environmental sciences proceedings

Proceeding Paper

Enhancing Photon Transport Simulation in Earth's Atmosphere: Acceleration of Python Monte Carlo Model Using Vectorization and Parallelization Techniques †

Jona Brügmann ‡, Dmitry Efremenko *,‡ and Thomas Trautmann ‡

Remote Sensing Technology Institute (IMF), German Aerospace Center (DLR), 82234 Oberpfaffenhofen, Germany
* Correspondence: dmitry.efremenko@dlr.de
† Presented at the 5th International Electronic Conference on Remote Sensing, 7–21 November 2023; Available online: https://ecrs2023.sciforum.net/.
‡ These authors contributed equally to this work.

Abstract: Photon transport within Earth's atmosphere is a vital aspect of atmospheric science. The accurate modeling of radiative transfer is crucial for remote sensing data analysis. Yet, simulating the photon transport in multi-dimensional models poses a significant computational challenge. Monte Carlo simulations are a common approach, but they demand a large number of photons for reliable results. Parallelization techniques can be employed to accelerate Monte Carlo computations by using multi-core CPUs and GPUs. This research delves into a comparative analysis of different parallelization techniques for the Python version of the Monte Carlo model. We consider conventional photon transport simulations that rely on iterative loops, the multithreading technique, NumPy's vectorization, and GPU acceleration via the CuPy library. It is shown that CuPy, harnessing GPU parallelism, significantly accelerates simulations, making them suitable for large-scale scenarios. It is shown that as the number of photons grows, the overhead from reading and retrieving data to the GPU decreases, making the CuPy library an effective and easy-to-use option for Monte Carlo simulations.

Keywords: Monte Carlo method; radiative transfer; parallel computing; GPU computing

Citation: Brügmann, J.; Efremenko, D.; Trautmann, T. Enhancing Photon Transport Simulation in Earth's Atmosphere: Acceleration of Python Monte Carlo Model Using Vectorization and Parallelization Techniques. *Environ. Sci. Proc.* **2024**, *29*, 53. https://doi.org/10.3390/ECRS2023-15841

Academic Editor: Luca Lelli

Published: 11 December 2023

1. Introduction

Photon transport within Earth's atmosphere plays a pivotal role in advancing our understanding of atmospheric science, enabling precise remote sensing techniques, and facilitating the development of accurate climate models. The behavior of photons as they traverse the atmosphere, interacting with various atmospheric components such as gases and aerosols, can be described by means of the radiative transfer equation [1,2]. Accurate modeling of these interactions is paramount for obtaining meaningful and quantitative insights from remote sensing data. In the context of the rapidly evolving field of remote sensing and the emergence of new sensors such as Sentinel 5 with fine spatial resolution, it has become increasingly important to design fast and accurate multi-dimensional radiative transfer models. These models are essential for interpreting the data generated by modern sensors and extracting valuable information about our planet's atmosphere and surface. However, this endeavor poses a computational challenge when dealing with multi-dimensional models.

One commonly employed technique to address the intricacies of multi-dimensional photon transport is Monte Carlo simulations [3]. These simulations, while powerful in their ability to yield reliable statistical data, require the consideration of a vast number of photons. In particular, according to the central limit theorem, the error of Monte Carlo simulations is proportional to the inverse square root of the number of photons that go into the result, and a huge number of photons are required to obtain accurate solutions in Monte Carlo

simulations. Despite this drawback, the Monte Carlo method has proved to be very efficient for solving the radiative transfer equation in inhomogeneous and broken clouds [4–6] as well as assessing the corresponding influence on retrieval results [7–9]. To accelerate these simulations and fully harness the capabilities of modern computing hardware, such as multi-core CPUs and GPUs, efficient parallelization techniques are essential. Monte Carlo simulation of photon transport in the atmosphere is a highly parallelizable problem, given that photons can be propagated independently while adhering to identical rules. The potential for achieving remarkable computational efficiency through GPU calculations using the Monte Carlo method was first demonstrated in early 2008 [10].

This research aims to contribute to the ongoing efforts to enhance the computational efficiency of photon transport simulations by conducting a comparative analysis of several simulation techniques. These techniques encompass traditional iterative loops, vectorized operations using the NumPy library, and the GPU-accelerated capabilities of CuPy [11].

In this pursuit, we explore the trade-offs and performance benefits of each approach, as well as assessing the adaptability of these techniques to existing Python code structures and the ease of transitioning from CPU-based calculations to GPU acceleration. Through empirical experiments, we demonstrate how these techniques impact simulation speed, particularly highlighting the substantial performance enhancement provided by CuPy when dealing with a significant number of photons. This investigation ultimately seeks to provide valuable insights into the optimization of photon transport simulations, with a focus on real-world applications in atmospheric science.

2. Monte Carlo Overview

We have implemented a basic Monte Carlo model in Python following the description given in [12]. The main steps of the algorithm can be summarized as follows.

2.1. Initialization

For each simulated photon, we begin by defining its initial propagation direction, which is characterized by three direction cosines (a, b, c) relative to the axes OX, OY, and OZ, respectively. In the context of a plane-parallel source, the photon's initial position is randomly selected from the top of the atmosphere.

2.2. Computation of the Step Size

Then, the mean free path (i.e. the step size) is computed as

$$l = -\tau \ln(\xi), \tag{1}$$

where τ is the optical thickness of the medium in the current location of the photon (being equal to the mean free path in this location), while ξ is a random number in range $(0, 1)$.

2.3. Photon Movement

Once a step size l and the direction cosines a, b and c are defined, the photon coordinates x, y and z are updated as follows:

$$x \leftarrow x + la, \tag{2}$$

$$y \leftarrow y + lb, \tag{3}$$

$$z \leftarrow z + lc. \tag{4}$$

2.4. Absorption and Scattering

With the probability equal to the single scattering albedo w, the photon can be absorbed. As suggested in [2], instead of killing the photon, its weight can be reduced by a factor of a single scattering albedo. Then, the scattering angle is computed from the single scattering

phase function. For scattering inside a cloud, we use the Henyey–Greenstein phase function. In this case, the scattering angle Θ can be chosen as

$$\cos\Theta = \frac{1}{2g}\left(1 + g^2 - \left(\frac{1-g^2}{1-g+2g\alpha}\right)^2\right), \tag{5}$$

where g is the asymmetry parameter, while α is a random number in range $(0,1)$. For scattering outside the cloud, we use the Reyleigh phase function. The corresponding scattering angle can be computed as

$$\cos\Theta = \sqrt[3]{z + \sqrt{z^2+1}} + \sqrt[3]{z - \sqrt{z^2+1}}, \tag{6}$$

where $z = 2(2\alpha - 1)$, and α is a random number in range (0,1). We define the direction before scattering $\omega' = (a', b', c')$ and after scattering $\omega = (a, b, c)$ through corresponding direction cosines. The azimuth angle φ (i.e., the angle between the planes (ω', OZ) and (ω', ω)) is chosen as a random number between 0 and 2π. The new direction can be computed using the following formulas:

$$a = a'\cos\Theta - (b'\sin\varphi + a'c'\cos\varphi)\sqrt{\frac{1-(\cos\Theta)^2}{1-c'^2}}, \tag{7}$$

$$b = b'\cos\Theta + (a'\sin\varphi - b'c'\cos\varphi)\sqrt{\frac{1-(\cos\Theta)^2}{1-c'^2}}, \tag{8}$$

$$c = c'\cos\Theta + \cos\varphi\sqrt{(1-(\cos\Theta)^2)(1-c'^2)}. \tag{9}$$

If $c' = \pm 1$, instead of Equations (7)–(9), we use the following formulas:

$$a = \sin\Theta\cos\varphi, \tag{10}$$

$$b = c'\sin\Theta\sin\varphi, \tag{11}$$

$$c = c'\cos\Theta. \tag{12}$$

When a photon reaches the surface, we employ the Lambertian model, which implies that the photon may be absorbed with a probability equal to the ground albedo. If the photon is not absorbed, it is instead reflected back into the upper hemisphere, and this reflection occurs with a uniform probability distribution across all possible angles.

2.5. Photon Termination

The photon travels through the atmosphere until it exits the atmosphere through reflection. Additionally, to expedite computations, we can terminate the photon's modeling when either its statistical weight becomes negligible (with the corresponding threshold determined experimentally) or its path length exceeds a specified limit.

3. Implementation

In this section, we provide outlines of certain implementation details that have been taken into account.

3.1. For-Loop Based Implementation

The aforementioned steps are implemented in a function having the following form:

```
def trace_photon(i):
    <initialization>
    while ((path_length<=max_path_length) and (not is_gone(x,y,z))):
        <determine the step size>
        <determine the scattering angles>
        <update the direction cosines>
```

```
        <update the position>
    return photon_parameters
```

that provides photon parameters as it leaves the atmosphere such as the path length and the direction cosines. Keeping that in mind, a custom implementation of the Monte Carlo algorithm that is based on the for-loop looks as follows:

```
photon_parameters_list = []
for i in range(number_of_photons):
    photon_parameters = trace_photon(i)
    photon_parameters_list.append(photon_parameters)
postprocess_data(photon_parameters_list)
```

In this simulation, photons are generated, and their behavior is simulated within a for-loop, while the collection and processing of statistical data occur after the loop has concluded.

3.2. Multithreading Implementation

A straightforward way to accelerate the computations on multicore CPUs is to use a multithreading library. The implementation looks as follows:

```
import multiprocessing
import os
num_processes = os.cpu_count()
with multiprocessing.Pool(processes=num_processes) as pool:
photon_indices = list(range(number_of_photons))  # Replace with the actual
    number of photons
photon_parameters_list = pool.map(trace_photon, photon_indices)
postprocess_data(photon_parameters_list)
```

Here, the pool of workers is created, and a list of photon indices is generated. The number of processes is equal to the number of available CPU cores. The pool is used to trace photons in parallel, resulting in a list of photon parameters.

3.3. NumPy Implementation

Vectorizing a Monte Carlo simulation code and using NumPy can significantly improve the efficiency of the code by taking advantage of NumPy's array operations and optimizations. In this case, Equations (1)–(12) have to be vectorized, and the whole batch of photons can be considered. For instance, a, b and c are not single values but rather numpy arrays for all photons in the batch.

```
def trace_photon_batch(N):
    <vectorized initialization for N photons>
    <determine the step sizes for N photons>
    <determine the scattering angles for N photons>
    <update the direction cosines for N photons>
    <update the positions for N photons>
    <check which photons escape the atmosphere>
    return photon_parameters_list
```

Note that certain photons may exit the atmosphere before others. To keep track of this, we employ a separate array that maintains information regarding whether a photon has already departed from the atmosphere. In such cases, photon parameters are no longer updated. The vectorized version avoids explicit loops across photons and instead utilizes masking. For example, considering Equation (2) and assuming that the array `is_gone` at the i-th position holds "True" if the i-th photon has escaped and "False" otherwise, the coordinate x is updated as follows:

```
# Update x where is_gone is False
x[~is_gone] += l[~is_gone]*a[~is_gone]
```

3.4. CuPy Implementation

After vectorizing the code to utilize NumPy arrays, transitioning to GPU acceleration using the CuPy library becomes straightforward. Essentially, the code remains unchanged,

except for replacing the import statement from NumPy to CuPy. For enhanced compatibility, CuPy can be imported as 'np,' allowing for a seamless integration of GPU capabilities into the existing codebase. This streamlined process empowers the code to harness the power of GPUs while maintaining code consistency and simplicity. We also check if GPU is available, and if that is the case, we substitute NumPy with CuPy.

```
import torch
import cupy
# Check if CUDA (GPU) is available
if torch.cuda.is_available():
    import cupy as np
else:
    import numpy as np
<vectorized Monte Carlo code for numpy>
```

4. Results and Discussion

In this section, we conduct an efficiency assessment of the implemented methodologies. Our testing scenario uses a three-dimensional box measuring 10 km in each dimension. At its center, we introduce a cloud layer with an optical thickness of 1 km and dimensions spanning 5 km along the x and y axes. The cloud's upper boundary is situated at 6 km above the ground. The simulations are performed for a wavelength of 350 nm and encompass ozone absorption effects, utilizing the ozone profile derived from the US standard atmosphere model.

The asymmetry parameter for the cloud phase function is specified as $g = 0.7$, with a cloud single scattering albedo of 0.9. Beyond the cloud region, we adopt the Rayleigh scattering phase function. The optical thickness of the cloud is set to 2.

In our simulation experiments, we use varying amount of photons, ranging from 10^3 to 10^9. To ensure the reliability of elapsed time measurements, we execute the simulations 10 times for scenarios with a relatively small photon count, computing the mean computation time. For CPU computations, we leverage an Intel Core i7 CPU running at 2.60 GHz with 12 cores, while GPU computations are conducted in a CoLab environment featuring the Tesla T4 GPU.

The results of these experiments are presented in Table 1. The elapsed time for the custom for-loop implementation exhibits a linear scaling relationship with the number of photons. When employing a multithreading library on 12 cores, we observe nearly an order of magnitude improvement in speed. The NumPy-based version outpaces the for-loop implementation by almost two orders of magnitude, establishing itself as the most efficient among those executed on the CPU. It is noteworthy that the impact of vectorization is an order of magnitude more efficient than the enhancement derived from explicit multiprocessing.

Interestingly, transitioning from NumPy to CuPy does not yield performance improvements for scenarios with fewer than 10^5 photons. This can be attributed to the overhead incurred when transferring data between the CPU and GPU. However, as the workload increases to $\geq 10^6$ photons, the CuPy implementation provides an order of magnitude speedup compared to the NumPy version and a three-order speedup compared to the for-loop implementation. It's worth noting that the NumPy and CuPy versions share the same code, with the only difference being the libraries they import.

Table 1. The computation time of Monte Carlo simulations.

Number of Photons	Elapsed Time, s			
	For-Loop	Multithreading	NumPy	CuPy
10^3	9	1.1	0.1	1.22
10^4	87	8.5	0.48	1.51
10^5	803	96	6.3	6.5
10^6	8187	125	71	7.9
10^7	—	—	690	62.6
10^8	—	—	6857	592
10^9	—	—	—	5743

5. Conclusions

This paper explores several strategies to accelerate Monte Carlo simulations of photon transport within Earth's atmosphere in the context of a Python-based Monte Carlo model, namely, traditional for-loop implementations, multithreading, vectorization using the NumPy library, and GPU acceleration with CuPy.

Vectorizing the code with NumPy surpasses the for-loop implementation by nearly two orders of magnitude in speed. However, the introduction of explicit multiprocessing via a multithreading library on a multi-core CPU yielded a speedup factor slightly below the number of cores. The transition from NumPy to CuPy, while initially incurring overhead for data transfer between CPU and GPU, proved to be highly advantageous for large-scale simulations. For workloads exceeding 10^6 photons, CuPy exhibited a one-order-of-magnitude speedup compared to NumPy, and a remarkable three-order speedup compared to the for-loop implementation. A notable advantage of CuPy lies in its ability to execute the NumPy version on the GPU without necessitating code changes while still achieving substantial performance enhancements. Note that a Fortran and C++ codes would require a substantial code refactoring while transitioning from CPU to GPU.

Author Contributions: Conceptualization, J.B., D.E. and T.T.; writing—original draft preparation, D.E.; writing—review and editing, J.B. and T.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Marshak, A.; Davis, A. (Eds.) *3D Radiative Transfer in Cloudy Atmospheres*; Springer: Berlin/Heidelberg, Germany, 2005. [CrossRef]
2. Mayer, B. Radiative transfer in the cloudy atmosphere. *Eur. Phys. J. Conf.* **2009**, *1*, 75–99. [CrossRef]
3. Marchuk, G.I.; Mikhailov, G.A.; Nazaraliev, M.A.; Darbinjan, R.A.; Kargin, B.A.; Elepov, B.S. *The Monte Carlo Methods in Atmospheric Optics*; Springer: Berlin/Heidelberg, Germany, 1980. [CrossRef]
4. Prigarin, S.M.; Marshak, A. A Simple Stochastic Model for Generating Broken Cloud Optical Depth and Cloud-Top Height Fields. *J. Atmos. Sci.* **2009**, *66*, 92–104. [CrossRef]
5. Zhuravleva, T.; Nasrtdinov, I. Simulation of Bidirectional Reflectance in Broken Clouds: From Individual Realization to Averaging over an Ensemble of Cloud Fields. *Remote Sens.* **2018**, *10*, 1342. [CrossRef]
6. Zhuravleva, T.; Nasrtdinov, I.; Chesnokova, T.; Ptashnik, I. Monte Carlo simulation of thermal radiative transfer in spatially inhomogeneous clouds taking into account the atmospheric sphericity. *J. Quant. Spectrosc. Radiat. Transf.* **2019**, *236*, 106602. [CrossRef]
7. Várnai, T.; Marshak, A. Observations of Three-Dimensional Radiative Effects that Influence MODIS Cloud Optical Thickness Retrievals. *J. Atmos. Sci.* **2002**, *59*, 1607–1618. [CrossRef]

8. Marshak, A.; Platnick, S.; Várnai, T.; Wen, G.; Cahalan, R.F. Impact of three-dimensional radiative effects on satellite retrievals of cloud droplet sizes. *J. Geophys. Res.* **2006**, *111*, 176. [CrossRef]
9. Efremenko, D.S.; Doicu, A.; Loyola, D.; Trautmann, T. Fast Stochastic Radiative Transfer Models for Trace Gas and Cloud Property Retrievals Under Cloudy Conditions. In *Springer Series in Light Scattering*; Springer International Publishing: Berlin/Heidelberg, Germany, 2018; pp. 231–277. [CrossRef]
10. Alerstam, E.; Svensson, T.; Andersson-Engels, S. Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration. *J. Biomed. Opt.* **2008**, *13*, 060504. [CrossRef] [PubMed]
11. Okuta, R.; Unno, Y.; Nishino, D.; Hido, S.; Loomis, C. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. In Proceedings of the Proceedings of Workshop on Machine Learning Systems (LearningSys) in the Thirty-First Annual Conference on Neural Information Processing Systems (NIPS), Long Beach, CA, USA, 4–9 December 2017.
12. Prigarin, M. *Modelling of Radiative Transfer by Means of Monte-Carlo Method*; Lambert Academic Publishing: London, UK, 2020. (In Russian)