GPU Accelerated Ray-tracing for Simulating Sound Propagation in Water

Joacim Stålberg & Mattias Ulmstedt
Abstract

The propagation paths of sound in water can be somewhat complicated due to the fact that the sound speed in water varies with properties such as water temperature and pressure, which has the effect of curving the propagation paths. This thesis shows how sound propagation in water can be simulated using a ray-tracing based approach on a GPU using Nvidia’s OptiX ray-tracing engine. In particular, it investigates how much speed-up can be achieved compared to CPU based implementations and whether the RT cores introduced in Nvidia’s Turing architecture, which provide hardware accelerated ray-tracing, can be used to speed up the computations. The presented GPU implementation is shown to be up to 310 times faster than the CPU based Fortran implementation Bellhop. Although the speed-up is significant, it is hard to say how much speed-up is gained by utilizing the RT cores due to not having anything equivalent to compare the performance to.
Acknowledgments

We would like to thank Saab Dynamics for providing us with the opportunity and resources to do this project. A special thanks to Stefan Persson, Andreas Gällström and Per-Ola Svensson for their help and direction throughout the project.

We would also like to thank Ingemar Ragnemalm for being our examiner, and August Ernstsson for being our academic supervisor at Linköping University.

A special thanks to Jens Hovem for his previous work on the topic and for providing us with the source code of PlaneRay which served as a way to validate our implementation.

Linköping, June 2019
Joacim Stålberg & Mattias Ulmstedt
## Contents

List of Figures .................................................. ix
List of Tables .................................................. xi

### 1 Introduction
1.1 Motivation ................................................. 1
1.2 Aim ......................................................... 2
1.3 Research Questions ........................................ 2
1.4 Delimitations .............................................. 2
1.5 Outline ..................................................... 2

### 2 Sound Propagation in Water
2.1 Bottom Reflection Loss .................................... 8
2.2 Surface Reflection Loss ..................................... 8
2.3 Acoustic Absorption ........................................ 9
2.4 Turning Points and Caustics ............................... 9
2.5 Eigenrays ................................................... 9
2.6 Transfer Function .......................................... 10
2.7 Practical Uses ............................................. 11

### 3 GPGPU Computing
3.1 Patterson’s Three Walls .................................... 13
3.2 GPU Architecture .......................................... 14
  3.2.1 The Turing Architecture ............................. 14
  3.2.2 Nvidia GeForce RTX 2060 ............................ 18
3.3 CUDA ....................................................... 19
  3.3.1 Intrinsic Functions ................................... 19

### 4 Ray-tracing
4.1 Bounding Volume Hierarchy .............................. 23
4.2 OptiX ....................................................... 25
  4.2.1 OptiX Basics .......................................... 26
  4.2.2 Scenes in OptiX ....................................... 27
## Contents

4.2.3 RTX Mode .................................................. 28  
4.3 DirectX Raytracing ......................................... 28  
4.4 Previous Work ............................................... 28  

5 Method .......................... 31  
5.1 Pre-study .................................................... 31  
5.2 Implementation ........................................... 32  
5.3 Evaluation .................................................. 33  
5.3.1 RT Core Evaluation ................................. 34  
5.3.2 Floating-point Precision Format and Intrinsics ... 34  
5.3.3 Error Metric ............................................ 35  
5.3.4 Hardware ............................................... 35  

6 Implementation .................. 37  
6.1 Initializations and Parameters ......................... 37  
6.2 Modes of Operation ...................................... 38  
6.3 Ray-tracing ............................................... 40  
6.3.1 The Scene .............................................. 40  
6.3.2 Ray Classification ................................. 43  
6.4 Analyzing and Sorting Collected Data ................ 43  
6.5 Interpolation of Eigenrays ............................ 44  
6.6 Signal Processing ....................................... 45  
6.7 Graphical Presentation ............................... 47  
6.8 Memory Considerations .............................. 49  

7 Results .................. 51  
7.1 Validation .................................................. 59  
7.2 Floating-point Precision Format and Intrinsics .... 59  

8 Discussion .................. 65  
8.1 Results .................................................... 65  
8.1.1 Floating-point Precision Format and Intrinsics ... 67  
8.2 Method .................................................... 68  

9 Conclusions ............... 71  
9.1 Future Work ............................................... 71  

A Scenario Parameters .... 75  

B Sound Speed Profiles .... 81  

Bibliography ............... 85
List of Figures

2.1 Example of a sound speed profile ....................................... 4
2.2 Relation between ray and wave-front propagation .................. 4
2.3 Example of the modeling of sound propagation in the ocean ....... 5
2.4 Small segment of a ray path .............................................. 6
2.5 Example of a ray's path through the layers ............................. 7
2.6 Eigenray example .......................................................... 10

3.1 Comparison between the architecture of CPUs and GPUs ............ 14
3.2 Architectural overview of a Turing SM .................................. 16
3.3 Shared memory structure .................................................. 17

4.1 Ray-tracing in graphics ..................................................... 22
4.2 Bounding volume hierarchy ............................................... 23
4.3 OptiX scene representation ............................................... 27

6.1 Data flow of OXSP from input to output ................................. 38
6.2 Ray trajectories for Scenario 8 with 50 rays ............................ 40
6.3 Simplified view of the structure of the OptiX scene in OXSP ....... 42
6.4 Eigenray interpolation ...................................................... 44
6.5 Ricker pulse with center frequency 200 Hz in the time domain ... 46
6.6 Amplitude spectrum of a Ricker pulse with center frequency 200 Hz 46
6.7 The GUI of OXSP ............................................................ 47
6.8 Heatmap for Scenario 8 ..................................................... 48
6.9 Heatmap for a Scenario 9 .................................................... 48
6.10 Heatmap for a Scenario 10 ................................................ 48
6.11 Heatmap for a Scenario 11 ................................................ 48

7.1 Amplitude spectrum of transfer function for Scenario 1 ............ 52
7.2 Phase spectrum of transfer function for Scenario 1 .................. 52
7.3 Eigenray contributions ...................................................... 53
7.4 Final time response for Scenario 1 ....................................... 53
7.5 Ray-tracing execution times for Scenario 1 with a varying number of rays .................................................. 54
7.6 Time response comparison using triangles/rectangles .............. 54
7.7 Ray-tracing execution times for Scenario 1 with a varying number of layers ............................................ 55
7.8 Full execution time of OXSP for Scenario 6 ............................................. 55
7.9 Memory transfer time ................................................................. 57
7.10 S&G comprehensive error factor for Scenario 3 with a varying number of rays ........................................... 57
7.11 S&G comprehensive error factor for Scenario 4 with a varying number of rays ........................................... 57
7.12 S&G comprehensive error factor for Scenario 5 with a varying number of rays ........................................... 58
7.13 S&G comprehensive error factor for Scenario 1 with a varying number of layers ........................................... 59
7.14 Comparison of time responses of OXSP and PlaneRay for Scenario 1 ......................................................... 60
7.15 Comparison of time responses of OXSP and PlaneRay for Scenario 2 ......................................................... 60
7.16 Comparison of time responses of OXSP and PlaneRay for Scenario 7 ......................................................... 61
7.17 Time responses for different precision formats and use of intrinsics for Scenario 1 ...................................... 61
7.18 Time responses for different precision formats and use of intrinsics for Scenario 5 with 1500 rays ..................... 62
7.19 Average ray-tracing execution times for different precisions for Scenario 1 with a varying number of rays .... 62
7.20 S&G comprehensive error factor for Scenario 3 with a varying number of rays ........................................... 63
7.21 S&G comprehensive error factor for Scenario 4 with a varying number of rays ........................................... 63
7.22 S&G comprehensive error factor for Scenario 5 with a varying number of rays ........................................... 64
7.23 S&G comprehensive error factor for Scenario 1 with a varying number of layers ........................................... 64

B.1 Sound speed profile 1 .............................................................. 82
B.2 Sound speed profile 2 .............................................................. 82
B.3 Sound speed profile 3 .............................................................. 83
List of Tables

3.1 Intrinsic functions in CUDA ........................................ 20

6.1 Parameter list ..................................................... 39
6.2 Data saved when a ray intersects the receiver depth ............ 41
6.3 Ray interactions and their associated values ..................... 43

7.1 RT-core speed-up ..................................................... 56

A.1 Scenario 1 parameters ............................................. 76
A.2 Scenario 2 parameters ............................................. 76
A.3 Scenario 3 parameters ............................................. 77
A.4 Scenario 4 parameters ............................................. 77
A.5 Scenario 5 parameters ............................................. 78
A.6 Scenario 6 parameters ............................................. 78
A.7 Scenario 7 parameters ............................................. 79
A.8 Scenario 8 parameters ............................................. 79
A.9 Scenario 9 parameters ............................................. 80
A.10 Scenario 10 parameters ......................................... 80
A.11 Scenario 11 parameters ......................................... 80
Simulating sound propagation in water can be useful for a number of reasons, e.g. in the development of sonars, and there exist a variety of models based on different approaches to do it. Calculating the propagation paths can be nontrivial due to the fact that sound speed in water varies with properties such as temperature and pressure, which results in the paths curving. One method for simulating this is through the use of ray-tracing.

Ray-tracing is a technique that is mostly known for its use in rendering graphics, but can be used in other areas as well. Ray-tracing basically means tracing the path of rays in space and finding where the ray intersects with objects. By modeling the water as planes stacked on top of each other with some space between them the effects of the differences in water speed can be approximated through the use of Snell’s law. This approximation gets better the more layers are used and the less distance there is between each layer. The idea is that by tracing a large number of rays sent out from a transmitter in different directions, all the propagation paths from a transmitter to a receiver can be found.

One of the downsides of ray-tracing is that it can be computationally demanding to trace a large number of rays in scenes with lots of objects to intersect with. As the rays are not affected by each other they can be traced in parallel, and as a result the problem is well suited for a Graphics Processing Unit (GPU) implementation. This thesis proposes one way this can be done using Nvidia’s OptiX ray-tracing engine.

1.1 Motivation

With the introduction of Nvidia's Turing GPU architecture that has inherent support for ray-tracing with its dedicated RT cores, ray-tracing performance is expected to improve. By simulating sound propagation in water on a GPU and
utilizing these RT cores, the execution time is expected to improve considerably compared to equivalent models running on a Central Processing Unit (CPU) or a GPU with an older architecture.

1.2 Aim

The aim of this thesis is to investigate how sound propagation in water can be simulated using ray-tracing, and in particular, how Nvidia’s ray-tracing engine OptiX can be used for this purpose. It will also investigate how much speed-up can be achieved by utilizing a modern GPU compared to pure CPU implementations.

1.3 Research Questions

- How can an algorithm for simulating sound propagation in water using ray-tracing be implemented on a GPU?
- Is it possible to gain performance with a GPU implementation of a sound propagation simulator using ray-tracing compared to CPU implementations?
- How much speed-up can be gained by utilizing the RT cores introduced in Nvidia’s Turing architecture with the proposed implementation?

1.4 Delimitations

Only a ray-tracing based approach for modeling sound propagation in water will be evaluated in this thesis. There are other approaches that could be well suited for parallel implementations as well, but the ray-tracing approach is especially interesting with the release of Nvidia’s Turing architecture because of its RT cores which are supposed to accelerate ray-tracing. The model will be limited to simulating sound propagation in a 2D environment.

1.5 Outline

The thesis starts out in Chapter 2 where a theoretical model of how sound propagates in water using ray acoustics is explained. Chapter 3 then gives a brief overview of General Purpose computing on Graphics Processing Units (GPGPU) and Chapter 4 explains the concept of ray-tracing and how a GPU can be utilized to accelerate ray-tracing.

In Chapter 5 the method is discussed followed by Chapter 6 which details the implementation and Chapter 7 where the results are presented. The results are then discussed in Chapter 8 and lastly the conclusions are found in Chapter 9.

All test scenarios that are used can be found in Appendix A and all sound speed profiles can be found in Appendix B.
Sound can be described in terms of sinusoidal plane waves with properties such as frequency and amplitude. The propagation path of sound depends on a lot of factors including the sound propagation speed which varies, making the path curve. The speed of sound in water in turn depends on a number of things such as temperature, salinity and depth. There are several formulas for approximating the sound speed in water. One such formula presented by Medwin [18] is

\[ c = 1449.2 + 4.6t - 0.055t^2 + 0.00029t^3 + (1.34 - 0.010t)(s - 35) + 0.016d \] (2.1)

where \( c \) [m/s] is the sound speed, \( t \) [°C] is the temperature, \( s \) [%] is the salinity and \( d \) [m] is depth. An example of a typical sound speed profile can be seen in Figure 2.1.

In ray acoustics, the propagation of sound is modeled by rays along the direction of energy being transported. The rays are straight lines that are parallel to the normals of the wave-fronts originating from a source as can be seen in Figure 2.2. The concept is very similar to the ray concept in optics where light propagation is modeled by rays. There are both pros and cons of modeling sound using rays as opposed to using classical wave theory. When using ray theory the solution is easy to interpret and boundary conditions such as a sloping bottom are easy to implement. Another advantage is that the contributions of different ray paths can be separately analyzed. It is also considerably faster if you are interested in a single point far away as you can calculate it directly, whereas with wave theory you have to calculate the entire sound field up to that point. The downsides are that it does not handle all conditions such as diffraction problems and it is only valid at high frequencies. The model also breaks down in the vicinity of caustics as it sometimes predicts infinite intensity. [39]

In order to calculate the paths of the rays through the water we make use of Snell’s law which expresses a relationship between the speed and angle of a ray.
Figure 2.1: Example of a sound speed profile, this type of sound speed profile is known as a knee or monk profile. To note is that the positive direction of the vertical axis is directed downwards.

Figure 2.2: Relation between ray and wave-front propagation. The sound waves originate from a source and propagate circularly in two dimensions. The rays are straight lines originating from the source and are parallel to the normals of the wave-fronts.
Figure 2.3: Example of the modeling of sound propagation in the ocean. The coordinate system’s horizontal and vertical coordinate is \( r \) and \( z \), respectively. The ocean is divided into equally thick layers of thickness \( \Delta z \). Sound propagates from a sender and reaches a receiver. Possible sound propagation paths are shown as lines with arrows. This figure also shows an example of a varying bathymetry in the form of the small mound on the ocean bottom floor.

that passes through one medium to another. Using the formulas from the works of Hovem [11, 12] we have a heuristic based on Snell’s law as

\[
\xi = \frac{\cos \theta(z)}{c(z)} = \frac{\cos \theta_0}{c_0}
\]

where \( \xi \) is the ray parameter, \( z \) is a coordinate in vertical axis (water depth), \( \theta_0 \) is the initial angle of the ray to the horizontal plane when going from one layer to another and \( c_0 \) is sound speed in the initial layer.

We have a two-dimensional coordinate system with axes \( r \) and \( z \) which denote horizontal and vertical position respectively. The coordinate system can be seen in Figure 2.3.

Because the speed of sound varies with depth the propagation paths are curved. Taking a small segment of a ray path, it will have components \( dr \) and \( dz \) with an angle \( \theta \) to the horizontal plane (see Figure 2.4), and the radius \( R \) of this curvature is

\[
R = \frac{ds}{d\theta} = \frac{1}{\sin \theta} \frac{dz}{d\theta}
\]
where $ds$ is the arc length of the small segment and $d\theta$ is the angle increment. We also define the sound speed gradient as

$$g(z) = \frac{dc(z)}{dz}$$

(2.4)

The radius of the curvature $R$ as a function of $z$ for a ray traversing layer $i$ becomes

$$R_i(z) = -\frac{1}{\xi g_i(z)}$$

(2.5)

The suggested method of doing ray-tracing to simulate sound propagation in water is to divide the whole water column into equally thick layers of thickness $\Delta z$. Within each such layer, the sound speed is approximated to be linear according to

$$c(z) = c_i + (z - z_i)g_i$$

(2.6)

for the layer $z_i < z < z_{i+1}$ where $c_i$ is the sound speed for layer $i$ and $g_i$ is the sound speed gradient of that layer. This means that we ignore the variation of speed in the horizontal plane and only consider the dependence on depth. Assuming a constant speed gradient for each layer we approximate it for a certain layer with

$$g_i = \frac{c(z_{i+1}) - c(z_i)}{z_{i+1} - z_i}$$

(2.7)

which means that the rays follow a circular arc within a layer.

The horizontal distance a ray propagates through a layer $z_i < z < z_{i+1}$ can now be calculated with

$$r_{i+1} - r_i = -R_i(\sin \theta_{i+1} - \sin \theta_i)$$

(2.8)

which can be written as

$$r_{i+1} - r_i = \frac{1}{\xi g_i} \left( \sqrt{1 - \xi^2 c^2(z_i)} - \sqrt{1 - \xi^2 c^2(z_{i+1})} \right)$$

(2.9)
Figure 2.5: Example of a ray’s path through the layers. Decreasing layer thickness will smooth out the curving appearance of the ray paths.

and the time distance through the same layer is given by

$$\tau_{i+1} - \tau_i = \frac{1}{|g_i|} \ln \left( \frac{c(z_{i+1})}{c(z_i)} \frac{1 + \sqrt{1 - \xi^2 c^2(z_i)}}{1 + \sqrt{1 - \xi^2 c^2(z_{i+1})}} \right)$$

(2.10)

Using Equation (2.9) it is now possible to trace a ray going through layer $i$ corresponding to an increment or decrement of $\Delta z$ in the vertical plane. An example of the path a ray might take can be seen in Figure 2.5. Using Equation (2.10) gives the corresponding travel time.

There is a special case when the ray makes a turn within a layer which is the case when $\xi^2 c^2(z_{i+1}) \geq 1$. In this case the formulas

$$r_{i+1} - r_i = \frac{2}{\xi g_i} \sqrt{1 - \xi^2 c^2(z_i)}$$

(2.11)

and

$$\tau_{i+1} - \tau_i = \frac{2}{|g_i|} \ln \left( \frac{1 + \sqrt{1 - \xi^2 c^2(z_i)}}{\xi c(z_i)} \right)$$

(2.12)

should be used instead.

Care must also be taken to the fact that the ray parameter $\xi$ is not constant with varying bathymetry$^1$. When a ray is reflected on the sea bottom with an inclination $\alpha$ the outgoing angle will then change according to

$$\theta_{out} = \theta_{in} + 2\alpha$$

(2.13)

This means that the ray parameter must change as well. Combining Equation (2.13) and Equation (2.2) we eventually get

$$\xi_{out} = \frac{\cos(\theta_{out})}{c} = \frac{\cos(\theta_{in} + 2\alpha)}{c} = \xi_{in} \cos(2\alpha) \pm \sqrt{1 - \xi_{in}^2 c^2} \sin(2\alpha)$$

(2.14)

To be able to calculate the sound intensity at a certain point you need to account for the transmission loss from multiple effects. The most significant trans-

---

$^1$The bathymetry specifies the ocean depth at every point.
mission loss is due to geometrical spreading. This can be calculated as

\[
\frac{I}{I_0} = \frac{r_0^2}{r} \left| \frac{\cos \theta_0}{\sin \theta} \frac{d \theta_0}{d r} \right| = \left( \frac{r_0^2}{r} \right) \left( \frac{c_0}{c} \right) \left| \frac{\cos \theta}{\sin \theta} \frac{d \theta}{d r} \right|
\]  
(2.15)

where \( I \) is the intensity after propagation at a distance \( r \) from the source, \( r_0 \) is a reference distance, \( I_0 \) is the intensity at the reference distance and \( d \theta_0 \) is the initial angular separation between two adjacent rays. An important assumption for the intensity calculations is that the energy remains between two adjacent rays with initial angular separation \( d \theta_0 \). With respect to a reference distance \( r_0 \) from the source the transmission loss can be calculated with

\[
TL = 10 \log \left( \frac{r}{r_0} \right) + 10 \log \left| \frac{d z}{d \theta_0} \right| + 10 \log \left( \frac{c}{c_0} \right)
\]  
(2.16)

In addition to this, bottom and surface reflection loss along with acoustic absorption in the sea water should be accounted for.

### 2.1 Bottom Reflection Loss

Reflections on the sea bottom are more complicated than surface reflections because of the different materials and multi-layering of materials. An easy way to model this is to treat the bottom as made up of only a single layer at the intersection point and use the Rayleigh formula for intensity ratios presented by Urick [39]. The ratio can be used as the bottom reflection coefficient according to

\[
R_{bot} = \frac{I_r}{I_i} = \left( \frac{m \sin \theta_1 - \sqrt{n^2 - \cos^2 \theta_1}}{m \sin \theta_1 + \sqrt{n^2 - \cos^2 \theta_1}} \right)^2
\]  
(2.17)

with

\[
m = \frac{\rho_2}{\rho_1}, \quad n = \frac{c_1}{c_2}
\]  
(2.18)

where \( I_r \) denotes the reflected intensity, \( I_i \) denotes the incident intensity, \( \theta_1 \) denotes the grazing angle, \( c_1 \) and \( \rho_1 \) is sound speed of water and density respectively in the layer above the bottom, and \( c_2 \) and \( \rho_2 \) denote the same attributes inside the bottom layer.

### 2.2 Surface Reflection Loss

The interaction with the sea surface is affected by waves and air bubbles near the surface. Other factors are frequency of the sound and grazing angle to the surface. This can be modeled by a coefficient as mentioned by Hovem [11] according to

\[
R_{coh} = \exp \left( -2 \left( \frac{2\pi}{\lambda} \sigma_h \sin \theta \right)^2 \right)
\]  
(2.19)
where $R_{coh}$ is the coherent reflection coefficient, $\lambda$ [m] is the wavelength of the sound, $\sigma_h$ [m] is the root-mean-square of the heights of the ocean waves and $\theta$ [rad] is the grazing angle. $R_{coh}$ can also be used to model the transmission loss from reflections with a rough bottom.

### 2.3 Acoustic Absorption

Acoustic absorption is the energy loss to the medium the sound travels within. The main components of absorption is relaxation absorption (due to the chemical reactions of magnesium sulphate ($\text{MgSO}_4$) and boric acid ($\text{H}_3\text{BO}_3$)) and viscous absorption. Ainslie and McColm [3] present a formula for modeling these effects under certain conditions given by

$$\alpha = 0.106 \left( \frac{f_1 f_2^2}{f_2^2 + f_1^2} \right) \exp \left( \frac{pH - 8}{0.56} \right)$$

$$+ 0.52 \left( 1 + \frac{T}{43} \right) \left( \frac{S}{35} \right) \left( \frac{f_2 f_2^2}{f_2^2 + f_2^2} \right) \exp \left( -\frac{z}{6} \right)$$

$$+ 0.00049 f_2^2 \exp \left( -\frac{T}{27} - \frac{z}{17} \right)$$

(2.20)

where $\alpha$ [dB/km] is the attenuation, $f$ [kHz] is the frequency, $z$ [km] is depth, $T$ [°C] is temperature, $S$ [%] is salinity and $pH$ is acidity. The constants $f_1$ and $f_2$ are the relaxation frequencies for boron and magnesium respectively, given by

$$f_1 = 0.78 \sqrt{\frac{S}{35}} \exp \left( \frac{T}{26} \right)$$

(2.21)

and

$$f_2 = 42 \exp \left( \frac{T}{17} \right)$$

(2.22)

This formula is within 10% of what is predicted by more rigorous methods. [3]

### 2.4 Turning Points and Caustics

Ray theory predicts infinite intensity when $\theta = 0$ or when $dr/d\theta_0 = 0$, see Equation (2.15). The first case occurs when a ray path changes vertical direction due to the curving caused by sound speed dependence on depth. The second case is when a small increment in starting angle $\theta_0$ gives no difference in horizontal coordinate $r$. The locus of all such points is called a caustic. In reality, the intensity at these points is high but not infinite.

### 2.5 Eigenrays

An eigenray is a ray that propagates from a source point and intersects a receiver point. The idea is to find all the eigenrays that intersect the receiver location.
and sum up the contributions of those rays. For example, Figure 2.6 shows all the eigenrays from a transmitter located at (300, 125) to a receiver at (2600, 180). During the ray-tracing, a record is kept for every ray from which things like transmission loss and phase can be derived. Rays can be sorted into different classes depending on properties such as:

- Positive/Negative starting angle
- Number of bottom reflections
- Number of surface reflections
- Number of upper turning points
- Number of lower turning points

Each class represents an eigenray and thus a unique propagation path from the source to the receiver. To find all the eigenrays in a given scenario a large number of rays are traced through the scene. The collected data is then classified and sorted within their respective classes so that the eigenrays can be calculated at a later stage. As it is unlikely that any of the traced rays passed through the exact location of the receiver, the eigenrays are interpolated from rays that passed by the receiver closest to each side.

### 2.6 Transfer Function

The signal at the receiver is according to

\[ Y(\omega) = H(\omega)X(\omega) \]  \hspace{1cm} (2.23)
where $Y(\omega)$ and $X(\omega)$ denote the frequency domain functions of the output signal and input signal respectively. Given an input signal $x(t)$ in the time domain, it is Fourier-transformed to get $X(\omega)$. The last part required to calculate Equation (2.23) is to calculate the transfer function $H(\omega)$. It is found by adding the transfer functions of all the eigenrays according to

$$H(\omega, r) = \sum_n A_n B_n(\omega) S_n(\omega) T_n \exp(i \omega \tau_n)$$  \hspace{1cm} (2.24)$$

where $n$ denotes the number of eigenrays, $A_n$ is the geometrical spreading loss factor defined as the square root of Equation (2.15), $B_n(\omega)$ is the effects of bottom reflections, $S_n(\omega)$ is the effect of surface reflections, $T_n$ accounts for phase shifts caused by caustics and turning points and $\tau_n$ is the travel time. The time domain response can then be calculated by inverse Fourier-transforming Equation (2.23).

### 2.7 Practical Uses

The techniques outlined in this chapter can be used in the simulation of a sonar that has the purpose of determining the distance to certain objects and even detect the presence of objects in water. Sonar systems are categorized either as being active or passive. An active sonar generates sound that propagates through the water and is partly reflected back to the sonar from various objects. A passive sonar listens to signals that are generated by other objects resulting in a one-way transmission through the water. [39]
Due to limitations in the possible growth of single-core performance, GPGPU is becoming an increasingly important option for accelerating the performance of certain computations, given that the problem at hand can be implemented in a way that can utilize the underlying architecture of modern GPUs. Not all problems fall into this class, but for the ones that do there is a huge potential speed-up that can be achieved by using GPUs instead of CPUs. So what are the limiting factors for the growth of single-core performance?

3.1 Patterson’s Three Walls

Patterson’s Three Walls, coined by Patterson [4], are the three main limiting factors for single-core performance. These are:

- **The power wall.** Increasing the clock frequency of processors leads to an exponential increase in power consumption, which in turn brings with it a number of problems such as power leakage and heat generation. As the cores get hotter, the performance degrades and you have to spend a lot of resources just to cool it down.

- **The memory wall.** Memory units are slower than CPUs which can make them huge bottlenecks in memory-bound programs, and this problem becomes even bigger when you have multiple CPUs sharing the same memory.

- **The ILP wall.** Instruction Level Parallelism (ILP) such as branch prediction, out-of-order execution and Very Long Instruction Word (VLIW) systems has reached its limit and trying to find more ILP has diminishing returns.

Parallel computing attempts to circumvent these problems. Instead of having one very fast core, we instead try to utilize a large number of slightly slower...
3.2 GPU Architecture

GPUs are designed for massive parallelism and have thousands of simple, data-parallel cores (e.g., Nvidia’s Turing TU102 GPU has 4608 CUDA cores) which operate in a Single Instruction Multiple Threads (SIMT) fashion. SIMT basically means that the same instruction is executed simultaneously across multiple threads. The idea behind SIMT is to reduce the overhead of fetching instructions [33].

GPUs are optimized for high memory throughput as opposed to CPUs which are optimized for low memory latency [16]. The higher latency is in part masked by efficient hardware supported context switching when threads are waiting for response from the memory. These points make possible a very different architecture compared to a traditional CPU. The high memory bandwidth and fast context switches reduce the need for having a large data cache, and the context switching combined with the SIMT execution model reduce the need for advanced flow control [25]. This means that a much larger portion of the transistors can be used for data processing instead (see Figure 3.1).

This thesis will focus on Nvidia GPUs, in particular Nvidia’s Turing architecture, but GPUs from other vendors are architecturally quite similar.

3.2.1 The Turing Architecture

Turing is the newest GPU architecture from Nvidia (as of 2019). The GPUs consist of thousands of CUDA cores split across a smaller number of Streaming Multipro-
3.2 GPU Architecture

Processors (SMs) (Figure 3.2), each containing 64 CUDA cores, a number of Tensor cores and one RT core (more on these later). The SMs are split into four processing blocks, each containing a L0 instruction cache and a 64 kB register file. Each SM has a combined 96 kB L1 data cache/shared memory which can be partitioned into 32 kB shared memory and 64 kB L1 cache or 64 kB shared memory and 32 kB L1 cache. [26]

There are also a number of Special Function Units (SFUs) which are specialized units that execute transcendental instructions called *intrinsics* (such as \(__sin()\) and \(__cos()\), see Section 3.3.1) very efficiently.

There are six types of memories in modern Nvidia GPUs, each with its own advantages and disadvantages. Using CUDA terminology, these are:

- **Registers.** Every thread has a number of registers dedicated to it. These registers are very fast and are used to store things like local variables and temporary values.

- **Local memory.** The local memory is used for local variables that do not fit in local registers. This memory is a part of the global memory but has its own address space.

- **Global memory.** The global memory is the main memory of the GPU and is the largest memory. It has high bandwidth but the throughput is affected by the memory access patterns. All accesses are via 32-, 64- or 128-byte transactions and they must be aligned to a multiple of the transaction sizes. For example, if a thread attempts to access a 4-byte section of the memory, a minimum transaction of 32 bytes still has to be done, effectively making the throughput just \(1/8\) \((4/32)\) of the maximum throughput. To avoid this kind of inefficient transaction, the threads in a warp can access the memory in a coalesced manner. In short, this means that consecutive memory addresses should be accessed by consecutive threads in a warp so that the accesses can be grouped together to fewer transactions. Coalesced accesses to global memory can be up to 10 times faster than non-coalesced accesses [9]. For a more detailed explanation of how to coalesce accesses to global memory, see the CUDA C programming guide [25].

- **Shared memory.** Every SM has access to an on-chip memory called shared memory which has both a higher bandwidth and much lower latency than the global memory. Shared memory is essentially a user-managed data cache which can be hundreds of times faster than the global memory [9] depending on the access pattern. The memory is split into a number of smaller modules called *memory banks*. Accesses to different banks can be done in parallel which is what enables the high bandwidth. If there are multiple accesses to the same bank simultaneously, however, the accesses have to be split into multiple transactions. This is called a *bank conflict*, and it is up to the programmer to manage the access patterns in order to avoid this. A simplified illustration of the bank structure can be seen in Figure 3.3.
Figure 3.2: Architectural overview of a Turing SM.
• **Constant memory.** Constant memory is a read-only memory that is broadcast to the entire half-warp when a thread in the half-warp reads from it. The advantage of having a read-only memory is that it is easier to cache.

• **Texture memory.** Texture memory is another read-only memory and so has the same cache benefits as constant memory. It is designed for 2D spatial locality and has hardware support for edge handling, texture filtering and linear interpolation.

Threads are scheduled to SMs in blocks of customizable size. Once a block has been scheduled to an SM it runs until all threads in the block have finished, without preemption. Every SM has four warp schedulers which are responsible for issuing instructions to the threads within its assigned block in groups of 32 (called warps). All the threads running concurrently in the same warp execute in a SIMT fashion and when a thread stalls, due to for example a memory access or because it is waiting for the Arithmetic Logic Unit (ALU), the warp schedulers switch context and dispatch instructions to another warp that is ready in order.
to increase the utilization of the available resources. Threads within a warp may diverge due to a data-dependent conditional branch. When this happens the performance suffers because they can no longer run in parallel, but in Nvidia’s Volta and later architectures the threads may re-converge at a later time.

**Tensor Cores**

Tensor cores are accelerators specialized in performing the tensor/matrix operations that are heavily used in Deep Learning [26]. They can speed up the training of neural networks and are also specialized for inferencing operations. These cores are quite important given the growing importance of neural networks and Artificial Intelligence (AI), but they will not be further discussed in this thesis. For more information on these the reader is referred to the architectural whitepapers for Turing and Tesla.

**RT Cores**

Turing introduces a new type of accelerator, the RT core. RT cores are accelerators specialized in some calculations used in ray-tracing and are intended to offload the SMs. The cores are designed specifically to accelerate Bounding Volume Hierarchy (BVH) (Section 4.1) traversal and ray/triangle intersection testing functions. Traversing BVHs and testing against all the possible bounding box and triangle intersections is very computationally expensive and would take thousands of instruction slots per ray if run on a normal processor. By offloading this to the RT cores the SM is free to do other work while waiting for the result of the traced ray from the accelerator. Turing is capable of tracing over 10 Giga rays/s using the RT cores, whereas the previous architecture Pascal was only capable of about 1.1 Giga rays/s [26].

There has been a number of previously proposed hardware accelerators for ray-tracing [19, 29, 34, 40], each with its own set of limitations. Hardware support has in the past primarily been limited by three things: the difficulty in handling the complex memory access patterns, support for flexible control flow and the sheer amount of floating point computations [40]. There are a lot of different algorithms for ray-tracing, but having accelerators for ray-tracing in general purpose GPUs were in part made possible because the usage of BVHs became standard. The ray-tracing libraries OptiX, DirectX Ray-tracing (DXR) and Vulkan all have the capability to utilize the RT cores.

### 3.2.2 Nvidia GeForce RTX 2060

Turing is the latest GPU architecture from Nvidia (as of 2019) and RTX 2060 is one of the cards in the new GeForce 20 series and is a downscaled version of the TU106 GPU. The card contains 30 SMs, each consisting of eight Tensor cores, one RT core and 64 CUDA cores for a total of 240 Tensor cores, 30 RT cores and 1920 CUDA cores.
3.3 CUDA

CUDA is Nvidia’s parallel computing platform and Application Programming Interface (API) made specifically for GPGPU. You can program in CUDA using extensions for many of the popular languages including C, C++, Fortran, Python, Java and others.

When writing CUDA code it is important to have an understanding of the underlying architecture of the GPU (see Section 3.2) and the CUDA programming model in order to produce efficient code as the difference in performance between good and bad CUDA code can be very significant. There are a lot of different optimizations that can be done, some of them covered by Che et al. [6] and Harris [9]. You could write a large number of papers just about writing efficient CUDA code, but we will cover just the most basic things here.

Occupancy is the ratio between the number of active warps on an SM to the supported maximum number of active warps on the SM. To ensure full occupancy of all SMs on the GPU the workload has to be to split over a lot of threads in such a way that as many threads as possible can execute in parallel. Threads in the same block can be synchronized through the use of `__syncthreads()` which acts as a barrier, and GPUs with Compute Capability 3.0 or higher (Kepler and later GPUs) can synchronize between blocks using cooperative groups [24]. In general it is a good idea to use as little synchronization as possible, especially between threads in different blocks, but when used in a clever way it can enable the use of some optimizations (e.g. using shared memory). A common strategy when multiple threads in the same block are going to access the same memory location is to first have the threads read the data from global memory in a coalesced manner, synchronize through the use of `__syncthreads()`, process the data and finally write the result back to global memory, again in a coalesced manner.

The threads are scheduled to the SMs in blocks of customizable size. Every SM then dispatches instructions to warps of 32 threads in a SIMT fashion. Due to this, block sizes should always be a multiple of 32, and if possible with a multiplier higher than one (e.g. 256 threads) so that when a warp stalls, there is always another warp ready to execute. For example, if the block size is 33, the threads will occupy 2 instruction slots in the SM by executing 32 threads in the first slot, then another slot for just the one remaining thread.

Because of the SIMT nature of the threads in a warp a GPU programmer should always avoid diverging paths of the threads to the extent that it is possible. Diverging paths increase execution time in two ways: the time it takes to execute all the diverged branches increases as they can no longer be executed in parallel, and there is some overhead from the divergence management mechanism that increases linearly with the number of diverged threads [5, 6].

3.3.1 Intrinsic Functions

Intrinsic functions (or intrinsics for short) are special CUDA functions that can only be executed in device code (i.e., on the GPU). These functions execute in
Table 3.1: Non-intrinsic versions of functions and their respective intrinsic versions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>x/y</td>
<td>__fdividef(x,y)</td>
</tr>
<tr>
<td>sinf(x)</td>
<td>__sinf(x)</td>
</tr>
<tr>
<td>cosf(x)</td>
<td>__cosf(x)</td>
</tr>
<tr>
<td>tanf(x)</td>
<td>__tanf(x)</td>
</tr>
<tr>
<td>sincosf(x, sptr, cp)</td>
<td>__sincos(x, sptr, cp)</td>
</tr>
<tr>
<td>logf(x)</td>
<td>__logf(x)</td>
</tr>
<tr>
<td>log2f(x)</td>
<td>__log2f(x)</td>
</tr>
<tr>
<td>log10f(x)</td>
<td>__log10f(x)</td>
</tr>
<tr>
<td>expf(x)</td>
<td>__expf(x)</td>
</tr>
<tr>
<td>exp10f(x)</td>
<td>__exp10f(x)</td>
</tr>
<tr>
<td>powf(x, y)</td>
<td>__powf(x,y)</td>
</tr>
</tbody>
</table>

the SFUs in the SMs which are units with hardware specialized to execute these functions as efficiently as possible. This results in lower execution times at the cost of lower accuracy and in some cases different special case handling. These functions only support single-precision floating-point format. The intrinsic functions are for the most part called the same as the standard ones prefixed by __, see Table 3.1. The compiler also has an option to automatically change all the functions to their intrinsic counterpart, although it is recommended to explicitly use the intrinsic versions where lower accuracy is acceptable.
Ray-tracing is a technique that is for the most part used for rendering graphics. It works by simulating the physics of photons as they traverse the scene that you render. It is capable of producing very realistic lighting effects such as shadows, reflections, indirect illumination and refractions. The technique is in principle very simple and easy to understand, and it makes it a lot easier to create realistic effects compared to traditional techniques used in conjunction with rasterization. Ray-tracing has one significant downside however: the computational cost. Scenes are usually represented as a set of geometric primitives such as triangles or quadrangles. When building an image from a scene using ray-tracing you operate pixel-by-pixel as opposed to rasterization which operates primitive-by-primitive [36]. Due to this, rasterization has historically been by far the most popular rendering technique. A simplified explanation of the difference between ray-tracing and rasterization can be stated as the following: rasterization computes a subset of rays that hit a given primitive, whereas ray-tracing computes the subset of primitives that are hit by a given ray [40]. An illustration of how ray-tracing works for rendering graphics can be seen in Figure 4.1.

Ray-tracing can also be used for a broader set of applications than just rendering graphics. Among other things it can be used for scientific computations. For example, Felbecker et al. [7] and others [32, 35] used ray-tracing to model the propagation of electromagnetic waves and in this thesis ray-tracing is used to model sound propagation in water. Ray-tracing is well suited for parallel implementations because every ray is independent and in most cases a large number of rays are cast.
Figure 4.1: The principles of ray-tracing in graphics. The figure shows how rays originate from a source point and are traced until they intersect with objects in 3D space. The colors of the intersected objects are projected onto pixels in a 2D image. By Henrik - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=3869326
4.1 Bounding Volume Hierarchy

A BVH is an acceleration structure used in ray-tracing to reduce the number of ray intersection tests that are made. In 3D space, the objects (geometric shapes) are enclosed by bounding volumes and then further subdivided into smaller bounding volumes that are contained within the larger volumes. This subdivision continues until some threshold is met. It is important that the shapes of the bounding volumes require as little computation as possible to perform intersection tests on for the sake of efficiency.

Goldsmith and Salmon [8] suggest a sphere or rectangular prism with sides aligned with coordinate axes as the simplest shapes, requiring on average 10 floating point operations per intersection test, but mention that the most effective shape is more complex and depends on the objects bounded by the bounding volume. These volume boundaries are then put into a tree structure called BVH where the root is the bounding volume containing all objects. Every bounding volume has at least two children consisting of either smaller bounding volumes (that further subdivide the space into smaller sections) or primitives (usually either triangles or quadrangles). All primitives are located in the leaves of the BVH. The result is a spacial hierarchy structure (see Figure 4.2). Instead of testing intersections of a ray to every possible object, the intersection tests are done by traversing the tree, following the positive outcomes from the intersection tests starting from the root node to its children until a leaf node is found in the case of a hit or if all children of a node test negative in the case of no intersection.

In the case where the BVH is a binary tree we are dealing with a binary search with an average time complexity of $O(\log(n))$, where $n$ is the number of leaves in the tree. This is much better than a linear search which is the case where intersection tests would be made on all objects. The trade-off of adding the extra nodes to reduce the number of intersection tests is that the extra nodes that are required take up memory space.

Because the gains of using BVHs are so great in terms of efficiency, it is a
worthwhile area of research and many different methods of constructing BVHs have been proposed. Many methods of constructing BVHs exist and some recent methods make use of the Surface Area Heuristic (SAH) presented by MacDonald and Booth [17] which is closely related to what is presented by Goldsmith and Salmon. SAH is a method of attributing a cost to the construction of a specific tree which is then used to guide the actual construction of the BVH.

The cost of a certain tree as presented by MacDonald and Booth can be calculated as

\[
c_t = \frac{\sum_{i=1}^{N_i} C_i \sum_{i=1}^{N_i} \text{SA}(i) + \sum_{i=1}^{N_l} C_l \sum_{i=1}^{N_l} \text{SA}(l) + \sum_{i=1}^{N_l} \text{SA}(l)N(l)}{\text{SA}(\text{root})}
\]  

(4.1)

where \( c_t \) is the cost of the tree, \( N_i \) is the number of interior nodes, \( N_l \) is the number of leaves, \( \text{SA}(i) \) is the surface area of interior node \( i \), \( \text{SA}(l) \) is the surface area of leaf node \( l \), \( N(l) \) is the number of objects stored in leaf \( l \), \( C_i \) is the cost of traversing an interior node, \( C_l \) is the cost of traversing a leaf and \( C_o \) is the cost of testing an object for intersection.

The construction of the BVH then becomes an optimization problem where the objective is to minimize the tree cost \( c_t \). SAH makes some simplifying assumptions such as that the probability of a ray to intersect a bounding volume is equal to the surface area of that bounding volume divided by the surface area of the root nodes bounding volume. MacDonald and Booth justify the use of this surface area metric by presenting some promising experimental results. The reason for using a heuristic method instead of an optimal method is because finding the optimal BVH topology for a BVH is believed to be an NP-hard problem [13].

Efforts to speed up the construction speed of BVHs by adapting and coming up with new algorithms that are well suited for GPUs have been made by Kariras and Aila [13]. They were able to show comparable ray-tracing performance numbers (Giga rays/s) compared to the best performing methods for constructing BVHs while reducing the construction time by a significant amount. According to their results, which method is best suited in the general case highly depends on the number of rays traced. The gains in construction speed could be useful where online construction of BVHs is required but it is a trade-off between construction speed of the BVH and its ray-tracing performance. Their algorithm is widely known by the name of TrBVH and is the recommended method for constructing a BVH in the OptiX programming guide [27].

Another approach that is available in OptiX is Split-Bounding Volume Hierarchy (SBVH) as presented by Stich et al. [38]. This method combines the idea of spacial splitting and surface area heuristic. The OptiX programming guide [27] suggests that this method may be outperformed by TrBVH even in the case of static geometries which is a case where SBVH used to be the best method.

In a paper by Ylitie et al. [41], further optimization methods were explored which showed promising results on the Pascal architecture and especially in regards to tracing incoherent rays. By compressing the BVH by quantizing child nodes, determining traversal order of the BVH according to the ray octant, efficiently encoding the traversal order in memory along with a specialized memory
layout scheme and more, they obtained better results for incoherent rays in comparison to other methods.

4.2 OptiX

OptiX is a general purpose ray-tracing engine developed by Nvidia that was released 2009 [22]. The intent behind OptiX was to create an accessible and flexible ray-tracing system for many-core architectures [28]. The reasoning behind this is that most ray-tracing applications build on a small set of operations such as ray generation, material shading, object intersection and scene traversal [28]. Instead of having to implement all these things from scratch for every application, it makes much more sense to utilize an API to do these things. That way development time is reduced and the application becomes much more flexible as the application can be compiled and optimized by the engine for different architectures. The downside is that you might lose some amount of performance compared to a highly optimized application specific implementation due to the engine overhead. Back in 2010, Ludvigsen and Elster [15] compared an OptiX implementation of a ray tracer to a ray tracer that was highly optimized in assembly. For that application they found that their OptiX implementation was three to five times slower than the manually optimized version. Some researchers have also used OptiX for scientific computations to great effect. For example, Felbecker et al. [7] used OptiX in an implementation of a polarimetric electromagnetical wave propagation simulator and managed to achieve a speed-up of up to 420x compared to a CPU based ray-tracer.

OptiX works by combining its hardcoded kernel code with programs supplied by the user. These programs are provided to the engine in the form of Parallel Thread Execution (PTX) code, which is a low-level intermediary representation of the program that is reminiscent of assembly [23]. The PTX code is then Just-In-Time (JIT) compiled by the OptiX engine when the program is executed. The compiler performs a number of optimizations when compiling the PTX code, many of which would not necessarily be considered safe in more general circumstances. This is possible because most of the data structures used in ray-tracing are read-only. The most common way of generating PTX code is through the use of Nvidia’s CUDA C/C++ compiler NVCC. A big advantage of using NVCC is that you have access to all the features and libraries available in C++. [28]

OptiX is a low-level engine that focuses on the most basic and general ray-tracing operations and does not contain concepts that are too application specific and high-level such as light sources and shadows. It also provides support for representing 3D structures in a way that is efficient for the GPU to traverse with objects called acceleration structures in the form of BVHs. This makes it a good fit for simulating sound propagation in water as the API provides a high degree of flexibility.
4.2.1 OptiX Basics

This section covers the basics of how OptiX programs are structured. For more details it is recommended to read the OptiX programming guide [27] from Nvidia.

The first thing you need in order to do ray-tracing is a scene with objects with which the rays can intersect. These objects are represented as polygon meshes. In order to speed up the ray-tracing OptiX utilizes BVHs as acceleration structures. These BVHs are generated and updated automatically from the polygon meshes of the objects in the scene.

The setup and launch of the ray-tracing engine is handled with a context object. All OptiX resources such as user-defined programs, geometries and buffers are encapsulated by a context object. Buffers are used to pass data between the CPU and the GPU and are created by the CPU before launching a context. There are a few different buffer types to choose from depending on application requirements such as one-way write buffers in either direction or two-way read and write buffers.

The way a ray interacts with different objects when they intersect are specified by certain user-defined programs associated with the object. These programs are provided by the user and can be reused for multiple objects. All programs (functions) that interact with rays are prefixed with the macro RT_PROGRAM. The different types of programs that can be bound to objects are:

- **Ray generation program.** The ray generation program is the point of entry for every ray. Subsequent computations done by the kernel are spawned by the ray generation program, e.g. tracing rays or reading/writing buffers. The ray generation program is executed in parallel, once for each specified thread when launching a context.

- **Closest-hit program.** The closest-hit program is executed when the ray intersects the closest object.

- **Any-hit program.** The any-hit program is executed once for every object the ray intersects.

- **Miss program.** The miss program is executed if a ray does not intersect with any object.

- **Exception program.** Exception programs are called when certain types of errors occur.

- **Bounding box program.** The bounding box program defines the bounding volume of the geometry.

- **Intersection program.** The intersection program is used to determine whether a ray intersects the geometry.

All rays carry a user-defined payload which can be used to store whatever data you need that is associated with the ray. It is also possible to submit different types of rays which can differ in what payload, closest-hit program, any-hit program and miss program they are associated with. This payload is also used to return the final result of the ray when it is terminated.
4.2 OptiX

4.2.2 Scenes in OptiX

Scenes in OptiX are represented as graphs, see Figure 4.3. There are a number of different node types in the graphs:

- **Group.** Group nodes are parents to other nodes and has an acceleration structure.

- **Geometry Groups.** These nodes contain primitives and materials. Like the group nodes it also has an acceleration structure. They can contain:
  
  - **Geometry instances.** Geometry instances are used to bind geometries to materials.
  
  - **Geometries.** These nodes contain the geometric primitives (e.g. triangles), a bounding box program and an intersection program.
  
  - **Materials.** The materials hold the intersection programs (closest-hit and any-hit). Each material can be associated with one closest-hit and one any-hit program per ray type.

- **Transforms.** Transform nodes have one child (of any type) and a matrix that is used to perform an affine transformation of the underlying geometry.

- **Selectors.** These nodes are like groups except they also have a visit program that selects one of its children when executed.

The acceleration structures for the nodes in the graph can differ for distinctive objects which provides great flexibility as a certain acceleration structures are
better suited than others for a specific type of object. For example, you may want to spend more computation time building an efficient tree for static objects in the scene that will only have to be built one time, but less time building the structures for the dynamic objects that will have to be rebuilt anyway. There are three types of builders supported in OptiX: Trbvh, Svth and Bvh. Nvidia recommends the Trbvh builder for most cases as it both builds very fast and provides great ray-tracing performance as well.

4.2.3 RTX Mode

RTX mode is an execution mode of OptiX that is available on Maxwell and later architectures. Enabling RTX mode changes the internal execution strategy of OptiX. The documentation on exactly what RTX mode affects is lacking, but it is supposed to provide better performance on newer GPUs and it is required in order to utilize RT cores on GPUs that have them.

4.3 DirectX Raytracing

DXR is an extension to Microsoft’s DirectX 12 (DX12) API that introduces some new concepts which support ray-tracing. The feature was announced in 2018 and can be run on both DX12’s Graphics and Compute engines [20].

At a high level, DXR works much the same as OptiX. Just like OptiX it introduces acceleration structures together with a function for dispatching rays. Programs (shaders) specifying the functionality of these rays are provided to DXR in the form of High Level Shading Language (HLSL) code, which is a C-like language for programming shaders in DirectX [21].

4.4 Previous Work

Previous research has been done about ray-tracing on GPUs. For example, Aila and Laine [2] brought up many techniques for improving Single Instruction Multiple Data (SIMD) efficiency for ray traversals on GPUs. Example techniques include replacing terminated rays and using work queues.

Building on Aila and Laine’s work, Lier et al. [14] researched ray traversal on the Pascal architecture (predecessor to Turing) and tried new techniques for improving the performance. They presented techniques such as Intersection sorting together with many other ways of modifying the acceleration structures which showed promising results for incoherent ray traversal but also a lower than standard performance for highly coherent ray traversal.

Architectural considerations in hardware for improving ray-tracing performance has been an area of research for some time now. In a paper by Aila and Karras [1], they presented a GPU architecture that aims to do just that. One of the supported techniques is called Work compaction that works by re-queueing non-terminated rays into the work pool if more than 50% of the rays (1 thread per ray) in a warp has terminated in order to increase warp efficiency. Because
queue operations are expected to happen so frequently in the architecture, dedicated hardware support is required. They also mentioned an optimization where a ray can be passed to another processor directly instead of being re-queued into memory if the corresponding queue is already bound to that processor.

Earlier work on GPU accelerated ray-tracing for simulation of sound propagation in water has been done by Haugehåtveit [10]. The results of that thesis showed great reduction in computation time for the GPU implementation versus a CPU implementation as the number of traced rays increased. Part of the reason why this is still an interesting area of research is because since then, GPUs have started to include hardware accelerated support for ray-tracing. Haugehåtveit published his thesis before CUDA was even released, and there has not been any research published on simulating sound propagation in water on the Turing architecture.
This chapter contains information about how the thesis project was conducted.

5.1 Pre-study

During the pre-study phase of the project it was concluded that the main focus of the work would revolve around GPGPU programming, ray-tracing and underwater acoustics. It was therefore important to find and read information on these subjects. The authors already had previous experience with GPGPU programming but the other subjects were unfamiliar. While the main focus of the project was to be about GPU computing it was still important to have a basic understanding of underwater acoustics in order to grasp the problem domain.

The propagation paths of the sound were to be modeled as rays using a ray-tracing algorithm. The options for ray-tracing were to implement it from scratch or make use of existing ray-tracing engines. Since the latest GPUs from Nvidia have hardware support for ray-tracing with support APIs to access the hardware acceleration, for constraint reasons the decision was made to make use of one of these ray-tracing engines. As for which ray-tracing engine to use, the choice stood between OptiX, DXR and Vulkan. After reading about the frameworks, OptiX appeared to be better suited for GPGPU whereas DXR and Vulkan are more focused on implementation in the graphics pipeline of computer games. Moreover, there exist more readily available documentation on OptiX. Another factor was that OptiX is built on CUDA programming which the authors already had experience with. Thus OptiX was chosen as the ray-tracing engine to solve the finer ray-tracing details such as parallelizing of the ray-tracing and detecting object collisions. This in turn allowed for more freedom to focus on the domain specific details of sound propagation in water which was the most unfamiliar subject.
The paper “Ray Trace Modeling of Underwater Sound Propagation” by Hovem [11] provided us with the most insights on the specifics of modeling sound propagation in water using ray-tracing and only some complementary reading was required to understand the principles of sound propagation in water. While there are other ways than ray-tracing to model sound propagation such as methods based on parabolic equations, wavenumber integration etc., ray-tracing naturally fits the parallelization opportunities available on GPUs as the problem of tracing a number of rays is an embarrassingly parallel problem due to every ray being independent.

Parallelization opportunities that were identified other than ray-tracing include:

- Classification of rays
- Sorting of rays
- Interpolation of eigenrays
- Graphics
- Signal generation
- Fast Fourier Transformation
- Point-wise multiplication of frequency functions

most of which have varying degrees of parallelizability depending on input parameters.

5.2 Implementation

The implementation is named OptiX Sound Propagator (OXSP) and will henceforth be mentioned as such for brevity. The development process was done in incremental steps. For example, the first main component that was built was the ray-tracing which could be tested and evaluated on its own and then other modules were added to the implementation one at a time. Because there were two people working on OXSP it was possible to work on different components in parallel which meant that two incremental steps were taken at once in many cases. This process was aided by the use of the version management tool Git which was useful when merging code or creating new experimental changes. While not following any specific framework, the main work flow for each increment consisted of the following steps:

1. Planning and researching
2. Implementation
3. Testing
4. Evaluation
5.3 Evaluation

These steps were then repeated until satisfactory conditions were met in the evaluation step. For example, towards the end of the implementation phase it was discovered that there were many previously unnoticed errors in the ray-tracing component which lead to many repeated cycles of the work flow steps. The resulting development method is an incremental and iterative one. Communication becomes an important factor in any project involving more than one person. By sitting back to back during the whole implementation phase, the internal communication was easily handled by directly talking to each other. The same holds true for the whole master thesis work. When working alone it is easy to go in a direction that leads no where which can result in time being wasted. A big advantage of not working alone in a project is that ideas are bounced around all the time and it is more likely that bad ideas are filtered out at an early stage.

5.3 Evaluation

To validate the results of OXSP it was necessary to compare them to some other, already established implementation’s results which could function as a frame of reference. It was also important to find an implementation that attempts to calculate the same thing, e.g., a transfer function or ray-tracing. One such implementation is PlaneRay [12] which is written in Matlab and is based on the same model as our implementation and it also calculates a transfer function and time response at a point. The idea is that by presenting some scenarios with different input parameters and comparing the results of OXSP to that of PlaneRay, it could lend some credibility to the results of OXSP. When validating OXSP against PlaneRay, PlaneRay was modified to use the same bottom model as OXSP in order to narrow down the validation to the features implemented in OXSP. Conversely, the sea absorption model was turned off in OXSP as PlaneRay does not have one implemented. All scenarios used for the tests can be found in Appendix A.

The execution time of OXSP was compared to PlaneRay and the Fortran program Bellhop, which is based on a similar model presented by Porter and Liu [30]. Both PlaneRay and Bellhop are pure CPU implementations and when measuring performance it was decided to only measure the ray-tracing parts of every implementation. The reasons for only comparing the ray-tracing performance are many. One reason is that ray-tracing tends to be the most computationally intensive part of the implementations, another reason is that it is desirable to change as little in the source code of the other implementations as possible. For instance, it would not have been a fair comparison if the time it takes to write results to a file was taken into account which is what is done without modification to the source code of Bellhop and PlaneRay. Also, because ray-tracing is the first step that is done for these implementations, there is less divergence in what tasks are performed between the implementations in the early steps.

All execution time data was captured using timers and taking the average of 25 runs per set of parameters. The functions used were cpu_time() in Fortran, tic and toc in Matlab and QueryPerformanceCounter() in C++.
5.3.1 RT Core Evaluation

The execution time of OXSP was compared with different parameters and on two different GPUs (Nvidia GeForce GTX 1060 and RTX 2060). One of the interesting things was to compare the performance with and without utilizing the RT cores for accelerating the ray-tracing. In order to use RT cores RTX mode has to be enabled and the scene has to be built with a specific triangle primitive. These triangle primitives have special intersection and bounding-box routines that are very fast even when RT cores are not available. They can only be used with RTX mode enabled and there is no way to disable the use of the RT cores when using the triangles, so there is no straightforward way to assess the performance of the RT cores. In order to compare the performance with and without RT core acceleration, some other primitive has to be used as a comparison. For this reason the performance of OXSP on the RTX 2060 was compared between triangles (using RT cores) and a custom rectangle primitive (not using RT cores).

The performance using the triangles and the rectangles was also compared on the GTX 1060 which does not have any RT cores in order to evaluate if there was any performance difference between the primitives to serve as a baseline difference when comparing the primitives on the RTX 2060. The spans of the speed-ups were calculated as

\[
S_{\text{min}} = \frac{m_r - d_r}{m_t + d_t}, \quad S_{\text{max}} = \frac{m_r + d_r}{m_t - d_t}
\]

where \(S_{\text{min}}\) is the minimum speed-up, \(S_{\text{max}}\) is the maximum speed-up, \(m_r\) is the mean execution time of the rectangles version, \(m_t\) is the mean execution time of the triangles version, \(d_r\) is the standard deviation of the rectangles version and \(d_t\) is the standard deviation of the triangles version. The RT core speed-up was then estimated as

\[
S_{\text{RT}} = \frac{S_{2060}}{S_{1060}}
\]

where \(S_{\text{RT}}\) is the RT core speed-up, \(S_{2060}\) is the RTX 2060 speed-up and \(S_{1060}\) is the GTX 1060 speed-up (in terms of triangles vs rectangles). The span of the RT core speed-up was based on the best and worst cases of \(S_{2060}\) and \(S_{1060}\). The performance of having RTX mode enabled/disabled was also compared on both GPUs when using the rectangles.

5.3.2 Floating-point Precision Format and Intrinsics

When comparing floating-point precision formats and intrinsics three different versions of OXSP were tested. The double-precision version was used in all other testing unless explicitly stated otherwise. The difference between the double-precision and the single-precision version is that all relevant variables and operations during ray-tracing were exchanged with their single-precision floating-point counterparts. The intrinsic version of OXSP is the same as the single-precision version except that the intrinsic versions of the functions were used...
When evaluating the different versions the time response resulting from tracing 1,000,000 rays with the double-precision version was used as a reference.

### 5.3.3 Error Metric

When evaluating the accuracy of the time responses for different parameters the Sprague and Geers (S&G) error metric for transient-response signals was used [37]. The metric consists of two parts; $M$ (Equation (5.3)), which measures the magnitude error without taking phase into account, and $P$ (Equation (5.4)), which measures the phase error without taking magnitude into account, where $s$ is the signal to evaluate and $r$ is the reference signal. During the evaluation using the S&G error metric while varying the number of rays, the reference signal was the time response resulting from tracing 1,000,000 rays, and when varying the number of layers the reference signal was the time response resulting from 8000 layers.

$$M = \sqrt{\int_{t_1}^{t_2} s^2(t)dt - \int_{t_1}^{t_2} r^2(t)dt} - 1 \quad (5.3)$$

$$P = \frac{1}{\pi} \arccos \left( \frac{\int_{t_1}^{t_2} r(t)s(t)dt}{\sqrt{\int_{t_1}^{t_2} r^2(t)dt \int_{t_1}^{t_2} s^2(t)dt}} \right) \quad (5.4)$$

$M$ and $P$ are then combined according to Equation (5.5) resulting in the comprehensive error factor denoted as $C$.

$$C = \sqrt{M^2 + P^2} \quad (5.5)$$

### 5.3.4 Hardware

All tests were performed with the CPU Intel Xeon W-2123 Processor, 16 GB memory and either Geforce GTX 1060 3 GB or Geforce RTX 2060 6 GB as the GPU. GTX 1060 is based on the Pascal architecture and RTX 2060 is based on the Turing architecture. If a test does not specify otherwise then the RTX 2060 was used.
The main task of OXSP is to calculate the transfer function of a sound signal propagating from a transmitter to a receiver in a 2D ocean. The transmitter and receiver are both modeled as points in the 2D space. For convenience, the task is broken down into smaller sub-tasks which are correspondingly represented in the source code of OXSP. These tasks can be broken down as follows:

1. Initializations
2. Scene building
3. Ray-tracing
4. Analyzing and sorting collected data
5. Interpolation of eigenrays
6. Signal processing
7. Graphical presentation

where the tasks after ray-tracing can collectively be referred to as the post-processing step. A data flow graph of OXSP can be seen in Figure 6.1. Each task in the graph is internally parallelized, but the tasks have to be performed in this sequential order. All these tasks will be explained in more detail in their respective sections below.

### 6.1 Initializations and Parameters

The beginning of the program consists mainly of setting the various input parameters and allocating memory on the GPU for upcoming calculations. For example,
6.1 Modes of Operation

OXSP has three modes of operation that can be changed in the GUI. These are:

- **Point Mode.** Point mode calculates the transfer function from a source point to a receiver point.

- **Sonar Mode.** In Sonar mode, an active sonar is simulated. A signal is transmitted from a source and propagated through the water while interacting with the surface and bottom. It is then reflected off a reflector target and eventually reaches the source again. The result is a transfer function for the path from source to the target and back to the source.

- **Heatmap Mode.** The Heatmap mode shows a visual presentation of the scene showing the signal’s relative intensity in different parts of the scene as the sound propagates. This is done in the form of a heatmap (see Figure 6.8) where “warmer” colors represent areas where the signal is relatively strong and “colder” colors represent areas where the signal is relatively weak. Signal intensity is compared to the point in the scene where the signal is at its highest intensity.
Table 6.1: List of the parameters that can be changed using the GUI.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitter Position [m]</td>
<td>Coordinates for signal origin point</td>
</tr>
<tr>
<td>Receiver Position [m]</td>
<td>Coordinates for point of interest</td>
</tr>
<tr>
<td>Launch Angle [°]</td>
<td>Range of angles to trace rays</td>
</tr>
<tr>
<td>Number of Rays</td>
<td>Number of rays to trace</td>
</tr>
<tr>
<td>Max range [m]</td>
<td>Maximum horizontal distance of the model</td>
</tr>
<tr>
<td>Max depth [m]</td>
<td>Maximum vertical distance of the model</td>
</tr>
<tr>
<td>Number of Layers</td>
<td>Number of layers used to model the ocean</td>
</tr>
<tr>
<td>Heatmap Samples</td>
<td>Resolution for the heatmap</td>
</tr>
<tr>
<td>Target Position [m]</td>
<td>Coordinates for the reflector</td>
</tr>
<tr>
<td>Target Size [m]</td>
<td>Reflector vertical length</td>
</tr>
<tr>
<td>fs [Hz]</td>
<td>Sampling frequency</td>
</tr>
<tr>
<td>Number of Samples</td>
<td>Number of samples</td>
</tr>
<tr>
<td>Simulate Ricker Pulse</td>
<td>Calculates time response of a Ricker pulse if selected and the transfer function if not</td>
</tr>
<tr>
<td>Write Results to File</td>
<td>Writes time response or transfer function to file</td>
</tr>
<tr>
<td>Center Frequency [Hz]</td>
<td>Denotes the frequency where the amplitude spectrum of the Ricker pulse peaks</td>
</tr>
<tr>
<td>Temperature [°C]</td>
<td>Ocean temperature</td>
</tr>
<tr>
<td>pH</td>
<td>Ocean acidity</td>
</tr>
<tr>
<td>Salinity [%o]</td>
<td>Ocean salinity</td>
</tr>
<tr>
<td>Surface Height rms [m]</td>
<td>Root-mean-square height of the surface waves</td>
</tr>
<tr>
<td>Bottom Height rms [m]</td>
<td>Root-mean-square height of the bottom</td>
</tr>
</tbody>
</table>
Figure 6.2: Ray trajectories for Scenario 8 with 50 rays. The bottom of Scenario 8 is flat with a big hill in the middle and the transmitter is located at the coordinates (100, 400). Some of the rays showcase the occurrence of turning points around depths 50 to 100 m.

6.3 Ray-tracing

The first step when calculating the sound field is the collecting of data through ray-tracing. The goal of this step is to collect all the data necessary for calculating the eigenrays at a later step. At this step only the receiver depth is specified and the horizontal coordinate is not specified until later at the eigenray interpolation step. Receiver depth is simply the vertical coordinate of the receiver. A large number of rays are traced from the transmitter with starting angles evenly spaced out between the minimum and maximum specified angles. These rays propagate through the scene and curve through the water because of the refraction that occurs due to the variation of sound speed with depth (see Figure 6.2). Whenever a ray intersects the chosen receiver depth, all the data that is needed for the calculations in the succeeding stages are recorded in a buffer. An overview of the data that is required to be stored for every ray-receiver intersection can be seen in Table 6.2.

A large portion of the calculations that are performed in this stage are the calculations from the refractions between every water layer. The new direction of a ray after a water layer intersection is calculated using a heuristic based on Snell’s law (see Chapter 2). Since all these computations are done in intersection programs in OptiX they are automatically parallelized to the best of OptiX’s ability.

6.3.1 The Scene

Before any rays can be traced, the OptiX scene has to be built. The scene in OXSP is a flat hierarchy with a Group node as the root node. This group node is the parent to every geometry in the scene (Figure 6.3). The different types of geometries that exist in the scene are:
Table 6.2: Ray specific data is continually updated during ray-tracing and is saved when a ray intersects with the receiver depth. The saved data for each of these intersections is what is shown in this table. This data is then used for calculations in later steps of OXSP.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>The horizontal coordinate of the intersection with the receiver depth.</td>
</tr>
<tr>
<td>B</td>
<td>The accumulated transmission loss coefficient due to reflections with the bottom.</td>
</tr>
<tr>
<td>S</td>
<td>The accumulated transmission loss coefficient due to reflections with the sea surface. The sign of S also determines the phase shift of the sound (negative for a phase shift of ±180°).</td>
</tr>
<tr>
<td>t</td>
<td>The time it took for the ray to propagate to the point of the intersection.</td>
</tr>
<tr>
<td>trDist</td>
<td>Total distance traveled up to the intersection point.</td>
</tr>
<tr>
<td>angle</td>
<td>The angle between the rays current direction and the horizontal plane.</td>
</tr>
<tr>
<td>initialAngle</td>
<td>The initial angle of the ray (the angle between the direction the ray was originally sent out from the transmitter and the horizontal plane).</td>
</tr>
<tr>
<td>turningPoints</td>
<td>Number of turning points the ray has had before the intersection point. This is used to calculate the correct phase shift due to the turning points.</td>
</tr>
<tr>
<td>history</td>
<td>The history contains the rays history in an encoded format that is used for classification (see Section 6.3.2).</td>
</tr>
</tbody>
</table>
• **Surface.** The surface is represented as a plane located at \( z = 0 \). When a ray intersects the surface plane the ray loses some of its energy based on the angle between the ray's direction and the horizontal surface plane and the rms height of the waves on the surface. The reflection loss is calculated and stored before the ray is reflected back downwards. Every time a ray hits the surface the signal is phase shifted by 180°.

• **Bottom.** The bottom is represented as a mesh of triangles and is built based on the bathymetry given as input to the program. Other than potentially having a more complicated shape than the surface it works much the same way, except that there is no phase shift associated with bottom reflections.

• **Target.** The target for the active Sonar mode is represented as a mesh of triangles in the shape of a circle and a transform which is used to move the target to the desired position. The reason for using a transform is so that the target can be moved without the scene requiring a rebuild, which makes it a lot faster when moving the target. Since the target is only used in the active Sonar mode, the target is moved outside of the scene when it is not needed. When a ray hits the target the ray is reflected.

• **Water layers.** The water layers are planes evenly spaced out between the sea surface and sea bottom. They are responsible for modeling the refraction that occurs due to the variation in sound speed with depth. Every time a ray intersects with a water layer plane, the ray's new direction is calculated based on the angle of the ray with the plane and the sound speeds in the layer above or below the layer. Also, if the layer is the closest one to the

---

**Figure 6.3:** Simplified view of the structure of the OptiX scene in OXSP. Every type of geometry group is assigned a certain material which comes with customizable hit programs.
Table 6.3: Ray interactions and their associated values.

<table>
<thead>
<tr>
<th>Interaction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>Surface Reflection</td>
<td>1</td>
</tr>
<tr>
<td>Bottom Reflection</td>
<td>2</td>
</tr>
<tr>
<td>Upper Turning Point</td>
<td>3</td>
</tr>
<tr>
<td>Lower Turning Point</td>
<td>4</td>
</tr>
<tr>
<td>Target hit</td>
<td>5</td>
</tr>
</tbody>
</table>

receiver depth the data required for later stages of the program is stored in a buffer.

6.3.2 Ray Classification

In order to be able to determine one eigenray for every type of path that the sound propagates from the transmitter to a receiver depth some method is needed to distinguish between rays with different history. One straightforward way to do this is to store all the necessary data for every ray that intersects the receiver depth and then in the post-processing stage split all the intersecting rays into different classes where all rays belonging to a specific class have the same history in terms of the ray interactions such as reflections and turning points. This is the approach that for example Hovem [12] took, but it has a few downsides. More data has to be stored in memory and it also adds another step in post-processing which takes some computation time.

A more efficient and flexible way to do this is to instead calculate each ray’s class on-the-fly. In OXSP this is done by storing an extra variable $q$ with each ray and incrementing it whenever the ray has an interaction that affects the ray’s class. The interactions used to determine the classes and their associated value can be seen in Table 6.3.

Whenever an interaction of any of these types occur, $q$ is updated according to

$$q = q + 6^n v$$  \hspace{1cm} (6.1)

where $6$ is the number of different interactions that can happen, $n$ is the number of previous interactions the ray has had and $v$ is the value number seen in Table 6.3. The value of $q$ can then later be decoded to get the actual interactions in the occurred order if needed, e.g. when debugging. This approach saves both memory (which is the biggest bottleneck of the application) and calculation time, and is automatically parallelized as OptiX handles the rays in parallel.

6.4 Analyzing and Sorting Collected Data

After all the data has been collected from the ray-tracing, the next step is to process this data in order to prepare it for the next step. When tracing rays in
Figure 6.4: Eigenray interpolation. The dashed red line is the actual eigenray. The eigenray is calculated by interpolating based on the properties of the blue ray (which intersects the point P1) and the green ray (which intersects the point P2).

OptiX, the way in which the ray intersection programs can communicate back data to the main program is through buffers. These buffers have to be preallocated which means that their sizes are not dynamic. Because of this, the buffer where all the data from the ray-tracing is stored is filled with a varying amount of unused padding. This padding is removed in this step in order to make the data more manageable. Next the data is sorted according to their classes, and internally each class is sorted by the r-coordinate at which they intersected the receiver depth. The number of unique classes that exist for the receiver depth are also counted and stored for later use.

All the calculations performed in this step are well suited for parallelization and are therefore done on the GPU using CUDA. By utilizing the supported OptiX/CUDA interoperability and doing the calculations on the GPU, transferring data back and forth between the GPU and the CPU is avoided. This alone provides substantial performance gains as there can be a lot of data and memory transfers are often the cause of bottlenecks in applications offloading calculations to the GPU. The sorting is done using the CUDA library Thrust.

6.5 Interpolation of Eigenrays

Since the receiver is modeled as a point it is extremely unlikely that any of the traced rays are going to intersect the receiver directly. To solve this, the eigenrays are interpolated from the rays that intersect the receiver depth closest to each side of the receiver (see Figure 6.4). One eigenray is calculated for every type of path (i.e. class) that the sound propagates from the transmitter to the receiver. Specifically, the properties that are interpolated at the receiver point are reflection loss, propagation time, propagation distance and geometrical spreading loss. The sorting performed in the previous step makes it faster to find the closest intersection points to the receiver depth to the left and right side of the receiver within every class. Again, the calculations for every eigenray can be done in parallel on the GPU.
6.6 Signal Processing

Signal processing is the step where the transfer function is calculated. The input to this module is the interpolated eigenray data. Another important input is the user specified number of samples and sample frequency, denoted $S$ and $f_s$ respectively, which together control the resolution of the transfer function. To avoid aliasing, a sample frequency above two times the Nyquist frequency of the source signal should be chosen. The transfer function is given in the frequency domain and internally one transfer function per eigenray is calculated but they are then added together to get the complete transfer function.

Partial terms of the transfer functions, such as the geometrical spreading loss and the interactions with the surface and the bottom, were calculated in previous steps and so only only require two additional multiplications in rectangular coordinates for this step. The signal delay due to travel time results in a phase shift according to

$$\exp\left(-\frac{i2\pi D f}{f_s}\right)$$

where $D$ denotes the actual number of samples that correspond to the real valued time delay which was calculated during the ray-tracing step. The number of samples and sample frequency influence the total size of the time window size, $W$, in the time domain according to

$$W = \frac{S}{f_s}$$

which means that in the case of a time delayed signal, it can become circularly shifted if the time window is not big enough and should therefore be taken into account.

As for the phase shifts associated with turning points, those are accounted for according to

$$\exp\left(-\frac{i\pi T}{2}\right)$$

where $T$ denotes the number of turning points. It is also in this step that the sea absorption loss model is taken into account which is detailed in Section 2.3.

There is potential for a high degree of parallelism when calculating the transfer function because the operations required to compute the transfer function of a single sample are independent from the operations on the other samples. Also, since the transfer function of each eigenray is calculated at least internally, every sample for every transfer function belonging to every eigenray can be calculated independently which gives a degree of parallelism that is dependent on $S$ and the number of eigenrays.

Another task assigned to signal processing is to generate a source pulse. OXSP can generate a Ricker pulse as a source signal and a time domain example can be seen in Figure 6.5. The corresponding frequency domain amplitude spectrum can be seen in Figure 6.6. The Ricker pulse can be computed by

$$x(t) = \left(1 - 2\pi^2 f_c^2 t^2\right)\exp\left(-\pi^2 f_c^2 t^2\right)$$
Figure 6.5: Ricker pulse with center frequency 200 Hz in the time domain.

Figure 6.6: Amplitude spectrum of a Ricker pulse with center frequency 200 Hz.
where \( f_c \) is the center frequency. The user can provide \( f_c \) as a parameter that is then used when generating the Ricker pulse and its amplitude spectrum will have the appearance of a band-pass filter around \( f_c \). This puts a requirement on choosing \( f_s \) such that the transformed source signal is well contained within \( f_s \), for example choosing \( f_s \) such that \( f_s > 10f_c \) holds should suffice.

To generate a time response, the source pulse is Fourier-transformed, multiplied with the transfer function and the result is then inverse Fourier-transformed. Just like the case of computing the transfer function, the generation of the Ricker pulse and multiplications in the frequency domain can be parallelized similarly as the calculation of each sample is independent. The transformations are done using the CUDA library cuFFT which provides a parallel implementation of the Fast Fourier Transform.

### 6.7 Graphical Presentation

The application comes with a GUI (see Figure 6.7) where parameters and modes of operation can be changed from. The scene is visually represented in another
Figure 6.8: Heatmap for Scenario 8. This figure showcases the program’s ability to visualize a heatmap with a varying bathymetry that has the form of a rounded hill. Also to note is that this heatmap corresponds to the same scenario as in Figure 6.2 which shows the trajectories of the rays.

Figure 6.9: Heatmap for Scenario 9.

Figure 6.10: Heatmap for Scenario 10.

Figure 6.11: Heatmap for Scenario 11.
part of the GUI where positional parameters can also be altered by clicking with the mouse. When operating in the Heatmap mode, a heatmap will be visualized as colors spanning between red and blue where red represents the highest intensity and blue the lowest (see Figures 6.8 to 6.11) using OpenGL. The heatmap shows the sound intensity based on incoherent transmission loss, which means that it does not take the phase of the sound waves into account. For this reason, the heatmap does not provide very high accuracy, but it can be used to get an idea of the propagation paths of the sound and in what areas the sound intensity is higher in general.

6.8 Memory Considerations

Because memory transfers between GPU and CPU are one of the common limiting factors when doing computations on a GPU, measures were taken to prevent as many of the transfers as possible. To this end, all data generated from the ray-tracing was kept on the GPU and subsequent computations on that data was further calculated and kept on the GPU. The problem is that every ray that is traced requires a memory footprint that is case dependent, and because it is not possible to know how much memory is required for each individual ray beforehand it is hard to find an optimal amount of memory to allocate. On top of that, there are the problems of trying to efficiently store the data generated between multiple threads without overwriting previously written memory and requirements of preallocating buffers in OptiX. Because of this, it seemed faster and safer to overallocate memory, divide that into segments and assign each segment to separate threads.

Since the GPU has a limited memory and it was desired to allow for a variable number of rays to be cast, it was important to save as little data from the ray-tracing step as possible as to not run out of memory. It turns out that a lot of data does not need to be saved if it is calculated at the same time as the ray-tracing is done. For example, if the classification of rays (Section 6.3.2) is partly done during ray-tracing, there is no need to save all data about ray history but instead the ray class is determined during the ray-tracing step and then only the class has to be saved which can be represented as a single number. If the objective is to find the transfer function in a single point, it is also possible to only save data on the rays that intersect with the receiver depth within a certain range in the horizontal coordinate as opposed to saving all intersection points which would require more memory. This is possible because the only intersection points you need in order to interpolate the eigenrays are the points closest to the left and right of the receiver. However, as it is impossible to know beforehand which intersection points are going to be the closest ones, you need to save all intersection points within a certain minimum range.

When optimizing it is important to recognize that there are optimizations that can have an impact on the accuracy of the result. For example, the numerical accuracy of doing the floating point calculations in single- or double-precision, and then saving the result in corresponding precision becomes a consideration of
execution time versus accuracy. OptiX uses single precision internally, for example when calculating intersections. While it is possible to use double-precision within an OptiX program or have double-precision variables in the payload, the internal ray-tracing operations are still done in single-precision and can be a limiting factor for certain applications in terms of accuracy.
OXSP produces a transfer function and time response from the transmitter to the receiver. Each eigenray’s contribution to the transfer function and time response can also be produced as separate transfer functions and time responses if desired, making it easy to further analyze the different propagation paths. Figure 7.1 and Figure 7.2 show an example of a transfer function for Scenario 1 in which there are eight different eigenray contributions. The time response resulting from this scenario when simulating a Ricker pulse can be seen in Figure 7.3 where every eigenray’s contribution can be seen side-by-side, and in Figure 7.4 which is the final time response acquired when summing up all the eigenrays.

Figure 7.5 shows the ray-tracing execution times of Scenario 1 for PlaneRay, Bellhop and OXSP with different settings. The S&G error factor (Section 5.3.3) when comparing the time responses of OXSP using triangles or rectangles is 0. There is no difference between the time responses, which can be seen in Figure 7.6. The execution time also depends on how many layers the water is split into. Figure 7.7 shows the execution time of OXSP for Scenario 1 with a varying number of layers. Table 7.1 shows the speed-ups achieved with triangles as compared to rectangles and the estimated speed-up gained by using the RT cores for a varying number of rays. Figure 7.8 shows the full execution time of OXSP from the ray-tracing stage until the transfer function has been calculated.

Because the end result might be needed on the CPU it is also relevant to take into account the time it takes to transfer the result from GPU to CPU memory. Figure 7.9 shows the average time it takes to transfer a varying number of samples.

The accuracy of the result depends on how many rays are traced and how many layers the water is split into. Figures 7.10 to 7.12 show the S&G error for three different scenarios with a varying number of traced rays and Figure 7.13 shows the S&G error for Scenario 1 with a varying number of layers.
Figure 7.1: Amplitude spectrum of transfer function for Scenario 1.

Figure 7.2: Phase spectrum of transfer function for Scenario 1.
Figure 7.3: Eigenray contributions to the time response for Scenario 1. Each color represents the contribution of one propagation path from the transmitter to the receiver.

Figure 7.4: Final time response for Scenario 1. This response is the sum of all the contributions shown in Figure 7.3.
Figure 7.5: Ray-tracing execution times for Scenario 1 with a varying number of rays. The execution times are measured as the average of 25 runs. The shaded areas show the standard deviations.

Figure 7.6: Comparison of the time responses calculated by OXSP for Scenario 7 using triangles/rectangles. There is no difference between the two time responses.
Figure 7.7: Ray-tracing execution times for Scenario 1 with a varying number of layers. The execution times are measured as the average of 25 runs. The shaded area shows the standard deviation.

Figure 7.8: Full execution time of OXSP for Scenario 6 excluding scene building. The execution times are measured as the average of 25 runs. The shaded areas show the standard deviations.
Table 7.1: RT-core speed-up. The spans in the parentheses are the speed-up spans based on the standard deviation. The RT core speed-up is taken as the RTX 2060 speed-up divided by the GTX 1060 speed-up. For more details, see Section 5.3.1.

<table>
<thead>
<tr>
<th>Rays</th>
<th>GTX 1060 Speed-up</th>
<th>RTX 2060 Speed-up</th>
<th>RT core Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^7</td>
<td>1.06 (0.85-1.32)</td>
<td>1.18 (0.78-1.72)</td>
<td>1.11 (0.59-2.03)</td>
</tr>
<tr>
<td>2^8</td>
<td>1.06 (1.04-1.08)</td>
<td>1.38 (1.31-1.45)</td>
<td>1.30 (1.21-1.39)</td>
</tr>
<tr>
<td>2^9</td>
<td>1.08 (1.06-1.09)</td>
<td>1.38 (1.31-1.45)</td>
<td>1.28 (1.20-1.37)</td>
</tr>
<tr>
<td>2^10</td>
<td>1.09 (1.06-1.11)</td>
<td>1.37 (1.32-1.43)</td>
<td>1.26 (1.19-1.34)</td>
</tr>
<tr>
<td>2^11</td>
<td>1.11 (1.08-1.14)</td>
<td>1.37 (1.31-1.43)</td>
<td>1.23 (1.15-1.32)</td>
</tr>
<tr>
<td>2^12</td>
<td>1.13 (1.11-1.15)</td>
<td>1.39 (1.33-1.45)</td>
<td>1.23 (1.16-1.31)</td>
</tr>
<tr>
<td>2^13</td>
<td>1.16 (1.15-1.18)</td>
<td>1.33 (1.28-1.39)</td>
<td>1.15 (1.09-1.21)</td>
</tr>
<tr>
<td>2^14</td>
<td>1.27 (1.24-1.30)</td>
<td>1.38 (1.34-1.42)</td>
<td>1.09 (1.03-1.14)</td>
</tr>
<tr>
<td>2^15</td>
<td>1.26 (1.24-1.28)</td>
<td>1.25 (1.22-1.29)</td>
<td>1.00 (0.95-1.04)</td>
</tr>
<tr>
<td>2^16</td>
<td>1.24 (1.23-1.25)</td>
<td>1.18 (1.15-1.21)</td>
<td>0.95 (0.92-0.98)</td>
</tr>
<tr>
<td>2^17</td>
<td>1.22 (1.22-1.23)</td>
<td>1.19 (1.17-1.21)</td>
<td>0.97 (0.95-0.99)</td>
</tr>
<tr>
<td>2^18</td>
<td>1.22 (1.22-1.23)</td>
<td>1.20 (1.19-1.20)</td>
<td>0.98 (0.97-0.99)</td>
</tr>
<tr>
<td>2^19</td>
<td>1.22 (1.22-1.22)</td>
<td>1.21 (1.20-1.21)</td>
<td>0.99 (0.98-1.00)</td>
</tr>
<tr>
<td>2^20</td>
<td>1.22 (1.22-1.22)</td>
<td>1.21 (1.21-1.21)</td>
<td>0.99 (0.98-1.00)</td>
</tr>
<tr>
<td>2^21</td>
<td>1.22 (1.22-1.22)</td>
<td>1.21 (1.21-1.22)</td>
<td>0.99 (0.99-1.00)</td>
</tr>
<tr>
<td>2^22</td>
<td>1.22 (1.22-1.23)</td>
<td>1.22 (1.22-1.22)</td>
<td>0.99 (0.99-1.00)</td>
</tr>
<tr>
<td>2^23</td>
<td>1.22 (1.22-1.23)</td>
<td>1.22 (1.22-1.22)</td>
<td>0.99 (0.99-1.00)</td>
</tr>
<tr>
<td>2^24</td>
<td>1.22 (1.22-1.22)</td>
<td>1.22 (1.22-1.22)</td>
<td>0.99 (0.99-1.00)</td>
</tr>
</tbody>
</table>
Figure 7.9: Memory transfer times from GPU to CPU. Each sample is 4 B in size. The transfer times are measured as the average of 25 runs. The shaded area shows the standard deviation.

Figure 7.10: S&G comprehensive error factor for Scenario 3 with a varying number of rays.
Figure 7.11: S&G comprehensive error factor for Scenario 4 with a varying number of rays.

Figure 7.12: S&G comprehensive error factor for Scenario 5 with a varying number of rays.
7.1 Validation

OXSP was validated by comparing the time responses from a few scenarios with the time responses that were output by PlaneRay using the same scenarios. Figure 7.14 shows the time responses for Scenario 1, Figure 7.15 shows the time responses for Scenario 2 and Figure 7.16 shows the time responses for Scenario 7.

7.2 Floating-point Precision Format and Intrinsics

The time responses from the three different precision format versions of OXSP for Scenario 1 and Scenario 5 with 1500 rays can be seen in Figure 7.17 and Figure 7.18 respectively. The versions differ by what floating point precision format is used and whether intrinsic functions were used or not. Figure 7.19 shows the execution times from the ray-tracing step when using intrinsics, single- and double-precision floating point format for Scenario 1. The S&G error of the different versions of OXSP for Scenarios 3 to 5 when varying the numbers of rays can be seen in Figures 7.20 to 7.22. The S&G error of the different versions of OXSP for Scenario 1 when varying the number of layers can be seen in Figure 7.23.
Figure 7.14: Comparison of time responses of OXSP and PlaneRay for Scenario 1. The time response of PlaneRay can just barely be seen behind the time response of OXSP.

Figure 7.15: Comparison of time responses of OXSP and PlaneRay for Scenario 2. The time response of PlaneRay can just barely be seen behind the time response of OXSP.
Figure 7.16: Comparison of time responses of OXSP and PlaneRay for Scenario 7. The time response of PlaneRay can just barely be seen behind the time response of OXSP.

Figure 7.17: Time responses for different precision formats and use of intrinsics for Scenario 1.
Figure 7.18: Time responses for different precision formats and use of intrinsics for Scenario 5 with 1500 rays.

Figure 7.19: Average ray-tracing execution times for different precisions for Scenario 1 with a varying number of rays. The execution times are measured as the average of 25 runs. The shaded areas show the standard deviations.
Figure 7.20: S&G comprehensive error factor for Scenario 3 with a varying number of rays.

Figure 7.21: S&G comprehensive error factor for Scenario 4 with a varying number of rays.
**Figure 7.22:** S&G comprehensive error factor for Scenario 5 with a varying number of rays.

**Figure 7.23:** S&G comprehensive error factor for Scenario 1 with a varying number of layers.
This chapter begins with an analysis of the results from the previous chapter and ends with a discussion about the methods that were used.

8.1 Results

The goal of speeding up the computations by utilizing a GPU was achieved and the ray-tracing stage of OXSP is up to 310 times faster than that of the CPU based Fortran implementation Bellhop. It should be noted that Bellhop is a single-threaded implementation, but even if it would be multi-threaded and run on a 16 core CPU with perfect speed-up OXSP would still be around 20 times faster. The execution time of OXSP on the RTX 2060 graphics card is nearly constant until around 8000 rays are traced, after which point the execution time scales linearly with the number of rays. This can be explained by the fact that the GPU has a total of 1920 CUDA cores and four warp schedulers per SM, so at around 8000 rays the GPU achieves full occupancy. Likewise, the execution time on the GTX 1060 is nearly constant until around 2000 rays are traced, due to the fact that the GPU has 1152 CUDA cores and two warp schedulers per SM. The execution time of OXSP scales linearly with the number of rays that are traced after full occupancy is achieved, and linearly with number of layers used. Doubling the number of layers has the effect of doubling the number of calculations for every ray as it traverses the scene.

Assessing the speed-up gained by using the RT cores is difficult as there is no way to run the same code on the same GPU with the RT cores toggled. One way to estimate the speed-up is to compare the difference between using triangles or rectangles on the RTX 2060 to the difference between using triangles or rectangles on the GTX 1060. As Figure 7.5 and Table 7.1 show, the RT cores appear to provide some speed-up when tracing 128-8192 rays, but then the speed-
up appears to converge to 1. It is hard to draw any conclusions about the RT cores from these tests as it is unclear exactly what causes the difference in execution time on both GPUs with the two versions (triangles/rectangles) tested. One possible explanation for why the speed-up is not larger is that the RT cores are designed to speed up BVH traversal and intersection testing in scenes with millions of rays and potentially millions of geometric primitives to intersect with. In a typical OXSP scenario there are only a few thousand rays and a few hundred or thousand polygons, mostly depending on the bathymetry, and most of the ray-traversal computation time is spent calculating the refraction that occurs at every intersection with a water layer. These refraction calculations are all performed in the CUDA cores, so the data has to be transmitted back and forth between the CUDA cores and the RT cores. However, as Figure 7.5 shows though, the fastest of all the tested versions was RTX 2060 using triangles (i.e., using RT cores). One interesting thing to note is that the best performance achieved for OXSP on the GTX 1060 was with RTX mode disabled. This may be because RTX mode is optimized for newer GPU architectures.

A downside of doing all the calculations on the GPU is that you may have to transfer the result to the CPU afterwards (if so desired). Figure 7.5 together with Figure 7.9 show that the time to transfer the result to the CPU is only a small fraction of the time it would take to calculate it directly with one of the CPU implementations. While the transfer time depends on the number of samples, for it to be a relevant part of the total run time the ratio of number of samples to the number of rays would have to be very large.

On top of being faster, OXSP can also handle a much larger amount of rays than the other programs before running out of memory due to being more efficient, but how many rays are necessary for a good result? The answer to this question varies from case to case and depends a lot on parameters such as the sound speed profile, the bathymetry and the positions of the transmitter and receiver. Figure 7.10 shows how the S&G error metric varies as the number of rays increases for a simple scenario with no turning points involved (Scenario 3). The best accuracy that can be expected from OXSP is with an error factor of around $C = 10^{-7}$. For Scenario 3 this is achieved already when tracing about 50 rays, after which point the error does not decrease much no matter how many more rays are traced. Figure 7.11 shows the error for a slightly more complicated scenario where one of the eigenrays has gone through an upper turning point (Scenario 4). Because the paths of the eigenrays for Scenario 4 are a bit more complicated, the error does not stabilize until after around 150 rays are traced. Figure 7.12 shows an even more complicated scenario in which the receiver is at a point where the nearby ray intersections are more spaced out (Scenario 5). In this case at least 1000 rays have to be traced in order to get the best result possible. The result also depends on how many layers the water is split into. Figure 7.13 shows how the S&G error gets smaller as the number of layers increase.

Based on testing in regards to both execution time and accuracy it is recommended to trace 1000-5000 rays and split the water into 200-1000 layers depending on the required accuracy, time constraints, what GPU is used and whether the GPU is used for other concurrent computations. Due to other limitations such as
the use of single-precision floating-point internally in OptiX, the result can not be expected to improve much after this point. In this interval OXSP is already 200-300 times faster than the fastest tested CPU program Bellhop. For the tested scenarios the time responses are very close to those of PlaneRay (Figure 7.14 and Figure 7.15). Based on the validation tests against PlaneRay, the time response results of OXSP are reliable.

### 8.1.1 Floating-point Precision Format and Intrinsics

According to Figure 7.19 it is clear that in terms of execution time it is faster to make use of the intrinsic functions which also implies the use single-precision (32-bit) floating-point format while the double-precision (64-bit) floating-point version of OXSP performed the slowest. This comes as no surprise as the whole point of intrinsic functions is to provide faster versions to some common functions at the cost of accuracy. It is also not surprising that the single-precision version of OXSP outperforms the double-precision version in terms of speed as each multiprocessor contains more single-precision floating-point cores than double-precision floating-point cores.

For a large number of rays, the ray-tracing speed for the different versions of OXSP scale linearly with the number of rays. Whether a problem is well suited for the use of single-precision and intrinsics is dependent on how accurate the results of the calculations are required to be. As can be seen in Figure 7.17, the time responses for the relatively simple Scenario 1 differ slightly depending on the precision used. The single-precision version can be seen to differ in amplitude for the later parts of the time response while the intrinsics version differs in amplitude as well as in the time offset. The error in the time offset also seems to be dependent on the travel distance of that particular eigenray’s contribution to the total time response. For example, the time distance between the two right-most peaks is 0.00171 s which corresponds to a 2.565 m difference in distance that the sound has traveled when comparing the double-precision version to the intrinsics version.

Scenario 5 is more complicated in part because it uses a more complex sound speed profile. Figure 7.18 shows that the resulting time responses for Scenario 5 with 1500 rays differ even more than the time responses of Scenario 1. While there might some cases where some inaccuracies may be considered acceptable, Figure 7.18 shows that it is easy to find cases where the time responses are very inaccurate if double-precision is not used. The differences in time responses indicate that some of the ray-tracing calculations are sensitive to the inaccuracies introduced by not using double-precision floating-point and in particular the ray paths seem to change, or at the very least the results for calculating the partial time steps (Equation (2.10)) become inaccurate. Another problem can be seen in Figure 7.23 where the time responses from the single-precision and intrinsics versions of OXSP do not seem to converge to some constant for any practical number of layers. In Figures 7.20 to 7.22 it is possible to see an oscillating behavior in the S&G errors for the non double-precision floating-point versions of OXSP when varying the number of rays. While convergence could be argued for some of the
scenarios, it is not conclusive for all presented scenarios which is a property only the double-precision version seems to have. This makes the non double-precision versions very hard to validate as the expected behavior when increasing the number of rays is to give more accurate results which would require a convergent behavior in the S&G errors as the number of rays increase. It is also problematic that an early inaccuracy in the ray-tracing may cause a cascading effect in the subsequent ray-tracing calculations meaning that the corresponding rays in the different versions take different paths.

For the reasons mentioned it was not possible to use any other version of OXSP than the double-precision floating-point version for validation and although there may be applications where the gain in speed of using the other versions outweighs the cost of the inaccuracies in the time responses and divergent behavior of the error function, they are not apparent.

8.2 Method

The OXSP ray model is all based on a paper by Hovem [11] and its validity therefore relies entirely on the validity of Hovem’s work. The output of OXSP was validated against the output of PlaneRay for the same parameters and they were nearly identical.

One of the biggest weaknesses of OXSP is the bottom reflection model. The bottom is modeled using only a coherent reflection coefficient which has the effect of weakening the signal. In reality the bottom is more complicated than this and one approach, which is used in PlaneRay, is to model the bottom as a fluid sedimentary layer over a homogeneous solid half space. This has the effect of introducing frequency dependent phase shifts and attenuation of the signal, which is something that OXSP does not take into account. In OXSP, the frequency used in the frequency dependent calculations (for acoustic absorption in the water and the coherent reflection coefficient) is also approximated by a center frequency for performance reasons. This is a reasonable approximation if the signal is a narrowband signal around the center frequency. As the results show, there is no need to send out more than at the most 10,000 rays (whereas OXSP can handle millions of rays), and as a result the performance considerations behind the simplifications were probably unnecessary as the execution time and memory requirements for tracing this amount of rays are already low.

Another simplification is that the model is entirely 2D-based. The accuracy of this for a real-world scenario is unclear. If there is no land mass nearby to reflect on, the sound that propagates in any other direction than the direction from the transmitter to the receiver will have no effect on the end result. However, if there is land nearby the sound would reflect on the land which could add even more eigenrays that contribute to the transfer function. These issues could be improved upon in the future if the model is to be used in scenarios where a high degree of realism is necessary.

When evaluating how many rays to trace, the time response resulting from tracing 1,000,000 rays was used as a reference. The result would be more reliable
with a better reference as there is no way to be sure that the result of 1,000,000 rays is actually any better than the result from, say, 100 rays. However it does still show that as the number of rays increases, the error tends to converge to some constant.

The method of evaluating the RT core speed-up is highly questionable, but no better method was found. It would have been a lot better comparison if it was possible to disable the use of the RT cores and then compare the performance with them enabled/disabled with the exact same program and GPU. Unfortunately this is not possible, so a roundabout method had to be used. However, the method used turned out to be inconclusive.

When comparing the different floating-point format versions of OXSP, the only operations and variables that were changed between the versions were the ones associated with ray-tracing. It is also likely that all partial results are not equally sensitive to numerical inaccuracies and that some could be done in single-precision floating point format for all versions. An underlying problem with presenting different scenarios is that it is difficult to draw any general conclusions but instead the investigation is limited to the cases that are brought to attention and can never be seen as definitive proof. That said, the potential performance gain of changing some calculations to single-precision format is relatively small.
Conclusions

In this thesis it has been shown how sound propagation in water can be simulated on a GPU using ray-tracing and the OptiX ray-tracing engine. Ray-tracing is a problem well suited for parallel implementations and as a result speed-ups of up to 310x were achieved on an Nvidia RTX 2060 graphics card compared to the CPU based Fortran implementation Bellhop. RTX 2060 is a graphics card based on Nvidia’s Turing architecture which has specialized hardware acceleration for ray-tracing in its RT cores. These cores are very efficient at traversing BVHs and testing for ray-primitive intersections, and according to Nvidia they can provide a large speed-up for ray-tracing. As there is no way to disable the use of RT cores for performance comparison some roundabout method had to be used when evaluating the speed-up gained from the RT cores. Although the best performance of all the tested versions of OXSP was achieved when using the RT cores, the method for estimating the RT core speed-up proved inconclusive.

Nevertheless, the main conclusion is that sound propagation in water can be simulated very efficiently on a GPU, and large speed-ups can be gained as compared to CPU implementations even when taking into account the time it takes to transfer the result from the GPU to the CPU.

9.1 Future Work

There were many possible features that were left out of OXSP due to the limited scope of this thesis. Some features that could be added are consideration of sound speed profiles that vary with range as well as depth, varying the sea surface geometry and direct plotting of ray paths, the time response and the transfer function. It would also be an interesting idea to trace the rays and perform the calculations in a 3D environment on a GPU. This would require a revised model and major changes to the current implementation but there are no obvious reasons for why
the current model could not be extended. Simulation of underwater sound propagation in a 3D environment has been done on CPUs in the past by Reilly [31] but not on GPUs as far as we can tell.

The bottom reflection model can be greatly improved upon by taking into account different bottom materials. For instance, reflections against a hard, rocky surface will naturally differ from that of sound propagating into a multi-layered bottom material with different densities which can be the case with a sediment or sand bottom. In reality, sand or sediment bottoms are a mixture of different materials, including water, and such bottoms could have a density that changes continuously. Sea bottom modeling is complicated and no model seems to fit every sea bottom which makes it an area that could be further researched.

If you know in advance exactly what parameters you want to use and limit the features of the application to only what you need, the code could easily be further optimized. For example, if you limit the maximum number of rays that can be traced then it would be possible to save more information per ray without the risk of running out of GPU memory. OXSP aims to minimize the required amount of memory in order to allow tracing millions of rays concurrently, but as the results show there is no point tracing that many rays. One situation where this can speed up the execution time significantly is if you want to calculate the sound field in a number of points with different depths concurrently. OXSP currently only saves ray data for one depth at a time and then loops the process once for every depth. Points on the same exact depth can all be calculated in parallel from the same data, which means that the execution time is almost constant until full GPU occupancy is reached, whereas adding points on new depths increases the execution time linearly. Although the memory management would have to be redesigned a bit this could easily be improved so that data is collected for all depths of interest at the same time, and as a result all the depths could be calculated in parallel.
Appendix
This appendix contains the program parameters of the scenarios used for the plots presented in this thesis.
### Table A.1: Scenario 1 parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitter Position [m]</td>
<td>(0, 50)</td>
</tr>
<tr>
<td>Receiver Position [m]</td>
<td>(3000, 250)</td>
</tr>
<tr>
<td>Launch Angle [°]</td>
<td>[-20, 20]</td>
</tr>
<tr>
<td>Number of Rays</td>
<td>1500</td>
</tr>
<tr>
<td>Max Range [m]</td>
<td>4000</td>
</tr>
<tr>
<td>Max Depth [m]</td>
<td>287</td>
</tr>
<tr>
<td>Number of Layers</td>
<td>287</td>
</tr>
<tr>
<td>fs [Hz]</td>
<td>16384</td>
</tr>
<tr>
<td>Number of Samples</td>
<td>131072</td>
</tr>
<tr>
<td>Center Frequency [Hz]</td>
<td>200</td>
</tr>
<tr>
<td>Surface Height rms [m]</td>
<td>0</td>
</tr>
<tr>
<td>Bottom Height rms [m]</td>
<td>0</td>
</tr>
<tr>
<td>Sound Speed Profile</td>
<td>Sound Speed Profile 1</td>
</tr>
<tr>
<td>Bathymetry</td>
<td>Flat</td>
</tr>
</tbody>
</table>

### Table A.2: Scenario 2 parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitter Position [m]</td>
<td>(0, 200)</td>
</tr>
<tr>
<td>Receiver position [m]</td>
<td>(2000, 100)</td>
</tr>
<tr>
<td>Launch Angle [°]</td>
<td>[-20, 20]</td>
</tr>
<tr>
<td>Number of Rays</td>
<td>1500</td>
</tr>
<tr>
<td>Max Range [m]</td>
<td>4000</td>
</tr>
<tr>
<td>Max Depth [m]</td>
<td>287</td>
</tr>
<tr>
<td>Number of Layers</td>
<td>287</td>
</tr>
<tr>
<td>fs [Hz]</td>
<td>16384</td>
</tr>
<tr>
<td>Number of Samples</td>
<td>131072</td>
</tr>
<tr>
<td>Center Frequency [Hz]</td>
<td>200</td>
</tr>
<tr>
<td>Surface Height rms [m]</td>
<td>0</td>
</tr>
<tr>
<td>Bottom Height rms [m]</td>
<td>0</td>
</tr>
<tr>
<td>Sound Speed Profile</td>
<td>Sound Speed Profile 1</td>
</tr>
<tr>
<td>Bathymetry</td>
<td>Flat</td>
</tr>
</tbody>
</table>
### Table A.3: Scenario 3 parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitter Position [m]</td>
<td>(100, 130)</td>
</tr>
<tr>
<td>Receiver Position [m]</td>
<td>(2150, 250)</td>
</tr>
<tr>
<td>Launch Angle [°]</td>
<td>[-20, 20]</td>
</tr>
<tr>
<td>Number of Rays</td>
<td>Varying</td>
</tr>
<tr>
<td>Max Range [m]</td>
<td>4000</td>
</tr>
<tr>
<td>Max Depth [m]</td>
<td>287</td>
</tr>
<tr>
<td>Number of Layers</td>
<td>4000</td>
</tr>
<tr>
<td>fs [Hz]</td>
<td>16384</td>
</tr>
<tr>
<td>Number of Samples</td>
<td>131072</td>
</tr>
<tr>
<td>Center Frequency [Hz]</td>
<td>200</td>
</tr>
<tr>
<td>Surface Height rms [m]</td>
<td>0</td>
</tr>
<tr>
<td>Bottom Height rms [m]</td>
<td>0</td>
</tr>
<tr>
<td>Sound Speed Profile</td>
<td>Sound Speed Profile 1</td>
</tr>
<tr>
<td>Bathymetry</td>
<td>Flat</td>
</tr>
</tbody>
</table>

### Table A.4: Scenario 4 parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitter Position [m]</td>
<td>(100, 130)</td>
</tr>
<tr>
<td>Receiver Position [m]</td>
<td>(3500, 125)</td>
</tr>
<tr>
<td>Launch Angle [°]</td>
<td>[-20, 20]</td>
</tr>
<tr>
<td>Number of Rays</td>
<td>Varying</td>
</tr>
<tr>
<td>Max Range [m]</td>
<td>4000</td>
</tr>
<tr>
<td>Max Depth [m]</td>
<td>287</td>
</tr>
<tr>
<td>Number of Layers</td>
<td>4000</td>
</tr>
<tr>
<td>fs [Hz]</td>
<td>16384</td>
</tr>
<tr>
<td>Number of Samples</td>
<td>131072</td>
</tr>
<tr>
<td>Center Frequency [Hz]</td>
<td>200</td>
</tr>
<tr>
<td>Surface Height rms [m]</td>
<td>0</td>
</tr>
<tr>
<td>Bottom Height rms [m]</td>
<td>0</td>
</tr>
<tr>
<td>Sound Speed Profile</td>
<td>Sound Speed Profile 1</td>
</tr>
<tr>
<td>Bathymetry</td>
<td>Flat</td>
</tr>
</tbody>
</table>
### Table A.5: Scenario 5 parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitter Position [m]</td>
<td>(230, 150)</td>
</tr>
<tr>
<td>Receiver Position [m]</td>
<td>(2900, 115)</td>
</tr>
<tr>
<td>Launch Angle [°]</td>
<td>[-20, 20]</td>
</tr>
<tr>
<td>Number of Rays</td>
<td>Varying</td>
</tr>
<tr>
<td>Max Range [m]</td>
<td>4000</td>
</tr>
<tr>
<td>Max Depth [m]</td>
<td>287</td>
</tr>
<tr>
<td>Number of Layers</td>
<td>4000</td>
</tr>
<tr>
<td>fs [Hz]</td>
<td>16384</td>
</tr>
<tr>
<td>Number of Samples</td>
<td>131072</td>
</tr>
<tr>
<td>Center Frequency [Hz]</td>
<td>200</td>
</tr>
<tr>
<td>Surface Height rms [m]</td>
<td>0</td>
</tr>
<tr>
<td>Bottom Height rms [m]</td>
<td>0</td>
</tr>
<tr>
<td>Sound Speed Profile</td>
<td>Sound Speed Profile 2</td>
</tr>
<tr>
<td>Bathymetry</td>
<td>Flat</td>
</tr>
</tbody>
</table>

### Table A.6: Scenario 6 parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitter Position [m]</td>
<td>(0, 100)</td>
</tr>
<tr>
<td>Receiver Position [m]</td>
<td>(3100, 225)</td>
</tr>
<tr>
<td>Launch Angle [°]</td>
<td>[-20, 20]</td>
</tr>
<tr>
<td>Number of Rays</td>
<td>Varying</td>
</tr>
<tr>
<td>Max Range [m]</td>
<td>4000</td>
</tr>
<tr>
<td>Max Depth [m]</td>
<td>287</td>
</tr>
<tr>
<td>Number of Layers</td>
<td>287</td>
</tr>
<tr>
<td>fs [Hz]</td>
<td>16384</td>
</tr>
<tr>
<td>Number of Samples</td>
<td>131072</td>
</tr>
<tr>
<td>Center Frequency [Hz]</td>
<td>200</td>
</tr>
<tr>
<td>Surface Height rms [m]</td>
<td>0</td>
</tr>
<tr>
<td>Bottom Height rms [m]</td>
<td>0</td>
</tr>
<tr>
<td>Sound Speed Profile</td>
<td>Sound Speed Profile 1</td>
</tr>
<tr>
<td>Bathymetry</td>
<td>Flat</td>
</tr>
</tbody>
</table>
### Table A.7: Scenario 7 parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitter Position [m]</td>
<td>(0, 200)</td>
</tr>
<tr>
<td>Receiver position [m]</td>
<td>(2000, 100)</td>
</tr>
<tr>
<td>Launch Angle [°]</td>
<td>[-20, 20]</td>
</tr>
<tr>
<td>Number of Rays</td>
<td>1500</td>
</tr>
<tr>
<td>Max Range [m]</td>
<td>4000</td>
</tr>
<tr>
<td>Max Depth [m]</td>
<td>287</td>
</tr>
<tr>
<td>Number of Layers</td>
<td>287</td>
</tr>
<tr>
<td>fs [Hz]</td>
<td>16384</td>
</tr>
<tr>
<td>Number of Samples</td>
<td>131072</td>
</tr>
<tr>
<td>Center Frequency [Hz]</td>
<td>200</td>
</tr>
<tr>
<td>Surface Height rms [m]</td>
<td>0</td>
</tr>
<tr>
<td>Bottom Height rms [m]</td>
<td>0</td>
</tr>
<tr>
<td>Sound Speed Profile</td>
<td>Sound Speed Profile 3</td>
</tr>
<tr>
<td>Bathymetry</td>
<td>Flat</td>
</tr>
</tbody>
</table>

### Table A.8: Scenario 8 parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitter Position [m]</td>
<td>(400, 100)</td>
</tr>
<tr>
<td>Launch Angle [°]</td>
<td>[-20, 20]</td>
</tr>
<tr>
<td>Number of Rays</td>
<td>1500</td>
</tr>
<tr>
<td>Max Range [m]</td>
<td>4000</td>
</tr>
<tr>
<td>Max Depth [m]</td>
<td>287</td>
</tr>
<tr>
<td>Number of Layers</td>
<td>574</td>
</tr>
<tr>
<td>Heatmap Samples</td>
<td>1000x250</td>
</tr>
<tr>
<td>Surface Height rms [m]</td>
<td>1.0</td>
</tr>
<tr>
<td>Bottom Height rms [m]</td>
<td>1.5</td>
</tr>
<tr>
<td>Sound Speed Profile</td>
<td>Sound Speed Profile 2</td>
</tr>
<tr>
<td>Bathymetry</td>
<td>Big hill</td>
</tr>
</tbody>
</table>
**Table A.9: Scenario 9 parameters.**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitter Position [m]</td>
<td>(350, 44)</td>
</tr>
<tr>
<td>Launch Angle [°]</td>
<td>[-20, 20]</td>
</tr>
<tr>
<td>Number of Rays</td>
<td>1500</td>
</tr>
<tr>
<td>Max Range [m]</td>
<td>4000</td>
</tr>
<tr>
<td>Max Depth [m]</td>
<td>287</td>
</tr>
<tr>
<td>Number of Layers</td>
<td>574</td>
</tr>
<tr>
<td>Heatmap Samples</td>
<td>1000x250</td>
</tr>
<tr>
<td>Surface Height rms [m]</td>
<td>1.5</td>
</tr>
<tr>
<td>Bottom Height rms [m]</td>
<td>2.0</td>
</tr>
<tr>
<td>Sound Speed Profile</td>
<td>Sound Speed Profile 2</td>
</tr>
<tr>
<td>Bathymetry</td>
<td>Flat</td>
</tr>
</tbody>
</table>

**Table A.10: Scenario 10 parameters.**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitter Position [m]</td>
<td>(740, 185)</td>
</tr>
<tr>
<td>Launch Angle [°]</td>
<td>[-20, 20]</td>
</tr>
<tr>
<td>Number of Rays</td>
<td>1500</td>
</tr>
<tr>
<td>Max Range [m]</td>
<td>4000</td>
</tr>
<tr>
<td>Max Depth [m]</td>
<td>287</td>
</tr>
<tr>
<td>Number of Layers</td>
<td>574</td>
</tr>
<tr>
<td>Heatmap Samples</td>
<td>1000x250</td>
</tr>
<tr>
<td>Surface Height rms [m]</td>
<td>1.5</td>
</tr>
<tr>
<td>Bottom Height rms [m]</td>
<td>2.0</td>
</tr>
<tr>
<td>Sound Speed Profile</td>
<td>Sound Speed Profile 2</td>
</tr>
<tr>
<td>Bathymetry</td>
<td>Big triangular hill</td>
</tr>
</tbody>
</table>

**Table A.11: Scenario 11 parameters.**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitter Position [m]</td>
<td>(350, 140)</td>
</tr>
<tr>
<td>Launch Angle [°]</td>
<td>[-20, 20]</td>
</tr>
<tr>
<td>Number of Rays</td>
<td>1500</td>
</tr>
<tr>
<td>Max Range [m]</td>
<td>4000</td>
</tr>
<tr>
<td>Max Depth [m]</td>
<td>287</td>
</tr>
<tr>
<td>Number of Layers</td>
<td>574</td>
</tr>
<tr>
<td>Heatmap Samples</td>
<td>1000x250</td>
</tr>
<tr>
<td>Surface Height rms [m]</td>
<td>2.0</td>
</tr>
<tr>
<td>Bottom Height rms [m]</td>
<td>3.0</td>
</tr>
<tr>
<td>Sound Speed Profile</td>
<td>Sound Speed Profile 1</td>
</tr>
<tr>
<td>Bathymetry</td>
<td>Multiple small hills</td>
</tr>
</tbody>
</table>
This appendix contains the sound speed profiles used for the scenarios used in this thesis.
**Figure B.1:** Sound speed profile 1. Linear profile.

**Figure B.2:** Sound speed profile 2. Smooth knee profile.
Figure B.3: Sound speed profile 3. Rough knee profile.


