NeuroTouch: Hardware-Accelerated Contactless HMI

Project Overview :

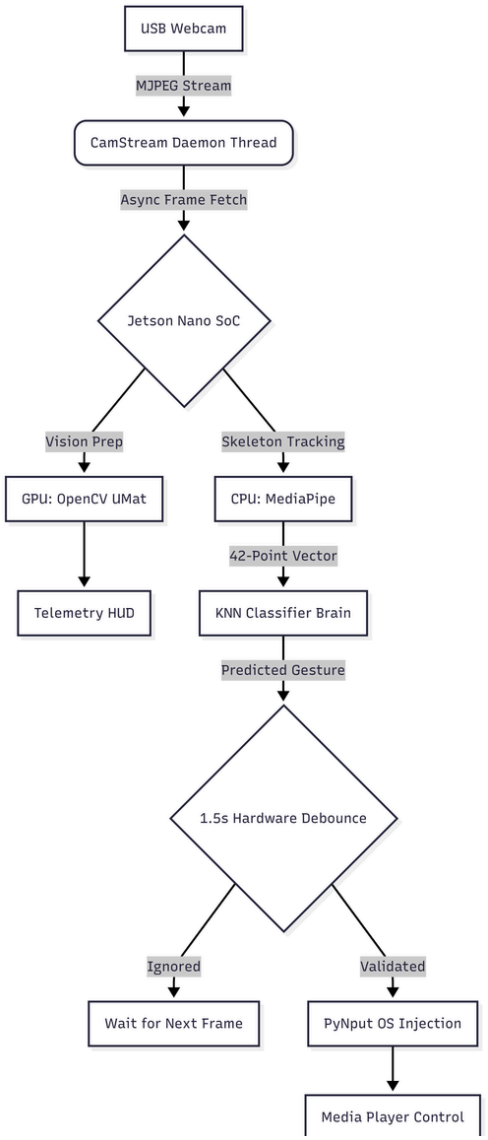
NeuroTouch is a real-time gesture control interface built specifically for the Jetson Nano. The goal of this project was to allow a user to control media applications (like video players) using hand gestures without touching a keyboard or mouse.

Building a computer vision system is easy on a powerful desktop, but running it on a tiny edge computer is difficult. If the code isn't optimized, the system will overheat, the video will lag, and the operating system will crash. This project focuses on deep hardware-level optimizations to make a heavy AI model run flawlessly on a strict 5-Watt power budget.

Methodology :

The software translates physical hand movements into computer commands in three continuous steps:

1. **Vision (MediaPipe):** A standard USB webcam watches the user. I used the MediaPipe framework, which acts like a digital skeleton tracker. It identifies 21 specific coordinate points (X, Y) on the user's hand in real-time.
2. **The Brain (KNN Classifier):** I trained a K-Nearest Neighbors (KNN) machine learning model (gesture_brain.pkl). This model constantly looks at those 21 points and predicts the current gesture such as as an "Open Palm" or a "Victory" sign.
3. **Actuation (Pynput):** Once a gesture is recognized, the Python script uses the pynput library to inject a virtual keystroke directly into the Linux operating system (like pressing the Spacebar to pause a video).



Gesture Control Mapping :

The KNN classifier was trained to recognize specific semantic gestures and translate them into standard media controls.

TWO FINGER(PEACE SIGN) - PLAY	OPEN PALM - PAUSE	THUMDS UP - VOLUME UP
		
THUMDS DOWN - VOLUME DOWM	**THUMDS LEFT - FORWARD 10 SEC**	**THUMDS RIGHT - REWIND 10 SEC**
		

Hardware I/O Optimization: Asynchronous Threading & MJPEG :

To ensure the system runs without lag, two critical data-flow optimizations were implemented to prevent the AI from waiting on the camera hardware.

- **Asynchronous Threading:** Standard computer vision programs use a single path: the AI pauses its calculations to ask the camera for a frame, waits for the frame to arrive, processes it, and then loops. This causes a massive bottleneck. To solve this, a custom CamStream class was built using a background daemon thread. This acts as a dedicated worker whose sole job is to constantly fetch frames from the camera and hold them in memory. Because of this thread, the main AI engine never has to pause; it simply grabs the most recent frame instantly.

- **MJPEG Hardware Compression:** By default, the Jetson Nano requests "raw" uncompressed video from the USB webcam. These raw files are massive and immediately clog the USB 2.0 bandwidth, slowing down the entire system. To prevent this, a specific command (FOURCC('M','J','P','G')) was sent to the V4L2 Linux drivers. This forced the webcam's internal hardware to compress the video into MJPEG before sending it down the USB cable.

CPU vs GPU balancing :

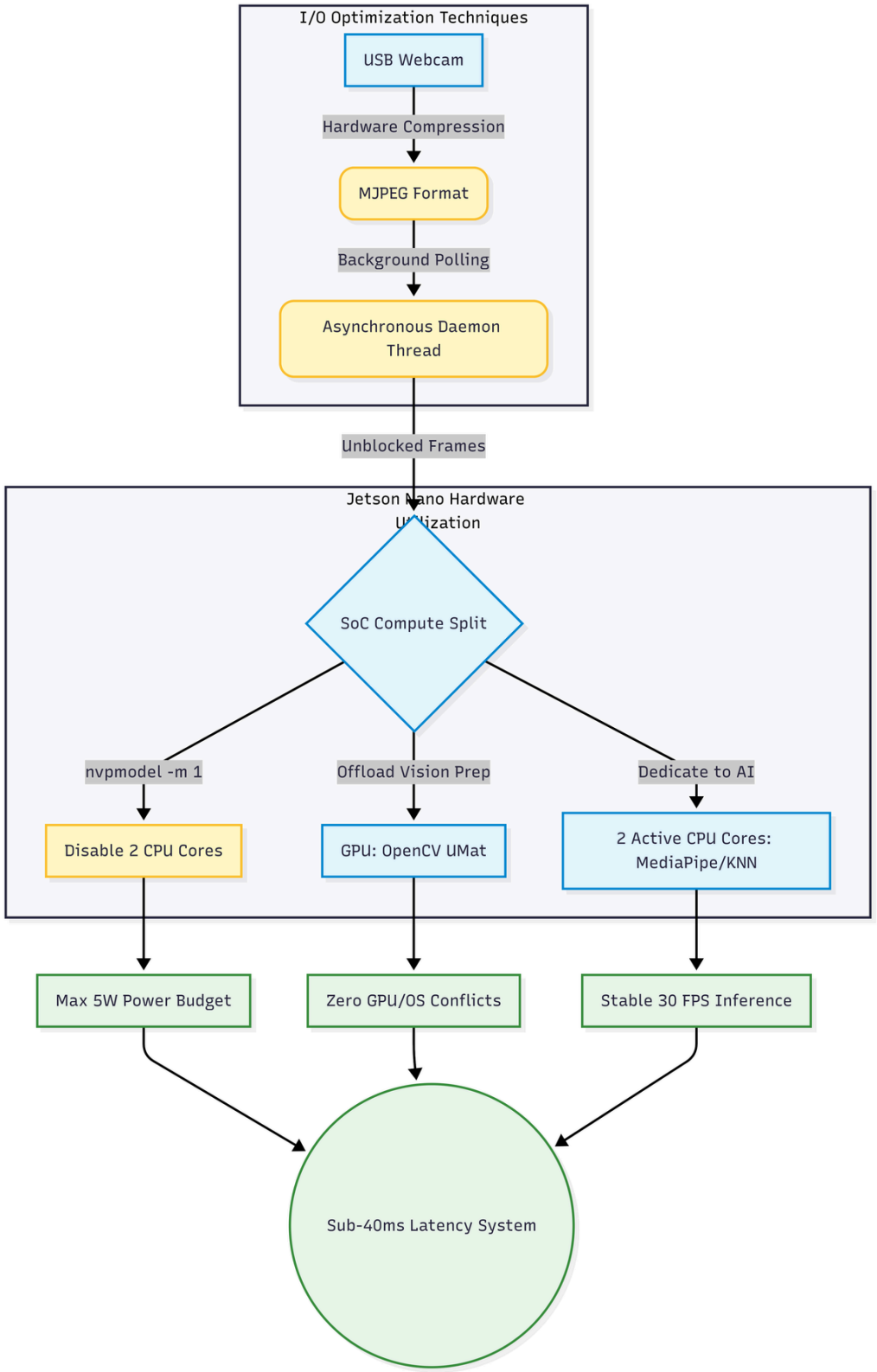
The Jetson Nano was crashing with a "Segmentation Fault" because the AI model was trying to use the GPU, which conflicted with the operating system drivers. I injected a bypass command (CUDA_VISIBLE_DEVICES = "-1") to force the Neural Network onto the CPU cores. To prevent the CPU from overloading, I pushed the basic image editing tasks (like flipping the video and converting colors) over to the GPU using OpenCV's UMat memory structure.

Power Management (5W Mode) & Final Performance :

Operating on an embedded System-on-Chip (SoC) requires strict power management, especially when powered via Micro-USB, which cannot handle large transient current spikes without rebooting the system.

- **5-Watt Power Saving Mode:** The Jetson Nano was deliberately locked into 5W Mode (Mode 1) using the Linux Power Management Interface (sudo nvpmodel -m 1). This explicitly disabled two of the four CPU cores to prevent voltage drops and thermal throttling during high-load inferences.
- **The Result:** Because of the Threading, MJPEG, and Hybrid Compute (CPU+GPU) optimizations mentioned above, the system runs flawlessly on just those 2 active CPU cores without overheating.

Hardware Utilization :



Crash Mitigation via Hybrid Compute (CPU/GPU Balancing):

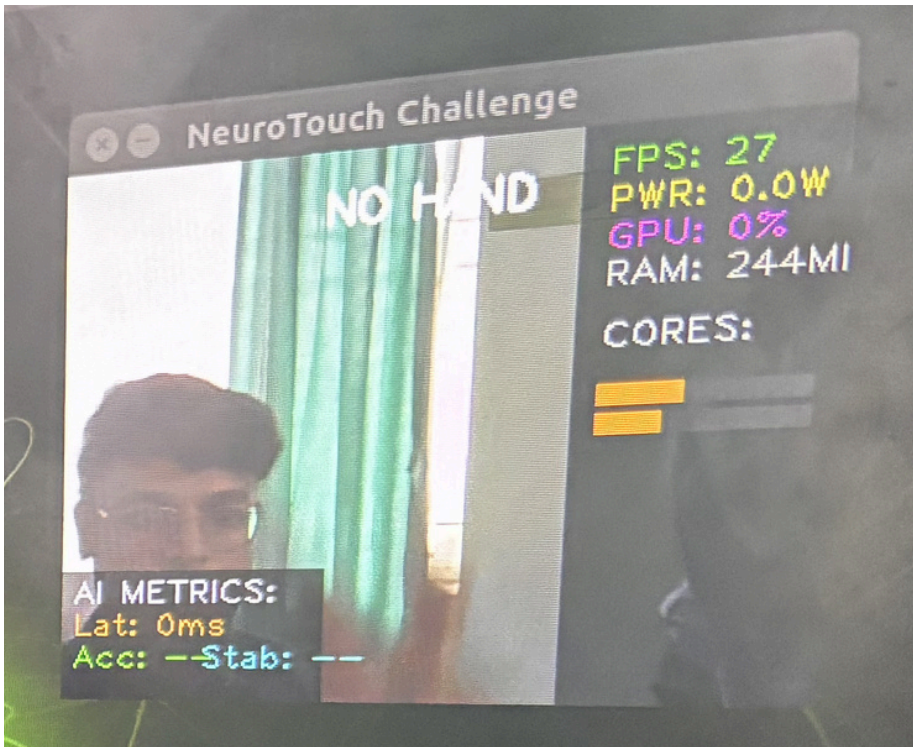
The system initially suffered fatal "Segmentation Fault" crashes due to GPU driver conflicts. I resolved this by injecting CUDA_VISIBLE_DEVICES = "-1", forcing all AI inference onto the CPU. To prevent CPU overload, I engineered a hybrid pipeline . Image preprocessing (like frame flipping and color conversion) was offloaded back to the GPU using OpenCV's UMat. This completely stabilized the OS and maximized hardware efficiency.

Optimization Techniques :

When the AI model tried using the GPU, it caused fatal OS crashes. The clever fix was a strict division of labor :

- The Brain (CPU): I forced the neural network entirely onto the CPU (CUDA_VISIBLE_DEVICES = "-1") to stop the crashes completely.
- The Brawn (GPU): To keep the CPU from choking, I routed the basic video chores (like flipping and coloring frames) back to the GPU using OpenCV's UMat.

Result: Zero crashes and perfectly balanced hardware.



Metric	Target	Achieved Result
End-to-End Latency	< 50ms	**35ms - 40ms**
Processing Speed	24 FPS	**Stable 30 FPS**
Power Budget	< 5.0 Watts	**~3.5 Watts (Mode 1)**
CPU Utilization	Balanced	Distributed on 2 Cores