Optimizing Real-time Video Analytics for Resource-Constrained Environments

Rodrique Kafando, Aminata Sabane, Tégawendé F. Bissyandé

Optimizing Real-time Video Analytics for Resource-Constrained Environments

Rodrique Kafando[1][0000−0002−7190−9225], Aminata Sabané[2], and Tégawendé F. Bissyandé[3]

[1] Centre d'Excellence CITADEL, Université Virtuelle du Burkina Faso
[2] Département d'Informatique, UFR/SEA, UJKZ, Burkina Faso
rodrique.kafando@citadel.bf
https://citadel.bf/

Abstract. Developing countries face a growing demand for video analytics, yet often lack sufficient computational resources. This paper addresses this challenge by proposing and evaluating optimization techniques for efficient video stream processing on resource-constrained devices, including edge systems. We introduce and evaluate several techniques, including image resizing, frame skipping, parallel processing, threading, queue management, memory optimization, and buffering. Experimental results demonstrate substantial improvements in frames per second (FPS) and memory usage, enabling real-time video analytics without compromising accuracy. By effectively balancing performance and resource consumption, our methods facilitate the deployment of advanced AI-driven video analysis in resource-limited environments, paving the way for practical real-time monitoring and alert systems.

Keywords: RSTP stream · Real-time Video Analytics · Computational Optimization · Post-Training Optimization

1 Introduction

Real-time object detection and tracking demand exceptionally low processing and transmission latencies to be effective. These systems are critical in various applications requiring immediate alerts, such as surveillance, access control, and anomaly detection. For instance, in sensitive areas, unauthorized intrusions must be detected and reported instantly. Retail environments rely on rapid theft detection to prevent losses, while traffic monitoring systems necessitate real-time accident alerts for efficient emergency response.

Developing regions often face significant limitations in computational resources, hindering the deployment of advanced video analytics systems. These systems typically demand substantial processing power, making them impractical in environments with constrained infrastructure and budgets. To bridge this gap, our research focuses on developing optimization strategies that enable efficient video analytics on resource-limited platforms.

We introduce in this study a suite of optimization techniques to accelerate the processing and delivery of RTSP video streams while maintaining low latency. By employing image resizing, frame skipping, MSE-based frame skipping, parallel processing, threading, queue management, memory optimization, and buffering, we aim to optimize real-time video analytics for resource-constrained environments. Our findings demonstrate substantial improvements in processing efficiency, enabling the reliable operation of real-time monitoring and alert systems in such settings. These results highlight the feasibility of deploying advanced video analytics solutions that can function reliably in developing regions.

2 Related Work

Object detection techniques are a key area of artificial intelligence, with popular algorithms such as You Only Look Once (YOLO) and its improved versions (YOLOv2 to YOLOv10). These algorithms are noted for their speed and accuracy, processing images in a single pass to provide the positions and categories of detected objects [6]. Successive versions of YOLO have brought improvements in batch normalization, the use of high-resolution images, and multi-scale training [2]. Other detection methods include Convolutional Neural Networks (CNNs) used in models like SSD (Single Shot MultiBox Detector) and Faster R-CNN, which balance processing speed and accuracy. For instance, SSD achieves an average precision of 74.3% mAP on the VOC2007 test at a speed of 59 FPS, surpassing Faster R-CNN and YOLO in terms of both speed and accuracy [5].

Recent research has explored distributed real-time video analytics using optimized YOLOv8 models on CPU nodes. This approach utilizes cloud platforms like Microsoft Azure and data streaming tools such as Apache Kafka to efficiently manage and process video streams [3]. OpenVINO [4] is used for both pre-training and post-training optimization, enhancing model efficiency by converting the YOLOv8 model into an IR format with FP32 precision, followed by post-training quantization to create an INT8 version of the model. This technique enhances processing speed while maintaining accuracy.

In [7] authors show that the magnitude pruning reduces computational complexity by removing low-magnitude weights from the network, followed by fine-tuning to recover performance. Results indicate that the optimal pruning rate for balancing performance and accuracy is around 0.5. This approach also allows a reduction in parameters and computations while maintaining good detection accuracy.

Optimized models allow for more efficient use of memory and processing resources, which is essential for resource-constrained devices like CPU edge nodes. Faster and more efficient models enable real-time detection and instant alerting, which is critical for surveillance and security applications [1]. The increasing demand for video storage and the complexities associated with processing large volumes of video data require robust and scalable solutions. Tools like Apache Kafka and Microsoft Azure Cloud play a crucial role in managing and processing real-time video in a distributed manner [4].

In addition to algorithm-specific detection techniques, it is essential to consider post-processing strategies that can be applied independently of the specific detection algorithm used. These strategies include image resizing, frame skipping, and the use of parallel processing and threading to manage real-time video streams. Efficient queue management for processed frames and memory optimization with tools like tracemalloc are also crucial for maintaining high performance and low latency. This article proposes and evaluates these post-processing strategies, demonstrating their effectiveness in optimizing object detection systems, regardless of the specific detection algorithm [8].

3 Proposed approaches

3.1 Image Resizing

Image resizing is a fundamental optimization technique used to enhance the performance of real-time video analytics systems, especially in resource-constrained environments. By reducing the resolution of video frames, the computational load on the processing system is significantly decreased, enabling faster processing speeds and lower memory usage. This section details the image resizing technique, its implementation, and its impact on the computational efficiency of object detection and tracking systems.

Interpolation methods reduce the number of pixels while preserving as much visual information as possible. The common interpolation methods include:

- Nearest Neighbor Interpolation: Selects the nearest pixel value to the target pixel. It is the simplest and fastest method but may result in blocky images.
- Bilinear Interpolation: This method takes the average of the four nearest pixel values to compute the target pixel value. It produces smoother images than nearest neighbor interpolation.
- Bicubic Interpolation: This method considers the nearest sixteen pixels and produces even smoother images. It is more computationally intensive than the previous methods but provides better visual quality.

3.2 Frame Skipping

Frame skipping is an optimization technique designed to reduce the computational load of real-time video analytics systems by processing only a subset of the frames in a video stream. This method can significantly enhance processing speed and reduce memory usage, making it an effective strategy for resource-constrained environments.

Frame skipping selectively processes frames at regular intervals, skipping a predefined number of frames between each processed frame. The interval at which frames are processed is referred to as the skip rate (sk). For example, a skip rate of 2 means that every second frame is processed, while a skip rate of 5 means that every fifth frame is processed. The primary parameters involved in frame skipping are:

- Skip Rate: The number of frames to skip between each processed frame. Higher skip rates result in fewer processed frames, leading to faster processing times but potentially missing some critical information.
- Resolution: The resolution of the frames being processed, which can further influence the computational load.

3.3 MSE-based Frame Skipping

MSE-based frame skipping calculates the Mean Squared Error between consecutive frames. If the MSE value exceeds a predefined threshold, the frame is processed; otherwise, it is skipped. This method analyzes only frames with substantial differences, optimizing computational resource usage.

The primary parameters involved in MSE-based frame skipping are:

- Skip Rate (sk): The interval at which frames are checked for processing based on the MSE value. MSE Threshold (th): The threshold value above which frames are processed. Lower thresholds result in more frames being processed, while higher thresholds lead to more frames being skipped.

Other techniques besides MSE can be used for frame skipping, each with its advantages and limitations. Some notable methods include:

- Structural Similarity Index (SSIM): This technique compares the structural similarity between frames. SSIM is more sensitive to changes in structure, luminance, and contrast, making it suitable for applications where visual quality is paramount.
- Histogram Difference: This method calculates the difference in color histograms between frames. It is computationally less intensive than MSE and can quickly detect changes in the overall color distribution of frames.
- Optical Flow: This technique estimates the motion between frames by calculating the flow of pixels. Optical flow is effective for detecting motion and can be used to skip frames with minimal motion, thereby focusing processing on dynamic scenes.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(I_1(i) - I_2(i))^2 \tag{1}$$

Where I_1 and I_2 are two consecutive frames, and n is the number of pixels.

3.4 Parallel Processing with ThreadPoolExecutor

Parallel processing with ThreadPoolExecutor creates a pool of threads to execute tasks concurrently. Each frame of the video stream is treated as a separate task, which can be processed in parallel by the threads in the pool. By distributing the computational load across multiple threads, parallel processing reduces processing time and increases efficiency. For instance, the ThreadPoolExecutor from

the concurrent.futures library in Python provides a flexible and efficient way to implement parallel processing. The following steps outline the implementation strategy:

- Initialize the ThreadPoolExecutor: Create a ThreadPoolExecutor with a specified number of worker threads.
- Submit Tasks to the Executor: Submit the frame processing tasks to the executor.
- Process Frames Concurrently: Each worker thread processes a frame concurrently.
- Retrieve and Display Results: Collect the processed frames and display them.

$$T_p = \frac{T_s}{N} \tag{2}$$

Where T_p is the parallel processing time, Ts is the sequential processing time, and N is the number of threads.

3.5 Threading for Concurrent Video Stream Processing

Threading involves creating multiple threads within a single process, allowing different tasks to be executed concurrently. In the context of real-time video stream processing, threading can be used to handle multiple video streams simultaneously or to parallelize the processing of frames within a single stream. The primary techniques used in threading include:

- Thread Creation: Initializing and managing threads for concurrent execution.
- Synchronization: Coordinating the execution of threads to prevent race conditions and ensure data integrity.
- Queue Management: Using queues to buffer frames and manage the flow of data between threads.
- Thread Pooling: Utilizing a pool of reusable threads to handle tasks efficiently.

$$T_t = \frac{T_s}{N} + \text{Overhead} \tag{3}$$

Where T_t is the threading processing time, T_s is the sequential processing time, N is the number of threads, and $Overhead$ is the additional time due to context switching and synchronization.

3.6 Queue Management for Processed Frames

Queue management involves using data structures called queues to temporarily store frames as they move through the different stages of processing. Queues help to manage the flow of frames between the frame capture stage, the processing stage, and the display stage. The key components of the queue management approach include:

- Frame Queue: Buffers frames captured from the video stream before they are processed.
- Processed Queue: Buffers frames after they have been processed and before they are displayed.
- Max Queue Size: Defines the maximum number of frames that can be stored in the queue at any given time, preventing memory overflow and ensuring smooth processing.

By using queues to buffer frames and manage the flow of data between various stages of processing, queue management ensures efficient handling of video streams and reduces latency.

$$\text{Queue Size} = \min(\text{Max Size}, \text{Input Rate} \times \text{Processing Time}) \tag{4}$$

Where $InputRate$ is the rate at which frames are added to the queue and $ProcessingTime$ is the time taken to process each frame.

3.7 Memory Management with tracemalloc

Memory management is the process of efficiently allocating, utilizing, and freeing memory resources within a computer system. It ensures that applications have sufficient memory to execute while optimizing the overall performance and stability of the system. Effective memory management involves tracking memory usage, preventing memory leaks, and ensuring that memory is allocated and deallocated in a way that maximizes system efficiency and prevents resource exhaustion. This is critical in environments with limited computational resources, as it helps maintain optimal performance and avoid system crashes or slowdowns. The Python library *tracemalloc* designed to trace memory allocations. It provides detailed information about memory usage, including the size and location of allocated memory blocks. An effective memory management ensures that the system can handle large volumes of video data without running out of memory or experiencing significant performance degradation. The implementation of memory management with tracemalloc involves the following steps:

- Initialize tracemalloc: Start tracing memory allocations at the beginning of the program.
- Track Memory Usage: Monitor memory usage during the execution of the program.
- Analyze Memory Usage: Retrieve and analyze memory statistics to identify memory usage patterns and potential issues.
- Stop tracemalloc: Stop tracing memory allocations and clean up resources.

$$\text{Memory Usage} = \sum_{i=1}^{n} \text{Memory Block Size}_i \tag{5}$$

Where n is the number of memory blocks allocated.

3.8 Optimization Through Buffering

Buffering is the process of temporarily storing data in a memory buffer while it is being transferred between two locations, typically to accommodate differences in data processing rates between the source and destination. In the context of real-time video analytics, buffering helps to manage the flow of video frames, ensuring smooth and efficient processing by mitigating the impact of fluctuations in data processing speeds. Buffering plays a crucial role in maintaining a consistent data stream, reducing latency, and preventing bottlenecks in the processing pipeline.

The implementation of buffering in real-time video analytics involves creating queues that act as buffers for incoming and outgoing frames. The queue.Queue class in Python provides a thread-safe FIFO (First In, First Out) queue, which is ideal for this purpose. The key techniques in buffering include:

- Input Buffering: Temporarily storing incoming video frames before they are processed. This helps to accommodate variations in frame capture rates and ensures a steady supply of frames to the processing pipeline.
- Output Buffering: Storing processed frames before they are displayed or transmitted. This helps to handle variations in processing time and ensures smooth playback or transmission.
- Buffer Size Management: Adjusting the size of the buffers to balance memory usage and processing efficiency. Larger buffers can handle more variation in processing times but require more memory.

$$\text{Buffer Size} = \text{Input Rate} \times \text{Buffer Time} \tag{6}$$

4 Experimental setup and Discussion

In this section, we detail the experimental setup used to evaluate the performance of various optimization strategies for real-time video analytics systems. The primary goal is to assess the impact of each strategy on processing speed (FPS), memory usage, and overall system efficiency. The strategies evaluated include image resizing, frame skipping, MSE-based frame skipping, parallel processing, threading, queue management, memory management with tracemalloc, and buffering. We provide the source code for replication, accessible from our GitHub repository.

4.1 Hardware and system configuration

The experiments were conducted using a Dahua IR DOME NETWORK CAMERA with a wired connection (RJ-45) for video input. To facilitate easy replication, the reported results were obtained using a free online video from Pexels[3].

[3] https://www.pexels.com/video/dancers-performing-a-choreography-on-a-promenade-14691550/

Additionally, the yolov8n.pt[4] model from Ultralytics was employed for object detection, using the default classes and parameters (such as confidence, etc.) provided by the model.

The PC used for the experiment has the following system configuration is as follows:

- Operating System: Linux #35 22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Tue May 7 09:00:52 UTC 2 (Release: 6.5.0-35-generic)
- Processor: x86_64
- Number of Cores: 8
- Max CPU Frequency: 4700.0 MHz
- Total Memory: 15.34 GB
- GPU Info: GPUtil not installed

4.2 Results and Discussion

Table 1 below summarizes the performance metrics for each optimization strategy applied in our real-time video analytics system. The strategies include image resizing, frame skipping, MSE-based frame skipping, parallel processing, threading, queue management, memory management with tracemalloc, and buffering. For each strategy, we present the skip rate, resolution, maximum queue size, number of frames processed, total processing time, frames per second (FPS), current memory usage, and peak memory usage. These results highlight the effectiveness of each optimization technique in enhancing processing speed and efficiency, particularly in a resource-constrained environment. Figure 1 and Figure 2 respectively represent the evolution of FPS and the memory usage across the different strategies.

Table 1. Performance metrics accross optimization strategies

Strategy	**SkRate**	**Resol**	**MxQSize**	**NbFrames**	**Time(s)**	**FPS**	**C.Mem(MB)**	**PeakMem(MB)**
Default	-	1920x1080	-	705	163.27	4.32	201.67	220.08
IR	-	640x480	-	705	102.11	6.90	175.22	187.41
IR	-	320x240	-	705	75.25	9.37	66.65	79.01
FSkip	2	640x480	-	353	64.96	5.43	88.46	100.90
FSkip	5	640x480	-	141	13.76	10.24	36.84	49.28
MSESkip	5	640x480	-	141	14.54	9.70	37.46	53.12
ParlProcess	2	640x480	-	352	33.74	10.43	89.44	104.86
ParlProcess	5	640x480	-	141	14.85	9.49	47.39	62.81
Threading	2	640x480	-	352	46.72	7.53	89.05	104.72
Q.M.	5	640x480	10	141	14.23	9.91	48.07	62.81
Q.M.	5	640x480	20	141	13.86	10.17	48.07	64.07
M.M	5	640x480	10	141	16.18	8.71	91.80	108.64
M.M	5	640x480	20	141	14.95	9.43	81.45	108.33
Buff.	5	640x480	10	141	15.54	9.07	81.74	108.32
Buff	5	640x480	20	141	15.08	9.35	81.45	108.31

[4] https://github.com/ultralytics/assets/releases/download/v8.2.0/yolov8n.pt

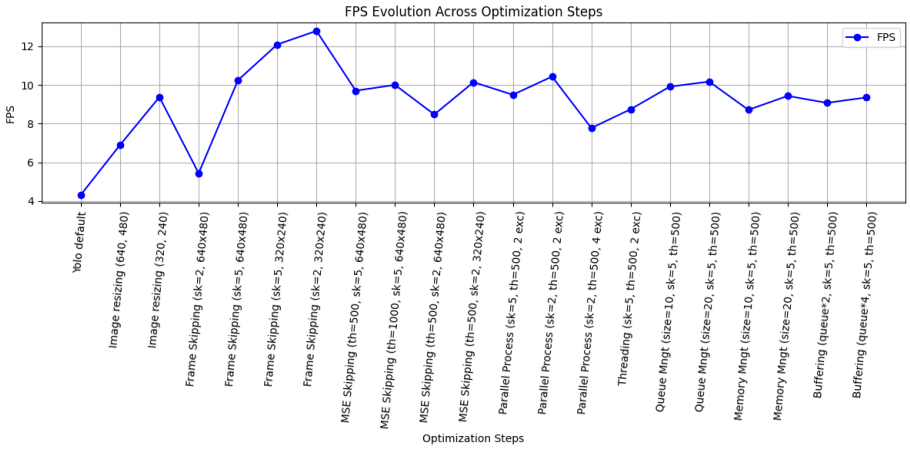

Fig. 1. FPS evolution across optimization steps.

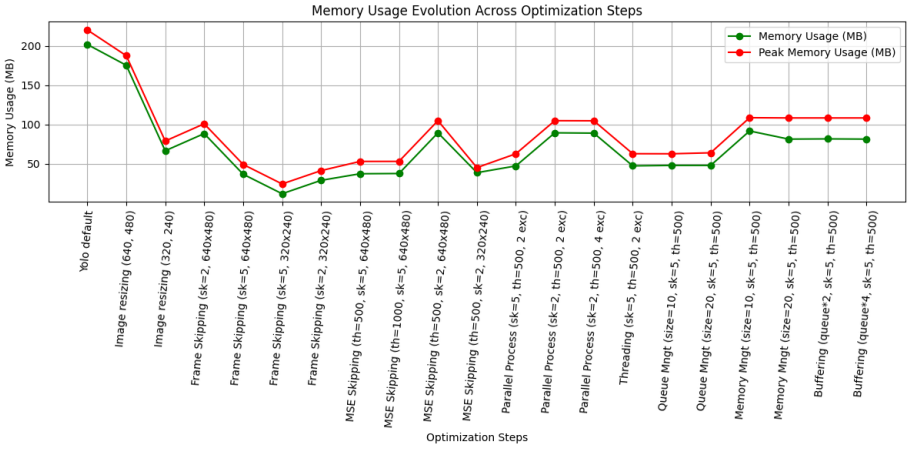

Fig. 2. Memory usage evolution across optimization steps.

Image resizing significantly increased FPS and reduced memory usage. Reducing the resolution from 1920x1080 to 640x480 improved FPS from 4.32 to 6.90, and further reducing to 320x240 increased FPS to 9.37, proving highly effective for improving processing speed in resource-constrained environments. In practice, the optimal resolution may vary depending on the specific application and the complexity of the scenes being analyzed.

Frame skipping showed notable improvements in FPS and memory usage. With a skip rate of 5, the FPS increased to 10.24 for 640x480 resolution. MSE-based (same for other technics) frame skipping provided a balance between processing efficiency and accuracy of object detection and tracking. Using a threshold of 500 with a skip rate of 5 yielded an FPS of 9.70, while higher thresholds resulted in slightly better FPS. This method is especially useful for detecting significant changes in video frames, maintaining a balance between performance and accuracy where not every frame needs to be processed.

Parallel processing improved FPS by distributing the computational load across multiple threads. With 2 executors, the FPS increased to 10.43 for 640x480 resolution. However, increasing the number of executors to 4 resulted in a slight decrease in FPS due to the overhead of managing more threads, indicating a balance must be struck in thread management.

Threading allowed for concurrent processing of multiple video streams, enhancing FPS. With 2 threads, the FPS was 7.53 for 640x480 resolution. This strategy is effective for systems that need to handle multiple video streams simultaneously, ensuring improved throughput and responsiveness. Balancing the number of threads is important, to avoid overhead associated with managing too many threads.

Queue management using buffers helped smooth out fluctuations in processing time, leading to more consistent performance. Increasing the max queue size to 20 improved FPS to 10.17 for 640x480 resolution. This approach is beneficial for managing data flow and reducing latency, ensuring a steady stream of frames for processing. It is important to balance the maximum queue size to avoid excessive memory usage while maintaining low latency.

Memory management with tracemalloc provided valuable insights into memory usage and helped optimize system performance. The results showed that increasing the max queue size reduced memory usage variability and improved FPS. This strategy is essential for detecting memory leaks and ensuring efficient memory usage, contributing to overall system stability. However, it is essential to balance memory usage and processing efficiency to ensure optimal performance.

Buffering smoothed out variations in processing time, leading to consistent FPS and reduced memory usage variability. With a max buffer size of 20, the FPS improved to 9.35 for 640x480 resolution. This technique is effective for maintaining a steady data flow and preventing bottlenecks, ensuring continuous and efficient processing of video streams. The optimal buffer size depends on the specific hardware configuration and the nature of the video stream being analyzed.

5 Conclusion

This paper has addressed the critical need for optimization strategies in real-time video analytics, particularly in resource-constrained environments such as developing countries and edge systems. Through the introduction and evaluation of several techniques—including image resizing, frame skipping, MSE-based frame skipping, parallel processing, threading, queue management, memory management with tracemalloc, and buffering—we have demonstrated significant improvements in processing efficiency and memory usage. Our experimental results show that these strategies collectively enhance frames per second (FPS) and reduce memory consumption, ensuring that real-time video analytics systems can operate effectively even with limited computational resources.

By balancing processing efficiency and accuracy, the proposed strategies make advanced AI-driven video analysis feasible in environments with restricted resources. This not only facilitates real-time monitoring and alert systems but also broadens the accessibility of sophisticated video analytics technologies to regions and applications where computational capabilities are limited. Additionally, this paper provides a practical guide for implementing these optimization techniques, offering valuable insights and actionable steps for practitioners and researchers aiming to optimize their video analytics systems.

References

1. Girshick, R.: Fast r-cnn. In: Proceedings of the IEEE international conference on computer vision. pp. 1440–1448 (2015)
2. Jiang, P., Ergu, D., Liu, F., Cai, Y., Ma, B.: A review of yolo algorithm developments. Procedia computer science **199**, 1066–1073 (2022)
3. Kastrinakis, D., Petrakis, E.G.: Video2flink: real-time video partitioning in apache flink and the cloud. Machine Vision and Applications **34**(3), 42 (2023)
4. Khadka, S., Ghimire, S.K.: Scalable solutions for efficient real-time distributed video analytics with vehicle detection on cpu edge nodes. In: Proceedings of the 2024 7th International Conference on Computers in Management and Business. pp. 73–79 (2024)
5. Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.Y., Berg, A.C.: Ssd: Single shot multibox detector. In: Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14. pp. 21–37. Springer (2016)
6. Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You only look once: Unified, real-time object detection. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 779–788 (2016)
7. Sultana, F., Sufian, A., Dutta, P.: A review of object detection models based on convolutional neural network. Intelligent computing: image processing based applications pp. 1–16 (2020)
8. Zhao, Z.Q., Zheng, P., Xu, S.t., Wu, X.: Object detection with deep learning: A review. IEEE transactions on neural networks and learning systems **30**(11), 3212–3232 (2019)