TU WIEN Informatics

CycleSafely Mobile

Porting the CycleSafely pipeline to mobile and evaluating its performance

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Matyas Hoffer-Toth
Registration Number 12122086

to the Faculty of Informatics

at the TU Wien

Advisor: Dr.techn. Michael Wimmer
Assistance: Stefan Ohnhallinger, PhD

Vienna, August 9, 2024

Matyas Hoffer-Toth Michael Wimmer

Erklärung zur Verfassung der Arbeit

Matyas Hoffer-Toth

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 9. August 2024

Matyas Hoffer-Toth

Kurzfassung

Fahrradfahren im Straßenverkehr kann wegen mangelndem Schutz im Fall einer Kollision gefährlich sein. Damit ein System entscheiden kann, ob eine Kollision in naher Zukunft geschehen wird, muss es in der Lage sein, Verkehrsteilnehmer zu erkennen, sie über mehrere Frames zu verfolgen, um Information über ihre vergangenen Positionen zu erhalten, und ihre zukünftigen Pfade vorherzusagen. In dieser Arbeit wird eine mobile Pipeline als plattformübergreifende Flutter-Applikation [1] implementiert. Die Pipeline wird von einer Python-Implementierung portiert, die dazu gemacht wurde, auf einem Desktop-Computer zu laufen. Während Objektverfolgung mittels einer algorithmischen Methode erfolgt, wird für die Objekterkennung und die Vorhersage der Pfade jeweils ein neuronales Netz verwendet. Aus Kompatibilitätsgründen wird das Modell zur Pfadvorhersage von einem einfachen Regressionsalgorithmus ersetzt. Die Auswertung zeigt, dass der Objektdetektor für Kollisionserkennung in Echtzeit zu langsam ist. Aus diesem Grund wird eine Lösung untersucht, bei der die meiste Rechenleistung an einen externen Server delegiert wird.

Abstract

Riding a bicycle in traffic can be dangerous due to little protection in case of a collision. To decide whether a collision will happen in the future, a system must be able to detect road users, track them to gain information about their past positions, and predict their trajectories. In this work, a mobile pipeline is implemented as a cross-platform Flutter [1] application. The pipeline is ported from a Python implementation designed to run on a desktop computer. While the object tracking is done via an algorithmic method, the object detection and the trajectory prediction each leverage a neural network. Due to compatibility issues, the trajectory prediction model is replaced by a simple regression algorithm. The evaluation shows that the object detector is too slow for real-time collision detection. Therefore, a solution where most processing is delegated to a remote server is also explored.

Contents

CHAPTER 1

Introduction

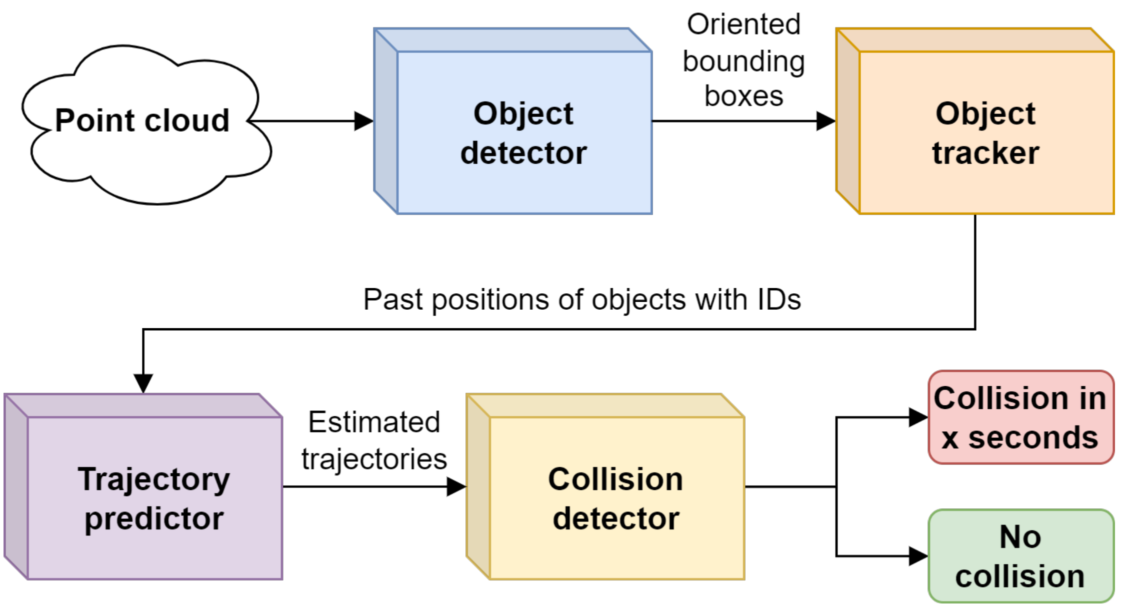


Figure 1.1: Visualization of the pipeline. The input is a point cloud and is passed to the object detector. Then, a set of oriented bounding boxes of vehicles is returned. The object tracker assigns IDs to them so that the trajectory predictor can estimate their future paths. Finally, the collision detector decides whether a collision will happen in the future.

Navigating traffic can be a dangerous endeavor, especially for a cyclist. In urban areas, cyclists commute alongside multi-ton vehicles, yet they have little to no protection in case of an accident. Would it not be great to have a simple and convenient tool to anticipate collisions and warn the biker of an upcoming hazardous situation? In light of this, a pipeline for collision prediction from 3D point cloud data was ported to a mobile application in this work.

A pipeline that can predict whether a crash will occur must first detect relevant objects in a scene, such as cars, cyclists, and pedestrians. The object detection needs to be done over multiple frames. Then, the pipeline requires an algorithm that can tell which object in one frame is the same in another. Once enough past information about an object is gathered, the pipeline can compute the future trajectory of an object. Using the predicted trajectories of the other road users, the pipeline can check whether any trajectory overlaps with the trajectory of the ego vehicle (in our case, the cyclist recording the data) at the same frame in the future. If so, our cyclist can be warned with a visual or auditory cue.

Figure 1.1 depicts each pipeline module and shows how the data flows through it to arrive at a final decision. The object detector detects other road users. The object tracker assigns IDs to objects across multiple frames to track them. The trajectory predictor estimates the future paths of the other commuters and the ego vehicle itself. Finally, the collision detector decides whether a collision will occur.

This paper will cover details of the implementation, evaluation, and potential improvements to the mobile pipeline that can be explored in future work. Furthermore, it will shed light on challenges faced during development. The final application runs on Android and iOS, but requires further optimization, which will be discussed in Chapter 5.

The mobile pipeline is based on a desktop pipeline developed and implemented by Simon Pointner [2]. This work ports the original pipeline to a mobile version using Flutter [1].

CHAPTER 2

Related Work

The pipeline presented in this paper consists of four main steps: Object detection, object tracking, trajectory prediction, and collision detection. Therefore, this chapter will look at related work for each topic separately.

2.1 Object Detection

Object detection concerns itself with recognizing specific objects in input data, which can be a 2D image or a 3D point cloud. The task may or may not include classifying the detected objects and assigning a location to them. The method used [3] in this work takes a point cloud as input, which is a set of points in 3D space. The point cloud is then converted into a bird's-eye-view map containing information about height, density, and intensity. Then, a trained model processes the map as input and outputs the center, dimensions, and angle of each object, providing enough information to construct an oriented bounding box.

Another, more challenging approach is using only a 2D image as input, also known as monocular detection. One solution is the RTM3D model [4], for example. It first finds 9 key points of an object that represent a projection of a bounding box including its center point onto the image. Then the bounding box is estimated by minimizing the reprojection error, i.e., the projected points of the bounding box onto the image should be as close to the key points as possible.

Multi-modal methods that combine LiDAR point cloud information with a corresponding image of the scene also exist [5, 6].

2.2 Object Tracking

Multi-object tracking aims to identify the same objects across multiple frames. This can be done by assigning a unique ID to each detected object in each frame. Most approaches have similar ideas, regardless of the tracking being 2D or 3D [7, 8]. First, the objects need to be detected by an object detector. Then some form of motion estimation is employed, for example with the help of a Kalman filter. Using the gathered data, the detections are associated across multiple frames with an association metric. Other features of the object can also be calculated to improve performance, such as appearance vectors used in the DeepSORT algorithm [9].

In this work, an algorithmic multi-object tracker [10] is used to track objects in 3D space. The algorithm takes bounding boxes of objects detected in a scene as input in each frame. It then computes a cost map between the currently detected objects and predicted trajectories from previous frames using a constant acceleration model. With the help of the cost map, either an existing ID is associated with a bounding box, or a box receives a new one.

2.3 Trajectory Prediction

By predicting an object's trajectory, one can estimate where an object may be located in the future at a certain time. The survey *Trajectory-Prediction with Vision* [11] gives a good overview of what data needs to be collected and what challenges one might need to overcome to make vehicle trajectory prediction work. Inputs for such prediction systems can be 2D images, LiDAR point clouds, or GPS information, for example. A good predictor must not only have a good understanding of the road environment but also be able to estimate the intentions of other road users.

The trajectory predictor used when offloading most of the pipeline's processing steps to a more powerful server is called ESP from the PRECOG paper [12]. It is a trained model aiming to predict the trajectories of agents in a scene from past positions and LiDAR data for environmental information. The paper also proposes the PRECOG model, which takes advantage of an agent's goal or planned path in addition to positional and environmental data. It is observed that this additional knowledge of the agent's intentions improves predictions of other road users as well.

2.4 Collision Detection

A simple approach to determining whether two vehicles collide is checking whether the Euclidean distance of their centers reaches a threshold. This would reduce the vehicles to perfect circles, which they are not.

To more accurately detect whether a collision of vehicles will happen in the future, the pipeline leverages the ideas of the Separating Axis Theorem (SAT) [13]. It tells us that if there exists a line that separates two shapes, they are not colliding as long as both shapes are convex. In other words, an axis must be found that separates both objects to conclude that they are not overlapping. Fortunately, the bounding boxes in the pipeline are rectangles and therefore convex. In total, at most four axes need to be checked for overlap, since each rectangle has 4 edges that are pair-wise parallel. If no overlap was found on one axis, the remaining axes no longer need to be considered.

CHAPTER 3

Implementation

To implement the CycleSafely pipeline on mobile, the Flutter framework [1] was chosen. It is a cross-platform development framework that compiles its applications into native machine code and is most useful for developing an app for Android and iOS simultaneously, which is why it was chosen. Furthermore, TensorFlow provides an official plugin [14] for running TensorFlow Lite models with Flutter.

The pipeline consists of 4 modules. It takes a set of points, called a point cloud, as input. A point cloud represents a traffic scene captured with a LiDAR camera.

The SFA3D neural-network model [3] acts as the object detector module that detects road users in the scene by providing an oriented bounding box for each detected object. The object tracker assigns an ID to each bounding box in a frame. The trajectory predictor predicts the trajectories of objects given their previous positions. For this, a simple regression is used when processing locally. Finally, a collision detector calculates which future bounding boxes overlap with the ego vehicle.

The pipeline's final output is the number of seconds the first collision will occur if one was predicted.

Since the inference performance of the object detector is slow on mobile, a solution where most processing is delegated to a server was also explored. When this solution is configured, the mobile app sends the point cloud to the server and receives the overlapping bounding boxes as the answer. The server pipeline uses the ESP model from the PRECOG paper [12] for object tracking instead of the simple regression that the on-device predictor uses.

The source code for the mobile app can be found here: `https://gitlab.cg.tuwien.ac.at/stef/cyclesafelymobile`

The source code for the server API can be found on the "mobile_api" branch of this repository: `https://gitlab.cg.tuwien.ac.at/stef/cyclesafely`

3.1 Object Detector

Since the SFA3D model [3] was implemented using PyTorch and Flutter does not have an official PyTorch plugin available, the model had to be converted to the TensorFlow Lite format. To achieve this, a converter called “Nobuco” [15] was used. It allows the conversion of a PyTorch model to Keras format. The Keras model can then be translated into TF Lite via TensorFlow. With that, the model is available for inference with Flutter.

However, before the model can process the point cloud it must be changed into a bird’s-eye-view map of shape (608, 608, 3). First, the point cloud is filtered by a bounding box for strict boundaries. Then, the x- and y-coordinates of the remaining 3D points need to be scaled to the size of the map. Each coordinate of the map encodes three pieces of information for that area:

- Height (of the highest point)
- Intensity (of the highest point)
- Density (of the points in that area)

This preprocessing is done twice for the front and back data. After inference, the detections are run through non-max suppression to filter out overlapping detections, i.e., the ones with the low scores are omitted. Then the top k (in this case 50) detections are selected. The model classifies the detections into three types of road users: car, pedestrian, and cyclist. Although, for the pipeline’s later steps, the object classes are not considered.

Each detected object has an x-, y-, and z-coordinate for the center point. It also has information about its height, width, and length. Lastly, each object has a yaw angle to tell which direction it is facing. With this, an oriented bounding box can be constructed around each object.

Before the results are passed to the object tracker, the coordinates, and dimensions need to be converted to world space, which can be done via a camera to world matrix. Furthermore, to avoid large and imprecise numbers, the origin of the coordinate system is set to the coordinates of the camera in the first frame.

The code for the pre- and postprocessing of the object detector was directly translated from the SFA3D repository’s Python implementation [3] into the Dart language which is used by Flutter [1].

3.2 Object Tracker

To track objects in 3D space, a simplified Python implementation [16] of a 3D multi-object tracking paper [10] was translated into Dart code to make it run with Flutter.

The tracker receives the detected oriented bounding boxes (center point, dimensions, and yaw), their detection scores, and the current frame index as input. Furthermore, it saves and uses processed data from previous frames.

Upon each iteration, objects that have not been tracked for a while are marked as “dead”, i.e., the IDs of those objects will not be used anymore. Then, the tracker computes a cost map between the current detections and the prediction data from previous frames based on a constant acceleration motion model. Finally, the IDs are greedily associated with the detections with the help of the cost map. Before the results are returned, the tracker updates prediction data based on the new results.

The final output of the tracker is the IDs associated with the detected oriented bounding boxes.

3.3 Trajectory Predictor

A simple approach is used to predict the trajectories of tracked road users, where only the x- and y-coordinates are considered. Initially, the idea was to fit a spline to the past positions of an object. Since Dart does not have a proper spline package, a polynomial regression was employed instead.

Two functions are fitted to the x- and y-positions each: $p_x(t)$ and $p_y(t)$, where t represents a timestamp index. To then predict the position (x_f, y_f) of an object at a future timestamp t_f the functions are evaluated:

$$x_f = p_x(t_f)$$
$$y_f = p_y(t_f)$$

For each object that has reached a certain number of past recorded positions n_{pos}, the positions for n_{pred} timestamps are predicted. $n_{pos} = 20$ and $n_{pred} = 20$ in the implementation and the values are adjustable. The predictions are then saved in a data structure accessible to the collision detector.

3.4 Collision Detector

For each predicted future frame (in this case, there are $n_{pred} = 20$ frames) and for each object including the ego vehicle, an oriented bounding box is calculated at their predicted position.

The width and length of the last recorded position are used. The direction of the vector from some frames (set to 4) previous to the current future frame and the current future frame is calculated for the yaw of the bounding box. The ego vehicle is assumed to have a width of 0.7 and a height of 1.8 meters, estimating the dimensions of a bicycle.

A collision is detected if a box overlaps with the bounding box of the ego vehicle in the same frame. To determine whether two oriented boxes overlap, the Separating Axis

Theorem (SAT) [13] is used. Four axes need to be checked (2 for each box). The corners of the oriented boxes are projected onto an axis and the minimum and maximum values of both projected boxes are found. The boxes A and B are overlapping on an axis if the following holds: $(min_A \leq max_B) \wedge (min_B \leq max_A)$. The two boxes touch if overlap on at least one axis is determined.

Eventually, the collision detector outputs the number of frames until the first collision, or zero if no collision was detected. One frame equates to 0.1 seconds in the pipeline implementation.

3.5 Remote Pipeline

The pipeline has also been implemented as remote calls to a server since the on-device inference times are slow. The conversion of ESP model [12] to the TF Lite format required modification of the TensorFlow source code. Furthermore, the model for trajectory prediction cannot run via the TF Lite mobile inference API because it uses float64 numbers, which are only properly supported on the desktop. These are also reasons for implementing a simple algorithm for trajectory prediction as described above.

When the remote pipeline is enabled, the mobile device sends the point cloud data to a server running the pipeline and ready for client requests. Once the data is transmitted, it runs all pipeline steps and returns all the predicted overlapping oriented bounding boxes and which future frames they belong to. This enables the mobile app to draw the received bounding boxes as visualization.

The app then informs about the nearest collision and sends the next frame's data to the server for another iteration. Web sockets were used to implement the two-way connection. The data is sent as protocol buffers [17], which help serialize it in an efficient and simple-to-use format. Two protocol buffers are defined, one for sending the input from the phone to the server and another for sending the outputs from the server to the phone.

Listing 3.1 shows the definition of the input protocol buffer. The data contains a list of float values. It is the client's responsibility to make sure that the length of this list is divisible by four since the server expects each point in a point cloud to be defined by an x-, y-, z-coordinate, and intensity. Furthermore, the latitude, longitude, and yaw are also sent to know the ego vehicle's position and orientation.

```
syntax = "proto3";

package pipeline_inputs;

message Data {
  repeated float pointCloud = 1 [packed=true];
  float lat = 2;
  float lon = 3;
  float yaw = 4;
}
```

Listing 3.1: Protocol buffer [17] definition for inputs

Listing 3.2 shows the definition of the output protocol buffer. It contains a list of collisions, where a collision is defined by the number of seconds the collision will occur in and the coordinates of the corners of the involved vehicles' bounding boxes. Therefore, each list needs to be divisible by eight and contain least 16 values. The first eight values always define the bounding box of the ego vehicle. The number of seconds to the first collision could have also been sent, but due to visualization purposes, this format was chosen. Compared to the input data, which is close to two megabytes, the output data is much smaller and takes less time to transfer.

```
syntax = "proto3";

package pipeline_outputs;

message Collision {
  int32 collision_in = 1;
  repeated float bounding_boxes = 2;
}

message Data {
  repeated Collision collisions = 1;
}
```

Listing 3.2: Protocol buffer [17] definition for outputs

3.6 Optimizations & further Details

The TF Lite inference API for Flutter [14] offers delegates to speed up the inference of a model. Therefore, the app allows one to configure those delegates for the object detector. The interface for choosing these optimizations is discussed in the next section. To enable one of the delegates, one has to pass them as options before creating the model's interpreter. Then, once inference is performed using the created instance, it will use the delegates specified upon creation.

The XNNPack delegate provides CPU optimizations to speed up inference. Furthermore, it allows one to choose the number of threads the model should use when running on the CPU. The Metal and Android GPU delegates are for allowing the model to leverage the GPU on iOS and Android, respectively. Only 103 out of 157 operations of the object detection model [3] can be executed on the GPU. Therefore, it is worth enabling a GPU and CPU delegate simultaneously if possible. Unfortunately, the Android test device (Galaxy A52s 5G) resulted in an error when the GPU delegate was enabled.

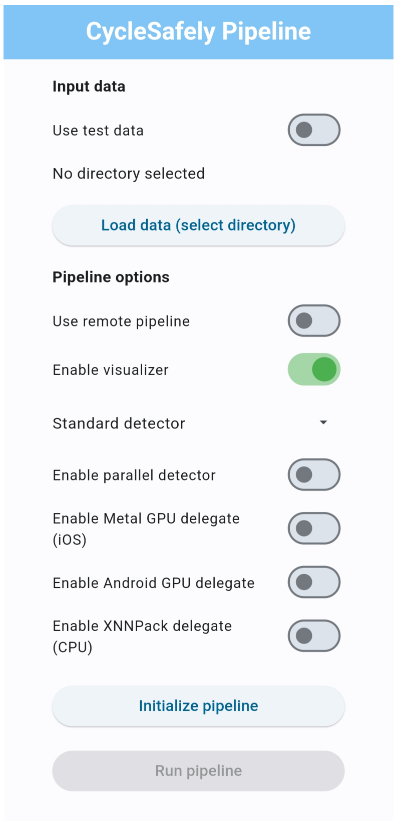
An optimization where the pipeline detects objects in the front and back simultaneously was also explored. Using this optimization slightly speeds up inference times, but lead to unpredictable and incorrect results when combined with a delegate. Furthermore, the object detection model [3] has been quantized using float16 and dynamic range quantization [18]. This aims to speed up processing time at the cost of quality. Chapter 4 demonstrates that, in this case, the quantized models are not worth using. The official TensorFlow Lite web page *Post-training quantization* [18] provides detailed information about these quantization techniques.

If the pipeline is run on the same thread as the main application, the interface freezes when the pipeline is running because of the heavy processing. To avoid this, the pipeline uses the pattern defined by the *Robust ports example* section of the Dart *Isolates* documentation [19]. Unlike most other programming languages, Dart does not have a thread API, instead, it offers isolates that communicate with each other via ports (not network ports). For example, the application running on the main isolate sends the inputs to the pipeline's isolate on the send port, and then the pipeline's isolate responds by sending the output of the run on the receive port.

Whenever the user presses "Initialize pipeline" the application spawns an isolate and creates the pipeline with the specified configuration. The most important objects and variables that are initialized on the new isolate are:

- An object detector instance that contains an initialized interpreter responsible for running the SFA3D model [3] with the specified delegates.
- An object tracker instance.
- A data structure responsible for storing past and predicted information about detected road users.
- A trajectory predictor instance that has access to the above-mentioned data structure.
- A collision detector instance that also receives a reference to the data structure.
- A run index that increments upon each iteration and assigns a number to each frame.
- A visualizer instance responsible for the bird's-eye-view map image shown when visualization is enabled.
- A remote runner instance that handles communication to a server in case the processing needs to be remotely delegated.

Furthermore, the new isolate is set to listen for incoming requests. Upon each iteration, the data is fed through each module in the right order. Eventually, collision information and, optionally, the BEV map image are sent back to the main isolate.


CycleSafely Pipeline

Input data

Use test data

No directory selected

Load data (select directory)

Pipeline options

Use remote pipeline

Enable visualizer

Standard detector

Enable parallel detector

Enable Metal GPU delegate (iOS)

Enable Android GPU delegate

Enable XNNPack delegate (CPU)

Initialize pipeline

Run pipeline

Figure 3.1: The screen when the application is opened.

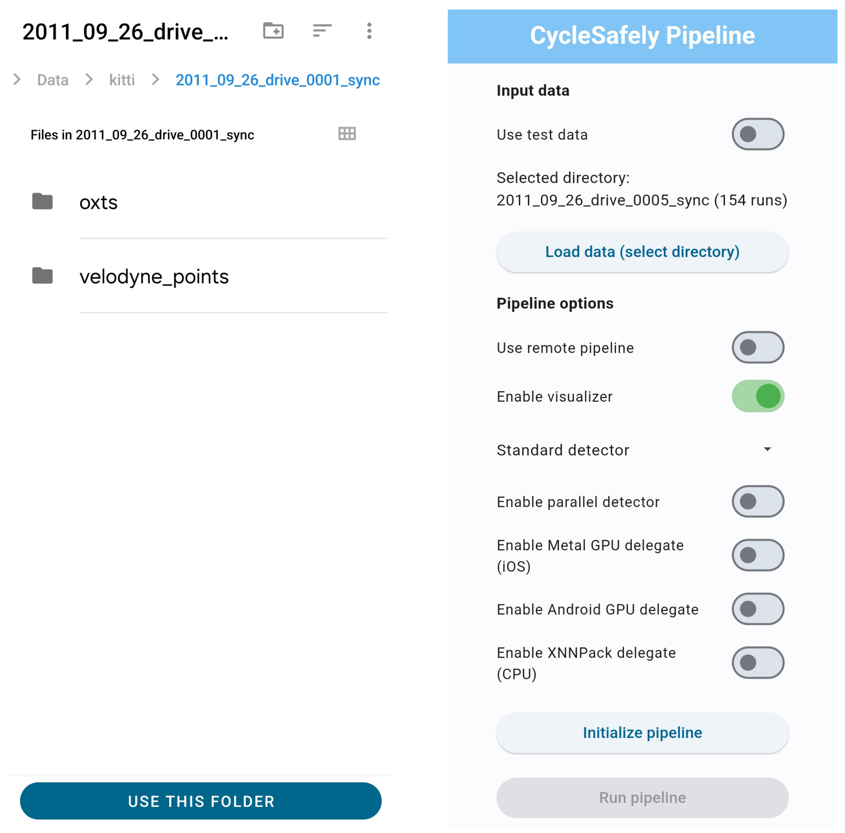
3.7 App Interface

The mobile application allows users to load LiDAR data, configure, and run the pipeline. Figure 3.1 shows the application screen right after it started.

Users can either load preinstalled data bundled with the application or load their own manually prepared data from the phone (only works on Android), as shown in Figure 3.2. The preinstalled test data contains the “2011_09_26_drive_0005_sync” raw KITTI city sequence [20] where the ego vehicle drives through and exits a roundabout. To load other point cloud data, the user must prepare it in the KITTI raw [20] format and select the directory that contains the “oxts” and “velodyne_points” folder with the appropriate data contained in them. Images are not needed to save storage space. If the remote pipeline is selected, the user must specify the server’s IP address, as shown in Figure 3.3a.

There are several options for on-device processing, all of which only concern the pipeline’s object detector and are discussed in Section 3.6 as well.

Firstly, the detector model can be changed from the standard version to either a float16 quantized or dynamic range quantized version [18]. The drop-down menu for this selection is displayed in Figure 3.3b.



2011_09_26_drive_...

> Data > kitti > 2011_09_26_drive_0001_sync

Files in 2011_09_26_drive_0001_sync

oxts

velodyne_points

USE THIS FOLDER

(a) The directory navigation screen.

CycleSafely Pipeline

Input data

Use test data: off

Selected directory:
2011_09_26_drive_0005_sync (154 runs)

Load data (select directory)

Pipeline options

Use remote pipeline: off

Enable visualizer: on

Standard detector

Enable parallel detector: off

Enable Metal GPU delegate (iOS): off

Enable Android GPU delegate: off

Enable XNNPack delegate (CPU): off

Initialize pipeline

Run pipeline

(b) A directory has been successfully loaded.

Figure 3.2: Selecting a data directory within the application.

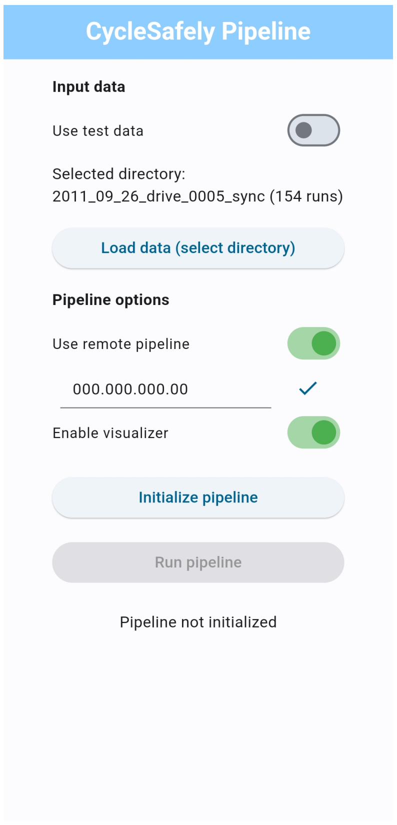
Users can enable the parallel detector, where front and back detections are run simultaneously in different isolates. However, this option is not compatible with any of the delegates.

To partially run the SFA3D model [3] on an iOS device's GPU (not all operations of the model can be executed on the GPU) the "Metal delegate" can be enabled. The option for the Android GPU delegate has also been implemented, but this leads to an error on the test device (Galaxy A52s 5G).

Lastly, the XNNPack delegate can be enabled for CPU optimizations and the number of threads to be used can also be specified, as shown in Figure 3.3c.

Once the data is ready and the desired options have been selected, the pipeline can be initialized and run. As displayed in Figure 3.4, while the pipeline is running the app displays information about the next predicted collision, the time it took for an iteration, and optionally the bird's-eye-view map of the points, detections, IDs, and trajectories. The bird's-eye-view map visualizer can be turned off to avoid a slight runtime increase.

The trajectories are only seen after the first 19 frames since the pipeline is configured to require 20 past positions for prediction, which is demonstrated in Figure 3.5. The visualizer also draws the bounding box of the colliding vehicles, as shown in Figure 3.6. Because of errors in tracking, the trajectories can sometimes suddenly speed up. Furthermore, filtering and marking stationary vehicles could increase collision detection quality.



CycleSafely Pipeline

Input data

Use test data

Selected directory:
2011_09_26_drive_0005_sync (154 runs)

Load data (select directory)

Pipeline options

Use remote pipeline

000.000.000.00

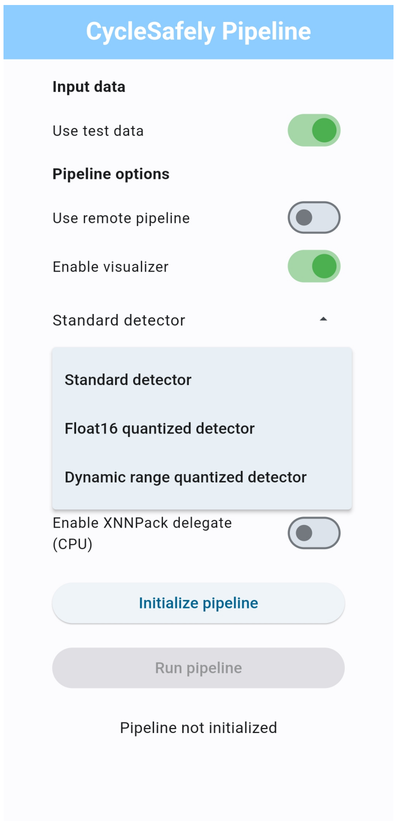
Enable visualizer

Initialize pipeline

Run pipeline

Pipeline not initialized

(a) The application with "Use remote pipeline" enabled.



CycleSafely Pipeline

Input data

Use test data

Pipeline options

Use remote pipeline

Enable visualizer

Standard detector

Standard detector

Float16 quantized detector

Dynamic range quantized detector

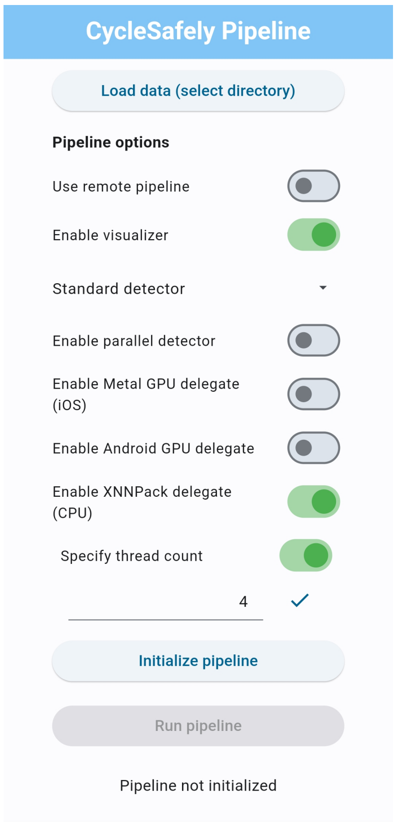
Enable XNNPack delegate (CPU)

Initialize pipeline

Run pipeline

Pipeline not initialized

(b) The application while selecting a quantized detector.



CycleSafely Pipeline

Load data (select directory)

Pipeline options

Use remote pipeline

Enable visualizer

Standard detector

Enable parallel detector

Enable Metal GPU delegate (iOS)

Enable Android GPU delegate

Enable XNNPack delegate (CPU)

Specify thread count

4

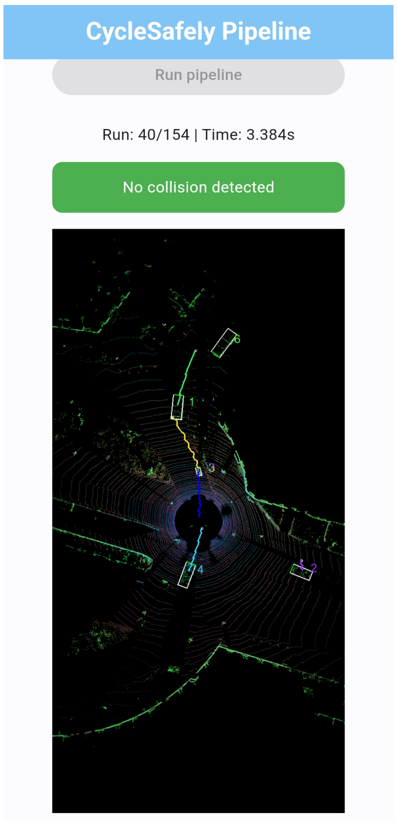
Initialize pipeline

Run pipeline

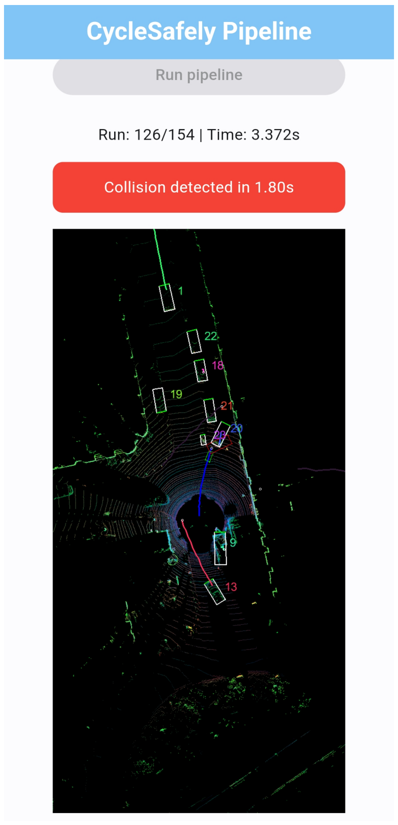
Pipeline not initialized

(c) The application with "Enable XNNPACK delegate" enabled.

Figure 3.3: The app interface shows additional input fields depending on which options are enabled and displays a dropdown for the detector options.

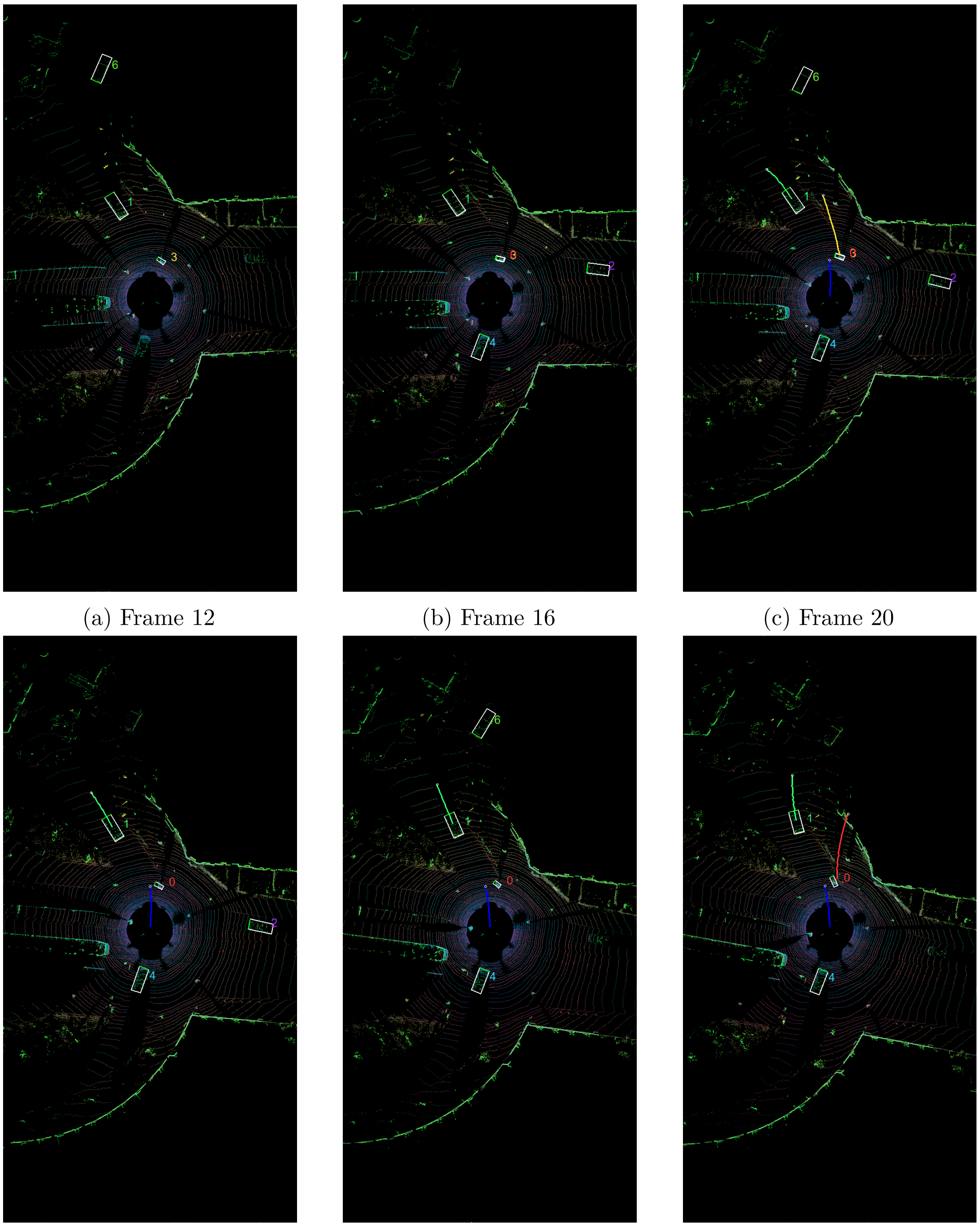


(a) The output of an iteration when no collision is detected.



(b) The output of an iteration when a collision is detected.

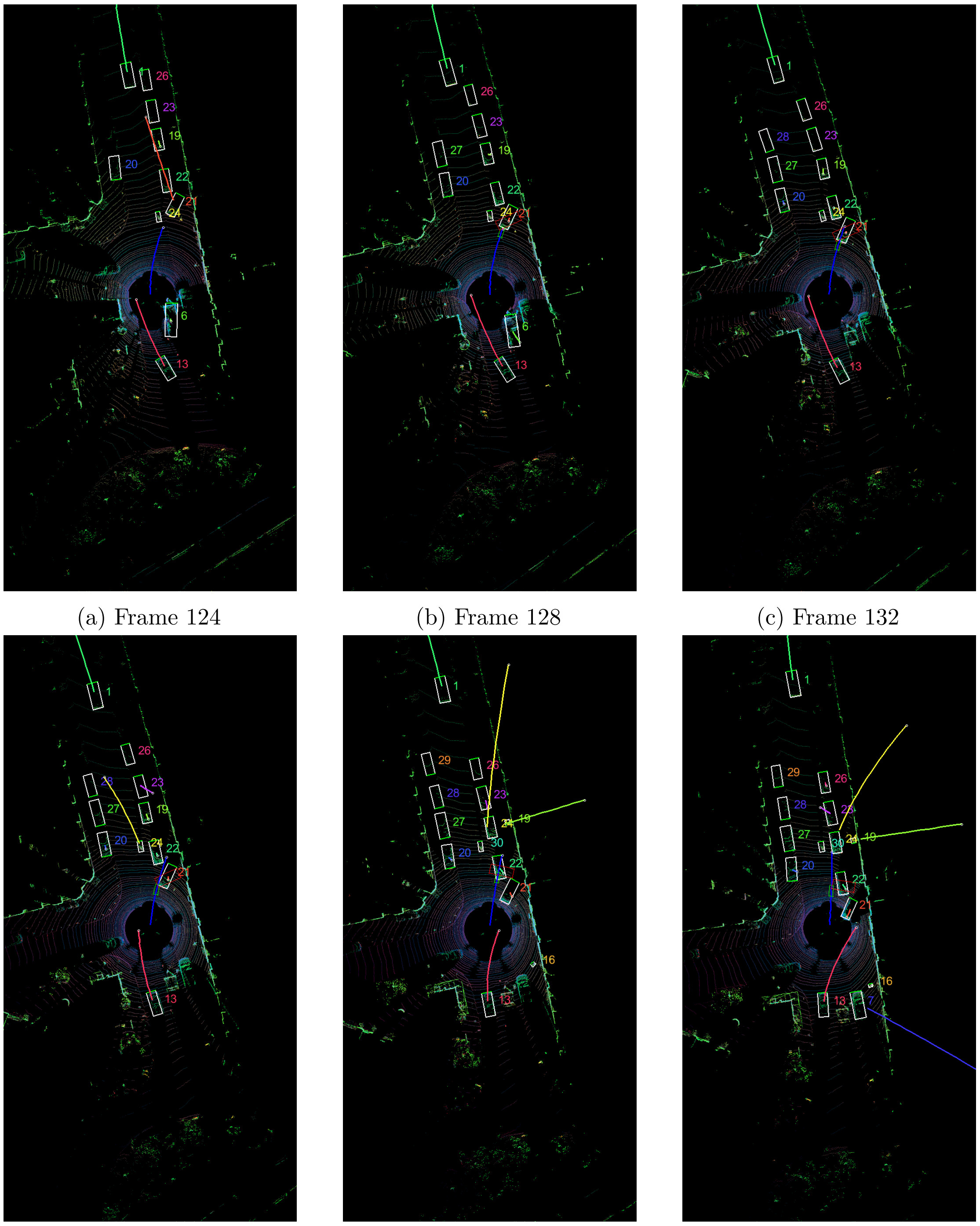
Figure 3.4: The information the app displays when the pipeline is running.



(a) Frame 12 (b) Frame 16 (c) Frame 20

(d) Frame 24 (e) Frame 28 (f) Frame 32

Figure 3.5: Bird's-eye-view map visualization of frames 12 to 32 in steps of four of sequence "2011_09_26_drive_0005_sync" [20].



(a) Frame 124 (b) Frame 128 (c) Frame 132

(d) Frame 136 (e) Frame 140 (f) Frame 144

Figure 3.6: Bird's-eye-view map visualization of frames 128 to 144 in steps of four of sequence "2011_09_26_drive_0005_sync" [20].

CHAPTER 4

Evaluation & Results

This chapter will cover the average runtimes of each component, the average runtimes when the remote server is enabled, and the quality of the on-device trajectory predictor.

4.1 Runtimes

The runtimes were tested on a Galaxy A52s 5G and an iPhone SE 2020 with the "2011_09_26_drive_0005_sync" KITTI raw [20] sequence (154 point clouds). Since the Galaxy heats up after some computation time, the numbers after an initial run were used for each test. The iPhone behaved similarly, but at some point, the runtimes severely slowed down to even 20 seconds. Therefore, all the runtime tests for the object detector on the iPhone were conducted with only the first ten iterations of the sequence to keep the phone's temperatures manageable.

The runtimes for the object tracker, trajectory predictor, and collision detector can be viewed in Table 4.1. The first 19 frames were ignored for the trajectory predictor and the collision detector since no trajectories are calculated at those frames. Compared to the object detector, the other modules do not significantly impact the overall runtimes.

	Obj. tracker	**Traj. predictor**	**Coll. detector**
Galaxy A52s 5G	0.00225s	0.00818s	0.00005s
iPhone SE 2020	0.00414s	0.01531s	0.00003s

Table 4.1: Average runtimes of object tracker, trajectory predictor, and collision detector on the Galaxy A52s 5G and the iPhone SE 2020.

Figure 4.1 shows the object detector runtimes with the XNNPack delegate. This delegate provides the option to specify the number of threads used when running inference on a model. The evaluation covers average runtimes with the thread count ranging from one to six. On the Galaxy, above 4 threads, the object detector processing times start to slow down again. The iPhone's detector runtimes continue to drop at a diminishing rate as the number of threads increases.

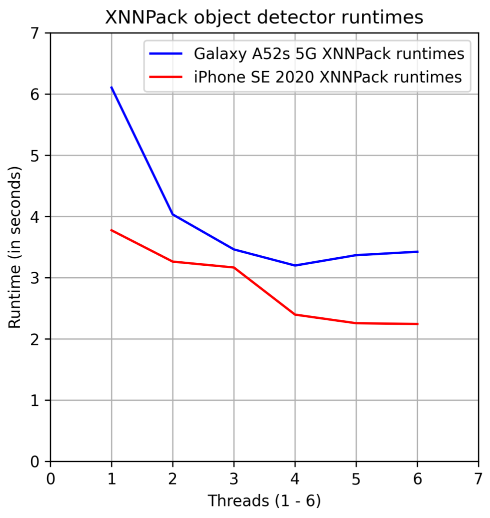


Figure 4.1: Average object detector runtimes with the XNNPack delegate configured to use one to six threads on the iPhone SE 2020 and Galaxy A52s 5G.

The object detector runtimes with various optimizations are shown in Table 4.2 for the Galaxy A52s 5G and Table 4.3 for the iPhone SE 2020. With the right configuration, the runtime can be more than halved. Using any of the two quantized models is not worth it since better performance can be achieved without them. Having the XNNPack delegate with 4 threads active on the Galaxy is best. On the iPhone, combining the Metal GPU delegate with the XNNPack 6 thread delegate is the fastest. This is because 54 out of 157 operations cannot be run on the GPU and need to be executed on the CPU instead.

	Standard	**F16 quant.**	**Dynamic range quant.**
No delegate	7.945s	7.750s	3.757s
XNNPack4T delegate	3.198s	3.367s	3.423s

Table 4.2: Average Galaxy A52s 5G object detector runtimes with various configurations.

	Standard	**F16 quant.**	**Dynamic range quant.**
No delegate	4.009s	6.320s	1.966s
XNNPack6T delegate	2.243s	2.174s	1.979s
Metal GPU delegate	2.034s	2.469s	1.627s
Metal&XNNPack6T d.	1.625s	1.729s	1.823s

Table 4.3: Average iPhone SE 2020 object detector runtimes with various configurations.

Before and after evaluating the remote pipeline's runtimes, a network speed test from the smartphone to the server was executed. Averaging the numbers yields the following results:

- Download speed: 10.6 Mbps (megabits per second)
- Upload speed: 10.15 Mbps
- Ping: 6.5 ms (milliseconds)

The runtime tests for the server pipeline using the Python implementation were conducted on a desktop computer using an AMD Ryzen 5 2600 CPU (no GPU). On average, the processing time of a request took 0.340 seconds. From the phone's perspective, an iteration lasted an average of 1.649 seconds. This results in an average client-server overhead of 1.309 seconds. Most of the overhead stems from sending the point cloud data from the client to the server, which is just below two megabytes in size. The information sent between the client and the server is explained in more detail in Section 3.5.

The average measured runtimes for each module on the server Python implementation are 0.084 seconds for the object detector, 0.003 seconds for the object tracker, 0.154 seconds for the trajectory predictor, and 0.003 for the collision detector. In this case, the trajectory predictor takes the longest, since it uses a much larger model than the object detector. The recorded performance for on-device processing and remote delegation are compared in Table 4.4. The collision detector is slower on the server since it does a more thorough check than the mobile implementation as discussed in Section 3.5. Note that the remote total in Table 4.4 does not exactly match the previously mentioned total from the phones' perspective since some processing before, after, and in between the pipeline steps is necessary.

	iPhone SE 2020	**Remote computer**
Object detector	1.625s	0.084s
Object tracker	0.00414s	0.003s
Trajectory predictor	0.01531s	0.154s
Collision detector	0.00003s	0.003s
Transfer overhead	0.0s	1.309s
Total	1.64448s	1.553s

Table 4.4: Local processing on iPhone SE 2020 with the best detector runtime vs. remote processing.

4.2 Trajectory Prediction Quality

Figure 4.2 displays the average error of the trajectory predictor using squared and cubed regression. Furthermore, the plot also shows the error of the custom-trained ESP [12] trajectory predictor, which ran on a Desktop computer. The errors were calculated by measuring the distance of a prediction to the actual detected position at each future frame for each tracked object. The average error over all other detected road users is meant by "other error" in the graph. The trajectory predictor was evaluated on five KITTI raw [20] sequences:

- "2011_09_26_drive_0001_sync" (city, 108 iterations)
- "2011_09_26_drive_0005_sync" (city, 154 iterations)
- "2011_09_26_drive_0020_sync" (residential, 86 iterations)
- "2011_09_26_drive_0046_sync" (residential, 125 iterations)
- "2011_09_28_drive_0038_sync" (campus, 110 iterations)

As expected, as the future frames increase, the error also increases. The squared prediction performs much better on average regarding both the ego vehicle and the other objects. When observing the visualizations, one can sometimes observe the trajectories jumping uncontrollably while the regression is cubed. Surprisingly, the neural network performed worse than the squared regression.

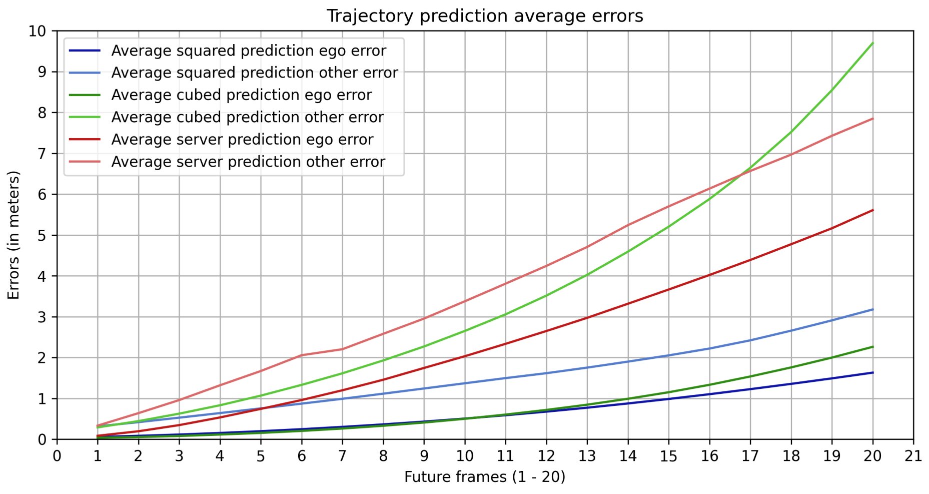


Figure 4.2: Average error of ego vehicle and other vehicles trajectory prediction using squared and cubed regression.

4.3 Performance of Latest Mobile Phones

The iPhone SE 2020, used for iOS testing, is powered by the A13 Bionic chip [21]. The most powerful iPhone available is the iPhone 15 Pro equipped with the A17 Pro chip

[22]. While the A13 bionic can perform 5 TOPS (trillion operations per second) with the neural engine, the A17 Pro is capable of nearly 35 TOPS. Theoretically, the newer chip can increase performance by up to seven times. Leveraging this performance requires implementing the application natively.

The Samsung Galaxy S24 is powered by the Snapdragon 8 Gen 3 Mobile Platform [23]. No details about the TOPS it can perform were found. The Android test device (Galaxy A52s 5G) uses the Snapdragon 778G 5G Mobile Platform. Compared to the Snapdragon 778G 5G, the Snapdragon 8 Gen 3 has a bit more than twice as powerful CPU and its GPU is six to seven times faster according to benchmarks [24, 25].

CHAPTER 5 Future Work & Conclusion

All in all, the mobile pipeline still requires optimizations to make it work properly. For the pipeline to be able to be used in a real-world scenario, an iteration needs to be much quicker. As we have seen in Chapter 4, the object detector is the bottleneck in this implementation. At best, it takes 1.625 seconds to detect objects from a point cloud, which is far too long to be viable in real-time. The rest of the modules' runtime is approximately at most 0.02 seconds together, which is acceptable.

The on-device trajectory predictor only uses the past positions of detected objects to calculate a trajectory. To reduce the error of the trajectory predictor, a neural network that can make use of the environment could be employed. The Python implementation on the server uses a custom-trained ESP model [12] to do this. Interestingly, the model on the server performs worse than the squared regression.

The current implementation runs inference on the SFA3D model [3] twice for object detection, once for the front and once for the back. If this was avoided, the object detector would be twice as fast. To achieve this, one could scale the front and back points so that they fit into the input map together. This adjustment might lead to a loss in precision, but it may also help detect other road users to the immediate right or left of the ego vehicle better.

Using the NCNN [26], ONNX [27], or Core ML [28] frameworks may be faster and more optimized on mobile devices than TF Lite. NCNN even allows for the leveraging of the GPU on supported Android and iOS devices. Unfortunately, these options do not officially support Flutter, so the pipeline would need to be reimplemented natively for Android and iOS.

The client-server overhead during remote processing is too large for a real-time scenario. Besides using a faster network speed, the point cloud data could be compressed via a smart algorithm before sending it to reduce the delay.

Once the runtime performances have been sufficiently optimized, the pipeline can leverage real-time data directly from the phone via a connected or built-in LiDAR sensor and warn the cyclist of a potentially hazardous situation.

Overview of Generative AI Tools Used

No AI tool were used to create this work.

Bibliography

[1] Flutter, "Flutter - Build apps for any screen," accessed: 2024-08-09. [Online]. Available: https://flutter.dev

[2] S. Pointner, "Cycle Safely - A collision prediction system," *TU Wien*, To appear.

[3] N. M. Dung, "Super-Fast-Accurate-3D-Object-Detection-PyTorch," 2020, accessed: 2024-08-09. [Online]. Available: https://github.com/maudzung/Super-Fast-Accurate-3D-Object-Detection

[4] P. Li, H. Zhao, P. Liu, and F. Cao, "RTM3D: Real-time Monocular 3D Detection from Object Keypoints for Autonomous Driving," in *European Conference on Computer Vision*, 2020.

[5] L.-H. Wen and K.-H. Jo, "Fast and Accurate 3D Object Detection for Lidar-Camera-Based Autonomous Vehicles Using One Shared Voxel-Based Backbone," *IEEE Access*, 2021.

[6] J. Wang, M. Zhu, B. Wang, D. Sun, H. Wei, C. Liu, and H. Nie, "KDA3D: Key-Point Densification and Multi-Attention Guidance for 3D Object Detection," *Remote Sensing*, 2020.

[7] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft, "Simple online and realtime tracking," in *2016 IEEE International Conference on Image Processing (ICIP)*, 2016.

[8] X. Weng, J. Wang, D. Held, and K. Kitani, "3D Multi-Object Tracking: A Baseline and New Evaluation Metrics," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[9] N. Wojke, A. Bewley, and D. Paulus, "Simple online and realtime tracking with a deep association metric," in *2017 IEEE International Conference on Image Processing (ICIP)*, 2017.

[10] H. Wu, W. Han, C. Wen, X. Li, and C. Wang, "3D Multi-Object Tracking in Point Clouds Based on Prediction Confidence-Guided Data Association," *IEEE Transactions on Intelligent Transportation Systems*, 2022.

[11] A. Singh, “Trajectory-Prediction with Vision: A Survey,” *2023 IEEE/CVF International Conference on Computer Vision Workshops (ICCVW)*, 2023.

[12] N. Rhinehart, R. T. McAllister, K. Kitani, and S. Levine, “PRECOG: PREdiction Conditioned on Goals in Visual Multi-Agent Settings,” *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[13] C. Ericson, *Real-Time Collision Detection.* Morgan Kaufmann Publishers, 2004.

[14] TensorFlow, “tflite_flutter,” accessed: 2024-08-09. [Online]. Available: https://pub.dev/packages/tflite_flutter

[15] A. Lutsenko, “Nobuco - Pytorch to Keras/Tensorflow/TFLite conversion made intuitive,” 2023, accessed: 2024-08-09. [Online]. Available: https://github.com/AlexanderLutsenko/nobuco

[16] Hailanyi, “3D Multi-Object Tracker - A project for 3D multi-object tracking,” 2021, accessed: 2024-08-09. [Online]. Available: https://github.com/hailanyi/3D-Multi-Object-Tracker

[17] Google, “Protocol Buffers,” 2008, accessed: 2024-08-09. [Online]. Available: https://protobuf.dev

[18] Tensorflow, “Post-training quantization,” accessed: 2024-08-09. [Online]. Available: https://www.tensorflow.org/lite/performance/post_training_quantization

[19] Google, “Isolates,” 2024, accessed: 2024-08-09. [Online]. Available: https://dart.dev/language/isolates

[20] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, “Vision meets Robotics: The KITTI Dataset,” *International Journal of Robotics Research (IJRR)*, 2013.

[21] Apple, “iPhone SE: A powerful new smartphone in a popular design,” 2020, accessed: 2024-08-09. [Online]. Available: https://www.apple.com/newsroom/2020/04/iphone-se-a-powerful-new-smartphone-in-a-popular-design

[22] ——, “iPhone 15 Pro,” 2024, accessed: 2024-08-09. [Online]. Available: https://www.apple.com/iphone-15-pro

[23] Samsung, “How the Galaxy S24 compares,” 2024, accessed: 2024-08-09. [Online]. Available: https://www.samsung.com/us/smartphones/galaxy-s24/compare/?device-1=samsung-galaxy-s24&device-2=samsung-galaxy-s24%2B&device-3=samsung-galaxy-s24

[24] Nanoreview, “Qualcomm Snapdragon 778G,” 2023, accessed: 2024-08-09. [Online]. Available: https://nanoreview.net/en/soc/qualcomm-snapdragon-778g-5g

[25] ——, “Qualcomm Snapdragon 8 Gen 3,” 2023, accessed: 2024-08-09. [Online]. Available: https://nanoreview.net/en/soc/qualcomm-snapdragon-8-gen-3

[26] H. Ni, “NCNN - High-performance neural network inference computing framework,” 2017, accessed: 2024-08-09. [Online]. Available: https://github.com/Tencent/ncnn

[27] J. Bai, F. Lu, K. Zhang *et al.*, “ONNX - Open Neural Network Exchange,” 2019, accessed: 2024-08-09. [Online]. Available: https://onnx.ai

[28] Apple, “Core ML - Integrate machine learning models into your app.” 2017, accessed: 2024-08-09. [Online]. Available: https://developer.apple.com/documentation/coreml