Optimizing Video Analytics for Embedded/Edge Devices

For a long time, I've been developing video analytics solutions for **embedded/edge devices**, particularly in **security systems**. One of the most critical challenges in such systems is **performance optimization**.

Below, I'll illustrate a common camera processing algorithm using **C-style pseudo-code**, followed by an optimized version that significantly improves speed—**at least 1.5x faster**.

Standard Approach (Using OpenCV)

The conventional method involves processing frames in a separate thread to avoid synchronization issues with the video source:

```
atomic<bool> ready = false;
cv::Mat image;

void thread_function()
{
    while(true) {
        if (ready) {
            // Processing (e.g., object detection, streaming)
            ready = false;
        }
    }
}

void camera_loop()
{
    cv::Mat img;
    cv::Capture capture("rtsp://....");
    create_thread(thread_function);

    while(true) {
        capture >> img;

        if (!ready) {
            img.copyTo(image);
            ready = true;
        }
    }
}
```

This ensures the main thread keeps capturing frames while the worker thread processes them. However, **copying frames (`img.copyTo`) introduces overhead**.

Optimized Algorithm: Double-Buffering Technique

After several experiments, I settled on a **more efficient approach** using **pointer swapping** and a **two-frame buffer**, eliminating unnecessary copies:

```
atomic<bool> ready = false;
cv::Mat* pimage;

void thread_function()
{
    while(true) {
        if (ready) {
            // Processing (e.g., object recognition, streaming)
```

```
            ready = false;
        }
    }
}

void camera_loop()
{
    cv::Mat img[2];
    int i = 0;

    cv::Capture capture("rtsp://....");
    create_thread(thread_function);

    while(true) {
        if (ready && !img[i].empty()) {
            pimage = &img[i];  // Swap pointer instead of copying
            i = 1 - i;          // Toggle buffer index
            ready = true;
        }

        capture >> img[i];  // Capture next frame into the inactive buffer
    }
}
```

Key Improvements

1. **Double-Buffering**: Uses two frame buffers (`img[0]` and `img[1]`) to avoid blocking the capture thread.
2. **Pointer Swapping**: Instead of copying data, the algorithm swaps pointers, **reducing memory operations**.
3. **Non-Blocking Capture**: The camera thread keeps fetching frames without waiting for processing to finish.

This method isn't limited to video—it can also optimize **sensor data processing** (e.g., microphones, lidar).

Would love to hear your thoughts or suggestions!