PARALLEL 2D SEISMIC RAY TRACING USING CUDA ON A JETSON NANO

Ban-Sok Shin, Luis Wientgens, and Dmitriy Shutin

German Aerospace Center (DLR)
Institute of Communications and Navigation
E-Mail: {ban-sok.shin, luis.wientgens, dmitriy.shutin}@dlr.de

ABSTRACT

We present a parallel implementation of a 2D seismic ray tracer on a graphics processing unit of the compact Jetson Nano by Nvidia. Ray tracing is commonly used in seismic imaging as an intermediate step in reconstructing subsurface structures. We employ a gradient-based ray tracer that requires a travel time map. Here, we make use of the fast iterative method that computes a travel time map in a parallel fashion. Since a ray path is independent from any other ray path, the tracing can be implemented in parallel as well. To this end, we use a graphics processing unit and implement ray tracing using the CUDA programming language. For performance evaluations we compare our implementation on the Jetson Nano to a state-of-the-art sequential seismic ray tracer on a desktop CPU and show that speedups with factors of up to three can be achieved. The results indicate that edge devices such as the Jetson Nano can play a relevant role for tomographic applications particularly in scenarios where mobility of processing devices and compactness are important.

***Index Terms*—** Parallel computing, CUDA, GPGPU, seismic ray tracing, Jetson Nano, edge devices, multi-agent seismic exploration

1. INTRODUCTION

Seismic data from extra-terrestrial bodies is highly relevant in current planetary missions [1]. In recent years, concepts for autonomous seismic surveys conducted by multiple mobile robotic platforms on e.g. Mars or Moon have been proposed [2, 3]. In particular, we proposed to perform seismic imaging within a network of multiple agents that are connected over wireless links without the need of a central entity. To this end, distributed subsurface imaging schemes for travel time tomography and full waveform inversion are applicable and have been proposed [4, 5, 6, 7]. However, to enable imaging on mobile robotic platforms, efficient hardware implementations of respective algorithms are required. For instance, in travel time tomography, ray tracing is required and an efficient implementation is desired to speed up imaging. Graphics processing units (GPUs) offer the capability to implement algorithms in a parallelized fashion and with high efficiency. In particular, edge devices such as Nvidia's Jetson Nano are equipped with a GPU and offer the possibility to implement highly efficient, parallelized software on a compact, mobile device. Especially, the compact form is highly relevant for mobile robotic applications as described above.

An important ingredient of travel time tomography is ray tracing [8]. The latter also finds applications in ,e.g., computer graphics [9], astronomy [10] and optical systems [11]. In seismic imaging it is required to reconstruct ray paths between source and receivers for the first arrival times. Based on the ray paths the forward calculation of travel times can be linearized and an inversion for a subsurface model with respect to (wrt.) the P-wave velocity can be performed. Under various ray tracing methods those that employ the gradient of travel times are commonly used [12]. To compute travel time gradients, a travel time map is required. Here, the fast iterative method (FIM) proposed in [13] can be used that performs a parallel computation of travel times on a GPU. Since a ray path is independent of any other ray path its computation can be parallelized. Such parallel implementations of seismic ray tracers on GPUs exist, see e.g. [14, 15, 16]. All of them achieve faster computation times than comparable implementations on a central processing unit (CPU). However, none of these considers the implementation and performance evaluation of ray tracers on an edge device such as the Jetson Nano where computational resources are rather limited compared to GPUs on graphics cards.

In this work, we present and evaluate a parallel implementation of a gradient-based seismic ray tracer on the compact Jetson Nano in the compute unified device architecture (CUDA) programming language. The required travel time map is provided by the FIM that is also implemented in parallel on the Jetson Nano. Since then the travel time map is in the GPU memory the ray tracer can directly access it without additional memory transfer. We investigate our parallelized implementation on the Jetson Nano to a state-of-the-art sequential seismic ray tracer on a desktop CPU for different subsurface models with differing grid sizes. Our parallel implementation shows total speed-ups by factors up to three.

2. PARALLEL EIKONAL SOLVER IN CUDA

In the following, we briefly summarize the FIM that computes a travel time map required for the gradient-based ray tracer. For a brief overview of CUDA the reader is referred to Appendix A.

To synthesize a map of first arrival travel times over a spatial domain $\Omega \subset \mathbb{R}^2$ the *eikonal equation* needs to be solved. With a subsurface model $m(\boldsymbol{x})$ that describes the P-wave velocity over the spatial coordinate $\boldsymbol{x} = (x, z) \in \mathbb{R}^2$ the eikonal equation is given as

$$|\nabla T(\boldsymbol{x})|^2 = \frac{1}{m(\boldsymbol{x})^2}, \quad \text{s.t.} \quad T(\boldsymbol{x}_s) = 0, \boldsymbol{x} \in \Omega, \qquad (1)$$

where ∇ is the gradient operator, $T(\boldsymbol{x})$ is the travel time function and $\boldsymbol{x}_s$ is the source position. At source position, the travel time needs to be 0 s, hence the respective initial condition at $\boldsymbol{x}_s$.

To solve the eikonal equation (1) wrt. the travel times $T(\boldsymbol{x})$ several numerical methods exist such as the fast marching method (FMM) or the fast sweeping method (FSM) [17, 18]. However, both methods do not allow for a parallel computation which is a prerequisite for an efficient implementation on a GPU. To enable a parallel implementation the fast iterative method (FIM) has been proposed in [13]. The main idea of FIM is to solve the eikonal equation using

an active list of grid points that keeps track of all travel times which require updating. All travel times in this active list are updated simultaneously without any order. This avoids the use of complex data structures such as a heap in FMM and thus enables parallelization and efficient implementation on GPU architectures. Travel times in the active list are updated until the absolute difference between current and updated travel time is below a certain threshold, at which the corresponding travel time is removed from the active list.

To solve (1) numerically finite differences with a regular grid discretization can be applied. To this end, the computational domain Ω is discretized with a distance Δx and Δz between grid points in the x and z-direction, respectively. A grid point is described by the index pair (i, j) that represents the spatial coordinate (x_i, z_j). To compute travel times at a specific grid point (i, j) Godunov upwind discretization is used and the eikonal equation is then changed into

$$[\max(T_{i,j} - T_{i,j}^{xmin}, 0)]^2 + [\max(T_{i,j} - T_{i,j}^{zmin}, 0)]^2 = \frac{1}{m_{i,j}^2}, \quad (2)$$

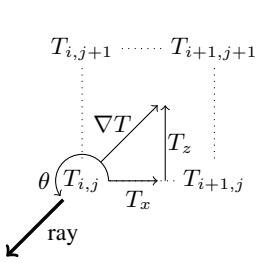
where $T_{i,j}^{xmin} = \min(T_{i-1,j}, T_{i+1,j})$ and $T_{i,j}^{zmin} = \min(T_{i,j-1}, T_{i,j+1})$. To obtain $T_{i,j}$ from (2), a quadratic solver can be used [13]. The travel time for one grid point in the active list is computed on one CUDA core enabling parallelization of these computations. Algorithmic and implementation details with pseudo-code can be found in [13]. In addition, FIM separates the discretized computational domain into blocks of grid points to optimize memory access. Each block consists of 64 grid points and is processed by one streaming multiprocessor (SM) on the GPU. Each block is indexed by the CUDA internal variable `blockIdx`. While processing one block the fast L1 cache of the SM can be accessed by all threads that update the travel times of the respective grid points. However, grid points that lie at the border of a block are accessed over the slower L2 cache that can be accessed by all SMs. This is done to enable a block to access grid points that belong to a neighboring block and thus, are not stored in its L1 cache.

3. PARALLEL GRADIENT-BASED SEISMIC RAY TRACING IN CUDA

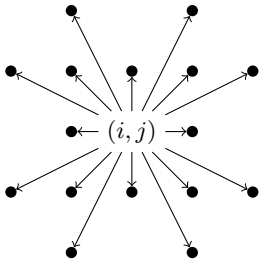
In seismic inversion ray tracing is an important step to obtain an image of the subsurface. Here, seismic waves are approximated by rays that travel from source to receivers. This simplification is valid for a high frequency approximation of the wave physics [8]. The ray paths are needed to build the sensitivity matrix that linearizes the forward model given by the eikonal equation (1). Doing so, enables a linearized inversion for the model parameters in $m(\boldsymbol{x})$, i.e., the spatial distribution of P-wave velocities in the subsurface Ω.

3.1. General concept

Various methods exist to trace seismic ray paths, see [8] for a detailed overview. When a travel time map $T(\boldsymbol{x})$ is available, as in our case, ray tracing based on gradient-descent can be used. The key idea here is to use a travel time gradient in order to reach the global minimum of the travel time map. This minimum is located at the source position $\boldsymbol{x}_s$ since here the travel time $T(\boldsymbol{x}_s)$ is zero, cf. (1). Hence, by following the steepest descent in the travel time map, i.e. the negative gradient, the ray can be traced back from receiver to source. Since each ray path between receiver and source is independent of any other ray path the ray tracer can be implemented in parallel. For instance, each ray path can be traced separately by one CUDA core on the GPU. Algorithm 1 summarizes the general pseudo-code of a parallel gradient-based ray tracer. For the steepest-descent update an appropriate step site $\mu > 0$ needs to be selected to enable convergence to the global minimum $T(\boldsymbol{x}_s) = 0$.



(a) Illustration of Eq. (4).



(b) Possible ray paths for 16 points.

Fig. 1: Computation of angle θ and possible paths for 16 grid points.

Algorithm 1 Gradient descent-based parallel ray tracer

Require: Travel time map T, receiver positions $\boldsymbol{x}_r$, ray array $\boldsymbol{R}$
  for each receiver position $\boldsymbol{x}_r$ **in parallel do**
    Initialize empty ray path vector $\boldsymbol{r}$
    Add $\boldsymbol{x}_r$ to $\boldsymbol{r}$
    $\boldsymbol{x}_k = \boldsymbol{x}_r$
    while $||\boldsymbol{x}_s - \boldsymbol{x}_k||_2^2 > \min(\Delta x, \Delta z)$ **do**
      Compute gradient $\boldsymbol{d}_k \leftarrow \nabla T(\boldsymbol{x})|_{\boldsymbol{x}=\boldsymbol{x}_k}$
      Compute next ray coordinate $\boldsymbol{x}_k \leftarrow \boldsymbol{x}_k + \mu \cdot \boldsymbol{d}_k$
      Add $\boldsymbol{x}_k$ to $\boldsymbol{r}$
    end while
    Add $\boldsymbol{r}$ to $\boldsymbol{R}$
  end for

3.2. Implementation details

To implement Algorithm 1 numerically, we first need to approximate the travel time gradient $\nabla T(\boldsymbol{x})$. Such approximation gives us a direction of the ray path at the grid point (i, j). To this end, a first order finite difference can be applied on the travel times as in [19]:

$$T_x = \frac{\partial T}{\partial x} \approx \frac{(T_{i+1,j} - T_{i,j}) + (T_{i+1,j+1} - T_{i,j+1})}{2\Delta x} \quad (3a)$$

$$T_z = \frac{\partial T}{\partial z} \approx \frac{(T_{i,j+1} - T_{i,j}) + (T_{i+1,j+1} - T_{i+1,j})}{2\Delta z} \quad (3b)$$

Here, the travel time gradients in x and z direction are approximated by taking the average of two adjacent gradients to improve accuracy of the numerical approximation. For receivers/sources placed at the borders of the domain, we assume that FIM provides these additional travel times by enlarging the domain accordingly. Based on the approximated gradients T_x and T_z an angle θ can be calculated that gives the direction of the ray path where θ is measured counterclockwise from the positive horizontal axis. Figure 1a illustrates the computation of the ray angle θ. The angle θ is then given by

$$\theta = \arctan(T_z/T_x) + \pi \quad (4)$$

with the approximated gradients in (3). Note that the ray path direction is opposite to the travel time gradient as the rays are traced in reverse direction using steepest descent. Therefore, π is added to the

Region of θ	Next grid point
$\theta \leq 22,5^\circ$ or $\theta > 337.5^\circ$	$(i+1, j)$
$22.5^\circ < \theta \leq 67.5^\circ$	$(i+1, j+1)$
$67.5^\circ < \theta \leq 112.5^\circ$	$(i, j+1)$
$112.5^\circ < \theta \leq 157.5^\circ$	$(i-1, j+1)$
$157.5^\circ < \theta \leq 202.5^\circ$	$(i-1, j)$
$202.5^\circ < \theta \leq 247.5^\circ$	$(i-1, j-1)$
$247.5^\circ < \theta \leq 292.5^\circ$	$(i, j-1)$
$292.5^\circ < \theta \leq 337.5^\circ$	$(i+1, j-1)$

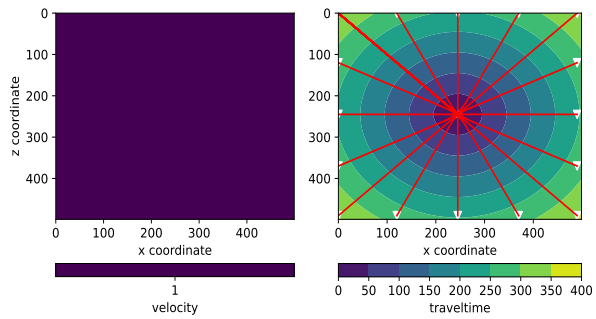
Table 1: Decision boundaries of angle θ and corresponding next grid points for an example of 8 possible ray paths from (i, j).

right-hand side of (4). Depending on the angle θ the next grid point in the vicinity of the current point (i, j) is chosen for the ray path. Table 1 gives the decision boundaries for a ray path for an example of eight possible grid points in the direct vicinity of (i, j). If more accurate ray paths are required the number of possible grid points can be analogously extended to include grid points that are further away. In our implementation, we used 16 possible grid points by including points that are two cells away from the current point (i, j) resulting in the possible paths shown in Figure 1b. Based on the decision boundaries, the next grid cell of the ray path is chosen. This step approximates the gradient computation and steepest descent update in Algorithm 1. By doing so, the ray path can be traced back until the respective grid point of the source position $\boldsymbol{x}_s$ is reached.

Again, due to independence of rays, the gradient computation can be parallelized. To this end, computations of gradients (3), angle θ in (4) and decision on the next grid point for the ray are done in parallel. On a GPU, each ray is then traced by one thread in one CUDA core where all threads are executed by one SM. The complete ray tracing is done without memory transfer from the GPU until termination using the travel time map in the global GPU memory that has been computed by FIM as described in Section 2. One issue is the initialization of arrays that represent each ray path. Each ray path is likely to include a different number of grid points. However, the CUDA kernels use fixed size static arrays as a means of representation. To mitigate this issue, the static arrays for the ray paths are initialized with the theoretically possible maximum length for a ray in the given environment. This maximum length is given by the sum of the grid points in x- and z-direction of the spatial domain Ω. Doing so, guarantees that each static array has sufficient size to store a ray path.

4. PERFORMANCE EVALUATION

We test our parallel ray tracing implementation in multiple scenarios and compare it against ttcrpy, a sequential implementation of seismic ray tracing from [12] that runs on a desktop CPU. We use a laptop running Python 3.9 on an i7-1185G7 to test the sequential implementation and an Nvidia Jetson Nano 2GB which has 2 SMs of the Maxwell architecture to run our parallelized ray tracer together with the FIM for travel time computation. We evaluate the performance in nine scenarios that include different velocity models $m(\boldsymbol{x})$ and domain sizes. We consider a homogeneous model with constant velocity, a model that contains two layers with differing velocity and a model with a velocity gradient. For each model we test three different domain sizes, namely 100×100, 200×200 and 500×500 grid points. Also, we use a regular grid with $\Delta x = \Delta z = 1$. Velocity models and parameters do not have a physical unit by default but can be given meaning by assigning the respective unit to the grid spacing or the velocity value. For the homogeneous model the source is placed in the middle of the domain while we place 16 receivers around the domain edges. In the layer and gradient case the seismic source is placed at $\boldsymbol{x}_s = (0, 0)$, i.e., the upper left corner of the domain while we employ 19 receivers placed in a line array on the surface.



(a) Homogeneous velocity model (b) Ray tracing results for homogeneous model

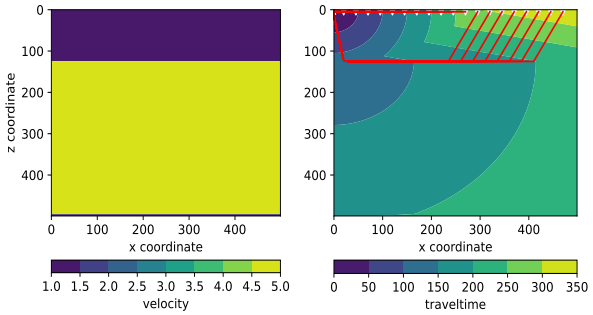
Fig. 2: Visualization of results for a 500×500 homogeneous model. Receivers are marked by white triangles.

Figures 2 - 4 show the velocity model and corresponding travel time maps with the computed ray paths for all three models assuming a domain size of 500×500. For the homogeneous model, rays follow a straight line between the source in the center to the receivers on the domain boundaries. Due to a constant velocity these ray paths give the shortest paths between source and receivers as expected. For the layered model, the phenomenon of refraction can be observed. Since the second layer has a higher velocity of 5 than the first layer, after a certain distance to the source rays start travelling in the second layer before they hit the receiver. This path gives the lowest travel time and is due to refraction known from Snell's Law. In particular, these rays travel at the critical angle, i.e., the refracted angle is 90° leading to a horizontal ray path in the second layer. If the distance between source and receiver is below a certain value, the direct ray over the surface gives the smallest travel time. This is visible for receivers that are closer to the source. A similar effect can be observed for the gradient model in Fig. 4. However, since here the velocity is gradually changing over the depth, rays do not take a distinct horizontal path but a rather bent path between source and receiver.

To evaluate the run time performance, we compare our implementation to ttcrpy [12]. For a fair comparison we disable the subgrids for ray tracing in ttcrpy such that rays are only traced between the grid vertices as in our implementation. We average the performance results over five runs for both implementations. Table 2 gives the profiling results for all tested scenarios. We report the computational time, the total time and the ratio between computational time and total time for both implementations. The computational time considers the time required for the pure computation of travel time map and ray tracing only. The total time additionally includes time consumption caused by memory allocation and data transfer between CPU and GPU. Such transfer is only relevant for the GPU implementation where data is moved between host and device. Furthermore, we list the resulting speedups of the parallel implementation for the computational and the total times. It can be clearly observed that despite having only 2 SMs on the Jetson Nano that our parallel implementation of the ray tracer is faster than ttcrpy running on

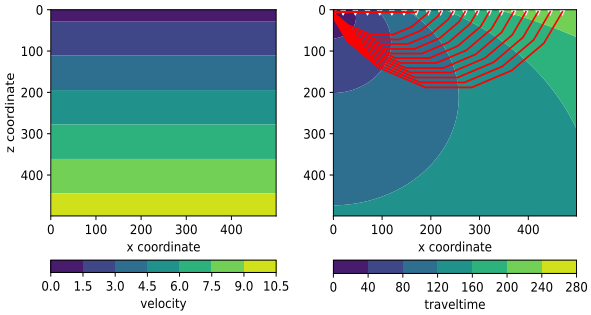
Scenario & grid size	Comp. time [s] CPU	Comp. ratio [%] CPU	Total time [s] CPU	Comp. time [s] Jetson	Comp. ratio [%] Jetson	Total time [s] Jetson	Comp. speedup Jetson/CPU	Total speedup Jetson/CPU
Homogeneous								
100×100	5.43	95.47	5.69	3	75.19	**3.98**	1.81	**1.43**
200×200	21.65	95.83	22.60	5.03	51.88	**9.70**	4.30	**2.33**
500×500	139.20	96.11	144.83	20.45	42.08	**48.63**	6.81	**2.98**
Layer								
100×100	5.56	96.15	5.79	3.84	78.27	**4.91**	1.45	**1.18**
200×200	22.24	96.01	23.16	7.89	63.52	**12.43**	2.82	**1.86**
500×500	145.56	96.04	150.33	29.36	51.07	**57.50**	4.96	**2.61**
Gradient								
100×100	5.20	95.17	5.46	4	80.37	**4.99**	1.30	**1.09**
200×200	21.62	95.66	22.60	7.99	65.13	**12.31**	2.71	**1.84**
500×500	142	95.80	148.23	30.45	51.67	**58.93**	4.66	**2.52**

Table 2: Profiling results for the considered models and domain sizes using ttcrpy [12] on a CPU and our implementation on a Jetson Nano.



(a) Velocity model with a layer interface (b) Ray tracing results for layer model

Fig. 3: Visualization of results for a 500×500 layered model. Receivers are marked by white triangles.



(a) Velocity model with a gradient (b) Ray tracing results for gradient model

Fig. 4: Visualization of results for a 500×500 gradient model. Receivers are marked by white triangles.

a desktop CPU. In particular, with increasing grid size the benefit of parallelization becomes more apparent, visible in speedup factors between two and nearly three. If we only consider the computational time, i.e., the time required for the computation of the travel time map and ray paths alone, the speedup is even higher with factors up to seven. Furthermore, for higher grid sizes we observe that more time is consumed for transferring data from the GPU to the CPU reflected in the decreasing ratio. This is due to an increasing amount of data for the travel time map and ray paths caused by a larger grid.

5. CONCLUSION

In this paper, we describe a parallel implementation of a seismic ray tracer for 2D subsurface models on an Nvidia Jetson Nano device. The ray tracer is based on gradient-descent and therefore, requires a travel time map. This travel time map is obtained by the FIM that is also implemented in parallel on the Jetson Nano. Numerical results show that our implementation on the Jetson Nano achieves speedup factors of up to three compared to a sequential seismic ray tracer running on a desktop CPU. If only the computation time is considered speedup factors of up to seven could be observed. The results indicate that GPU-based edge devices such as the Jetson Nano can be promising candidates for autonomous seismic explorations by a multi-agent network of mobile rovers. Here, our implementation will allow for a fully distributed architecture where computations are done by each rover carrying a GPU. For future work, implementing a complete seismic inversion on a Jetson Nano is of great interest.

A. APPENDIX: BRIEF INTRODUCTION TO CUDA

We give a short introduction to the CUDA hardware architecture to provide context. Further information can be found in [20]. CUDA allows offloading of computationally heavy but parallelizable code from execution on the CPU to the massively parallel GPU. This is done in the form of kernels that are launched on the GPU. In the CUDA terminology the CPU is known as the host and the GPU as the device. Host and device have separated dedicated memory and memory hierarchy. The device is made up of a global device memory and the graphics processor itself with its core and cache hierarchy. The graphics processor is comprised of a shared L2 cache, a global thread scheduler and multiple streaming multiprocessors (SM). SMs are further subdivided into 64 CUDA cores where each core executes one thread. Threads running on the same SM additionally have access to the L1 cache which allows faster access than the L2 cache. Parallelization is achieved by running computations concurrently on multiple CUDA cores.

6. REFERENCES

[1] Brigitte Knapmeyer-Endrun, Mark P. Panning, Felix Bissig, et al., "Thickness and structure of the martian crust from InSight seismic data," *Science*, vol. 373, no. 6553, pp. 438–443, 2021.

[2] S. W. Courville, N. E. Putzig, P. C. Sava, M. R. Perry, and D. Mikesell, "ARES: An Autonomous Roving Exploration System for Planetary Active-Source Seismic Data Acquisition," in *AGU Fall Meeting*, Dec. 2018.

[3] Armin Wedler, Martin J. Schuster, Marcus G. Müller, et al., "German Aerospace Center's advanced robotic technology for future lunar scientific missions," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 379, no. 2188, Nov. 2020.

[4] Ban-Sok Shin and Dmitriy Shutin, "Adapt-then-combine full waveform inversion for distributed subsurface imaging in seismic networks," in *IEEE ICASSP*, June 2021, pp. 4700–4704.

[5] Ban-Sok Shin and Dmitriy Shutin, "Distributed Traveltime Tomography Using Kernel-based Regression in Seismic Networks," *IEEE Geoscience and Remote Sensing Letters*, 2022.

[6] Sili Wang, Fangyu Li, Mark Panning, Saikiran Tharimena, Steve Vance, and Wenzhan Song, "Ambient Noise Tomography with Common Receiver Clusters in Distributed Sensor Networks," *IEEE Trans. Signal Inf. Process. Netw.*, vol. 6, pp. 656–666, 2020.

[7] Wenzhan Song, Fangyu Li, Maria Valero, and Liang Zhao, "Toward creating a subsurface camera," *MDPI Sensors*, vol. 19, no. 2, pp. 1–20, 2019.

[8] N. Rawlinson and M. Sambridge, "Seismic Traveltime Tomography of the Crust and Lithosphere," *Advances in Geophysics*, vol. 46, pp. 81–198, 2003.

[9] Timothy J. Purcell, Ian Buck, William R Mark, and Pat Hanrahan, "Ray tracing on programmable graphics hardware," in *ACM SIGGRAPH 2005 Courses*. 2005.

[10] S. Paltani and C. Ricci, "RefleX: X-ray absorption and reflection in active galactic nuclei for arbitrary geometries," *Astronomy & Astrophysics*, vol. 607, pp. A31, Nov. 2017.

[11] Michael J. Kidger, *Fundamental Optical Design*, SPIE, Dec. 2001.

[12] Bernard Giroux, "ttcrpy: A Python package for traveltime computation and raytracing," *SoftwareX*, vol. 16, 2021.

[13] Won Ki Jeong and Ross Whitaker, "A Fast Iterative Method for Eikonal Equations," *SIAM Journal of Scientific Computing*, 2008.

[14] Jorge Monsegny, Jonathan Monsalve, Kareth León, et al., "Fast Marching Method in Seismic Ray Tracing on Parallel GPU Devices," in *Communications in Computer and Information Science*, pp. 101–111. Springer International Publishing, 2019.

[15] Martin Sarajaervi and Henk Keers, "Ray-based modeling and imaging in viscoelastic media using graphics processing units," *Geophysics*, vol. 84, no. 5, pp. S425–S436, Sept. 2019.

[16] P.-C. Liao, C.-C. Lii, Y.-C. Lai, et al., "A Graphics Processing Unit Implementation and Optimization for Parallel Double-Difference Seismic Tomography," *Bulletin of the Seismological Society of America*, vol. 104, no. 2, pp. 953–961, Mar. 2014.

[17] Eran Treister and Eldad Haber, "A fast marching algorithm for the factored eikonal equation," *J. Comput. Phys.*, vol. 324, pp. 210–225, 2016.

[18] Yen-Hsi Richard Tsai, Li-Tien Cheng, Stanley Osher, and Hong-Kai Zhao, "Fast Sweeping Algorithms for a Class of Hamilton–Jacobi Equations," *SIAM Journal on Numerical Analysis*, vol. 41, no. 2, pp. 673–694, 2003.

[19] Xin-Xin Qu, Si-Xin Liu, and Fei Wang, "A new ray tracing technique for crosshole radar traveltime tomography based on multistencils fast marching method and the steepest descend method," in *Proceedings of the 15th International Conference on Ground Penetrating Radar*. June 2014, IEEE.

[20] John Cheng, Max Grossman, and Ty McKercher, *Professional CUDA C Programming*, John Wiley & Sons, 2014.