

Trajectory Prediction and Simulation for Improved Control of Connected Autonomous Vehicles

Avikam Chauhan



Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-77

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-77.html>

May 10, 2024

Copyright © 2024, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Trajectory Prediction and Simulation for Improved
Control of Connected Autonomous Vehicles

by

Avikam Chauhan

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Alexandre M. Bayen, Chair
Professor Maria Laura Delle Monache

Spring 2024

Trajectory Prediction and Simulation for Improved Control of Connected Autonomous Vehicles

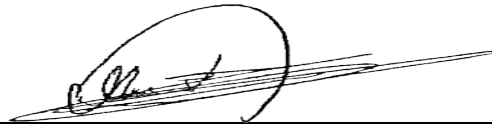
by Avikam Chauhan

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:



Professor Alexandre Bayen
Research Advisor

5/6/2024

(Date)

* * * * *



Professor Maria Laura Delle Monache
Second Reader

5/8/2024

(Date)

Trajectory Prediction and Simulation for Improved
Control of Connected Autonomous Vehicles

Copyright 2024
by
Avikam Chauhan

Abstract

Trajectory Prediction and Simulation for Improved
Control of Connected Autonomous Vehicles

by

Avikam Chauhan

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alexandre M. Bayen, Chair

Connected autonomous vehicles (CAVs) are uniquely positioned to serve as mobile actuators of human-driven traffic, smoothing out or even completely eliminating phantom traffic jams on highways — this can significantly improve safety, traffic flow, and fuel efficiency for all vehicles on the highway. This work builds upon existing reinforcement learning (RL) based controller algorithms for CAVs, and introduces a Transformer-based machine learning (ML) model designed to effectively predict the future trajectories of human-driven vehicles. Additionally, this work presents a GPU-optimized parallel simulator to test CAV controller performance by simulating large numbers of mixed autonomy traffic trajectories in the highway domain.

Contents

Contents	i
1 Introduction	1
1.1 Background	1
1.2 Related Work	2
1.3 Contributions Of This Work	4
2 Vehicle Trajectory Prediction Model	5
2.1 Dataset Formats and Pre-Processing	5
2.2 Model Architecture and Training	6
2.3 Experiments and Results	7
2.4 Conclusion	10
3 Parallelized Vehicle Trajectory Simulation	11
3.1 Dataset Formats and Pre-Processing	11
3.2 Simulator Design and Implementation	12
3.3 Experiments and Results	17
3.4 Conclusion	19
Bibliography	20

Acknowledgments

This thesis has been a culmination of dedicated research and learning, made possible by the invaluable support and guidance of many. First and foremost, I would like to express my deepest gratitude to my research advisor, Professor Alexandre Bayen. I sincerely appreciate the opportunity to be part of your lab and to engage in such innovative work. The knowledge and experience I have gained over the past years would not have been possible without your unwavering guidance and support. I also extend my heartfelt thanks to Professor Maria Laura Delle Monache for her invaluable assistance in reviewing my report and supporting my application for the research fellowship. Additionally, I would like to acknowledge my colleagues in the Bayen lab; their willingness to share their expertise greatly facilitated my growth and understanding of our projects.

I am immensely grateful to my family for their constant love and support, without which this achievement would not have been possible. Their encouragement to always keep learning and growing, as well as their inspiration to take on challenging endeavors, has been truly invaluable. Last, but not least, I want to thank my friends for making my four-year journey at UC Berkeley truly unforgettable.

This work was supported in part by the United States Department of Transportation, through the Dwight David Eisenhower Transportation Fellowship Program, under Grant Number 693JJ32445046. The views expressed herein do not necessarily represent the views of the U.S. Department of Transportation or the United States Government.

Chapter 1

Introduction

1.1 Background

Phantom traffic jams are spontaneous slowdowns of vehicles that arise for no apparent reason — rather they are the result of inefficiencies in human driving. When one driver lightly taps the brakes, it can send a chain reaction upstream that is so unstable that it causes stop-and-go waves, forcing many vehicles to come to a halt. Because drivers have to quickly react to sudden changes in traffic speed, there is a higher risk of collision, potentially causing injuries and deaths. Furthermore, the cycles of accelerating and braking (and also idling, when vehicles come to a full stop) result in increased energy consumption. As the number of vehicles on highways and freeways continues to increase, this problem becomes increasingly important to hundreds of millions of people all over the world.

However, modern advances in lane keeping assist systems (LKAS) and adaptive cruise control (ACC) technologies have transformed our highways into a mixed autonomy regime where autonomous and human drivers operate together. When autonomous vehicles (AVs) share information about the traffic around them with other AVs, they form a system of connected autonomous vehicles (CAVs) that can serve as mobile traffic actuators that can optimize traffic flow and safety [1]. In past experiments, automated speed controllers have been shown to eliminate phantom traffic jams for vehicles traveling in a loop.

With the CIRCLES consortium, we have taken this to the next level by training deep reinforcement learning (RL) based controllers to optimize the flow of highway traffic by controlling the ACC speed and gap settings for a system of CAVs. Ultimately, these controllers can reduce phantom traffic jams, improve the throughput of highways, and increase the fuel efficiency of all vehicles on the road significantly. Paired with real-time traffic data and custom kernel smoothing algorithms, these deep RL controllers were deployed to 100 CAVs and tested in real-world highway traffic on I-24 near Nashville, TN, in November 2022. [2]. This first-of-its-kind experiment produced a massive dataset that can be extended for further AV research [3].

To train an RL agent to smooth flows of traffic, the problem is formulated as a partially-observable Markov decision process (POMDP), where the agent is the AV, and its action space includes the control of either the vehicle’s ACC speed and gap settings, or the vehicle’s acceleration directly. The observation space can include data from onboard sensors (e.g. leader speed, leader gap) as well as aggregated traffic flow information (using INRIX). The reward function is designed to incentivize fuel efficiency, smoother flow of traffic, and safety.

Although these models have a probabilistic component that attempts to predict the likelihood of another vehicle cutting in front of the AV (based on the gap and relative speed between the AV and the leader vehicle), they do not utilize local sensing of neighboring vehicles, and thus cannot accurately predict lane changes or adjust their speed and gap accordingly. Furthermore, the RL agent training environment only consists of single lanes and fixed size arrays of vehicle platoons, and as such, there is scope for more sophisticated training and simulation environments where vehicles make lane changes like human drivers do in the real-world. This could also reduce the “sim2real” gap in performance when deploying such RL controllers to real-world experiments.

However, to be able to effectively build upon this work and design next-generation controllers that can optimize traffic even further, these improved controller models must be paired with a performant and scalable simulator designed to handle these large volumes of data (i.e. hundreds of thousands of trajectories per hour). Ultimately, the development of perception, planning, and control software for autonomous vehicles (AVs) is a highly data-intensive and computationally demanding process, and it requires fast, scalable virtual simulation software to collect relevant data, train models (with software-in-the-loop feedback), and run experiments to quantify controller performance. Because it is often infeasible to collect such large amounts of multi-modal data (e.g. video, sensors, traffic, road maps) with physical vehicles and unsafe to directly test models in real-world scenarios, AV simulation is crucial to the development of next-generation control algorithms and software.

1.2 Related Work

Vehicle Trajectory Prediction Models

For effective control of autonomous vehicles, the prediction of neighboring vehicle trajectories is a highly relevant and important problem. Past works have demonstrated the capabilities of deep learning architectures such as long short-term memory (LSTM) and recurrent neural networks (RNNs) for highway traffic, and Transformers for city traffic.

One such example of a vehicle trajectory prediction model is the Wayformer, developed by Waymo. The authors present a Transformer-based architecture to predict vehicle trajectories in city traffic, and their model is tightly coupled with the associated Waymo Open Motion Dataset. At its core, it uses Transformer encoder and decoder blocks, in a variety of arrangements, that allow it to take in multimodal inputs (e.g. traffic light states, roadgraph

information, vehicle trajectory histories, and agent interactions) and predict the future positions and velocities of vehicles within each example [4]. However, as this model architecture is designed specifically for the Waymo dataset (it relies on roadgraph information as an input) and the dynamics of city traffic are different from that of highway traffic, it is not as directly applicable for this specific domain.

Another related example utilizes LSTM models to predict future trajectories of vehicles in dense traffic on highways, using an end-to-end prediction method (i.e. the model does not extract features that could identify a particular maneuver, rather it directly predicts the future vehicle positions and velocities) [5]. The authors highlight that their model architecture includes spatio-temporal information of neighboring vehicles to allow for better predictions, yet the trajectories tend to diverge as the prediction horizon gets larger [5].

Parallelized Vehicle Trajectory Simulators

Although there have been prior works in developing parallel, optimized simulators for autonomous vehicles, they generally have not been designed for this scale of simulation (either focusing on smaller-scale city traffic, or lacking the spatial sensing that is needed for effective control of AVs in highway traffic).

One such example of a parallelized vehicle simulator is Webots.HPC, which is a toolkit for running Webots-SUMO simulations at scale on high-performance computing (HPC) resources. The authors describe their implementation of batching large numbers of simulations at the same time (tested with 6 compute nodes on the Palmetto cluster, and 8 simulation instances per node), which allows them to generate large datasets that contain rare edge cases that are dangerous to recreate in real-world traffic [6]. However, this work is focused on generating comprehensive datasets for AV driving scenarios, and does not focus on optimizing the parallel simulation of individual large-scale experiments.

Another example of a state-of-the-art AV simulator is the Waymax simulator, which offers a parallelized, differentiable AV simulator that makes it easier to train ML models. This simulator provides the capability to simulate non-vehicle agents (e.g. pedestrians, bicyclists) which are important for AVs that drive in city traffic [7]. This simulator is written in JAX, which allows for efficient simulation on GPUs or TPUs, but this simulator is tightly coupled to the associated Waymo Open Motion Data dataset, and it is restricted to city traffic with only 128 individual agents being simulated at the same time [7].

Another related work, titled “Scaling Is All You Need”, builds upon the Waymax simulator, and allows for 64-256 individual agents to be simulated simultaneously [8]. The authors use JAX vector operations to perform batched inference on observation data and batched updates to the state representation, and present a method of preparing and pre-loading batch data to the GPU memory to maximize simulation throughput [8]. However, it still does not provide a large enough scale to simulate the I-24 MOTION dataset that is the subject of this work, and it is also restricted to city traffic with the same dataset as the Waymax simulator.

Lastly, LPSim is a large-scale parallel regional traffic simulation that enables microsimulation analysis of network traffic assignments for both on-road vehicles and aircraft. The simulator is optimized to run on multiple GPUs, and can simulate millions of trips across a network of hundreds of thousands of nodes and edges in minutes [9]. The fundamental state representation for this simulator is created by spatially segmenting the edges in the network (each of which is an element in a large array), and storing the speed of each vehicle at its corresponding physical location [9]. This simulator is designed to generate new human-driven trajectories between origin and destination pairs, and it utilizes the intelligent driver model (IDM) to control vehicle dynamics, but it doesn't allow for the re-simulation of previously collected trajectories or the insertion of CAVs into the flow.

1.3 Contributions Of This Work

Given the limitations and constraints of existing vehicle trajectory prediction models and large-scale parallelized simulators, this work presents a Transformer-based ML model to predict the future trajectories of human-driven vehicles in highway traffic, along with a GPU-optimized parallel simulator co-designed to test CAV controller performance by simulating large numbers of mixed autonomy traffic trajectories in the highway domain.

The trajectory prediction model presented in Chapter 2 is a hybrid approach, combining a Transformer encoder architecture (similar to the Wayformer model) with a feature-based prediction model (similar to the maneuver-based model described in the LSTM paper). The architecture of the parallelized trajectory simulator, presented in Chapter 3, builds upon the spatial segmentation methodology from LPSim, and allows for the accurate re-simulation of pre-recorded trajectories and the introduction of custom CAV controllers (that rely on local sensing and aggregate downstream traffic data) into the flow.

Chapter 2

Vehicle Trajectory Prediction Model

2.1 Dataset Formats and Pre-Processing

I-24 MOTION Dataset

The dataset being used for this work is the I-24 MOTION dataset, which is collected using a system of 294 high-resolution cameras mounted on a 4.2-mile stretch of Interstate 24 in southeast Nashville, TN. Video recordings are captured from the cameras, and then computer vision algorithms are used to identify, classify, and track vehicles trajectories at 25 *Hz* on this segment of the highway. The dataset consists of 4-hour recordings during morning rush-hour traffic (5 AM to 9 AM) across a span of 10 days. At this time of day, vehicles heading westbound (into Nashville) generally experience stop-and-go waves, while vehicles heading eastbound (away from Nashville) are in free-flow.

The I24-MOTION dataset is formatted as JSON objects, which provides convenience and flexibility in data access, but at the cost of efficiency and performance. For each day, there are hundreds of thousands of trajectories (and each JSON file could be anywhere from 10 to 20 GB). The dataset needs to be pre-processed and converted into a different format that is optimized specifically for model training.

Trajectory Prediction Dataset

Because the trajectory prediction model is implemented using *TensorFlow*, the JSON objects within the I-24 MOTION dataset must be converted into *TFRecord* objects. The trajectory is broken up into 2 segments: the past (25 samples, over a 1 second interval) and the future (the remaining portion of the initial trajectory). Each sample also includes basic vehicle information, such as the length, width, height, and class of vehicle (sedan, midsize, van, pickup, semi, or truck).

Since the trajectories are of varying lengths, it is important to appropriately select and sort the trajectories for efficient model training and optimal model performance. The trajectories are filtered by trajectory length, and only the trajectories that are within 1 standard

deviation of the mean trajectory length are used for training and testing the model. Still, it is inefficient to naively sample trajectories from the dataset and pad all samples to the length of the longest trajectory, and it is ineffective to truncate trajectories to the length of the shortest trajectory. To resolve this issue, the dataset is chunked into trajectories that are similar lengths (i.e. bins of size 100), and samples are padded to the next multiple of 100.

The *TfRecord* data structure is defined as follows:

```
features = {
    '_id': tf.io.FixedLenFeature([1], tf.string),
    'timestamps': tf.io.RaggedFeature(tf.float32),
    'x_positions': tf.io.RaggedFeature(tf.float32),
    'y_positions': tf.io.RaggedFeature(tf.float32),
    'length': tf.io.FixedLenFeature([1], tf.float32),
    'width': tf.io.FixedLenFeature([1], tf.float32),
    'height': tf.io.FixedLenFeature([1], tf.float32),
    'coarse_vehicle_class': tf.io.FixedLenFeature([1], tf.int64),
    'direction': tf.io.FixedLenFeature([1], tf.int64),
    'first_timestamp': tf.io.FixedLenFeature([1], tf.float32),
    'last_timestamp': tf.io.FixedLenFeature([1], tf.float32),
    'starting_x': tf.io.FixedLenFeature([1], tf.float32),
    'ending_x': tf.io.FixedLenFeature([1], tf.float32)
}
```

2.2 Model Architecture and Training

The trajectory prediction model is designed to allow an AV to effectively predict the future trajectory of human-driven vehicles around it. The model architecture includes a Transformer encoder block that takes in the vehicle’s past states and physical characteristics, and allows it to learn key hidden features (about the vehicle dynamics and driver behavior) that allow for more accurate predictions of future states. The hidden features are combined with the current state and fed into a recurrent neural network (RNN) block, which outputs the predicted state of the vehicle at the next timestep.

More specifically, the model architecture is as follows:

- Input: $(B \times H \times 6)$
- Positional Embedding: $(B \times H \times 6) \rightarrow (B \times H \times 6)$
- Projection Layer (Dense): $(B \times H \times 6) \rightarrow (B \times H \times F_P)$
- Transformer Encoder (6 layers, 8 attention heads): $(B \times H \times F_P) \rightarrow (B \times H \times F_P)$

- Dense Layer: $(B \times \{H \cdot F_P\}) \rightarrow (B \times F_H)$
- Repeat + Concatenate: $(B \times O \times F_H) + (B \times O \times F_P) \rightarrow (B \times O \times \{F_H + F_P\})$
- RNN Intermediate Layer (Dense): $(B \times O \times \{F_H + F_P\}) \rightarrow (B \times O \times R)$
- RNN Output Layer (Dense): $(B \times O \times R) \rightarrow (B \times O \times 2)$
- Output: $(B \times O \times 2)$

The model was trained with a batch size $B = 32$, history $H = 25$ samples, projected feature dimension $F_P = 256$, hidden feature dimension $F_H = 512$, and RNN intermediate dimension $R = 128$. O is the length of the output sequence, which depends on the specific trajectory that inference is being performed on.

The model is trained with teacher forcing, where the RNN input is the ground truth vehicle state at the current timestep, and the loss function (the predicted future vehicle state at the next timestep) is defined as the mean squared error (MSE) from the ground truth vehicle state at the next timestep. The purpose of training with teacher forcing is to avoid the problem of vanishing or exploding gradients that can occur over such long sequences.

During inference, the RNN input will be the observed vehicle state at the current timestep, which mimics a live prediction of the vehicle's trajectory as new sensor data is received. That being said, it is also possible to use the RNN output at a given timestep as the RNN input for the next timestep (as this would allow for the generation of predictions further ahead into the future). However, as the time horizon increases, the predictions may diverge from the ground truth because the range of possible future coordinates gets wider and routing information (e.g. the vehicle's destination) are not known.

2.3 Experiments and Results

To determine the performance of the model, it was run on the test set, which it had not seen during the training and validation steps. Despite not having seen these examples before, the model is able to generally predict the shapes of the trajectories quite successfully. The diagram below depicts the predicted and actual trajectories for 2 different examples from the test set, where the respective vehicle remains in the same lane for the duration of the trajectory.

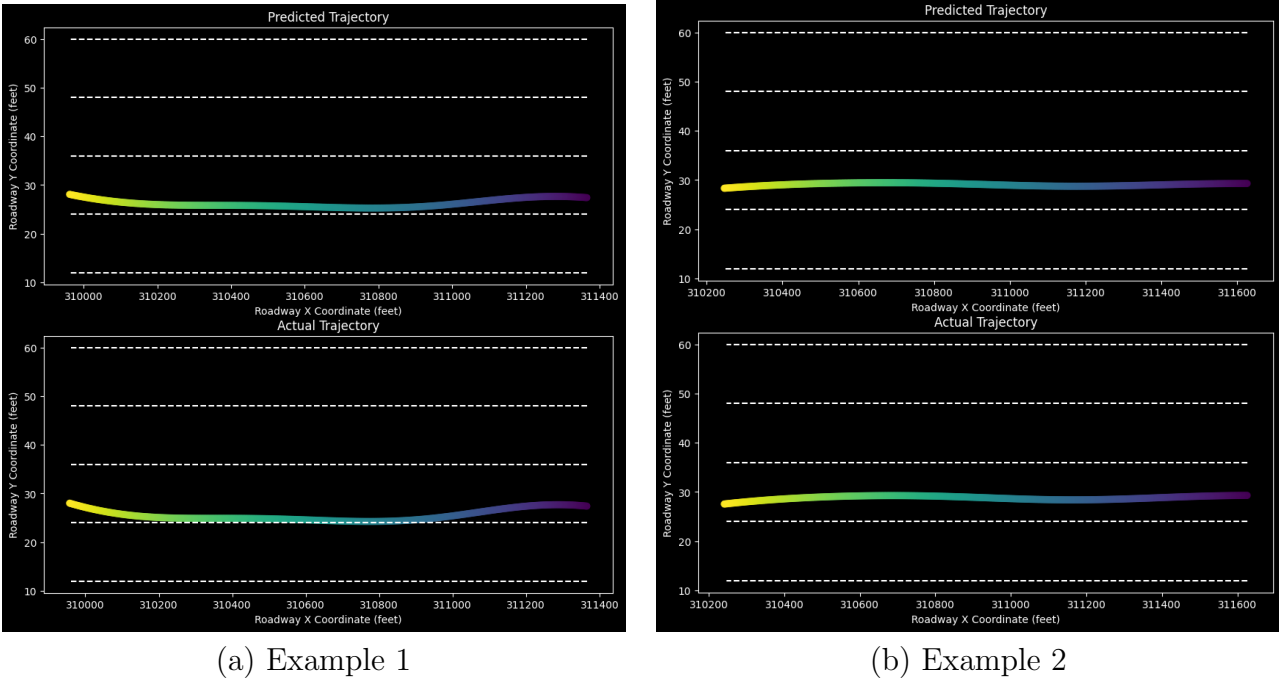


Figure 2.1: Examples of predicted and ground-truth trajectories from the test dataset, where the vehicles do not change lanes during the trajectory.

The model was also evaluated on trajectories where the selected vehicle changed lanes one or more times during the course of the trajectory, and this is especially useful because the AV can use these predictions to preemptively avoid accelerating or even slow down if a nearby vehicle is likely to change lanes in front of the AV. The diagram below depicts the predicted and actual trajectories for 2 such examples in the test set.

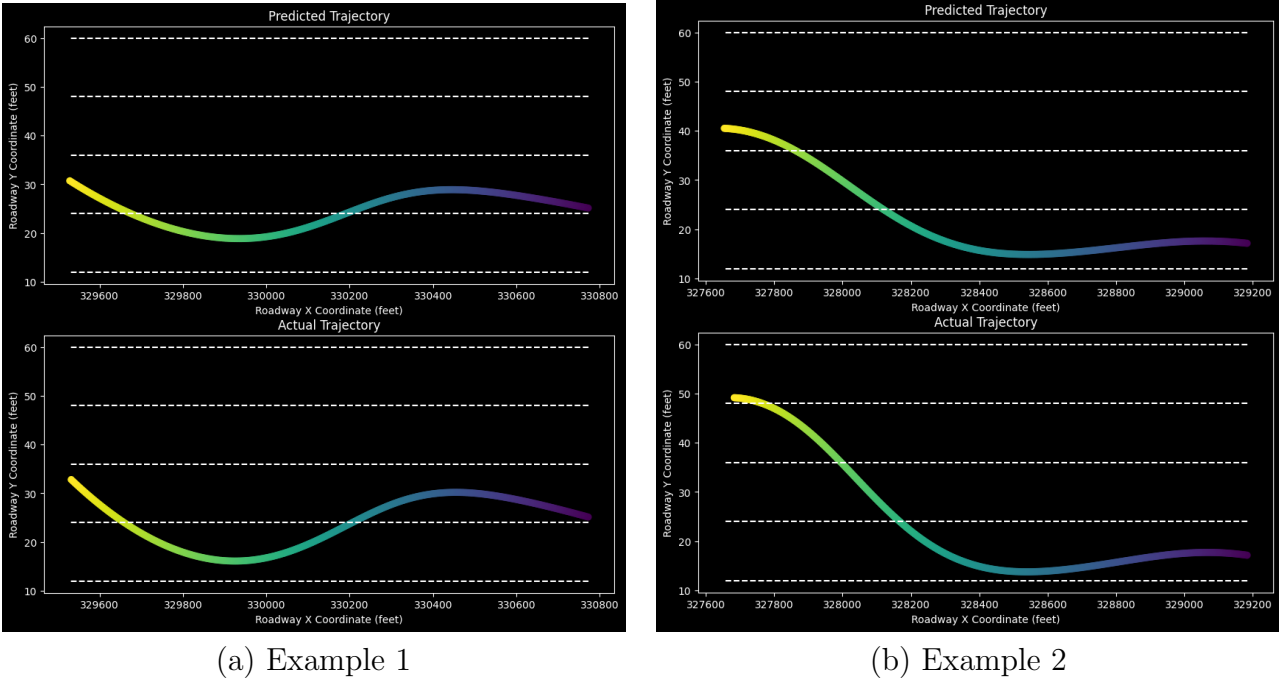


Figure 2.2: Examples of predicted and ground-truth trajectories from the test dataset, where the vehicles change lanes during the trajectory.

Although the shape of the predicted trajectory captures the trend of the actual trajectory, it is not a perfect prediction due to differences in the y -coordinates in positions at the extreme ends of the trajectory. Nevertheless, this model provides added predictive capabilities to AV controllers, which is expected to improve the performance with respect to flow smoothing and fuel efficiency.

To understand the overall model performance across a wide variety of trajectories, the model was run on a test set of over 65,000 trajectories, and the average distance between the predicted coordinates and actual coordinates for each trajectory was calculated and plotted in a histogram, as depicted below.

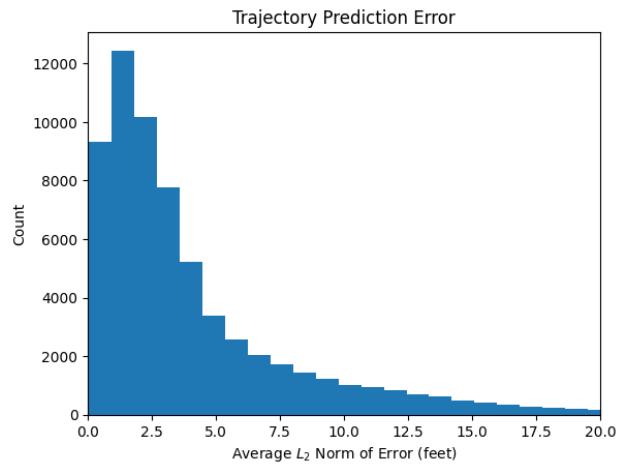


Figure 2.3: The average L_2 norm of the error in predicted vehicle coordinates, across a test set of over 65,000 trajectories.

2.4 Conclusion

The Transformer-based vehicle trajectory prediction model presented in this section is a powerful and capable model that can allow a CAV to forecast the future positions of neighboring human-driven vehicles around it. This can provide additional boosts in the flow-smoothing and energy-saving capabilities of RL-based controllers, as they allow for preemptive adjustments for vehicles that will change lanes into or out of the lane the AV is currently in. This means that AVs can avoid closing gaps that are expected to be filled by another vehicle, and will also accelerate more cautiously when the leader vehicle changes lanes and leaves a gap in front of the AV.

Chapter 3

Parallelized Vehicle Trajectory Simulation

3.1 Dataset Formats and Pre-Processing

The dataset being used for this work is the I-24 MOTION dataset, which is the same dataset used to train the human-driven vehicle trajectory prediction model in the previous section. Just as before, the dataset needs to be pre-processed and converted into a different format that is specifically optimized for trajectory simulation.

Trajectory Simulation Dataset

To effectively parse and simulate the trajectories in these large files, the dataset must be pre-processed and converted into a format that is optimized for the simulator. The purpose of the simulator is to recreate human driver trajectories as accurately as possible, while also inserting CAVs in the flow that are running custom controller algorithms. As a result, the trajectory is not sampled from input data (and in fact, if that was the case, introducing CAVs into the flow would not make sense), but rather re-simulated with the key features extracted beforehand. To do this, the key features (e.g. the start time, start position, end time, end position, and the physical coordinates and directions of all lane changes) need to be extracted from each trajectory during the pre-processing step. As a result, the size of this processed data is much smaller than the original dataset.

The C++ data structure is defined as follows:

```
struct Trajectory {  
    unsigned int id;  
    double length;  
    double width;  
    double height;  
    int coarse_vehicle_class;
```

```
    int direction;  
    double first_timestamp;  
    double last_timestamp;  
    double starting_x;  
    double starting_velocity;  
    int starting_lane;  
    double ending_x;  
    std::vector<std::tuple<double, int>> lane_changes;  
};
```

3.2 Simulator Design and Implementation

The purpose of this component is to accelerate the large-scale simulation of AV trajectories in highway traffic, which operates with on the order of hundreds of thousands of trajectories per experiment. Because different segments of the highway are independent of each other, the process of querying local sensing data and updating vehicle speeds and positions can be parallelized spatially. Furthermore, this lends itself to a memory-efficient simulator state representation, which also allows for faster lookup of neighboring vehicle speeds and positions, as needed for evaluating the Intelligent Driver Model (IDM) algorithm and, in the future, performing inference with the vehicle trajectory prediction model (described in the previous section) as part of an improved RL controller for CAVs. Paired with a GPU-optimized kernel, this implementation can provide significant speedups over the baseline. But first, the concept must be implemented serially to test for simulation correctness, and to evaluate the improvements in performance by parallelizing the simulator on GPU hardware. The following sections describe the implementation of the serial and parallel algorithms.

Baseline Serial Algorithm

The baseline serial implementation was developed as a reference point against which the correctness and performance of the GPU-optimized parallel implementation could be compared. The algorithm, implemented in Python, is defined as follows:

Algorithm 1 Baseline Serial Trajectory Simulation Algorithm

```

active_vehicles ← [ ]
active_trajectories ← [ ]
t ← trajectories[0].first_timestamp
while len(active_trajectories) > 0 or len(trajectories) > 0 do
  indices_to_remove ← [ ]
  for i ← 1 to len(active_trajectories) do
    curr_vehicle ← active_vehicles[i]
    curr_traj ← active_trajectories[i]
    if t ≥ active_trajectories[i].last_timestamp then
      indices_to_remove.append(i)
    else
      leader_vehicle ← GET_LEADER_VEHICLE(curr_vehicle)
      new_velocity ← INTELLIGENT_DRIVER_MODEL(curr_vehicle, leader_vehicle)
      curr_vehicle.velocity ← new_velocity
      if t ≥ curr_traj.lane_changes[0][0] then
        lane_change ← curr_traj.lane_changes.pop(0)
        curr_vehicle.lane ← lane_change[1]
      end if
    end if
  end for
  for i ← len(indices_to_remove) to 1 do
    idx ← indices_to_remove[i]
    active_vehicles.remove(idx)
    active_trajectories.remove(idx)
  end for
  for i ← 1 to len(trajectories) do
    if t ≥ trajectories[0].first_timestamp then
      new_traj ← trajectories.pop(0)
      new_vehicle ← Vehicle(new_traj)
      active_vehicles.append(new_vehicle)
      active_trajectories.append(new_traj)
    else
      break
    end if
  end for
end while

```

GPU-Optimized Parallel Algorithm

In the baseline serial implementation, the trajectories and vehicles were stored as arrays of objects, but this is not the optimal data structure (in terms of both lookup and update time). Inspired by the implementation of LP-Sim (described in Chapter 1), I developed the following optimized data structure and algorithm for simulating mixed-autonomy trajectories in the highway domain.

As depicted in the diagram below, the 4.2 mile long stretch of the I-24 highway can be divided up spatially into segments that are 10 feet long, such that only 1 vehicle can be in a segment at any given point in time. With this structure, 2D arrays can be used to store the exact location (represented as an offset within the segment, ranging from 0 to 9 feet) and speed (in feet per second) of all active vehicles in the simulation. Three arrays are needed (for positions, speeds, and IDs), and they are each of shape (2218, 4). Since the range of position values is only 0 – 9, and the range of speed values is 0 – 150, both values can be represented by unsigned chars (1 byte). The IDs are stored as unsigned integers (4 bytes). The space needed to store these arrays is around 53 KB, which is independent of the number of vehicles being simulated. To ensure correctness when querying the states of some vehicles and updating others simultaneously, there need to be copies of each array, and the current and next states will alternate each timestep to avoid unnecessary memory copy operations.

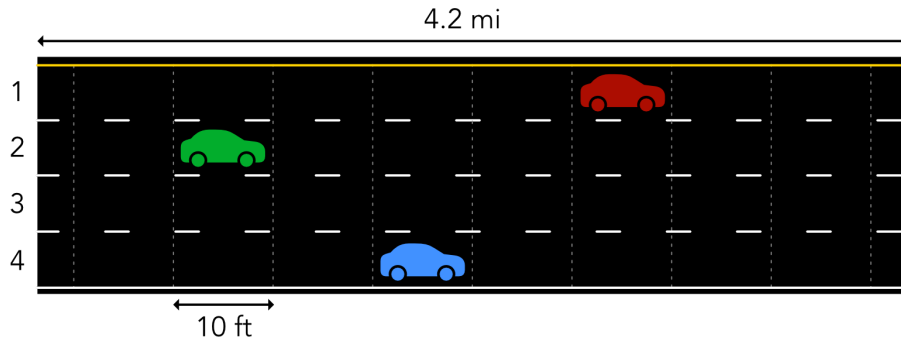


Figure 3.1: A diagram depicting the spatial segmentation of the 4.2 mile long stretch of I-24, into segments that are each 10 feet long. This structure is used for three 2D arrays that store the position, velocity, and unique ID of each active vehicle in the simulation.

Now, to be able to handle lane changes, a different representation must be used that allows GPU threads to quickly lookup the next lane change for any given vehicle. To achieve this, the structs containing lane change information are written into shared GPU memory by the CPU function that parses the JSON file when the simulator is initialized. The structs are then referenced in a hash table (where the key is the unique integer ID for each trajectory), which allows for constant time lookup of lane change parameters.

To handle the introduction of new vehicles into the simulator at various timesteps throughout the experiment, the CPU thread will check the current simulation time against the `first_timestamp` for trajectories that have not yet started simulation. If a vehicle needs to be added, the CPU thread will initialize the struct and copy it to shared GPU memory, and insert the vehicle’s initial state into the 3 simulation state arrays. Since the trajectories are already sorted by `first_timestamp` before the simulation is run, this is highly efficient. Furthermore, the struct that contains lane change information also stores the `ending_x` coordinate of the trajectory, and the GPU thread compares the current vehicle position with this `ending_x` coordinate to determine when to remove the vehicle from the simulator.

Formally, the CPU logic is defined as follows:

Algorithm 2 GPU-Optimized Parallel Simulator — CPU Logic

```

active_trajectory_id ← 0
d_pos_1, d_speed_1, d_pos_2, d_speed_2 ← cudaMalloc(num_vehicles)
d_id_1, d_id_2 ← cudaMalloc(num_vehicles × sizeof(unsigned int))
h_pos, h_speed ← new unsigned char[num_vehicles]
h_id ← new unsigned int[num_vehicles]
trajectories ← READ_TRAJECTORIES(filename)
h_lane_changes ← new LaneChange[trajectories.size()]
d_lane_changes ← cudaMalloc(trajectories.size() × sizeof(LaneChange))
current_time ← trajectories[0].first_timestamp
while current_time < simulation_end_time do
    active_trajectory_id ← INSERT_NEW_VEHICLES(current_time, active_trajectory_id,
        trajectories, h_pos, h_speed, h_id, h_lane_changes)
    d_pos_1, d_speed_1, d_id_1, d_lane_changes ← h_pos_1, h_speed_1, h_id_1, h_lane_changes
    d_pos_2, d_speed_2 ← cudaMemset(UCHAR_MAX, num_vehicles)
    d_id_2 ← cudaMemset(0, num_vehicles)
    GPU_KERNEL(d_pos_1, d_speed_1, d_id_1, d_pos_2, d_speed_2, d_id_2, d_lane_changes)
    cudaDeviceSynchronize()
    d_pos_1, d_pos_2 ← d_pos_2, d_pos_1
    d_speed_1, d_speed_2 ← d_speed_2, d_speed_1
    d_id_1, d_id_2 ← d_id_2, d_id_1
    h_pos_1, h_speed_1, h_id_1, h_lane_changes ← d_pos_1, d_speed_1, d_id_1, d_lane_changes
    UPDATE_LANE_CHANGE_INFO(active_trajectory_id, trajectories, h_lane_changes)
    current_time ← current_time + timestep
end while

```

The CPU thread calls a GPU kernel to perform the local sensing data lookup, calculate the instantaneous acceleration of each vehicle, and update its position and velocity for the next timestep. This GPU kernel is defined as follows:

Algorithm 3 GPU-Optimized Parallel Simulator — GPU Kernel

```

idx ← blockIdx.x × blockDim.x + threadIdx.x
stride ← blockDim.x × gridDim.x
while idx < num_segments × num_lanes do
  lane ← idx ÷ num_segments
  segment ← idx % num_segments
  current_id ← id[idx]
  if current_id = 0 then
    idx ← idx + stride
    continue
  end if
  curr_pos ← pos[idx]
  curr_speed ← speed[idx]
  for i ← segment to num_segments do
    check_idx ← lane × num_segments + i
    if id[check_idx] ≠ 0 then
      lead_pos ← pos[check_idx]
      lead_speed ← speed[check_idx]
      break
    end if
  end for
  accel ← INTELLIGENT_DRIVER_MODEL(curr_pos, curr_speed, lead_pos, lead_speed)
  curr_speed ← curr_speed + (accel × timestep)
  curr_pos ← curr_pos + (curr_speed × timestep)
  lane_change ← lane_changes[id[idx]]
  if not lane_change.lane_change_completed and lane_change.next_lane ≠ -1 then
    if curr_pos ≥ lane_change.lane_change_x then
      lane ← lane_change.next_lane
      lane_change.lane_change_completed ← true
    end if
  end if
  next_segment ← curr_pos ÷ 10
  next_idx ← lane × num_segments + next_segment
  if curr_pos < 4.2 × 5280 then
    pos_2[next_idx] ← curr_pos
    speed_2[next_idx] ← curr_speed
    id_2[next_idx] ← current_id
  else
    lane_change.trajectory_completed ← true
  end if
  idx ← idx + stride
end while

```

3.3 Experiments and Results

Both the baseline serial implementation and the GPU-optimized parallel implementation of the simulator were benchmarked on a complete single-day dataset of more than 580,000 trajectories. The serial implementation ran in 63.32 minutes, while the parallel implementation ran in 1.27 minutes, resulting in a speed-up of 49.85x. The GPU algorithm was tested on Perlmutter, a HPE Cray EX supercomputer, with 1 AMD EPYC 7763 CPU and 4x NVIDIA A100 GPUs. The graph below breaks down the execution time of the parallelized simulator into the various steps of the process (e.g. initialization, GPU kernel, memory operations, and CUDA synchronization).

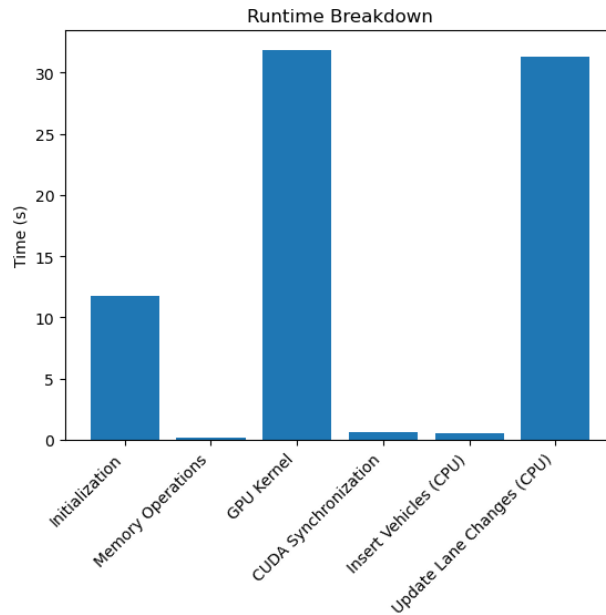


Figure 3.2: A breakdown of the runtimes of high-level operations during the simulation.

The subprocesses that take the majority of the runtime are: initialization, the GPU kernel, and updating lane change information (CPU). The most expensive component of the simulator initialization is reading the trajectories in from a JSON file, but this is necessary to set up the data structures that capture the simulator state. The GPU kernel is generally quite optimized, as it minimizes interference between threads by maintaining separate read-only and write-only versions of the 3 state arrays (position, speed, ID). The CPU function that updates lane change information is a bottleneck for the simulator performance because it iterates through all the active trajectories to check if their respective lane changes have been completed (and then updates the struct with the next lane change if applicable).

Below, the graphs display how the runtime scales with the number of simulator timesteps and the number of trajectories being simulated. There is an initial cost associated with the initialization of the parallel simulator (which means that it is slower than the serial implementation for smaller-scale simulations), but as the simulation scale increases, the parallel simulator demonstrates much better runtime and scaling performance than the baseline serial algorithm.

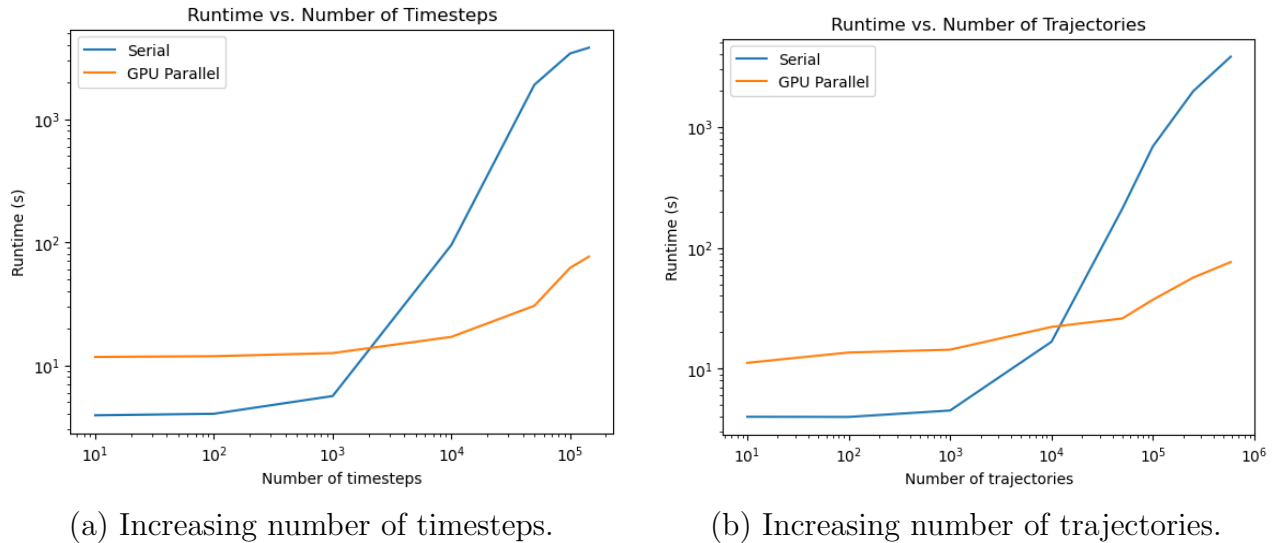


Figure 3.3: Performance graphs that depict the runtime as the number of simulation timesteps increase (left) and as the number of trajectories simulated increases (right).

The parallel simulator demonstrates great scaling properties. Specifically, when modeling the scaling in the form $O(n^{1+\epsilon})$, the runtime scales with a factor of $\epsilon = -0.8$ with respect to both the number of timesteps and the number of trajectories. This is because of the simulator's unique state representation (which is only dependent on the size of the highway, not the number of trajectories). Of course, there are components of the program that do scale with the number of trajectories, such as lane change information (which is not stored in the spatially segmented format).

To determine the optimal GPU grid size and block size, additional experiments were run to evaluate the simulator performance while varying parallelization parameters. The graphs below depict the runtime as the grid size increases (but the block size remains constant), and vice versa, respectively. For the other experiments, the block size was chosen to be 256, and the grid size was determined by dividing the maximum number of concurrently active vehicles by 256.

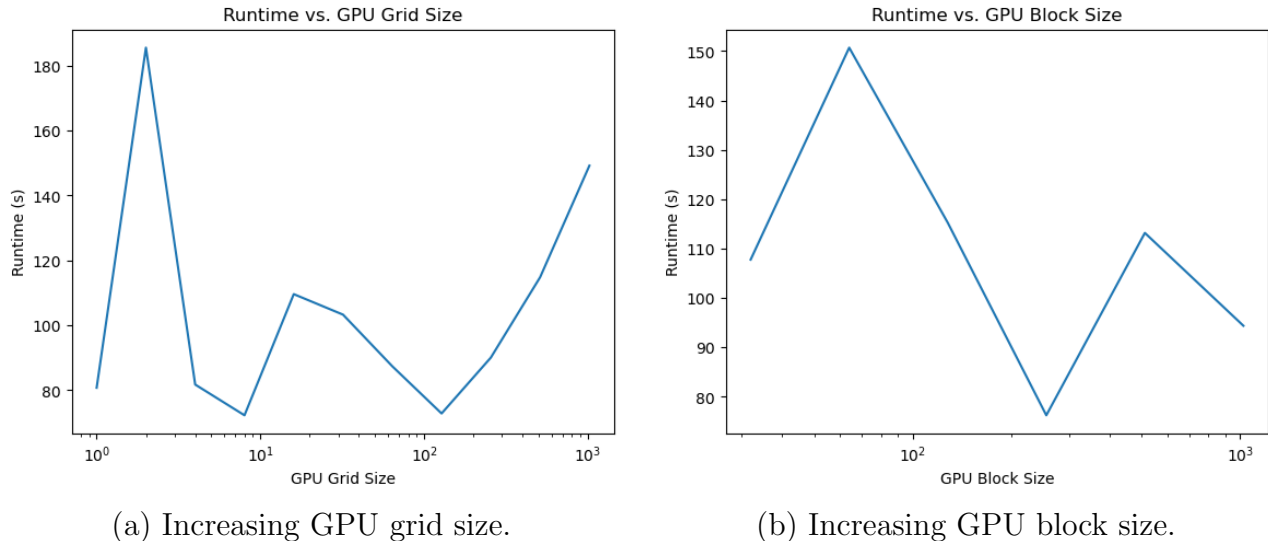


Figure 3.4: GPU strong scaling graphs that depict the runtime as the GPU grid size increases (left) and as the GPU block size increases (right).

3.4 Conclusion

This large-scale, GPU-optimized parallel simulator is designed to efficiently simulate hundreds of thousands of mixed-autonomy trajectories in the highway domain. As demonstrated through benchmarks, the parallelization and use of GPU kernels allow for a speed-up of nearly 50x over the serial algorithm. This simulator can be used during the training process for next-generation RL controllers for CAVs, potentially improving the resulting policies even further. It can also be used to benchmark the performance of existing RL controllers, with a realistic multi-lane environment that accurately re-creates real-world trajectories of human driven vehicles.

One caveat of this dataset is that it includes AV trajectories as well, but in the future, these can either be held out by filtering the trajectories that correlate with the onboard AV data recorded during the MegaVanderTest, or the simulator can be run on new datasets collected from the I-24 MOTION system, where the recorded trajectories would consist of purely human-driven trajectories.

There is scope to push the envelope of large-scale mixed-autonomy simulation, and even this simulator can be optimized further to provide higher performance. Innovation in the field of high-performance computing, along with highly-optimized vehicle simulation software, will continue to support the development of next-generation CAV controller software and improve highway throughput, traffic safety, and fuel efficiency for users all over the world.

Bibliography

- [1] N. Lichtlé, K. Jang, A. Shah, E. Vinitsky, J. W. Lee, and A. M. Bayen, *Traffic Smoothing Controllers for Autonomous Vehicles Using Deep Reinforcement Learning and Real-World Trajectory Data*, 2024. arXiv: 2401.09666 [eess.SY].
- [2] J. W. Lee, H. Wang, K. Jang, A. Hayat, M. Bunting, A. Alanqary, W. Barbour, Z. Fu, X. Gong, G. Gunter, S. Hornstein, A. R. Kreidieh, N. Lichtlé, M. W. Nice, W. A. Richardson, A. Shah, E. Vinitsky, F. Wu, S. Xiang, S. Almatrudi, F. Althukair, R. Bhadani, J. Carpio, R. Chekroun, E. Cheng, M. T. Chiri, F.-C. Chou, R. Delorenzo, M. Gibson, D. Gloudemans, A. Gollakota, J. Ji, A. Keimer, N. Khoudari, M. Mahmood, M. Mahmood, H. N. Z. Matin, S. Mcquade, R. Ramadan, D. Urieli, X. Wang, Y. Wang, R. Xu, M. Yao, Y. You, G. Zachár, Y. Zhao, M. Ameli, M. N. Baig, S. Bhaskaran, K. Butts, M. Gowda, C. Janssen, J. Lee, L. Pedersen, R. Wagner, Z. Zhang, C. Zhou, D. B. Work, B. Seibold, J. Sprinkle, B. Piccoli, M. L. D. Monache, and A. M. Bayen, *Traffic Control via Connected and Automated Vehicles: An Open-Road Field Experiment with 100 CAVs*, 2024. arXiv: 2402.17043 [eess.SY].
- [3] D. Gloudemans, Y. Wang, J. Ji, G. Zachar, W. Barbour, E. Hall, M. Cebelak, L. Smith, and D. B. Work, “I-24 MOTION: An instrument for freeway traffic science,” *Transportation Research Part C: Emerging Technologies*, vol. 155, p. 104311, 2023.
- [4] N. Nayakanti, R. Al-Rfou, A. Zhou, K. Goel, K. S. Refaat, and B. Sapp, *Wayformer: Motion Forecasting via Simple & Efficient Attention Networks*, 2022. arXiv: 2207.05844 [cs.CV].
- [5] S. Dai, L. Li, and Z. Li, “Modeling vehicle interactions via modified lstm models for trajectory prediction,” *IEEE Access*, vol. 7, pp. 38287–38296, 2019. DOI: 10.1109/ACCESS.2019.2907000.
- [6] M. Franchi, R. Kahn, M. Chowdhury, S. Khan, K. Kennedy, L. Ngo, and A. Apon, “Webots.HPC: A Parallel Simulation Pipeline for Autonomous Vehicles,” in *Practice and Experience in Advanced Research Computing*, ser. PEARC ’22, Boston, MA, USA: Association for Computing Machinery, 2022, ISBN: 9781450391610. DOI: 10.1145/3491418.3535133. [Online]. Available: <https://doi.org/10.1145/3491418.3535133>.

- [7] C. Gulino, J. Fu, W. Luo, G. Tucker, E. Bronstein, Y. Lu, J. Harb, X. Pan, Y. Wang, X. Chen, J. D. Co-Reyes, R. Agarwal, R. Roelofs, Y. Lu, N. Montali, P. Mougin, Z. Yang, B. White, A. Faust, R. McAllister, D. Anguelov, and B. Sapp, *Waymax: An Accelerated, Data-Driven Simulator for Large-Scale Autonomous Driving Research*, 2023. arXiv: 2310.08710 [cs.R0].
- [8] M. Harmel, A. Paras, A. Pasternak, N. Roy, and G. Linscott, *Scaling Is All You Need: Autonomous Driving with JAX-Accelerated Reinforcement Learning*, 2024. arXiv: 2312.15122 [cs.LG].
- [9] X. Jiang, J. F. Agerup, and Y. Tang, *Benchmarking and preparing lpsim for scalability on multiple gpus*, May 2023. DOI: 10.31219/osf.io/ezejrc. [Online]. Available: osf.io/ezejrc.