Rogério de Moraes Calazan

Numerical enhancements and parallel GPU implementation of the TRACEO3D model

University of Algarve

Faculty of Sciences and Technology

2018

Rogério de Moraes Calazan

Numerical enhancements and parallel GPU implementation of the TRACEO3D model

Ph.D. Thesis in Electronics and Telecommunications (Signal Processing)

Developed under supervision of:

Prof. Dr. Orlando Camargo Rodríguez

University of Algarve

Faculty of Sciences and Technology

2018

Numerical enhancements and parallel GPU implementation of the TRACEO3D model

Declaração de autoria do trabalho

Declaro ser o autor deste trabalho, que é original e inédito. Autores e trabalhos consultados estão devidamente citados no texto e constam da listagem de referências bibliográficas incluída.

Rogério de Moraes Calazan

To Leonardo and Andréa

&

To my parents

Acknowledgements

I would like to thank my colleagues of the Signal Processing Laboratory (SiPLAB), for their friendship, scientific discussions and coffee-time where anxiety was healed: Ana Bela, Lussac Maia, Vicente Barroso, Agni Mantouka, Jef Philippine and Ana Catarina. In addition, I want to thank the SiPLAB research team with whom I always learned about high interdisciplinary field of underwater acoustics: Prof. António Silva, Prof. Paulo Santos, Cristiano Soares, Friedrich Zabel, specially Prof. Sérgio Jesus, the Lab coordinator, and Prof. Paulo Felisberto who gave me fundamental lessons in signal processing.

I would like to thank my supervisor, Prof. Orlando Camargo Rodríguez, for the vast experience transmitted during the development of this research, for the fundamental suggestions and for the constructive criticism that contributed to improve this work and specially for his friendship.

I am also deeply grateful to my wife and my son for their unconditional support, being patient during this journey that consumed precious time that was reserved for my family.

I would like to thank the Brazilian Navy for the possibility of full-time dedication and support which were fundamental to develop this PhD thesis.

I thank God for all the goodness and for giving me the strength to overcome this challenge.

This work was conducted with the support of the CNPq, Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil (grant 202027/2015-5), and Brazilian Navy Port. 290/MB-09/07/2015.

Name: Rogério de Moraes Calazan
College: Faculty of Sciences and Technology
University: University of Algarve
Supervisors: Orlando Camargo Rodríguez, Assistant Professor at the Faculty of Sciences and Technology, University of Algarve
Thesis title: Numerical enhancements and parallel GPU implementation of the TRACEO3D model

Abstract

Underwater acoustic models provide a fundamental and efficient tool to parametrically investigate hypothesis and physical phenomena through varied environmental conditions of sound propagation underwater. In this sense, requirements for model predictions in a three-dimensional ocean waveguide are expected to become more relevant, and thus expected to become more accurate as the amount of available environmental information (water temperature, bottom properties, etc.) grows. However, despite the increasing performance of modern processors, models that take into account 3D propagation still have a high computational cost which often hampers the usage of such models. Thus, the work presented in this thesis investigates a solution to enhance the numerical and computational performance of the TRACEO3D Gaussian beam model, which is able to handle full three-dimensional propagation. In this context, the development of a robust method for 3D eigenrays search is addressed, which is fundamental for the calculation of a channel impulse response. A remarkable aspect of the search strategy was its ability to provide accurate values of initial eigenray launching angles, even dealing with non-linearity induced by the complex regime propagation of ray bouncing on the boundaries. In the same way, a optimized method for pressure field calculation is presented, that accounts for a large numbers of sensors. These numerical enhancements and optimization of the sequential version of TRACEO3D led to significant improvements in its performance and accuracy. Furthermore, the present work considered the development of parallel algorithms to take advantage of the GPU architecture, looking carefully to the inherent parallelism of ray tracing and the high workload of predictions for 3D propagation. The combination of numerical enhancements and parallelization aimed to achieve the highest performance of TRACEO3D. An important aspect of this research is that validation and performance assessment were carried out not only for idealized waveguides, but also for the experimental results of a tank scale experiment. The results will demonstrate that a remarkable performance was achieved without compromising accuracy. It is expected that the contributions and remarkable reduction in runtime achieved will certainly help to overcome some of the reserves in employing a 3D model for predictions of acoustic fields.

Keywords: Underwater acoustics, numerical modeling, Gaussian beams, 3D propagation, parallel computing, GPU.

Nome: Rogério de Moraes Calazan
Faculdade: Faculdade de Ciências e Tecnologia
Universidade: Universidade do Algarve
Orientador: Orlando Camargo Rodríguez, Professor Auxiliar da Faculdade de Ciências e Tecnologia, Universidade do Algarve
Título da Tese: Melhorias numéricas e implementação paralela em GPU do modelo TRACEO3D

Resumo

Modelos de previsão acústica submarina são ferramentas eficientes e fundamentais para investigar parametricamente hipóteses e fenómenos físicos através de variadas condições ambientais da propagação do som subaquático. Tais modelos resolvem a equação da onda, a qual descreve matematicamente a propagação do som no oceano, para gerar previsões de campos acústicos através do cálculo do campo de pressão transmitido por um conjunto de fontes acústicas, e recebido em um conjunto de hidrofones. Além de resolver a equação da onda o modelo deve ser capaz de lidar com fenómenos adicionais como, por exemplo, perdas devido a reflexão no fundo, atenuação volumétrica e/ou espalhamento volumétrico e espalhamento devido a reflexão nas fronteiras. A necessidade de gerar previsões que levem em consideração um guia de ondas a três dimensões tem se tornado mais relevante nos últimos anos, em simultâneo com o requisito de ir melhorando as previsões a medida que aumenta a quantidade de informação ambiental (temperatura da água, propriedades do fundo oceânico, etc) disponível. Uma abordagem simples para gerar previsões em 3D consiste em "cortar" o guia 3D de ondas em transectos (planos 2D verticais), e utilizar um modelo 2D para calcular a previsão no transecto (técnica conhecida como modelagem $N \times 2D$). Entretanto, uma batimetria 3D pode induzir propagação não confinada dentro do plano 2D mesmo em casos simples, um efeito conhecido por "propagação fora-do-plano". Em termos gerais, para calcular apropriadamente o campo acústico, um modelo de propagação 3D precisa levar em consideração a variabilidade do ambiente em distância, profundidade e azimute, bem como possíveis interferências dessa variabilidade no cálculo da propagação.

A busca de autoraios (*eigenrays*, em inglês) é igualmente um aspeto importante das previsões 3D. Os autoraios podem ser definidos como raios específicos, que para uma geometria dada de um guia de ondas conectam a fonte ao recetor. O cálculo preciso de autoraios é um problema de grande interesse, porque eles são utilizados para o cálculo das previsões do sinal recebido, o qual é extremamente sensível ao tempo de propagação e ao ângulo de lançamento do raio. Num guia de ondas bidimensional o problema pode ser resolvido de modo eficiente usando um algoritmo de cálculo de raízes a uma dimensão, a qual corresponde ao ângulo de elevação; a extensão deste método de busca para encontrar autoraios em um guia de ondas tridimensional é uma tarefa complexa, a qual requer que a busca aconteça em um plano de elevação e azimute, sendo ela guiada principalmente pela minimização da distância entre a posição final do raio e a posição do hidrofone; além disso, a busca não pode acontecer ao longo de uma determinada direção devido ao regime complexo de propagação, o qual frequentemente precisa levar em conta a "propagação fora-do-plano" ou variações ambientais complexas, tais como ondas internas ou variações espaciais das fronteiras. O problema também é computacionalmente intenso, visto que requer o cálculo inicial de uma grande quantidade de raios. De fato, desde o início do seu desenvolvimento, os cálculos de

previsões de propagação em 3D são bem conhecidos por consumirem tempos elevados de processamento; apesar do aumento de desempenho dos processadores modernos tal situação verifica-se ainda hoje, o que dificulta frequentemente o emprego de tais modelos. A modo de exemplo pode ser referido que nas aplicações de processamento por ajustamento do campo (*matched-field processing*, em inglês), que requer a geração de milhares de previsões do campo acústico, os modelos 3D são preteridos em favor dos modelos 2D. As melhorias em termos de precisão que podem advir da utilização de um modelo 3D neste caso representam de fato uma das principais razões de desenvolvimento e aprimoramento deste tipo de modelos.

No contexto da discussão previamente referida foi desenvolvido um método robusto para busca de autoraios 3D, baseado no método Simplex, que foi implementado no modelo de traçamento de raios 3D TRACEO3D. A estratégia computacional de otimização Simplex foi projetada para se apoiar em uma seleção eficiente de candidatos na região inicial que inclui um determinado recetor, de modo que a pesquisa possa ser realizada eficientemente utilizando uma antena vertical ou horizontal. Um aspeto notável da estratégia de busca foi sua habilidade de prover valores precisos de ângulos iniciais de lançamento dos autoraios, mesmo lidando com a não-linearidade induzida pelo regime complexo de reflexão dos raios nas fronteiras. O método fornece uma estimativa precisa do tempo de propagação e amplitude de cada raio, que são fundamentais para prever a resposta impulsiva do canal. Adicionalmente, é apresentado um método otimizado para cálculo do campo de pressão usando um elevado número de sensores. A combinação das melhorias acima referidas permitem que o código sequencial do TRACEO3D seja computacionalmente eficiente e preciso.

Além das melhorias o desempenho do modelo foi aprimorado por intermédio da computação paralela. Regra geral (e consoante a arquitetura paralela adotada) um algoritmo sequencial precisa ser reescrito como um algoritmo paralelo para reduzir o seu tempo de computação e melhorar o seu desempenho. Paralelizar implica igualmente adicionar extensões ao algoritmo, especificando conjuntos de etapas que podem ser realizadas simultaneamente; o código paralelo também pode exigir o tratamento da sincronização de processadores nos vários estágios de execução do programa, ou o gerenciamento de acessos a posições de memória compartilhada por vários núcleos de processamento. Do ponto de vista do hardware verifica-se que para aumentar o desempenho dos programas a indústria de microprocessadores tem apostado no desenvolvimento de processadores com múltiplos núcleos dadas as limitações inerentes ao aumento da frequência de cálculo do processador. Embora na atualidade as CPUs possam conter vários núcleos de cálculo (variando entre as unidades e as dezenas) verifica-se em contraste que as GPUs possuem um elevado número de processadores (normalmente de centenas a milhares), dedicados exclusivamente ao processamento paralelo; tal número de processadores aumenta a cada nova geração de GPUs. Estas diferenças substanciais entre processadores motivaram a transferência de partes computacionalmente intensas do modelo 3D para execução paralela na GPU.

Assim, o trabalho apresentado nesta tese considera o desenvolvimento de algoritmos paralelos que possam tirar proveito da arquitetura da GPU, verificando atentamente o inerente paralelismo do algoritmo de traçamento de raios, assim como o alto volume de processamento no cálculo em 3D do TRACEO3D. A combinação de aprimoramento numérico e paralelização visou alcançar o mais alto desempenho do modelo, exibindo aumentos de desempenho combinados de até 692 vezes superior ao da versão original. Um aspeto importante desta pesquisa é que a validação e avaliação de desempenho foram realizadas não apenas para guias de ondas idealizados, mas também para resultados experimentais coletados em um tanque de testes localizado no *Laboratoire de Mécanique des Fluides et d'Acoustique – Centre National de*

la Recherche Scientifique (LMA-CNRS) laboratório em Marseille. A experiência do tanque decorreu em 2007 com o objetivo de coletar dados de propagação acústica 3D usando um fundo inclinado em um ambiente controlado. A busca de autoraios 3D baseada no método Simplex (e implementada no TRACEO3D) foi validada através de comparações com resultados do modelo 2D TRACEO e com os resultados da experiência no tanque. As previsões do método Simplex exibiram uma semelhança notória com os resultados da experiência, revelando zonas modais de sombra, interferência entre modos, e chegadas múltiplas de modos; neste contexto foram observadas conexões importantes na estrutura de equivalência raio/-modo. Foram detetadas igualmente algumas discrepâncias, que podem estar relacionadas com a falta de conhecimento sobre o sinal emitido e/ou não ter tido em conta deslocamento do feixe nas reflexões no fundo (um efeito que melhora as previsões do modelo quando aplicado na fronteira de sua validade). De modo geral os resultados demonstram que foi alcançado um desempenho notável sem ter comprometido a precisão do modelo. Espera-se que as contribuições apresentadas nesta tese (em particular a redução notável do tempo de execução) tornem atrativa a utilização do TRACEO3D nos problemas de processamento por ajustamento do campo.

As contribuições científicas deste trabalho são:

1. Desenvolvimento de uma solução para o cálculo de autoraios 3D baseada na otimização Simplex. A estratégia de busca baseada no Simplex foi considerada capaz de calcular autoraios 3D de forma precisa e eficiente para um guia de ondas com um fundo penetrável inclinado, gerando previsões de padrões de chegada ao longo do plano, a qual replicou aspetos elaborados de zonas de sombra modais, interferência entre modos e múltiplas chegadas de modos.

2. Desenvolvimento de uma estratégia para o cálculo das influências dos raios baseada em uma grade de recetores, que é atualizada dinamicamente ao longo da trajetória do raio. O método foi considerado computacionalmente eficiente utilizando antenas com um grande número de recetores.

3. Desenvolvimento de algoritmos paralelos para execução em GPU do modelo TRACEO3D, os quais foram validados para busca de autoraios 3D e para o cálculo de influências, exibindo melhorias significativas entre a versão sequencial e a versão paralela do modelo. Pretende-se partilhar o código paralelo para permitir a validação adicional e eventual aplicação do modelo por outros grupos de investigação, assim como para servir de referência de paralelização de um modelo 3D.

Palavras-chave: Acústica submarina, modelagem numérica, feixes Gaussianos, propagação 3D, computação paralela, GPU.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Synopsis: *This chapter presents initial considerations regarding underwater acoustic modeling and three-dimensional propagation, reviews the state of the art and describes the motivation for numerical enhancements and parallel GPU development of the TRACEO3D underwater acoustic model. Section 1.1 briefly outlines 3D propagation, Section 1.2 reviews the state of the art, Section 1.3 presents the motivation and Section 1.4 presents the thesis organization.*

1.1 Three-dimensional propagation

Underwater acoustic models provide a fundamental and efficient tool to parametrically investigate hypothesis and physical phenomena through varied environmental conditions of sound propagation underwater [1]. Such models solve the wave equation, which mathematically describes sound propagation in the ocean, to generate field predictions through the calculation of the pressure field transmitted by a set of acoustic sources, and received on a set of hydrophones [2]. Besides solving the wave equation the model should be capable to handle additional phenomena like, for instance, bottom loss, volume attenuation and/or boundary and volume scattering.

Ocean acoustic models can be classified into different types, depending on the particular

analytical approximation of the wave equation that the model implements numerically. Ray tracing models, for instance, are based on geometrical optics, and address the solution of the wave equation using a high frequency approximation, which leads to the calculation of wavefronts based on ray trajectories. Ray tracing theory has some inherent drawbacks like, for instance, the prediction of perfect shadow zones and caustic singularities. The theory is not well suited for problems of geoacoustic inversion (which require predictions at low frequencies), yet it is ideal if modeling is required for an environment with complex boundaries and/or a complex sound speed distribution [3] (as long as high frequencies are being considered). In this sense, ray theory seems to be an ideal choice for such problems as underwater communications [4] and source tracking [5], for which execution time is a critical factor.

Requirements for model predictions in a three-dimensional ocean waveguide are expected to become more relevant, and thus expected to become more accurate as the amount of available environmental information (water temperature, bottom properties, etc.) grows [1]. A simple approach to provide three-dimensional predictions is to "slice" the waveguide with different transects (i.e. vertical $2D$ planes), and to rely on a two-dimensional model to produce a prediction along the transect (a technique, known as $N \times 2D$ *modeling*). Yet, a three-dimensional boundary (either by itself or combined with a sound speed field) can induce propagation not confined to the $2D$ plane even in the simplest of cases, an effect known as *out-of-plane* propagation [3]. Generally speaking, to calculate properly the acoustic field a three-dimensional propagation model needs to take into account the environmental variability in range, depth and azimuth, as well as possible interferences of this variability in the calculation of propagation. Research in three-dimensional acoustic propagation modeling

is not recent [6–8]. However, the topic was often regarded as being too complex, lacking accurate environmental data, and requiring computational capabilities available only in supercomputers [3]. Nowadays interest in three-dimensional modeling is again receiving attention thanks to the availability of detailed environmental data [9–11], combined with the steady growth of computational power [2]. Research had been conducted regarding 3D propagation effects considering sea mountains [3], submarine canyons [12–14] and other bathymetries with elaborated features [15–17], with most of the 3D predictions obtained using a 3D parabolic equation model. The impact of out-of-plane effect comes mostly from bottom topography but, as discussed in [18] and [19], ocean fronts and wedges can also modify significantly the acoustic field affecting the estimation of source distance and creating shadow coastal zones. Additionally, predictions of sonar performance can take advantage of full 3D modeling to improve accuracy, with a ray model playing a central role in such task due to its capability to handle high frequencies [1, 20]. Furthermore, monitoring of shipping noise represents also an important field of research since shipping noise propagates at long distances, with 3D effects becoming more relevant as distances increase. On the other side, bottom interactions are significant in shallow waters and littoral environments, making 3D effects important in the vicinity of harbors, where ship density is high [21, 22].

Calculation of eigenrays is also an important aspect of 3D predictions. Eigenrays can be defined as particular rays, that for a given waveguide geometry connect the source to the receiver [2]. In two-dimensional waveguides the problem can be solved efficiently using root finder algorithms in one dimension; in this case the problem can be stated as searching for the zeros of a cost function, which depends only on the elevation angles. The extension of such root finder algorithms to find eigenrays in a three-dimensional waveguide is a cumbersome

task, which requires the search to take place on the two-dimensional plane of elevation and azimuth, and would be guided mainly by the minimization of the distance between the final position of the ray and the position of the hydrophone; besides, unlike the one-dimensional search, the search for a minimal value of the cost function on the elevation/azimuth plane can not take place along a particular direction due to the complex regime of propagation, which often needs to account for out-of-plane effects, non-linear internal waves or boundary features [3, 23]. The problem is also computationally demanding, since it often relies on the shooting of a large amount of initial rays [24].

3D predictions and eigenray calculations within the context of the TRACEO3D model will be addressed in detail in Chapter 2. Before the model discussion the state of the art is to be reviewed in the following section.

1.2 State of the art

1.2.1 3D modeling

The 3D Hamiltonian ray-tracing model HARPO was one of the first implementations of 3D propagation based on ray theory [25]; HARPO was able to provide field predictions, ray travel times and field phase within the corresponding limitations of ray-theory. The model was later updated in order to calculate 3D eigenrays using a method which considered the final distance of the ray to the receiver (hereafter called *proximity*) [26]; given two pairs of shooting angles and corresponding proximities one could use linear interpolation to shoot a ray with a smaller proximity; the process was repeated iteratively and the iteration stopped when the proximity was less than 5 m. For the method to be efficient ray trajectories were required to change smoothly over iterations, thus the method was not able to handle non-

linearity due to complex boundary interactions. The discussion presented in [26] considered a speed field typical of the ocean mesoscale, showing only eigenrays (i.e. without results of amplitudes or travel times) and ignoring boundary reflections. Additional results after [26] were not found in the literature, possibly because at that time HARPO was no longer supported by its authors (and so it remains).

A different approach to ray tracing can be found in [27,28], which relies on the *Gaussian beam method* to avoid the generation of shadow zones and infinities of intensity at caustics; the method calculates a pressure field as a sum of beam influences at each receiver. The Gaussian beam method is discussed in detail in [27,29], and was the basis for the development of the 3D models BELLHOP3D [30,31] and TRACEO3D [11,30].

Regarding eigenrays the discussion presented in [32] avoids their direct calculation by considering a dense fan of rays, which can be discretized from the source to a final range of interest over a predefined set of spatial mesh cells. In a given cell the field intensity can be calculated as an average from ray contributions, in proportion to each ray arclength within the cell. Additional computation is needed sorting rays into families in order to compute the coherent ray pressure, and travel time within the cell is then associated to each ray family. To this end a root finding algorithm is needed to determine the position within the cell in which the normal to the ray intersects the position of the receiver; linear interpolation is further used to calculate the travel time between a given cell and the receiver. Results regarding only 2D calculations are shown for a parallel implementation in a high-end computer workstation; the reverberation model MOC3D (renamed later as REV3D) is based on this method [33]. Results presented in [34] suggest that the method is highly time consuming.

An analytic approach to the eigenray problem was proposed in [35], which stated the

calculation of eigenrays as a variational problem. Thus, an initial set of eigenrays calculated for a receiver close to the source can be used to calculate eigenrays for an arbitrary receiver position; caustics could be taken into account by considering a ray amplitude, which was frequency dependent. The numerical implementation of the method for general sound profiles required the introduction of parameterized smoothing functions, and the performance of the method accounting for 3D bathymetries was not considered. A summation method based on the superposition of complex source beams proposed to rely on beam shooting to avoid eigenray calculations [36, 37]; to this end the beams need to be properly collimated through the proper selection of beam parameters for the given geometry of propagation. The discussion was again limited to 2D propagation and did not account for boundary reflections.

A rather different approach for a 3D Gaussian ray model using geodetic coordinates is discussed in [20], looking to calculate transmission loss for sonar training systems. Eigenrays are to be found for each sensor position by considering the closest point of approach (CPA) of the ray to the receiver; then, second order Taylor series can be used to calculate launching angles and travel time corrections by taking into account the CPA. The approach was found to be less efficient than an implementation based on Cartesian coordinates, and very time consuming when considering a large numbers of sensors. The model was used to investigate horizontal refraction although it exhibited a limited success predicting experimental data [38].

As will be shown in Chapters 3 and 5 this thesis will discuss and validate an efficient and robust strategy of 3D eigenray calculations, based on the Simplex method. The approach relies on a small set of parameters (which need to be determined only once) and is able to handle arbitrary 3D waveguide features, such as sound speed distributions and/or bathyme-

tries. The computational strategy of Simplex optimization was designed in order to rely on an efficient selection, within the original region of candidates that encloses a given receiver, such that the search can be accomplished efficiently with either a vertical or a horizontal array. In fact, Simplex optimization guides the ray solution accounting for all environmental influences (even non-linearity induced by the complex regime of ray bouncing on the boundaries), finding take-off angles that allow a given ray to pass near the receiver within a user-defined distance. In this context, the method provides an accurate estimate of ray travel time and amplitude, which is fundamental to predict the channel impulse response.

1.2.2 GPU-based ray tracing

Since its early development, predictions of 3D propagation are well known to be highly time consuming; initial research in this area relied in fact on special computers to carry on model execution [3, 39]. Even today, despite the increasing performance of modern processors, models that take into account 3D propagation still have a high computational cost [2]. This high runtime can easily explain why 3D models are generally put aside to generate replicas for acoustic inversion [9, 34], which is based on matching the acoustic field recorded at an array of sensors with replicas from a numerical model, which are generated for a broad set of parameters [1]; matched field methods are in fact one of the main reasons driving the development of underwater acoustic models. Model performance can be significantly improved through parallel computing. To reduce computing time, improve performance and solve more complex problems, a serial algorithm needs to be rewritten as a parallel algorithm by taking advantage of the underlying parallel hardware. Generally speaking, a serial algorithm is a sequence of steps that solve a given problem using a single processor. In the same way,

a parallel algorithm is a set of steps that solve the same problem, using multiple processors. However, defining the steps is not sufficient. Parallelization also implies adding extensions to the algorithm, specifying sets of steps that can be performed simultaneously [40]. Additionally, the parallel code may require dealing with the synchronization of processors at the various stages of program execution, or managing accesses to data shared by multiple processors. Often, different choices yield the best performance on different parallel architectures or under different parallel programming paradigms.

The microprocessor industry has followed the many-core direction to improve performance due to the designs limitation, by boosting clock speed. Although *central processing units* (hereafter CPU) can be found with a few to dozens of cores, *graphic processing units* (hereafter GPU) have a larger number of cores (usually from hundreds to thousands), devoted to parallel processing; such number of cores increases at each GPU generation. The design differences between CPUs and GPUs resulted in a large performance gap between parallel and sequential program execution, which motivated the transfer of computationally intensive parts of a code to the parallel execution on a GPU [41]. This capability motivated several implementations of scientific applications, including underwater acoustic models. For instance, a split-step Fourier parabolic equation model implemented in a GPU is discussed in [42]. The work considered only high idealized waveguides and the results showed a significant improvement, with the parallel version of the code being 20-35 times faster than the sequential one. As further indicated in the reference GPU computing has a potential to enable interesting new approaches to 3D modeling. The discussion presented in [43] describes a GPU-based version of a Beam-Displacement Ray-Mode code; although it considers only 2D propagation in idealized waveguides the parallel model was found to be 30 times faster

than the sequential one. A parallel implementation of BELLHOP (not to be confused with BELLHOP3D) addressed to a GPU architecture was presented in [44]; the parallel model had the capability to calculate only ray trajectories and amplitudes. The discussion presents only runtime results, with the performance being increased for a high numbers of rays. However, pressure computation was kept in the CPU since the runtime of the parallel version was worse than the one of the sequential version due to memory transfers. A parallel version of the C-based version of TRACEO (called cTRACEO), based on a GPU architecture, was discussed in detail in [45]; the discussion showed that parallelization drastically reduces the computational burden when a large number of rays needs to be traced. Such parallel version of cTRACEO was able to calculate travel times, amplitudes, eigenrays and pressure. Performance results for such 2D model indicated a promising advantage addressing the GPU architecture for the 3D case.

The present work considered the development of parallel algorithms with the recent tools available to take advantage of the GPU architecture, looking carefully to the inherent parallelism of ray tracing and the high workload of predictions for 3D propagation. The results, to be presented in Chapter 5, will demonstrate through comparisons based on simulations and experimental results that a remarkable performance was achieved without compromising accuracy.

1.3 Motivation of this work

An important component of the work developed in this thesis was the availability of the TRACEO3D ray tracing model, which is described in detail in Chapter 2. Within this context the thesis was motivated by the interest in providing accurate predictions, that can

take into account out-of-plane effects in high frequency based models; one therefore needs to address the development of a robust method for 3D eigenrays search, which is fundamental for the calculation of a channel impulse response; additionally, it was addressed also the issue of long runtime, which often hampers the usage of 3D models. It was explored the possibility for numerical enhancements and optimization regarding calculations in the sequential version of TRACEO3D, leading to improvements in performance and accuracy; conclusion of such exploration was to be followed by a careful analysis of the GPU hardware multithread, coding the sequential model structure into a parallel algorithm. The combination of numerical enhancement and parallelization aimed to achieve the highest performance of TRACEO3D. Finally, an important aspect of this research is that validation and performance assessment were carried out not only for idealized waveguides, but also for the experimental results of the tank scale experiment described in [9].

The objectives of this thesis can then be summarized as follows:

- To investigate and develop a search method to calculate 3D eigenrays for channel impulse response predictions, and to optimize the method for pressure field calculation that account for horizontal effects using a large numbers of sensors. In both cases the enhancement should allow the sequential code to be computationally efficient and accurate.

- To develop parallel algorithms that take advantage of the GPU architecture, and restructure the memory access pattern to improve performance.

- To validate the model through comparisons between original and enhanced versions (prior to parallelization), and between the sequential and parallel versions, not only in

terms of performance, but also in terms of accuracy.

1.4 Thesis organization

This thesis is organized as follows: Chapter 2 describes the theoretical and numerical formalisms in which the TRACEO3D Gaussian beam model is based. Enhancements are presented in Chapter 3, describing the strategies to calculate 3D eigenrays and optimizing the calculations of ray influence. The detailed structure of the parallel GPU implementation is presented in Chapter 4, providing a brief introduction to the GPU architecture and CUDA programing. Chapter 5 presents the validation results, in which simulations and experimental data are considered. Conclusions and future work are in Chapter 6, presenting the contributions and indicating future directions of research. The appendix explains how to compile the model, the structure of the input file, and an example of 3D predictions using the sequential and the parallel versions of the model.

Chapter 2

The TRACEO3D Gaussian beam model

Synopsis: *This chapter describes the TRACEO3D Gaussian beam model and it is organized as follows: Section 2.1 provides a compact description of the model, Section 2.2 describes the theory behind TRACEO3D calculations, and Section 2.3 discusses important numerical issues.*

2.1 General description

The TRACEO3D Gaussian beam model represents the three-dimensional extension of the TRACEO Gaussian beam model [46]. The first version of TRACEO3D was written by Orlando Camargo Rodríguez of the Signal Processing Laboratory (SiPLAB), but current authorship has been extended to the thesis author given the relevance of his contributions to the model. TRACEO3D was developed in order to provide different types of predictions, namely:

- ray trajectories;
- ray travel time and amplitude along the trajectory;
- eigenrays (i.e. rays connecting a source to a receiver);

- predictions of acoustic pressure for different array configurations (horizontal, vertical, linear, planar);
- particle velocity calculations (of relevance for the development of Vertical Sensor Arrays).

Additionally, TRACEO3D can account for a rather large class of waveguide features, such as

- water sound speed profiles and sound speed fields;
- non-flat boundaries (wavy surfaces, bottom wedges, sea mountains, canyons, etc.);
- spatial variability of boundary properties (density, attenuation, sound speed).

However, TRACEO's capability to consider ray bouncing on underwater objects located between the surface and the bottom is still not implemented in TRACEO3D due to the complexity of defining 3D meshes, and the associated problem of ray/mesh intersection and mesh interpolation. Important theoretical and numerical issues of the model are discussed in the following sections.

2.2 Theoretical background

By order of importance the theoretical aspects of the model can be organized into the following items:

- calculation of ray trajectories;
- calculation of amplitude parameters;

- calculation of beam influence;
- calculation of particle velocity;
- calculation of amplitude corrections due to ray-boundary reflections and volume attenuation.

Each item will be described in detail in the following sections.

2.2.1 The Eikonal equations

The starting point for the calculation of three-dimensional ray trajectories is given by the solution of the *Eikonal equations*, which can be written in different ways [2, 29, 47]; in the TRACEO3D model they correspond to

$$\begin{aligned} \frac{dx}{ds} &= c(s)\sigma_x \quad , \quad \frac{dy}{ds} = c(s)\sigma_y \quad , \quad \frac{dz}{ds} = c(s)\sigma_z \quad , \\ \frac{d\sigma_x}{ds} &= -\frac{1}{c^2}\frac{\partial c}{\partial x} \quad , \quad \frac{d\sigma_y}{ds} = -\frac{1}{c^2}\frac{\partial c}{\partial y} \quad , \quad \frac{d\sigma_z}{ds} = -\frac{1}{c^2}\frac{\partial c}{\partial z} \quad , \end{aligned} \tag{2.1}$$

where $c(s)$ represents the sound speed along the ray, σ_x, σ_y and σ_z stand for the components of the vector of sound slowness, and s stands for the ray arclength. The derivatives dx/ds, dy/ds and dz/ds define the unitary vector $\mathbf{e}_s$, which is tangent to the ray; the plane perpendicular to $\mathbf{e}_s$ defines the plane normal to the ray. On this plane one can introduce a pair of unitary and orthogonal vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ (known as the *polarization vectors* [47], see Fig.2.1), which define a ray normal $\mathbf{n}$ as:

$$\mathbf{n} = n_1\mathbf{e}_1 + n_2\mathbf{e}_2 \ , \tag{2.2}$$

where n_1 and n_2 represent the normal components in the ray-centered system of coordinates.

The integration of Eq.(2.1) requires the knowledge of the source position $(x(0), y(0), z(0))$

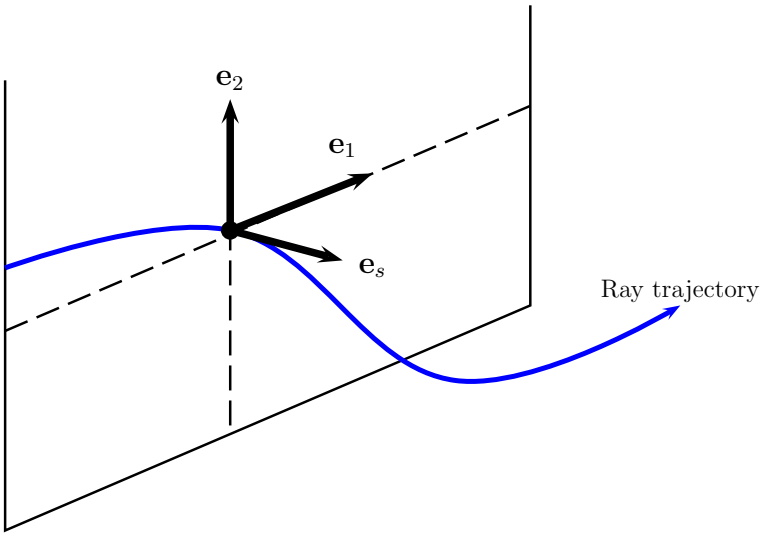


Figure 2.1: Ray tangent $\mathbf{e}_s$, and polarization vectors $\mathbf{e}_1$ and $\mathbf{e}_2$.

and of the initial direction of propagation, which is given by $\mathbf{e}_s(0)$ and can be written as

$$\mathbf{e}_s(0) = \begin{bmatrix} \cos\theta(0)\cos\phi(0) \\ \cos\theta(0)\sin\phi(0) \\ \sin\theta(0) \end{bmatrix} \tag{2.3}$$

where $\theta(s)$ stands for the ray *elevation* (i.e. the ray slope relative to the plane XY) and $\phi(s)$ stands for the ray *azimuth* (i.e. the slope of the ray projection on the XY plane relative to the X axis; see Fig.2.2). The travel time $\tau(s)$ is further obtained after integration of $ds/c(s)$ along the ray trajectory.

2.2.2 The dynamic equations

Besides ray trajectories TRACEO3D relies on the Gaussian beam approximation to compute the ray amplitude [48]. To this end one needs to calculate a set of 2×2 matrices represented generally as $\mathbf{C}$, $\mathbf{M}$ and $\mathbf{P}$; the system is given by the following relationships [47]:

$$\mathbf{M} = \mathbf{P}\mathbf{Q}^{-1} \ , \tag{2.4}$$

and

$$\frac{d}{ds}\mathbf{Q} = c(s)\mathbf{P} \quad , \quad \frac{d}{ds}\mathbf{P} = -\frac{1}{c^2(s)}\mathbf{C}\mathbf{Q} \quad , \tag{2.5}$$

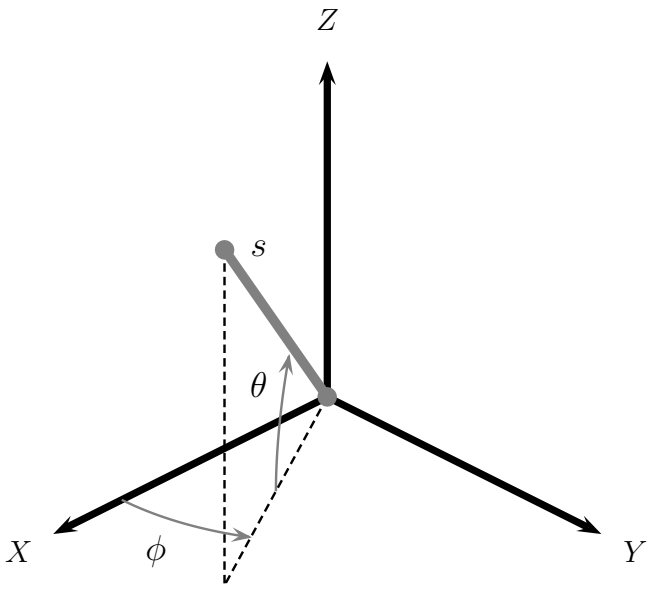


Figure 2.2: Ray elevation θ and azimuth ϕ.

where

$$C_{ij} = \frac{\partial^2 c}{\partial n_i \partial n_j} \; ; \tag{2.6}$$

the elements of $\mathbf{C}$ correspond to second order derivatives of sound speed along the polarization vectors $\mathbf{e}_1$ and $\mathbf{e}_2$. Generally speaking $\mathbf{P}$ describes the beam slowness in the plane perpendicular to $\mathbf{e}_s$, while $\mathbf{Q}$ describes the beam spreading. The pair of expressions given by Eq.(2.5) is called the *dynamic equations* of the Gaussian beam formulation.

Generally speaking the polarization vectors are related to $\mathbf{e}_s$ through the ray torsion and curvature, which can be cumbersome to determine numerically; a simplified approach, valid for a sound speed profile, or for a sound speed field with cylindrical symmetry, is to calculate both vectors using the relationships

$$\mathbf{e}_1(s) = \begin{bmatrix} -\sin\theta(s)\cos\phi(s) \\ -\sin\theta(s)\sin\phi(s) \\ \cos\theta(s) \end{bmatrix} \quad \text{and} \quad \mathbf{e}_2(s) = \begin{bmatrix} -\sin\phi(s) \\ \cos\phi(s) \\ 0 \end{bmatrix} . \tag{2.7}$$

The update of matrices $\mathbf{P}$ and $\mathbf{Q}$ after a boundary reflection is discussed in detail in [29].

2.2.3 Beam influence

The solution of the Eikonal and dynamic equations allows to calculate the beam influence along a given normal based on the expression [48]

$$P(s,\mathbf{n}) = \frac{1}{4\pi}\sqrt{\frac{c(s)}{c(0)}\frac{\cos\theta(0)}{\det\mathbf{Q}}}\exp\left\{-i\omega\left[\tau(s)+\frac{1}{2}\left(\mathbf{Mn}\cdot\mathbf{n}\right)\right]\right\}\ , \tag{2.8}$$

where $\cdot$ represents an inner vector product; generally speaking the imaginary part of the product $\mathbf{Mn}\cdot\mathbf{n}$ induces a Gaussian decay of beam amplitude along $\mathbf{n}$, while the real part introduces phase corrections to the travel time. As long as $\det\mathbf{Q} \neq 0$ the solution given by Eq.(2.8) is free of the singularities of the classic solution (based on ray tubes); phase corrections due to caustics can be also easily included. The expression given by Eq.(2.8) behaves near the source as an spherical wave emitted by a point source, through the choice of initial conditions [47]

$$\mathbf{P}(0) = \begin{bmatrix} 1 & 0 \\ 0 & \cos\theta(0) \end{bmatrix}/c(0) \tag{2.9}$$

and [48]

$$\mathbf{Q}(0) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}\ . \tag{2.10}$$

2.2.4 Calculation of particle velocity

Calculation of particle velocity requires the gradient of the pressure field, which can be written in ray coordinates as

$$\nabla P = \frac{\partial P}{\partial s}\,\mathbf{e}_s + \frac{\partial P}{\partial n_1}\,\mathbf{e}_1 + \frac{\partial P}{\partial n_2}\,\mathbf{e}_2\ . \tag{2.11}$$

Partial derivatives in the Cartesian coordinates can be obtained through the expressions

$$\frac{\partial P}{\partial x} = \nabla P\cdot\mathbf{e}_x\ ,\ \frac{\partial P}{\partial y} = \nabla P\cdot\mathbf{e}_y\ ,\ \frac{\partial P}{\partial z} = \nabla P\cdot\mathbf{e}_z\ , \tag{2.12}$$

where $\mathbf{e}_x$, $\mathbf{e}_y$ and $\mathbf{e}_z$ stand for the unitary vectors along the X, Y and Z axes, respectively.

2.2.5 Boundary reflections and volume attenuation

After each boundary reflection the amplitude needs to be multiplied by a decaying factor ϕ_r, which is given by the expression

$$\phi_r = \prod_{i=1}^{n_r} R_i \ , \tag{2.13}$$

where n_r represents the total number of boundary reflections, and R_i is the reflection coefficient at the ith reflection. The case with no reflections ($n_r = 0$) corresponds to $\phi_r = 1$. Generally speaking, boundaries can be one of four types:

- Absorbent: the wave energy is transmitted completely to the medium above the boundary, so $R = 0$ and ray propagation ends at the boundary.
- Rigid: the wave energy is reflected completely on the boundary, with no phase change, so $R = 1$.
- Vacuum: the wave energy is reflected completely on the boundary, with a phase change of π radians, so $R =$ -1.
- Elastic: the wave energy is partially reflected, with R being a complex value and $|R| < 1$.

The calculation of the reflection coefficient for an elastic medium requires the knowledge of the following (often depth-dependent) parameters:

- compressional wave speed c_p,
- shear wave speed c_s,

- compressional wave attenuation α_{cp},
- shear wave attenuation α_{cs},
- density ρ,

(see Fig. 2.3) and it is based on the following expression [49]:

$$R(\theta) = \frac{D(\theta)\cos\theta - 1}{D(\theta)\cos\theta + 1}\ , \tag{2.14}$$

where

$$D(\theta) = A_1\left(A_2\frac{1-A_7}{\sqrt{1-A_6^2}} + A_3\frac{A_7}{\sqrt{1-A_5/2}}\right)\ ,$$

$$A_1 = \frac{\rho_2}{\rho_1}\quad ,\quad A_2 = \frac{\tilde{c}_{p2}}{c_{p1}}\quad ,\quad A_3 = \frac{\tilde{c}_{s2}}{c_{p1}}\quad ,$$

$$A_4 = A_3\sin\theta\quad ,\quad A_5 = 2A_4^2\quad ,\quad A_6 = A_2\sin\theta\quad ,\quad A_7 = 2A_5 - A_5^2\quad ,$$

$$\tilde{c}_{p2} = c_{p2}\frac{1-i\tilde{\alpha}_{cp}}{1+\tilde{\alpha}_{cp}^2}\quad ,\quad \tilde{c}_{s2} = c_{s2}\frac{1-i\tilde{\alpha}_{cs}}{1+\tilde{\alpha}_{cs}^2}\quad ,$$

$$\tilde{\alpha}_{cp} = \frac{\alpha_{cp}}{40\pi\log e}\quad ,\quad \tilde{\alpha}_{cs} = \frac{\alpha_{cs}}{40\pi\log e}\quad ,$$

where the units of attenuation should be given in dB/λ.

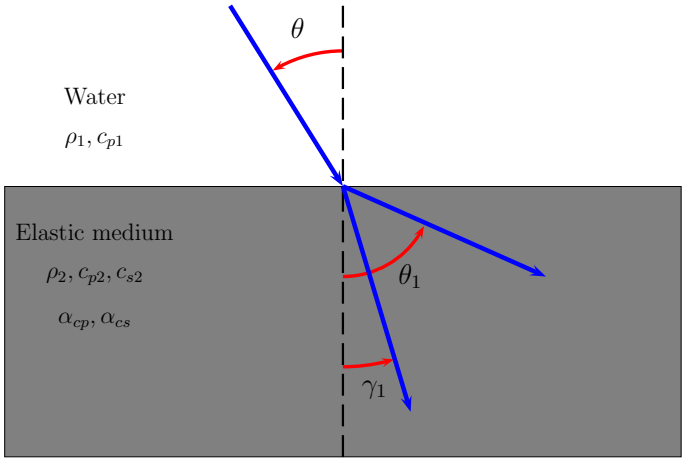


Figure 2.3: Ray reflection on an elastic medium.

In general the reflection coefficient is real when $\alpha_{cp} = \alpha_{cs} = 0$, and the angle of incidence θ is less than the critical angle θ_{cr}, with θ_{cr} given by the expression

$$\theta_{cr} = \arcsin\left(\frac{c_{p1}}{c_{p2}}\right) \ . \tag{2.15}$$

Moreover, attenuation is negligible when $\theta < \theta_{cr}$, and for small θ the energy transferred to shear waves in the elastic medium is only a small fraction of the total energy transfered.

Ray amplitude needs to be corrected also along a ray trajectory with a factor ϕ_V to account for volume attenuation, which in the ocean has a chemical nature, and it is induced by relaxation processes of salt constituents like $MgSO_4$, $B(OH)_3$ and $MgCO_3$. The factor ϕ_V is given by the decaying exponential

$$\phi_V = \exp\left(-\alpha_T s\right) \ , \tag{2.16}$$

where s is the ray arclength and α_T is the Thorpe (frequency dependent) attenuation coefficient in dB/m, given by [2]

$$\alpha_T = \frac{40 f^2}{4100 + f^2} + \frac{0.1 f^2}{1 + f^2} \ , \tag{2.17}$$

with the frequency given in kHz.

2.3 Numerical issues

2.3.1 Solving the Eikonal

In order to solve numerically the Eikonal one can rewrite Eq.(2.1) as a linear differential vector equation:

$$\frac{d\mathbf{y}}{ds} = \mathbf{f} \ , \tag{2.18}$$

where

$$\mathbf{y} = \begin{bmatrix} x \\ y \\ z \\ \sigma_x \\ \sigma_y \\ \sigma_z \end{bmatrix} \quad \text{and} \quad \mathbf{f} = \begin{bmatrix} \sigma_x/\sigma \\ \sigma_y/\sigma \\ \sigma_z/\sigma \\ \partial\sigma/\partial x \\ \partial\sigma/\partial y \\ \partial\sigma/\partial z \end{bmatrix} ; \tag{2.19}$$

this system can be integrated using a Runge-Kutta-Fehlberg method, which provides two solutions at each step of integration; if the solutions differ in more than a particular threshold the ray step ds is halved, and the integration is restarted with the new step; this process is repeated until the difference falls below the threshold. Particular care was taken to avoid infinite loops by setting a maximal number of repetitions.

2.3.2 Solving the dynamic equations

The set given by Eq.(2.5) can be solved using Euler's method, providing that the ray trajectories are calculated accurately.

2.3.3 Calculation of derivatives along the polarization vectors

Elements of the matrix $\mathbf{C}$ in Eq.(2.6) need to be calculated as derivatives of sound speed $c(x, y, z)$ along the polarization vectors $\mathbf{e}_1$ and $\mathbf{e}_2$. Such derivatives can be calculated explicitly in Cartesian coordinates using the expressions

$$\frac{\partial}{\partial n_1} = \left(\frac{\partial x}{\partial n_1}\right)\frac{\partial}{\partial x} + \left(\frac{\partial y}{\partial n_1}\right)\frac{\partial}{\partial y} + \left(\frac{\partial z}{\partial n_1}\right)\frac{\partial}{\partial z}$$

and

$$\frac{\partial}{\partial n_2} = \left(\frac{\partial x}{\partial n_2}\right)\frac{\partial}{\partial x} + \left(\frac{\partial y}{\partial n_2}\right)\frac{\partial}{\partial y} + \left(\frac{\partial z}{\partial n_2}\right)\frac{\partial}{\partial z} .$$

Second-order derivatives follow directly from the above expressions; for instance:

$$\left(\frac{\partial}{\partial n_1}\right)\left(\frac{\partial}{\partial n_2}\right) =$$

$$= \left(\frac{\partial x}{\partial n_1}\right)\left(\frac{\partial x}{\partial n_2}\right)\frac{\partial^2}{\partial x^2} + \left(\frac{\partial x}{\partial n_1}\right)\left(\frac{\partial y}{\partial n_2}\right)\frac{\partial^2}{\partial x \partial y} + \left(\frac{\partial x}{\partial n_1}\right)\left(\frac{\partial z}{\partial n_2}\right)\frac{\partial^2}{\partial x \partial z}$$
$$+ \left(\frac{\partial y}{\partial n_1}\right)\left(\frac{\partial x}{\partial n_2}\right)\frac{\partial^2}{\partial y \partial x} + \left(\frac{\partial y}{\partial n_1}\right)\left(\frac{\partial y}{\partial n_2}\right)\frac{\partial^2}{\partial y^2} + \left(\frac{\partial y}{\partial n_1}\right)\left(\frac{\partial z}{\partial n_2}\right)\frac{\partial^2}{\partial y \partial z}$$
$$+ \left(\frac{\partial z}{\partial n_1}\right)\left(\frac{\partial x}{\partial n_2}\right)\frac{\partial^2}{\partial z \partial x} + \left(\frac{\partial z}{\partial n_1}\right)\left(\frac{\partial y}{\partial n_2}\right)\frac{\partial^2}{\partial z \partial y} + \left(\frac{\partial z}{\partial n_1}\right)\left(\frac{\partial z}{\partial n_2}\right)\frac{\partial^2}{\partial z^2} \ .$$

For the choice of polarization vectors discussed in Section 2.2.2 it can be found that

$$\frac{\partial x}{\partial n_1} = -\sin\theta\cos\phi \quad , \quad \frac{\partial y}{\partial n_1} = -\sin\theta\sin\phi \quad , \quad \frac{\partial z}{\partial n_1} = \cos\theta \quad ,$$

and

$$\frac{\partial x}{\partial n_2} = -\sin\phi \quad , \quad \frac{\partial y}{\partial n_2} = \cos\phi \quad , \quad \frac{\partial z}{\partial n_2} = 0 \quad .$$

2.3.4 Beam influence

For a given normal $\mathbf{n}$ the calculation of beam influence using Eq.(2.8) requires the calculation of matrix $\mathbf{M}$; however, during the development of the TRACEO3D model it was found an alternative (and equally accurate) expression of beam influence, given by

$$P(s, n_1, n_2) = \frac{1}{4\pi}\sqrt{\frac{c(s)}{c(0)}\frac{\cos\theta(0)}{\det\mathbf{Q}}}\Phi_{11}\Phi_{12}\Phi_{21}\Phi_{22}\exp\left[-i\omega\tau(s)\right] \ , \tag{2.20}$$

where the coefficients Φ_{ij} are given by

$$\Phi_{ij} = \exp\left[-\left(\frac{\sqrt{\pi|n_i n_j|}}{\Delta\theta}Q_{ij}^{-1}\right)^2\right] \ , \tag{2.21}$$

with $\Delta\theta$ standing for the elevation step between successive rays, and Q_{ij}^{-1} representing the elements of $\mathbf{Q}^{-1}$; n_1 and n_2 are calculated through the projection of $\mathbf{n}$ onto $\mathbf{e}_1$ and $\mathbf{e}_2$. Calculations with Eq.(2.20) are faster than with Eq.(2.8) because the step of matrix multiplication between $\mathbf{P}$ and $\mathbf{Q}^{-1}$ is not required.

2.3.5 Calculation of normals

In the original version of TRACEO3D ray influence at a receiver located at the position $\mathbf{r}_h$ was calculated using the following procedure:

- Divide the ray trajectory into segments between successive transitions (surface/bottom reflection, or bottom/surface reflection, etc.);
- Proceed along *all* segments to find *all* ray normals to the receiver; to this end:
  - Consider the ith segment; let $\mathbf{r}_A$ and $\mathbf{r}_B$ be the coordinates of the beginning and end of the segment, respectively, and let $\mathbf{e}_A$ and $\mathbf{e}_B$ be the vectors corresponding to $\mathbf{e}_s$ at A and B.
  - Calculate the vectors $\Delta\mathbf{r}_A = \mathbf{r}_h - \mathbf{r}_A$ and $\Delta\mathbf{r}_B = \mathbf{r}_h - \mathbf{r}_B$.
  - Calculate the inner products $P_A = \mathbf{e}_A \cdot \Delta\mathbf{r}_A$ and $P_B = \mathbf{e}_B \cdot \Delta\mathbf{r}_B$.
  - If $P_A \times P_B < 0$ a normal exists and it can be found through bisection along the segment; once the normal is found the corresponding influence at the receiver can be calculated.
  - If $P_A \times P_B > 0$ there is no normal (and no influence at the receiver); therefore, one can move to segment $i + 1$.
- The ray influence at the receiver is the sum of influences from all segments.

The influence of a Gaussian beam decays rapidly along a normal, but it never reaches zero; therefore, the procedure is to be repeated for *all* rays and *all* receivers.

As shown in [11] field predictions using this method exhibit a good agreement with experimental data, but the runtime is often high and increases drastically as range, number

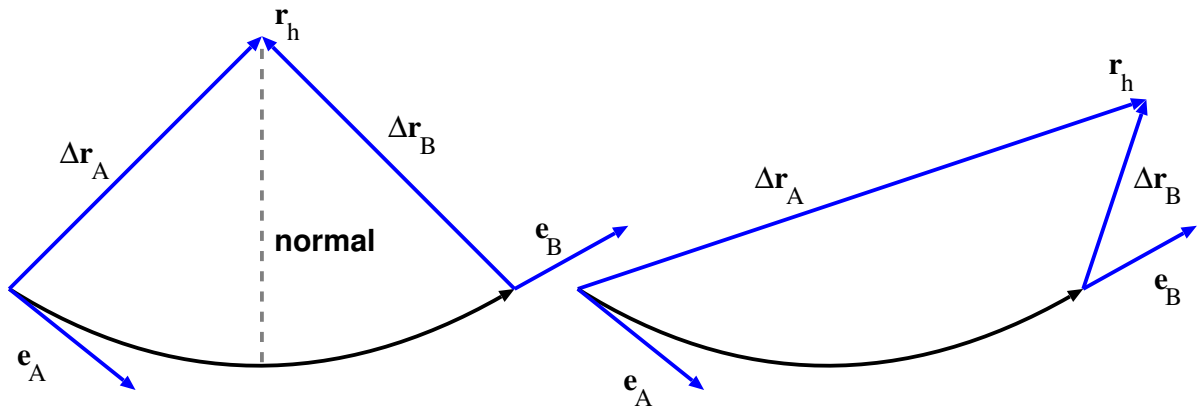


Figure 2.4: Normal search along a ray segment. *Left:* the hydrophone is at a position for which $P_A \times P_B < 0$; thus a normal exists, and it can be found by bisection somewhere along the segment. *Right:* the hydrophone is at a position for which $P_A \times P_B > 0$; thus, there is no normal and the ray segment has no influence at the hydrophone position.

of rays and number of sensor increases. The numerical enhancement of normal calculations is described in detail in Section 3.2.

2.3.6 Interpolation and calculation of derivatives

N dimensional piecewise n-point barycentric polynomials are used for interpolation and calculation of derivatives [46]. For each space dimension n points of tabulated data are used to calculate relative distances between the points and build a polynomial of order $n-1$. Interpolation and calculation of derivatives is then performed at a new point using the polynomial coefficients and polynomial derivatives. This strategy of interpolation can be used with a uniform or non-uniform grid of data points, is numerically stable, robust, and easy to implement for any number of dimensions. The following discussion illustrates the interpolation of curves, surfaces and volumes using a parabolic ($n = 3$) barycentric interpolation[1].

[1]Generally speaking parabolic interpolation is not used very often, because it can lead to unbalanced estimates of function values depending on the position of the new point relative to the tabulated ones. Yet the expressions of the polynomials are ideal to illustrate the method.

- **Preliminary definitions:** in order to provide a compact description of barycentric parabolic interpolation the following notation will be introduced:

$$P_j^n(x) = \prod_{i=1, i\neq j}^{n} (x - x_i) \ ; \tag{2.22}$$

for instance:

$$P_1^2(x) = (x - x_2) \quad , \quad P_2^2(x) = (x - x_1) \quad ,$$

$$P_1^3(x) = (x - x_2)(x - x_3) \ , \ P_2^3(x) = (x - x_1)(x - x_3) \ , \ P_3^3(x) = (x - x_1)(x - x_2) \ ,$$

$$P_1^4(x) = (x - x_2)(x - x_3)(x - x_4) \ , \ P_2^4(x) = (x - x_1)(x - x_3)(x - x_4) \ ,$$

$$P_3^4(x) = (x - x_1)(x - x_2)(x - x_4) \ , \ P_4^4(x) = (x - x_1)(x - x_2)(x - x_3) \ \ldots$$

Additionally, let be

$$S_j(x) = 2x - \sum_{i=1, i\neq j}^{3} x_i \ ; \tag{2.23}$$

for instance

$$S_1(x) = (2x - x_2 - x_3) \ , \ S_2(x) = (2x - x_1 - x_3) \ , \ S_3(x) = (2x - x_1 - x_2) \ .$$

- **Line interpolation:** Consider a set of three points x_1, x_2 and x_3 and a set of function values $f(x_1)$, $f(x_2)$ and $f(x_3)$. It is required to interpolate the function and its first and second derivatives at a point x, located between x_1 and x_3 (see Fig. 2.5).

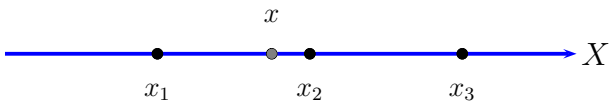


Figure 2.5: One-dimensional grid considered for piecewise barycentric parabolic interpolation.

The barycentric parabolic polynomial can be written as

$$f(x) = f(x_1) + a_2 P_2^3(x) + a_3 P_3^3(x) \ , \tag{2.24}$$

where

$$a_2 = \frac{f(x_2) - f(x_1)}{P_2^3(x_2)} \quad \text{and} \quad a_3 = \frac{f(x_3) - f(x_1)}{P_3^3(x_3)} \ .$$

The condition $P_i^3(x_j) = 0$ when $j \neq i$ implies automatically that the polynomial provides the function values at the grid points. The expressions for the derivatives become:

$$\frac{df}{dx} = a_2 S_2(x) + a_3 S_3(x) \quad \text{and} \quad \frac{d^2 f}{dx^2} = 2\,(a_2 + a_3) \ .$$

- **Surface interpolation:** consider a two-dimensional grid of points (x_1, y_1), (x_2, y_1), ..., (x_3, y_3), with function values $f(x_1, y_1)$, $f(x_2, y_1)$, ..., $f(x_3, y_3)$. It is required to interpolate the function and its first and second partial derivatives at a point (x, y) located inside the grid (see Fig. 2.6). The biparabolic barycentric polynomial can be written as

$$\begin{array}{rcccccc} f(x,y) & = & f(x_1,y_1) & +a_{12}P_2^3(x)P_1^3(y) & +a_{13}P_3^3(x)P_1^3(y) & + \\ & + & a_{21}P_1^3(x)P_2^3(y) & +a_{22}P_2^3(x)P_2^3(y) & +a_{23}P_3^3(x)P_2^3(y) & + \\ & + & a_{31}P_1^3(x)P_3^3(y) & +a_{32}P_2^3(x)P_3^3(y) & +a_{33}P_3^3(x)P_3^3(y) & , \end{array} \tag{2.25}$$

where the general expression for the coefficient a_{ij} given by

$$a_{ij} = \frac{f(x_j, y_i) - f(x_1, y_1)}{P_j^3(x_j) P_i^3(y_i)}$$

with $i, j = 1, 2, 3$ and $a_{11} = 0$; for instance

$$a_{12} = \frac{f(x_2, y_1) - f(x_1, y_1)}{P_2^3(x_2) P_1^3(y_1)} \quad , \quad a_{13} = \frac{f(x_3, y_1) - f(x_1, y_1)}{P_3^3(x_3) P_1^3(y_1)} \quad , \quad \ldots$$

Expressions for partial derivatives can be written as

$$\frac{\partial f}{\partial x} = \sum_{i=1}^{3} \sum_{j=1}^{3} a_{ij} P_i^3(y) S_j(x) \quad , \quad \frac{\partial f}{\partial y} = \sum_{i=1}^{3} \sum_{j=1}^{3} a_{ij} P_i^3(x) S_j(y) \quad ,$$

$$\frac{\partial^2 f}{\partial x^2} = 2 \sum_{i=1}^{3} \sum_{j=1}^{3} a_{ij} P_i^3(y) \quad , \quad \frac{\partial^2 f}{\partial y^2} = 2 \sum_{i=1}^{3} \sum_{j=1}^{3} a_{ij} P_i^3(x) \quad ,$$

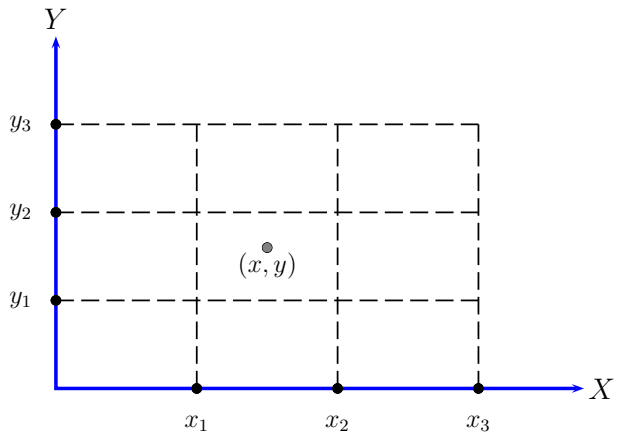


Figure 2.6: Two-dimensional grid considered for piecewise barycentric biparabolic interpolation.

and

$$\frac{\partial^2 f}{\partial x \partial y} = \sum_{i=1}^{3}\sum_{j=1}^{3} a_{ij} S_i(x) S_j(y) \ .$$

- **Volume interpolation:** consider a three-dimensional grid of points (x_1, y_1, z_1), (x_2, y_1, z_1), (x_3, y_1, z_1), ..., (x_3, y_3, z_3), with function values $f(x_1, y_1, z_1)$, $f(x_2, y_1, z_1)$, $f(x_3, y_1, z_1)$, ..., $f(x_3, y_3, z_3)$. It is required to interpolate the function and its first and second partial derivatives at a point (x, y, z) located inside the grid (see Fig. 2.7). The triparabolic barycentric polynomial can be written as

$$f(x, y, z) = \sum_{i=1}^{3}\sum_{j=1}^{3}\sum_{k=1}^{3} a_{ijk} P_k^3(x) P_j^3(y) P_i^3(z) \ , \tag{2.26}$$

where the coefficients a_{ijk} are given by the expression

$$a_{ijk} = \frac{f(x_k, y_j, z_i) - f(x_1, y_1, z_1)}{P_k^3(x_k) P_j^3(y_j) P_i^3(z_i)}$$

with $i, j, k = 1, 2, 3$ and $a_{111} = 0$.

Expressions for partial derivatives can be written as

$$\frac{\partial f}{\partial x} = \sum_{i=1}^{3}\sum_{j=1}^{3}\sum_{k=1}^{3} a_{ijk} S_k(x) P_j^3(y) P_i^3(z) \ ,$$

$$\frac{\partial f}{\partial y} = \sum_{i=1}^{3}\sum_{j=1}^{3}\sum_{k=1}^{3} a_{ijk} P_k^3(x) S_j(y) P_i^3(z) \ ,$$

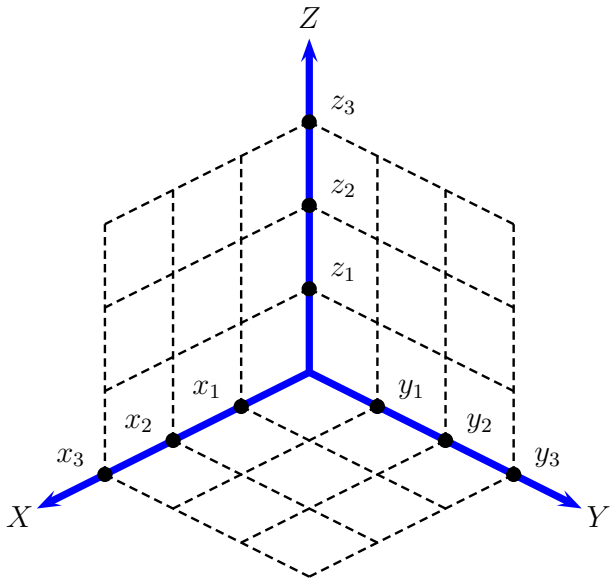


Figure 2.7: Three-dimensional grid considered for piecewise barycentric triparabolic interpolation.

$$\frac{\partial f}{\partial z} = \sum_{i=1}^{3}\sum_{j=1}^{3}\sum_{k=1}^{3} a_{ijk} P_k^3(x) P_j^3(y) S_i(z) \ ,$$

$$\frac{\partial^2 f}{\partial x^2} = 2\sum_{i=1}^{3}\sum_{j=1}^{3}\sum_{k=1}^{3} a_{ijk} P_i^3(y) P_i^3(z) \ ,$$

$$\frac{\partial^2 f}{\partial y^2} = 2\sum_{i=1}^{3}\sum_{j=1}^{3}\sum_{k=1}^{3} a_{ijk} P_k^3(x) P_i^3(z) \ ,$$

$$\frac{\partial^2 f}{\partial z^2} = 2\sum_{i=1}^{3}\sum_{j=1}^{3}\sum_{k=1}^{3} a_{ijk} P_k^3(x) P_j^3(y) \ ,$$

$$\frac{\partial^2 f}{\partial x \partial y} = \sum_{i=1}^{3}\sum_{j=1}^{3} a_{ijk} S_k(x) S_j(y) P_i^3(z) \ ,$$

$$\frac{\partial^2 f}{\partial x \partial z} = \sum_{i=1}^{3}\sum_{j=1}^{3} a_{ijk} S_i(x) P_j^3(y) S_i(z) \ ,$$

$$\frac{\partial^2 f}{\partial y \partial z} = \sum_{i=1}^{3}\sum_{j=1}^{3} a_{ijk} P_k^3(x) S_j(y) S_i(z) \ .$$

2.3.7 Ray/boundary intersection

In TRACEO3D the numerical integration of the Eikonal is accomplished in parallel testing the intersection of a ray segment with any of the waveguide boundaries, which in general can

be expected to exhibit any degree of roughness. When the end of the ray segment is found to be above the surface or below the bottom the point of intersection (x_i, y_i, z_i) is determined. Using the interpolation polynomials one can find the normal to the boundary $f(x, y)$ at the intersection point, which can be written as

$$\mathbf{n} = \mathbf{N} / |\mathbf{N}|$$

where

$$\mathbf{N} = \begin{bmatrix} -\partial f/\partial x \\ -\partial f/\partial y \\ 1 \end{bmatrix} ;$$

the partial derivatives are to be calculated at (x_i, y_i, z_i). The vector $\mathbf{n}$ is required to calculate the reflection coefficient at the point of intersection; besides, $\mathbf{n}$ is also needed to determine the new direction of propagation after reflection. In fact, let be $\mathbf{e}_s$ the ray tangent before reflection; the law of specular reflection requires the new tangent to become

$$(\mathbf{e}_s)' = \mathbf{e}_s - 2\,\mathbf{n}\,(\mathbf{n} \cdot \mathbf{e}_s) \ ; \tag{2.27}$$

knowing $(\mathbf{e}_s)'$ one can restart the integration the Eikonal at the intersection point along the direction of specular reflection.

In order to provide accurate estimates of (x_i, y_i, z_i) for any degree of boundary roughness the following strategy is used:

- Divide the ray segment into a sequence of linearly distributed points, starting on one side of the boundary and ending on the other side;
- Use the interpolation polynomials to calculate the vertical distance from both start and end of the ray segment to the boundary (this distance will change sign as one moves along the segment);

- Use bisection to determine the pair of points (x_k, y_k, z_k) and $(x_{k+1}, y_{k+1}, z_{k+1})$ where the vertical distance changes sign.

- Determine (x_i, y_i, z_i) from (x_k, y_k, z_k) and $(x_{k+1}, y_{k+1}, z_{k+1})$ using linear interpolation.

This strategy can be expected to be accurate, but it is also computationally demanding. Yet, it has been found to converge very rapidly in most cases because the ray step is always smaller than the typical roughness of the boundary.

Chapter 3

Numerical enhancements

Synopsis: *A description of the numerical enhancements of the TRACEO3D ray tracing model is presented in this chapter, namely, the Simplex-based eigenray search, and the optimization of ray influence calculations. The eigenray Simplex-based search was developed to efficiently and accurately calculate 3D eigenrays, providing predictions that account for horizontal effects. Ray influence calculations were also improved with the main goal of reducing the computational time, which often increases drastically as range, number of rays and number of sensor increases. The structure of this chapter is as follows: Section 3.1 presents the eigenray Simplex-based search, while Section 3.2 describes the procedures that compound the ray influence calculations.*

3.1 The Simplex-based eigenray search

In the original version of TRACEO3D eigenray search was based on the "proximity" method, i.e. by launching as many rays as possible, and keeping only rays ending inside a sphere centered on a given receiver, with the sphere radius being defined by the user. This approach was found to be computationally demanding and inaccurate. The Simplex method was used to address the problem efficiently, with the 3D search of eigenrays being based on three different strategies:

1. To start the search determine a reliable candidate region that encloses the receiver.

2. Apply the general rules of Simplex optimization using the candidate region to find an eigenray.

3. Avoid the storage of duplicated eigenrays.

These strategies are discussed in detail in the following three sections (a compact discussion and validation is also presented in [50]). The full algorithm is presented in section 3.1.4.

3.1.1 Selection of a reliable candidate region

Let θ and ϕ be the ray elevation and azimuth, respectively. For a given set of receivers the initial choice of take-off angles (defined by a set of θ and ϕ pairs at the source) depends on many waveguide features, such as boundary variations over the horizontal plane, source-receiver alignment, and the existence or absence of environmental variations. In any case a given choice should aim at sweeping the waveguide in such a way, that a large number of rays should be propagating among all receivers; in such conditions it can be expected that "enough" eigenrays will be found at every receiver, allowing to predict accurately the corresponding impulse response. For a given receiver, a vertical plane is calculated using the normal vector connecting the source to the receiver, and the crossings of rays through the plane determine the closest distance from each ray to the receiver. Let θ_i and ϕ_j define the take-off angle of the (i, j)th ray; a candidate search space is then built with the region defined by the corners

$$\begin{bmatrix} \theta_i, \phi_j & \theta_i, \phi_{j+1} \\ \theta_{i+1}, \phi_j & \theta_{i+1}, \phi_{j+1} \end{bmatrix}$$

These corners are changed over iterations according to the following rules:

- fix i and increment j until the horizontal deviation of the closest distance vanishes;

- increment i and repeat the previous step until it covers the vertical deviations.

At each iteration a new search region is created; the corresponding corners are used to divide the region in triangles using the following combinations:

1. $[\theta_i, \phi_j \quad \theta_{i+1}, \phi_j \quad \theta_i, \phi_{j+1}]$;

2. $[\theta_i, \phi_j \quad \theta_{i+1}, \phi_j \quad \theta_{i+1}, \phi_{j+1}]$;

3. $[\theta_i, \phi_j \quad \theta_i, \phi_{j+1} \quad \theta_{i+1}, \phi_{j+1}]$;

4. $[\theta_{i+1}, \phi_j \quad \theta_i, \phi_{j+1} \quad \theta_{i+1}, \phi_{j+1}]$;

The method calculates the barycentric coordinates $\boldsymbol{\lambda}$ to determine which triangle contains the receiver, with $\boldsymbol{\lambda}$ given by

$$\begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \end{pmatrix} \boldsymbol{\lambda} = \begin{pmatrix} x_r \\ y_r \\ z_r \end{pmatrix} \ ; \tag{3.1}$$

in Eq(3.1) (x_1, y_1, z_1), (x_2, y_2, z_2) and (x_3, y_3, z_3) represent the coordinates of the triangle vertex, and (x_r, y_r, z_r) are the coordinates of the receiver. The search considers all triangles; it decides that the receiver lies inside a given triangle when the components of the normalized $\boldsymbol{\lambda}$ are all positive. When this happens the take-off angles (θ, ϕ) of the corresponding vertex are considered for the next step (i.e. for Simplex optimization). A visualization of the four corners used for selection is presented in Fig. 3.1, with the candidate region located at the first combination of launching angles (i.e., at the corners corresponding to $[\theta_i, \phi_j \quad \theta_{i+1}, \phi_j \quad \theta_i, \phi_{j+1}]$). All triangles are considered until the one containing the receiver is found. The selection step is fundamental in order to overcome the chaotic distribution of vertex corners induced by the waveguide. In fact, rays from an initially narrow pyramid will

end up producing an amorphous cloud of corners near the receiver, with consecutive rays following completely different trajectories. For instance, one corner can be produced by a ray coming from the bottom, while another corner can be produced by a ray coming from the surface.

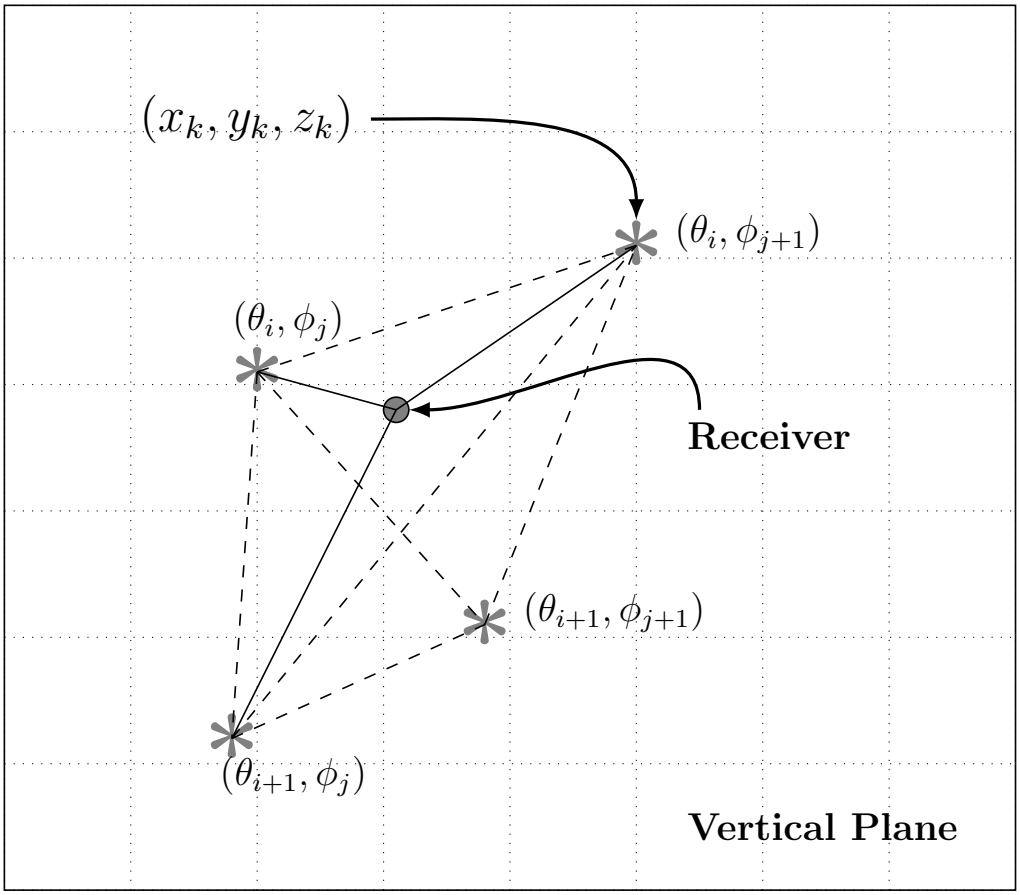


Figure 3.1: The four corners (represented as asterisks) used to find a reliable candidate region containing a receiver; all points are located on a vertical plane, associated to the receiver. To each corner corresponds a set of coordinates (x_k, y_k, z_k), which define the point of ray-plane intersection. The region is divided into triangles (dashed lines), and barycentric coordinates (solid lines) λ_1, λ_2 and λ_3 are used to determine which triangle contains the receiver.

3.1.2 Simplex optimization

The Simplex method was developed as a general strategy to optimize a function of N variables [51]. A simplex can be idealized as a geometric figure in N dimensions, defined

by a set of $N + 1$ points; for instance, a simplex is a triangle in two dimensions, and in three dimensions it is a tetrahedron. The method is able to achieve convergence in few iterations, and requires few function evaluations, a feature which is important when dealing with complicated objective functions [52].

Within the context of eigenray search the objective function to be minimized can be defined as

$$f(\theta,\phi) = \sqrt{\begin{array}{ll}[x_r - x(\theta,\phi)]^2 & + \\ [y_r - y(\theta,\phi)]^2 & + \\ [z_r - z(\theta,\phi)]^2 & \end{array}} \quad , \tag{3.2}$$

where $x(\theta,\phi)$, $y(\theta,\phi)$ and $z(\theta,\phi)$ represent the ray coordinates on the vertical plane of the receiver. Each calculation of the objective function requires solving the system of Eikonal equations for a given pair of angles (θ,ϕ). The selection of a candidate region (as discussed in section 3.1.1) delivers a high quality initial guess for the simplex algorithm to start the minimization of the objective function; each corner corresponds to a point combination $P_k = (\theta,\phi)$, with $k = 0 \ldots N$. An initial simplex is computed between each vertex of the triangle and its centroid. Each calculation of new points produces a simplex with the same triangular shape inside the initial region. Additionally, overlapping triangles can be used to restart the optimization in regions in which the convergence is failing. Once the simplex is started it uses three operations called *reflection*, *contraction* and *expansion* based on the centroid $\bar{P}$, which are defined as follows:

- The reflection is denoted by P^* and its coordinates are calculated by the relation

$$P^* = (1+\alpha)\,\bar{P} - \alpha P_h \tag{3.3}$$

  where α stands for a positive constant *reflection coefficient*, and h represents the point that gives the *highest* function value. Whether $f(P^*)$ lies between $f(P_h)$ and $f(P_l)$,

with l corresponding to the point that gives the *lowest* function value, P_h is replaced by P^* and the optimization restarts with the new simplex.

- However, if the reflection produces a new minimum, then we expand P^* to P^{**}, given by

$$P^{**} = \gamma P^* + (1 - \gamma)\,\bar{P} \tag{3.4}$$

where γ corresponds to the *expansion coefficient*, with $\gamma > 1$. If the new point gives a successful expansion, P_h is replaced by P^{**} and the process is restarted; otherwise P_h is replaced by P^*.

- If reflecting P to P^* gives $f(P^*) > f(P)$ for all $k \neq h$ then a new P_h is defined to be either the old P_h or P^*, whichever produces the closest distance, and forms the contraction, denoted by

$$P^{**} = \beta P_h + (1 - \beta)\,\bar{P} \tag{3.5}$$

where β stands for the *contraction coefficient*, with $0 < \beta < 1$. A successful contraction replaces P_h by P^{**}, while a failed one replaces all points by $(P_k + P_l)/2$.

The optimization stops when the value of $f(P)$ at a given vertex is below a predefined threshold. For the sake of clarity all steps of simplex optimization are illustrated in the pseudo-code of Algorithm 1.

During initial tests for a single receiver the algorithm achieved a remarkable convergence with $\alpha = 1.5$, $\gamma = 1.65$ and $\beta = 0.5$. Those values were found to guarantee also the convergence of the method for multiple receiver configurations[1]. Yet, it was also found

[1] It should be noticed that parallel tests using swarm optimization (not shown here) with different combinations of "proper" optimization parameters often failed to achieve the desired accuracy, besides requiring significant amounts of computational time.

Algorithm 1 Pseudo-code of Simplex optimization

calculate initial simplex for points P_k
while $f(P_l) > threshold$ **do**
  form $P^* = (1+\alpha)\,\bar{P} - \alpha P_h$
  if $f(P^*) < f(P_l)$ **then**
    form $P^{**} = \gamma P^* + (1-\gamma)\,\bar{P}$
    if $f(P^{**}) < f(P_l)$ **then**
      replace P_h by P^{**}
    else
      replace P_h by P^*
    end if
  else
    if $f(P^*) > f(P_k)$ **then**
      if $f(P^*) < f(P_h)$ **then**
        replace P_h by P^*
      end if
      form $P^{**} = \beta P_h + (1-\beta)\,\bar{P}$
      if $f(P^{**}) > f(P_h)$ **then**
        replace all P_k by $(P_k + P_l)/2$
      else
        replace P_h by P^{**}
      end if
    else
      replace P_h by P^*
    end if
  end if
end while
return P_l

that a "blind" application of Simplex optimization could lead to the calculation of the same eigenray using different candidate regions. To avoid this duplication a final step was implemented, as described in the following section.

3.1.3 Avoiding storage of duplicated eigenrays

To avoid the storage of duplicated eigenrays the following procedure was adopted:

- After the calculation of a given eigenray it was verified that the corresponding pair (θ, ϕ) was found to be inside the candidate region. The eigenray was discarded when the pair was outside.

- As each eigenray was being calculated the information regarding (θ, ϕ) together with the number of bottom and surface reflections was stored in memory; the information regarding a new eigenray was compared with the information of old ones, and the eigenray was discarded if already present.

- At the end of calculations the eigenrays were sorted according to the time of arrival.

3.1.4 The Simplex-based algorithm of 3D eigenray search

The complete pseudo-code of Simplex-based 3D eigenray search implemented in TRACEO3D is shown in Algorithm 2, with proper line identification; two main processing stages can be noted. The first one (lines 5 to 14), corresponds to the computation of corners for the reliable candidate region, which are calculated tracing all rays sequentially. For each new ray segment an intersection test against a given receiver plane is performed. When the intersection occurs the crossing coordinates are stored. However, depending on the launching angle, rays can propagate without crossing any receiver plane, or crossing only some of them. In both cases "invalid" corners are stored in memory to prevent searches in regions without rays. For horizontal arrays it is desirable to reduce runtime avoiding shooting rays repeatedly for different receiver ranges. In such case ray-plane intersections are calculated progressively as rays propagate through the waveguide (see Fig 3.2).

The second working stage (lines 15 to 29) is related to the search itself, and it is performed over the number of candidate regions. When a candidate region encloses a receiver the Simplex-based search strives to find an optimized pair of take-off angles (θ_o, ϕ_o), which fulfills both threshold and duplicated conditions. If that is the case the pair (θ_o, ϕ_o) is used to calculate the ray trajectory, together with the travel time and amplitude, and the

Algorithm 2 Pseudo-code for the Simplex-based 3D eigenray search

```
load environmental data
let ϕ = set of azimuth angles
let θ = set of elevation angles
let r = set of receivers
for j := 1 → length(ϕ) do
  for i := 1 → length(θ) do
    while ray (θ_i, ϕ_j) exists do
      solve the Eikonal equations for segment k
      if segment crosses r_l vertical plane then
        store ray crossing coordinates
      end if
    end while
  end for;
end for;
for j := 1 → (length(ϕ) − 1) do
  for i := 1 → (length(θ) − 1) do
    for l := 1 → length(r) do
      compute barycentric coordinates for (θ_i, ϕ_j), (θ_{i+1}, ϕ_j), (θ_i, ϕ_{j+1}), (θ_{i+1}, ϕ_{j+1})
      combination
      if encloses r_l then
        perform Simplex optimization to find (θ_o, ϕ_o)
        if threshold is satisfied then
          avoid duplicated eigenray
          solve the Eikonal equations using (θ_o, ϕ_o)
          solve the dynamic equations using (θ_o, ϕ_o)
        end if
      end if
    end for
  end for
end for
return the eigenrays
```

information is stored as an eigenray.

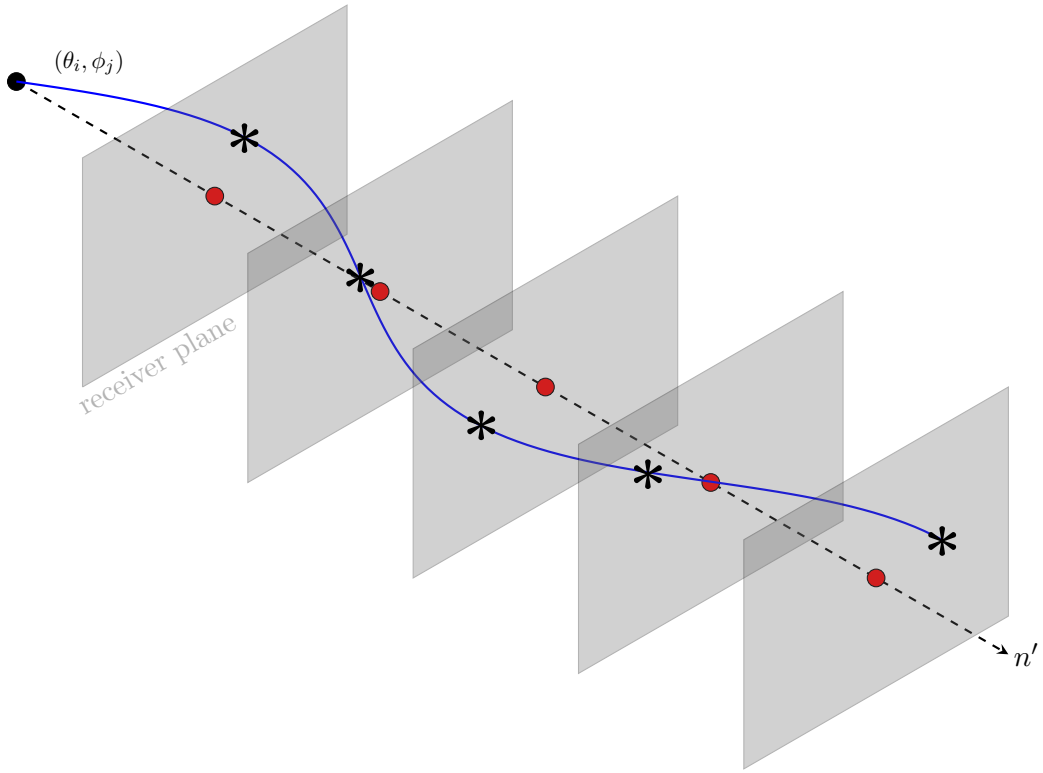


Figure 3.2: Ray-plane intersections, represented as asterisks, for a horizontal line array; the dashed line corresponds to the planes normal n'.

3.2 Calculations of ray influence

3.2.1 The receiver grid strategy

To address the problem described in Section 2.3.5 and reduce drastically the runtime without compromising accuracy one can follow the approach described in [2], which suggests that for each ray segment one considers *not all* receivers, but only those "insonified" (i.e. bracketed) between the endpoints of a ray segment. For a given subset of receivers one can move from the ocean surface to the ocean bottom within the subset, and rely on simple algebra to determine the parameters of ray influence; the procedure is then repeated for all ray segments. An examination of the BELLHOP3D ray tracing code [31] reveals that the determination of the subset of receivers is achieved by testing *all* receiver positions within the array, looking to find the ones within the endpoints of the ray segment. The approach implemented in TRACEO3D goes further and looks to determine an even smaller subset of receivers (called the *receiver grid*, see Fig. 3.3) based on the following considerations:

- A "finite" beam width W is defined along the ray, given by the expression

$$W = \left| \frac{Q_{11}(s)\Delta\theta}{\cos\theta(s)} \right| . \tag{3.6}$$

- There is no need to consider all receivers from the ocean surface to the ocean bottom, but only those within the neighborhood defined by W

The main idea on the basis of this strategy is that beyond the distance defined by W the influence is too small to be of any importance. Therefore, as one moves along each ray segment the receiver grid is determined by the receivers bracketed by both *the ray segment and* W. In this way one can avoid not only the query in the entire set of receivers forming

the array, but also the query of all receivers bracketed by the ray segment. The method can take advantage of Cartesian coordinates to determine efficiently the indexes of the receivers lying inside the receiver grid. The specific details of this enhancement are described in the next Section.

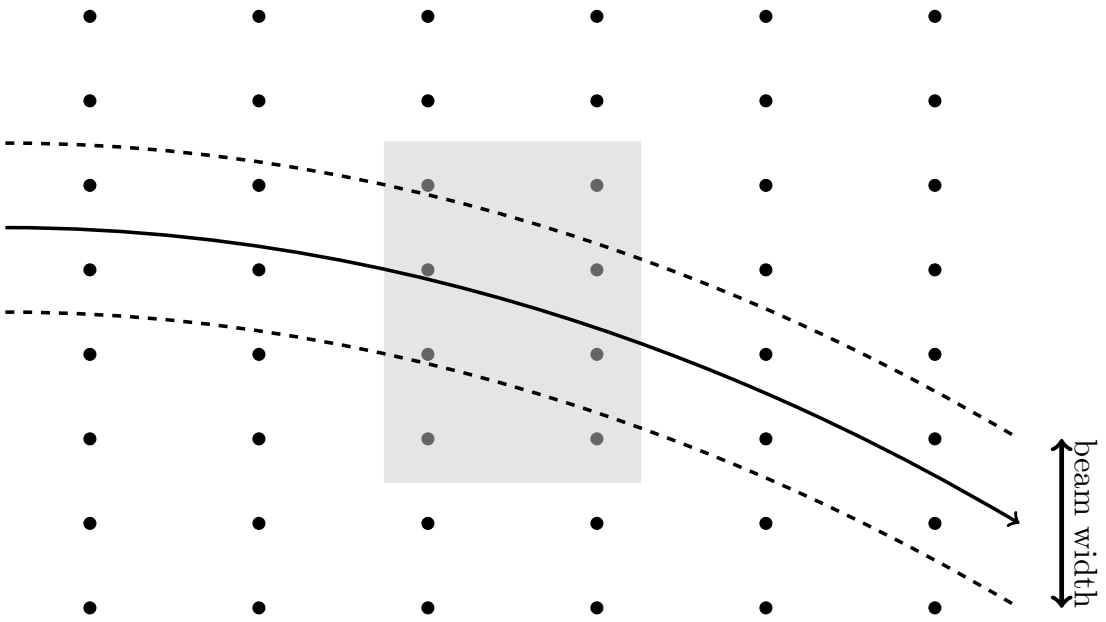


Figure 3.3: The receiver grid: the black dots represent *all* the receivers of a rectangular array, while the solid line represents the ray trajectory; the ray influence is only relevant within the limits of the beam width, represented by the dashed lines, and the gray rectangle represents the grid of receivers considered for the calculation of ray influence.

3.2.2 Ray influence calculation algorithm

The specific details of optimization are shown in the pseudo-code of Algorithm 3, which summarizes the sequential steps regarding field calculations. Let n and r stand for the number of rays and receivers, respectively. The optimization starts by tracing the ray for a given pair of launching angles. Then, the algorithm marches through the ray segments, and solves the dynamic equations to calculate the ray amplitude and the beam spreading. As shown in lines 13 and 14 a subset of receivers is computed from r for each segment k of the ray. The ray influence is computed only if a normal to the receiver is found at a given

segment (see lines 15 and 16). Line 22 presents the final step, in which coherent acoustic pressure for each receiver is calculated. As will be shown in Chapter 4 the set of nested loops constitutes a fundamental stage of the algorithm, allowing a a substantial improvement of the model's performance. Details regarding the computation of the receiver grid are shown in the pseudo-code of Algorithm 4; in it the integers l_{low} and l_{high} control the array indexes that form the receiver grid according to the finite beam width bw at each coordinate axis. The receiver indexes increase or decrease their values depending on the position of the ray segment inside the receiving array; the entire procedure is designed to be flexible enough to account for different ray directions.

Algorithm 3 Pseudo-code for sequential ray influence calculation

```
 1: load environmental data
 2: let ϕ = set of azimuth angles
 3: let θ = set of elevation angles
 4: let r = set of receivers
 5: consider n = length(ϕ) × length(θ)
 6: for j := 1 → length(ϕ) do
 7:    for i := 1 → length(θ) do
 8:       while ray (θ_i, ϕ_j) exists do
 9:          solve the Eikonal equations for segment k
10:       end while
11:       for k := 1 → raylength do
12:          solve the dynamic equations of segment k
13:          calculate w at segment k
14:          compute the receiver grid g from r according to w
15:          for l := 1 → length(g) do
16:             if ray_k and g_l are ⊥ then
17:                compute ray_k influence at g_l
18:             end if
19:          end for
20:       end for
21:    end for
22: end for
23: return the coherent acoustic pressure for each receiver
```

Algorithm 4 Pseudo-code to compute the receiver grid

1: **let** l_{low} = lower array index inside grid
2: **let** l_{high} = high array index inside grid
3: **consider** bw as beam width at ray_k
4: **while** l_{high} or l_{low} are inside grid **do**
5: **if** $bw < \boldsymbol{r}\,(l_{low} - 1)$ **then**
6: **decrement** l_{low}
7: **else if** $bw > \boldsymbol{r}\,(l_{low})$ **then**
8: **increment** l_{low}
9: **else**
10: **exit**
11: **end if**
12: **if** $bw < \boldsymbol{r}\,(l_{high})$ **then**
13: **decrement** l_{high}
14: **else if** $bw > \boldsymbol{r}\,(l_{high} + 1)$ **then**
15: **increment** l_{high}
16: **else**
17: **exit**
18: **end if**
19: **end while**

Chapter 4

Parallel GPU Implementation

Synopsis: *The goal of this chapter is to describe in detail the structure of the algorithm leading to the parallel GPU implementation of the TRACEO3D model, and to explain the design decisions for memory organization and execution configuration. A glimpse of the GPU architecture and the CUDA C programing model is also provided, by introducing some of the concepts used in the implementation. The structure of this chapter is as follows: Section 4.1 presents the Data-Parallel execution Model, while Section 4.2 describes the parallel GPU implementation & memory organization, and discusses the algorithm design.*

4.1 Data-Parallel execution Model

4.1.1 CUDA parallel organization

A GPU is a specialized electronic circuit, which is designed with the main goal of accelerating the creation of images for the corresponding output on a display. In contrast with the CPU, a GPU is highly parallel, and has optimized many-core processors for high-definition 3D graphics with high memory bandwidth. While the CPU consists of few cores optimized for sequential processing, the GPU has a massively parallel architecture consisting of thousands of cores with more transistors, dedicated to data processing rather than data caching and flow control [53]. The GPU computing is intended to address problems with a large amount of data, which is executed by the same program. Among several programming frameworks that

handle with GPU cores the mostly widely used is the Compute Unified Device Architecture (CUDA), developed by NVIDIA [54]. CUDA is a scalable parallel programming model and software platform which provides C, C++ and Fortran extensions to develop parallel programs. This model implements a data-parallel function, denominated *kernel*, which is executed by all threads during a parallel step. Generally speaking, a CUDA program starts in the *host*, as a CPU sequential program, and when a kernel function is launched, it is executed in a grid of parallel threads into the GPU or *device*, as shown in Fig 4.1.

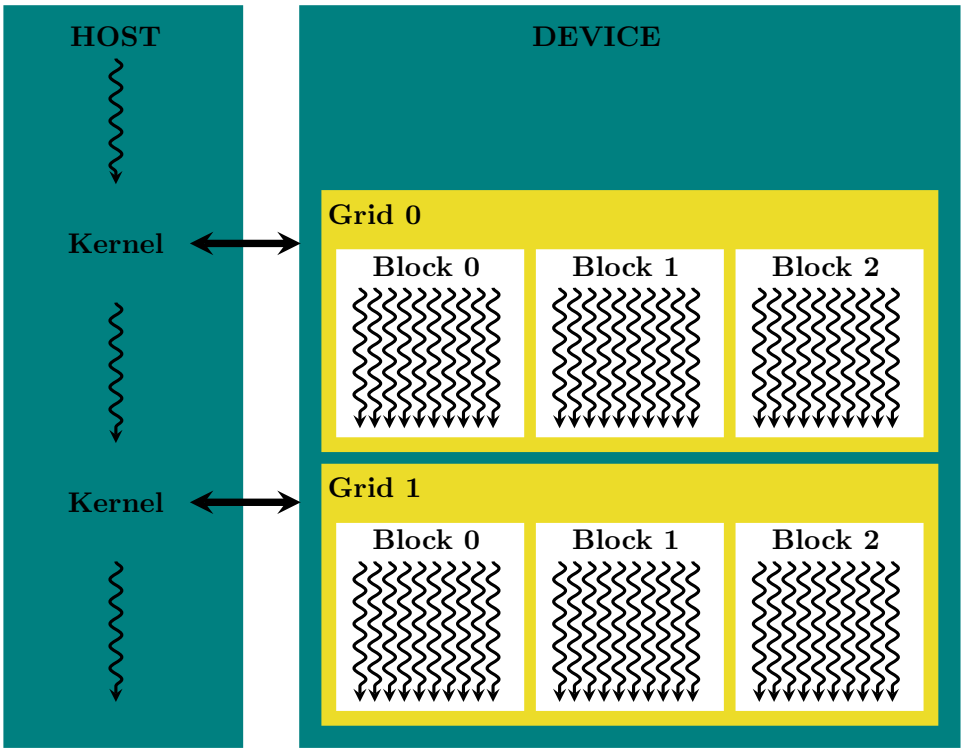


Figure 4.1: CUDA heterogeneous computing organization, with kernels launching grids of threads blocks for parallel processing into a device.

Each grid is organized as an array of blocks, where each block is compounded by an array of threads. The number of threads in a block and the number of blocks in a grid are denominated as *grid size* and *block size*, respectively. The grid size and block size have to be specified as execution configuration parameters of the kernel function. Each thread in

a block has a unique identification value, and each block has a unique identification value in a grid. These identifications are a 3-component vector, which can identify the thread using one, two or three dimensional indexes. Thus, a given thread can combine each index identification with the block size to produce a global identification for itself in the entire grid. Since each thread executes the same code it creates an efficient way to directly access a particular part of the data.

4.1.2 Device memories

CUDA relies on additional methods to access different types of memory, that help to overcome long access latencies and finite bandwidth. As described in Table 4.1 each memory has its own scope, access type and lifetime, whose combination provides distinct strategies to improve performance. Memory organization of a generic CUDA device is illustrated in Fig. 4.2: the host can transfer data to the device memory through global, constant or texture memory, and these spaces are visible among kernels calls. They are accessible by all threads and present an on-chip cache to improve performance.

Table 4.1: CUDA device memory types.

Memory	*Device access*	*Scope*	*Lifetime*
Global	R/W	All threads / host	Application
Constant	R	All threads / host	Application
Texture	R	All threads / host	Application
Shared	R/W	Thread block	Kernel
Local	R/W	Thread	Kernel
Register	R/W	Thread	Kernel

The global memory can be read and written from both host and device; the constant and texture memory can be read and written from the host. However, they are read-only

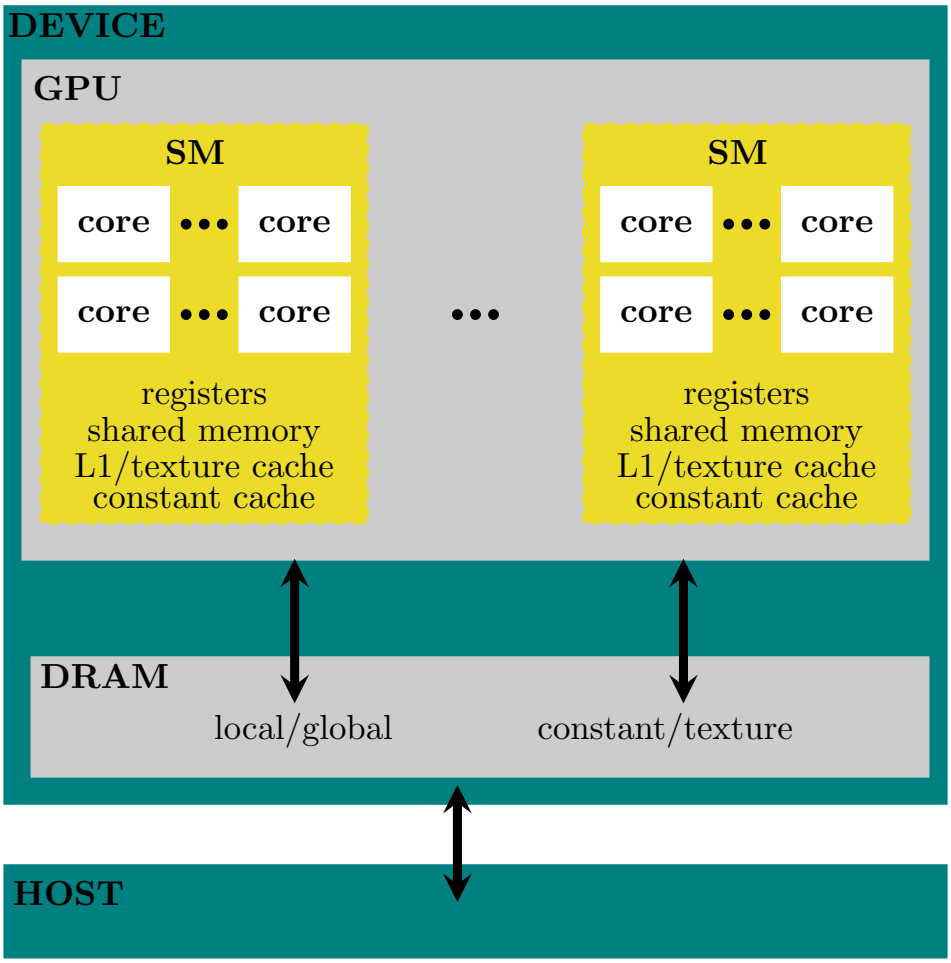


Figure 4.2: Generic CUDA device memory organization; it can be manipulated according to hardware computing capabilities.

from the device. Furthermore, texture memories are accessed through a dedicated read-only cache, that can be used for linear interpolation as part of the read process through hardware filtering. While global, constant and texture memories are located in *off-chip* memory, register and shared memories are located in *on-chip* memories, meaning that they have a very low latency to be accessed, roughly 100 times lower than uncached global memory. Shared memory can be accessed by all threads in the same block, and provides an efficient way to combine their partial results. Local variables in the device code are stored in the register if there is available space, otherwise they are stored in the global memory with *local* scope [55].

4.1.3 Thread execution

In the current GPU generation thread blocks are assigned to hardware resources and organized into streaming multiprocessors (hereafter SM). When a block is assigned to a given SM, it is further divided into a group of thread units denominated *warp*, which is the unit of thread scheduling. This is illustrated in Fig. 4.3, which presents an hypothetic partition of blocks into warps for scheduling in a SM. The SM is meant to execute threads in a warp based on the single instruction multiple thread model (hereafter SIMT). In this way, any threads in a warp are addressed to an execution unit to perform the same instruction at the same execution time. The *coalescing technique* [54] takes advantage of the SIMT model to improve global memory performance, since optimum access arises when all threads in a warp access consecutive global memory positions.

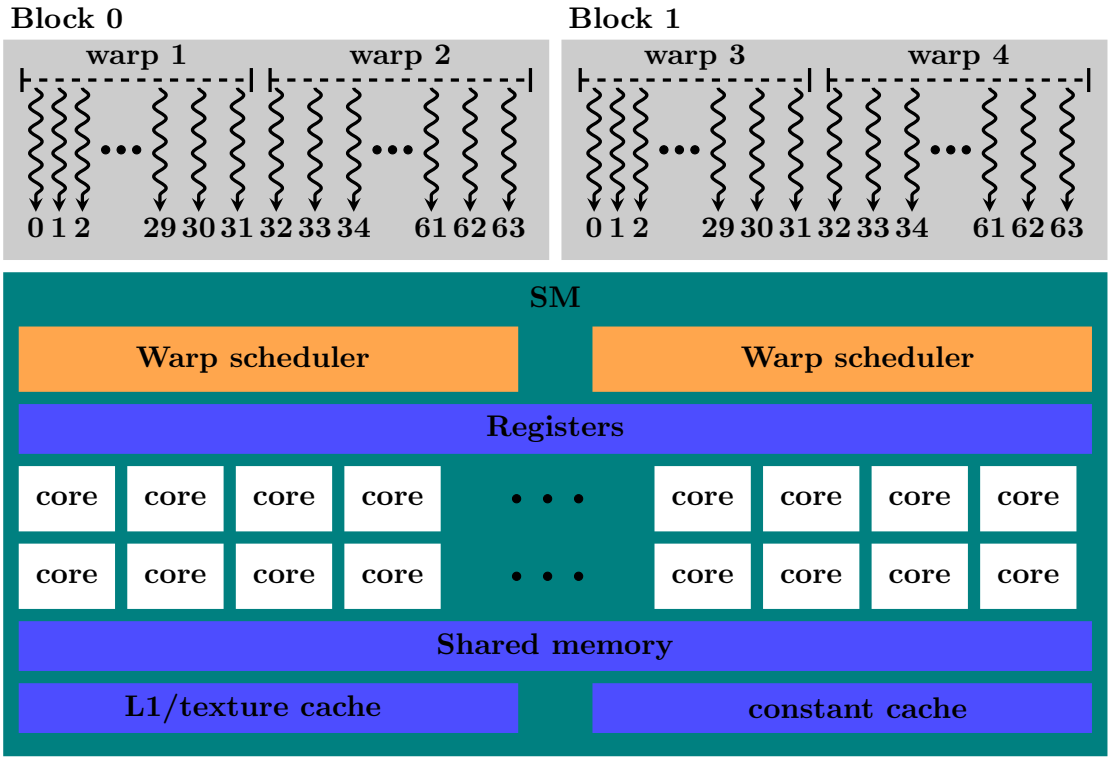


Figure 4.3: Schematic of blocks partition into warps for scheduling, and multiprocessor streaming architecture.

Each SM can execute instructions for a small number of warps at any instant in time. This strategy allows to fill in latency time operations with work from other threads, belonging to other warps, ready to be executed, and this is often called *latency hiding* [41]. These skills to tolerate long latency operations demonstrate how GPU overcomes the lack of chip area to cache memories and branch predictions, and dedicate instead more area to float-point execution resources. However, if threads in the same warp follow different paths due to a given branch condition, the SIMT hardware will take multiple steps to execute a different control flow. That is, the divergent path will be executed in sequence for each thread group. In general, the resource constraints in a given device can have a prominent impact in the execution speed of CUDA kernels. For instance, registers and shared memory can be useful in reducing the number of accesses to global memory. However, if the amount available is exceeded, the number of threads that can reside in a given SM decreases.

The classical approach of CUDA development relies in keeping the SM as busy as possible, meaning that the execution configuration should be optimized to launch a sufficient amount of lightweight threads [56]. In this way, it will maximize the hardware *occupancy*, which is defined as the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps [41]. However, some works show that the application may achieve better performance with low occupancy [57, 58], once it is allowed to share more register memory per thread which is the fastest on-chip memory. Although this increase in number of registers certainly decreases the number of threads per SM, it is compensated by increasing the numbers of operations per thread. An analysis regarding the SM occupancy and the execution configuration parameters will be presented in Sections 5.3.2 and 5.4.2 for each validation result.

4.2 Parallel TRACEO3D implementation

4.2.1 Ray tracing algorithm considerations

Generally speaking, ray tracing has an inherent parallelism, since rays can be computed independently or in any order. Furthermore, 3D propagation involves launching thousands of rays to cover the waveguide in elevation and azimuth, a task which represents a computationally demanding workload [59]. Such ray independence, combined with a high workload that can achieve a massive parallelism, is the main attraction in a GPU hardware multithreading. Preliminary research into parallelization in a coarse-grained fashion [24] was developed using OpenMPI [60]. Performance analysis showed that the parallel implementation followed a linear speedup when each process was addressed to a single physical CPU core. However, such performance was achieved at the cost of using high-end CPUs, designed for computer servers without network communication (which probably would decrease the overall performance). Thus, the best parallel implementation was 12 times faster than the sequential one, meaning that the execution took place in a CPU with 12 physical cores.

As introduced in Chapter 2, TRACEO3D relies on the 3D solution of the Eikonal equations to calculate ray trajectories, and on the solution of dynamic equations to calculate ray amplitudes. A preliminary timing analysis in the sequential code of a generic ray calculation is presented in Fig. 4.4, which shows that the solution of the Eikonal is the most time consuming task; given the dependence of the dynamic equations on ray trajectories a substantial portion of the code to be parallelized needs to be focused on a balanced combination of such tasks. The implementation thus adopted the inherent ray tracing parallelism, addressing each pair of launching angles (θ, ϕ) as a single parallel thread, even though it could lead to

the concentration of additional work per thread.

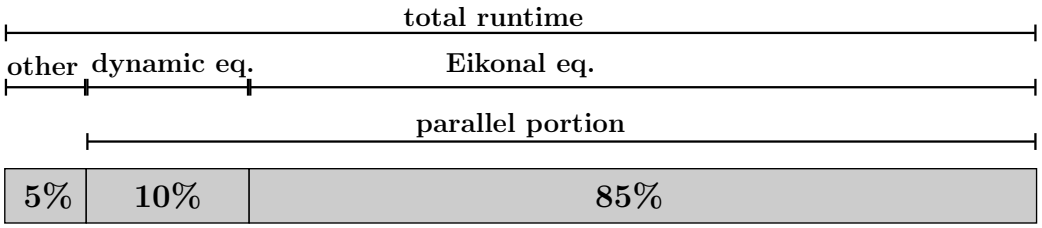


Figure 4.4: TRACEO3D timing analysis, showing roughly the percentage of runtime required to perform a generic calculation.

Another important issue is that once initial conditions are loaded into the GPU memory, rays are computed until the end without any additional request. Since the GPU is a co-processor this strategy in fact contributes to the achievement of high performance execution in a GPU architecture, since it avoids the bottleneck represented by data transfer between the host and the device. Yet, a drawback of the ray tracing algorithm is that each ray follows a different path, since some rays can experience only refraction, while others may be bouncing on different boundaries; this diversity of behaviors leads to the execution of different instructions. When this happens the warp schedule executes the divergent threads sequentially. Additionally, the access to different environmental information depending on the ray path can change drastically the pattern of memory access and hamper the use of the coalescing technique, which is fundamental to improve global memory access performance. Strategies to deal with those issues are presented in the following sections.

4.2.2 Memory organization

Acoustic predictions in a three-dimensional scenario demand the tracing of a high number of rays. In the sequential algorithm the ray trajectory information (such as, for instance, Cartesian coordinates, travel time, complex decay, polarized vectors, caustics, matrices **P**

and **Q**, etc.) are stored in memory to be used at later steps, such as eigenray searching or ray influence. Such storage makes sense considering that the sequential algorithm keeps one ray at a time in memory. However, handling thousands of rays in parallel rapidly exceeds the available memory in a given device. To circumvent this issue calculations of ray paths and amplitudes are performed in a single step, for each ray segment at a time, storing in memory only the values required to execute such computation. A sketch of this strategy is presented in Fig 4.5, where the horizontal arrow represents the direction in which memories are updated, and t corresponds to the current time in which calculations are taking place; $t-1$ and $t-2$ represent previous values, that are required to be held in memory. Small arrows connecting memory positions represent the values accessed by the corresponding function to perform computations in time t. The vertical arrow indicates the order in which functions are computed in the current time. The same structure is valid for both functions, eigenray search and ray influence calculation. After calling the functions for a given ray segment the values stored in memory at time $t-1$ are copied to the position $t-2$, and the values regarding t are copied to position $t-1$, meaning that the values at $t-2$ are discharged. A new iteration then starts to solve the next ray segment, following the same rules. In this way, the storage requires only three segments to be held in memory, reducing drastically the amount of data stored. This organization allows further updates of data into registers to be kept, reducing the global memory access and overcoming the problems of divergences in the pattern of memory access. The performance of memory access is also increased by loading part of the environmental information into the shared memory at the kernel initialization.

An overview of how data from the parallel implementation is organized into device memories is shown in Table 4.2. The memory type was chosen considering the respective

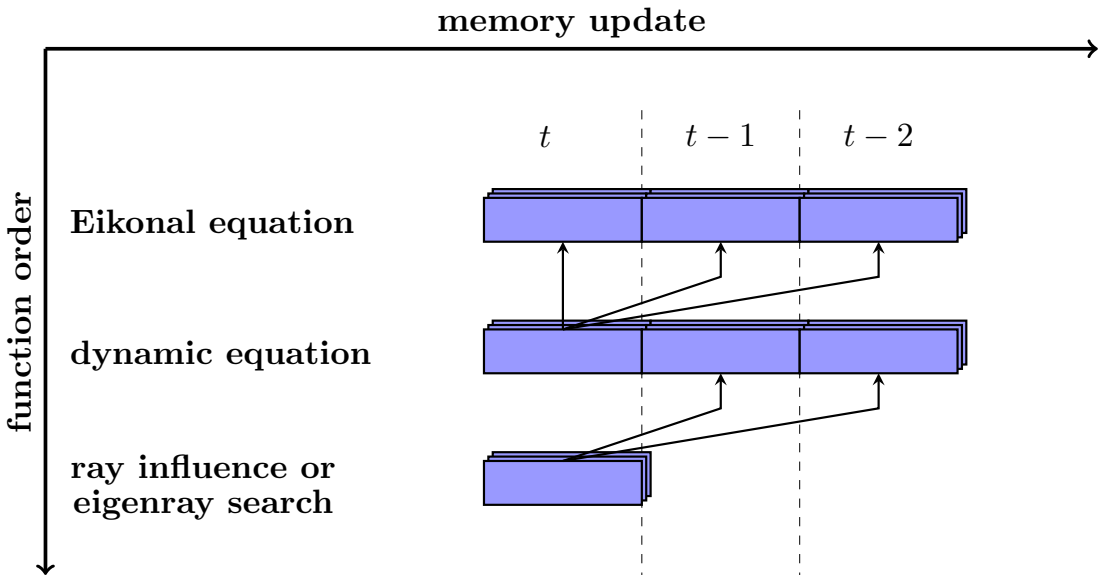


Figure 4.5: Schematic representing the memory update sequence (horizontal arrow), where t stands for current computation time and $t-1$ and $t-2$ for previous times when values are held in memory. Small arrows connecting memory positions represent the values accessed for the corresponding function to perform computations in time t. The vertical arrow represents the order in which the functions are executed for a single ray segment.

data size and the frequency in which the data is accessed. For instance, data regarding environmental boundaries (surface and bottom) was initially put into the shared memory. However, when representing 3D waveguides, the number of coordinates became too big to fit in this type of memory and the data was thus moved to the global memory. On the other hand, the sound speed data was kept in shared memory since it was frequently accessed during ray trajectory calculations and the access takes place in an unpredictable order.

4.2.3 Parallel eigenray Simplex-based search

A general view of the parallel version of eigenray Simplex-based search is presented in Algorithm 5. The proposed parallel algorithm is logically organized in a grid of b blocks, where each block has p threads. Generally speaking, p has great impact in the GPU throughput and needs to be calculated taking into account the GPU features and local memory utiliza-

Table 4.2: TRACEO3D memory organization into a parallel implementation: n_{ssp} is the number of points in the sound speed profile, n_{sur} and n_{bot} is the number of grid points defining the surface and bottom, respectively; n stands for the number of rays, h represents the number of receivers and m is the number of candidate regions.

Data	*Symbol or name*	*Type*	*Size*
source information		shared	12
environment parameters		shared	14
sound speed profile		shared	$3 + n_{ssp}$
array coordinates		global	$3 + 3 \times h$
surface coordinates		global	$7 + n_{sur}$
bottom coordinates		global	$7 + n_{bot}$
candidate region corners	*reg*	global	$3 \times n \times h$
eigenray values	*eig*	global	$5 \times m \times h$
coherent acoustic pressure	*cpr*	global	h
cpr all rays	*ncpr*	global	$n \times h$
ray coordinates		register/local	3×3
travel time	τ	register/local	3
complex amplitude	A	register/local	3
polarized vectors		register/local	3×3
P	**P**	register/local	3×4
Q	**Q**	register/local	3×4
general parameters		constant	12

tion. Furthermore, p needs to be chosen as a multiple of the warp size of a given device, since it helps coalescing and increases computing efficiency. Two kernels were used to implement the eigenray search method. The first kernel (line 7) computes the candidate region corners, whose total number of parallel threads corresponds to n. Algorithm 6 depicts the *kernel corners calculation*, where parallel threads sequentially compute each corresponding ray and the global thread identification index, which corresponds to ti, is calculated as shown in line 1. When a ray (θ_i, ϕ_j) intersects a receiver plane, the respective intersection coordinates, represented as q, are stored into reg. The memory index is calculated taking advantage of the coalescence access as shown in line 8. After the kernel execution a device synchronization is performed, to ensure that all intersection coordinates were calculated before starting the

eigenray search.

Algorithm 5 Parallel eigenray Simplex-based search

1: **let** $\boldsymbol{\phi}$ = set of azimuth angles
2: **let** $\boldsymbol{\theta}$ = set of elevation angles
3: **let** $\boldsymbol{r}$ = set of receivers
4: **consider** $n = length\,(\boldsymbol{\phi}) \times length\,(\boldsymbol{\theta})$
5: **let** p = number of threads per block
6: **let** b = n/p (number of blocks)
7: **kernel** $\ll b, p \gg$ corners calculation
8: **synchronize**
9: **consider** $m = (length\,(\boldsymbol{\phi}) - 1) \times (length\,(\boldsymbol{\theta}) - 1)$
10: **let** p = number of threads per block
11: **let** b = m/p (number of blocks)
12: **kernel** $\ll b, p \gg$ eigenray search
13: **select** eigenrays
14: **return** eigenrays values

The second kernel (line 12 from Algorithm 5) performs the *eigenray search* using the candidate region corners computed during the first kernel execution. For this configuration the total amount of threads launched into the device corresponds to m. The details of the second kernel are shown in Algorithm 7. Each thread is addressed to a search region in order to perform triangle divisions that may enclose a given receiver. Otherwise, the thread computation goes to the next receiver (if there is another one), or it is concluded. When the Simplex algorithm finds an optimized pair of launching angles (θ_o, ϕ_o) it uses the pair to compute ray coordinates, travel time and amplitude decay, and stores the information as an eigenray. As in the first kernel, the memory index was calculated to perform a coalescence access to the global memory, as shown in line 12. Because of the unpredictability in which the receiver/region combination allows to find an eigenray, the size of *eig* (see Table 4.2) is calculated with enough space in memory, so each thread can store directly the corresponding eigenray data; this also allows to prevent a sequential thread memory access that might produce unnecessary synchronization or race conditions. At the end, a sequential procedure

is performed by the host to select valid eigenrays among those computed in parallel (line 13 from Algorithm 5).

Algorithm 6 Kernel corners calculation

1: **let** ti = block index × grid index + thread index;
2: **let** $\theta_i = ti$ mod $length\,(\boldsymbol{\theta})$
3: **let** $\phi_j = ti/length\,(\boldsymbol{\theta})$
4: **while** ray (θ_i, ϕ_j) exists **do**
5: l = first receiver index
6: **solve** the Eikonal equations for each ray segment
7: **if** segment crossing vertical plane regarding r_l **then**
8: $reg[ti + n \times l] = q$
9: **increment** l
10: **end if**
11: **end while**

Algorithm 7 Kernel eigenray search

1: **let** ti = block index × grid index + thread index;
2: **let** $\theta_i = ti$ mod $(length\,(\boldsymbol{\theta}) - 1)$
3: **let** $\phi_j = ti/\,(length\,(\boldsymbol{\theta}) - 1)$
4: **for** $l := 1 \rightarrow length\,(\boldsymbol{r})$ **do**
5: **compute** barycentric coordinates for (θ_i, ϕ_j), (θ_{i+1}, ϕ_j), (θ_i, ϕ_{j+1}), $(\theta_{i+1}, \phi_{j+1})$ combination
6: **if** encloses r_l **then**
7: **perform** Simplex optimization to find (θ_o, ϕ_o)
8: **if** threshold is satisfied **then**
9: **avoid** duplicated eigenray
10: **solve** the Eikonal equations using (θ_o, ϕ_o)
11: **solve** the dynamic equations using (θ_o, ϕ_o)
12: $eig[ti + m \times l] = A,\ \tau,\ (\theta_o, \phi_o)$
13: **end if**
14: **end if**
15: **end for**

There are two possibilities to speedup the 3D eigenray search. The first one would calculate ray coordinates in parallel, using only the kernel corners calculation, and would then perform the search sequentially into the host, meaning that Algorithm 5 executes only lines 1 to 8, and then jumps to line 15 of Algorithm 2. This approach will produce ray coordinates, travel time and amplitude along the entire trajectory, and it is suitable for an

array with a small number of receivers, or when the ray information is of interest. The second option is to call all functions in parallel, using the two kernels presented in Algorithm 5. This configuration delivers only eigenray values at the receiver position, and is well suited for large arrays (which is of interest for matched field processing [61,62]), or when the ray trajectories are not required. However, pairs of launching angles for each eigenray are provided allowing the corresponding information to be recovered if needed.

4.2.4 Parallel field calculation

Algorithm 8 shows a summarized version of the parallel calculation of the pressure field. Two kernels were used to implement the parallel code. The first kernel (line 8) calculates the ray influence, where the number of threads launched into the device corresponds to n. An overview of the kernel *ray influence calculation* is shown in Algorithm 9. Each

Algorithm 8 Parallel field calculation

1: **load** environmental data
2: **let** $\boldsymbol{\phi}$ = set of azimuth angles
3: **let** $\boldsymbol{\theta}$ = set of elevation angles
4: **let** $\boldsymbol{r}$ = set of receivers
5: **consider** $n = length\,(\boldsymbol{\phi}) \times length\,(\boldsymbol{\theta})$
6: **let** p = number of threads per block
7: **let** b = n/p (number of blocks)
8: **kernel** $\ll b, p \gg$ ray influence calculation
9: **synchronize**
10: **let** p = number of threads per block
11: **let** b = h/p (number of blocks)
12: **kernel** $\ll b, p \gg$ pressure by sensor reduction
13: **return** the coherent acoustic pressure

thread computes the propagation of a single ray and its contributions to the entire field; the contributions are stored separately by ray. It should be noted that, as shown in Table 4.2, the size of *ncpr* corresponds to $n \times h$. After the kernel execution a device synchronization is performed to ensure that the acoustic field calculation for all rays was concluded. Then,

a second kernel is launched to perform a parallel reduction over the values in $ncpr$. Each thread is addressed to a given receiver, and it adds sequentially the contribution of each ray to the corresponding receiver, as described in Algorithm 10.

Algorithm 9 Kernel ray influence calculation

let ti = block index $\times$ grid index + thread index
let $\theta_i = ti \bmod length\,(\boldsymbol{\theta})$
let $\phi_j = ti/length\,(\boldsymbol{\theta})$
while ray (θ_i, ϕ_j) exists **do**
  solve the Eikonal equations for segment k
  compute the dynamic equations for segment k
  compute the receiver grid $\boldsymbol{g}$ from $\boldsymbol{r}$
  for $l := 1 \rightarrow length\,(\boldsymbol{g})$ **do**
    if ray_k and g_l are $\perp$ **then**
      $ncpr[ti + m \times l]$ = acoustic pressure regarding ray_k at g_l
    end if
  end for
end while

Algorithm 10 Kernel pressure by sensor reduction

let ti = block index $\times$ grid index + thread index;
consider n number of rays;
consider r as a thread local memory;
for $k := 1 \rightarrow n$ **do**
  $r = r + ncpr\,[k + (n \times ti)]$;
end for;
$cpr\,[ti] = r$;

Chapter 5

Validation

Synopsis: *This chapter presents the validation results regarding model predictions of 3D eigenray search and transmission loss. Details regarding the software/hardware platform are presented in Section 5.1 addressing the enhanced parallel model for performance analysis. The comparisons were carried out based on simulations and experimental data. The 3D acoustic propagation data were acquired in 2007 during a laboratory-scale experiment, that took place at the LMA-CNRS laboratory in Marseille. The experiment is described in Section 5.2, while the validation (together with a performance analysis) is discussed in Sections 5.3 and 5.4.*

5.1 Implementation

The original version of TRACEO3D was written using the FORTRAN programing language in double precision. Therefore, the first GPU parallel version of the model was developed using the CUDA FORTRAN 17.1 Community Edition compiler, developed by PGI [63]. Preliminary calculations exhibited in fact unsatisfactory performance, which was later found to be a result of particular restrictions regarding the device subprograms [64]. To overcome this problem the CUDA C platform was chosen to encode the parallel portion (as discussed in Section 4.2.1), meaning that the only interface kept in TRACEO3D was that of FORTRAN, using its functions to read the inputs and write the outputs. The CUDA C and FORTRAN

environments were connected using the ISO C Binding library [65], which is a standardized way to generate procedures, derived-type declarations and global variables, which are interoperable with C. The parallel implementation was compiled in a single precision version (numerical stability was already addressed in [45]); to properly clarify this issue comparisons regarding precision will be shown between the parallel and the sequential version. The single precision version allows the use of low-end devices or mobile equipments to provide predictions with high performance. Additionally, the FORTRAN sequential implementation was compiled with the optimization flag $-O3$, which was found to decrease the total runtime in 50%. The hardware and software features that were addressed when comparing the sequential and parallel model implementations of TRACEO3D are shown in Table 5.1. Fifteen runs per test case were performed for every validation case. The maximum and minimum values were then discarded, and the average run time was computed from the remaining thirteen runs.

Table 5.1: *Host*/*Device* hardware and software features.

Feature	Value	Unit
Host - CPU Intel i7-3930k		
Clock frequency	3500	MHz
Compiler	gfortran 5.4.0	–
Optimization flag	$-O3$	–
Device - GPU GeForce GTX 1070		
CUDA capability	6.1	–
CUDA driver	9.1	–
Compiler	nvcc 9.1.85	–
Optimization flag	none	–
Clock frequency	1683	MHz
Number of SM	15	–
Max threads per SM	2048	–
Warp size	32	–

5.2 The tank experiment

The laboratory-scale experiment took place at the indoor tank of the *Laboratoire de Mécanique des Fluides et d'Acoustique – Centre National de la Recherche Scientifique* (LMA-CNRS) laboratory in Marseille. The experiment was carried out in 2007 in order to collect 3D acoustic propagation data using a tilted bottom in a controlled environment. A brief description of the experiment (which is described in great detail in [9, 66]) is presented here. The inner tank dimensions were 10 m long, 3 m wide and 1 m deep. The bottom was filled with sand and a rake was used to produce a mild slope angle $\alpha \approx 4.5°$ (see Fig.5.1). The *ASP-H* data set (for horizontal measurements of across-slope propagation) is composed of time signals, recorded at a fixed receiver depth denominated z_r, and source/receiver distances starting from $Y = 0.1$ m until $Y = 5$ m in increments of 0.005 m, providing a sufficiently fine representation of the acoustic field in terms of range.

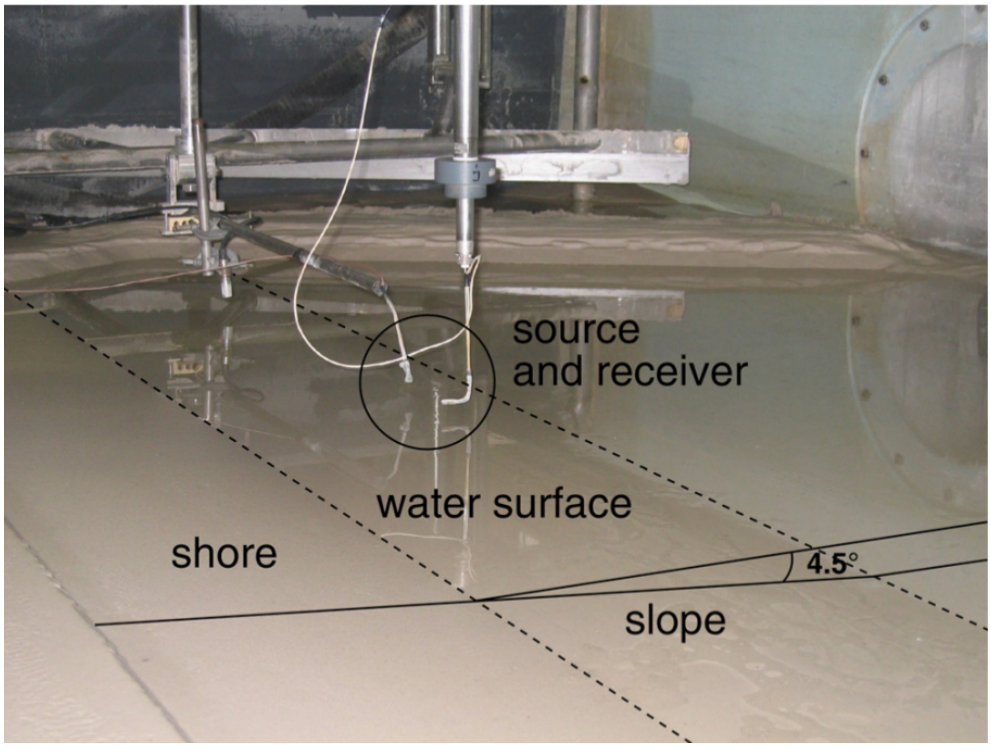


Figure 5.1: Indoor shallow-water tank of the LMA-CNRS laboratory of Marseille (from [9]).

The transmitted signal was a five-cycle pulse with a Gaussian envelope and with 0.04 ms duration. The signal presents a frequency spectrum with a main lobe centered at 150 kHz, with 100 kHz bandwidth, and a secondary lobe above 200 kHz. The source and the receiver were both aligned along the across-slope direction, as depicted in Fig 5.2. The receiver was located at 10 mm depth from the surface, bottom depth at the source position was $D(0)$ = 48 mm. Three different source depths were considered, namely z_s = 10 mm, 19 mm and 26.9 mm, corresponding to data subsets referenced as ASP-H1, ASP-H2 and ASP-H3, respectively. Bottom parameters corresponded to c_p = 1700 m/s, ρ = 1.99 g/cm^3 and α_p = 0.5 dB/λ. Sound speed in the water was considered constant, and corresponded to 1488.2 m/s for ASP-H1 and 1488.7 m/s for ASP-H2 and ASP-H3.

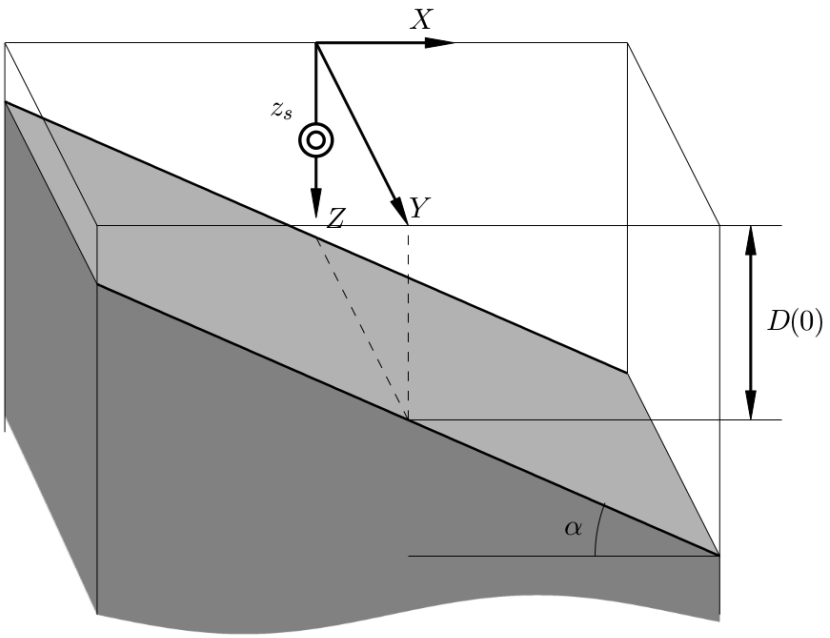


Figure 5.2: Across-slope geometry: α corresponds to the bottom slope, $D(0)$ is the bottom depth at the source position, z_s stands for the source depth (shown as a double circle), the horizontal array is located along the Y axis.

For simulation purposes a scale factor of 1000 : 1 is required to properly account for the

frequencies and lengths of the experimental configuration in the model. Thus, experimental frequencies in kHz become model frequencies in Hz, and experimental lengths in mm become model lengths in m. For instance, an experimental frequency of 150 kHz becomes a model frequency of 150 Hz, and an experimental distance of 10 mm becomes a model distance of 10 m. Sound speed remains unchanged, as well as compressional and shear attenuations.

Validation and performance of predictions of three-dimensional eigenray search were obtained through comparisons against an equivalent (flat) two-dimensional waveguide, and against results presented in [9] and are discussed in Section 5.3; validation and performance of predictions of transmission loss (hereafter TL) were obtained through comparisons against an experimental TL curve from the ASP-H1 subset at frequency of 180.05 kHz, and are discussed in Section 5.4.

5.3 Eigenray predictions

5.3.1 Validation results

Given the "mild" slope of the experimental setup described in the previous section a preliminary set of comparisons was performed between the TRACEO3D and TRACEO models, for a source frequency of 150 Hz. The horizontal array was idealized starting at 0.1 km until 5 km, in increments of 0.1 km. Given the lack of knowledge regarding the source spectrum a *synthetic* five-cycle pulse with a Gaussian envelope was considered as the emitted signal. The received signal was computed using the model output of amplitudes and delays for each receiver range and depth. For Fourier synthesis only frequencies between 100 Hz and 200 Hz were considered, outside the interval the acoustic field was set to zero; the signal in the time domain was calculated using an inverse Fourier transform.

Table 5.2: Geometric parameters used in the numerical predictions for the ASP-H data set.

	z_s [m]	$z_r[m]$	$D(0)[m]$	slope[degrees]
ASP-H1	6.7	11.0	43.9	4.5
ASP-H2	15.0	11.0	43.9	4.5
ASP-H3	27.0	11.0	43.9	4.5

Preliminary TRACEO3D predictions (not shown here) failed to produce satisfactory results using the parameters provided by the refinement discussed in [9]; therefore, alternative geometries were considered. The configuration shown in Table 5.2 was found to best replicate the results presented in Fig.3 from the reference. Three-dimensional predictions, together with equivalent TRACEO calculations for a flat waveguide, are shown in Fig.5.3. Not only the patterns of propagation between the 2D and 3D predictions are strikingly different, but additional inspection of Fig.5.3(d-f) reveals that the set of parameters given by Table 5.2 allows TRACEO3D to predict the features visible in the experimental data, such as the numbers and position of the modes, as well as mode shadow zones, intra-mode interference and mode arrivals. The only exception was the ASP-H3 data set; it is believed that most discrepancies are due to the proximity of the source to the bottom for the corresponding geometry, for which beam displacement corrections can be relevant [1].

As suggested in [7, 8, 23] the 3D effects can be explained based on ray/mode analogies. A mode can be considered as a standing wave in the vertical plane, and as a traveling wave describing a hyperbolic path on the horizontal plane, with the ray initially propagating itself upslope; at some point in the range the hyperbolic path crosses the across-slope direction; this analogy is fundamental for the discussion that follows. Predictions of normalized amplitudes for 2D and 3D calculations, regarding the ASP-H1 configuration, are shown in Fig.5.4. The

3D results in the figure also reveal modes in the (θ, ϕ) plane, allowing to determine take-off angles for different modes. The dashed lines approximately represent the edges of the shadow zones for each mode, with each shadow zone being a complex function of different parameters, such as frequency, wedge slope and bottom properties. The across-slope direction where the source is aligned with the synthetic horizontal array is taken as $\phi = 0$; this angle increases towards the wedge apex.

The waveforms presented in Fig.5.3(a) correspond to 2D predictions for the ASP-H1 configuration, with a source depth of 6.7 m. At short ranges the predicted time signals seem to merge together. Above a certain range they start to separate, increasing the relative time delay between them as the receiver moves away from the source. As a receiver approaches the range of 5 km late arrivals progressively lose more energy. Similar patterns can be seen in the other two configurations (see Fig.5.3(b-c)). The ASP-H1 2D prediction is further supported by Fig.5.4(a-e), in which the behavior of amplitudes over range exhibit a typical distribution for a flat waveguide: amplitudes can be seen to decrease steadily over elevations θ, while the number of eigenrays increase with range. Such steady decay can be explained by taking into account that 2D eigenrays are confined exclusively to the vertical plane, and thus often bounce off the bottom losing more and more energy as elevation and range increase. A completely different pattern can be seen in Fig.5.3(d), in which the waveforms were calculated accounting for full 3D effects. The figure shows an interesting pattern of mode arrivals: above 2 km the modes M1 and M2 exhibit well resolved first and second arrivals, and the time delay between them decreases as the receiver moves away from the source; near 2 km the expected first and second arrivals from mode M3 merge together, and the mode quickly vanishes due to the transition of M3 into a shadow zone; additionally, as

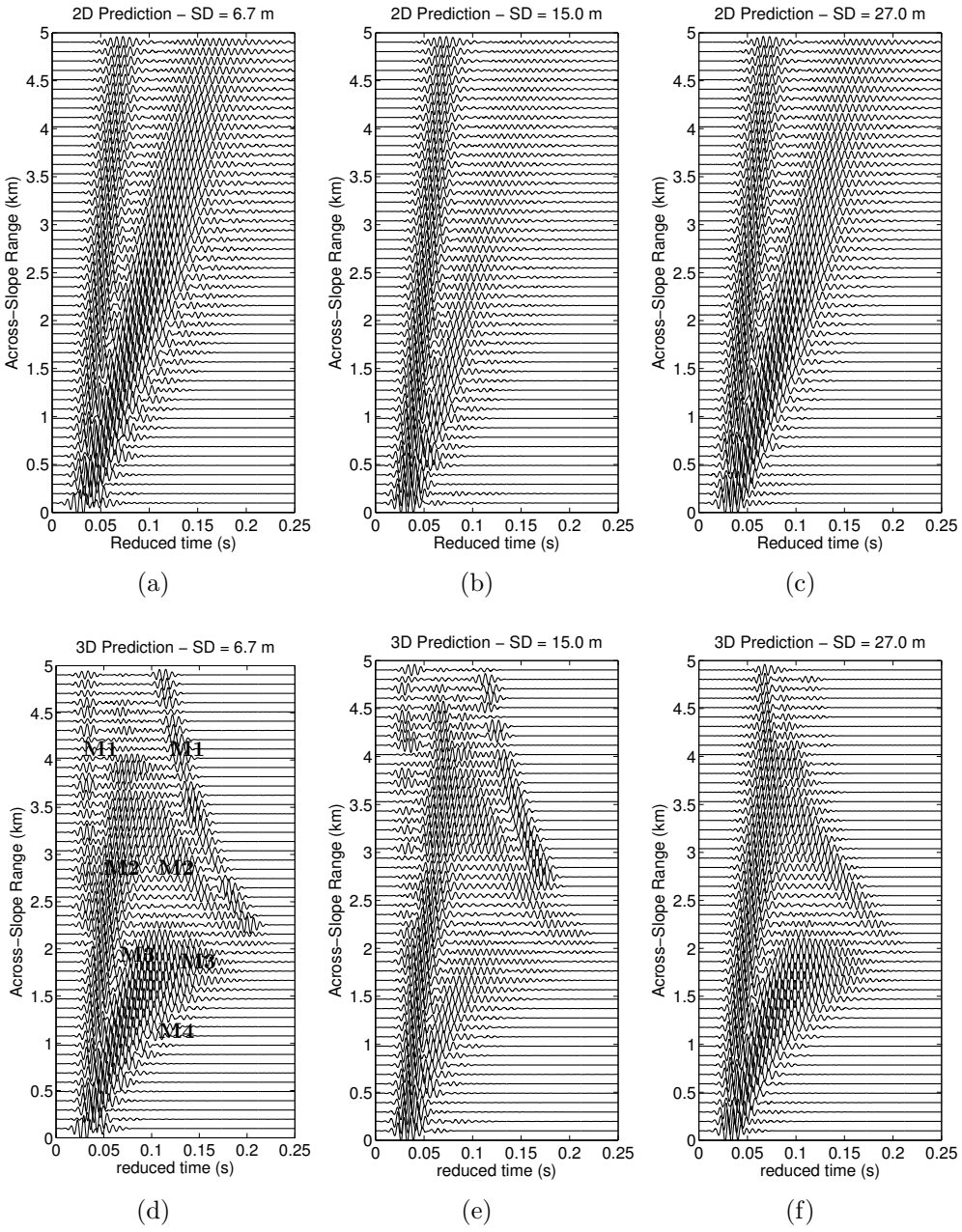


Figure 5.3: Numerical simulations calculated with TRACEO (top) and TRACEO3D (bottom) for the geometry presented in Table 5.2; four modes can be identified regarding 3D predictions for the ASP-H1 configuration.

range decreases below 2 km, modal refraction on the horizontal plane is such that the mode M4 becomes well resolved in time, but exhibiting only a single arrival.

Similar modal patterns can be seen in Fig.5.3(e-f). All mentioned features can be explained in more detail in Fig.5.4(f-j), which shows that higher order modes are more intensively refracted at short ranges due to their large initial elevation θ; such modes rapidly bounce off the bottom at the critical angle and thus vanish (i.e. enter into a shadow zone) after being absorbed. Low order modes, on the other hand, are able to produce first and second arrivals at larger ranges due to an interesting combination of propagation conditions: for a single "small" elevation θ one can find a pair of azimuths ϕ_1 and ϕ_2 (with $\phi_1 < \phi_2$), in which the ray with take-off angles (θ, ϕ_2) propagates over shallower regions, but bounces more often off the bottom than the ray propagating with angles (θ, ϕ_1), and therefore leaks energy more rapidly. Thus, the entire 3D set of eigenray, travel time and amplitude calculations allows for the establishment of a remarkable connection between eigenray azimuth/elevation (θ, ϕ), mode order n and receiver range r, with the parameters (θ, ϕ, n) increasing simultaneously as r decreases. These general conclusions, based mostly on ray theory, coincide with the discussion presented in [66]. Obviously there are some amplitude discrepancies between the results shown in Fig.5.3(d-f) and those presented in Fig. 3 from [9]; the discrepancies were in fact expected. During the calculations of arrival patterns different synthetic pulses were considered, besides the Gaussian one; it was found that the structure of propagating modes was highly sensitive to the particular choice of emitted signal. Such sensitivity can perhaps explain the usage in [9] of the *recorded* transmitted signal, instead of the *synthetic* one, to predict the arrival patterns. A final insight into the problem can be found in the comparison of eigenrays, calculated with TRACEO for the flat case, and calculated with TRACEO3D

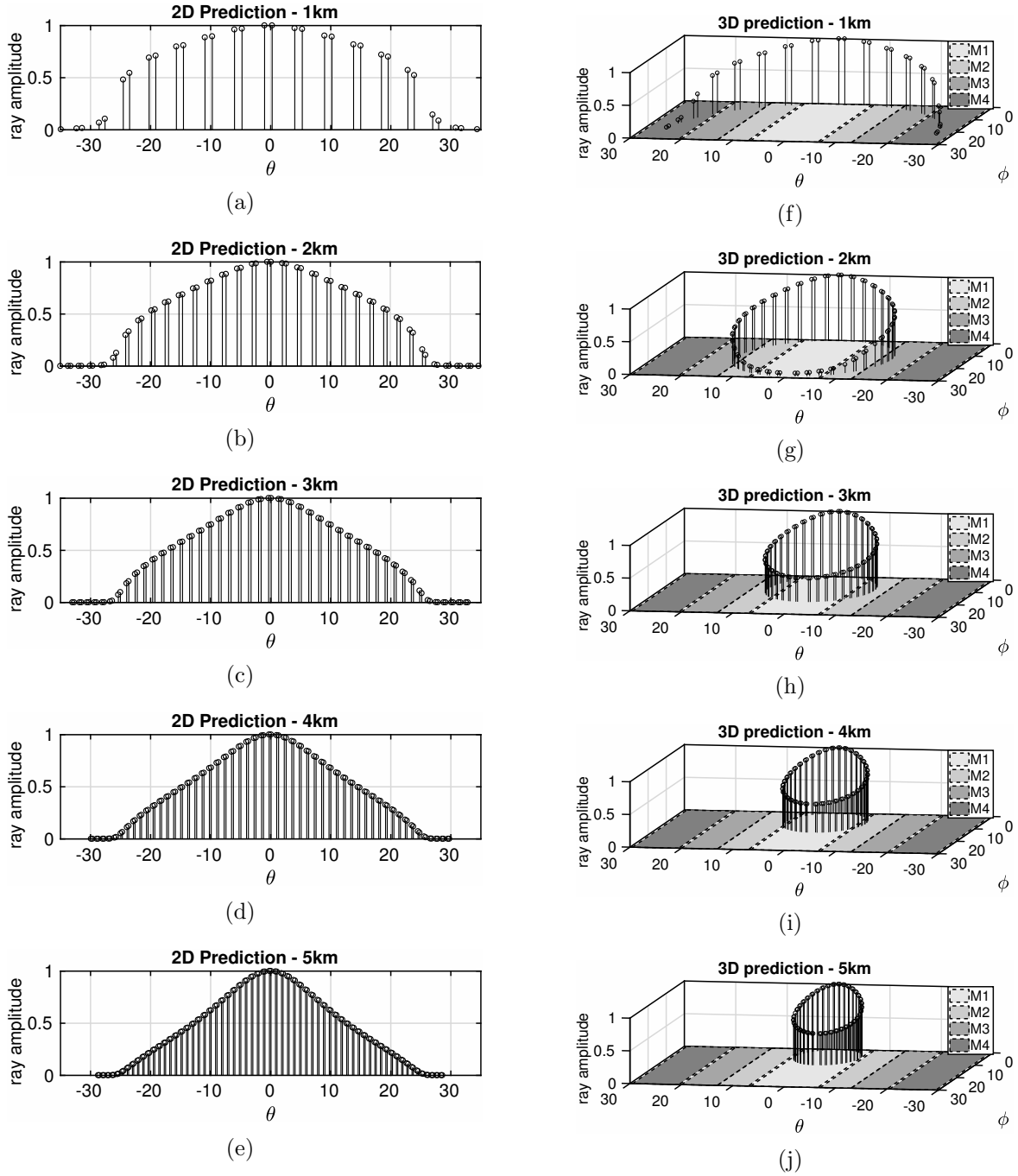


Figure 5.4: Predictions of normalized amplitudes versus launching angles for the ASP-H1 configuration over range: TRACEO (left); TRACEO3D (right). The corresponding regions where modes can exist are indicated over the (θ, ϕ) plane. The dashed lines stand roughly for the critical launching angle.

for the wedge waveguide (see Fig.5.5). At a first glance there seems to be a perfect one-to-one correspondence of eigenrays in terms of elevations θ, and thus one could expect both 2D and 3D amplitudes to exhibit a similar correspondence. In fact that is not the case; in the wedge waveguide most eigenrays propagating up then down slope are bouncing on regions where bottom depth is smaller than the one of the 2D waveguide; as a consequence, instead of spreading progressively over elevations as shown in Fig.5.4(b), the amplitudes of arrivals become clustered between the limits of an elevation interval, as shown in Fig.5.4(g).

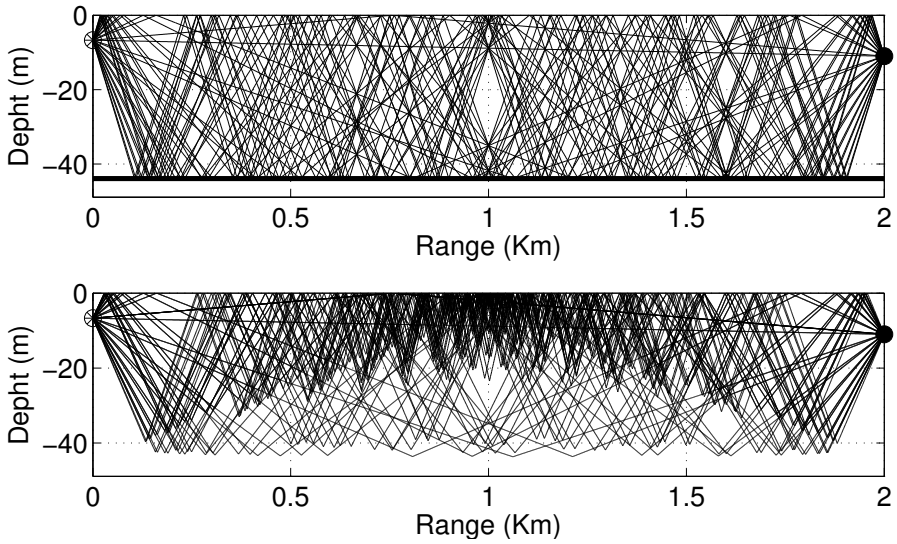


Figure 5.5: Eigenray predictions for the ASP-H1 configuration: TRACEO, flat waveguide (top); TRACEO3D, across-slope propagation on the wedge waveguide (bottom). Source and receiver depth corresponds to 6.7 m and 11.0 m, respectively.

5.3.2 Performance analysis

To properly address the performance analysis of eigenray search an optimization step is required, based on the device resources, looking to define the best parameters of kernel execution configuration, together with the number of registers per thread, which influence the SM occupancy rate. Thus, model runs using the parallel eigenray search algorithm for the predictions of Fig.5.3(d) were performed with block sizes from 32 to 1024, and number

of registers from 32 to 256, increasing each as a factor of the warp size and power of two, respectively. Runtime results for such combinations are presented in Fig.5.6, and show that the performance improves for high numbers of registers per thread, which means low occupancy. In general, better results occurred for an occupancy rate lower than 25% (up to 128 registers per thread), and the best result was achieved with $p = 64$, using 255 registers per thread which means 12.5% of occupancy.

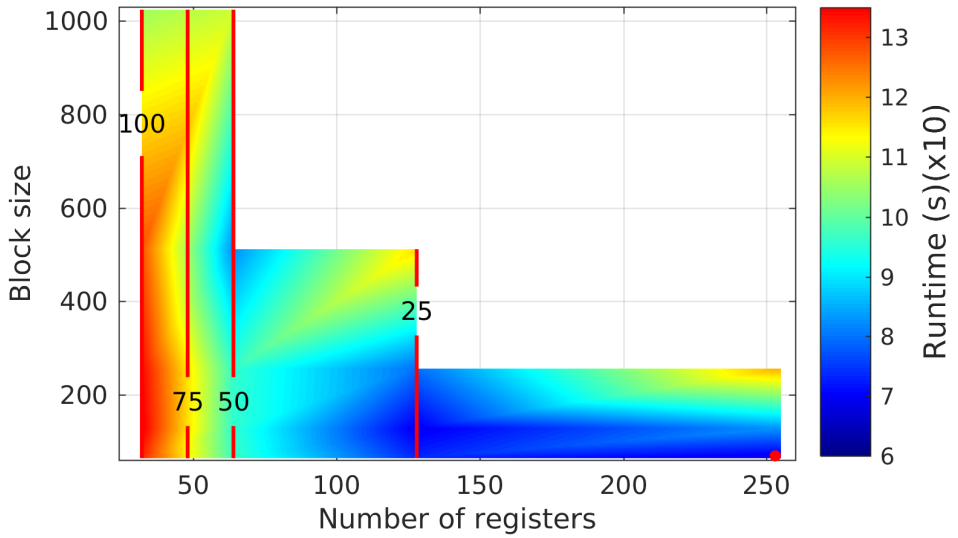


Figure 5.6: Execution configuration results for different block sizes p and number of registers per thread; vertical lines stands for the occupancy rate (%) in each SM. The best option (red dot) corresponds to $p = 64$ with 255 registers per thread; areas with no data represent parameter combinations that the device can not handle due to lack of resources.

One can therefore conclude that performance can increases as threads individually have more registers available, even when less threads share the SM simultaneously. It is important to remark that runtime interpolated results provided an important guide regarding the configuration of parameters, despite the specific choice of p multiples of warp size. Additionally, it was also found that when using the occupancy-based launch configurator of the application programming interface (API) [54], which *heuristically* calculates the block size, runtime results *increased* around 70%; such unexpected result is believed to be due to the

fact that the API strives mostly to achieve high occupancy, which not always guarantees the highest performance in every case.

Table 5.3: Results of runtime and speedup ratio regarding predictions of the LMA CNRS H1 @ 150 Hz in the time domain.

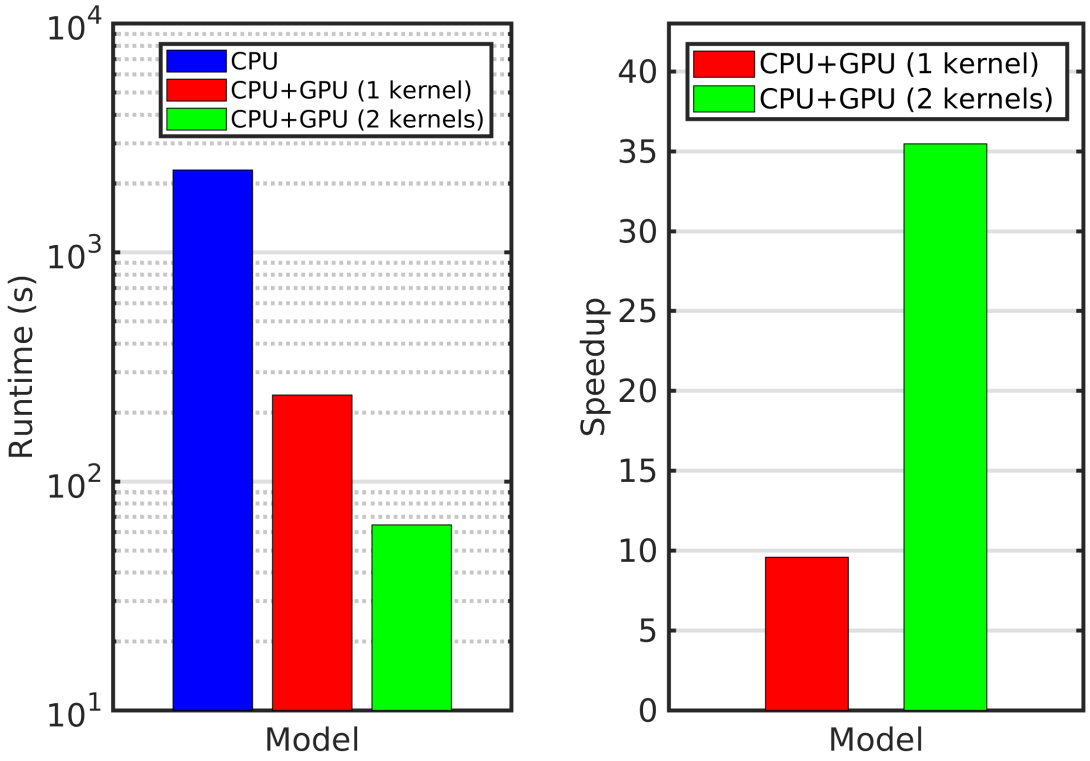
Model	CPU	CPU + GPU (1 kernel)	CPU + GPU (2 kernels)
Runtime (s)	2287.8	238.22	64.48
Speedup ratio	1	9.6	35.47

The best results during the execution of the configuration optimization are presented in Table 5.3 and Fig.5.7, where CPU corresponds to the sequential algorithm, while CPU + GPU (1 kernel) and CPU + GPU (2 kernels) correspond to the two different parallel implementations, discussed in Section 4.2.3. The results show that the parallel implementation was over 35 times faster than the sequential one, reducing the runtime from 2,287.8 s to 64.48 s. The mean square error (MSE) between the sequential and parallel implementations is presented in Fig.5.8. One can see that the difference between the values is lower than 10^{-4}. The parallel implementation only achieves such accuracy by using IEEE 754 compatible mathematical functions [67], and compiling without the flag *–fast-math*; it is believed that this flag enables performance optimization at the cost of introducing some numerical inaccuracies. Comparisons using the proximity method are not presented because the method failed to provide 3D predictions.

5.4 Numerical predictions of transmission loss

5.4.1 Comparisons with experimental data

The set of waveguide parameters provided by the tank scale experiment was also used to calculate predictions in the frequency domain. TL results are presented in Fig.5.9, where



(a) (b)

Figure 5.7: (a) Runtime and (b) speedup of model predictions using the 3D eigenray search algorithm for the tank scale experiment.

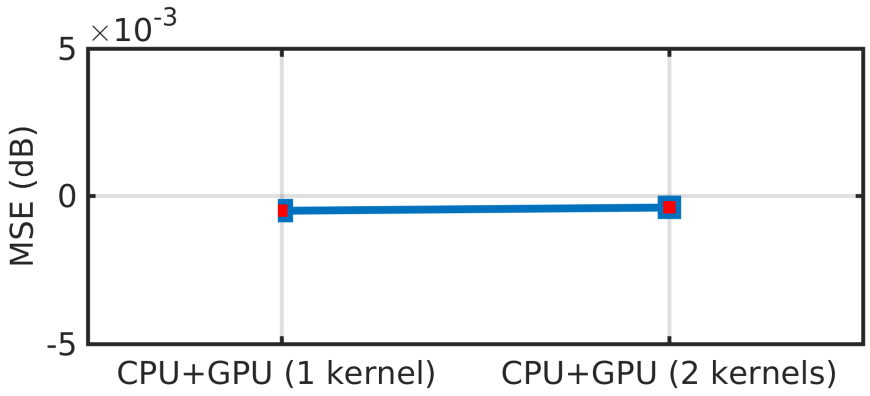


Figure 5.8: MSE of TRACEO3D predictions against parallel implementations.

Bisection means the original algorithm that the sequential version of TRACEO3D uses to calculate ray influence, *Grid* stands for the sequential method presented in Section 3.2, and *GPU Grid* corresponds to the parallel implementation. In general, model predictions were

able to follow accurately the experimental curve over the full across-slope range. Nevertheless, a slight shift in phase can be observed at 2 Km and 2.4 Km in all simulation predictions. Besides, minor discrepancies can be noted between the predictions at the far field.

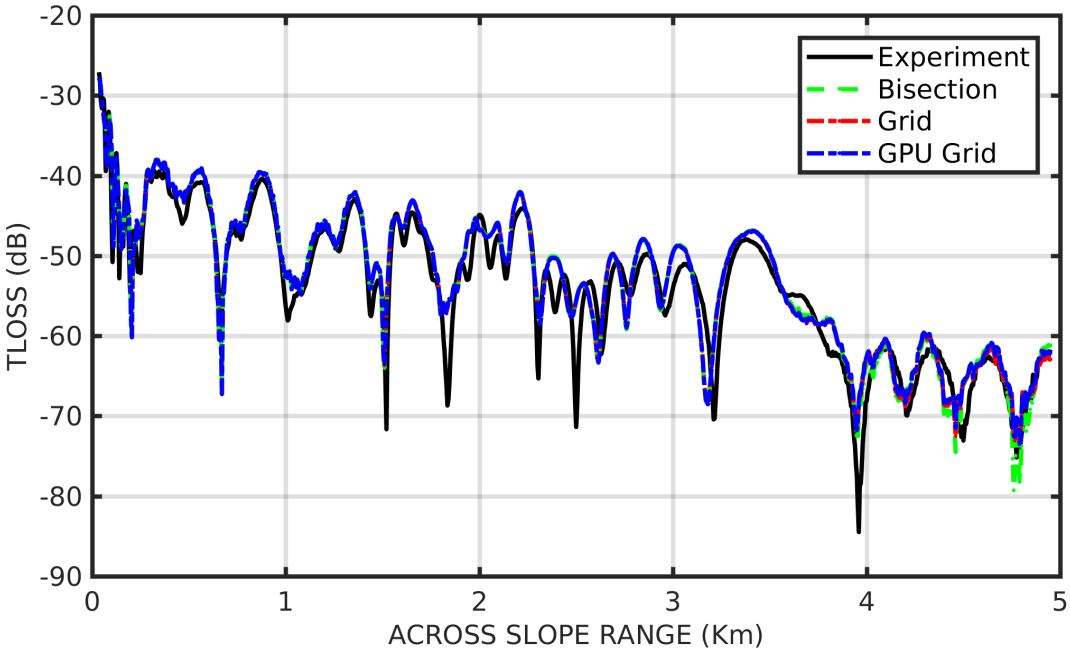


Figure 5.9: Comparisons with the experimental data for LMA CNRS H1 @ 180.05 Hz.

5.4.2 Performance analysis

As previously, the algorithm of parallel field calculation required an optimization step in order to define the best parameters of the kernel execution configuration and the number of registers per thread. Thus, model runs generating the predictions presented in Fig.5.9 were performed with block sizes from 32 to 1024, and number of registers from 32 to 256, increasing as a factor of the warp size and power of two, respectively. Runtime results for such combinations are presented in Fig.5.10, which shows that the performance improves for number of registers per thread between 64 and 128, which means the occupancy rate is between 50% and 25%. The best result was achieved with $p = 64$ and using 64 registers per

thread, which means 50% of occupancy.

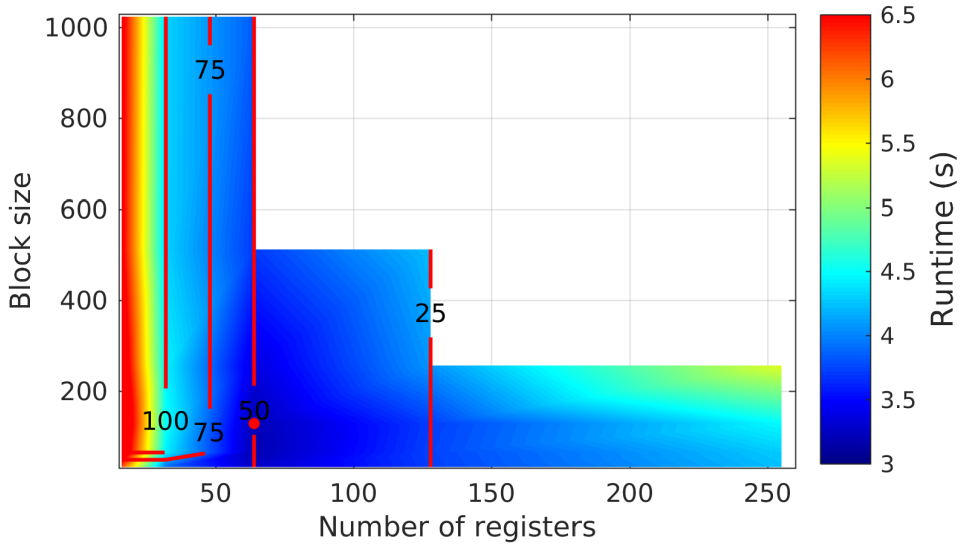


Figure 5.10: Execution configuration results for different block sizes p and number of registers per thread; vertical lines stands for the occupancy rate (%) in each SM. The best option (red dot) corresponds to $p = 64$, with 64 registers per thread; areas with no data represent parameter combinations that the device can not handle due to lack of resources.

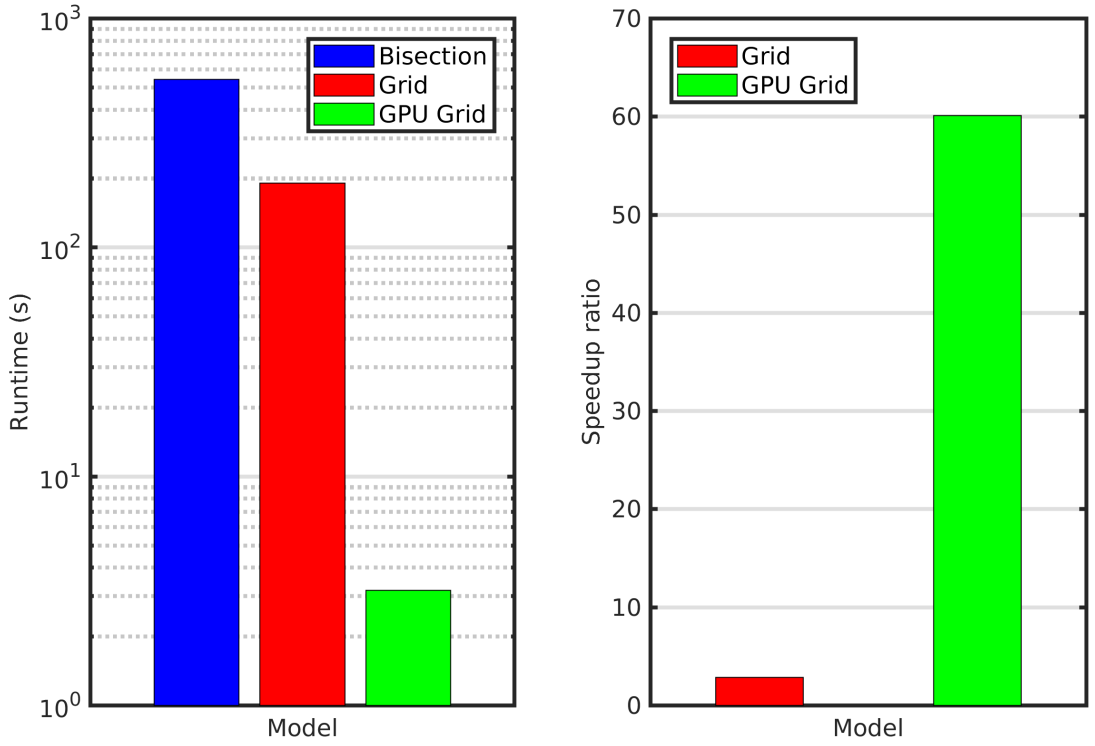
Again, as discussed in Section 5.3.2, runtime interpolated results are valid only for values of p multiples of warp size. The occupancy-based API delivered an occupancy of 100% with $p = 256$ and 32 registers per thread, which represented an increase of 90% in the runtime.

Table 5.4: Results of runtime and speedup ratio for TL predictions.

Model	CPU (Bisection)	CPU (Grid)	CPU + GPU (Grid)
Runtime (s)	542.3	191.16	3.18
Speedup ratio	1	2.83	60.11

The best result found during the execution configuration optimization is presented on both Table 5.4 and Fig.5.11. Different from the comparison shown in Section 5.3.2, speedup rates are presented separately, comparing the improvement regarding the numerical enhancement and the improvement achieved with the parallel GPU implementation. Thus, the speedup ratio of CPU (Grid) is calculated dividing the Bisection runtime by the Grid run-

time, and for the CPU + GPU (Grid) dividing the Grid runtime by the GPU runtime. It is important to remark that the CPU (Grid) was able to decrease the runtime in 2.83 times, while the parallel GPU implementation achieved 60 times of performance, which indeed represents a outstanding improvement. Combining both speedups the total improvement was about 170 times (2.83×60.11), reducing the runtime **from 542.3 s to 3.18 s**. The mean square error (MSE) between the model implementations and the experimental data is shown in Fig.5.12. One can see that the difference among the implementations are of the same order of magnitude. As already indicated in Section 5.3.2 the GPU achieves such accuracy using IEEE 754 compatible mathematical functions, and compiling without the flag *–fast-math*.



(a) (b)

Figure 5.11: (a) Runtime and (b) speedup for TL predictions of the tank scale experiment.

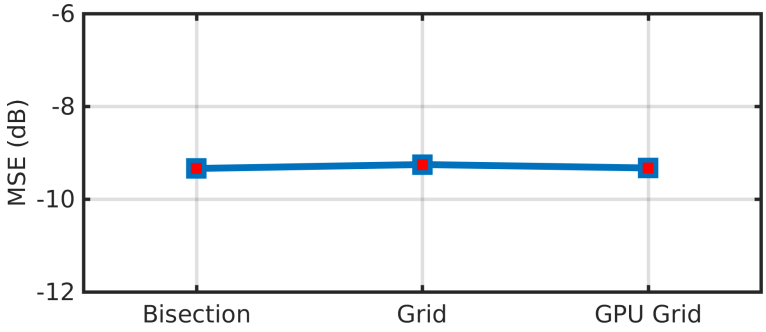


Figure 5.12: MSE of TRACEO3D predictions against experimental data (LMA CNRS H1 @ 180.05 Hz) using three different approaches: Bisection, Grid and GPU Grid.

5.4.3 Comparisons with an analytical solution

The analytical solution for sound propagation in a 3D penetrable wedge discussed in [10] represented also an important reference for additional model predictions [11]. The solution is inspired by the image method presented in [68], in which the contribution of each image is represented in terms of a Bessel function expansion inside a certain improper integral. For small wedge angles acoustic propagation can be considered adiabatic; in the case of constant sound speed the corresponding expression for the acoustic field can be written compactly using the wavenumbers of the Pekeris problem, calculated at the position of the acoustic source. Again a geometry of across-slope wedge propagation was considered, similar to the one shown in Fig.5.2, but considering a wedge angle $\alpha = 0.5°$. Waveguide parameters and corresponding values are summarized in Table 5.5; the parameters for the non-adiabatic case correspond to the well-known 3D ASA wedge benchmark [69]. A rectangular array (RA) was considered for predictions; the RA was aligned along the Y axis, with $X = 0$, and was composed of 44 receivers in depth from $z_r = 1$m until $D(0) - 1$, and 501 receivers in range, starting from $Y = 35$m until $Y = 5000$m, providing a mesh of 44×501 receivers equally spaced in range and depth; source frequency corresponded to 122 Hz. The results are shown

in Fig.5.13 and show that the predictions replicate the elaborate pattern of interference of the analytical solution as range increases. Some discrepancies can be seen between transition areas, but they were in fact expected because the low value of frequency is on the edge of validity of ray theory. Even so, the main goal of the comparisons was to demonstrate the ability of the finite beam width strategy to calculate TL fields preserving the accuracy, while decreasing significantly runtime. Additionally, model predictions can be provided for any type of waveguide, while the analytical solution is valid only in wedges with small slopes.

Table 5.5: Wedge parameters and corresponding notation.

Parameter	Symbol	RA	Units
Bottom slope	α	0.5	degrees
Source frequency	f	122	Hz
Depth at source position	$D(0)$	44.4	m
Source depth	z_s	8.3	m
Maximal range	R	5000	m
Water sound speed	c_w	1500	m/s
Bottom compressional speed	c_b	2000	m/s
Bottom compressional density	ρ_b	2	g/cm^3
Bottom compressional attenuation	α_b	0.5	dB

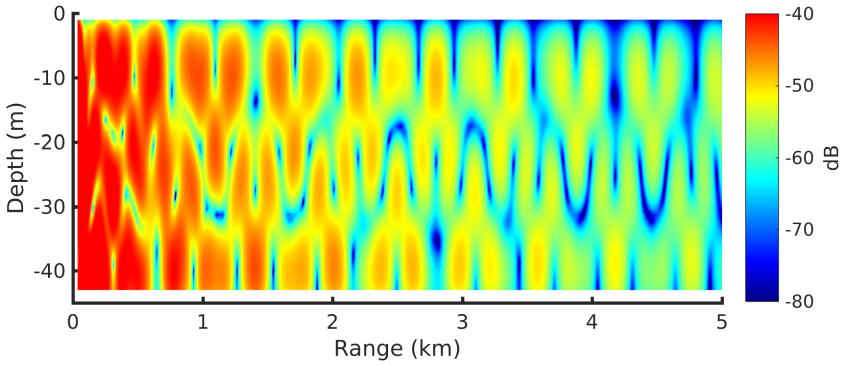
5.4.4 Performance analysis

Since the tunning procedure was already achieved for the kernel of parallel field calculation (as described in Sec. 5.4.2), the same execution configuration parameters were used to generate the predictions of Fig.5.13. The performance analysis regarding such calculations is presented in both Table 5.6 and Fig.5.15, which shows that the CPU (Grid) was able to decrease the runtime in 32.76 times, while the parallel GPU implementation performed 21 times better. Note that the speedup ratio was calculated as explained in Section 5.4.2. Combining both speedups the total improvement was about 692 times (32.76×21.13),

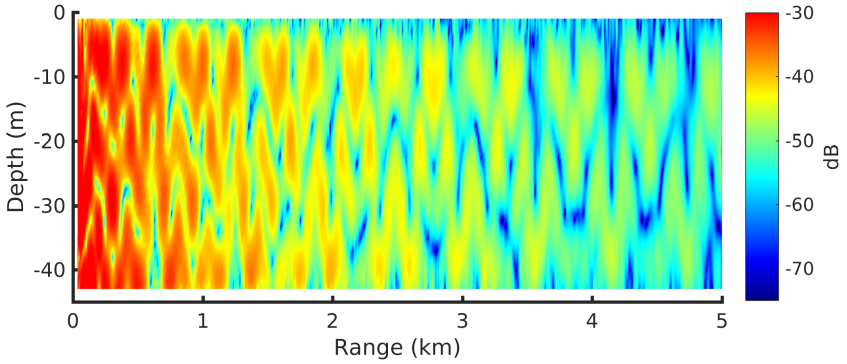
reducing the runtime **from 18,279.1 s to 26.40 s**, which indeed represents a remarkable improvement. The difference between these results and the ones presented in Section 5.4.2 can be explained by keeping in mind that the receiver grid strategy becomes more efficient as the number of sensors increases, while the opposite happens for the bisection algorithm. Besides, memory requirements for these computations are too large to fit into the GPU at once. Thus, the calculation was divided into a serial loop for calling kernels and execute memory transfers between the host and the device, which decreases the GPU performance. The MSE between the sequential and parallel model implementations and the experimental data is shown in Fig.5.12. One can see that the differences between the implementations are almost of the same order of magnitude, except for the Grid result, in which the divergence was about 3 dB. As discussed in Section 5.3.2 the GPU achieved such accuracy using IEEE 754 compatible mathematical functions, and compiling again without the flag *–fast-math*.

Table 5.6: Runtime and speedup ratio regarding the calculations of TRACEO3D predictions of wedge problem @ 122 Hz using different methods.

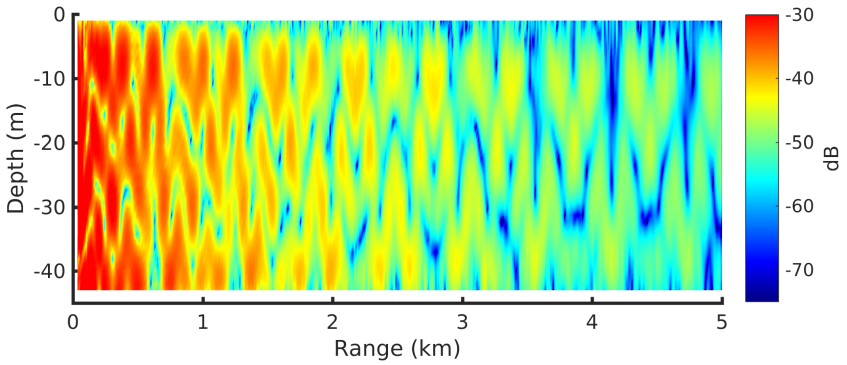
Model	Analytical Solution	CPU (Bisection)	CPU (Grid)	CPU + GPU (Grid)
Runtime (s)	2.54	18,279.1	557.87	26.40
Speedup ratio	–	1	32.76	21.13



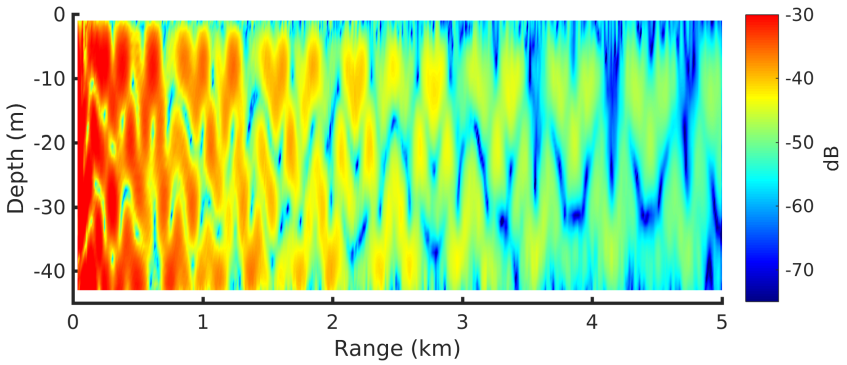
(a) Analytic Solution



(b) CPU Bisection



(c) CPU Grid



(d) GPU Grid

Figure 5.13: Adiabatic wedge: TL results.

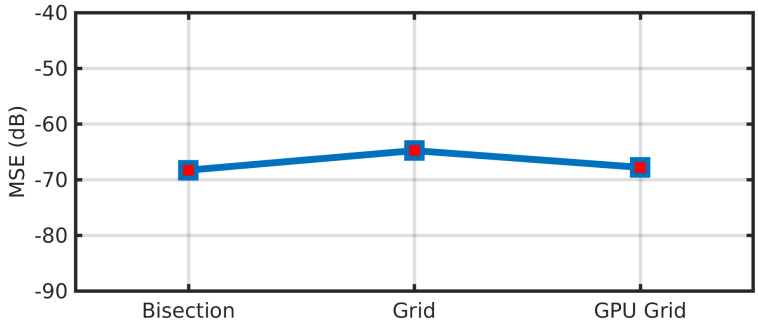
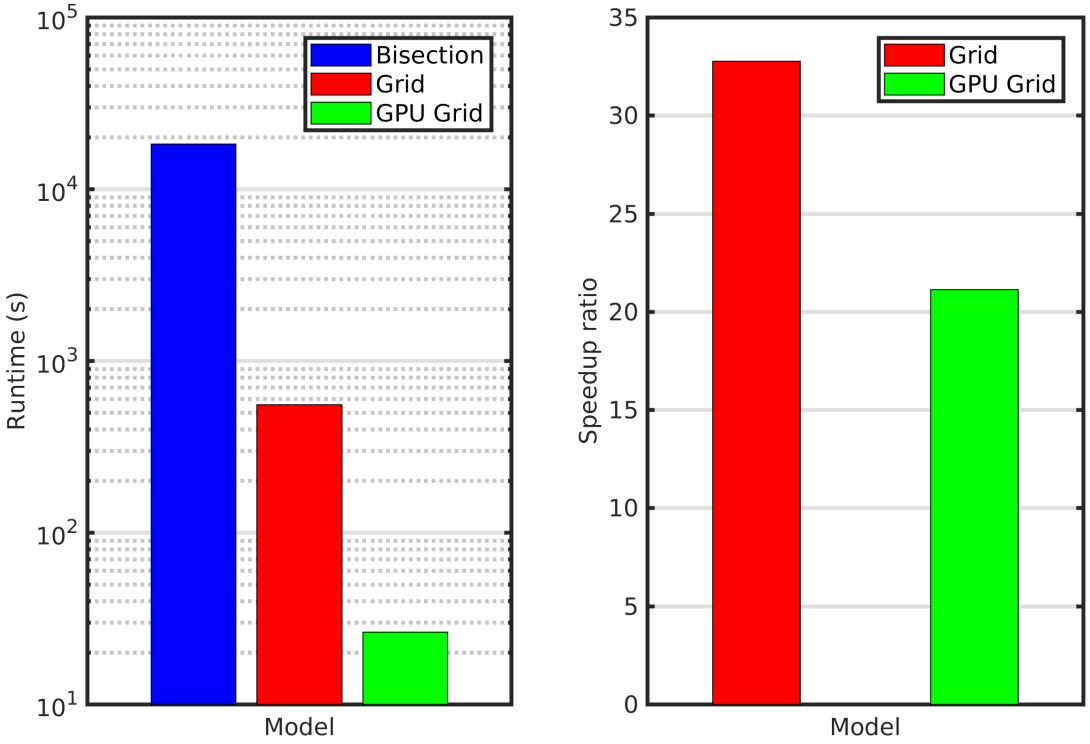


Figure 5.14: MSE of TRACEO3D predictions of the analytic solution of the wedge problem using three different approaches: bisection, Grid and GPU Grid.



(a) Runtime

(b) Speedup

Figure 5.15: Runtime and speedup for TL model predictions of the wedge problem.

Chapter 6

Conclusions

Synopsis: *This chapter presents an overview of the research developed within the framework of the thesis, including published results, and suggestions for future work. Concluding remarks are presented in Section 6.1, while publications are listed in Section 6.2; Section 6.3 discusses future directions of research.*

6.1 Concluding remarks

The theoretical background and numerical issues of the TRACEO3D Gaussian beam model were discussed in detail in Chapter 2 in order to establish a firm basis for additional issues regarding model enhancements, parallelization and validation. Such discussion allowed to conclude that the main task of TRACEO3D is to keep the calculation of ray trajectories as accurate as possible, to which end a high order integrator was used; that task was considered an immutable clause for further optimization and parallelization. In the original version of TRACEO3D the eigenray search was based on the method of proximity (see Section 3.1), which was found to be computationally demanding and inefficient. Additionally, calculations of ray influence were found to be accurate, but time consuming, with runtime increasing drastically as range, number of rays and number of sensor increased (see Section

2.3.5). Both issues were analyzed carefully before the development of the parallel algorithms (in which optimization strives mainly to deal with data bottlenecks) by developing new methods that parallel computing was not able to overcome. In this context the Simplex method to find 3D eigenrays was implemented in TRACEO3D, and the corresponding validation was carried out against predictions from the TRACEO 2D model, and against results from a tank scale experiment. The 3D predictions exhibited a remarkable similarity with most experimental features, replicating mode shadow zones, intra-mode interference, and mode arrivals; important connections in the ray/mode equivalence framework were noticed. TRACEO predictions, unsurprisingly, were found to be valid only close to the source. Simplex-based eigenray search allows an efficient and accurate calculation of 3D eigenrays by determining values of the corresponding take-off angles, which lead to the shooting of rays passing as close as desired to the position of a given receiver after multiple (and complex) boundary reflections. Minor discrepancies found in the comparisons against experimental results are believed to be related to beam displacement and/or signal processing issues, and to ray theory being applied on the edge of its validity. Yet such discrepancies are completely independent of the proposed method of eigenray search, which was found to be extremely efficient and robust.

The calculation of ray influence was addressed using a receiver grid, i.e. a subset of adjacent receivers within the array, with the goal of decreasing runtime while keeping accuracy. The validation results were performed using experimental data collected from a tank scale experiment, and against simulated results from an analytical solution for sound propagation in a 3D penetrable wedge. The method was able to achieve the same precision as the original (and much slower) version of TRACEO3D using bisection. Besides, the method was found

to be computationally efficient even when dealing with arrays containing a large number of sensors, although some optimization was required in order to define the proper borders of ray influence given by the finite beam width.

After the enhancement of numerical issues parallel algorithms were developed considering a GPU architecture, that could take advantage of the inherent ray tracing parallelism and the high workload of 3D propagation, keeping in mind that the memory access pattern was a serious drawback to consider. The parallelism was based on the natural ray tracing organization, addressing each pair of launching angles (θ, ϕ) as a single parallel thread. A detailed description of the parallel algorithms for 3D eigenrays search and ray influence calculation was presented in Chapter 4. Besides, a device memory organization was also proposed, which allowed for the improvement of performance in a non-classical fashion by requiring low occupancy and high register use per thread. For each validation result a performance analysis was carried out looking to optimize the execution configuration parameters and number of registers per thread; this optimization procedure delivered the double of performance at the final speedup. Considering the 3D eigenray search, the parallel implementation was 35 times faster than the sequential version, reducing runtime from 2,287.8 s to 64.48 s. Performance comparisons with the numerical enhancement for eigenray calculations were not shown because the proximity method failed to provide 3D predictions. Regarding TL calculations and comparisons with experimental data the enhanced implementation was 2.83 times faster than the bisection method, while the parallel implementation was 60.11 times faster than the sequential one. Combining both speedups the improvement was around 170 times faster (2.83×60.11), reducing the runtime from 542.3 s to 3.18 s. Considering the comparison with the analytical solution the enhanced implementation was 32.76 times faster

than the bisection method, while the parallel implementation was 21.13 times faster than the sequential one. Combining both speedups the improvement was around 692 times faster (32.76×21.13), reducing the runtime from 18,279.1 s to 26.40 s. In general, parallelization does not degrade the accuracy as long as compatible IEEE 754 mathematical functions are used, and as long as one avoids using CUDA compilation flags for optimization. Despite the significant improvements in speedup it can be not guaranteed that the adopted parallel algorithms exhausted all solutions of parallelization. It is believed that additional combinations of thread granularities and memory organization can have the potential to achieve a greater performance, but the exploration of such cases was beyond the original goal of this work. The speedup issue is certainly related to the requirements needed to use a 3D model, a topic which is currently under intense discussion. This thesis stands on the firm conviction that the contributions and remarkable reduction in runtime achieved will certainly help to overcome some of the reserves in employing a 3D model for predictions of acoustic fields.

6.2 Contributions

The contributions of this work can be summarized as follows :

1. Development of a solution for the calculation of three-dimensional (3D) eigenrays based on Simplex optimization. It was found that the search strategy based on Simplex optimization was able to calculate 3D eigenrays efficiently and accurately for a wedge waveguide, thus providing predictions of arrival patterns along cross-slope range, which replicated elaborate patterns of mode shadow zones, intra-mode interference, and mode arrivals.

2. Development of an strategy for ray influence calculations, by relying on a grid of

receivers, that were updated along a ray trajectory. The method was found to be computationally efficient when dealing with arrays with a large number of receivers.

3. Development of GPU-based parallel algorithms for the TRACEO3D model with validation for 3D eigenray search and ray influence, showing significant improvements between the sequential and parallelized versions of the model. The parallel code will be made available to allow other researchers to carry out 3D calculations, or to be used as a reference for code parallelization.

All contributions were presented progressively in the following publications [22, 24, 50, 59, 70, 71]:

1. R.M. Calazan and O.C. Rodríguez, "TRACEO3D Ray Tracing Model and its Parallel Implementation", *Poster in Ciência 2016, Lisboa, Portugal*, July 2016.

2. R.M. Calazan and O.C. Rodríguez, "TRACEO3D ray tracing model for underwater noise predictions", in *Doctoral Conference on Computing, Electrical and Industrial Systems*, pp. 183–190, Springer, 2017.

3. R.M. Calazan and O.C. Rodríguez, and N. Nedjah, "Parallel ray tracing for underwater acoustic predictions", in *International Conference on Computational Science and Its Applications*, pp. 43–55, Springer, 2017.

4. R.M. Calazan and O.C. Rodríguez, "Three-dimensional eigenray search for vertical line array", in *UACE2017 - 4th Underwater Acoustics Conference and Exhibition*, pp. 941–946, UACE Proceedings, 2017.

5. R.M. Calazan and O.C. Rodríguez, "Simplex based three-dimensional eigenray search

for underwater predictions", *The Journal of the Acoustical Society of America*, vol. 143, no. 4, pp. 2059–2065, 2018.

6. R.M. Calazan and O.C. Rodríguez, "GPU-Based 3D eigenrays search for underwater acoustics predictions", *Poster in Ciência 2018, Lisboa, Portugal*, July 2018.

6.3 Future work

Future directions of research can be described as follows:

- Further validation looking to assess the model's performance and accuracy, through the calculation of 3D eigenrays and ray influence in typical ocean environments with complex bathymetries like sea canyons, or complex sound speed fields like the one produced by an upwelling regime. It is believed that such complex waveguide features will require the development of smoothing criteria in order to handle eventual erratic behaviors of propagating rays.

- Incorporation of additional theoretical methods into the TRACEO3D model in order to improve its accuracy at low frequencies.

- Addition of code to allow TRACEO3D to read tabulated reflection coefficients.

- Tests with different thread granularities, requiring new memory organization and execution configuration parameters; within this context the following issues are to be considered:

  - Take advantage, when possible, of GPU hardware filtering for 2D/3D interpolation using texture memory for boundary intersections and sound speed calcula-

tions.

  - Improve the copy *host* - *device* using asynchronous transfers during the kernel computation; this improvement can be important for calculations when the device memory is not sufficient to store the data, as discussed for the RA TL results (see Section 5.4.4).

- Improve the OpenMPI version to calculate 3D eigenrays exploiting multiple CPU cores in super computers (clusters, for instance) and scaling the parallel model in multiple GPU nodes.

- Unlike the sequential version of TRACEO3D the parallelized version of the model lacks the code to support calculations of particle velocity. Thus, a code update to fix this issue can be expected in the future.

Bibliography

[1] Paul C Etter. *Underwater Acoustic Modeling and Simulation*. CRC Press, 4th edition, 2013.

[2] Finn B Jensen, William A Kuperman, Michael B Porter, and Henrik Schmidt. *Computational Ocean Acoustics*. Springer Science & Business Media, New York, 2th edition, 2011.

[3] A Tolstoy. 3-D propagation issues and models. *Journal of Computational Acoustics*, 4(03):243–271, 1996.

[4] Lussac P Maia, António Silva, and Sérgio M Jesus. Environmental model-based time-reversal underwater communications. *IEEE Access*, 6:10041–10051, 2018.

[5] Martin Gassmann, Sean M Wiggins, and John A Hildebrand. Three-dimensional tracking of cuvier's beaked whales' echolocation sounds using nested hydrophone arrays. *The Journal of the Acoustical Society of America*, 138(4):2483–2494, 2015.

[6] DE Weston. Horizontal refraction in a three-dimensional medium of variable stratification. *Proceedings of the Physical Society*, 78(1):46, 1961.

[7] Henry Weinberg and Robert Burridge. Horizontal ray theory for ocean acoustics. *The Journal of the Acoustical Society of America*, 55(1):63–79, 1974.

[8] Chris H Harrison. Acoustic shadow zones in the horizontal plane. *The Journal of the Acoustical Society of America*, 65(1):56–61, 1979.

[9] Frédéric Sturm and Alexios Korakas. Comparisons of laboratory scale measurements of three-dimensional acoustic propagation with solutions by a parabolic equation model. *The Journal of the Acoustical Society of America*, 133(1):108–118, 2013.

[10] Pavel S. Petrov and Frédéric Sturm. An explicit analytical solution for sound propagation in a three-dimensional penetrable wedge with small apex angle. *The Journal of the Acoustical Society of America*, 139(3):1343–1352, 2016.

[11] Orlando C. Rodriguez, Frédéric Sturm, Pavel Petrov, and Michael Porter. Three-dimensional model benchmarking for cross-slope wedge propagation. In *Proceedings of Meetings on Acoustics*, volume 30, page 070004, 25–29 June, Boston, MA, 2017. ASA.

[12] Frederic Sturm. Numerical study of broadband sound pulse propagation in three-dimensional oceanic waveguides. *The Journal of the Acoustical Society of America*, 117(3):1058–1079, 2005.

[13] Pavel S Petrov and Tatyana N Petrova. Asymptotic solution for the problem of sound propagation in a sea with an underwater canyon. *The Journal of the Acoustical Society of America*, 136(4):EL281–EL287, 2014.

[14] Ying-Tsong Lin, Timothy F Duda, Chris Emerson, Glen Gawarkiewicz, Arthur E Newhall, Brian Calder, James F Lynch, Philip Abbot, Yiing-Jang Yang, and Sen Jan.

Experimental and numerical studies of sound propagation over a submarine canyon northeast of taiwan. *IEEE Journal of Oceanic Engineering*, 40(1):237–249, 2015.

[15] Frédéric Sturm, Sven Ivansson, Yong-Min Jiang, and N Ross Chapman. Numerical investigation of out-of-plane sound propagation in a shallow water experiment. *The Journal of the Acoustical Society of America*, 124(6):EL341–EL346, 2008.

[16] Megan S Ballard and Jason D Sagers. Numerical analysis of three-dimensional acoustic propagation in the catoche tongue. *The Journal of the Acoustical Society of America*, 138(4):EL365–EL369, 2015.

[17] David R Dall'Osto and Peter H Dahl. Observations of water column and bathymetric effects on the incident acoustic field associated with shallow-water reverberation experiments. *IEEE Journal of Oceanic Engineering*, 42(4):1146–1161, 2017.

[18] Kevin D Heaney and Richard L Campbell. Three-dimensional parabolic equation modeling of mesoscale eddy deflection. *The Journal of the Acoustical Society of America*, 139(2):918–926, 2016.

[19] Leandro Calado, Orlando Camargo Rodríguez, Gabriel Codato, and Fabio Contrera Xavier. Upwelling regime off the cabo frio region in brazil and impact on acoustic propagation. *The Journal of the Acoustical Society of America*, 143(3):EL174–EL180, 2018.

[20] Sean M Reilly, Gopu R Potty, and Michael Goodrich. Computing acoustic transmission loss using 3D Gaussian ray bundles in geodetic coordinates. *Journal of Computational Acoustics*, 24(01):1650007/1–24, 2016.

[21] Cristiano Soares, Friedrich Zabel, and Sérgio M Jesus. A shipping noise prediction tool. In *OCEANS 2015-Genova*, pages 1–7. IEEE, 2015.

[22] Rogério Calazan and Orlando C. Rodríguez. TRACEO3D ray tracing model for underwater noise predictions. In *Doctoral Conference on Computing, Electrical and Industrial Systems*, pages 183–190. Springer, 2017.

[23] Michael J Buckingham. Theory of three-dimensional acoustic propagation in a wedgelike ocean with a penetrable bottom. *The Journal of the Acoustical Society of America*, 82(1):198–210, 1987.

[24] Rogério Calazan, Orlando C. Rodríguez, and Nadia Nedjah. Parallel ray tracing for underwater acoustic predictions. In *Proceedings of the 17th ICCSA2017*, volume 10404, pages 43–55, 3–6 July, Trieste, Italy, 2017.

[25] Richard Michael Jones, Jack Parker Riley, and Thomas Martin Georges. Harpo: A versatile three-dimensional hamiltonian ray-tracing program for acoustic waves in an ocean with irregular bottom. *NOAA Report*, 1986.

[26] JA Mercer, WJ Felton, and JR Booker. Three-dimensional eigenrays through ocean mesoscale structure. *The Journal of the Acoustical Society of America*, 78(1):157–163, 1985.

[27] Michael B Porter and Homer P Bucker. Gaussian beam tracing for computing ocean acoustic fields. *The Journal of the Acoustical Society of America*, 82(4):1349–1359, 1987.

[28] Homer P Bucker. A simple 3-D Gaussian beam sound propagation model for shallow water. *The Journal of the Acoustical Society of America*, 95(5):2437–2440, 1994.

[29] Vlastislav Červenỳ and Ivan Pšenčík. Ray amplitudes of seismic body waves in laterally inhomogeneous media. *Geophysical Journal International*, 57(1):91–106, 1979.

[30] Ocean acoustics library. http://oalib.hlsresearch.com/. Accessed 2018-07-03.

[31] Michael B. Porter. BELLHOP3D user guide. Technical report, Heat, Light, and Sound Research, Inc., 2016.

[32] Lewis Dozier and Pierre Lallement. Parallel implementation of a 3-D range-dependent ray model for replica field generation. In *Full Field Inversion Methods in Ocean and Seismo-Acoustics*, pages 45–50. Springer, 1995.

[33] S Ivansson. Stochastic ray-trace computations of transmission loss and reverberation in 3-D range-dependent environments. In *8th European Conference on Underwater Acoustics*, pages 131–136, 2006.

[34] Trond Jenserud and Sven Ivansson. Measurements and modeling of effects of out-of-plane reverberation on the power delay profile for underwater acoustic channels. *IEEE Journal of Oceanic Engineering*, 40(4):807–821, 2015.

[35] Nick Maltsev. Enhanced ray theory. *Journal of computational acoustics*, 9(01):169–182, 2001.

[36] Tal Heilpern, Ehud Heyman, and Vadim Timchenko. A beam summation algorithm for wave radiation and guidance in stratified media. *The Journal of the Acoustical Society of America*, 121(4):1856–1864, 2007.

[37] Yael Gluk and Ehud Heyman. Pulsed beams expansion algorithms for time-dependent point-source radiation. a basic algorithm and a standard-pulsed-beams algorithm. *IEEE Transactions on Antennas and Propagation*, 59(4):1356–1371, 2011.

[38] Sean M Reilly, Gopu R Potty, and David Thibaudeau. Investigation of horizontal refraction on florida straits continental shelf using a three-dimensional gaussian ray bundling model. *The Journal of the Acoustical Society of America*, 140(3):EL269–EL273, 2016.

[39] O. G. Johnson. Three-dimensional wave equation computations on vector computers. *Proceedings of the IEEE*, 72(1):90–95, Jan 1984.

[40] Ananth Grama, Vipin Kumar, Anshul Gupta, and George Karypis. *Introduction to parallel computing*. Pearson Education, 2003.

[41] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2013.

[42] Paul Hursky and Michael B Porter. Accelerating underwater acoustic propagation modeling using general purpose graphic processing units. In *OCEANS 2011*, pages 1–6. IEEE, 2011.

[43] Xuehai Sun, Lianglong Da, and Yuyang Li. Study of BDRM asynchronous parallel computing model based on multiple cuda streams. In *Computational Intelligence and Design (ISCID), 2014 Seventh International Symposium on*, volume 1, pages 181–184. IEEE, 2014.

[44] Matteo Lazzarin. Parallel implementation of a ray tracer for underwater sound waves using the cuda libraries: description and application to the simulation of underwater networks. Master's thesis, 2012.

[45] Emanuel Ey. Adaptation of an acoustic propagation model to the parallel architecture of a graphics processor. Master's thesis, University of Algarve, 2013.

[46] Orlando C. Rodriguez, Jon M Collis, Harry J Simpson, Emanuel Ey, Joseph Schneiderwind, and Paulo Felisberto. Seismo-acoustic ray model benchmarking against experimental tank data. *The Journal of the Acoustical Society of America*, 132(2):709–717, 2012.

[47] Mikhail Mikhailovich Popov. *Ray theory and Gaussian beam method for geophysicists.* EDUFBA, Salvador, Bahia, 2002.

[48] MM Popov, Ivan Pšenčík, and V Červenỳ. Computation of ray amplitudes in inhomogeneous media with curved interfaces. *Studia Geophysica et Geodaetica*, 22(3):248–258, 1978.

[49] P. Papadakis, M. Taroudakis, and J. Papadakis. Recovery of the properties of an elastic bottom using reflection coefficient measurements. In *Proceedings of the 2nd. European Conference on Underwater Acoustics*, volume II, pages 943–948, Copenhagen, Denmark, 1994.

[50] Rogério Calazan and Orlando C. Rodríguez. Simplex based three-dimensional eigenray search for underwater predictions. *The Journal of the Acoustical Society of America*, 143(4):2059–2065, 2018.

[51] John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[52] Jeffrey C Lagarias, James A Reeds, Margaret H Wright, and Paul E Wright. Convergence properties of the nelder–mead simplex method in low dimensions. *SIAM Journal on optimization*, 9(1):112–147, 1998.

[53] David A Patterson and John L Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes, 2013.

[54] CUDA C programming guide. Technical report, Nvidia Corporation, 2018. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed 2018-05-16.

[55] M Fatica and G Ruetsch. *CUDA FORTRAN for Scientists and Engineers*. Morgan Kaufmann, Burlington, 2014.

[56] CUDA C best practices guide. Technical report, Nvidia Corporation, 2018. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html. Accessed 2018-05-10.

[57] Vasily Volkov and James W Demmel. Benchmarking GPUs to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.

[58] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, volume 10, page 16. San Jose, CA, 2010.

[59] Rogério Calazan and Orlando C. Rodríguez. TRACEO3D ray tracing model and its parallel implementation. Poster in Ciência 2016, Lisboa, Portugal, July 2016.

[60] Open source high performance computing. https://www.open-mpi.org/. Accessed 2018-06-13.

[61] Alexandra Tolstoy. *Matched field processing for underwater acoustics*. World Scientific, 1993.

[62] Cristiano Soares, Sérgio M Jesus, and Emanuel Coelho. Environmental inversion using high-resolution matched-field processing. *The Journal of the Acoustical Society of America*, 122(6):3391–3404, 2007.

[63] PGI version 17.10 documentation for x86 and NVIDIA processors. https://www.pgroup.com/resources/docs/17.10/x86/index.htm. Accessed 2018-06-07.

[64] CUDA FORTRAN programming guide. https://www.pgroup.com/resources/docs/17.10/x86/cuda fortran-prog-guide/index.htm. Accessed 2018-06-17.

[65] The GNU FORTRAN compiler. https://gcc.gnu.org/onlinedocs/gfortran/Interoperability-with-C.html. Accessed 2018-06-05.

[66] Alexios Korakas, Frédéric Sturm, Jean-Pierre Sessarego, and Didier Ferrand. Scaled model experiment of long-range across-slope pulse propagation in a penetrable wedge. *The Journal of the Acoustical Society of America*, 126(1):EL22–EL27, 2009.

[67] Floating point and IEEE 754 compliance for NVIDIA GPUs. Technical report, Nvidia Corporation, 2018. https://docs.nvidia.com/cuda/floating-point/index.html. Accessed 2018-05-31.

[68] GB Deane and MJ Buckingham. An analysis of the three-dimensional sound field in a penetrable wedge with a stratified fluid or elastic basement. *The Journal of the Acoustical Society of America*, 93(3):1319–1328, 1993.

[69] John A Fawcett. Modeling three-dimensional propagation in an oceanic wedge using parabolic equation methods. *The Journal of the Acoustical Society of America*, 93(5):2627–2632, 1993.

[70] Rogério Calazan and Orlando C. Rodríguez. Three-dimensional eigenray search for vertical line array. In *UACE2017 - 4th Underwater Acoustics Conference and Exhibition*, pages 941–946. UACE Proceedings, 2017.

[71] Rogério Calazan and Orlando C. Rodríguez. GPU-BASED 3D eigenray search for underwater acoustic predictions. Poster in Ciência 2018, Lisboa, Portugal, July 2018.

[72] CUDA Toolkit documentation. Technical report, Nvidia Corporation, 2018. https://docs.nvidia.com/cuda/index.html. Accessed 2018-07-05.

Appendices

Appendix A

Installation

A.1 Pre-installation tasks

The parallel version was developed based on the heterogeneous programing CPU + GPU considering the CUDA platform. Thus, gfortran GNU and nvcc CUDA® compilers are needed. CUDA is a parallel computing platform and programming model developed by NVIDIA. Users have to follow the NVIDIA CUDA Installation Guide procedures to install the CUDA toolkit [72]. The validation tests were performed using an Ubuntu 16.04 LTS operation system.

The input file generates a structure with the environmental information and model configuration. In the same file, the model is called to perform calculations. The input file have to be loaded using MATLAB® or Python. More details about the input file are presented in Appendix B.

A.2 Model installation

Before running the `install` script on the command line the user should review the correspondent definitions and adapt them to his local machine. After a successful compilation the user can place the resulting binary (`gputraceo3d.exe`) in a folder, where the system

can find it.

To install the model open a command line and execute the install script

```
$ ./install.sh
```

This script creates the objects, the execution file and clean the objects code; specific details can be found inside the script file.

A.3 Compiling options

The following list shows the definitions used in the installation file:

```
CC          = GPU compiler
FC          = FORTRAN compiler
CFLAGS      = CUDA capability
SDFLAGS     = library flags location
EXECUTABLE = executable file name
CMODULES    = cuda model source files
COBJECTS    = cuda objects
SOURCES     = FORTRAN model source files
PSOURCES    = FORTRAN model source files for bridge pourposes
```

A.4 Compilation file example

A particular makefile is shown below:

```
# MAKEFILE CPU/GPU TRACEO3D
# by Rogerio Calazan and Orlando Rodriguez
# Faro-PT, Sat Jul  7 18:21:27 WEST 2018

# *****  COMPILER *****
CC          = nvcc
FC          = gfortran

# *****  COMPILER FLAGS *****
CFLAGS  = --gpu-architecture=sm_50
SDFLAGS = -L /usr/local/cuda/lib64 -I /usr/local/cuda/include/thrust -lcuda

# *****  EXEC NAME *****
EXECUTABLE = gputraceo3d.exe
```

```
# ****** GPU TRACEO3D Files ******
CMODULES =          cudabridge.cu \
                       kernels.cu \
                      fdevices.cu \

COBJECTS =          cudabridge.o \
                       kernels.o \
                      fdevices.o \

SOURCES =       baryco2d.for \
                baryco3d.for \
                  brcket.for \
                   bdryr.for \
                  blii1d.for \
                  bcui1d.for \
                  blii2d.for \
                  bcui2d.for \
                  blii3d.for \
                  bcui3d.for \
                  bpai2d.for \
                  calrco.for \
                  caleis.for \
                  cnvnts.for \
                  cramer.for \
                   cross.for \
                  csvals.for \
                  cvals1.for \
                  cvals2.for \
                  cvals3.for \
                  dtelcm.for \
                       . . .
                       . . .
                       . . .
                   rkf45.for \
                   sbdyi.for \
                  si2p2d.for \
                  si3p2d.for \
                  si4p2d.for \
                   sorti.for \
                    r2x2.for \
                   raybi.for \
                  reflct.for \
                  seikeq.for \
                  sdyneq.for \
                  thorpe.for \
                  calcpf.for \
                 gpressf.for \
```

```
        bracketReceiver.for \
                invmat.for \
            sdyneqPress.for \
              calcpf_ii.for \
             gpressf_ii.for \

PSOURCES =    caleisgpu.for \
               calesgpu.for \
              calcprgpu.for \
                 calgpu.for \
               simplex.for \
       seikeqSegPlanInt.for \
            intsegplane.for \

allp: modc execp install clean

modc:    $(CMODULES)
         $(CC) $(CFLAGS) --device-c $(CMODULES) --resource-usage
         $(CC) $(CFLAGS) --device-link $(COBJECTS) --output-file link.o
         --resource-usage

execp: $(OBJECTS)
         $(FC) -fbounds-check -o $(EXECUTABLE_GPU) traceo3d.for
         $(SOURCES) $(PSOURCES) $(COBJECTS) link.o $(SDFLAGS) -O3

execs: $(OBJECTS)
         $(FC) -o $(EXECUTABLE) traceo3ds.for $(SOURCES) -O3

install:
         mv *.exe ~/bin
clean:
         rm *.o
```

Appendix B

Input file structure

B.1 Running options

The general structure of the input file (hereafter called INFIL) can be better understood if one thinks of it as composed of blocks; each block describes a particular element of the waveguide, from top to bottom. In order to provide a friendly view of the INFIL the blocks are separated with a long line, which is ignored by the model. The structure of the INFIL is as follows:

```
Title
Source          Block
Altimetry       Block
Sound Speed     Block
Objects         Block
Bathymetry      Block
Array           Block
Output          Block
```

The Title is a character string, which is written in the LOGFIL (the file with the *.log extension). The structure of each block is as follows:

Source Block:

```
source_data.ds          ray step
source_data.position    source coordinates
source_data.f           source frequency
source_data.thetas      elevation angles
source_data.phi         azimuth angles
```

```
source_data.nthetas      number of elevation angles
source_data.nphi         number of azimuth angles
source_data.xbox         range box in x axis
source_data.ybox         range box in y axis
```

Altimetry Block:

```
surface_data.type        surface type
surface_data.ptype       surface properties
surface_data.itype       surface interpolation
surface_data.x           surface coordinates
surface_data.y           surface coordinates
surface_data.z           surface coordinates
surface_data.units       attenuation units
surface_data.properties  surface properties
```

surface type can be one of the following characters:

'A'	absorbent surface
'E'	elastic surface
'R'	rigid surface
'V'	vacuum over surface

surface properties can be one of the following characters:

'H'	homogeneous surface
'N'	non-homogeneous surface

interpolation type can be one of the following characters:

'FL'	flat surface
'2P'	piecewise linear interpolation

units can be one of the following characters:

'F'	dB/kHz
'M'	db/meter
'N'	dB/neper
'Q'	Q factor
'W'	db/λ

Sound Speed Block:

```
ssp_data.ctype  type of sound speed distribution
ssp_data.x      point coordinates
ssp_data.y      point coordinates
ssp_data.z      point coordinates
ssp_data.c      sound speed data
```

For a sound speed field both range and depth derivatives are calculated using a bi-dimensional barycentric parabolic interpolator, on the grid of points. For a sound speed profile all range derivatives are zero; depth derivatives are calculated depending on the value of type, which can be one of the following strings:

'ISOV'	isovelocity	profile
'TABL'	tabulated	profile

When specifying the sound speed profile or field it is highly recommended to use an evenly spaced grid, avoiding vertical segments where a smooth variation is followed by an isovelocity layer. Including such segments introduce unrealistic artifacts, which result from the calculation of inaccurate sound speed gradients.

Object Block:

The 3D object capability is under development, but the following line is mandatory:

```
object_data.nobjects = 0;
```

Bathymetry Block:

The structure of this block is identical to the structure of the altimetry block.

Array Block:

```
output_data.x sensors in x coordinate
output_data.y sensors in y coordinate
output_data.z sensors in z coordinate
output_data.nxa number of sensors along x
output_data.nya number of sensors along y
output_data.nza number of sensors along z
```

Array geometry definition:

- **single receiver**: x,y,z with one element;
- **vertical line array**: x and y with one element; z with depth coordinates;
- **horizontal line array in x**: y and z with one element; x with range coordinates;
- **horizontal line array in y**: x and z with one element; y with range coordinates;
- **vertical plane array in x**: y with one element; x with range coordinates; z with depth coordinates;
- **vertical plane array in y**: x with one element; y with range coordinates; z with depth coordinates.

Output Block:

```
output_data.ctype       output type
output_data.miss        eigenray parameter
```

The option outype defines the type of output and can correspond to one of the following strings:

'CPR'	output Coherent acoustic PRessure
'EIR'	output Eigenrays, parallel regions
'EIS'	output Eigenrays, parallel regions and Simplex

The miss parameter is used as a threshold to find the 3D eigenrays.

B.2 Model output

After creating the input file (for instance, munk.in) the user can run the model with the command

```
$ gputraceo3d.exe munk
```

according to the desired output the model will create one of the following Matlab mat files:

1. `cpr.mat`: coherent acoustic pressure; and

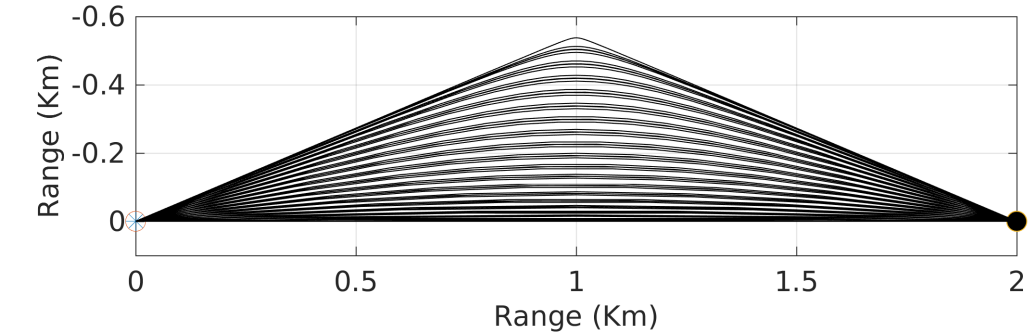
2. `eig.mat`: eigenray information

B.3 Example

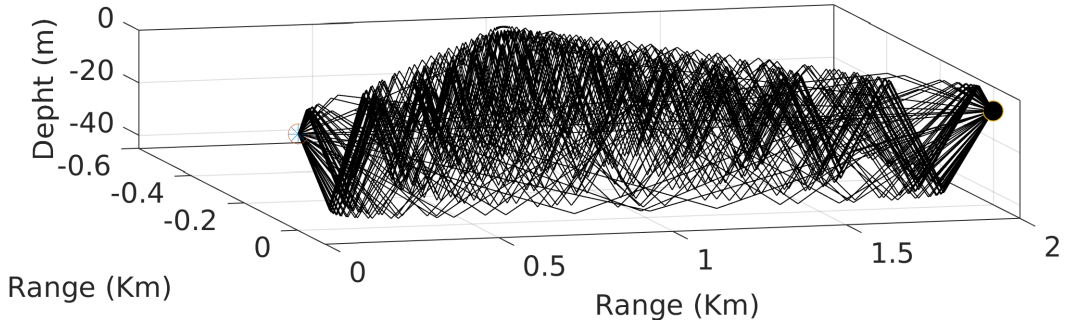
A example using the environmental and geometry information from the tank scaled experiment (see Section 5.2) and considering a source-hydrophone range of 2 km is provided here to illustrate the model utilization. The plots shown in Figs B.1 and B.2 are produced by running the model with the output option 'EIR' and using the follow command under the Matlab prompt

```
$tank_gputraceo3d
```

All the files mentioned in these appendix are distributed together with the model code.



(a)



(b)

Figure B.1: Eigenray predictions for TRACEO3D, across-slope propagation on the wedge waveguide; (a) horizontal plane and (b) perspective view. Source-receiver ranger corresponds to 2 km.

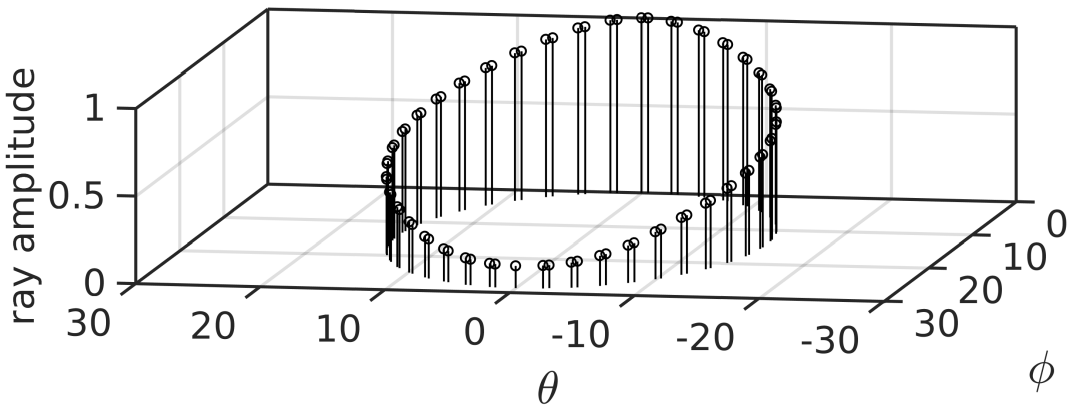


Figure B.2: Predictions of normalized amplitudes versus launching angles for a receiver at 2km.