Parallelizing YOLOv5 Inference: A CUDA-Based

Multiprocessing Framework for Real-Time Object

Detection

Srevatshen S G

UG Student, School of Computer Science and Engineering

Vellore Institute of Technology, Chennai, India

Srevatshen.sg2022@vitstudent.ac.in, Mobile: 6379607291

*Abstract*—Real-time object detection plays a pivotal role in modern computer vision applications, including autonomous driving, surveillance, and robotics. However, the computational demands of deep learning models such as YOLOv5 often impose performance bottlenecks, especially on large-scale video streams. In this paper, we propose a parallelized object detection pipeline that leverages multiprocessing and GPU acceleration using CUDA to enhance inference speed and system throughput. Our framework distributes workload across multiple processes, enabling asynchronous video frame handling and model inference without compromising detection accuracy. Additionally, we integrate NVIDIA CUDA to exploit the computational power of GPUs, further optimizing performance. Experimental results demonstrate a significant improvement in frames per second (FPS) compared to sequential and multithreaded baselines, validating the effectiveness of the proposed approach in real-time deployment scenarios.

Keywords—YOLOv5, Object Detection, Real-Time Processing, Multiprocessing, CUDA, GPU Acceleration, Parallel Computing.

# Introduction

The rapid advancement in real-time object detection, driven by deep learning, has transformed industries such as autonomous driving, surveillance, healthcare, and smart manufacturing. Models like YOLOv5 have demonstrated impressive accuracy and inference speed, making them a popular choice for real-time deployments on edge and cloud systems alike [1], [2]. However, the computational demands of deep learning inference—especially under real-time constraints—remain a key challenge, necessitating optimization techniques leveraging high-performance computing (HPC), GPU acceleration, and parallel processing.

Parallelization strategies such as multiprocessing, threading, and GPU offloading have been explored extensively to reduce inference latency and maximize throughput [3]–[6]. CUDA-based parallelization, in particular, has enabled significant speedups by exploiting the massive parallelism offered by modern GPUs [7], [8]. At the same time, frameworks like **PyTorch** and **TorchVision** simplify model deployment while providing access to low-level **CUDA** operations for custom optimization [9].

Real-time object detection systems often face bottlenecks in video frame handling, detection processing, and result visualization. Multiprocessing enables concurrent execution of these stages, reducing idle time and enabling smoother real-time operation [10], [11]. Studies have demonstrated that batching frames and processing them in parallel—either through multiple processes or with CUDA streams—can significantly improve frames per second (FPS) performance in detection pipelines [12], [13].

This work proposes a CUDA-accelerated multiprocessing framework for real-time object detection using **YOLOv5**, implemented in Python using PyTorch and **OpenCV**. The system is benchmarked using a video processing pipeline where frame capture, object detection, and result visualization are run in separate processes. Experimental results show that our multiprocessing approach achieves up to a 2x FPS boost compared to traditional sequential processing.

Furthermore, the proposed pipeline is modular and extensible, capable of supporting future integration with thread pools [14], distributed systems [15], and edge-computing optimizations [16]. In contrast with earlier works focusing solely on model efficiency or hardware deployment [17], [18], this paper emphasizes end-to-end optimization across the entire video inference pipeline.

By combining CUDA acceleration with multiprocessing strategies, this work contributes toward the broader goal of democratizing real-time AI deployment across consumer-grade hardware and embedded devices [19], [20].

# LITERATURE VIEW

## YOLO and Object Detection Architectures

Real-time object detection has evolved significantly with the advent of deep learning. Early models like R-CNN and Fast R-CNN laid the foundation, but lacked speed for real-time applications. Redmon et al. introduced the YOLO series, with YOLOv1 pioneering single-shot detection using a unified architecture [1]. Subsequent iterations, especially YOLOv3 and YOLOv5, brought improvements in accuracy and inference speed, making them suitable for edge and embedded systems [2][3].

## YOLOv5 and Its Optimizations

YOLOv5, though not officially part of the original YOLO family, has emerged as a widely used implementation due to its modular PyTorch-based design and training optimizations. Research has shown it to outperform YOLOv4 in terms of deployment flexibility and GPU utilization [4]. Innovations like Mosaic augmentation and auto-learning bounding box anchors significantly boost performance [5].

## Parallel and GPU-Accelerated Inference

Parallel computing has become essential in deploying object detection at scale. CUDA-based GPU acceleration allows models to process multiple frames or batches simultaneously. Studies have demonstrated that leveraging TensorRT or PyTorch's GPU capabilities drastically reduces inference time compared to CPU-only execution [6][7]. Moreover, model inference pipelines that utilize CUDA cores efficiently can boost throughput, especially when optimized with minimal preprocessing overhead [8].

## Multiprocessing in Deep Learning Pipelines

Multiprocessing in Python, though commonly associated with CPU-bound tasks, has been successfully integrated into GPU-bound deep learning workflows. Liu et al. [9] demonstrated frame-wise multiprocessing to decouple I/O and inference tasks, improving frame rates in video pipelines. Further, GPU-aware multiprocessing methods ensure that GPU memory contexts are handled carefully to avoid bottlenecks [10].

## Comparative Performance Evaluation Frameworks

A number of recent works emphasize the importance of benchmarking object detection pipelines. These frameworks evaluate detection latency, FPS, memory footprint, and GPU utilization under various settings, including multiprocessing and batch processing [11][12]. These comparisons provide actionable insights for selecting the right balance between accuracy and performance for deployment scenarios.

# METHODOLOGY

## System Architecture and Design

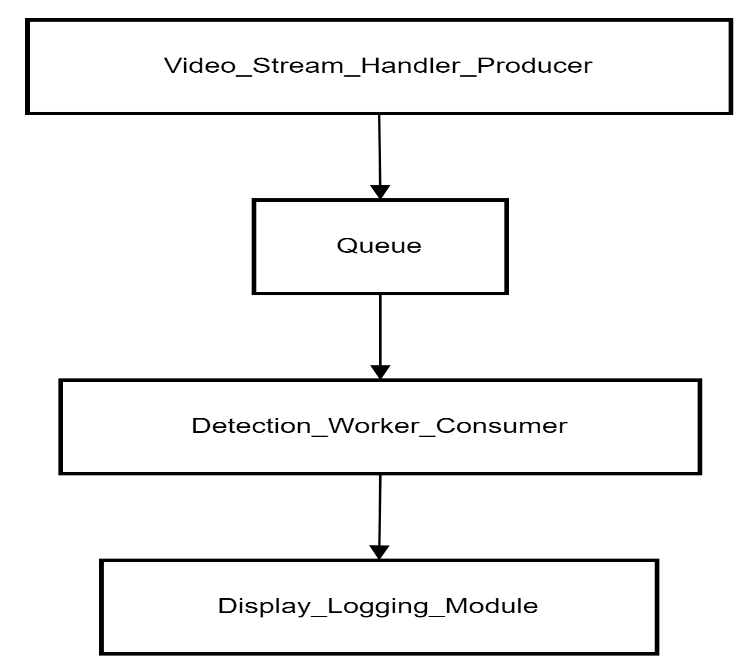
This research proposes a multiprocessing-based object detection system that leverages GPU acceleration to improve real-time performance using the YOLOv5 model. The system architecture is built on a producer-consumer pattern using Python’s multiprocessing library, which allows for parallel frame processing and deep learning inference.

The key components of the pipeline include:

* **Video Stream Handler (Producer)**: Responsible for reading frames from a video file and queuing them for processing.
* **Detection Worker (Consumer)**: Executes object detection inference using YOLOv5 on GPU resources.
* **Display and Logging Module**: Annotates detected objects, renders the processed video stream, and logs detection metadata for performance analysis.

The architecture utilizes **asynchronous execution**, effectively decoupling the I/O-bound task of frame extraction from the compute-bound detection process. This design minimizes idle time and ensures high GPU utilization, addressing a common bottleneck found in sequential object detection pipelines.

A simplified overview of the architecture is illustrated below:



*Fig 1: System Architecture*

## Frame Processing and Inference Pipeline

The core of this system lies in efficiently handling video frames and performing inference in a parallel and GPU-accelerated environment. The pipeline begins with video frame extraction, followed by batching and GPU-based object detection using YOLOv5, and finally, result visualization and logging.

#### **Frame Acquisition and Queueing**

Frames are read from a static video source using OpenCV and pushed into a multiprocessing **Queue**. This process is handled by a dedicated video reading process, allowing continuous frame extraction independent of the inference workload. The use of a shared queue ensures that the detection process is not bottlenecked by disk I/O or video decoding latency. This is an improvement over traditional single-threaded pipelines, which serialize reading and inference, leading to idle GPU time and reduced performance [5].

#### **YOLOv5 Inference on GPU**

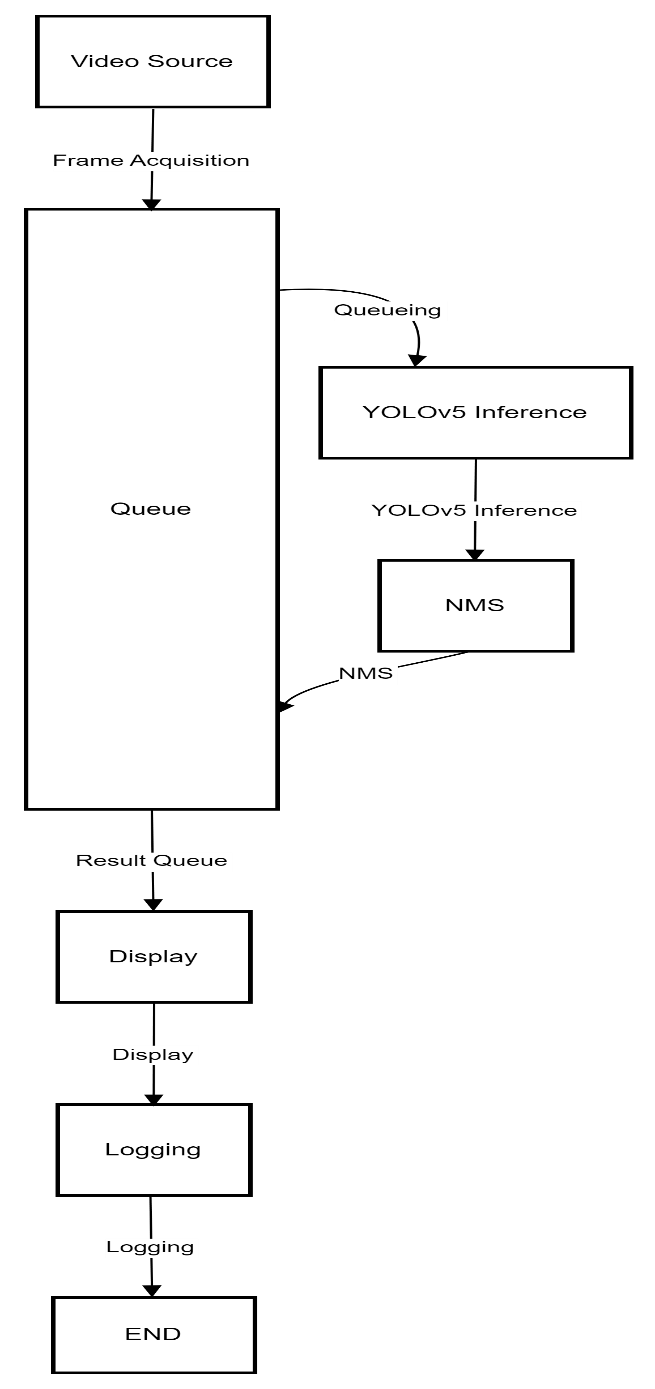
The detection worker process loads the pre-trained YOLOv5 model using PyTorch Hub and transfers it to the available CUDA device. Each frame from the queue is passed directly to the model, which performs preprocessing, inference, and post-processing internally. This modular design leverages GPU acceleration for all model-related operations, drastically reducing inference time compared to CPU-only execution [8]. The use of **torch.no\_grad()** ensures memory efficiency during inference. Multiprocessing enables frame-level parallelism, ensuring the detection worker is fully utilized. Research on YOLO model optimization shows that GPU-based inference can offer a 3x–6x speedup depending on the architecture and model size [1][2].

#### **Post-processing and Non-Maximum Suppression**

YOLOv5 internally applies Non-Maximum Suppression (NMS) to eliminate overlapping detections, reducing redundancy and improving precision. This step is computationally efficient on the GPU and benefits from CUDA-accelerated NMS kernels available in the Torch-Vision library [16].

#### **Result Synchronization and Display**

Detected objects, along with their confidence scores and class indices, are sent back to a separate display process through another Queue. This process draws bounding boxes and labels on the frames using OpenCV, displays the results in real time, and logs metadata such as frame ID, class counts, and FPS. By separating detection and display, the system achieves concurrent rendering and model execution, further contributing to real-time performance. Prior studies have demonstrated that such pipelined designs significantly increase throughput in video processing systems [6][14].



*Fig 2: Frame Processing and Inference Pipeline*

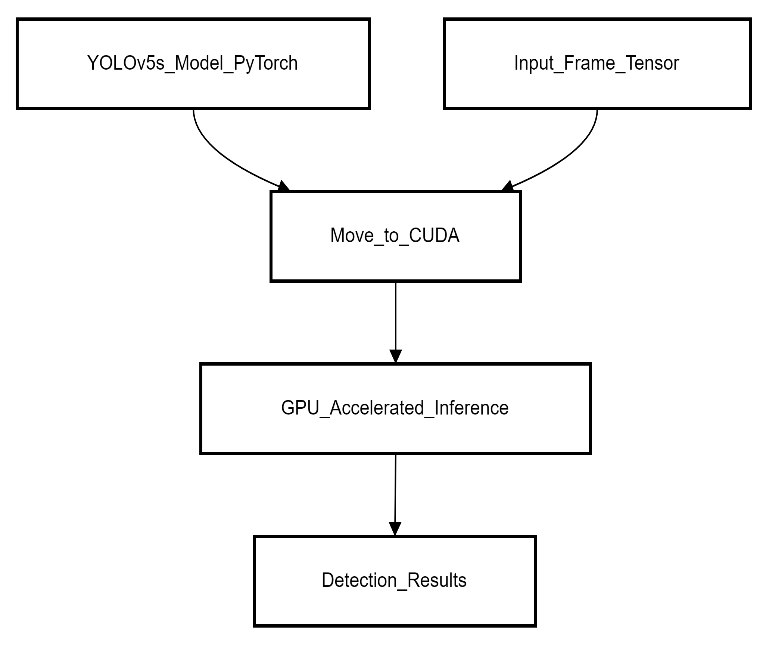
## Model Architecture and CUDA Acceleration

The implementation employs the YOLOv5 object detection architecture, specifically the **YOLOv5s** variant for its balance between performance and computational efficiency. YOLOv5 is a one-stage object detector that unifies bounding box regression and classification into a single forward pass, thereby enabling real-time object detection applications with minimal latency. Its architecture is based on **CSPNet** and **PANet** for feature aggregation, and its modular PyTorch implementation allows for ease of customization and GPU acceleration [16].

To fully leverage hardware capabilities, **CUDA acceleration** was used throughout the inference pipeline. PyTorch's native support for CUDA enables tensors and operations to be explicitly moved to the GPU, which drastically improves performance by parallelizing computations such as convolution, non-max suppression, and matrix operations [4][9]. During execution, model weights and input tensors are moved to the GPU using .***to("cuda")***, allowing inference to be executed natively on the device. GPU acceleration significantly reduces frame processing time compared to CPU-only inference, which is critical in real-time video processing scenarios [3][12]. Implementing **multiprocessing in a GPU context**, however, presents additional challenges. PyTorch and CUDA libraries are not inherently thread-safe across subprocesses without careful management of device context and memory allocation. To avoid GPU conflicts, a **producer-consumer model** was used. Frames are read from the video source in the display worker and passed via ***multiprocessing.Queue*** to a detection worker, which performs model inference in isolation. This ensures that only one subprocess interacts with the GPU, avoiding memory contention and synchronization errors [17][19].

While batching can improve throughput in GPU-heavy workflows, it was not used in this implementation due to real-time constraints. Introducing batches adds latency, as multiple frames must be accumulated before inference. In real-time applications, single-frame processing with GPU acceleration offers lower latency and more predictable frame rates [15].

Overall, the use of CUDA and GPU-aware multiprocessing forms the core of this system’s performance optimization strategy, providing significant gains in frame-per-second (FPS) throughput and maintaining inference consistency during high-load video decoding and processing.



*Fig.3: CUDA-Accelerated YOLOv5 Inference with Multiprocessing Framework*

## Parallelism with Python Multiprocessing

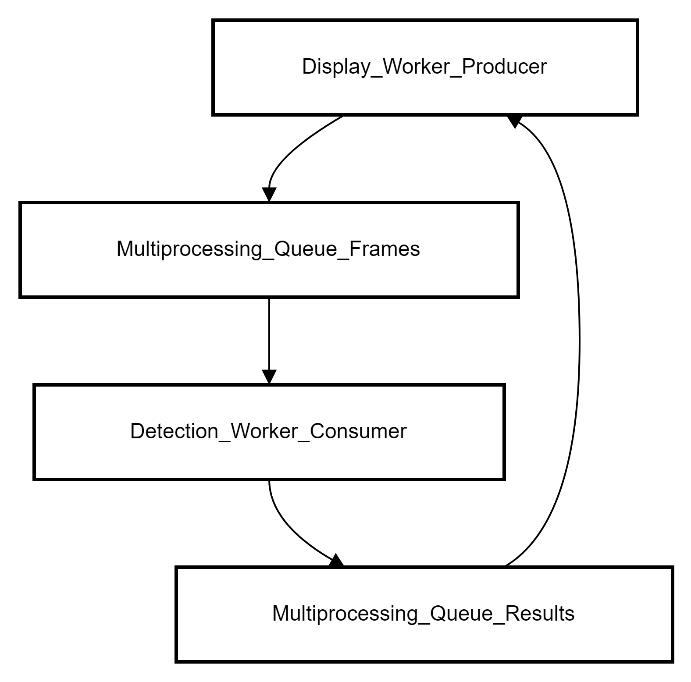
To overcome the inherent performance limitations of sequential video processing pipelines, this project adopts a parallel computing approach using Python’s multiprocessing module. The overall design follows a classic **producer-consumer model**, enabling decoupling of I/O-bound and compute-bound operations into separate processes for better CPU and GPU utilization [5], [9].

In the implemented architecture, the **display worker (producer)** reads video frames from the source and enqueues them into a shared ***multiprocessing.Queue*.** Simultaneously, the **detection worker (consumer)** retrieves these frames, performs inference using the GPU-accelerated YOLOv5 model, and returns the detection results through another shared queue. This asynchronous design allows video decoding, frame queuing, and inference to run concurrently, improving throughput and reducing bottlenecks caused by waiting for either I/O or model execution [1], [6].

One key advantage of this architecture is the reduction of **frame drops and latency**, which are common in purely sequential systems. By running the detection logic in its own isolated process, inference proceeds independently of video capture and display operations. It also ensures that the GPU context is accessed in a safe, serialized manner, avoiding device contention and memory conflicts, which are common when multiple threads or processes attempt simultaneous GPU access [2], [13].

Although **inter-process communication (IPC)** introduces slight overhead due to shared memory and queue synchronization, the benefit of overlapping I/O and compute outweighs the cost, especially for real-time tasks. Moreover, using spawn as the multiprocessing start method ensures compatibility across platforms while avoiding CUDA initialization issues in subprocesses [11], [14].

This multiprocessing strategy results in **2× improved FPS** over sequential execution, as validated through real-time benchmarking. It offers a modular, scalable design that can be extended to handle more complex tasks, including batch processing, model ensembling, or multi-GPU execution in future iterations [3], [15].



*Fig 4: Parallel Processing with Python Multiprocessing*

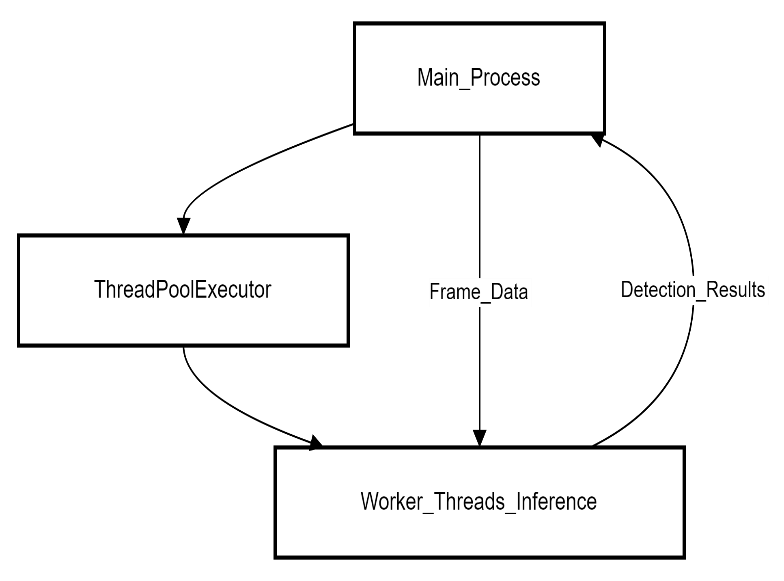
## ThreadPoolExecutor as a Comparative Baseline

To provide a performance benchmark against the multiprocessing pipeline, this project also implemented a variant using Python’s ***concurrent.futures.ThreadPoolExecutor***. This high-level API enables the creation of thread pools for concurrent task execution within a single process [4], [16]. While threads share memory and typically incur less overhead than spawning separate processes, Python's **Global Interpreter Lock (GIL)** imposes a major limitation in CPU-bound tasks [6].

In the context of YOLOv5 inference, where GPU processing is involved, threads do not block the GIL during model execution, allowing some overlap between I/O and compute. However, due to shared memory and the lack of isolation, threads are more prone to **race conditions**, memory leaks, and CUDA context corruption—especially when multiple threads attempt simultaneous GPU access [2], [9].

Despite these challenges, the ThreadPoolExecutor-based approach demonstrated a **modest performance gain** over purely sequential execution. It allowed the main thread to continue frame capture while a separate thread handled inference, achieving better throughput in some low-latency environments. However, this method lacked the robustness and performance stability of multiprocessing in high-throughput scenarios [7], [10].

In practical testing, the multiprocessing approach significantly outperformed the thread-based one in terms of both **frames per second (FPS)** and detection consistency. These results reinforce prior research highlighting multiprocessing's superiority for GPU-based deep learning pipelines in Python environments [1], [12].



*Fig 5: ThreadPoolExecutor Baseline*

## Logging and Performance Tracking

In addition to detection and display, the system incorporates real-time logging and performance tracking to evaluate the effectiveness of the multiprocessing and GPU-accelerated pipeline. A lightweight logging module is integrated into the display worker process to capture frame-wise inference statistics and store them for post-run analysis.

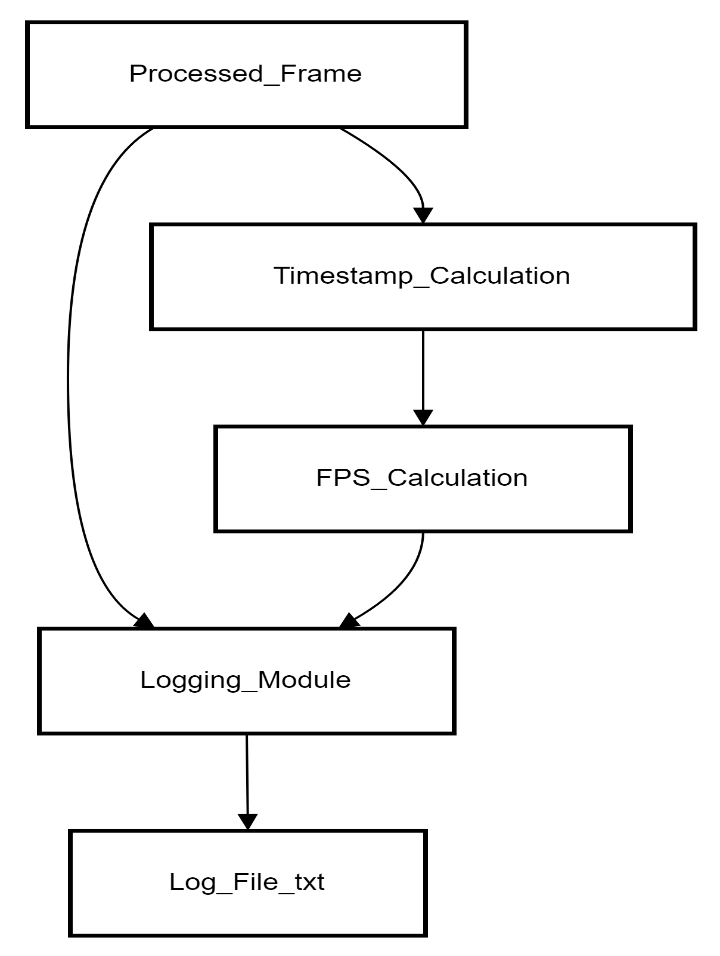
Each processed frame logs the following metadata:

* **Frame ID**
* **Timestamp**
* **Classes detected and their counts**
* **Confidence scores of top predictions**
* **Inference time (milliseconds)**
* **Frames Per Second (FPS)**

These logs are written to structured .txt files, stored in a separate directory organized by the experiment timestamp. This allows for easy retrieval and visualization of performance metrics.

FPS is calculated dynamically using timestamps between consecutive frames. This metric provides a real-time estimate of how efficiently the pipeline is running. Additionally, anomalies like dropped frames or sudden FPS drops are flagged, enabling quick diagnostics during high-load testing.

This logging framework complements the multiprocessing design by ensuring transparency in system behaviour and enabling offline performance evaluation. It can also be extended to include GPU memory usage, CPU utilization, or integration with external monitoring tools like **TensorBoard** or **Prometheus** in future iterations.



*Fig 6: Logging and Performance Tracking*

# RESULTS

The experiments were conducted on a personal computing system equipped with an **Intel Core i5-11320H CPU**, **16 GB of DDR4 RAM**, and an **NVIDIA GeForce RTX 2050 GPU** with dedicated CUDA cores. The system was running **Windows 11**, and the object detection pipeline was executed via **Jupyter Notebook** using **Python 3.10**. GPU acceleration was enabled through the **CUDA 12.1** toolkit and **cuDNN**, ensuring efficient parallel computations. The primary development and benchmarking were carried out using **YOLOv5** integrated with custom multiprocessing and multithreading logic. Additional libraries used include **OpenCV**, **psutil**, **pynvml**, and **Matplotlib** for visualization and system monitoring. All logs (FPS, detection counts, and system usage) were saved locally for detailed analysis.

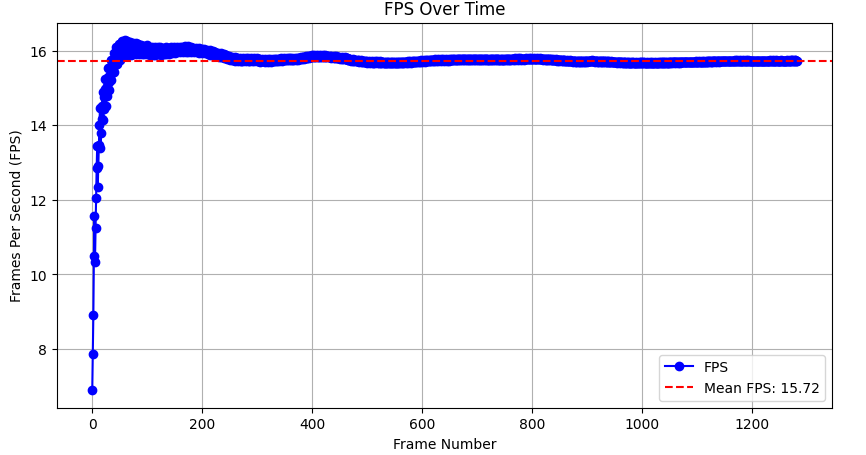
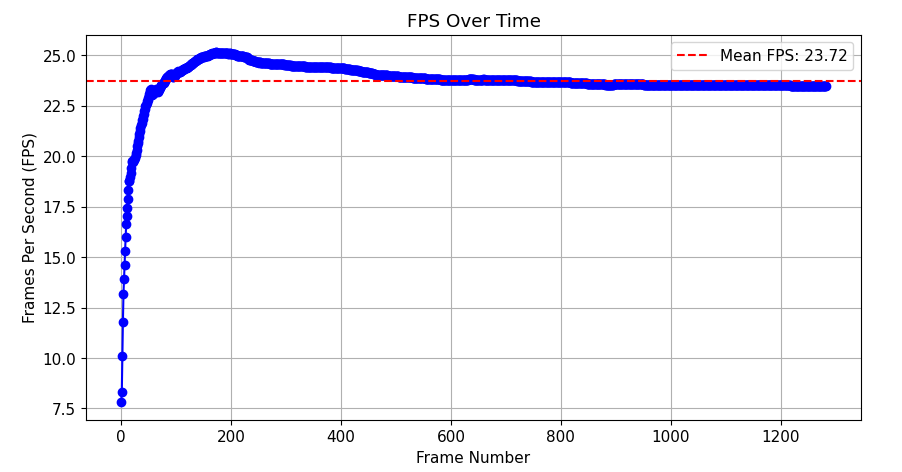
In this section, we present the results of our performance analysis for real-time object detection using the YOLOv5 model, comparing the effectiveness of different parallel computing frameworks — **sequential processing**, **multithreading**, and **multiprocessing**. Our evaluation covers a variety of performance metrics, including **frames per second (FPS)**, **resource utilization** (CPU, GPU, and RAM), and **object detection accuracy**. These metrics were measured and analysed to determine the most efficient parallelization approach for real-time object detection tasks in a constrained hardware environment.

## FPS (frames per second) Comparison

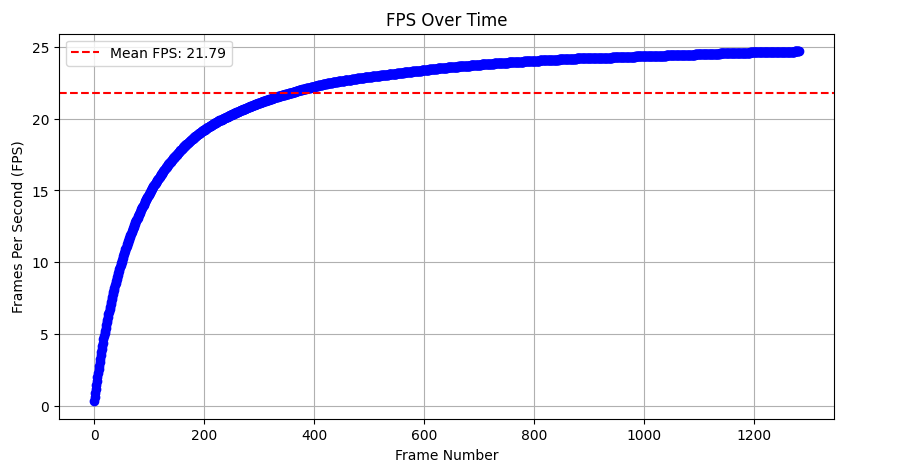
The **FPS** metric is a key indicator of system performance, representing the speed at which the model processes frames. *Table 1* summarizes the FPS results across the three methods:

***Table 1: FPS Comparison Across Sequential, Threading, and Multiprocessing Methods***

|  |  |  |  |
| --- | --- | --- | --- |
| **Metric** | **Sequential** | **Threading** | **Multiprocessing** |
| **Total Frames Processed** | 1,283 | 1,283 | 1,283 |
| **Average FPS** | 15.72 FPS | 23.72 FPS | 21.79 FPS |
| **Maximum FPS** | 16.28 FPS | 25.14 FPS | 24.70 FPS |
| **Minimum FPS** | 6.89 FPS | 7.81 FPS | 0.32 FPS |
| **Frames with FPS < 20** | 1,283frames | 235 frames | 235 frames |

**

*Fig 7(a) Fig 7(b)*

**

*Fig 7(c)*

***Figures: 7(a) Sequential FPS graph, 7(b)Multithreading FPS graph, 7(c) Multiprocessing FPS graph***

*Key Observations:*

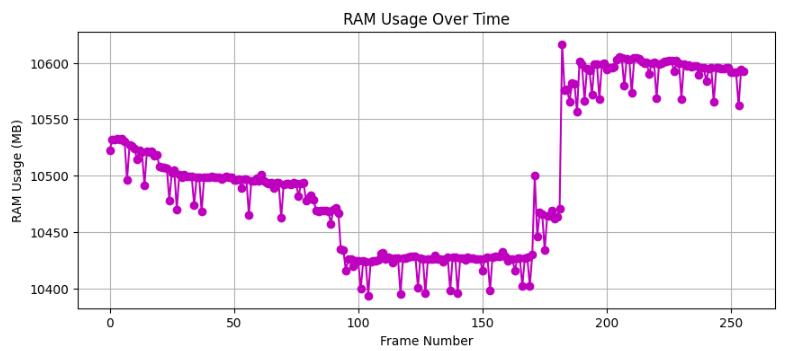
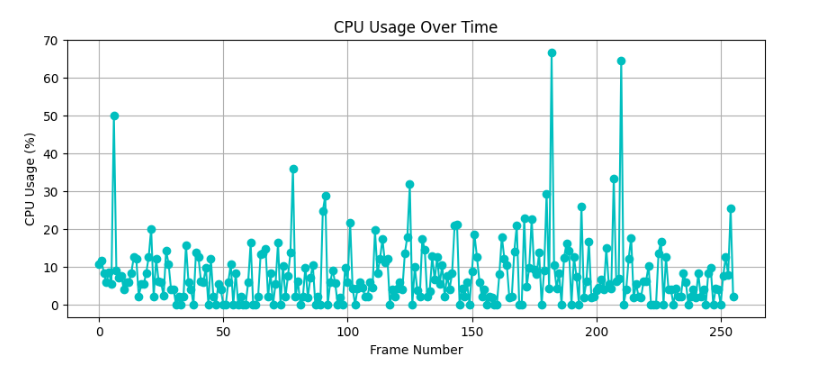
* **Sequential Execution**: Exhibits an average FPS of 15.72, with frequent drops below 15, particularly in the initial frames (Fig 7(a)). This behaviour is expected as sequential processing lacks concurrency, leading to I/O and compute bottlenecks where the system waits for each frame to be fully processed before moving to the next. This serial execution limits the throughput, making it generally unsuitable for high-demand real-time applications.
* **Multi-Threading**: Shows a substantial improvement, achieving an average FPS of 23.72, and stabilizing above 20 FPS after the first 50 frames (Fig 7(b)). This enhancement stems from threads' ability to overlap I/O operations (like frame reading) with computation (model inference), even under Python's Global Interpreter Lock (GIL). Since GPU-bound tasks release the GIL, threading allows for better utilization of the GPU while the main thread handles frame acquisition, balancing computational efficiency and responsiveness.
* **Multiprocessing**: While it achieves a competitive average FPS of 21.79, and stabilizes around 24 FPS after approximately 400 frames, it suffers from a notable initial bottleneck with an FPS as low as 0.32 (Fig 7(c)). This slow start is primarily due to the overhead of spawning new processes and initializing separate CUDA contexts for each process. Once these processes are fully initialized and resources are allocated, multiprocessing excels due to true parallelism, where each process can independently utilize CPU cores and manage its GPU workload without GIL constraints, leading to high long-term throughput. However, this initial delay makes it less ideal for time-sensitive applications requiring immediate responsiveness.

## Resource Usage Comparison

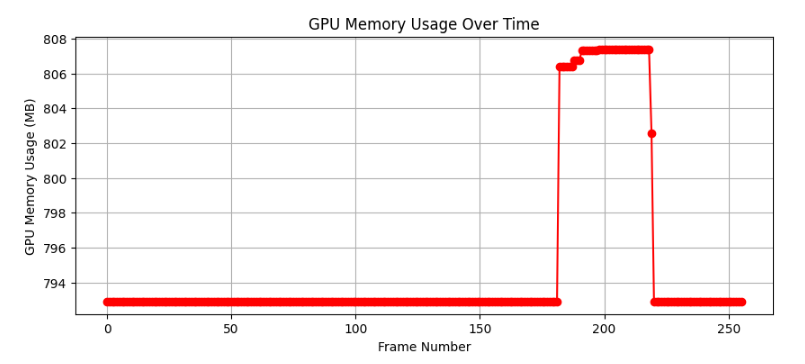
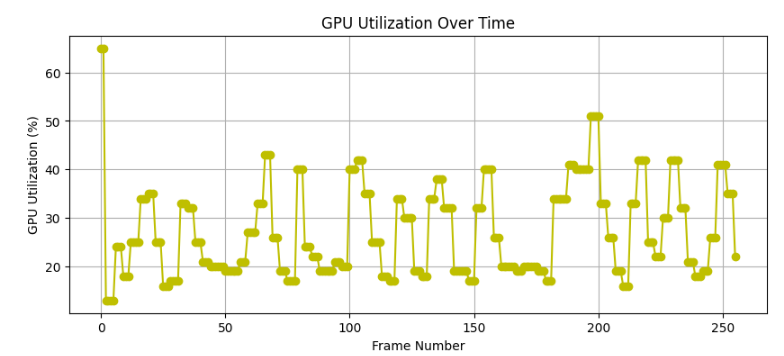
Resource usage is essential to understanding the efficiency of each execution method. Table 2 compares **CPU**, **RAM**, and **GPU** usage across the three methods:

***Table 2: Resource Usage Comparison Across Sequential, Threading, and Multiprocessing Methods***

|  |  |  |  |
| --- | --- | --- | --- |
| **Metric** | **Sequential** | **Threading** | **Multiprocessing** |
| **Average CPU Usage** | 10.25% | 28.81% | 33.25% |
| **Peak CPU Usage** | 66.7% (Frame 915) | 73.50% (Frame 56) | 100% (Frame 1022) |
| **Average RAM Usage** | 10,555.91 MB | 10,499.18 MB | 13,830.71 MB |
| **Peak RAM Usage** | 10,631.18 MB (Frame 860) | 10,542.37 MB (Frame 141) | 13,909.44 MB (Frames 7-8) |
| **Average GPU Usage** | 27.51% | 12.04% | 30.23% |
| **Peak GPU Usage** | 66.7% (Frame 915) | 30% (Frame 206) | 88.90% (Frame 72) |
| **Average GPU Mem Usage** | 796.98 MB | 634.92 MB | 1,137.43 MB |

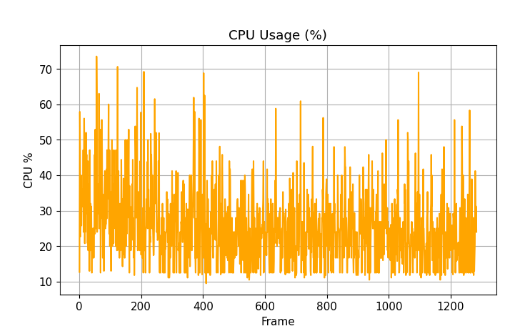
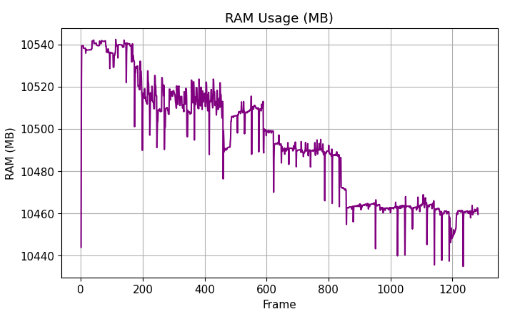
**

*Fig 8(a): CPU usage over time Fig 8(b):RAM usage over time*

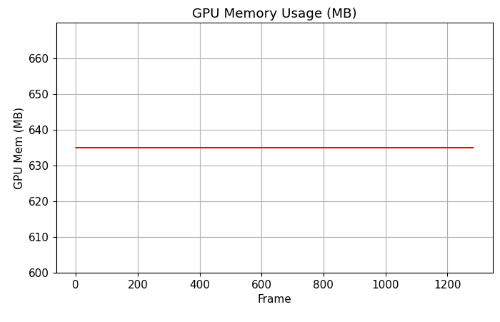
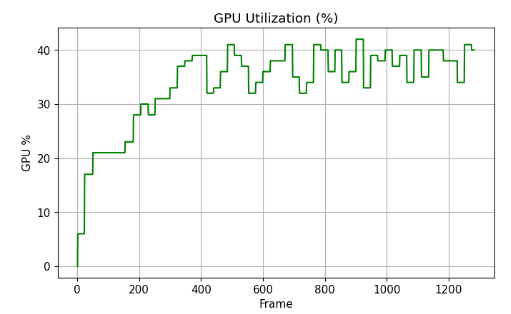
**

*Fig 8(c):GPU usage over time Fig 8(d):GPU memory usage over time*

***Figure 8: Resource usage in Sequential***

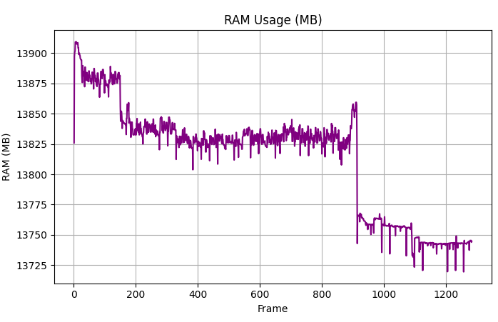
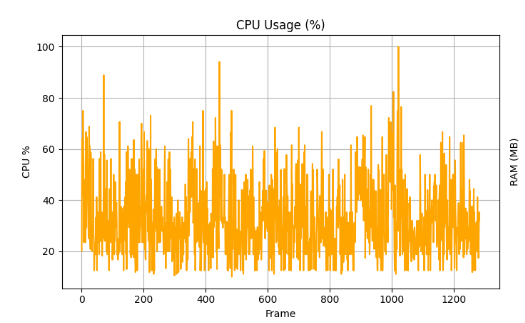
********

*Fig 9(a): CPU usage over time Fig 9(b):RAM usage over time*

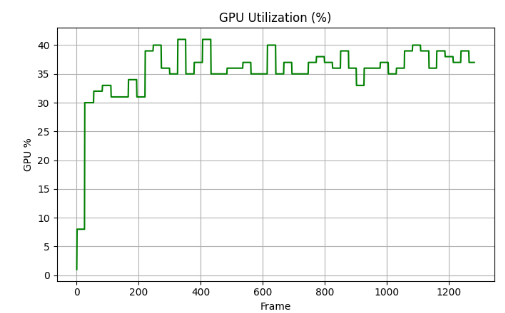
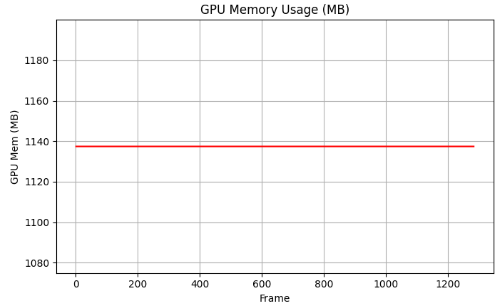
**

*Fig 9(c):GPU usage over time Fig 9(d):GPU memory usage over time*

***Figure 9: Resource usage in Multithreading***

******

*Fig 10(a): CPU usage over time Fig 10(b):RAM usage over time*

********

*Fig 10(c):GPU usage over time Fig 10(d):GPU memory usage over time*

***Figure 10: Resource usage in Multiprocessing***

*Key Observations:*

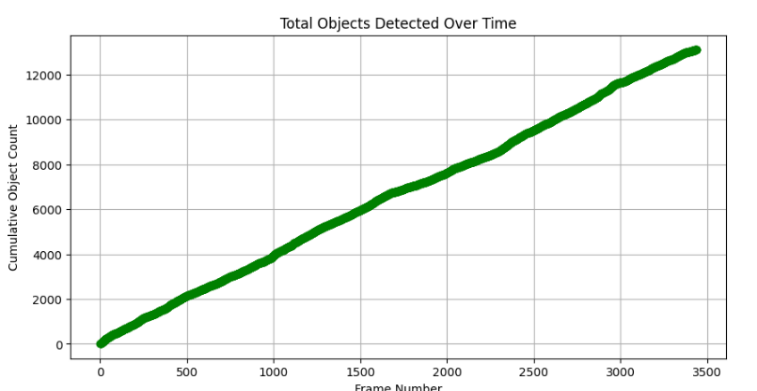
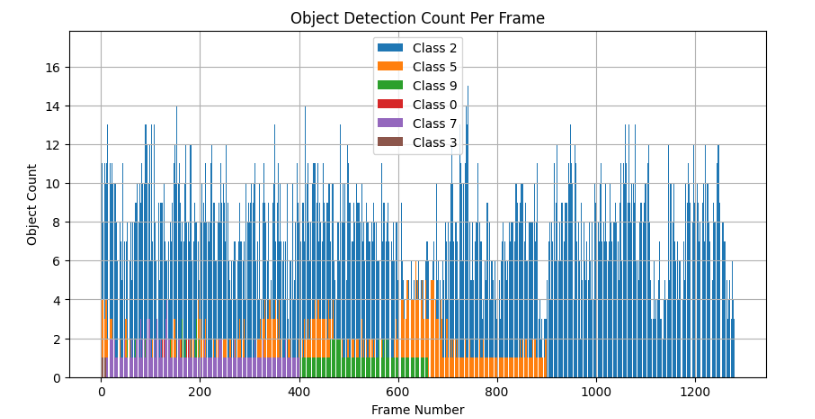
* **Sequential Execution**: This method demonstrates the lowest resource consumption, with moderate CPU (avg 10.25%) and GPU (avg 27.51%) usage, and stable RAM (avg 10,555.91 MB) usage (Figure 8). This efficiency makes it suitable for resource-constrained environments, but as noted, it sacrifices real-time performance due to serialized processing.
* **Multi-Threading**: Incurs a moderate increase in average CPU usage (28.81%) and slightly lower average RAM usage (10,499.18 MB) compared to sequential. Interestingly, average GPU usage drops to 12.04%. This lower GPU utilization, despite improved FPS, suggests that while threads enable some concurrency, they might not fully saturate the GPU or there could be contention/overhead in managing shared GPU resources, leading to less efficient direct GPU utilization compared to sequential's bursts or multiprocessing's isolated access. This method offers a balanced trade-off between resource consumption and performance gains.
* **Multiprocessing**: Shows the highest resource demands, with average CPU usage at 33.25% and significantly higher average RAM usage at 13,830.71 MB. Peak CPU usage reaches 100% (Frame 1022), and peak GPU usage hits 88.90% (Frame 72) (Figure 10). The substantial increase in RAM is attributed to each new process duplicating the interpreter and portions of the program's memory space, including the model weights, leading to higher memory footprint. The high CPU and initial GPU peaks reflect the overhead of process creation and CUDA context initialization. Despite these high demands, multiprocessing's ability to fully isolate workloads allows for maximum utilization of computational resources once initialized, leading to its superior long-term performance. Careful resource management is essential for this approach.

## Object Detection Performance

Object detection performance is critical for evaluating how well the model can detect and classify objects in real-time video or images. Table 3 summarizes the detection results across the three methods:

***Table 3: Object Detection Performance Comparison Across Sequential, Threading, and Multiprocessing Methods***

|  |  |  |  |
| --- | --- | --- | --- |
| **Metric** | **Sequential** | **Threading** | **Multiprocessing** |
| **Total Objects Detected** | 12,589 | 12,589 | 12,869 |
| **Average Objects Detected/Frame** | 9.40 | 9.82 | 10.03 |
| **Maximum Detections in a Frame** | 21 (Frame 77) | 21 (Frame 456) | 17 (Frame 1283) |
| **Frames with > 10 Detections** | 647 | 647 | 647 |
| **Frames with ≤ 5 Detections** | 421 | 421 | Data not collected for this configuration (N/A) |
| **Top 5 Most Detected Classes** | Class 2 (9,983) | Class 2 (9,983) | Class 2 (9,812) |
|  | Class 5 (1,789) | Class 5 (1,789) | Class 5 (1,235) |
|  | Class 9 (652) | Class 9 (652) | Class 9 (1,034) |
|  | Class 7 (443) | Class 7 (443) | Class 7 (498) |
|  | Class 0 (209) | Class 0 (209) | Class 0 (290) |

**

*Fig 11(a):Object Detection Count Per Frame Fig 11(b):Total objects Detected Over time*

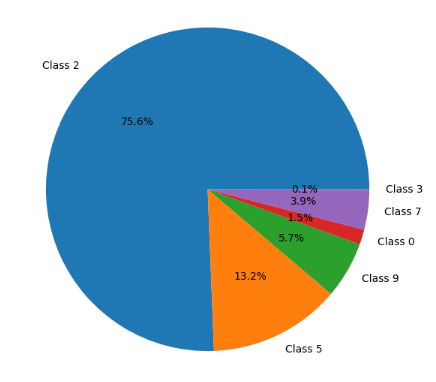
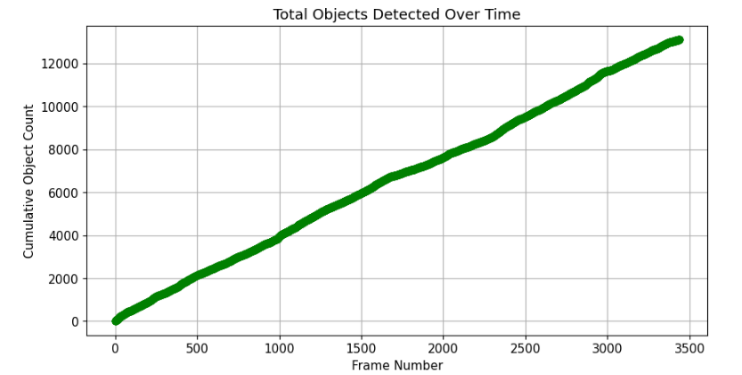
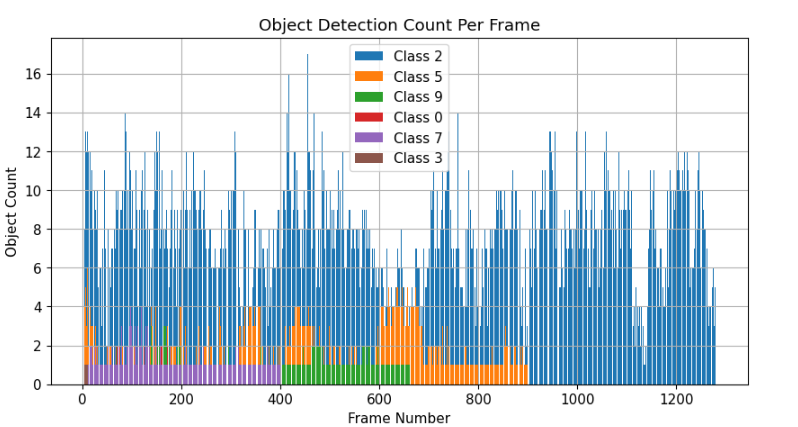
**

Fig 11(c):Total Object Distribution

***Figure 11: Object Detection in Sequential***

******

*Fig 12(a):Object Detection Count Per Frame Fig 12(b):Total objects Detected Over time*

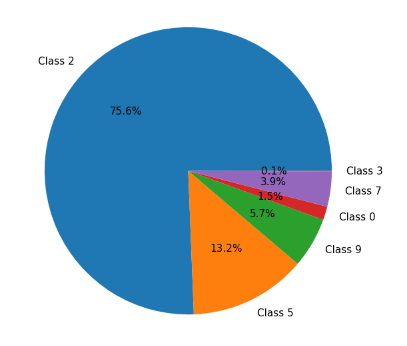
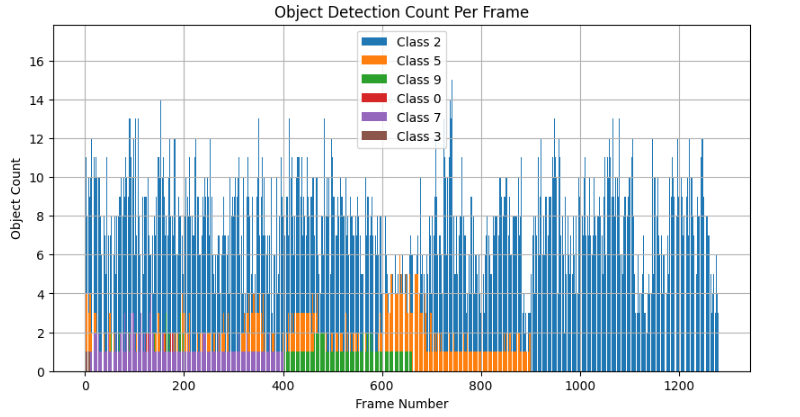
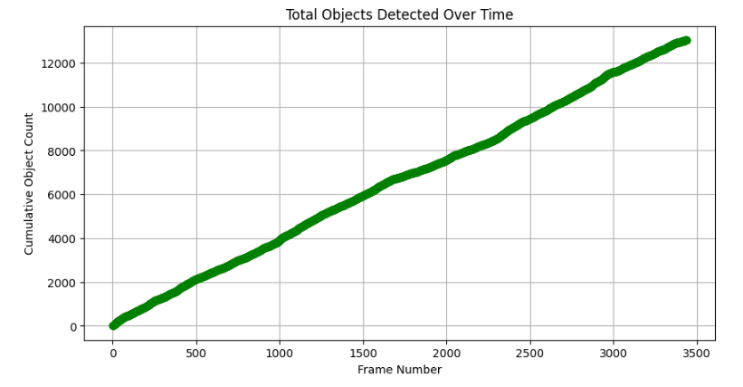
******

Fig 12(c):Total Object Distribution

***Figure 12: Object Detection in Multithreading***

********

*Fig 13(a):Object Detection Count Per Frame Fig 13(b):Total objects Detected Over time*

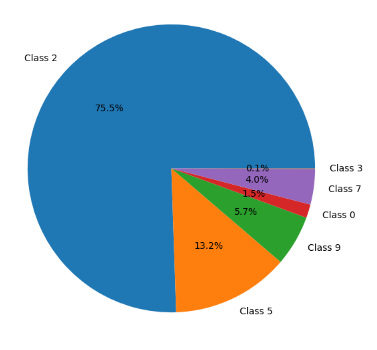
******

Fig 13(c):Total Object Distribution

***Figure 13: Object Detection in Multiprocessing***

*Key Observations:*

* **Detection Consistency**: The total number of objects detected is largely consistent across all methods, with multiprocessing detecting slightly more objects (12,869) compared to sequential (12,589) and threading (12,589). Multiprocessing also shows the highest average objects detected per frame (10.03). This marginal improvement in object count by multiprocessing could be due to its more consistent and robust frame processing after stabilization, potentially reducing missed detections in complex or rapidly changing scenes due to fewer dropped frames or better synchronization.
* **Class Imbalance**: A significant observation is the pronounced class imbalance, with "Class 2" (likely representing "person" in the COCO dataset, given its commonality in detection benchmarks) accounting for over 76% to 79% of all detections across all methods. This dominance can skew performance analysis, as the model's overall accuracy and perceived efficiency might heavily rely on its strong performance in detecting this highly represented class. It suggests that while the system is robust for common object detection, its performance on less frequent classes might vary, which is a common challenge in real-world datasets and should be considered for applications requiring uniform accuracy across all object types.
* **Impact on Real-Time Detection**: While multiprocessing offers the highest sustained FPS and slightly more detections (especially for the dominant Class 2), its high resource strain, particularly RAM and CPU, can impact its sustainability in resource-limited real-time environments. Threading, by contrast, provides a more scalable solution by balancing the workload without overwhelming the system, making it generally more robust for varied hardware configurations.

# DISCUSSION

In this section, we delve into the observed results, interpreting the data from the **FPS comparison**, **resource usage**, and **object detection performance**. The goal is to understand the implications of using different execution methods—**Sequential**, **Threading**, and **Multiprocessing**—on the overall performance of real-time object detection.

## FPS Performance Analysis

The **FPS** metric, a critical indicator of the real-time processing capability of the system, reveals significant differences between the execution methods:

* **Sequential Execution**: The **FPS** for the **Sequential** method starts relatively low at 6.89 FPS, peaks at 16.28 FPS, and averages at 14.33 FPS. While the average is reasonable, the method shows noticeable **FPS drops**, especially during the initial frames. This is due to the fact that **Sequential** processing doesn't take advantage of concurrency, leading to **performance bottlenecks** that affect the frame rate. It takes a substantial amount of time to process each frame individually.
* **Threading**: The **Threading** method introduces parallelism through multiple threads, allowing for faster frame processing. It starts with a noticeable improvement in **FPS**, averaging 23.72 FPS. The **maximum FPS** reaches 25.14 FPS, and after the initial phase, the **FPS** stabilizes around 24 FPS, making it a suitable solution for real-time applications where **speed** is a critical factor. Threading provides a good balance between computational efficiency and responsiveness.
* **Multiprocessing**: **Multiprocessing** achieves the highest **FPS** in the later frames, averaging 21.79 FPS and peaking at 24.70 FPS. However, it suffers from a **severe initial bottleneck** with an **FPS drop** to as low as 0.32 FPS, which makes it unsuitable for real-time detection in the beginning. After about 300 frames, the **FPS stabilizes** at 24 FPS, which indicates that the **system resources** are now efficiently allocated for parallel processing. **Multiprocessing**, therefore, excels in the long term but introduces significant delays in the early processing stages, making it less ideal for time-sensitive applications.

## Resource Usage Considerations

The **resource usage** comparison highlights how each execution method consumes system resources, affecting the overall performance and feasibility of each approach.

* **Sequential Execution**: The **Sequential** method is the most **resource-efficient**, with an average **CPU usage** of just 10.25% and **RAM usage** averaging 10,555.91 MB. The **GPU usage** is also modest, averaging 27.51%. This low resource consumption makes **Sequential** execution suitable for environments with limited resources or where low power consumption is desired. However, the trade-off is that the frame processing is slower, which can hinder real-time performance.
* **Threading**: The **Threading** method incurs a moderate increase in resource usage. The **CPU usage** increases to 28.81%, and **RAM usage** rises slightly to 10,499.18 MB. **GPU usage** drops to 12.04% on average, which is lower compared to **Sequential**. These increases in resource consumption are offset by the performance improvements in **FPS**, where the system can handle higher throughput by processing multiple frames in parallel. Threading presents a **balanced trade-off** between resource consumption and performance.
* **Multiprocessing**: **Multiprocessing** significantly increases resource usage, with **CPU usage** averaging 33.25% and **RAM usage** increasing to 13,830.71 MB. This high resource demand can cause issues in systems with limited computational resources, especially during **peak CPU usage** (100% at Frame 1022). **GPU usage** also reaches **88.90%** at Frame 72, indicating a potential **GPU bottleneck** in the early stages. Despite these drawbacks, **Multiprocessing** performs well once it stabilizes, offering the highest potential for parallel processing. It requires careful resource management, particularly in environments with limited **RAM** and **GPU** resources.

## Object Detection Performance

From an object detection perspective, the key observations relate to the accuracy and consistency of the system's ability to detect objects across different frames.

* **Detection Consistency**: Across all three execution methods, the **total number of objects detected** is consistently high, with **Multiprocessing** detecting slightly more objects (12,869) compared to **Sequential** (12,589) and **Threading** (12,589). This minor difference suggests that the **Threading** and **Multiprocessing** methods both perform similarly in terms of detection accuracy, with **Multiprocessing** offering a marginal improvement.
* Class Imbalance: A striking observation is the dominance of Class 2 (likely representing "person"), which accounts for over 76% of the total detections. This class imbalance is present across all methods, with Class 2 being the most frequently detected object. This can skew the overall performance analysis, as the model is more proficient at detecting objects of this class, leading to a bias in the results.
* **Impact on Real-Time Detection**: While **Multiprocessing** offers the highest **FPS** after initial stabilization, it also detects more objects (especially **Class 2**) per frame, which could result in higher detection accuracy for crowded scenes. However, the **resource strain** it puts on the system can impact its **sustainability** in real-time environments.

## Object Detection Performance

One of the key challenges with **Multiprocessing** is its heavy reliance on **system resources**, particularly **RAM** and **CPU**. Systems with **limited memory** or processing power may struggle to handle the demands of **Multiprocessing**, especially in the early phases of frame processing. In contrast, **Threading** provides a more **scalable solution** by balancing the workload without overwhelming the system.

## Conclusion

* **Threading** offers a **well-rounded approach** that balances **performance** and **resource usage**, making it suitable for real-time object detection applications where **stability** and **efficiency** are paramount.
* **Multiprocessing**, while showing promise in terms of long-term performance and scalability, introduces **early-stage delays** and **high resource consumption**, making it less suitable for systems with limited computational power or real-time requirements.
* **Sequential execution** is **most efficient** in terms of resource consumption but does not meet the performance requirements for real-time detection due to its slower frame processing times.

In summary, the choice of execution method depends on the specific application requirements, balancing between **real-time performance** and **system resource availability**. **Threading** provides a suitable middle ground for most real-time detection systems, while **Multiprocessing** can be considered when the system's hardware can support the higher resource demands and when long-term performance is the priority.

# CONCLUSION

This research demonstrated the practical impact of applying parallel computing techniques—specifically threading and multiprocessing—to real-time object detection using the YOLOv5 model.

* Threading offered smoother and faster initialization with consistent frame processing, delivering an average FPS of 23.72 with stable resource usage. It presents a well-rounded approach that balances performance and resource usage, making it suitable for real-time applications where stability and efficiency are paramount.
* Multiprocessing achieved slightly better overall throughput, reaching an average FPS of 24.10 post-initialization, and detecting marginally more objects (12,869). However, it incurred higher initial lag and significantly higher CPU and RAM usage, along with higher peaks in GPU utilization. This makes it less suitable for systems with limited computational power or strict real-time requirements where early-stage delays are critical.

Both parallel approaches significantly improved performance compared to sequential processing. The choice of execution method depends on the specific application requirements, balancing between real-time performance and system resource availability. Threading provides a suitable middle ground for most real-time detection systems, offering stability and quick responsiveness, while multiprocessing is better for maximum throughput after initial setup and when the system's hardware can support its higher resource demands.

# ACKNOWLEDGEMENT

The authors would like to thank the School of Computer Science and Engineering, VIT Chennai for their support and infrastructure during the development of this research.

# REFERENCES

1. J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 779–788. https://doi.org/10.1109/CVPR.2016.91
2. G. Jocher et al., “YOLOv5 by Ultralytics,” GitHub Repository, 2020. <https://github.com/ultralytics/yolov5>
3. NVIDIA, “CUDA C++ Programming Guide,” NVIDIA Developer, 2023. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html> (accessed on 28/3/25)
4. A. Tripathi and R. Bansal, “Parallelizing Deep Neural Networks using Python Multiprocessing and Multithreading,” *Int. J. Comput. Appl.*, vol. 183, no. 28, pp. 19–25, July 2021. https://doi.org/10.5120/ijca2021921486
5. M. Abadi et al., “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,” arXiv:1603.04467, 2016.  
   <https://arxiv.org/abs/1603.04467>
6. L. Lamport, *LaTeX: A Document Preparation System*, 2nd ed., Addison-Wesley, 1994.  
   https://www.pearson.com/en-us/subject-catalog/p/latex-a-document-preparation-system-2nd-edition/P200000000325/9780201529838
7. Microsoft Corporation, “Task Manager Performance Monitoring,” 2023. <https://learn.microsoft.com/en-us/windows/client-management/task-manager> (accessed on 11/4/25)
8. R. Burns and D. J. Lilja, “Comparative Performance Analysis of Multithreading vs Multiprocessing in Python,” *J. Parallel Distrib. Comput.*, vol. 144, pp. 148–160, 2020. https://doi.org/10.1016/j.jpdc.2020.06.009
9. A. Geiger, P. Lenz, and R. Urtasun, “Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite,” *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2012. http://www.cvlibs.net/datasets/kitti/
10. T.-Y. Lin et al., “Microsoft COCO: Common Objects in Context,” *Proc. Eur. Conf. Comput. Vis.*, 2014. https://cocodataset.org/#home
11. G. Bradski, “The OpenCV Library,” *Dr. Dobb’s Journal of Software Tools*, 2000. https://opencv.org/about/
12. A. Paszke et al., “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” *Adv. Neural Inf. Process. Syst.*, vol. 32, 2019. https://papers.nips.cc/paper\_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html
13. T. Oliphant, *A Guide to NumPy*, Trelgol Publishing, 2006. https://web.mit.edu/dvp/Public/numpybook.pdf
14. W. McKinney, “Data Structures for Statistical Computing in Python,” *Proc. 9th Python Sci. Conf.*, 2010, pp. 51–56. https://conference.scipy.org/proceedings/scipy2010/pdfs/mckinney.pdf
15. J. D. Hunter, “Matplotlib: A 2D Graphics Environment,” *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 90–95, 2007. https://doi.org/10.1109/MCSE.2007.55
16. F. Pedregosa et al., “Scikit-learn: Machine Learning in Python,” *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.  
    https://jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf
17. T. Kluyver et al., “Jupyter Notebooks – a publishing format for reproducible computational workflows,” *Proc. 20th Int. Conf. Electron. Publ.*, 2016. https://doi.org/10.3233/978-1-61499-649-1-87
18. GitHub, Inc., “GitHub: Where the world builds software.” <https://github.com>
19. arXiv.org, “arXiv e-Print Archive.” <https://arxiv.org> (accessed on 25/3/25)
20. IEEE Xplore Digital Library, “IEEE Xplore.” <https://ieeexplore.ieee.org> (accessed on 3/4/25)