8-10-2018

A Real-Time Predictive Vehicular Collision Avoidance System on an Embedded General-Purpose GPU

Andrew Hegman

A real-time predictive vehicular collision avoidance system on an embedded general-purpose GPU

By

Andrew Hegman

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Electrical and Computer Engineering
in the Department of Electrical and Computer Engineering

Mississippi State, Mississippi

August 2018

A real-time predictive vehicular collision avoidance system on an embedded general-purpose GPU

By

Andrew Hegman

Approved:

_____

Jian Shi
(Major Professor)

_____

Michael Mazzola
(Committee Member)

_____

James E. Fowler
(Committee Member and Graduate Coordinator)

_____

Jason M. Keith
Dean
Bagley College of Engineering

Name: Andrew Hegman

Date of Degree: August 10, 2018

Institution: Mississippi State University

Major Field: Electrical and Computer Engineering

Major Professor: Jian Shi

Title of Study: A real-time predictive vehicular collision avoidance system on an embedded general-purpose GPU

Pages in Study 84

Candidate for Degree of Master of Science

Collision avoidance is an essential capability for autonomous and assisted-driving ground vehicles. In this work, we developed a novel model predictive control based intelligent collision avoidance (CA) algorithm for a multi-trailer industrial ground vehicle implemented on a General Purpose Graphical Processing Unit (GPGPU). The CA problem is formulated as a multi-objective optimal control problem and solved using a limited look-ahead control scheme in real-time. Through hardware-in-the-loop-simulations and experimental results obtained in this work, we have demonstrated that the proposed algorithm, using NVIDA's CUDA framework and the NVIDIA Jetson TX2 development platform, is capable of dynamically assisting drivers and maintaining the vehicle a safe distance from the detected obstacles on-the-fly. We have demonstrated that a GPGPU, paired with an appropriate algorithm, can be the key enabler in relieving the computational burden that is commonly associated with model-based control problems and thus make them suitable for real-time applications.

DEDICATION

This thesis is dedicated to my always-supportive and loving family: my fiancé Elizabeth, my sister Lauren, my mother Ann, and my father Albert. Without them, I would have never made it through my undergraduate or my graduate degree. To Elizabeth, it has been a long journey full of very late, very stressful nights, and I can't thank you enough for staying so supportive through all of it. To my parents, you both served as inspiration to me to follow my dreams and never give up, no matter how hard it may have been. The values you taught me as a child continue to prove invaluable today. And to Lauren, you are everything a sister should be to a younger brother and having you as someone to look up to helped make this journey that much easier.

ACKNOWLEDGEMENTS

This thesis would never have been completed had it not been for my supervisor, Jim Gafford, who never ceased to keep pushing me harder to do things I never thought possible, my major professor, Dr. Jian Shi, for helping me understand the entire concept and guiding me throughout the whole process, as well as all of the students I worked so closely with. I would especially like to recognize Tyler Hannis, Dennison Iacominni, Collin Hoffman, Raj Patel, Van Kingma, Lucas Cagle, Brandon Powell, and Andrew LeClair. The many late nights spent working on this project were made much easier with your company.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Vehicular Collision Avoidance System

It is evident that vehicles can be very dangerous, resulting in nearly 37,000 deaths in 2016 alone [1]. While full-autonomy may be several years from realizing its full potential, Advanced Driver Assistance Systems (ADAS) may be an interim solution [2]. ADAS describe a vast range of various systems, one of which being ways of improving driver safety. Modern cars offer systems such as adaptive cruise control, which is capable of maintaining a pre-determined gap based on distance to the car in front and affecting the speed of the vehicle to maintain this gap, and lane departure warning, which alerts the driver when the vehicle begins to drift out of a road lane [3]. We extend this idea by creating a Collision Avoidance System (CAS). Due to the complexity of the task, simplistic reactive controllers are not well-suited to for this and so a more intelligent control approach is used.

1.2 Vehicular Collision Avoidance System

A Model Predictive Control (MPC) technique describes a method of control where future states are predicted based on current system states, environment disturbances and potential future control inputs to the system. Fundamental elements of this approach include a numerical model that is able to estimate the future state of the system, constraints that represent practical design limitations, and physical constraints

that the system needs to be operated within. A cost function that describes the ideal outcome that the system is steered into and a solution engine to solve the resulted numerical optimization problem and determine the optimal control solution are used [4].

For the CA system we developed, we used a Model Predictive Control approach that utilizes a tree-search paradigm to find the optimal control solution as described in [5]. The MPC algorithm works by evaluating all of the possible outcomes over a set period of time given the current system operating conditions and a set of input variables known as the control inputs. These sequences of inputs are known as *input sequences*. Following the evaluation of all possible input sequences, a cost function is computed for each sequence and based on the determined validity of each sequence, the sequence with the most optimal cost is chosen and actuated. Due to the rapidly changing environment, the control solution must be recalculated every control interval.

In the scope of a MPC algorithm, the computational complexity is the main limiting factor for real-time use. To help circumvent these struggles, alternative means to traditional homogeneous computing environments are explored in the following section.

1.3 Utilizing a Heterogeneous Computing Environment

MPC algorithms can be computationally taxing on a processor. When the processor is placed in an embedded environment, with limited resources, the problem becomes even more challenging. Alternatives to traditional CPU computation of an MPC algorithm have been explored, utilizing devices such as FPGAs, or *Field Programmable Gate Array,* and GPUs, or *Graphical Processing Unit* [6, 7]. The concept of using multiple processors in tandem, specifically processors of different types, creates a system

known as a *Heterogeneous Computing Environment*. In particular, GPUs are an appealing alternative to conventional CPUs due to their high floating-point performance as well as the massive amount of parallelism. GPUs are also popular due to the fact that many computers have GPUs either built-in with the CPU itself or have a dedicated GPU that works in tandem with the CPU, albeit mostly for graphical computations. Due to the great potential that GPUs show, much research has been done in the field of GPGPU computing, or *General Purpose GPU Computing* [8, 9, 10].

1.4 Existing Work

1.4.1 Application of MPC in CAS Design

MPC algorithms have proven to be useful in a variety of contexts of control problems. For example, diesel engines are equipped with special hardware to help keep their emissions in check and within government regulated limits [11]. In a regular diesel-powered vehicle, the engine control unit (ECU) would handle the control of such devices using a rather simplistic approach ignoring key features such as the fact that these devices are tightly coupled functionally as they are both controlled by exhaust gasses. In order to take better advantage of such devices, Ferreau et al. used an MPC algorithm to better control these devices are achieve better emissions from the engine. The MPC algorithm was computed on a single Autobox by dSpace, which is powered by a 480MHz IBM Power PC processor, a homogenous computing environment. They were able to achieve a control horizon of 250 milliseconds and a prediction horizon of 5 seconds. The control horizon was determined to be 250 milliseconds due to having 5 control intervals lasting 50 milliseconds each. Due to limitations in the model, they came to the conclusion that the full potential of the MPC algorithm could not be fully realized, but their results

showed great promise [11]. It is important to note that without a proper model, the performance of the MPC algorithm is bound to suffer. The performance of the model is directly hinged on the quality of the model [12].

Alrifaee, Maczijewski, and Abel used an MPC algorithm to affect the steering angle of a vehicle on a pre-defined path to avoid a detected object on the path. A dynamics model is used to predict future positions of the vehicle and uses the current steering angle of the vehicle as the input into the MPC algorithm. They tuned the algorithm so that it favors no steering change, but if a collision is imminent, the most minimal amount of steering change is applied to avoid the collision. This method does not address the possibility of detecting objects at varying distances and does not account for speed changes. In addition, it assumes that the vehicle will stay absolutely true to the predefined path. This could become problematic if this algorithm was used in an environment where the vehicle could not be trusted to follow the pre-defined path, for example the situation of a human operator driving a vehicle [13]. While it appears that this algorithm works very well in a static environment, given a changing environment, with obstacles that may move, vehicles that change speed, and other parts of the environment that may change, the algorithm may not perform as well.

To address the concern of real-time computation, Zhou, Wang, Bandyopadhyay, and Schwager demonstrate an on-line collision avoidance algorithm, but it requires that all vehicles are aware of the relative position of all vehicles. In contrast of other existing methods, this method *only* requires position while others require more detailed information such as position in addition to velocity. The relative position is used to create

what they have called *Buffered Voronoi Cells*, or BVC. They differ from regular Voronoi Cells in that their edges do not touch, but instead of a distance between each other such that if a vehicle, in this case a robot, has its center in the Voronoi Cell, the entire robot will be inside the BVC. When calculating the future BVCs for each robot, they used a receding horizon approach [14] as did our MPC algorithm. For applications in environments such as vehicular traffic, creating a communication network so that all vehicles are aware of the relative position of others could prove to be challenging. Further, this method does not account for obstacles other than other robots. The algorithm does not account for static, or dynamic, obstacles in the environment, only other robots that can communicate their current position.

To address the issue of collision avoidance in terrestrial vehicles, Liu et al demonstrate their use of a model predictive control algorithm to operate an unmanned vehicle through a field of obstacles, minimizing travel time while maximizing vehicle safety. They utilize a LIDAR sensor to detect potential obstacles. Because the algorithm will have full control of the vehicle, one concern is tire lift off. While tuning the algorithm to minimize travel to get to the destination, it may be optimal to allow the vehicle to travel at a high speed and turn very sharply to avoid an obstacle. This could have consequences that would compromise the safety of the vehicle, such as tire-lift. To help predict when such a condition may occur, and to, in turn, prevent such an event, a model of vehicle dynamics is used. To help with the computational burden of the MPC algorithm a much simpler two DoF model is used in the MPC algorithm with a fourteen DoF model being used offline to create look-up tables for the MPC algorithm to use [15]. In a real-world implementation, multiple stages of the system will introduce latency into

the system to varying degrees. Latency could be introduced in communication channels between various controllers, computational time of the MPC algorithm, computational time of the LIDAR, etc. To account for these in their simulation, all types of delays were grouped into two different categories, sensing or control delay. Sensing delays mean that the LIDAR data is from some specified time previously and control delays mean that the solution that is calculated at a given time will not be actuated until some time in the future [15]. Their results show when the control latency exceeds 0.2s or the sensor latency exceeds 0.4s, the algorithm is no longer capable of safely avoiding the detected obstacle(s). However, if such delays are compensated for in the algorithm, the algorithm never fails to successfully prevent a collision [15].

Anderson, Peters, Pilutti, and Iagnemma demonstrate a method of collision avoidance that operates in real-time by computing a model predictive control algorithm using radar, LIDAR, vision-based lane-recognition systems along with various methods of sensor fusion to sense the environment and provide inputs to the control algorithm. Utilizing a finite-horizon approach to solve for the optimal control solution, the physical limitations of the vehicle are used as constraints for the path planning algorithm. The dynamic vehicle model ensures that the paths that are planned by the MPC algorithm keep the tires from losing traction. A method of threat detection is used to determine how much control should be given to the driver, ranging from full control to full autonomy. As threats become greater, the control gradually changes to the controller, removing control from the user. This is a valuable feature because it can be jarring to have control removed suddenly [16]. While very promising results were demonstrated, with their system proving capable of assisting a human operator to avoid a collision by altering the

angle of the vehicle's trajectory, no testing was done to modify the speed of the vehicle. The vehicle was kept at a relatively constant 50km/h. In addition, the algorithm was computed in real-time by a dSPACE processor, instead of utilizing the parallel capabilities of a GPU [16].

1.4.2 Hardware: GPGPU

Moore's Law, which states that the number of transistors on an integrated circuit must double every two years to match the performance demands of modern computer users. Throughout the lifetime of Moore's law, many have speculated the imminent doom of the law, but the law continues to remain valid. Currently, researchers and scientists are discovering new ways to ensure the continuation of Moore's law, but at some point, innovations in the current technology will begin to produce only marginal returns [17]. Currently, photolithography is predicted to reach a practical limitation by the 2020s [18]. This essentially leaves the industry with two options, either find a breakthrough in the development of manufacturing integrated circuits or look for an alternative to packing more transistors on an IC. Enter heterogeneous computing environments. The idea is to use multiple processors, particularly different *types* of processors, and have them work in tandem instead of making one single processor more efficient. In this sense, the *total* performance of the system is valued over the performance of a single processor. Traditionally, this has been a difficult task due to the difficulty in developing for both a traditional CPU and a less-traditional FPGA or GPU and spreading the workload among the devices [19].

Routers require fast processing of packets, with modern hardware achieving rates of up to 100 Gbps [20]. The alternative is to use software, but current processors cannot match the speeds of the hardware alternative. This would allow routers to handle more complex packet routing and manufacturers to offer updates on routers that would add functionality. In simple terms, a router is used to determine which port a specific packet should be sent out on given information in the packet. Given that a GPU is a SIMD device, they are very well suited for packet routing in a router [20]. To test the potential benefits of such an architecture, Mukerjee, et al. created a software router framework that can either be executed using only the CPU or both the CPU and GPU of a laptop. After several iterations of optimizations, the results were somewhat underwhelming. They explain that the CPU+GPU router architecture only achieves "meager gains" over the CPU-only architecture [20]. They go on to explain that they discovered this was due to a bottle-neck in their packet generation. They re-tested the system, using a new method of packet generation that emulated instantaneous I/O (impossible under real-world conditions) and showed that the GPU+CPU router architecture achieved three times the bandwidth and one fifth of the latency of the CPU only version. Even after they refactored the CPU-only version to utilize multiple cores, the GPU+CPU version still performed nearly 20% faster [20].

In the past few years, there has been a surge in GPGPU development in embedded applications, more specifically applications that stray away from using ultra-high end GPUs to acceleration the computations and use much lower-power GPUs instead [21]. To examine how far the embedded GPGPU development has progressed recently, Maghazeh et al. tested five benchmarks to measure relative performance increases. The five they

chose were the following: Rijndael algorithm bitcount, genetic programming, and convolution and pattern matching. Using an NVIDIA Tesla M2050, they were able to achieve a speedup of 19.1 over a single-core ARM Cortex A9 processor. They noted that they tested two versions, using two different memory access paradigms, and demonstrated that a naïve approach to memory access in GPGPU programming could yield a 9.2 speedup while an efficient, although not always intuitive, approach yielded the aforementioned 19.1 speedup. Similar results were shown for the remaining benchmarks as well [21]. Perhaps one of the most intriguing conclusions they drew was despite the speedup they achieved, an inefficient or naïve implementation can severely degrade performance.

1.4.3 Real-time MPC with GPU

MPC algorithms can be computationally intensive to calculate, especially with an increasing number of look ahead steps. In the case of an exhaustive tree search, the number of look ahead steps as well as the number of possible control solutions causes the size of the tree to grow exponentially. In an offline simulation, computation time is much less of a concern than in a real-time system. Real-time systems, as their name suggests, operate under real-world time constraints. In fact, in many control environments, a real-time system must operate *faster* than real-time so as to give other devices, such as actuators in the system, time to respond. In addition to the time that the actuators may need, additional latencies are introduced in communication delays and sensor processing. A real-time algorithm must be highly optimized as there is often little margin for overrun. The type of algorithm that is used is also an important factor that must be considered.

Automotive, as well as aerospace, factory automation, and robotics are all fields that have seen a rising amount of MPC applications and require computation of quickly changing dynamics models as well as high demand to keep costs low. In these fields, latency of the algorithm trumps the throughput of the as the outputs that are needed tend to be fewer but must be re-computed more often [22]. Yu et al. demonstrate a potential solution to this hurdle—the use of GPUs [22]. The incredible throughput of a GPU could be used to off-set the relatively higher latency compared to its single- or few-threaded relative, the CPU. To compare how much the use of GPUs actually improves their performance, they implement a single-threaded, as well as quad-threaded, application to be executed on an ARM A57 processor as well as a massively-parallel application to be executed on an NVIDIA Tegra X1 GPU, utilizing the numerous cores of the GPU. It has been shown that the GPU implementation of a generic MPC problem, with problem size greater than 300 variables, can achieve over 40x speedup compared with the single-threaded CPU version. It is also reported that GPU platform is better suited with MPC problems with slow system dynamics and larger problem sizes to meet the requirement of real-time operation. The paper concluded that "significant work on the deterministic scheduling of real-time GPU applications" is still to be done in order to leverage GPU for real-time control applications [22].

1.5 Organization

This thesis is organized as follows: the methodology of the tree-search based MPC algorithm used is explained in detail in Chapter II. Chapter III discusses the development of MPC algorithm in a parallel environment and how the algorithm was modified to utilize the parallel capabilities of a GPU. The implementation of the

algorithm on an actual vehicle is discussed in Chapter IV, including various optimizations and methods used to improve performance given real-world uncertainties and disturbances, such as latency in communication between processing units. Chapter V discusses the overall performance of the completed system and Chapter VI explores future work.

CHAPTER II

MODEL PREDICTIVE CONTROL ALGORITHM FOR A COLLISION AVOIDANCE SYSTEM

2.1 Model Predictive Control

2.1.1 The Control Inputs

Crucial to the operation of the MPC algorithm is the control inputs. This is not to be confused with the environmental inputs, which include parameters such as current vehicle speed, heading, acceleration, and detected objects. The control inputs are discrete values that are actions that the MPC algorithm can command of the VAC. In our system, these take the form of different acceleration rates. Four discrete control inputs were used, one that allows the operator to accelerate the vehicle at a maximum rate of $1 m/s^2$ which is limited by the VAC, one that allowed the operator to maintain the vehicle's current speed but *not* accelerate (acceleration must remain at or below zero), or two different negative acceleration rates that were dependent upon the vehicles current speed. The two dynamic inputs are further explored in chapter 4.

Using these four control inputs, *control input sequences* are created. The MPC system operates with five lookahead steps, so there are a total of 1,024 possible input sequences. Each input sequence is evaluated and the resulting position of the vehicle after the last timestep determines if that specific input sequence is a viable choice to use. A

cost function is used to determine the most optimal sequence given multiple viable sequences.

2.1.2 The Control Inputs

Given a list of 1,024 input sequences, each sequence must be evaluated to determine if a collision occurs with any detected object given the vehicle's heading. This is accomplished by applying the given input to the current speed of the vehicle and evaluating the resulting position of the vehicle in relation to the detected object. This process is repeated for each timestep. Because the MPC algorithm operates by simulation of future vehicle actions, it is impossible to know what actions the operator will perform in regards to heading changes. To solve this problem, a method of calculating the worst path is used.

2.1.2.1 Worst Path Planning

The nature of the CA system is to prevent collisions with detected objects. If a hypothetical situation is considered where an adversarial operator is operating the vehicle, the worst action that can be taken is the to drive directly at an object and at maximum speed. For simplicity in the evaluation of the MPC algorithm, the concept of this adversarial operator is used. For example, when an input that allows the vehicle to accelerate is evaluated, the maximum acceleration rate that is allowed by the VAC is assumed and it is assumed that the speed will remain constant if an input of zero acceleration is applied, rather than considering the chance that the operator will allow the vehicle to coast to a slower speed. Further, the heading of the vehicle will become such

that the vehicle is placed on a direct collision course with the detected object. The algorithm used to calculate the worst path is demonstrated below:

Algorithm 1 Calculate Next Heading

1: **procedure** CALCULATENEXTHEADING
2: $\Delta\phi_{max} \leftarrow v_t/(WheelBase/sin(MaxTurningRadius))$
3: $\Delta\phi \leftarrow arctan(Object.x - X_t, Object.y - Y_t)$
4: **if** $|\Delta\phi| > \Delta\phi_{max}$ **then**
5: $\Delta\phi_{t+1} \leftarrow \phi_t + (\Delta\phi_{max} * sign(\Delta\phi))$
6: **else**
7: $\phi_{t+1} \leftarrow \phi_t + \Delta\phi$
8: **if** $\phi_{t+1} > \pi$ **then**
9: $\phi_{t+1} \leftarrow \phi_{t+1} - 2\pi$
10: i
11: **else if** $phi_{t+1} < -\pi$ **then**
12: $\phi_{t+1} \leftarrow \phi_{t+1} + 2\pi$

Figure 2.1 Algorithm to calculate next heading

Algorithm Sign

if $variable == 0$ **then**
return 0
else
return $|variable|/variable$

Figure 2.2 Algorithm that returns the sign of a given number

In figure 2.1, φ represents the heading while *WheelBase* and *MaxTurningRadius* are assumed to be known when the algorithm is executed.

2.1.3 Determining the Optimal Control Solution

After all input sequences have been evaluated, a cost function is calculated for each input sequence to determine the *optimal* input sequence. The cost function is computed as follows:

Algorithm Cost

$cost \leftarrow 0$
for $input$ in $ControlInputSequences$ **do**
    $cost \leftarrow cost + input^2 * \text{sign}(input)$
end for

Figure 2.3 Algorithm to compute the cost of each control input sequence

In this manner, the optimal input sequence will be the sequence that achieves the smallest cost at the end of the control horizon. Because the actual input values may change between evaluations of the MPC algorithm due to the speed-dependent input values, the cost function must be re-evaluated every time the algorithm is computed.

Once the input sequence that has the optimal cost has been determined, the first input of the input sequence is applied to the system and the rest are discarded for this current time instant. The algorithm that is implemented in the realization of the CA System is closely based on the work of Bai and Abdelwahed in [5].

2.2 Collision Avoidance System

The CAS is capable of affecting the speed of the vehicle, but it is not capable of changing the trajectory of the vehicle by means of any method such as differential

braking or steering wheel control. At a high level, the CAS system will detect potential obstacles, then attempt to prevent a collision with these objects by means of requiring increasingly slower vehicle speeds until the vehicle is brought to a stop. To allow some margin for error, the CAS attempts to stop the vehicle approximately three or seven meters from the detected object, depending on which mode the system is operating in, namely *Beacon Collision Avoidance* (BCA) or *Front Guard Collision Avoidance* (FGCA). This distance is referred to as the Keep Out Distance (KOD). In addition to changing the KOD, the different modes are designed to detect specific objects and utilize different sensor approaches.

2.2.1 Beacon Collision Avoidance

In BCA mode, the sensor package is used to detect *beacons* (see figure 2.4). Due to the position of the LiDAR sensor that is used by the sensor package, objects that are 135° to the right and left of the front of the vehicle can be detected, with 0° being directly in front of the vehicle. Beacons that exist beyond that range are not able to be detected. Techniques to mitigate this blind spot are explored in section 4.3.1. When a beacon is detected, the MPC algorithm will assume the operator will drive directly at the beacon, assuming a worst-case trajectory, and will attempt to stop the vehicle approximately three meters from the beacon. The detection and subsequent collision avoidance of these beacons outside of the FoV that is directly in front of the vehicle is unique to this mode of operation as FGCA does not utilize this wider FoV.

Figure 2.4 An example of a beacon used in BCA mode

2.2.2 Front Guard Collision Avoidance

In FGCA mode, the sensor package is used to detect objects *only if* they exist directly in front of the vehicle. Further, unlike BCA mode, *any* object that is detected by the sensor package is considered an object that must be avoided by the MPC algorithm, such as boxes, chocks, and people. It is important to note that FGCA will continue to prevent the driver from colliding with beacons as well, as long as they are in front of the

vehicle. Because of this tunnel-vision-like FoV, severe blind spots are created for the CAS on either side of the vehicle. Consider a scenario where the vehicle approaches an object on the right side of the vehicle, as shown in 2.5. If the operator of the vehicle were to quickly turn the vehicle so that the object is now in the FoV of the CAS, the CAS would respond by quickly applying the brakes rather forcefully. This quick change could upset the balance of the vehicle and cause the vehicle to lose traction. This problem is exacerbated when the vehicle is towing several trailers. To mitigate the effects of the blind spots created by the FGCA mode, a method of a dynamic FoV was created and explored in section 4.3.2.

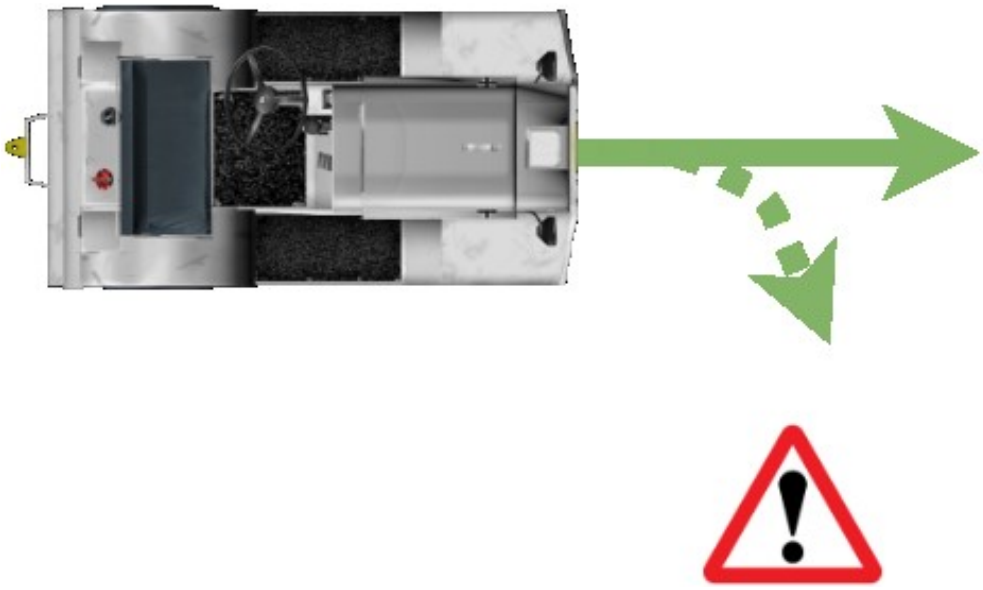

Figure 2.5 Vehicle abruptly turning towards a detected object

2.3 The Kinematics Model

Essential to the successful execution of the model predictive control algorithm is a model. The model must accurately represent the environment and the system that is attempting to be controlled. In the case of a collision avoidance system for a vehicle, a model of the vehicle must be employed. A highly detailed model can greatly improve the performance of such an algorithm. Some have modeled the suspension system of vehicles and using this information, the algorithm is capable of preventing events such as tire lift off under hard turning. Tire physics can help predict stopping distance as well as maximum lateral load which would indicate the projected maximum speed the vehicle can be turned at before losing traction [23]. A requirement that is unique to our CAS is that collisions with the lead vehicle must be prevented as well as any number of towed trailers. As such, without the possibility of using sensors to know the position of the towed trailers, the development of a model that would accurately predict the location of the trailers given known inputs such as heading and speed was an essential part of the success of the system as a whole.

2.3.1 Predicting the Location of the Lead Vehicle

The vehicle that the CAS was implemented on turns about the center of the rear axle, so it is convenient to have the model begin with the center of the rear axle located at the origin of a cartesian plane. Given the current velocity, acceleration, the change in heading, and time since the model was last calculated, the new location of the rear axle is determined by simple use of basic kinematics equations.

$$v_x = v_t * \sin(\Delta\phi) \tag{1.1}$$

$$a_x = a_t * \sin(\Delta\phi) \tag{1.2}$$

$$x_{t+1} = x_t + v_x t + \frac{1}{2} a_x t^2 \tag{1.3}$$

Equations 2.1, 2.2, and 2.3 can be similarly applied to the y-axis as well, except the equations would use cosine instead.

Now that the position of the rear axle is known, the rest of the relevant positions of the vehicle itself can be calculated. Depending on the relation to the rear axle, the distance is either added or subtracted. For example, to calculate the position of the center of the rear bumper, the distance calculated is subtracted from the position of the rear axle, whereas the center of the front bumper is calculated by adding the known distance between the rear axle and the front bumper to the new position of the rear axle. Equation 2.4 demonstrates how to calculate the position of the rear bumper's x coordinate.

$$B_{Rear_{n+1_x}} = A_{Rear_{n_x}} - d_{A_{Rear}B_{Rear}} * \sin(\Delta\phi) \tag{1.4}$$

Here, *n* represents the current object being calculated, with *n+1* representing the object in front. As was the case with equations 2.1-3, equation 2.4 can be modified to calculate the y-coordinate of the rear axle by substituting cosine into the equation. It is important to point out that the position of the front axle or the bumpers is ultimately inconsequential in the computation of the kinematics model for the purpose of the CA system, the

calculation of these intermediate points makes the calculation of the corners of the vehicle easier, and it is possible that these vehicle points could be used in future functionalities of the system.

Using the method demonstrated in equations 2.1-4, it is trivial to calculate the rest of the points of the vehicle such as the four corners of the vehicle as well as the location of the hitch, which is crucial in the calculation of the location of the towed trailers.

2.3.2 Predicting the Location of the Towed Trailers

To predict the location of the towed trailers, the most important value to determine is the angle at which the trailer has been rotated. A towed trailer will follow a tighter path around a corner than the lead vehicle. This effect is known as *off-tracking*. With more trailers, each subsequent trailer follows increasingly tighter paths than the object directly in front of it [24]. To be able to accurately predict the location of the trailers as they are being towed, it is essential that the angle between the lead vehicle's hitch and the trailer's previous rear axle position are known. This method of solving for the angle difference is the main factor that enables the high accuracy of the model.

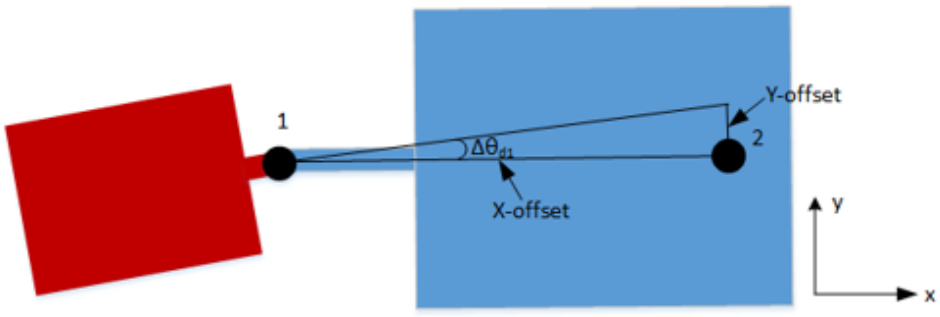

Figure 2.6 Calculation of the angle of rotation of the towed object, denoted by $\Delta\theta_{d1}$

The first time that the model is calculated, the position of the axle of the rear axle at the previous timestep of any given towed object must be initialized, given there are no previous time steps. Initially, this position is determined as follows:

$$A_{Rear}{}_{n_{x_{t-1}}} = H_{n+1}{}_{x_t} - \left(d_{HA_{Rear}} * \sin(\Delta\phi)\right) \tag{2.5}$$

In equation 2.5, *n* indicates the body, with *n+1* being in front of *n* and *t* represents the timestep. *A* indicates an axle, and the subscript indicates *which* axle, front or rear. *H* is the position of the hitch, and d_{HA} is the distance between the hitch and axle. Following this initialization, the angle of rotation of the trailer is calculated as follows:

$$\theta = \arctan 2\left(\left(H_{n+1_{x_t}} - A_{Rear}{}_{n_{x_{t-1}}}\right), \left(H_{n+1_{y_t}} - A_{Rear}{}_{n_{x_{t-1}}}\right)\right) \tag{1.6}$$

In equation 2.6, arctan2 computes the inverse tangent of the first parameter, divided by the second and determines the correct quadrant depending on the signs of the arguments [25]. Once this angle is computed, the determination of the location of the trailer is carried out in a similar manner to the way that the location of the corners of the lead vehicle was calculated, however, this new angle is used instead of the change in heading.

Once the location of the rear axle of the towed object is determined, that location is saved to be used in the next computation of the model. In this way, the initialization step is only necessary for the first time that the model is computed.

2.3.3 Maintaining a Relative Frame of Reference

The sensor package reports detections using polar coordinates, in the form of an angle in meters possibly ranging from -180° to +180° (although the actual angles that are reported are in a smaller range, due to the position of the sensor), and a distance in meters, centered about the center of the front bumper of the vehicle. To prevent the positions of the vehicle from theoretically approaching +/- infinity, due to the vehicle travelling very far from the original location, and to simplify the arithmetic needed to compare the location of the vehicle and trailers to a detected object, various methods are used to maintain a local relative frame of reference, placing the vehicle's rear axle at the origin of the cartesian plane.

The model works by allowing the lead vehicle to propagate through space and measuring the new angle of the towed trailers given the change in heading of the lead vehicle. As such, during the computation of the model, the vehicle's position deviates from the starting position of (0, 0), but will be translated back to that position at the conclusion of the computation of the model. To translate the vehicle back, the distance traveled from the starting location is calculated. In practice, this is simply a calculation of the Pythagorean distance between the origin and the current location of the vehicle's rear axle. Because the starting position can be assumed to always be (0, 0), the Pythagorean distance calculation is not necessary. All positions of all vehicles are translated by this amount, so that the entire system is translated correctly as shown in figures 2.7-9.

For simplicity of calculation, the vehicle model uses locations based on a cartesian plane. To maintain consistency, the location of detected objects are converted

into cartesian coordinates and placed in the cartesian plane along with the vehicle and trailers. Due to the naivety of the sensor package, it becomes imperative that the model not only be translated to the origin, but more importantly be rotated so that it faces along the positive y-axis of the cartesian plane. This ensures that the locations of the detected objects are placed correctly in the cartesian plane relative to the full vehicle system. As was previously stated, it is essential to the model that the vehicle and all of the trailers propagate through the plane before being translated back to the origin, and the rotation is no special case. It too must occur after all positions of the vehicle and trailers are determined. The following expression demonstrates how the rotation is performed:

$$x' = (x * \cos(\Delta\phi)) - (y * \sin(\Delta\phi)) \tag{1.7}$$

$$y' = (y * \sin(\Delta\phi)) + (y * \cos(\Delta\phi)) \tag{1.8}$$

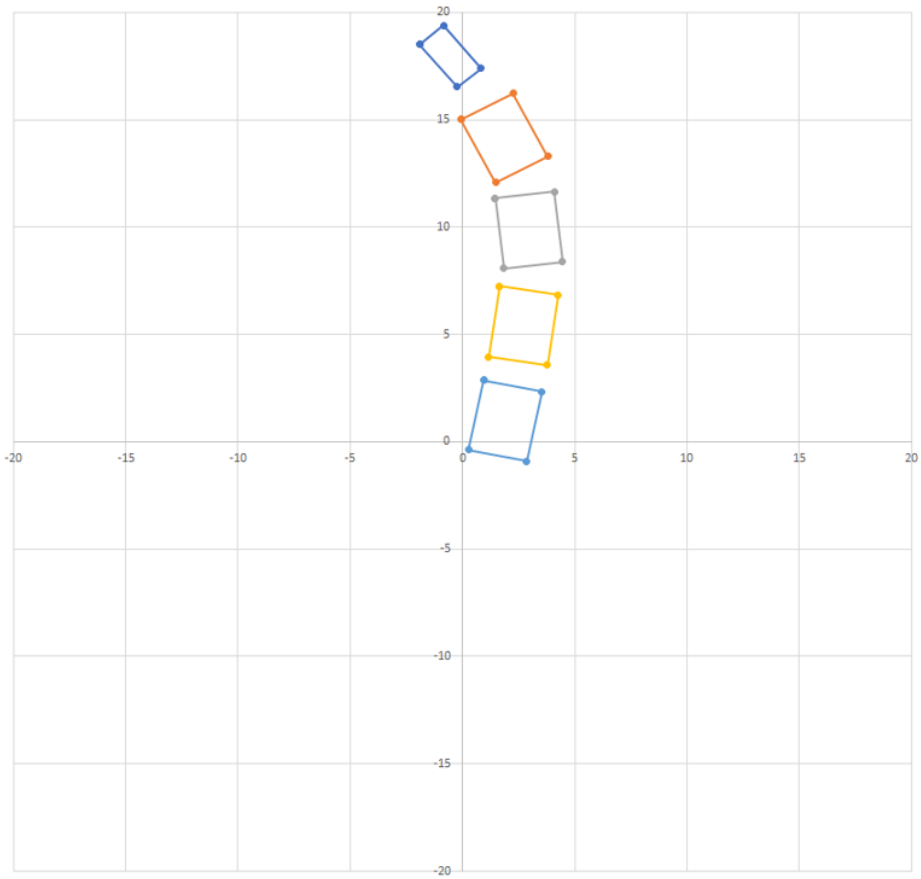

Figure 2.7 Image of vehicle with four trailers before the system has been centered and rotated

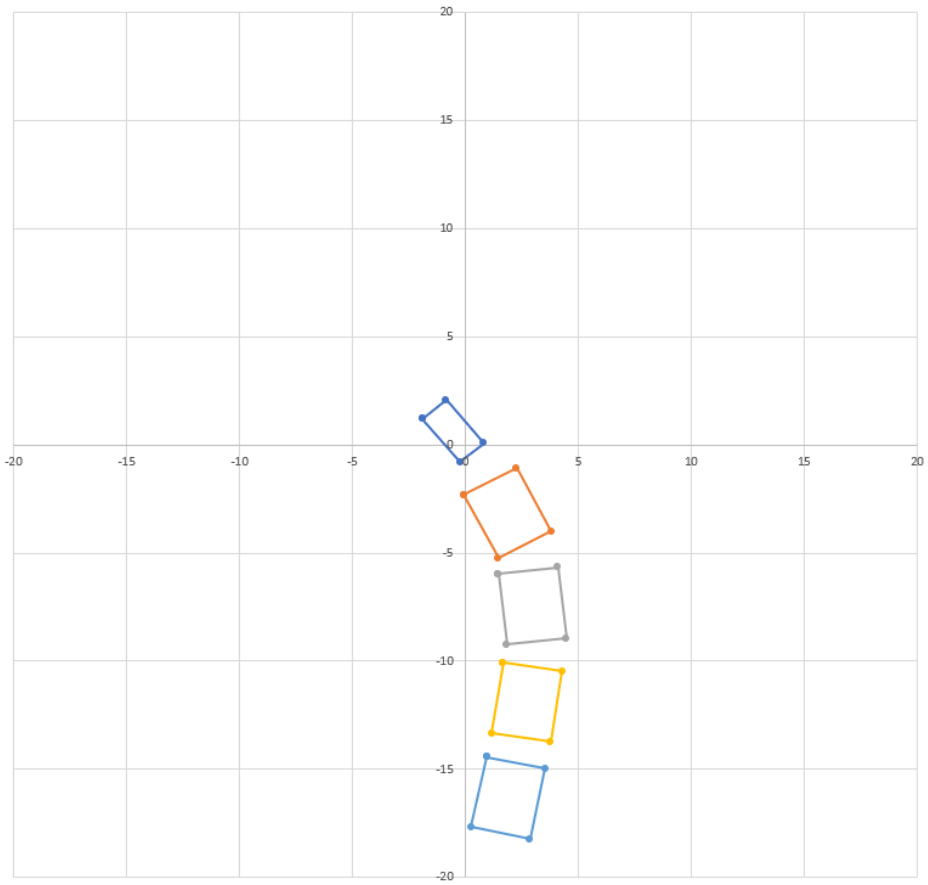

Figure 2.8 Image of the vehicle with four trailers before the system has been rotated but after the system has been centered

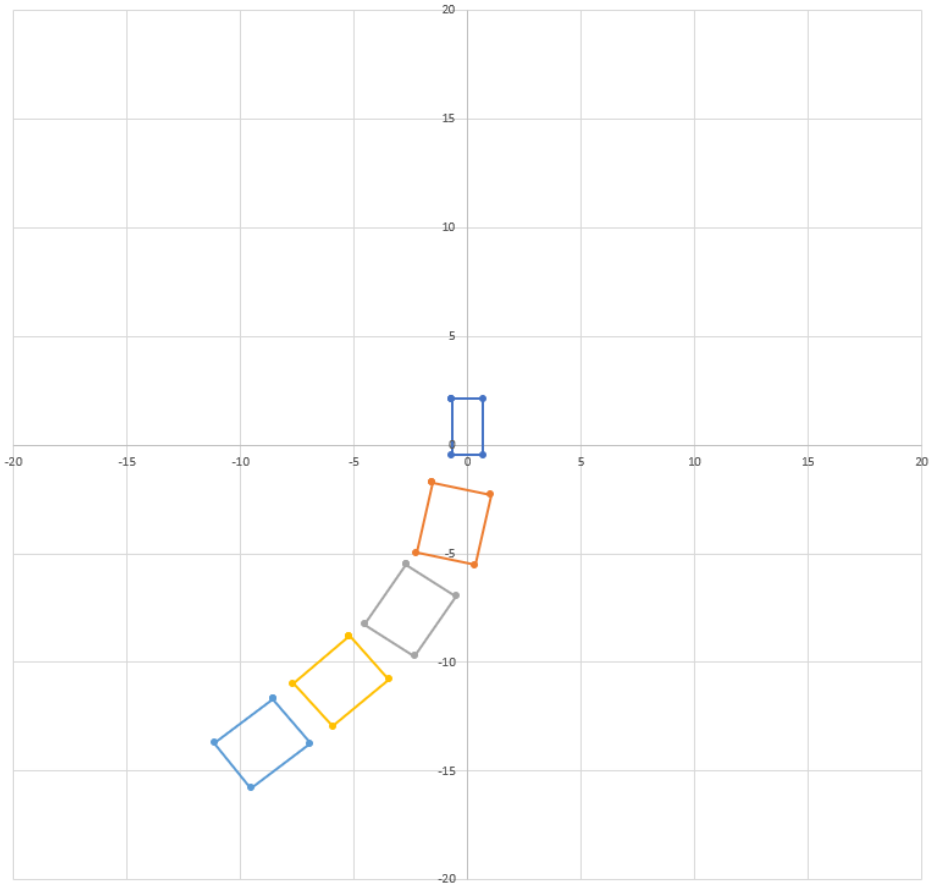

Figure 2.9 Image of the vehicle with four trailers after the system has been rotated and centered

CHAPTER III

PARALLEL MPC ALGORITHM DEVELOPMENT

3.1 Background on Graphical Processing Units

Graphical Processing Units were developed due to the need of a greater computational power demanded by consumers of their computers. The first computers that supported a graphical interface used the Central Processing Unit, or CPU, to render both the graphics as well as perform the complex calculations that are expected of a computer. As time went on, graphics became more demanding but CPUs were typically able to keep up. At some point, the duty of graphics rendering was off-loaded to a specialized processor known as a GPU. The purpose of a GPU is solely to render graphics, and to do it quickly. With this new architecture, the CPU could continue its normal operation and the GPU would handle all of the graphical operations. Many modern computers offer CPUs that have graphics rendering capabilities built-in to the CPU, meaning a discrete GPU is not necessary, however optimal performance is only achieved when the two devices are separated.

1984 marks the year in which IBM first developed a processor-based video card for the PC. This meant that the processor on the video card could handle all video-related tasks allowing the CPU more computational power [26].

GPUs attain their unique performance advantages over CPU by virtue of their vastly different underlying architecture. Firstly, it is important to note that GPUs have been developed with different goals in mind than a CPU, namely they are developed to favor throughput over latency, whereas CPUs tend to favor the opposite. While both processors utilize forms of pipelining, they do so in different fashions. Given a stream of data that must go through several different stages of computations, the CPU would operate on the data by executing the first stage of the pipeline using the first element of data, then feeding the output of that data into the next stage of the pipeline. GPUs accomplish the same effect of parallelism by performing the same operation of the first pipeline stage on all elements at once before continuing onto the next stage of the pipeline. The overall result is a high throughput, with the CPU operations taking many fewer clocks to finish the calculations but outputting data in small chunks, or even single elements, at a time, whereas the GPU may take many more clock cycles to run but outputting all elements of the data simultaneously [27].

Given the huge potential for high-throughput computing, the idea of a programmable GPU was very appealing. In 1999, NVIDIA introduced the first GPU that was programmable using DirectX7 and OpenGL. The problem was that programmers were required to learn a graphics programming language instead of C/C++ in order to program the GPU [28]. In 2006, NVIDIA changed all of this when they released the GeForce 8800 which was the first GPU to support C and CUDA. This meant that programmers no longer needed to learn programming languages unique to graphics development, to take advantage of the GPU's horsepower as a computational platform [28].

3.1.1 NVIDIA CUDA

Since the introduction of the GeForce 8800, the most efficient way to utilize the parallelism offered by GPUs was to use NVIDIA's CUDA. The programmer writes *kernels* which look like conventional single-threaded CPU functions. The difference is that the kernels are executed across many threads in parallel using the SIMD paradigm. These threads can be further organized into blocks, each block having up to three different dimensions of organizational structure. Moving further beyond blocks, blocks are then organized into structures known as *grids*, which also offer up to three dimensions of organization. The specific sizes of threads per block, threads per block dimension, blocks per grid dimension, etc. is fully dependent on the specific architecture of the. The GPU that is used in this system is capable of executing 1,024 threads per block with four blocks simultaneously. Figure 3.1 shows the organization of threads, inside a block, inside a grid, all of which are of two dimensions.

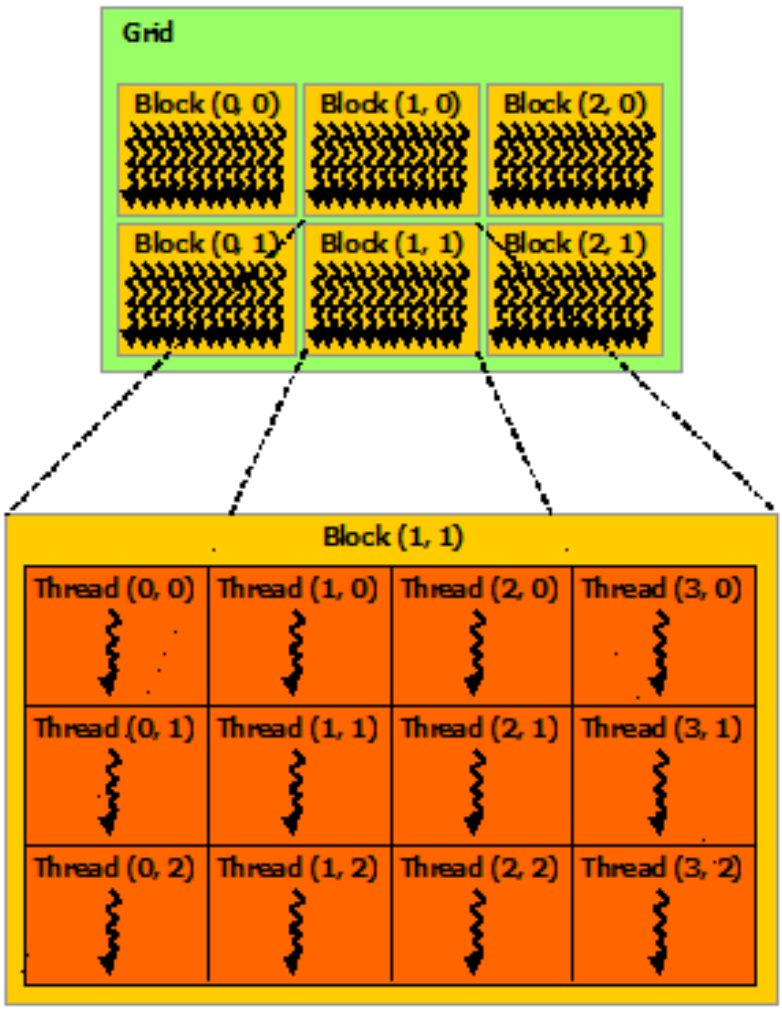

Figure 3.1 Organization of grids, blocks, and threads in CUDA [29]

When the kernel is declared, the programmer specifies how many blocks should be used and how many threads should be in each block. Each thread executes an instance of the kernel and is given its own unique thread ID so that individual threads can be identified in the kernel. In our development, we noticed that memory allocation can be a time-consuming operation so it is very beneficial that special attention is payed to this area

during the development phase to ensure that memory allocation occurs as sparingly as possible.

3.1.2 CUDA Memory Model

One of the most important concepts to remember when developing in CUDA is the memory model. *Global memory* is accessible by all threads across all blocks and is even accessible by the CPU, through memory transfers. This memory can have excessively long latencies and can be very detrimental to the performance of the program if the kernel must read and write to this memory space frequently. It is the duty of the programmer to ensure that global memory accesses are limited, favoring shared memory access where appropriate [30].

While all threads have access to global memory, only threads in the same block have access to *shared memory*. Shared memory is expected to be very low-latency and close to each processor core. Because it is visible to all threads in a block, special attention must be paid to prevent race conditions [30].

Finally, each thread has its own set of memory that is only accessible by that specific thread, known as *local memory*. Data that is declared in the kernel is considered local memory. The compiler will decide whether to allocate the memory into registers or into the local memory space, which is the same memory space as global memory and so latency can be very high with low bandwidth [30].

Regarding memory accesses, when threads must access global memory, ensuring that consecutive elements of an array are accessed can significantly improve performance

as this allows the GPU to coalesce the memory reads which reduces the total number of memory access operations, improving the overall performance [29]. Figure 3.2 demonstrates the relationship between the different scopes of memory in CUDA.

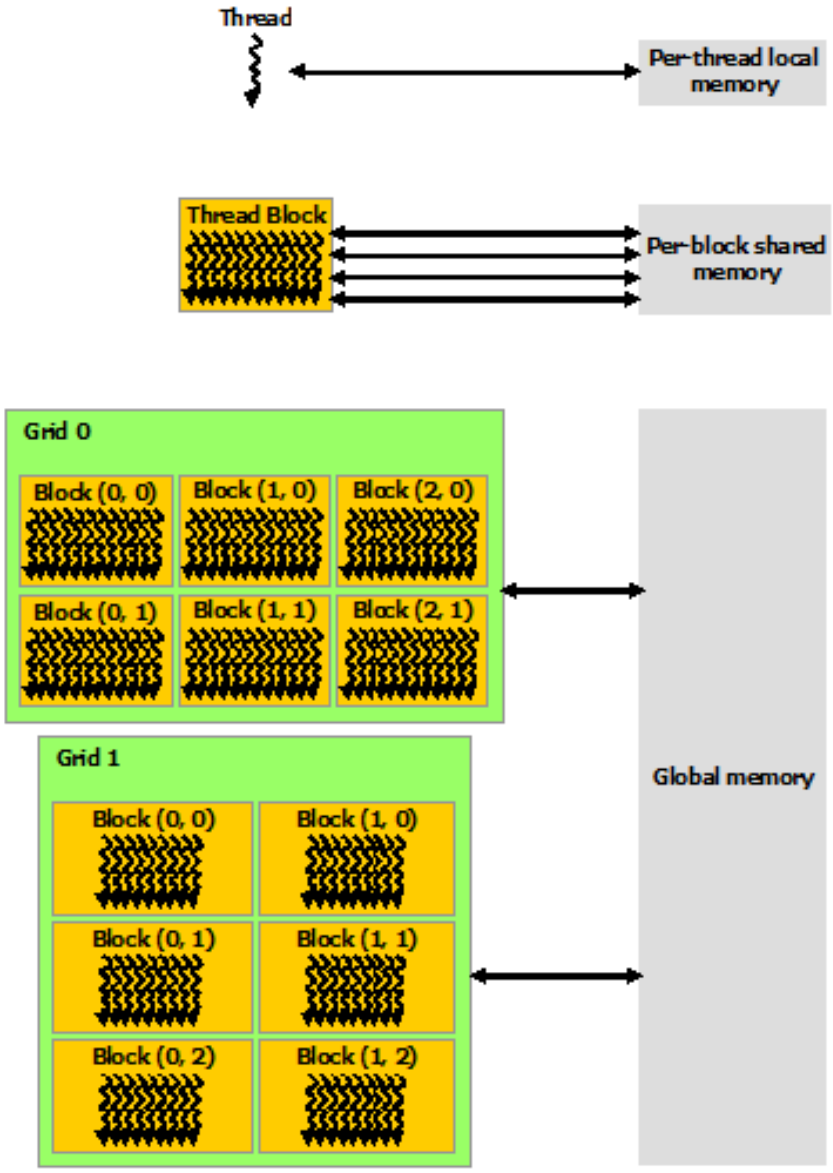

Figure 3.2 Illustration of the organization and scope of memory in CUDA [30]

3.1.3 Types of Memory in CUDA

Making intelligent design decisions on the type of memory that should be used is critical for optimizing performance. This section is not meant to be an exhaustive description of the various types of memory that are available to a CUDA developer, rather it serves to explain some of the nuances of the types of memory and aims to explain why certain types of memory were chosen over other types in the implementation of the MPC algorithm.

Page-locked host memory is memory that is guaranteed to not be available to the system for paging. In other words, this memory is guaranteed to not be swapped out, ensuring faster memory transfers. Page-locked memory makes concurrent kernel execution and memory transfers possible, whereas pageable memory does not offer this advantage. Allocating memory as page-locked decreases the amount of memory that the system has for paging. Decreasing the total amount of pageable memory can have detrimental effects on the performance and memory allocations for page-locked memory can start to fail much sooner than memory allocations for pageable memory.

NVIDIA GPUs utilize the SIMD paradigm to execute threads in parallel. This means that optimal efficiency is achieved when the threads in a *warp* all execute the same instruction. If any conditional branches cause some threads to execute different instructions than another thread in the same warp, parallelism is broken. The SM must then execute one branch and then execute the next branch. This phenomenon is known as *thread divergence* and can have severe consequences on the performance of GPU applications.

3.2 Implementing the MPC Algorithm using the GPU

In order to ensure that the CAS algorithm is executed in time, efficiency is very critical. The nature of the tree search algorithm lends itself to a parallel execution idea as all input sequences can be executed independently of each other. Each branch of the tree will represent the evaluation of one input sequence and will be executed by one thread of a thread block. Each detected object will have its own set of 1,024 input sequences, so each detected object will utilize one thread block, each utilizing the maximum 1,024 threads of the block. This section explains the details of how the MPC algorithm was developed to utilize the parallel processing capabilities of the GPU.

To determine the number of threads per block, the limitation of the GPU is considered. For our system, we are using NVIDIA's Jetson TX2 platform, which allows a maximum of 1,024 threads per block. In formula 3.1, the number of branches of each tree is found using the following expression where N is the number of lookahead steps and U is the number of inputs:

$$Number\ of\ Branches = N^U \tag{2.1}$$

When the algorithm was first developed on the GPU, it generated a two-dimensional array of input sequences and copied this array to the GPU and each thread would need to access this global memory. All of these memory transactions caused a severe degradation in performance and so a new method was developed so that each thread could dynamically calculate the appropriate control input.

Each thread in each block relates to a specific input sequence. To simplify the execution of the MPC algorithm, an array of the control inputs is created. Using the current thread's index, the current timestep that is being evaluated, and the following expression, a number between zero and the number of control inputs minus one is returned. This is then used to index the array of control inputs.

$$floor\left(\frac{i}{U^{N-n-1}}\right) \% U \tag{2.2}$$

In equation 3.2, *n* represents the current timestep, *i* is the index of the current thread, the % represents the modulo operation, *floor* rounds the result of the expression inside the parenthesis to the *down* to the nearest integer.

Each thread utilizes a for-loop that loops the number of look ahead steps that have been specified. The control solution that pertains to that thread and that timestep is determined and using the current speed and heading (local to that thread), the kinematics model is calculated. After the model is finished, four corners of each body are compared to the detected object. If any corner of any body is inside the KOD, that specific input sequence is considered invalid.

3.2.1 Optimizing the MPC Algorithm

Because the algorithm must be computed in real-time, efficiency and computation time are of the upmost importance. As such, special attention to detail has been paid to implement optimizations which seek to minimize the execution time of the MPC algorithm as a whole.

The method used to determine the optimal control solution after all threads have been evaluated was developed using two different approaches. The first approach was used as a benchmark and was known as the *naïve* approach due to its naivety in its method of processing large arrays. The second approach utilizes NVIDIA's *Thrust* library which offers GPU-accelerated performance to common operations such as finding the largest element in an array.

3.2.1.1 The Naïve Approach

Figure 3.3 demonstrates the *naïve* algorithm that was developed as a bench mark to compare the performance improvement of using NVIDIA's *Thrust* library.

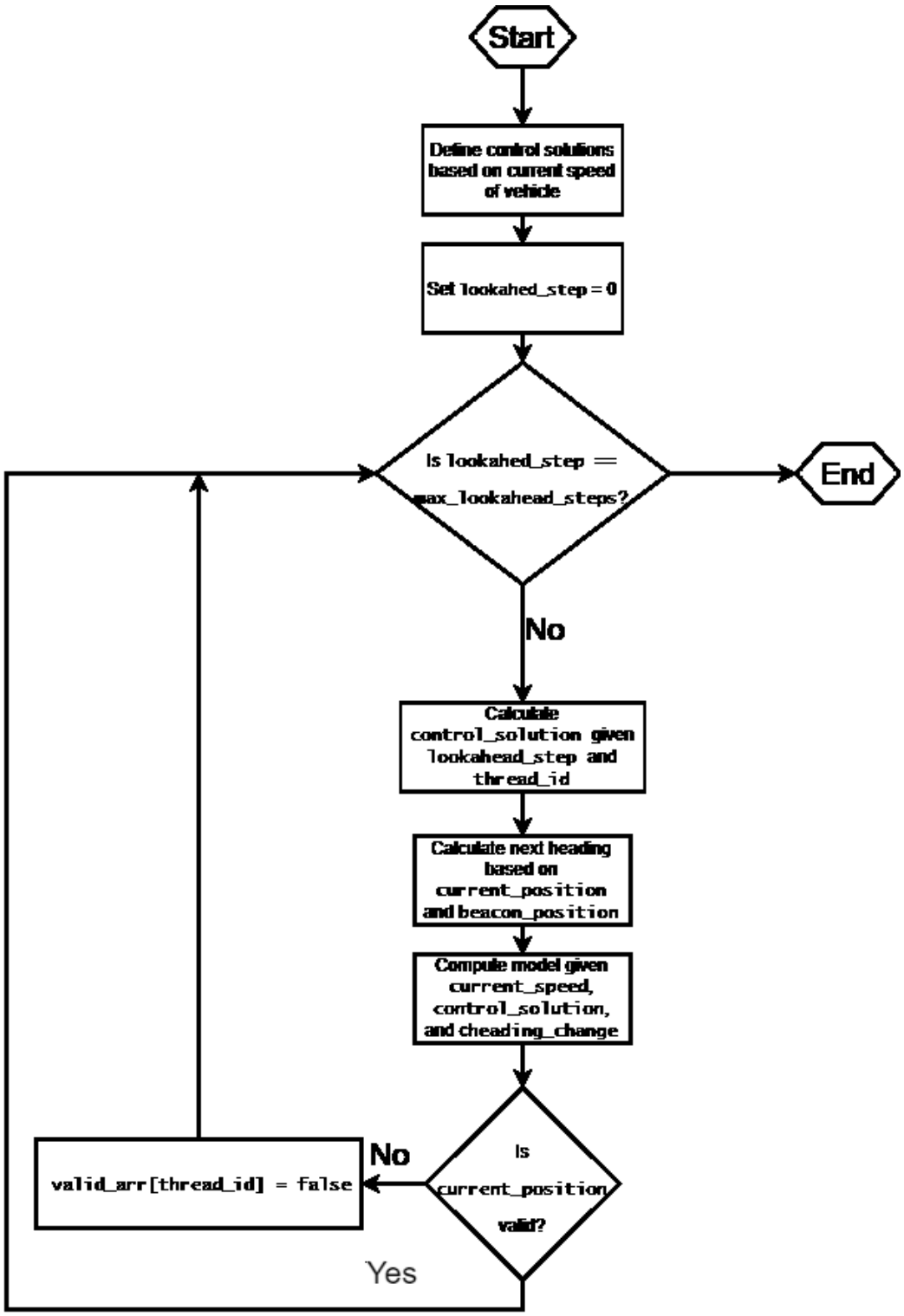

Figure 3.3 MPC Algorithm using the naïve approach

The *naïve* approach created a multi-dimensional* array of booleans, with the number of rows equal to the number of detected objects, or the number of thread blocks, and the number of columns equal to the number of control solutions to be evaluated. In the case of four inputs and five lookahead steps, there are 1024 input sequences. If a certain index of the array is *true*, the input sequence that relates to that index is considered *valid* whereas an index that is *false* indicates that that input sequence is *invalid*. The so-called validity of an input sequence is somewhat dependent on the current mode of operation of the system as the KOD for each mode can differ, but in general, it indicates that given the vehicle parameters, i.e. given the current speed, the current heading, and the *speed-dependent* control solutions (with an absolute maximum of -0.5m/s^2), the vehicle will not be able to stop without passing the KOD.

Following the execution of the GPU kernel, the multi-dimensional array must be transferred from the GPU's memory to the CPU's memory, or from the *device* to the *host*. To help with this, page-locked memory was allocated which allows the host and device to directly access device memory, meaning explicit memory transfers between the host and device are not necessary. Due to the relatively small size of the array being allocated, it is safe to use this method of memory allocation, however, with larger memory allocations, the overall performance may degrade as the overall memory that is available to the system for paging is reduced [30]. Following the memory transfer, each row of the array was searched. If the array element was *true* and it passed the pruning filter, the utility of the input sequence was iteratively calculated. The pruning filter will be discussed in greater detail in in a later section of this chapter. The index that produced the maximum utility was saved for each row of the array. Due to the pruning filter, the

optimal input sequence may not always be the lowest indexed input sequence. Once the entire array had been processed, a number of input sequence indices would result, equal to the number of blocks, or beacons processed by the GPU kernel. The utility of the resulting indices was then compared against each other and the optimal utility is chosen. This time, however, the index that produces the *worst* utility is the most important, as this indicates the worst-case scenario. Each row represents the path to a detected object. If the vehicles speed must be limited to a greater degree if the vehicle's path turns towards a specific beacon, the controller must abide by this control routine, even if no planned trajectory would require such a control response.

3.2.1.2 Using NVIDIA's Thrust Library

Figure 3.4 illustrates the improved algorithm that utilized the Thrust library.

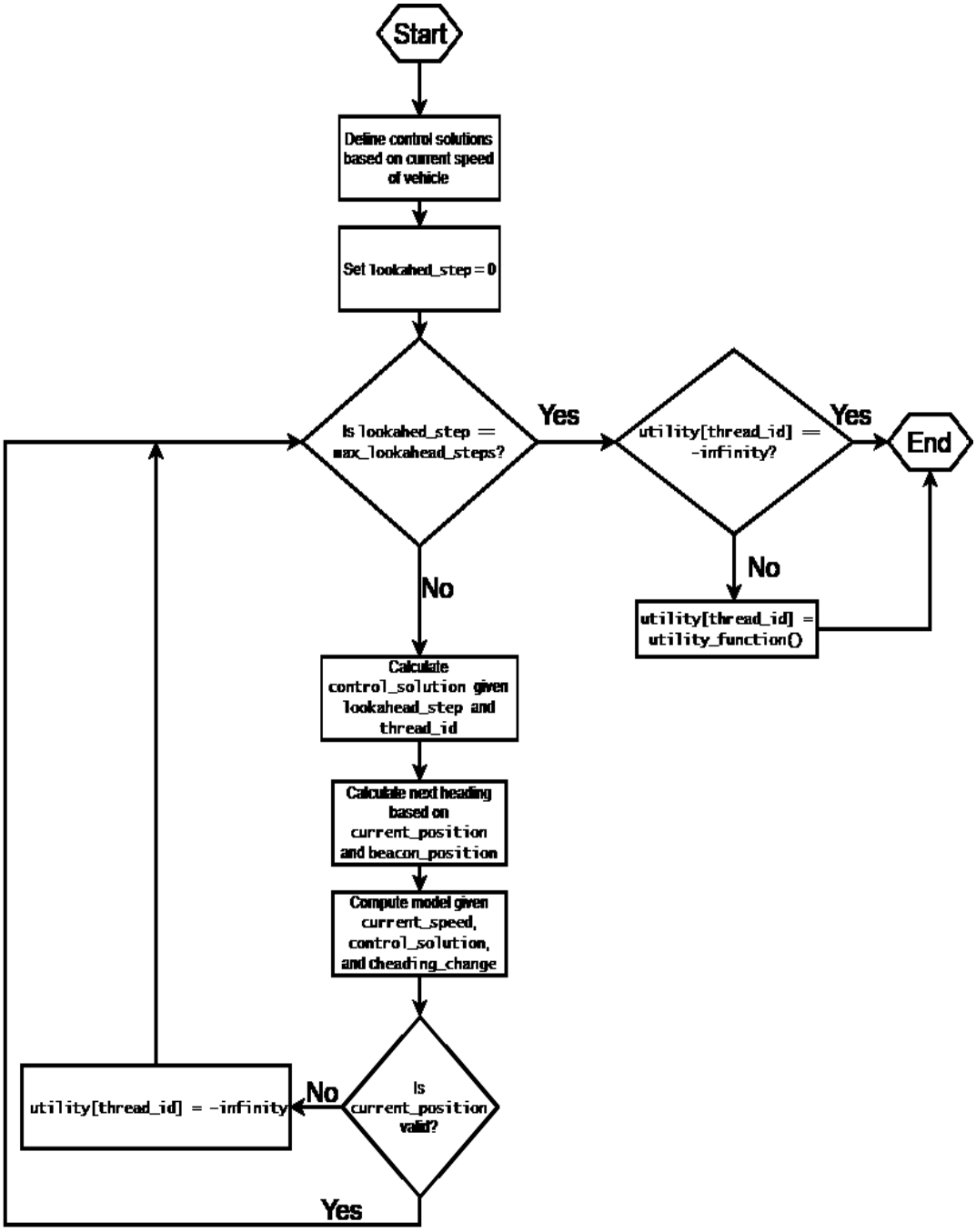

Figure 3.4 The MPC algorithm using the Thrust library

To utilize NVIDIA's *Thrust* library, the MPC algorithm must be slightly modified. The main algorithm need not be changed, however. The array which indicates

the validity of each index of the input sequences is now an array of floating point numbers rather than boolean values. Further, the method that determines whether the input sequence is valid or invalid must no longer set the index of the array to *true* or *false*, rather it has the ability to set the current index to negative infinity in the event of the input sequence being invalid. The specifics of this algorithm, including our techniques to optimize its performance and avoid thread divergence are explored in this chapter and in chapter 4.

After all lookahead steps have been evaluated by a thread, if the index of the array, which is unique to the thread, is *not* negative infinity, then the utility of the input sequence is computed and stored in the array.

Following the completed execution of the MPC algorithm on the GPU, an algorithm that follows the same basic idea of the naïve approach is executed. The goal is still to find the optimal control solution for each column of the array, or for each block or beacon, then to find the worst control solution of those. Instead of explicitly transferring the memory from the device to the host and then performing an exhaustive search on the array, NVIDIA's *thrust* library offers a method called *max_element* which takes a pointer to the beginning of an array in device memory and a pointer to the end of an array in device memory and returns a data type that can be used to resolve the index that had the best utility. This is performed on each row of the array. The resulting optimal input sequence indices are then compared, and the worst index is chosen as the control solution. The results of utilizing *thrust* over the naïve approach are examined in chapter 5.

3.2.2 The Pruning Filter

The concept of the MPC Tree Search algorithm is to examine *every possible* combination of inputs, even ones that are infeasible or impossible. Due to the way that the MPC algorithm is executed on the GPU, it is not feasible to apply this pruning filter on the tree before execution, but it is beneficial to prune the tree after execution when determining the optimal control solution given the current environmental parameters.

A simplistic filter was developed and applied. A more complex filter may be more beneficial and may offer better performance, this is explored further in section 6. Figure 3.5 demonstrates the pruning filter algorithm.

Algorithm Prune Input Sequence

```
IsPositive ← false
IsNegative ← false
IsValid ← false
for Step in LookAheadSteps do
    if ControlInput > 0 then
        IsPositive ← true
    else if ControlInput < 0 then
        IsNegative ← false
end for

if IsPositive = true and IsNegative = false then
    IsValid ← false
else
    IsValid ← true
```

Figure 3.5 Pruning filter algorithm

The effect is to restrict valid input sequences to only those that have only *positive* control solutions and control solutions equal to zero for all timesteps *or* input sequences

that have only *negative* control solutions and control solutions equal to zero for all timesteps. This prevents the MPC algorithm from being able to choose an input sequence that allows the vehicle to accelerate for the first timestep, or first few timesteps, but then requires the vehicle to suddenly decelerate in the next timestep, or later timesteps. This type of input sequence proves to be problematic as the MPC algorithm must be re-computed before the control solution for the next timestep is executed due to the quickly changing, and often unpredictable, environment.

3.2.3 Avoiding Thread Divergence

After a single lookahead step has been evaluated and executed in the MPC algorithm, the current distance to the detected object is computed. If the distance is less than the currently determined threshold, the sequence of inputs is considered invalid. Thread divergence occurs when two threads "diverge" due to the control path changing due to a conditional branch. To avoid the possibility of thread divergence, each thread continues to execute the remaining time steps even if it has already been determined that that specific sequence is considered invalid.

The method which computes the distance to the detected object and determines the validity of the input sequence is structured in such a way that it is only able to modify the validity variable if the validity goes from *true* to *false*. In this way, it becomes trivial to ensure that the algorithm begins with the assumption that all input sequences are *valid*, and hence can only become *invalid* or remain *valid*. By doing this, no conditional branching is needed in the method which determines the validity of a sequence of inputs and thread divergence is avoided.

CHAPTER IV

IMPLEMENTATION OF THE CA SYSTEM ON A VEHICLE

4.1 The Sensor Package

For the CA system to function appropriately, an accurate system of sensors is imperative. For our system, a combination of sensors along with sensor fusion is utilized to improve the sensing capabilities.

4.1.1 LiDAR

Of the two types of sensors that are used, the first type that will be discussed is the Light Detection and Ranging, LiDAR, sensor. LiDAR sensors have been used extensively in autonomous vehicle research and have proven to be very accurate in object detection. The specific LiDAR used is the Quanergy M8, offering eight evenly spaced vertical beams. While the sensor performs very well in object *detection*, it often fails to show similar performance in object *classification*. Furthermore, short objects can sometimes prove to be difficult to detect due to the sparse resolution of the sensor.

The LiDAR sensor utilizes a Support Vector Machine, SVM, to identify the objects it detects, most notably beacons. When clusters of points are returned from the LiDAR, the SVM is used to determine whether the detected object is a beacon based on a set of features that can be extracted from the cluster of points. The threshold of the SVM can be tuned to be conservative, thus accepting more *false positives* in exchange for

minimizing *false negative*. Conversely, it can be tuned to be more aggressive, rejecting false positives, but potentially also introducing false negatives. If the SVM is tuned to be very aggressive, thus rejecting most false positives, the likelihood that a true positive will be rejected becomes higher. In this scenario, the CA system is doomed to fail. As such, in the presence of only a LiDAR sensor, it is better to have the SVM tuned much more conservatively.

4.1.2 Monocular Cameras

In addition to the LiDAR, two monocular RGB cameras are used. Unlike the LiDAR, the cameras offer much higher resolution images as well as color images, which greatly improve the classification abilities. Unlike the LiDAR, however, the cameras are not able to accurately measure distance as well. In some cases, if the size of the object is known, the distance to the object can be inferred. This is based on the fact that distant objects appear smaller than objects that are closer. For many objects, such as people, size may vary significantly, rendering this distance estimation method useless.

4.1.3 Sensor Fusion

These sensors are used in tandem to improve the overall performance of the sensor package. This method of cooperatively using a heterogeneous set of sensors is known as *sensor fusion*. For the area where the FoV of the LiDAR and the cameras intersect, the decision threshold of the SVM for the LiDAR can be made to be much more aggressive, as the cameras are able to help identify beacons and other objects and reject false positives. For the rest of the FoV of the LiDAR, a much more conservative decision threshold is used for the SVM so as to prevent false negatives.

4.2 Mitigating the Granularity Problem

The number of input sequences is correlated with the number of inputs and the number of lookahead steps. Because only a limited number of threads could be run on the GPU at one time, the optimal number of lookahead steps and inputs must be determined. We chose five lookahead steps and four inputs as this gave us the most lookahead steps while allowing us enough inputs to operate efficiently. However, with so few inputs, we experienced a problem that we called the 'granularity' problem.

The MPC algorithm could be computed with a maximum of four discrete inputs, meaning the control solution could be one of four different values. It was determined that one of the solutions should be to allow the vehicle to accelerate, otherwise the operator could be put in such a state that the MPC algorithm would never let them accelerate. Another should be a control solution that instructed the VAC to not force the driver to decelerate but not let the driver accelerate, effectively limiting the vehicle to a maximum acceleration of $0 m/s^2$. In this case, the driver could either hold their current speed, or decelerate by use of allowing the vehicle to coast or applying the brakes. Given these conditions, the MPC was limited to only two solutions that would command the vehicle to decelerate. This made it difficult for the MPC to *smoothly* control the vehicle. The following sections explain how we attempted to improve the performance of the CA system with the limited number of inputs.

4.2.1 Dynamic Control Solutions

To mitigate the effects of the so-called 'granularity' problem, control solutions that varied according to the current speed of the vehicle were developed. The following expressions represent the formula that is used to calculate the possible control solutions:

$$u = \left(u_{max} * \left(\frac{v_0}{v_{max}}\right)^2\right) \tag{3.1}$$

In equation 4.1, u is the control input of the current timestep and u_{max} is the maximum value this input value should achieve given that v_0, which is the current velocity of the vehicle, is equal to v_{max}, which is the maximum attainable speed of the vehicle. In our system, v_{max} is determined by the VAC. Figure 4.1 demonstrates the effect that different u_{max} values have on the resulting control inputs.

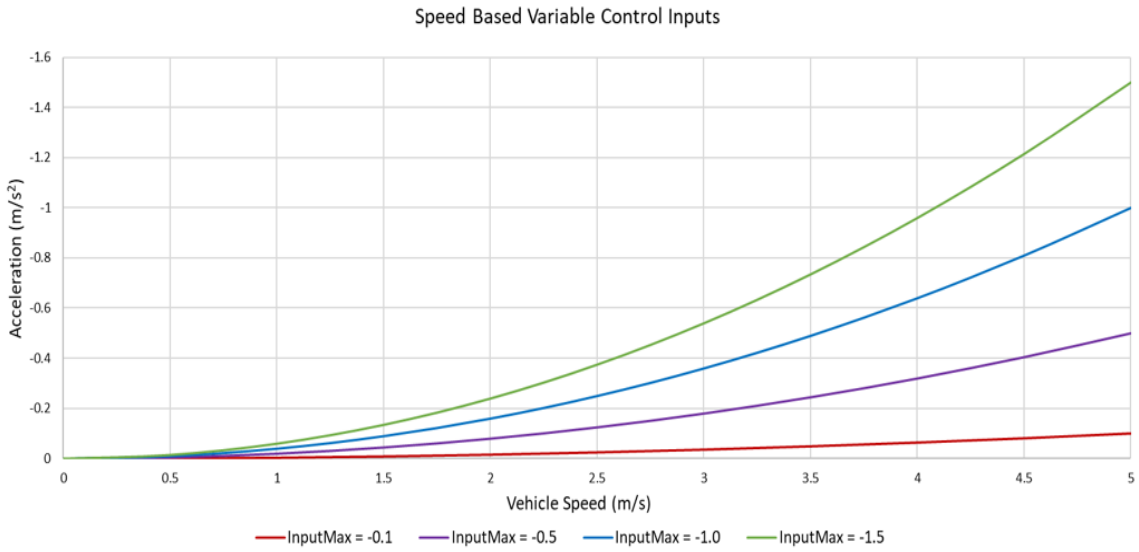

Figure 4.1 This chart demonstrates how the u_{max} parameter can affect the control input values

As the current speed of the vehicle increases, the aggressiveness of the control solutions becomes exponentially greater. This allows the VAC to apply a gradual amount of braking that will decrease at an quadratic rate as the vehicle speed slows down. Given a constant amount of mass and a decreasing rate of deceleration, the *force* required to slow the vehicle down becomes increasingly less. When towing several trailers, or under poor road conditions, this gradual slow down becomes imperative as an abrupt stop can cause the vehicle to lose traction and the driver could be seriously injured.

4.2.1.1 Creating a Floor for the Control Inputs

Adding more complexity to the formula used to calculate the input values allows finer control over the performance of the overall CA system. The following expressions show the addition of a new parameter, the $input_{min}$.

$$u = (u_{max} * \left(\frac{v_0}{v_{max}}\right)^2 + u_{min} \tag{3.2}$$

The u_{min} is a very useful parameter and is simplistic in nature. While the speed-dependent control solutions do help to mitigate abrupt stopping, at sufficiently low speeds the control solutions become so small that the VAC is not capable of achieving such deceleration rates. To alleviate this problem, a constant scalar value is added to the control solution. This creates a so-called *floor* for the control solutions as no control solution will be able to achieve a deceleration rate less than the floor value. Further tuning could be implemented by applying some logical expression to the floor value, making it variably dependent on the current speed of the vehicle. Such an expression could be developed so that no scalar factor is applied while the vehicle is above a certain

speed, and the scalar factor is only applied when the vehicle's speed falls below a certain threshold which would cause the control solutions to become a value such that the VAC could no longer achieve the requested deceleration rates. Figure 4.2 demonstrates the concept of utilizing a floor value, showing that setting u_{min} equal to 0.25 prevents a control solution that is less than 0.25m/s^2. In practice, the u_{min} must be less than zero, as the control solution itself is negative, however it makes more sense intuitively to use positive values for the u_{min}.

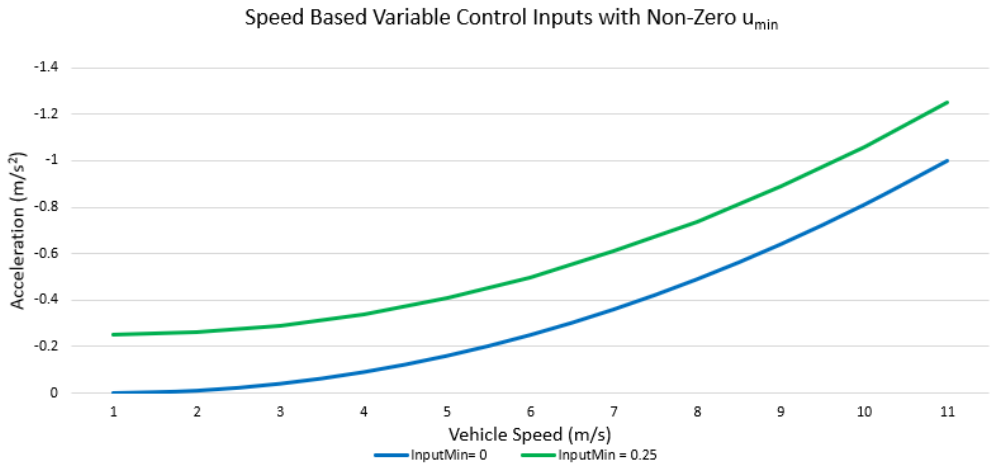

Figure 4.2 Comparison of speed based control inputs where there is a non-zero u_{min}

4.2.2 Gain Factors

There are many factors of the environment that are unmeasurable with our system but have a great effect on the overall response of the system such as the number of towed trailers, which significantly affects the stopping distance of the vehicle. As such, it quickly became apparent that control inputs that were optimal when the vehicle was towing no trailers proved to be not aggressive enough when towing multiple trailers. To

aid in the tuning and development of different control inputs for different environments, the concept of *gain factors* was developed. A gain factor was applied to the control inputs that the MPC algorithm used and a separate and independently-tuned gain factor was applied to the control inputs that were sent to the VAC. These two values allow greater control over tuning of the system and each affect the system in a different manner. In addition, and perhaps most importantly, it allowed us to tune the system to *feel* more natural to the driver and mitigate locking the brakes under heavy loads. Equation 4.3 and figure 4 demonstrate the effect that different gain factors have on the control inputs as well as how the control inputs are calculated.

$$u = \left(u_{min} * \left(\frac{v_0}{v_{max}}\right) * g\right) + u_{min} \tag{3.3}$$

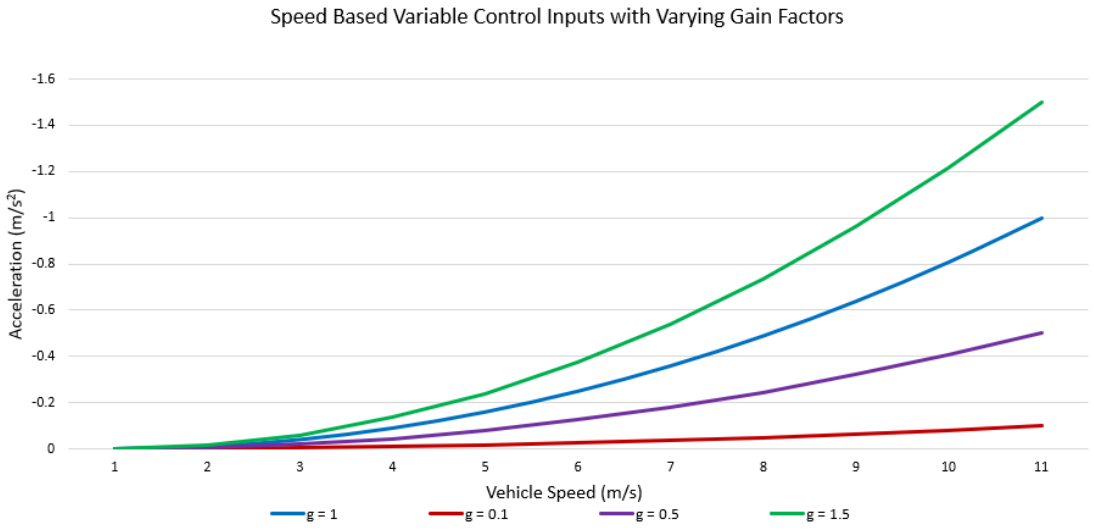

Figure 4.3 Comparison of speed based control inputs with varying values of g

4.2.2.1 MPC Gain Factor

As described in [16], suddenly removing control from the operator of a vehicle can be disturbing and may unnerve the driver. In our system, the revocation of control from the operator is done in the form of applying the brakes and ignoring the operators throttle input. Under certain conditions, the application of brakes may be rather aggressive. Given unknowns such as road conditions and towed mass, it may be dangerous to suddenly and forcefully apply the brakes. Therefore, it becomes imperative to blend driver control with control from the MPC algorithm.

The gain factor applied to the MPC algorithm is known as the MPC-GF. In essence, the MPC algorithm is "tricked" into believing that the vehicle is capable of greater deceleration rates, or, as the case may be, only capable of much smaller deceleration rates than it actually is. The MPC-GF effectively changes the *timing* of the application of the control solution, controlling whether the control solution should be applied earlier or later.

To fully understand the effect that the MPC-GF has on the overall system performance, the difference between the MPC-GF and the VAC-GF must be examined. For the purposes of this section, the VAC-GF will be assumed to be unity, for simplicity sake.

If a *very* high MPC-GF is used, the MPC algorithm will assume that the vehicle is capable of greater deceleration rates than are actually possible approaching instantaneous deceleration rates at the limit. As the MPC algorithm attempts to allow the driver to travel as fast as possible for as long as possible, the MPC algorithm will allow the vehicle to

travel at a greater velocity for longer and attempt to slow down the vehicle much later than if the MPC-GF was at unity or less. This may have detrimental consequences as the MPC-GF approaches increasingly large values, infinity at the extreme limit. In this extreme case, the MPC algorithm will believe instantaneous decelerations are possible, and so it is not necessary to request a deceleration before the vehicle has crossed the KOD. Under virtually no conditions in the real-world is instantaneous stopping possible, so it is nearly impossible to imagine a scenario where the MPC algorithm will successfully be able to bring the vehicle to a full stop safely before breaching the KOD.

The alternative to a very *high* value for the MPC-GF is a very *low* MPC-GF. Similar to the way that high values change the behavior of the MPC algorithm, sufficiently low values of the MPC-GF will cause the MPC algorithm to attempt to slow the vehicle down excessively early. This too can have detrimental consequences as the MPC-GF value approaches a lower limit, zero. The actual response of the vehicle with very low values of the MPC-GF is very dependent on how aggressively the VAC decelerates the vehicle, which is defined to some extent by the VAC-GF, which is assumed to be unity in this case. Assuming a relatively aggressive reaction from the VAC, the vehicle will slow down very quickly when the MPC algorithm requests a deceleration rate. Due to the large mismatch in requested speed and actual speed, the MPC algorithm may allow the vehicle to accelerate. A rapid acceleration of the vehicle may cause the MPC algorithm to again request a deceleration quickly after allowing the vehicle to accelerate. This cycle continues, causing a “bucking” sensation. In an extreme case, with large amounts of latency in the system, it is possible that the vehicle could breach KOD as it allows the vehicle to accelerate before it requests the vehicle

decelerate. If the KOD is not large enough, it is even possible that the vehicle could collide with the detected object.

4.2.2.2 VAC Gain Factor

While the MPC-GF directly affects the performance and response of the MPC algorithm and can be considered to affect the *timing* of the actuation of the control solution, the VAC-GF has a direct effect on the response and the *aggressiveness* of the VAC. For the purposes of explaining the behavior of various values of the VAC-GF, the MPC-GF is assumed to be unity.

The VAC-GF is applied to the input values in the exact same way the MPC-GF is, except the VAC-GF is applied to the input values that are sent to the VAC. As such, these gain factors have a direct impact on the behavior of the VAC and an indirect impact on the performance of the MPC algorithm.

Sufficiently high values of the VAC-GF have a fairly intuitive effect on the overall performance of the CA system—a high value will allow very aggressive decelerations. When the MPC algorithm calculates the optimal control solution given the set of environmental variables, the VAC-GF will effectively scale the control solution upwards, requesting a much more aggressive deceleration rate. In turn, the MPC algorithm may allow the vehicle to accelerate after slowing down quickly and abruptly, as was the case with a sufficiently small MPC-GF. Again the “bucking” sensation may occur and under certain conditions a collision may even occur.

Sufficiently low values of the VAC-GF will force the vehicle to decelerate slower than the MPC algorithm requests. Given this scenario, the MPC algorithm should continually increase the aggressiveness of its requests until the maximum deceleration rate is requested. However, given the limited number of control inputs, the most aggressive solution the MPC algorithm is able to request is limited. It is likely that the MPC algorithm will quickly *saturate* by requesting the maximum deceleration rate possible, given the current speed of the vehicle and assuming dynamic control inputs are being used. Like the case of a very high MPC-GF, it is possible that the vehicle will never slow down enough to stop before breaching the KOD.

It quickly becomes clear that gain factors that approach either upper or lower bounds exacerbate the negative effects of the *granularity problem*. Only when the relationship between the VAC-GF and MPC-GF is optimized do the gain factors successfully mitigate the effects of the *granularity problem*. It is important to realize that the specific values of the gain factors are less important than the discrepancy between the MPC-GF and the VAC-GF.

4.3 Extending Beyond Sensor Performance Limitations

The proper execution of the proposed control system is dependent on an effective sensor package to provide accurate and on-time measurements. Due to limitations based on the physical placement of the sensors, attaining optimal performance can be challenging. In the following section, we have introduced a few methods used in the MPC controller that can help alleviate these problems.

4.3.1 Beacon Persistence

Due to the position of the LiDAR sensor on the vehicle, the actual visible spectrum of the sensor was limited to approximately 270°. This became problematic when the vehicle was towing several trailers as the trailer would be able to collide with the beacon, but the MPC algorithm was unaware of such an event occurring as the sensor was no longer able to see the beacon and so the MPC algorithm did not know that the trailer was in danger of hitting a beacon. To solve this problem, detected beacons were persisted by the MPC algorithm. There were two methods of persistence that were applied, depending on where the beacon was detected in space. The first method is based on *time* and the second method is based on *area*.

4.3.1.1 Time-Based Persistence

Any detection of a beacon that was inside of a certain angular threshold would be persisted by time. The optimal angular threshold for this type of persistence was determined, empirically, to be +/-125°. If the angle of the detected beacon was within that range, it would be persisted by time. For 500 milliseconds, the position of the beacon would be approximated by utilizing the current heading of the vehicle along with the speed and acceleration. When the sensors returned a new set of detections, the new detections were compared to the persisted beacons. If a persisted beacon is within one meter of a newly detected beacon, it is assumed to be the same detection and the newly detected beacon's position is saved and the previously persisted beacon is forgotten. A margin of error of one meter was introduced to account for inaccuracies in predicted beacon location. If, after 500 milliseconds, no beacon has been detected in nearly the same position of a persisted beacon, it is assumed to have been a false-positive and the

beacon is no longer persisted. This method works well to filter false positives and prevent the list of detected objects from growing too large too quickly, however, it does not help with the problem of persisting beacons when they go beyond the range of sensors because they will not return to the vision of the sensors before the 500 millisecond life of the persisted beacon expires. To solve this, area-based persistence was used.

4.3.1.2 Area-Based Persistence

If a detected beacon is outside of the angular threshold of +/-125°, the type of persistence is switched to area-based. The beacon's new position is still predicted in the same way as time-based persistence, but the beacon is not "forgotten" when 500 milliseconds have elapsed and no beacon has been detected near the persisted beacon. Instead, once the beacon's distance to the rear axle of the vehicle is greater than twenty-five meters, the beacon is no longer persisted. Twenty-five meters was chosen as this is the length of the vehicle plus four trailers. Using this distance ensures that all trailers must be completely clear of the beacon before it is no longer persisted. This allows the MPC algorithm to prevent collisions with the dollies and beacons.

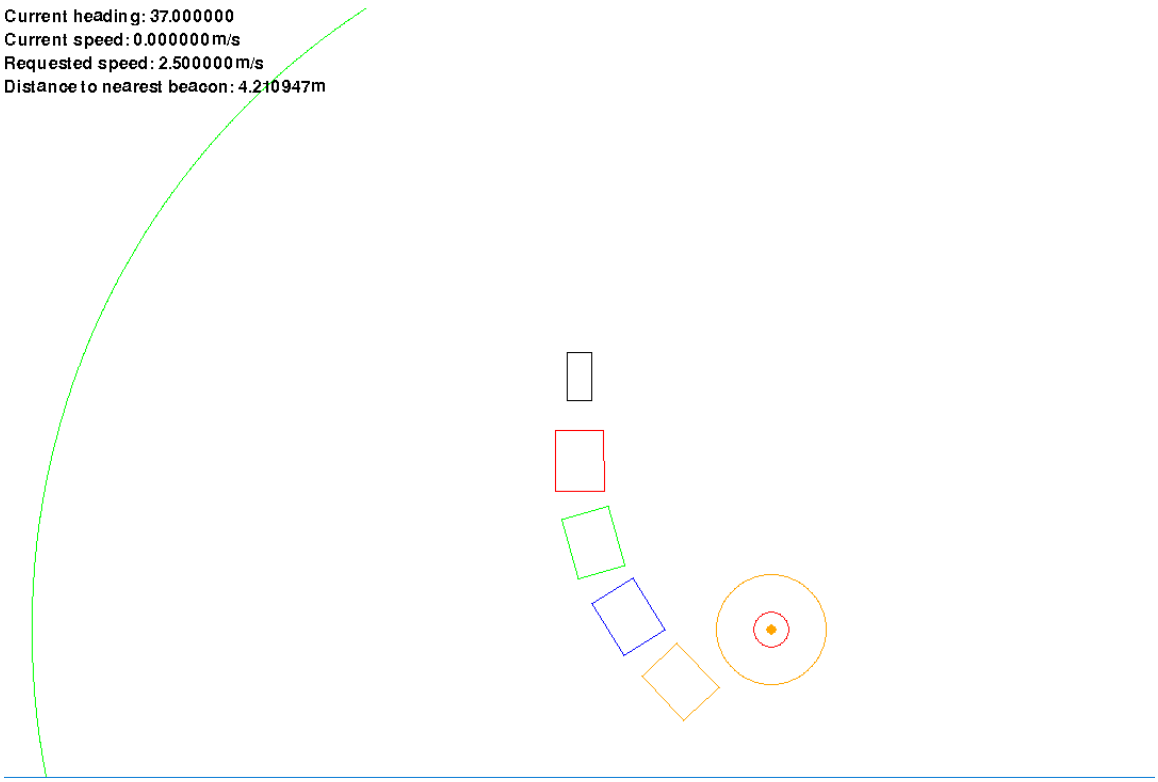

Figure 4.4 Area-based beacon persistence

Figure 4.4 demonstrates the concept of area-based persistence, where the detected beacon, signified by the orange dot, with 3 concentric circles around it, one red, one orange, and one green, is beyond the FoV of the LiDAR but the MPC algorithm has persisted it. While the current speed of 0m/s, as shown in the top left corner of the figure, the requested speed of 2.5m/s signifies that the operator is allowed to accelerate up to a maximum of 2.5m/s. Given the current heading of the vehicle and the current orientation and position of the vehicle and towed trailers, it is very unlikely that the vehicle or any of the trailers will collide with the beacon, so the MPC algorithm does not need to limit the speed of the driver beyond the current maximum allowable speed of 2.5m/s as the algorithm is currently operating in BCA mode.

4.3.2 Dynamic Field of View for Front Guard Collision Avoidance Mode

When the CAS is operating in FGCA mode, the sensors are designed to detect objects only directly in front of the vehicle. The effective FoV is demonstrated in figure 4.5.

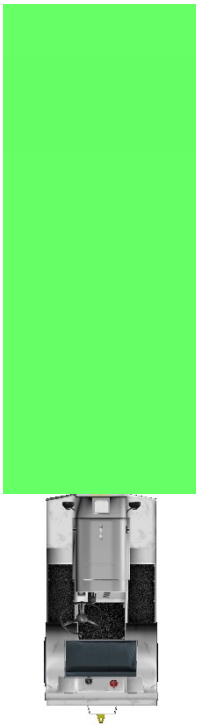

Figure 4.5 Static FoV for FGCA mode

The shape of the FoV is essential to the correct operation of the CA system as this specific shape will allow the vehicle to travel without being limited unless an object is directly in front of the vehicle. It is undesirable to limit the vehicle's speed if there are objects around the vehicle, but not directly in front. An example of such a scenario is demonstrated in figure 4.6.

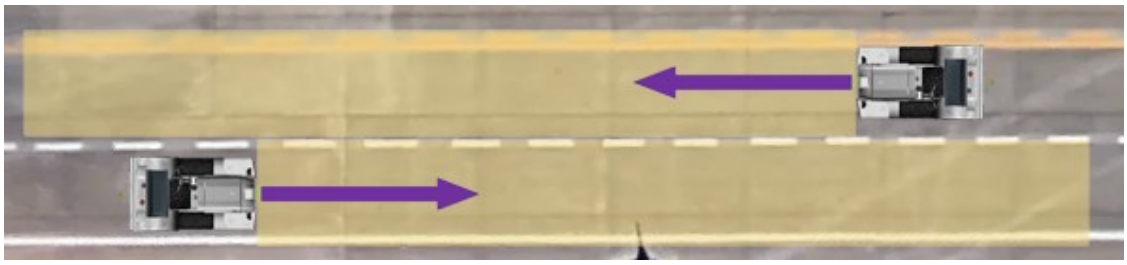

Figure 4.6 A realistic example demonstrating the necessity of the shape of the FoV for FGCA

As demonstrated in figure 4.6, if the FoV of the CA system while in FGCA mode was any wider or was not rectangular, the vehicles would be limited and potentially even prevented from passing each other, depending on how close the two vehicles were to each other.

Given the rectangular FoV of the CA system in FGCA mode, if the operator turned sharply and an object was then directly in the path of the vehicle, the MPC algorithm would react very abruptly, which could create a dangerous situation. Given poor road conditions, tires with less-than-optimal amounts of lateral grip due to tire wear over time, large amounts of towed mass, or a number of any other variables, the vehicle could quickly lose control which could make the situation worse. To help prevent such a situation, FoV of the sensors changes dynamically with the current angular velocity of the vehicle. A rectangular FoV that is directly in front of the vehicle is always present, but it may be expanded to the left or the right if the vehicle is turning. As the vehicle turns to the right, the FoV is expanded to look further to the right, and the FoV expands in a similar manner when the vehicle turns to the left. This effect is demonstrated in figures 4.8-4.10. This gives the MPC more time to process and calculate a control solution and

bring the vehicle to a safe and controlled stop in time. Figure 4.7 shows the algorithm used for the dynamic FoV.

Algorithm Dynamic Field of View

```
if detection is a front guard detection then
    if ω = 0 then
        if detection in FoV then
            save detection
    else if ω > 0 then
        Expand FoV in the positive direction
        if detection in FoV then
            save detection
    else if ω < 0 then
        Expand FoV in the negative direction
        if detection in FoV then
            save detection
else
    save detection
```

Figure 4.7 Algorithm to expand the FoV based on angular velocity

In practice, this proved to greatly improve the performance of front guard in the described situations. The operator was still able to travel next to the object without being impeded by the MPC algorithm. As soon as the operator changed the trajectory of the vehicle towards the object, the MPC algorithm brought the vehicle to a stop. A more realistic image demonstrating the effect of the dynamic FoV is demonstrated in 4.10.

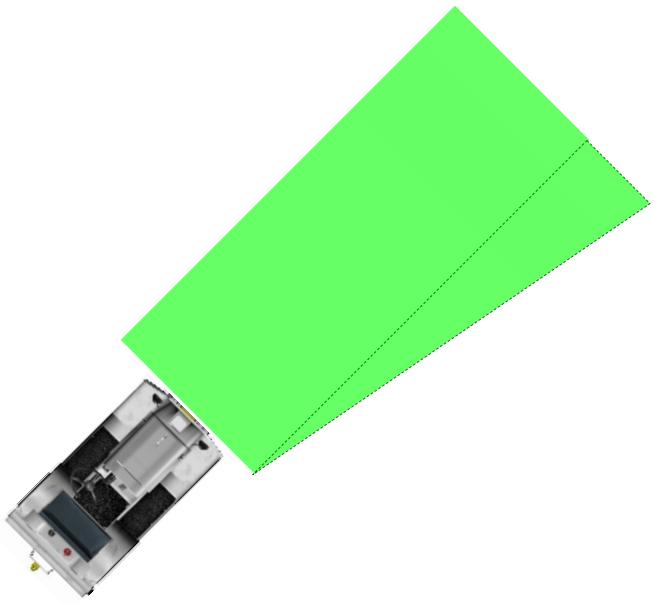

Figure 4.8 Dynamic FoV when the vehicle is turning to the right

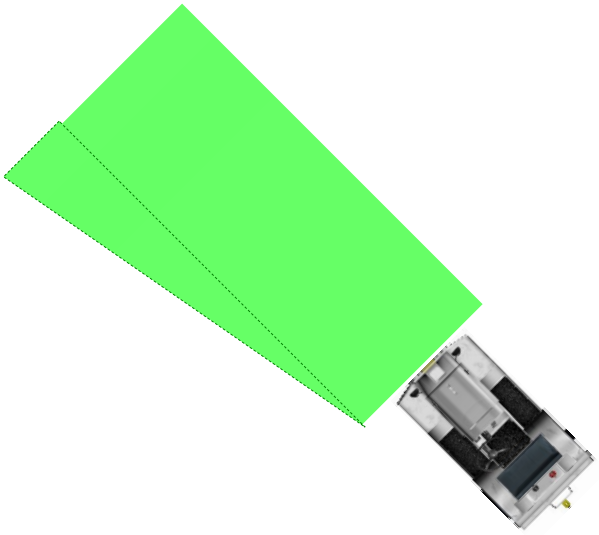

Figure 4.9 Dynamic FoV when the vehicle is turning to the left

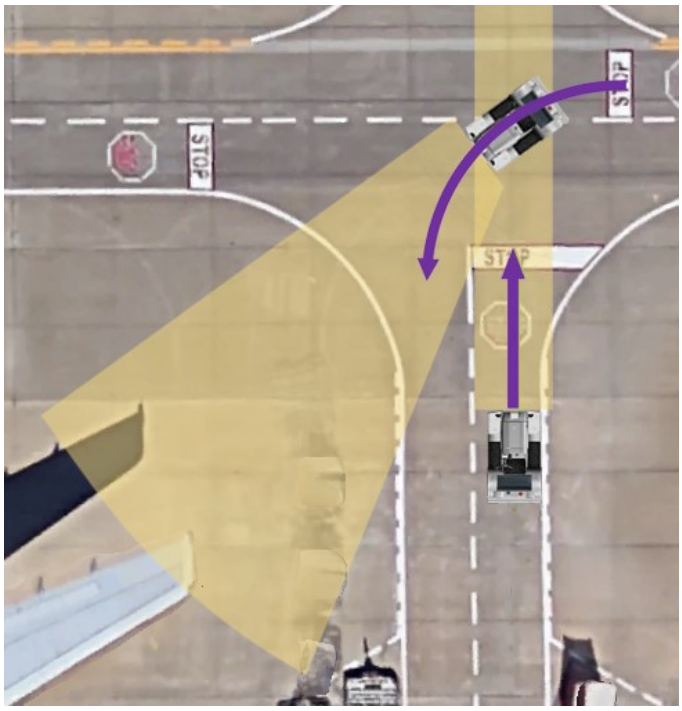

Figure 4.10 Realistic example of dynamic FoV when the vehicle is operating in FGCA mode

4.4 Mitigating Latency

As we observed, and as is pointed out in [15], real-world applications usher in several places for latency to be injected into the system. In our system, we experienced significant latency in the processing time of the sensors, the computational time of the MPC algorithm, as well as the vehicle actuators, not to be confused with the vehicle actuation controller. Under the worst conditions, the total latency could be as much as several hundred milliseconds. Latency can have extremely detrimental consequences. The algorithm that Liu, et al developed fails when the control latency, which is the summation of the MPC computational time and the vehicle actuator latency, exceeds 200 milliseconds and when the sensor latency, which is the processing time of the sensor

platform, exceeds 400 milliseconds [15]. Unlike the authors of [15] the latency of our system is not known and is prone to varying due to many different factors.

Sensor latency can be very detrimental to the overall system, as such, we developed a system to mitigate the effects. When the sensor package reports a beacon to the CA system, using the current speed and heading and an empirically tuned time-based parameter, the beacon was moved a certain distance closer to the vehicle. This simulated the object being detected later and attempted to approximate the actual location of the object when the VAC would be applying the commanded control solution. This constant was known as the *Latency Correction Factor* (LCF) and was tuned empirically due to the fact that the latency varied so much and there were so many different places where latency could present itself. The most optimal value of LCF that we found was 400 milliseconds. At 400 milliseconds, the MPC algorithm appeared to perform the most accurately and successfully.

CHAPTER V

RESULTS AND PERFORMANCE EVALUATION

5.1 Hardware Implementation

To test the viability of the algorithm, utilizing an embedded GPU, the algorithm was implemented and ran on an NVIDIA Jetson TX2 development platform, shown in figure 5.1.

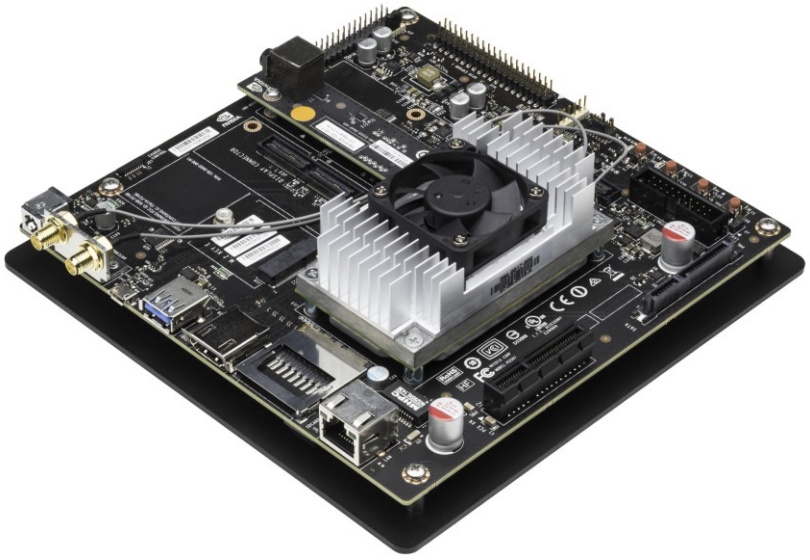

Figure 5.1 NVIDIA Jetson TX2 Embedded GPU Development Platform [31]

The Jetson TX2 development board features an NVIDIA GPU based on NVIDIA's Pascal architecture utilizing 256 CUDA cores. In the realm of processors with fewer threads, the development board has two CPUs, a Denver 2 and a Quad ARM A57.

The Denver 2 is a new addition to the Jetson family as the Jetson TX1 made do with only a Quad ARM A57. The Jetson TX2 doubles the amount of RAM from its predecessor, raising from 4 gigabytes up to 8 gigabytes of LPDDR4. In addition to the additional RAM, the total on board data storage has been increased from 16 gigabytes to 32 gigabytes [32].

5.2 The Vehicle

The vehicle that the CA system was implemented on was a Tug Inc cargo tractor, pictured in figure 5.2.

Figure 5.2 Tug Inc cargo tractor, model Tug MA

At the front of the vehicle, the black cylindrical object is the LiDAR sensor and the two cameras were installed behind the front grill of the vehicle, with the lenses

peering through the holes in the grill. By installing the cameras in this fashion, they are protected from people or objects accidentally bumping into them and affecting the angle at which they are installed, which would render the computer vision methods that are employed ineffective.

Figure 5.3 The weather-proof housing of the Jetson TX2 development boards and the custom designed power-conditioning board installed on the vehicle

The white box shown in figure 5.3 is the weather-proof housing which contains three separate Jetson TX2 platforms, two of which are used for sensing processing and the third which is used for the MPC computation. The VAC is housed under the floor board on the passenger side.

5.3 Optimizing the Control Solution Search

As was described in chapter 3, two methods of determining the optimal control solution were tested in order to compare performance improvements when NVIDIA's Thrust library was used. In figure 5.4 and 5.5, the relative performance difference is demonstrated.

Figure 5.4 Execution time using the naïve implementation of the MPC algorithm

Figure 5.5 Execution time using the Thrust library

Figure 5.3 shows that the majority of the execution time is spent during memory transfers between the device and host and searching for the optimal control solution. In fact, up to 80% of the execution time may be spent during this part of the execution of the algorithm. In contrast, using NVIDIA's Thrust library, the percentage of execution time that finding the optimal control solution occupies is as low as just under 3%. Further, it is important to note that the relative difference between the execution time of only the MPC algorithm is statistically irrelevant.

5.4 Comparison with a serial implementation

The algorithm was re-written in a serial manner to compare the performance advantage the GPU has over the few-threaded CPU. From figure 5.5, it is very evident of the performance advantage of the GPU over the CPU. For fifteen beacons, the average execution time averaged over twenty iterations for the PUG was approximately 8.3ms

where as the execution time of the CPU under the same test conditions was a stagger 1,022.86ms.

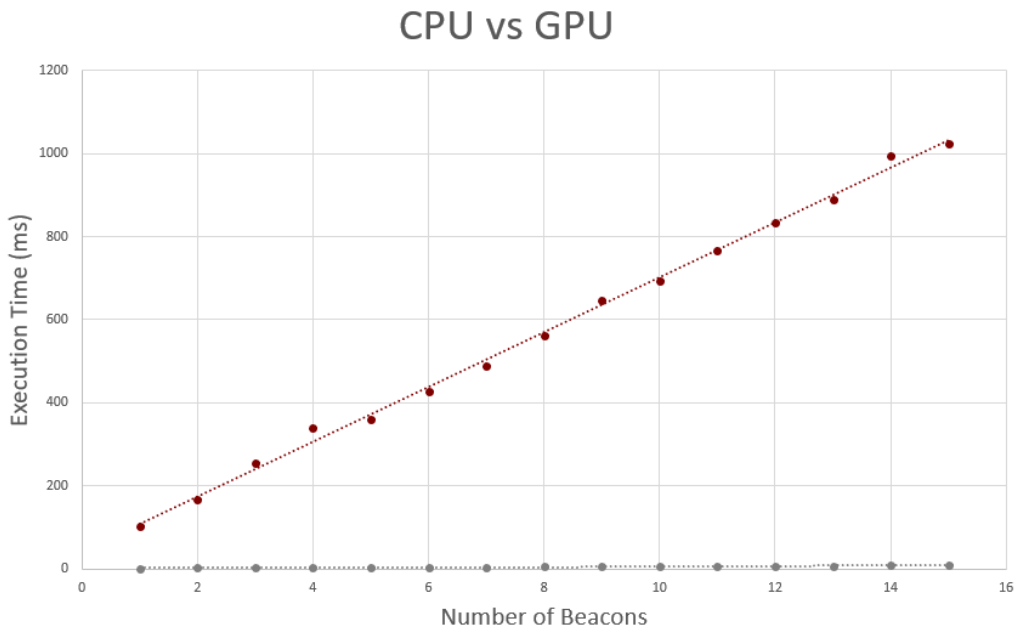

Figure 5.6 Time comparison between CPU and GPU

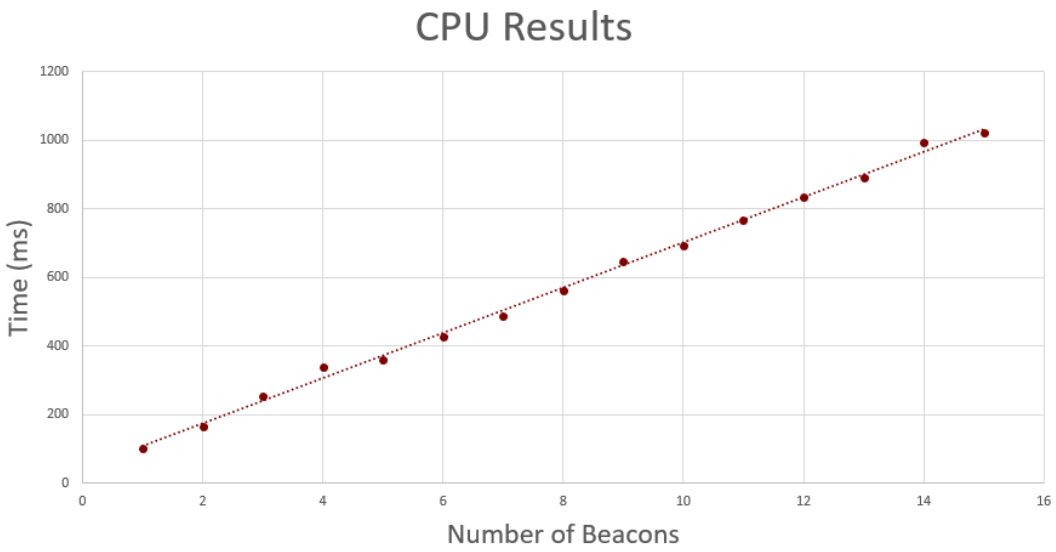

Figure 5.7 CPU execution time given a varying number of beacons

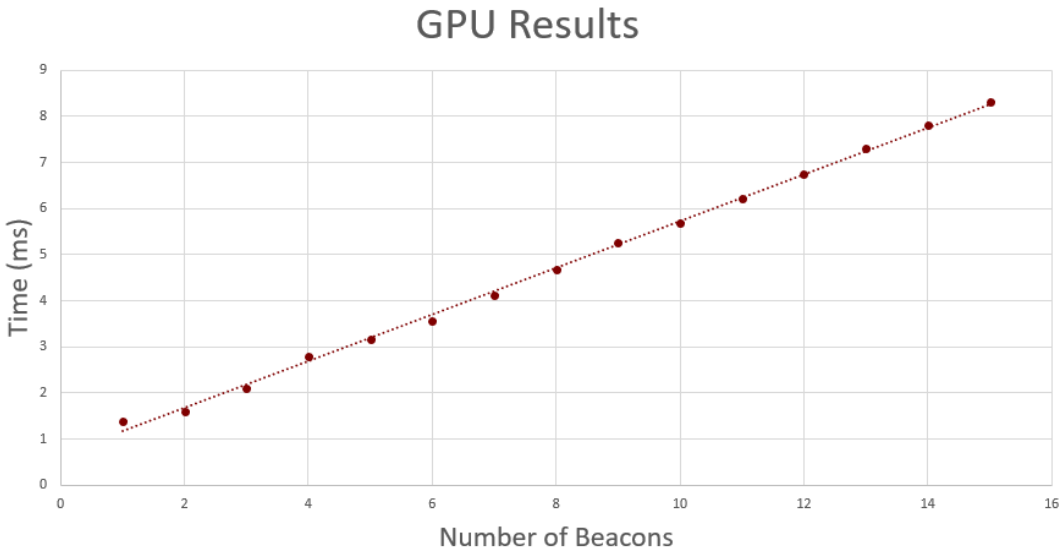

Figure 5.8 GPU execution time given a varying number of beacons

In addition to comparing the execution time of the CPU and GPU with varying numbers of beacons, a comparison of GPU and CPU execution time with varying

numbers of control inputs is illustrated in figures 5.9 and 5.10. In this figure, the execution time is in fact exponential which is explained by equation 3.1. Further, this test was run with five lookahead steps, which means that the GPU is only capable of executing the MPC with up to four control inputs, as was discussed in chapter 3.

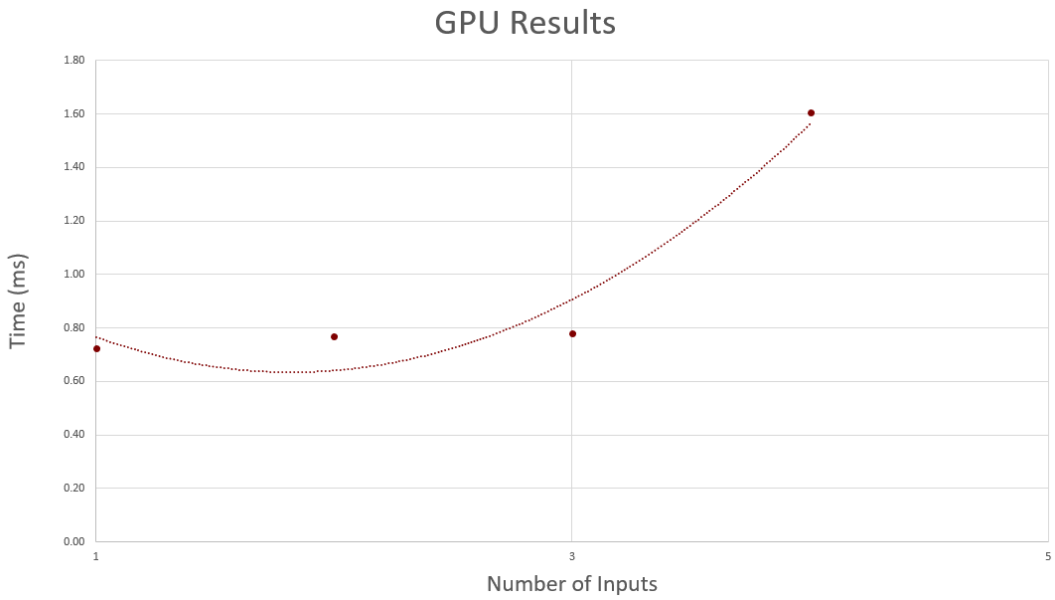

Figure 5.9 GPU execution time with a varying number of control inputs

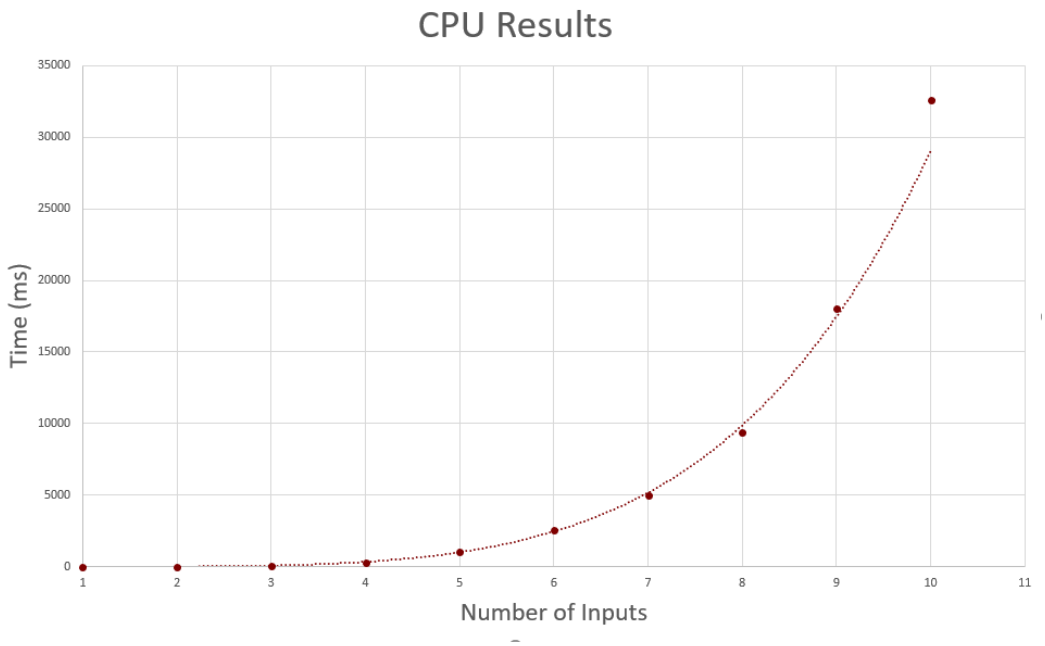

Figure 5.10 CPU execution time with a varying number of control inputs

Figures 5.9 and 5.10 demonstrate the effect that an increasing number of control inputs has on the execution time of the algorithm. While both methods produce exponential results, it is clear that the CPU implementation produces results that increase at a much faster rate than does the GPU.

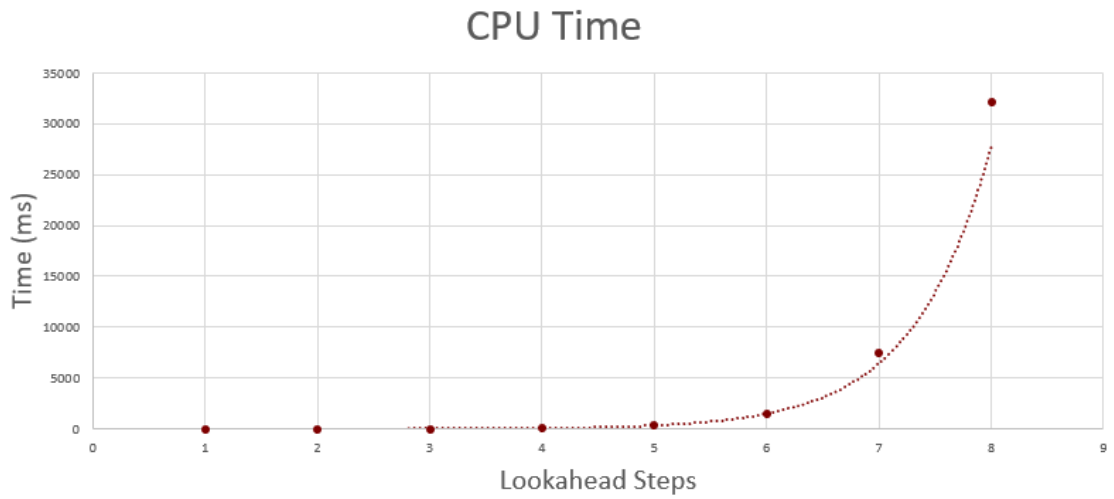

Figure 5.11 CPU execution time with varying number of lookahead steps

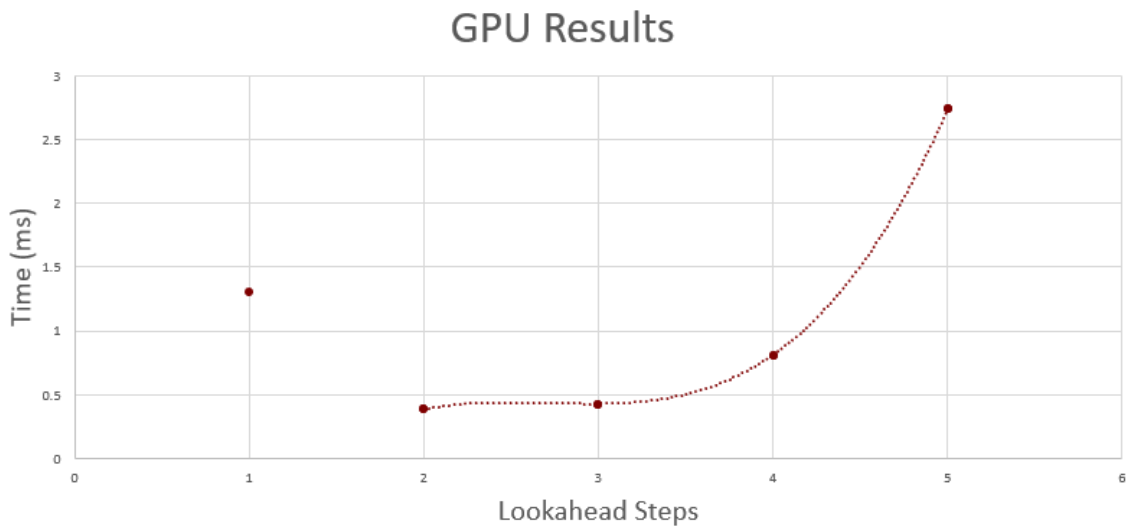

Figure 5.12 GPU execution time with varying number of lookahead steps

Again, the rapid exponential increase in execution time of the CPU is demonstrated in figures 5.11 and 5.12 which show how increasing the prediction horizon affects the computation time. The GPU takes significantly longer to execute given one

lookahead step than it does for two, three, or even four lookahead steps because the number of threads being executed is very sparse and therefore the total occupancy is low.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

6.1 Conclusion

Considering the inherent danger of vehicles given the statistics on vehicle-related casualties in recent years [1], the need for some type of augmented safety device in vehicles is apparent. The efficacy of a predictive controller and the advantages over a more simplistic reactive controller are clear with the major downside of a predictive controller being the computational power required to process such a control problem in real-time, which is necessitated by the vehicular application. Given the imminent demise of Moore's Law, it quickly becomes apparent that an alternative to traditional computing platforms must be explored to achieve the computational power necessary. Heterogeneous computing is one such alternative [17, 18, 19].

We have developed and implemented such a system that uses a predictive control algorithm that is computed in real-time using a GPU in tandem with a CPU on an embedded platform for a collision avoidance system on an industrial vehicle. In addition, it has been shown that the computational time of the CPU is exponentially greater than the computational time of the GPU and increases at a faster rate as the problem space becomes larger.

6.2 Future Work

Future work should include further development and refinement of the system as a whole. While further refinements can be made from a variety of different perspectives, the focus of this chapter will be on improvements to the controls of the system. More specifically, no problems that are directly related to the successful operation of the system for the proposed task have been left unsolved.

Through various tests that have been performed on the system, we have noticed that sufficiently small KoDs can sometimes result in undetected collisions due to the sparse number of points in which a collision is detected. Increasing this could improve the system's performance given a different vehicular system.

Improvements to the system include methods of improving the prediction horizon while not sacrificing computational simplicity by utilizing a Monte Carlo Tree Search approach and improving the control response by use of a dynamic timestep length. The control response could further be improved by perhaps investigating more complex methods of calculating dynamic control inputs.

6.2.1 Finer Grained Collision Detection

The CA system that we developed only considers collisions between the corners of each body and the detected object. If this system were to be used with a vehicle that was significantly longer, or trailers that were significantly longer, it is possible that collisions could occur without the corners entering a certain radius around the beacon as demonstrated in the following figure.

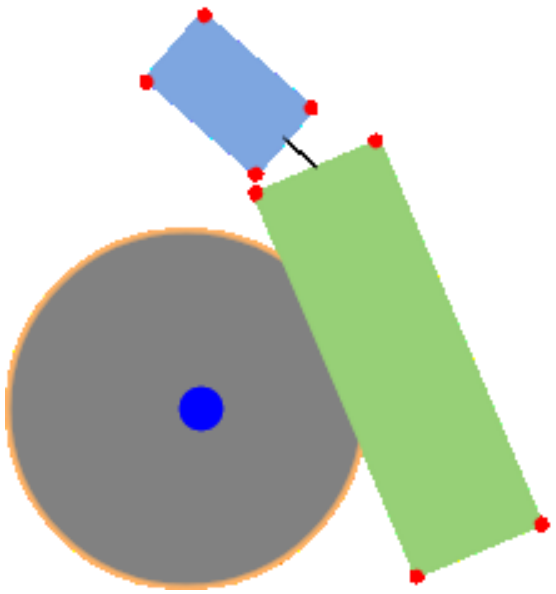

Figure 6.1 Collision with only corner detection

As shown in figure 6.1, using the current CA system, it is possible to the towed object to enter the indicated radius around the detected object without triggering an emergency stop from the CA system because no corner is close enough to the detected object. This could be mitigated by not only monitoring the corners of the object, but also monitoring intermediate points along each edge of the object as shown in the following figure.

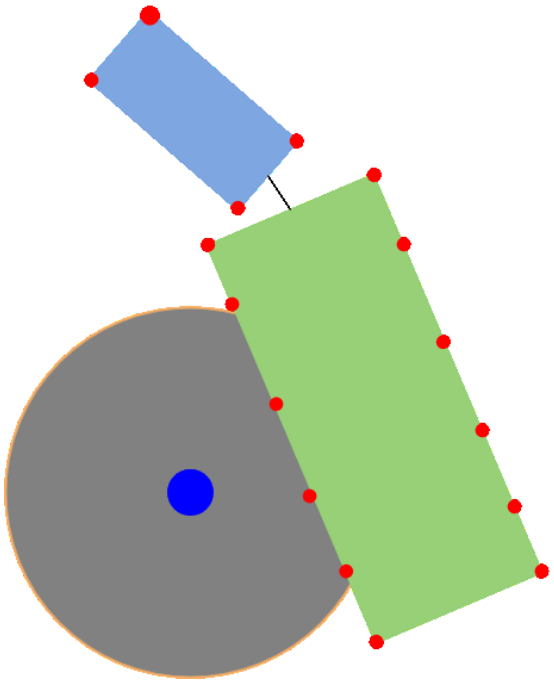

Figure 6.2 Collision detection with more points of detection

6.2.2 Finer Grained Control Input Creation

We showed that the use of control inputs that dynamically changed with the velocity of the vehicle significantly improved overall performance of the system. The actual formula that was used could be considered, to some extent, naïve and could potentially benefit from a more complex expression. Further, the use of a minimum input value to create a *floor* could be further explored. Using a non-constant value for the floor could create even finer grained control and response from the overall system.

6.2.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a modification of the traditional exhaustive tree search where every branch of the tree must be evaluated before a decision is made. In machine learning and other fields of artificial intelligence (AI), this method is also known

as *minimax*. MCTS allows only part of the tree to be evaluated, using previous experiences to predict the most optimal branches to continue evaluating. Using this method, it may be possible to evaluate either more lookahead steps or more control inputs, though it would likely be more beneficial to evaluate more lookahead steps due to the improvements that dynamic control solutions made to the system.

6.2.4 Dynamic Timestep Length

As the speed of the vehicle increases, the distance the vehicle travels between calculations of the MPC algorithm increases. As such, it becomes more critical to look further ahead than have fine-grained timesteps. At higher speeds, it is desirable to have timesteps that are longer, sacrificing granularity for a further prediction horizon and at lower speeds, such granularity is more important whereas a distant prediction horizon is less important as the distance travelled over the course of n timesteps will decrease. Developing an expression to relate the current velocity of the vehicle to the length of the timestep could possibly help improve the performance of the system.

REFERENCES

[1] NHTSA, "National Center for Statistics and Analysis (NCSA) Motor Vehicle Traffic Crash Data Resource Page," 2017. [Online]. Available: https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812451. [Accessed 2018].

[2] B. Reimer, "Driver Assistance Systems and the Transition to Automated Vehicles: A Path to Increase Older Adult Safety and Mobility?," Oxford University Press, 2013.

[3] D. Geronimo, A. Lopez, A. Sappa and T. Graf, "Survey of Pedestrian Detection for Advanced Driver Assistance Systems," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 32, no. 7, pp. 1239-1258, July 2010.

[4] J. Shi, "Development of a Dynamic Performance Management Framework for Naval Ship Power System Using Model-Based Predictive Control," 2014.

[5] J. Bai and S. Abdelwahed, "Efficient Algorithms for Performance Mangement of Computing Systems," in *Fourth International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks FeBID*, 2009.

[6] J. L. Jerez, P. J. Goulart, S. Richter, G. A. Constantinides, E. C. Kerrigan and M. Morari, "Embedded Online Optimization for Model Predictive Control at Megahertz Rates," vol. 59, no. 12, pp. 3238-3251, December 2014.

[7] N. F. Gade-Nielsen, J. B. Jorgensen and B. Dammann, "MPC Toolbox with GPU Accelerated Optimization Algorithms," in *The 10th European Workshop on Advanced Control and Diagnosis (ACD 2012)*, 2012.

[8] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum,* vol. 26, no. 1, pp. 80-113, 2007.

[9] E. Wu and Y. Liu, "Emerging Technology About GPGPU," in *IEEE Asia Pacific Conference on Circuits and Systems*, Macao, 2008.

[10] S. Singh, S. Singh, V. K. Banga and D. Chauhan, "National Conference on Contemporary Techniques & Technologies in Electronics Engineering," Murthal, 2013.

[11] H. J. Ferreau, P. Ortner, P. Langthaler, L. d. Re and M. Diehl, "Predictive Control of a Real-world Diesel Engine Using an Extended Online Active Set Strategy," *Annual Reviews in Control,* no. 31, pp. 293-301, 2007.

[12] A. S. Badwe, R. S. Patwardhan, S. L. Shah, S. C. Patwardhan and R. D. Gudi, "Quantifying the Impact of Model-Plant Mismatch on Controller Performance," *Journal of Process Control,* no. 20, pp. 408-425, 2010.

[13] B. Alrifaee, J. Maczijewski and D. Abel, "Sequential Convex Programming MPC for Dynamic Vehicle Collision Avoidance," in *IEEE Conference on Control TEchnology and Applications (CCTA)*, Kohala Coast, 2017.

[14] D. Zhou, Z. Wang, S. Bandyopadhyay and M. Schwager, "Fast, On-line Collision Avoidance for Dynamic Vehicles Using Buffered Voronoi Cells," *IEEE Robotics and Automation Letters,* vol. 2, no. 2, pp. 1047-1054, 2017.

[15] J. Liu, P. Jayakumar, J. L. Stein and T. Ersal, "A Multi-stage Optimization Formulation for MPC-based Obstacle Avoidance in Autonomous Vehicles Using a LiDAR Sensor," in *Dynamic Systems and Control Conference*, San Antonio, 2014.

[16] S. J. Anderson, S. C. Peters, T. E. Pilutti and K. Iagnemma, "An Optimal-control-based Framework for Trajectory Planning, Thread Assessment, and Semi-Autonomous Control of Passenger Vehicles in Hazard Avoidance Scenarios," *Int. J. Vehicle Autonomous Systems,* vol. 8, no. 2,3,4, pp. 190-216, 2010.

[17] R. R. Schaller, "Moore's Law: Past, Present, and Future," *IEEE Spectrum,* pp. 53-59, June 1997.

[18] J. M. Shalf and R. Leland, "Computing Beyond Moore's Law," *Computer,* pp. 14-23, December 2015.

[19] A. Shan, "Heterogeneous Processing: A Strategy for Augmenting Moore's Law," *Linux Journal,* vol. 2006, no. 142, p. 7, February 2006.

[20] M. Mukerjee, D. Naylor and B. Vavala, "Packet Processing on the GPU".

[21] A. Maghazeh, U. D. Bordoloi, P. Eles and Z. Peng, "General Purpose Computing on Low-Power Embedded GPUs: Has It Come of Age?," in *International Conference on Embedded Computer Systems: Architecutres, Modeling, and Simulation (SAMOS XIII)*, Agios Konstantinos, 2013.

[22] L. Yu, A. Goldsmith and S. D. Cairano, "Efficient Convex Optimization on GPUs for Embedded Model Predictive Control," in *Proceedings of the General Purpose GPUs*, Austin, 2017.

[23] J. Liu, P. Jayakumar, J. L. Overholt, J. L. Stein and T. Ersal, "The Role of Model Fidelity in Model Predictive Control Based Hazard Avoidance in Unmanned Ground Vehicles Using LiDAR Sensors," in *Dynamic Systems and Control Conference*, Palo Alto, 2013.

[24] Y. Liu, C. Davenport, D. Iacomini, J. Gafford and M. Mazzola, "Development, Testing, and Assessment of A Dynamic Path FOllowing Model for Twoing Vehicle Systems".

[25] "CppReference.com," 8 February 2018. [Online]. Available: http://en.cppreference.com/w/cpp/numeric/math/atan2. [Accessed 2018].

[26] T. S. Crow, "Evolution of the Graphical Processing Unit," 2004.

[27] J. D. Owens, M. Houston and D. Luebke, "GPU Computing," *Proceedings of the IEEE,* vol. 96, no. 5, pp. 879-899, May 2008.

[28] J. Nickolls and W. J. Dally, "The GPU Computing Era," *IEEE Micro,* vol. 30, no. 2, pp. 56-69, 12 April 2010.

[29] NVIDIA, "CUDA Toolkit Documentation," NVIDIA, 5 March 2018. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#page-locked-host-memory. [Accessed 2018].

[30] NVIDIA, "CUDA Toolkit Documentation," NVIDIA, 5 March 2018. [Online]. Available: http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1gab84100ae1fa1b12eaca660207ef585b. [Accessed 2018].

[31] [Online]. Available: https://www.cnx-software.com/wp-content/uploads/2017/08/Nvidia-Jeston-TX1-Developer-Kit-Large.jpg.

[32] NVIDIA, "NVIDIA Jetson," NVIDIA, 2018. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/. [Accessed 2018].

[33] D. D. Donno, A. Esposito, L. Tarricone and L. Catarinucci, "Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD [EM Programmer's Notebook]," *IEEE Antennas and Propagation Magazine,* vol. 52, no. 3, pp. 116-122, June 2010.