REAL TIME MONOCULAR DEPTH ESTIMATION ON EDGE AI DEVICE

TEAM PIXEL-PAC

VIVEK
SHWETANK
AMRIT
RATISH

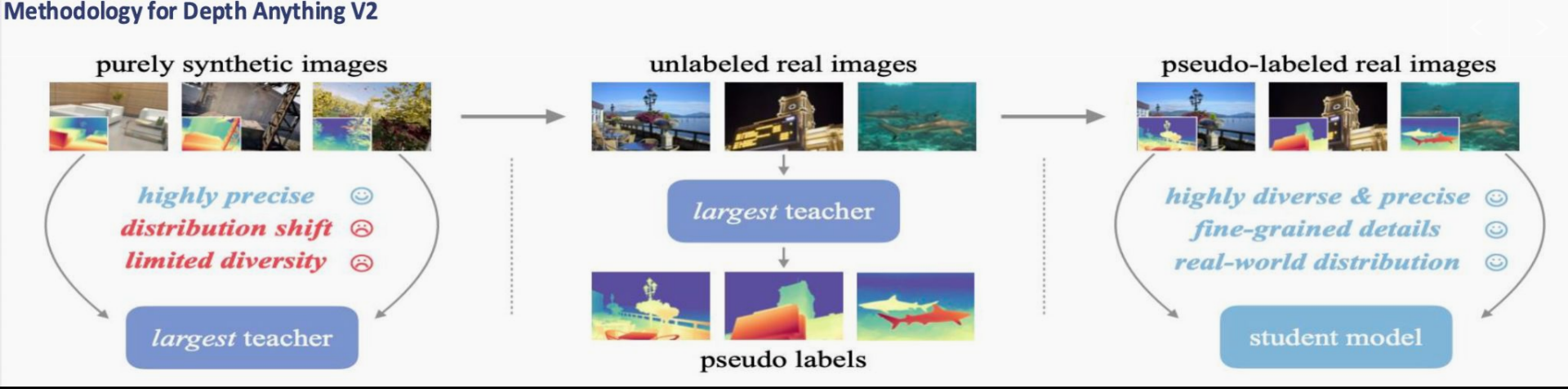
EXECUTIVE SUMMARY

Project Goal: A Deterministic AI System for Collision Mitigation

The primary objective of this project is to develop a low-cost, real-time system for monocular depth estimation, suitable for deployment on edge AI devices like the Raspberry Pi 5. The system is designed to enhance 3D perception and provide timely collision avoidance alerts for operational vehicles.

This requires an operating system foundation that can guarantee **timely execution of critical tasks**—from image capture to hardware alerts.

Key Achievement:
We have successfully transformed a standard Raspberry Pi 5 into a specialized, low-latency platform that can handle Real-Time Depth Estimation by meticulously building, deploying, and validating a custom Linux kernel with real-time capabilities (PREEMPT_RT).



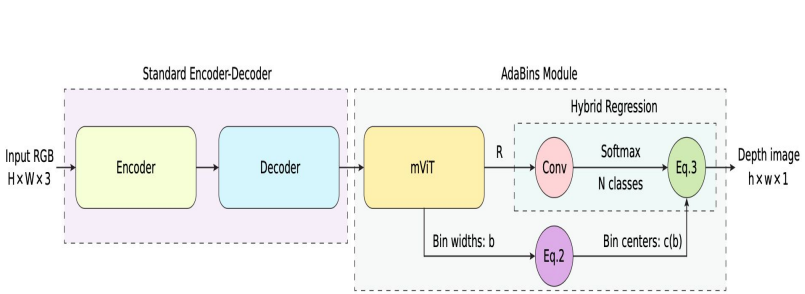
Methodology for Depth Anything V2

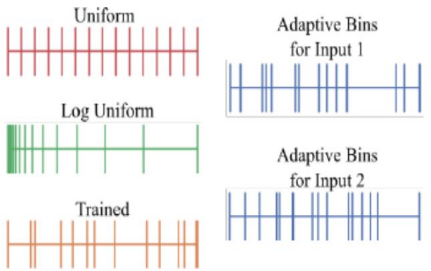
Then, to mitigate the distribution shift and limited diversity of synthetic data, we annotate unlabeled

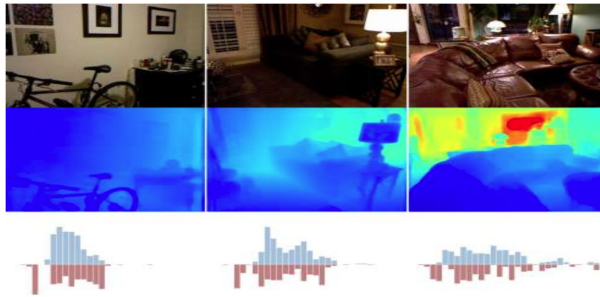
real images with the teacher. Finally, we train student models on high-quality pseudo-labeled images.

- 3 step process
  1. Uses already available labelled dataset to make a teacher network
  2. Now unlabeled datasets are passed through this network to get the annotations(pseudo labelled dataset)
  3. now the student network is trained on both labelled and pseudo labelled dataset (won't really surpass teacher)
- Depths are initially mapped to a disparity space and normalized, 2 Loss functions are used
  1. Scale and shift invariant- simply- the shift and scale of output is same as of Input and is independent of time
  2. Gradient matching loss

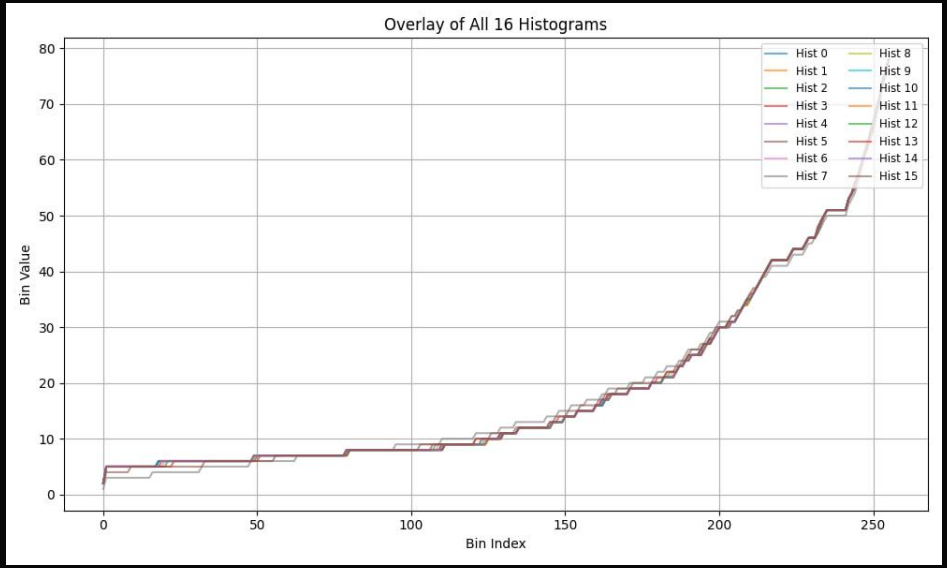
Model Motivation Adabins







- Encoder decoder for feature extraction
- Light vision model for depth estimation
- The model outputs attention score
- We use softmax to get the probability for depth of each pixel
- We get the bin centers
- Use the bin centers and probability score to build a depth map using softmax



Used Mobilenet based custom encoder and NNconv5 Upscaling decoder, with a simple and light custom transformer based adabins module

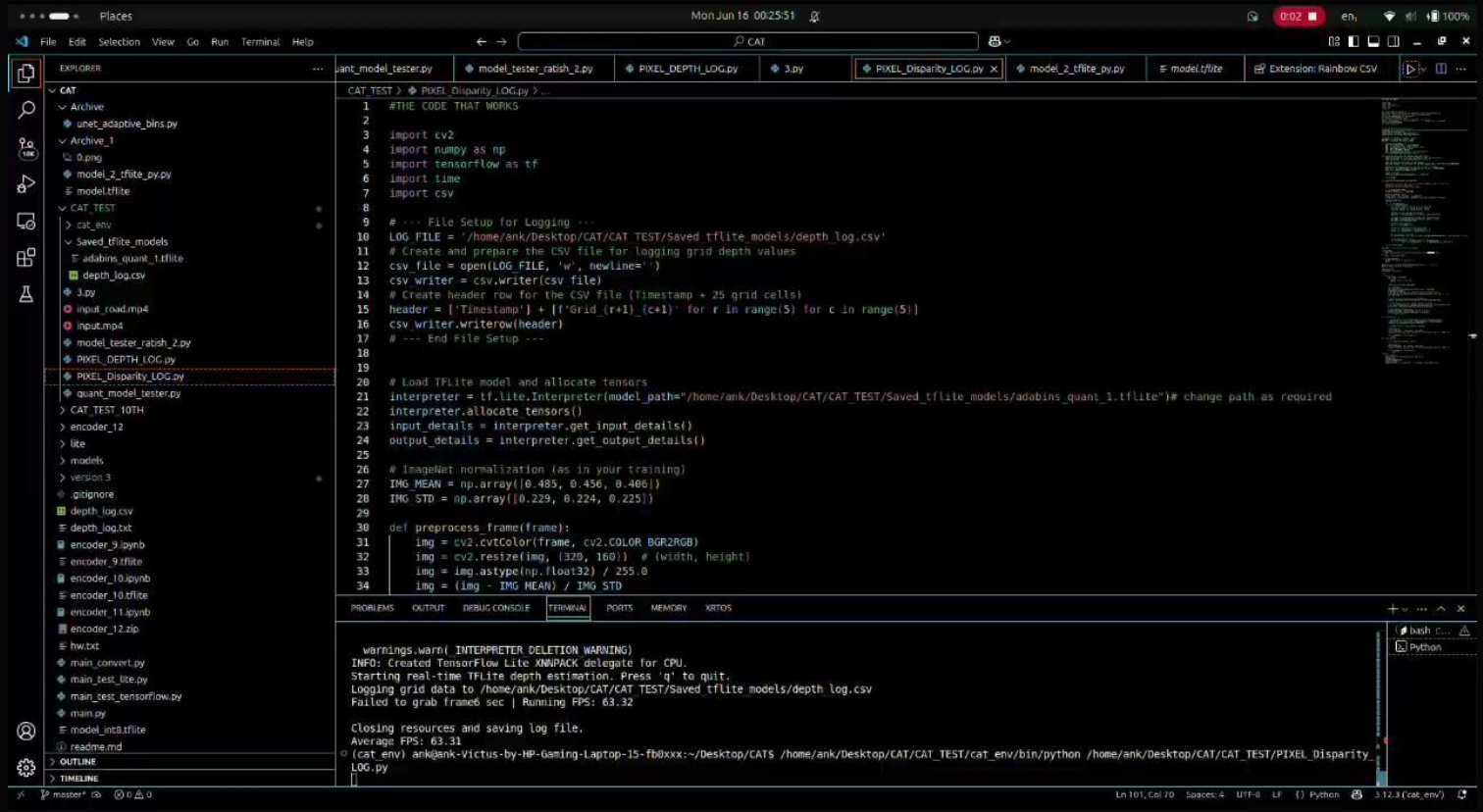
Hyper parameters:
Input : (240,240,3)
Embedding dim : (32)
Num_heads = 2
Num_Layers = 1
Patch size = 10
Batch size = 16
No of images = ~87000 (50Gb)
Total parameters < 6,00,000 (very light)
Silog and distillation loss using adabins
0.4 gt + 0.75 depth loss + 0.75 distill loss

Training methodology:
Knowledge distillation:
Teacher: Efficient B5 based Adabins model
Student: our custom model
Features taken : Pen-ultimate outputs (depth-maps,bin centers and widths)

Training iterations : 5, 5 iterations - 5 challenges

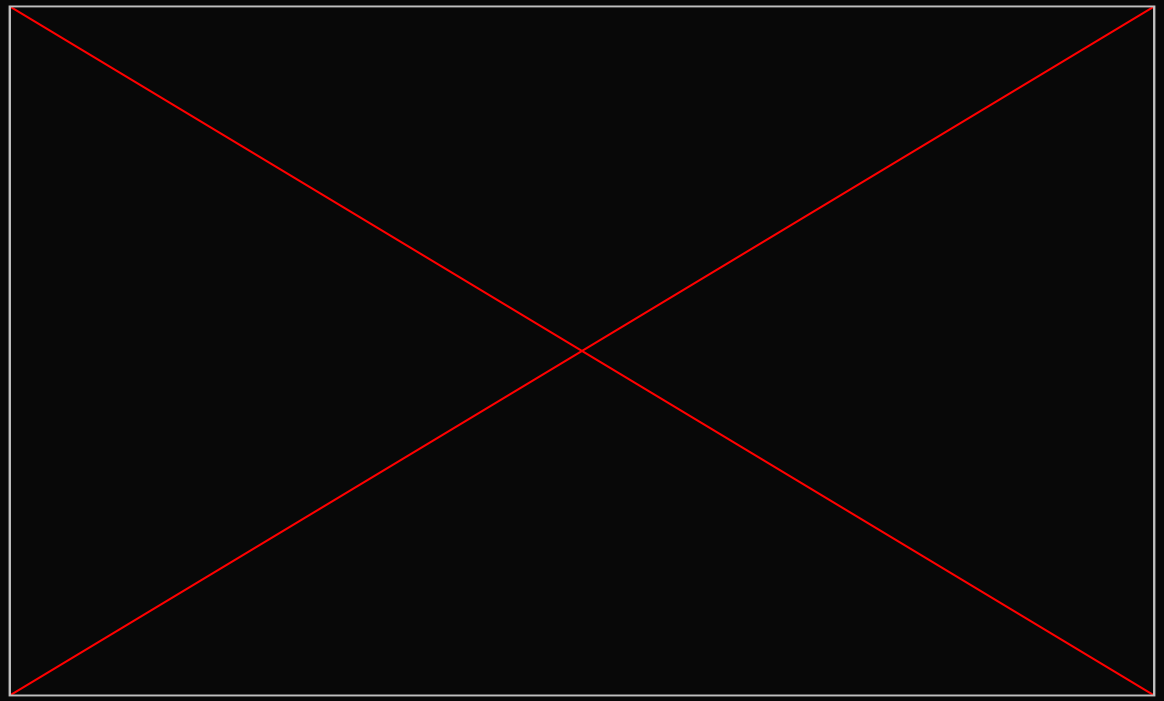
Challenge	Solution
Data collection / preprocessing and management	214GB KITTI dataset, Only ~55Gb usable)
Knowledge distillation - Slow teacher inference	Offline distillation
Offline distillation-(resource constraints)- more time taken	Usage of h5py, tfrecords
Incorrect feature extraction	Used feature extraction checks and live monitoring
RAM & storage constraints	Producer consumer approach

Result of Phase 1



What now?

Ok let’s make it stronger!



Adabins - training pipeline

Version 1

Untrained encoder decoder with miniVit model
Parameters :
Input : (240,240,3)
Embedding dim : (64)
Num_heads = 4
Patch size = 16
Batch size = 16
No of images = ~7000
Epochs = 50

Silog and distillation loss using adabins
0.4 gt + 0.75 depth loss + 0.75 distill loss

Version 3

Untrained encoder decoder with miniVit model
Parameters :
Input : (240,240,3)
Embedding dim : (64)
Num_heads = 4
Patch size = 16
Batch size = 16
No of images = ~7000
Epochs = 50
AdamW optimizer
Silog and distillation loss using adabins
0.3 gt + 0.5 depth loss + 0.8 distill loss

Version - 7

Untrained different (with swish activation)encoder decoder with miniVit model

Parameters :
Input : (240,240,3)
Embedding dim : (64)
Num_heads = 4
Patch size = 16
Batch size = 16
No of images = ~7000
Epochs = 50
AdamW optimizer

```
Processing frame 228
Inference time: 177.79 ms
Processing frame 229
Inference time: 178.89 ms
Processing frame 230
Inference time: 181.82 ms
```

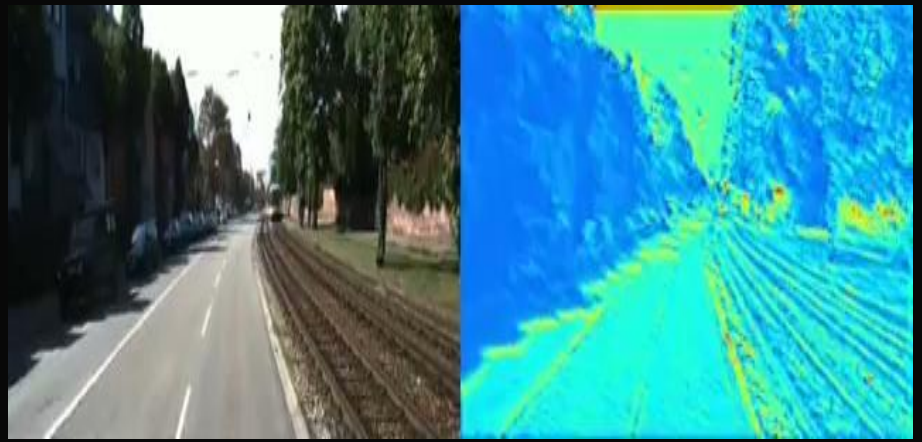
Silog and distillation loss using adabins

0.4 gt + 0.75 depth loss + 0.75 distill loss

Lr-Scheduler

Version - 12

Untrained encoder decoder with miniVit model
Parameters :
Input : (240,240,3)
Embedding dim : (64)
Num_heads = 4
Patch size = 16
Batch size = 16
No of images = ~7000
Epochs = 50
AdamW optimizer

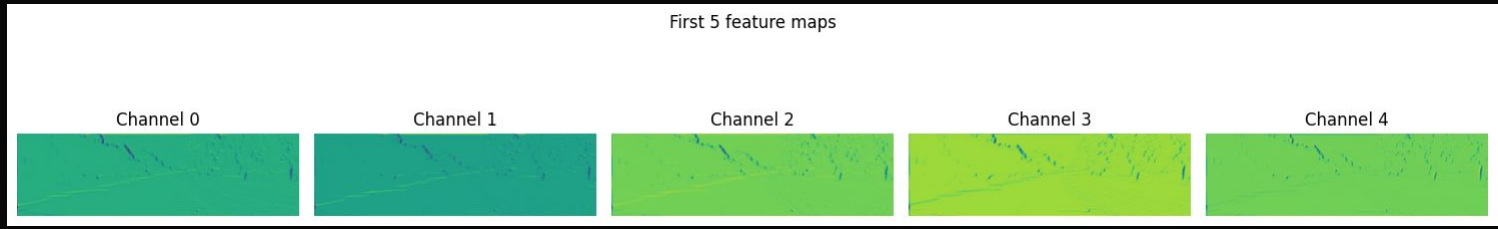


Silog and distillation loss using adabins

0.3 gt + 0.5 bins loss
Learning from GT and bins from teacher

Model 2 version - 1 (total 3 hybrid model)

Untrained encoder decoder with miniVit model
Parameters :
Input : (176,608,3)
Embedding dim : (64)
Num_heads = 4
Patch size = 16
Batch size = 16
No of images = ~7000
Epochs = 50
Trained with output of range attention map R and bin b
R - shape is in 64 dims
Silog and distillation loss using adabins
0.5 depth loss + 0.75 bin loss + 0.1 cosine loss
Cosine loss uses max pool over H*W

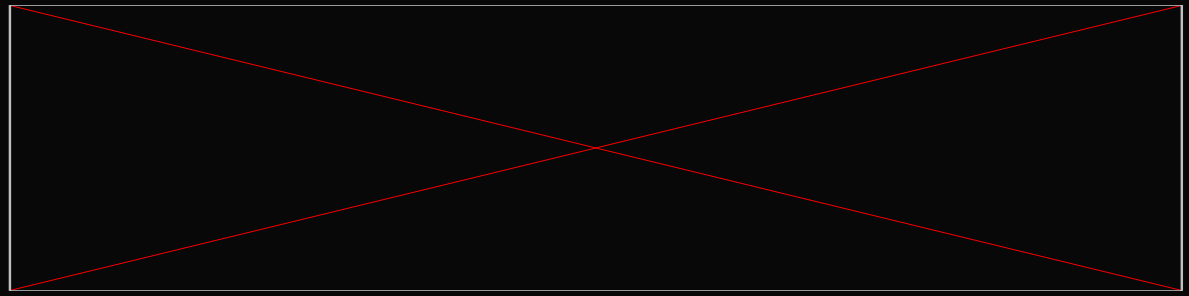


Untrained encoder decoder with miniVit model
Parameters :
Input : (256,256,3)
Embedding dim : (64)
Num_heads = 4
Patch size = 16
Batch size = 16
No of images = ~7000
Epochs = 50
Trained with output of range attention map R and bin b
Silog and distillation loss using adabins
0.5 depth loss + 0.75 bin loss + 0.1 cosine loss
Cosine loss uses max pool over H*W

ADALite - CNN + MHSA

Untrained CNN with with multi head in deep layers

Parameters :
Input : (256,256,3)
Num_heads = 4
No of images = ~17000
Epochs = 250
Silog loss
Lr scheduler



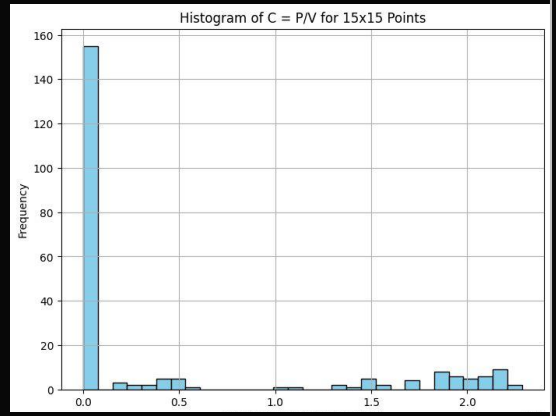
mapping relative map to depth map

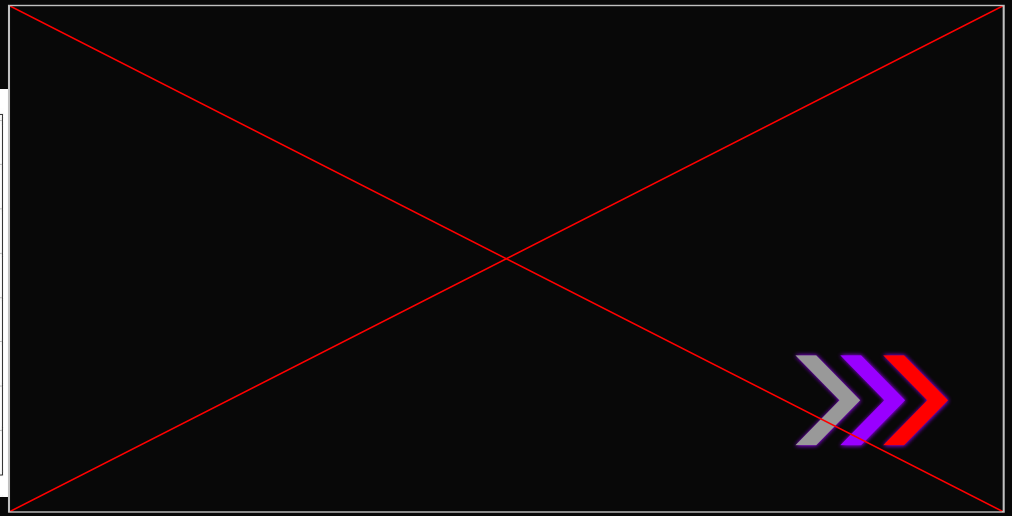
1)Lightweight cnn

2)Linear regression

Rapi inference time
(For model only)

```
ty_parameters': {}}]
time taken 111.00935935974121 ms
Model output: (1, 256, 256, 1)
time taken 110.90779304504395 ms
Model output: (1, 256, 256, 1)
time taken 111.71174049377441 ms
```



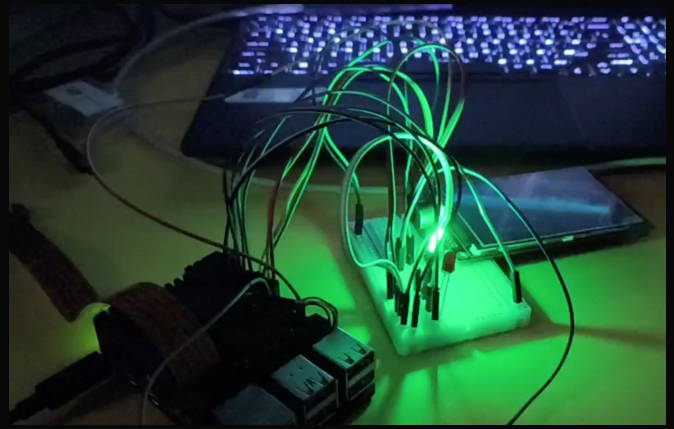


CI/CD pipeline for training & deployment

CI/CD FLOW

1) Data ingestion
2) Data Preprocessing
3) Loading the teacher model and student model based on defined architecture
4) Training pipeline
5) Model conversion from tensorflow to tflite
6) All the above steps are logged
7) Dockerization of the above project has been done
8) Github workflows was setup and tested

HARDWARE SYSTEM ARCHITECTURE



Software Stack:

- **Base OS:** Raspberry Pi OS (64-bit, Debian Bookworm)
- **Kernel: Custom 6.15.0-rc7 PREEMPT_RT Kernel**
- **Application Layer:** Python 3 Virtual Environment (cat_venv)
- **Core Libraries:** TFLite-Runtime, OpenCV, Picamera2, gpiozero, matplotlib

Hardware Platform:

- **Compute:** Stock Raspberry Pi 5 (8GB RAM, 2.4GHz Quad-Core Arm Cortex-A76)
- **Input:** Raspberry Pi Camera Module 3
- **Output:**
  - **Visual Feedback:** HDMI Display (via Console Framebuffer)
  - **Alert System:** GPIO-driven Red/Green LEDs & Active Buzzer

Why a Real-Time Kernel is Essential for This Project?

THE CORE CHALLENGE

A standard Operating System (like stock Raspberry Pi OS) is designed for **high average throughput**, not for timeliness. A critical task can be delayed by lower-priority system activities. This unpredictable delay is called **jitter**.

- **General-Purpose OS (e.g., Stock Linux):**
  - Optimized for multitasking and fairness.
  - **High Jitter:** An alert might trigger in 10ms one time, but 150ms the next.
  - Unacceptable for safety-critical collision alerts.

- **Real-Time Operating System (RTOS / PREEMPT_RT Kernel):**
  - Optimized for **predictability and determinism**.
  - **Low Jitter:** Ensures that high-priority tasks execute within a strict, predictable time bound.
  - **Our Goal:** To provide a guarantee that when a depth estimation is complete, the subsequent **alert happens *now*, EVERY TIME.**

OUR SOLUTION:

CUSTOM-BUILT PREEMPT_RT KERNEL

Modified the Linux kernel itself to provide real-time capabilities while retaining the extensive hardware support and software ecosystem of Raspberry Pi OS.

Kernel Specification:

- **Version:** 6.15.0-rc7-v8-16k-NTP+
- **Build Method:** Natively compiled on the Raspberry Pi 5.
- **Core Feature:** PREEMPT_RT patchset functionality enabled directly in the kernel configuration.

Key Real-Time Configurations Applied:

01	**CONFIG_PREEMPT_RT=y**	Enabled the fully preemptible kernel, minimizing sources of non-deterministic latency.
02	**CONFIG_HZ_1000=y**	Set the system timer frequency to 1000 Hz, allowing for finer-grained task scheduling (1ms resolution).
03	**CONFIG_NO_HZ_FULL=y**	Enabled "full tickless" mode, eliminating periodic timer interrupts on isolated CPU cores to reduce jitter.
04	**Performance CPU Governor**	Locked the CPU at its maximum frequency (2.4GHz) to eliminate latency from frequency scaling.
05	**Disabled Kernel Debugging**	Removed all debugging overhead from the final kernel image.

- **Problem 1: Kernel & Patch Mismatch**

Issue: Initial attempts to use the latest kernel sources (rpi-6.6.y) failed because no matching PREEMPT_RT patch was available.

Solution: Pivoted to an older but more stable Long-Term Support (LTS) branch (rpi-6.1.y) where we could successfully find and apply a compatible patch (6.1.90-rt30).

Problem 2: Critical Boot Failure

Issue: After the first deployment, the Raspberry Pi 5 failed to boot. Reverting to a backup of the original kernel also failed, indicating a corrupted boot partition.

Solution: A full **system re-flash** with a clean Raspberry Pi OS image was performed. This provided a known-good baseline and led to the adoption of a safer deployment strategy (using os_prefix) that did not overwrite the main boot files.

Problem 3: Application & Environment Conflicts

Issue: The application faced numerous Python errors, including RPi.GPIO failing on the new kernel, ModuleNotFoundError for libcamera, and X11 display errors when running with sudo.

Solution: Systematically resolved by:

- Switching from RPi.GPIO to the more modern gpiozero library.
- Recreating the Python virtual environment with the --system-site-packages flag.
- Using sudo -E to preserve the user environment when launching the GUI application with real-time priority.

SYSTEM VALIDATION AND RTOS BENCHMARKING

Validation - Latency Benchmarking (Idle & CPU Load)

Quantifying Real-Time Performance: *cyclictest*

cyclictest is the industry standard for measuring scheduling latency (jitter). It measures the difference between when a high-priority task is supposed to run and when it actually does. Lower values are better.

Results on our PREEMPT_RT Kernel:

- **Mostly Idle System:**
  - Achieved a **maximum latency of ~15µs** (microseconds).
  - Demonstrates exceptional responsiveness when the system is not under heavy application load.
- **Under Extreme CPU Stress** (stress-ng --cpu 4):
  - Even with all 4 CPU cores at 100% utilization, the maximum observed latency remained incredibly low, **under 20µs**.
  - This proves the PREEMPT_RT scheduler is effective at preempting low-priority work for critical tasks.

```
# Total: 000855978
# Min Latencies: 00002
# Avg Latencies: 00003
# Max Latencies: 00015
# Histogram Overflows: 00000
```

```
pi@depth-pi:~$ uname -a
Linux depth-pi 6.15.0-rc7-v8-16k-NTP+ #1 SMP PREEMPT_RT Mon May 26 23:21:13 IST 2025 aarch64 GNU/Linux
pi@depth-pi:~$ sudo cyclictest -t -p 90 -N -i 1000 -l 500000 -a
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.05 0.03 0.02 1/271 3094

T: 0 ( 3079) P:90 I:1000 C: 500000 Min:   2061 Act:    3199 Avg:    3395 Max:   15178
T: 1 ( 3080) P:90 I:1500 C: 333335 Min:   2094 Act:    3037 Avg:    3706 Max:   21715
T: 2 ( 3081) P:90 I:2000 C: 250001 Min:   2169 Act:    2983 Avg:    4006 Max:   14659
T: 3 ( 3082) P:90 I:2500 C: 200001 Min:   2162 Act:    2902 Avg:    3940 Max:   12349
pi@depth-pi:~$ sudo nmcli device wifi list
```

Validation - Performance Under Combined Load

Robustness Under CPU and Memory Pressure

The most challenging test involved stressing both CPU and memory simultaneously to simulate a demanding real-world scenario.

Test Conditions:

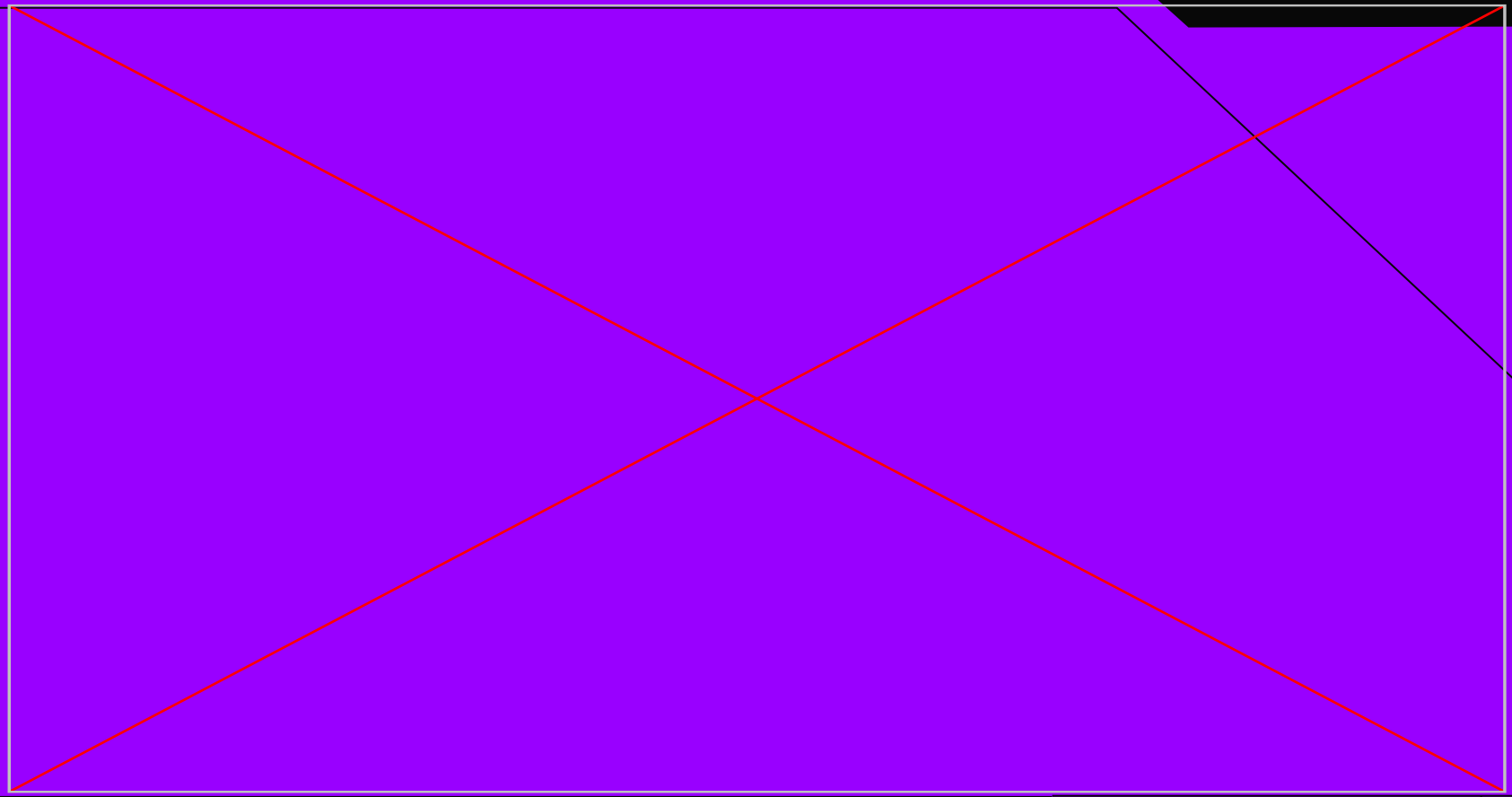
- **CPU Load:** stress-ng --cpu 4 (all cores at 100%)
- **Memory Load:** stress-ng --vm 4 --vm-bytes 1G (4GB of RAM actively stressed)
- **Monitoring Tool:** cyclictest running with high priority (-p 90).

Results:

- The system remained stable with **zero swap usage**.
- The maximum observed scheduling latency across all cores was **~116µs to 191µs**.

Conclusion:

Even under extreme, combined load, our PREEMPT_RT kernel maintained worst-case scheduling latencies well below one millisecond. This confirms the system is robust and deterministic, a critical requirement for a safety-oriented application.



BENCHMARKING PICTURES

BASE RTOS

```
pi@depth-pi:~ $ sudo cyclictest -t -p 90 -N -i 1000 -l 500000 -a
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 1.04 1.03 0.62 1/435 5555

T: 0 ( 3469) P:90 I:1000 C: 500000 Min:   1317 Act:    5110 Avg:    3800 Max:   35690
T: 1 ( 3470) P:90 I:1500 C: 333338 Min:   1363 Act:    5914 Avg:    3978 Max:   32358
T: 2 ( 3471) P:90 I:2000 C: 250003 Min:   1367 Act:    5049 Avg:    4133 Max:   32929
T: 3 ( 3472) P:90 I:2500 C: 200002 Min:   1407 Act:    4174 Avg:    4294 Max:   33768
```

UNDER HEAVY STRESS

```
(tflite_env) pi@depth-pi:~ $ sudo cyclictest -t -p 90 -N -i 1000 -l 500000 -a
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 1.71 1.50 0.95 2/419 4775

T: 0 ( 4295) P:90 I:1000 C: 500000 Min:   1258 Act:    3126 Avg:    3592 Max:   70978
T: 1 ( 4296) P:90 I:1500 C: 333338 Min:   1305 Act:    4732 Avg:    3853 Max:   46832
T: 2 ( 4297) P:90 I:2000 C: 250004 Min:   1324 Act:    3621 Avg:    4200 Max:   65323
T: 3 ( 4298) P:90 I:2500 C: 200003 Min:   1372 Act:    2715 Avg:    4511 Max:   44753
(tflite_env) pi@depth-pi:~ $
```

Min latency: 2007 ns (~2.0 µs)
Average latency: 3177 ns (~3.2 µs)
Maximum latency: 15290 ns (~15.3 µs)

Application Performance & Real-Time Tuning

Optimizing the Monocular Depth Application

To achieve the best performance, we applied real-time tuning principles directly to the application.

1. CPU Core Isolation:

- We configured the kernel's boot arguments (isolcpus=0,1,2,3, nohz_full=0,1,2,3) to dedicate CPU cores 0, 1, 2 and 3 exclusively for our application, protecting them from other OS-level tasks and interrupts.

2. Thread & Process Pinning:

- Using the *taskset -c 1,2,3* command, we pinned the Python application process directly to our isolated cores (0,1, 2 and 3).

3. Real-Time Priority Scheduling:

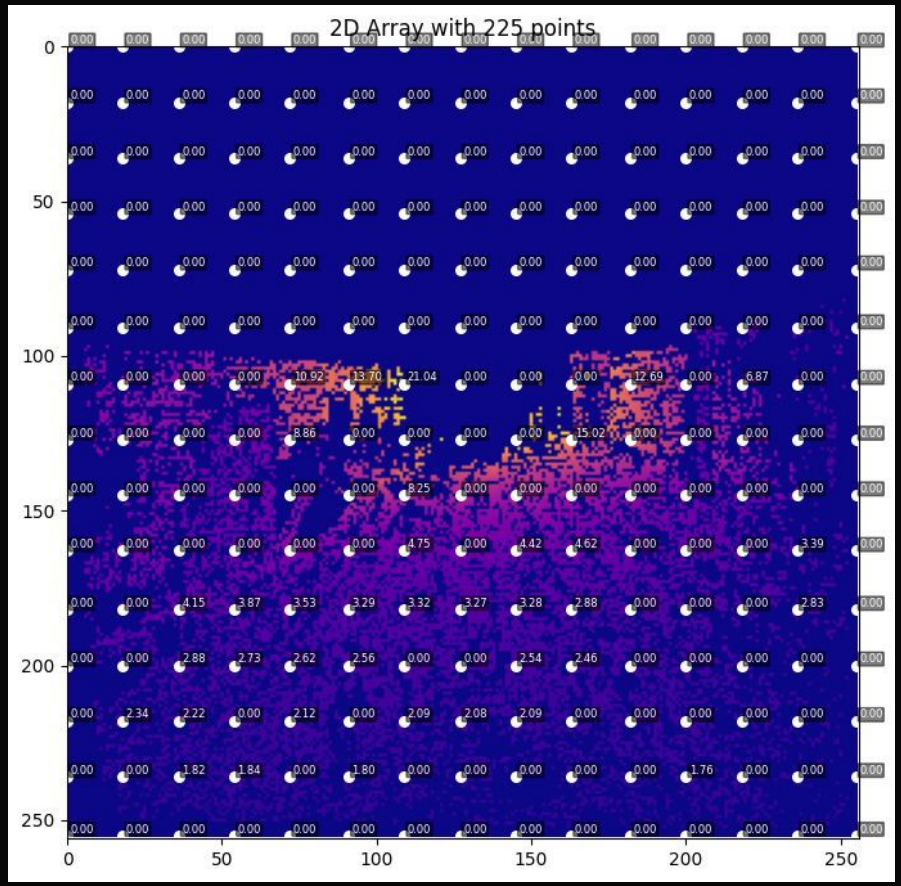
- Using *chrt -f 90*, we launched the application with SCHED_FIFO, a high-priority, non-timesliced real-time policy.

4. Performance Bottleneck Analysis:

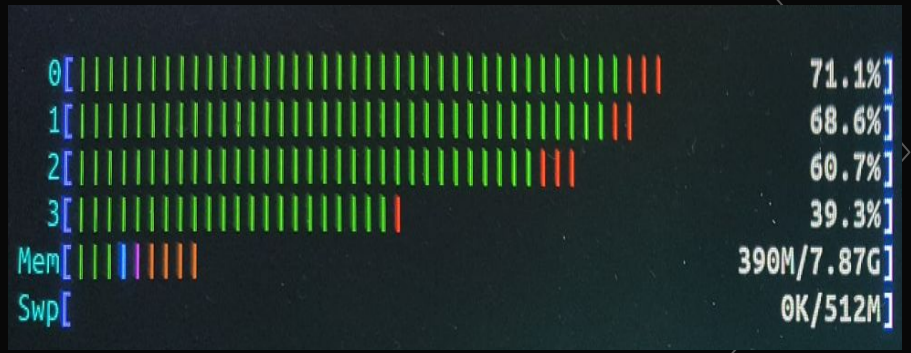
- We identified an initial performance degradation caused by a mismatch between the number of TFLite threads and the number of assigned cores (4).
- **Solution:** By modifying the application to use num_threads=4, we eliminated thread contention and achieved optimal performance for the given core affinity.

Result: A fully optimized execution environment that provides the application with dedicated, high-priority access to CPU resources with minimal jitter.

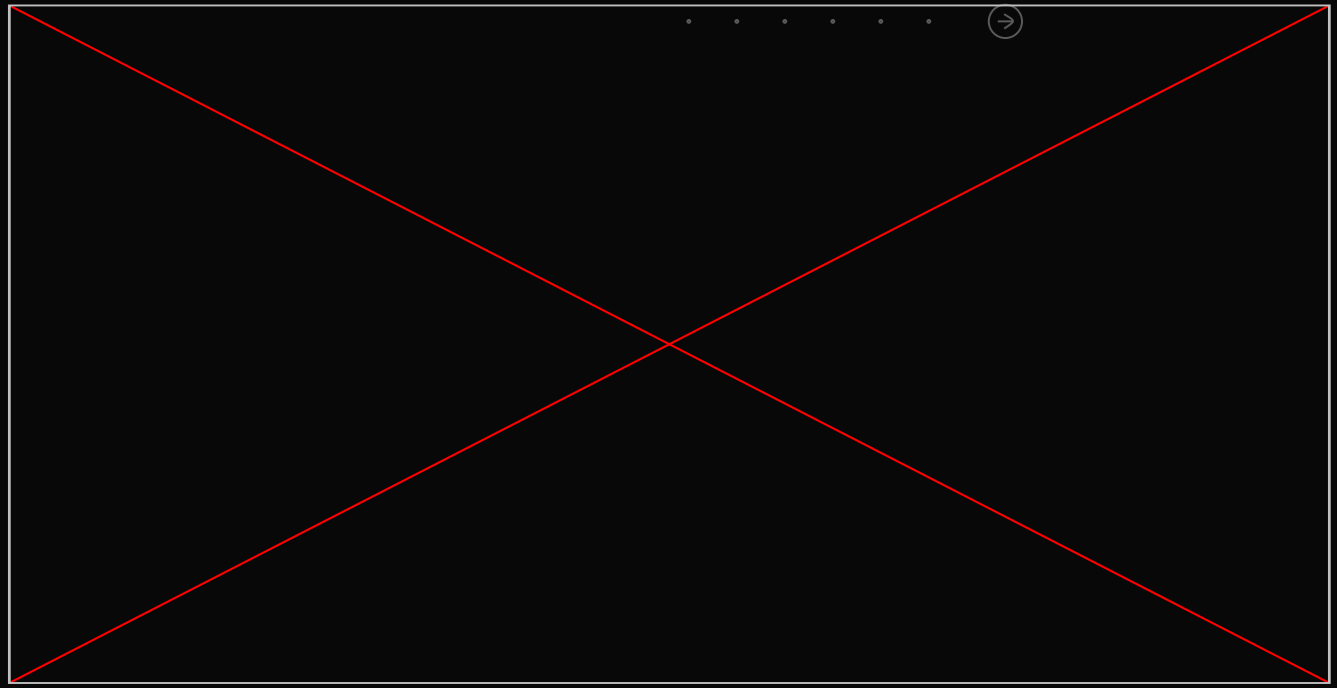
PIPELINE BENCHMARKING & VALIDATION:



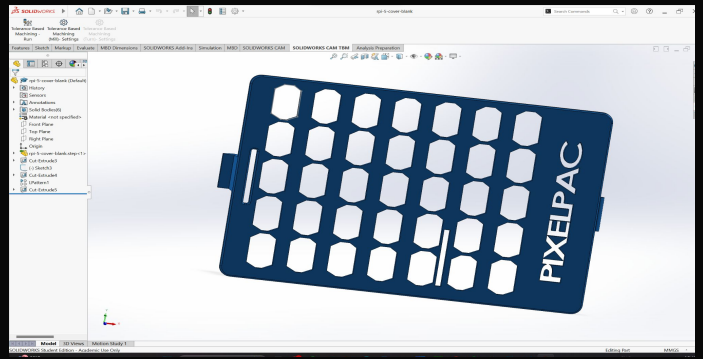
CORE AND MEMORY UTILIZATION

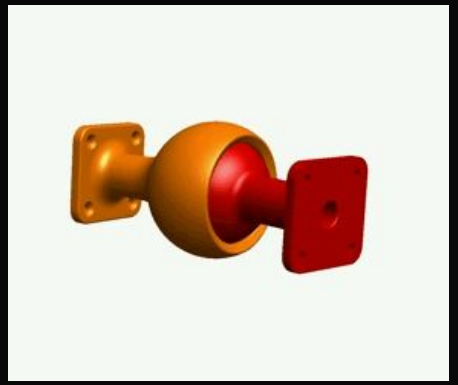


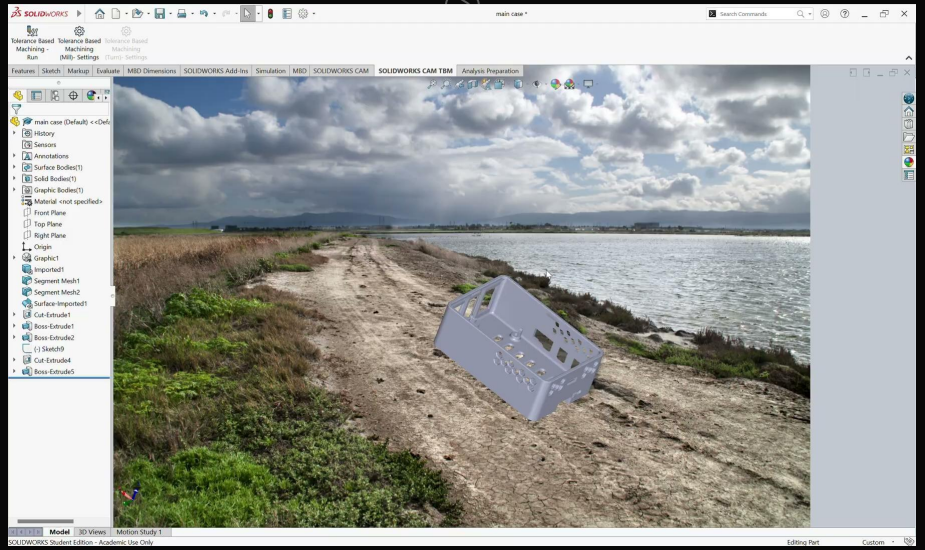
Performance Simulation of Cooling Fans



ON-FIELD DEPLOYMENT 3D-MODELS

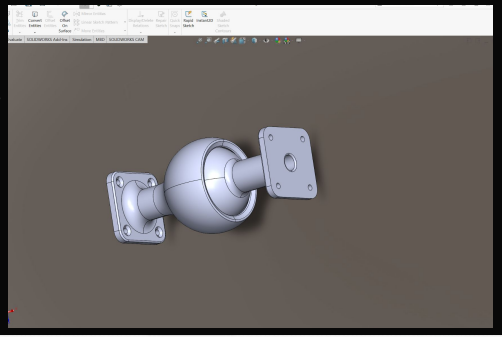






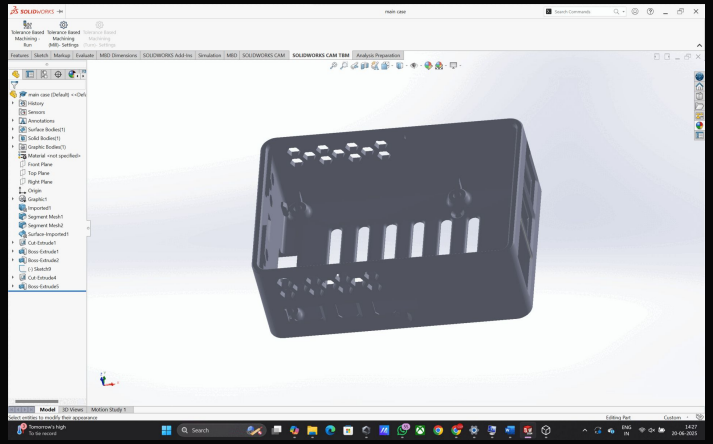
ON-FIELD DEPLOYMENT 3D-MODELS

The entire assembly was optimized for FDM 3D printing, using high-temperature-resistant filaments like
1.PETG
2. ABS



THE DESIGN INCLUDES

- **Mounting brackets** for vibration-dampened installation on vehicle surfaces,
- **A dedicated camera holder** with an adjustable tilt mechanism to fine-tune the visual field of view,
- **Integrated ventilation slots** and **mounting holes** for **dual cooling fans** to ensure thermal efficiency,
- And space for **future expansion**, such as additional sensors or battery modules.



FUTURE IMPROVISATIONS IN DESIGN

- **Material Upgrade**
  We aim to transition from regular 3D-printed plastics to **carbon fiber-reinforced nylon**,

- **Integrated Cable Management**
  The next revision of the design will include internal routing channels for wiring to improve safety,

- **Quick-Release Mounts**
  For field maintenance and modular swapping, we plan to design **quick-release latches** and **snap-fit mounts** for both the main unit and the camera assembly.

- **IP-rated Enclosure Design**
  We're working toward achieving an **IP65 or higher** rating by adding gaskets and sealed entry points—ensuring protection from dust, water, and impact.

- **Active Camera Stabilization**
  As part of long-term R&D, I'm exploring the integration of a **miniature 2-axis gimbal** into the camera holder to compensate for vibration or movement during operation.

- **Digital Twin Simulation**
  Finally, for better predictive maintenance and design validation, we intend to integrate our 3D models into a **digital twin environment**, allowing for simulation of real-world performance under various stress and load conditions.

DEPTH ESTIMATION

Limitations of current approach

1. Limited resources for training ~7000 for adabins distillation and ~15000 for CNN based.
2. Latency for building and evaluation - All the models were trained on cloud.
3. Encoder decoder cannot be deep for feature extraction

Future Exploration

1. Use encoder decoder with lightweight MHSA along with vision model training with different teacher models like monodepth2 and DPT
2. Explore Unsupervised training approach

Final H/W System Specifications & Achievements

- General-purpose Raspberry Pi 5 into a specialized, low-latency real-time platform.
- **Quantitatively Validated Real-Time Performance:** Proved via cyclictest that the system maintains worst-case scheduling latencies under 200µs even under extreme system load.

Architected a Complete, Integrated Application Pipeline: Developed a Python application that handles:

- Live camera input (Picamera 3)
- Direct console framebuffer display for efficient visual feedback
- Real-time hardware alerts (GPIO-driven LEDs and Buzzer)
- CPU-based TFLite model inference

Solved Complex System-Level Challenges: Successfully navigated and resolved issues related to kernel boot failures, driver incompatibilities, and Python environment conflicts.

Created a Robust Foundation for Final Model Deployment: The current platform is stable, validated, and ready to incorporate the final optimized depth estimation model and the hardware-accelerated AI Kit.

WHY DIDN'T WE USE HAILO KIT?

EASY VS EFFICIENT:

PRIORITISED COST EFFECTIVENESS AND GENERATIONAL CROSS COMPATIBILITY OVER IMMEDIATE PERFORMANCE.

STICK TO STOCK STRATEGY

WE DECIDED TO USE THE BASE RASPBERRY PI 5 TO:

- ENSURE **COMPATIBILITY ACROSS MULTIPLE GENERATIONS** AND ITERATIONS OF THE SYSTEM **ACROSS THE DEVELOPMENT CYCLE**.
- ACHIEVE A **STABLE AND CROSS FUNCTIONAL SYSTEM** THAT SERVES AS THE **FOUNDATION FOR ALL FUTURE ADVANCEMENTS**.
- MAKE A **UNIVERSALLY DEPLOYABLE SYSTEM** THAT IS ALSO **COST EFFICIENT**.
- **STRONG MODEL ALREADY:** OUR CURRENT MODEL PERFORMS VERY WELL, MEETING ALL ESSENTIAL REQUIREMENTS.
- **AI KIT'S EDGE:** AI KIT INTEGRATION WILL OFFER EVEN HIGHER FPS, LOWER LATENCY WITH ALL THE BENEFITS OF THE **CUSTOM REAL-TIME SYSTEM.**

CONCLUSION

We have successfully engineered and validated a high-performance, real-time capable platform on a Raspberry Pi 5. The custom PREEMPT_RT kernel provides the deterministic foundation necessary to meet the timing-critical demands of the monocular depth estimation project, ensuring that collision mitigation alerts are both timely and predictable.

IMPROVEMENT PROSPECTS:

1. **Field Testing:** Deploy the prototype in a controlled environment to test its real-world effectiveness.

2. **Integrate Raspberry Pi AI Kit:**Installing the Hailo-8L NPU and its software stack. Offload the TFLite inference from the CPU to the hardware accelerator to achieve a significant increase in FPS and reduce CPU load.

3. **End-to-End Latency Measurement:** With the final model and hardware accelerator in place, perform end-to-end pipeline latency measurements (from frame capture to alert activation).

THANK YOU!