Hardware/Software Co-Design for the Acceleration of RRT* Algorithm for Robotics Motion Planning

1st Samuel Akinola
Institute for Theoretical Electronics and Microelectronics
University of Bremen
Germany
akinolasamuel762@gmail.com

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—Motion planning is a critical component of robotics and autonomous vehicles, and the Rapidly Exploring Random Trees (RRT) Star algorithm, based on stochastic search strategies, is widely used for its ability to find optimal paths even in complex environments. However, RRT* can be computationally intensive, especially in environments with high-dimensional state spaces. Therefore, this paper introduces a hardware/software co-design approach to accelerate the RRT* motion planning algorithm on low-power embedded FPGA devices.**

The proposed solution includes the development of a hardware accelerator known as the "neighbourhood processor." This accelerator is designed to speed up essential computations such as nearest neighbor identification, collision detection, and neighboring node calculations. Notably, the approach is versatile and can be used for both RRT and RRT* algorithms, although the primary focus of this paper is on RRT*.

The architecture of the hardware accelerator is validated using a bare metal software approach, ensuring efficient utilization of hardware resources while retaining the adaptability and scalability inherent in software based methods. The hardware accelerator is developed using High Level Synthesis (HLS) and is seamlessly integrated with the RRT* algorithm through a custom software interface.

Remarkable performance improvements are achieved with the accelerator when it runs on a SoC Xilinx Zynq-7020. Specifically, it achieves a remarkable 60X runtime acceleration for the neighborhood algorithm (for 10,000 nodes) and a 40X improvement in the energy-delay product. In conclusion, this paper demonstrates that a hardware/software co-design approach can serve as a practical solution for accelerating motion planning algorithms on low-power embedded systems.

***Index Terms*—Motion planning, RRT-Star, FPGA, fixed-point, Embedded Linux**

I. Introduction

Motion planning in robotics is a crucial task, particularly in dealing with complex and high-dimensional environments. The Rapidly Exploring Random Trees (RRT) algorithm has emerged as a popular choice for this purpose. However, an optimized version of RRT, known as RRT*, designed to minimize solution cost, often struggles with computational intensity, especially when applied to large-scale problems. In response to this challenge, researchers have explored the use of hardware accelerators to enhance the algorithm's performance. Despite the promise of this approach, there remains a notable gap in research in this specific domain.

Field-Programmable Gate Arrays (FPGAs) hold substantial promise due to their ability to strike a balance between flexibility, performance, and energy efficiency. However, realizing this potential necessitates meticulous optimization in the design process [1].

As the landscape of chip design evolves, specialized hardware accelerators are increasingly favored over general-purpose processors. This preference is accentuated by growing power constraints, making the use of dedicated hardware an even more compelling choice [2].

A closer examination of the RRT* algorithm's performance profile, as illustrated in Figure 2, reveals a significant bottleneck. Approximately 85% of the total computational burden lies in the processes related to generating random nodes and computing the neighbors of specific nodes. Accelerating these critical aspects holds the promise of substantially improving the overall computational efficiency of the algorithm.

This paper introduces a novel approach to address these challenges—a hardware/software co-design strategy aimed at accelerating the RRT* motion planning algorithm, with a particular focus on low-power embedded FPGA devices. The core of our design methodology leverages High-Level Synthesis (HLS), an automated design technique that facilitates the transformation of high-level algorithm descriptions into highly optimized hardware implementations. Additionally, we adopt float16 and fixed-point (32-bit) number representations to minimize latency introduced by conditional if statements. These are thoughtfully replaced with signed shift implementations, as detailed in Section IV.

To validate the effectiveness of our implementation, we subject it to testing using bare metal software on a Xilinx Zynq-7020 SoC (Zybo-Z7 development board). The outcomes of this validation effort are presented in the subsequent sections of this paper.

Our main contributions are:

- We present a dedicated hardware that accelerates the computation of the nearest neighbour, neighbourhood, and obstacle detection.
- We developed a hardware/software co-design framework targeting low-power and resource constrained embedded applications.

- We have implemented two variations of the proposed architecture: one utilizing IEEE 754 float16 representation and the other employing 32-bit fixed-point number representation.

II. RELATED WORK

The primary goal of motion planning in robotics is to discover a safe and obstacle-free path for a robot to traverse from its initial position to a designated target position [4]. This objective remains consistent, although the environmental conditions may vary substantially, as depicted in Figure 3, Figure 4, or even complex mazelike settings as shown in Figure 5. Achieving efficient and effective motion planning in high-dimensional spaces is crucial for successful robot navigation in diverse and challenging surroundings.

To address the computational complexities involved in high-dimensional motion planning, researchers have explored various strategies to accelerate the RRT* algorithm through a combination of hardware and software design.

One approach, exemplified in [6], entails the integration of both combinatorial and hierarchical architectures within a hybrid framework. To determine the optimal distribution of RRT* nodes between these architecture blocks, researchers propose a cost function that takes into account speed-up and power parameters, which are inversely related. By maximizing this cost function while adhering to specific constraints, they utilize a modified branch and bound technique to achieve the best allocation of RRT* modules across both architecture blocks. The drawback to this approach is that it is limited to RRT algorithm and the power consumption was as high as 34W which is restrictive for embedded systems applications.

Joshua B. and colleagues have pursued a distinct avenue by harnessing the power of Graphics Processing Units (IPs) to accelerate RRT* algorithm computations [7]. IPs are renowned for their parallel processing capabilities, which can significantly enhance the algorithm's performance. However, elevated power consumption and size of GPUs representing restrictions for embedded systems application

In another innovative approach detailed in [8], S. Xiao and his team have implemented the RRT* algorithm on an FPGA with a unique refinement step. This refinement step operates in parallel with the ongoing tree-extension process, thereby enhancing the overall efficiency of the RRT* algorithm. This showcases the versatility of hardware acceleration methods, such as FPGAs, in enhancing motion planning within high-dimensional spaces. However, their computation was only limited to 2048 nodes which is a considerable restriction in practical applications. Moreover, this work does not specify the number format which can impact the quality of the result.

These diverse approaches exemplify the ongoing efforts to improve motion planning algorithms, making them more efficient and effective for a wide range of robotic applications.

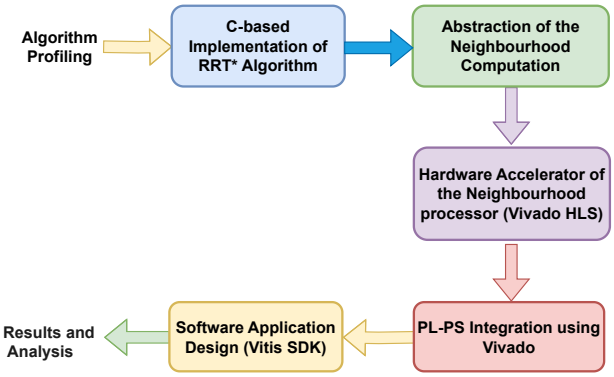

Fig. 1. Implementation workflow.

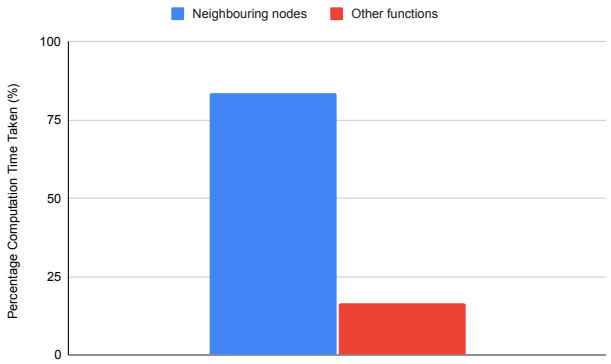

Fig. 2. Profiling of RRT* algorithm.

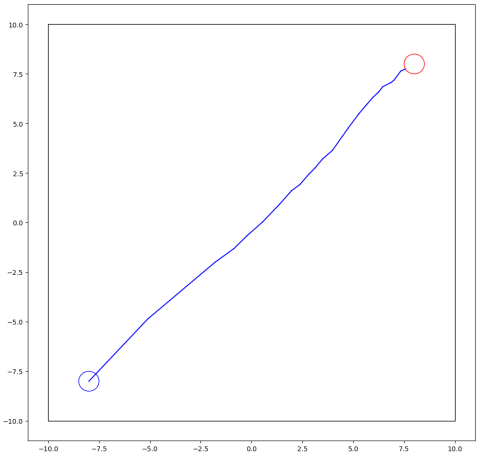

Fig. 3. Motion in a free space.

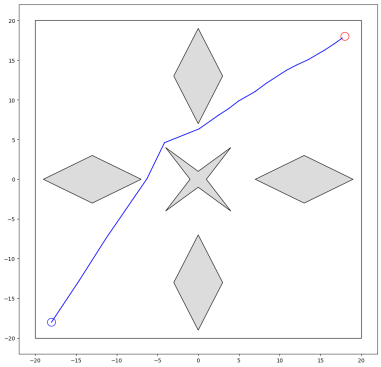

Fig. 4. Motion in a constrained environment.

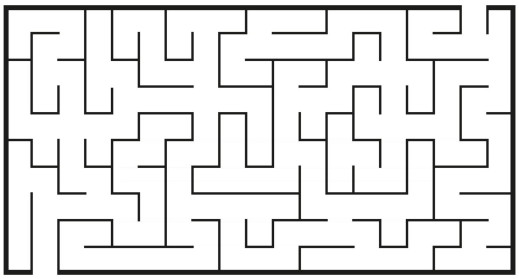

Fig. 5. Motion in a more constrained environment.

III. BACKGROUND

A. RRT*

RRT* is an optimal sampling-based motion planning algorithm that was first introduced by Karaman and Frazzoli in 2011 [9]. The key idea of RRT* is to construct a tree by iteratively connecting random samples to the existing tree, while also re-wiring the tree to improve the solution quality. The re-wiring step involves checking if a new path from the root to the new node is shorter than the existing path, and if so, modifying the tree accordingly. RRT* maintains a set of nodes that have been visited and a set of nodes that are yet to be explored. The algorithm terminates when a feasible path is found or a maximum number of iterations are reached.

IV. SYSTEM DESIGN

The proposed system architecture in this study focuses on a hardware and software co-design for the RRT* algorithm, with a specific target on its implementation within embedded FPGAs. This implementation employs both float16 and fixed-point (32 bits) data representations. To validate the system, tests are conducted on bare metal software utilizing the Xilinx Zynq-7020 SoC (Zybo-Z7 development board). The core of this hardware acceleration approach is the "neighbourhood processor," responsible for delivering information about neighbouring nodes. This processor optimizes off-chip communication through the AXI-Stream interface, establishes direct communication with the CPU via the AXI4-Lite interface, and utilizes on-chip memory (BRAM) for storage.

Algorithm 1 RRT* Algorithm

Require: Start configuration $start$, goal configuration $goal$, maximum iterations max_iter
Ensure: Optimal path from $start$ to $goal$
  $RRT\_Nodes \leftarrow InitializeTree()$
  $RRT\_Nodes \leftarrow InitializeNode(init\_position)$
  while $N < total\_node$ **do**
    // Iterative step
    $Rand\_Node \leftarrow GenerateRandomNode()$
    $Nrst\_Index \leftarrow Nearest(RRT\_Nodes, Rand\_Node)$
    $xnew\_pos \leftarrow StreamRdr(RRT\_Nodes, Rand\_Node)$
    if $Obstacle\_free(RRT\_Nodes, Environment)$ **then**
      $X\_Near \leftarrow Near(RRT\_Nodes, xnew\_pos)$
      $Cost(RRT\_Nodes, xnew\_pos)$
      $SelectParent(RRT\_Nodes, xnew\_pos)$
      $RRT\_Nodes \leftarrow InsertNode(RRT\_Nodes, xnew\_pos)$
      $RRT\_Nodes \leftarrow RewireTree(RRT\_Nodes, xnew\_pos)$
    end if
  end while
  return ConstructPath($Tree$, $goal$)

The foundational structure of the hardware system is depicted in Figure 6. The hardware architecture is engineered using High-Level Synthesis (HLS). Initially, the accelerator is configured based on parameters derived from the AXI-Stream and AXI-Lite interfaces.

The design is based on the following properties of motion planning:

- The environment configuration is known and does not change. Hence, it can be stored on the accelerator on-chip memory.
- Since the information of the parent is not needed to compute the neighbourhood of the generated node, it is completely ignored, thereby saving memory.

The architecture for the fixed-point implementation of the proposed Neighbourhood Processor is elaborated in Figure 7. Data is initially converted to a fixed-point format before computations begin. While achieving complete parallelization is challenging due to data dependencies, efforts are made to reduce these dependencies. An effective strategy for mitigating latency arising from data dependencies involves replacing conditional if statements with an alternative implementation utilizing signed shifted bits. This approach employs an array of boolean logic in conjunction with a logic gate, which is particularly implemented within the obstacle detection function. This substitution results in a significant enhancement in function latency, achieving an improvement of over 20 times.

In C/C++, the signed shift $n >> 31$ returns -1 for every negative number and 0 for non-negative number. [3].

Hence for the expression:

$$f = 1 + (n >> 31) = \begin{cases} \text{f=0}, & \text{if } n < 0 \\ \text{f=1}, & \text{if } n >= 0 \end{cases}$$

Consider the two equations below

$$x = a - b \tag{1}$$

$$c = 1 + x >> 31 \tag{2}$$

The value of c will be $zero$ if b is greater than a , hence, equation 1 and 2 is an alternative way of writing

$$if(a < b)$$

Using the parallelised implementation as shown in Listing IV-B rather than Listing IV-A ensures that the fucntion reduces the required clock cycle significantly.

Not all the if statement could be implemented this way- one important implementation is in the collision detection function where the latency improved by a factor of 20X for the fixed-point implementation. This as well increased the resource utilisation by 10%.

A. *Collision detection using if-statement*

```
static bool Collision_detected(Point_2 &A,
          float Squared_R_Robot,
          Segment_3 Edges[MAX_EDGES])
  {
     bool res = false;
     //MAX_EDGES=4
     for (int i=0;i<MAX_EDGES;i++)
     {
        if( Edges[i].squared_distance( A ) <
           Squared_R_Robot)
        return true;
     }
     return res;
  }
```

B. *Parrallelised implementation of the collision detection function*

```
static bool Collision_detectedhw( Point_2 &A,
               int32_t Squared_R_Robot,
               Segment_3hw Edges[MAX_EDGES])
{
   int32_t ff;
   bool res[MAX_EDGES]; //MAX_EDGES=4
   for (int i=0;i<MAX_EDGES;i++)
   {
      ff=Edges[i].squared_distance( A) -
            (Squared_R_Robot);
      res[i]=1 + (ff >> 31);
    }
    return !(res[0]&res[1]&res[2]&res[3]);
}
```

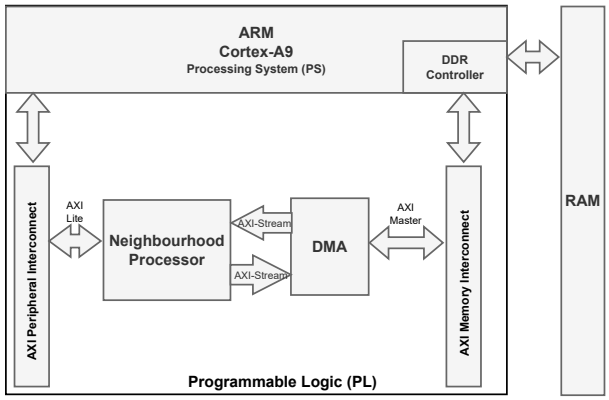

Fig. 6. Base system architecture.

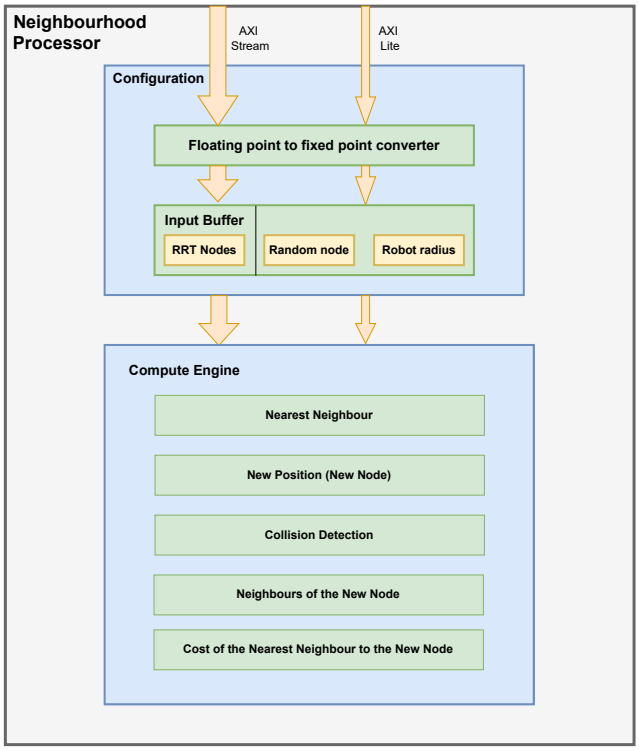

Fig. 7. Neighbourhood processor architecture.

V. EXPERIMENTAL RESULTS

The proposed hardware/software co-design framework is demonstrated as a bare metal software implementation on a Xilinx Zynq-7020 SoC (Zybo-Z7 development board). On the PL, we implemented the proposed hardware architecture with a clock frequency at 150MHz. For evaluation, the initial position is given as -8,-8 and the goal position is 8,8. The precomputed number of nodes required by the robot to reach the goal position based on the given environmental constraint was 10,000. The robot's search diameter was given as 0.25. The memory range required to save the intermittent computation such as the number of the neighbours for each node was as well precomputed.

TABLE I
PROFILING OF THE RRT* ALGORITHM- THE COLOURED REPRESENTS THE ACCELERATED PART

Function	Calls	Self (s)	Time (%)
Cal_Sqrd_Dist	105311459	0.50	34.02
Sqrd_Dist	105322493	0.23	15.65
Point_2	211659445	0.89	10.55
Point_2	219715720	0.12	8.17
Nearest	11034	0.11	7.48
Point_2::y	137419124	0.07	4.76
Near	9999	0.07	4.76
Point_2::x	137419124	0.07	4.42
Coll_detection	1003419	0.05	3.40
Segment_2	4012095	0.02	1.36
Point_2	32082	0.02	1.36
Sqrd_Dist	4012076	0.01	0.68
Obstacle_Free	168099	0.01	0.68
Cal_Dist	11034	0.01	0.68

A. *Result Accuracy*

The accuracy of the implementation is determined based on if the robot was able to get to the goal position after 10,000 nodes and how long it took using the neighbourhood processor in comparison to the software implementation.

1) Goal Position Reachability: After 10,000 nodes, the goal position was reached. Fig 8 shows the plot of the trajectory.

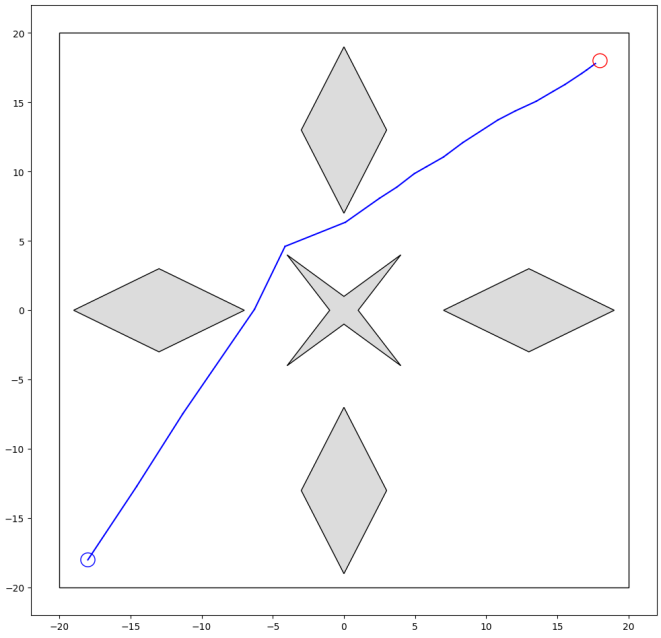

Fig. 8. Trajectory plot generated after 10,000 Nodes

2) Computational Time: The computational time is evaluated on three basis

a) Total time required to calculate the neighbourhood for 10,000 nodes using the fixed-point IP (In hardware, IP refers to intellectual property related to hardware designs and components.), float16 IP and software (without acceleration) with bare metal software implementation on Xilinx Zynq-7020 SoC (Zybo-Z7 development board). The result is as shown in Table II.

b) Total time taken for the overall RRT* computation using the fixed-point IP, float16 IP and software (without acceleration) implementation with bare metal software implementation on Xilinx Zynq-7020 SoC (Zybo-Z7 development board). The result is as shown in Table III.

TABLE II
TIME TAKEN TO COMPUTE NEIGHBOURHOOD NODES FOR 10,000 NODES

Compute Engine	Time(s)
CPU	124.04
CPU + IP(float16)	3.27
CPU + IP(fixed-point)	2.07

TABLE III
TIME TAKEN TO COMPUTE THE OVERALL RRT* ALGORITHM

Compute Engine	Time(s)
CPU	129.34
CPU + IP(float16)	8.57
CPU + IP(fixed-point)	7.61

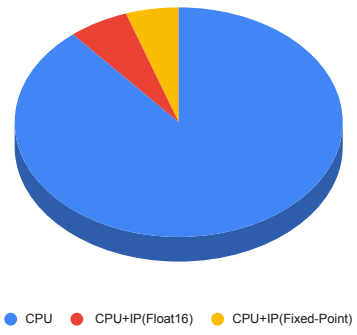

Fig. 9. Plot of time taken to compute the overall RRT* algorithm

3). Resource Utilisation: The resource utilisation in terms of BRAM, DSPs, LUT and power are as shown in Table IV

B. *Clock Frequency Design Exploration*

The fixed-point (32 bits) architecture design was explored for 50MHz, 100MHz and 150MHz clock frequencies. The computational time and the power-delay product were compared for various clock frequencies. The obtained result is as shown in table V

TABLE IV
RESOURCE UTILISATION

Compute Engine	BRAM(%)	DSP(%)	LUT(%)	Power(W)
CPU + IP(fixed-point)	33.93	64.09	12.33	1.83
CPU + IP(float16)	37.5	81.36	9.46	1.88

TABLE V
COMPARISON OF COMPUTATIONAL TIME AND POWER-DELAY PRODUCT OF THE CPU AND THE FIXED-POINT ACCELERATOR AT VARIOUS CLOCK FREQUENCIES

Compute Engine	Time (s)	Energy-Delay Product (J)
CPU	124.7	173.9
CPU + IP@50MHz	4.94	8.8
CPU + IP@100MHz	2.73	6.0
CPU + IP@150MHz	2.07	4.3

VI. CONCLUSION

In this paper, a hardware accelerator called a neighbourhood processor was presented. It is a dedicated hardware that accelerates the computation of the neighbours of a particular node in a RRT* computation. It takes as input the RRT nodes and the configuration parameters such as the random node, the robot radius. The proposed architecture was implemented using fixed-point and float16 number representation which is demonstrated using bare metal software on Xilinx Zynq-7020 (zybo development board).

The accelerator processor running at 150MHz on Xilinx Zynq-7020 achieves 60X runtime acceleration in calculating the neighbours for 10,000 nodes and improved the overall computation of the whole RRT* algorithm by 17X when compared with the software implementation.

REFERENCES

[1] A. Misra, C. He and V. Kindratenko, "Efficient HW and SW Interface Design for Convolutional Neural Networks Using High-Level Synthesis and TensorFlow," 2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC), St. Louis, MO, USA, 2021, pp. 1-8, doi: 10.1109/H2RC54759.2021.00006.

[2] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris and D. J. Sorin, "The microarchitecture of a real-time robot motion planning accelerator," 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, 2016, pp. 1-12, doi: 10.1109/MICRO.2016.7783748.

[3] https://www.geeksforgeeks.org/check-number-positive-negative-zero-using-bit-operators/

[4] Masehian, E., Sedighizadeh, D. Multi-objective robot motion planning using a particle swarm optimization model. J. Zhejiang Univ. - Sci. C 11, 607–619 (2010). https://doi.org/10.1631/jzus.C0910525

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] G. Malik, K. Gupta, R. Dharani and K Krishna "FPGA based hybrid architecture for parallelizing RRT,"

[7] J. Bialkowski, S. Karaman and E. Frazzoli, "Massively parallelizing the RRT and the RRT," 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, San Francisco, CA, USA, 2011, pp. 3513-3518, doi: 10.1109/IROS.2011.6095053

[8] S. Xiao, N. Bergmann and A. Postula, "Parallel RRT* architecture design for motion planning," 2017 27th International Conference on Field Programmable Logic and Applications (FPL), Ghent, Belgium, 2017, pp. 1-4, doi: 10.23919/FPL.2017.8056773.

[9] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," 2011 30th International Journal of Robotics Research USA, 2011, pp. 846-895, doi: 10.1177/0278364911406761.

[10] J. Nasir1, F. Islam, U. Malik, Y Ayaz, O. Hasan, M. Khan and M. Muhammad, "RRT*-SMART: A Rapid Convergence Implementation of RRT* motion planning," 2013 30th International Journal of Advanced Robotics System USA, 2013, pp. 1-12, doi: 10.5772/56718

[11] Y. Nevarez et al., "CNN Sensor Analytics With Hybrid-Float6 Quantization on Low-Power Embedded FPGAs," in IEEE Access, vol. 11, pp. 4852-4868, 2023, doi: 10.1109/ACCESS.2023.3235866.

[12] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.