HLS-Based Approach for Embedded Real-Time Ray Tracing in Wireless Communications

Jintong An and Selma Saidi

***Abstract*—With the development of wireless communication technology, complex and dynamic scenarios pose great challenges to the Quality of Service (QoS) of wireless communication, especially in indoor scenarios. The quality of beam management can be greatly improved if signal ray-tracing module is embedded in wireless devices to handle synthetic multipath transmissions in real time. In this article, a novel reflection path derivation algorithm for ray tracing of signal beams is proposed, which builds the core mechanism of the proposed FPGA accelerator for ray tracing: by decomposing the computation of the entire ray path into mutually independent subproblems associated with the respective planes involved in the reflection and implemented by independent processing element on FPGAs, the parallelization of the entire ray tracing is realized, which significantly improves the convergence speed of the ray tracing; meanwhile, a new high-level synthesis workflow corresponds to the proposed algorithm and hardware architecture is proposed, which opens the door on synthesizing embedded hardware dedicated for robust and real-time wireless communication. After validation, the method proposed in this article can generate FPGA accelerator for real-time ray-tracing effectively, which achieves ray-tracing simulation in milliseconds.**

***Index Terms*—Beamforming, FPGA, high-level synthesis (HLS), real-time ray tracing.**

I. Environmental Awareness in Wireless Communication and Ray Tracing

With the booming development of communication technologies, the resulting increase in frequency and bandwidth poses challenges to the robustness of the Quality of Service (QoS). For example, in indoor scenarios, due to the complexity of the environment (e.g., placement of furniture, movement of people, etc.), the multipath transmission effect caused by multiple reflections of signals on reflection planes has a significant impact on beam tracking and link maintenance [1]. This effect is especially true for beamforming techniques of antenna arrays: by focusing the signal power in a specific direction through beamforming to compensate for the path loss due to frequency increase, the signal coverage is greatly enhanced and the data rate is also increased. While this is an advantage over omni-directional antennas, it also introduces complex tracking mechanisms for dynamic targets. When transmitting in complex indoor environments, the multipath effect makes beam management more difficult, and the beamformer (beam management unit) must detect, predict and select the optimal direction for transmission, which is highly dependent on the environment, and once the environment suddenly changes, such as opening the door, causing the original beam configuration to fail, the beamformer has to re-establish the link, which causes link outage and affects the QoS of the communication, as shown in Fig. 1.

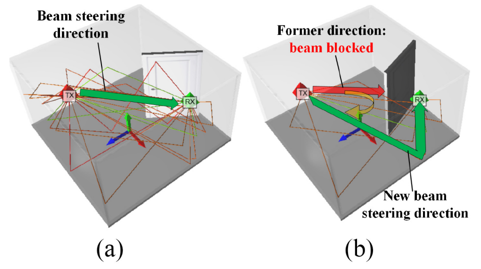

Fig. 1. Ray-tracing simulation showing the effect on beams when the door is suddenly opened: Beamformer should re-establish the link between transmitter (TX) to receiver (RX) that was disconnected by the outage. (a) Before door's opening. (b) After door opens.

Therefore, if the wireless devices can sense the dynamic changes in the environment through sensors, such as depth cameras/LiDARs, the impact on signal propagation can be efficiently simulated by the virtualized digital twin, as shown in Fig. 2, in order to derive the optimal beamforming configurations for current situation.

In the field of wireless communication, ray-tracing technology is widely used due to its low cost and high accuracy, meeting the needs of the virtualized digital twin in wireless devices. In [2], authors experimentally confirmed that ray-tracing simulation results can well match real measurements. In their experiment, 87% of the received signal strengths were predicted with an error of less than 3 dB, and all the simulations had an error of less than 5 dB. In addition, [3] and [4] also made a similar conclusion, that is, the simulation of wireless communication scenarios by ray tracing can accurately extract critical parameters (e.g., received signal strength, path loss, power delay profile, etc.) that matches real measurements, which demonstrates practicality of ray tracing to help in design aspect of wireless communication.

Manuscript received 11 August 2024; accepted 12 August 2024. This work was supported by the German Federal Ministry of Education and Research (BMBF) in the course of the 6GEM Research Hub under Grant 16KISK038. This article was presented at the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES) 2024 and appeared as part of the ESWEEK-TCAD Special Issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Jintong An.)*

Jintong An is with the Chair of Embedded Systems, TU Dortmund University, 44227 Dortmund, Germany (e-mail: jintong.an@tu-dortmund.de).

Selma Saidi is with the Institute of Computer and Network Engineering, TU Braunschweig, 38106 Braunschweig, Germany (e-mail: saidi@ida.ing.tu-bs.de).

Digital Object Identifier 10.1109/TCAD.2024.3446710

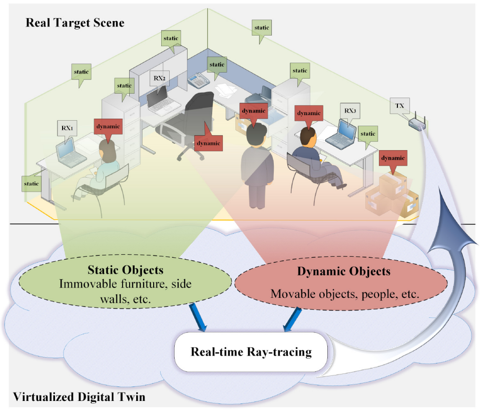

Fig. 2. Digital twin of ray tracing in wireless device (here is the wireless router): Objects in the scene are organized into dynamic and static, and are annotated in red and green, respectively. The motion of dynamic objects is fully explored in the digital twin and the resulting consequences are simulated by ray tracing, which guides the wireless device to sufficiently master the possible situations in the dynamic scene and to perform correctly in communication settings, such as antenna array and beamforming configurations.

In this article, the real-time performance of the ray-tracing routine impacts greatly on the immediacy of the digital twin, i.e., the ray-tracing calculations should be done in milliseconds to ensure that the simulation results are correctly and timely assisted for the actual beamformer configuration. Reference [5] used point cloud data (PCD) as a static simulation scenario and utilized NVIDIA frameworks for acceleration. Although the computation rate was improved by 49.8 times compared with that without GPU acceleration, it is still inapplicable to the embedded real-time systems as targeted by this article due to the long ray-tracing execution time (487 s). Reference [6] also accelerated ray tracing via GPUs, which, although 16 times faster than using CPUs for computation, still did not meet the millisecond simulation requirements targeted by our article (ray tracing finished in 8756ms in [6]). Furthermore, the experimental results in [7] also showed that rationally optimizing the ray-tracing algorithm and parallelizing it using GPUs can effectively accelerate the simulation, resulting in the computational time consumed for ray tracing being reduced to within a few seconds, which, however, clearly falls short of the millisecond level of elapsed time in ray tracing that we expect.

For widely used shooting-and-bouncing-rays (SBRs) algorithm in ray tracing, it is necessary to traverse all reflection planes to derive the correct ray-plane intersection points given by the incident vectors, which is recursively computed until the rays approach the RX to obtain the final set of paths. Calculating the intersection of rays and scatters in SBR algorithm occupies more than 90% of the computation time [8], so if the scatters which are likely to intersect with the rays at a certain direction are known in advance, it will speed up the computation significantly. For indicating interplane visibility relationships, Liu and Guo [9] and Hu et al. [10] proposed a new data structure called Virtual Source Tree, in which the possible paths can be obtained through routing from transmitter (TX) to receiver (RX). While a ray-tracing simulation of an outdoor street grid scene can be done in 2 s if the potential reflection paths obtained by routing in the VST in advance, again, the experimental results of this article clearly did not meet the millisecond timing requirements we propose.

Compared with ray tracing in static environments, dynamic scenes lead to more complex ray tracing due to the dynamic changing of positions and attitudes of the reflection planes involved in signal reflection. Hussain and Brennan [11], [12], [13] derived the rules for updating the interplane visualization relations for linear motion in accordance with the principles of geometrical optics (GOs) with the assistance of the proposed intravisibility matrix, which greatly improves the ray-tracing efficiency in processing scenes with dynamic motion, resulting in an acceleration of ray tracing for typical urban scenes for V2X application, reducing the processing time to few seconds (indicator τ_{it} in the papers). However, compared to the V2X scenarios where vehicles follow fixed paths, indoor scenarios with random object movements (e.g., movement of people, opening and closing of doors, placement of furniture and objects, etc., all affecting the final signal propagation and being simulated by ray tracing) require more complex geometrical processing in order to determine the interplanar visualization relations, which should be implemented in an efficient way for the ray-tracing simulation on the digital twin for real-time processing.

From the above discussion of the existing work, it can be concluded that although the ray-tracing algorithms are optimized by improving the data structure and accelerating the ray-tracing routines in parallelization with the support of GPUs, the various ray-tracing methods proposed in the existing work still fail to satisfy the timing requirements (in our article is milliseconds) and computational speeds of real-time ray tracing for the applications that we expect to be applied in the virtualized digital twins in wireless devices. On the one hand, this is due to the angular sweeping mechanism in the implemented SBR algorithm, i.e., the SBR algorithm meaninglessly consumes the vast majority of the runtime for the exclusion of noncritical paths: for example, sweeping the space with one-degree angular interval at TX yields $180 \times 360 = 64\,800$ rays. However, the final result leaves only one ray (ideally) as the critical path, which has the lowest path-loss/propagation delay and thus contributes the most to the signal propagation and channel states; the vast majority of the remaining rays are regarded as noncritical paths, which are discarded in the final ray-tracing results, yet the computation of which takes up the vast majority of the runtime. On the other hand, GPU acceleration uses batch processing to parallelize the same operations, which is not suitable for forward-serial ray-tracing computations: coordinates of the next intersection point and the direction in the next round of iteration depend on the forward calculation from the previous round of iterations. And most of the reflections in the batch will not lead the ray to the end point RX, which, however, cannot be eliminated earlier in the forward computation.

Therefore, in order to improve the ray tracing applied on embedded wireless devices for real-time simulation, this article first proposes a novel algorithm for reflection path derivation, i.e., the iterative path convergence (IPC) algorithm. By decomposing the overall derivation of one potential reflection path into mutually independent subproblems related to the individual planes involved in reflection, the computation of the reflection path is parallelized and globally accelerated. Meanwhile, since geometric operations, such as projection of vectors, are involved in each subproblem routine, we propose to instantiate the geometric operations involving each plane in the IPC algorithm using independent processing elements (PEs) in order to map the parallelization of the IPC algorithm to physically parallel circuits on FPGA. The main contributions are as follows.

1) An IPC algorithm for computing signal reflection paths is proposed, which is designed for parallel processing of the ray paths and easy to implement in hardware;
2) An high-level synthesis (HLS) workflow is proposed: geometric data of the scene is compiled to obtain weak visibility relationship networks (WVRN) and used to infer potential reflection paths; the IPC algorithm is then achieved on FPGA to accelerate ray-tracing speed to meet the timing requirement of milliseconds for real-time embedded applications;
3) The idea of embedding real-time ray-tracing platform in wireless devices is not to reconstruct the signal propagation (e.g., reconstruct radiation pattern) in a target scene together with high-precision physical models (e.g., frequency bands, path-loss models, reflective plane materials and reflection coefficients, etc.), but rather to assist in beamforming for faster and more effective beam management: by quickly calculating potential signal reflection paths, the ray directions can be graded based on path length, reflection order, and reflection angle, thereby effectively assisting beamforming by filtering out the beam steering angle that has the greatest impact on signal propagation.
4) To the best of our knowledge, the HLS workflow proposed in this article is not covered by any existing work, which opens the door on synthesizing embedded hardware dedicated for robust and real-time wireless communication, especially for real-time beam management; the objective of our proposed HLS workflow is to provide a new solution targeting environment sensing and processing in 6G wireless communications aided by hardware-software co-design.

II. Iterative Path Convergence Algorithm

The most important aspect of ray tracing is the calculation of the reflection paths of the signal between the reflection planes. Based on the obtained reflection paths applied to signal models one can derive information, such as path loss, phase shift, time delay, etc., according to which the beamformer (i.e., beam management unit) can adjust the weight matrix of the antenna array to achieve beam steering. For an Nth-order reflection path ($Path_1$) as shown in Fig. 3, there are involved reflection planes $\{\Omega_1, \Omega_2, \dots, \Omega_N\}$, set of ray-plane intersection points on each planes $\{s_1, s_2, \dots, s_N\} \subset S$ where each point $s_n = [x_n, y_n, z_n]^T, n = 1, 2, \dots, N$ is represented by the 3-D coordinates in vector form, and fixed terminals

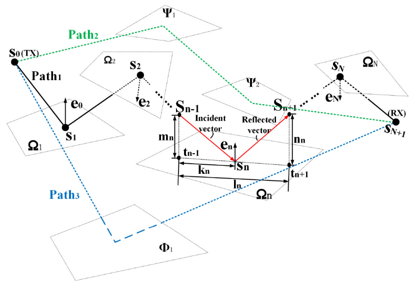

Fig. 3. Signal propagation via reflection: Nth-order reflection path ($Path_1$) with the involved reflection planes $\{\Omega_1, \Omega_2, \dots, \Omega_n\}$; another first- and second-order reflection paths $Path_3$ and $Path_2$ consist of $\{\Phi_1\}$ and $\{\Psi_1, \Psi_2\}$ as reflection planes, respectively.

$\{s_0, s_{N+1}\} \subset S$ which are, respectively, recognized as TX and RX coordinates. Then the plane Ω_n should satisfy the Householder transformation

$$H_n(s_{n-1} - s_n) = \alpha(s_{n+1} - s_n) \tag{1}$$

where H_n is the Householder matrix of plane Ω_n and α is the ratio of the modulus of the incident vector $\overrightarrow{s_n s_{n-1}}$ to the reflected vector $\overrightarrow{s_n s_{n+1}}$

$$\alpha = \frac{||s_n - s_{n-1}||}{||s_{n+1} - s_n||}. \tag{2}$$

The reversed incident vector $\overrightarrow{s_n s_{n-1}}$ and the reflected vector $\overrightarrow{s_n s_{n+1}}$ should, respectively, be related to the normal vector of the plane e_n as follows:

$$||s_{n-1} - s_n|| = \frac{e_n^T(s_{n-1} - s_n)}{\cos\theta_i} \tag{3}$$

$$||s_{n+1} - s_n|| = \frac{e_n^T(s_{n+1} - s_n)}{\cos\theta_r} \tag{4}$$

in which $\cos\theta_i \equiv \cos\theta_r$ is fulfilled when the reflection at s_n exists. Thus, the ratio α in the Householder equation could be derived as

$$\alpha = \frac{||s_{n-1} - s_n||}{||s_{n+1} - s_n||} = \frac{e_n^T(s_{n-1} - s_n)}{e_n^T(s_{n+1} - s_n)} = \frac{m_n}{n_n} \tag{5}$$

where $m_n = e_n^T(s_{n-1} - s_n)$ and $n_n = e_n^T(s_{n+1} - s_n)$ represent the distance from s_{n-1} and s_{n+1} to the reflection plane Ω_n, respectively.

Substituting (5) into (1), then the intersection point s_n could be described after rearranging as

$$s_n = (m_n I - n_n H_n)^{-1}(m_n s_{n+1} - n_n H_n s_{n-1}) \tag{6}$$

where I represents a 3×3 identity matrix.

Thus, the predicted value of s_{n+1} can be derived from s_n represented by s_{n+1}^p as

$$s_{n+1}^p = \frac{1}{m_n}\big[n_n H_n s_{n-1} + (m_n I - n_n H_n)s_n\big]. \tag{7}$$

Meanwhile, the calculated value of s_{n+1} described as s_{n+1}^c should satisfy

$$s_{n+1}^c = (m_{n+1} I - n_{n+1} H_{n+1})^{-1}(m_{n+1} s_{n+2} - n_{n+1} H_{n+1} s_n) \tag{8}$$

which formulates a constrained optimization problem as

$$\underset{\forall s_i \in S, i=1,2,\ldots,N}{\text{minimize}} \sum_{n=1}^{N} ||s_{n+1}^{p} - s_{n+1}^{c}||$$
$$\text{subject to} \quad s_n \in \Omega_n, n = 1, 2, \ldots, N \tag{9}$$

which implies that each intersection point should satisfy the minimum difference between the predicted and computed values at convergence of the path, where each point is constrained to be within the boundary of the corresponding plane. Although this constrained optimization problem can be solved by transforming it into Karush–Kuhn–Tucker conditions (KKT conditions) by applying the Lagrange multiplier method, the solution process is complex and not conducive to FPGA applications. In order to solve the above constrained optimization problem efficiently, we design an iterative method in the proposed IPC algorithm, where the computation of the overall path is decomposed into iterative computation for the ray-plane intersection point s_n on each scattering plane Ω_n, and the computation on each plane is independent to the others: the overall solution can thus be transformed into separate subproblems that can be solved in parallel for easy hardware implementation.

At each intersection point $s_n \in S$ satisfies

$$s_n = f(s_{n-1}, s_{n+1}; \Omega_n), n = 1, 2, \ldots, N \tag{10}$$

namely, the coordinates of the intersection point s_n on Ω_n are computed from the leading and following points on the reflection path (s_{n-1} and s_{n+1}, respectively), as shown in Fig. 3; t_{n-1} and t_{n+1} are the projection points of s_{n-1} and s_{n+1} on the plane Ω_n, respectively; l_n is the distance between the two projection points on the plane and k_n is the distance from the projection point p_{n-1} to s_n, then the following relation is obtained due to the Theorem of Similar Triangles:

$$\triangle s_{n-1} t_{n-1} s_n \sim \triangle s_{n+1} t_{n+1} s_n \Rightarrow \tag{11}$$
$$\frac{m_n}{n_n} = \frac{k_n}{l_n - k_n} \Leftrightarrow \frac{k_n}{l_n} = \frac{m_n}{m_n + n_n} \tag{12}$$
$$s_n = t_{n-1} + k_n \frac{t_{n-1} - s_n}{||t_{n-1} - s_n||} = t_{n-1} + l_n \frac{m_n}{m_n + n_n} e_l \tag{13}$$

with e_l indicates the direction of the vector $\overrightarrow{t_{n-1}t_{n+1}}$. Thus, the coordinates of the ray-plane intersection point can be obtained by the leading and following points on the path, which leads to an intuitive solution for the mentioned constrained optimization problem: through iterations the new coordinates of intersection points $s_1, s_2, \ldots, s_N$ in the reflection path P_N can be updated in parallel (or sequentially in software implementations) until the displacement of the new coordinates with respect to the old ones is less than threshold ϵ_N, which depicts computation convergence, as shown in Algorithm 1.

Algorithm 1: IPC Algorithm on One Nth Order Reflection Path

```
Input: Terminal points $TX$ and $RX$, list of bounded reflection
       planes $\Omega = \{\Omega_1, \Omega_2, ...\}$, list of potential reflection
       paths $P = \{P_1, P_2, \ldots\}$, convergence threshold $\epsilon_N$
Output: List of confirmed ray paths in exact coordinates $Q$
1  $s_0 \leftarrow TX$
2  $s_{N+1} \leftarrow RX$
3  $Q \leftarrow \emptyset$
4  while $P \neq \emptyset$ do
5      $path \leftarrow$ pop($P$)
6      $\Delta \leftarrow \infty$
7      Initialize intersection points $s_i, i = 1, 2, \ldots, N$
8      while $\Delta > \epsilon_N$ do
9          for $1 \leq i \leq N$ || do
10             $planeIndex \leftarrow path[i]$
11             $\omega \leftarrow \Omega[planeIndex]$
12             $s_i^{new} \leftarrow f_{IntersectionPoint}(s_{i-1}, s_{i+1}; \omega)$
13             $\Delta \leftarrow \Delta + ||s_i^{new} - s_i||$
14             $s_i \leftarrow s_i^{new}$
15         $\Delta \leftarrow \frac{1}{N}\Delta$
16     if $\forall i \in N, s_i \in \Omega[path[i]]$ then
17         $Q \leftarrow Q \cup \{s_0, s_1, ..., s_N, s_{N+1}\}$
18
19 return $Q$
```

Algorithm 2: Compute Ray-Plane Intersection Point $f_{IntersectionPoint}$

```
Input: Leading and following points $s_{n-1}$ and
       $s_{n+1}$,respectively; target reflection plane $\Omega_n$
Output: Ray-object intersection point ($s_n$) on the given plane
        $\Omega_n$
1 $t_{n-1} \leftarrow$ Projection of $s_{n-1}$ on $\Omega_n$
2 $t_{n+1} \leftarrow$ Projection of $s_{n+1}$ on $\Omega_n$
3 $m_n \leftarrow ||s_{n-1} - t_{n-1}||$
4 $n_n \leftarrow ||s_{n+1} - t_{n+1}||$
5 $l_n \leftarrow ||t_{n+1} - t_{n-1}||$
6 $e_l \leftarrow \frac{t_{n+1} - t_{n-1}}{l_n}$
7 $s_n \leftarrow t_{n-1} + l_n \frac{m_n}{m_n + n_n} e_l$
8 return $s_n$
```

The IPC algorithm traverses all potential ray paths, where each path consists of the index numbers of the planes involved in the reflection sorted by the order of the reflection. On the reflection path $P_i, i = 1, 2, \ldots,$ the first point s_0 and last point s_{N+1} are fixed TX and RX coordinates, respectively, and the remaining points $s_i, i = 1, 2, \ldots, N$ correspond to ray-plane intersections on each plane involved in the reflection. Iterations are performed after initializing each point. The stopping rule ensures that the error of the coordinates calculated for each intersection points finally converges to ϵ_N. Calculating and updating the new coordinates of the intersection points on each reflection plane is implemented in parallel, without any demand on the order of reflections. The function $f_{\text{IntersectionPoint}}$ calculates new coordinates of the intersection points on the given reflection plane by the leading and following points, as shown in Algorithm 2 and (10)–(13). The error between the newly computed coordinates and the old coordinates for each point is accumulated and averaged for fitting the iterative stopping rule indicated by ϵ_N. After the ray path has converged, it is necessary to ensure that each intersection point is within the boundary of the corresponding plane in order to determine the validity of the given potential path. All potential paths given by the list P are checked for validity and further recorded by the list Q only for confirmed paths, which is returned for subsequent processing. As an example to demonstrate the convergence of iterative paths, a third-order

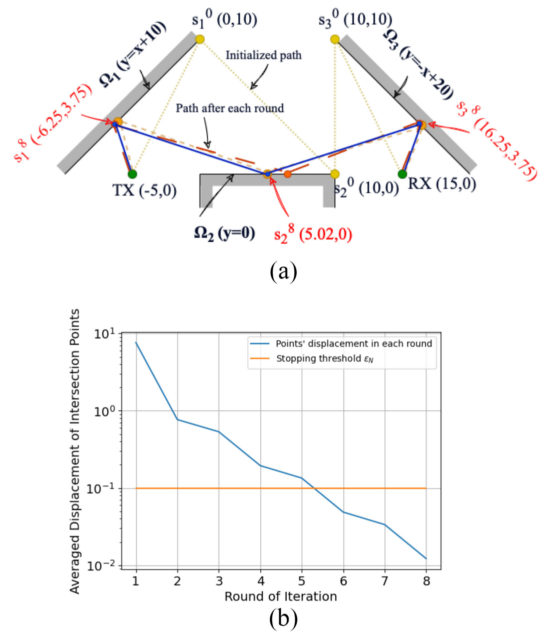

Fig. 4. Example for IPC algorithm: With fixed TX and RX the reflection paths will gradually converged: clearly the intersection converges as the iterations proceed. (a) Reflection path derivation using IPC. (b) Convergence in computing intersection points.

reflection path is shown in Fig. 4. The purpose of the IPC is to derive the coordinates of the ray-plane intersections in each of the planes based on an ordered sequence of reflection planes (here is TX $\rightarrow \{\Omega_1 \rightarrow \Omega_2 \rightarrow \Omega_3\} \rightarrow$ RX) in the space connecting the TX and RX. After each intersection has been initialized, a round of path iteration is launched: the intersection point s_1^0 on the plane Ω_1 is updated to a new value s_1^1 based on the current coordinates of TX and s_2^0, while at the same time s_2^0 is computed to an updated value s_2^1 based on s_1^0 and s_3^0, and s_3^0 is updated to a new value s_3^1 based on s_2^0 and RX. The average displacement of each intersection point in this path does not satisfy the iteration convergence threshold $\epsilon_N = 0.1$ at this point, and thus the next round of iteration is launched. The exact reflection path and its nodes coordinates are obtained after five rounds of iterations, as shown in Fig. 4.

The IPC algorithm discussed above allows the forward serial path derivation procedure to be fragmented by disassembling one given potential reflection path into independent geometrical calculation $f_{\text{IntersectionPoint}}$ for each planes involved in the reflection, which allows the derivation of the path to break through the limitation of the order of the ray-plane intersection points, and thus allows the path to be computed in parallel until the convergence is achieved.

III. Validation and Performance of IPC Algorithm

In order to validate the proposed IPC algorithm, this section uses software to compare and analyze the performance of the proposed IPC algorithm and the widely used SBR algorithm in ray tracing in terms of computational accuracy and speed, respectively. In order to simplify the validation process, this section uses 2-D closed contours regarded as the top views of indoor scenarios as the simulation environments for the validation of path derivation; meanwhile, in order to ensure that the validation results are generalizable, 100 sets of randomly generated closed contours of different shapes are tested which makes to cover the L-, T-, U- and convex-polygon-shaped rooms in a (numerically) 10 × 10 square-shaped environment, which is used to imitate the common room shapes in real-world. The configurations is shown in Table I.

In addition, one vertex in the closed contour is freely selected so as to make it move according to a certain trajectory in time steps, which is equivalent to the attitude change of the two edges adjacent to this vertex and is used to imitate the dynamics of the environment, as shown in Fig. 5. The transmitter TX and receiver RX are randomly placed within the generated 2-D closed contour and the reflection paths are computed, respectively, using the proposed IPC algorithm and SBR algorithm, which are realized in software using Python. The performance of the respective algorithms will be compared in terms of accuracy and speed of path derivation (only the case of 3rd-order or lower reflections are considered due to the high-reflection loss for RF signals in GHz-band) as

TABLE I
Configurations of Generated Shapes

Shape	Edges Number	Count
L	4-6	25
T	8-10	25
U	8-10	25
Convex polygon	4-10	25

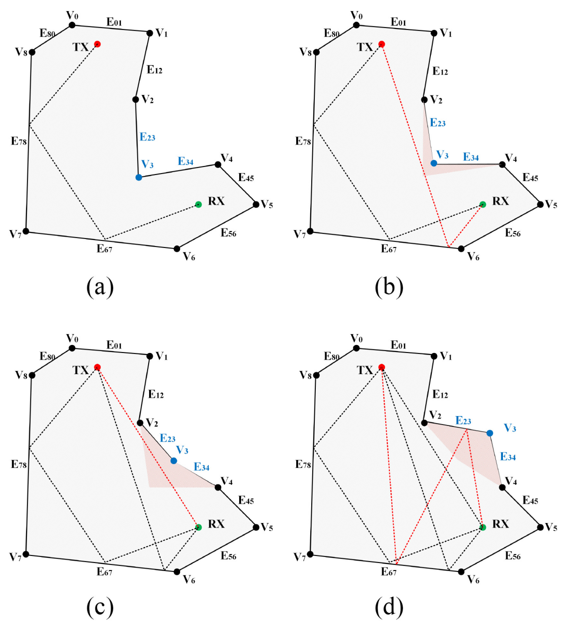

Fig. 5. L-shaped room with dynamic slide walls captured in four time steps: Dynamic sidewalls E_{23} and E_{34} are affected by the dynamic vertex V_3, which expands a new area (marked with a red shaded region) for signal propagation at each time step; newly established signal propagation link at each time step is denoted by red dotted lines. (a) Time step 0: Original signal propagation pattern. (b) Time step 1: Environment changing leads to a new first-order reflection path in ray tracing. (c) Time step 2: Environment changing leads to a new line-of-sight path in ray tracing. (d) Time step 3: Environment changing leads to a new second-order reflection path in ray tracing.

TABLE II
SOFTWARE SETTINGS FOR VALIDATIONS

CPU	Intel Core i7-8750H
Python	3.11.7 (64-bit)
Numpy	1.26.3
Open3D [14]	0.18.0
Logic Synthesis	Quartus II 11.0 on Windows 10 22H2

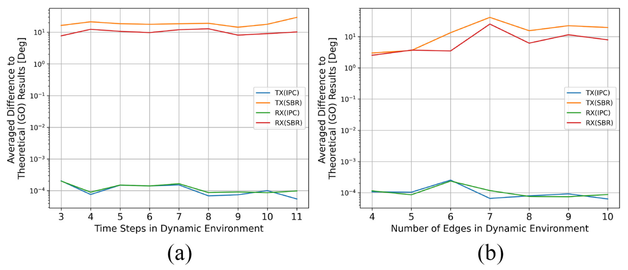

Fig. 6. Computational errors in ray paths derivation: Angular errors on TX and RX sides are depicted, respectively, to (a) dynamics of environment and (b) complexity of environment. Accuracy of ray tracing via IPC algorithm is compared with the results obtained by SBR algorithm with one-degree sweeping interval.

indicated by the time step (i.e., dynamics of the scene) and the number of edges of the closed contour (i.e., complexity of the scene), respectively. The software settings are shown in Table II.

A. Accuracy in Path Derivation

The accuracy of the reflection path computation is expressed in terms of the angular error between the direction of the ray computed by the respective algorithms at TX and the theoretical ray direction computed using GOs, in order to avoid the ambiguity associated with using actual units (e.g., meters) to describe the coordinates of ray-plane intersections and the corresponding displacement errors.

As can be seen in Fig. 6(a), the angular error obtained using the SBR algorithm with one-degree sweeping interval is about 10 degrees, whereas the result obtained by convergence in the proposed IPC algorithm can be improved by five orders of magnitude in terms of accuracy, which fully demonstrates the advantages of the proposed IPC algorithm. Besides, the simulation accuracy of the IPC algorithm does not vary widely with the dynamics and complexity of the environment, which is robuster compared to the SBR algorithm that has an angular error ranges from 3 degrees to 30 degrees, as shown in Fig. 6(b).

B. Ray-Tracing Simulation Speed

For the convergence speed of the computation, the advantages of the proposed IPC algorithm can be clearly demonstrated in Fig. 7(a) and (b): by obtaining the potential reflection paths in advance, the IPC algorithm can improve the ray-tracing speed by more than two orders of magnitude compared to the SBR algorithm; although the IPC algorithm is not accelerated in parallel (due to software implementation using Python), it still achieves milliseconds of elapsed time in the software ray-tracing implementation. It can be noticed that as the environment becomes more complex (more edges)

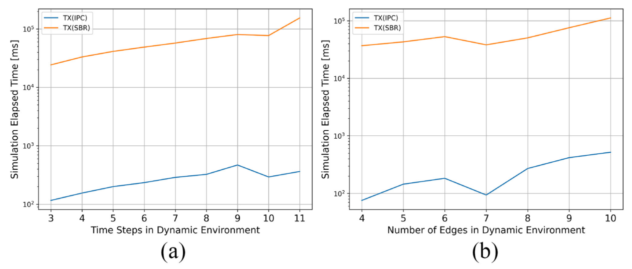

Fig. 7. Elapsed time in ray paths computation: The software-implemented IPC algorithm achieves a speedup of about 200 times compared to the SBR algorithm for ray tracing within 2-D closed contours. (a) Ray-tracing elapsed time to the dynamics of environment. (b) Ray-tracing elapsed time to the complexity of environment.

the ray tracing's elapsed time slowly increases. When the number of edges is low, the L- and convex-polygon-shaped rooms dominate. When the number of edges grows to seven there is a significant speedup of the ray tracing due to the fact that it is at the boundary between the number of edges of the L-shaped and the T/U-shaped, so that convex-polygon-shaped rooms dominate, which in turn makes the computation of reflection paths easier; as the number of edges continues to grow, the T/U-shaped rooms gradually dominate, which makes the ray-tracing environment complex and thus leads to a further increase in the elapsed time.

From the above analysis and comparison of the performance of the IPC algorithm and the SBR algorithm for ray tracing in 2-D scenes, it can be concluded that the proposed IPC algorithm compresses the ray-tracing elapsed time in milliseconds while maintaining high-computational accuracy, which shows the expectation and feasibility of further accelerating the IPC algorithm using FPGAs.

IV. HARDWARE ARCHITECTURE OF RAY-TRACING PLATFORM ON FPGAS

The IPC algorithm proposed in this article can greatly accelerate the ray path convergence process with the help of computational parallelization. If the IPC algorithm is implemented using an FPGA, the parallel subroutines in the algorithm can be mapped to the intrinsic parallelism of the logic circuits, giving full play to the advantages of FPGAs: compared to the batch-based parallel acceleration of GPUs, FPGAs are more flexible solutions due to the high degree of freedom in functional implementation and efficient tradeoffs between resource utilization and timing constraints. For example, when the timing constraints of a ray-tracing platform for virtual digital twins of wireless devices are violated, i.e., the ray tracing does not converge fast enough to meet the design requirements, the ray tracing can be physically accelerated by adding logic function modules to share the computational efforts. This approach is not available to GPUs, which demonstrates the unique freedom of implementation provided by FPGAs.

The hardware architecture adapted to the IPC algorithm implemented on FPGA proposed in this article is shown in Fig. 8. The ray-tracing platform implemented in FPGA consists of three main modules: 1) the PE matrix; 2) the PE scheduler; and 3) the coordinates updater-buffer. In this

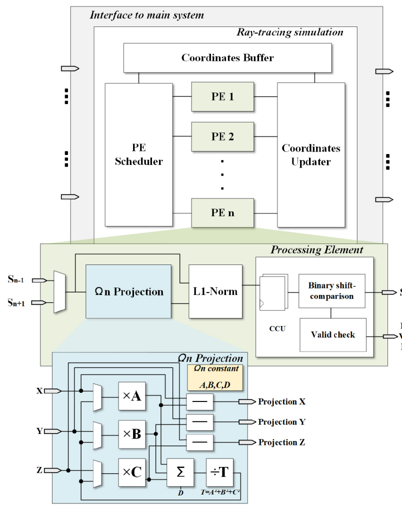

Fig. 8. Module architecture of the real-time ray-tracing platform on FPGA.

Algorithm 3: Halving-Comparison in Proportion Operations of (13)

```
   Input: $m_n, n_n, l_n$in (13)
   Output: Proportion $l_n \frac{m_n}{m_n+n_n}$
1  $L \leftarrow (m_n + n_n)/2$; $B \leftarrow L$
2  $L_p \leftarrow l_n/2$; $B_p \leftarrow L_p$
3  while Ł > 0 do
4      $L \leftarrow L/2$; $L_p \leftarrow L_p/2$
5      if $B < m_n$ then
6          $B \leftarrow B + L$; $B_p \leftarrow B_p + L_p$
7      else
8          $B \leftarrow B - L$; $B_p \leftarrow B_p - L_p$
9
10 return $B_p$ as $l_n \frac{m_n}{m_n+n_n}$
```

case, the PE matrix consists of various different mutually independent PEs related to the individual planes participating in the reflection in the target environment. The function of each PE corresponds to $f_{\text{intersectionPoint}}$ in the IPC algorithm, as shown in Algorithm 2, and is used to compute the coordinates of the ray-plane intersections in a given reflection plane from the leading and following points. As a result of the geometric preprocessing, the reflection planes in the scene are extracted with critical information, such as plane equations, normal vectors, and boundary points, which can be considered as constants during the ray-tracing process, thereby reducing the size of the circuit. Thus, geometric operations in PE, such as calculating vector projections, can be physically realized by combinational blocks or small-scale sequential blocks: multiplication operations for constants can be realized using the shift-add principle, while division/proportion operations can be realized using halving-comparison in order to balance the size of circuits and the speed of computation, as shown in Algorithm 3. In Algorithm 3 the denominator $m_n + n_n$ is fitted to the numerator m_n by halving successively in iterations, and by mapping the same action to the coefficients l_n the displacement of the intersection point with respect to the projection point of the preceding point on the plane is obtained as $l_n(m_n/[m_n + n_n])$, and then the actual coordinates of the intersection point can be computed according to (13). As only addition, subtraction and shift operations are involved, it can be easily converted into sequential logic circuit implemented in each PE.

According to the previously mentioned, if all potential reflection paths obtained based on geometrical preprocessing of the target scene can be rationally used for scheduling the PE matrix, the ray-tracing simulation can be accomplished in the shortest possible elapsed time. Hence, the PE scheduler uses a finite state machine or ROM to save the derived scheduling sequence by which the PE matrix is scheduled and manipulated. The coordinates updater-buffer buffers the newly calculated intermediate coordinates in each round of path iteration so that it can be used in the next round of the IPC iteration. As mentioned earlier, the purpose of our proposed embedded real-time ray tracing is to quickly and accurately compute the reflection paths that contribute the most to signal propagation, which will guide the beamformer for efficient beam management. Therefore, instead of deriving all possible reflection paths completely, we selectively buffer the intermediate values in path iterations according to the following principles.

1) *Priority of Reflection Order:* First order reflections are prioritized over second order, and so forth.
2) *Priority of Path Lengths:* Only one intersection per reflection plane is buffered, which indicates the best reflection path through this plane.

Based on the above principles, an upper limit of FIFOs can be set in the HLS workflow, i.e., only storing coordinates up to the number of reflection planes, which can greatly reduce the amount of memory/registers and make the final on-chip resource utilization greatly reduced. For example, for a target environment with nine reflection planes as shown in Fig. 5, the reflection needs to be buffered up to 1665 coordinates (9 first-order, 72 second-order, and 504 third-order intersection points), whereas if only the intersection of the best reflection paths involved is selected for each reflection plane then only 9 coordinates should be buffered, which is equivalent to releasing 99.46% of the FIFO capacity and makes the on-chip performance better.

The ray-tracing platform implemented on FPGA based on the IPC algorithm can be obtained by instantiating the PE matrix, PE scheduler and coordinates updater-buffer, which pursues a further parallelization and acceleration for the reflection path derivation. As stated earlier, FPGA-based real-time ray tracing is deployed to provide the beam management with a fast response to dynamic environmental changes more than providing a complete and accurate simulation of signal propagation, with the runtime workflow shown in Fig. 9. When sensors (e.g., depth camera/LIDAR) detect a change in the environment, the ray-tracing wrapper in the embedded wireless device activates the real-time FPGA-based ray-tracing platform to launch ray-tracing simulations for the instantaneous scene, and the results are used to assist further in beam management for robust beam- forming and tracing. Applying the proposed IPC algorithm and parallelizing the mapping of the algorithm using FPGAs makes it possible to reduce the ray-tracing time to milliseconds (e.g., about 1 ms as shown

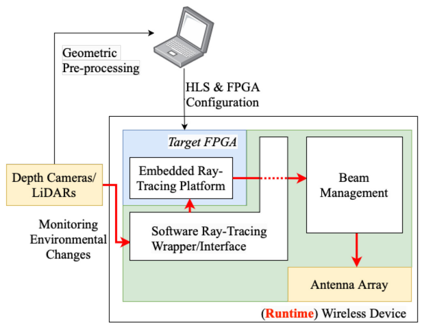
Geometric Pre-processing
HLS & FPGA Configuration
Target FPGA
Embedded Ray-Tracing Platform
Beam Management
Depth Cameras/ LiDARs
Monitoring Environmental Changes
Software Ray-Tracing Wrapper/Interface
Antenna Array
(Runtime) Wireless Device

Fig. 9. Runtime workflow of embedded wireless device with FPGA-based real-time ray-tracing platform.

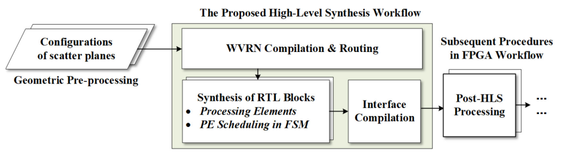
The Proposed High-Level Synthesis Workflow
Configurations of scatter planes
Geometric Pre-processing
WVRN Compilation & Routing
Synthesis of RTL Blocks
Processing Elements
PE Scheduling in FSM
Interface Compilation
Post-HLS Processing
Subsequent Procedures in FPGA Workflow

Fig. 10. Overview of the proposed HLS workflow for real-time ray tracing.

in Section VI), which allows the beamformer to achieve an instantaneous response to the environmental changes, thereby effectively improving the robustness of the beam management.

V. High-Level Synthesis Workflow

Aiming to automate the process from the acquired 3-D PCD of the target scene to the final RTL modules, so that the resulting FPGA implementation has high efficiency and flexibility, this article designs a novel HLS workflow for the hardware architecture of the proposed ray-tracing platform on FPGAs. The overall workflow is shown in Fig. 10.

After geometric preprocessing of the captured 3-D PCD successfully extracted critical information (e.g., plane equations, normal vectors, and boundaries, etc.) for the main planes that may be involved in signal propagation, the HLS workflow begins. The first step is to derive all potential reflection paths at each time step in the given dynamic scene: by using the Simple Funnel algorithm [15], [16], [17] on the cross section of the 3-D target scene in the direction perpendicular to the ground to extract the interplane weak visibility relations, a WVRN can be generated, as shown in Fig. 11. In the WVRN, if two nodes are linked by an edge, which indicates that the planes corresponding to these two nodes are weakly visible, i.e., each of the two planes has at least one pair of points that are directly visible (Line of sight, LOS), then it is possible that there is a reflection path between these two planes denoted by the nodes. All potential reflection paths are derived separately by time step by routing over the WVRN using A*-algorithm [18], leading to the final scheduling sequence used for the control of PEs. The PE scheduling problem is similar to the open-job shop scheduling problem (OSSP) [19] in that a definite number of jobs (i.e., derived potential paths in WVRN) are assigned time slots to a known number of machines (i.e., PEs) for the sequences of operations (i.e., PEs in the iteration) [20] with same computation time of each path calculation, which can be achieved by genetic algorithm (GA) [21] to perform heuristic search for a balance between the complexity of the algorithm and success rate of the PE scheduling, as shown in Fig. 12

$$\mathrm{LF} \overset{\text{def}}{=} \sum_{i=1}^{n} |T_{\text{latest}}(\mathrm{Path}_i) - T_{\text{earliest}}(\mathrm{Path}_i)|. \tag{14}$$

The objective of PE scheduling is to minimize the latency factors (LFs) of different PEs in the same path while achieving the shortest makespan, as defined in (14). That is, it is required that each PE operation involved in iterating the same path should be completed in the same time slot (or the nearest time slots) to achieve the maximum parallelization of the distributed PEs. Iteration can be accelerated by duplicating highly utilized PEs, but at the cost of increased on-chip resource utilization, which is not a critical factor for performance in our proposed FPGA implementation (see Section VI). Hence, a three-pass scheduling procedure is applied: 1) initializing the scheduling sequence by GA-algorithm; 2) applying PE duplications to expand the “room” for the parallelization of path computation; and 3) by ASAP-scheduling the blank timeslots are scheduled to achieve shorter makespan.

After compiling and routing the WVRN, the potential reflection paths and the critical information about the reflection planes corresponding to each path are obtained, which are used to, respectively, generate the PE scheduler and PEs, as discussed in the previous section. Finally, input and output interfaces are added to the generated ray-tracing platform to accommodate the embedded main system and system bus, thus realizing the eventual ray-tracing platform in RTL modules for further logic synthesis and subsequent processing for the final FPGA configuration. The proposed HLS workflow allows the target scene to be hardwarized on FPGAs in the form of FPGA modules aiming at fast reflection path derivation in milliseconds. As shown in Fig. 9, although the processing of the target scene using the HLS workflow to generate a scene-specific FPGA ray-tracing platform imposes a higher overhead compared to the accelerator using software program (e.g., GPU-accelerated ray tracing, etc.), the resulted FPGA-based real-time ray-tracing platform has better performance at runtime (see Section VI). Once the system is (re)configured, high speed and high-accuracy real-time ray tracing can be achieved at runtime, which is not affected by the overhead of the (re)configuration process.

VI. Performance of Generated FPGA Platform for Ray Tracing

In this section the ray-tracing platform on FPGA via the proposed HLS workflow for a given indoor scene is validated, an experimental platform as shown in Fig. 13 is constructed. The augmented ICL-NUIM dataset [22], [23] as shown in Fig. 14 is used as the target scene for ray tracing to compare the performances of different algorithms (SBR and proposed IPC) and platforms (multicore CPU and FPGA). The preprocessing program filters out the planes with an area larger than a quarter of one square meter for ray tracing, while the rest of the small-area planes will be

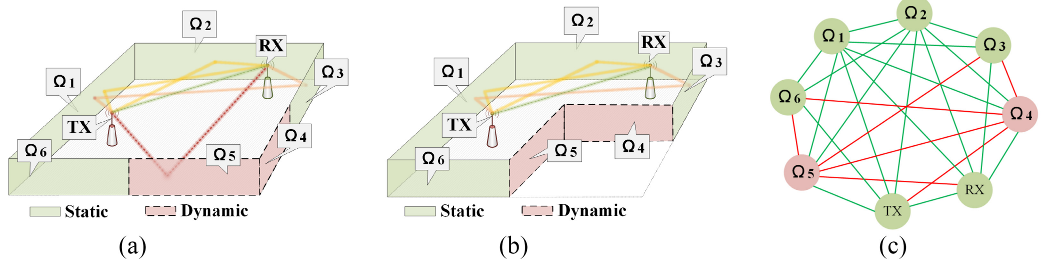

Fig. 11. Deriving dynamic weak visibility relationship networks (WVRN) from 3-D scene. (a) Original 3-D dynamic scene: Side walls Ω_1, Ω_2, Ω_3 and Ω_6 work as static scatters whereas Ω_4 and Ω_5 as dynamic scatters. (b) Dynamic scatters Ω_4 and Ω_5 move to new positions, which leads to environmental changes in next frame. (c) Corresponding dynamic WVRN, where the red edges indicate weak visibility relationships (WVR) that change with the environment while green edges represent static WVR.

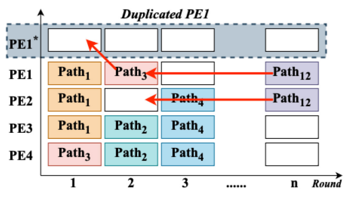

Fig. 12. PE scheduling: Aims to maximize parallelization for PE matrix. Makespan can be effectively reduced by duplicating PEs that have a high-utilization frequency: duplicated $PE1^*$ makes the calculation of $Path3$ executed without latency, which further guides $Path12$ to be scheduled earlier and reduces the total makespan in path iterations.

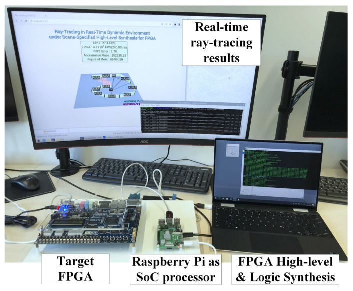
Real-time ray-tracing results
Target FPGA
Raspberry Pi as SoC processor
FPGA High-level & Logic Synthesis

Fig. 13. Hardware validation platform: the platform for real-time ray tracing is synthesized on PC and implemented on the target FPGA (Intel Cyclone IV E, EP4CE115F29C7). The Raspberry Pi is used as a post-processor for the rays extracted from the FPGA ray-tracing platform.

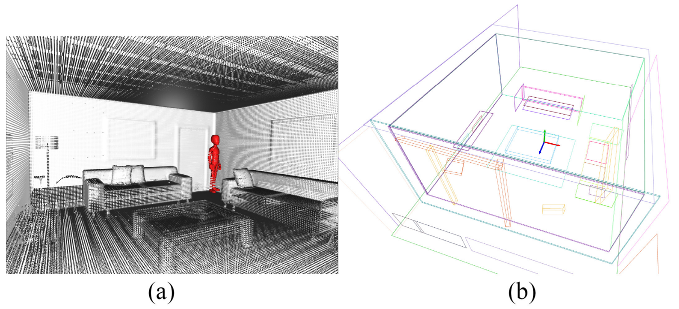

Fig. 14. Target scene of living room environment: (a) PCD of ICL-NUIM Dataset is used as static background environment where the introduced human body (in red) moves around to create a dynamic scene and (b) 25 extracted planes involved in signal reflection by the geometric preprocessing for further ray tracing.

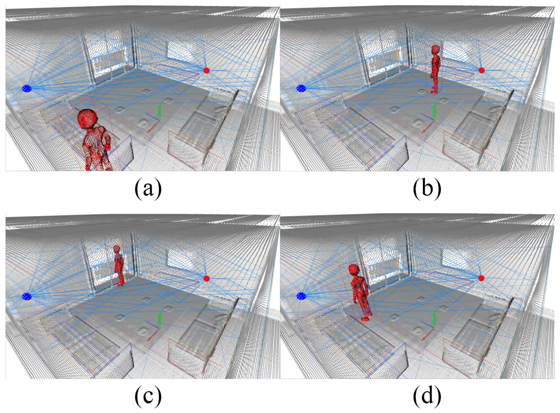

Fig. 15. Ray tracing in indoor scene with dynamic blockage (human body): The environment changes dynamically as the human body moves, while the TX (red point) and RX (blue point) are fixed in position to depict the influence of environmental change on signal propagation via ray tracing. (a) Time step 1. (b) Time step 5. (c) Time step 8. (d) Time step 9.

considered purely as blockages, which do not participate in the calculation of reflections. A moving human body is introduced in the static living room scene, which results in dynamic changes of the target environment. Four of the ten time steps sampled from the scene are shown in Fig. 15, which depict typical dynamic movement in position. As the geometric information of the reflection planes is extracted, the ray-tracing platform proposed in this article is implemented on FPGA according to the mentioned workflow. In order to fully demonstrate the performance of the IPC algorithm and the corresponding FPGA acceleration, the optimization approach for FIFO utilization mentioned in Section IV is not used in this section, i.e., iterative computation of ALL possible reflection paths on FPGAs using the IPC algorithm is performed in order to demonstrate the upper bound of accuracy and the lower bound of speed of the algorithm's implementation on FPGAs. Besides, the SBR algorithm is transplanted into rational RTL modules implemented on the same FPGA, the functional block diagram of which is shown in Fig. 16. Both of the hardware implementations are achieved by 32-bits word length in arithmetic calculations. In order to make floating-point geometric operations efficiently implementable on FPGAs, the coordinate values in which the operations are performed in FPGAs are all scaled up by a factor of one thousand, which corresponds to a coordinate precision of millimeters. Meanwhile, the ray-tracing simulation for the same scenario is accelerated on a PC using multicore CPU (32-bits float point numbers in arithmetic

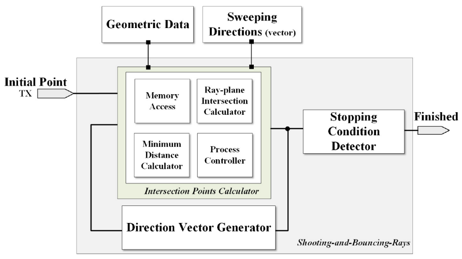
Geometric Data
Sweeping Directions (vector)
Initial Point
TX
Memory Access
Ray-plane Intersection Calculator
Minimum Distance Calculator
Process Controller
Intersection Points Calculator
Stopping Condition Detector
Finished
Direction Vector Generator
Shooting-and-Bouncing-Rays

Fig. 16. SBR algorithm achieved on FPGA: The RTL module consists of an intersection points calculator, a direction vector generator, and a stopping condition detector. For the given initial point TX, the intersection calculator traverses all the planes in the scene, calculates the set of intersection points in the given direction, and selects the one with the shortest distance to be the starting point of the next segment of rays in the reflective path; the direction vector generator calculates the direction of the next segment of rays, and repeats the cycle until it reaches the stopping condition to complete the ray tracing in one sweeping direction. The ray tracing of the whole space is completed by traversing all the sweeping vector stored in the memory block. In order to avoid the use of IP cores provided by vendors, such as DSPs, which makes the comparison of on-chip resource utilization with the IPC implementation complicated, this section uses combinational logic to implement general-purpose multipliers and divisors, such as Wallace tree multipliers.

TABLE III
UTILIZATION OF ON-CHIP RESOURCES FOR IPC AND SBR IMPLEMENTATIONS ON FPGA

Intel Cyclone IV E (EP4CE115F29C7) with 50MHz System Clock		
On-chip Resources	IPC (Min.-Max.Utiliz.)	SBR
Total logic elements	4%-43%	12 %
Total combinational functions	4%-41%	12 %
Dedicated logic registers	2%-13%	<1 %

calculations) with hardware and software parameters as shown in Table II. By comparing the simulation results obtained by different algorithms implemented on different platforms, the performance of each implementation is demonstrated in terms of simulation accuracy and speed.

A. Ray-Tracing Accuracy

By comparing the simulation results of the ray directions on the TX and RX sides with the theoretical results, which have been calculated by GOs, the accuracy of the ray tracing is derived. In each time step the rays obtained by the simulations (IPC and SBR, respectively) and the theoretical results are calculated in vectors to obtain the angular errors, which are then averaged to obtain the performance of ray-tracing accuracy as shown in Fig. 17. It is clear that the use of the IPC algorithm can improve the accuracy of ray tracing implemented on FPGAs by a factor of more than 100 compared to the implementation of the SBR algorithm. More detailed comparisons are shown in Fig. 18. Demonstrating the ray direction by azimuth and elevation for a given time step on the TX and RX sides, the ray-tracing accuracy of the IPC and SBR algorithms implemented on the FPGA can be derived, respectively. It is clear that the IPC algorithm demonstrates advantages in terms of accuracy: the simulation results obtained by the IPC algorithm are all very close to the theoretical results, whereas the SBR algorithm spreads the rays around the theoretical results in clusters. Furthermore, when the ray directions are not gathered, the SBR algorithm tends

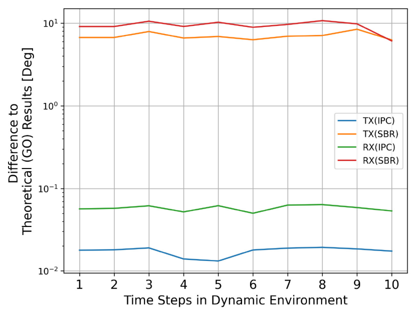

Fig. 17. Angular errors in direction of simulated rays on TX and RX sides: The FPGA-based IPC implementation has a significant advantage in terms of ray-tracing accuracy, as it can reduce the angular error with respect to the GO to less than 0.1 degrees.

to miss these independent directions, in which the signal can still propagate via reflection.

B. Ray-Tracing Speed

As can be concluded from Fig. 19, accelerating the ray-tracing algorithm using an FPGA has a clear advantage in simulation speed:

FPGA-based SBR implementation can increase the simulation speed by nearly 500 times relative to single-core CPU implementations, reducing the simulation elapsed time to around 200 ms. Nevertheless, the IPC algorithm is accelerated by the FPGA to compress the ray-tracing elapsed time to around 1 ms, which shows a much greater advantage in simulation speed. Similarly, the acceleration ratios shown in Fig. 20 also demonstrate the advantages of the IPC algorithm implemented on FPGA, which achieves a speedup of about 181 times compared to SBR algorithm implemented on FPGA. Such a large class of speedups is partly due to the proposed IPC algorithm: by decomposing the derivation of each path into mutually independent subroutines consisting of the respective planes in which the reflections are involved, it is possible to change the path iteration from a sequential process of derivation according to the order of reflection of each intersection point to parallel implementations, which accelerates the convergence of the paths; Besides the IPC algorithm is further accelerated using FPGAs: the PE matrix can well map the respective independent subroutines of the IPC algorithm, which in combination with rational PE scheduling can achieve timing optimization and thus further accelerate the ray-tracing procedure.

C. Discussions and Conclusion

From the above comparisons, it can be concluded that the implementation of the IPC algorithm on FPGA has a great advantage over PC simulators in terms of accuracy and speed of embedded real-time ray tracing for wireless communications. Moreover, the decrease in ray-tracing speed for more complex scenarios can be balanced by increasing the scales of the PE matrix, which gives the embedded system design a higher degree of freedom to make tradeoffs. Besides, refer to Table III resource utilization varies between 4%-43%, depending on the movement trajectory of dynamic objects. Forcing the logic synthesizer to generate PEs for all reflection

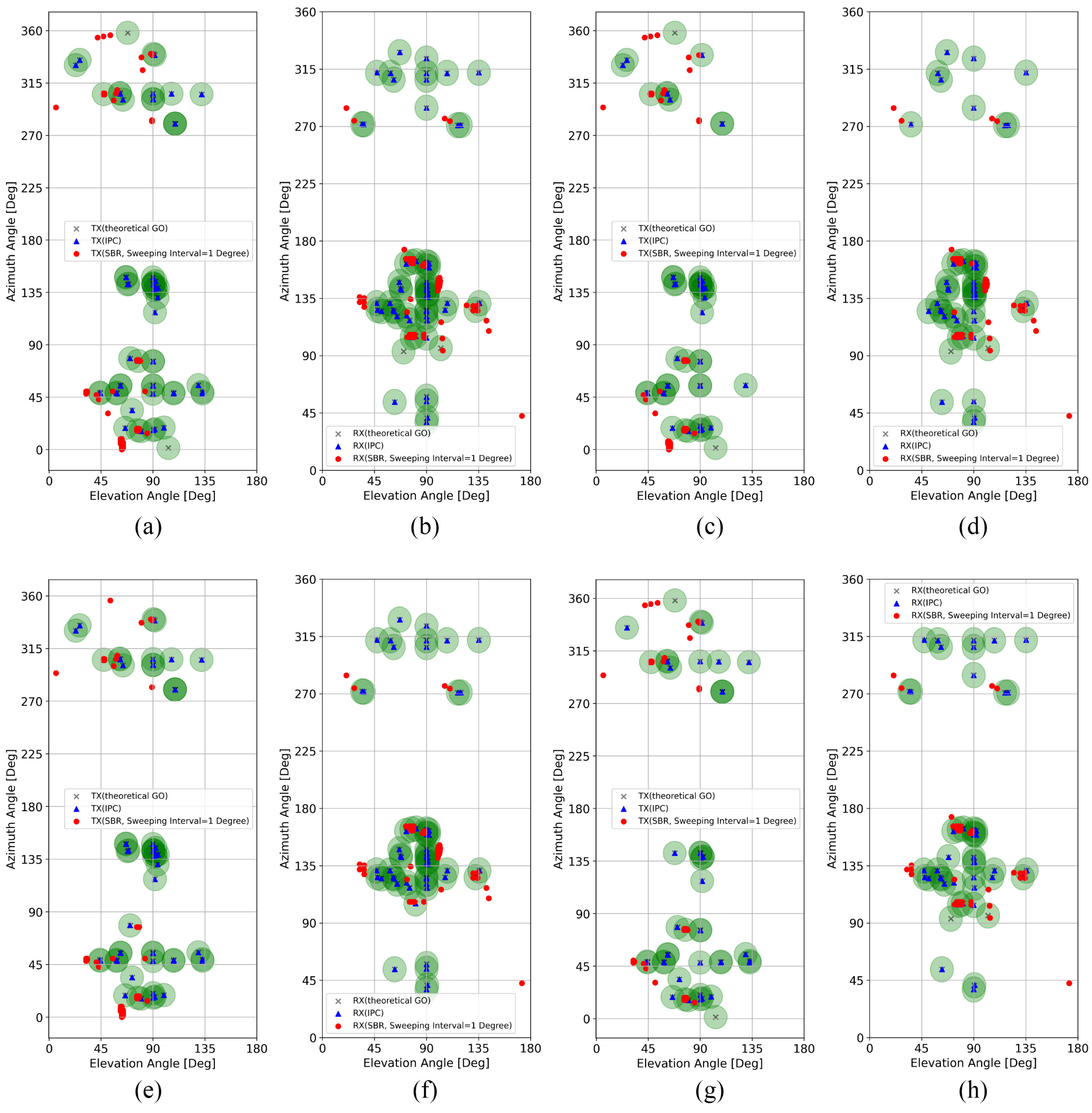

Fig. 18. Direction of rays: Black crosses indicate the theoretical values calculated by GOs theory, the blue triangles represent the results calculated by the proposed FPGA-based IPC implementation while the red circles indicate the results simulated by SBR algorithm implemented on same FPGA board; light green circle indicates the ten-degree tolerance centered on the result of the GO calculation. Although a few isolated rays are missed due to the quantization error (i.e., the error caused by using integers instead of floats for arithmetic operations) of FPGAs, it is clear that the results obtained using the IPC algorithm are very close to the theoretical results, which are all located within the corresponding tolerances. However, the SBR algorithm suffers from more significant errors: due to the ray-sweeping mechanism of the SBR algorithm, the resulting ray-tracing results are presented as a set of clusters of rays in the neighborhood of the theoretical results, which require further processing to acquire the unique true result; due to the accumulation of errors in the SBR algorithm caused by the forward serial operation according to the reflection orders, a large number of independent rays are mistakenly discarded, which needs a finer sweeping interval for compensation. (a) Time step 1, TX side. (b) Time step 1, RX side. (c) Time step 5, TX side. (d) Time step 5, RX side. (e) Time step 8, TX side. (f) Time step 8, RX side. (g) Time step 9, TX side. (h) Time step 9, RX side.

planes can lead to a final resource utilization of up to 43%, demonstrating the worst-case condition, which is four times that of SBR but still achieves a speedup of over a factor of hundreds for SBR, as shown in Fig. 20. Since the architecture of the ray-tracing platform proposed in this article is designed in register transfer level, the synthesized logic circuits can also be transplanted to the most advanced state-of-the-art FPGA chips for implementation toward greater optimization and better performance. Moreover, with the implementation flexibility of FPGAs tradeoffs can be made between ray-tracing accuracy, speed, and on-chip resource usage. As shown in Fig. 18, these isolated scattered rays depict the details of the signal propagation in the target scene compared to the concentrated paths, and thus the ray-tracing accuracy should be optimized in favor of the scattered-direction rays. An early stopping strategy should therefore be adopted for freezing the paths that show concentrated trend in the first few rounds, which is used to provide available PEs for the computation of the paths in the scattered directions. Similarly, the reduction in simulation speed and increase in resource utilization caused by the growing complexity of the target scenario can still be counterbalanced by dynamically adjusting the computational accuracy using early stopping strategy: freezing the computation of paths that clearly tend to be invalid (e.g., intersections are outside of the boundaries of the planes) saves PE-occupancy to be provided for rational path derivations, thus accelerating path convergence without loss of robustness.

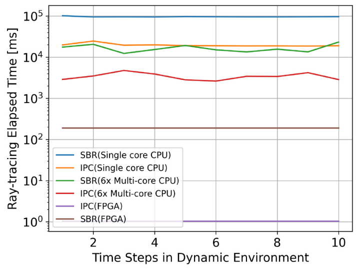

Fig. 19. Run time of ray-tracing simulation on different platforms: Generally speaking, the IPC algorithm is faster than the SBR algorithm implemented on the same platform. Accelerating the IPC algorithm using an FPGA can significantly speed up ray-tracing computation, resulting in simulation convergence time reduced to approximately 1 ms for each time step.

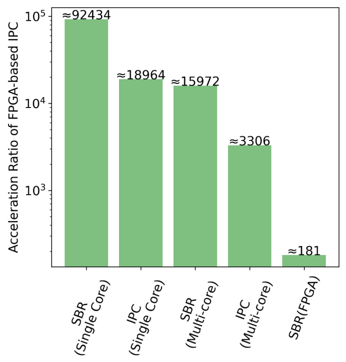

Fig. 20. Acceleration ratios of the proposed FPGA-based ray-tracing platform via IPC algorithm.

VII. Conclusion

In this article, the IPC algorithm for improving ray-tracing accuracy and facilitating parallelization on FPGA for acceleration is first proposed. The HLS workflow proposed in this article then obtains RTL-level modules that can be implemented on FPGAs by processing the geometric data of a given dynamic scene: compiling weak visibility relationship networks (WVRN) for each time step of the scene and routing in WVRN to obtain potential reflection paths; generating the IPC-based PEs and the corresponding control modules to achieve real-time ray-tracing platform on FPGAs. With the help of the IPC algorithm proposed in this article, the FPGA-implemented ray-tracing platform is validated to greatly parallelize the process of deriving the reflection paths, improve the ray-tracing speed while increasing the computational accuracy, and realize an assistance platform for environmental awareness of wireless communications demanding to complete the ray tracing in milliseconds. Although the ray-tracing accuracy and speed have been improved by FPGAs, there are still shortcomings, such as the geometric processing in the workflow, is very time-consuming and the resource utilization of RTL modules should further be optimized, etc., which will continue to be improved in subsequent work.

References

[1] W. Jiang, B. Han, M. A. Habibi, and H. D. Schotten, “The road towards 6G: A comprehensive survey,” *IEEE Open J. Commun. Soc.*, vol. 2, pp. 334–366, 2021.

[2] A. Ichkov, P. Mähönen, and L. Simić, “Is ray-tracing viable for millimeter-wave networking studies?” in *Proc. IEEE 31st Annu. Int. Symp. Pers., Indoor Mobile Radio Commun.*, 2020, pp. 1–7.

[3] A. W. Mbugua, Y. Chen, L. Raschkowski, L. Thiele, S. Jaeckel, and W. Fan, “Review on ray tracing channel simulation accuracy in sub-6 GHz outdoor deployment scenarios,” *IEEE Open J. Antennas Propag.*, vol. 2, pp. 22–37, 2021.

[4] M. U. Sheikh, R. Jäntti, and J. Hämäläinen, “Performance comparison of ray tracing and 3GPP street canyon model in microcellular environment,” in *Proc. 27th Int. Conf. Telecommun. (ICT)*, 2020, pp. 1–5.

[5] M. Pang, H. Wang, K. Lin, and H. Lin, “A GPU-based radio wave propagation prediction with progressive processing on point cloud,” *IEEE Antennas Wireless Propag. Lett.*, vol. 20, pp. 1078–1082, 2021.

[6] D. Shi, X. Tang, and C. Wang, “The acceleration of the shooting and bouncing ray tracing method on GPUs,” in *Proc. 31st Gener. Assem. Sci. Symp. Int. Union Radio Sci. (URSI GASS)*, 2017, pp. 1–3.

[7] S. Kasdorf, B. Troksa, C. Key, J. Harmon, S. Pasricha, and B. M. Notaroš, “Parallel GPU optimization of the shooting and bouncing ray tracing methodology for propagation modeling,” *IEEE Trans. Antennas Propag.*, vol. 72, no. 1, pp. 174–182, Jan. 2024.

[8] Z. Yun and M. F. Iskander, “Ray tracing for radio propagation modeling: Principles and applications,” *IEEE Access*, vol. 3, pp. 1089–1100, 2015.

[9] Z.-Y. Liu and L.-X. Guo, “A quasi three-dimensional ray tracing method based on the virtual source tree in urban microcellular environments,” *Prog. Electromagn. Res.*, vol. 118, pp. 397–414, 2011.

[10] S. Hu, L.-X. Guo, and Z.-Y. Liu, “A fast ray-tracing algorithm for rugged terrain,” in *Proc. Cross Strait Quad-Reg. Radio Sci. Wireless Technol. Conf. (CSQRWC)*, 2019, pp. 1–3.

[11] S. Hussain and C. Brennan, “An intra-visibility matrix based environment pre-processing for efficient ray tracing,” in *Proc. 11th Eur. Conf. Antennas Propag. (EUCAP)*, 2017, pp. 3520–3523.

[12] S. Hussain and C. Brennan, “Efficient preprocessed ray tracing for 5G mobile transmitter scenarios in urban microcellular environments,” *IEEE Trans. Antennas Propag.*, vol. 67, no. 5, pp. 3323–3333, May 2019.

[13] S. Hussain and C. Brennan, “A dynamic visibility algorithm for ray tracing in outdoor environments with moving transmitters and scatterers,” in *Proc. 14th Eur. Conf. Antennas Propag. (EuCAP)*, 2020, pp. 1–5.

[14] Q.-Y. Zhou, J. Park, and V. Koltun, “Open3-D: A modern library for 3-D data processing,” 2018, *arXiv:1801.09847*.

[15] S. Ghosh, *Visibility Algorithms in the Plane*. Cambridge, U.K.: Cambridge Univ. Press, 2007.

[16] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan, “Linear time algorithms for visibility and shortest path problems inside simple polygons,” in *Proc. 2nd Annu. Symp. Comput. Geomet.*, 1986, pp. 1–13.

[17] D.-T. Lee and F. P. Preparata, “Euclidean shortest paths in the presence of rectilinear barriers,” *Networks*, vol. 14, no. 3, pp. 393–410, 1984.

[18] M. R. Wayahdi, S. H. N. Ginting, and D. Syahputra, “Greedy, a-star, and Dijkstra’s algorithms in finding shortest path,” *Int. J. Adv. Data Inf. Syst.*, vol. 2, no. 1, pp. 45–52, Feb. 2021. [Online]. Available: http://ijadis.org/index.php/ijadis/article/view/greedy-a-star-and-dijkstras-algorithms-in-finding-shortest-path

[19] P. Brucker, *Scheduling Algorithms*. Berlin-Heidelberg, Germany: Springer, 2013.

[20] E. Anand and R. Panneerselvam, “Literature review of open shop scheduling problems,” *Intell. Inf. Manage.*, vol. 7, no. 1, p. 33, 2015.

[21] G. Campos Ciro, F. Dugardin, F. Yalaoui, and R. Kelly, “Open shop scheduling problem with a multi-skills resource constraint: A genetic algorithm and an ant colony optimisation approach,” *Int. J. Prod. Res.*, vol. 54, no. 16, pp. 4854–4881, 2016.

[22] S. Choi, Q.-Y. Zhou, and V. Koltun, “Robust reconstruction of indoor scenes,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2015, pp. 5556–5565.

[23] A. Handa, T. Whelan, J. McDonald, and A. Davison, “A benchmark for RGB-D visual Odometry, 3-D reconstruction and SLAM,” in *Proc. IEEE Int. Conf. Robot. Autom.*, Hong Kong, China, 2014, pp. 1524–1531.