Live camera flow

Computation steps

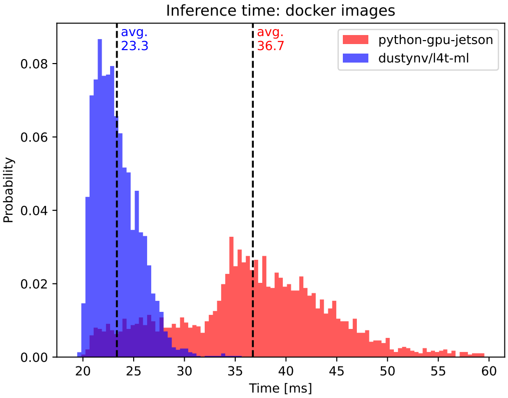
1. capture image with camera (disabled JPEG encode for Jupyter)
2. run model on image
3. filter anchors, apply non-maximum suppression
4. draw bounding boxes and fps
5. compress image to JPEG (OpenCV should use libjpeg-turbo already)
6. show image to user (MJPEG stream)

Jupyter notebook unreliable: kernel crashes, unresponsive, . . .
→ **run on a custom web server inside docker**

Docker container

- provided image outdated (2021)
- use "l4t-ml" from dustynv (11/2023)
  `https://github.com/dusty-nv/jetson-containers`
- chose matching NVIDIA JetPack/L4T
- newer libraries, **faster inference**
- unchanged performance on other parts

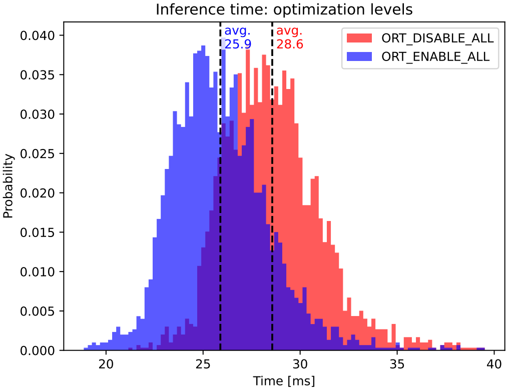
- simple web server with Flask
- low resource usage
- has endpoint serving MJPEG stream
- UI to start/stop camera



ONNX runtime

- use for inference on GPU
- supposedly faster than PyTorch
- represents model computations as graph
- use optimization `ORT_ENABLE_ALL`
  → fuses conv and batch-norm layers

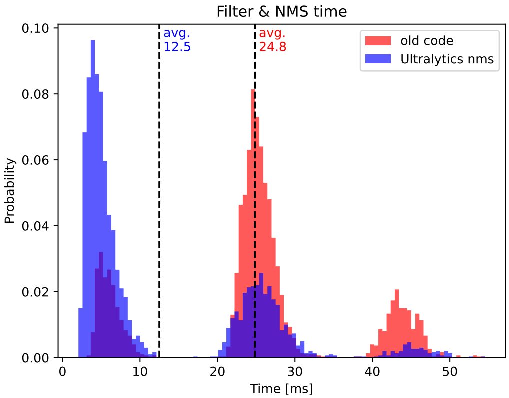
- Quantization
  - int8 → not supported by GPU
  - float16 → no speed up



Filter boxes and NMS

- original code: slow, pure Python impl.
- use `non_max_suppression()` from Ultralytics library

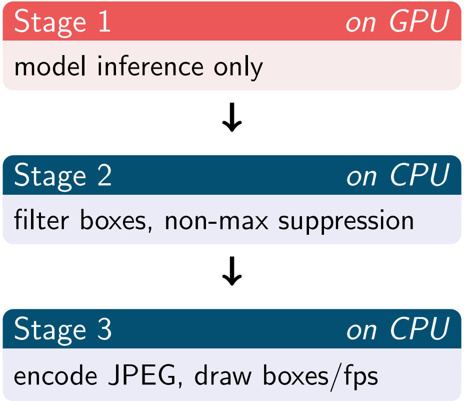
- licensed as AGPL v3
- everything combined in one function
- based on native `torchvision.ops.nms()`



Pipelining

- split work between GPU and CPU
- parallelize steps
- as Python `threading.Thread`
  → GIL mostly no issue
- connected by queues, length limited

- **Input**: image from camera callback
- **Output**: byte stream for HTTP response



Stage 1 *on GPU*

model inference only

↓

Stage 2 *on CPU*

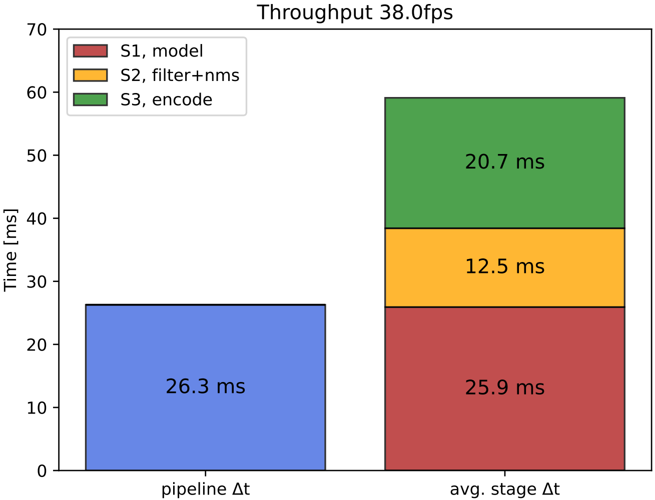
filter boxes, non-max suppression

↓

Stage 3 *on CPU*

encode JPEG, draw boxes/fps

Pipelining: results


Throughput 38.0fps
Time [ms]
S1, model
S2, filter+nms
S3, encode
26.3 ms
20.7 ms
12.5 ms
25.9 ms
pipeline Δt
avg. stage Δt