Embedded Reachability for Autonomous Racecar

by
Nathan Jewell

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Scholar)

Presented March 8, 2021
Commencement June 2021

AN ABSTRACT OF THE THESIS OF

Nathan Jewell for the degree of Honors Baccalaureate of Science in Computer Science presented on March 8, 2021. Title: Embedded Reachability for Autonomous Racecar.

Abstract approved:____

Houssam Abbas

Reachability analysis enables the safety assurance of control systems despite uncertain initial conditions and control inputs, and can be an important component to run on-board an autonomous system. This thesis explores the characteristics of reachability analysis with different algorithmic configurations and runtime parameters running onboard the F1Tenth 1/10th-scale autonomous racing platform. The results demonstrate that it is possible to obtain accurate, real-time reachability computations using only simple kinematics. Parallelizing reachability algorithms on a GPU without algorithmic changes is feasible and reduces runtime. This is all achieved while running the algorithm alongside a fully autonomous racing stack. This implementation uses simple linearized kinematics, yet it is accurate; runs in Python, yet is real-time; shares the hardware with a full autonomous racing stack; and is parallelized to the GPU without algorithmic changes. These features lower the barrier to running reachability significantly. In testing, accurate 10-step reachability as quickly as 150 milliseconds is achieved. It was found that porting portions of reachability onto the GPU can improve the runtime for 10-step reachability by 62% over a CPU-only implementation.

Key Words: Reachability, Safety Analysis, CUDA, GPU

Corresponding e-mail address: ncjewell@gmail.com

©Copyright by Nathan Jewell
March 8, 2021

Embedded Reachability for Autonomous Racecar

by
Nathan Jewell

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Scholar)

Presented March 8, 2021
Commencement June 2021

Honors Baccalaureate of Science in Computer Science project of Nathan Jewell presented on March 8, 2021.

APPROVED:

____

Houssam Abbas, Mentor, representing EECS

____

Weng-Keen Wong, Committee Member, representing EECS

____

Stanley Bak, Committee Member, representing Computer Science at Stony Brook University

____

Toni Doolen, Dean, Oregon State University Honors College

I understand that my project will become part of the permanent collection of Oregon State University, Honors College. My signature below authorizes release of my project to any reader upon request.

____

Nathan Jewell, Author

Embedded Reachability for Autonomous Racecar*

Nathan Jewell[1]

Oregon State University, Corvallis OR 97333

Abstract. Reachability analysis enables the safety assurance of control systems despite uncertain initial conditions and control inputs, and can be an important component to run on-board an autonomous system. This thesis explores the characteristics of reachability analysis with different algorithmic configurations and runtime parameters running onboard the F1Tenth 1/10th-scale autonomous racing platform. The results demonstrate that it is possible to obtain accurate, real-time reachability computations using only simple kinematics. Parallelizing reachability algorithms on a GPU without algorithmic changes is feasible and reduces runtime. This is all achieved while running the algorithm alongside a fully autonomous racing stack. This implementation uses simple linearized kinematics, yet it is accurate; runs in Python, yet is real-time; shares the hardware with a full autonomous racing stack; and is parallelized to the GPU without algorithmic changes. These features lower the barrier to running reachability significantly. In testing, accurate 10-step reachability as quickly as 150 milliseconds is achieved. It was found that porting portions of reachability onto the GPU can improve the runtime for 10-step reachability by 62% over a CPU-only implementation.

Keywords: Reachability · Safety Analysis · CUDA · GPU
Code URL: *https://github.com/NathanJewell/ppcm_reachability_public*

*Supported by NSF Award 1925500.

Acknowledgements

I could not have completed this thesis without the support and efforts of many people. Dr. Houssam Abbas, my mentor through the whole process and the root of the F1Tenth research at OSU was, of course, essential in identifying these research topics and helping me throughout with the calculations, ideation, writing, and so much more. Niraj Basnet was instrumental in deploying the underlying autonomous software, coordinating laboratory logistics, integrating reachability with the MPCC, and creating the racetrack for testing. Dr. Stanley Bak was another essential collaborator and without his adaptation of HYLAA into the QuickZonoReach tool, profiling would not have left the ground. Another huge thank you to the entire Sabotage Lab for their insight, friendliness, and interest. The entire F1Tenth class of Spring 2019, our TA Abhi, and my immediate team, Brian and Amrita, must also be recognized for their work in inspiring my interest for this research, assembling the test vehicles, and troubleshooting innumerable hangups prior to my beginning this thesis. And finally, I have so much gratitude for my friends and family, who were always there to bounce ideas around with, edit my work, and keep me motivated over the course of this project. Thank you all so much.

Table of Contents

1 Introduction 4

1.1 Embedded Reachability 4

1.2 Research Topics 5

2 Platform 5

2.1 Hardware 5

2.2 Software 6

3 Reachability 8

3.1 Kinematic Model 8

3.2 Hylaa 9

3.3 QuickZonoReach 10

3.4 Successive Linearization 10

3.5 Successive Linearization of Kinematics 11

3.6 Runtime Modes 13

4 Setup and Methods 14

4.1 Offline and Online Execution 14

4.2 Implementation 14

4.3 Parallelization 15

4.4 Characterizing Performance 17

5 Results 19

6 Conclusion 21

1 Introduction

1.1 Embedded Reachability

Autonomous vehicles operate in an uncertain space where the exact position and control inputs at any time are not known precisely. Because of this uncertainty, the safety outcomes of autonomous vehicles are called into question. For a reliable system, it is necessary to understand the full range of possibilities which may occur at future times under these uncertain conditions. Autonomous vehicles are safety-critical systems in that they cannot fail under normal operation. It must be assured that the planned actions of the vehicle will be safe, not resulting in collision or other disaster, despite inherent uncertainty. Reachability analysis is a methodology for generating these future states while accounting for the uncertainty and it can be used to plan around unsafe events. Effective reachability tools must run quickly enough to generate future states, evaluate their safety implications, and still allow time for the vehicle to respond to the situation and guarantee a safe outcome. This type of useful reachability analysis is real-time.

In this research the characteristics of real-time reachability on an autonomous embedded system was explored in different runtime configurations and algorithmic variations. Reachability is a type of safety assurance which is used to determine the possible, that is, the “reachable,” system states at some point in the future. In an embedded system, the computational capacity is constrained by the design parameters for the device, resulting in limited processing power available for execution of control, mapping, and safety algorithms. Despite this, computer controlled systems are being deployed in increasingly critical applications, clarifying the need for highly reliable and efficient algorithms for safety assurance, such as reachability.

Focusing on real-time reachability in this work has particular importance for developing meaningful improvements in control outcomes for autonomous systems. Real-time reachability is reachability computation where the output states are accurate estimators for the system state at a future time. Enabling real-time safety analysis in this manner allows for the system to analyze and react to the safety results prior to collision or other safety problems, improving confidence in the safety of the underlying control systems at runtime.

For these experiments, the target system was the embedded Jetson TX2, standardized for the F1Tenth Autonomous Racing Platform (See Section 2). The F1Tenth platform provides a common hardware platform and base software packages for autonomous mapping and navigation. By increasing access to these emerging technologies, the F1Tenth platform facilitates rapid development of improved autonomous algorithms.

1.2 Research Topics

The foremost research goal was to integrate and test reachability on an embedded system without doing time-intensive physical system identification. The intention was to create implementations with sufficient accuracy and precision to allow management of safety outcomes in real-time. Meeting these criteria without system identification was very important since developing an accurate physical model of the system dynamics is not feasible for many researchers. This research demonstrates the promise of using a simple kinematic model along with technique of successive linearization to reduce configuration and testing overhead.

Increasing access to effective safety algorithms is also a significant goal. By improving the speed and accessibility of safety assurance, the visibility of these techniques is improved within the F1Tenth community and other autonomous control applications. This visibility can play an important role in creating opportunity for continuing research and improvement by reducing the roadblocks to creating and testing new techniques.

The final research goal was to examine the applications for parallel computing in this field and demonstrate the impacts of CPU- and GPU-based parallelization on reachability algorithm performance in the context of embedded systems concurrently running fully functional autonomous navigation stacks. Due to the limited nature of embedded computation, increasing utilization of modern GPU computing would allow for higher effective onboard computing capacity by freeing CPU cycles for sequential algorithms and other sequential tasks.

Related work GPU-based implementations of dynamical systems reachability include Xspeed which introduces algorithms optimized for GPU performance [6]. Xspeed modifies reachability algorithms to enable GPU parallelization, unlike our approach. Using simple linearized kinematics enabled faster runtimes compared to what's reported in [6]. The work in [1] also does real-time reachability but uses a non-linear model which requires systems identification, which is a step avoided in this work.

2 Platform

The F1Tenth platform was created to encourage research in autonomy, control systems, computer vision, and other autonomous technologies at institutions around the world. Races in various styles are held multiple times each year to encourage competition and facilitate comparison of unique perception and control algorithms on a level playing field.

2.1 Hardware

The autonomous vehicle is built from conversion guidelines provided by F1Tenth. It is based on the chassis of a remote control car, heavily modified to support sensors and onboard computers (Fig. 1). The physical capabilities of the platform easily allow speeds greater than 10 meters per second if the control algorithms can handle it.

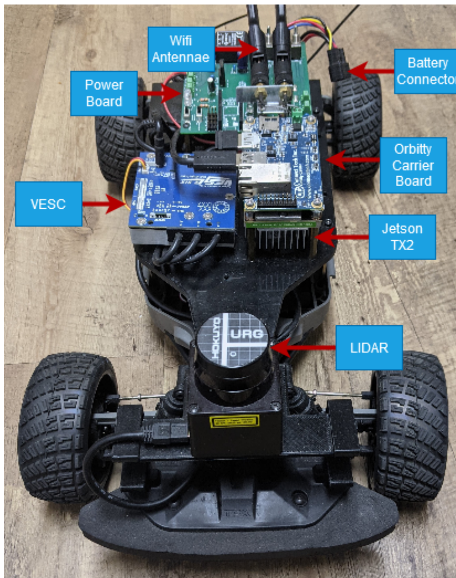

Fig. 1: A standard configuration for the F1Tenth platform. Components are not centrally sourced, so different vehicles have different hardware but comparable specifications. The motor and gearbox (not in frame) are located in the rear of the vehicle in gray plastic undercarriage beneath mounted electronic controllers. The undercarriage also holds the 5000mah Lithium Polymer battery on one side and the steering control servo at the front axis.

The drivetrain is a fixed gear electric motor mounted in the rear of the vehicle with a driveshaft enabling power delivery to both axes concurrently. An improved electronic speed controller (VESC) is included on the vehicle. It was necessary to perform manual tuning of motor actuation curves to mitigate gear slippage at lower speeds and when accelerating. Steering actuation is controlled by a front servo.

The compute platform is a Nvidia Jetson TX2 equipped with the Orbitty breakout board. This board provides I/O ports in a much smaller form factor than the included TX2 development board. The Jetson TX2 has a dual ARM CPU setup with six (4+2) physical cores, as well as a CUDA capable Nvidia GPU. A power board with outputs for the Jetson TX2, Wifi Antenna, and servo is also installed on the system as part of the standard conversion.

The primary sensor is a single horizontal axis LIDAR. This sensor returns depth data for thousands of points at equally spaced angles multiple times per second. Other sources of sensor data in the car include the VESC and servo feedback, which are used to estimate drive speed and steering angle.

2.2 Software

Nvidia enforces a proprietary Ubuntu-based distribution called Jetpack for their Jetson TX2 devices. This distribution includes necessary support drivers for the

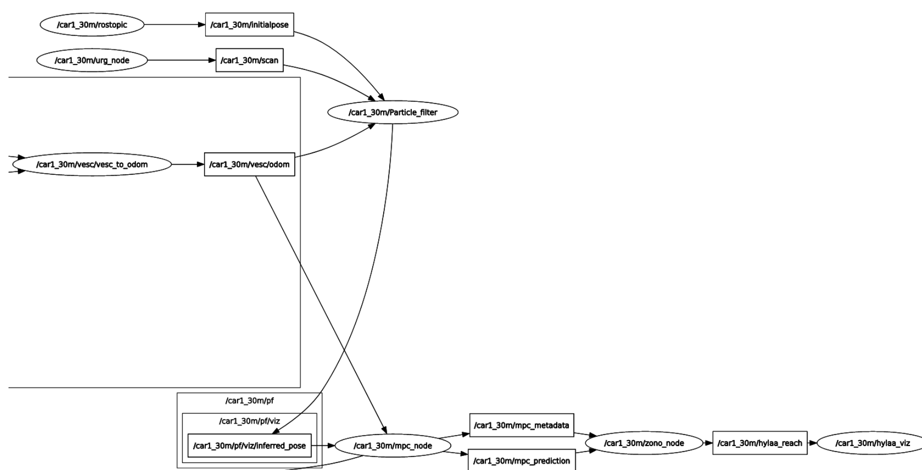

Fig. 2: High-level nodes in a graph showing the node layout in the autonomous pipeline. Many nodes which control low-level interactions with input devices or the VESC are not pictured.

hardware platform as part of the package. It also includes support for other Nvidia technologies like CUDA. To interface with our software components it was necessary to install various third party libraries, many of which had to be compiled specifically for the TX2 due to its unique hardware and operating system configuration. Temporary roadblocks due to complex installation procedures specific to the TX2 were a common feature in setting up the development environment for this research.

The Robotic Operating System (ROS) was used to manage concurrently running the array of autonomous algorithms and low level sensor data collection software. ROS is a networked operating system for robotics and allows various disparate software functionalities to be abstracted as ROS nodes. Nodes communicate over a messaging pipeline and can run on different hosts within one ROS network. Hardware manufacturers for many of the onboard components of the F1Tenth platform provide low-level ROS nodes, which transform sensor output to ROS messages. There are also publicly available nodes which were used to direct motor and servo actuation. The basic configuration for these low-level systems is part of the guidelines provided by F1Tenth platform, but some unique ROS nodes and configuration changes were necessary to achieve basic hardware-software interface specific to the selected models of LIDAR and VESC. Additional publicly available nodes were used for basic parts of the autonomous pipeline. These include a node implementing the Google Cartographer SLAM algorithm and a particle filter aiding in localization was also used [3].

Custom nodes were developed to address higher level navigation and control problems. These include nodes implementing various reachability techniques discussed in this thesis and a Model Predictive Contouring Control (MPCC) [4] node developed as a part of other work. MPCC is an algorithm which handles the error correction and input selection for the vehicle. In this research the MPCC node was used as part of the base autonomous driving pipeline during profiling of reachability performance.

Orchestrating reachability as ROS nodes constituted the bulk of the development work in this research. The original linear reachability tool, Hylaa was integrated separately from the pared down QuickZonoReach tool (See Section 3). In each case, reachability configuration and setup was handled by a wrapper class which was imported by the ROS nodes and test scripts. The reachability nodes themselves were primarily responsible for loading configurations specific to the online testing modes and executing reachability via the wrapper classes. The nodes also pushed resultant reach sets to the ROS network in real-time. A partial graph showing the most important nodes in our ROS network is shown in Figure 2. It can be seen how the data flows from the VESC, pose estimation, and LIDAR to the MPCC and then to reachability nodes.

3 Reachability

For dynamical systems, *reachability analysis* is the computation of possible future system states, given a *set* of initial states and a *set* of possible inputs at each time step.

In discrete-time reachability, a range of N reach sets are computed for N future times dt, $2dt$,..., Ndt, with the guarantee that any state actually reached at time $i \cdot dt$ is within the ith computed reach set; thus the latter can be utilized for safety analysis [5].

Specifically, with safety-critical systems, it is necessary to verify that no error state could be reached. In autonomous vehicles, error states may involve intersections with walls, other cars or people, etc. By checking that the reach set does not contain error states, a guarantee is obtained that the next N time steps are safe. If not, planned inputs can then be modified to avoid danger, preventing potentially catastrophic system failure.

3.1 Kinematic Model

The vehicle is represented by a simple 3 variable kinematic model:

$$\dot{x} = v\cos(\psi + d),\ \dot{y} = v\sin(\psi + d),\ \dot{\psi} = v\frac{1}{L}\sin(d) \tag{1}$$

where the state variable ψ is the current yaw of the vehicle. The input variable v is the velocity of the car. The input variable d is the steering angle in the vehicle's frame. The constant $L = .32$ is the wheel length of the vehicle in meters.

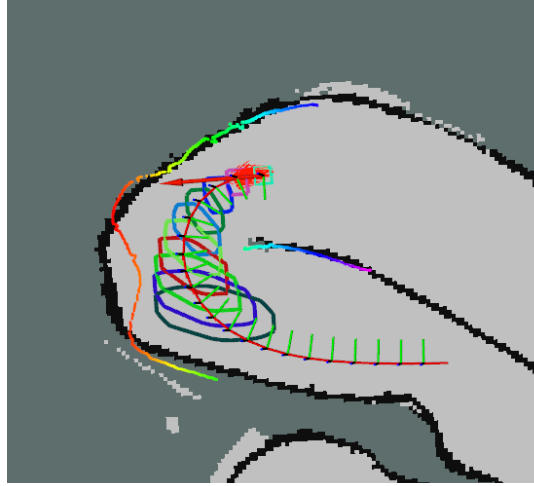

Fig. 3: Zonotope projections from a 10-step reachability problem on-board F1Tenth vehicle (multicolored polygons). Visualization shows the rasterized map (in grey), overlayed by LIDAR scan data (multicolored points), MPCC trajectory (red line), and vehicle position/heading (red arrows). Note the MPCC trajectory lies centered on the reach sets.

Define the state vector, $S = [x, y, \psi]$, and the input vector $U = [v, d]$. The onboard MPCC algorithm provides a sequence of optimal inputs for the next N steps, $U_1, ..., U_N$, and the corresponding predicted state trajectory is $S_1, ..., S_N$. The non-linear kinematics are linearized around each (S_t, U_t), thus obtaining the matrices A_t, B_t, and the linearized dynamics at step t are then $S_{t+dt} = S_t \cdot A_t + U_t \cdot B$. A correction parameter α is introduced to scale the steering angle prior to evaluation of the Jacobian matrices: $\bar{S}_t = [\bar{x}, \bar{y}, \alpha \cdot \bar{\psi}]$.

This parameter was added to address the problem of over or under steering in the linearized kinematics and was tuned experimentally to $\alpha = .5$. This is the only parameter needing tuning, which is done by comparing the center of output zonotopes to the computed trajectory and tuning experimentally until a sufficient match is found. The parameter is quickly determined experimentally since the relation between alpha and the difference in center lines is generally monotonic and thus can be deduced without falling into local optimalities. See Figure 3.

3.2 Hylaa

Hylaa [2] is a state-of-the-art linear reachability analysis tool which can handle linear hybrid dynamical systems. Hylaa output is a collection of polytopic reach sets and performs checking for safety conditions inline with computing these sets. In this way, system trajectories coincident with error modes are not necessarily computed through the entire reachability time horizon, potentially saving computing time. Hylaa represents error modes as polytopes (a bounded intersection

of half-spaces). Checking the intersection of polytopes reduces to a linear feasibility problem.

3.3 QuickZonoReach

[https://github.com/stanleybak/QuickZonoReach] QuickZonoReach is a variant of Hylaa obtained by disabling certain unneeded features and by delaying the process of error checking to after computation of reachable sets instead of inlining this as in Hylaa. By delaying error checking, it was possible to explore applications of both CPU- and GPU-based parallelism in this part of the algorithm. The core Hylaa algorithms were not modified. The core reason for delaying error checking was the computation intensity of that task. The linearity of the system allows for very fast computation of reachable system states. But, while the system state can be n dimensional for any n, the unsafe sets for this application vehicles are 2-dimensional obstacles in the (x, y) plane. This is a slower task than applying the linear dynamics since n-dimensional surfaces must be projected into the (x, y) plane.

3.4 Successive Linearization

Successive linearization is a technique of creating multiple linearizations around time-sequential points to reduce inaccuracies from linearizing equations. A nonlinear function F linearized point K is generally an approximation for F which is more accurate closer to k and less accurate further from k. So, in the case of the vehicle's dynamics, which are nonlinear, if a single linearization were used for the dynamics in the entirety of the reachability computation, the accuracy would be awful. As linear reachability tools, Hylaa and QuickZonoReach would generally use a single linear system to do calculations, but we know this would not give useful results in our case.

Performing successive linearizations changes things. A single linearization point can be computed accurately and quickly using nonlinear equations and then transformed to a unique set of linear dynamics at a particular time. The reachability dynamics are used to transform multidimensional reach sets with many vertices. Since all of the vertices in a reach set can be assumed to be relatively near the center point, the accuracy of any given linearization is decent for the entire reach set. In this way, multiple linearizations greatly improve over a single linearization. Additionally, the computation time is reduced since fast linear algebra from the linearized dynamics can be applied over the entire reach set more efficiently than nonlinear dynamics. The input points for the successive linearization are given by outputs of the simulated car trajectory, either from the MPCC or as a standalone simulation. A demonstration follows for the successive linearization used with the kinematic models for the F1Tenth research vehicle.

3.5 Successive Linearization of Kinematics

The process of successive linearization outlined in Section 3.4 is demonstrated algebraically. Let the variables $\dot{x}, \dot{y}, \dot{\psi}, v, d$ and the vectors S, U be defined as in Section 3.1.

It will be shown that this nonlinear system can be linearized and rewritten in terms of S and U at a discrete time step k.

$$S_{k+1} = A_k S_k + B_k U_k \tag{2}$$

Where $S_k = [x_k, y_k, \psi_k]$ the system's state at a kth step and $U_k = [v_k, d_k]$, the system's inputs at a kth step. Such that the matrix products $A_k S_K$ and $B_k U_k$ are defined.

Start by deriving a discrete time form of the nonlinear kinematics

Consider the definition of the derivative for a function $f(t)$

$$\dot{f}(t) = \lim_{a \to 0} \frac{f(t+a) - f(t)}{a}$$

with a constant small step $a = dt$, we can write a definite form approximating $\dot{f}(x)$ without the limit.

$$\dot{f}(t) \approx \frac{f(t+dt) - f(t)}{dt}$$

Substituting with the state variables, and discretizing by $t = k * dt$ we are left with:

$$\dot{S_k} \approx \frac{S_{k+1} - S_k}{dt}$$

Create the function $F(S_k, U_k)$ such that $\dot{S_k} = \left[\dot{x_k}, \dot{y_k}, \dot{\psi_k},\right] = F(S_K, U_k)$

$$\begin{aligned}
\dot{S_k} &= F(S_k, U_k) && \text{by Definition} \\
\frac{S_{k+1} - S_k}{dt} &= F(S_k, U_k) && \text{by Substitution} \\
S_{k+1} &= S_k + dt * F(S_k, U_k) && \text{multiply by dt, add } S_k
\end{aligned} \tag{3}$$

It is necessary to linearize $F(S_k, U_k)$ so we have $L_F(S_k, U_k)$ linearized in terms of $F(\bar{S}, \bar{U})$ where $(\bar{S}, \bar{U})$ is the *point of linearization*

$$L_F(S_k, U_k) = F(\bar{S}, \bar{U}) + \frac{\partial F}{\partial S}\Big|_{\bar{S}} S_k - \frac{\partial F}{\partial S}\Big|_{\bar{S}} \bar{S} + \frac{\partial F}{\partial U}\Big|_{\bar{U}} U_k - \frac{\partial F}{\partial U}\Big|_{\bar{U}} \bar{U}$$

This linearization is approximating F, so we can say $L_F \approx F$ and then, by substitution into (3)

$$S_{k+1} \approx S_k + dt * L_F(S_k, U_k) \tag{4}$$

Now expand L_F in (4) and collect constant terms, leaving

$$S_{k+1} \approx S_k + dt\left[F(\bar{S}, \bar{U}) + \frac{\partial F}{\partial S}\Big|_{\bar{S}} S_k + \frac{\partial F}{\partial U}\Big|_{\bar{U}} U_k\right] + dt\left[F(\bar{S}, \bar{U}) - \frac{\partial F}{\partial S}\Big|_{\bar{S}} \bar{S} - \frac{\partial F}{\partial U}\Big|_{\bar{U}} \bar{U}\right]$$

Multiplying dt, factoring S_k leaves

$$S_{k+1} \approx \left[I + dt\frac{\partial F}{\partial S}\Big|_{\bar{S}}\right] S_k + \frac{\partial F}{\partial U}\Big|_{\bar{U}} U_k + C$$

Where the constants have been grouped in the constant term C

It is easy to see that this equation is now affine in the form

$$S_{k+1} \approx A_k S_k + B_k U_k + C$$

where

$$A_k = \left[I + dt\tfrac{\partial F}{\partial S}\Big|_{\bar{S}} S_k\right]$$
$$B_k = \left[\tfrac{\partial F}{\partial U}\Big|_{\bar{U}} U_k\right]$$

A final transformation is required to combine the affine term and find the equation in our desired format,

$$S_{k+1} = A_l S_l + B_k U_k \text{ s.t. } A_l S_l = A_k S_k + C$$

By expanding the A_k and S_k matrices, a solution can be found. Define these matrices

$$A_l = \begin{bmatrix} A & I \\ 0 & I \end{bmatrix} \quad S_l = \begin{bmatrix} S_k \\ C \end{bmatrix} \tag{5}$$

Note: for the constant matrix C, $C = C_0 = C_1 = C_2 = ... = C_k$

Then

$$A_l S_l = \begin{bmatrix} A & I \\ 0 & I \end{bmatrix} \begin{bmatrix} S_k \\ C \end{bmatrix} = \begin{bmatrix} AS_k + IC \\ IC \end{bmatrix} = \begin{bmatrix} AS_k + C \\ C \end{bmatrix}$$

$$S_{k+1} = A_l S_l + B_k U_k$$

So, it has been shown that the definitions in (5) are correct and the successive linearization is achieved in the format of (2). In practice, B_k and U_k are also augmented to match the dimensionality of the augmented A_l and S_l matrices and carry constants between successive linearizations.

3.6 Runtime Modes

Four ways of distributing the workload between the CPU cores and the GPU cores were evaluated by differing implementations of the reachability tool:

Hylaa (HYLAA) is the original Hylaa reachability tool, configured to use successively linearized dynamics. Computation is completely bound to the CPU on one core.

QuickZonoReach Single Core (QZ_CPU) is the abridged QuickZonoReach adaptation of Hylaa, pared down for fast execution and easy integration with the successively linearized dynamics. Executed on a single CPU core.

QuickZonoReach Multiple Core (QZ_MP) is the same as the QZ_CPU mode with a trivial process-based parallelization for the projection step of QuickZonoReach. This is configured to allow concurrent execution on up to 4 physical CPU cores.

QuickZonoReach GPU Hybrid (QZ_HYBRID) introduces major changes to the projection step by moving parts of computation to GPU. Zonotopes are copied to GPU and a hybrid approach is used for generating 2d polygons with intermittent host-device copy cycles. As in the other modes, the CPU still performs reachability on a single core - only the projection steps are parallelized.

4 Setup and Methods

4.1 Offline and Online Execution

Successively linearized dynamics were tested in two primary execution modes. In Offline Execution, reachability code was tested without other software running concurrently. Profiling tools specific for this mode were created to run tests without the overhead of a ROS node or other pieces of the autonomous pipeline competing for computational capacity. When offline, the reachability tool generates its own single-point predictions for use in the successive linearization. This does take some time but is ignored when profiling the reachability performance.

In Online Execution, reachability code was tested alongside the entire autonomous pipeline including SLAM and MPCC. Profiling was integrated into the ROS Nodes responsible for the reachability. A live visualization was also produced by rendering polygons for the output reach sets in RVIZ. In this mode, point predictions for successive linearization are provided by the MPCC via ROS messages. The MPCC continually reports these estimations over the ROS network as it plans control inputs. This means that there is reduced overhead for performing the successive linearization since the estimations do not need to be computed specifically for reachability. It also means that the dt, time between reachability steps, had to be configured as an multiple of the dt used in the MPCC and limited the maximum number of reachability steps to the maximum number of predictions generated by the MPCC. However, since the reachability performance is only impacted by the number of timesteps, not the time between them, and since the MPCC produces many more predictions than are feasible to use for safety analysis, the caveats are of little consequence.

4.2 Implementation

Reachability was implemented in three layers.The base algorithm (Hylaa or QuickZonoReach), a wrapper class to standardize calling semantics between the algorithms and configurations, and finally the high level ROS node which interfaces with the rest of the autonomous networks and controls the timing

The specifics of the Hylaa algorithm and the QuickZonoReach adaptation are covered in the reachability section (3). Hylaa and QuickZonoReach are primarily implemented in Python3 using Numpy for heavy math.

Wrapper classes were written to facilitate the configuration and calling of reachability in both Hylaa and QuickZonoReach. The calling conventions between these classes are relatively synonymous but do have important differences reflective of the simplified and stripped-down nature of the QuickZonoReach algorithm. In the offline mode, this class also handles generating a simulated set of future

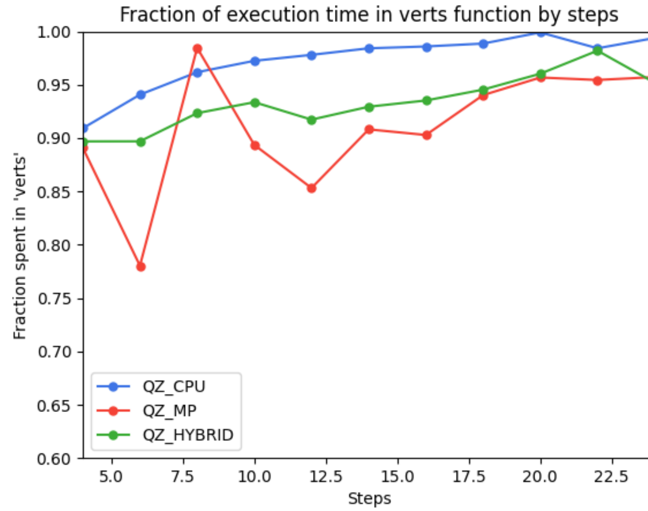

Fig. 4: The "verts" function is where all of the projection for each reach set is done. Profiling indicated that most execution time was spent in this function, this graph shows that proportion. The vertical axis is the fraction of total execution in the verts function compared to total execution time for reachability.

states using nonlinear dynamics which are used in the successive linearization. In the offline mode, a small utility script initializes the wrapper class and starts the reachability computation. In the online mode, however, a ROS node is used to allow for specific configuration changes and communication with other parts of the autonomous pipeline.

The reachability ROS node instantiated the wrapper class for Hylaa or QuickZonoReach. Alongside this, a variety of builtin ROS features are used to load configuration files used to configure the parameters for reachability including the horizon time (T), timestep (dt), desired latency, and the runtime mode. The node also uses the ROS messaging system to receive predicted trajectories from the MPCC and relay them to the reachability algorithm using the wrapper functions. After the reach sets are computed, the ROS node reports them in real-time to the ROS network using the messaging system. When running visualization, these messages are consumed by a visualization node so they may be rendered as lines in Rviz.

4.3 Parallelization

Initial profiling of QuickZonoReach was done using pycallgraph and cProfile in the QZ_CPU mode. These tools measure the time for each function call over the

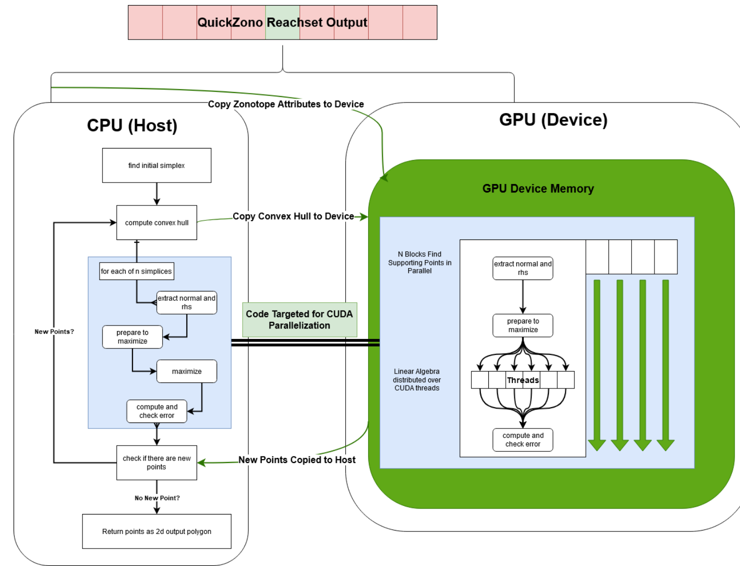

Fig. 5: The QZ_HYBRID runtime mode implements a CPU + GPU hybrid computation model. Synchronous nested loops are translated to asynchronous grids of CUDA blocks in 2 dimensions. Matrix functions are distributed across 3 dimensions of threads within each block. The light blue blocks on the host and device perform analogous computations but, when executed on the GPU, as in QZ_HYBRID, the work is distributed across hundreds of worker threads.

entire course of execution for a reachability computation. This testing revealed that the largest portion of CPU time was spent projecting from n-dimensional zonotopes into the 2-dimensional polygons useful for checking safety parameters (Fig. 4). This strongly indicated the projection algorithm as a prime target for parallelization. It had the most potential in terms of reducing CPU time and the projection step is particularly well suited for parallel computing. The projection is especially good for parallelism because it depends only on the target zonotope and thus the algorithm can be replicated across all N output zonotopes concurrently without race conditions.

CPU parallelization was accomplished trivially by mapping the projection function across the list of reach set outputs using Python3.6 built-in *subprocess.ProcessPool* object. This method was effective in reducing the execution time and representation of a CPU-based parallelization paradigm (See Section 5).

GPU parallelization involves significantly more setup but opens doors to significant increases in computing power (Fig. 5). Importantly, offloading portions of CPU work to the GPU allows that CPU time to be used for other work which cannot be parallelized, increasing the utilization of limited computational resources. The onboard TX2 includes an NVIDIA GPU capable of running generalized algo-

rithms written in CUDA. CUDA is a C based language which exposes GPU functionality to the programmer by way of CUDA kernels. These kernels are pieces of code which execute on the GPU hardware. Because the code is executed on the GPU hardware, the concept of a host (the CPU) and a device (the GPU) is relevant. All code and pieces of memory needed on the GPU must be copied there before execution. This poses an overhead time cost to GPU computing and minimizing copies is an important piece of designing parallel applications.

A CUDA kernel was written to parallelize the most time intensive function in the projection step by moving it onto the GPU. Sequential CPU code was translated into this single kernel as directly as possible (Fig. 5). Synchronous iteration was not entirely avoided for the projection step, so a hybrid implementation of QuickZonoReach, QZ_HYBRID, resulted. Specifically, computation of the convex hull was not moved to the GPU. This necessitated periodic copies between the host and the device as the convex hull was repeatedly computed as part of the iterative algorithm for finding lower dimensional points. Effort was taken to minimize the number of other memory copies between the host and the device.

The other primary challenge in writing CUDA code is organizing the massively parallel nature of GPU execution. CUDA divides the GPU into a 3-dimensional grid of blocks and each block into a 3-dimensional grid of threads.However, the dimensionality of this grid and blocks is only a software level abstraction from the physical execution paradigm. So, when designing efficient CUDA code it is necessary to understand both the software abstraction and how the underlying hardware is assigned execution tasks. In computing the projection, the size of the grid and blocks were defined by the dimensionality of the reachability problem and also dynamically adjusted for each iteration depending on the size of problem. Rewriting the code directly as a single CUDA kernel resulted in some suboptimal design. The implementation for QZ_HYBRID abuses CUDA thread abstractions to simplify translation. Using shared memory only available between threads of the same block eased the implementation of this algorithm on the GPU. However, block size (dimensionality of allocated threads) suffers from a severely constrained scalability compared to grids (dimensionality of allocated blocks). Simplifying the implementation sacrificed the significantly larger parallelism afforded by larger grid dimensionality.

4.4 Characterizing Performance

Timing data was collected as the primary method of understanding the performance characteristics of reachability. For the offline mode, standalone scripts were written to automatically run trials for some permutation of the configurations. In the online mode, reachability was executed as quickly as possible inline with the rest of the ROS timing code but the exact same function call was still times as in the offline mode. In both cases, resulting timing data was written to output files for further analysis and graphing. Specific performance characteris-

tics are described to facilitate collection and comprehension of the timing data. Each characteristic has an empirical measure which was used to analyze it quantitatively.

Correctness is the guarantee that the outputs are a valid solution to the problem. It is important to know that the algorithm is producing results consistent with the desired output and underlying mathematics. This is especially important for a reachability algorithm which is utilized in safety analysis protocols. To validate correctness, Hylaa model output was assumed to be correct. Outputs from other runtime modes were compared to the Hylaa output to discover if there are any significant differences. It was not necessary to validate correctness separately for online and offline computation since the same code was executed in each case.

Scalability is how well the algorithm scales with increased problem size. In this case, problem size is dictated by the number of time steps. This corresponds to either or both of an increase in horizon or decrease discretization time step. It is important to understand how the algorithm scales so intelligent decisions can be made about optimal horizon and precision in real-time applications. To validate scalability, each runtime model is tested with a range of values for the number of reachability steps which are representative of useful workloads. The number of steps ,N, is equal to the ratio between total horizon time in seconds, T, and the discretization time-step, dt. This varying of step values is done for both online and offline modes. However, for online trials, only models capable of running at approximately 200ms per iteration (5hz) are considered.

Runtime is the time to solve the given problem. Minimizing runtime is a critical component of this research. Running a real-time safety algorithm is only effective when there is a low latency between consecutive executions. If this criteria is not met, safety analysis could be out of date when the agent is developing a response, resulting in ineffective performance. In contrast, low latency (runtime) will improve the safety margin of the agent by providing reachability information to the controller which is very up to date with the environment. To test runtime, each combination of runtime mode and steps is evaluated over multiple trials. The recorded runtimes will be the average over 200 trials. This data collection is done for both online and offline testing.

Runtime variability is the standard deviation in the runtime. While latency is of primary importance, the consistency of the algorithm runtime is paramount for critical systems. Latency spikes at the wrong time could create dangerous problems for the agent when reacting to unsafe conditions. To test runtime variability, the recorded runtimes from all trials are analyzed by computing standard deviation in runtime. From this, a 95% runtime confidence intervals is generated around the mean, assuming normality holds due to the large sample size. This analysis is performed for both online and offline testing.

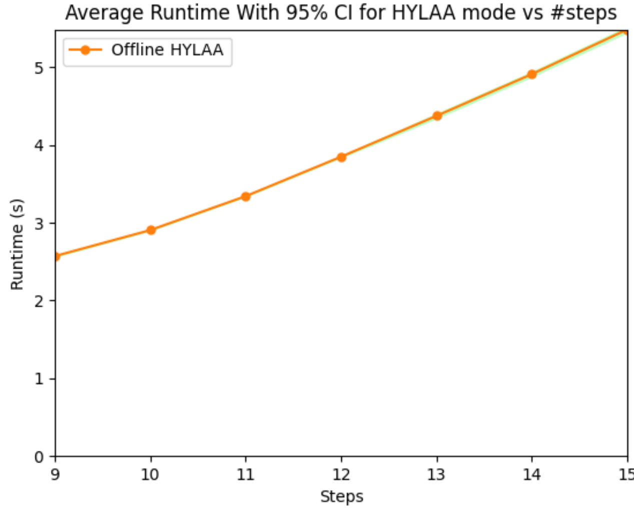

Fig. 6: The runtimes compared to number of reachability steps in offline Hylaa.

5 Results

Performance of the Hylaa implementation was only evaluated in offline mode. This is because the runtimes were excessive for performing reachability in real-time (Fig. 6). Notice that 10-step reachability takes over around 3 seconds. To be real-time at this speed, the dt (reachability timestep) would have to be a full $\frac{1}{3}$ second which is not likely to be accurate for practical purposes. These high runtimes were expected, and they were the reason for pursuing lighter-weight implementations in QuickZonoReach. Especially with these longer runtimes, the exponential increase in runtime with respect to the number of reachability steps, N, means that the computation becomes prohibitively expensive very quickly.

real-time reachability performance at 4hz or greater was achieved for both QZ_MP and QZ_HYBRID. This represents a significant performance improvement over QZ_CPU (Fig. 7). CPU-based parallelization in QZ_MP showed improvement larger than 60% at 10 time steps over QZ_CPU. But QZ_MP strained four CPU cores which may be better utilized in other autonomous navigation or control functions. This highlights the need to leverage the GPU. An improvement in runtime based on straightforward techniques for GPU Parallelization with NVIDIA CUDA was achieved. This is important because GPU resources are more powerful than CPU resources if they can be properly utilized.

Despite improved runtimes of QZ_HYBRID versus QZ_CPU, the two modes have comparable timings at the lowest number of reachability steps. The overhead of

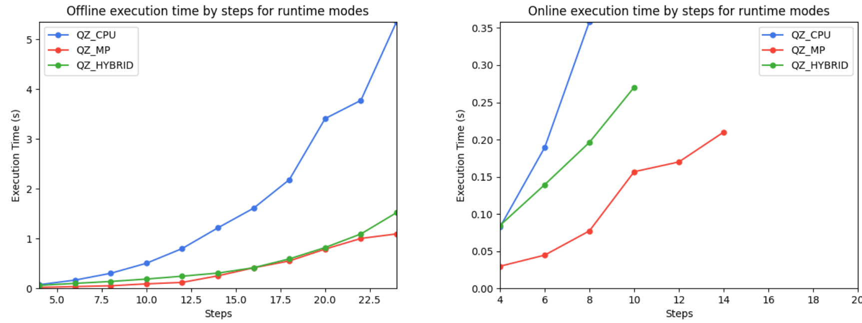

Fig. 7: Average offline (left) and online (right) runtimes for all runtime modes.

making CUDA device calls for initializing GPU memory in QZ_HYBRID may explain this.

Comparison of QZ_MP and QZ_HYBRID shows a much larger 95% confidence interval for CPU_MP in the offline mode (Fig. 9). This is unexpected given anticipated variation introduced by calling CUDA functions. This could be explained by unrelated system processes being scheduled on the 4 CPU cores at varying times during reachability execution, causing delays and inconsistency.

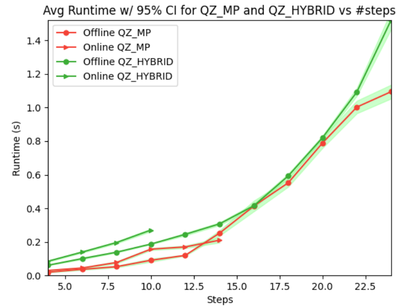

Fig. 9: Comparison between QZ_MP and QZ_HYBRID runtimes in online and offline configurations. The 95% confidence interval is indicated with in light green shading.

The standard deviation was recorded to provide insight on the variability within timings for various configurations (Fig. 8). As a percent of the total runtime, the standard deviation is relatively constant. As an absolute, the variation is also quite small. This is a good sign for real-time use cases. Low deviation in runtimes means that the controller can expect consistent feedback from the reachability computations at relatively predictable intervals. This translates into a higher confidence for responses taken based on the reachability results since they may be generated very predictably.

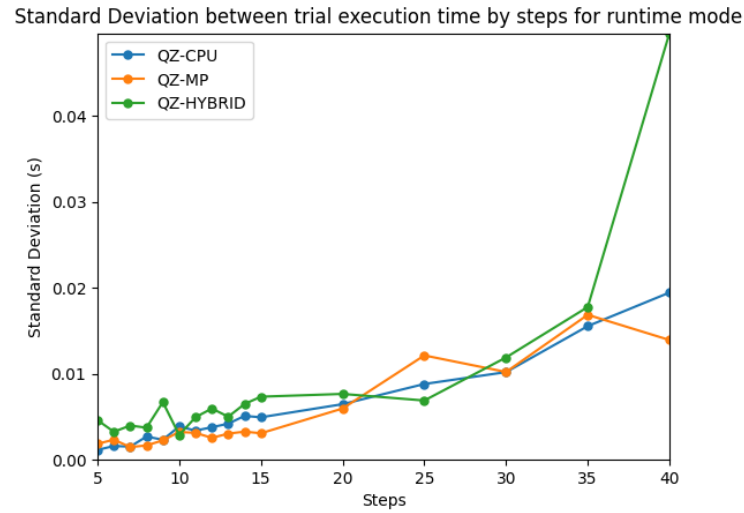

Fig. 8: The standard deviation in runtimes for different runtime modes.

6 Conclusion

It has been shown that real-time reachability can be performed at above 10hz and provide accurate results using a successively linearized model. Importantly, it is also effective to utilize parallel computing techniques to achieve these increases in performance. These developments lower the barrier to entry for real-time safety analysis by reducing the need for system identification. The promise of parallelization in this field also bodes well for the further development of more accurate and faster systems. Trivial parallelization techniques used in this research may certainly be improved and applied to larger pieces of code for performance benefit. By developing these tools on the F1Tenth platform, it will be more accessible for future researchers to play with the configuration of these tools and find improvements that contribute to the safety and reliability of autonomous systems.

References

1. Althoff, M., Dolan, J.M.: Online verification of automated road vehicles using reachability analysis. IEEE Transactions on Robotics **30**(4), 903–918 (2014)
2. Bak, S., Duggirala, P.S.: Hylaa: A tool for computing simulation-equivalent reachability for linear systems. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control. p. 173–178. HSCC '17 (2017)
3. Google Inc.: Cartographer github (2021), https://github.com/cartographer-project/cartographer
4. Lam, D., Manzie, C., Good, M.: Model predictive contouring control. In: 49th IEEE Conference on Decision and Control (CDC). pp. 6137–6142 (2010)
5. Maler, O.: Computing reachable sets : An introduction (2008)
6. Ray, R., Gurung, A., Das, B., Bartocci, E., Bogomolov, S., Grosu, R.: Xspeed: Accelerating reachability analysis on multi-core processors. In: Piterman, N. (ed.) 11th International Haifa Verification Conference, HVC. Springer (2015)