point in polygon
    ], dtype=np.int32)


class MemoryMonitor:
    """
    Monitors memory usage during video processing.
```

```
    """
    def __init__(self):
        self.peak_memory = 0
        self.start_memory = psutil.Process(os.getpid()).memory_info().rss / (1024 * 1024)
        self.lock = threading.Lock()

    def update(self):
        current = psutil.Process(os.getpid()).memory_info().rss / (1024 * 1024)  # MB
        with self.lock:
            self.peak_memory = max(self.peak_memory, current)
        self.peak_memory = max(self.peak_memory, current)
        return current

    def get_peak(self):
        with self.lock:
            return self.peak_memory

    def get_increase(self):
        with self.lock:
            return self.peak_memory - self.start_memory

class FrameProcessor:
    """
    Processes a single frame with YOLO detection and tracking.
    This class is designed to be used by multiple threads.
    """
    def __init__(
        self,
        model: YOLO,
        tracker: sv.ByteTrack,
        zones_in: List[sv.PolygonZone],
        zones_out: List[sv.PolygonZone],
        detections_manager: DetectionsManager,
        conf_threshold: float,
        iou_threshold: float,
        use_fp16: bool,
        downsample_factor: float = 1.0,
    ):
        self.model = model
        self.tracker = tracker
        self.zones_in = zones_in
        self.zones_out = zones_out
        self.detections_manager = detections_manager
        self.conf_threshold = conf_threshold
        self.iou_threshold = iou_threshold
        self.use_fp16 = use_fp16
        self.downsample_factor = downsample_factor

        # Initialize thread-local storage
```

```
        self.thread_local = threading.local()

        # Model lock for thread safety if model is not thread-safe
        self.model_lock = threading.Lock()

    def resize_frame(self, frame: np.ndarray) -> Tuple[np.ndarray, Tuple[int, int]]:
        """Resize frame based on downsample factor."""
        if self.downsample_factor == 1.0:
            return frame, frame.shape[:2]

        h, w = frame.shape[:2]
        new_h, new_w = int(h * self.downsample_factor), int(w * self.downsample_factor)
        return cv2.resize(frame, (new_w, new_h)), (h, w)

    def process(self, frame: np.ndarray) -> sv.Detections:
        """Process a single frame and return detections."""
        # Resize frame if needed
        resized_frame, original_dims = self.resize_frame(frame)

        # Use thread-local storage for per-thread caching

        # CPU Optimization

        if not hasattr(self.thread_local, 'device_set'):
            # Set device for this thread
            self.thread_local.device_set = True
            if torch.cuda.is_available():
                # If multiple GPUs are available, we could distribute across them
                device_id = threading.current_thread().ident % torch.cuda.device_count() if
torch.cuda.device_count() > 1 else 0
                self.thread_local.device = torch.device(f'cuda:{device_id}')
            else:
                self.thread_local.device = torch.device('cpu')

        # Run inference with model lock to ensure thread safety
        with self.model_lock:
            with torch.inference_mode():
                # Run model inference
                results = self.model(
                    resized_frame,
                    verbose=False,
                    conf=self.conf_threshold,
                    iou=self.iou_threshold,
                    half=self.use_fp16
                )[0]

                # Convert results to detections
                detections = sv.Detections.from_ultralytics(results)
```

```
            if len(detections) > 0:
                detections.class_id = np.zeros(len(detections))

                # Update with tracker (tracker should be thread-safe)
                detections = self.tracker.update_with_detections(detections)

                # Process zones
                detections_in_zones = []
                detections_out_zones = []

                for zone_in, zone_out in zip(self.zones_in, self.zones_out):
                    detections_in_zone = detections[zone_in.trigger(detections=detections)]
                    detections_in_zones.append(detections_in_zone)
                    detections_out_zone = detections[zone_out.trigger(detections=detections)]
                    detections_out_zones.append(detections_out_zone)

                # Update detections manager
                detections = self.detections_manager.update(
                    detections, detections_in_zones, detections_out_zones
                )

                # Clear CUDA cache if needed
                if torch.cuda.is_available():
                    torch.cuda.empty_cache()

        return detections


class FrameAnnotator:
    """
    Handles the annotation of frames with detections.
    """
    def __init__(
        self,
        zones_in: List[sv.PolygonZone],
        zones_out: List[sv.PolygonZone],
        detections_manager: DetectionsManager,
    ):
        self.zones_in = zones_in
        self.zones_out = zones_out
        self.detections_manager = detections_manager

        # Initialize annotators
        self.box_annotator = sv.BoxAnnotator(color=COLORS)
        self.label_annotator = sv.LabelAnnotator(color=COLORS, text_color=sv.Color.BLACK)
        self.trace_annotator = sv.TraceAnnotator(
            color=COLORS, position=sv.Position.CENTER, trace_length=50, thickness=2
        )
```

```
    def annotate(self, frame: np.ndarray, detections: sv.Detections) -> np.ndarray:
        """Annotate a frame with detections."""
        annotated_frame = frame.copy()

        # Draw zones
        for i, (zone_in, zone_out) in enumerate(zip(self.zones_in, self.zones_out)):
            annotated_frame = sv.draw_polygon(
                annotated_frame, zone_in.polygon, COLORS.colors[i]
            )
            annotated_frame = sv.draw_polygon(
                annotated_frame, zone_out.polygon, COLORS.colors[i]
            )

        # Only annotate if there are detections
        if len(detections) > 0 and detections.tracker_id is not None:
            # Generate minimal labels
            labels = [f"#{tid}" for tid in detections.tracker_id]

            # Apply annotations
            annotated_frame = self.trace_annotator.annotate(annotated_frame, detections)
            annotated_frame = self.box_annotator.annotate(annotated_frame, detections)
            annotated_frame = self.label_annotator.annotate(
                annotated_frame, detections, labels
            )

        # Draw counts
        counts = self.detections_manager.get_counts()
        for zone_out_id, zone_out in enumerate(self.zones_out):
            if zone_out_id in counts:
                zone_center = sv.get_polygon_center(polygon=zone_out.polygon)
                zone_counts = counts[zone_out_id]

                for i, zone_in_id in enumerate(zone_counts):
                    count = zone_counts[zone_in_id]
                    text_anchor = sv.Point(x=zone_center.x, y=zone_center.y + 40 * i)
                    annotated_frame = sv.draw_text(
                        scene=annotated_frame,
                        text=str(count),
                        text_anchor=text_anchor,
                        background_color=COLORS.colors[zone_in_id],
                    )

        return annotated_frame

class VideoProcessor_Optimize:
    """
    Processes a video using a YOLO model and tracks objects across frames.
    Multi-threaded / Memory implementation with I/O optimization.
    """
```

```
    def __init__(
        self,
        source_weights_path: str,
        source_video_path: str,
        target_video_path: str,
        confidence_threshold: float,
        iou_threshold: float,
        downsample_factor: float = 1.0,
        skip_frames: int = 0,
        use_fp16: bool = False,
        annotate_frames: bool = True,
        num_threads: int = 4,
        queue_size: int = 32,
    ) -> None:
        self.conf_threshold = confidence_threshold
        self.iou_threshold = iou_threshold
        self.source_video_path = source_video_path
        self.target_video_path = target_video_path
        self.downsample_factor = downsample_factor
        self.skip_frames = skip_frames
        self.use_fp16 = use_fp16
        self.annotate_frames = annotate_frames
        self.num_threads = num_threads
        self.queue_size = queue_size
        self.memory_monitor = MemoryMonitor()

        # Initialize model with optimized settings
        self.model = self.initialize_model(source_weights_path)
        self.tracker = sv.ByteTrack()

        # Get video info
        self.video_info = sv.VideoInfo.from_video_path(source_video_path)

        # Initialize zones with proper scaling if needed
        if self.downsample_factor != 1.0:
            scaled_zones_in = [scale_polygon(p, downsample_factor) for p in ZONE_IN_POLYGONS]
            scaled_zones_out = [scale_polygon(p, downsample_factor) for p in
ZONE_OUT_POLYGONS]
            self.zones_in = initiate_polygon_zones(scaled_zones_in, [sv.Position.CENTER])
            self.zones_out = initiate_polygon_zones(scaled_zones_out, [sv.Position.CENTER])
        else:
            self.zones_in = initiate_polygon_zones(ZONE_IN_POLYGONS, [sv.Position.CENTER])
            self.zones_out = initiate_polygon_zones(ZONE_OUT_POLYGONS, [sv.Position.CENTER])

        # Initialize the detection manager
        self.detections_manager = DetectionsManager()

        # Initialize the frame processor
```

```
        self.frame_processor = FrameProcessor(
            model=self.model,
            tracker=self.tracker,
            zones_in=self.zones_in,
            zones_out=self.zones_out,
            detections_manager=self.detections_manager,
            conf_threshold=self.conf_threshold,
            iou_threshold=self.iou_threshold,
            use_fp16=self.use_fp16,
            downsample_factor=self.downsample_factor,
        )

        # Initialize annotators only if needed
        if self.annotate_frames:
            self.frame_annotator = FrameAnnotator(
                zones_in=self.zones_in,
                zones_out=self.zones_out,
                detections_manager=self.detections_manager,
            )

        # Initialize thread-safe queues for pipeline
        self.frame_queue = queue.Queue(maxsize=queue_size)
        self.result_queue = queue.Queue(maxsize=queue_size)
        self.output_queue = queue.Queue(maxsize=queue_size)

        # Initialize threading control
        self.stop_event = threading.Event()
        self.threads = []

        # Statistics
        self.start_time = None
        self.frame_count = 0
        self.processed_count = 0
        self.stats_lock = threading.Lock()

    def initialize_model(self, weights_path: str) -> YOLO:
        """Initialize the YOLO model with optimized settings."""
        model = YOLO(weights_path)

        # Optimize model settings
        if torch.cuda.is_available():
            model.to('cuda')
            if self.use_fp16 and torch.cuda.is_available():
                model.model.half()  # Use FP16 precision

        return model

    def frame_reader(self):
        """Thread function for reading frames from video."""
```

```
        try:
            cap = cv2.VideoCapture(self.source_video_path)
            frame_idx = 0

            while not self.stop_event.is_set():
                success, frame = cap.read()
                if not success:
                    break

                # Update frame counter
                with self.stats_lock:
                    self.frame_count += 1

                # Skip frames if needed
                if self.skip_frames > 0 and frame_idx % (self.skip_frames + 1) != 0:
                    frame_idx += 1
                    continue

                # Put frame in queue
                try:
                    self.frame_queue.put((frame_idx, frame), timeout=1.0)
                    frame_idx += 1
                except queue.Full:
                    if self.stop_event.is_set():
                        break
                    time.sleep(0.01)  # Small sleep to prevent CPU hogging

            # Signal end of frames
            self.frame_queue.put((None, None))

        except Exception as e:
            logger.error(f"Error in frame reader: {e}")
            self.stop_event.set()
        finally:
            cap.release()

    def resize_frame(self, frame: np.ndarray) -> Tuple[np.ndarray, Tuple[int, int]]:
        """Resize frame based on downsample factor."""
        if self.downsample_factor == 1.0:
            return frame, frame.shape[:2]

        h, w = frame.shape[:2]
        new_h, new_w = int(h * self.downsample_factor), int(w * self.downsample_factor)
        return cv2.resize(frame, (new_w, new_h)), (h, w)

    def process_frames(self):
        """Thread function for processing frames."""
        try:
            while not self.stop_event.is_set():
```

```
            # Get frame from queue
            try:
                frame_idx, frame = self.frame_queue.get(timeout=1.0)
                if frame_idx is None:  # End signal
                    self.frame_queue.put((None, None))  # Forward the signal
                    break

                # Process frame
                detections = self.frame_processor.process(frame)

                # Put result in queue
                self.result_queue.put((frame_idx, frame, detections), timeout=1.0)

                # Mark task as done
                self.frame_queue.task_done()

                # Update processed count
                with self.stats_lock:
                    self.processed_count += 1

                # Monitor memory
                self.memory_monitor.update()

            except queue.Empty:
                continue

        except Exception as e:
            logger.error(f"Error in frame processor: {e}")
            self.stop_event.set()
        finally:
            # Signal end of processing
            self.result_queue.put((None, None, None))

    def annotate_frames_thread(self):
        """Thread function for annotating frames."""
        try:
            while not self.stop_event.is_set():
                # Get result from queue
                try:
                    frame_idx, frame, detections = self.result_queue.get(timeout=1.0)
                    if frame_idx is None:  # End signal
                        self.result_queue.put((None, None, None))  # Forward the signal
                        break

                    # Annotate frame if needed
                    if self.annotate_frames:
                        annotated_frame = self.frame_annotator.annotate(frame, detections)
                    else:
                        annotated_frame = frame
```

```
                # Put annotated frame in output queue
                self.output_queue.put((frame_idx, annotated_frame), timeout=1.0)

                # Mark task as done
                self.result_queue.task_done()

            except queue.Empty:
                continue

        except Exception as e:
            logger.error(f"Error in frame annotator: {e}")
            self.stop_event.set()
        finally:
            # Signal end of annotation
            self.output_queue.put((None, None))

    def writer(self):
        """Thread function for writing frames to video."""
        try:
            # Setup video writer
            sink = None
            if self.target_video_path:
                # Adjust output resolution if needed
                if self.downsample_factor != 1.0 and self.annotate_frames:
                    # Output will be at the processed resolution
                    h, w = int(self.video_info.height * self.downsample_factor), int(self.video_info.width *
self.downsample_factor)
                    fps = self.video_info.fps
                    try:
                        sink = cv2.VideoWriter(self.target_video_path, cv2.VideoWriter_fourcc(*'mp4v'), fps,
(w, h))
                    except Exception as e:
                        logger.error(f"Error creating video sink: {e}")
                        sink = None
                else:
                    try:
                        sink = cv2.VideoWriter(
                            self.target_video_path,
                            cv2.VideoWriter_fourcc(*'mp4v'),
                            self.video_info.fps,
                            (self.video_info.width, self.video_info.height)
                        )
                    except Exception as e:
                        logger.error(f"Error creating video sink: {e}")
                        sink = None

            # Process frames from output queue
            frames_written = 0
```

```
        buffer = {}  # Buffer to ensure frames are written in order
        next_frame_idx = 0

        while not self.stop_event.is_set():
            try:
                frame_idx, frame = self.output_queue.get(timeout=1.0)
                if frame_idx is None:  # End signal
                    break

                # Store frame in buffer
                buffer[frame_idx] = frame

                # Write frames in order
                while next_frame_idx in buffer:
                    next_frame = buffer.pop(next_frame_idx)
                    if sink is not None:
                        sink.write(next_frame)
                    frames_written += 1
                    next_frame_idx += 1

                # Mark task as done
                self.output_queue.task_done()

                # Display frame if needed
                if sink is None and self.annotate_frames:
                    cv2.imshow("Processed Video", frame)
                    if cv2.waitKey(1) & 0xFF == ord("q"):
                        self.stop_event.set()
                        break

            except queue.Empty:
                continue

    except Exception as e:
        logger.error(f"Error in frame writer: {e}")
        self.stop_event.set()
    finally:
        # Clean up
        if sink is not None:
            sink.release()
        if sink is None and self.annotate_frames:
            cv2.destroyAllWindows()

def process_video(self):
    """Process video using multi-threading pipeline."""
    self.start_time = time.time()
    self.stop_event.clear()

    # Muitithreading (Process Optimization)
```

```
# Start reader thread
reader_thread = threading.Thread(target=self.frame_reader)
reader_thread.daemon = True
reader_thread.start()
self.threads.append(reader_thread)

# Start processor threads
for _ in range(self.num_threads):
    processor_thread = threading.Thread(target=self.process_frames)
    processor_thread.daemon = True
    processor_thread.start()
    self.threads.append(processor_thread)

# Start annotator thread
annotator_thread = threading.Thread(target=self.annotate_frames_thread)
annotator_thread.daemon = True
annotator_thread.start()
self.threads.append(annotator_thread)

# Start writer thread
writer_thread = threading.Thread(target=self.writer)
writer_thread.daemon = True
writer_thread.start()
self.threads.append(writer_thread)

# Create a progress bar
cap = cv2.VideoCapture(self.source_video_path)
total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
cap.release()

with tqdm(total=total_frames) as pbar:
    last_count = 0
    while any(thread.is_alive() for thread in self.threads):
        if self.stop_event.is_set():
            break

        # Update progress bar
        with self.stats_lock:
            new_count = self.frame_count
            if new_count > last_count:
                pbar.update(new_count - last_count)
                last_count = new_count

        time.sleep(0.1)

# Wait for all threads to finish
self.stop_event.set()
for thread in self.threads:
```

```
            thread.join(timeout=2.0)

        # Print statistics
        duration = time.time() - self.start_time
        peak_memory = self.memory_monitor.get_peak()
        memory_increase = self.memory_monitor.get_increase()

        print(f"\nProcessing completed:")
        print(f"- Total frames: {self.frame_count}")
        print(f"- Processed frames: {self.processed_count}")
        print(f"- Processing time: {duration:.2f} seconds")
        print(f"- FPS: {self.processed_count / duration:.2f}")
        print(f"- Peak memory usage: {peak_memory:.2f} MB")
        print(f"- Memory increase: {memory_increase:.2f} MB")
        print(f"- Threads used: {self.num_threads}")

        # Clean up
        self.cleanup()

    def cleanup(self):
        """Clean up resources."""
        # Memory / CPU Optimization
        # Clear CUDA cache
        if torch.cuda.is_available():
            torch.cuda.empty_cache()

        # Move model to CPU
        if hasattr(self.model, 'cpu'):
            self.model.cpu()

        # Clear queues
        while not self.frame_queue.empty():
            try:
                self.frame_queue.get_nowait()
                self.frame_queue.task_done()
            except queue.Empty:
                break

        while not self.result_queue.empty():
            try:
                self.result_queue.get_nowait()
                self.result_queue.task_done()
            except queue.Empty:
                break

        while not self.output_queue.empty():
            try:
                self.output_queue.get_nowait()
                self.output_queue.task_done()
```

```
            except queue.Empty:
                break

        # Delete large objects
        self.model = None
        self.tracker = None
        self.frame_processor = None
        if hasattr(self, 'frame_annotator'):
            self.frame_annotator = None

        # Force garbage collection
        gc.collect()


class Application(tk.Frame):
    """Application for processing videos using a YOLO model and tracker."""

    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.master.title("Video Processing Tool")
        self.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)
        self.current_process = None
        self.progress_window = None
        self.create_widgets()

    def create_widgets(self):
        """Creates the application widgets."""
        # Input variables
        self.source_weights_path = tk.StringVar()
        self.source_video_path = tk.StringVar()
        self.target_video_path = tk.StringVar()
        self.confidence_threshold = tk.DoubleVar(value=0.3)
        self.iou_threshold = tk.DoubleVar(value=0.7)
        self.downsample_factor = tk.DoubleVar(value=1.0)
        self.skip_frames = tk.IntVar(value=0)
        self.use_fp16 = BooleanVar(value=False)
        self.annotate_frames = BooleanVar(value=True)
        self.num_threads = tk.IntVar(value=4)
        self.queue_size = tk.IntVar(value=32)

        # Create main frames
        file_frame = tk.LabelFrame(self, text="File Selection", padx=5, pady=5)
        file_frame.pack(fill=tk.X, expand=False, pady=5)

        config_frame = tk.LabelFrame(self, text="Configuration", padx=5, pady=5)
        config_frame.pack(fill=tk.X, expand=False, pady=5)

        advanced_frame = tk.LabelFrame(self, text="Advanced Settings", padx=5, pady=5)
```

```
        advanced_frame.pack(fill=tk.X, expand=False, pady=5)

        action_frame = tk.Frame(self, padx=5, pady=5)
        action_frame.pack(fill=tk.X, expand=False, pady=10)

        # File selection
        row = 0
        tk.Label(file_frame, text="Source Weights Path").grid(row=row, column=0, sticky="w", pady=2)
        tk.Entry(file_frame, textvariable=self.source_weights_path, width=40).grid(row=row, column=1,
sticky="ew", padx=5)
        tk.Button(file_frame, text="Browse", command=self.browse_source_weights).grid(row=row,
column=2, padx=5)

        row += 1
        tk.Label(file_frame, text="Source Video Path").grid(row=row, column=0, sticky="w", pady=2)
        tk.Entry(file_frame, textvariable=self.source_video_path, width=40).grid(row=row, column=1,
sticky="ew", padx=5)
        tk.Button(file_frame, text="Browse", command=self.browse_source_video).grid(row=row,
column=2, padx=5)

        row += 1
        tk.Label(file_frame, text="Target Video Path").grid(row=row, column=0, sticky="w", pady=2)
        tk.Entry(file_frame, textvariable=self.target_video_path, width=40).grid(row=row, column=1,
sticky="ew", padx=5)
        tk.Button(file_frame, text="Browse", command=self.browse_target_video).grid(row=row,
column=2, padx=5)

        # Configuration options
        row = 0
        tk.Label(config_frame, text="Confidence Threshold").grid(row=row, column=0, sticky="w",
pady=2)
        conf_scale = tk.Scale(config_frame, variable=self.confidence_threshold, from_=0.0, to=1.0,
                              resolution=0.05, orient=tk.HORIZONTAL)
        conf_scale.grid(row=row, column=1, sticky="ew")
        tk.Label(config_frame, text=f"{self.confidence_threshold.get():.2f}").grid(row=row, column=2,
padx=5, sticky="w")
        conf_scale.config(command=lambda val: self.update_label(conf_scale, row, 2,
f"{float(val):.2f}"))

        row += 1
        tk.Label(config_frame, text="IOU Threshold").grid(row=row, column=0, sticky="w", pady=2)
        iou_scale = tk.Scale(config_frame, variable=self.iou_threshold, from_=0.0, to=1.0,
                             resolution=0.05, orient=tk.HORIZONTAL)
        iou_scale.grid(row=row, column=1, sticky="ew")
        tk.Label(config_frame, text=f"{self.iou_threshold.get():.2f}").grid(row=row, column=2, padx=5,
sticky="w")
        iou_scale.config(command=lambda val: self.update_label(iou_scale, row, 2, f"{float(val):.2f}"))

        row += 1
```

```
        tk.Label(config_frame, text="Downsample Factor").grid(row=row, column=0, sticky="w",
pady=2)
        down_scale = tk.Scale(config_frame, variable=self.downsample_factor, from_=0.1, to=1.0,
                              resolution=0.1, orient=tk.HORIZONTAL)
        down_scale.grid(row=row, column=1, sticky="ew")
        tk.Label(config_frame, text=f"{self.downsample_factor.get():.1f}").grid(row=row, column=2,
padx=5, sticky="w")
        down_scale.config(command=lambda val: self.update_label(down_scale, row, 2,
f"{float(val):.1f}"))

        row += 1
        tk.Label(config_frame, text="Skip Frames").grid(row=row, column=0, sticky="w", pady=2)
        skip_scale = tk.Scale(config_frame, variable=self.skip_frames, from_=0, to=10,
                              resolution=1, orient=tk.HORIZONTAL)
        skip_scale.grid(row=row, column=1, sticky="ew")
        tk.Label(config_frame, text=f"{self.skip_frames.get()}").grid(row=row, column=2, padx=5,
sticky="w")
        skip_scale.config(command=lambda val: self.update_label(skip_scale, row, 2,
f"{int(float(val))}"))

        # Advanced settings
        row = 0
        options_frame = tk.Frame(advanced_frame)
        options_frame.pack(fill=tk.X, expand=True, pady=5)

        tk.Checkbutton(options_frame, text="Use FP16 (half precision)",
variable=self.use_fp16).pack(side=tk.LEFT, padx=5)
        tk.Checkbutton(options_frame, text="Annotate Frames",
variable=self.annotate_frames).pack(side=tk.LEFT, padx=5)

        threading_frame = tk.Frame(advanced_frame)
        threading_frame.pack(fill=tk.X, expand=True, pady=5)

        tk.Label(threading_frame, text="Processing Threads:").pack(side=tk.LEFT, padx=5)
        tk.Spinbox(threading_frame, from_=1, to=16, textvariable=self.num_threads,
width=5).pack(side=tk.LEFT, padx=5)

        tk.Label(threading_frame, text="Queue Size:").pack(side=tk.LEFT, padx=5)
        tk.Spinbox(threading_frame, from_=8, to=64, textvariable=self.queue_size,
width=5).pack(side=tk.LEFT, padx=5)

        self.use_mmap = BooleanVar(value=True)
        tk.Checkbutton(options_frame, text="Use Memory Mapping",
variable=self.use_mmap).pack(side=tk.LEFT, padx=5)

        # Info section
        info_text = "This tool processes video using YOLO object detection with tracking.\n" + \
                    "Higher confidence threshold = fewer detections but more accurate.\n" + \
                    "Lower downsample factor = faster processing but less accurate."
```

```
        info_label = tk.Label(advanced_frame, text=info_text, justify=tk.LEFT,
                              fg="gray", font=("Arial", 8))
        info_label.pack(fill=tk.X, expand=True, pady=5)

        # Process button
        self.process_btn = tk.Button(action_frame, text="Start Processing",
command=self.start_processing,
                                     bg="#4CAF50", fg="white", height=2)
        self.process_btn.pack(fill=tk.X, expand=True)

        # Status bar
        self.status_var = tk.StringVar(value="Ready")
        status = tk.Label(self, textvariable=self.status_var, bd=1, relief=tk.SUNKEN, anchor=tk.W)
        status.pack(side=tk.BOTTOM, fill=tk.X)

        # Configure grid weights
        file_frame.columnconfigure(1, weight=1)
        config_frame.columnconfigure(1, weight=1)

    def update_label(self, scale_widget, row, col, text):
        """Updates the label next to a scale widget."""
        scale_widget.master.grid_slaves(row=row, column=col)[0].config(text=text)

    def browse_source_weights(self):
        """Browses for the source weights file."""
        file_path = filedialog.askopenfilename(filetypes=[("PyTorch Models", "*.pt"), ("All Files",
"*.*")])
        if file_path:
            self.source_weights_path.set(file_path)
            self.status_var.set(f"Selected weights: {os.path.basename(file_path)}")

    def browse_source_video(self):
        """Browses for the source video file."""
        file_path = filedialog.askopenfilename(
            filetypes=[("Video Files", "*.mp4 *.avi *.mov *.mkv"), ("All Files", "*.*")])
        if file_path:
            self.source_video_path.set(file_path)
            self.status_var.set(f"Selected source video: {os.path.basename(file_path)}")

            # Auto-fill target path with _processed suffix
            if not self.target_video_path.get():
                base, ext = os.path.splitext(file_path)
                self.target_video_path.set(f"{base}_processed{ext}")

    def browse_target_video(self):
        """Browses for the target video file."""
        file_path = filedialog.asksaveasfilename(
            defaultextension=".mp4",
            filetypes=[("MP4 Files", "*.mp4"), ("AVI Files", "*.avi"), ("All Files", "*.*")])
```

```
        if file_path:
            self.target_video_path.set(file_path)
            self.status_var.set(f"Selected target path: {os.path.basename(file_path)}")

    def start_processing(self):
        """Starts processing the video."""
        # Validate inputs
        if not self.source_weights_path.get():
            tk.messagebox.showwarning("Missing Input", "Please provide YOLO model weights path.")
            return

        if not self.source_video_path.get():
            tk.messagebox.showwarning("Missing Input", "Please provide source video path.")
            return

        if not os.path.exists(self.source_weights_path.get()):
            tk.messagebox.showerror("File Not Found", f"Model weights file not found:
{self.source_weights_path.get()}")
            return

        if not os.path.exists(self.source_video_path.get()):
            tk.messagebox.showerror("File Not Found", f"Source video file not found:
{self.source_video_path.get()}")
            return

        # Check if output directory exists
        if self.target_video_path.get():
            output_dir = os.path.dirname(self.target_video_path.get())
            if output_dir and not os.path.exists(output_dir):
                try:
                    os.makedirs(output_dir)
                except Exception as e:
                    tk.messagebox.showerror("Error", f"Cannot create output directory: {str(e)}")
                    return

        # Create progress window
        self.progress_window = tk.Toplevel(self.master)
        self.progress_window.title("Processing Video")
        self.progress_window.geometry("400x150")
        self.progress_window.resizable(False, False)
        self.progress_window.transient(self.master)
        self.progress_window.grab_set()

        # Add widgets to progress window
        frame = tk.Frame(self.progress_window, padx=10, pady=10)
        frame.pack(fill=tk.BOTH, expand=True)

        tk.Label(frame, text="Processing video. Please wait...", font=("Arial", 10, "bold")).pack(pady=5)
```

```
        tk.Label(frame, text="This may take several minutes depending on the video length and
settings.").pack()

        # Progress info
        info_frame = tk.Frame(frame)
        info_frame.pack(fill=tk.X, expand=True, pady=5)

        self.progress_status = tk.StringVar(value="Initializing...")
        tk.Label(info_frame, textvariable=self.progress_status).pack(side=tk.LEFT)

        self.cancel_button = tk.Button(frame, text="Cancel", command=self.cancel_processing)
        self.cancel_button.pack(pady=10)

        # Disable main window controls
        self.process_btn.config(state=tk.DISABLED)
        self.status_var.set("Processing video...")

        # Start processing in a separate thread to keep UI responsive
        self.master.after(100, self._run_processing)

    def cancel_processing(self):
        """Cancels the current processing operation."""
        if self.current_process and hasattr(self.current_process, 'stop_event'):
            self.current_process.stop_event.set()
            self.progress_status.set("Canceling... Please wait.")
            self.cancel_button.config(state=tk.DISABLED)

    def _run_processing(self):
        """Run the processing operation in a background thread."""
        try:
            # Create processor
            self.current_process = VideoProcessor_Optimize(
                source_weights_path=self.source_weights_path.get(),
                source_video_path=self.source_video_path.get(),
                target_video_path=self.target_video_path.get(),
                confidence_threshold=self.confidence_threshold.get(),
                iou_threshold=self.iou_threshold.get(),
                downsample_factor=self.downsample_factor.get(),
                skip_frames=self.skip_frames.get(),
                use_fp16=self.use_fp16.get(),
                annotate_frames=self.annotate_frames.get(),
                num_threads=self.num_threads.get(),
                queue_size=self.queue_size.get(),
            )

            modify_frame_reader_to_use_mmap(self.current_process)

            # Start monitoring thread for updating progress
            self.monitor_thread = threading.Thread(target=self._monitor_progress)
```

```
            self.monitor_thread.daemon = True
            self.monitor_thread.start()

            # Start processing in a separate thread
            self.process_thread = threading.Thread(target=self._process_video_thread)
            self.process_thread.daemon = True
            self.process_thread.start()

        except Exception as e:
            self._processing_error(f"Failed to start processing: {str(e)}")

    def _process_video_thread(self):
        """Thread function to run video processing."""
        try:
            self.current_process.process_video()

            # Update UI on main thread
            if not self.current_process.stop_event.is_set():
                self.master.after(0, self._processing_complete)
            else:
                self.master.after(0, self._processing_canceled)

        except Exception as e:
            self.master.after(0, lambda: self._processing_error(str(e)))

    def _monitor_progress(self):
        """Monitor progress and update UI."""
        while self.current_process and not self.current_process.stop_event.is_set():
            # Access frame counts safely
            with self.current_process.stats_lock:
                total = self.current_process.frame_count
                processed = self.current_process.processed_count

            if total > 0:
                status = f"Processing: {processed}/{total} frames"
                self.master.after(0, lambda s=status: self.progress_status.set(s))

            time.sleep(0.5)

    def _processing_complete(self):
        """Called when processing completes successfully."""
        if self.current_process:
            duration = time.time() - self.current_process.start_time
            peak_memory = self.current_process.memory_monitor.get_peak()

            # Close progress window
            if self.progress_window:
                self.progress_window.destroy()
                self.progress_window = None
```

```
        # Show completion message
        tk.messagebox.showinfo(
            "Processing Complete",
            f"Video processing completed successfully!\n\n"
            f"Processed {self.current_process.processed_count} frames in "
            f"{duration:.2f} seconds ({self.current_process.processed_count / duration:.2f} FPS).\n\n"
            f"Peak memory usage: {peak_memory:.2f} MB"
        )

        # Clean up
        self._cleanup_processing()

        # Update status
        self.status_var.set("Processing complete")

    def _processing_canceled(self):
        """Called when processing is canceled."""
        # Close progress window
        if self.progress_window:
            self.progress_window.destroy()
            self.progress_window = None

        tk.messagebox.showinfo("Canceled", "Video processing was canceled.")

        # Clean up
        self._cleanup_processing()

        # Update status
        self.status_var.set("Processing canceled")

    def _processing_error(self, error_msg):
        """Called when an error occurs during processing."""
        # Close progress window
        if self.progress_window:
            self.progress_window.destroy()
            self.progress_window = None

        tk.messagebox.showerror("Error", f"An error occurred during processing:\n\n{error_msg}")

        # Clean up
        self._cleanup_processing()

        # Update status
        self.status_var.set(f"Error: {error_msg[:30]}...")

    def _cleanup_processing(self):
        """Clean up after processing completes, errors, or is canceled."""
        # Clean up processor
```

```
        if self.current_process:
            self.current_process.cleanup()
            self.current_process = None

        # Re-enable controls
        self.process_btn.config(state=tk.NORMAL)


if __name__ == "__main__":
    root = tk.Tk()
    app = Application(master=root)
    app.mainloop()
```

Optimization Code:

1. Process Priority and Scheduling:

- **OS Level:** You would use OS-specific commands or tools to adjust the priority of the Python process running your YOLO application. For example, on Linux, you might use the nice or chrt commands. Real-time scheduling policies (like SCHED_FIFO or SCHED_RR on Linux) can be configured for critical processes.
- **Python Level (Indirectly):** While you don't set OS-level priority directly in Python, you can design your application to have separate processes for time-critical tasks. You could then use libraries like multiprocessing to spawn these processes and manage their communication. You would then adjust the priority of these *processes* at the OS level.

```
import multiprocessing
import time

def video_processing_task():
    # Your YOLO inference and traffic analysis code
    print("Processing video frame...")
    time.sleep(0.1) # Simulate processing time

if __name__ == "__main__":
    processing_process = multiprocessing.Process(target=video_processing_task)
    processing_process.start()
    # You would then use OS commands to prioritize 'processing_process'
    # (e.g., using its PID)
    print("Main application running...")
    processing_process.join()
```

2. CPU Affinity:

- **OS Level:** You would use OS-specific tools (like taskset on Linux or process affinity settings in Windows Task Manager or via system calls) to bind specific threads or processes of your YOLO application to particular CPU cores. This can improve cache locality and reduce context switching.
- **Python Level (Indirectly):** Libraries like multiprocessing allow you to create and manage processes. You can then use OS-level tools to set the CPU affinity for these processes. Some Python libraries like psutil provide ways to get process information that can be used with OS-level commands.

```
import multiprocessing
import os
import time

def worker():
    print(f"Worker process running on CPU {os.sched_getaffinity(0)}")
    time.sleep(1)

if __name__ == "__main__":
    num_cores = multiprocessing.cpu_count()
    processes = []
    for i in range(num_cores):
        p = multiprocessing.Process(target=worker)
        processes.append(p)
        p.start()
        # You would then use OS commands (e.g., taskset -c i p.pid)
        # to bind each process to a specific core

    for p in processes:
        p.join()
```

3. Real-Time Libraries and System Calls (Less Common in Standard Python):

- **OS Level:** This involves having a real-time kernel or using specific OS APIs for precise timing and control.
- **Python Level (Limited):** Standard Python has limitations in achieving hard real-time guarantees due to the Global Interpreter Lock (GIL) and garbage collection. However, libraries like sched can be used for event scheduling, and for more demanding real-time tasks, you might consider:
    - **Cython or C Extensions:** Writing performance-critical parts of your application in C or using Cython to compile Python-like code to C can give you more control and the ability to interact with lower-level system APIs.
    - **Specialized Real-Time Python Distributions:** Some specialized Python distributions aim to address real-time constraints.

4. Hardware Acceleration (GPU):

- **OS Level:** Ensuring the correct drivers for your GPU are installed and configured is crucial.
- **Python Level (Directly):** Deep learning frameworks like TensorFlow and PyTorch have extensive support for GPU acceleration. You would write your YOLO model and inference code using these frameworks, and they will handle the offloading of computations to the GPU. You need to ensure these frameworks are built with GPU support (e.g., CUDA for NVIDIA GPUs).

```
import torch
if torch.cuda.is_available():
    device = torch.device("cuda")
    print("CUDA is available. Using GPU.")
else:
    device = torch.device("cpu")
    print("CUDA not available. Using CPU.")

# Load your YOLO model and data, and move them to the selected device
model = torch.hub.load('ultralytics/yolov5', 'yolov5s', pretrained=True).to(device)
# ... your inference code ...
```

5. Memory Management (Influenced by OS):

- **OS Level:** The OS's virtual memory manager, swapping mechanisms, and memory allocation strategies play a significant role. Tuning these (e.g., swapiness on Linux) can impact performance. Using large/huge pages can sometimes improve memory access times for large datasets.
- **Python Level (Indirectly):** While you don't directly control the OS's memory management from Python, you can write your Python code to be memory-efficient:
    - **Avoid unnecessary object creation.**
    - **Use data structures that are appropriate for the task.**
    - **Explicitly release resources when they are no longer needed.**
    - **Be mindful of large data copies.**
    - Libraries like mmap can be used to work with large files without loading them entirely into memory.

Code (Performance Test)

```
import time
import psutil
import tracemalloc
import os
import gc
import matplotlib.pyplot as plt
import numpy as np
from main import VideoProcessor
from main_optimize import VideoProcessor_Optimize

def plot_comparison(original_results, optimized_results, orig_trials=None, opt_trials=None):
    # Bar plots (existing)
    fig, axs = plt.subplots(3, 2, figsize=(14, 15))

    # Bar plot: Execution Time
    axs[0, 0].bar(['Original', 'Optimized'],
                  [original_results['execution_time'], optimized_results['execution_time']], color=['skyblue',
'lightgreen'])
    axs[0, 0].set_title('Average Execution Time (s)')
    axs[0, 0].set_ylabel('Seconds')

    # Bar plot: Peak Memory Usage
    axs[0, 1].bar(['Original', 'Optimized'],
                  [original_results['peak_memory'], optimized_results['peak_memory']], color=['skyblue',
'lightgreen'])
    axs[0, 1].set_title('Average Peak Memory Usage (MB)')
    axs[0, 1].set_ylabel('MB')

    # Bar plot: CPU Time
    axs[1, 0].bar(['Original User', 'Optimized User', 'Original System', 'Optimized System'],
```

```
            [original_results['user_cpu_time'], optimized_results['user_cpu_time'],
             original_results['system_cpu_time'], optimized_results['system_cpu_time']],
            color=['skyblue', 'lightgreen', 'deepskyblue', 'lightseagreen'])
    axs[1, 0].set_title('Average CPU Usage (s)')
    axs[1, 0].set_ylabel('Seconds')

    # Bar plot: Memory Increase
    axs[1, 1].bar(['Original', 'Optimized'],
            [original_results['memory_increase'], optimized_results['memory_increase']],
color=['skyblue', 'lightgreen'])
    axs[1, 1].set_title('Average Memory Usage Increase (MB)')
    axs[1, 1].set_ylabel('MB')

    # Line plot: Execution Time per Trial
    if orig_trials and opt_trials:
        axs[2, 0].plot(orig_trials['execution_time'], label='Original', marker='o')
        axs[2, 0].plot(opt_trials['execution_time'], label='Optimized', marker='s')
        axs[2, 0].set_title('Execution Time per Trial')
        axs[2, 0].set_ylabel('Seconds')
        axs[2, 0].set_xlabel('Trial')
        axs[2, 0].legend()

        # Boxplot: Memory Increase Spread
        axs[2, 1].boxplot([orig_trials['memory_increase'], opt_trials['memory_increase']],
labels=['Original', 'Optimized'])
        axs[2, 1].set_title('Memory Increase Spread per Trial')
        axs[2, 1].set_ylabel('MB')

    plt.tight_layout()
    plt.savefig('performance_comparison_extended.png')
    plt.show()

    # Radar chart (optional, separate figure)
    labels = ['Execution Time', 'Peak Memory', 'Memory Increase', 'User CPU', 'System CPU']
    orig_values = [original_results[k] for k in ['execution_time', 'peak_memory', 'memory_increase',
'user_cpu_time', 'system_cpu_time']]
    opt_values = [optimized_results[k] for k in ['execution_time', 'peak_memory', 'memory_increase',
'user_cpu_time', 'system_cpu_time']]

    angles = np.linspace(0, 2 * np.pi, len(labels), endpoint=False).tolist()
    orig_values += orig_values[:1]
    opt_values += opt_values[:1]
    angles += angles[:1]

    fig = plt.figure(figsize=(6, 6))
    ax = fig.add_subplot(111, polar=True)
    ax.plot(angles, orig_values, 'o-', label='Original')
    ax.plot(angles, opt_values, 'o-', label='Optimized')
    ax.fill(angles, orig_values, alpha=0.25)
```

```
    ax.fill(angles, opt_values, alpha=0.25)
    ax.set_xticks(angles[:-1])
    ax.set_xticklabels(labels)
    ax.set_title('Radar Chart: Performance Metrics')
    ax.legend(loc='upper right')
    plt.savefig('performance_radar_chart.png')
    plt.show()


def measure_performance(processor_class, source_weights_path, source_video_path,
target_video_path,
                        name="Original", trials=1):
    results = {
        "execution_time": [],
        "memory_increase": [],
        "peak_memory": [],
        "user_cpu_time": [],
        "system_cpu_time": []
    }

    for trial in range(trials):
        # Force garbage collection before each trial
        gc.collect()

        # Start memory and time tracking
        process = psutil.Process(os.getpid())
        start_cpu_times = process.cpu_times()
        start_time = time.time()
        tracemalloc.start()
        start_mem = process.memory_info().rss

        # Instantiate and run processor
        processor = processor_class(
            source_weights_path=source_weights_path,
            source_video_path=source_video_path,
            target_video_path=target_video_path,
            confidence_threshold=0.4,
            iou_threshold=0.5,
        )

        try:
            processor.process_video()
        finally:
            # Clean up resources if the optimized version
            if hasattr(processor, 'cleanup'):
                processor.cleanup()

        # Stop memory and time tracking
        end_time = time.time()
```

```
        end_mem = process.memory_info().rss
        end_cpu_times = process.cpu_times()
        current, peak = tracemalloc.get_traced_memory()
        tracemalloc.stop()

        # Store results
        results["execution_time"].append(end_time - start_time)
        results["memory_increase"].append((end_mem - start_mem) / 1024**2)  # MB
        results["peak_memory"].append(peak / 1024**2)  # MB
        results["user_cpu_time"].append(end_cpu_times.user - start_cpu_times.user)
        results["system_cpu_time"].append(end_cpu_times.system - start_cpu_times.system)

        print(f"Trial {trial+1}/{trials} completed for {name}")

    # Calculate averages
    avg_results = {k: np.mean(v) for k, v in results.items()}
    std_results = {k: np.std(v) for k, v in results.items()}

    print(f"\n=== Performance Report [{name}] ===")
    print(f"Total Execution Time     : {avg_results['execution_time']:.2f} ±
{std_results['execution_time']:.2f} seconds")
    print(f"Memory Increase          : {avg_results['memory_increase']:.2f} ±
{std_results['memory_increase']:.2f} MB")
    print(f"Peak Memory Usage        : {avg_results['peak_memory']:.2f} ±
{std_results['peak_memory']:.2f} MB")
    print(f"User CPU Time            : {avg_results['user_cpu_time']:.2f} ±
{std_results['user_cpu_time']:.2f} seconds")
    print(f"System CPU Time          : {avg_results['system_cpu_time']:.2f} ±
{std_results['system_cpu_time']:.2f} seconds")

    return avg_results, std_results

if __name__ == "__main__":
    # Use the same files for both versions to ensure fair comparison
    YOLO_WEIGHTS = "data/traffic_analysis.pt"
    SOURCE_VIDEO = "data/traffic_analysis.mov"
    OUTPUT_ORIG = "data/traffic_analysis_original.mov"
    OUTPUT_OPT = "data/traffic_analysis_optimized.mov"

    # Number of trials for each version
    TRIALS = 3

    print("Running original version...")
    orig_avg, orig_std = measure_performance(
        VideoProcessor,
        YOLO_WEIGHTS, SOURCE_VIDEO, OUTPUT_ORIG,
        name="Original",
        trials=TRIALS
    )
```

```
    print("\nRunning optimized version...")
    opt_avg, opt_std = measure_performance(
        VideoProcessor_Optimize,
        YOLO_WEIGHTS, SOURCE_VIDEO, OUTPUT_OPT,
        name="Optimized",
        trials=TRIALS
    )

    # Calculate improvement percentages
    time_improvement = (orig_avg['execution_time'] - opt_avg['execution_time']) /
orig_avg['execution_time'] * 100
    memory_improvement = (orig_avg['peak_memory'] - opt_avg['peak_memory']) /
orig_avg['peak_memory'] * 100

    print("\n=== Improvement Summary ===")
    print(f"Execution Time Reduction  : {time_improvement:.2f}%")
    print(f"Peak Memory Reduction     : {memory_improvement:.2f}%")

    # Create visual comparison
    plot_comparison(orig_avg, opt_avg)
```

References

[1] Roboflow, supervision, GitHub repository, 2023. [Online]. Available: https://github.com/roboflow/supervision

[2] S. Nitjaphant, traffic_analysis, GitHub repository, 2024. [Online]. Available: https://github.com/sahatsawat-pks/traffic_analysis

[3] Lecture Slide and Lab hand-on from subject ITCS225 – Principles of Operating Systems, 2025.