A Lightweight Camera Stack for Edge AI

Karthick Kumaran, Jui Garagate

Staff Software Engineer / Camera Software Engineer

Meta Platforms Inc

Meta

Topics

- Problem statement
- Approach
- Measurement methodology
- Threading model
- Memory allocations
- Camera metadata
- Camera HAL API
- Results
- Summary

Problem Statement

- Mobile phones use one camera for video calling, recording, snapshots, even if they have multiple cameras.
- Camera applications on mobile phones run in the foreground and consume all the hardware resources they require.
- Typically, the camera software stack is not well optimized!
- Silicon vendors provide the same camera stack for all the product segments.
- IOT devices, automotive, robots, AR/VR devices often have multiple cameras and their operations are quite different from mobile

Problem Statement – Emerging Technologies

- Emerging technologies requirements are quite different from mobile.
  - Multiple cameras operate concurrently
  - Multiple camera applications and services running in the background
  - Hardware resources should be shared across the features and applications.
  - Requires lightweight camera software to run more features (AI DNN, CV, recording, streaming, etc.,)

Approach – Lightweight Strategy

- Performance of the camera software is a major concern.
- Camera software stack should
  - Consume less compute, memory, power
  - Allow multiple camera applications / services to run in parallel
- Default camera software stack from silicon vendors
  - Consumes > 40% of the CPU
  - Allocates more memory and is not fixed (dynamically increasing)
  - Written for color camera use cases
  - Not optimized for mono cameras used for various CV / AI applications
- Develop tools to identify bottlenecks
- Develop a strategy for optimization

Why Focus on Mono Camera Optimization?

- Color cameras used for video streaming, recording, snapshot (for human visual) and object recognition (computer vision)
- Mono cameras used for various AI and CV use cases
  - Object detection
  - Object tracking
  - Simultaneous localization and tracking
  - Eye tracking
  - Hand tracking
  - Scene understanding etc.,

Measurement Methodology

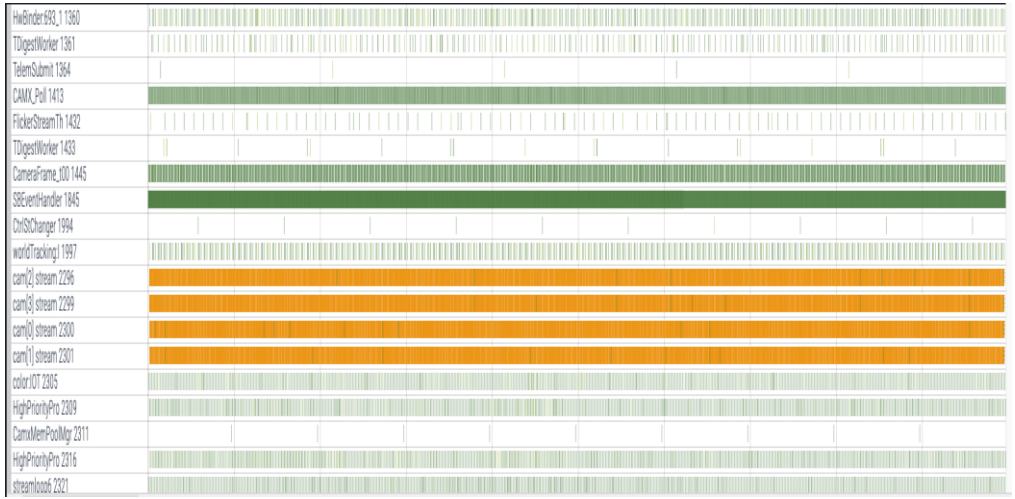
- Use same terminology for measuring CPU performance
  - CPU clusters – Big Little architecture
  - Ultimate performance CPU : ARM Cortex-X1
  - High performance CPU : ARM Cortex-A73
  - High efficiency CPU : ARM Cortex-A53
- We came up with the methodology "CPU Units"
  - Overcomes differences between
    - CPU cores, frequencies and SoCs
  - Allows apple-to-apple CPU performance comparison

Tools Used

- Pnp
- Perfetto
  - https://perfetto.dev
- Flamegraph
  - Graphical representation of CPU consumption
- CPU Units – Meta internal dhrystone benchmark based tool
- Simpleperf
  - Cache misses
  - Page faults
  - CPU cycles

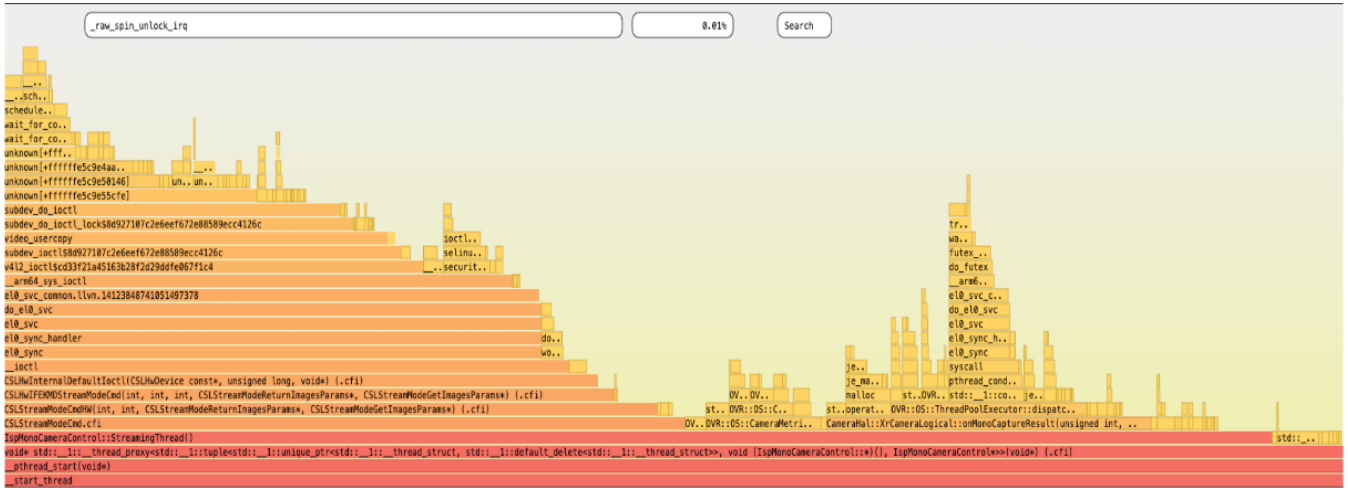
Perfetto

- Generate perfetto trace using a tool like record_android_trace from Android
- Graphical representation of
  - Thread timelines
  - Priorities
  - Scheduling etc.,



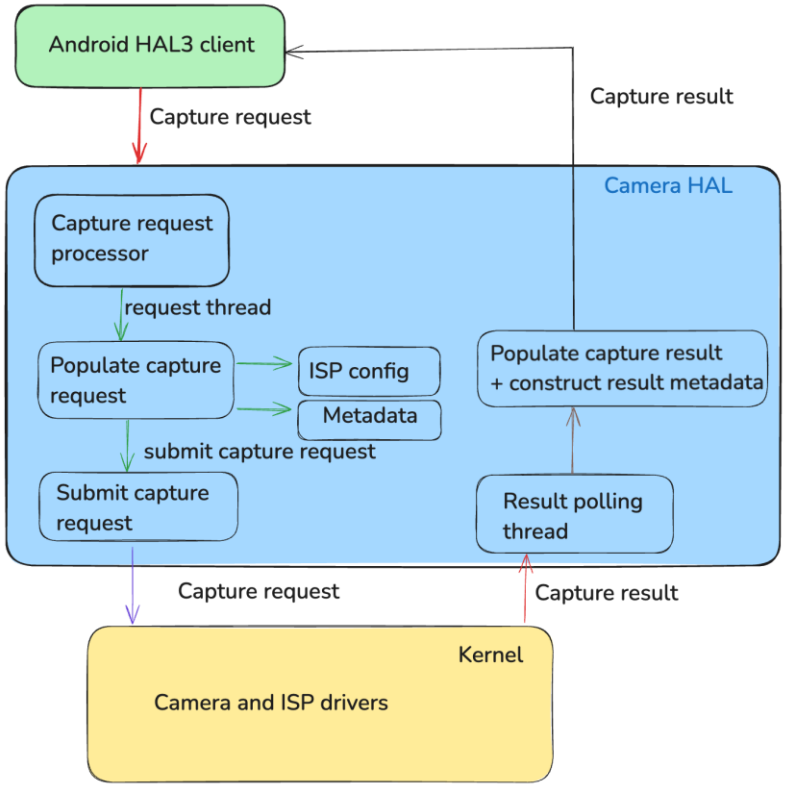
Flamegraph

- Graphical representation of
  - CPU consumption of the code
  - Identify threads consuming more CPU



Bottlenecks

- Android HAL3 based APIs
  - Request result model
  - Significant CPU overhead for each camera frame
  - Multiple cameras running at high fps result in significantly high CPU overhead
- Threading model
  - Multiple threads for request, result
  - Context switching
  - Synchronization
- Per-frame memory allocations
- Camera metadata and logging



Memory Allocations

- Per-frame requests have mallocs()
  - Degrades performance significantly
- We found ~40K mallocs() per second
- mallocs() are expensive
  - Can cause cache misses
  - Impacts system performance
  - Varying memory load
- Use tools like "bpftrace"
  - For measuring mallocs()

Memory Allocations (continued)

- Developed internal tools for profiling memory
- We created containers
  - Operate like C++ STL containers
  - But won't allocate dynamic memory
  - Vector usage throughout camera software
- Explicitly allocate memory at startup
  - For high frame rate per-frame operations
- Reduced mallocs() from ~40k to 8k per second
- Resulted in significant CPU load reduction

Camera Metadata

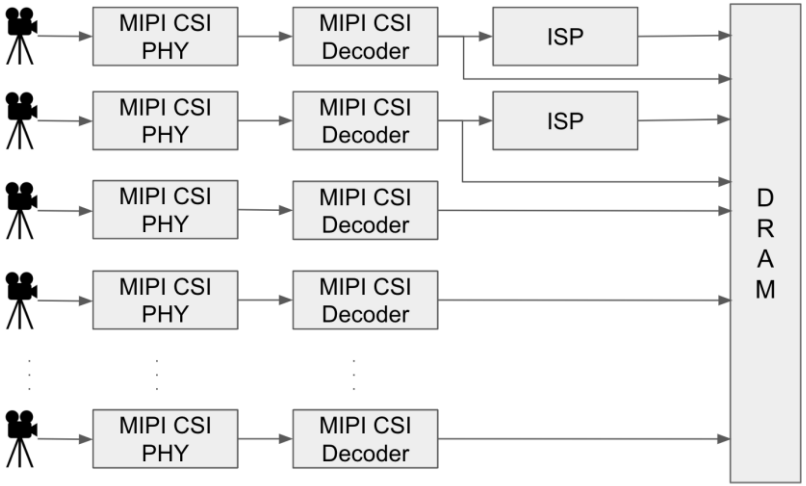
- Performance profiling showed camera software code is memory-bound
  - Memory bound – CPU stalls for memory access
  - Camera metadata – main contributor
- Android camera metadata
  - Contains huge set of metadata tags
  - Searching / reading / writing / updating metadata is expensive
- Some use cases don’t need all the metadata
- Identify the required metadata for your features / use cases

Camera Metadata (continued)

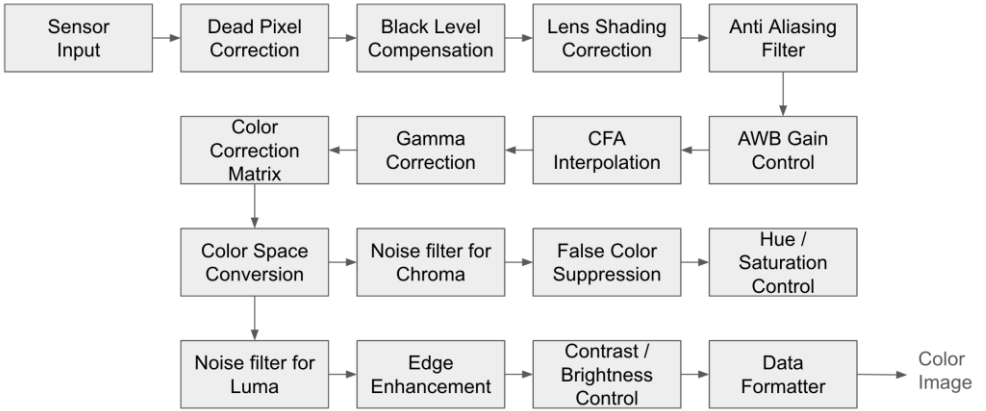
- Mobile phones require an extensive set of camera metadata
- But always running (multiple) cameras used for AI / CV features don't:
  - Several metadata tags not relevant for AI / CV
  - Remove unnecessary metadata
  - Simpler caching
  - Refactor the metadata search / update code
- Above optimizations resulted in significant performance improvement

Camera Hardware Pipeline

Camera Hardware Architecture



ISP Hardware Blocks



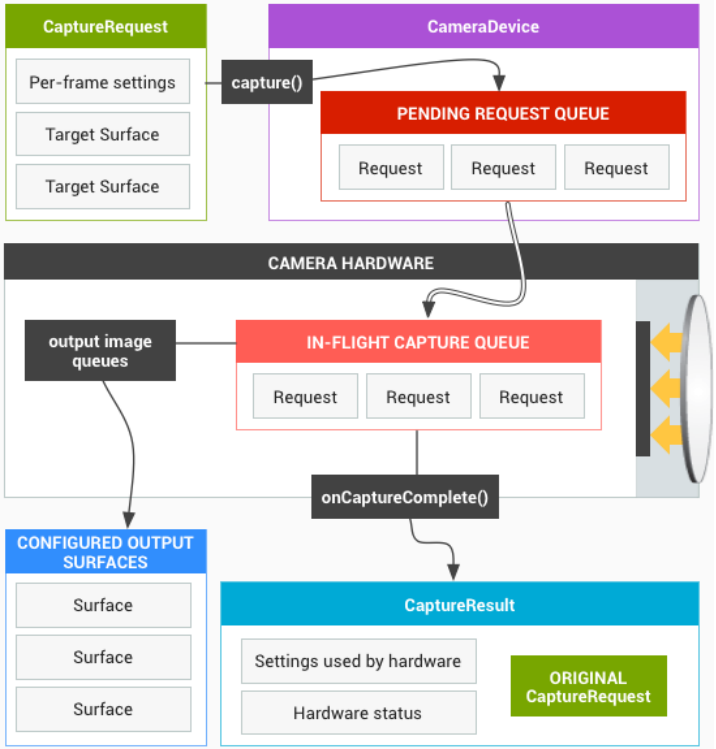
Credits: OpenISP

Threading Model

- Multiple threads
  - Capture request
  - Capture result
  - Synchronization
- Having the right thread priorities for each thread is critical
- Real time priorities for camera threads
  - Stream multiple cameras
  - Work under high memory pressure scenarios
- Reduce number of threads and context switching

Camera HAL3 API

- Silicon vendors camera software is based on Android HAL3 APIs
- Simpler camera pipelines don't need the complex HAL3 APIs
- For AI/CV use cases with mono cameras
  - Capture request not required
  - Per-frame settings not required
  - Multiple threads for multiple cameras not required
  - Simpler camera metadata is sufficient
  - Simpler ISP pipeline
  - Lightweight camera software for high performance and frame rate

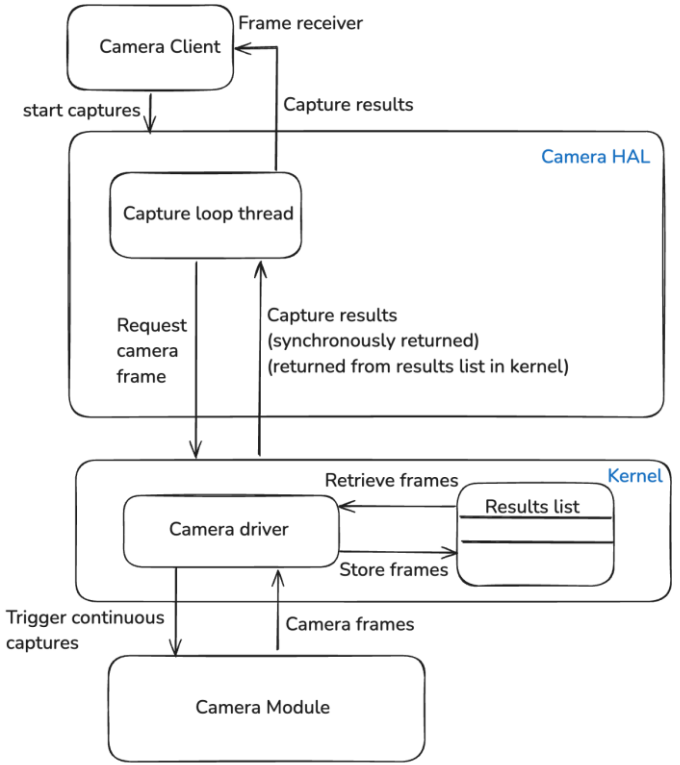


Camera HAL3 API (continued)

- Developed a lightweight camera software architecture
  - No capture requests – continuously streams frames until the camera is stopped
  - Simple and lightweight metadata
  - Circular queue created between kernel driver and HAL for synchronization
  - Kernel driver stores the captured frame in a per-camera list
  - Single thread retrieves camera frames from the list
  - Notifies client when frame is available
  - Updates circular queue after frame consumption
- Observed significant improvements in performance
- 80% reduction in CPU consumption

Simplified Camera Architecture for Mono Cameras

- Initialize the camera
- Allocate and queue the buffer
- Start Capture
- Camera module streams continuously
- Camera driver captures frames and writes to the buffer in a loop
- Capture loop thread retrieves frames from the list
- Capture results delivered to the client
- Only one thread
- Simple camera metadata (ex., timestamp, frame number etc.,)



Results

Cameras	CPU Utilization (CPU Units)		Memory Allocations	
	Before	After	Before	After
Mixed Reality (Color cameras)	1200	326 (73% gain)	22k-40k	7.5k (66% gain)
Virtual Reality (Mono cameras)	530	20 (95% gain)	18k	1.3k (90% gain)

Summary

- Use and build tools to identify bottlenecks in the camera software
- Build a strategy to optimize the code
- Identify and remove unused features
- Refactor / Re-architect the code if necessary
- Optimizations explained in this talk can be applied to most mono cameras software and applies to all silicon vendors camera software stack
- Some optimizations can be used to any software stack
- Lightweight camera software helps to
  - Add new features
  - Reduce memory throughput, CPU load, power consumption

Resources

- https://www.brendangregg.com/flamegraphs.html
- https://android.googlesource.com/platform/external/perfetto/+/master/tools/record_android_trace
- https://android.googlesource.com/platform/hardware/libhardware/+/refs/heads/main/include_all/hardware/camera3.h
- https://perfetto.dev/
- https://android.googlesource.com/platform/system/media/+/refs/heads/main/camera/include/system/camera_metadata_tags.h