UNIVERSITY OF OULU

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Niklas Vaara

TOOLS FOR RAY TRACING BASED RADIO CHANNEL MODELING AND SIMULATION

Master's Thesis
Degree Programme in Computer Science and Engineering
June 2022

Vaara N. (2022) Tools for Ray Tracing Based Radio Channel Modeling and Simulation. University of Oulu, Degree Programme in Computer Science and Engineering, 70 p.

ABSTRACT

Ray tracing-based methods have become the state of the art for radio channel propagation modeling simulations. They provide a way to deterministically simulate field strength and multidispersive characteristics of the radio channel, and thus, offer a faster and easier alternative to measuring. Ray tracing is also an important tool for validating algorithms, and many applications can utilize the simulation results. As the wireless networks suffer from increasing complexity, the interest in machine learning and artificial intelligence solutions is increasing as well, and in this context the simulation results can be utilized as training data.

We introduce the relevant theory in radio propagation modeling in the context of ray tracing, followed by theory of graphics processing unit-based computing, architecture, and ray tracing. We present multiple existing graphics processing unit and ray tracing-based radio channel propagation modeling implementations from the literature. We then develop multiple optimized versions of an existing environment discretization-based path search implementation and develop a path refiner for refining the coarse paths generated by the path search. The path refiner computes the optimal paths, and then validates them by utilizing ray tracing. Experiments for the developed solutions are conducted with an indoor and an outdoor model on two different computer setups. We achieve on average over 25 times faster computation in the outdoor scene and over 4 times faster computation in the indoor scene when compared to the original path search implementation. The path refiner is able to find the optimal paths fulfilling the Fermat's principle of least time on average for over 96% of the coarse paths in the outdoor scene, and for over 99% in the indoor scene. From these refined paths, on average about 62% pass the validation phase in the outdoor case, and around 30% in the indoor case. The results show that the path refinement combined with validation is essential for improving the quality of the paths found by the initial discretization-based search.

Keywords: graphics processing unit computing, ray launching, optimization, deterministic radio channel modeling.

Vaara N. (2022) Työkaluja säteenseurantaan perustuvaan radiokanavamallinnukseen ja simulointiin. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 70 s.

TIIVISTELMÄ

Säteenseurantaan perustuvat menetelmät ovat edistyneintä tekniikkaa radiokanavien etenemisen mallinnussimulaatioissa. Ne tarjoavat tavan deterministisesti arvioida radiokanavan kentänvoimakkuutta ja monidispersiivisiä ominaisuuksia ja siten tarjoavat nopeamman ja helpomman vaihtoehdon mittaamiselle. Säteenseuranta on myös tärkeä työkalu algoritmien validoinnissa ja useissa sovelluksissa voidaan hyödyntää simulointien tuloksia. Langattomien verkkojen monimutkaisuuden lisääntyessä myös kiinnostus koneoppimis- ja tekoälypohjaisiin ratkaisuihin lisääntyy, ja tässä yhteydessä simulointien tuloksia voidaan hyödyntää opetusdatana.

Tässä työssä esitellään teoriaa radiokanavan etenemisen mallinnuksesta säteenseurantaan perustuen, jonka jälkeen esitellään näytönohjainpohjaisen laskennan, arkkitehtuurin, sekä säteenseurannan teoriaa. Tämän jälkeen tarkastellaan useita olemassa olevia näytönohjain- ja säteenseurantapohjaisia radiokanavan etenemistä mallintavia toteutuksia. Työssä kehitetään useita optimoituja versioita olemassa olevasta ympäristön diskretisointiin perustuvasta polunetsintätoteutuksesta ja kehitetään poluntarkentaja tarkentamaan sen tuottamia epäoptimaalisia polkuja. Poluntarkentaja laskee optimaaliset polut ja validoi ne hyödyntämällä säteenseurantaa. Ratkaisuiden tehokkuutta arvioidaan sekä ulko- että sisätilan malleille tehtävillä laskennoilla kahdella eri tietokoneella. Paras polunetsintäversio saavuttaa keskimäärin yli 25 kertaa nopeamman laskennan ulkotilassa ja yli 4 kertaa nopeamman laskennan sisätilassa verrattaessa alkuperäiseen toteutukseen. Poluntarkentaja löytää optimaaliset polut, jotka täydentävät Fermat'n periaatteen lyhyimmästä ajasta keskimäärin yli 96 prosentille karkeista poluista ulkotilassa ja yli 99 prosentille sisätilassa. Näistä tarkennetuista poluista keskimäärin noin 62 prosenttia pääsee läpi validoinnista ulkotilassa ja noin 30 prosenttia sisätilassa. Tulokset osoittavat, että polkujen tarkennus ja validointi ovat tärkeitä alkuperäisen diskretisointipohjaisen haun löytämien polkujen laadun parantamiseksi.

Avainsanat: laskenta näytönohjaimella, säteenlaukaisu, optimointi, determinisitinen radiokanavan mallinnus.

TABLE OF CONTENTS

FOREWORD

This thesis was done at the Centre of Wireless Communications (CWC) - Radio Technologies. I would like to thank AI4Green and ARIADNE projects for the funding, as well as the 6G Flagship program. I would like to thank Prof. Markku Juntti for the opportunity to work in CWC - Radio Technologies, Dr. Pekka Sangi for the thesis supervision and technical assistance, and Lic. Tech. Juha Pyhtilä for the assistance with the theoretical and technical details.

Oulu, June 8th, 2022

Niklas Vaara

LIST OF ABBREVIATIONS AND SYMBOLS

AABB	axis-aligned bounding box
AI	artificial intelligence
API	application programming interface
BSP	binary space partitioning
BV	bounding volume
CPU	central processing unit
CUDA	Compute Unified Device Architecture
CWC	Centre of Wireless Communications
DED-RL	discrete environment-driven ray launching
EM	electromagnetic
GPGPU	general purpose computing on the graphics processing unit
GPU	graphics processing unit
GTD	geometrical theory of diffraction
LOS	line of sight
MIMO	multiple-input multiple-output
ML	machine learning
MT	multi-threaded
NLOS	non-line of sight
RL	ray launching
RT	ray tracing
RX	receiver
SDK	software development kit
SIMD	single instruction multiple data
SLVA	second-level visibility analysis
ST	single-threaded
TX	transmitter
UTD	uniform theory of diffraction
VANET	vehicular ad hoc networks
V2V	vehicle-to-vehicle

1. INTRODUCTION

Ray tracing (RT) provides a deterministic approach for radio channel modeling. It has been studied since the 1990's with the goal of fast computation and accurate field calculation [1]. RT-based implementations prove to be a reliable method for approximating features of the radio channel and they have many use cases such as helping in the design of beamforming and multiple-input multiple-output (MIMO) systems, where the performance depends on the multidispersive characteristics in polarization, time and space domains on top of signal-to-noise ratio [2, 3]. Vehicular ad hoc networks (VANET) greatly benefit from RT based radio channel modeling simulations as well. They provide a cheap and fast way to accurately simulate vehicle-to-vehicle (V2V) communication channel, which is essential as the safety-critical applications require widescale testing [4].

Recently, an interest in the area of artificial intelligence (AI) and machine learning (ML) for green and efficient communication networks has peaked as the large amount of traffic generated by the rapidly growing number of connected devices make effective monitoring and modeling of network traffic difficult [5]. This creates a new use case for RT-based radio channel propagation modeling simulations, as the results can be used as training data for ML models. ML requires a lot of training data, which is hard and time consuming to acquire through measurements. As a result, RT-based accurate simulations become an interesting application for this use case.

RT is computationally challenging due to the nature of tracing the rays. It might take multiple interactions before before a ray launched from the transmitter (TX) reaches a receiver (RX) in the scene. Each interaction might result in multiple new rays that have to be traced, which is why the maximum number of interactions is usually limited. Especially interaction types such as diffraction and scattering are very expensive computationally, as they result in many new rays. Angular discretization can be used to reduce the number of new rays caused by scattering or diffraction. Reducing the number of rays does not come without a cost, as it might result in important paths being missed. As each ray is traced independently, parallelism can be utilized to achieve faster execution speed.

The introduction of programmable graphics processing units (GPUs) has massively accelerated the computation speed of RT-based simulations due to their highly parallel nature [1]. Due to the architecture of GPUs, optimizations in the GPU side code might have a significant impact on the performance of RT-based algorithms. Even with GPU acceleration, RT-based solutions such as ray launching (RL) can still be extremely time consuming, thus, ways to reduce the computational complexity are needed.

The accuracy of the simulation is affected by multiple factors. Acceleration methods such as environment-based discretization simplify the RT process with the cost of decreasing the accuracy [6]. The accuracy is also determined by other factors such as the geometrical representation of the environment, and the electromagnetic (EM) material properties of the objects in the environment.

For ML applications, it is important to get the optimal paths that fulfill the Fermat's principle of least time to get as accurate training data as possible. To overcome the accuracy issues of acceleration methods such as discretization, a path refinement solution for correcting the paths has to be developed. As mentioned, ML applications require training data in large quantities, thus, the components involved in the data generation such as RT and path refinement should be optimized for fast execution speed.

In this thesis, we will optimize an existing discretization-based path search implementation. As the second task, we develop a path refinement solution to compute the optimal paths fulfilling the Fermat's principle of least time for the paths generated by the path search. In Chapter 2, theory and methods of radio channel propagation modeling in the context of RT are presented. Chapter 3 introduces GPU architecture and relevant GPU computing theory, as well as GPU-based implementations for radio channel propagation modeling. In Chapter 4, we develop optimized versions of the path search implementation, as well as the path refiner for computing the optimal paths. In Chapter 5, all of the path search versions are benchmarked and further experiments are conducted with the best performing version in an indoor and an outdoor scene. The path refinement experiments are conducted with the paths that are generated during the path search experiments. In Chapter 6, we discuss and analyze the results and findings, and present the future work ideas. In Chapter 7 we summarize what was done in this thesis.

2. RADIO CHANNEL PROPAGATION MODELING

Scientific exploration of EM waves started to progress remarkably when Hertz made the discoveries in 1880's. In wireless communications, the wireless channel is the carrier of the information, which is propagated through radio waves [1]. In this chapter we introduce the general theory behind the ray concept, ray-based methods in radio channel propagation modeling, as well as the EM properties that should be considered when tracing the rays.

2.1. Ray Tracing

Ray optic-based algorithms such as RT can be used to approximate properties of radio waves, as they solve the Maxwell's equations in the high frequency regime [1]. For electric and magnetic fields in the free space, the Maxwell's equations introduce the Faraday's law, the Ampere's law, and the Gauss's laws. For rays in the high frequency regime, [7] illustrates that the Fermat's principle of least time can be proven, as well as the Maxwell's equations.

2.1.1. Types of Ray Interactions

RT-based radio channel propagation modeling follows certain rules, such as the laws of reflection, diffraction and refraction, and that the ray travels in a straight line in a homogeneous medium [1].

Line of Sight

Line of sight (LOS) rays are known as the rays which have a direct connection between TX and RX without obstruction from the environment in the scene [1]. Based on the used RT method, LOS rays can be determined with low computational effort.

Reflection and Refraction

When a ray encounters any object with a different medium, it causes reflection and refraction [8]. The Snell's law defines the law of refraction as [9]

$$n_1 \sin\theta_1 = n_2 \sin\theta_2, \tag{1}$$

where the refractive indices n_1 and n_2 are defined based on the relevant medium. θ_1 defines the angle between surface normal and the incoming ray. The angle defined by θ_1 is also the same between the reflected ray and the surface normal. θ_2 defines the angle between surface normal and refracted angle. Illustration of the mentioned variables can be seen in Figure 1.

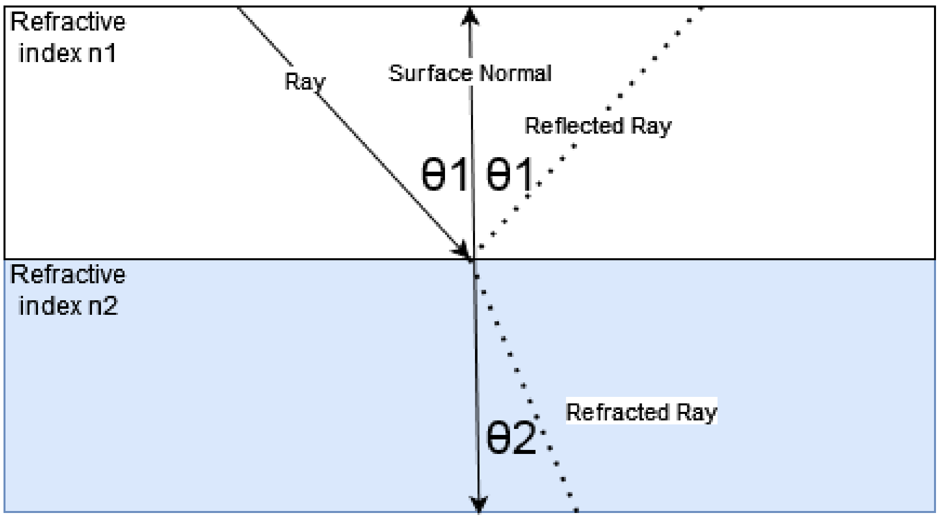

Figure 1. Reflection and refraction behavior.

Diffraction

In the geometrical theory of diffraction (GTD), Keller extends the geometrical optics theory to account for diffraction [10]. Diffraction happens for example, when a ray hits an edge. The diffracting rays are represented as a cone, as in Figure 2, which is determined by the law of edge diffraction. The law states that the angle between the incident and diffracting rays have an equal angle with the edge in the same medium. GTD was later refined in the uniform theory of diffraction (UTD) by Kouyoumjian & Pathak [11]. Many RT-based implementations for example, in [4, 6] have their diffraction coefficients based on the UTD.

Simulating diffractions is rather expensive, as the diffracting edges result in multiple new rays from the incident ray [1]. The complexity of diffractions does not end there, as the diffraction coefficients are computationally more expensive to calculate than the coefficients for the other interaction types. Due to the complexity, multiple methods for calculating the diffraction coefficients have been developed.

Yun & Iskander [1] present multiple methods for diffraction calculations, such as for lossy wedges [12], transparent wedges [13], lossy dielectric wedges [14], the uniform asymptotic theory [15], and a method for multiple straight wedge diffraction in 3D space [16].

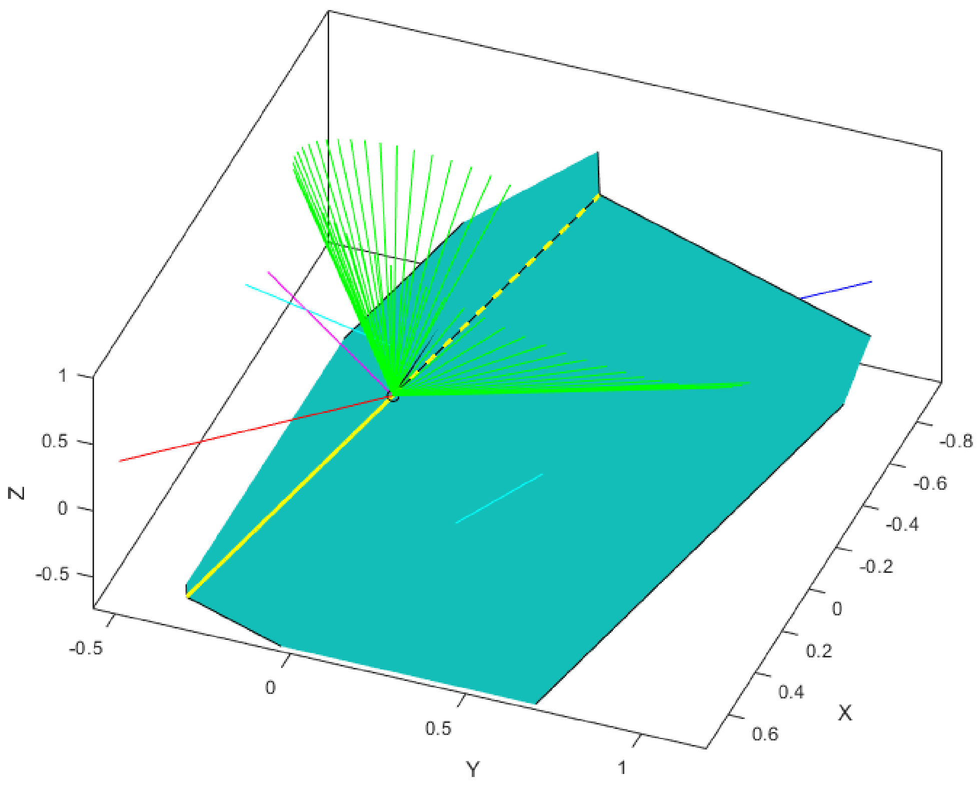

Figure 2. Sampled diffracting rays determined by the law of edge diffraction.

Scattering

Scattering happens when a ray encounters a rough surface. In [17], 60 GHz RT-based simulations were done, where the effect of rough surface in non-line of sight (NLOS) urban areas was deemed considerable. Thus, it is essential to take the roughness of surfaces into consideration when using RT-based simulations in order to achieve accurate results. For very rough surfaces the power scatters into all directions, meaning that the interaction cannot be represented as specular [17, 18]. Illustration of scattering can be seen in Figure 3.

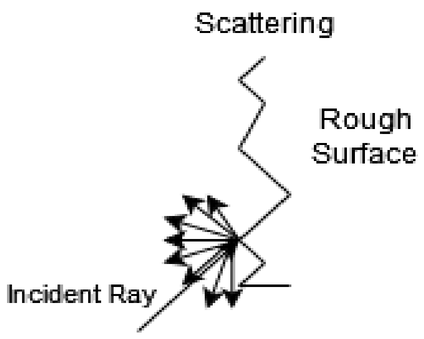

Figure 3. Scattering rays.

2.1.2. Ray Tracing Methods

RT is time consuming, as millions of rays might have to be traced in order to get adequate results. Multiple methods have been developed with varying computational complexity and accuracy.

Ray-based algorithms are often divided into RT and RL algorithms [19]. The RT methods, such as the image method, often revolve around figuring out valid optical paths between a single TX and RX. These optical path methods are not very optimal for coverage related simulations with multiple RXs, as each TX and RX pair has to be calculated separately.

Image method

In the image RT method, the optical path is solved by dividing the environment into components such as planes and edges [20]. The path between these components that reaches the RX is then searched while satisfying the requirements of the relevant laws, such as the Snell's law and the law of diffraction.

The image method can be used to retrieve the intersection point Q, presented in Figure 4. The images of RX and Ri are retrieved with respect to the reflecting plane, followed by forming a line between TX and Ri to determine the intersection point Q on the plane [1]. Diffraction points can be computed by solving systems of nonlinear equations, as shown in [21].

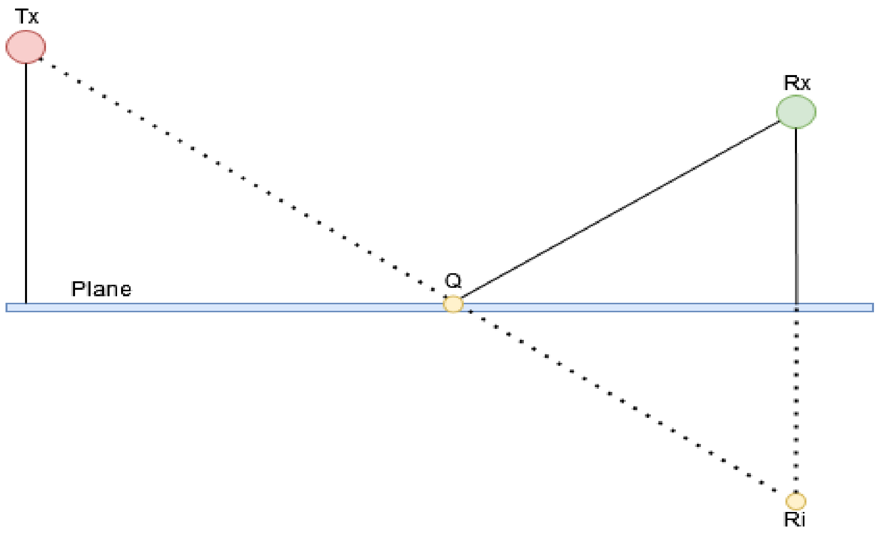

Figure 4. Reflection using the image method.

Ray launching

In RL, the rays are launched in the relevant directions from the TX with a small angular separation, and traced until they reach any RXs, or until the power threshold or the maximum number of interactions is reached. This makes RL suitable for coverage related simulations, as the computation time is not largely effected by the number of RXs in the scene [19]. Thus, RL can be used to gain adequate path loss estimates [1].

RL has its challenges, as the discretization of launched rays causes analytical errors due to the spatial resolution, which means that the paths cannot be considered optimal [20]. Ray splitting [22] was developed to combat the angular dispersion [23]. Each ray is split into four new rays when the spatial separation reaches a threshold. While ray splitting reduces the angular dispersion, it increases the computational complexity due to the additional rays that have to be traced. The RX sphere radius has to be chosen so that it captures the relevant rays, while also avoiding duplicate paths. A visualization of paths generated with a ray launcher can be seen in Figure 5.

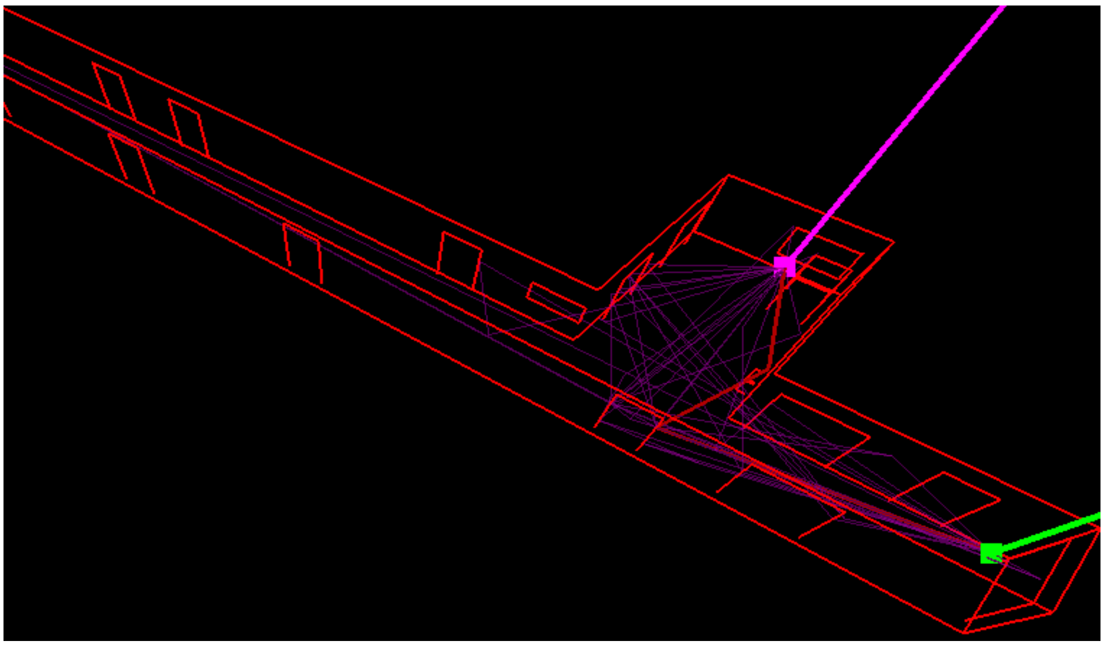
Figure 5. Paths generated with a ray launcher.

Combined method

Some implementations such as [24, 25] combine the RL and image methods. The valid paths are first determined using RL, followed by using the image method.

Determination of valid RL paths reduces the computational complexity of the image method, as the sequence, as well as the relevant planes and edges on the path are known [1]. In the combined approach, the overhead of applying the image method is insignificant, while resulting in improved accuracy of the paths.

2.1.3. Acceleration Methods

Visibility graph

The visibility graph method presented in [26] provides information of the visibilities between the objects in the scene. Methods such as the binary space partitioning (BSP) and bounding volume (BV) can be used to generate visibility graphs in high speed [20]. BSP divides the planes into subspaces sequentially. The division happens iteratively until all the planes are added to the tree. In BV, the scene is divided into multiple blocks of same width and depth, but the height of the block depends on the tallest object in the block.

Space division

Space division methods divide the objects in the scene into small cells. When a ray is being traced, the next cell can be determined by the neighboring information, which reduces the number of possible intersections in the scene. Multiple methods of space division exist, such as triangular and rectangular grid methods [1]. A 2D rectangular space division approach [27] presented in [20] reported 86% reduction in computational time compared to the visibility graph method.

Discretization is a form of space division, where the different elements, such as walls and streets are divided into cells to simplify the RT process with the cost of reduced accuracy of the simulation [6]. This method is used in papers such as [6, 28]. An example where the environment is discretized into tiles can be seen in Figure 6.

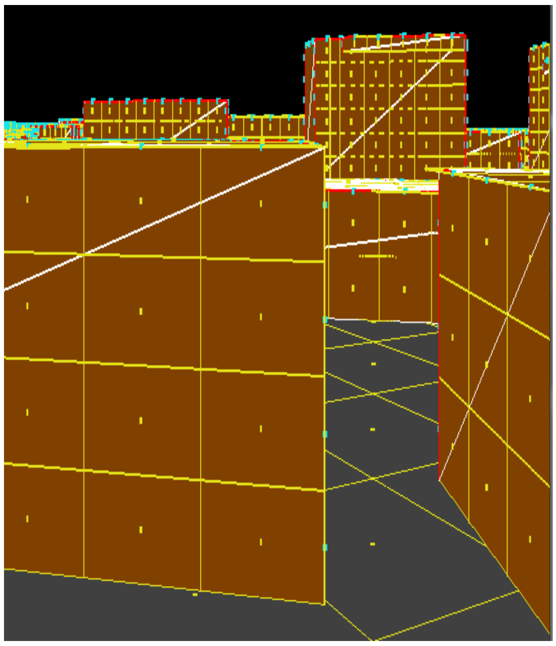
Figure 6. Discretized geometry.

Combined methods

Visibility graph and space division methods have been combined to achieve even better performance. In [28], which was published in 1999, the visibility relations between the center point of the discretized tiles and edges are determined to form a visibility graph in the form of a tree structure. Similar approach is still used in recent implementations, such as [6] published in 2018, where the visibility relations between the discretized elements are saved in a matrix. The latter implementation is explained in detail in Section 3.2.1.

2.2. Electromagnetic Properties

RT-based solutions for radio channel propagation modeling are used in different environments such as indoor [29, 30, 31, 32] and outdoor [4, 6, 33, 34], where the calculation models for the EM properties and the used RT-based algorithms differ.

2.2.1. Electromagnetic Field

Acquiring an approximation of the EM field with the given parameters is the main purpose of radio propagation modeling [1]. For free space radio propagation, the simplest model is known as the Friis equation [35]

$$\frac{P_\mathrm{r}}{P_\mathrm{t}} = G_\mathrm{t} G_\mathrm{r} \left(\frac{\lambda}{4\pi r}\right)^2, \tag{2}$$

where P_r is the received power, P_t is the transmitted power, G_t is the transmitter antenna gain, G_r is the receiver antenna gain, r represents the distance between TX and RX with assumed matching impedance and polarization, and λ is the wavelength of the EM waves.

As a ray traverses in the scene, it might interact with the geometry before reaching a RX. The interactions contribute to the field strength, and the effect is modeled through dyadic coefficients for each interaction, where they decompose the field into orthogonal polarizations [36]. In [6], the field strength for a ray is represented as

$$\vec{E} = \mathrm{SF} \cdot \left(\prod_{i=1}^{N_\mathrm{b}} C_i\right) \cdot \vec{E_0} \cdot e^{-j\beta r_\mathrm{tot}}, \tag{3}$$

where SF is the spreading factor, which represents the propagation loss caused by spatial broadening of the wavefront [37]. N_b is the number of interactions for the path, β denotes the wave number, r_tot represents the total length of the path, $\vec{E_0}$ represents the field of the ray from the TX launched into the direction of departure, and lastly, C_i is the dyadic coefficient of the ith interaction.

2.2.2. Dyadic Coefficients

Dyadic coefficients represent the decomposition of the field in the ray-fixed coordinate system and the multiplication by the interaction coefficients [38]. The coefficients are used when considering the changes in amplitude and phase shift [6]. As an example, the dyadic reflection coefficient is presented by the equation [38, 39]

$$\overline{R} = \vec{e_{i\|}}\vec{e_{r\|}}\Gamma_\| + \vec{e_{i\perp}}\vec{e_{r\perp}}\Gamma_\perp, \tag{4}$$

where $\vec{e_{i\|}}$ and $\vec{e_{r\|}}$ are the incident and reflected parallel unit vectors, $\Gamma_\|$ and $\Gamma_\perp$ are the parallel and perpendicular Fresnel reflection coefficients, and lastly, $\vec{e_{i\perp}}$ and $\vec{e_{r\perp}}$ are the incident and reflected perpendicular unit vectors. The unit vectors $\vec{e_{i||}}$, $\vec{e_{r||}}$, $\vec{e_{i\perp}}$, and $\vec{e_{r\perp}}$ are defined by the following equations [38]

$$\vec{e_{i||}} = \frac{\vec{s_i} \times (\vec{n} \times \vec{s_i})}{\|\vec{s_i} \times (\vec{n} \times \vec{s_i})\|}, \tag{5}$$

$$\vec{e_{r||}} = \frac{\vec{s_r} \times (\vec{n} \times \vec{s_r})}{\|\vec{s_r} \times (\vec{n} \times \vec{s_r})\|}, \tag{6}$$

$$\vec{e_{i\perp}} = \vec{s_i} \times \vec{e_{i||}}, \tag{7}$$

and

$$\vec{e_{r\perp}} = \vec{s_r} \times \vec{e_{r||}}, \tag{8}$$

where $\vec{s_i}$ is the incident ray direction, $\vec{s_r}$ is the reflected ray direction, and $\vec{n}$ is the normal vector of the surface. From these equations we can conclude that the vectors $\vec{e_{i||}}$ and $\vec{e_{r||}}$ lie in the *parallel plane* spanned by the vectors $\vec{s_i}$, $\vec{s_r}$, and $\vec{n}$. Additionally, the vectors $\vec{e_{i\perp}}$ and $\vec{e_{r\perp}}$ are perpendicular to this plane.

The reflection and refraction dyadic coefficients are modeled by the famous Fresnel equations. Based on the Fresnel's equation [40], the reflection's perpendicular $\Gamma_\perp$ and parallel $\Gamma_{||}$ polarizations can be calculated using the equations

$$\Gamma_\perp = \frac{Z_2 \cos\theta_i - Z_1 \cos\theta_t}{Z_2 \cos\theta_i + Z_1 \cos\theta_t} \tag{9}$$

and

$$\Gamma_{||} = \frac{Z_2 \cos\theta_t - Z_1 \cos\theta_i}{Z_2 \cos\theta_t + Z_1 \cos\theta_i}, \tag{10}$$

where θ_i is the angle between the surface normal and the ray, θ_t is the angle between the surface normal and the reflected ray, Z_1 and Z_2 represents the impendances of the mediums, which are defined by relative permittivity ϵ_r, permeability and conductivity.

For refractions, the Snell's law can be used to get the refraction coefficient multipliers, if we assume that the medium is non-magnetic [31]. The Fresnel's reflection coefficients can be simplified to [40]

$$\Gamma_\perp = \frac{\cos\theta_i - \sqrt{\epsilon_r - \sin^2\theta_i}}{\cos\theta_i + \sqrt{\epsilon_r - \sin^2\theta_i}} \tag{11}$$

and

$$\Gamma_{||} = \frac{\epsilon_r \cos\theta_i - \sqrt{\epsilon_r - \sin^2\theta_i}}{\epsilon_r \cos\theta_i + \sqrt{\epsilon_r - \sin^2\theta_i}}. \tag{12}$$

For diffractions, often UTD-based coefficients are used, described in [11], and a well detailed example can be found in [38]. Scattering has its own coefficients. In [6], diffuse scattering is supported, where the approach for calculating the coefficients is described in [41]. For example, in [17], the Fresnel reflection coefficients are multiplied with a roughness correction factor ρ_s, which is introduced in [42] by the equations

$$\rho_s = \max\left[\exp\left(-\frac{1}{2}g^2\right), 0.15\right] \tag{13}$$

and

$$g = \frac{4\pi\sigma}{\lambda}\cos\varphi, \tag{14}$$

where σ is the standard deviation of the surface roughness, λ is the wavelength and φ is the angle between the ray and the normal of the surface. The value 0.15 was decided as the threshold for avoiding too small roughness factor.

3. GPU COMPUTING FOR RAY TRACING

In this chapter, we introduce the relevant GPU application programming interfaces (APIs), as well as the common GPU APIs used for RT. Then, we present multiple existing GPU accelerated RT based radio channel propagation modeling implementations and discuss their benefits and drawbacks.

3.1. GPU Computing

In the past decade, GPUs have been widely used in research as they evolved from configurable to programmable processors [1]. They are extremely suitable for parallel processing, as they contain many cores executing thousands of threads in parallel. Another point of attraction is that the computers equipped with such devices are very affordable [43].

3.1.1. Application Programming Interfaces

NVIDIA released Compute Unified Device Architecture (CUDA) [44] in 2007, which was the first general-purpose languange for programming GPUs [45]. Graphics APIs or abstractions of these were used previously. The introduction of CUDA resulted in an increased interest in GPU computing, as it provided a more convenient interface to use.

After the introduction of CUDA, multiple alternative APIs for general purpose computing on the GPU (GPGPU) have been developed. DirectCompute [46] is restricted to Windows only, and is only supported on versions starting from Vista. OpenCL [47] on the other hand is not restricted to a single operating system, and is more widely used. Vulkan [48] is the newest GPGPU API promising cross-platform support [49]. It also promises high performance and efficiency due to its low level control over the hardware, as the API is extremely explicit. Many of the operations in other APIs such as memory allocation and work submission are handled by the API, but in Vulkan these are done explicitly. Error checking and validation is present in the form of layers that can be turned off in release builds, resulting in low driver overhead.

Mammeri & Juurlink [49] benchmarked Vulkan against CUDA and OpenCL and concluded that Vulkan with low level optimizations on desktop platforms could offer 1.53 times faster execution when compared to CUDA, and 1.66 times faster execution when compared to OpenCL. On mobile platforms, 1.59 times faster execution was achieved with Vulkan when compared to OpenCL.

Although Vulkan could potentially bring better performance and enable execution on devices and platforms that do not support CUDA or OpenCL, most of the existing implementations of radio channel propagation modeling on the GPU are developed using CUDA, as well as the implementations developed in this thesis.

3.1.2. CUDA

NVIDIA GPU computing architecture consist of multi-threaded (MT) streaming multiprocessors [43]. These streaming multiprocessors provide threads, shared memory and processor cores for the CUDA thread blocks that have been scheduled for execution. Streaming multiprocessors contain parallel processor cores that execute instructions for threads in parallel. These parallel processors execute hundreds of threads in parallel. As there are multiple streaming multiprocessors, different kernels might be executing in parallel.

When a kernel is launched, the execution of it is represented as a 3D grid consisting of blocks [45]. A block consist of threads, and the threads in the same block can synchronize and cooperate using shared memory, illustrated in Figure 7. Each block is executed on a single multiprocessor, but the multiprocessor can execute multiple blocks in a preemptive way. Each kernel invocation can resolve its global index by accessing its block dimension and thread indices. The global index is a useful tool to retrieve and even write specific data in the context of a specific kernel invocation.

Transfering a single element from the main GPU memory takes hundreds of clock cycles, thus, memory latencies in the GPU are hidden through massively threaded execution model [45]. The GPU automatically switches to a waiting thread when a memory fetch occurs, which serves well for hiding the latencies when there are lots of available threads.

GPUs have a maximum number of active concurrent threads [45]. The number of active threads with respect to the maximum number of active concurrent threads is known as GPU occupancy, which can be used to represent the effectiveness of a kernel. In most cases when developing a kernel, it is beneficial to achieve as high GPU occupancy as possible, though in some cases where all the latencies are hidden, high GPU occupancy might result in lower performance.

A major difference to CPUs is that the GPU executes instructions in a way that 32 threads execute the same instructions, comparable to single instruction multiple data (SIMD) instructions [45]. This execution unit is known as a warp, which executes the same instruction on all of the 32 threads. If the threads do not take the same code paths, divergence occurs. In such a case, the diverging code paths have to be executed separately. It is important to aim for kernel code that takes the same path as in the worst case scenario the kernel execution could be 32 times slower than kernel code where each thread takes the same code path. Divergence can be avoided by for example, executing the diverging branch on the CPU side, or by sorting elements so that each thread executes their code with minimal diverging. Due to the mentioned effects of divergence, writing multiple kernels should also be considered to avoid clear cases of divergence.

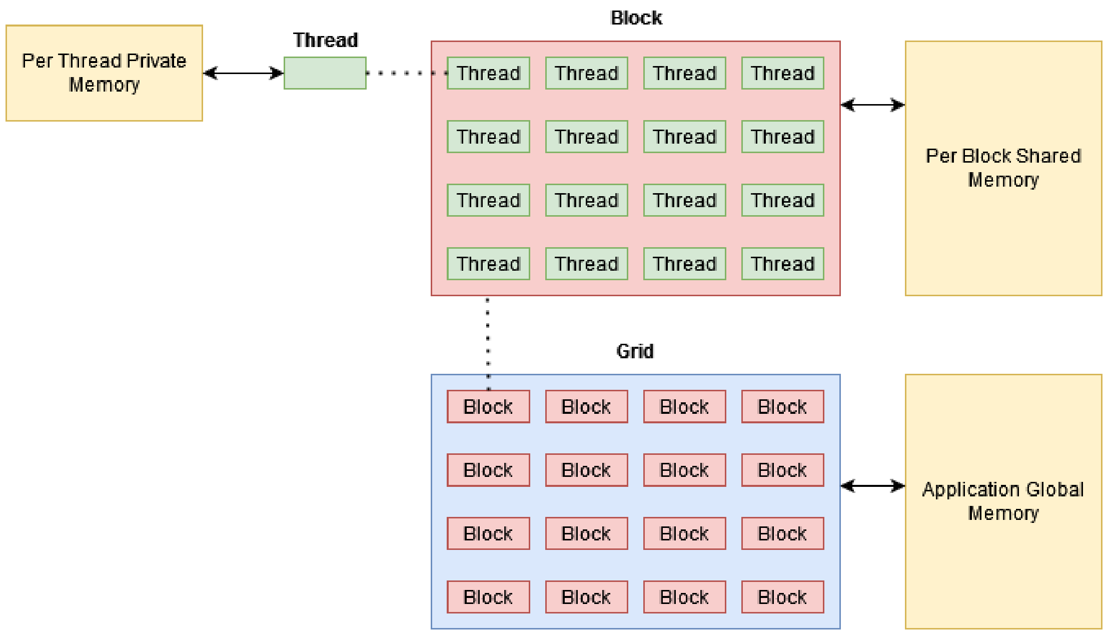

Figure 7. CUDA architecture.

3.1.3. Application Programming Interfaces for Ray Tracing

Building custom RT solutions can be hard and time consuming. Modern graphics- and compute-based GPU APIs are starting to support RT, and modern GPUs are developed with RT in mind. For example, NVIDIA introduced RT cores dedicated for RT in their RTX generation of GPUs [50].

The OptiX RT engine by NVIDIA is a general purpose, low level RT engine that is built around the CUDA architecture [51]. In OptiX, the geometry is stored in acceleration structures. An acceleration structure is a spatial representation of the geometry that is used by the ray traversal algorithm to efficiently find the intersecting primitives.

As of OptiX version 7 released in 2019, the API became more CUDA-centric and explicit, as the GPU memory management was changed to be handled by the application instead of OptiX [52]. The removal of OptiX host state was introduced as well, which meant that the application was now responsible for the management of resources such as scene graphs and materials.

The OptiX RT pipeline consists of user-defined programs, illustrated in Figure 8. Tracing of rays consists the following of three parts [53]. First, the ray is created, followed by scene traversal, and lastly, the result of scene traversal is interpreted by the user defined programs. When the ray being traced finds the closest point of intersection, the closest-hit program will be called. Any-hit program gets called for every new intersection point, and as one might expect, the miss program is called if no intersection points are found. The callable programs allow for additional programmability. The direct callable programs are invoked instantly, while the continuation callable programs are invoked by the scheduler. OptiX also has support for an user-defined exception program, which will be called if the programs in the RT pipeline cause an exception.

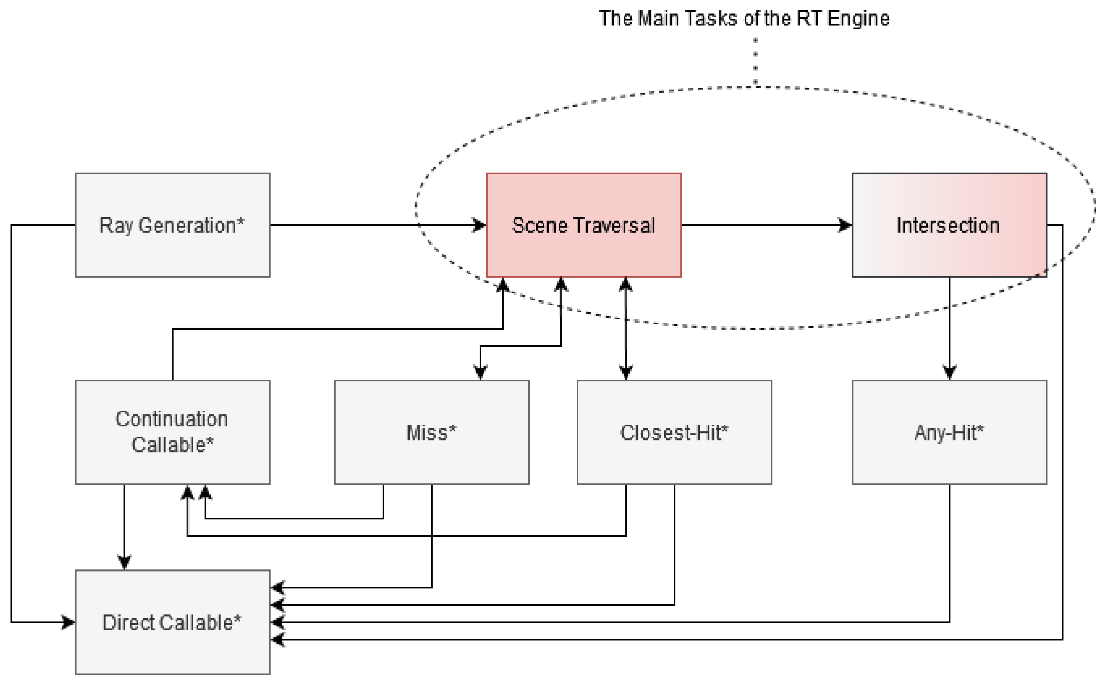

Figure 8. OptiX ray tracing pipeline, the user-defined programs are marked with an asterisk.

In November 2020, the Vulkan RT specification was released, and in December 2020, the first Vulkan software development kit (SDK) with RT extensions was released [54]. The Vulkan RT extensions were built with existing RT APIs such as OptiX in mind, making one of the goals of the extensions to enable easy porting to Vulkan from existing implementation with other RT APIs.

Due to the popularity of CUDA, OptiX is mostly used in the radio channel propagation modeling implementations where a RT API is used. Vulkan RT extensions could bring radio channel propagation modeling solutions to platforms that do not support CUDA. Hardware accelerated RT for ARM Mali mobile GPUs [55] and Vulkan RT extensions for mobile platforms [56] are already under development.

3.2. Implementations for Radio Channel Propagation Modeling

In this section, we present GPU accelerated RT-based radio channel propagation modeling implementations in depth on an algorithmic level. As GPU accelerated RT-based radio channel propagation modeling is the most efficient way of conducting simulations, many implementations such as [4, 6, 32, 57, 58, 59] utilize this acceleration method.

3.2.1. Ray Launching with a Discretized Environment

Discrete environment-driven ray launching (DED-RL) algorithm by Lu *et al.* [6] is a GPU-based ray launching solution for fast 3D prediction in urban areas, developed with CUDA. The environment is first discretized[1] as in Figure 6, where each surface is filled with a nonoverlapping grid of tiles. If a wall being discretized does not fulfill the dimension requirements of the tile, the wall is discarded. The tile square area increases the computation accuracy the smaller it is. However, choosing a small tile square area also increases the time needed for computation.

Visibility preprocessing is done to determine the visibilities between the tiles [6]. The visibilities of each tile is saved in an $N \times N$ binary matrix, where N denotes the number of tiles. The visibility between two tiles can be determined by their respective indices, which can then be used to query the visibility matrix.

The center point of each visible tile is used as the RX for each interaction [6]. The incident field, total incident ray field, angle of arrival and time delay are computed and stored for each ray if the ray passes the visibility analysis. The first level visibility analysis, presented in Figure 9 consists of a visibility matrix lookup, which is done for each discretized element.

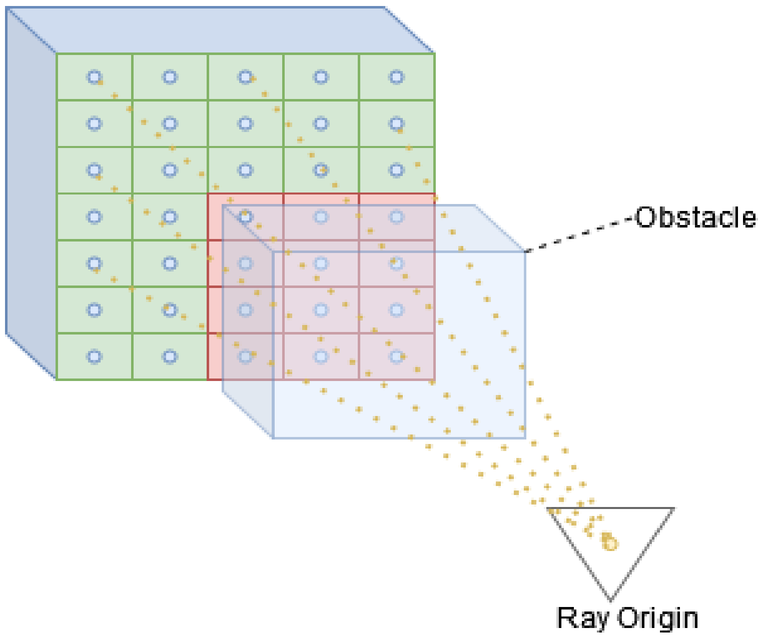

Figure 9. First level visibility analysis. The green tiles are visible from the ray origin.

After the discretized element passes the first level visibility analysis, simple processing methods depending on the interaction type are used to determine whether the interaction is valid or not [6]. For reflections, the second-level visibility is determined by back-projecting the tile center point to the wall plane of the spawning tile. The projected point is then checked whether it is in the spawning tile or not. A visualization of the second-level visibility analysis (SLVA) of reflections can be seen in Figure 10. SLVA of diffractions is determined based on two Keller's cones that define the visibility region, which is then used to test if the center point of the tile is inside the region, illustrated in Figure 11. The implementation also supports diffuse scattering, which does not require any more visibility processing, as all the visible tiles in the visibility matrix can be reached through scattering. The rays are traced until the

[1]The idea of discretization was first considered by Hoppe *et al.* in 1999 [28].

maximum number of interactions is reached, or until the incident power falls below the threshold.

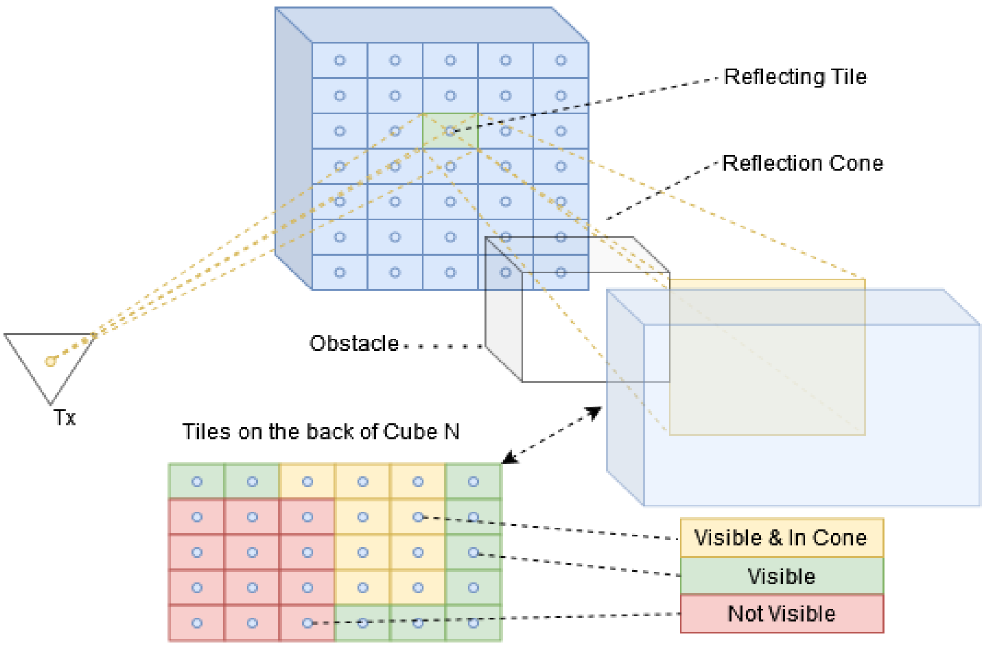

Figure 10. Second-level visibility analysis of reflections.

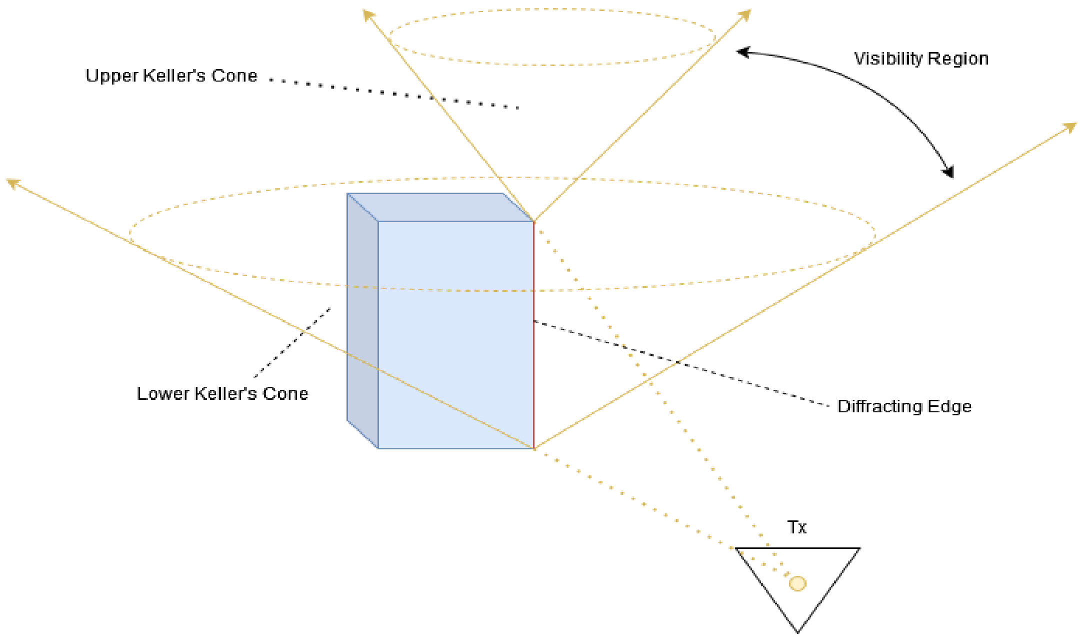

Figure 11. Second-level visibility analysis of diffractions.

3.2.2. Ray Launching for VANET Application Simulations

The ray launching implementation by Schiller *et al.* [4] aims to bring fast and highly detailed simulations of V2V communication channel. The implementation is used for virtual test driving of VANET applications. It is written with CUDA, and the rays are traced using OptiX.

The algorithm is cut into two phases [4]. In the first phase, the rays are traced in the scene by using ray launching. In the second phase, the EM calculations are done for the paths that were acquired in the first phase.

The first step in in the ray launching phase is resolving the LOS paths [4]. For the other interaction types, the initial rays are uniformly distributed to random directions in order to gain an acceptable number of rays, as well as an accurate representation of the possible paths. The ray directions are sorted using Z-order curve [60] in order to avoid divergence. The computation and sorting of the primary ray directions is done as a preprocessing step before tracing the paths.

The RX antennas are modeled as spheres, where the size of the sphere denotes how many rays are collected at each RX [4]. Each triangle in the scene has information regarding its material and interaction type. The reflections are traced in the direction of the reflection and the tracing continues until the configured number of reflections is reached. The diffracting edges are resolved in a preprocessing stage based on an angle threshold. They are modeled as cylinders centered around the edge to allow efficient intersection testing during the RT. Multiple rays are launched from diffracting edges in the shape of a Keller's cone as in Figure 2, according to the UTD.

A methodology called focusing is applied to the RL algorithm, which aims to reduce the required computation while maintaining the accuracy [4]. The motivation for this methodology comes from the many possible paths that will never reach a RX, which results in a waste of computation resources. The scene is sampled coarsely with few rays and with a big RX sphere, of which the rays that lead to an intersection with the RX sphere are then sampled in more detail.

As the RL supports multiframe simulation, the previous frame is considered when the movement of objects is minimal [4]. The direction information of the successful paths in the previous frame is used in the coarse phase to assess similarity between the frames. This information can then be used to retrieve paths that have not changed significantly since the last frame.

The valid propagation paths are saved to a scene query tree, which might contain duplicate paths [4]. The paths are then sorted based on primitive indices and interaction types. The sorted path list is then processed to remove duplicate paths, resulting in each remaining path being unique. Finally, the EM calculations are done for the remaining valid paths based on the principles of geometrical optics and the UTD.

3.2.3. Beam-Tracing Method

As traditional RT is time consuming due to the discrete sampling of the space, Tan *et al.* [32] developed 3D GPU-based beam-tracing solution for predicting radio channel propagation in indoor environments to overcome issues in traditional RT.

A kD-tree [61] is constructed as a preprocessing step for indexing the triangles of the scene [32]. In kD-tree, an axis-perpendicular plane is inserted into axis-aligned bounding box (AABB), causing division of the current AABB into two AABBs.

A fast 3D stack-based beam-triangle intersection algorithm was developed for the beam-tracer [32]. As the beam can intersect the triangle in multiple ways, there are different ways of launching secondary beams from the interaction. The fully intersecting beams are mirrored, while the partially intersecting beams are clipped to the intersecting area. The partial and fully intersecting beams are then divided into two lists that will be further processed in their respective kernels. These kernels are then used to launch the secondary beams to finalize beam-triangle intersection tests.

The generated beams are stored in a beam tree structure, where the TX acts as the root [32]. From the tree, all the paths can be found. For each beam, a reference is stored in the structure to the beams parent, the corresponding beam, and the triangle primitive from which the beam starts.

The rays are traced starting from the bottom of the tree [32]. If the bottom level beams contain a RX, a ray is launched to the apex of the beam with the RX as the ray origin. If the ray does not hit the referred triangle of the node, the path is discarded. The tracing runs iteratively until the TX is reached. If the interaction is a reflection, a reflected ray is launched towards the parent node's apex of the beam. In the case of diffractions, the diffracting edges are stored during beam-tracing, that can then be used to compute intersection points between the edge and the beam to form two Keller's cones, that are then used to confirm the validity of the diffraction. The path is then determined by calculating the diffraction point for the valid diffractions.

After the valid paths have been determined, a hash is calculated for each path, which is then inserted into a hash map in order to remove duplicate paths [32]. Next, the EM calculations for the paths are performed on the GPU. In the first step of the EM calculations, a kernel calculates the interaction coefficients at all frequencies for all the paths, followed by another kernel calculating the fields of the paths. In the last step, the total field contribution of the paths is calculated on a separate kernel.

3.2.4. Discussion

The presented implementations have their benefits and drawbacks. Especially with diffractions, issues start to surface. For example, in the ray launcher by Schiller *et al.* [4] angular discretization is used to launch the diffracting rays, meaning that balancing the number of diffracting rays for each edge is hard due to the unknown distances to the next interaction points. In other words, if an object is hit by diffracting rays from the same edge, the number of diffracting rays is purely determined by the distance due to the angular dispersion. This might cause some paths to be missed if the object is far away, and on the other hand if the object is close, similar paths are generated, causing unnecessary computation. As mentioned in Section 2.1.2, the angular dispersion can be reduced with the ray splitting technique with the drawback of increasing the computational complexity due to the additional rays [22, 23]. DED-RL [6] solves this issue by utilizing environment-based discretization, where each discretized element is considered when checking for a valid diffraction from an edge. The accuracy is purely determined by the discretization resolution, and only one ray for each valid discretized element is generated. The beam-tracing method [32] is restricted to only a single diffraction, which has to be the last interaction. Thus, it is not very viable for simulating diffractions or comparable to the other methods.

Both the beam-tracing implementation [32], and the ray launcher by Schiller *et al.* [4] perform the EM calculations as a postprocessing step, while in DED-RL they are already performed during the RL phase. Performing them during the RL allows discarding of the paths that would not contribute to the EM field in a significant manner. It reduces unnecessary computation, as many new rays might be launched from each interaction point along the path.

Considering that the effect of scattering was deemed significant in NLOS areas [17], as mentioned in Section 2.1.1, taking it into consideration is important if rough surfaces are present. DED-RL supports diffuse scattering [6], while the ray launcher by Schiller *et al.* [4] and the beam-tracing method [32] in the current state do not. The latter implementations in their current state cannot sufficiently support diffuse scattering, as it introduces similar problems as simulating diffractions. As mentioned in Section 2.1.1, when scattering occurs, the power scatters into all directions [17, 18], meaning that rays or beams would have to be launched into all directions.

DED-RL [6] has its downsides. One of them is that the scene has to be represented with a tile-based discretization, which requires additional approximation of the original geometry representation. Another downside is that the traced paths do not represent the optimal paths, thus, additional processing is needed if the coarse approximations are not enough. Methods such as the image method presented in Section 2.1.2 can be applied to get more accurate paths. In this thesis work, a gradient descent-based approach is developed for this purpose in Section 4.4.

4. IMPLEMENTATION

In this chapter, optimizations and improvements are implemented to an existing path search implementation developed at the University of Oulu. Additionally, a path refinement solution is developed to compute the valid optimal paths fulfilling the Fermat's principle of least time. The development work is performed using C++, CUDA and NVIDIA OptiX RT engine.

4.1. Overview

The path generation consists of input preparation, coarse path search, and refinement, which are presented in Figure 12. The input preparation consists of geometry discretization, and placement of TXs and RXs in the scene. All of the components that use geometry information use an internal format, which contains information about vertices, indices, and materials, as well as information about the diffraction edges and walls in the scene. A tool was developed, which automatically detects the mentioned attributes from glTF 2.0 [62] models and converts the relevant model data into the internal format.

The coarse path search takes the data generated in the input preparation stage as an input. Its main objective is to find paths between TXs and RXs determined in the input preparation stage according to the rules presented in Section 3.2.1 for a given number of interactions.

The refinement component takes the paths generated by the coarse path search with the objective of computing the shortest path along the geometric elements that the path interacts with. The refined paths are then validated using OptiX to make sure that the interaction points did not move out of the bounds of the geometry or get occluded.

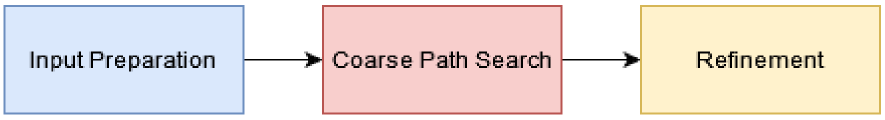

Figure 12. Overview of the different steps involved.

4.2. Coarse Path Search

Before diving into the optimizations and enhancements, it is necessary to understand the existing implementation. The implementation is based on the DED-RL solution by Lu *et al.* [6], which is presented in Section 3.2.1. The steps involved in the coarse path search are presented in Figure 13.

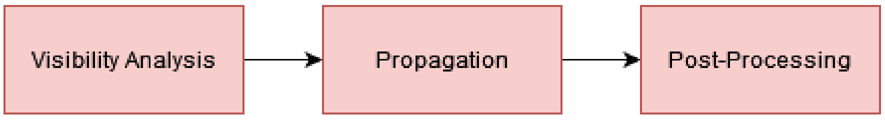

Figure 13. Coarse path search.

The first step is the determination of visibilities between the discretized elements, which is used in the first level visibility checks, as described in Section 3.2.1. The propagation is implemented by launching two types of kernels, a transmission and a propagation kernel. In the transmission kernel, the initial interactions are determined from the current TX. The coarse path search process is performed for each TX separately. If the initial interaction is with a RX, the path is deemed as a LOS path. If the initial interaction is valid and the type is reflection or diffraction, the path is prepared for the propagation kernel.

The launch size of the propagation kernel is the same which was used in the transmission kernel. Each launch index handles the generated rays recursively in its own thread, until the maximum interaction depth, or until all of the paths have been processed. For each RX point, a buffer of chosen size is allocated for storing the paths. CPU side synchronizations are required if the path buffer is full, thus, the propagation kernel is called in a loop until all the paths have been processed. As a post-processing measure on the CPU, the paths are sorted, and then processed in an iterative fashion to discard the duplicate paths.

In the implementation, the propagation is limited by thresholds for the maximum number of interactions and diffractions. Additionally, the weak rays are discarded. Assuming an omnidirectional radiator at a TX point (1W power), the field vectors are computed at the interaction points, and if their power value is below a chosen threshold, the path is discarded. The power value for the complex field vector $\vec{E} = [E_\phi, E_\theta]^T$ is computed using

$$P\left(\vec{E}\right) = 10\log_{10}\left(|E_\phi|^2 + |E_\theta|^2\right) \text{ [dB]}. \tag{15}$$

4.3. Coarse Path Search Optimizations

In this thesis, the visibility analysis is taken from the original implementation as is, and we focus on optimizing the propagation. The initial guess for the bottleneck in the existing implementation was low GPU occupancy, as the launch size for the transmission and propagation kernels was the number of generated initial transmission invocations. As an example, if the initial transmission invocation count would be 100, it would mean that only 100 GPU threads would actively be processing all the possible paths at most. These same threads would be handling the newly generated rays at each new interaction, resulting in a high workload for each thread and the majority of the available GPU computing resources being idle.

Another problem is that the generated ray count at each interaction point depends on the environment. Even if the initial transmission invocation count would be high, it is very likely that the total number of rays launched by each thread would not be equal, resulting in an increasingly lower GPU occupancy as the interaction depth advances.

4.3.1. Optimizing the Existing Approach

The first valid propagation points are processed in the transmission kernel to get the number of interactions in the next depth level. A buffer is then allocated for these interactions, and a kernel is launched to fill the buffer with the interactions. After the buffer is filled, the rest of the depth levels are propagated the same way as in the original implementation with the difference that the filled buffer is used in the propagation kernel, resulting in a higher kernel launch size and one less interaction depth level to process in the propagation kernel.

The initial optimized version was mostly developed to confirm the suspected bottlenecks. The GPU memory usage is unpredictable and possibly very high, as it depends on the number of generated rays from the first interactions in the transmission kernel.

The GPU occupancy in this implementation still gets worse as the interaction depth advances, though the GPU workload is distributed more evenly as the kernel launch dimensions are larger. It still suffers from the same problem as the original implementation, because the interactions are processed recursively on the same GPU thread.

4.3.2. Chaining Implementations

To overcome the problems with the original implementation, a new approach called chaining was designed for managing the computations. The idea is to have separate kernels for handling reflections, diffractions and receival of rays at RX points. Dividing the workload into smaller kernels that only handle specific interactions reduces the divergence, and thus, increases the performance. The transmission kernels are similar to the existing implementation, with the difference that only a specific type of interaction is handled at once. Five variants of this approach were developed, which are presented in this section. In addition, the sorting and removal of duplicate paths adopted from the original implementation was replaced with a hash-based approach in the best performing chaining implementation.

The launching starts at the interaction depth level zero. If the depth level is zero, transmission kernels are used. Each level will first call the RX kernel, followed by the reflection and diffraction kernels. The interaction processing functions will keep launching the relevant processing kernel of the interaction type until all of the interactions have been processed successfully. The reflection and diffraction kernels will always call the top level launching function when a CPU side synchronization occurs. The high level logic of the launching can be seen in figure 14.

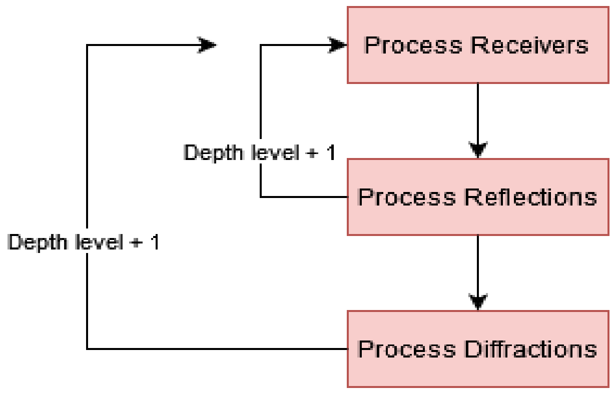

Figure 14. Launching process in the chaining implementation.

The propagation kernel only handles one interaction based on the launch index, which results in a higher GPU occupancy, as each invocation requires only a small amount of work. The kernel is launched for each parent in the current depth level, which was filled in the previous depth level. The correct launch size is resolved dynamically. In the case of valid reflections and diffractions, the relevant information is written to the parent buffer of the next depth level.

CPU-based chaining (CPU ST)

CPU-based chaining is the basic version of the chaining implementations. The launching is executed on a single CPU thread. Each processing function downloads the required parent data from the GPU before the launching can be started for the parents of the current depth level.

Multi-threaded CPU-based chaining (CPU MT)

For the MT CPU-based implementation, a thread pool implementation was developed for better CPU utilization. Each thread has its own context and buffers required for launching. CUDA streams are utilized, which make it possible to do asynchronous operations with the GPU, meaning that we do not have to wait for other cores to finish their GPU work. For example, it enables us to handle memory operations between the CPU and the GPU asynchronously, and allows us to wait for the completion of specific kernel launches.

The initial invocation count for the transmission kernels is divided based on the available number of CPU cores, so that the workload can be distributed as evenly as possible for each launching context.

GPU-based chaining (GPU ST)

With the help of dynamic parallelism feature of CUDA [63], it is possible to launch a kernel from within a kernel. This means that we can avoid some memory transfers between the CPU and the GPU, as we can interact with the GPU memory in the launching kernel. The launching happens on a single GPU thread, and synchronization with the CPU is only required when the valid path buffer is full.

Multi-threaded GPU-based chaining (GPU MT, GPU MT Async)

Single-threaded (ST) GPU implementation was developed into a MT version as well. The GPU-based launching is also able to exploit CUDA streams, which in this implementation rises in importance, as the MT GPU launching is in most cases performed on a lot more threads than in CPU MT.

The logic between the MT implementations are quite the same, but the GPU-based implementation has the advantage of not having to do memory transfer operations, as it is interacting with the GPU memory directly. The launching block size is by default 32, meaning that there are 32 invocations for the launching.

In the GPU implementations during the propagation, the CPU downloads the path buffer and processes the contents once the buffer is full. Downloading and processing large buffers takes a decent amount of time, during which the GPU has to wait for the CPU to finish its work before the propagation can continue. To overcome the delay caused by the CPU synchronizations, GPU MT was further developed to support double buffering. As the name suggests, the path buffer is divided into two separate buffers, which allows for the CPU and the GPU to work asynchronously. The thread pool implementation which was developed for CPU MT is utilized in the double buffering version as well. When the CPU synchronization happens, the current buffer index is set on the GPU, and the launching is continued immediately. An available thread is retrieved from the thread pool on the CPU side, which downloads the valid paths from the previously filled buffer asynchronously by utilizing a CUDA stream.

Duplicate path processing

The original duplicate path removal post-processing step was replaced with a hash-based solution, which checks for the uniqueness of the paths during the propagation. As the comparison in the original implementation was based on integers, implementing a function for calculating the hash of a path was straightforward, as in [32]. When the valid paths are downloaded from the GPU, the hash for each path is calculated. The hashes are then used to query the hashmap to check if the paths already exists. If the hash for a path is not found in the hashmap, the path is saved.

4.4. Path Refinement

Paths generated by the coarse path search are coarse approximations and do not represent the optimal path, as the intermediate paths come from discretization. Gradient descent algorithm was applied as the solution for finding the optimal paths fulfilling the Fermat's principle of least time. The different steps in the refinement process can be seen in Figure 15.

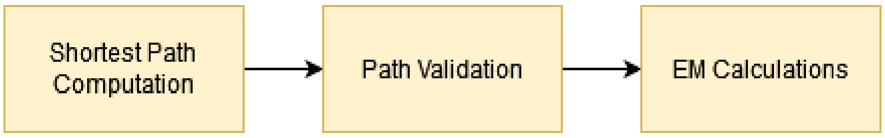

Figure 15. Overview of Refinement.

The pre-processing code is written in C++, and the gradient descent algorithm is written in CUDA compatible C++. As the algorithm is written in CUDA compatible C++, both CPU- and GPU-based implementations were developed. CPU-based implementation runs on a single thread only and could easily be expanded to run on multiple threads using the thread pool implementation, which was developed for CPU MT. Since all of the paths that are being processed can be refined in parallel, as they do not interfere with each other in any way, it is more beneficial to keep the CPU implementation as simple as possible for potential debugging purposes. The GPU implementation should be used when fast execution speed is desired.

4.4.1. Gradient Descent

Gradient descent algorithm can be used to minimize a function $f(x)$ by updating the parameters in the direction of the function's negative gradient [64]. In our case, we want to minimize the path length to find the path fulfilling the Fermat's principle of least time. The gradient descent algorithm can be represented by the following equation [64]

$$x_{n+1} = x_n - \gamma_n \nabla f(x_n), n > 0, \tag{16}$$

where γ_n is the step size, x_n represents the parameters, and $\nabla f(x_n)$ represents the gradient of the function.

Each interaction point is updated with (16) iteratively until one of the thresholds is reached. The convergence and maximum iteration thresholds are taken as arguments. In the current implementation the step size γ is halved every 25 iterations, which should be further developed to be computed by a step size algorithm in the future. As the current supported interaction types in the coarse path search are reflection and diffraction, we do not have to consider the index of refraction in the equation, as the rays travel only in a homogenous medium. Thus, we can represent the function for the path length which we aim to minimize by the following equation

$$f(\vec{x}) = \sum_{k=0}^{N} \|I_{k+1} - I_k\|, \tag{17}$$

where N is the number of interactions and I_k is the notation for the function of the interaction type. Especially I_0 and I_{N+1} correspond to TX and RX points of the path, respectively. The path minimization for boundless planes and straight edges is convex, thus, the converged path found with gradient descent algorithm represents an unique global minimum, which is validated afterwards.

In the case of N_r reflections and N_d diffractions, the vector of variables for the path can be represented as

$$\vec{x} = (r_1, \ldots, r_{N_r}, s_1, \ldots, s_{N_r}, t_1, \ldots, t_{N_d})^T, \tag{18}$$

where the elements r_i, s_i, and t_i are updated with the function f differentiated with respect to the relevant variable. Thus, the gradient becomes

$$\nabla f(\vec{x}) = \left(f_{r_1}, \ldots, f_{r_{N_r}}, \ldots, f_{s_1}, \ldots, f_{s_{N_r}}, \ldots, f_{t_1}, \ldots, f_{t_{N_d}}\right)^T, \tag{19}$$

where f_u denotes the partial derivative $\partial f / \partial u$.

Reflections

In the case of reflections, we have the surface point p and orthonormal vectors $\vec{u}$ and $\vec{v}$. A reflection point R_i on the plane can be represented as

$$I_k = R_i = p_i + r_i \vec{u_i} + s_i \vec{v_i}. \tag{20}$$

For each reflecting interaction the path consists of the source point I_{k-1}, the interaction point I_k, and the destination point of the reflected ray I_{k+1}. The path length is determined by the variables r_i and s_i, thus, we get the equations

$$\frac{\partial}{\partial r_i}\|I_{k+1} - I_k\| = \frac{(I_k - I_{k+1})}{\|I_k - I_{k+1}\|} \cdot \vec{u_i}, \tag{21}$$

$$\frac{\partial}{\partial r_i}\|I_k - I_{k-1}\| = \frac{(I_k - I_{k-1})}{\|I_k - I_{k-1}\|} \cdot \vec{u_i}, \tag{22}$$

$$\frac{\partial}{\partial s_i}\|I_{k+1} - I_k\| = \frac{(I_k - I_{k+1})}{\|I_k - I_{k+1}\|} \cdot \vec{v_i}, \tag{23}$$

and

$$\frac{\partial}{\partial s_i}\|I_k - I_{k-1}\| = \frac{(I_k - I_{k-1})}{\|I_k - I_{k-1}\|} \cdot \vec{v_i}, \tag{24}$$

from which we can compose the following functions that are used for updating the vector elements r_i

$$f_{r_i} = \left(\frac{(I_k - I_{k+1})}{\|I_k - I_{k+1}\|} + \frac{(I_k - I_{k-1})}{\|I_k - I_{k-1}\|}\right) \cdot \vec{u_i} \tag{25}$$

and s_i

$$f_{s_i} = \left(\frac{(I_k - I_{k+1})}{\|I_k - I_{k+1}\|} + \frac{(I_k - I_{k-1})}{\|I_k - I_{k-1}\|}\right) \cdot \vec{v_i}. \tag{26}$$

Diffractions

For diffractions, we have the surface point p and the normalized direction vector $\vec{w}$ along the edge. The diffraction points can be represented with the equation

$$I_k = D_i = p_i + t_i\vec{w_i}. \tag{27}$$

The same way as for reflections, the path consists of the source point I_{k-1}, the interaction point I_k, and the destination point of the diffracted ray I_{k+1}. For diffractions we only have the normalized direction vector $\vec{w}$, thus, the length of the path is determined by the variable t_i, resulting in the equations

$$\frac{\partial}{\partial t_i}\|I_{k+1} - I_k\| = \frac{(I_k - I_{k+1})}{\|I_k - I_{k+1}\|} \cdot \vec{w_i} \tag{28}$$

and

$$\frac{\partial}{\partial t_i}\|I_k - I_{k-1}\| = \frac{(I_k - I_{k-1})}{\|I_k - I_{k-1}\|} \cdot \vec{w_i}, \tag{29}$$

from which we can compose the function for updating the vector elements t_i

$$f_{t_i} = \left(\frac{(I_k - I_{k+1})}{\|I_k - I_{k+1}\|} + \frac{(I_k - I_{k-1})}{\|I_k - I_{k-1}\|}\right) \cdot \vec{w_i}. \tag{30}$$

4.4.2. Path Validation

After the path refinement was developed, integration with the existing path validation and EM calculation code was performed. The path validation is performed using OptiX, similarly to the visibility analysis of coarse path search. The refinement algorithm takes the paths generated by the coarse path search as an input, does the refinement, and then validates the paths and performs EM calculations for the refined paths that converged.

Path validation is needed, as the refinement code might move the interaction points outside of the surface or diffraction edge bounds, or there might be an object in the way of the ray between interaction points after the refinement. Another issue is that due to the design of DED-RL presented in Section 3.2.1, SLVA may result in infinity bouncing. This increases the likelihood of the refined paths to fail the validation. An example of this is illustrated in Figure 16, where in situation A, the ray launched from the source point P is reflected at the tile Q. SLVA determines that the tile R is suitable for reflection. In situation B, the tile R reflects the ray coming from the point Q, and SLVA determines the tile Q suitable for reflection. In situation C, the tile Q reflects the ray coming from the point R, and SLVA determines the tile R suitable for reflection. Thus, it is possible for the algorithm to keep bouncing between the situations B and C infinitely.

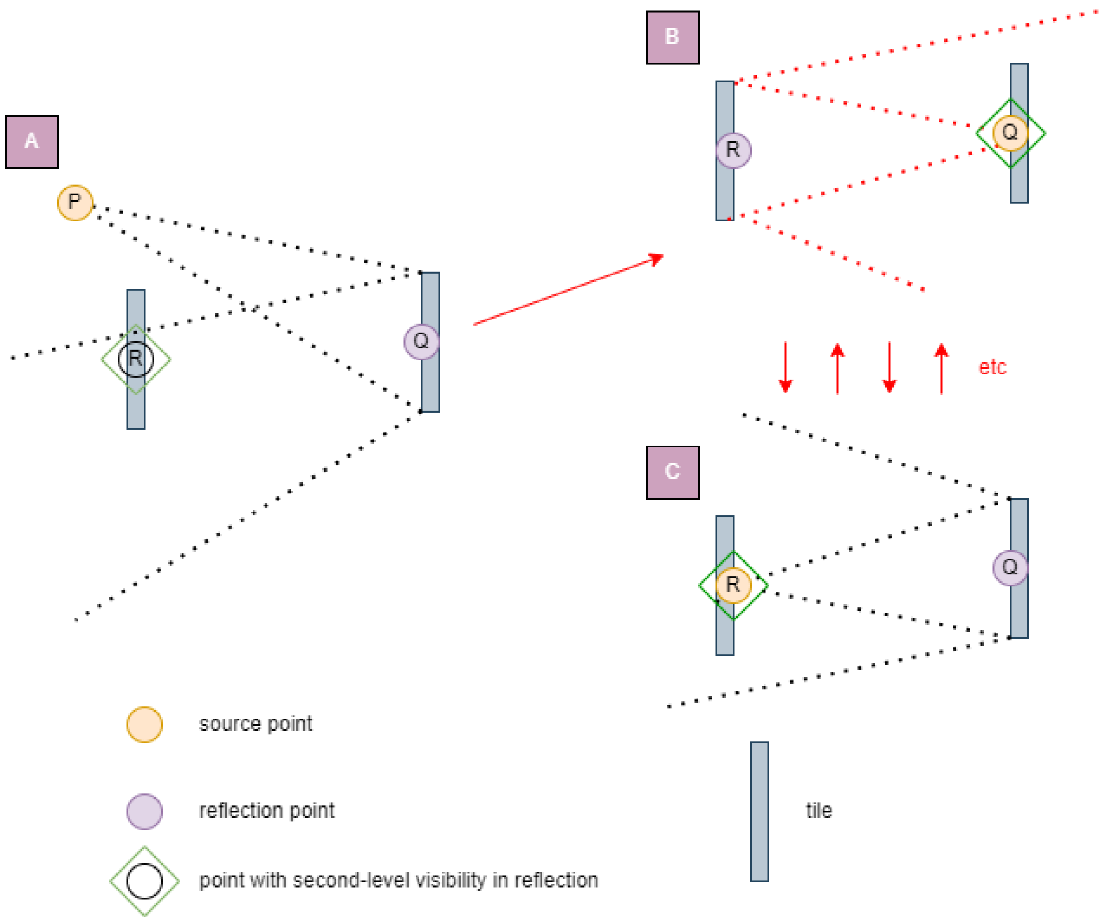

Figure 16. Infinity bouncing problem of DED-RL.

5. EXPERIMENTS

In this chapter, we conduct benchmarks with the original and the optimized implementations of the propagation step in the coarse path search, which are presented in Section 4.2. The benchmarks are conducted on two computer setups, System G and System R. The components of these setups are presented in Table 1. Varying parameters will be used for the experiments that are evaluated with an indoor and an outdoor model, which are real locations from Oulu, Finland. The outdoor scene is a model of Etu-Lyötty, which can be seen in Figure 17. The indoor scene is a model of a corridor in the CWC research unit located in the University of Oulu, presented in Figure 18. For the path refinement experiments, we will refine the final paths gathered in the coarse path search experiments and benchmark the execution time on the CPU and GPU. The development work was done on System G. Some additional experiments are conducted on System R.

Table 1. The computer setups System G and System R used in the experiments.

	System G	System R
CPU	Intel i5-2500k	Intel i7-8700k
CPU Cores	4	6
RAM (GB)	8	32
GPU	NVIDIA GeForce GTX 1070	GeForce RTX 2080 Ti
CUDA Cores	1920	4352
VRAM (GB)	8	11
OS	Windows 7	Windows 10

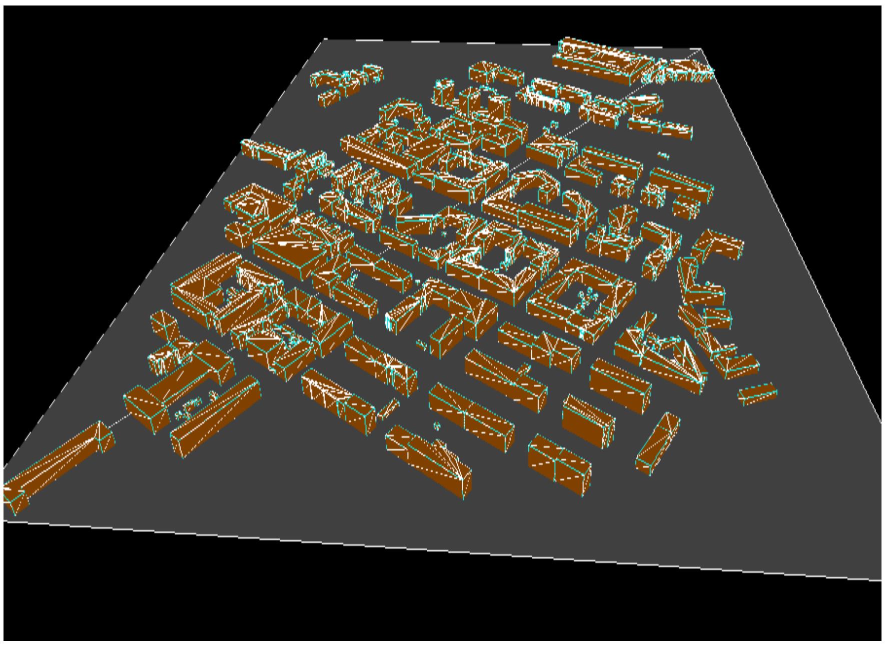
Figure 17. The Etu-Lyötty scene.

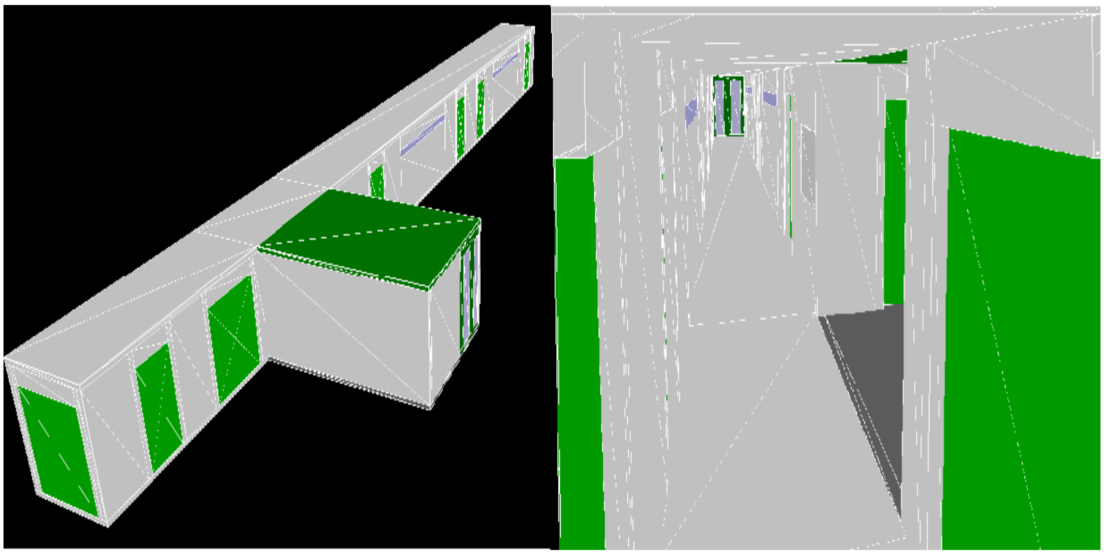

Figure 18. An inside and outside view of the CWC corridor scene.

5.1. Coarse Path Search

The first benchmarks of propagation were performed without saving the interaction points, as saving them is only required by the path refiner. The details of the implementations can be found in Section 4.2. The maximum number of interactions will be referred as #Ia in the tables. For the initial benchmarks, a 5 minute threshold was decided as the time limit for each execution, thus, execution times exceeding this limit are not reported.

5.1.1. Outdoor Scene

The Etu-Lyötty experiments were conducted with 5$\times$5m discretization for the walls and 10$\times$10m discretization for the ground plane, which resulted in 7389 tiles and 3239 diffraction edges. The scene has one TX and a RX grid of 719 points. 1.4GHz carrier frequency was used, and -100dB power threshold for the field vector powers. The size of the path buffer for each RX point was 10000, unless mentioned otherwise.

In Table 2, common path statistics are presented for the mentioned setup. In the fourth column, the number of RX points receiving at least one path is given, and in the fifth column, the average number of paths for those RX points. The Figures 19, 20 and 21 include benchmarks conducted on System G. The details can be found in Tables 1, 2, and 3 of Appendix 1. In these benchmarks, the interaction points are not saved. Similar benchmarks were done on System R. The initial benchmarks can be found in Table 3. The GPU MT Async benchmarks can be found in Figure 22 and the details are presented in Table 4 of Appendix 1.

Table 2. Path statistics of the Etu-Lyötty experiments.

#Ia	Diffraction Limit	#Paths	#Receivers	#Paths Per Receiver
2	0	1170	348	3.36
3	0	4527	459	9.86
4	0	17295	529	32.69
5	0	64683	564	114.69
6	0	242087	579	418.11
2	1	33085	566	58.45
3	1	153041	579	264.32
4	1	510818	587	870.22
5	1	1286550	588	2188.01
6	1	2807669	588	4774.95
2	2	88009	585	150.44
3	2	379181	587	645.96
4	2	1118404	588	1902.05
5	2	2572653	588	4375.26
6	2	5024738	588	8545.47

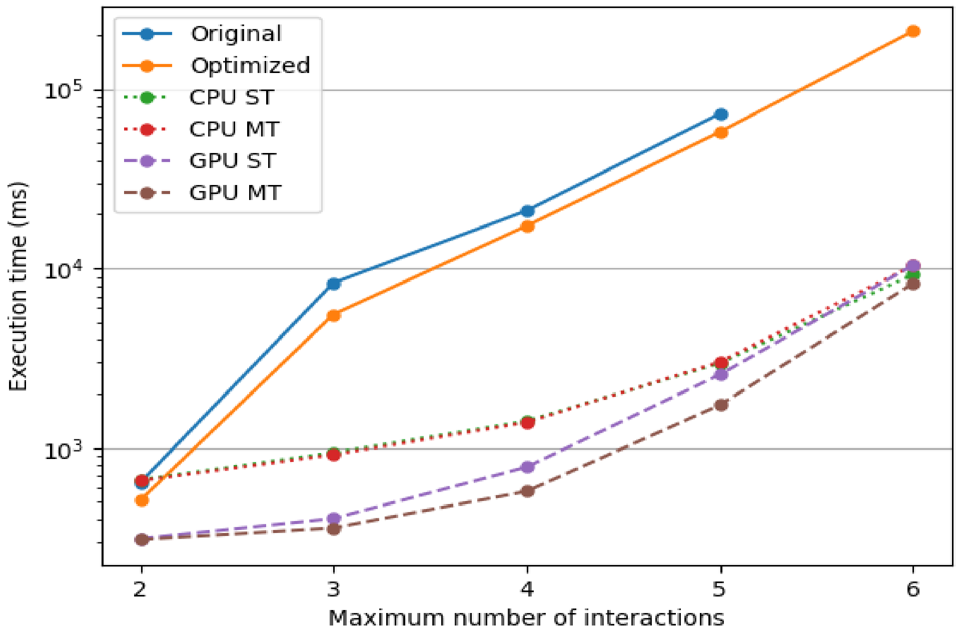

Figure 19. Propagation time of the implementations on System G in the Etu-Lyötty scene when diffractions are limited to 0.

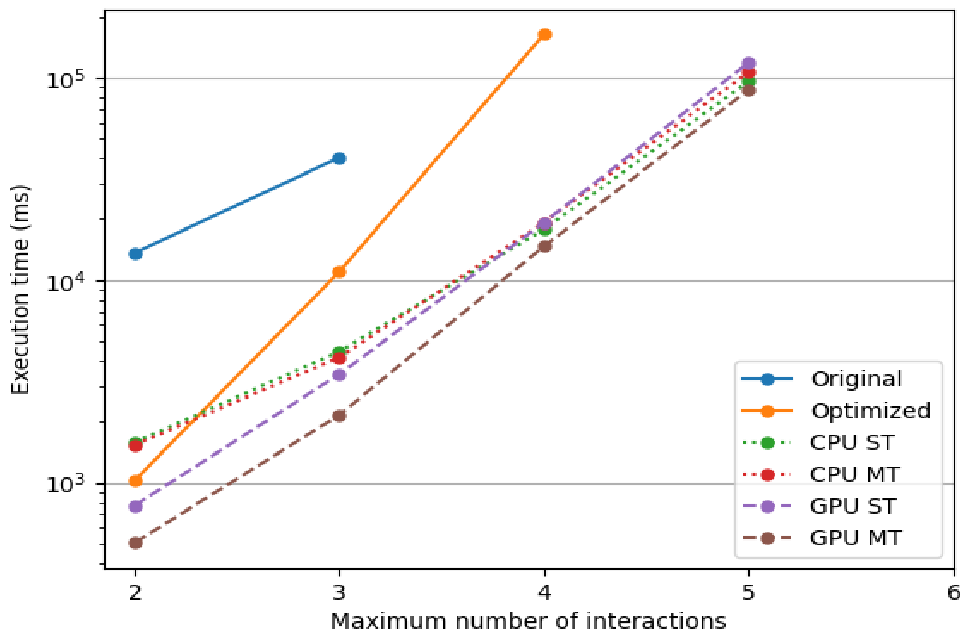

Figure 20. Propagation time of the implementations on System G in the Etu-Lyötty scene when diffractions are limited to 1.

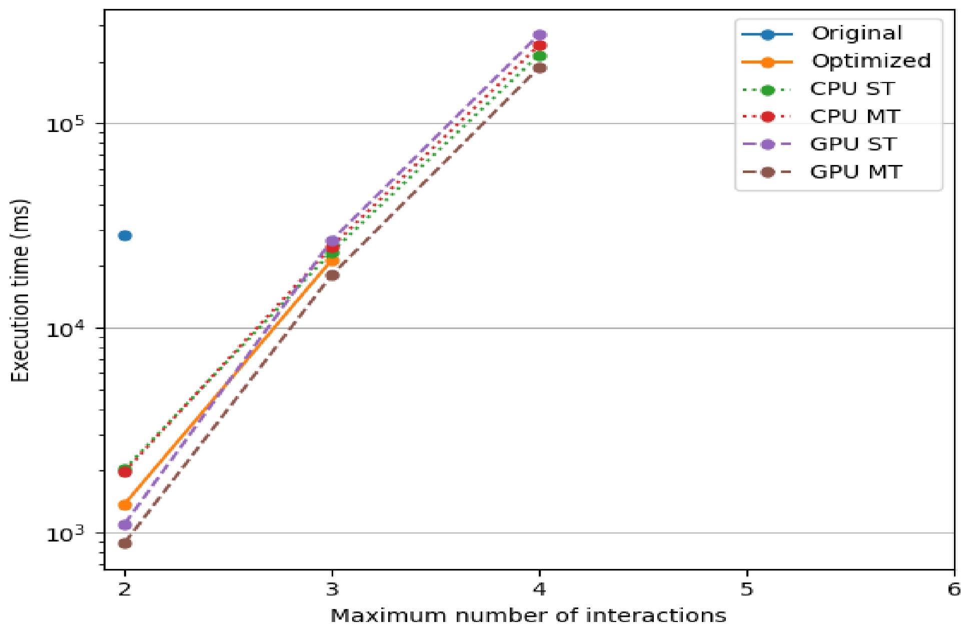

Figure 21. Propagation time of the implementations on System G in the Etu-Lyötty scene when diffractions are limited to 2.

Table 3. Benchmarks of different implementations on System R with the limits of 3 interactions and 1 diffraction, and a buffer size of 10000.

Implementation	Propagation Time (ms)
Original	11321
Optimized	3890
CPU ST	3257
CPU MT	2646
GPU ST	8836
GPU MT	2000

The optimized version did not have a significant difference in the execution time in the case of zero diffractions. On System G, the execution time is only slightly faster when compared to the original implementation with the different number of interactions. We can see large improvement when the diffraction limit is increased. The improvement is caused by the increase in the GPU occupancy, as the diffractions cause a large increase in the number of valid interactions, and thus, more invocations for the propagation kernel are generated. On System R, the optimized version was only ran with the limits of three interactions and one diffraction. When comparing to the original implementation, the performance improvement was about the same as for System G. In terms of execution time, System R was nearly three times faster.

On System G, CPU ST on average outperformed the original and optimized implementation. CPU MT was slower than CPU ST, which is quite odd considering that asynchronous CUDA operations were used. System G has four cores, thus, the launching was running on four different threads. Each thread had its own complete context, meaning that no synchronization was needed between the threads until the propagation on each thread had finished its work.

System R outperformed the original and optimized version as well. The launching was running on six threads, as System R has six CPU cores. Interestingly, the MT version greatly outperformed the ST version. The System G results indicate that the launching is not causing overhead, as there is no significant difference between the ST and MT results. However, the benchmarks conducted on System R suggests great improvement from MT launching.

The GPU MT implementation was the most performant implementation. On System G, GPU ST was about as fast as CPU ST. Unlike CPU MT, GPU MT sustained great benefit from MT launching. GPU MT was the most performant version for System R as well. For some reason, on System R, GPU ST was the slowest optimized version. Some undocumented experiments were done to hunt down the possible cause, but nothing was found. As GPU MT was the fastest on both setups, it was decided as the final optimized version that would be further developed to include path hashing and double buffering.

In the original, optimized, and the initial chaining implementations, the valid paths are first sorted followed by a simple removal of duplicate paths in an iterative fashion. The execution time of the sorting-based duplicate path removal is presented in Figure 23 and the details can be found in Table 5 of Appendix 1. The benchmarks were performed with GPU MT on System G.

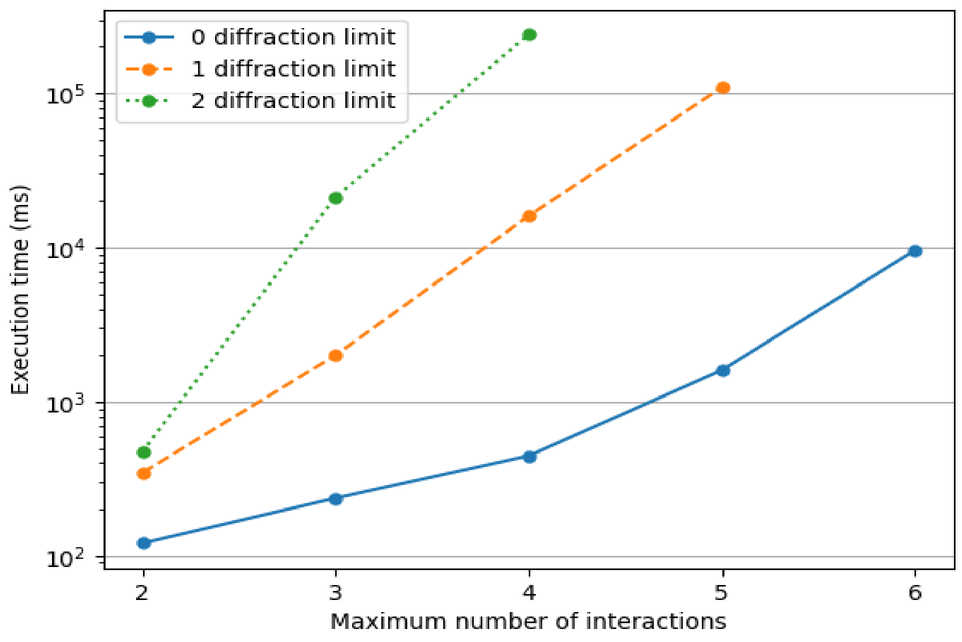

Figure 22. Propagation time of GPU MT with varying parameters in the Etu-Lyötty scene on System R.

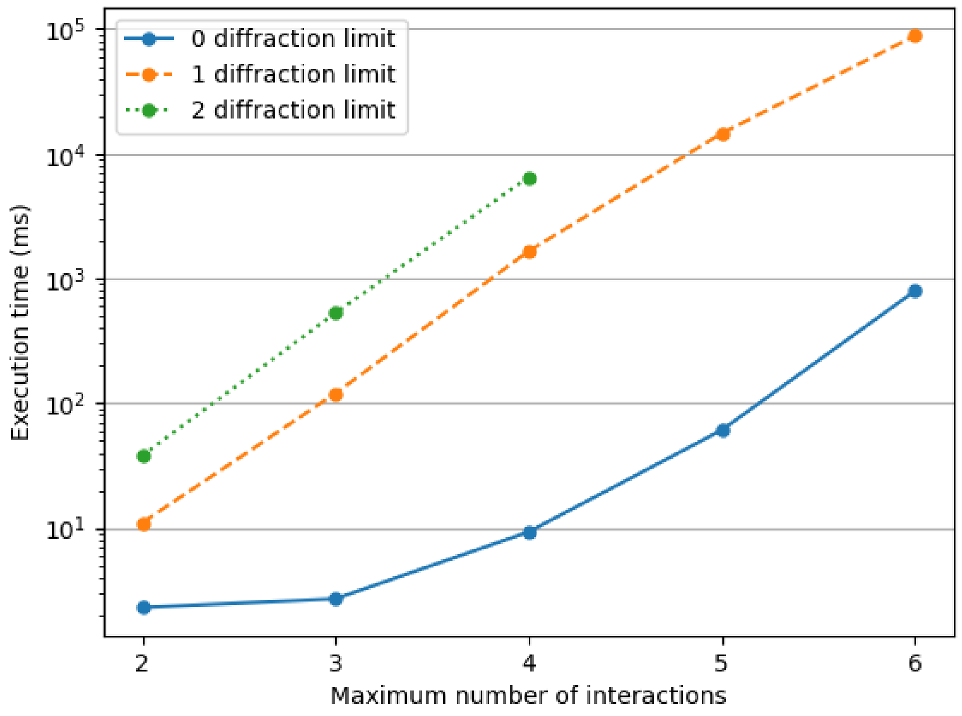

Figure 23. The execution time of the original duplicate path removal algorithm with GPU MT on System G.

Comparison of GPU MT and GPU MT Async performance can be seen in Figure 24. The details of the GPU MT Async benchmarks can be found in Table 6 of Appendix 1. The implementation contains hashing of paths to avoid duplicates, which means that when comparing the execution time to the GPU MT benchmarks, the execution time of the duplicate path removal shown in Table 5 of Appendix 1 should be added to the propagation time found in Tables 1, 2, and 3 of Appendix 1.

When comparing the GPU MT Async benchmarks to the GPU MT benchmarks, the propagation time is generally faster with few exceptions. With a high diffraction limit and interaction count, small benefit from double buffering can be seen, as there will be more CPU synchronizations causing GPU to idle on the initial implementation. However, it should be noted, that the benefit of double buffering would be higher if we saved the interactions points, as there would be more data to transfer and process. On top of the general performance increase, GPU MT Async does not have to worry about removing duplicates after the propagation. The uniqueness of each path is guaranteed through the hash-based validation during the CPU synchronizations.

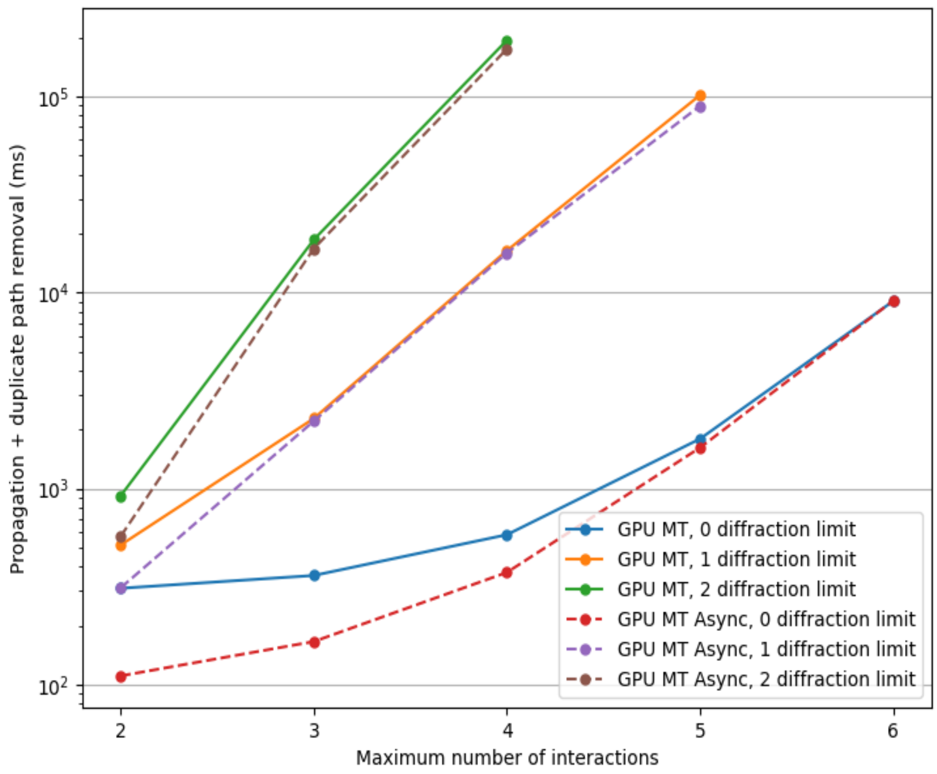

Figure 24. Comparison of the benchmarks of GPU MT and GPU MT Async in the Etu-Lyötty scene.

The final experiments were done with the interaction points saved. The original implementation benchmarks can be seen in Figure 25, and the details can be found in Table 7 of Appendix 1. The experiments were ran with 5000 and 7500 buffer sizes for both computer setups. For System G, the results can be seen in Figure 26. The details can be found in Tables 8 and 9 of Appendix 1. For System R, Figure 27 presents the benchmarks and Tables 10 and 11 of Appendix 1 contain the details.

Increasing the buffer size with GPU MT Async increased the performance on both setups, as CPU synchronizations do not have to happen as often. With double buffering, the downloading of the data happens asynchronously on a CPU thread while the GPU continues its work, thus, increasing the buffer size has no drawbacks other than higher memory usage.

When comparing the original implementation to GPU MT Async with the diffraction and interaction limits from Table 7 of Appendix 1, GPU MT Async on System G and with a buffer size of 5000 is 11.61-52.44 (on average 25.54) times faster when considering propagation and removal of duplicate paths, presented in Figure 29. The details for the execution speed improvement can be found in Table 19 of Appendix 1.

System R was slower than System G setup with higher interaction and diffraction limits, presented in Figure 28. Considering that the GPU in System R is a newer generation and higher tier graphics card, further investigation should be done.

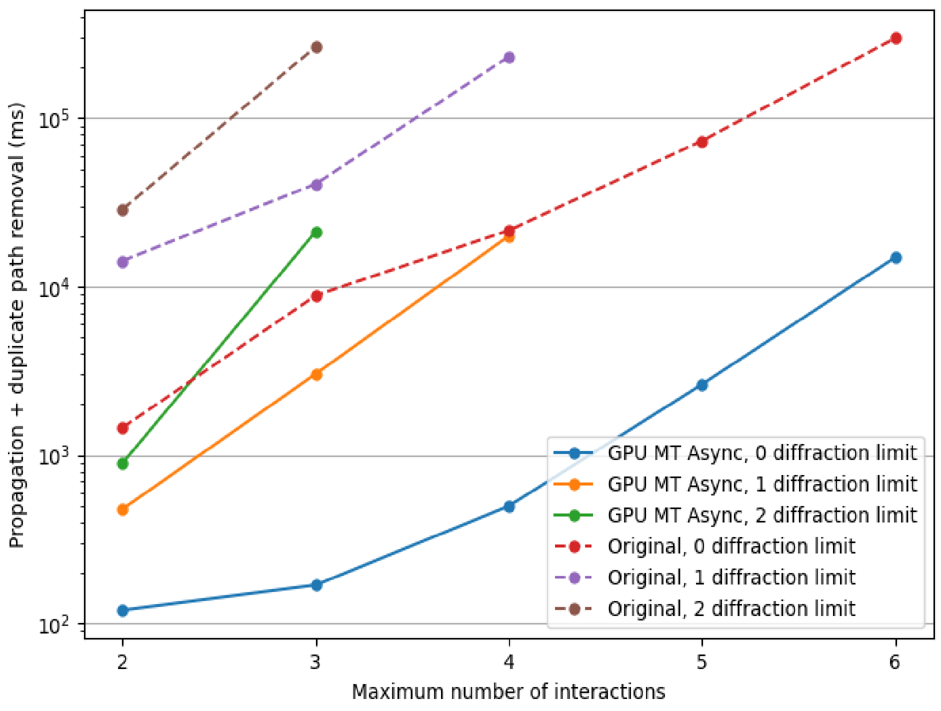

Figure 25. Original versus GPU MT Async in the Etu-Lyötty scene on System G and with a buffer size of 5000. The interaction points are saved.

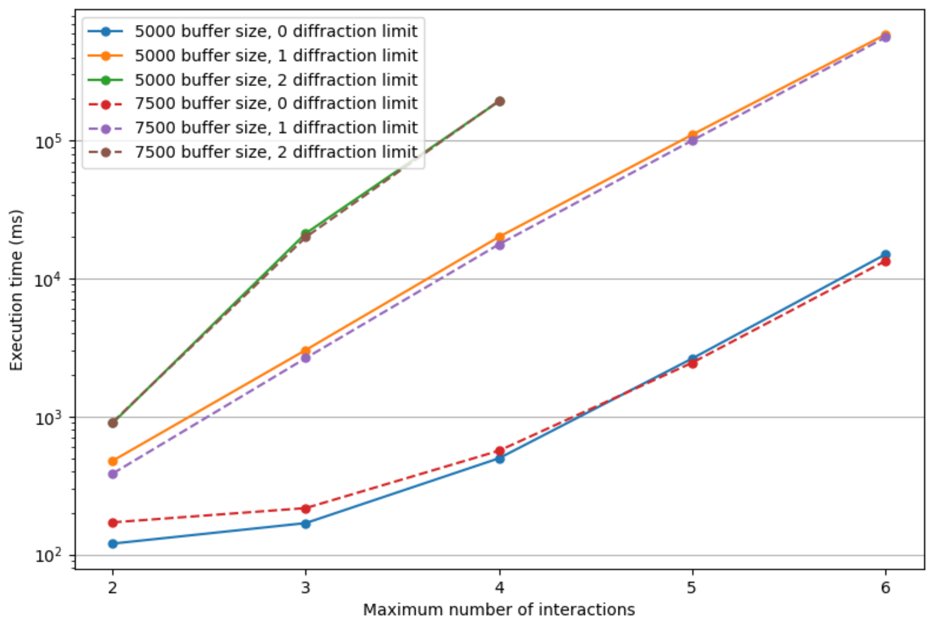

Figure 26. System G GPU MT Async performance with 5000 and 7500 buffer sizes in the Etu-Lyötty scene.

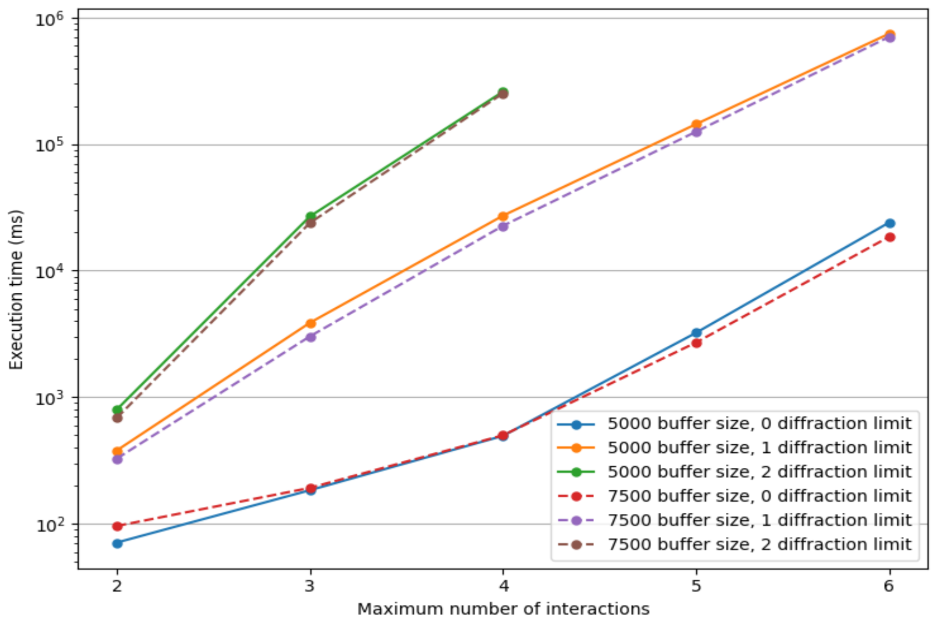

Figure 27. System R GPU MT Async performance with 5000 and 7500 buffer sizes in the Etu-Lyötty scene.

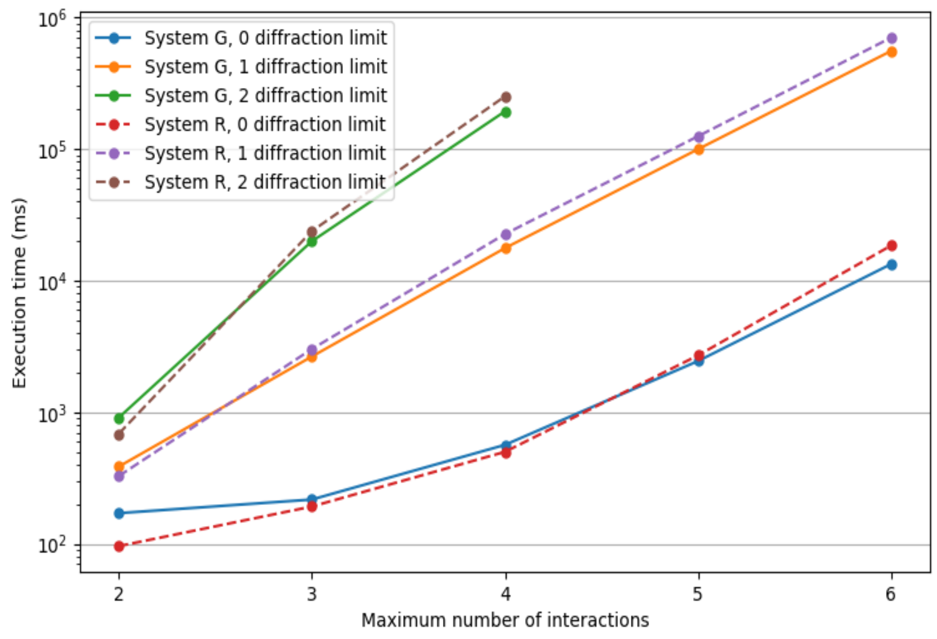

Figure 28. System G versus System R propagation time in the Etu-Lyötty scene with GPU MT Async and with a buffer size of 7500.

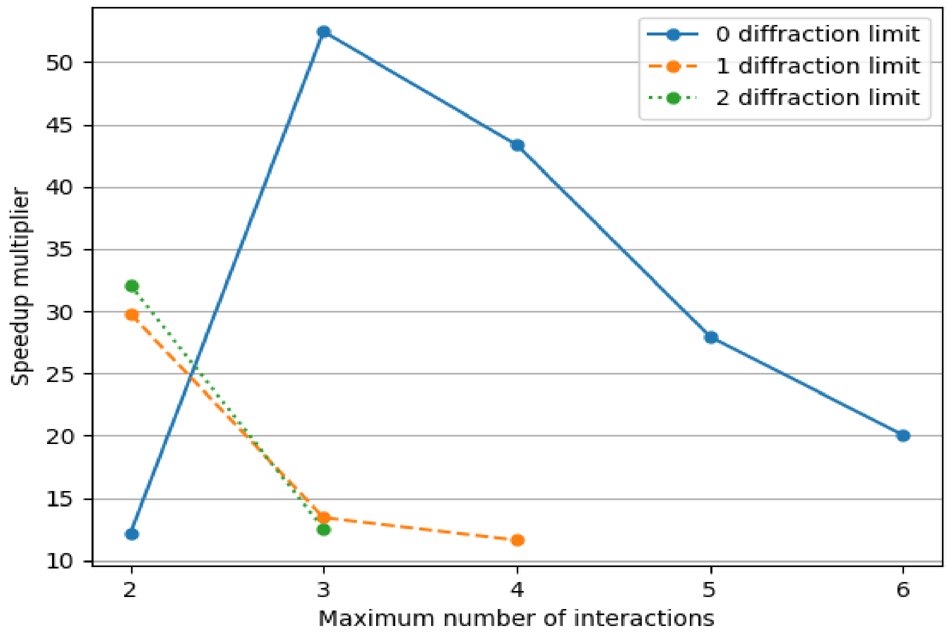

Figure 29. Speedup of GPU MT Async on System G in the Etu-Lyötty scene when compared to the original implementation.

5.1.2. Indoor Scene

The CWC corridor experiments were conducted with 1×1m discretization for walls which resulted in 611 tiles and 245 diffraction edges. The scene has one TX and a RX grid of 53 points. 110GHz carrier frequency was used, and -100dB power threshold for the field vector powers.

Table 4 contains general path statistics for this scenario. In the fourth column, the number of RX points receiving at least one path is given, and in the fifth column, the average number of paths for those RX points. Figure 30 contains benchmarks of the original implementation and GPU MT Async, and Figure 31 contains the GPU MT Async benchmarks conducted on both computer setups. The details can be found in Tables 12 and 13 of Appendix 1 for System G, and in Table 14 of Appendix 1 for System R.

As presented in Figure 31, GPU MT Async on System R was outperformed with higher interaction and diffraction limits by System G which has older hardware. One interesting exception is the case with the limits of six interactions and one diffraction. Out of the documented cases, it was the most computationally expensive case, resulting in over 4 million unique paths. System G was 1.78-7.26 (on average 4.02) times faster with GPU MT Async when compared to the original implementation with the specified interaction and diffraction limits presented in Figures 30 and 32. The details for the execution speed improvement can be found in Table 20 of Appendix 1.

Table 4. Path statistics of the CWC corridor experiments.

#Ia	Diffraction Limit	#Paths	#Receivers	#Paths Per Receiver
2	0	318	21	15.1
3	0	1798	28	64.2
4	0	10569	53	199.4
5	0	62476	53	1178.8
6	0	360188	53	6796
2	1	2521	53	47.6
3	1	21410	53	404
4	1	148214	53	2796.5
5	1	858414	53	16196.5
6	1	4053568	53	76482.4
2	2	4535	53	85.6
3	2	41040	53	774.3
4	2	257086	53	4850.7

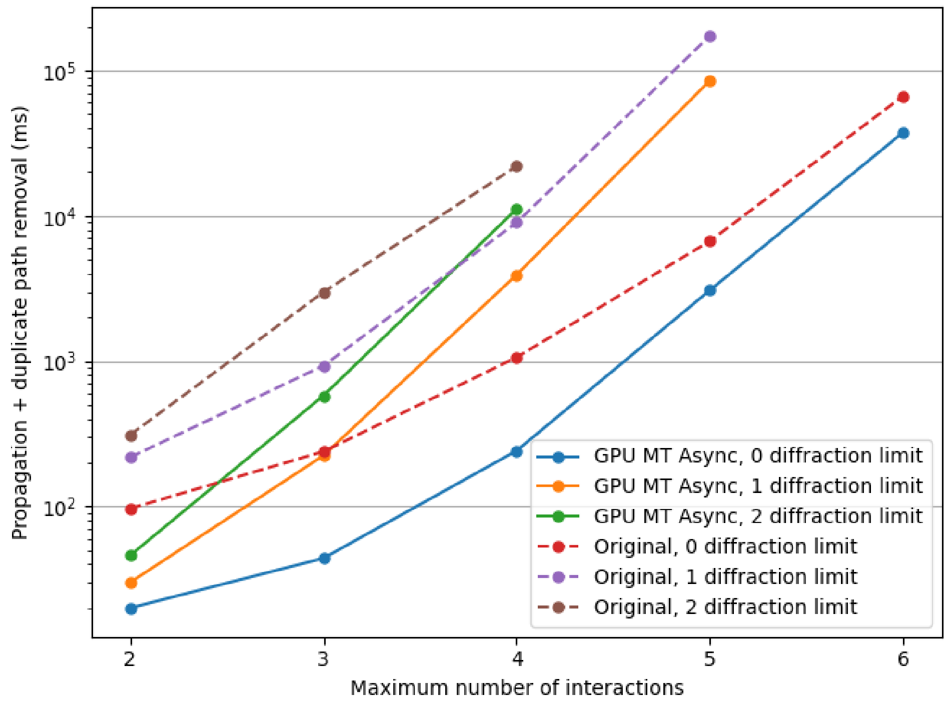

Figure 30. Original versus GPU MT Async in the CWC corridor scene on System G and with a buffer size of 5000. The interaction points are saved.

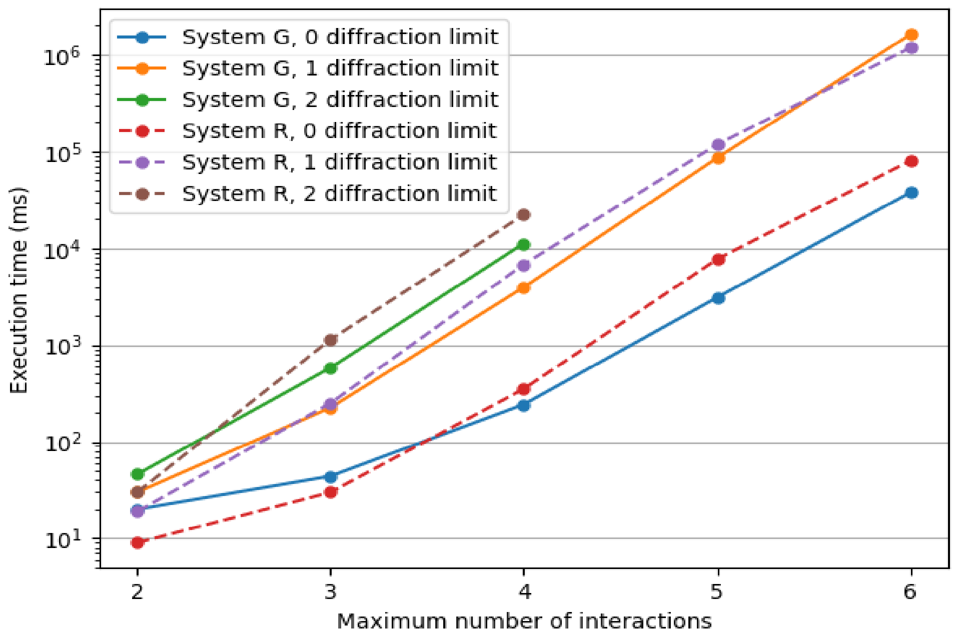

Figure 31. System G versus System R in the CWC corridor scene with GPU MT Async and with a buffer size of 5000.

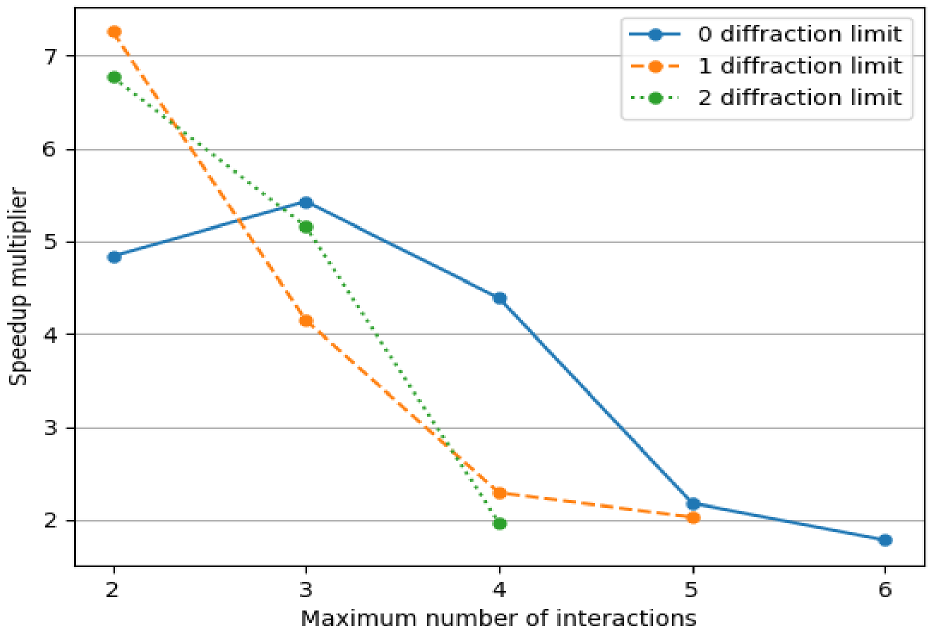

Figure 32. Speedup of GPU MT Async on System G when compared to the original implementation in the CWC corridor scene.

5.2. Path Refinement

For the path refinement experiments, CPU and GPU execution time of the refinement implementation will be benchmarked on System G with the paths from the indoor and outdoor scenes gathered from the coarse path search experiments. The optimal paths will then be computed by the path refiner to find the paths fulfilling Fermat's principle of least time.

The paths for Etu-Lyötty outdoor scene were saved from the benchmarks that can be seen in Table 8 of Appendix 1. For the CWC corridor indoor scene, the benchmarks that generated the paths are presented in Table 13 of Appendix 1.

The path refinement experiments were conducted with γ of 0.5, the number of maximum iterations was 500, and the convergence threshold was $1\mathrm{e}{-5}$.

5.2.1. Outdoor Scene

The Etu-Lyötty benchmarks can be seen in Figure 33 and the details can be found in Table 15 of Appendix 1. Figure 34 presents the percentage of converged paths, and the percentage of remaining converged paths after validation can be seen in Figure 35. The details for converged and validated paths can be found in Tables 16 and 21 of Appendix 1. As expected, the GPU-based refining implementation was faster. 88.0-99.8% (on average 96.7%) of paths converged and 31.5-91.2% (on average 62.1%) of the converged paths remained after validation.

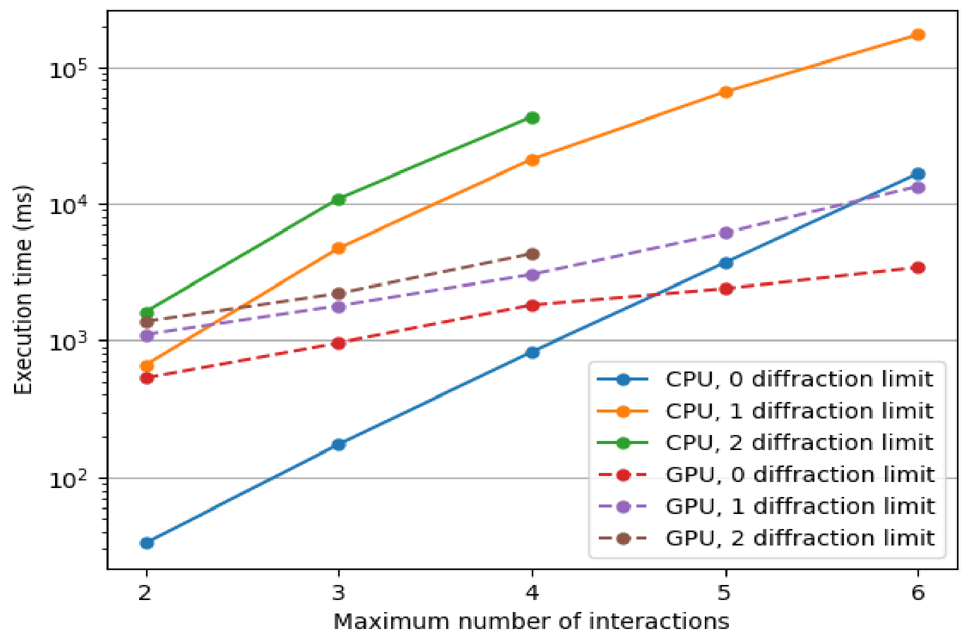

Figure 33. CPU- and GPU-based path refiner performance in the Etu-Lyötty scene.

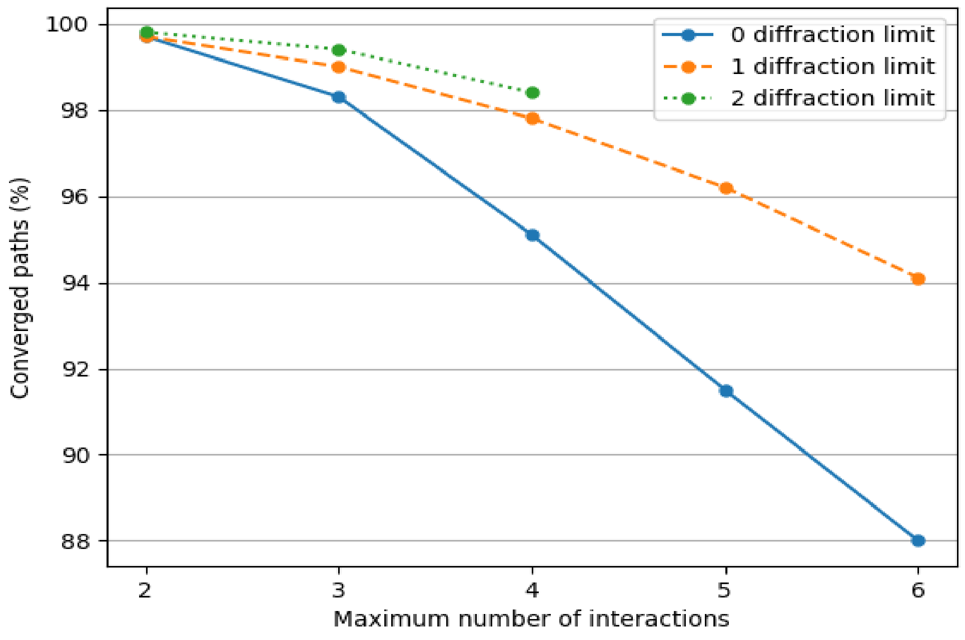

Figure 34. Converged paths remaining after refinement in the Etu-Lyötty scene.

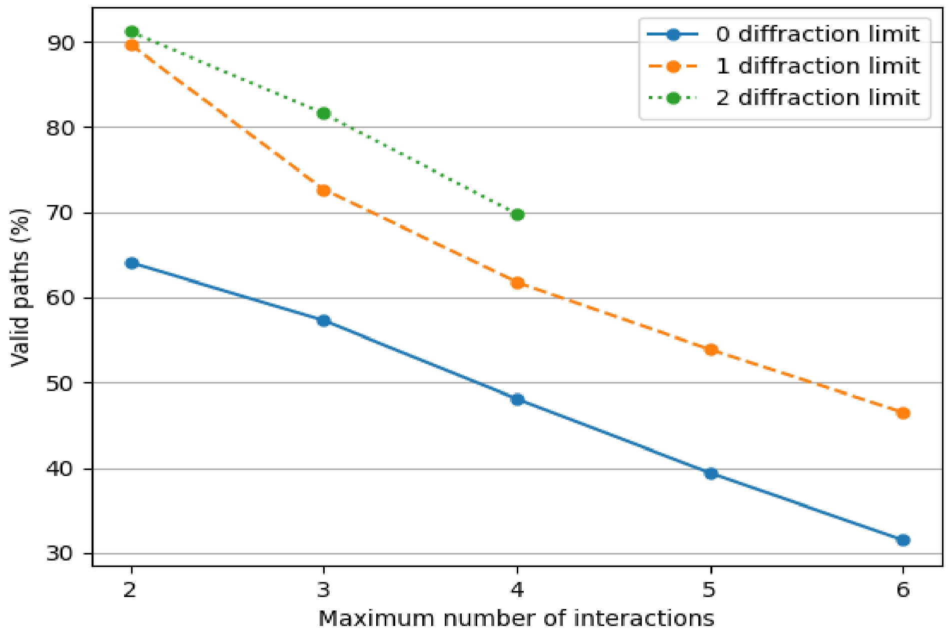

Figure 35. Remaining converged paths after validation in the Etu-Lyötty scene.

5.2.2. Indoor Scene

For the CWC corridor indoor scene, the benchmarks are presented in Figure 36 and the details are in Table 17 of Appendix 1. 98.3-100% (on average 99.6%) of the initial paths converged and 1.71-91.2% (on average 30.0%) of the converged paths remained after validation. The percentage of paths that converged is presented in Figure 37, and the percentage of remaining converged paths after validation is presented in Figure 38. The details for converged and validated paths can be found in Tables 18 and 22 of Appendix 1.

The validation percentages demonstrate the importance of the validation after the refinement, as many of the computed paths become unsuitable due to occlusion. The validation percentages in the CWC corridor experiments are lower than in the Etu-Lyötty experiments. The CWC corridor scene has a lot more details in the geometry, which leads to more paths being discarded. Another reason for the low CWC corridor scene validation percentages comes from the infinity bouncing problem of SLVA described in Section 4.4.2, as in closed small scenes such as the CWC corridor scene, the problem occurs more frequently, resulting in unsuitable paths. In both scenes, the benefit of GPU-based refining becomes noticeable when the number of interactions and diffractions is increased.

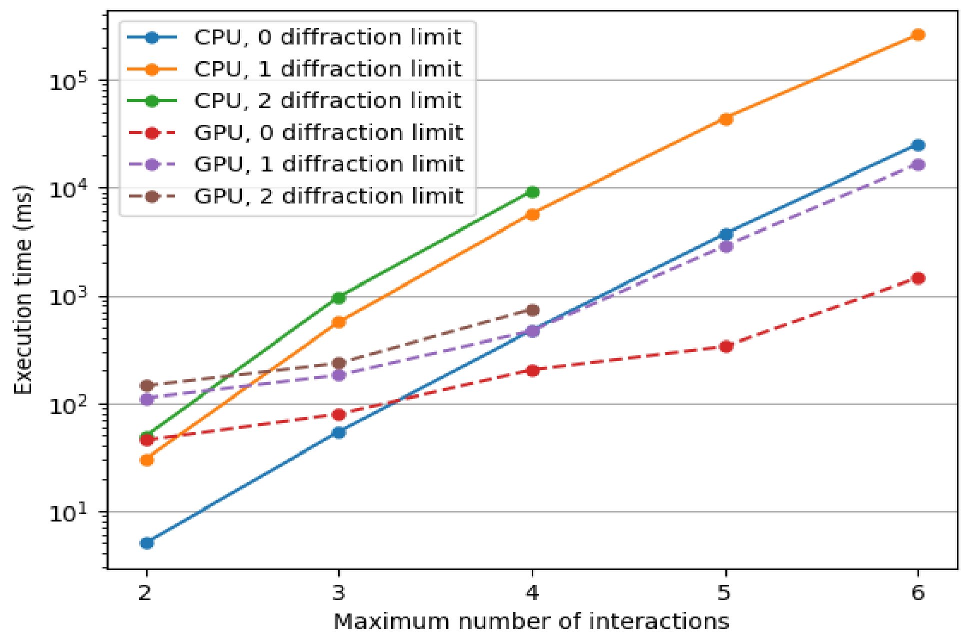

Figure 36. CPU- and GPU-based path refiner performance in the CWC corridor scene.

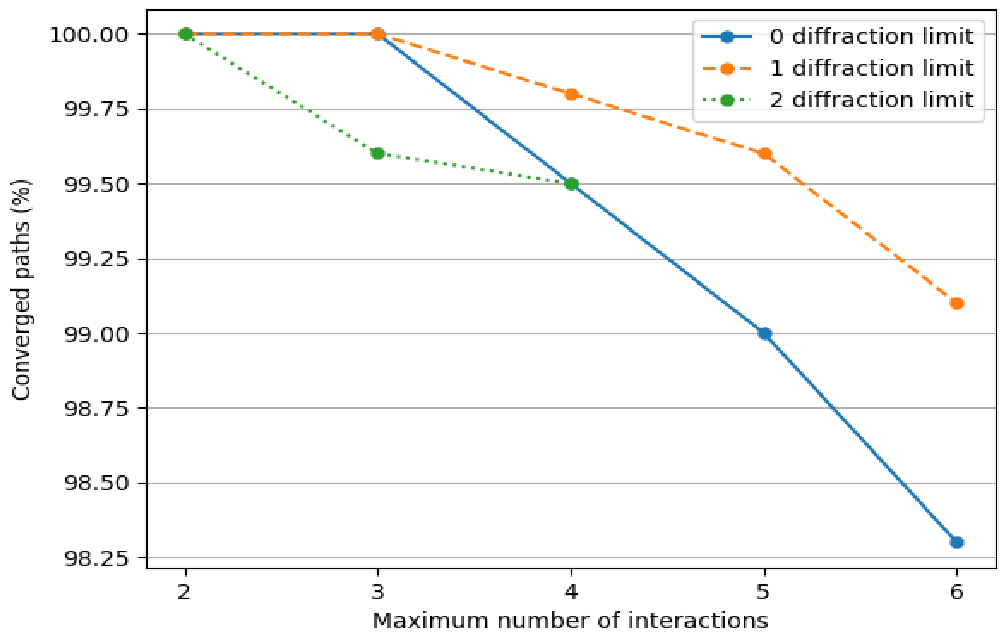

Figure 37. Converged paths remaining after refinement in the CWC corridor scene.

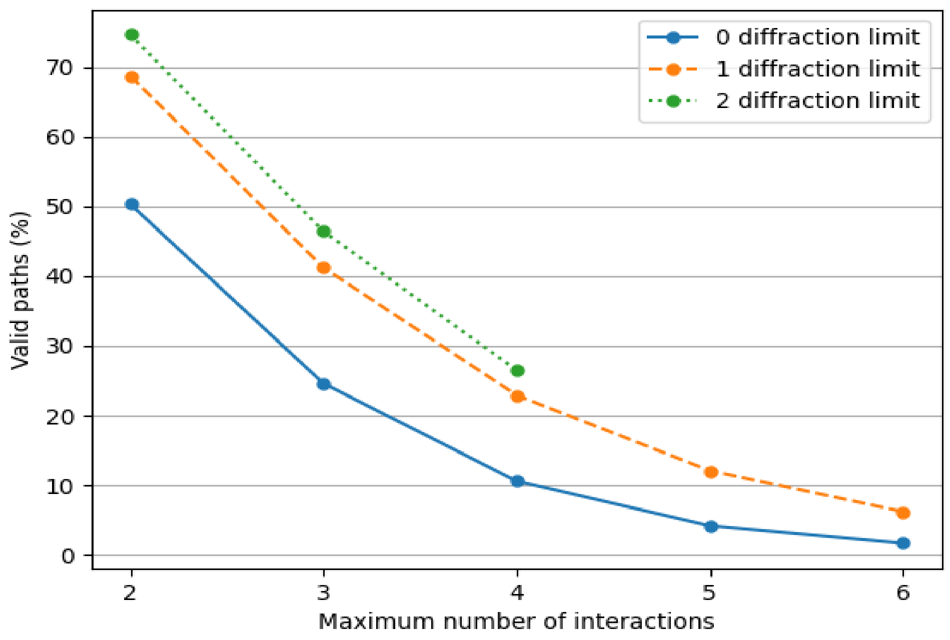

Figure 38. Remaining converged paths after validation in the CWC corridor scene.

6. DISCUSSION

6.1. Results

GPU MT Async on System G was on average over 25 times faster in the outdoor Etu-Lyötty scene, and on average over 4 times faster in the indoor CWC corridor scene, as presented in Chapter 5. Comparing the performance to other implementations is not viable, as nearly all implementations use their own 3D models. Different implementations also include varying amount of time consuming pre-processing measures, which might greatly impact the path searching phase. Thus, we have to assess the developed implementations by considering the found optimization methods and considerations presented in Chapters 2 and 3.

RT-based algorithms are highly divergent by nature due to multiple interaction types and the possibility of a ray not hitting anything. Thus, it is important to minimize the divergence where its possible, as mentioned in 3.1.2. Divergence was considered in the chaining implementations by handling the different interaction types separately, which reduces the divergence of the kernel launches. Each kernel for the parents still results in an invocation for all the relevant discretized elements as described in Section 4.3.2, which might cause a small performance hit. Fixing this divergence problem would not come without a cost, as separating the interaction types into separate visibility matrix pairs would increase the memory usage. As an example, if the first interaction type is reflection, and we were running the reflection kernel for determining the next valid reflections, we could have a visibility matrix between all the reflecting elements, which could then be used in the first-level visibility analysis.

When comparing the results of indoor and outdoor performance gains presented in Tables 19 and 20 of Appendix 1, it can be seen that GPU MT Async performs better in the outdoor scenarios. This is most likely caused by the additional computations required in the outdoor scene, as there are more tiles and edges that need to be checked for visibility.

In the indoor scene, the overhead of launching a kernel seems to become quite high when comparing to the computation of the following interactions. For example, the case with 5 interactions and 1 diffraction limit in the original implementation, presented in Table 12 of Appendix 1, results in faster propagation time. However, in this case the removal of duplicates is nearly twice as slow as the time required for propagation.

The potential overhead of launching could be reduced with Vulkan API, as the synchronization between kernel executions is explicitly determined by the user, which means that the launched kernels in each launching context would potentially be executing concurrently [65]. In CUDA, all kernels launched within a same CUDA stream are guaranteed to be executed in the order they were launched [66].

For both scenes, increasing the number of diffractions had a positive impact on the converged paths passing the validation. This is most likely caused by the fact that diffractions can only move along the edge, unlike reflections that can move along the orthonormal vectors of the surface normal. As a result, reflecting points are more likely to move outside the wall it was originally in, thus, failing the validation.

The difference between the Etu-Lyötty and CWC corridor average validation percentage stems from the scene complexity. We can see in Figure 18 that the CWC corridor scene has a lot of details. For example, every door along the corridor is a recessed door, which could easily cause the validation to fail. Etu-Lyötty scene on the other hand mostly consists of large planes, as we can see in Figure 17. Another factor is the infinity bounce problem of SLVA described in Section 4.4.2, which often results in unsuitable refined paths.

6.2. Future Work

Implementing support for refraction would be useful for indoor environments, as the indoor models often contain a lot more details than the outdoor scenes, which allows for correctly approximating the contribution of refraction. Refraction might also be the only way to reach a RX in an indoor scene. For large outdoor scenes in urban areas, simulating refraction correctly is difficult, as modeling large buildings from the inside is extremely time consuming and hard, even if access to the floor plans is granted. Modeling large buildings from the inside also significantly increases the complexity of the scene. Another reason to disregard refraction in urban outdoor scenes is that the buildings are large, which means that the contribution of refraction to the field strength might be minimal due to the attenuation.

The effect of diffuse scattering in NLOS areas in urban environments was deemed significant in [17], as mentioned in Section 2.1.1. Thus, implementing support for diffuse scattering would be beneficial. Supporting it would also increase the complexity of the models as the roughness of materials would have to be defined.

Researching the efficiency of hardware accelerated RT could be interesting, as hardware accelerated RT has been gaining support since 2018 on modern hardware [55]. It would be interesting to see if the performance of RL could be improved by utilizing RT cores in the coarse path search. Of course, due to the nature of diffuse scattering and diffractions some form of discretization is needed, but perhaps the performance of querying the valid scatterers or diffracting elements could be increased by hardware accelerated RT on modern hardware.

The current tile-based discretization model is suitable for scenes where a low amount of detail is acceptable, such as in large urban outdoor environments. Each object has to be represented using tiles, which leads to crude approximations. In the indoor scenes where the details are important, support for round and curved objects is ideal, which is limited with the tile-based discretization. Thus, the discretization could be modified to be triangle- or point-based, allowing for the introduction of triangle-based models without limitations. Changing the discretization would require quite a bit of work, as many components such as the automatic detection of diffraction edges assume models that are suitable for tile-based discretization. Another possible issue is that the increased number of details would most likely result in a decrease of optimal paths passing the validation phase of the path refiner.

The current gradient descent implementation presented in Section 4.4.1 has step size halving every 25 iterations. An algorithm for calculating the step size, such as backtracking line-search should be integrated for a more robust approach.

In terms of optimization, the usage of shared GPU memory could be researched, as access to the shared GPU memory is much faster than global access [67]. Coming up with more ways to reduce divergence should be researched as well.

Vulkan API could potentially bring performance improvements, as well as provide support on platforms that do not support CUDA. On top of the potential advantages in terms of performance, Vulkan would also bring compute, rendering, and RT under the same API that could be abstracted into a reusable library, which would speed up the development of future tools and features for radio channel modeling.

7. SUMMARY

In this thesis, tools for radio channel propagation modeling were optimized and developed. The general theory of radio channel propagation modeling in the context of RT was studied. We optimized the existing environment discretization-based path search implementation by developing multiple new optimized versions, which utilize features such as MT and GPU-based launching. As the second task, we developed a path refinement solution for computing the optimal paths fulfilling the Fermat's principle of least time, as the paths generated by the path search are coarse approximations due to the discretized environment. We integrated the path refiner with RT-based path validation, which is needed as many of the paths can be unsuitable due to the geometric characteristics of the scene.

Experiments were conducted on two different computer setups with the different optimized path search versions, and the best performing version, as well as the path refiner were evaluated using an indoor and an outdoor model. The best performing path search version was on average 25 times faster in the outdoor scene, and on average 4 times faster in the indoor scene when compared to the original implementation. The path refiner found optimal paths on average for over 96% of the paths in the outdoor scene, and for over 99% in the indoor scene. Out of these paths, on average about 62% passed the validation phase in the outdoor case, and around 30% in the indoor case.

In future work, refraction support could be developed, as it might be the only way to reach a RX, and thus, it plays an important role in indoor simulations. Scattering support would be beneficial for outdoor scenarios, as its effect to the channel was deemed significant in the literature. Studies should also address improvements and extensions in dealing with the geometry, increasing the computational efficiency, and providing a better cross-platform support.

8. REFERENCES

[1] Yun Z. & Iskander M.F. (2015) Ray tracing for radio propagation modeling: Principles and applications. IEEE Access 3, pp. 1089–1100.

[2] Degli-Esposti V. (2014) Ray tracing propagation modelling: Future prospects. In: The 8th European Conference on Antennas and Propagation (EuCAP 2014), IEEE, pp. 2232–2232.

[3] Fuschini F., Vitucci E.M., Barbiroli M., Falciasecca G. & Degli-Esposti V. (2015) Ray tracing propagation modeling for future small-cell and indoor applications: A review of current techniques. Radio Science 50, pp. 469–485.

[4] Schiller M., Knoll A., Mocker M. & Eibert T. (2015) GPU accelerated ray launching for high-fidelity virtual test drives of vanet applications. In: 2015 International Conference on High Performance Computing & Simulation (HPCS), IEEE, pp. 262–268.

[5] AI4Green Celtic. `https://www.celticnext.eu/project-ai4green/`. Accessed: 28.4.2022.

[6] Lu J.S., Vitucci E.M., Degli-Esposti V., Fuschini F., Barbiroli M., Blaha J.A. & Bertoni H.L. (2018) A discrete environment-driven GPU-based ray launching algorithm. IEEE Transactions on Antennas and Propagation 67, pp. 1180–1192.

[7] Born M. & Wolf E. (2013) Principles of optics: electromagnetic theory of propagation, interference and diffraction of light. Elsevier.

[8] electronics-notes Radio Wave Reflection. `https://www.electronics-notes.com/articles/antennas-propagation/propagation-overview/radio-em-wave-reflection.php`. Accessed: 18.4.2022.

[9] The University of British Columbia The Law of Refraction . `https://personal.math.ubc.ca/~cass/courses/m309-01a/chu/Fundamentals/snell.htm`. Accessed: 18.4.2022.

[10] Keller J.B. (1962) Geometrical theory of diffraction. Josa 52, pp. 116–130.

[11] Kouyoumjian R.G. & Pathak P.H. (1974) A uniform geometrical theory of diffraction for an edge in a perfectly conducting surface. Proceedings of the IEEE 62, pp. 1448–1461.

[12] Luebbers R. (1984) Finite conductivity uniform GTD versus knife edge diffraction in prediction of propagation path loss. IEEE Transactions on Antennas and Propagation 32, pp. 70–76.

[13] Tiberio R., Pelosi G. & Manara G. (1985) A uniform GTD formulation for the diffraction by a wedge with impedance faces. IEEE Transactions on Antennas and Propagation 33, pp. 867–873.

[14] Demetrescu C., Constantinou C. & Mehler M. (1997) Scattering by a right-angled lossy dielectric wedge. IEE Proceedings - Microwaves, Antennas and Propagation 144, pp. 392–396.

[15] Ahluwalia D., Lewis R. & Boersma J. (1968) Uniform asymptotic theory of diffraction by a plane screen. SIAM Journal on Applied Mathematics 16, pp. 783–807.

[16] Carluccio G. & Albani M. (2008) An efficient ray tracing algorithm for multiple straight wedge diffraction. IEEE Transactions on Antennas and Propagation 56, pp. 3534–3542.

[17] Rasekh M.E., Shishegar A.A. & Farzaneh F. (2009) A study of the effect of diffraction and rough surface scatter modeling on ray tracing results in an urban environment at 60 GHz. In: 2009 First Conference on Millimeter-Wave and Terahertz Technologies (MMWaTT), IEEE, pp. 27–31.

[18] Hammoudeh A., Pugliese J.P., Sanchez M. & Grindrod E. (1999) Comparison of reflection mechanisms from smooth and rough surfaces at 62 GHz. In: IEE National Conference on Antennas and Propagation, IEE, pp. 144–147.

[19] Gschwendtner B., Wölfle G., Burk B. & Landstorfer F. (1995) Ray tracing vs. ray launching in 3-D microcell modelling. In: First European Personal and Mobile Communications Conference, IEE, pp. 74–79.

[20] Imai T. (2017) A survey of efficient ray-tracing techniques for mobile radio propagation analysis. IEICE Transactions on Communications E100.B, pp. 666–679.

[21] Funkhouser T., Tsingos N., Carlbom I., Elko G., Sondhi M., West J.E., Pingali G., Min P. & Ngan A. (2004) A beam tracing method for interactive architectural acoustics. The Journal of the Acoustical Society of America 115, pp. 739–756.

[22] Kreuzgruber P., Unterberger P. & Gahleitner R. (1993) A ray splitting model for indoor radio propagation associated with complex geometries. In: IEEE 43rd Vehicular Technology Conference, IEEE, pp. 227–230.

[23] Fortune S. (1998) Efficient algorithms for prediction of indoor radio propagation. In: VTC'98. 48th IEEE Vehicular Technology Conference. Pathway to Global Wireless Revolution (Cat. No. 98CH36151), vol. 1, IEEE, vol. 1, pp. 572–576.

[24] Chen S.H. & Jeng S.K. (1996) SBR image approach for radio wave propagation in tunnels with and without traffic. IEEE Transactions on Vehicular Technology 45, pp. 570–578.

[25] Tan S. & Tan H. (1996) A microcellular communications propagation model based on the uniform theory of diffraction and multiple image theory. IEEE Transactions on Antennas and Propagation 44, pp. 1317–1326.

[26] Agelet F.A., Formella A., Rabanos J.H., De Vicente F.I. & Fontan F.P. (2000) Efficient ray-tracing acceleration techniques for radio propagation modeling. IEEE Transactions on Vehicular Technology 49, pp. 2089–2104.

[27] Yun Z., Iskander M.F. & Zhang Z. (2000) Fast ray tracing procedure using space division with uniform rectangular grid. Electronics Letters 36, pp. 895–897.

[28] Hoppe R., Wolfle G. & Landstorfer F. (1999) Accelerated ray optical propagation modeling for the planning of wireless communication networks. In: RAWCON 99. 1999 IEEE Radio and Wireless Conference (Cat. No. 99EX292), IEEE, pp. 159–162.

[29] Lott M. & Forkel I. (2001) A multi-wall-and-floor model for indoor radio propagation. In: IEEE VTS 53rd Vehicular Technology Conference, Spring 2001. Proceedings (Cat. No. 01CH37202), vol. 1, IEEE, vol. 1, pp. 464–468.

[30] Teh C.H., Chung B. & Lim E. (2019) Multilayer wall correction factors for indoor ray-tracing radio propagation modeling. IEEE Transactions on Antennas and Propagation 68, pp. 604–608.

[31] Hossain F., Geok T.K., Rahman T.A., Hindia M.N., Dimyati K., Ahmed S., Tso C.P. & Abd Rahman N.Z. (2019) An efficient 3-D ray tracing method: Prediction of indoor radio propagation at 28 GHz in 5G network. Electronics 8, p. 286.

[32] Tan J., Su Z. & Long Y. (2015) A full 3-D GPU-based beam-tracing method for complex indoor environments propagation modeling. IEEE Transactions on Antennas and Propagation 63, pp. 2705–2718.

[33] Wang G.y., Liu Y.j., Zhang X.j., Chen Z.p. et al. (2016) Study on the outdoor wave propagation at 28GHz by ray tracing method. In: 2016 IEEE International Conference on Microwave and Millimeter Wave Technology (ICMMT), vol. 1, IEEE, vol. 1, pp. 476–478.

[34] Nguyen H.C., MacCartney G.R., Thomas T., Rappaport T.S., Vejlgaard B. & Mogensen P. (2014) Evaluation of empirical ray-tracing model for an urban outdoor scenario at 73 GHz E-band. In: 2014 IEEE 80th Vehicular Technology Conference (VTC2014-Fall), IEEE, pp. 1–6.

[35] Friis H.T. (1946) A note on a simple transmission formula. Proceedings of the IRE 34, pp. 254–256.

[36] Degli-Esposti V., Fuschini F., Vitucci E.M., Barbiroli M., Zoli M., Tian L., Yin X., Dupleich D.A., Müller R., Schneider C. et al. (2014) Ray-tracing-based mm-wave beamforming assessment. IEEE Access 2, pp. 1314–1325.

[37] Bertoni H.L. (1999) Radio Propagation for Modern Wireless Systems. Pearson Education.

[38] Mani F. (2012) Improved ray-tracing for advanced radio propagation channel modeling. Ph. D. dissertation .

[39] Oestges C., Clerckx B., Raynaud L. & Vanhoenacker-Janvier D. (2002) Deterministic channel modeling and performance simulation of microcellular wide-band communication systems. IEEE Transactions on Vehicular Technology 51, pp. 1422–1430.

[40] Rappaport T.S. et al. (1996) Wireless Communications: Principles and Practice, vol. 2. Prentice Hall PTR New Jersey.

[41] Degli-Esposti V., Fuschini F., Vitucci E.M. & Falciasecca G. (2007) Measurement and modelling of scattering from buildings. IEEE Transactions on Antennas and Propagation 55, pp. 143–153.

[42] Series P. (2013) Propagation data and prediction methods required for the design of terrestrial broadband radio access systems operating in a frequency range from 3 to 60 GHz. Recommendation ITU-R , pp. 1410–1415.

[43] Nickolls J. & Dally W.J. (2010) The GPU computing era. IEEE micro 30, pp. 56–69.

[44] NVIDIA Corporation CUDA Toolkit Documentation. `https://docs.nvidia.com/cuda/`. Accessed: 10.4.2022.

[45] Brodtkorb A.R., Hagen T.R., Schulz C. & Hasle G. (2013) GPU computing in discrete optimization. part i: Introduction to the GPU. EURO Journal on Transportation and Logistics 2, pp. 129–157.

[46] NVIDIA Corporation Microsoft's DirectCompute API. `https://developer.nvidia.com/directcompute`. Accessed: 10.4.2022.

[47] The Khronos Group The OpenCL Specification. `https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_API.html`. Accessed: 10.4.2022.

[48] The Khronos Group Vulkan Specification. `https://www.khronos.org/registry/vulkan/specs/1.3-extensions/html/index.html`. Accessed: 10.4.2022.

[49] Mammeri N. & Juurlink B. (2018) Vcomputebench: A Vulkan benchmark suite for GPGPU on mobile and embedded GPUs. In: 2018 IEEE International Symposium on Workload Characterization (IISWC), IEEE, pp. 25–35.

[50] NVIDIA Corporation RTX Ray Tracing. `https://developer.nvidia.com/rtx/ray-tracing`. Accessed: 10.4.2022.

[51] Parker S.G., Bigler J., Dietrich A., Friedrich H., Hoberock J., Luebke D., McAllister D., McGuire M., Morley K., Robison A. et al. (2010) OptiX: a general purpose ray tracing engine. ACM Transactions on Graphics 29, pp. 1–13.

[52] OptiX 7 Delivers New Levels of Flexibility to Application Developers. `https://developer.nvidia.com/blog/optix-7-delivers-new-levels-of-flexibility-to-application-developers/`. Accessed: 21.5.2022.

[53] Xu G., Dong C., Zhao T., Yin H. & Chen X. (2021) Acceleration of shooting and bouncing ray method based on OptiX and normal vectors correction. Plos one 16, p. e0253743.

[54] Ray Tracing in Vulkan. https://www.khronos.org/news/press/vulkan-sdk-tools-and-drivers-are-ray-tracing-ready. Accessed: 10.4.2022.

[55] Developing Ray tracing content for mobile games. https://community.arm.com/arm-community-blogs/b/graphics-gaming-and-vr-blog/posts/developing-ray-tracing-content-for-mobile-games. Accessed: 6.5.2022.

[56] MediaTek partners with ARM and Tencent to bring ray tracing to mobile devices. https://www.gizchina.com/2021/10/15/mediatek-partners-with-arm-and-tencent-to-bring-ray-tracing-to-mobile-devices/. Accessed: 6.5.2022.

[57] Pyhtila J., Sangi P., Karvonen H., Berg M., Lighari R., Salonen E., Parssinen A. & Juntti M. (2019) 3-D ray tracing based GPU accelerated field prediction radio channel simulator. In: 2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring), IEEE, pp. 1–7.

[58] Pang M., Wang H., Lin K. & Lin H. (2021) A GPU-based radio wave propagation prediction with progressive processing on point cloud. IEEE Antennas and Wireless Propagation Letters 20, pp. 1078–1082.

[59] Abdellatif A. & Safavi-Naeini S. (2014) GPU accelerated channel modeling ray tracing tool. In: 2014 IEEE Radio and Wireless Symposium (RWS), IEEE, pp. 238–240.

[60] Moon B., Byun Y., Kim T.J., Claudio P., Kim H.S., Ban Y.J., Nam S.W. & Yoon S.E. (2010) Cache-oblivious ray reordering. ACM Transactions on Graphics 29, pp. 1–10.

[61] Bentley J.L. (1975) Multidimensional binary search trees used for associative searching. Communications of the ACM 18, pp. 509–517.

[62] glTF 2.0 Specification. https://www.khronos.org/registry/glTF/specs/2.0/glTF-2.0.html. Accessed: 22.5.2022.

[63] Adaptive Parallel Computation with CUDA Dynamic Parallelism. https://developer.nvidia.com/blog/introduction-cuda-dynamic-parallelism/. Accessed: 2.6.2022.

[64] Ruder S. (2016) An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747 .

[65] The Khronos Group, Understanding Vulkan Synchronization. https://www.khronos.org/blog/understanding-vulkan-synchronization. Accessed: 28.4.2022.

[66] Steve Rennich NVIDIA, CUDA C/C++ Streams and Concurrency. https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf. Accessed: 28.4.2022.

[67] Using Shared Memory in CUDA C/C++. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/. Accessed: 23.5.2022.

9. APPENDICES

Table 1. Propagation benchmarks in milliseconds of the different implementations on System G with diffractions limited to 0 and with a buffer size of 10000.

#Ia	Original	Optimized	CPU ST	GPU ST	CPU MT	GPU MT
2	640	510	664	312	657	308
3	8310	5542	941	404	916	358
4	20960	17253	1409	782	1387	573
5	72695	57600	2967	2563	3014	1734
6	Halted	209601	9243	10413	10385	8249

Table 2. Propagation benchmarks in milliseconds of the different implementations on System G with diffractions limited to 1 and with a buffer size of 10000.

#Ia	Original	Optimized	CPU ST	GPU ST	CPU MT	GPU MT
2	13617	1024	1581	771	1535	505
3	40231	10992	4445	3432	4158	2160
4	Halted	163138	17815	19245	19288	14710
5	Halted	Halted	95766	118760	106384	86973
6	Halted	Halted	Halted	Halted	Halted	Halted

Table 3. Propagation benchmarks in milliseconds of the different implementations on System G with diffractions limited to 2 and with a buffer size of 10000.

#Ia	Original	Optimized	CPU ST	GPU ST	CPU MT	GPU MT
2	28159	1366	2027	1091	1966	882
3	Halted	21408	23452	26738	24903	18081
4	Halted	Halted	216092	271137	241188	187057
5	Halted	Halted	Halted	Halted	Halted	Halted
6	Halted	Halted	Halted	Halted	Halted	Halted

Table 4. Benchmarks of GPU MT on System R in the Etu-Lyötty scene with varying parameters.

#Ia	Diffraction Limit	Buffer Size	Propagation Time (ms)
2	0	10000	121
3	0	10000	237
4	0	10000	446
5	0	10000	1603
6	0	10000	9602
2	1	10000	346
3	1	5000	2316
3	1	10000	2000
3	1	20000	2192
4	1	10000	16010
5	1	10000	109211
6	1	10000	Halted
2	2	10000	473
3	2	10000	21205
4	2	5000	234385
4	2	10000	239934
4	2	20000	255416
5	2	10000	Halted
6	2	10000	Halted

Table 5. Execution times of the duplicate path removal algorithm with GPU MT on System G in the Etu-Lyötty scene.

#Ia	Diffraction Limit	#Paths	Duplicate Path Removal Time (ms)
2	0	1170	2.3
3	0	4527	2.7
4	0	17295	9.3
2	1	33085	11
5	0	64683	61
2	2	88009	38
3	1	153041	120
6	0	242087	797
3	2	379181	530
4	1	510818	1657
4	2	1118404	6497
5	1	1286550	14621
6	1	2807669	88460

Table 6. Propagation time with GPU MT Async on System G in the Etu-Lyötty scene with hashing included and with varying parameters.

#Ia	Diffraction Limit	Buffer Size	Propagation Time (ms)
2	0	10000	111
3	0	10000	166
4	0	10000	374
5	0	10000	1618
6	0	10000	9086
2	1	10000	312
3	1	10000	2210
4	1	10000	15921
5	1	10000	89460
2	2	10000	571
3	2	5000	17130
3	2	10000	16762
4	2	5000	173815
4	2	10000	174743

Table 7. Execution times in the Etu-Lyötty scene with the original implementation on System G with a buffer size of 5000. Interaction points are saved.

#Ia	Diffraction Limit	Propagation Time (ms)	Duplicate Removal Time (ms)
2	0	1455	1.5
3	0	8861	2.2
4	0	21581	7.6
5	0	73450	58
6	0	298271	855
2	1	14240	9.5
3	1	40636	115
4	1	230517	1858
2	2	28763	35.7
3	2	265855	582

Table 8. Propagation times in the Etu-Lyötty scene with GPU MT Async on System G with a buffer size of 5000. Interaction points are included.

#Ia	Diffraction Limit	Propagation Time (ms)
2	0	120
3	0	169
4	0	498
5	0	2629
6	0	14927
2	1	478
3	1	3035
4	1	20013
5	1	110060
6	1	584607
2	2	897
3	2	21257
4	2	192136

Table 9. Propagation times in the Etu-Lyötty scene with GPU MT Async on System G with a buffer size of 7500. Interaction points are included.

#Ia	Diffraction Limit	Propagation Time (ms)
2	0	171
3	0	217
4	0	565
5	0	2451
6	0	13385
2	1	386
3	1	2651
4	1	17679
5	1	99891
6	1	557467
2	2	909
3	2	19938
4	2	193499

Table 10. Propagation times in the Etu-Lyötty scene with GPU MT Async on System R with a buffer size of 5000. Interaction points are included.

#Ia	Diffraction Limit	Propagation Time (ms)
2	0	71
3	0	184
4	0	495
5	0	3230
6	0	24014
2	1	379
3	1	3870
4	1	27183
5	1	143718
6	1	745547
2	2	797
3	2	26787
4	2	259400

Table 11. Propagation times in the Etu-Lyötty scene with GPU MT Async on System R with a buffer size of 7500. Interaction points are included.

#Ia	Diffraction Limit	Propagation Time (ms)
2	0	96
3	0	192
4	0	500
5	0	2705
6	0	18509
2	1	327
3	1	3021
4	1	22597
5	1	125282
6	1	703627
2	2	685
3	2	23792
4	2	251548

Table 12. Propagation times in the CWC corridor scene with the original implementation on System G with a buffer size of 5000. Interaction points are included.

#Ia	Diffraction Limit	Propagation Time (ms)	Duplicate Removal Time (ms)
2	0	95	1.7
3	0	237	1.8
4	0	1042	21
5	0	6101	610
6	0	40175	26697
2	1	216	1.9
3	1	904	26
4	1	7453	1551
5	1	59435	114347
6	1	408421	Halted
2	2	309	2.3
3	2	2904	84
4	2	17627	4349

Table 13. Propagation times in the CWC corridor scene with GPU MT Async on System G with a buffer size of 5000. Interaction points are included.

#Ia	Diffraction Limit	Propagation Time (ms)
2	0	20
3	0	44
4	0	242
5	0	3082
6	0	37566
2	1	30
3	1	224
4	1	3934
5	1	85529
6	1	1614140
2	2	46
3	2	579
4	2	11173

Table 14. Propagation times in the CWC corridor scene with GPU MT Async on System R with a buffer size of 5000. Interaction points are included.

#Ia	Diffraction Limit	Propagation Time (ms)
2	0	9
3	0	30
4	0	350
5	0	7678
6	0	79980
2	1	19
3	1	249
4	1	6785
5	1	117538
6	1	1174540
2	2	30
3	2	1134
4	2	22093

Table 15. Refinement and validation benchmarks in the Etu-Lyötty scene on System G.

#Ia	Diffraction Limit	Paths	CPU Based (ms)	GPU Based (ms)	Validation & EM (ms)
2	0	1170	33	532	134
3	0	4527	174	962	179
4	0	17295	827	1819	219
5	0	64683	3713	2396	252
6	0	242087	16677	3149	266
2	1	33085	662	1104	236
3	1	153401	4731	1796	264
4	1	510818	21105	3035	318
5	1	1286550	65662	6124	412
6	1	2807669	172165	13439	606
2	2	88009	1617	1378	262
3	2	379181	10878	2212	291
4	2	1118404	43275	4312	427

Table 16. Remaining paths after refinement and validation for the Etu-Lyötty scene.

#Ia	Diffraction Limit	Total Paths	Converged Paths	Validated Paths
2	0	1170	1165	747
3	0	4527	4452	2550
4	0	17295	16453	7917
5	0	64683	59215	23308
6	0	242087	213155	67230
2	1	33085	32997	29596
3	1	153401	151848	110324
4	1	510818	499700	308578
5	1	1286550	1237246	667380
6	1	2807669	2642573	1228086
2	2	88009	87852	80154
3	2	379181	376961	307673
4	2	1118404	1100826	768697

Table 17. Refinement and validation benchmarks in the CWC corridor scene on System G.

#Ia	Diffraction Limit	Paths	CPU Based (ms)	GPU Based (ms)	Validation & EM (ms)
2	0	318	5	45	52
3	0	1798	54	79	56
4	0	10569	472	203	60
5	0	62476	3713	335	72
6	0	360188	25201	1451	102
2	1	2521	30	111	58
3	1	21410	562	181	67
4	1	148213	5702	466	81
5	1	858413	43926	2866	162
6	1	4053563	260994	16557	826
2	2	4535	49	144	67
3	2	41040	964	234	76
4	2	257085	9230	739	105

Table 18. Remaining paths after refinement and validation in the CWC corridor scene.

#Ia	Diffraction Limit	Total Paths	Converged Paths	Validated Paths
2	0	318	318	160
3	0	1798	1798	442
4	0	10569	10511	1110
5	0	62476	61875	2606
6	0	360188	353906	6039
2	1	2521	2521	1732
3	1	21410	21402	8815
4	1	148213	147977	33951
5	1	858413	854876	103317
6	1	4053563	4016950	250199
2	2	4535	4521	3374
3	2	41040	40858	18970
4	2	257085	255873	67727

Table 19. Speedup of GPU MT Async with a buffer size of 5000 on System G when compared to the original implementation in the Etu-Lyötty scene.

#Ia	Diffraction Limit	Original (ms)	GPU MT Async (ms)	Speedup multiplier
2	0	1456.5	120	12.14
3	0	8863.2	169	52.44
4	0	21588.6	498	43.35
5	0	73508	2629	27.96
6	0	299126	14927	20.04
2	1	14249.5	478	29.81
3	1	40751	3035	13.43
4	1	232375	20013	11.61
2	2	28798.7	897	32.11
3	2	266437	21257	12.53

Table 20. Speedup of GPU MT Async with a buffer size of 5000 on System G when compared to the original implementation in the CWC corridor scene.

#Ia	Diffraction Limit	Original (ms)	GPU MT Async (ms)	Speedup multiplier
2	0	96.7	20	4.84
3	0	238.8	44	5.43
4	0	1063	242	4.39
5	0	6711	3082	2.18
6	0	66872	37566	1.78
2	1	217.9	30	7.26
3	1	930	224	4.15
4	1	9004	3934	2.29
5	1	173782	85529	2.03
2	2	311.3	46	6.77
3	2	2988	579	5.16
4	2	21976	11173	1.97

Table 21. Refinement statistics for different interaction and diffraction limits in the Etu-Lyötty scene.

#Ia	Diffraction Limit	Initial Paths	Converged (%)	Validation (%)
2	0	1170	99.6	64.1
3	0	4527	98.3	57.3
4	0	17295	95.1	48.1
5	0	64683	91.5	39.4
6	0	242087	88.0	31.5
2	1	33085	99.7	89.7
3	1	153401	99.0	72.7
4	1	510818	97.8	61.8
5	1	1286550	96.2	53.9
6	1	2807669	94.1	46.5
2	2	88009	99.8	91.2
3	2	379181	99.4	81.6
4	2	1118404	98.4	69.8

Table 22. Refinement statistics for different interaction and diffraction limits in the CWC corridor scene.

#Ia	Diffraction Limit	Initial Paths	Converged (%)	Validation (%)
2	0	318	100	50.3
3	0	1798	100	24.6
4	0	10569	99.5	10.6
5	0	62476	99.0	4.21
6	0	360188	98.3	1.71
2	1	2521	100	68.7
3	1	21410	100	41.2
4	1	148213	99.8	22.9
5	1	858413	99.6	12.1
6	1	4053563	99.1	6.23
2	2	4535	100	74.6
3	2	41040	99.6	46.4
4	2	257085	99.5	26.5