Hardware and Software Acceleration for Hamilton-Jacobi Reachability Analysis

by

Minh Nhat Bui

B.Ap.Sc., University of British Columbia, 2019

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Minh Nhat Bui 2021
SIMON FRASER UNIVERSITY
Summer 2021

Declaration of Committee

Name: **Minh Nhat Bui**

Degree: **Master of Science**

Thesis title: **Hardware and Software Acceleration for Hamilton-Jacobi Reachability Analysis**

Committee: **Chair:** Tianzheng Wang
Assistant Professor, Computing Science

Arrvindh Shriraman
Co-Supervisor
Associate Professor, Computing Science

Mo Chen
Co-Supervisor
Assistant Professor, Computing Science

Alaa Alameldeen
Committee Member
Associate Professor, Computing Science

Hang Ma
Examiner
Assistant Professor, Computing Science

Abstract

Hamilton-Jacobi (HJ) reachability analysis is a powerful technique with applications in robotic safety, game theory. etc. HJ reachability analysis is advantageous in analyzing nonlinear systems with disturbances and flexible set representations. A drawback to this approach is that the associated Partial Differential Equation (PDE) is solved numerically on a multidimensional grid, hence scales poorly as the number of dimensions increases.

There has been an extensive body of work that addresses the computational complexity reduction of the problem with or without introducing overapproximation. In this thesis, without changing the numerical solution approach, we address the speedup of solving HJ PDE using software and hardware acceleration.

Our first contribution is OptimizedDP, a python-based software toolbox optimized for dynamic programming algorithms arising in optimal control and reinforcement learning. The software toolbox reduces computational time ranging from 7x-75x compared to common toolboxes written in MATLAB and implementation in Python.

Our second contribution is a customized hardware design that accelerates HJ PDE solving procedure on a Field Programmable Gate Array (FPGA). The design can accelerate 4D grid-based HJ reachability analysis up to 14 times compared to OptimizedDP and 103 times compared to the existing MATLAB toolbox on a 16-thread machine. The methodology presented here is without loss of generality: it can potentially be applied to different systems dynamics, and moreover, leveraged for higher dimensional systems. In addition, we experiment online HJ PDE solving algorithm, using on-cloud FPGA, on a robot car that can safely avoid obstacles.

Keywords: FPGA; Optimal Control; Software toolbox; Hardware Acceleration; Reachability Analysis

Dedication

To my parents, my brother, and my grandmother.

Acknowledgements

I'm grateful for the constant guidance, support, and encouragement from my advisors Mo Chen and Arrvindh Shriraman during the past 2 years. I have learned from you both the technical knowledge and wisdom. Thanks to that I have improved and became better at conducting research as well as writing papers and giving presentations. Thanks Mo for introducing me opportunities to give talks about my research whenever possbile. Those talks and presentations have helped me become more confident in myself and my research. I also would like to say thanks to Reza Hojabr, Amirali Sharifian, Michael Lu, Georgios Giovanis for their assistance in my work presented here, without you guys this thesis would not be possible.

I would like to also say thanks to friends I have met in the past 2 years including Ali Sedaghati, Tien Vu, Parmida Vahdatniya, Anjian Li, and everyone in MARS lab who made my graduate school journey less lonely and more enjoyable.

Last but not least, I would like to express my gratitude to my parents and my brother, who've always supported me unconditionally with everything throughout this endeavor.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As autonomous systems, such as self-driving cars, unmanned aerial vehicles, and rescue robots, become more prevalent in our daily lives, one key factor that will allow wider adoption of these systems is their guaranteed safety. Despite tremendous progress in autonomous systems research in areas such as motion planning, perception, and machine learning, deployment of these systems in environments that involve interactions with humans and other robots remains limited due to the potential danger these robotic systems can cause. Formal safety verification methods can help autonomous robots reach their untapped potential.

Formal verification via reachability analysis can provide guaranteed safety and goal satisfactions of autonomous systems under adversarial disturbances. Reachability analysis can characterize a set of states called Backward Reachable Tube (BRT) that a system must stay out of in order to avoid obstacles. There are different methods to do reachability analysis for dynamical systems. For example, the authors in [3] propose an approach that would scale up to one billion dimensions for affine systems, and the authors in [4] provide reachable sets using specified shapes. Most of these methods trade off generalization for scalability by making specific assumptions about linear dynamic systems, systems with no input and disturbances, and set representations with polytypes. The HJ formulation, on the other hand, is very powerful in handling control and disturbances, nonlinear system dynamics, and flexible set representations as well as synthesizing controllers to get the system out of the unsafe states. HJ reachability analysis has been successfully applied in practical applications such as aircraft safe landing [5], multi-vehicle path planning, multiplayer reach avoid games [6]. The main downside to HJ reachability is that the associated HJ partial differential equation (PDE) is solved numerically on a multi-dimensional grid with the same number of dimensions as the number of state variables and thus computational complexity scales exponentially with the number of dimensions.

In this thesis, we make the following two contributions. Our first contribution is a computationally efficient toolbox for solving HJ reachability and other related dynamic programming algorithms that emphasizes efficiency and user-friendliness. The second contribution is a computing architecture on FPGA that further accelerates HJ PDE computation to perform real-time reachability for 4D systems. This computing architecture can used as a computing service hosted by AWS cloud that can potentially be useful for roboticists trying to apply reachability analysis for real-time collision avoidance of robots.

Chapter 2

Optimized Dynamic Programming Toolbox

Dynamic programming algorithms are crucial to many optimization problems. Despite its poor scalability and exponential complexity, global optimal solutions to many control and optimization problems are only feasible via dynamic programming. Dynamic programming also provides a framework that is important for reasoning about solutions to many complex problems.

Applications of dynamic programming we would like to address in this work is continuous Markov Decision Process (MDP) value iteration, and reachability analysis via solving the HJ PDE, whose solution is crucial for guaranteeing the safety of autonomous systems and the surrounding environment, especially in robotics where there are increasing presences of autonomous agents in services. While there exist many packages for solving value iteration in MDP, none of them have support for value iteration with continuous state space and action.

In addition, as the numerical algorithm for solving the HJ PDE to obtain the Backward Reachable Tube (BRT) and Backward Reachable Set (BRS) defined in [7] is quite complex and involves many floating-point operations on a large dimensional grid, it takes a lot of effort and time to write the algorithm, prototype the system dynamics, waiting for output results (which can be hours/days), and validating the results. Scalability is probably the biggest downside of the framework but these aforementioned factors shy roboticists away more from applying reachability analysis to their research. To address some of these problems, there have been some toolboxes that were implemented: HelperOC as a wrapper of the level set toolbox ToolboxLS [2], and the BEACLS library written in C++ and CUDA [8]. HelperOC and ToolboxLS are both written in MATLAB, which contains a rich set of visualizing plots and contours functions, and is user-friendly and quite powerful in prototyping mathematical models. However, this toolbox suffers from slow runtime with the MATLAB software package being proprietary. BEACLS, on the other hand, executes the level-set based numerical algorithm much faster than the MATLAB counterpart but has a very difficult interface to specify a problem setting and hence is not user-friendly.

In this section, we introduce our new toolbox that not only obtains the BRS and BRT more efficiently but also includes other optimal control algorithms based on dynamic programming that assist researchers better in prototyping and applying optimal control algorithms to their system model. The advantages of our toolbox compared to the existing ones are the significant improvement of the execution runtime and the user-friendly interface for problem specifications in Python. The efficient implementation of the toolbox also allows reachability analysis to be done on dynamical systems of up to six dimensions, which was not the case previously.

The toolbox supports the following algorithms: level-set based Hamilton-Jacobi PDEs to obtain BRT, BRS, and time-to-reach (TTR) value function, value iterations for Markov Decision Process (MDP) with continuous state space and action space. Our toolbox is implemented in Python and HeteroCL [9]. The front-end used to initialize various problem formulation is written in Python while the backend implementing the algorithms are written in HeteroCL. HeteroCL is a python-based domain-specific language (DSL) that is based on Tensor Virtual Machine (TVM) [10], a framework that optimizes deep learning programs as computation graph structures. HeteroCL is built on top of TVM that allows imperative programming in its syntax, which allows more flexibility in writing diverse algorithm implementations. Similar to TVM, HeteroCL decouples algorithm definitions from the scheduling transformations that can optimize the runtime of the programs. For our implementation, we attribute the significant improvement in running time to the scheduling optimizing scheme available that we use and also the optimization done on graphs by the TVM framework. Our toolbox is available online at **https://github.com/SFU-MARS/optimized_dp**.

In the next few subsections, we will provide an overview of related software packages, optimizedDP's software structure, features, and algorithms supported in our toolbox, and then describe the structure of our library and implementation details.

2.0.1 Related work

We are aware of other existing toolboxes that are most commonly used for solving HJ PDE:

ToolboxLS [2] is a library that contains many subroutines written in MATLAB for solving a variety of cases of an HJ PDE. HelperOC is a wrapper around ToolboxLS that utilizes these subroutines for convenient computation of BRT and BRS through solving the time-dependent HJ PDE. HelperOC contains many different examples of system dynamics used in BRT computation. In comparison with optimizedDP, ToolboxLS and HelperOC is more mature and contains more advanced numerical schemes to approximate derivatives and numerical integration as well as diverse MATLAB subroutines used for visualizing plots. One downside to ToolboxLS and HelperOC is that the toolbox can be quite slow for large problems and not possible for problems with systems that have higher than 4 dimensions.

Another minor disadvantage of ToolboxLS is that it is written in MATLAB, which is a proprietary software package whose licenses have to be renewed yearly.

BEACLS is a library that contains implementations of all the features available in **ToolboxLS** and **HelperOC** in C++ and CUDA with support on GPU. This toolbox tries to solve the computational inefficiency issue that ToolboxLS faces. However, the biggest downside of this toolbox is that it's quite hard to use due to the problem specification having to be written in C++.

Our toolbox optimizedDP introduced here is an ongoing effort that tries to combine the best features of the two toolboxes: codes that are easy to use, understand while keeping computations efficient. In addition, we are also aware of other software libraries for solving value iteration in MDP:

Markov Decision Process for Python is a software package written in Python that includes many algorithm implementations for MDP such as value iteration, policy iteration, relative value iteration, etc. However, the package does not support continuous state space and action space in value iteration. This package is available at https://pymdptoolbox.readthedocs.io/en/latest/index.html.

POMDP [11] is a software package written in Julia that contains a variety of algorithm solvers for MDP and reinforcement learning algorithms such as Deep Q-learning, Monte Carlo Tree Search, etc. The package also contains many examples for different types of problem initialization and has an easy-to-use interface. But like **Markov Decision Process for Python**, the package does not support continuous state space and action space value iteration in MDP.

2.1 Overview of the software toolbox structure

The general structure of our toolbox is shown in **Figure** 2.1. To begin the computation, users first specify a problem specification file that imports the file *solver.py*, which contains definitions of APIs calls for three different types of computations. The problem specification file can be quite straightforward to Python users as it is mostly written with Python and Numpy libraries. To assist users in initializing their problem specifications, there are provided python libraries that include grid generations (file *grid_processing.py*), initialization of signed distance initial value function (file *ShapesFunction.py*). These functions could be extended or customized towards users' needs, as long as the APIs are called correctly. The only HeteroCl part of the problem specifications is the system dynamics description, which is passed to the backend solvers to build a computational graph at the beginning.

The *solver.py* file maps the problem specification to the corresponding algorithm implementations. Each algorithm is implemented in HeteroCL which creates a computational graph of the algorithm and then returns an executable. The executables are used as a function to which parameters of the problem are passed. Once the results are computed

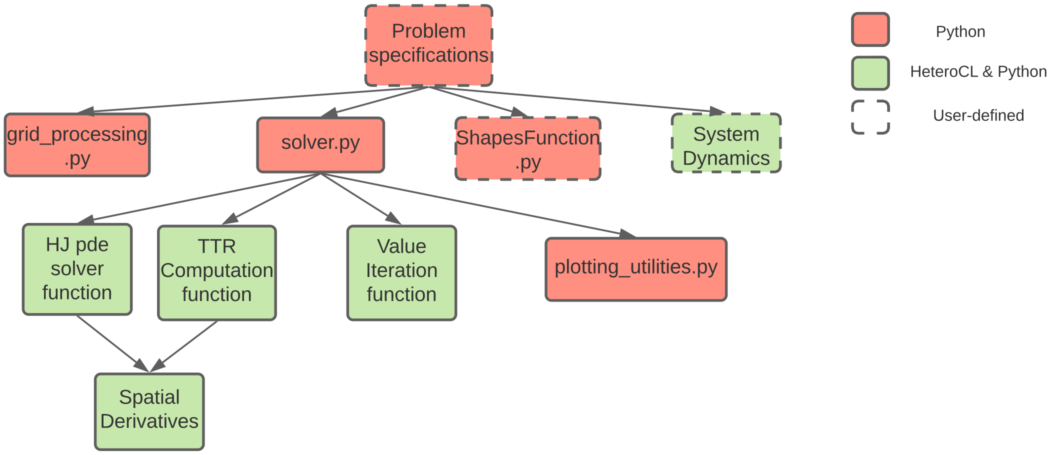


Figure 2.1: The overall structures of OptimizedDP. The red-colored files are written in Python and uses Numpy library for problem specifications, grid initialization, and plotting utilities. The green-colored files are written in a mix of HeteroCL and Python. The *system dynamics* file has to be provided by users and needs to be a Python object that contains problem parameters, and subroutines that determine optimal controls, optimal disturbances and compute the rate of change for each system state. These system object's subroutines are then called by the rest of the green-colored files that provide implementations of the core algorithms.

and converted to a Numpy array, available visualization package libraries in Python can be used to display the result. To make visualization of high dimension array easier, the file *plotting_utilities.py* creates a wrapper around the plotly's *Isosurface* function that can be called to visualize 3D value function representing various slices of the multidimensional result array. In addition, certain computation modules can be cross-used among different algorithms implementation. Specifically, the module spatial derivatives computation is both used in solving HJ PDE and TTR value function.

2.2 Algorithms supported and common features

2.2.1 Continuous Markov Decision Process (MDP) & Value Iteration

Markov Decision Process is a model that is useful to study the optimal behavior of a target system in react to the change of external environments. An MDP is usually described by a tuple (S, A, T, R, γ, H) where S is the state space, A is the action space, T is the transition between states probability matrix, γ is the discount factor and H is the time horizon. The key assumption of MDP is that the next state transition of a system is only dependent on the current state and action. This assumption is described by the following equation

$$\mathbf{P}(S_{t+1}|S_t, A_t) = \mathbf{P}(S_{t+1}|S_t, A_t..., S_0, A_0) \qquad (2.1)$$

where $S_t \in S$, and $A_t \in A$. In MDP, the discounted return G_t at time step t is defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \Sigma_{k=0}^{n} \gamma^k R_{t+k} \tag{2.2}$$

And the state value function $V_\pi(s)$ under a policy $\pi : S \to A$ is as follow

$$V_\pi(s) = E_\pi[G_t | S_t = s] = E_\pi[R_t + \gamma V_\pi(s') | S_t = s] \tag{2.3}$$

In MDP, the objective of the target system is to act according to an optimal policy $\pi^* : S \to A$ that can maximize the expected rewards received at each state over time. The main goal in MDP is to discover π^* along with the expected rewards received at every state. This objective and the basic properties of MDP are the backbone of all reinforcement learning algorithms.

Our toolbox provides an implementation of the value iteration algorithm for continuous state and action space (shown in **Algorithm** 1), which computes expected rewards $V_{\pi^*}(s)$ at every state given all the possible actions a state s can take.

Algorithm 1 Continuous MDP - Value Iteration Algorithm

```
1:  Discretize $S, A$
2:  $V_{t=0} \leftarrow 0$
3:  Repeat:
4:  for $s$ in $S$ do
5:      for $a$ in $A$ do
6:          $Q(s,a) \leftarrow R(s,a) + \sum_{s'} p(s'|s,a) V_{t-1}(s')$
7:          $V_{t+1}(s) \leftarrow max(V_{t+1}(s), Q(s,a))$
8:          $\Delta \leftarrow |V_t(s) - V_{t+1}(s)|$
9:          If $\Delta <$ threshold:
10:             Break
11:     end for
12: end for
```

Note that on line 6 of algorithm 1, (s') is obtained by considering the nearest neighbour that is the closest discretized state on grid.

2.2.2 Time-dependent Hamilton-Jacobi (HJ) Partial Differential Equation (PDE)

Our toolbox supports numerical computation for solving two HJ pdes. The first pde, which is solved in order to obtain BRS and BRT defined in [7], is as follow:

$$\begin{gathered} D_s\phi(z,s) + H(z, \nabla\phi(z,s)) = 0 \\ H(z, \nabla\phi(z,s)) = \min_{d(\cdot)} \max_{u(\cdot)} \nabla\phi(z,s)^T f(z,u,d) \\ \phi(z,0) = \phi_0(z), s \in [t, 0] \end{gathered} \tag{2.4}$$

The algorithm based on level-set methods for solving the above equation is implemented as in **algorithm** 2.

Algorithm 2 Time-dependent HJ PDE algorithm

```
Initialize $\phi_0$
// Compute Hamiltonian term, max and min derivative
for every grid point index i do
    Compute $\nabla\phi(z,s)$
    $u_{opt} \leftarrow \arg\max_{u \in \mathbb{U}} \nabla\phi(z,s)^\top f(z,u)$
    $\dot{z} \leftarrow f(z, u_{opt})$
    $H_i \leftarrow \nabla\phi(z,s)^\top \dot{z}$
    $minD \leftarrow min(minD, \nabla\phi(z,s))$
    $maxD \leftarrow max(maxD, \nabla\phi(z,s))$
end for
// Compute artificial dissipation
for every grid point index i do
    $\alpha_i \leftarrow \max_{p \in [minD, maxD]} \left|\frac{\partial H}{\partial p}\right|$
    $H_i \leftarrow H_i - \alpha_i \frac{D^+V_i - D^-V_i}{2}$
    $\alpha_d^{max} \leftarrow max(\alpha_d^{max}, \alpha_i)$
end for
// Compute integration time step
$\Delta t \leftarrow (\Sigma_{d=1}^N \frac{|\alpha_d^{max}|}{\Delta z_d})^{-1}$
// First order Runge-Kutta (RK) integrator.
$V_{t+1} \leftarrow H\Delta t + V_t$
```

Currently, the toolbox only supports algorithms for up to 6 dimensions. Although the toolbox developed by [2] supports an arbitrary number of dimensions through the usage of various operation tricks supported by MATLAB, each of the temporary variables such as spatial derivatives, system dynamics, etc. for each grid point is stored in a multidimensional array. This approach is not ideal for the performance of an already expensive computation in two ways (illustrated in **Fig.** 2.2). Firstly, the approach does introduce extra overhead of memory in the implementation. These redundant overheads increase linearly as we go up the dimensional ladder, which can limit the number of dimensions to which the algorithm can be applied. Secondly, each of the components for all grid points has to be computed before the final output, which results in bad locality for high-dimensional problems.

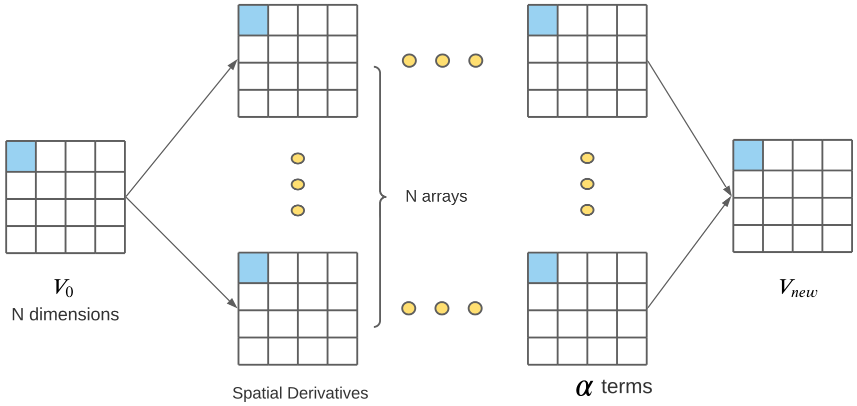


Figure 2.2: In [2], each temporary variables are stored in multidimensional arrays. As we increase N, the number of dimensions, the number of temporary multi-dimensional array goes up linearly. If the depth of the computation is large, the total amount memory used for temporary variables will exceed system's DRAM capabilities, limiting computations to small problems only.

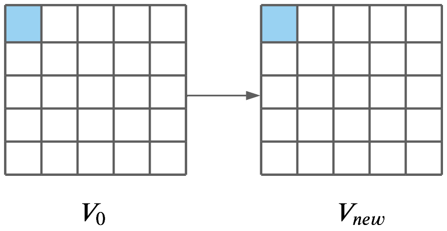


Figure 2.3: OptimizedDP's implementation of **algorithm 2** does not buffer temporary variables into multidimensional arrays. Instead, within each grid iteration, a grid point value in V_{new} is directly computed.

2.2.3 Time-independent Hamilton-Jacobi (HJ) Partial Differential Equation (PDE)

In addition, optimizedDP provides an implementation of the Lax-Friedrich sweeping algorithm described in [12] (also shown in **algorithm** 3) for efficiently computing time-to-reach BRS without numerical integration. Given a target set $\mathcal{T} \subseteq \mathcal{R}^n$, the time-to-reach function is defined as follow:

$$\phi(z) = \min_{d(\cdot)} \max_{u(\cdot)\in\mathbb{U}} \min\{t \mid z(t) \in \mathcal{T}\} \tag{2.5}$$

By dynamic programming principle, this TTR $\phi(z)$ can be obtained by solving the following HJI PDE:

$$\begin{aligned} H(z, \nabla\phi(z)) &= 0 \\ \phi(z) &= 0, z \in \mathcal{T} \\ H(z, \nabla\phi(z)) &= \min_{d(\cdot)} \max_{u(\cdot)} (-\nabla\phi(z)^T f(z, u, d) - 1) \end{aligned} \tag{2.6}$$

The advantage of **algorithm** 3 compared to obtaining the TTR function through solving the time-dependent HJ PDE is that less memory is required and the final result for TTR generally converge much faster. In solving both equations, users can set the objective of the u and d to either min or max for the term H.

Algorithm 3 Lax-Friedrich sweeping algorithm for TTR

```
Initialize $\phi(z) \leftarrow 0$ for $z \in \mathcal{T}$ and $\phi(z) \leftarrow 100$ for $z \notin \mathcal{T}$
while $|\phi - \phi_{old}| < \epsilon$ do
    $\phi \leftarrow \phi_{old}$
    for  grid index i not in boundary do:
        Compute $\nabla\phi(z, s)$
        $u_{opt} \leftarrow \arg\max_{u \in \mathbb{U}} \nabla\phi(z, s)^\top f(z, u)$
        $\dot{z} \leftarrow f(z, u_{opt})$
        $H_i \leftarrow \nabla\phi(z, s)^\top \dot{z}$
        $\sigma \leftarrow \left|\frac{\partial H}{\partial p}\right|$
        $c \leftarrow \frac{\Delta z}{\sigma}$
        $\phi_i^{new} \leftarrow c(-H_i + \sigma\frac{\phi_{i+1}+\phi_{i-1}}{2\Delta z})$
        $\phi_i \leftarrow min(\phi_i^{new}, \phi_i)$
    end for
    // Update the grid points at boundary
    $\phi_1^{new} \leftarrow min(max(2\phi_2 - \phi_3, \phi_3), \phi_1)$
    $\phi_N^{new} \leftarrow min(max(2\phi_{N-1} - \phi_{N-2}, \phi_{N-2}), \phi_N)$
end while
```

2.2.4 Common Components and Features

Grid

Similar to the ToolboxLS [2], our toolbox allows users to create a Cartesian grid by specifying the number of grid nodes, upper bound, and lower bound for each dimension. Users can also specify which dimension is periodic. The ghost points at the boundary for non-periodic dimension, by default, are extrapolated based on the formula described in the file *addGhostExtrapolate.m* in ToolboxLS. The *grid* is implemented as a Python object.

Initial Condition

To initialize different implicit surface shapes, we have implemented many initial conditions which represent shapes such as cylinders, spheres, and lower/upper planes. In addition, there are utility functions that operate on these geometry shapes such as union, intersection. All of these functions are written with Python and Numpy, and could be easily extended by users using the attribute *grid.vs* exposed by the *grid* object.

Time Integration

OptimizedDP provides the standard first-order accurate strong stability preserving (SSP) Runge-Kutta (RK) integrator. The maximum timestep used for integration is determined by the Courant–Friedrichs–Lewy (CFL) [13] condition.

Spatial Derivatives

Currently, OptimizedDP provides an implementation of derivatives approximation method that includes first order upwind approximation and second order accurate essentially non-oscillatory (ENO) [14] scheme.

Visualization

OptimizedDP provides an interface that helps visualize 3D isosurface of implicit surface for high dimensional systems. This interface allows users to specify the slice indices for higher dimensional systems and set the threshold value of isosurface for visualization. At its core, the interface calls the function *Isosurface* available in *plotly* library, which will show the isosurface plot in a browser. Users can also opt to use other software packages for visualization once the final result is obtained.

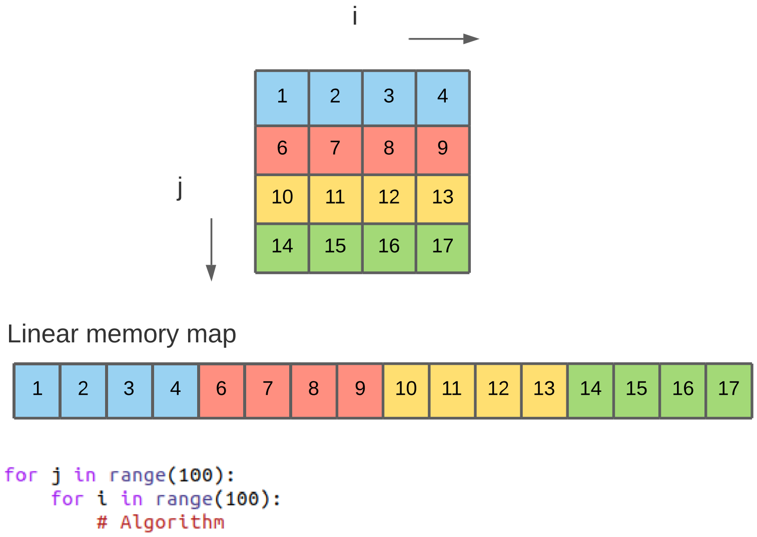
2.3 Implementation Details

We decided to implement each algorithm mentioned in the above section for every dimension separately, each with its own nested loop implementation. Even though this can be a tedious process for development, there are few reasons for this approach. First, we would like to keep the algorithm implementation descriptive, intuitive, and easy to be understood by users who are familiar with the algorithms. More importantly, we would like to optimize the computation using some of the schedule transformations available in HeteroCL without introducing extra redundancy and overhead in the code. Currently, the toolbox only supports algorithms for up to 6 dimensions. In our experience, this is the limit beyond which tabular dynamic programming is no longer practical on a personal computer of maximum 32GB DRAM.

In this section, we are going to discuss in more detail the optimization techniques enabled in HeteroCL we use in our implementations, and note that not all of them are applicable to all the algorithms implementation.

2.3.1 Loop ordering (Algorithm 1, 2, 3)

This optimization applies to all of the algorithm implementations. One important factor that can have a substantial impact on the performance of a program when dealing with high dimensional arrays is memory locality. By knowing the memory layout of the N-dimensional array, we have a nested loop order that follows this layout order which results in more cache efficiency. To abide by Numpy's memory layout, the implementation, by default, has the highest dimension being the most inner loop and the lowest dimension being the most outer loop. We are aware that in some cases depending on the system dynamics, this default order might not be the most optimal. However, on average, we see that this implementation already gives a very good performance.



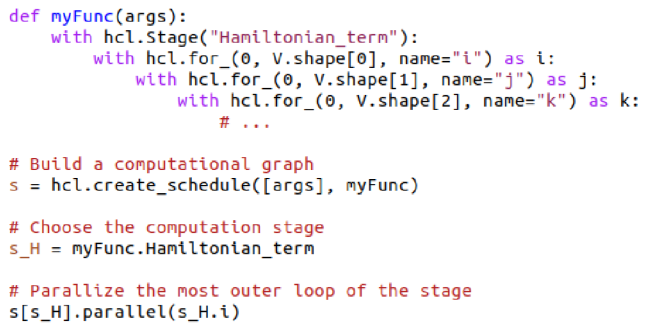
```
for j in range(100):
    for i in range(100):
        # Algorithm
```

Figure 2.4: Nested loop order that follows the linear memory map will take advantage of the fast cache memory access

For users who are keen on experimenting with different loop orders for their specific system dynamics problem, they could easily customize and swap the loop orders without introducing errors into the algorithm already implemented by using transformative primitives available in HeteroCL.

2.3.2 Parallel threading (Algorithm 2)

One important characteristic of **algorithm** 2 is that each grid point, within the same iteration, can be computed independently, and therefore in parallel. Note that this parallelization of computation only applies to solve time-dependent HJ PDE equations. This property can be taken advantage of to accelerate the computation greatly. In particular, each computation of **algorithm** 2 on a grid point can be assigned a thread to it (shown in Fig. 2.6). In HeteroCL, this could be done by applying the transformation primitive *parallel* to a loop computation as follow.



```
def myFunc(args):
    with hcl.Stage("Hamiltonian_term"):
        with hcl.for_(0, V.shape[0], name="i") as i:
            with hcl.for_(0, V.shape[1], name="j") as j:
                with hcl.for_(0, V.shape[2], name="k") as k:
                    # ...

# Build a computational graph
s = hcl.create_schedule([args], myFunc)

# Choose the computation stage
s_H = myFunc.Hamiltonian_term

# Parallize the most outer loop of the stage
s[s_H].parallel(s_H.i)
```

Figure 2.5: Computation can be parallelized by applying the *parallel* primitive

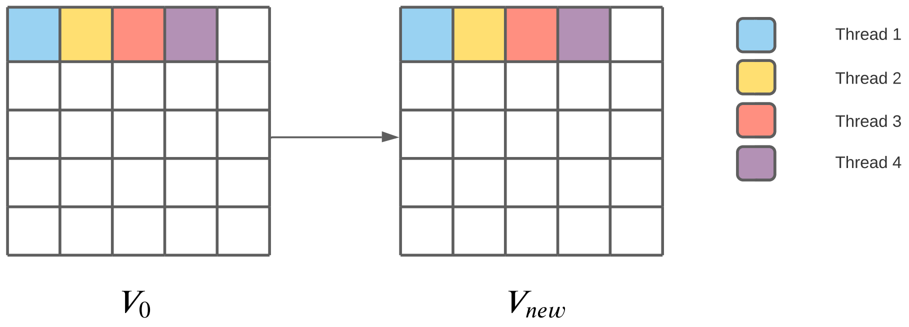


Figure 2.6: Each grid computation is assigned a thread for parallel computation

Under this primitive is an implementation of multi-threading in C++ provided by the TVM framework. The general idea of this multi-threading implementation is that there is a pool of threads where each thread pops and assigns itself a task (computation) from a task queue. The number of threads used is equal to the maximum number of hyper-threads available in the system. More details about the implementation are available online in the HeteroCL code base.

2.3.3 Alternating sweeping directions (Algorithm 1, 3)

This optimization is more algorithmic and less on the computer system level, and is applied to in-place value updating. In our toolbox, this approach is used in the implementation of value iteration algorithm and time-to-reach value function. The main idea is that the traversing directions on a multidimensional grid do not have to be fixed and can be alternated in different iterations until convergence. This technique has been shown to compute time-to-reach value function for 2D systems [12].

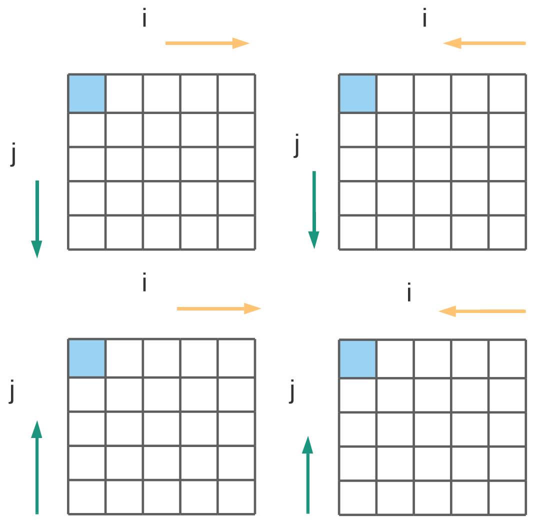


Figure 2.7: Each grid iteration can have alternating traversing direction for each dimension

The benefit of this optimization is that final value results would converge at a faster rate than a fixed iterating direction. As we go up the dimensional ladder, the total number of different possible alternating directions increases exponentially. Because of that, we do not implement all possible iterating directions for each dimensional problem. Instead, for each dimensional problem, we have a total of 8 different iterating directions.

2.4 Result

In this section, we first compare the performance of optmizedDP against the time-dependent HJ PDE implementation in ToolboxLS and BEACLS for various number of dimensions. These results are performed on a 16-thread Intel(R) Core(TM) i9-9900K CPU at 3.60GHz.

Table 2.1: First order ENO scheme performance against ToolboxLS and HelperOC

Dimensions	3D	4D	5D	6D
Grid points	100^3	60^4	40^5	25^6
OptimizedDP	0.56 seconds	19 seconds	7 seconds	1 day
MATLAB	3.8 seconds	196 seconds	223 seconds	Not possible
Speedup	$\times$**7**	$\times$ **10**	$\times$ **32**	N/A
Maximum difference	$\mathbf{1.4}\times10^{-6}$	$\mathbf{7.0}\times10^{-6}$	$\mathbf{1.4}\times10^{-6}$	N/A

Table 2.2: First order ENO scheme performance against BEACLS on CPU

Dimensions	3D	4D
Grid points	100^3	60^4
OptimizedDP	0.56 seconds	19 seconds
BEACLS	1.5 seconds	244 seconds
Speedup	×**3**	× **13**

Table 2.3: Second order ENO scheme performance against Toolbox and HelperOC

Dimensions	3D	4D	5D	6D
Grid points	100^3	60^4	40^5	25^6
OptimizedDP	0.7 seconds	23 seconds	10 seconds	1 day
MATLAB	12 seconds	678 seconds	754 seconds	Not possible
Speedup	×**17**	× **29**	× **75.4**	N/A
Maximum difference	**0.037**	**0.25**	**0.1**	N/A

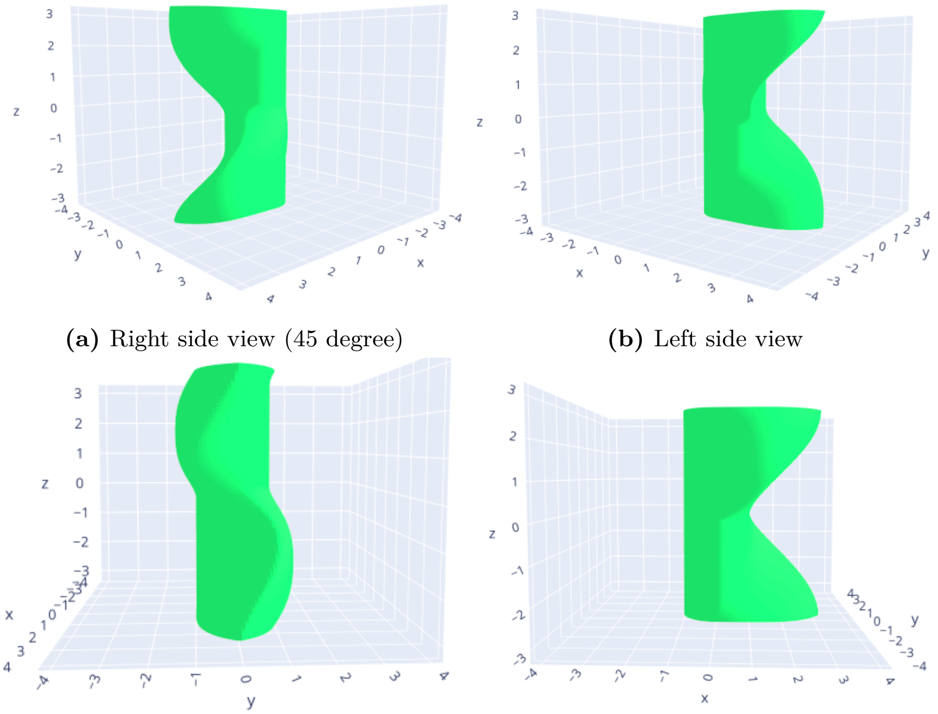
Table 2.4: Second order ENO scheme performance against BEACLS on CPU

Dimensions	3D	4D
Grid points	100^3	60^4
OptimizedDP	0.7 seconds	23 seconds
BEACLS	3 seconds	6420 seconds
Speedup	×**4**	× **279**

For 3D system example, we compute BRT for the following canonical pairwise Dubins Car's system dynamics:

$$\begin{aligned} \dot{x} &= -v_a + v_b \cos\theta + ay \\ \dot{y} &= v_a \sin\theta - ax \\ \dot{\theta} &= b - a \end{aligned} \tag{2.7}$$

where x, y, θ are the relative positions and heading, v_a and v_b are the evaders and pursuer's velocity, a and b are the control input of the evader and pursuer respectively. 3D plots of the BRT are shown in Figure. 2.8.



(a) Right side view (45 degree)

(b) Left side view

(c) Front view

(d) Right side view (90 degree)

Figure 2.8: Sub-zero level set is the green surface shown in the plots

For 6D system example, we have the following system dynamics used for computing tracking error bound of an underwater vehicle with disturbances as defined in [15]:

$$\begin{aligned}
\dot{x}_\alpha &= u_r + V_{f,x}(x,z,t) + d_x - b_x \\
\dot{z}_\alpha &= w_r + V_{f,z}(x,z,t) + d_z - b_z \\
\dot{u}_r &= \frac{1}{m - X_{\dot{u}}}((\bar{m} - m)A_{f,x}(x,z,t) \\
&\quad - (X_u + X_{|u|u}|u_r|)u_r + T_A) + d_u \\
\dot{w}_r &= \frac{1}{m - Z_{\dot{w}}}((\bar{m} - m)A_{f,z}(x,z,t) \\
&\quad - (-g(m - \bar{m})) - (Z_w + Z_{|w|w}|w_r|)w_r \\
&\quad + T_B) + d_w \qquad (2.8) \\
\dot{x} &= u_r + V_{f,x}(x,z,t) + d_x \\
\dot{z} &= w_r + V_{f,z}(x,z,t) + d_z
\end{aligned}$$

where x, z denote the vehicle position, u_r, w_r represent relative velocities between vehicle and water flow, x_α, z_α denote relative position between tracker and planner. The control inputs are T_A, T_B, planning inputs are b_x, b_z, and disturbances are d_x, d_z, d_u, d_w. The problem parameters are $m, \bar{m}, X_{\dot{u}}, Z_{\dot{w}}, X_u, X_w, X_{|u|u}, Z_{|w|w}$. Contour plots of distances between the tracker and planner are shown in Figure 2.9.

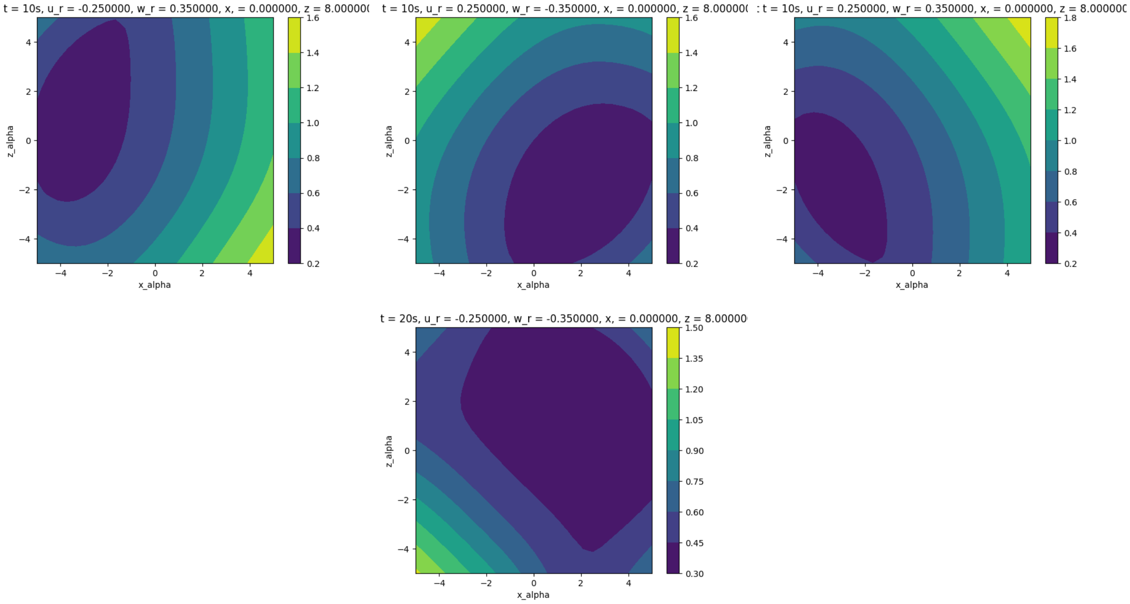


Figure 2.9: 2D contour plots of relative distances between the planner and tracker at different array indices and time. Each of the color on the vertical bar represents a distance value

Since there is no existing library that implements value iteration and algorithm 3 for time-independent HJ PDE, we only compare a version of value iteration written in pure Python, a commonly used language for reinforcement learning and MDP, with optimizedDP's implementation. This result is performed on a machine running on AMD A10-8700P CPU with 4 cores at 3.2GHz.

Table 2.5: Value Iteration for 3D grid

Grid points	25 × 25 × 9	40 × 40 × 20	60 × 60 × 40
OptimizedDP	1.35 seconds	6.44 seconds	28 seconds
Python	4.9 seconds	42 seconds	266 seconds
Speedup	×**3.6**	× **6.5**	× **9.5**

It can be observed that as the problem size increases, the gap in performance between optimizedDP and other existing implementations becomes larger. This proves that our toolbox is better for working with high-dimensional problems.

2.5 Limitation and future work

OptimizedDP toolbox is still a work in progress. Despite having better performance in terms of computational efficiency, optimizedDP is still missing some features that are available in other toolboxes. To make the toolbox more complete, we plan on adding new features to the toolbox such as higher order ENO scheme for derivatives approximation, more complex custom functions such as interpolation that can be used within a HeteroCL graph.

Chapter 3

Real-Time Hamilton-Jacobi Reachability Analysis of Autonomous System With An FPGA

In the previous chapter, we have provided a toolbox that allows roboticists to easily apply reachability analysis to system models. While 3D or smaller systems could be computed quickly with multi-core CPUs, practical systems that usually involve 4 to 5 state variables can take several minutes to hours to solve the HJ PDE. This prevents the HJ formulation to be applied to real-time systems on which safety is increasingly demanded. There have been works that proposed decomposing high dimensional systems into smaller tractable sub-systems that can exactly compute [16] or overapproximate the BRT in certain cases [17]. However, the challenge of applying HJ formulation on real-time systems remains, since some systems cannot be decomposed lower than four dimensions, and over-approximation is introduced if projection methods are used.

In this chapter, we expand the limit of the number of dimensions for which we can directly compute the BRT in *real time* through the use of an FPGA. As general-purpose computers no longer double their performance every two years due to the end of Moore's law, we have seen examples of successful hardware accelerations in other areas such as machine learning's training/inference [18, 19, 20], robot's motion planning [21]. This is the source of our inspiration, as we aim to apply hardware acceleration to safety verification.

We will show that we could accelerate HJ reachability analysis for a 4D system by solving Algorithm 4 efficiently on an FPGA. Our computing architectures can used as a computing service hosted by AWS cloud that can potentially be useful for roboticists trying to apply reachability analysis for real-time collision avoidance of robots. The codes of the hardware design and instructions for running it on an FPGA is online at **https://github.com/sfu-**

arch/HJ_solver. Before going into the details of this design, we will introduce some background and terminologies that are relevant to the understanding of our methodologies.

3.1 Background

Let $s \leq 0$ be time and $z \in \mathbb{R}^n$ be the state of an autonomous system. We assume that the evolution of the system state over time is described by a system of ordinary differential equations (ODE) below:

$$\dot{z} = \frac{\mathrm{d}z(s)}{\mathrm{d}s} = f(z(s), u(s), d(s)), u(s) \in \mathcal{U}, d(s) \in \mathcal{D}, s \in [t, 0] \tag{3.1}$$

where $u(\cdot)$ and $d(\cdot)$ denote the control and disturbance function respectively and defined in [7] and are drawn from the set of measurable functions:

$$u(\cdot) \in \mathbb{U} := \{\phi : [t, 0] \to \mathcal{U}, \phi(\cdot) \text{ is measurable}\} \tag{3.2}$$

$$d(\cdot) \in \mathbb{D} := \{\phi : [t, 0] \to \mathcal{D}, \phi(\cdot) \text{ is measurable}\} \tag{3.3}$$

where $\mathcal{U} \subseteq \mathbb{R}^{n_u}$ and $\mathcal{D} \subseteq \mathbb{R}^{n_d}$ are compact and $t < 0$. The system dynamics $f : \mathbb{R}^n \times \mathcal{U} \times \mathcal{D} \to \mathbb{R}^n$ are assumed to be uniformly continuous, bounded and Lipschitz continuous in z for fixed $u(\cdot)$ and $d(\cdot)$. Given $u(\cdot)$ and $d(\cdot)$, there exists a unique trajectory that solves equation (3.1) [22]. The trajectory or solution to equation (3.1) is denoted as $\zeta(s; z, t, u(\cdot), d(\cdot)) : [t, 0] \to \mathbb{R}^n$ which starts from state z at time t under control $u(\cdot)$ and disturbances $d(\cdot)$. ζ satisfies (3.1) almost everywhere with the initial condition $\zeta(t; z, t, u(\cdot), d(\cdot)) = z$.

3.2 Reachability Analysis & Hamilton-Jacobi (HJ) Partial Differential Equation (PDE)

In reachability analysis, we begin with a system dynamics described by an ODE and a target set $\mathcal{T} \subseteq \mathcal{R}^n$ that represents unsafe states/obstacles [7]. We then solve a HJ PDE to obtain the Backward Reachable Tube (BRT), defined as follows:

$$\begin{aligned} \bar{\mathcal{A}} = \{z : \exists d(\cdot) \in \mathbb{D}, \forall u(\cdot) \in \mathbb{U}, \exists s \in [t, 0], \\ \zeta(s; z, t, u(\cdot), d(\cdot)) \in \mathcal{T}\} \end{aligned} \tag{3.4}$$

The target set is represented by the implicit surface function $V_0(z)$ as $\mathcal{T} = \{z : V_0(z) \leq 0\}$. The BRT is then the zero sub level set of a value function $V(z, t)$ defined as below.

$$V(z, t) = \min_{d(\cdot)} \max_{u(\cdot) \in \mathbb{U}} \min_{s \in [t, 0]} V_0(\zeta(s; z, t, u(\cdot), d(\cdot))) \tag{3.5}$$

We follow the standard assumption that the disturbance function is taken from the set of non-anticipative strategies defined in [23]. To guarantee safety under the worst-case assumption, we model the interaction between control and disturbance as zero-sum differential game, in which the control input and disturbances have opposite objectives. This ensures that outside of the BRT defined in (3.4), there exists a control that guarantees avoidance of the target set.

The value function $V(z,t)$ can be obtained as the viscosity solution of the following HJ variational inequality:

$$\begin{gathered}\min\{D_s V(z,s) + H(z, \nabla V(z,s)), V(z,0) - V(z,s)\} = 0 \\ V(z,0) = V_0(z), s \in [t, 0] \\ H(z, \nabla V(z,s)) = \min_{d(\cdot)} \max_{u(\cdot)} \nabla V(z,s)^T f(z,u,d)\end{gathered} \tag{3.6}$$

The optimal control can be obtained from the value function as follows:

$$u_{opt} = \arg\max_{u \in \mathbb{U}} \nabla V(z,s)^\top f(z,u,d) \tag{3.7}$$

Exact analytical solutions to (3.6) are rarely possible and typically obtained through numerical methods. Numerical toolboxes based on level set methods such as [2] are commonly used to obtain the solution on a multi-dimensional grid for (3.6). The grid is first initialized with V_0, and integrated backward in time until convergence.

3.3 Numerical solution

The simplest scheme used for approximation of spatial derivaties on a grid is a central-difference scheme, which is defined as follow for one dimension.

$$\begin{aligned} D^- V_i &= \frac{V_i - V_{i-1}}{\Delta z}, \\ D^+ V_i &= \frac{V_{i+1} - V_i}{\Delta z}, \\ p = DV_i &= \frac{D^+ V_i + D^- V_i}{2} \end{aligned} \tag{3.8}$$

.

When V_i is at the grid boundary (ie. $i = N-1, i = 0$ where N is the size of the dimension), we apply the following formulas for extrapolating the left and right value grid point as implemented in toolbox [2]:

$$V_{i-1} = V_i + |V_i - V_{i+1}| * sign(V_i) \tag{3.9}$$

$$V_{i+1} = V_i + |V_i - V_{i-1}| * sign(V_i) \tag{3.10}$$

The algorithm used to solve equation 3.6 for 4D systems, based on level-set method, is described in algorithm 4.

Algorithm 4 4D Grid Value Function Solving Procedure

```
 1: Initialize $V_0[N_1][N_2][N_3][N_4]$
 2: //Compute Hamiltonian term, max and min derivative
 3: for $i = 0 : N_1 - 1;\ j = 0 : N_2 - 1;\ k = 0 : N_3 - 1;\ l = 0 : N_4 - 1$ do
 4:     Compute (3.8) for $0 \leq dim \leq 3$
 5:     $minD_{dim} \leftarrow min(minD_{dim}, D_{dim}V_{i,j,k,l})$
 6:     $maxD_{dim} \leftarrow max(maxD_{dim}, D_{dim}V_{i,j,k,l})$
 7:     $u_{opt} \leftarrow \arg\max_{u \in \mathbb{U}} \nabla V(z,s)^\top f(z,u)$
 8:     $\dot{z} \leftarrow f(z, u_{opt})$
 9:     $H_{i,j,k,l} \leftarrow \nabla V(z,s)^\top \dot{z}$
10: end for
11: // Compute dissipation and add to H
12: for $i = 0 : N_1 - 1;\ j = 0 : N_2 - 1;\ k = 0 : N_3 - 1;\ l = 0 : N_4 - 1$ do
13:     $\alpha_{dim,(i,j,k,l)} \leftarrow \max_{p \in [minD_{dim}, maxD_{dim}]} \left| \frac{\partial H}{\partial p_{dim}} \right|$
14:     $H_{i,j,k,l} \leftarrow H_{i,j,k,l}$
          $- \Sigma_{dim=1}^{4} \alpha_{dim,(i,j,k,l)} \frac{D_{dim}^{+} V_{i,j,k,l} - D_{dim}^{-} V_{i,j,k,l}}{2}$
15:     $\alpha_{dim}^{max} \leftarrow max(\alpha_{dim}^{max}, \alpha_{dim,(i,j,k,l)})$
16: end for
17: //Compute stable integration time step
18: $\Delta t \leftarrow (\Sigma_{d=1}^{4} \frac{|\alpha_d^{max}|}{\Delta z_d})^{-1}$
19: $V_{t+1} \leftarrow H\Delta t + V_t$
20: $V_{t+1} \leftarrow min(V_t, V_{t+1})$
21: $\epsilon \leftarrow |V_{t+1} - V_t|$
22: if $\epsilon < threshold$ then
23:     $V_t \leftarrow V_{t+1}$
24:     Go to line 3
25: end if
```

3.4 Field Programmable Gated Array (FPGA)

FPGA are intergrated circuits that can be configured and programmed for specific applications. FPGAs consist of multiple configurable logic blocks (CLB) that implement digital logic using look-up table (LUT). In order to correctly behave according to programmer's logic gate specifications, these LUTs store the correct output and input combination of the gates that corresponds to programmer's logic specification.

The advantages of FPGAs over CPUs and GPUs are energy efficiency, deterministic latency and high level of parallelism. On FPGA, efficient systems comprises of fast computing cores and fast data distribution from the memory. Depending on the application, the memory access and computing pattern will vary. Compute cores must request and receive all the necessary data from the memory component before proceeding with the computation. If the memory component cannot provide all data accesses the application requires to proceed at once, cores have to stall and wait, slowing down the computation.

General-purpose CPU/GPU are often architected towards a reasonable performance for a wide variety of applications, but not optimized for any particular application. FPGAs, on the other hand, can be customized by leveraging knowledge about the details of the computing workload to design an efficient system. With an FPGA, one could control and achieve a higher degree of parallelism from the digital hardware level at the cost of programmability.

3.4.1 Memory Technologies

Taking into account the distinction between different types of available memory technologies is crucial for designing efficient hardware accelerator. Here we will focus mainly on two types of memory: Dynamic Random Access Memory (DRAM) and Blocks of Random Access Memory (BRAM).

DRAM works by storing a data bit (1 or 0) inside a memory cell consisting of a capacitor and a transistor. The electric charge inside the cell leaks over time and hence a DRAM requires a controller that need to refresh the data periodically. DRAM typically come in size of 8GB to 32GB with maximum bandwidth of 20GB/s.

BRAM is a small unit amount of memory implemented on FPGA that typically has a size of 4 to 32 Kilobits. An FPGA can have from hundred of kilobytes to eight megabytes of BRAM. Each of the BRAM could provide at most two memory accesses every clock cycles. All the memory accesses are synchronized to the logic clock and hence, the latency is solely dependent on the the clock period.

3.5 Algorithm Optimization

In this section, we will show that Algorithm 1, with three full grid iterations, repeats redundant computation that can be reduced to one iteration instead using the bounded input property in the dynamics on a grid. We will analyze this property for our target dynamical system, which is an extended 4D Dubins car model:

$$\begin{aligned} \dot{x} &= v\cos(\theta) & \dot{y} &= v\sin(\theta) \\ \dot{v} &= a & \dot{\theta} &= v\frac{\tan(\delta)}{L} \end{aligned} \tag{3.11}$$

where (x, y) represents the positions, v represents the speed, and θ represents the heading. The control inputs are acceleration $a \in [-a, a]$ and the steering angle $\delta \in [-b, b]$. The explicit formula for the Hamiltonian term is: $H = \left(\frac{\partial V}{\partial x}\dot{x} + \frac{\partial V}{\partial y}\dot{y} + \frac{\partial V}{\partial v}\dot{v} + \frac{\partial V}{\partial \theta}\dot{\theta}\right)$. In this case, the term $\left|\frac{\partial H}{\partial p_{dim}}\right|$, which is used as scaling factor (line 14 - Algorithm 5), is not dependent on p_{dim} and equal to the absolute value of the rate of change of state component dim. Hence, we do not need to keep track of the maximum and minimum spatial derivatives; this allows us to merge second grid iteration (line 12 - Algorithm 4) into the first one . This change is reflected in line 7 of Algorithm 5. Furthermore, we can observe that the rates of change of each state component are only dependent on other state's components and the control input. The maximum rate of changes α_{dim}^{max} is therefore, resulted from the combination of largest value from each state component and largest possible control input, which is constant over time. Instead of re-computing Δt every time and then iterating through the 4D grid array again, we pre-compute Δt and re-use it for all of the time iterations. Combining these ideas together, throughout this paper, we will use Algorithm 5 with one grid iteration for our FPGA implementation, which is more computationally efficient. We empirically evaluate the correctness of this Algorithm 5 in the Result section.

Algorithm 5 Optimized 4D Grid Value Function Solving Procedure

```
 1: Initialize $V_0[N_1][N_2][N_3][N_4]$
 2: for $i = 0 : N_1 - 1$; $j = 0 : N_2 - 1$; $k = 0 : N_3 - 1$; $l = 0 : N_4 - 1$ do
 3:     Compute (3.8) for $1 \leq dim \leq 4$
 4:     $u_{opt} \leftarrow \arg\max_{u \in \mathbb{U}} \nabla V(z, s)^\top f(z, u)$
 5:     $\dot{z} \leftarrow f(z, u_{opt})$
 6:     $H_{i,j,k,l} \leftarrow \nabla V(z, s)^\top \dot{z}$
 7:     $H_{i,j,k,l} \leftarrow H_{i,j,k,l}$
              $- \Sigma_{dim=1}^{4} |\dot{z}_{dim}| \frac{D_{dim}^{+} V_{i,j,k,l} - D_{dim}^{-} V_{i,j,k,l}}{2}$
 8:     $V_{t+1,(i,j,k,l)} \leftarrow H_{i,j,k,l} \Delta t_{precomputed} + V_{t,(i,j,k,l)}$
 9:     $V_{t+1,(i,j,k,l)} \leftarrow \min(V_{t,(i,j,k,l)}, V_{t+1,(i,j,k,l)})$
10: end for
11: $\epsilon \leftarrow |V_{t+1} - V_t|$
12: if $\epsilon < threshold$ then
13:     $V_t \leftarrow V_{t+1}$
14:     Go to line 2
15: end if
```

3.6 Hardware Architecture

In this section, our goal is to build a custom hardware design on an FPGA that solves Algorithm 5 efficiently. Before going into details of the design, we will introduce some terminologies that will be relevant throughout the next section. In digital systems, time is discretized into the unit of a *clock cycle*, which is the amount of time it takes for an operation such as computing, loading, and storing to proceed. Each clock cycle typically is a few nanoseconds.

Our custom hardware comprises two main components: an on-chip memory buffer, and processing elements (PE) (shown on the right side of Fig. 3.1). The memory buffer is on-chip storage, providing access to all the grid points a PE needs to compute a new value function. Each PE is a digital circuit that takes in value grid points from the memory buffer to compute a new value function at a particular grid point according to Algorithm 5 (lines 3-10). In the following subsections, we will go into the details of each component.

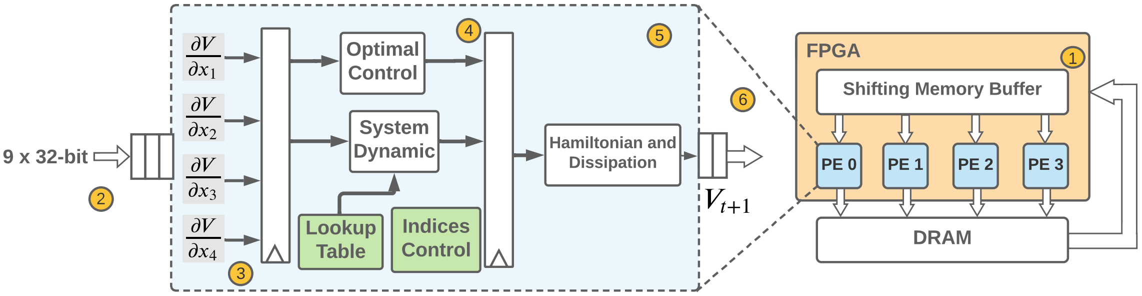

9 x 32-bit
2
$\frac{\partial V}{\partial x_1}$
$\frac{\partial V}{\partial x_2}$
$\frac{\partial V}{\partial x_3}$
$\frac{\partial V}{\partial x_4}$
3
Optimal Control
System Dynamic
Lookup Table
Indices Control
4
5
Hamiltonian and Dissipation
6
V_{t+1}
FPGA
1
Shifting Memory Buffer
PE 0
PE 1
PE 2
PE 3
DRAM

Figure 3.1: Overview of the accelerator architecture implemented on an FPGA ① The accelerator on an FPGA consists of Memory buffer and PEs. The value array V, residing in DRAM, is copied to FPGA's memory buffer which distributes data to the PEs to concurrently execute algorithm 5. ② Every clock cycle, each PE takes new input from the memory buffer to start executing Algorithm 5 for new indices. ③ Spatial derivatives in all dimensions, defined in (3.8), are computed (line 3 in Algorithm 5). ④ Each PE has its own lookup table that's used to quickly index the state values for system dynamics computation (line 5 in Algorithm 5). Optimal control is also determined at the same time. ⑤ The Hamiltonian term and the dissipation in (line 6-7 in Algorithm 5) are then computed using the output from previous steps. ⑥ New output values are computed (line 8-9 Algorithm 5) written back to DRAM.

3.6.1 Memory Buffer Micro - Architecture

The memory buffer has the following key design objectives: (1) minimize on-chip memory usage and external DRAM accesses while (2) concurrently providing value grid points to each PE every clock cycle.

One challenge of working with a multi-dimensional grid is that the value function over the whole grid can take up tens of megabytes and therefore its entirety cannot fit into a state-of-the-art FPGA's on-chip memory. In our design, instead of storing the entire grid on-chip, the value of V at grid points are streamed continuously from DRAM into the on-chip memory buffer, which is implemented as a First In First Out (FIFO) queue data structure. Every clock cycle, values of new grid points are fetched from DRAM and pushed onto the back FIFO queue while the oldest grid points are popped from the front of the queue. Our objective is to minimize this length of this FIFO queue, and hence the amount of memory usage. Shown in [24] the minimum length of the FIFO queue is equal to the maximum reuse distance of a value grid point, which is dependent only on smallest $N-1$ dimensions of the grid. Based on the grid's size in each dimension, the minimum length of the FIFO queue is as follows:

$$2N_1N_2N_3 + pe + 1 \tag{3.12}$$

where pe is the number of PEs used in the design. The expression in (3.12) is an order of magnitude smaller than the grid's size of $N_1N_2N_3N_4$ and can fit on FPGA's on-chip memory.

The next challenge is to design the FIFO queue such that it can provide value of the grid points for the PEs to simultaneously start computing new value function at grid points every clock cycle. On FPGA, a FIFO queue is physically mapped to standard *Blocks of Random Access Memory* (BRAM). Each BRAM has two-ports and can concurrently request at most two read/write in the same clock cycle. One way to increase the number of access per clock cycle is to duplicate the FIFO queue with multiple BRAMs, but this would not work well for multi-dimensional arrays since these array copies easily exceed an FPGA's on-chip memory.

A different technique, called *memory banking*, partitions a FIFO queue into multiple smaller FIFO queues mapped into multiple BRAMs. Our FIFO queue structure (shown in Fig. 3.2) is adapted from the parallel memory buffer microarchitecture described in [24], which is specialized for a 2D/3D grid. We extend [24] to be optimized for a 4D grid as follows. The on-chip FIFO structure is partitioned into four line buffers, which corresponds to the number of PEs in our design. Each line buffer is a sequence of connected FIFO queues of varying sizes (shown as the horizontal stacks of rectangles in Fig. 3.2) depending on which grid points need to be accessed by the PEs. The endpoints of these FIFO queues (shown as colored arrows in Fig. 3.2) are connected both the next FIFO queue and to the inputs to the PEs. At every clock cycle, values from four grid points are streamed from DRAM and enter the four buffer lines (left side of Fig. 3.2) then travel towards the right of the buffer lines to fill up the FIFOs until they are popped at the end of each line (right side of Fig. 3.2). As the values move along the FIFO queues, they are accessed by the PEs. To perform the computations in line 3 of Algorithm 5, the PEs need access to the adjacent grid points values in all 4 dimensions as illustrated by Fig. 3.2. The indices of the grid value needed by the PEs determine the sizes of each FIFO queue, which are shown in Fig. 3.2 for the first line. Note each line buffer supplies values to multiple PEs, as shown by the coloured arrows.

3.6.2 Processing Element Micro - Architecture

The PE has the following target design objectives: (1) increase compute throughput (defined as the number of value output generated per second) through pipelining, a technique that overlaps execution between iterations, (2) and ensure the correctness of the result while minimizing data transfer between DRAM and the FPGA using a lookup table.

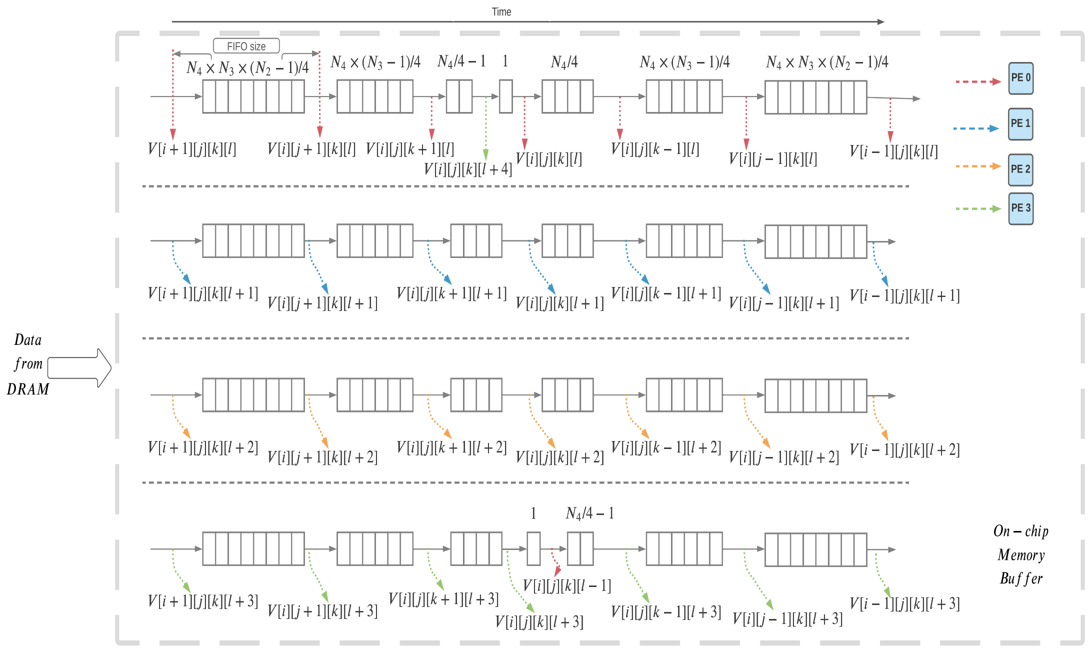
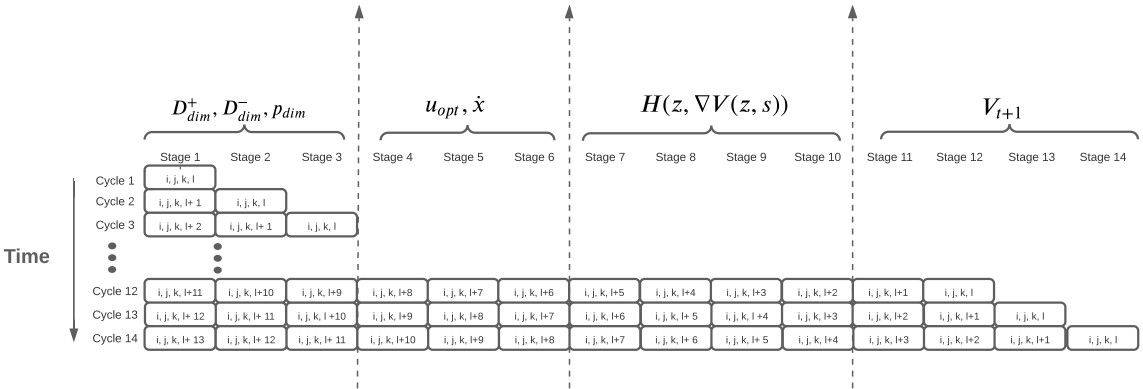


Figure 3.2: Four lines of memory buffer supply grid data to all four PEs. Each rectangle block is a FIFO queue synthesized to Block RAM (BRAM). New grid points are streamed from DRAM every clock cycle. They enter each buffer line (left-hand side) and old grid points at the end of the lines are discarded (right-hand side). The overhead notation is the size of the FIFO queue with N_1, N_2, N_3, N_4 as the four grid dimensions. Note that the FIFO's size depends only on three dimensions, while still able to supply all the necessary grid data to all PEs. The FIFO sizes are equal to the linear distances between every two grid points in memory.

To increase computation throughput, each PE is fully pipelined. Similarly to an assembly line, each PE operation is divided into multiple stages, and each stage within the pipeline is responsible for executing a step in Algorithm 5 (lines 3-10) for a particular index i, j, k, l. Following the sequential order of Algorithm 5, every clock cycle, the result from the previous stages will be forwarded into the next stage. At any time during the operations, the processing element is executing Algorithm 5 for multiple indices concurrently (explained in Fig.3.3).

To ensure that the computation is correct, inside each of the PE, there are indices that keep track of loop variables i, j, k, l, with the innermost loop variable incrementing by one every clock cycle. These indices are used to correctly address the state vectors during the system dynamics computation (line 5 of Algorithm 5). To avoid accessing external DRAM as much as possible, we store the list of values of each state variable in z or any fixed non-linear functions such as $\cos(\cdot)$ and $\sin(\cdot)$ of these states over the entire grid as a lookup table stored in the on-chip memory, since state vectors only depend on the grid configuration and do not change with the environment. Each PE has access to its own copy of look-up table to avoid communications between PEs. Having this data on-chip only requires a few kilobytes of memory.



	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Stage 6	Stage 7	Stage 8	Stage 9	Stage 10	Stage 11	Stage 12	Stage 13	Stage 14
Cycle 1	i, j, k, l													
Cycle 2	i, j, k, l+ 1	i, j, k, l												
Cycle 3	i, j, k, l+ 2	i, j, k, l+ 1	i, j, k, l											
Cycle 12	i, j, k, l+11	i, j, k, l+10	i, j, k, l+9	i, j, k, l+8	i, j, k, l+7	i, j, k, l+6	i, j, k, l+5	i, j, k, l+4	i, j, k, l+3	i, j, k, l+2	i, j, k, l+1	i, j, k, l		
Cycle 13	i, j, k, l+ 12	i, j, k, l+ 11	i, j, k, l +10	i, j, k, l+9	i, j, k, l+8	i, j, k, l+7	i, j, k, l+6	i, j, k, l+ 5	i, j, k, l +4	i, j, k, l+3	i, j, k, l+2	i, j, k, l+1	i, j, k, l	
Cycle 14	i, j, k, l+ 13	i, j, k, l+ 12	i, j, k, l+ 11	i, j, k, l+10	i, j, k, l+9	i, j, k, l+8	i, j, k, l+7	i, j, k, l+ 6	i, j, k, l+ 5	i, j, k, l+4	i, j, k, l+3	i, j, k, l+2	i, j, k, l+1	i, j, k, l

Figure 3.3: The pipelining schedule of a single PE. The PE's operation is an assembly line where multiple grid points are processed simultaneously. Within the pipeline, each stage computes a step of algorithm 5, and is occupied by a tuple comprised of the indices i, j, k, l. Every clock cycle, the results from these indices are forwarded to the next stage towards the end of the pipeline.

In our design, we use 4 PEs (shown in Fig. 3.1). The number of PEs is largely determined by how many grid points can be provided by the on-chip memory, which in turn is dependent on the DRAM speed. Each PE has an offset index idx, $0 \leq idx \leq 3$ associated with it. At the start of Algorithm 5, each PE takes as input a grid index $(i, j, k, l + idx)$ and its eight neighbours from the on-chip memory buffer to start computing $V_{t+1}(i, j, k, l{+}idx)$ according to Algorithm 5 (line 2-10).

3.6.3 Data representations

Computing a new value function based on Algorithm 5 involves multiple addition operations on floating-point numbers. At the hardware level, the addition of floating-point numbers is as computationally expensive as fixed-point multiplication, which would take up lots of resources and chip's area. Instead, we use fixed-point representations for our data to reduce the burden on the hardware. We will show in the next section that this has little impact on the correctness of the computation if the radix point is chosen carefully for the grid configuration.

3.7 FPGA implementation on AWS

3.7.1 Memory Interface

Fig. 3.4 shows the hardware modules that are used to correctly and effectively transfer data from DRAM to our hardware accelerator. Our FPGA design communicates data with external DRAM using the AXI protocol. Every external data requests (read/write commands) on FPGA has to pass through the module arbiter, which ensures fairness among the modules that require DRAM accesses. Each read/write command's addresses and request length are controlled and tracked by a Finite State Machine (FSM).

Data are accessed in bursts from/to DRAM with the data length being fixed throughout the computation. In particular for our design, we use a data length of 512 for our AXI modes. As array values are read sequentially, using this maximum possible data length can help us

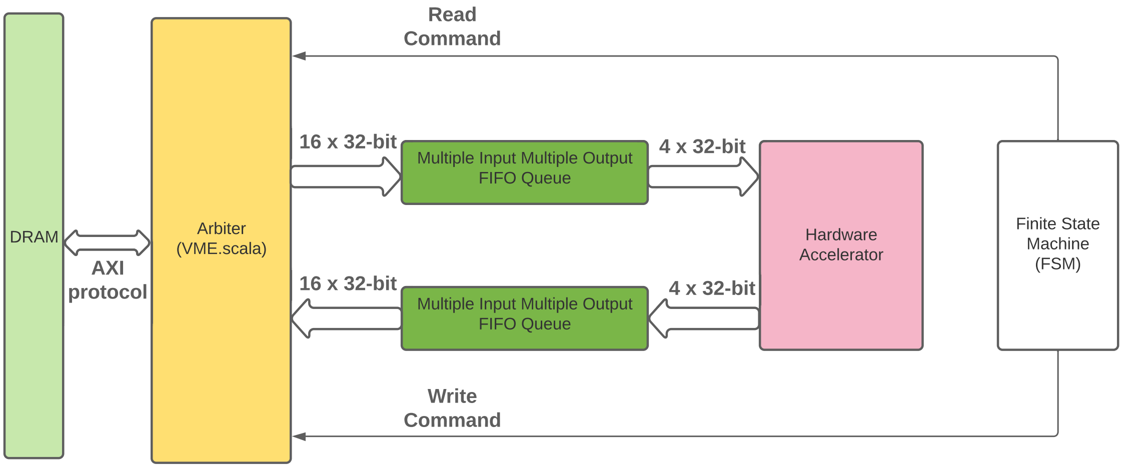
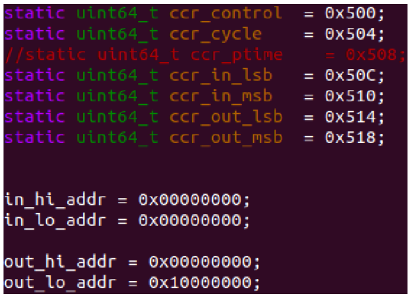


Figure 3.4: Components that are required for data communication between target accelerator and DRAM

achieve near optimal usage of the DRAM bandwidth. In addition, because the accelerator is not guaranteed to keep up with input supply rate, there has to be a FIFO queue (dark green blocks in Fig . 3.4) that buffers grid points read from DRAM. Each of the FIFO takes different size for the input width and the output width, and the depth of the queues are computed using the burst size and data length used for the AXI communications.

3.7.2 Interface between host and FPGA

To correctly offload the specialized computation from the host program to the FPGA, the following information has to be shared between the host and the FPGA: input data addresses, output data addresses, when the computation should start, and when the computation has finished. On FPGA, these information are stored in a set of control registers that can be written and read by both host program and fpga's custom logic. In particular, the design's control registers include two write address registers, two read address registers, a cycle counter register and a launch register. Each register is 32-bit wide. Note that the two 32-bit address registers make up a 64-bit address in the host system. In addition, the

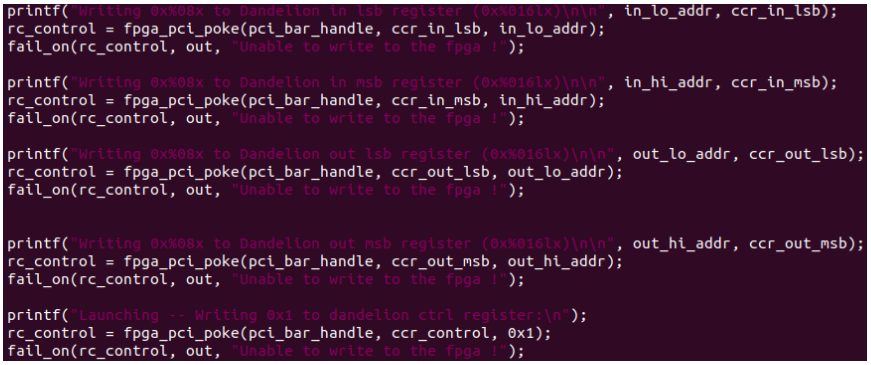


```
static uint64_t ccr_control   = 0x500;
static uint64_t ccr_cycle     = 0x504;
//static uint64_t ccr_ptime     = 0x508;
static uint64_t ccr_in_lsb    = 0x50C;
static uint64_t ccr_in_msb    = 0x510;
static uint64_t ccr_out_lsb   = 0x514;
static uint64_t ccr_out_msb   = 0x518;


in_hi_addr = 0x00000000;
in_lo_addr = 0x00000000;

out_hi_addr = 0x00000000;
out_lo_addr = 0x10000000;
```

Figure 3.5: PCIe mapped addresses of the control registers on FPGA

cycle counter register keeps track of the number of clock cycles that has passed since the beginning of computation and can be read back by the host program. The definitions of these control registers are defined in the file DCRF1.scala.

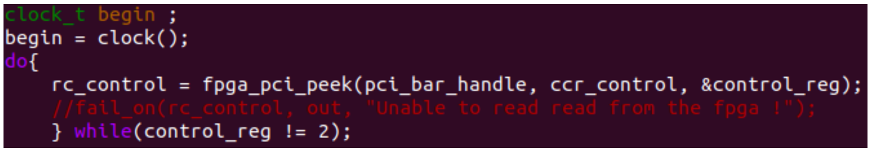
In the host program, these registers' addresses are mapped to the custom logic's registers through the PCIe bus (**Figure 3.5**). By writing to these mapped addresses, we can interface to the custom logic design from our host program (shown in **Figure 3.6**).



```
printf("Writing 0x%08x to Dandelion in lsb register (0x%016lx)\n\n", in_lo_addr, ccr_in_lsb);
rc_control = fpga_pci_poke(pci_bar_handle, ccr_in_lsb, in_lo_addr);
fail_on(rc_control, out, "Unable to write to the fpga !");

printf("Writing 0x%08x to Dandelion in msb register (0x%016lx)\n\n", in_hi_addr, ccr_in_msb);
rc_control = fpga_pci_poke(pci_bar_handle, ccr_in_msb, in_hi_addr);
fail_on(rc_control, out, "Unable to write to the fpga !");

printf("Writing 0x%08x to Dandelion out lsb register (0x%016lx)\n\n", out_lo_addr, ccr_out_lsb);
rc_control = fpga_pci_poke(pci_bar_handle, ccr_out_lsb, out_lo_addr);
fail_on(rc_control, out, "Unable to write to the fpga !");


printf("Writing 0x%08x to Dandelion out msb register (0x%016lx)\n\n", out_hi_addr, ccr_out_msb);
rc_control = fpga_pci_poke(pci_bar_handle, ccr_out_msb, out_hi_addr);
fail_on(rc_control, out, "Unable to write to the fpga !");

printf("Launching -- Writing 0x1 to dandelion ctrl register:\n");
rc_control = fpga_pci_poke(pci_bar_handle, ccr_control, 0x1);
fail_on(rc_control, out, "Unable to write to the fpga !");
```

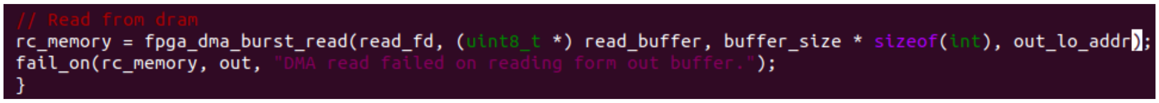
Figure 3.6: Control registers on FPGA can be accessed in C++ program by writing to PCI mapped addresses using function call fpga_pci_poke provided by AWS FPGA software library

To kickstart the computation on FPGA, the launch register is written 1 by the host program, and written 2 by the FPGA when the computation has finished. To determine if the computation on FPGA has finished, the host program will poll for the computation status on the FPGA by continuously evaluating the value of the launch register(shown in **Figure 3.7**). Finally, input data from the host program can be written to the FPGA's DRAM using DMA function calls, and output results on the FPGA side can be read back by the host program using the DMA burst read function (shown in **Figure 3.8**)



```
clock_t begin ;
begin = clock();
do{
    rc_control = fpga_pci_peek(pci_bar_handle, ccr_control, &control_reg);
    //fail_on(rc_control, out, "Unable to read read from the fpga !");
    } while(control_reg != 2);
```

Figure 3.7: The host program polls for the control status register on the FPGA



```
// Read from dram
rc_memory = fpga_dma_burst_read(read_fd, (uint8_t *) read_buffer, buffer_size * sizeof(int), out_lo_addr);
fail_on(rc_memory, out, "DMA read failed on reading form out buffer.");
}
```

Figure 3.8: Data residing on FPGA's DRAM can be transferred back to host program by calling function fpga_dma_burst_read provided by AWS FPGA software library.

3.8 Result

3.8.1 Hardware correctness

We evaluate the correctness of our FPGA on three different environments (shown in Fig. 3.9, 3.10, 3.11) in a room of size 6m x 5m. The target system is an RC car modeled by the dynamics described in equation 3.11 with the control input range and car length as follows: $a \in [-1.5, 1.5]$, $\delta \in [-\frac{\pi}{12}, \frac{\pi}{12}]$, and $L = 0.3$m.

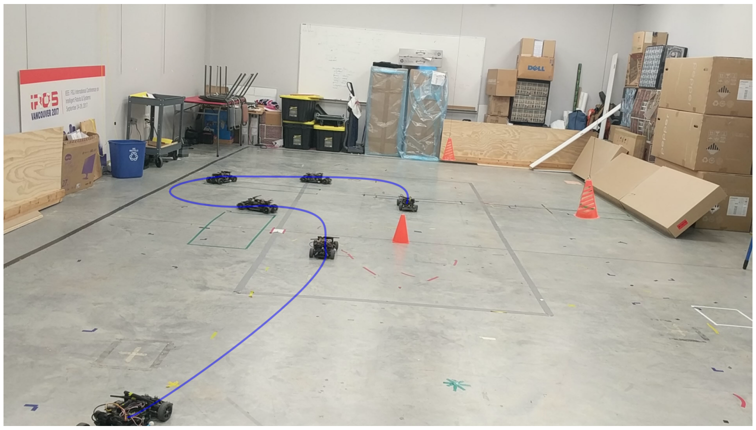


Figure 3.9: Environment 1

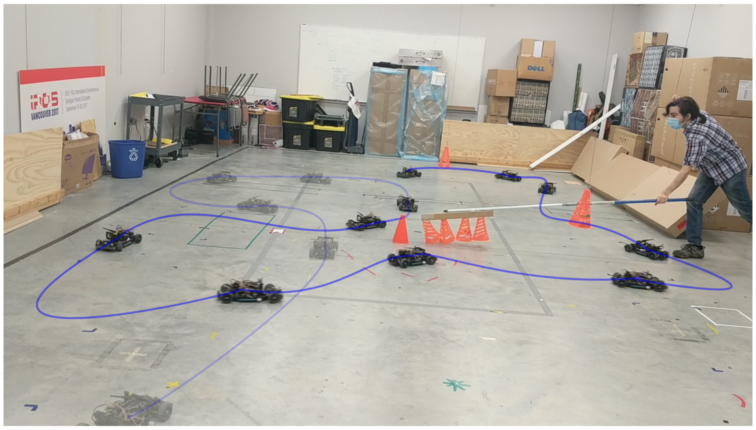


Figure 3.10: Environment 2

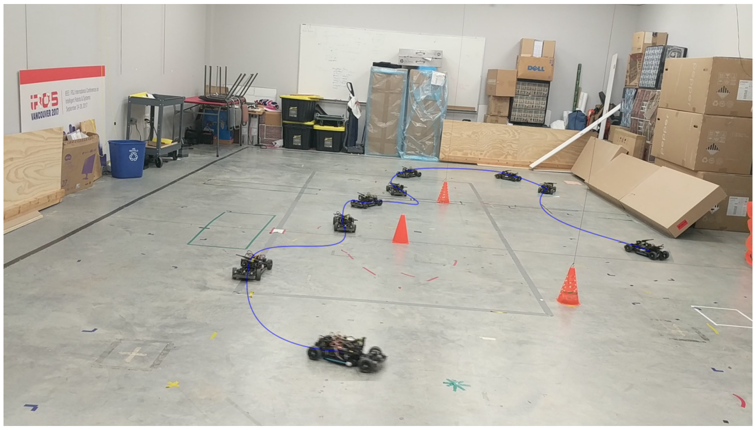


Figure 3.11: Environment 3

Each of the cone in the scenarios are used as obstacles and the implicit value function is as follow:

$$V_0(x, y, v, \theta) = \sqrt{(x - x_o)^2 + (y - y_o)^2} - R \tag{3.13}$$

where x_o and y_o are the obstacle's positions and $R = 0.08m$ is the radius of the cone. The sub-zero level set of the intial value function is shown in Figure. 3.12, 3.13, 3.14

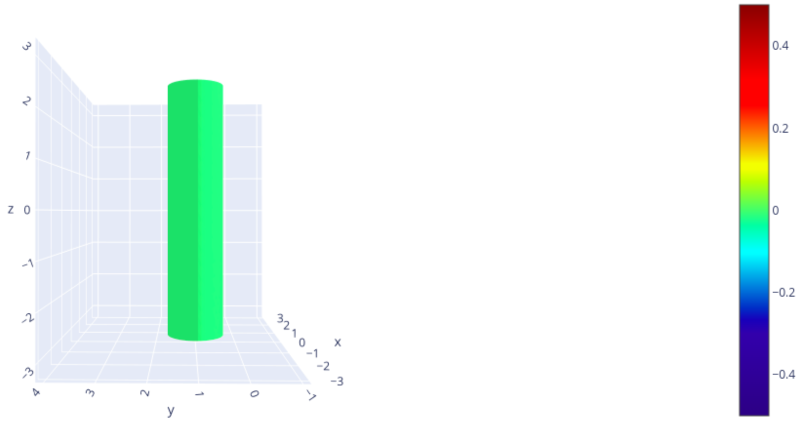


Figure 3.12: Initial value function for environment 1

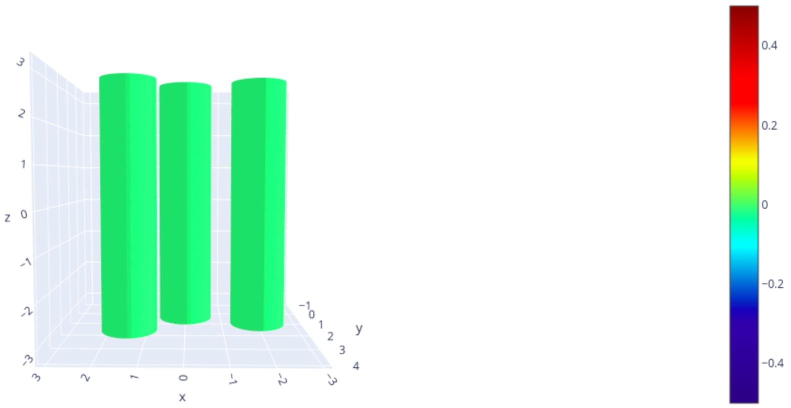


Figure 3.13: Initial value function for environment 2

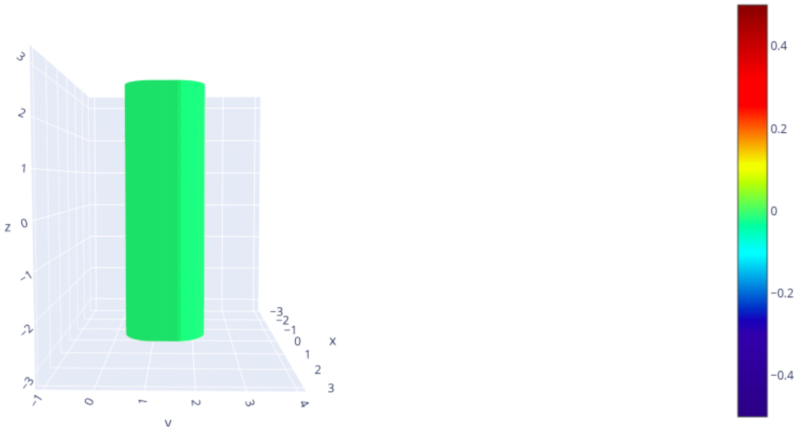


Figure 3.14: Initial value function for environment 3

Each of these initial level set function is passed to the FPGA in order to be solved for the converged value function. Because our FPGA works with fixed-point data representation, each initial value function has to be correctly converted before computation can start. In particular, we use 32 bits with 5 bits to represent the integer part (including sign) and 27 bits for the decimal part. With this choice, the precision of the result is $2^{-27} = 7.45 \times 10^{-9}$ and the range is from -16 to 16. The room area is 4m × 6m, hence the largest absolute distance is the diagonal of 7.2m. Therefore, the number of integer bits is enough to represent all possible values in the solution V, which has the physical interpretation of minimum distance to collision over time, given (3.5) and the choice of V_0 in (3.13).

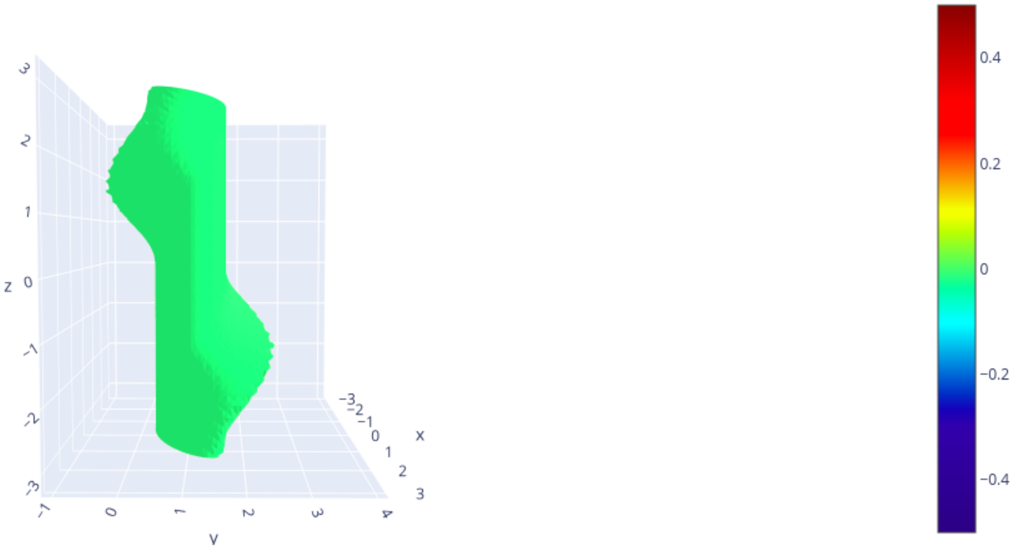


Figure 3.15: A 3D slice of 4D Backward Reachable Tube (BRT) for environment 1

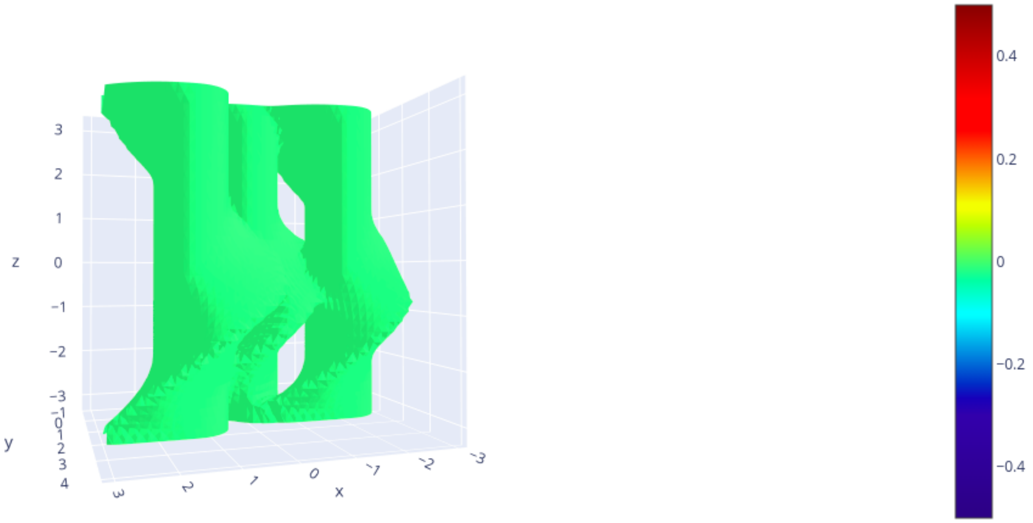


Figure 3.16: A 3D slice of 4D Backward Reachable Tube (BRT) for environment 2

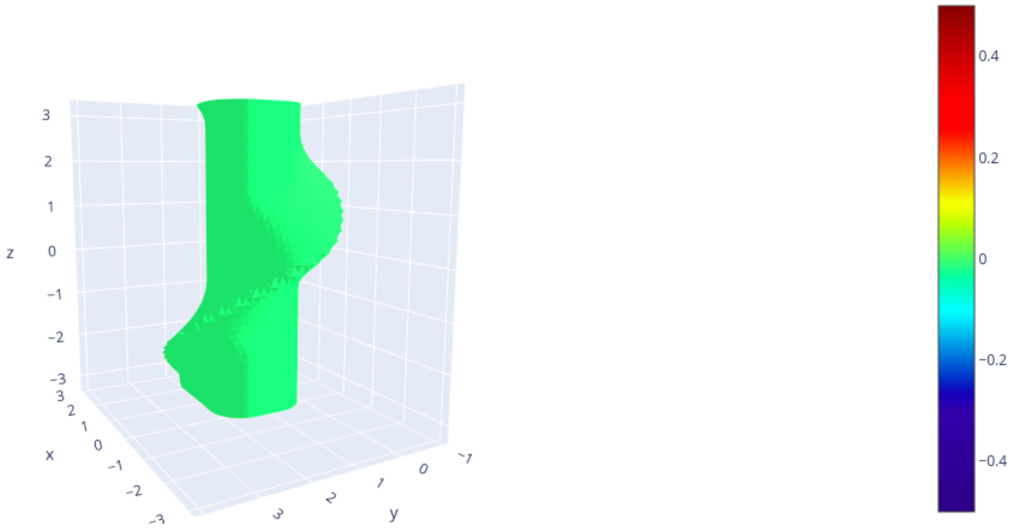


Figure 3.17: A 3D slice of 4D Backward Reachable Tube (BRT) for environment 3

The numerical error resulting from the different representations is shown for each three environments in Table 3.1. The errors are negligible due to precision difference between fixed-point and floating point number.

Table 3.1: Error Comparison

	Env. 1	**Env. 2**	**Env. 3**
Maximum Error	3.1×10^{-6}	1.67×10^{-6}	2.62×10^{-6}

Even though the computation is repeated for many iterations, the maximum error does not grow dramatically over time. We believe that is because of the convergence property of the value function. As time increases, the rate of changes in the value function at the grid points slows down, leading to a stable discrepancy between the correct floating point and fixed-point values.

3.8.2 Latency & speedup

To measure the speed up for all three environments, we compare the computation time on an AWS FPGA running at 196MHz against the toolboxes [1] and [2] running on a 16-thread Intel(R) Core(TM) i9-9900K CPU at 3.60GHz. The results are summarized in Tables 3.2 and 3.4. *Latency* refers to the time it takes to compute the the value function over the time horizon of 0.5s. For an FPGA, latency can be computed by multiplying the clock cycles with the clock period.

Table 3.2: FPGA

	Clock cycles	Period	Iterations	Latency
Env. 1	47965066	5.1 ns	67	0.2447s
Env. 2	47964977	5.1 ns	67	0.2447s
Env. 3	47965021	5.1 ns	67	0.2447s

Table 3.3: optimized_dp[1]

	Latency	Iterations	FGPA speed up
Env. 1	3.35 s	67	×**13.7**
Env. 2	2.99 s	67	×**12.2**
Env. 3	3.42 s	67	×**14**

Table 3.4: ToolboxLS[2]

	Latency	Iterations	FPGA speed up
Env. 1	25.11 s	70	×**103**
Env. 2	25.14 s	70	×**103**
Env. 3	25.18 s	70	×**103**

3.8.3 Resource utilization

On an FPGA, arithmetic operations on numbers are implemented using Digital Signal Processing (DSP) hardware or Look Up Table (LUT) that perform logical functions. Our design does not consume a significant portion of the available resources and thus could be scaled up to a larger grid size if needed. The resources usage of our design for 4 PEs is shown in the table below.

Table 3.5: RESOURCE CONSUMPTION

	LUT	BRAM	DSP
Used	26319	519	598
Available	111900	1680	5640
Utilization	**14.03%**	**30.89%**	**10.6%**

Note that since we are synthesizing our hardware design on AWS FPGA, the number of resources used also include other components that are specific for AWS's general application but not part of the core hardware accelerator. This means that our accelerator can potentially be synthesized on smaller, local FPGAs with fewer resources.

3.9 Robotic Experiment

3.9.1 Software Architecture

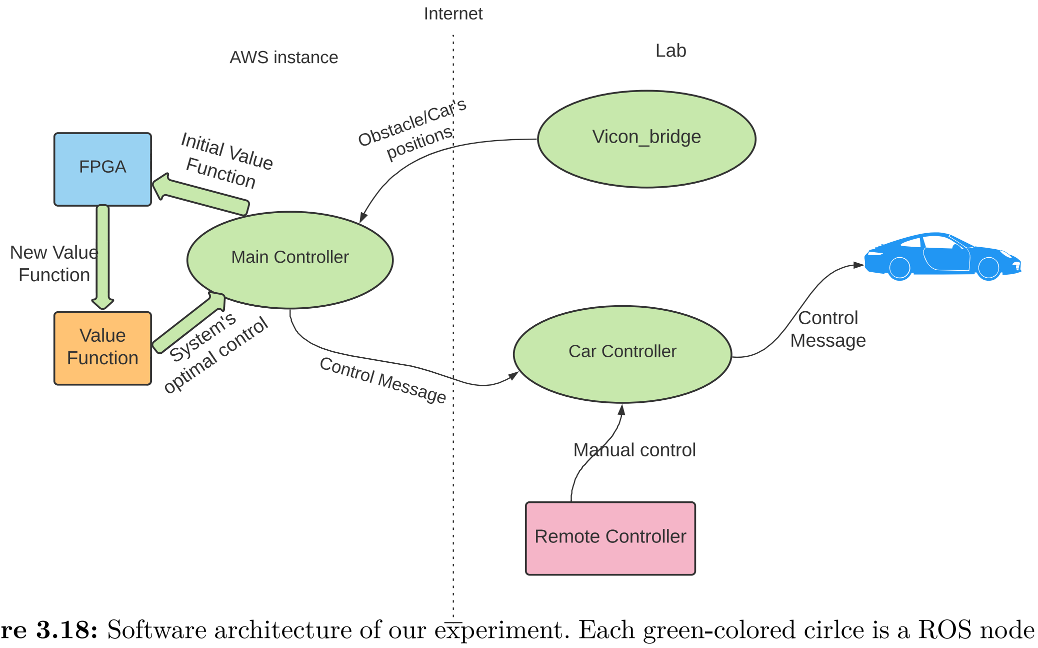


Figure 3.18: Software architecture of our experiment. Each green-colored cirlce is a ROS node that has inward arrow as subscribed message and outward arrow as publishing message.

The software architecture used in our experiment is shown in **Figure. 3.18** and is written in Python language. Our software uses Robotic Operating Systems (ROS) framework in order to handle the communications between different modules locating in different locations in a communication network over the internet. The software architectures include three ROS nodes (green-colored circle in **Figure. 3.18)**: the Vicon bridge node, the car controller node, and the main controller node. We will go into detail about the functionality of each node below.

The Vicon is responsible for detecting and tracking the positions of obstacles and the target system in real-time. The Vicon bridge ROS node takes in these input from the Vicon and publishes them in the form of ROS messages over the network. The main controller locating at the AWS server subscribes to these messages to complete two tasks. First, it computes the new value function multidimensional array using equation 3.13 with new obstacle's positions in the local environment. In order to use the hardware accelerator implemented on FPGA in a Python program, we use Pybind to wrap the C++ driver code that triggers computation on FPGA. Second, the ROS node sends back optimal control computed by taking the spatial derivatives at the car's current state. It should be noted that the car does not have to be applied the most optimal control input all the time, but only necessary when the car is close to the BRT (configuration states in which inevitable collisions happen in the future) in order to guarantee safety. Based on the car's positions, the ROS node will evaluate how safe the car currently is by checking the value function of the car's present position against a positive threshold value, and then decides if optimal control has to be sent to the car. If the current value function of the car is less than the threshold value, an override flag is set to true and the car must take the optimal control (line 8-9 of algorithm 6). These two tasks of the main controller are performed concurrently by multithreading. This approach has the advantage that the HJ PDE solving procedure does not block the optimal control computation from proceeding. In fact optimal control has to be sent at a much higher frequency of 50-70 Hz than the HJ PDE computation. If the latest value function is being updated, an older multidimensional value array is used to determine the optimal control for current state input instead.

The third ROS node (car controller in **Figure. 3.18**) subscribes to the control message published by the main controller and to the control sent from a remote manual controller. If the override flag of the optimal control message is false, the manual control is chosen to be applied to the car and a person can freely control it. However, when the car's position is less than a pre-defined threshold, the optimal control will then take over.

The algorithms used in the main controller and car controller are shown below in **Algorithm 6** and **7**.

Algorithm 6 AWS Controller

```
 1: Initialize $V[N_1][N_2][N_3][N_4]$
 2: Thread 1 (z):
 3:     OptTakeover $\leftarrow$ false
 4:     StateValue $\leftarrow V(z)$
 5:     $u_{opt} \leftarrow \arg\max_{u \in \mathbb{U}} \nabla V(z, s)^\top f(z, u)$
 6:     If StateValue $<$ threshold:
 7:         OptTakeover $\leftarrow$ true
 8:     Publish (OptTakeover, $u_{opt}$)
 9: Thread 2 (obstacle):
10:     Compute V based on the obstacle's positions
```

Algorithm 7 Car Controller

```
Global ctrl
Thread 1 (manualControl):
    ctrl ← manualControl
Thread 2 (AWSControl):
    If AWSControl.OptTakeover == True:
        Publish AWSControl.$u_{opt}$
    Else:
        Publish ctrl
```

3.9.2 Real-time Robot Demonstration

We plan out our experiments as follows: two obstacles locating in a room where one obstacle is static while other obstacle is constantly moved by a person trying to hit a moving car. The car is freely remotely controlled by another person when the flag *OptTakeover* is false. A video demonstrating obstacle avoidance is available at https://www.youtube.com/watch?v=8q6cJHmHNS8.

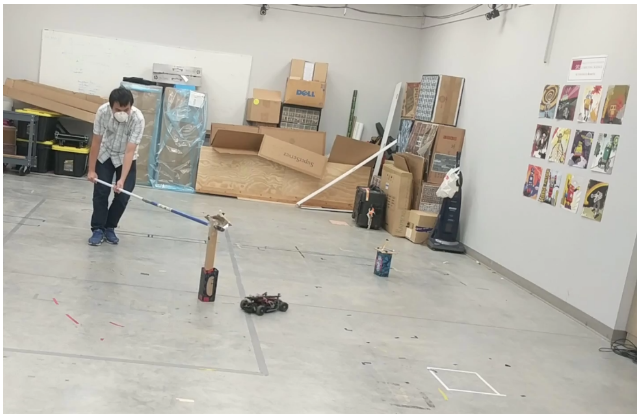


Figure 3.19: There are two obstacles inside the room where a person actively tries to move one obstacle around while the other obstacle is stationary

Chapter 4

Conclusion and Future Work

As autonomous systems are increasingly demanded in many industrial sectors, safety guarantees is an important aspect that is crucial in turning autonomous systems into reality. In order to achieve that, we need better, faster tools for theoretical research as well as more reliable, efficient systems on which safety-critical computation can run. Our work presented in this is one step towards that goal. In this thesis, we have contributed OptimizedDP, a toolbox that can be useful for researchers to solve HJ PDEs and value iteration problems more efficiently with an easy-to-use Python interface. We have also shown that a specialized computing architecture on FPGA could accelerate reachability analysis an order of magnitude faster than a multi-core CPU machine while having a more consistent computational latency.

4.1 Future work

OptimizedDP's core computation is written in HeteroCL, which lacks mathematical libraries that can be useful for problem specifications. The next step to resolve this issue is extending HeteroCL and TVM to include more complex mathematical functions. Several use cases of Optimized DP indicate that utility functions found in Numpy can be used as a guideline for this extension. To make OptimizedDP even more complete, features such as accurate, higher order numerical scheme of spatial derivates approximation, numerical integration, etc. will be added in the future.

Currently, the architecture we presented in Chapter 3 has to be manually specified using hardware description language Chisel/Scala. This process consumes a lot of time and effort. For systems with varying characteristics such as different system dynamics and different number of dimensions, the memory buffer and PE presented will have to be designed from scratch again. One potential direction to solve this issue is to build an automation framework that can produce a description of digital circuits given a high-level user's problem specification input. Another aspect of this work we would like to address in the future is better computational latency. The accuracy of solutions to HJ PDEs can be tuned by both the number of grid points and the numerical scheme of computation. Because of that, higher order numerical scheme can be adopted to compensate for fewer grid points used. As the current architecture presented is limited by DRAM bandwidth, this trade-off can potentially result in even faster computation of HJ PDE for high dimensional systems.

Bibliography

[1] M. Bui, "Optimized dynamic programming," 2020. Available at `https://github.com/SFU-MARS/optimized_dp`.

[2] I. Mitchell, "The flexible, extensible and efficient toolbox of level set methods," *J. Sci. Comput.*, vol. 35, pp. 300–329, 06 2008.

[3] S. Bak, H.-D. Tran, and T. T. Johnson, "Numerical verification of affine systems with up to a billion dimensions," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '19, (New York, NY, USA), p. 23–32, Association for Computing Machinery, 2019.

[4] A. B. Kurzhanski and P. Varaiya, "Ellipsoidal techniques for reachability analysis," in *Hybrid Systems: Computation and Control* (N. Lynch and B. H. Krogh, eds.), (Berlin, Heidelberg), pp. 202–214, Springer Berlin Heidelberg, 2000.

[5] A. K. Akametalu, C. J. Tomlin, and M. Chen, "Reachability-based forced landing system," *Journal of Guidance, Control, and Dynamics*, vol. 41, no. 12, pp. 2529–2542, 2018.

[6] M. Chen, Q. Hu, C. Mackin, J. F. Fisac, and C. J. Tomlin, "Safe platooning of unmanned aerial vehicles via reachability," in *2015 54th IEEE Conference on Decision and Control (CDC)*, pp. 4695–4701, 2015.

[7] M. Chen and C. J. Tomlin, "Hamilton–Jacobi Reachability: Some Recent Theoretical Advances and Applications in Unmanned Airspace Management," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 1, no. 1, pp. 333–358, 2018.

[8] K. Tanabe and M. Chen, "Beacls," 2021. Available at `https://github.com/HJReachability/beacls`.

[9] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2019.

[10] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," 2018.

[11] M. Egorov, Z. N. Sunberg, E. Balaban, T. A. Wheeler, J. K. Gupta, and M. J. Kochenderfer, "POMDPs.jl: A framework for sequential decision making under uncertainty," *Journal of Machine Learning Research*, vol. 18, no. 26, pp. 1–5, 2017.

[12] I. Yang, S. Becker-Weimann, M. J. Bissell, and C. J. Tomlin, “One-shot computation of reachable sets for differential games,” in *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, HSCC ’13, (New York, NY, USA), p. 183–192, Association for Computing Machinery, 2013.

[13] R. Courant, K. Friedrichs, and H. Lewy, “On the partial difference equations of mathematical physics,” *IBM Journal of Research and Development*, vol. 11, no. 2, pp. 215–234, 1967.

[14] S. Osher and C.-W. Shu, “High-order essentially nonoscillatory schemes for hamilton–jacobi equations,” *Siam Journal on Numerical Analysis - SIAM J NUMER ANAL*, vol. 28, 08 1991.

[15] S. Siriya, M. Bui, A. Shriraman, M. Chen, and Y. Pu, “Safety-guaranteed real-time trajectory planning for underwater vehicles in plane-progressive waves,” in *2020 59th IEEE Conference on Decision and Control (CDC)*, pp. 5249–5254, 2020.

[16] M. Chen, S. L. Herbert, and C. J. Tomlin, “Exact and efficient hamilton-jacobi guaranteed safety analysis via system decomposition,” in *2017 IEEE International Conference on Robotics and Automation, ICRA 2017, Singapore, Singapore, May 29 - June 3, 2017*, pp. 87–92, IEEE, 2017.

[17] A. Li and M. Chen, “Guaranteed-safe approximate reachability via state dependency-based decomposition,” in *2020 American Control Conference (ACC)*, pp. 974–980, 2020.

[18] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-datacenter performance analysis of a tensor processing unit,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA ’17, (New York, NY, USA), p. 1–12, Association for Computing Machinery, 2017.

[19] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[20] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “Eie: Efficient inference engine on compressed deep neural network,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 243–254, 2016.

[21] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. Sorin, “The microarchitecture of a real-time robot motion planning accelerator,” pp. 1–12, 10 2016.

[22] E. A. Coddington and N. Levinson, *Theory of ordinary differential equations*. McGraw-Hill New York, 1955.

[23] S. Bansal, M. Chen, S. Herbert, and C. J. Tomlin, “Hamilton-jacobi reachability: A brief overview and recent advances,” in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pp. 2242–2253, IEEE, 2017.

[24] Y. Chi, J. Cong, P. Wei, and P. Zhou, “Soda: Stencil with optimized dataflow architecture,” in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2018.