2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

Hardware-Software Co-Design for Path Planning by Drones

Ayushi Dube[§], Omkar Patil[§], Gian Singh, Nakul Gopalan, and Sarma Vrudhula

***Abstract*— This work consists of two main components: designing a hardware-software co-design, MT$^+$, for adapting the Mikami-Tabuchi algorithm for on-board path planning by drones in a 3D environment; and development of a specialized custom hardware accelerator *CDU*, as a part of MT$^+$, for parallel collision detection. Collision detection is a performance bottleneck in path planning. MT$^+$ reduces the delay in path planning without using any *heuristic*. A comparative analysis between the state-of-the-art path planning algorithm A* and Mikami-Tabuchi is performed to show that Mikami-Tabuchi is faster than A* in typical real-world environments. In custom-generated environments, path planning using Mikami-Tabuchi shows a latency improvement of $1.7\times$ across varying average sizes of obstacles and $2.7\times$ across varying obstacle density over state-of-the-art path planning algorithm, A*. Further, the experiments show that the co-design achieves speedups over a full software implementation on CPU, averaging between 10% to 60% across different densities and sizes of obstacles. *CDU* area and power overheads are negligible against a conventional single-core processor.**

Fig. 1: ***The neighborhood environment for drone path planning simulated with AirGen, a drone simulator. The other simulated environments for evaluating the algorithms are based on grassland and industrial domains.***

I. INTRODUCTION

Existing path planners for drones are mostly offline and use planning algorithms that are inherently sequential [15]. Therefore, these planners witness a huge path computation delay (T_p) as well as a communication delay (T_c) between the planner and the drone. Since the demand for autonomous drones is increasing at a rapid rate, there is a need for fast path planners [4]. One solution to reduce the T_c is to design on-board planners. On-board planners sit on the drone and do not communicate with an external controller. However, several existing path planners still employ algorithms that are sequential in nature and thus, do not improve the T_p [2].

In this paper, a hardware-software co-design, MT$^+$, is proposed for an on-board path planner that implements the *Mikami-Tabuchi* (MT) [20] algorithm. MT is a *Line Search Algorithm (LSA)* which is highly parallelizable in nature without using any *heuristic*. The aim of this design is, (a) to reduce the T_p by using MT for the first time in drone path planning and (b) to reduce the T_c by implementing the planner on-board. MT$^+$ further improves on T_p by implementing a *Collision Detection Unit* (CDU) in hardware. *CDU* parallelizes collision detection function used in MT$^+$ for faster execution.

MT$^+$ is ideally suited for *local path planning*. In local path planning, the drone only has information about the environment around it (*observable environment (OE)*) instead of a full map [16]. Drones either perceive information about the OE through sensors like LIDARs, cameras, ultrasonic sensors, etc. or receive the information from an off-board GPS or controller. The next step involves path planning which guides the drone from the present location to the goal location within the OE. The path traversed within the OE is called a *local path*. A series of local path plannings leads to *global path planning*, where the global path is an unobstructed path between the source and the final goal location as illustrated in Fig. 2. The local environment is updated at $t = t_1$ and $t = t_2$ time stamps. These local source and goal locations are updated periodically along with the OE as the drone moves towards the global goal.

In summary, our work has the following contributions:

- To the best of our knowledge, this is the first work to adapt the Mikami-Tabuchi (MT) algorithm for on-board path planning. Moreover, the use of MT is demonstrated in realistic 3D drone path planning problems.
- A hardware-software co-design MT$^+$ is proposed to accelerate MT. The collision detection operation in MT performed by the CPU is three orders of magnitude slower than the *Collision Detection Unit* (CDU). CDU is a proposed hardware accelerator in the co-design MT$^+$. The collision detection function constitutes up to 30% of the total operations executed on MT-based planners. An average performance improvement of MT$^+$ over MT implemented on the CPU ranges from 10% to 60% for different sizes of the OE and obstacle densities.
- A detailed comparison of the MT against the A* algorithm [15] in custom environments with various obstacle densities and sizes is carried out on simulated grids and

School of Computing and Augmented Intelligence, Arizona State University, Tempe, Arizona, United States. Contact: {adube9, opatil3, gsingh58, ng, vrudhula}@asu.edu
[§]Equal contribution

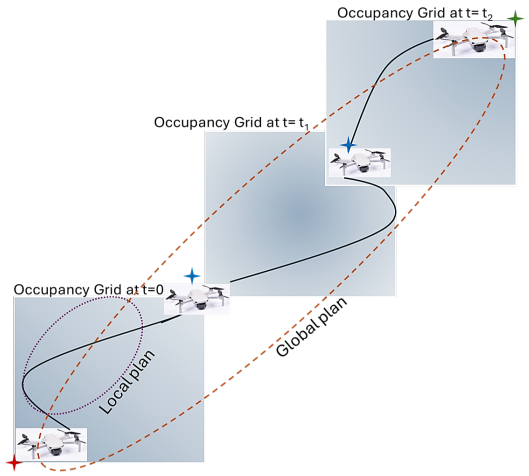

Fig. 2: ***Local path planning versus global path planning. The red and green markers are global source and goal and the blue markers are local source or goal as per the time stamp (t_1 and t_2).***

on a realistic drone simulator to establish the use case of the proposed approach.

- The CPU implementation of MT performs better than A* by an average of 2× better in terms of planning latency for real-world environments such as neighborhoods and factories.

II. Setup

Definition 2.1 (Occupancy grid (O)): produced by a drone perception unit or received by the drone consists of 3D matrix representing the physical OE around the drone. This is a standard representation that consists of cells with values 0 or 1 where the former represents *empty* and the latter represents *blocked* cells. O doesn't change during the planning stage. The *forward kinematics (FK)* of path planning involves finding a collection of connected empty cells and connecting the source and goal locations.

Definition 2.2 (Path): A set of contiguous empty cells between the source and the goal. Here, a path is a collection of continuous empty cell addresses, also called *waypoints* connecting the source and the goal cells.

Definition 2.3 (Optimal path): The path with minimum length OR with minimum bends OR computed in minimum time. Optimality is dictated by the application and the constraints imposed by the drone's capabilities.

Definition 2.4 (Cost): Cost of finding a path is the total energy (or time) expended to compute the path or the path length of the found path.

Problem Statement: A 3D occupancy grid O(N×N×N) along with a source location (x_s, y_s, z_s) and a goal location (x_g, y_g, z_g) (addresses of a certain pair of cells in the grid) are given. The source and goal cells are contained in the grid i.e., x_s, y_s, z_s, x_g, y_g, z_g < N. A path has to be computed connecting the source and the goal in minimum time.

III. Background

A. Path-Planning Algorithms

There are various existing classes of approaches to path planning which are discussed in this section. Since we are working with an occupancy grid O as the spatial representation of the environment, the discussed algorithms are broadly classified as *search-based* and *sample-based* algorithms [1]. Sample-based methods *sample* the environment uniformly and randomly to find the path and therefore, does not produce the shortest path. Examples of sample-based methods are Probabilistic RoadMap (PRM) and Rapidly-Exploring Random Trees (RRT) [1]. Search-based algorithms like A* and D* search in a particular direction of motion trying to optimize the cost of the path which is based on a *heuristic*. These methods guarantee an optimal path [1]. We are using the MT algorithm which is a special class of search-based algorithms called *Line Search Algorithms*. These methods do not guarantee an optimal path (or shortest path). However, they offer using the current massively parallel hardware accelerators to definitely find a path if one exists.

B. Line Search Algorithms (LSAs)

As the name indicates, LSAs search along defined directions. LSAs are an example of depth-first search (DFS) algorithms [11]. These are designed for a grid-based environment and have been previously employed for channel routing in Integrated Chips (ICs) which are 2D planes [11]. Examples of LSAs are Mikami-Tabuchi (MT) and Hightower [19].

MT algorithm relies on generating orthogonal lines from source (S) and goal (G) locations and then checking orthogonal pairs of lines from source and goal for an intersection. These are called *Level0* lines. A line is a collection of successive empty cells in the defined direction. If an intersection isn't found, all possible orthogonal lines to each of the *Level0* lines are generated. These are called *Level1* lines. All the lines generated at any level are stored to the source or goal set of lines. *Level0* and *Level1* lines are shown in Fig. 3.

All possible combinations of orthogonal sets of lines from the source and goal sets are checked for intersection before the next level of lines are generated. This procedure is repeated to generate *Level2, Level3,...* lines and finding intersections. Once an intersection is found, backtracking through the lines explored leads to the whole path. In Fig. 3, lines from S and G intersect at multiple locations to generate multiple paths. One of them is $(0,0)$, $(0,1)$, $(0,2)$, $(1,2)$, $(2,2)$, $(3,2)$,$(3,3)$. An alternate path is $(0,0)$, $(1,0)$, $(2,0)$, $(2,1)$, $(2,2)$, $(2,3)$, $(3,3)$. There are other possible paths as well. MT algorithm guarantees a solution if it exists [11]. Also, the algorithm can be easily parallelized as opposed to A* which is inherently sequential [2]. This infers that careful design and architectural choices can significantly speed up the algorithm. The Hightower algorithm is similar to MT and uses less memory compared to MT. However, Hightower doesn't guarantee a solution [11].

Memory size is not as great a challenge as it was a couple of decades ago when these algorithms were designed.

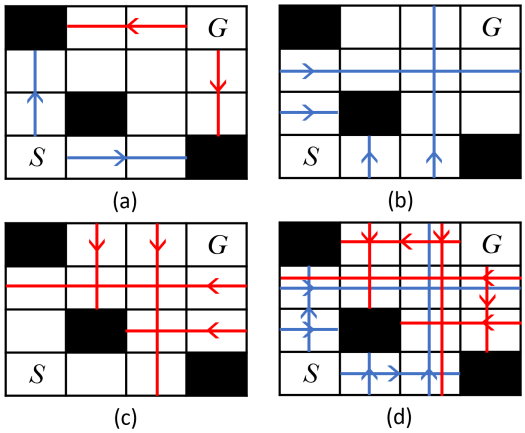

Fig. 3: ***A 2D Occupancy grid*** $(4X4)$ ***with free (empty) and blocked (marked with 'X') cells: (a) shows Level0 lines from S and G (blue and red respectively); (b) shows Level1 lines from the source set; (c) shows Level1 lines from the goal set; (d) shows Level0 and Level1 lines from source and goal sets together; where*** $\boldsymbol{S}(0,0)$ ***and*** $\boldsymbol{G}(3,3)$***.***

However, memory still is a bottleneck to the performance of a design [17]. LSAs use lesser memory compared to A* and RRT where cost functions are computed for every node and its neighbors to keep track of the cost values and the connectivity information of randomly generated neighbors respectively. These values increase dramatically with larger grid sizes. Therefore, the memory requirements and delay also increase. Other variants of A* have been formulated to tackle the issue but the problem persists [15]. A notable drawback with LSAs is that all paths generated are rectilinear. It suggests that diagonal cells to the current location are not explored at time t. However, the diagonal cell may be a part of the path through a set of two or more rectilinear translations at time $t+T$ (later instant). The idea is using maximum available parallel resources to reduce T_p.

IV. Proposed Approach/Methodology

We implement a hardware-software co-design MT$^+$ which adapts the Mikami-Tabuchi algorithm for an on-board drone path planner. To reduce the latency of the planner, parallelism is used in the collision detection function by implementing a hardware accelerator *CDU*, described in the next section. Before moving towards the algorithm, it is important to define the terminology used in the explanation of the code.

1) Point $P(x,y,z)$*:* A point here is an address of a cell in the 3D occupancy grid O.

2) Array: A x or y or z array is a complete 1D row or column in the corresponding direction in O. Therefore, the length of an array is N in a O (N×N×N). As shown in Fig. 5(a), x array 1, 2,..., 7 are seven arrays in a 2D grid. For a 3D grid O with $N=7$, total number of x arrays are $N \times N = 49$. Same number of y and z arrays also exist.

3) Line: A x or y or z line is the collection of empty cells in the corresponding direction, containing the input point P. As shown in Fig. 5(b), $x=0$ line is derived from x array 1

Algorithm 1: Algorithm for 3D path planning using Mikami-Tabuchi. The notation applies symmetrically to Y and Z.

Input: Start point S, Occupancy grid O around S, Goal point G

Output: List of waypoints W

Initialize : Struct $LineX$ that stores:

- Origin point P {Spawn point}
- $xStart$, $xEnd$ {Contiguous line along x direction}
- $parentLineY$, $parentLineZ$ {For backtrackking}

Initialize : Separate structs $LineSetX$ for S and G:

- $N \times N \times N$ $pointerArray$ {Pointer array to $LineX$}
- $N \times N \times N$ $included$ {True if $LineX$ exists for point}
- $size$, $prevSize$ {Number of lines in this and previous level}

Level 0 :

1: Create $LineX$'s for point S and G and add to $LineSetX$
2: Check for intersections b/w LineSets from S and G

Level i:

3: **for** $i=1$ to Inf **do**
4: Check for intersections b/w LineSets from S and G
5: **if** Intersection I found **then**
6: Backtrack from I to S and G
7: Record spawn points for each $LineX$ in W
8: **end if**
9: Create new $LineX$'s $\forall$ points on $LineY$ and $LineZ$ generated in the previous level, add to $LineSetX$
10: **end for**
11: **return** W

with limits as $x_1=0$ and $x_2=4$. Using this information, an efficient representation of the line is presented in the following subsection.

V. Hardware Acceleration

In this section, details of MT$^+$ are discussed with emphasis on the hardware design of the accelerator for collision detection *CDU*. MT$^+$ comprises of a tightly coupled CPU and *CDU* with an assumed communication latency of 1 clock cycle. The clock frequency of MT$^+$ is 1 GHz.

A. CDU Operations

CDU essentially generates lines which is part of the Create *LineX* in step 1 and step 9. It refers to inputting a point $P(x,y,z)$ and generating orthogonal lines through P in the desired direction (x or y or z). The format used to represent a y direction line is (x,z,y_1,y_2) where x and z are the co-ordinates of the line and y_1 and y_2 are the limits/ends of the line in the y direction. In other words, all the cells/points between y_1 and y_2 are empty and the points (x,y_1-1,z) and (x,y_2+1,z) are the locations of the first obstacle encountered while going from P in the negative and positive y direction respectively. The P is located in the line. Similar computation is done for x and z lines which are represented as (y,z,x_1,x_2) and (x,y,z_1,z_2) respectively.

Create *LineX* aims to represent a line in the above-explained format. This simplifies the procedure to find an

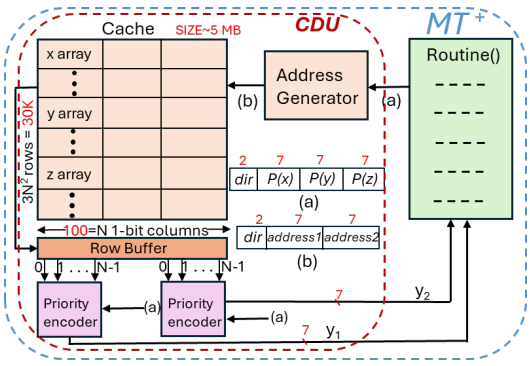

Fig. 4: ***Hardware Software co-design MT$^+$: CDU as the hardware accelerator for collision detection and the routine is a software program running on a single core host CPU which is tightly coupled with CDU on-board the drone.***

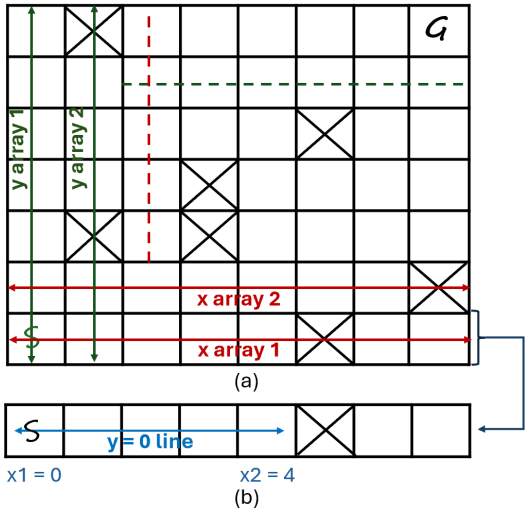

Fig. 5: ***(a) Definition of an array; (b) Definition of a*** $x = 0$ ***line with limits*** $x_1 = 0$ ***and*** $x_2 = 4$

intersection between two lines in step 5. For example, to check if the lines (y, z, x_1, x_2) and (x', y', z_1, z_2) intersect, only two comparison operations are performed which are: $x_1 \leq x' \leq x_2$ OR $z_1 \leq z \leq z_2$ and $y == y'$. As evident, collision detection is ingrained in the format as it finds the limits on the given line.

As the algorithm in the previous section suggests as the level i increases, the number of lines generated and added to the source and goal set increases exponentially. These are orthogonal lines to each line of the set which will be further checked for intersection in the next level. These lines can be independently checked which is an O(1) complexity step, explained by the example just discussed. This step is carried out by the software routine on the host CPU. However, for a grid size of dimension N, the total number of such intersection operations can be huge for increasing levels. Instead, we focus on hardware acceleration of creating a given line along any direction of O. This requires a constant time checking for all N points in the given array. Hence, a hardware accelerator *CDU* is designed for this operation. *CDU* produces a line in the required format in one cycle.

B. CDU Architecture

The hardware architecture of *CDU* is shown in Fig. 4. It consists of an address generator, a cache, a row buffer, and two priority encoders. They are described as follows:

1) Cache: A cache memory is used to store the occupancy grid information. The cache size is $NX3N^2$ bits or 480 KB for N=100. It stores all arrays in x, y, and z directions of O in successive rows as shown in Fig. 4.

2) Address Generator: The input to the address generator comes from the routine running on the host CPU. The inputs are a point location $P(x, y, z)$ and a signal called *dir* signifying the direction in which the line through P is to be fetched from the cache. Address to the cache is generated from the address generator in the format shown in Fig. 4(a). It consists of 2 bits to signify direction: 00 for x, 01 for y, and 10 for z direction. The next two fields, address1, and address2 are y, z co-ordinate locations for a x direction array; x, y co-ordinate locations for a z array, and x, z co-ordinate locations for a y array. These fields are $\log_2 N$ in length each. For N=100, address size is $2 + 2 * 7 = 16$ bits.

3) Row Buffer: It is a 1D register memory that holds the array $(x/y/z)$ fetched from the cache. Note that the row buffer size, the cache block size, and the dimension of O are all equal to N.

4) Priority Encoders: Two priority encoders are employed to find the limits or ends of a line i.e., y_1, y_2 for a y direction line. One encoder is connected to the row buffer in such a way that it produces y_1 value in the negative y direction starting from P and the other encoder generates y_2 in the positive y direction starting from P.

VI. Experimental Set-up

There are three sets of evaluations performed in this work. The first is to compare the Mikami-Tabuchi algorithm against the state-of-the-art A* algorithm in terms of performance for drone path planning for simulated grid/synthetic data. Second, is comparing the two algorithms in real-world environments using a drone simulator *AirGen*[12]. Third is evaluating the performance of the proposed co-design over a single-core CPU for the most suited application.

A. Mikami-Tabuchi versus A* on Simulated Grids

For the first set of experiments, the C code implementation of the two algorithms for a grid of size N×N×N is written and compiled using *gcc* compiler on the 12th Gen Intel(R) Core(TM) i9-12900KF CPU. The results are shown in Fig. 6. We evaluate the two algorithms for N=10 and N=100 across different obstacle densities (1%, 10%, 20%, 30% and 40%) and sizes (1×1, 3×3, 5×5, 7×7). The results demonstrate that MT has an average of 5× lower planning latency for lower obstacle densities, which is representative of real-life scenarios, shown in section VI-B. MT suffers from higher planning latencies at higher obstacle densities, prompting a hardware speed-up that helps to alleviate this issue.

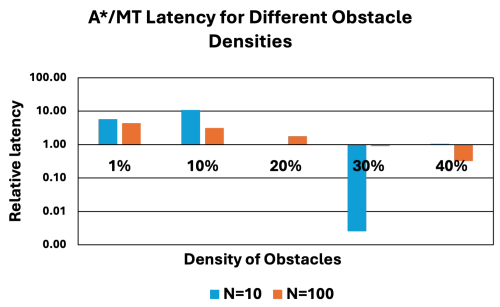

Fig. 6: ***Speedup (%) of Mikami-Tabuchi over A* across environment densities of 1% to 40% averaged over different obstacle sizes of 1×1, 3×3, 5×5, 7×7.***

	A* Latency (ms)			MT Latency (ms)		
Grid Discretization (meters) (N=100)	**2**	**3**	**5**	**2**	**3**	**5**
Cable Factory (D=2%)	5.51	3.8	1.74	2.77	1.67	1.38
Electric Central (D<1%)	4.23	3.63	3.63	1.47	1.35	1.27
Neighborhood (D=5%)	5.12	5.55	1.88	5.76	3.61	1.90

TABLE I: ***Latency of Mikami-Tabuchi and A* algorithms for real-world environments simulated in AirGen where D is the density of obstacles in the environments and N is the O dimension.***

B. Realistic Drone Simulation

Performance improvements are also demonstrated using a drone simulator *AirGen*, built on top of AirSim. Environments representative of common scenarios such as neighborhoods and factories are simulated and MT and A* are compared across random sets of source and goal locations. Table I shows the planning latency for 100×100×100 grid and discretization levels of 2, 3, and 5 meters in different simulated environments. Since a global path is not known beforehand, escape heuristics are designed to be similar for both the algorithms. We run the MT algorithm based on Algorithm 1 for two levels. It is evident that MT is faster than A* in almost all scenarios by an average of 2×. We also found the paths generated by the MT algorithm to be safer and more acceptable for neighborhood and factory scenarios. Videos of a drone following the paths generated by MT and A* algorithm in different environments can be found on the project website [1].

C. Co-design versus CPU Implementation

After MT is shown to perform better than A* for the environment type classified through the previous set of experiments, a hardware-software co-design MT$^+$ is implemented to efficiently run MT. As explained in section IV, the collision detection is offloaded to the hardware accelerator *CDU*. The architecture explained in the section V is described using Verilog and synthesized in TSMC's 45 nm ASIC flow using *genus* compiler. The clock frequency is 1 GHz. The cache memory used in *CDU* of size 480 KB and block size 16 KB is simulated using the open source cache memory simulator

[1]https://sites.google.com/asu.edu/drone-planning-mt/

TABLE II: ***Hardware details for CDU*** *($N = 100$)*

	Power (mW)	Area (mm^2)
Cache	288.49	1.281
Address generator and priority encoders	5.78	0.004
Total	294.27	1.285

cacti [18]. The area and power of *CDU* is 1.285 mm^2 and 294.273 mW. Against a full implementation on a single-core CPU, the area and power overheads of the co-design (*CDU* + single core CPU) are 5% and 2% respectively. The area and power of *CDU* is shown in Table II.

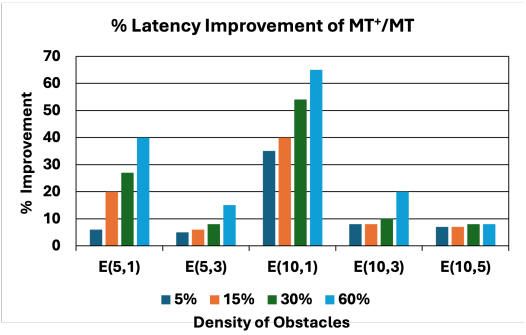

Fig. 7: ***Speedup (%) of the hardware-software co-design over a full CPU (Software) implementation. Here, E(N, S) stands for the environment with occupancy grid size N×N×N and S×S is the obstacle size.***

Fig. 7 shows a performance improvement of the proposed co-design over a full software implementation (CPU) with increasing obstacle density for an environment setting $E(N, S)$. $S \times S$ is the size of an obstacle in O. The reason is the increasing number of cycles expended by the CPU, checking for blocked cells. *CDU* checks for collision in an array in one cycle and hence, reduces combined cycles expended by the co-design (*CDU* + host CPU). Another observation is that as we move from a small grid ($N = 5$) to a large grid ($N = 10$), the speedup increases as the number of parallel checks for collision is directly proportional to N. Similar to the first reason, the combined cycles are reduced.

VII. Related Work

Existing works mostly use A* and RRT with their extensions which are heuristics designed as per the nature of data and the application [15]. One such work is [2], where the baseline A* is expanded with *RASExp's* extension in order to increase parallelism in the algorithm. As per the *RASExp*, a heuristic is followed that assumes that the direction of motion is constant for most of a drone's journey. Therefore, the next cell and its neighbors are explored for checking collision depending on the previous direction of motion. Collision detector accelerator units called *CODAcc* are used for collision detection in parallel. The shortcoming of this approach is that the level of parallelism depends on the heuristic and may not perform well in unstructured/complicated environments.

Another notable work is [3] where RRT is used along with a proposed memoization using Morton codes. A specialized hardware accelerator is proposed that accelerates memoization using a content-addressable memory (CAM). Many other co-design techniques aim to accelerate path planning with a main focus on the task of collision detection but they fail to address its sequential execution leading to an increase in latency, especially with increasing obstacle density.

Besides co-design techniques, FPGAs are also employed for the complex task of path planning [9]. The work presented in [8] uses a hybrid approach, combining hierarchical decision-making with combinatorial execution of multiple RRTs. A multi-port memory is designed that could store and prioritize over multiple RRT solutions running in parallel. Prioritization and the number of RRTs running are formulated mathematically, and subjected to constraints. Another class of FPGA-based path planning uses a radically different approach of Reinforcement learning [7], [6]. A state space of possible actions for a set of states is used to evaluate a reward for an action in a state. This concept is used to train a neural network on an FPGA. FPGAs are power-hungry devices that may not be energy-efficient for on-board path planning, especially with a resource-constraint drone. Therefore, they are used as offline path planners.

All of the above approaches are useful or optimized for different sets of constraints and inputs. The proposed work is different from the existing literature as it focuses on a highly parallelizable path-planning algorithm. There are no heuristics involved and the focus is to improve performance using minimal resources on-board the drone. Though a hardware-software co-design is proposed in the work, the market is flooded with specialized hardware to perform parallel computation faster and energy-efficiently. GPUs are also used for faster computation of the Mikami-Tabuchi algorithm [5]. MT is highly parallelizable and can be optimized at various design abstraction levels.

VIII. Limitations

Our work has several limitations arising from the nature of the algorithm MT. Firstly, the drone cannot move on diagonal lines as MT spawns straight lines from all the nodes at each step in search for a path. Secondly, the time required by the MT algorithm increases exponentially with the levels of search used in the algorithm. We choose a threshold of two for levels of search after observing sufficient success rates at fast computation times. We do want to point out that thresholding the levels of MT removes the assurance of finding a path to the goal, requiring its practical usage to setup interim goal points for finding an indirect path. Finally, the actual path followed by MT is sub-optimal to that of other algorithms such as A*. This is because the algorithm is not designed to return shortest paths.

IX. Conclusion

This work aims to accelerate on-board drone path planning as a mapless planner. A hardware-software co-design technique MT$^+$ is proposed that executes the Mikami-Tabuchi algorithm in parallel. The proposed co-design technique achieves significant speedups over CPU implementations of A* and Mikami-Tabuchi due to the following reasons: (a) it eliminates the communication between a drone and the off-board controller, (b) it exploits the parallelism in the Mikami-Tabuchi algorithm to increase the speed of path planning, and (c) the proposes hardware accelerator *CDU* computes the offloaded collision detection function from the host CPU in parallel. MT$^+$ shows significant improvement over MT and A* which are full CPU implementations.

X. Acknowledgement

This work was supported in part by the NSF I/UCRC for Intelligent, Distributed, Embedded Applications and Systems (IDEAS) and from NSF grant #2231620

References

[1] Gugan G, Haque A., Path Planning for Autonomous Drones: Challenges and Future Directions. Drones. 2023; 7(3):169.
[2] Mohammad B. et. al. RACOD: algorithm/hardware co-design for mobile robot path planning. Proceedings of the 49th ISCA. 2022.
[3] M. Luo and G. E. Suh, Accelerating Path Planning for Autonomous Driving with Hardware-Assisted Memorization. IEEE 33rd ASAP, 2022.
[4] Borghetti F. et. al., The Use of Drones for Last-Mile Delivery: A Numerical Case Study in Milan, Italy. Sustainability, 2022.
[5] Chiu Y. Chan et. al., GPU-based Line Probing Techniques for Mikami Routing Algorithm, https://api.semanticscholar.org/CorpusID:6343039, 2012.
[6] Tu G-T, Juang J-G. UAV Path Planning and Obstacle Avoidance Based on Reinforcement Learning in 3D Environments. Actuators. 2023.
[7] Y. Li et. al., A UAV Path Planning Method Based on Deep Reinforcement Learning, IEEE USNC-CNC-URSI North American Radio Science Meeting (Joint with AP-S Symposium), 2020.
[8] Malik, Gurshaant et. al., FPGA based hybrid architecture for parallelizing RRT. International Journal of Mechanical Engineering and Robotics Research. 2016.
[9] A. Bayrak and M. Ö. Efe, FPGA based offline 3D UAV local path planner using evolutionary algorithms for unknown environments, IECON 2016.
[10] Nasrollahi, S., Mirrezaei, S.M., Toward UAV-based communication: improving throughput by optimum trajectory and power allocation, J Wireless Com Network, 2022.
[11] S. Q. Zheng, J. S. Lim and S. S. Iyengar, Efficient maze-running and line-search algorithms for VLSI layout, Proceedings of Southeastcon '93, Charlotte, NC, USA, 1993.
[12] Vemprala S, Chen S, Shukla A, Narayanan D, Kapoor A. GRID: A Platform for General Robot Intelligence Development. arXiv. 2023
[13] Gupta A, Fernando X. Simultaneous Localization and Mapping (SLAM) and Data Fusion in Unmanned Aerial Vehicles: Recent Advances and Challenges. Drones. 2022.
[14] Azzam, R., Taha, T., Huang, S. et al. Feature-based visual simultaneous localization and mapping: a survey. SN Appl. Sci. 2, 224 (2020).
[15] Zammit, Christian & Van Kampen, Erik-Jan. Comparison between A* and RRT Algorithms for 3D UAV Path Planning. Unmanned Systems. 2021.
[16] S. Zhang and R. Zhang, Radio Map Based Path Planning for Cellular-Connected UAV," IEEE Global Communications Conference (GLOBECOM), 2019.
[17] Y. Yang, X. Chen and Y. Han, Dadu-CD: Fast and Efficient Processing-in-Memory Accelerator for Collision Detection, 57th DAC, 2020.
[18] Jouppi, N. P. et. al., CACTI-IO: With Off-chip Power Area-Timing Models, IEEE/ACM ICCAD, 2012.
[19] David W. Hightower. A solution to line-routing problems on the continuous plane, DAC, 1969.
[20] Mikami and Tabuchi, A computer program for optimal routing of printed circuit connectors, IFIP, H47, 1968.