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

Minh Bui, Michael Lu, Reza Hojabr, Mo Chen, Arrvindh Shriraman

Abstract—Hamilton-Jacobi (HJ) reachability analysis is a powerful technique used to verify the safety of autonomous systems. HJ reachability is ideal for analysing nonlinear systems with disturbances and flexible set representations. A drawback to this approach is that it suffers from the curse of dimensionality, which prevents real-time deployment on safety-critical systems. In this paper, we show that a customized hardware design on an Field Programmable Gate Array (FPGA) could accelerate 4D grid-based HJ reachability analysis up to 14 times compared to an optimized implementation and 103 times compared to state-of-the-art MATLAB toolboxes on a 16-thread CPU. Because of this, we are able to achieve guaranteed real-time collision avoidance in dynamic environments that abruptly change with a 4D car model by re-solving the HJ partial differential equation (PDE) at a frequency of 4Hz on an FPGA. Our design can overcome the complex data access pattern while taking advantage of the parallel nature of the computations for solving the HJ PDE. The low latency of our computation is consistent, which is crucial for safety-critical systems. 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. We validate our approach in real world collision avoidance experiments with a robot car in a changing environment. We also provide the code of our hardware design and an AWS AFI image.

I. INTRODUCTION

As autonomous systems, such as self-driving cars, unmanned aerial vehicles, and rescue robots, are becoming 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 characterizes 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 [1] propose an approach that would scale up to one billion dimension for affine systems, and the authors in [2] 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 polytopes. 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 [3], multi-vehicle path planning, multi-player reach avoid games [4].

The main downside to HJ reachability is that the associated HJ partial differential equation (PDE) 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. This prevents the HJ formulation to be applied to real-time systems on which safety is increasingly demanded. 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 compute. There have been works that proposed decomposing high dimensional systems into smaller tractable sub-systems that can exactly compute [5] or overapproximate the BRT in certain cases [6]. 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 paper, 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 [7]–[9], robot’s motion planning [10]. This is the source of our inspiration, as we aim to apply hardware acceleration to safety verification.

In this paper, our contributions are as follows:

- We prototype a customized hardware design on an FPGA that can numerically solve the HJ PDE up to 14 times compared to an optimized implementation and 103 times compared to a state-of-the-art toolbox [11] on 16-thread CPU for 4D systems.
- We demonstrate that the system meets real-time requirement of guaranteeing safety in an abruptly changing environments by re-computing BRTs at 4Hz.
- We demonstrate obstacle avoidance with a robot car driving in an environment in which obstacles are moved abruptly during run time.
To the best of our knowledge, this is the first time online HJ reachability computations are used for real-time collision avoidance in an unknown, changing environment.

II. PRELIMINARIES

A. Hamilton-Jacobi Reachability Analysis

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:

$$
\dot{z} = \frac{dz}{ds}(s) = f(z(s), u(s), d(s)),
$$

$$
u(s) \in \mathcal{U}, d(s) \in D, s \in [t, 0]
$$

where $u(\cdot)$ and $d(\cdot)$ denote the control and disturbance function respectively and defined in [12]. The system dynamics $f: \mathbb{R}^n \times \mathcal{U} \times D \to \mathbb{R}^n$ 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 (1) [13]. The trajectory or solution to equation (1) is denoted as $z(t; z, u(\cdot), d(\cdot)) : [t, 0] \to \mathbb{R}^n$ which starts from state $z$ at time $t$ under control $u(\cdot)$ and disturbances $d(\cdot)$. $z$ satisfies (1) almost everywhere with initial condition $z(t; z, u(\cdot), d(\cdot)) = z$.

In reachability analysis, we begin with a system dynamics described by an ODE and a target set $T \subseteq \mathbb{R}^n$ that represents unsafe states/obstacles [12]. We then solve a HJ PDE to find the Backward Reachable Tube (BRT), defined as follows:

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

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

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

We follow the standard assumption that the disturbance function is taken from the set of non-anticipative strategies defined in [14]. 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 (2), 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:

$$
\min\{ D_s V(z, s) + H(z, \nabla V(z, s)), V(z, 0) - V(z, s) \} = 0
$$

$$
V(z, 0) = V_0(z), s \in [t, 0]
$$

$$
H(z, \nabla V(z, s)) = \min_{d(\cdot)} \max_{u(\cdot)} \nabla V(z, s)^T f(z, u, d)
$$

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

$$
u_{opt} = \arg \max_{u(\cdot) \in \mathcal{U}} \nabla V(z, s)^T f(z, u, d)
$$

Exact analytical solutions to (4) are rarely possible and typically obtained through numerical methods. Numerical toolboxes based on level set methods such as [11] are commonly used to obtain the solution on a multi-dimensional grid for (4). The grid is first initialized with $V_0$, and integrated backward in time until convergence.

B. Basic Numerical Solution

In this paper, we specialize the applications of HJ reachability to solving (4) for a system without adversarial disturbance on a 4D state space. To accomplish this, the first step is to discretize the state space on a grid, and initialize the grid with an implicit surface function.

\begin{algorithm}
\caption{4D Grid Value Function Solving Procedure}
\begin{algorithmic}[1]
\State Initialize $V_0[N_1][N_2][N_3][N_4]$
\State //Compute Hamiltonian term, max and min derivative
\For{$i = 0 : N_1 - 1; \quad j = 0 : N_2 - 1; \quad k = 0 : N_3 - 1; \quad l = 0 : N_4 - 1$}
\State Compute (6) for $0 \leq \text{dim} \leq 3$
\State $\min D_{\text{dim}} \leftarrow \min(D_{\text{dim}}, D_{\text{dim}}V_i,j,k,l)$
\State $\max D_{\text{dim}} \leftarrow \max(D_{\text{dim}}, D_{\text{dim}}V_i,j,k,l)$
\State $u_{\text{opt}} \leftarrow \arg \max_{u(\cdot) \in \mathcal{U}} \nabla V(z, s)^T f(z, u(\cdot))$
\State $\dot{z} \leftarrow f(z, u_{\text{opt}})$
\State $\nabla V(z, s) \leftarrow \nabla V(z, s)^T \dot{z}$
\EndFor
\State // Compute dissipation and add to $H$
\For{$i = 0 : N_1 - 1; \quad j = 0 : N_2 - 1; \quad k = 0 : N_3 - 1; \quad l = 0 : N_4 - 1$}
\State $\alpha_{\text{dim},(i,j,k,l)} \leftarrow \max_{p \in [\min D_{\text{dim}}, \max D_{\text{dim}}]} \left| \frac{\partial H}{\partial p_{\text{dim}}} \right|$
\State $H_{i,j,k,l} \leftarrow H_{i,j,k,l} - \sum_{\text{dim}=1}^4 \alpha_{\text{dim},(i,j,k,l)} D_{\text{dim}}^+ V_i,j,k,l - D_{\text{dim}}^- V_i,j,k,l$\
\State $\alpha_{\text{dim}} \leftarrow \max(\alpha_{\text{dim},0}, \alpha_{\text{dim},(i,j,k,l)})$
\EndFor
\State //Compute stable integration time step
\State $\Delta t \leftarrow (\sum_{d=1}^{4} \frac{\Delta z_d}{\alpha_{\text{dim}}})^{-1}$
\State $V_{t+1} \leftarrow H\Delta t + V_t$
\State $V_{t+1} \leftarrow \min(V_t, V_{t+1})$
\State $\epsilon \leftarrow |V_{t+1} - V_t|$
\If{$\epsilon < \text{threshold}$}
\State $V_t \leftarrow V_{t+1}$
\State Go to line 3
\EndIf
\end{algorithmic}
\end{algorithm}
Let $N_{\text{dim}}$ be the number of grid points of the $\text{dim}^{\text{th}}$ dimension of the grid ($0 \leq \text{dim} \leq 3$), and let $V_{i,j,k,l}$ be the value function at some state on a 4D grid. We will adopt the central differencing scheme for approximating partial derivatives for every direction. Equation (6) demonstrates the central differencing scheme for $\text{dim} = 0$, where $\Delta z_{\text{dim}} = \text{Range}_{\text{dim}}/N_{\text{dim}}$ and $\text{Range}_{\text{dim}}$ is the grid range of state $z_{\text{dim}}$. The two terms $D_{\text{dim}}^{-}$ and $D_{\text{dim}}^{+}$ are the left and right approximations derivatives respectively.

$$D_{\text{dim}}^{-} = \frac{V_{i+1,j,k,l} - V_{i-1,j,k,l}}{\Delta z_{\text{dim}}},$$

$$D_{\text{dim}}^{+} = \frac{V_{i+1,j,k,l} - V_{i-1,j,k,l}}{\Delta z_{\text{dim}}},$$

$$p_{\text{dim}} = D_{\text{dim}} = 0V_{i,j,k,l} = \frac{D_{\text{dim}}^{+}V_{i,j,k,l} + D_{\text{dim}}^{-}V_{i,j,k,l}}{2} \tag{6}$$

When $V_{i,j,k,l}$ is at the grid boundary (ie. $\text{dim} = 0$, and $i = N_0 - 1$, $i = 0$), we follow the popular toolbox [11] to extrapolate the left and right value grid point as follows:

$$V_{i-1,j,k,l} = V_{i,j,k,l} + |V_{i,j,k,l} - V_{i+1,j,k,l}| \ast \text{sign}(V_{i,j,k,l}) \tag{7}$$

$$V_{i+1,j,k,l} = V_{i,j,k,l} + |V_{i,j,k,l} - V_{i-1,j,k,l}| \ast \text{sign}(V_{i,j,k,l}) \tag{8}$$

The basic algorithm for solving (4) on a grid for 4D systems is described in Algorithm 1. The algorithm loops through the 4D array three times. In the first grid iteration (lines 3-9), the Hamiltonian term $H(z, \nabla V(z, s))$ is computed while maximum and minimum values of the partial derivative components are tracked in order to compute the dissipation term. The dissipation term and $\Delta t$ have to be determined via the Courant-Friedrichs-Lewy condition [15] in order for the $V$ solution to be stable. In the next grid iteration (lines 12-15), the dissipation term is computed and added to the Hamiltonian. In the third grid iteration (line 19), each grid point is integrated with an interval of $\Delta t$. At the same time, the maximum value of $\alpha_{\text{dim}}$ in every dimension defined in line 13 is computed. These $\alpha_{\text{dim}}^{\text{max}}$ variables are used to determine the step bound $\Delta t$.

C. Field Programmable Gate Arrays (FPGA)

FPGAs are configurable intergrated circuits programmed for specific applications using a hardware description language (HDL). Computing platforms such as CPUs, GPUs, and FPGAs consist of a memory component and computing cores. 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.

Efficient systems need to have both fast computing cores and fast data distribution from the memory. Depending on the application, the memory access and computing pattern will vary. General-purpose CPU/GPU are often architected towards a reasonable performance for a wide variety of applications, but not optimized for any particular application. On the other hand, FPGAs 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.

D. Problem Description

A key observation of Algorithm 1 is that new value of $V$ at each grid point can be computed independently from each other within a grid iteration and therefore, in parallel. We can then leverage a high degree of parallelism on an FPGA by having many processing elements (PE) update as many grid points concurrently as possible. However, before that, two challenges must be addressed. Firstly, an FPGA’s on-chip memory needs to efficiently distribute data to the PEs in order to execute Algorithm 1, each PE needs a nine-point stencil input (shown in Fig. 2). Secondly, a 4D grid takes up tens of megabytes in memory and therefore cannot fully load onto an FPGA’s on-chip memory for fast access compared to Dynamic Random Access Memory (DRAM), which has a higher memory capacity but significantly clock cycles to access.

In this paper, our goal is threefold. First, we introduce an alternative algorithm to Algorithm 1 that is more computationally efficient. Second, we propose a hardware design that can solve the above challenges and maximally parallelize the algorithm while making efficient use of an FPGA’s on-chip memory. Finally, through real world robotic experiments, we demonstrate that this enables real-time BRT computations on an FPGA which allows autonomous systems to react to abruptly changing environments.

III. 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:

$$\dot{x} = v \cos(\theta) \quad \dot{y} = v \sin(\theta)$$

$$\dot{v} = a \quad \dot{\theta} = \frac{v \tan(\delta)}{L} \tag{9}$$
where \((x, y)\) represents the positions, \(v\) represents the speed, and \(\theta\) 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: 

\[
\dot{x} = \frac{\partial V}{\partial x} \quad \dot{y} = \frac{\partial V}{\partial y} \quad \dot{\theta} = \frac{\partial V}{\partial \theta}
\]

In this case, the term 

\[
\frac{\partial H}{\partial \text{dim}}
\]

is used as scaling factor (line 14 - Algorithm 1), is not dependent on \(\text{dim}\) and equal to the absolute value of the rate of change of state component \(\text{dim}\). 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 1) into the first one. This change is reflected in line 7 of Algorithm 2. 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 \(\alpha_{\text{dim}}\) 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 \(\Delta t\) every time and then iterating through the 4D grid array again, we pre-compute \(\Delta t\) and re-use it for all of the time iterations. Combining these ideas together, throughout this paper, we will use Algorithm 2 with one grid iteration for our FPGA implementation, which is more computationally efficient. We empirically evaluate the correctness of this Algorithm 2 in section V.

**Algorithm 2** Optimized 4D Grid Value Function Solving Procedure

1. Initialize \(V_0[N_1][N_2][N_3][N_4]\)
2. for \(i = 0 : N_1 - 1; \quad j = 0 : N_2 - 1; \quad k = 0 : N_3 - 1; \quad l = 0 : N_4 - 1\) do
3. Compute (6) for \(1 \leq \text{dim} \leq 4\)
4. \(u_{\text{opt}} \leftarrow \arg\max_u \nabla V(z, s)^T f(z, u)\)
5. \(\hat{z} \leftarrow f(z, u_{\text{opt}})\)
6. \(H_{i,j,k,l} \leftarrow \nabla V(z, s)^T \hat{z}\)
7. \(H_{i,j,k,l} \leftarrow H_{i,j,k,l} - \sum_{\text{dim}=1}^{\text{dim}} D_{\text{dim}} V_{i,j,k,l} - D_{\text{dim}} V_{i,j,k,l}\)
8. \(V_{i+1}(i,j,k,l) \leftarrow H_{i,j,k,l} + \Delta t_{\text{precomputed}} + \frac{2}{\epsilon} V_{i+1}(i,j,k,l)\)
9. \(V_{i+1}(i,j,k,l) \leftarrow \min(V_{i+1}(i,j,k,l), V_{i+1}(i,j,k,l))\)
10. end for
11. \(\epsilon \leftarrow |V_{i+1} - V_i|\)
12. if \(\epsilon < \text{threshold}\) then
13. \(V_i \leftarrow V_{i+1}\)
14. Go to line 2
15. end if
IV. FPGA DESIGN

In this section, our goal is to build a custom hardware design on an FPGA that solves Algorithm 2 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). 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 2 (lines 3-10). In the following subsections, we will go into the details of each component.

A. Minimizing On-Chip Memory Buffer

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 [16] 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:

\[
2N_1N_2N_3 + pe + 1
\]

where \( pe \) is the number of PEs used in the design. The expression in (10) 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. 5) is adapted from the parallel memory buffer microarchitecture described in [16], which is specialized for a 2D/3D grid. We extend [16] 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. 5) depending on which grid points need to be accessed by the PEs. The endpoints of these FIFO queues (shown as colored arrows in Fig. 5) 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. 5) 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. 5). As the values move along the FIFO queues, they are accessed by the PEs. To perform the computations in line 3 of Algorithm 2, the PEs need access to the adjacent grid points values in all 4 dimensions as illustrated by Fig. 2. The indices of the grid value needed by the PEs determine the sizes of each FIFO queue, which are shown in Fig. 5 for the first line. Note each line buffer supplies values to multiple PEs, as shown by the coloured arrows.

B. Indexed Processing Element (PE)

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.

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 2 (lines 3-10) for a particular index \( i, j, k, l \). Following the sequential order of Algorithm 2, 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 2 for multiple indices concurrently (explained in Fig.4).

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 2). 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.

In our design, we use 4 PEs (shown in Fig. 3). 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 \( \text{idx} \), \( 0 \leq \text{idx} \leq 3 \) associated with it. At the start of Algorithm 2, each PE takes as input a grid index \((i,j,k,l + \text{idx})\) and its eight neighbours from the on-chip memory buffer to start computing \( V_{i+1}(i,j,k,l + \text{idx}) \) according to Algorithm 2 (line 2-10).

C. Fixed-Point Representation

Computing a new value function based on Algorithm 2 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.

V. EXPERIMENT & RESULTS

In this section, we first evaluate the correctness and show the performance of our hardware design on an FPGA that can re-solve (4) in a changing environment at an update frequency of 4Hz consistently. Next, we will show a real world experiment demonstrating that re-solving (4) at this rate can be used to ensure safety of a robot car.

A. Hardware Correctness

To evaluate the correctness of the computation, we considered three different environments inside a room, each with different cone placements (shown in Fig. 6). Inside the environment, orange cones are used as obstacles. The corresponding implicit surface function is as follows:

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

(11)

where \( x_o \) and \( y_o \) are the obstacle’s positions, and \( R \) is the radius of the cone.

We use fixed-point data representations for hardware computation. 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 \(4m \times 6m\), 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) and the choice of \( V_0 \) in (11). We choose to synthesize and implement our design on AWS F1 FPGA because of its flexibility and availability. To correctly input data to the FPGA, the initial value array based on (11) is generated and then converted from floating-point to fixed-point based on the bit choice discussed above.
Afterward, the value array is passed to the FPGA for the HJ variational inequality solving procedure to start. For all three environments, the correctness of the value function of the grid points generated by our hardware is verified against the toolbox at [17] by comparing the maximum absolute error among corresponding grid points. The toolbox uses 32-bit floating-point numbers. The numerical error resulting from the different representations is shown for each three environments in Table I. The errors are negligible due to precision difference between fixed-point and floating point number. 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.

B. Computational Speed and Resources Usage

To measure the speed up for all three environments, we compare the computation time on an AWS FPGA running at 196MHz against the toolboxes [17] and [11] running on a 16-thread Intel(R) Core(TM) i9-9900K CPU at 3.60GHz. The results are summarized in Tables II and IV. 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>
<thead>
<tr>
<th>Env.</th>
<th>Clock cycles</th>
<th>Period</th>
<th>Iterations</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Env. 1</td>
<td>4,796,5066</td>
<td>5.1 ns</td>
<td>67</td>
<td>0.2447s</td>
</tr>
<tr>
<td>Env. 2</td>
<td>4,796,4977</td>
<td>5.1 ns</td>
<td>67</td>
<td>0.2447s</td>
</tr>
<tr>
<td>Env. 3</td>
<td>4,796,5021</td>
<td>5.1 ns</td>
<td>67</td>
<td>0.2447s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Env.</th>
<th>Latency</th>
<th>Iterations</th>
<th>FGPA speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Env. 1</td>
<td>3.35 s</td>
<td>67</td>
<td>×13.7</td>
</tr>
<tr>
<td>Env. 2</td>
<td>2.99 s</td>
<td>67</td>
<td>×12.2</td>
</tr>
<tr>
<td>Env. 3</td>
<td>3.42 s</td>
<td>67</td>
<td>×14</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Env.</th>
<th>Latency</th>
<th>Iterations</th>
<th>FGPA speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Env. 1</td>
<td>25.11 s</td>
<td>70</td>
<td>×103</td>
</tr>
<tr>
<td>Env. 2</td>
<td>25.14 s</td>
<td>70</td>
<td>×103</td>
</tr>
<tr>
<td>Env. 3</td>
<td>25.18 s</td>
<td>70</td>
<td>×103</td>
</tr>
</tbody>
</table>

It can be observed that the latency of computation on FPGA is more fixed and consistent for all three environments while the latency on CPUs varies even though the computation remains the same. Note that the slight difference of number of clock cycles in three environment is due to communication latency between DRAM and FPGA, which can vary. With the worst latency of 0.25s, we are able to update the value function at a frequency of 4Hz.

C. Experiment Setup

We used a Tamiya TT02 model RC car [18] controlled by an on-board Nvidia Jetson Nano microcontroller inside a 4m × 6m room for our experiment. The dynamics defined in (9) is used to model the car.

For our car, the control input range and car length are as follows: \( \alpha \in [-1.5, 1.5] \), \( \delta \in \left[ -\frac{\pi}{12}, \frac{\pi}{12} \right] \), and \( L = 0.3\text{m} \). We use a grid size of \( 60 \times 60 \times 20 \times 36 \) with resolutions of 0.1m, 0.067m, 0.2m/s and 0.17 rad for \( x \)-position, \( y \)-position, speed and angle, respectively. The computation ranges for \( x, y, v, \) and \( \theta \) are \( [-3.0, 3.0], [-1.0, 4.0], [0.0, 4.0], [-\pi, \pi] \) respectively.

For the experiment, we have two obstacles where one of them is constantly moved by a person and the other is static. Each obstacle is modeled as a cylinder shape with \( R = 0.08\text{m} \). A motion capture system is deployed to accurately track the car’s state and the position of the obstacles. These positions are then sent over to the FPGA in order to initialize the implicit surface function defined in (11) and computation can then start.
We note that the optimal control expressed in (5) implies that the RC car system is able to continuously change its control input. However, in practice, our RC car is a sampled data system, and can only change its control input at every time sample. Several previous works have investigated sampled data reachability analysis [19], [20]. Since the focus of this paper is on demonstrating the effectiveness of real-time BRT computations on an FPGA, we instead ignore the effect of sampled data, and compensate for modeling error and account for the size of the car by enlarging the effective radius of the obstacles to 0.55m. This amount of obstacle augmentation was experimentally determined to be sufficient.

In each setting, while the car is manually controlled by a user who tries to steer the car into the obstacles, the $V$ function is constantly evaluated by the FPGA to determine how safe the car’s current state is. When the $V$ value is less than some threshold value defined as, $V(x,y,v,\theta) < 0.5$, meaning that the car is close to the zero-level set, optimal control generated by (5) will then take over the manual control to prevent the car from entering the BRT.

A video of these experiments can be found at https://www.youtube.com/watch?v=8q6cJHmHNS8

VI. CONCLUSION

This paper introduces a novel customized hardware design on an FPGA that allows HJ reachability analysis to be computed 14 times faster than the newest state-of-the-art implementation on a 16-thread CPU and 103 times faster than the widely used MATLAB toolbox [11]. Because of this, we are able to solve the HJ variational inequality at a frequency of 4Hz. The latency of our computation on an FPGA is deterministic for all computation iterations, which is crucial for safety-critical systems. Our design approach presented here can be applied to system dynamics and potentially higher dimensional systems. Finally, we demonstrate that at 4Hz, a robot car can safely avoid obstacles and guarantee safety. We believe that customized hardware accelerators could complement well with decomposition methods in making higher dimensional systems provably safe in real-time. The exploration of this is left as future work.

REFERENCES


