Parallel RRT* Architecture Design for Motion Planning

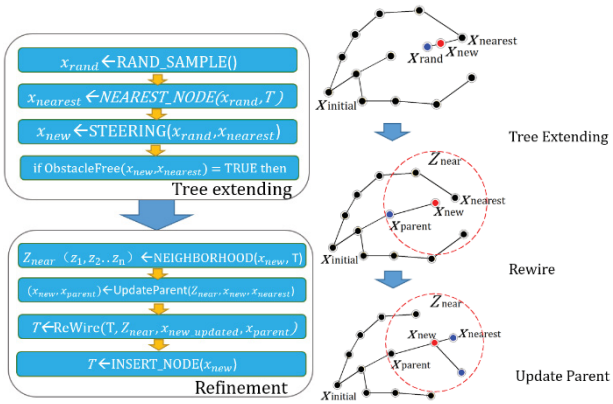
Size Xiao[1], Neil Bergmann[1] and Adam Postula[1]
[1]School of Information Technology and Electrical Engineering
The University of Queensland, QLD 4072, Australia

***Abstract*—A motion planning algorithm aims to calculate one obstacle-free trajectory which meets the dynamical constraints of a vehicle and leads the vehicle from the start state to the target state. RRT* (RRT star) is one sampling-based algorithm which is widely used in many applications because of its speed in quickly finding a trajectory. In contrast with basic RRT (Rapidly-exploring Random Trees) algorithm, RRT* improves trajectory optimality significantly by introducing the refinement step after setting up each tree node. This paper describes a new parallel version of the RRT* algorithm and its corresponding architecture on an FPGA (Field Programmable Gate Arrays). The refinement step is extracted and runs as a parallel process alongside a continuous tree-extending process. In order to avoid traversing all nodes in every iteration, tree nodes are stored in separate memory blocks and each block is assigned its own tree extending and refining pipelines to enhance the memory throughput. In the experimental evaluation, we take a 3-dimensional state spaces and implement the proposed architecture. The hardware implementation gives a 90 times speed improvement compared with an embedded software implementation, and a 30 times speedup compared to a desktop software implementation.**

I. Introduction

Motion planning is a significant issue for autonomous vehicles. In order to achieve good computation efficiency even in high-dimensional state space, sampling based motion planning algorithms such as RRT [4], RRT*[1] and PRM [9] (Probabilistic RoadMap) are widely used. Usually these algorithms are implemented on general-purpose processors. Due to the relatively low efficiency of software execution in some cases, using dedicated motion planning hardware on a reconfigurable device becomes another option for better flexibility and power-efficiency.

Among a variety of sampling based motion planning algorithms, RRT-based algorithms are relatively easy to implement without establishing a detailed map and also they are efficient for high dimensional state space. In Listing 1, RRT* is described where basic RRT procedures are included. States are represented as nodes. The algorithm starts with the initial state, and then new nodes are added into the tree in every iteration while the tree edges are refined in every iteration. More precisely, one random obstacle-free state is sampled and the NN (Nearest Neighbor) search returns the nearest node of this sampled state. Then, a new node is computed by moving one step from the nearest node toward the random state. "Nearest" normally uses a Euclidean-distance metric. This basic RRT gives a solution which is not optimal because the consideration of total moving cost is absent when choosing the parent for the new node. To improve the solution, RRT* adds a refinement step after every RRT iteration. As shown in Listing 1, the neighborhood of the new node within a certain hyper-ball [1] is temporarily picked. The refinement process updates the parent of the new node by comparing the total cost when each neighbor is assumed to be the parent. This total cost contains two parts: the cost of moving from the parent to the new node, and cost estimated from initial state to the parent. The neighbor which gives minimum total cost is finally chosen as the parent of the new node. The rewiring procedure is the reverse check where the above parent-updated new node is selected as the parent of any neighbor if it can reduce the total cost. Then, RRT* enters into the next iteration after the new node and updated connections are added to the tree.



Listing 1: Main procedures of RRT* algorithm

The RRT* algorithm shows asymptotical optimality [1]. The executable solution is continuously updated during iterations, i.e., the vehicle can follow the real-time optimized trajectory. Meanwhile, an improved solution at the cost of more computation. This feature gives potential for a parallelized implementation. Tree extending and tree refinement are designed as two parallel pipelines which provide better implementation efficiency compared to the conventional sequential software execution. For the time-consuming NN and neighborhood search, we split the tree such that tree nodes are stored in distributed RAMs for multiple access.

II. Related Work

Parallel features and implementations of RRT related algorithms has been previously researched. In [7], two RRT processes are working on the same solution: one tree grows from the initial state and expects to connect with another tree which grows back from the destination. As mentioned in [4], the NN search and collision detection of RRT are bottlenecks

for its performance. Research in [10] [11] implements collision detection on a GPU (Graphics Processing Unit) so that tasks are divided into sub-tasks and executed on different dimensions of CUDA (Compute Unified Device Architecture) thread blocks. In [6], an effort is made to utilize a dedicated data structure for efficient indexing. The author presents one KD-tree based NN search architecture where multiple query modules are stored in the Block RAM in an FPGA to increase the memory throughput. In contrast with the above software based approach for accelerating certain algorithmic procedures, we aim to develop a complete scalable RRT* hardware architecture to fit various reconfigurable devices.

III. HARDWARE ARCHITECTURE FOR RRT*

This section describes the top-level system structure, followed by the key function modules.

A. RRT Architecture*

According to the execution steps of RRT* in Listing 1, once a new node is returned by the steering function, the state information in each dimension is fixed and only the parent could be updated in further iterations. To utilize this property, our approach is to design tree extending and refining as separate pipelines connected to one dual-port memory which maps to the Block RAM of the FPGA. In Figure 1, nodes are distributed into Block RAMs and each of them represent one sub-tree. The node frame consists of four parts. *ID* is unique and given to the new node according to a generating sequence. As every node only has one parent, the *index* indicates its connection status which is the ID of its current parent. "Cost" measures the total cost counted from the initial state to the current state via the parent. The cost metric is important and affects the performance of the path planner. So, one distance metric function is essential to define the optimal cost of the target moving between nodes. In this current work, we adopt a Euclidean distance metric that is estimated based on a weighted state information [4] such as d_n shown in Figure 1:

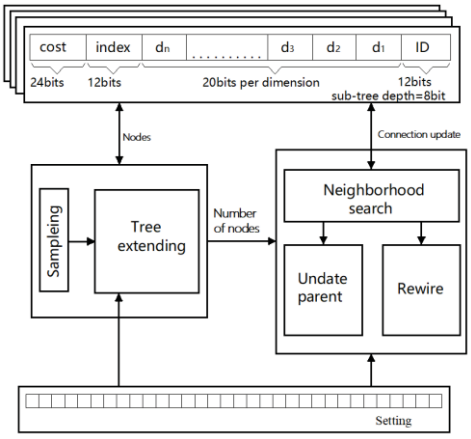


Figure1. Fundamental architecture of RRT path planner*

To simplify the operation in NN and neighborhood search, instead of calculating the square root, squared Euclidean distance is used for the comparison operation that only consumes DSP (Digital Signal Processing) resources. For computing the total cost which will be stored into sub-tree, vectoring-mode CORDIC (COordinate Rotation DIgital Computer) is employed to calculate the square root.

The performance improvement caused by parallel extending-refining implementation is now discussed. For sequential RRT* execution, refinement always works on the most recent new node. By contrast, in this work, the tree extending process approximates the basic RRT algorithm and refinement is comprised of two pipelines as shown in Figure 2. Especially in the Rewire pipeline, all Steering and Collision Detection are done before the parent-updated new node is ready. Due to the refinement process having more operations, tree extending progress is always ahead of the refinement progress such as the example shown in Figure 3. This progress gap depends on the set value of the search hyper-radius. Within a certain computation time, this proposed parallel RRT* architecture generates more nodes and explores more configuration space, and also could be faster to provide one partially optimal solution for the total path. For Neighborhood search, there is more possibility to find more neighbors and rewire more tree connections.

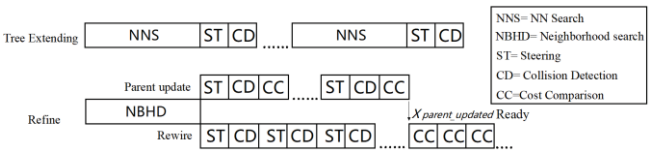


Figure2. Pipeline sequence of Tree extending and Refinement

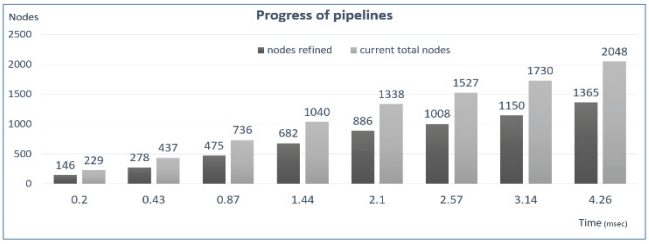


Figure3. Example of tree extending progress versus refinement progress for generating 2048 nodes

B. Tree Extending

The above section describes the algorithmic feature that are used parallelize RRT*. In Figure 4, we give the detailed architecture for tree extending. For convenience of description, the number of dimensions is set to three and the size of sub-tree is 256 x 108bits including 60bits of dimensional state.

Sub-NN controller

This controller works as the first level in the pipelines. It samples the random dimensional state from the LFSR (Linear-Feedback Shift Register) based pseudo-random number generator. The sampled state is applied as a query node for NN search and the returned nearest node is sent to steering module to get a new node. As the tree nodes are distributed into separate Block RAMs for parallel access, each sub-NN module will return one answer for the query, hence, the best answer is selected.

Sub-NN

Each sub-NN is directly connected to one dual-port Block RAM. In read mode, the sub-NN implements the NN search algorithm corresponding to the data structure in the sub-tree. As mentioned in section 3A, the calculated squared Euclidean is the criterion for nearness measurement. After searching, Sub-NN returns the nearest neighbor of the query node with the minimum distance. In write mode, the new node from sub-NN controller is returned to the sub-tree and the write address is determined by the total number of nodes.

Steering

The Steering module is a dedicated calculator for the dynamic model of controlled vehicle. For a pair of nodes $(x_{new,}x_{rand})$ as input, it can generate the path from x_{new} to x_{rand}. This path is represented by discrete nodes for the collision detection step. Considering the possible arithmetic requirement, one 8-level pipelined CORDIC (COordinate Rotation DIgital Computer) with a 32-bit divider is used for computation.

Collision Detection

The Obstacle Detection module is designed to traverse all obstacle centers and check if the discretized path collides with any obstacle region, where the squared Euclidean distance is used. The result will be returned to the sub-NN controller to determine if collision occurs or not.

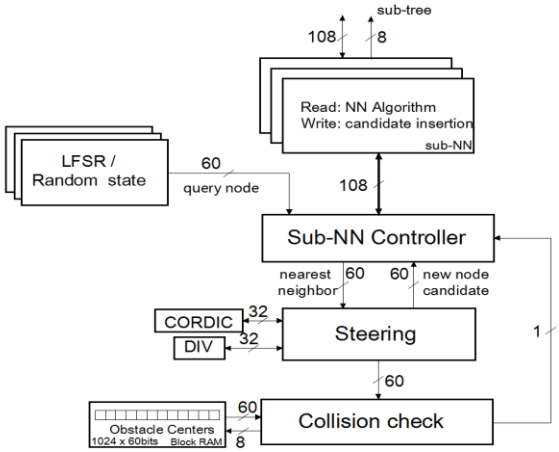


Figure.4 Tree extending architecture

C. *Refining Pipeline of RRT**

This pipeline is designed for refining the node which follows the order of nodes generated. Once the progress of refinement is behind or equal to the tree extending progress, the pipeline will be enabled.

The detailed refinement architecture is shown in Figure 5. The second port of each sub-tree (dual-port Block RAM) is assigned to one sub-NBHD (NeighBorHooD) module. After sub-NBHD retrieves the node, it traverses the sub-tree to find all neighbors within the given hyper-radius r. Every found neighbor is transferred to a data sink named Neighborhood. This data sink is mapped to the distributed RAM on the FPGA to meet the Single-Write Dual-Read demand, as the neighbor z_{new} may be written to Neighborhood while Parent Update and Rewire processes are reading from Neighborhood.

The Parent Update and Rewire are constructed as two parallel sub-pipelines. In the Parent Update process, for any newly found neighbor z_{near}, it tries to make z_{near} as the parent of x_{new} which is the new node candidate under refining. After inspecting all z_{near}, it outputs the recent updated x_{new} to both Rewire process and sub-NBHD modules, where $DONE_{PU}$ is used to indicate the completion of the task. For the Rewire process, there are similar procedures with the difference that Rewire is conducting steering and collision detection first when x_{new} is being updated. After both neighborhood search and x_{new} update are complete, x_{new} is made the parent of any collision-free z_{near}. The Rewire process will return the number of the updated z_{near} as an address which is used to access its output buffer named as Updated z_{near} in Figure 5.

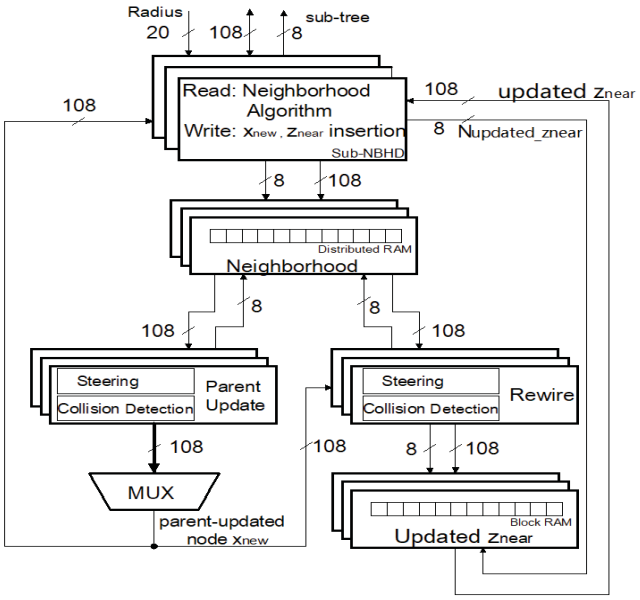


Figure.5 Architecture of refinement pipeline

Algorithm 1 Process_Parent_Update

If $N_{PU} \le N_{z_{near}}$ **and** $N_{z_{near}} \ge 1$
  $z_{near} \leftarrow$ ReadZnear (N_{PU});
  path_nodes1 $\leftarrow$ Steering(z_{near}, x_{new});
  If Collision _free(path_nodes1) = true
    c_1 $\leftarrow$ Cost(z_{near})+Cost($\overrightarrow{z_{near}x_{new}}$);
    If c_1<Cost(x_{new})
      $x_{new} \leftarrow$ MakeParent(z_{near}, x_{new});
    End if;
  End if;
  $N_{PU} = N_{PU} + 1$;
Elsif $DONE_{NBDH} = 1$ **and** $N_{PU} > N_{z_{near}}$
  $DONE_{PU} = 1$;
  Return x_{new};
End if;

Algorithm 2 Process_Rewire

If $N_{RW} \le N_{z_{near}}$ **and** $N_{z_{near}} \ge 1$
  $z_{near} \leftarrow$ ReadZnear (N_{RW});
  path_nodes2 $\leftarrow$ Steering(x_{new}, z_{near});
  If Collision_free(path_nodes2) = true
    // temporarily save
    WriteBRAM(z_{near} , $N_{Collision\_free}$,);
    $N_{Collision\_free} = N_{Collision\_free} + 1$;
  End if;
  $N_{RW} = N_{RW} + 1$;
End if;

If $DONE_{NBDH} = 1$ **and** $DONE_{PU} = 1$;
  For i = 1 to $N_{Collision\_free}$ **do**
    $z_{near} \leftarrow$ ReadBRAM (i) ;
    c_2 $\leftarrow$ Cost($\overrightarrow{x_{new}z_{near}}$);
    If c_2+ Cost(x_{new})<Cost(z_{near});
      $z_{near} \leftarrow$ MakeParent(x_{new}, z_{near});
WriteBRAM(z_{near} , $N_{Collision\_free}$,);
      $N_{Updated\_z_{near}} = N_{Updated\_z_{near}} + 1$;
    Enf if;
  End for;
  $DONE_{RW} = 1$;
  Return $N_{updated\_z_{near}}$;
End if;

Listing 2: Procedures of Parent Update and Rewire

IV. RRT* ARCHITECTURE EVALUATION

The proposed RRT* architectures are implemented on a Xilinx Artix-7 FPGA. In this experiment, the size of each sub-tree is 256 x 108bits and the complete tree consists of 8 sub-trees which are connected to 8 parallel NN/NBDH search and refinement pipelines. For performance comparison, the RRT* algorithm program [12] is also executed on a desktop PC (Xeon E3 1241V3 3.5GHz) and an embedded processor, Raspberry Pi 3 (ARM Cortex-A53 CPU, 1.2GHz) running a Linux environment. The averaged time cost which concluded from 50 times implementations is presented in TABLE I. According to the results, the proposed architecture is 33 times faster than the PC and 90 times faster than the ARM processor. This acceleration greatly benefits from the paralleled NN/neighborhood search and the optimized refinement pipeline. It is should be noted that for a given task generating 2048 nodes, the progress of the refining process lags behind the tree extending process as illustrated in Figure 3. Therefore, the computation time is determined by the slower refinement process. In TABLE.II, the synthesis result shows the hardware cost of the current 8-pipeline structure for a 3-dimensional configuration space. This result will vary with different application parameters and the number of pipelines. The current architecture can be scaled to fit more complicated applications like increasing the dimensional configuration space, changing the steering function and replacing the NN search algorithm, or adding more NN search and refinement pipelines, which will cause more FPGA resource use.

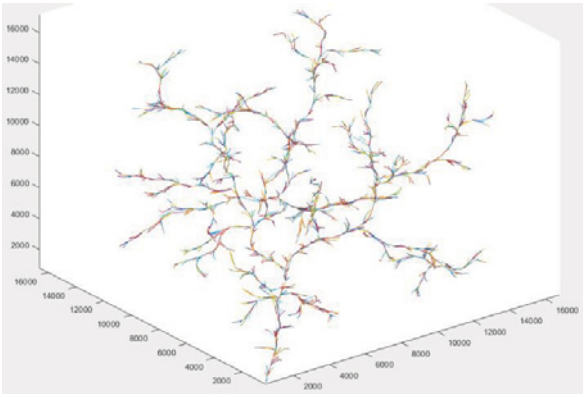


Figure.7 Demonstration of 3D Parallel RRT implementation Range=16384×16384, tree size =2048 nodes, root at (129,129,129),*

TABLE I. AVERAGE TIME COST OF EXPERIMENT

Stop condition: 2048 nodes Obstacles number = 3	Averaged time cost	Freq.(Hz)/Power
Parallel RRT* on FPGA	9.43ms	100MHz/Est. 0.561W
C code Xeon E3 3.5GHz	312ms	3.7GHz
C code ARM A53 1.2GHz	843ms	1.2GHz/2.9 W

TABLE II. RESOURCE USAGE OF ARTIX-7 (XC7A100T)

	Flip Flop	LUT RAM	LUTs	BRAM.	DSP
RRT* on FPGA	3898 (126800) (13%)	3904 19000 (20.55%)	46914 (63400) (74%)	88 135 (65%)	174 240 (72.5%)

V. CONCLUSION AND FUTURE WORK

In this work, a fully pipelined RRT* architecture has been developed in which the tree extending and node refining processes are parallelized to improve the implementation efficiency. For the core nearest neighbor and neighborhood search procedures, we divide the search tree and distribute its nodes into separate memory blocks with independent access. This method can dramatically improve the NN search and data throughput. In the refinement process, we optimize the parent-update and rewiring pipelines to operate them in parallel. In the experiment, the RRT* architecture is implemented on FPGA and proved much faster compared to software RRT* execution on both PC and ARM platforms. In future work, the developed RRT* architecture can be tested for its operation on a practical platform such as a UAV or robot. Currently, the limited on-chip RAM is the bottleneck for storing large numbers of nodes and obstacles. In future work, the use of off-chip memory will be examined.

VI. REFERENCES

[1] Karaman, Sertac, and Emilio Frazzoli. "Incremental sampling-based algorithms for optimal motion planning." Robotics Science and Systems VI 104 (2010).

[2] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli and S. Teller, "Anytime Motion Planning using the RRT*," 2011 IEEE International Conference on Robotics and Automation (ICRA), Shanghai, 2011, pp. 1478-1483.

[3] Goerzen C, Kong Z, Mettler B. A survey of motion planning algorithms from the perspective of autonomous UAV guidance. Journal of Intelligent and Robotic Systems, 2010, 57(1-4): 65-100.

[4] LaValle S M. Rapidly-Exploring Random Trees A Цew Tool for Path Planning. 1998.

[5] Brown, Russell A. "Building kd Tree in O (knlog n) Time." Journal of Computer Graphics Techniques 4.1 (2015).

[6] T. Kuhara, T. Miyajima, M. Yoshimi, and H. Amano, "An FPGA Acceleration for the Kd-tree Search in Photon Mapping", *Reconfigurable Computing: Architectures, Tools and Applications.* Springer Berlin Heidelberg, 2013. 25-36.

[7] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation, San Francisco, CA, 2000, pp. 995-1001 vol.2.

[8] Andraka R. "A survey of CORDIC algorithms for FPGA based computers" , Proceedings *1998 ACM/SIGDA sixth International Symposium on Field Programmable Gate Arrays (FPGA)* , pp.191-200.

[9] N. Atay and B. Bayazit, "A motion planning processor on reconfigurable hardware," Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006., Orlando, FL, 2006, pp. 125-132.

[10] 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, 2011, pp. 3513-3518.

[11] 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, 2016, pp. 1-12.

[12] Sertac Karaman, Emilio Frazzoli, RRT(*) Library, https://svn.csail.mit.edu/rrtstar