University of Colorado Boulder

Faculty of Electrical, Computer & Energy Engineering

ECEN5763

Written Document: Face-Tracker Group Report

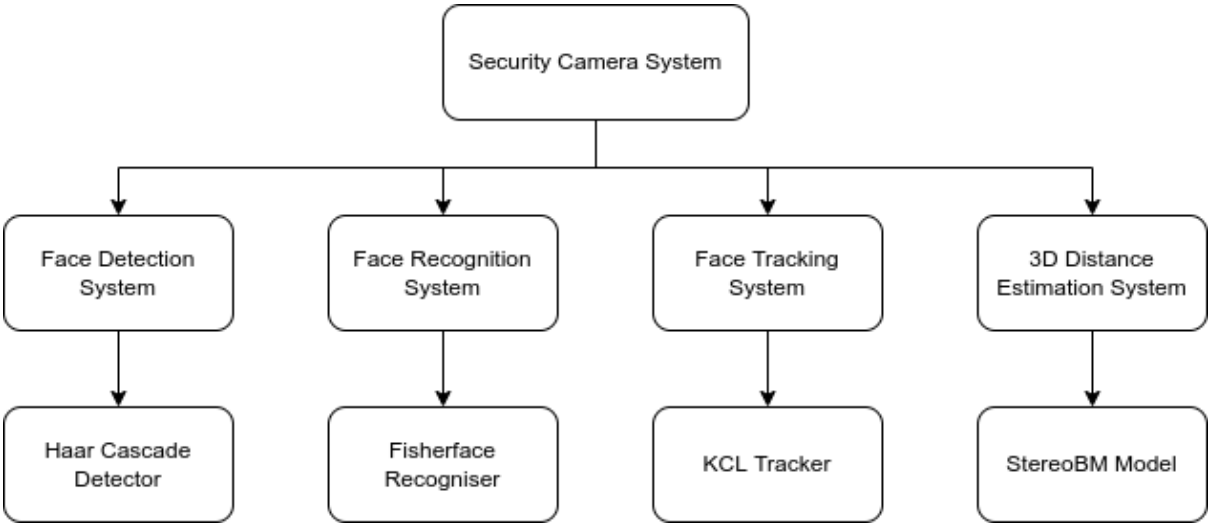
Date	2022/08/07
Prepared by (Name)	Shuran Xu
Prepared by (Name)	Pradyumna Gudluru

Outline

1.Introduction

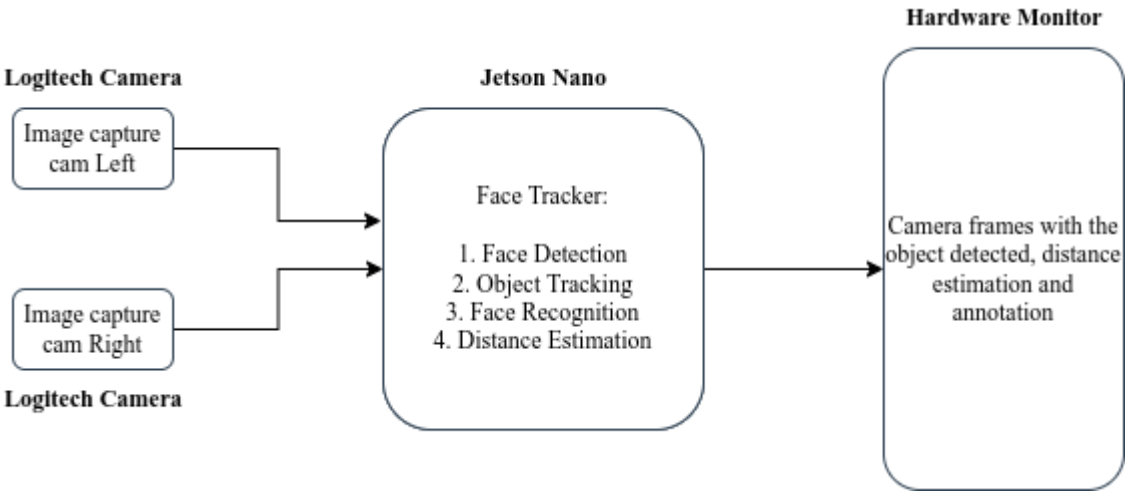
As home security camera systems are excellent tools for home protection and are great applications of computer vision, the team designs and implements a security camera system called “Face Tracker” as the final project. “Face Tracker” detects and recognizes strangers nearby the protected property, tracks the individuals continuously and warns the owner of the property when the stranger is too close to the property.

Structurally speaking, the system is composed of a facial detection subsystem, a facial recognition subsystem, a tracking subsystem and a distance estimation subsystem. Such composition is illustrated by the following block diagram:



All four subsystems are interconnected via data flows. Specifically, the face detection subsystem, which is implemented using the Haar Cascade detector, detects individual faces from camera frames in real-time, and forwards the face data to the face recognition subsystem, which is implemented via the Fisherface recogniser model because of its capability of facial recognition. The recognition system annotates the detected individual and forwards the facial image information to the track subsystem, which is realized by the KCL tracker. The KCL tracker not only tracks the target but also delivers the target information to the distance estimation subsystem so that the StereoBM model can perform the distance estimation while the KCL tracker is tracking the target. Therefore, the program produces a closed-loop data flow system so that any threats to the protected property can be detected and alarmed.

In terms of the system architecture, the team currently uses two cameras arranged in parallel to calculate the depth map and estimate the distance from the target to the camera pair. The video processing is performed on a Jetson Nano board and the processed camera frames are dumped to the attached monitor. The following diagram illustrates such logical architecture:



The following photos show the actual system setup for development:

The chessboard image attached to the wall is used for camera calibration [3] and two identical cameras are connected to the Jetson nano as the system inputs. The monitor is used for output display and program execution.

In terms of the design planning, the team schedules ‘Face Tracker’ design into three stages. And the following table illustrates the general work to be performed in each stage:

Stage(Goal)	**Workload**
Minimum	Camera Input/output operations in opencv C++
	Face Detection via Haar Cascade Detector
	Target Tracking via KCL Tracker
Target	Face Recognition via FisherFace model
	Frame Annotation
	3D Depth Estimation
Optimal	Warning Alarm
	System Performance Acceleration

It is worthy mentioning that performance benchmarking is not included in the above table as the profiling will be performed for all stages. For performance profiling in particular, the team designed a suite of tools for data collection and analysis for the following metrics:

- CPU Load
- Physical Consumed Memory
- GPU Load
- GPU Temperature
- FPS for cameras

Provided ‘Face Tracker’ is successfully prototyped, such a system can be used in authentication of people at home, office, or surveillance areas. The project scope can also be extended in tracking an invasion and warning the owner of the respective authorities.

2.Functional (capability) Requirements

The project contains a set of requirements based on the goal levels and the following table illustrates the functional requirements of the project at each goal level as well as the completion status:

Minimum Goal Requirements		**Status**
1	The face detection and tracking system shall take inputs from two cameras.	Y
2	The system shall detect the objects/persons continuously.	Y
3	The system shall track the object of interest continuously.	Y
4	The average system FPS shall not be below 5 frames/sec.	Y
Target Goal Requirements		
1	The system shall find the distance between the camera pair and the object of interest.	Y
2	The system shall utilize all CPU cores during the execution.	Y
3	The system shall annotate camera frames for facial recognition purposes.	Y
4	The system shall recognize faces for the individual under tracking.	Y
5	The average system FPS shall not be below 10 frames/sec.	Y
Optimal Goal Requirements		

1	The system shall warn the user/owner of the property for the incoming threat if an invader is detected and close to the protected property.	Y
2	The system shall utilize all CPU cores and the GPU core for performance boosting.	Y
3	The average system FPS shall not be below 20 frames/sec.	Y
Common Requirements		
1	The system shall log information including fps for post processing purposes.	Y
2	The system shall smoothly run on the Jetson Nano board.	Y

P.S. “Y” shown in the above table means the particular requirement has been met and ‘N’ means the particular requirement has been failed to meet.

3. Machine Vision and Machine Learning Requirements

“Face Tracker” contains various machine vision requirements that are necessary for development the prototype, and the following table describes each of requirements as follows along with the completion status:

MVML Requirements		**Status**
Face Detection	The system shall detect faces and deliver data to the tracking system.	Y
Target Tracking	The system shall track the target anytime the target is detected.	Y
Distance Estimation	The system shall measure and display the distance from the camera pair to the object on the attached system output, which can be a monitor.	Y
Face Recognition	The system shall recognize faces that have been trained so that only strangers approaching the property can trigger the warning/alarm.	Y

P.S. “Y” shown in the above table means the particular requirement has been met and ‘N’ means the particular requirement has been failed to meet.

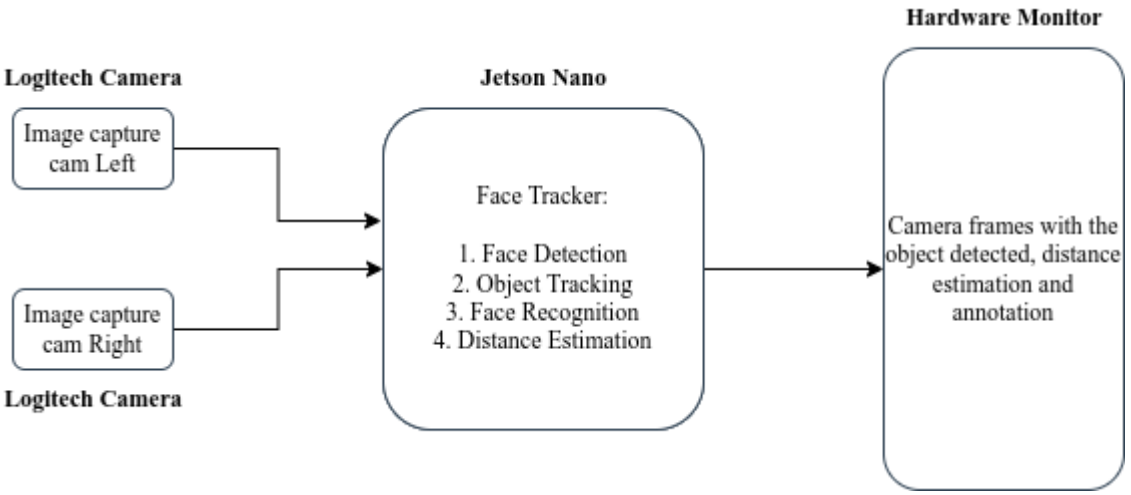
Although all machine vision and machine learning requirements have been met, there are still improvements that can be made in the future for quality advancement, especially for the distance estimation feature.

4.Functional Design Overview and Diagrams

In this section, the system design will be described along with several diagrams for explanation and demonstration.

4.1. System-Level Block Diagram

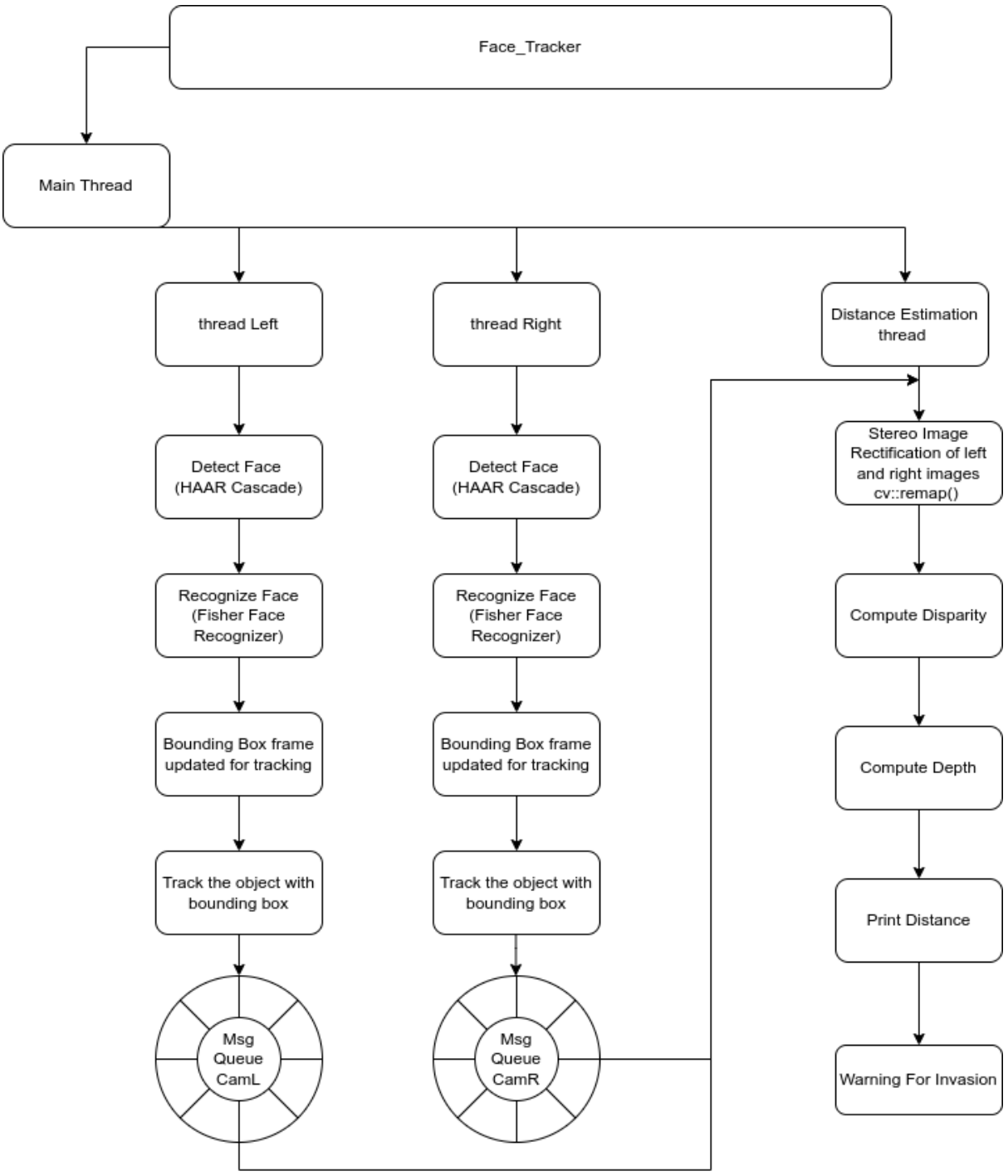
The following diagram shows the overall high-level block diagram of the proposed system.



As can be seen above, ‘Face Tracker’ consists of one camera pair, one Jetson Nano board and a hardware monitor. The camera pair is used for frame capture and the Jetson Nano is employed for frame processing, and the processed output is dumped to the attached monitor for display. The USB3.x bus should be used for the camera connection to the Jetson Nano, but currently only one of the cameras can be connected to the board via USB3.x port due to the hardware limitation imposed by the Jetson board. A USB2.x-to-USB3.x adaptor is needed to resolve this issue in the future.

4.2. Flow Chart

The following diagram shows the data flow of 'Face Tracker' in the course of its execution:

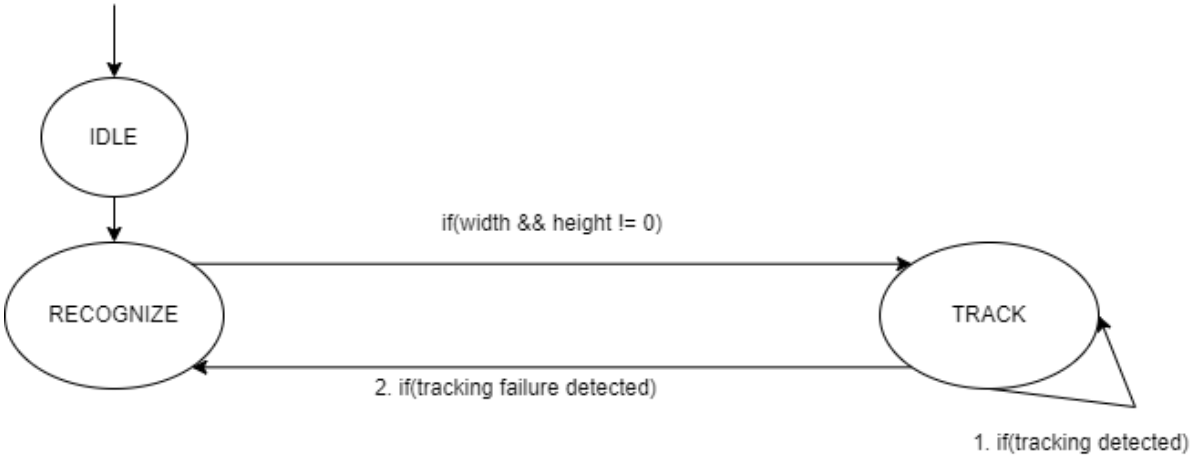


As can be seen above, the main thread creates and initiates the other three threads. The left thread and the right thread essentially go through the same series of operations except that the left thread captures frames from the left camera and the right thread captures frames from the right camera. Whenever camera frames are successfully captured, the Haar Cascade classifier

[2][7] on both threads will be used for face detection, and then the fisherface models will try to recognize the detected faces[8]. The facial recognition is based on the pre-trained data with the database of people to recognise as per the application. Upon the facial recognition, the KCF trackers [9] will update the tracking and push the facial information to the concurrent message queues, which will be pulled by the distance estimation thread, which uses the StereoBM model [11][12] for distance estimation. During the distance estimation process, the images are remapped for stereo image rectification. The disparity is computed using the api call, compute() and the disparity is normalized. By using the formula, depth = M * (1/disparity)[1], the depth is estimated and printed on the annotated frames.

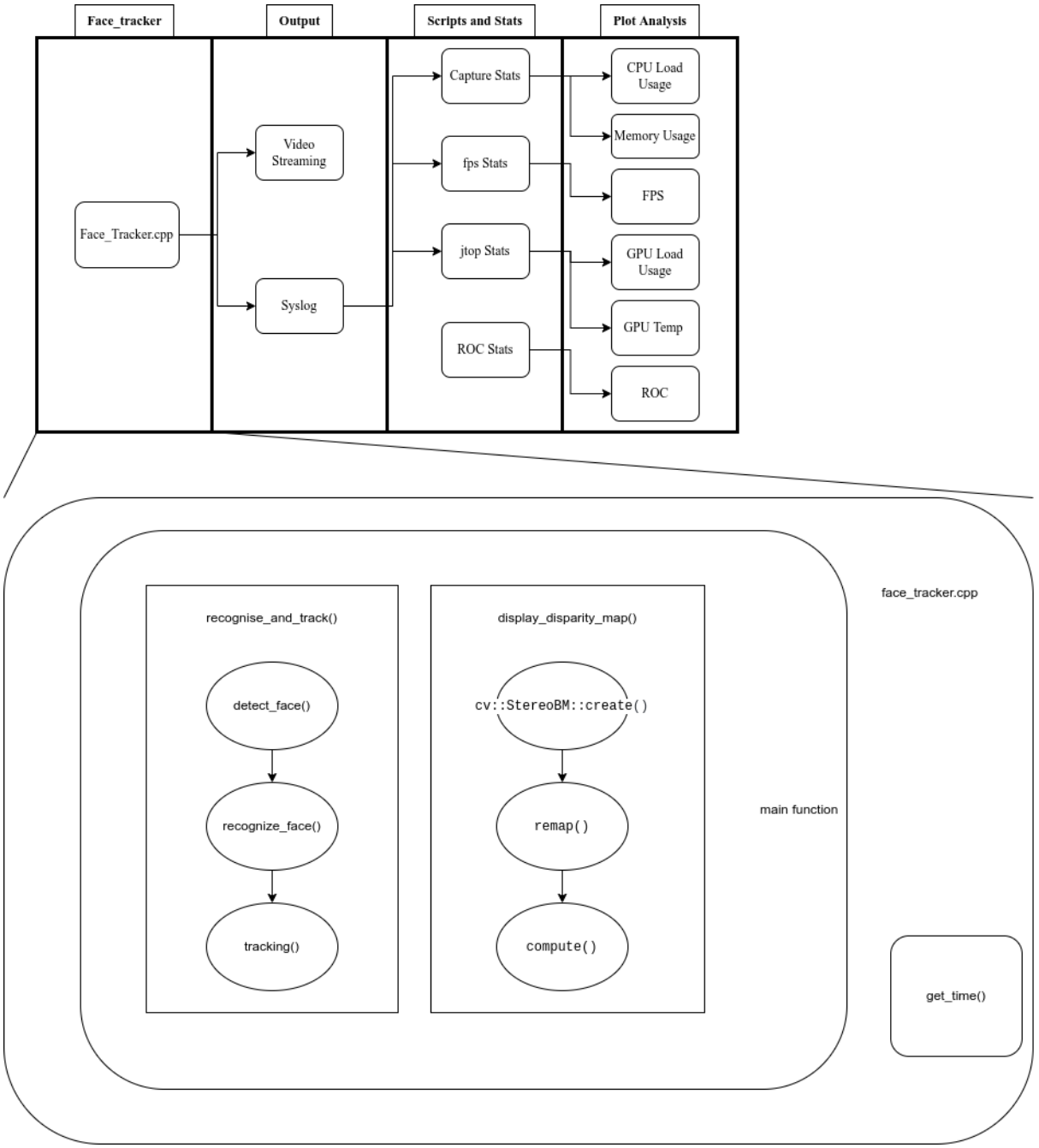
4.3. Finite State Machine Diagram

The entire system is composed of two states: the RECOGNIZE state and the TRACK state. The two threads that handle facial recognition and tracking run separately and go through both the RECOGNIZE state and the TRACK state during their lifetime, but the distance estimation thread only runs in the TRACK state. Specifically, the two threads initially reside in IDLE state but immediately move to the RECOGNIZE state for facial detection and recognition. Once a face is recognized, the thread is advanced to the TRACK state for the target tracking. The distance estimation is only possible to be performed when the both facial recognition threads are in the TRACK state.



4.4 Composite diagram

The development work of the project not only involves the design and the implementation of each required subsystem but also the performance profiling and benchmark. The following diagram shows the components the team constructed for both the development and the profiling purpose.



The ‘Scripts and Stats’ section above lists the utility tools that the team developed for metric measurement, and the following table describes them in detail:

Utility Tool	Description
capture_stats.sh	Collects the CPU load and the actual consumed physical memory for the given process.
gpu_logger.py	Logs the GPU core load and the GPU temperature during the program execution period.
roc_curve.py [4]	Performs the train-test data split, train the fisherface model and test the fisherface model. The test results are then used for the ROC curve generation as well as the associated AUC score.
extract_syslog_for_fps.sh	Extracts the FPS values for both cameras from the syslog.
data_analysis.ipynb	Performs the statistical summary analysis for FPS, GPU load, CPU load and Memory usage.

5. ACV Analysis and Design

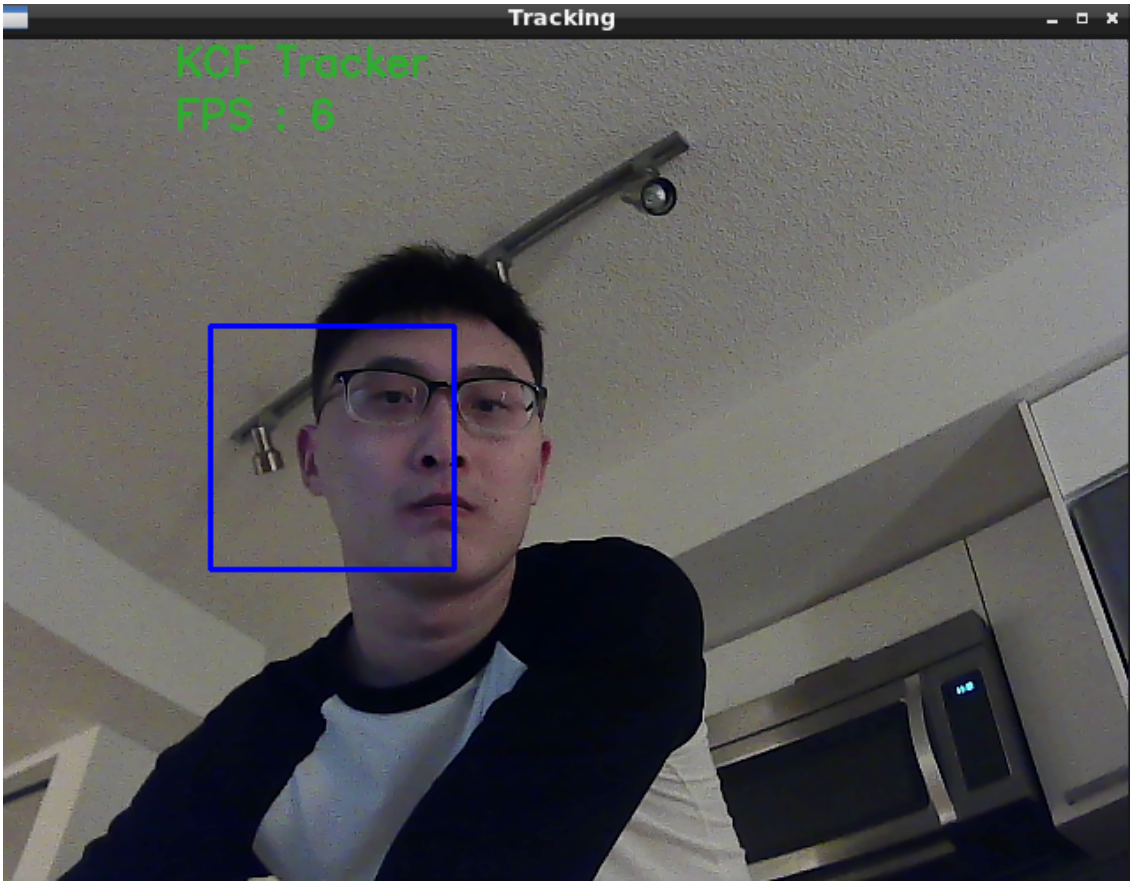
The team performed unit testing on important subsystems and conducted performance analysis of the program over a range of metrics. This section discusses both in detail.

5.1 Unit Testing

Since the system is composed of several subsystems, the team performed unit testing on each subsystem. Specifically, the video frame capture, facial detection and tracking are tested by a simple face tracker program that uses the KCF tracker and the Haar cascade classifier with the haarcascade_frontalface_default.xml [6] as the model parameter file. The following screenshots show the case when the camera fails to detect a person and the case that the same person is successfully detected:

Case where the person is not detected

Case where the person is detected



Additionally, the facial recognition was tested with the following steps:

No	Steps Taken
1	Construct a custom training database that is based on the Yale Face database.
2	Construct a custom testing database that includes both detectable individuals and strangers.
3	Train the fisherface model.
4	Test the fisherface model with the testing database.
5	Print the test result summary.
6	Draw the ROC with the AUC value.

The above five steps are implemented in C++ and the resulting C++ program is run on the Jetson Nano. The last step is implemented in Python due to the high complexity of ROC implementation in C++.

The following screenshot shows the partial contents of the training data file and the testing data file. The files are in .csv format and contain the absolute path of each image for training/testing as well as the associated labels:

```
shuran@shuran-desktop:~/face-model$ cat yalefaces.csv | head -n 5
/home/shuran/face-model/yalefaces/subject18/subject18.sad.pgm,18
/home/shuran/face-model/yalefaces/subject18/subject18.wink.pgm,18
/home/shuran/face-model/yalefaces/subject18/subject18.sleep.pgm,18
/home/shuran/face-model/yalefaces/subject18/subject18.surprise.pgm,18
/home/shuran/face-model/yalefaces/subject18/subject18.happy.pgm,18
shuran@shuran-desktop:~/face-model$ cat yalefaces.csv | wc -l
159
shuran@shuran-desktop:~/face-model$ cat testfaces.csv
/home/shuran/face-model/testfaces/subject33.normal.pgm,33
/home/shuran/face-model/testfaces/subject14.sad.pgm,14
/home/shuran/face-model/testfaces/subject08.noglasses.pgm,8
/home/shuran/face-model/testfaces/subject15.happy.pgm,15
/home/shuran/face-model/testfaces/subject17.happy.pgm,17
/home/shuran/face-model/testfaces/subject16.happy.pgm,16
/home/shuran/face-model/testfaces/subject03.sad.pgm,3
/home/shuran/face-model/testfaces/subject19.normal.pgm,19
/home/shuran/face-model/testfaces/subject08.normal.pgm,8
/home/shuran/face-model/testfaces/subject30.happy.pgm,30
/home/shuran/face-model/testfaces/subject01.sleepy.pgm,1
/home/shuran/face-model/testfaces/subject21.sleep.pgm,21
/home/shuran/face-model/testfaces/subject31.sad.pgm,31
/home/shuran/face-model/testfaces/subject20.sad.pgm,20
/home/shuran/face-model/testfaces/subject11.happy.pgm,11
/home/shuran/face-model/testfaces/subject32.normal.pgm,32
/home/shuran/face-model/testfaces/subject18.normal.pgm,18
/home/shuran/face-model/testfaces/subject30.normal.pgm,30
shuran@shuran-desktop:~/face-model$ cat testfaces.csv | wc -l
18
```

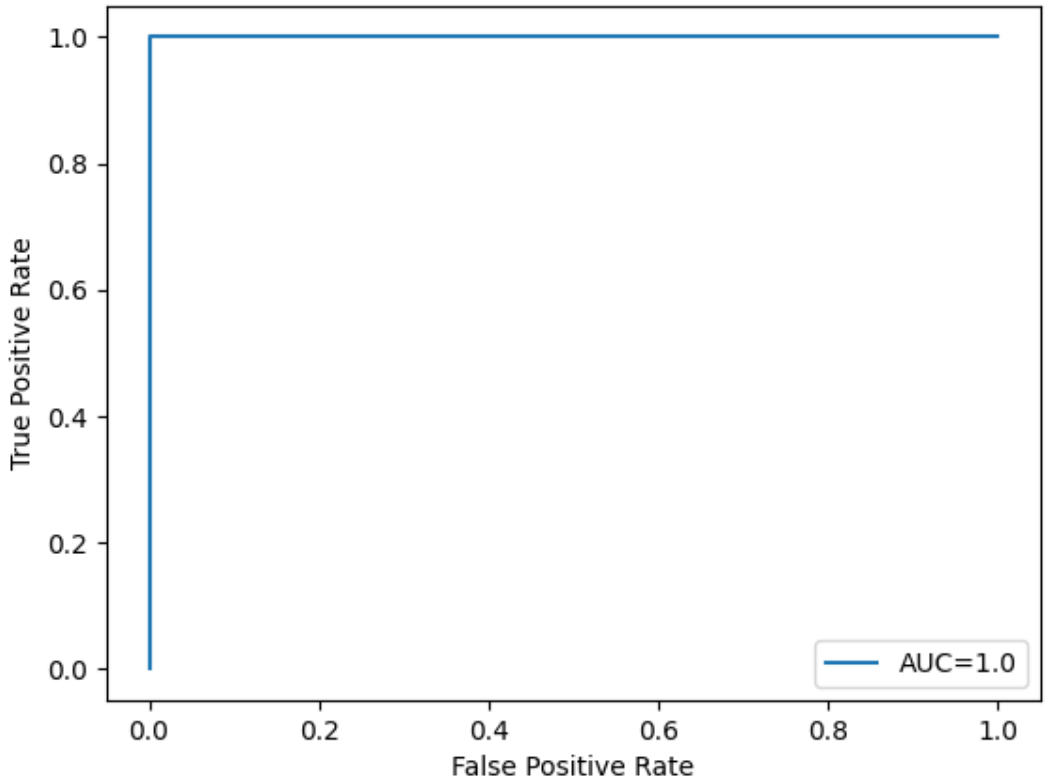
The ‘wc -l’ command shows the number of images used for training/testing. It is obvious to see that 159 images are used for training and 18 images are used for testing. All images are in .pgm format with the same resolution.

The following screenshot shows the testing result summary:

```
shuran@shuran-desktop:~/face-model/fisherface/build$ ./fisherface --train=/home/shuran/face-model/ya
lefaces.csv --test=/home/shuran/face-model/testfaces.csv
Successful Predicted Images = 13 / Total Test Images = 18.
shuran@shuran-desktop:~/face-model/fisherface/build$
```

The above result basically means that the model successfully recognized all recognizable faces and failed the rest which are not supposed to be recognized.

Additionally, the ROC curve for the fisherface model is shown as follows:



It is clear to see that the model is too perfect and this could be due to the overfitting issue. Currently the model is performing relatively well but further improvement is needed in the near future to resolve the overfitting issue.

5.2 Performance analysis

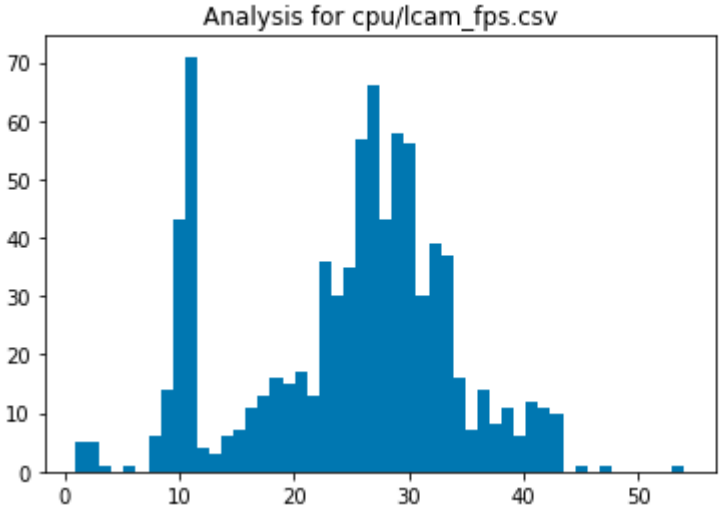
To perform the performance profiling, the team developed a suite of tools to collect data including FPS and CPU load. The following table demonstrates such development work:

Utility Tool	Description
capture_stats.sh	Collects the CPU load and the actual consumed physical memory for the given process.
gpu_logger.py	Logs the GPU core load and the GPU temperature during the program execution period.
jtop_stats.py [5]	Periodically updates the GPU related statistical information. The program works as the top command.
roc_curve.py	Performs the train-test data split, train the fisherface model and test the fisherface model. The test results are then used for the ROC curve generation as well as the associated AUC score.
extract_syslog_for_fps.sh	Extracts the FPS values for both cameras from the syslog.
data_analysis.ipynb	Performs the statistical summary analysis for FPS, GPU load, CPU load and Memory usage.

The aforementioned tools are used to collect the performance metrics while the program is running. The scripts make use of syslog and various Linux commands such as top and jtop for data collection. The collected metric data is then post-processed by data_analysis.ipynb for the statistical summary analysis. To quantify the performance gain from the GPU optimization, the team profiled the runtime performance of both the target-level program, which is executed fully on CPU cores and the optimal-level program, which runs on both CPU cores and GPU cores. The post-processing data analysis showed that the optimal-level program outperformed the CPU-version in terms of the distance estimation and the CPU usage reduction.

5.2.1 Performance Evaluation of the target-level program

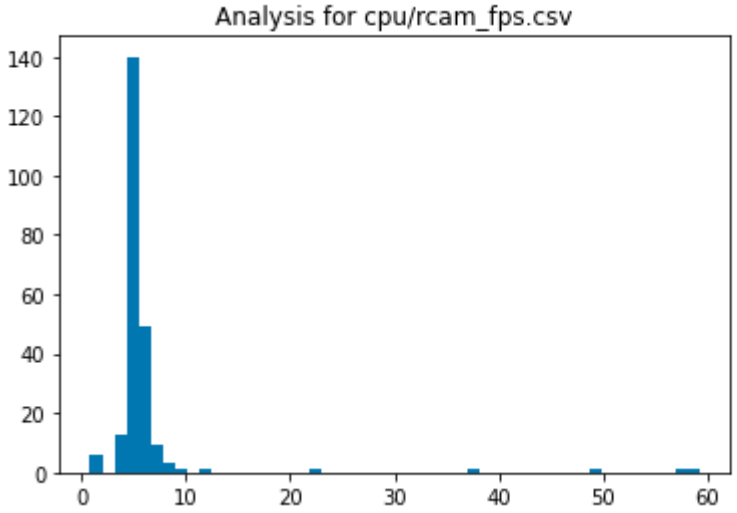
The following diagram shows the FPS rate histogram and the associated summary statistics of the left camera:



count	836.000000
mean	24.809844
std	9.181253
min	0.900000
25%	19.012500
50%	26.560000
75%	30.602500
max	54.040000
Name:	fps,
dtype:	float64

As can be seen above, the average FPS rate on the left camera is 24.8 and the maximum rate is 54. Although the FPS rate is above 26 most of the time, the rate varies widely. Such large standard deviation implies the drastic changes of the runtime overhead of program's video processing.

The following diagram shows the FPS rate histogram and the associated summary statistics of the right camera:



count	227.000000
mean	6.054626
std	6.327606
min	0.810000
25%	4.760000
50%	5.090000
75%	5.690000
max	59.200000
Name:	fps,
dtype:	float64

It is obvious to see that the FPS rate on the right camera also varies dramatically, ranging from 1 to 59. One can notice that the average rate on the right camera is only 6, and this is due to the USB bus connection. Since the Jetson Nano only provides one USB-3 port but two USB 2.x ports, only one of the cameras can be connected to the board via USB-3 and the

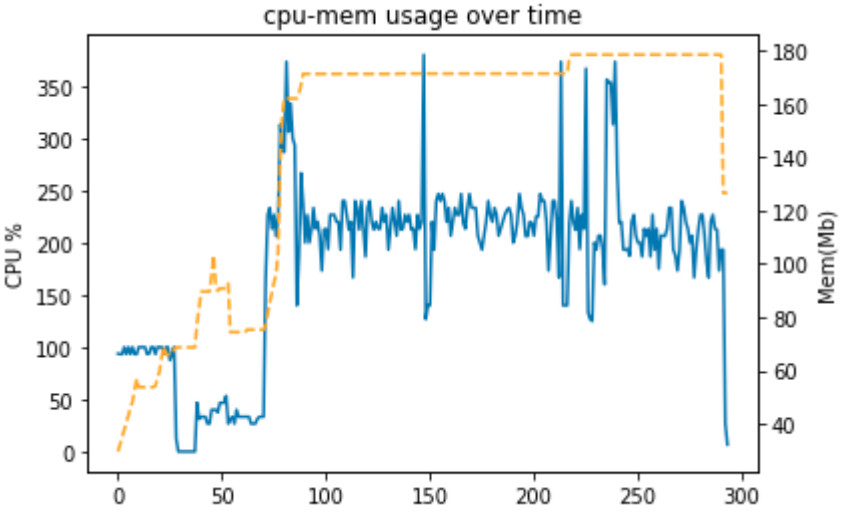
team observed that the camera connected via USB-3 always runs considerably faster than its peers. The following screenshot shows such connection setup on Jetson nano:

```
shuran@shuran-desktop:~$ v4l2-ctl --list-devices
USB 2.0 Camera (usb-70090000.xusb-2):
        /dev/video0

USB Camera (usb-70090000.xusb-3.2):
        /dev/video1
```

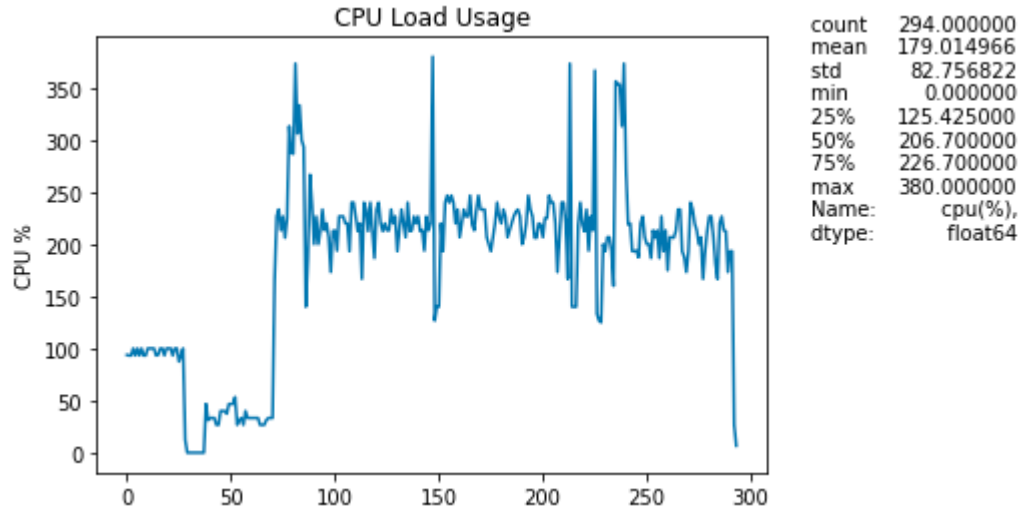
As USB-3 is 10 times faster than USB 2.0, the left camera is therefore running at a much higher rate than the right camera.[17]

The following composite diagram shows the variation of both the CPU load and the memory usage in the course of the program execution:



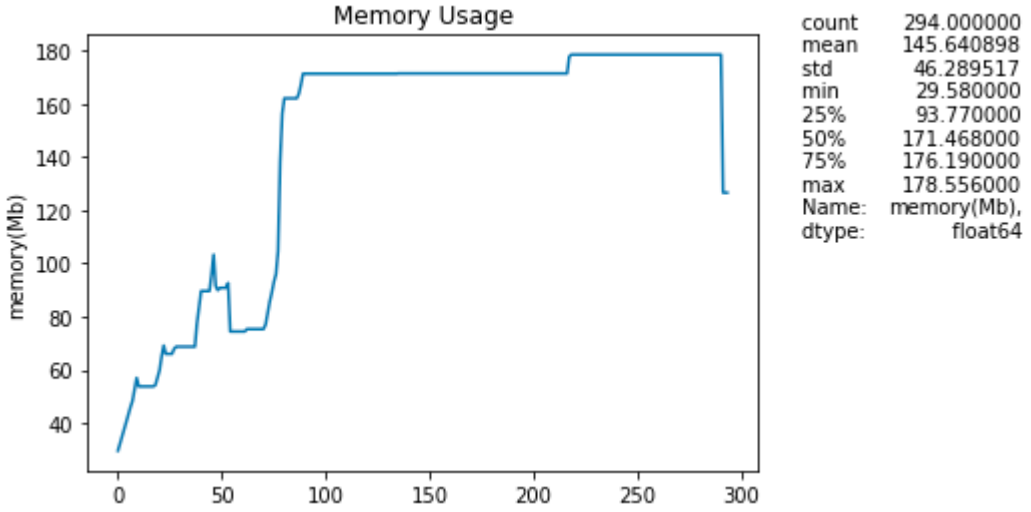
It can be seen that the program occupies 2 CPU cores mostly and consumes roughly 178 MB most of the time. The detailed discussion is shown below.

The following diagram shows the CPU load histogram and the associated summary statistics:



It can be seen that the program primarily makes use of two CPU cores despite the fact the program is designed to fully utilize four CPU cores. However, such a statistical summary is reasonable since the main thread sleeps upon launching two facial recognition threads and one distance estimation thread. And during the time where the program tracks the target individual, only the distance estimation thread is heavily performing the computation and two facial detection threads simply update tracking status and text displays. Therefore, it makes sense to see approximately only two CPU cores are actively involved in the course of the program execution.

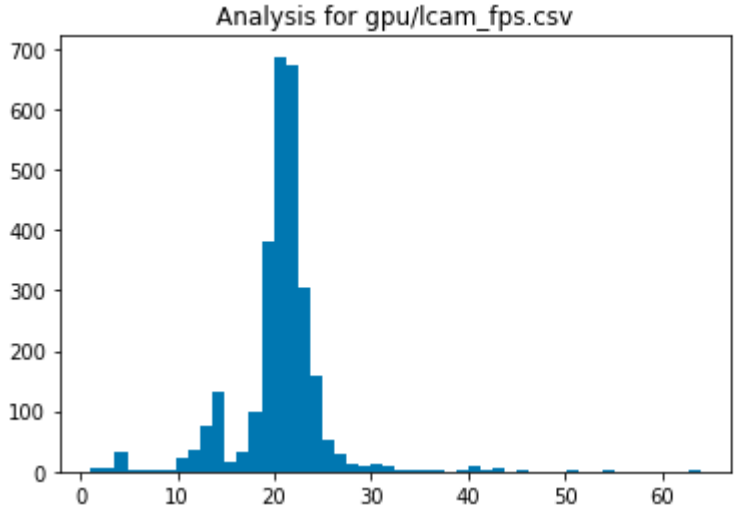
The following diagram shows the memory usage histogram and the associated summary statistics:



It is clear to see that the program consumes about 176 MB of physical memory most of the time during its execution. The high jump shown from the diagram from 80 MB to 170 MB indicates the launch of facial recognition threads and the distance estimation thread.

5.2.2 Performance Evaluation of the optimal-level program

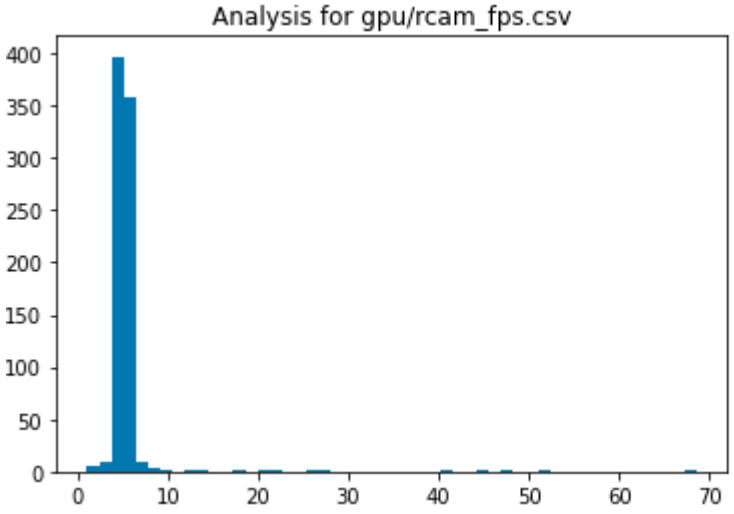
The following diagram shows the FPS rate histogram and the associated summary statistics of the left camera:



count	2827.000000
mean	20.512037
std	4.489972
min	1.070000
25%	19.650000
50%	20.890000
75%	22.215000
max	63.900000
Name:	fps,
dtype:	float64

As can be seen above, the average FPS rate on the left camera is approximately 20 and the maximum rate is 64. It is noticeable that the FPS distribution tends to form a normal distribution with little standard deviation. Such little fluctuation implies the program is executing more reliably in a consistent manner, compared to the target version.

The following diagram shows the FPS rate histogram and the associated summary statistics of the right camera:



count	797.000000
mean	5.492033
std	4.018320
min	1.040000
25%	4.980000
50%	5.090000
75%	5.230000
max	68.560000
Name:	fps,
dtype:	float64

Similarly, the right camera has a much lower FPS rate than the left camera due to the USB connection issue. The average FPS rate is about 5.5, which is only slightly lower than that of the target version.

It could be a confusing observation as the FPS rates from both cameras in the optimal-versioned program are lower than the rates found in the target-versioned program. However, based on the team's analysis of the CUDA implementation, such observation could be due to the following overhead introduced by the CUDA implementation of the facial detection function:

1). I/O transfer overhead: data uploading and downloading

For each call of the CUDA-versioned facial detection, uploading of matrices and downloading of matrices inevitably introduce overhead.

2). Serialization of GPU usage from two threads

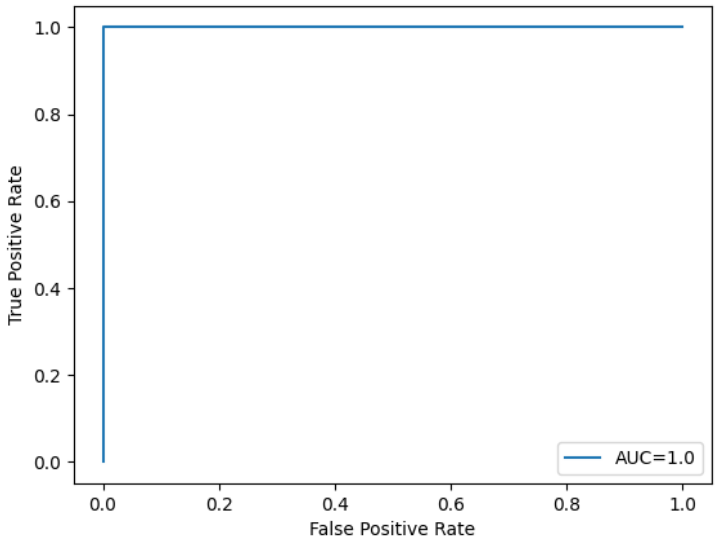
The two threads have to share the single GPU device with each other. Although the NVIDIA GPU device consists of hundreds of GPU cores and CUDA programming allows for the parallel computation via asynchronous streaming, the OpenCV::cuda framework offered from the opencv-contrib-4.x only supports single-stream processing for certain operations such as multi-scale detection. Therefore, the facial detection work inevitably has to be serialized and this introduces the blocking overhead.

3). Insufficient amount of data for GPU processing

Leveraging the power of NVIDIA GPU certainly produces the performance gain when running computationally intensive applications. However, when the data being processed is relatively small, such performance gain might not be obvious. In the case of the optimal-versioned program, the GPU only processes very little data: only two cropped images of resolution 640x480 every time. Therefore, the program might not benefit from offloading work to the GPU.

5.3. Segmentation, detection, tracking, and/or classification/recognition performance

The performance analysis of the fisherface model is evaluated as a ROC curve with the AUC value calculated. The following diagram shows the ROC curve with the associated AUC value:



There is no denying that the ROC curve looks too perfect and the model might be facing the overfitting issue. However, from the experimentation and observation, the trained model meets the team's expectation: the trained fisherface model always successfully recognized faces that were trained before and failed to recognize stranger faces, which are faces that were never trained. Hence, the ROC curve is currently aligned with the team's expectation, but more work should be done as the future improvement to remove the potential overfitting issue.

6. Challenges

In the course of program design and development, the team experienced numerous challenges. In this section, several key challenges will be discussed in detail.

Terminology: The two threads dedicated for the facial recognition and tracking will be labelled as T1 and T2, and the thread for the distance estimation will be labelled as T3.

6.1 Data Communication among threads

Challenge Statement
The program is designed to use T1 to handle frames from the left camera and T2 to handle frames from the right camera. In order to perform the real-time distance estimation, T3 should estimate the distance of the target individual to the camera pair from both the left camera frame and the right camera frame consistently and continuously. To do that, those two threads have to supply facial information to the third thread without any delays that can be introduced by data race. In addition, T3 also has to supply the distance value back to T1 and T2 so that they can display such information into their frames respectively.
Solution

The team designed a concurrent queue structure that utilizes the mutex, condition variable and C++11 synchronization primitives such as unique_lock and lock_guard as a vector for matrix delivery from T1/T2 to T3. And to avoid data race, the std::map structure is used to ensure that each thread has its own queue. The following declaration presents such idea:
`std::map<std::string,concurrent_queue_cv<cv::Mat>> mat_map;`
Here, std::string indicates the thread name and the concurrent_queue<cv::Mat> indicates the queue for matrix delivery from T1/T2 to T3.
Inside the concurrent_queue definition, the condition variable is used in the pop and push operations so that T3 will not proceed until it receives data from both T1 and T2. Such producer-consumer scheme ensures that T3 execution is strictly following the T1 and T2 execution. In case either T1 or T2 fails to detect the target or fails to track the target T3 will not run but wait for data arrival.
Such structure is also used in defining a data structure to deliver distance value from T3 to T1/T2:
`std::map<std::string,concurrent_queue_cv<std::string>> dist_map;`

6.2 Facial Recognition thread synchronization

Challenge Statement
Since T1 and T2 perform the facial recognition and target tracking tasks separately, there are chances that they fail to recognize the same person at the same time, and this can lead to both bad user experience and extra overhead for T3 to wait for the data arrival from the late submission of facial data matrix. The assumption made is that both threads successfully detect and recognize the same person from the frames.
Solution
The team designed a concurrent barrier that forces T1 and T2 to reach a common point before proceeding. Such a common point is placed after the facial recognition task so that T1 and T2 can proceed with the target tracking at the same time. Inside the concurrent barrier definition, the condition variable is used to force either T1 or T2 calling wait( ) to block until both T1 and T2 have called wait( ) to release them at once.

6.3 Tracking Failure Handling and Facial Recognition Misalignment Handling

Challenge Statement
In order to achieve real-time video stream analysis, T1 and T2 were designed to perform the facial recognition and target tracking tasks independently so that no communication or synchronization overhead would occur. However, the team found that there are following cases that cause the program fail to proceed: ● T1 or T2 fails to detect a person ● T1 or T2 fail to recognize a person ● T1 and T2 recognize different persons
Solution
The program checks the facial recognition labels as well as the tracking status for both T1 and T2 when the program is running in the tracking state. In case the labels are mismatched or the tracking fails on either one of these two threads, then the program will reset the states for T1 and T2 to force them to repeat the facial recognition tasks.

6.4 GPU Optimization

The team encountered a series of troubles when working on the GPU optimization during the development, and three key challenges are covered in this subsection.

6.4.1 Framework Setup

Challenge Statement
The released Opencv-Contrib-4.x does not support CUDA implementation of the facial detection functionality. Such a limitation restricted the usage of the NVIDIA GPU to the optimization work.
Solution
The team found a custom branch of Opencv-Contrib-4.x called “2048_cuda_cascade_no_longer_available” so the team decided to reinstall OpenCV4.1.1 with this branch of OpenCV-Contrib-4.x. But soon the team found that the associated Python3 installation failed due to the lack of SIFT support in Python3 from OpenCV4.1.1. Such lack of support is due to the patent expiration issue. To solve this issue, the team ended up with installing Opencv 4.5 with this custom branch of Opencv-Contrib-4.x.[15][16]

6.4.2 Facial Detection Optimization

Challenge Statement
Since both T1 and T2 are leveraging GPU to optimize the facial detection, the way to share the GPU with these threads determines the performance gain from the GPU utilization. If the team forces T1 and T2 to serialize the access to the GPU then the synchronization overhead undoubtedly cancels out the performance gain from the GPU processing.
Solution
The team managed to parallelize most of the operations involved in the facial detection by uploading data and executing operations in different GPU streams so that no data overlap or data race would occur. However, due to the implementation of the cuda::cascadeclassifer.cpp, the critical multiscale detection work has to be performed on the GPU stream0, which is the default GPU stream. Such restriction forced T1 and T2 to serialize the access to the GPU via std::lock_guard. However, the partial parallel processing was realized in the GPU-optimized implementation and therefore the CUDA implementation of facial detection produced positive performance gain.

6.5 Distance Estimation

Challenge Statement
The team used the StereoBM model and implemented the distance estimation steps according to the following standard equation below: `distance = (baseline * focal_length) / disparity` However, the team found that the implementation always produced 0 values, regardless of the StereoBM parameter settings.
Solution
Currently, the issue has not been fully resolved yet as the CPU implementation still produced zero values all the time, but the CUDA implementation produced non-zero values. The CUDA implementation has the exact same steps as the CPU implementation although the results of the two are drastically different. The resolution of this issue will be part of the future improvement for this program.

6.6 Different FPS values for two identical cameras

Challenge Statement
The FPS rates for the left camera and the right camera are different although both T1 and T2 are executing the same function, meaning the same flow path. Such an issue occurs in both the CPU implementation and the CUDA implementation.
Solution
The team found that the issue was due to the different USB ports used to the left camera and the right camera. Since the Jetson Nano only has one USB3.x port and two USB2.x, inevitably one camera has to be connected to the Jetson via USB2.x. It is known that the USB3.x is at least 10 times faster than USB2.x and such speed difference introduced FPS differences. The team tried to switch the cameras with USB ports and the observation was aligned with the hypothesis: the camera connected via USB3.x always produced a much higher FPS rate. Currently, the team decided to leave the system setup as what it is right now, because purchasing a USB2.x-to-USB3.x adaptor takes at least a day and the team might not be able to show the result during the demo.

7. Incomplete Work/Future Improvement

In general, the team has accomplished all functionality required for the project, but several places can be further improved. Firstly, more time and effort should be allocated to resolve the distance estimation issue so that the CPU implementation of the distance estimation function can produce time-varying non-zero values and the CUDA implementation can produce much more accurate values. Additionally, a USB2.x-to-USB3.x adaptor should be equipped so that both cameras can connect to the Jetson via USB3.x port, resolving the FPS rate difference issue. Furthermore, the fisherface model performance evaluation is needed to be done extensively to resolve the potential overfitting issue. Lastly, different Opencv versions can be tried to explore the possibility of more CUDA options from other contrib modules.

8. Proof Of Concept

The team shot three video recordings that represent the minimum goal, the target goal and the optimal goal. All recordings will be submitted along with the report. Specifically, the video recording for the minimum goal demonstrates the facial detection and the target tracking on one camera;the recording for the target goal demonstrates the facial recognition, the target tracking and the distance estimation by utilizing all CPU cores; the recording for the optimal goal demonstrates the same operations as the target goal in addition to the distance warning annotation with the program executing on all CPU cores and the GP-GPU.

9. Conclusion

As considered the status of the project, the minimal and target goals are reached with respect to the requirements for the application of security cameras. Specifically, the object is detected and tracked in real-time. In addition, facial recognition is also successful in the sense that the individual can be correctly recognized most of the time. The disparity map and distance estimation is calculated but the quality is not as expected, so more development work is needed for quality improvement. In addition, the optimization work including GPU acceleration via CUDA will be carried out as the next major development work.

The major takeaway from this project is design and implementation of the heterogeneous parallel computing using GP-GPU and CPU combo as well as exercising computer vision techniques including facial recognition and object tracking algorithms, etc. In addition, performance profiling and analysis over a wide range of metrics including GPU load and memory size over the period of the program execution is also practiced.

10. References

1. “Depth Estimation using Stereo Camera and OpenCV (Python/C++).” LearnOpenCV – OpenCV, PyTorch, Keras, Tensorflow examples and tutorials https://learnopencv.com/depth-perception-using-stereo-camera-python-c/#from-disparity-map-to-depth-map
2. “OpenCV: Cascade Classifier.” Docs.opencv.org https://docs.opencv.org/4.1.1/db/d28/tutorial_cascade_classifier.html
3. “Camera Calibration using OpenCV | LearnOpenCV #.” LearnOpenCV – OpenCV, PyTorch, Keras, Tensorflow examples and tutorials. https://learnopencv.com/camera-calibration-using-opencv/
4. View Zach. “How to Plot a ROC Curve in Python (Step-by-Step) - Statology.” Statology https://www.statology.org/plot-roc-curve-python/
5. “jtop package — jetson-stats 3.1.2 documentation.” Rnext.it https://rnext.it/jetson_stats/jtop.html
6. “opencv/haarcascade_frontalface_alt.xml at master · opencv/opencv.” Github https://github.com/opencv/opencv/blob/master/data/haarcascades_cuda/haarcascade_frontalface_alt.xml
7. “OpenCV: Cascade Classifier.” Docs.opencv.org https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html
8. “OpenCV: Face Recognition with OpenCV.” Docs.opencv.org https://docs.opencv.org/3.4/da/d60/tutorial_face_main.html
9. “OpenCV: Introduction to OpenCV Tracker.” Docs.opencv.org https://docs.opencv.org/4.x/d2/d0a/tutorial_introduction_to_tracker.html
10. “C++11 Multithreading – Part 1 : Three Different ways to Create Threads – thisPointer.” Thispointer.com https://thispointer.com/c-11-multithreading-part-1-three-different-ways-to-create-threads/
11. “cv2.StereoBM | LearnOpenCV.” LearnOpenCV – OpenCV, PyTorch, Keras, Tensorflow examples and tutorials https://learnopencv.com/tag/cv2-stereobm/
12. “OpenCV: Depth Map from Stereo Images.” Docs.opencv.org https://docs.opencv.org/3.4/dd/d53/tutorial_py_depthmap.html
13. Meel Vidushi. “Object Tracking in Computer Vision (Complete Guide) - viso.ai.” viso.ai https://viso.ai/deep-learning/object-tracking/
14. “Object Tracking using OpenCV (C++/Python).” LearnOpenCV – OpenCV, PyTorch, Keras, Tensorflow examples and tutorials https://learnopencv.com/object-tracking-using-opencv-cpp-python/
15. “GitHub - nosajthenitram/opencv_contrib at 2048_cuda_cascade_no_longer_available.” GitHub https://github.com/nosajthenitram/opencv_contrib/tree/2048_cuda_cascade_no_longer_available
16. “Can't use SIFT in Python OpenCV v4.20.” Stack Overflow https://stackoverflow.com/questions/60065707/cant-use-sift-in-python-opencv-v4-20
17. “USB 2.0 vs USB 3.0 - Difference and Comparison | Diffen.” Diffen.com https://www.diffen.com/difference/USB_2.0_vs_USB_3.0

11. Code and Appendices

1).face_tracker.cpp of the optimal goal:

```
#include <stdio.h>
#include <iostream>
#include <iomanip>
#include <chrono>
#include <ctime>
#include <atomic>
#include <syslog.h>
#include <thread>
#include <map>
#include <opencv2/core/base.hpp>
#include <opencv2/calib3d/calib3d.hpp>
#include <opencv2/opencv.hpp>
#include <opencv2/tracking.hpp>
#include <opencv2/tracking/tracker.hpp>
#include <opencv2/core/ocl.hpp>
#include "opencv2/objdetect.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/imgcodecs.hpp"
#include "opencv2/imgproc.hpp"
#include "opencv2/face.hpp"
#include "concurrent_queue_cv.h"
#include "opencv2/cudaobjdetect.hpp"
#include <opencv2/core/cuda.hpp>
#include <opencv2/cudastereo.hpp>
#include <opencv2/cudaimgproc.hpp>
#include <opencv2/cudawarping.hpp>
#include <opencv2/cudaarithm.hpp>
#include "cv_barrier.h"

using namespace std::chrono;
using namespace std;
using namespace cv;
using namespace cv::face;


#define WIDTH                   (640)
#define HEIGHT                  (480)
#define LEFT_DISPLAY_WINDOW     "Left-Tracking"
#define RIGHT_DISPLAY_WINDOW    "Right-Tracking"
#define TEST_IMG_WIDTH          (320)
#define TEST_IMG_HEIGHT         (243)
#define MIN_DISTANCE            (2)
```

```
#define INFO_LOG(...) \
do {    \
   syslog(LOG_INFO, ##__VA_ARGS__); \
}while(0)

#define ERR_LOG(...) \
do {    \
   syslog(LOG_ERR, ##__VA_ARGS__); \
}while(0)

#define CRIT_LOG(...) \
do {    \
   syslog(LOG_CRIT, ##__VA_ARGS__); \
}while(0)

// Convert to string
#define SSTR( x ) static_cast< std::ostringstream & >( \
( std::ostringstream() << std::dec << x ) ).str()


// initialize values for StereoBM parameters
int numDisparities = 144;
int blockSize = 39;
int preFilterType = 1;
int preFilterSize = 9;
int preFilterCap = 48;
int textureThreshold = 9;
int uniquenessRatio = 16;
int speckleRange = 6;
int speckleWindowSize = 24;
int disp12MaxDiff = 0;
float minDisparity = 0.0f;
float M = 4.1320842742919922e+01;


// Global mutex instance for protecting cascade
mutex cascade_mutex;
// Global stereo pointer
cv::Ptr<cuda::StereoBM> stereo;

// Global concurrent barrier to synchornize the two facial recognition threads
CVBarrier barrier {2u};

// declare atomic bool variable as the program terminartion flag
atomic<bool>terminate_flag{false};

// vector<std::string>faces{"P1", "P2", "P3", "P4", "P5", "P6", "P7", "P8", \
// "P9", "P10", "P11", "P12", "P13", "P14", "P15", "Shuran", "P17", "P18", "P19",
"P20", "P21"};
```

```
vector<std::string>faces{"P1", "P2", "P3", "P4", "P5", "P6", "P7", "P8", \
"P9", "P10", "P11", "P12", "P13", "P14", "P15", "Shuran"};

// declare a map of (string, int) pairs for detected labels of both cameras
std::map<std::string,std::atomic<int>> label_map;
std::map<std::string,concurrent_queue_cv<std::string>> dist_map;
std::map<std::string,concurrent_queue_cv<cv::Mat>> mat_map;
std::map<std::string,cv::cuda::Stream> stream_map;


// Program Status Enum types
typedef enum {
   RECOGNIZE=1,
   TRACK
}prog_status_t;


void estimate_distance()
{
   cv::Mat imgL, imgR;
   cv::Mat mean;
   cuda::GpuMat d_mean;
   cuda::GpuMat d_imgL, d_imgR, d_imgL_gray, d_imgR_gray;
   cuda::GpuMat d_disp, d_disparity, d_depth_map;
   int min_cols, min_rows;

   while(true) {

       // return the function if the termination flag is set
       if(terminate_flag){
           break;
       }

       // Pop matries from both left and right cameras
       mat_map[LEFT_DISPLAY_WINDOW].pop(imgL);
       mat_map[RIGHT_DISPLAY_WINDOW].pop(imgR);
       //upload matrices
       d_imgL.upload(imgL);
       d_imgR.upload(imgR);
       // Convert matries to gray-scaled
       cuda::cvtColor(d_imgL, d_imgL_gray, cv::COLOR_BGR2GRAY);
       cuda::cvtColor(d_imgR, d_imgR_gray, cv::COLOR_BGR2GRAY);
       // Resize matrices
       min_cols = min(imgL.cols, imgR.cols);
       min_rows = min(imgL.rows, imgL.rows);
       cuda::resize(d_imgL_gray, d_imgL_gray, Size(min_rows, min_cols),
INTER_LINEAR);
       cuda::resize(d_imgR_gray, d_imgR_gray, Size(min_rows, min_cols),
```

```
INTER_LINEAR);

        try{
            stereo->compute(d_imgL_gray,d_imgR_gray,d_disp);
        }
        catch(exception &e){
            cout << "exception caught: " << e.what() << endl;
            break;
        }

        // NOTE: compute returns a 16bit signed single channel image,
        // CV_16S containing a disparity map scaled by 16. Hence it
        // is essential to convert it to CV_16S and scale it down 16 times.
        d_disp.convertTo(d_disparity,CV_8U);

        // Scaling down the disparity values and normalizing them
        // disparity = (disparity/(float)16.0 -
(float)minDisparity)/((float)numDisparities);

        // Calculating disparity to depth map using the following equation
        // ||    depth = M * (1/disparity)   ||
        // depth_map = (float)M/disparity;

        cv::cuda::GpuMat d_M(min_cols, min_rows, CV_8U);
        d_M.setTo(M);
        cv::cuda::divide(d_M, d_disparity, d_depth_map);

        // Calculating the average depth of the object closer than the safe
distance
        cuda::meanStdDev(d_depth_map, d_mean);

        // Download mean
        d_mean.download(mean);
        // Printing the warning text with object distance
        char dist[10];
        std::sprintf(dist, "%.2f",mean.at<double>(0,0));
        // Push the dist text to the dist_map
        dist_map[LEFT_DISPLAY_WINDOW].push(dist);
        dist_map[RIGHT_DISPLAY_WINDOW].push(dist);
    }
}

Rect2d&& detect_face( Ptr<cuda::CascadeClassifier> &cascade, Mat frame, \
cv::cuda::Stream& stream )
{
    cuda::GpuMat d_frame, d_frame_gray, d_buf;
    // Detect faces
    std::vector<Rect> faces;
```

```
    d_frame.upload(frame, stream);
    cuda::cvtColor( d_frame, d_frame_gray, COLOR_BGR2GRAY, 0, stream);
    cuda::equalizeHist( d_frame_gray, d_frame_gray, stream);

    try{
        lock_guard<mutex> cm(cascade_mutex);
        cascade->detectMultiScale( d_frame_gray, d_buf);
        cascade->convert(d_buf, faces);
    }
    catch (const std::exception& e){
        std::cout << "Face detection exception caught: " << e.what() << std::endl;
        return std::move(Rect2d(0,0,0,0)); // mark Rect (0,0,0,0) as the failure
to detect faces
    }

    if(faces.size() == 0){
        return std::move(Rect2d(0,0,0,0)); // mark Rect (0,0,0,0) as the failure
to detect faces
    }

    // Return the coordinate of the first detected face
    return std::move(Rect2d(faces[0].x, faces[0].y, faces[0].width,
faces[0].height));
}


int inline recognize_face(Mat face, Ptr<FisherFaceRecognizer> &model)
{
    // Convert the matrix from RGB to Grayscale
    Mat face_gray;
    cvtColor( face, face_gray, COLOR_BGR2GRAY );
    // Resize the face_mat to be the same size as the train image size
    resize(face_gray, face_gray, Size(TEST_IMG_WIDTH, TEST_IMG_HEIGHT),
INTER_LINEAR);
    // Predict and return the label
    return model->predict(face_gray);
}

std::string get_time()
{
    auto now = std::chrono::system_clock::now();
    std::time_t now_time = std::chrono::system_clock::to_time_t(now);
    return std::ctime(&now_time);
}

void recognise_and_track(VideoCapture&& cap, Ptr<cuda::CascadeClassifier>
&cascade, \
std::string orientation)
{
```

```
    Mat img;
    Rect2d face;
    Rect2d bbox;
    Ptr<Tracker> tracker;
    double timer{0.0};
    float fps{0.0};
    int predictedLabel;
    prog_status_t thread_mode {RECOGNIZE};
    // Create the recognizer
    Ptr<FisherFaceRecognizer> model = FisherFaceRecognizer::create();
    // Load the trainer model
    model->read("../../train.yml");

    while (true)
    {
        // return the function if the termination flag is set
        if(terminate_flag){
            // destroy the current display window
            destroyWindow(orientation);
            // push a dummy matrix to lcq and rcq to make the disparity map thread
proceed
            mat_map[LEFT_DISPLAY_WINDOW].push(img);
            mat_map[RIGHT_DISPLAY_WINDOW].push(img);
            return;
        }

        timer = (double)getTickCount();
        cap >> img;

        if( img.empty())
        {
            cout << "No captured frame !\n";
            continue;
        }

        // Display tracker type on frame
        putText(img, "KCF Tracker", Point(100,20), FONT_HERSHEY_SIMPLEX, \
            0.75, Scalar(50,170,50),2);

        switch(thread_mode)
        {
            case RECOGNIZE:
            {
                // Apply the classifier to img
                face = detect_face(cascade, img,
std::ref(stream_map[orientation]));

                if(face.width != 0 && face.height != 0 ){
```

```
                    // Recognize the face by predicting the label
                    predictedLabel = recognize_face(cv::Mat(img, face), model);
                    // cout << orientation << " is about to enter wait state" <<
endl;
                    barrier.Wait();
                    // Update the corresponding label_map entry
                    label_map[orientation] = predictedLabel;
                    // Update the bounding box
                    bbox.x = face.x;
                    bbox.y = face.y;
                    bbox.width = face.width;
                    bbox.height = face.height;
                    // Create a KCF tracker
                    tracker = TrackerKCF::create();
                    // initialize the tracker
                    tracker->init(img, std::ref(bbox));
                    // update thread mode
                    thread_mode = TRACK;
                }
                break;
            }
            case TRACK:
            {
                bool success = tracker->update(img, std::ref(bbox));
                if(!success || (label_map[LEFT_DISPLAY_WINDOW] !=
label_map[RIGHT_DISPLAY_WINDOW])){

                    // print the error message if tracking update failed
                    if(!success){
                        // cout << orientation << "tracking failed" << endl;
                        putText(img, "Tracking failure detected", Point(100,80), \
                            FONT_HERSHEY_SIMPLEX, 0.75, Scalar(0,0,255),2);
                    }

                    // cout << orientation << "left label = " <<
label_map[LEFT_DISPLAY_WINDOW] << endl;
                    // cout << orientation << "right label = " <<
label_map[RIGHT_DISPLAY_WINDOW] << endl;

                    // Clear the label_map value
                    label_map[orientation] = -1;
                    // Clear the tracker
                    tracker->clear();
                    // update thread mode
                    thread_mode = RECOGNIZE;

                    break;
                }
```

```
                // Tracking success : Draw the tracked object
                rectangle(img, bbox, Scalar( 255, 0, 0 ), 2, 1 );
                // Create cropped matrix based on the bbox size
                Rect roi{(int)bbox.x, (int)bbox.y, (int)bbox.width,
(int)bbox.height};
                Mat cropped_img = cv::Mat(img, roi);
                mat_map[orientation].push(cropped_img);

                // Display recognition anotation on frame
                if(predictedLabel > faces.size()){
                    putText(img, "Stranger", Point(bbox.x-6,bbox.y),
FONT_HERSHEY_SIMPLEX, \
                    0.75, Scalar(50,170,50),2);
                }
                else{
                    putText(img, faces[predictedLabel - 1],
Point(bbox.x-6,bbox.y), FONT_HERSHEY_SIMPLEX, \
                    0.75, Scalar(50,170,50),2);
                }

                // Display the distance value if possible
                putText(img, "Distance : ", Point(100,70), FONT_HERSHEY_SIMPLEX, \
                    0.75, Scalar(50,170,50),2);

                if(dist_map[orientation].size()){
                    std::string dist;
                    dist_map[orientation].pop(dist);
                    putText(img, dist + " cm", Point(235,70),
FONT_HERSHEY_SIMPLEX, \
                    0.75, Scalar(50,170,50),2);
                    if(stoi(dist) < MIN_DISTANCE){
                        putText(img, "Warning, too close !", Point(100,93),
FONT_HERSHEY_SIMPLEX, \
                    0.75, Scalar(0,0,255),2);
                    }
                }
                break;
            }

            default:
                break;
        }

        // Calculate Frames per second (FPS)
        fps = getTickFrequency() / ((double)getTickCount() - timer);
        // Display FPS on frame
        putText(img, "FPS : " + SSTR(int(fps)), Point(100,50),
FONT_HERSHEY_SIMPLEX, 0.75, Scalar(50,170,50), 2);
        // Log the FPS value
```

```
        INFO_LOG("camera=%s FPS=%.2f",orientation.c_str(), fps);
        // Display frame.
        imshow(orientation, img);
        // Exit if ESC pressed.
        int k = waitKey(1);
        if(k == 27)
        {
            terminate_flag = true;
        }
    }
}


int main(int argc, char **argv)
{
    int CamL_id{1}; // Camera ID for left camera
    int CamR_id{0}; // Camera ID for right camera

    // Set terminate_flag to false
    terminate_flag = false;

    // Reading the stored the StereoBM parameters
    cv::FileStorage cv_file =
cv::FileStorage("../../data/depth_estimation_params_cpp.xml",
cv::FileStorage::READ);
    cv_file["numDisparities"] >> numDisparities;
    cv_file["blockSize"] >> blockSize;
    cv_file["preFilterType"] >> preFilterType;
    cv_file["preFilterSize"] >> preFilterSize;
    cv_file["preFilterCap"] >> preFilterCap;
    cv_file["minDisparity"] >> minDisparity;
    cv_file["textureThreshold"] >> textureThreshold;
    cv_file["uniquenessRatio"] >> uniquenessRatio;
    cv_file["speckleRange"] >> speckleRange;
    cv_file["speckleWindowSize"] >> speckleWindowSize;
    cv_file["disp12MaxDiff"] >> disp12MaxDiff;


    // stereo = cv::cuda::StereoBM::create();
    stereo = cuda::createStereoBM();

    // updating the parameter values of the StereoSGBM algorithm
    stereo->setNumDisparities(numDisparities);
    stereo->setBlockSize(blockSize);
    stereo->setUniquenessRatio(uniquenessRatio);
    stereo->setSpeckleRange(speckleRange);
    stereo->setSpeckleWindowSize(speckleWindowSize);
    stereo->setDisp12MaxDiff(disp12MaxDiff);
    stereo->setMinDisparity(minDisparity);
```

```
   // initialize cuda-version CascadeClassifier
   Ptr<cuda::CascadeClassifier> cascade;
   cascade =
cv::cuda::CascadeClassifier::create("../haarcascade_frontalface_alt.xml");
   bool findLargestObject = false;
   bool filterRects = true;
   cascade->setFindLargestObject(findLargestObject);
   cascade->setScaleFactor(1.2);
   cascade->setMinNeighbors((filterRects || findLargestObject) ? 4 : 0);

   // initialize stream map
   stream_map[LEFT_DISPLAY_WINDOW] = cv::cuda::Stream();
   stream_map[RIGHT_DISPLAY_WINDOW] = cv::cuda::Stream();

   // initialize label map
   label_map[LEFT_DISPLAY_WINDOW] = -1;
   label_map[RIGHT_DISPLAY_WINDOW] = -1;

   cv::VideoCapture camL(CamL_id), camR(CamR_id);

   // Check if left camera is attched
   if (!camL.isOpened())
   {
       std::cout << "Could not open camera with index : " << CamL_id <<
std::endl;
       return -1;
   }

   // Check if right camera is attached
   if (!camR.isOpened())
   {
       std::cout << "Could not open camera with index : " << CamR_id <<
std::endl;
       return -1;
   }

   // Configure the camera resolutions
   camL.set(CAP_PROP_FRAME_WIDTH, WIDTH);//Setting the width of the video
   camL.set(CAP_PROP_FRAME_HEIGHT, HEIGHT);//Setting the height of the video//

   camR.set(CAP_PROP_FRAME_WIDTH, WIDTH);//Setting the width of the video
   camR.set(CAP_PROP_FRAME_HEIGHT, HEIGHT);//Setting the height of the video//

   thread tl{ recognise_and_track, std::move(camL) , std::ref(cascade),
LEFT_DISPLAY_WINDOW };
   thread tr{ recognise_and_track, std::move(camR) , std::ref(cascade),
RIGHT_DISPLAY_WINDOW };
   thread td{ estimate_distance };
```

```
    tl.join();
    tr.join();
    td.join();

    return 0;
}
```

2). Capture_stats.sh:

```
#!/bin/sh
# Usage: ./monitor-usage.sh <PID of the process> <Output folder>
# Output: top.dat with lines such as `305m 2.0`, i.e. memory with m/g suffix -
CPU load in %

PID="$1"
OUTPUT_DIR="$2"
OUTPUT="$OUTPUT_DIR"/top.csv
echo 'memory(Mb),cpu(%)' > "$OUTPUT"
while true;
do
    # check if the process is running
    t_pid=$( ps aux | awk '{print $2 }' | grep "$PID" )
    if [ -z "$t_pid" ];
    then
        echo "Process ""$PID"" has exited, stop capturing stats"
        break
    fi
    top -p $PID -bn 1 | egrep "$PID" | awk '{print $6/1000,","$9}' >> "$OUTPUT"
done
```

3). extract_syslog_for_fps.sh

```
#!/bin/bash
# This script extracts FPS data from the syslog file for both the left camera and
the
# right camera. The extracted files are saved as .csv files for post-processing
by
# data-analysis.ipynb

BASE_DIR="$1"
```

```
DIRECTORY="$BASE_DIR"/fps_logs

if [ ! -d "$DIRECTORY" ];
then
   mkdir "$DIRECTORY"
fi

cd "$DIRECTORY"

# loop through the directory
for d in */ ; do

   SOURCE="$d"/fps_log

   # skip this directory if fps_log is not found
   if [ ! -f "$SOURCE" ];
   then
       continue
   fi

   LCAM_CSV="$d"/lcam_fps.csv
   RCAM_CSV="$d"/rcam_fps.csv

   #Extract FPS values for the left camera
   echo "fps" > "$LCAM_CSV"
   cat ${SOURCE} | grep "Left-Tracking" | awk '{print $7}' | cut -d "=" -f2 >>
"$LCAM_CSV"

   #Extract FPS values for the right camera
   echo "fps" > "$RCAM_CSV"
   cat ${SOURCE} | grep "Right-Tracking" | awk '{print $7}' | cut -d "=" -f2 >>
"$RCAM_CSV"

done
```

4). jtop_stats.py

```
# Jetson stat tutorial page can be found as follows:
# https://rnext.it/jetson_stats/jtop.html

from jtop import jtop
import pprint
import time
import os

def read_stats(jetson):
```

```
    # clear the Screen
    os.system('clear')
    print("\n" * 3)
    print("JTOP Statistic Summary:")
    stats = jetson.stats
    print_stats = pprint.pformat(stats, indent=4)
    print(print_stats)
    # delay 1 second prior to collect data again
    time.sleep(1)



jetson = jtop()
jetson.attach(read_stats)
jetson.loop_for_ever()
```

5).gpu_logger.py

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# This program logs the GPU usage data as well as the GPU temperaure data
consistently to the log.csv file.
# The implementation is based on the example given from the jetson-stat
repository:
# https://github.com/rbonghi/jetson_stats/blob/master/examples/jtop_logger.py

from jtop import jtop, JtopException
import csv
import argparse


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Simple jtop logger')
    # Standard file to store the logs
    parser.add_argument('--file', action="store", dest="file",
default="gpu_log.csv")
    args = parser.parse_args()

    print("Simple jtop logger")
    print("Saving log on {file}".format(file=args.file))

    try:
        with jtop() as jetson:
            with open(args.file, 'w') as csvfile:
                stats = jetson.stats
                print(type(stats))
```

```
                # Initialize cws writer
                writer = csv.DictWriter(csvfile, fieldnames=["GPU", "Temp_GPU"])
                # Write header
                writer.writeheader()
                # This is to ensure that the data is actually written to the file,
                # especially when running the script as a background process
                csvfile.flush()
                # Write first row
                gpu_stats = {"GPU": stats["GPU"], "Temp GPU" : stats["Temp GPU"]}
                writer.writerow(gpu_stats)
                csvfile.flush()
                # Start loop
                while jetson.ok():
                    stats = jetson.stats
                    gpu_stats = {"GPU": stats["GPU"], "Temp GPU" : stats["Temp
GPU"]}
                    # Write row
                    writer.writerow(gpu_stats)
                    csvfile.flush()
                    print("Log at {time}".format(time=stats['time']))
    except JtopException as e:
        print(e)
    except KeyboardInterrupt:
        print("Closed with CTRL-C")
    except IOError:
        print("I/O error")
# EOF
```

6). Run.sh

```
#!/bin/bash
# This script runs both the executable as well as capture_stats.sh to monitor &
collect
# CPU,mem,GPU data used by the program over the its lifetime. All collected data
will
# be saved as .csv files for post-processing by data-analysis.ipynb

BASE_DIR="$1"
OPTION="$2"
WORK_DIR="$BASE_DIR"/"$OPTION"
BUILD_PATH="$WORK_DIR"/build/
EXPR_ITER_FN="$WORK_DIR"/expr_iter
EXPR_ITER=0

cd "$BASE_DIR"

# Launch gpu_logger.py for GPU statistical sampling
```

```
python3 gpu_logger.py --file="$WORK_DIR"/gpu_log.csv &
GPU_LOGGER_PID=$!

# Run the target face_tracker
cd "$BUILD_PATH"
./face_tracker &

# Obtain the face_tracker process ID
FT_PID=$!

# Update the experiment number
if [ ! -f "$EXPR_ITER_FN" ];then
   EXPR_ITER=1
   echo "1" > "$EXPR_ITER_FN"
   mkdir -p "$WORK_DIR"/"top-1"
else
   iter=$(cat "$EXPR_ITER_FN")
   EXPR_ITER=$(( $iter + 1 ))
   echo ${EXPR_ITER} > "$EXPR_ITER_FN"
   mkdir -p "$WORK_DIR"/"top-""$EXPR_ITER"
fi

cd "$BASE_DIR"

# Capture GPU & Memory data to .dat
./capture_stats.sh "$FT_PID" "$WORK_DIR"/"top-""$EXPR_ITER"

# Kill gpu_logger.py
kill -9 "$GPU_LOGGER_PID"
```