Diese Arbeit wurde vorgelegt am
Lehr- und Forschungsgebiet Theorie der hybriden Systeme

Raytracer for Central Receiver Systems using GPU

Raytracer für Solarturmkraftwerke mit GPU

Masterarbeit
Informatik

Juni 2021

Vorgelegt von Presented by	Lukas Aldenhoff Römerstraße 311 46519 Alpen Matrikelnummer: 333731 lukas.aldenhoff@rwth-aachen.de
Erstprüfer First examiner	Prof. Dr. rer. nat. Erika Ábrahám Lehr- und Forschungsgebiet: Theorie der hybriden Systeme RWTH Aachen University
Zweitprüfer Second examiner	Prof. Dr. rer. nat. Matthias S. Müller Lehr- und Forschungsgebiet: Hochleistungsrechnen RWTH Aachen University
Fachlicher Betreuer Supervisor	Dr. rer. nat. Pascal Richter Lehr- und Forschungsgebiet: Theorie der hybriden Systeme RWTH Aachen University

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen meiner Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht. Dasselbe gilt sinngemäß für Tabellen und Abbildungen. Diese Arbeit hat in dieser oder einer ähnlichen Form noch nicht im Rahmen einer anderen Prüfung vorgelegen.

Alpen, im Juni 2021

Lukas Aldenhoff

Contents

1 Introduction

In the modern day surge of renewable energy investment and innovation, concentrating solar power (CSP) plants represent a promising link between clean energy production and high capacity energy storage in the form of thermal energy. The Gemasolar power plant in Spain, launched in 2011 by Torresol Energy, is an example of such a commercial solar tower plant with molten salt heat storage and 2650 heliostats, see Figure 1.

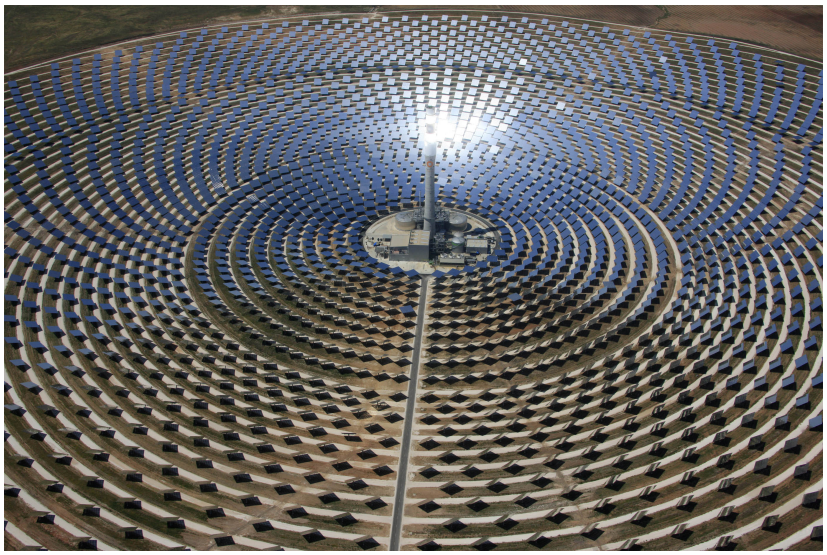


Figure 1: Solar tower power plant Gemasolar in the province of Seville, Spain. In operation since 2011, it involves a molten salt heat storage system and 2650 heliostats [9].

Maximizing the commercial viability of central receiver system (CRS) based CSP plants is tightly bound to the optimization of the heliostat field layout, which is a computationally challenging task. To fully optimize the heliostat field layout and other subsystems of a CRS plant an immense number of simulations have to be done. As future CRS plants are trending towards huge heliostat fields beyond ten thousand heliostats, this optimization becomes even more costly.
Todays technology offers many innovations to tackle computationally intensive tasks, however with single core frequency of CPUs stagnating for well over a decade, most of these innovations nowadays have to be exploited explicitly during implementation. The most glaring example is the increasing amount of cores on CPUs and the arising general purpose many-core architectures such as GPGPUs. Exploiting the computational power of these parallel architectures today is paramount in the quick solution of computationally intensive tasks. This leads to the requirement of efficiently parallelizing computation wherever possible.
The vast majority of computation during the simulation of a CRS plant is required for the evaluation and tracing of sun rays. For each heliostat a ray tracing module approximates the amount of solar power that is successfully redirected to the cen-

tral receiver. This is typically achieved by discretizing sun rays and evaluating their trajectory. With this methodology a high discretization resolution is key to achieve reasonable accuracy, which leads to millions or even billions of rays being traced in common settings. As rays can be calculated and traced independently, ray tracing has immense parallelization possibilities, uniquely suitable to the powerful data parallelism of the GPU architecture.

This work aims to accelerate the existing *SunFlower* tool for simulation and optimization of CRS plants, initially introduced by Richter et al. [43]. The goal is to improve the incorporated ray tracing methodology and to efficiently parallelize the computationally intensive ray tracing module.

1.1 State of the art

The simulation and optimization of solar thermal power plants has been topic of research since the 1970s, resulting in various tools, approaches and innovations in this field. There also have been a number of reviews on the range of available tools, eg. by Garcia et al. [18], Bode and Gauché [8], Cruz et al. [13] and most recently Jafrancesco et al. [27]. Behar et al. [4] and Li et al. [31] also have done a more general review of studies and the state of the art in this field. Given the comprehensive available literature summarizing the state of the art, this Section only gives a brief overview and focusses on more recent studies.

There are two predominant approaches to the simulation of the produced flux on the receiver by a heliostat field. The first and most common approach is the Monte-Carlo ray tracing method, pseudo-randomly generating large amounts of rays and tracing them to approximate the error functions encountered in a CSP environment.
The tool *MIRVAL* [30], launched by the Sandia National Laboratory in 1979, is considered as one of the first Monte-Carlo based raytracers and today is available commercially under the name SPRAY via the German Aerospace Center (DLR).
In 2003 the US National Renewable Energy Laboratory (NREL) launched the free Monte-Carlo ray tracing tool *SolTrace* [49] and since improved it considerably [50], making it a well established state of the art tool.
Another free and open-source Monte-Carlo raytracer was released in 2007 by the University of Texas under the name *Tonatiuh* [7]. *MIRVAL*, *SolTrace* and *Tonatiuh* compute the flux distribution on the receiver with a standard forward ray tracing procedure, generating rays on a plane above the heliostat.
The DLR incorporated a more efficient backward ray tracing procedure in their *STRAL* tool [1], originating rays directly on the reflective surface (the heliostat).
TieSOL [26] represents the first established GPU implementation of Monte-Carlo ray tracing for CSP plants in 2011. It is developed and commercially available by Tietronix Software Inc. and showcased the potential performance of highly parallelized GPU-based ray tracing.
Another implementation of Monte-Carlo ray tracing was done by Duan et al. [15],

named Quasi-Monte-Carlo ray tracing (*QMCRT*). This implementation uses bidirectional ray tracing and limits the random generation of Monte-Carlo rays, instead using two precomputed random variable sets. *QMCRT* was also implemented on GPU with impressive performance.
A very recent commercial and on GPU parallelized Monte-Carlo raytracer is *sbpRAY*. Developed by Schlaich Bergermann Partner and introduced in 2019, this tool can also be used for heliostat field layout optimization and is extremely fast due to its GPU parallelization.

The second approach to the flux approximation is of analytical nature and bases on numerical convolution of the sun irradiation to avoid the generation of large amounts of rays as in Monte-Carlo methods, instead estimating errors statistically. This allows for very quick and efficient simulation of annual power generation at the cost of further simplifications to the optical model.
An early implementation of this approach can be found in the tool *HELIOS* [5], developed at the Sandia National Laboratory in the late 1970s. This tool is not available anymore.
The *HFLCAL* tool [46], first developed in the early 1980s at Interatom and since acquired and improved by the DLR, is well established and also makes use of this analytical approach.
During the last decade the methodology incorporated in *HFLCAL* was picked up and improved several times. It was revised by García et al. [17] in 2015 and this revised model was picked up by He et al. [22] to implement it in a GPU rendering pipeline. Most recently this implementation was again improved in the *iHFLCAL* model [24], which again reports remarkable performance on GPU.

While this is by no means an exhaustive list of tools and studies, it is clear that the landscape of research around the optical simulation of the heliostat field is well saturated. However, most of these modern tools only offer simulation of heliostat field layouts without field optimization [13]. They also do not allow simulation of any other sub-system of the CSP plant such as the thermal receiver, storage or power block. Especially the new and well performing tools are often highly specialized or oversimplified and lack general purpose usability.
Therefore this landscape still leaves much to be desired when looking for a state-of-the-art, versatile and powerful tool to simulate and optimize central receiver systems.

1.2 Outline

This work will discuss and implement a number of techniques to accelerate the optical simulation and optimization of heliostat fields. This includes methodological improvements as well as optimization and parallelization of the code. The tool SunFlower, which is used as basis for this work, and the therein incorporated optical model will be introduced in Section 2. The following Section 3 will explain the in SunFlower implemented ray tracing methodology including the Monte-Carlo and Convolution tech-

niques to handle optical errors. Next, in Section 4, the current SunFlower code will be evaluated using the Intel VTune Profiler and optimizations to methodology and code will be described. Adding to these optimizations, Section 5 will present how the most computationally intensive parts of the optical simulation can be parallelized on both CPU and GPU. In case of the GPU parallelization, kernel performance will be analysed with the NVIDIA Nsight Compute profiler and a number of optimizations will be layed out. Section 6 will investigate the impact of the optimizations and code parallelization and will compare the final versions of both raytracers on CPU and GPU respectively. The performance of the final GPU implementation will additionally be compared to other state-of-the-art GPU parallelized raytracers. After discussing the results of the study in Section 6.4, the main findings of this work will be summarized in Section 7.1. Finally Section 7.2 will give an outlook on possible future research.

2 Optical model

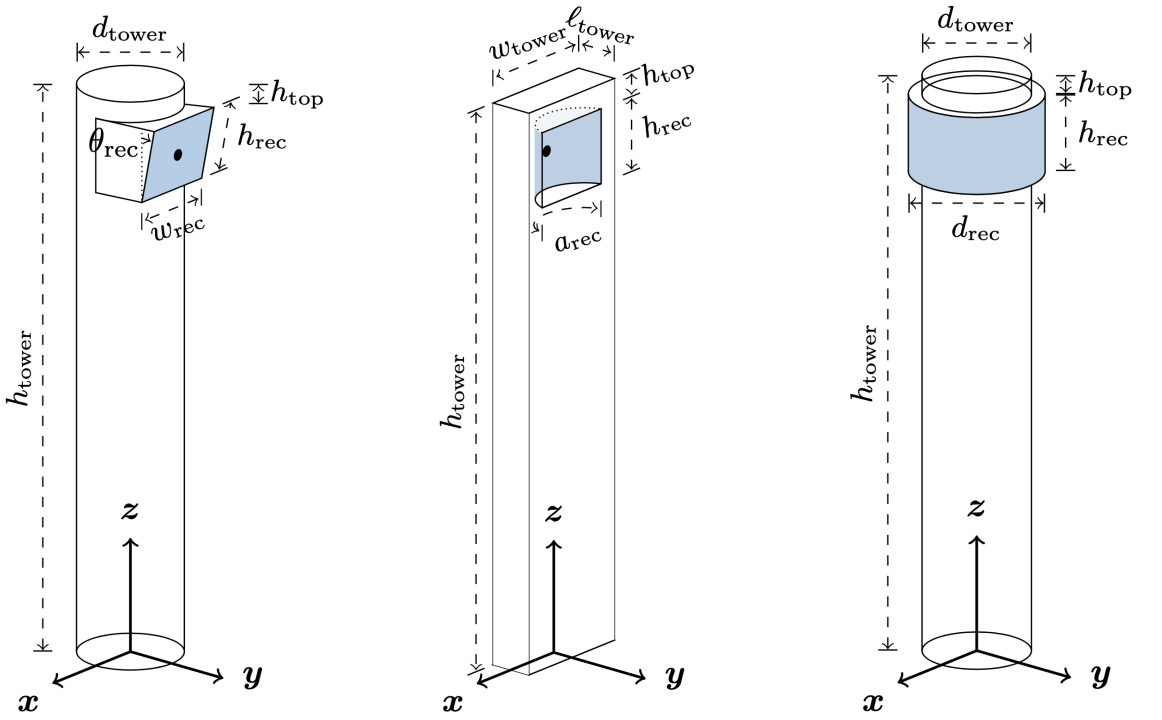
The simulation tool this work bases on is called *SunFlower*, is developed at RWTH Aachen University and was first introduced by Richter et al. [43] in 2018. SunFlower allows simulation and optimization of the heliostat field layout and a full simulation of the optical and thermal system of central receiver power plants. The optical simulation is able to calculate efficiencies for individual heliostats as well as the flux on the receiver at a given moment or throughout a full year.
This Section will introduce the optical model of SunFlower before the ray tracing methodology is presented in the following Section.
For the simulation of heliostat field flux distributions a sophisticated optical model is needed that precisely defines the interacting sub-systems. The herein described model is adapted from the model presented by Richter et al. [43, 42], Franke [16] and Hoevelmann [25].
In Section 2.1 the modelling of tower and receiver will be explained. Section 2.2 briefly describes considerations about the plant side. The following Section 2.3 elaborates on the modelling of the sun and its localization. Finally, Section 2.4 will illustrate the heliostat model.

2.1 Tower and receiver

To define position and expansion of sub-systems a global cartesian coordinate system is placed at the position of the solar tower. The z-axis points into the sky and x-axis and y-axis point towards East and North respectively. The position of the tower is given as latitude ϕ and longitude θ.
The tower in this model is assumed to be cylindric, has a total height of h_{tower} and diameter of d_{tower}. The receiver is installed close to the top of the tower with the distance from the top of the receiver to the top of the tower denoted as h_{top}.
We distinguish three receiver types:

- **Flat tilted receiver:** rectangular shaped flat cavity receiver, tilted down by angle θ_{rec}, with height h_{rec} and width w_{rec}.
- **Internal cylindric receiver:** cavity receiver that is embedded with a window into the tower as rough half-cylinder. It is defined by height h_{rec} and the arc length a_{rec} of the cut cylinder. The receiver is raised by h_{raise} from the lower horizontal edge of the window to avoid blocking incoming rays. The PS10 power plant in Spain uses this type of receiver.
- **External cylindric receiver:** full cylinder that encloses part of the tower on 360°. It again can be defined by height h_{rec} and diameter d_{rec}. An example of this receiver type can also be found in Spain, at the Gemasolar power plant.

A sketch of towers with each of these receiver types can be found in Figure 2 with Figure 2a showing a flat tilted cavity receiver, Figure 2b showing an internal cylindric



(a) Flat tilted cavity receiver (b) Internal cylindric receiver (c) External cylindric receiver

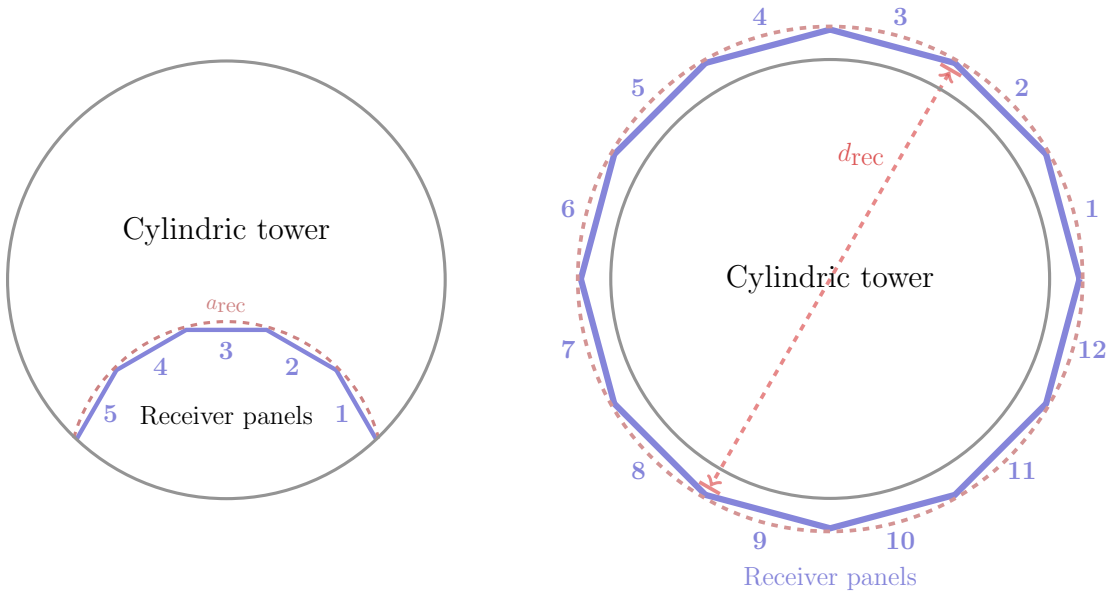
Figure 2: Sketch of receiver types [16].

receiver and Figure 2c an external cylindric receiver.
The cylindric receiver types consist of a number of identical flat receiver panels arranged around the theoretical receiver arc. They can be described as edges of a regular polygon. Figure 3a skteches the internal cylindric receiver and Figure 3b the external cylindric receiver type.

The shown panels are further partitioned into smaller pieces for the simulation to be able to more accurately approximate the flux distribution on each panel.

2.2 Site

The boundaries of the power plant site are defined as vertices of a polygon. Any additional restricted area within the bounding polygon is again stated as polygon vertices. The topography of the site is considered as elevation data which can be specified in a cell format and is interpolated for exact positions. For any heliostat placed on the site it then is checked whether it fits inside the polygon and whether it crosses any restricted area. If the heliostat position is valid it is elevated according to the given elevation data.



(a) Internal cylindric receiver with five panels (b) External cylindric receiver with twelve panels

Figure 3: Two dimensional sketch of cylindric receivers with identical flat panels arranged according to a regular polygon. The viewpoint is on the z-axis.

2.3 Sun

For the optical simulation we need to model the sun position and irradiation. The sun position is defined by the sun angles azimuth γ_{solar} and altitude θ_{solar}. The azimuth γ_{solar} is measured clockwise starting on the y-axis (North) and altitude θ_{solar} is measured from the earth's surface upwards. With that we can calculate the sun vector $\vec{\tau}_{\text{solar}}$ as

$$\vec{\tau}_{\text{solar}} = \begin{pmatrix} -\cos(\theta_{\text{solar}}) \cdot \sin(-\gamma_{\text{solar}}) \\ \cos(-\gamma_{\text{solar}}) \cdot \cos(\theta_{\text{solar}}) \\ \sin(\theta_{\text{solar}}) \end{pmatrix}. \tag{1}$$

The sun's irradiation can be calculated with the clear sky model MRM [2] or taken from standardized measurement data, eg. from the weather data provided by EnergyPlus. We only consider the direct normal irradiation I_{DNI} as diffuse radiation can not be concentrated.

2.4 Heliostat

The position for any heliostat i is specified in (x, y) coordinates. Additionally all heliostats are raised on a pedestal of height h_{ped} which also is elevated according to the topography as mentioned in Section 2.2. With this we have the global position p_i of the heliostat as (x, y, z) coordinates.
Each heliostat typically consists of multiple smaller mirrors, called facets. In our model facets can be triangular or rectangular, constructing either polygonal or rectangular

heliostats. Individual facet position and alignment is calculated and stored relative to a local coordinate system for each heliostat. The total mirror area A_i of the heliostat can be calculated as the sum of the facet areas. This does not necessarily equal the total heliostat area as there can be gaps between facets. Therefore the maximum expansion for the polygonal heliostat is then given by the diameter d_{Circ_H} of the cirumscribed circle of the polygon and for the rectangular heliostat by the diagonal d_H of the heliostat.
Given the incoming sun vector $\vec{\tau}_{\text{solar}}$ and the aim point p_i^{aim} for the heliostat i, we can calculate the normalized ideal reflective vector $\vec{r}_i$ as

$$\vec{r}_i = \frac{p_i^{\text{aim}} - p_i}{||p_i^{\text{aim}} - p_i||}. \tag{2}$$

With $||\cdot||$ in this case referring to the euclidean norm of the vector.
Next we calculate the heliostat normal $\vec{n}_i$ such that the above reflective vector $\vec{r}_i$ is given for the incoming sun vector $\vec{\tau}_{\text{solar}}$ as

$$\vec{n}_i = \frac{\vec{r}_i + \vec{\tau}_{\text{solar}}}{||\vec{r}_i + \vec{\tau}_{\text{solar}}||}. \tag{3}$$

Therefore aligning Heliostat i with normal vector $\vec{n}_i$ will aim the reflected sun ray at the given aim point p_i^{aim}. The calculation of the sun ray trajectory $\vec{\tau}_{\text{solar}}$ was shown in equation 1. It remains to determine the aim point p_i^{aim}. Here we again have to distinguish the receiver types. For flat tilted receivers we simply take the center of the aperture p_{rec}^{center} which in case of a north facing receiver on a cylindric tower is calculated as

$$p_{rec}^{\text{center}} = \begin{pmatrix} 0 \\ \frac{d_{\text{tower}}}{2} + \frac{h_{\text{rec}}}{2} \cdot \sin(\theta_{\text{rec}}) \\ h_{\text{tower}} - h_{\text{top}} - \frac{h_{\text{rec}}}{2} \cdot \cos(\theta_{\text{rec}}) \end{pmatrix}. \tag{4}$$

For receiver facing directions deviating from the y-axis (north), the coordinates just have to be rotated around the z-axis.

In case of the cylindric receivers the determination of the aim point has to be more careful. Previous work assumed an aiming point at the center of the aperture for internal cylindric receivers. Taking the center of the back wall leads to problems for heliostats on the outside of the field, as can be seen in Figure 4a. Rays reflected by heliostats on the far outside would often be blocked by the tower. Similarly, taking the theoretical center of the internal receiver would lead to problems for heliostats close to the tower, as can be seen in Figure 4b. Here, rays would often be spilled above the receiver because of the steep angle.
To avoid these problems, a simple new aim point strategy is used. As the cylindric receivers consist of multiple flat panels, as mentioned in Section 2.1, we can take the center of a specific panel j, denoted as $p_{\text{rec}_{P_j}}^{\text{center}}$, as aim point. The decision of which panel j to take depends on the receiver type. Given an internal cylindric receiver, we

take the panel that is *farthest away* from the heliostat. For external cylindric receivers we instead take the panel that is *closest* to the heliostat. This concept applied to the exemplary situation of Figure 4a is illustrated in Figure 5a. Figure 5b shows the choice of aiming points for external cylindric receivers. The calculation of the coordinates of the cylindric receiver panel centers is omitted.

Besides the alignment of the overall heliostats according to their aim points, there is also the possibility to align heliostat facets individually to minimize spillage. The alignment of individual facets is called **canting**. We differentiate two approaches: *on-axis* and *off-axis* canting. On-axis canting assumes the sun, receiver and heliostat to be perfectly aligned on one common axis. Using a paraboloid with the given axis as symmetry axis and vertex at the heliostat center, the facets are then positioned around the symmetry axis.
Off-axis canting uses a specified sun vector that deviates from the axis going through heliostat and receiver. A similar paraboloid is used, however the facets are now positioned on the side of the paraboloid.
To make facet alignment easier, each heliostat uses a local coordinate system, which we define using the heliostat normal vector $\vec{n}_i$ as local z-axis. The remaining x and y-axis are then defined as cross product with the global z-axis $(0,0,1)^T$ as

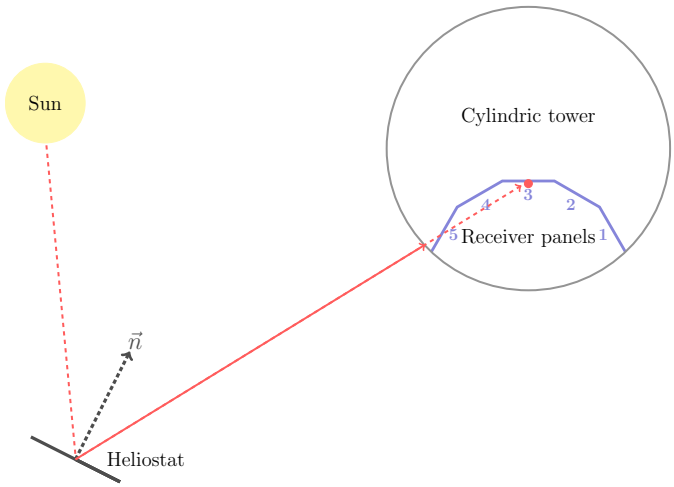
$$\vec{x}_i = \frac{\vec{n}_i \times (0,0,1)^T}{||\vec{n}_i \times (0,0,1)^T||}, \qquad \vec{y}_i = \vec{x}_i \times \vec{n}_i. \tag{5}$$

For a heliostat normal vector $\vec{n}_i$ almost equal to the global z-axis, we have to take another approach to avoid hazards. In this case we instead take the normalized vector from the on the global z-axis projected heliostat center $(p_i^x, p_i^y, 0)^T$ to the global coordinate origin as local y-axis and calculate the x-axis as cross product:

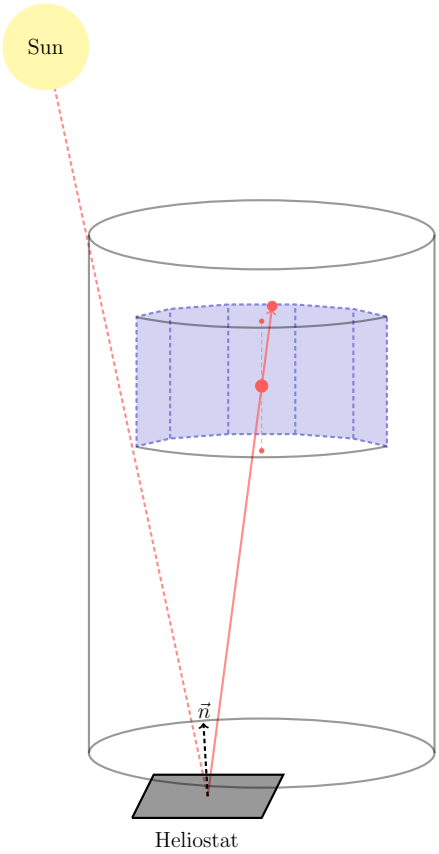
$$\vec{y}_i = \frac{(-p_i^x, -p_i^y, 0)^T}{||(p_i^x, p_i^y, 0)^T||}, \qquad \vec{x}_i = \vec{y}_i \times \vec{n}_i. \tag{6}$$

Furthermore we define a coordinate system for each heliostat facet which will be used in the optical error considerations of the next Section. The approach is completely analogous, however each facets coordinate system is defined relative to that of its heliostat. After canting each facet j may have a surface normal vector $\vec{n}_{i,j}$ that differs from the surface normal $\vec{n}_i$ of the heliostat.

Using this optical model, we now can discuss the ray tracing methodology that will be the focus of the optimization and acceleration done in this work.

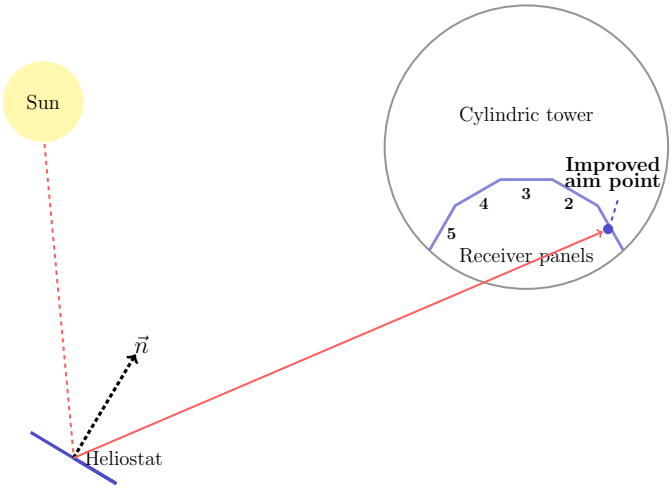


(a) Problematic situation with aim point at center of the back wall of internal cylindric receivers

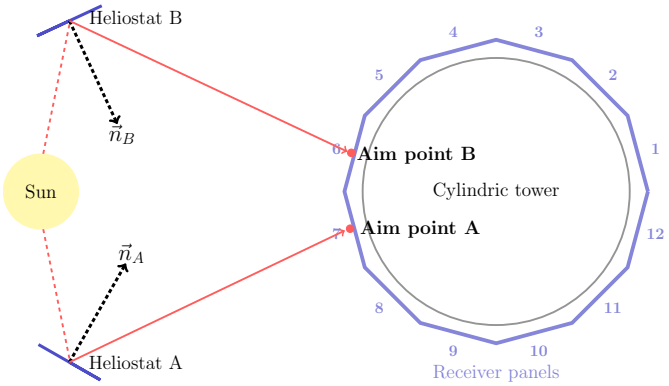


(b) Problematic situation with aim point at theoretical center of internal cylindric receivers

Figure 4: Hazards encountered with aim point strategies for internal cylindric receivers



(a) New aim point strategy for internal cylindric receivers: aim at center of farthest away receiver panel



(b) Aim point strategy for external cylindric receivers: aim at center of closest receiver panel

Figure 5: Aim point strategies for cylindric receivers

3 Ray tracing methodology

This Section will introduce the in SunFlower incorporated ray tracing methodology. First the consideration of optical errors is established in Section 3.1. The two available methods to handle optical errors, the Monte-Carlo method and the analytical Convolution method, are presented in sections 3.1.1 and 3.1.2 respectively.
Subsequently the general ray tracing procedure is described, beginning with the ray generation in Section 3.2. Finally, in sections 3.3 and 3.4 the handling of blocking and shading effects and optical losses is discussed. The closing subsection 3.5 will summarize the procedure and combine the considerations of this Section to calculate the power of a ray.

3.1 Optical errors

In the optical environment of CSP power plants there are multiple sources for optical inaccuracies. Due to necessary abstractions optical models of CSP plants typically are not able to implicitly model these optical errors and as such they have to be modeled explicitly.
We consider three main sources for optical errors in this model:

Heliostat tracking error

The vertical and horizontal alignment of heliostats in reality is inaccurate and introduces a slight deviation from the wanted heliostat normal vector $\vec{n}_H$. This error can be modeled as gaussian distribution and vertical and horizontal errors are calculated independently, hence we distinguish the two standard deviations $\sigma^{\text{ver}}_{\text{track}}$ and $\sigma^{\text{hor}}_{\text{track}}$.

Heliostat surface error

The actual mirror surface is not perfectly smooth, instead having very slight irregularities. This error deviates for different parts of the mirror surface and also may deviate horizontally and vertically. Therefore we model this error with a gaussian distribution and use two matrices of standard deviations $\sigma^{\text{ver}}_{\text{slope}} \in \mathbb{R}^{m\times n}$ and $\sigma^{\text{hor}}_{\text{slope}} \in \mathbb{R}^{m\times n}$. The matrix entries then are mapped to the mirror surface and the appropriate standard deviations $\sigma^{\text{ver}}_{\text{slope}}(x,y)$ and $\sigma^{\text{hor}}_{\text{slope}}(x,y)$ are used for calculations at location (x,y).

Sun error

As the sun is modeled as two-dimensional plane while being a three-dimensional sphere, actual ray directions may deviate from modeled directions. Using the proposal of Rabl [40], we again model this error as gaussian distribution with standard deviation σ_{sun}.

Note that we will use the heliostat facet x-axis as horizontal axis and the y-axis as vertical axis.
Given these three gaussian distributions we make use of the Central Limit Theorem

which predicts the sum of multiple gaussian distributions to again be a gaussian distribution. Thus, we model the above mentioned errors for a given location (x, y) on the mirror surface with one vertical and one horizontal gaussian distribution with standard deviations calculated as

$$\begin{aligned}\sigma_{\text{ang}}^{\text{ver}} &= \sqrt{\left(\sigma_{\text{track}}^{\text{ver}}\right)^2 + \left(\sigma_{\text{slope}}^{\text{ver}}(x,y)\right)^2 + \left(\sigma_{\text{sun}}\right)^2},\\ \sigma_{\text{ang}}^{\text{hor}} &= \sqrt{\left(\sigma_{\text{track}}^{\text{hor}}\right)^2 + \left(\sigma_{\text{slope}}^{\text{hor}}(x,y)\right)^2 + \left(\sigma_{\text{sun}}\right)^2}.\end{aligned} \tag{7}$$

These gaussian distributions then give a statistical view on the angle of deviation of actual rays from ideal rays in vertical and horizontal direction respectively. This can also be described as the standard deviation length for a ray of unit length with the tangent function:

$$\begin{aligned}\sigma_{\text{len}}^{\text{ver}} &= \tan\left(\sqrt{\left(\sigma_{\text{track}}^{\text{ver}}\right)^2 + \left(\sigma_{\text{slope}}^{\text{ver}}(x,y)\right)^2 + \left(\sigma_{\text{sun}}\right)^2}\right),\\ \sigma_{\text{len}}^{\text{hor}} &= \tan\left(\sqrt{\left(\sigma_{\text{track}}^{\text{hor}}\right)^2 + \left(\sigma_{\text{slope}}^{\text{hor}}(x,y)\right)^2 + \left(\sigma_{\text{sun}}\right)^2}\right).\end{aligned} \tag{8}$$

We now want to incorporate this error information into our ray calculations, which in Section 2 assumed ideal rays.
In the following the two widely used approaches to this problem are explained, beginning with the Monte-Carlo method.

3.1.1 Monte-Carlo method

The Monte-Carlo method is a probabilistic approach to a wide range of numerical problems. It estimates numerical results with a simple computational algorithm based on repeated random sampling. Although this method is easy to use and well suited to solve problems with probabilistic distribution, it tends to be quite slow as accurate and reliable results can only be guaranteed with the law of large numbers. This means, depending on the application, the number of random samplings needed to obtain a representative result can be enormous [10].
In ray tracing the Monte-Carlo method is useful to realistically model the impact of optical errors that can be described with probabilistic distributions. Moreover it is very robust and easily adaptable for any environment, which can be difficult for analytical methods. Thus, many modern ray tracing tools use the Monte-Carlo method, as mentioned in Section 1.1.

Similarly, we use Monte-Carlo to incorporate the above mentioned gaussian distributed optical errors by sampling perturbed reflected rays. The deviation length of the reflected ray in horizontal direction $\delta_{\text{len}}^{\text{hor}}$ and in vertical direction $\delta_{\text{len}}^{\text{ver}}$ is determined by evaluation of the respective gaussian distributions with a random variable. To obtain the directions of each respective error the horizontal and vertical axes of the heliostat facet are rotated towards the ideal reflected ray. Figure 6 sketches this rotation.

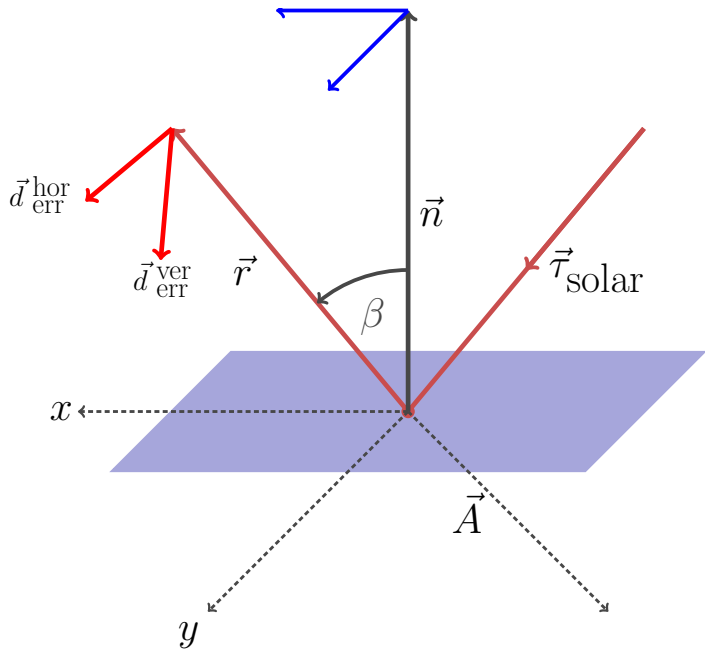


Figure 6: Sketch of the rotation of the heliostat facet axes onto the ideal reflected ray. They are rotated around rotation axis $\vec{A}$ by angle β to obtain the ray error directions $\vec{d}^{\,\mathrm{hor}}_{err}$ and $\vec{d}^{\,\mathrm{ver}}_{\mathrm{err}}$.

The rotation axis $\vec{A}$ for the ideal reflected ray $\vec{r}$ and the heliostat facet normal $\vec{n}$ is given by their cross product

$$\vec{A} = \vec{r} \times \vec{n}. \tag{9}$$

And the rotation angle β is obtained with their dot product

$$\beta = \arccos(\vec{r} \cdot \vec{n}). \tag{10}$$

Rotating the heliostat facets x and y axes around axis $\vec{A}$ by angle β we obtain the error directions $\vec{d}^{\,\mathrm{hor}}_{err}$ and $\vec{d}^{\,\mathrm{ver}}_{\mathrm{err}}$. Now the deviation of the ray in each direction can be calculated with the above mentioned respective deviation lengths δ_{len} as

$$\begin{aligned} \vec{r}^{\,\mathrm{hor}}_{\mathrm{err}} &= \delta^{\mathrm{hor}}_{\mathrm{len}} \cdot \vec{d}^{\,\mathrm{hor}}_{\mathrm{err}}, \\ \vec{r}^{\,\mathrm{ver}}_{\mathrm{err}} &= \delta^{\mathrm{ver}}_{\mathrm{len}} \cdot \vec{d}^{\,\mathrm{ver}}_{\mathrm{err}}. \end{aligned} \tag{11}$$

And with that the perturbed ray is given as

$$\vec{r}_{\mathrm{err}} = \vec{r} + \vec{r}^{\,\mathrm{hor}}_{\mathrm{err}} + \vec{r}^{\,\mathrm{ver}}_{\mathrm{err}}. \tag{12}$$

Finally, the perturbed ray is ready to be evaluated. When it passes the blocking and shading check the raytracer verifies it hits the receiver and flux corresponding to the

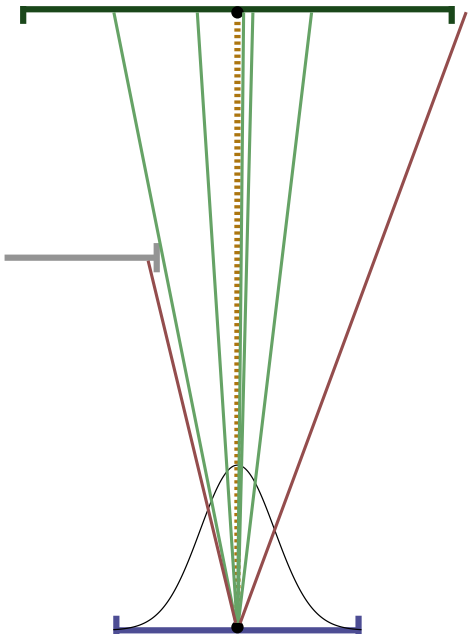


Figure 7: Repeated random sampling of the perturbation of a single ray according to an exemplary gauss distribution. The original ideal ray is illustrated by the dashed orange line. Receiver hitting rays are colored in green, blocked and missing rays are colored in red. Only five out of seven perturbed rays hit the receiver successfully. One ray is blocked by another heliostat (in grey) and one ray misses the receiver.

heliostat facet area is added to the receiver. This procedure and further considerations will be explained in detail beginning in Section 3.2.
Following the Monte-Carlo idea of repeated random sampling, each ray that is generated by the simulation may be evaluated N times with a new random ray perturbation being applied each time according to the above procedure. The flux that is added to the receiver then depends on the number of samplings successfully hitting the receiver. For k out of N rays hitting the receiver successfully, the added flux would be scaled by $\frac{k}{N}$. This is done by scaling the power of each sampling of the ray. The exact calculation of a rays power is given in Section 3.5, once all necessary considerations are layed out. This is one way in which we are able to trade additional computation time for higher simulation accuracy by increasing N or in other words by increasing the number of repeated random samplings we do. Figure 7 illustrates the repeated random sampling of a single ray according to a gauss distribution. In the shown example only five out of seven samplings of the ray successfully hit the receiver, hence the added flux is scaled by $\frac{5}{7}$.

SunFlower also implements a Quasi-Monte-Carlo method which bases on predefined sequences rather than random numbers. The idea is to avoid the generation of large amounts of random numbers and create a uniform set of samplings with fewer evalua-

tions, possibly accelerating convergence.

In contrast to the easy to use Monte-Carlo method which offers great accuracy at the expense of high computational costs, the second common approach to handling optical errors in Convolution based methods is more complex. The idea of this approach and its implementation in SunFlower is presented briefly in the following Section.

3.1.2 Convolution method

Convolution based methods generally seek to determine for each receiver piece the probability P_{hit} that an ideal reflected ray is perturbed such that it hits this piece. More specifically, they try to approximate the flux density that a ray produces as mathematical function. For each receiver piece this function then is integrated for an area representative of the receiver piece.
As the exact derivations necessary for the in SunFlower implemented Convolution methods are quite lengthy and not subject of the optimizations discussed in the later sections, this Section will only give an overview on the idea of and general approach to the Convolution methods. A detailed derivation can be found in the work of Hoevelmann [25].

The probability P_{hit} is defined using the probability density function $f(x, y)$, corresponding to the bivariate gaussian distribution of our optical errors in horizontal and vertical direction, given by

$$f(x,y) = \frac{1}{2\pi\sigma_{\text{ang}}^{\text{hor}}\sigma_{\text{ang}}^{\text{ver}}} \cdot \exp\left(-\frac{1}{2}\left(\frac{x^2}{{\sigma_{\text{ang}}^{\text{hor}}}^2} + \frac{y^2}{{\sigma_{\text{ang}}^{\text{ver}}}^2}\right)\right). \tag{13}$$

Integrating this function over a region D then gives the probability $P_{\text{int}}(D)$ that the ray intersects this region. Choosing D as representative region D_{ang} of our receiver piece, this probability is equivalent to the probability P_{hit} of the ray hitting the receiver.
The receiver region D_{ang} is defined by the angles α_i in horizontal and β_i in vertical direction of each corner of the receiver piece relative to the perfect reflected ray direction $\vec{r}$. Encorporating computer graphics methodology, the probability density function $P_{\text{int}}(D)$ and the receiver region D_{ang} can be adapted to instead work on the spanned region D_{span} of the receiver on the ray image plane.
The evaluation of the integral $P_{\text{int}}(D)$ is based on the in 1978 depicted numerical approach by Didonato et al. [14]. The main idea of this approach is to integrate over the to D complementary region $\overline{D}$. This is feasible as for a probability density function $f(x, y)$ we know it holds that

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)\,dx\,dy = 1. \tag{14}$$

Therefore we can derive that

$$P_{\text{int}}(D) = 1 - P_{\text{int}}(\overline{D}). \tag{15}$$

The integral $P_{\text{int}}(\overline{D})$ then is evaluated by partitioning $\overline{D}$ into four semi-infinite angular regions corresponding to the four corners of the region D and approximating the four individual integrals numerically, see [25, 14]. Applying this method to the receiver region defined by D_{ang} or D_{span} gives us the desired probability P_{hit}.
While further derivations are omitted, a summary of the implemented integration of a two dimensional region D with error terms σ_{hor} and σ_{ver} can be found in algorithm 1 and 2, which were adopted from [25]. The used polynomial (P_3) for the angular region integration can be replaced to slightly improve accuracy at the cost of additional computation.

With its analytical approach, the Convolution method in contrast to the Monte-Carlo method gives deterministic results and offers higher accuracy for the same amount of rays. However, the mathematical complexity of the method introduces new computational intensity, making the evaluation of each ray more expensive.

3.2 Ray generation

The first step of the ray tracing procedure is to determine the rays we trace. For this, we generate rays directly on the heliostat surface instead of on a plane above the heliostat. This saves computational effort as significantly less rays have to be evaluated. For each heliostat facet j on heliostat i we may generate multiple rays by partitioning the facet into M smaller cells for which we each generate one ray. The ray origin is localized at the center of the cell k, given as $p_{i,j,k}^{\text{center}}$. The power of the ray is corresponding to the area $A_{i,j,k}$ of its respective cell. Note that this single ray may be sampled multiple times when using the Monte-Carlo method, as explained in Section 3.1.1.
The uniform partitioning of rectangular heliostat facets into smaller cells is illustrated in Figure 8. The sum of the cell areas $\sum_{k=1}^{k\leq M} A_{i,j,k}$ always equals the total mirror area $A_{i,j}$ of the heliostat facet.
With the Convolution method and before perturbation in the Monte-Carlo method, the generated rays for one facet all share the same direction according to the ideal reflected ray for that facet. The ray direction may differ for individual facets as their surface normal may differ after canting, see Section 2.4.

Given the localization of the ray on the heliostat, we first do a blocking and shading check before tracing the ray against the receiver.

3.3 Blocking and shading

To verify that a ray successfully travels from the sun to the given point $p_{i,j,k}^{\text{center}}$ on the heliostat surface and then is reflected to the receiver without being blocked, we perform a blocking and shading check before actually tracing the ray against the receiver surface. We test for two shading and one blocking effect:

Algorithm 1 Bivariate Polygon Integration

function integrateGaussianPolygon(D, σ_{hor}, σ_{ver})
  if not isSimplePolygon(D) **then**
    return -1 ▷ Error, Polygon has to be simple
  if isClockwiseOriented(D) **then**
    $D :=$ reverse(D)
  $D' :=$ scaleToCircular($D, \sigma_{hor}, \sigma_{ver}$)
  $res := 0$
  for each vertex V_i in D' **do**
    $R := |V_i|$
    if $R == 0$ **then**
      $res \mathrel{+}= \arccos\left((v_1 \cdot v_2)/(2\pi)\right)$
      continue
    $v_1 := (V_i - \text{predecessor}(V_i))/|V_i - \text{predecessor}(V_i)|$
    $v_2 := (\text{successor}(V_i) - V_i)/|\text{successor}(V_i) - V_i|$
    $l := V_i/|V_i|$
    $\theta_1 := \text{sign}(\det(v_1,\, l)) \cdot \arccos\left(v_1 \cdot l\right)$
    $\theta_2 := \text{sign}(\det(v_2,\, l)) \cdot \arccos\left(v_2 \cdot l\right)$
    $g_1 := \cos\left(\theta_1\right) R/\sqrt{2}$
    $g_2 := \cos\left(\theta_2\right) R/\sqrt{2}$
    $h_1 := \sin\left(\theta_1\right) R/\sqrt{2}$
    $h_2 := \sin\left(\theta_2\right) R/\sqrt{2}$
    if $g_1 \geq 0$ and $g_2 \geq 0$ **then**
      $res \mathrel{+}=$ integrateAngularRegion$(R, \theta_1, \theta_2, g_1, g_2, h_1, h_2)$
    else
      $\theta_1 \mathrel{+}= (g_1 < 0) \cdot \pi \cdot (1 - 2 \cdot (\theta_1 > 0))$
      $\theta_2 \mathrel{+}= (g_2 < 0) \cdot \pi \cdot (1 - 2 \cdot (\theta_2 > 0))$
      $g_1' := \cos\left(\theta_1\right) R/\sqrt{2}$
      $g_2' := \cos\left(\theta_2\right) R/\sqrt{2}$
      $h_1' := \sin\left(\theta_1\right) R/\sqrt{2}$
      $h_2' := \sin\left(\theta_2\right) R/\sqrt{2}$
      if $g_1 \geq 0$ and $g_2 < 0$ **then**
        $res \mathrel{+}= \text{erfc}(h_2)/2$
          $-$integrateAngularRegion$(R, \theta_2, \theta_1, g_2', g_1, h_2', h_1')$
      else if $g_1 < 0$ and $g_2 \geq 0$ **then**
        $res \mathrel{+}= \text{erfc}(-h_1)/2$
          $-$integrateAngularRegion$(R, \theta_2, \theta_1, g_2', g_1, h_2', h_1')$
      else if $g_1 < 0$ and $g_2 < 0$ **then**
        $res \mathrel{+}= (\text{erfc}(h_2) - \text{erfc}(h_1))/2$
          $-$integrateAngularRegion$(R, \theta_1, \theta_2, g_1', g_2, h_1', h_2')$
  return $1 - res$ ▷ $\equiv P_{\text{int}}(D)$

Algorithm 2 Angular Region Integration

$P_3[0] := 0.885777518572895$ ▷ Constants
$P_3[1] := -0.981151952778050$
$P_3[2] := 0.759305502082485$
$P_3[3] := -0.353644980686977$
$P_3[4] := 0.0695232092435207$
function INTEGRATEANGULARREGION($R, \theta_1, \theta_2, g_1, g_2, h_1, h_2$)
    $res := 0$
    $j_{\text{last}} := \theta_2 - \theta_1$
    $j := h_2 - h_1$
    for $k = 0; k < \text{size}(P_3); k++$ **do**
        $res \mathrel{+}= P_3[k] \cdot j$
        $copy := j$
        $j := (h_2 g_2^{k+1} - h_1 g_1^{k+1} + j_{\text{last}}(k+1)R^2/2)/(k+2)$
        $j_{\text{last}} := copy$
    return $\exp(-R^2/2) \cdot ((\theta_2 - \theta_1)/2 - res)/\pi$

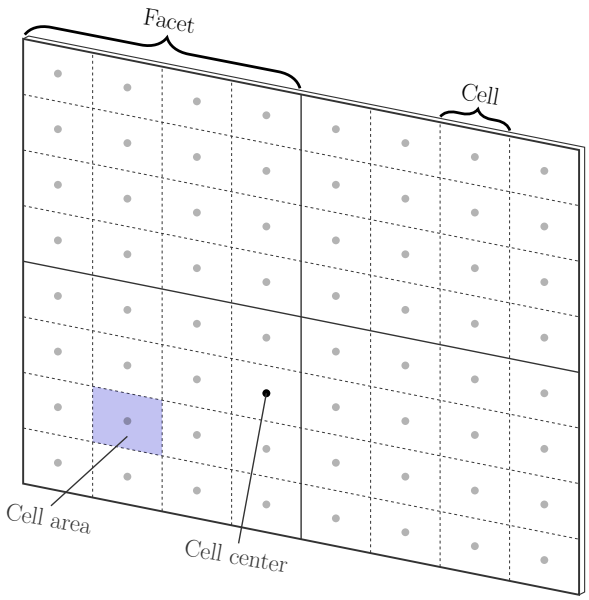


Figure 8: Exemplary heliostat facet partitioning into cells. The heliostat consists of 2x2 facets which are partitioned into 4x4 cells.

Tower shading

This effect may occur when the tower is aligned with the sun and the heliostat. The tower casts a shadow onto the heliostat and intercepts any sun rays potentially being cast onto the heliostat. To check for this effect, the minimal distance of the line from tower position p_{tower} into the sky to the line from sun to the point on the heliostat $p_{i,j,k}^{\text{center}}$ is evaluated. When this distance is smaller than the tower radius, the point on the heliostat is possibly shaded. In this case we check the z-coordinate of the point at which the distance of the two lines is minimal. The heliostat area is shaded when this z-coordinate is smaller than the position of the tower top plus the tower radius.

Heliostat shading

Any heliostat of the power plant between the point $p_{i,j,k}^{\text{center}}$ and the sun may intercept rays cast from the sun on to the heliostat cell. We check whether this is the case by first preselecting the possibly shading heliostats for each heliostat. We consider the line from sun to the center of each heliostat i. Any heliostat m with center point p_m^{center} at a distance to this line that is smaller than the heliostat expansion potentially shades heliostat i. We again filter these candidates for each facet of the heliostat by using half the facet expansion plus half the heliostat expansion as threshold. These filters can also be described as bounding sphere method: we determine whether the distance of a heliostat's bounding sphere to the ray is less than half the heliostat or facet expansion respectively. An illustration of the second filter is given in Figure 9. For all potentially shading heliostats m we then check whether a ray from our heliostat cell center $p_{i,j,k}^{\text{center}}$ towards the sun intersects with any facet of heliostat m.

Heliostat blocking

Similar to heliostat shading, after the sun ray hits the heliostat and is reflected, any heliostat inbetween $p_{i,j,k}^{\text{center}}$ and the receiver may block the reflected ray. The procedure to check this is analogous to that of heliostat shading: we preselect possibly blocking heliostats m by evaluating their distance to the line from center of the reflecting heliostat i to the aim point p_i^{aim} on the receiver. Is this distance smaller than the heliostat expansion, heliostat m possibly blocks the ray. Again, we filter the preselected candidates a second time for each heliostat facet analogously to the approach for the shading filter. We test all remaining possibly blocking heliostats for intersection with the reflected ray.

The blocking and shading test for preselected candidates is narrowed down by an axis-aligned bounding box (AABB) tree intersection check. We consider the ray from sun to heliostat cell center (shading) or from heliostat cell center to receiver aim point (blocking) and the AABB tree of the occlusion candidate which encloses the entire heliostat surface and the heliostat facets in its leaf nodes. See Figure 10 for an exemplary intersection check with a shading candidate. After the AABB tree intersection check we either confirm that there is no intersection or have narrowed down the intersection

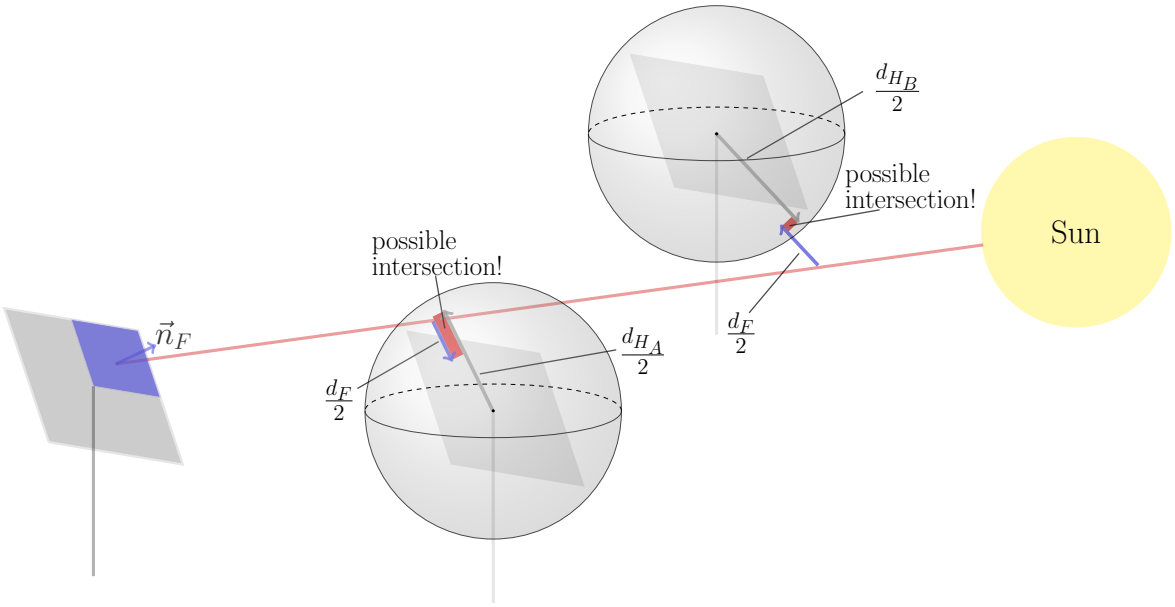


Figure 9: Filtering possible shading (or blocking) candidates for each heliostat facet by the bounding sphere method. Is the distance between ray and bounding sphere of a heliostat less than half the facet expansion, the heliostat remains a shading candidate. In this case both heliostat H_A and H_B are confirmed shading candidates for the considered facet.

to a small (possibly singular) number of heliostat facets. In this case we check for intersection with each of these facets with a Möller-Trumbore ray-triangle intersection check by splitting the facet into two triangles. Finally we can declare with certainty whether the ray is intercepted by the occluding candidate.
Each step of this intricate approach is done to reduce the computational expense of the blocking and shading check as much as possible.

When no blocking or shading effects occur, we trace the reflected ray against the receiver surface, verifying that it does not miss it. It remains to determine the actual power of the ray. For this matter, the next Section will discuss other optical losses.

3.4 Other optical losses

To accurately model the power P_{ray} of a traced ray we factor in further optical losses. There are three additional sources for optical losses we take into account that influence the power of a ray:

Cosine effect

Due to the alignment of the heliostats such that they reflect the sun rays towards the receiver, their surface is not perpendicular to the sun vector. With this angle deviating farther from perpendicularity, the projected heliostat area decreases and less sun light

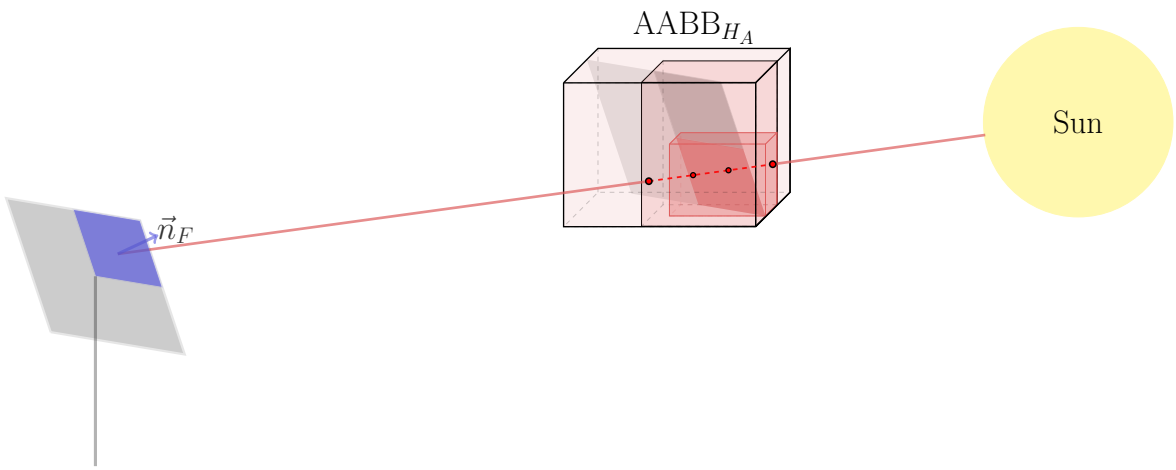


Figure 10: Narrowing down the intersection of ray and shading (or blocking) candidate by means of an AABB tree intersection check. It is determined that the ray intersects the bounding box of the lower right facet of the shading candidate. Note that this example uses the facet center as reflection point for better visibility. In actual simulation this check is done for each ray and therefore for multiple cell centers within the facet.

reaches the receiver, which is called cosine effect [20]. For low solar altitudes θ_{solar} the projected area of the heliostat in fact shrinks significantly. The relative reduction of the projected area can easily be modeled as cosine of the angle of incidence, which we denote as $\eta_{\cos,i}$ for heliostat i [20, 33]. This is calculated by the dot product of the heliostat surface normal and the solar vector. So for heliostat i we get

$$\eta_{\cos,i} = \vec{n}_i \cdot \vec{\tau}_{\text{solar}}. \tag{16}$$

Atmospheric attenuation

As the ray travels from heliostat to receiver it loses a small amount of power due to absorption by water vapor and scattering by air molecules. This effect is modeled as a function of the distance between heliostat i and the aim point on the receiver p_i^{aim}. This distance is denoted as d_i. The function for the relative remaining power after atmospheric attenuation is taken from Schmitz et al. [45] as

$$\eta_{\text{aa}} = \begin{cases} 0.99321 - 0.0001176 d_i + 1.97 \cdot 10^{-8} d_i^2 & d_i \leq 1000\text{m} \\ \exp(-0.0001106 d_i) & d_i > 1000\text{m} \end{cases}. \tag{17}$$

Heliostat reflectivity

Part of the sun light is absorbed or scattered when hitting the heliostat surface rather than reflected due to dust on the surface and an imperfect reflective property of the mirror surface. This typically is modeled as constant relative loss [50]. Adopting this

approach, we model the relative remaining power as constant value η_{ref}.

With the simulation model set up and the optical losses considered, we now calculate the power $P_{i,j,k}$ for a traced ray and briefly summarize the whole ray tracing procedure.

3.5 Procedure summary

The procedure consists of three steps:

1. **Ray generation:** we partition the heliostat facet j of each heliostat i into M smaller heliostat cells and generate a ray for each cell. The rays power is scaled to the cell area $A_{i,j,k}$.
   a) Using *Monte-Carlo:* to perform repeated sampling we duplicate the generated ray N times and perturb it each time according to a random evaluation of the horizontal and vertical gaussian distribution with $\sigma_{\text{len}}^{\text{hor}}$ and $\sigma_{\text{len}}^{\text{ver}}$.

2. **Blocking and shading check:** the generated rays are checked for blocking and shading effects. Only those rays that are neither blocked nor shaded are evaluated in the final step.

3. **Ray evaluation:** according to the employed methodology we evaluate all remaining rays to determine the flux distribution on the receiver.
   a) Using *Monte-Carlo:* rays are traced against the receiver surface to verify that they indeed hit the receiver. To identify flux distribution we partition the receiver panels into smaller pieces and check for intersection with individual pieces, as mentioned in Section 2.1. If the ray hits a receiver piece, we add flux to the piece in extent of $P_{i,j,k}^{\text{mc}}$.
   b) Using *Convolution:* for each receiver piece we calculate the probability P_{hit} that the ray hits the piece. We add flux to every piece corresponding to $P_{i,j,k}^{\text{conv}}$.

The ideal power $P_{i,j,k}^{\text{ideal}}$ of a traced ray from cell k of facet j on heliostat i is simply given by the direct normal irradiation I_{DNI} times the area $A_{i,j,k}$ of the cell the ray represents:

$$P_{i,j,k}^{\text{ideal}} = I_{\text{DNI}} \cdot A_{i,j,k}. \tag{18}$$

This power then is scaled by the in Section 3.4 mentioned factors, giving us $P_{i,j,k}$ as

$$P_{i,j,k} = P_{i,j,k}^{\text{ideal}} \cdot \eta_{\cos} \cdot \eta_{\text{ref}} \cdot \eta_{\text{aa}}. \tag{19}$$

Finally, the power is scaled according to our used methodology. For the Monte-Carlo method we divide by the number N of samplings done:

$$P_{i,j,k}^{\text{mc}} = P_{i,j,k} \cdot \frac{1}{N}. \tag{20}$$

When using the Convolution method, we instead scale by the probability P_{hit} that the perfect reflected ray will be perturbed in a way such that it hits the receiver piece:

$$P_{i,j,k}^{\text{conv}} = P_{i,j,k} \cdot P_{\text{hit}}. \tag{21}$$

Now that the simulation methodology is layed out, the next Section will begin to present optimizations to the current procedure and its code implementation.

4 General optimization

The optical simulation of SunFlower, as portrayed in the previous two sections, already employs state-of-the-art methods to reduce the computational intensity.

Among these is the bidirectional raytracing approach, where rays are generated directly on the heliostat surface. In comparison to the traditional method of generating rays in the sky, this significantly reduces the amount of rays that need to be traced as no rays are considered that miss all heliostats.

Another already present optimization is the predetermination of potential shading and blocking heliostat pairs according to bounding spheres. With this approach, the blocking and shading checks for each ray only have to test for intersection with a small number of heliostats rather than every single one. For larger heliostat fields this methodology is all but necessary, as the number of blocking and shading checks grows quadratically with the number of heliostats.
In this regard SunFlower even does another step not found in other tools. It uses the found potential blocking and shading candidates to predetermine the potential blocking and shading heliostats for each heliostat facet to further reduce the number of intersection tests necessary for rays generated on this facet. As the number of candidates that have to be evaluated for this step already is quite small, the added computational intensity is manageable in comparison to the disregarded intersection tests and this additonal step proves to be advantageous, especially for high discretization resolutions.

Furthermore SunFlower uses bounding volume hierarchies (BVH) consisting of axis-aligned bounding boxes (AABB) for both heliostats and receivers to speed up remaining intersection checks with either object.
These optimizations accelerate the simulation by orders of magnitude for non-trivial heliostat fields.

To find further optimization pathways, the first step is to identify where most time is spent. Table 1 shows a superficial summary of the top hotspots during a serial execution of a typical annual simulation of Gemasolar with the Monte-Carlo raytracer. The profiling was done using the Intel VTune Profiler.

Function	CPU-Time [%]
CGAL AABB tree ray-intersection check	27.3
Mersenne twister normal-distributed RNG	19.6
traceMonteCarlo	8.9

Table 1: Top computation hotspots during serial execution of exemplary annual simulation of the Gemasolar power plant with the Monte-Carlo raytracer

Inspite of the aforementioned optimizations, 27.3% of CPU-time was still spent with ray-intersection checks by the Computational Geometry Algorithms Library (CGAL). The second most time is spent with the generation of the normal-distributed random numbers, which is a common computational expense in Monte-Carlo based methods. In this case a mersenne twister random number generator (RNG) and normal distribution of the C++ standard library (std) is employed and used close to 20% of CPU-time. The next most time is spent in the *traceMonteCarlo* function which coordinates the raytracing, does some calculations and calls multiple functions to perform ray generation, perturbation and evaluation.
Apart from these top three hotspots the CPU-time is spread across many smaller functions, most of whom are geometric utility functions like vector normalization or coordinate transformation.
An important takeaway is that a lot of time is spent on code that should respond well to parallelization.
Lastly, there is also a considerable amount of time spent with data access and storage.
Taking a closer look at the CGAL AABB tree ray-intersection check that uses up more than one fourth of the CPU-time, we have to differentiate the two situations in which it is called: ray-heliostat-intersections and ray-receiver-intersections. We will inspect the former case first.

Ray-heliostat-intersection checks

Heliostat-intersection checks are done during the blocking and shading tests, to confirm whether one of the predetermined potential blocking and shading candidates intercepts the ray.
As mentioned before, the predetermination of potential blocking and shading heliostats for each heliostat facet according to their bounding spheres reduces the amount of intersection checks significantly, but the bounding sphere method is still overestimating and will consider many heliostats that are aligned in a way such that they clearly will not intercept the ray.
In most cases we use high cell discretization resolutions for the heliostats to obtain accurate simulation results, which means the blocking and shading candidates of a heliostat facet are checked for hundreds or thousands of rays. Thus it might be beneficial to do an additional step and again filter the remaining potential blocking

and shading heliostats for each facet by considering current heliostat alignment. This can be done by considering planes along the projection of the facet in ray direction and evaluating the position of heliostat corners relative to these planes. We need a total of five planes: the facet surface itself and the four planes that connect the facets corners to the corresponding corners of its projection in ray direction. For the determination of blocking candidates the ray direction references the reflected ray that is directed towards the receiver and for the determination of shading candidates the ray direction corresponds to the sun vector.
If we calculate the side planes such that their normal faces inwards (as in: towards the facet center), it only remains to check whether for all planes there is at least one corner of the AABB of the candidate that is on the positive side of the plane. Note that it does *not* have to be the same corner for all five planes. If this condition holds for all five planes, the heliostat still has to be considered a potential blocker or shader respectively.
Considering just heliostat corners instead of the corners of its AABB would lead to inaccuracies, as even for flat facets the heliostat is not perfectly flat due to canting.

To underline the advantage of this additional filter, Figure 11 shows the same exemplary situation that was also used in Figure 9, which illustrated the preceding bounding sphere filter in Section 3.3.
For facet F the two heliostats H_A and H_B are checked as potential shading candidates. The four facet side planes are constructed connecting the facet corners to those of the projection of the facet on the sun. Next we test for all four side planes and the facet surface plane whether there is at least one corner of the bounding box of the candidate that is on the positive side of the plane. In Figure 11 the corner V_i of the AABB of heliostat H_A is on the positive side of all five planes. Thus, H_A is confirmed as shading candidate. For heliostat H_B all corners of its AABB are found to be on the negative side of the top or left plane and with that H_B, in contrast to the bounding sphere method, is determined to not be a valid shading candidate.

It is important to note that this filter does not replace the bounding sphere method but instead is applied to the remaining candidates confirmed by the bounding sphere method. This multi-step approach exhibited the best performance during testing. While the computational intensity of the additional condition is non-negligible, it in this way only has to be applied to the very few remaining (if any) filtered candidates and is able to again reduce the number of intersection-checks, which trump the expense of all applied filters. Furthermore the benefit accumulates for higher cell discretization resolutions.

Ray-receiver-intersection checks

They occur when the blocking and shading test is passed and the ray is traced against the receiver surface to confirm it does not miss. Note that this step is only

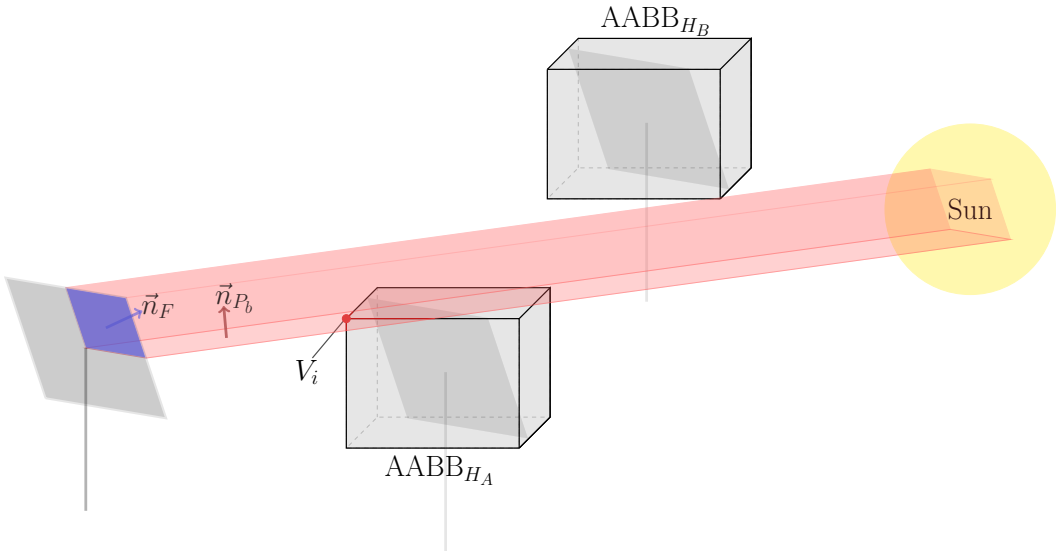
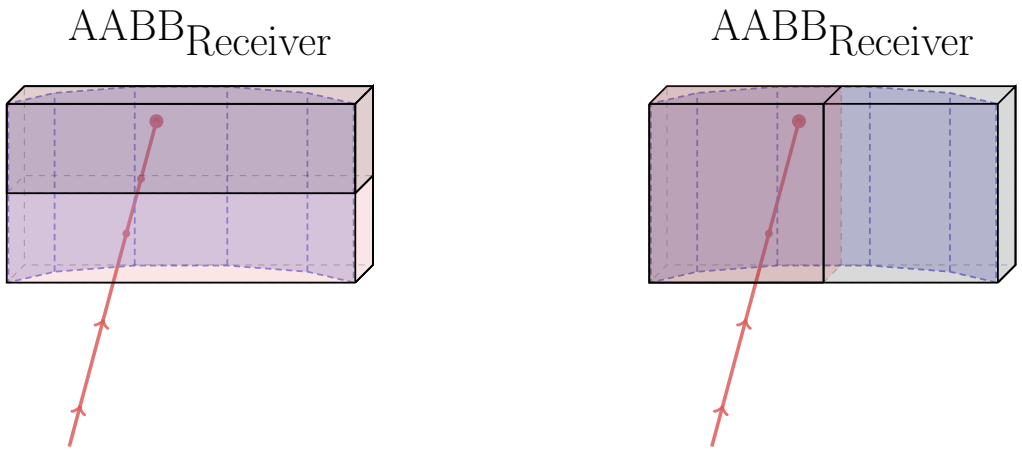


Figure 11: Filtering potential shading (or blocking) candidates for a heliostat facet by checking whether any corner of a candidate's AABB lies on the positive side of all five planes that encompass the facet. Heliostat H_A is confirmed as shading candidate, H_B is found to not be a valid candidate.

done by the Monte-Carlo raytracer and is omitted by the Convolution raytracer. Depending on the receiver discretization the cost of this check can severely outweigh that of ray-heliostat-intersection checks.
As each ray that is not blocked nor shaded has to be checked for intersection with the receiver, we cannot reduce the number of checks but will have to make the check itself more efficient. Therefore we will discuss how to optimize the implemented AABB tree approach which will impact both receiver- and heliostat-intersection tests.
To have better control over the AABB tree implementation and its traversal during the intersection check, we implement our own, very simple AABB tree. An alternative solution to the current CGAL AABB tree will be necessary for later use on GPU anyway, as the GPU does not directly support high level libraries like CGAL or boost.
Picking a different method like another bounding volume should also be considered. There are two reasons to stick with AABBs: firstly we have to build a BVH from heliostat facets for every single heliostat and also for receivers from receiver pieces. This already is a considerable workload. Additionally, during an annual simulation we have to adapt the BVH for every single heliostat before every simulated moment as heliostats are aligned to the changing sun position. Therefore we need a BVH that is either easy to adapt or really quick to create, which is a clear strength of AABBs. Secondly, the receiver is typically perfectly (besides from the rare slightly tilted flat receivers) aligned with the z-axis, making AABBs a good fit.



(a) Combining horizontally first: the top most levels split the vertical dimension, unable to narrow down the intersected region.

(b) Combining vertically first: the top most levels split the horizontal dimension, immediately discarding half the region.

Figure 12: Exemplary impact of combining the AABB-nodes vertically vs. horizontally first during BVH creation on intersection check with a ray coming from below. It is advantageous to build such that the top most levels split the region that has typically a smaller entrance angle to the ray.

The new AABB tree is a simple binary tree that we build bottom-up from the receiver pieces or heliostat facets respectively. In case of the receiver, we choose to combine *vertically* adjacent AABB-nodes first and horizontally second. This means during ray-receiver-intersection tests the top-down traversal will first narrow down the horizontal region. The reasoning for this choice is that the vertical angle between receiver panels and incoming rays most often will be steeper than the horizontal angle, making it harder for BVHs to narrow down the vertical region. Figure 12 depicts an exemplary intersection check of a horizontal-first vs. vertical-first build AABB tree with a ray coming from below.
Conversely, the heliostat AABB trees are build bottom-up by combining *horizontally* adjacent AABB-nodes first. The reasoning is analogous, as shading and blocking typically occurs when the ray's vertical angle is small.
Testing the opposite approaches as well as alternating combination of horizontal and vertical neighbors, we confirmed that this way of building the AABB trees on average makes later intersection checks most efficient.
With the AABB tree built it remains to decide how we check for intersection of AABBs and rays and how we will traverse the tree.

To test whether a ray intersects a given AABB, an improved [51] and branchless [3] implementation of the *slab* method [29] is used. This method truncates the ray in each dimension such that only the section of the ray remains that is within the

bounding box. When the ray does not intersect the box the length of the truncated ray is zero.
The best performing method of tree traversal heavily depends on the executing hardware, the size of the tree and the way we save our AABB nodes.
As our AABB trees are well balanced and the depth typically quite small, a recursive preorder traversal can be used without concerns on CPU. Heliostat AABB trees are especially small and easily fit into the large CPU cache. Thus, the order in which we save the nodes is not as significant. On GPU this topic has to be evaluated with more care, so we will touch on this point again in Section 5.2.
Finally, once potentially intersected receiver pieces or heliostat facets are determined through AABB tree traversal, a Möller-Trumbore ray-triangle-intersection test [32] is done on the two triangles that make up the corresponding rectangle (piece or facet).

Generation of normal-distributed random numbers

Of the top three identified hotspots the random number generation offers the most straight forward optimization: using a faster random number generator. SunFlower previously incorporated a mersenne twister, which is a very commonly used RNG. However, there are many more efficient alternatives that deliver pseudo-randomness appropriate for a Monte-Carlo simulation.
One of the fastest and a very robust way to generate 64 random bits is the xoshiro256+ algorithm developed by Blackman and Vigna [6] in recent years.
As we need normal-distributed random numbers, the generation actually consists of an additional step to transform from uniform-distributed numbers to normal-distributed numbers. Using two random numbers from xoshiro256+ as input, we use a Box-Muller transform to get two normal-distributed numbers [21]. The transformation to normal-distributed numbers N_1 and N_2 for uniform-distributed numbers U_1 and U_2 is given by:

$$\begin{aligned} N_1 &= \sqrt{-2\ln U_1} \cdot \cos\left(2\pi U_2\right) \\ N_2 &= \sqrt{-2\ln U_1} \cdot \sin\left(2\pi U_2\right) \end{aligned} \tag{22}$$

As CUDA offers its own random number generation library in cuRAND, we will briefly touch on this topic again for the GPU in Section 5.2.

Next we take a look at the Convolution raytracer. We do a basic hotspot profile using the same settings as before. A summary of the top hotspots is shown in table 2. As the Convolution raytracer evaluates non-intercepted rays by a set of integrations, this calculation is also where most time was spent. Even the in Monte-Carlo raytracing very influential blocking and shading test was not amongst the top five hotspots. Further inspection shows that calls to the top three functions (*pow*, *sincos*, *acos*) are done almost exclusively from within either one of the last two hotspots *integrateAngularRegion* and *integrateGaussianPolygon*. When attributing functions

Function	CPU-Time [%]
pow	22.7
sincos	12.3
acos	8.8
integrateAngularRegion	7.8
integrateGaussianPolygon	5.9

Table 2: Top computation hotspots during serial execution of exemplary annual simulation of the Gemasolar power plant with the Convolution raytracer

to their respective caller, the entire CPU-Time spent in the top five hotspots is actually spent inside *integrateGaussianPolygon* or in functions called by it. Although we do not want to alter the integration method itself, thoroughly inspecting its code implementation reveals many ways to make it more efficient. Let's take a look at the in algorithm 2 layed out *integrateAngularRegion* function, which is called repeatedly for each ray by the Convolution raytracer. Inside the functions for-loop, in line 13, there are three occasions where a number is raised to a power. In the C++ implementation this is done by calling the *pow* function. This code line is in fact by far the main caller to this top hotspot function of the Convolution raytracer. We can restructure this loop, completely eliminate the usage of *pow* and instead incrementally calculate the factors g_1^{k+1} and g_2^{k+2} on the go. Additionally we precalculate $R^2/2$ in *integrateGaussianPolygon*, as R^2 is a byproduct of calculating R, and hand it as parameter instead of R. The improved *integrateAngularRegion* function is depicted in algorithm 3.

Algorithm 3 Optimized Angular Region Integration

1: **function** INTEGRATEANGULARREGION2($R', \theta_1, \theta_2, g_1, g_2, h_1, h_2$)
2: $res := 0$
3: $j_{\text{last}} := \theta_2 - \theta_1$
4: $j := h_2 - h_1$
5: $g_1' := g_1$
6: $g_2' := g_2$
7: **for** $k = 0; k < \text{size}(P_3); k++$ **do**
8: $res \mathrel{+}= P_3[k] \cdot j$
9: $copy := j$
10: $g_1' \mathrel{*}= g_1$
11: $g_2' \mathrel{*}= g_2$
12: $j := (h_2 g_2' - h_1 g_1' + j_{\text{last}}(k+1)R')/(k+2)$
13: $j_{\text{last}} := copy$
14: **return** $\exp(-R') \cdot ((\theta_2 - \theta_1)/2 - res)/\pi$

Due to the optimizations to the Convolution raytracer being purely code related and

hence SunFlower specific, further changes will not be layed out in detail. Instead we will take a broader view and discuss the core concepts behind applied code optimizations to the entire optical model including the Convolution raytracer.

4.1 Code optimization

Accelerating serial code without altering the underlying functionality in part is viewed skeptically as many optimizations are already handled by the compiler. The compiler typically also does a better job at optimizing code for a specific architecture. However, the compiler is not always able to understand the inner workings of a complex program or mathematical calculations.

The above described adaption to the *integrateAngularRegion* function shows one such case. Here is another example: at the start of a SunFlower simulation the sun vector is calculated and stored. During raytracing the sun vector then is repeatedly used for different calculations such as the heliostat alignment or a ray's reflective direction. For multiple of these calculations the sun vector is normalized on the spot. As developer it is easy to determine that we can do every single one of these calculations with a normalized vector. Hence we directly normalize the sun vector after its initial computation and save it as such at the start of the simulation. The compiler is not able to understand the underlying geometry, cannot know that this does not alter the functionality of the code and thus would not apply this optimization.
We look at every expensive function that is repeatedly called and evaluate whether or not some sort of precalculation is feasible. This includes substituting calculations with a constant if possible.
The general goal is to eliminate redundant calculations and reduce the excessive use of expensive functions, such as *sqrt*, *pow*, trigonometric functions or division.

The next step is to eliminate unnecesssary checks and branches. In SunFlower many functions act as independent and uninformed entities. This means they check every input parameter for potentially hazardous cases, often with significant computational expense. In general this is a good approach to write robust code, but for performance it is desirable to avoid these checks. However, to ensure correctness, we cannot just remove the checks and hope to never encounter hazardous cases. Instead we move the responsibility to the caller of the function to only call it with reasonable input parameters. This type of code manipulation represents a tradeoff between robust and performant code. Working in an environment where code performance is paramount, we want to try to minimize the impact of such safety measures without completely discarding them. This entails additional work for the developer but speeds up specific functions considerably. Look at the *integrateGaussianPolygon* function in algorithm 1 for an example. The two initial checks done in lines 2 and 4 confirm the input polygon is a simple polygon and correctly oriented, which is necessary for the integration to produce correct results. Both checks have significant performance impact and can be discarded if we instead validate the way in which we construct the

input polygon before calling the function.

Another point is to ask the compiler to inline functions that are called excessively to reduce function call overhead. The compiler may do this anyway for certain functions or inspite of the developers *inline* directive might decide against it.

Lastly we take a brief look at how we can exploit characteristics of the annual simulation.

4.2 Annual simulation considerations

The annual simulation consists of a set of optical moment simulations for a fixed heliostat field setup. Thus we can reuse the heliostat field dataset and just adapt information that is tied to the changing sun position.
This basically amounts to the alignment of heliostats, which subsequently entails rebuilding their AABB trees and re-evaluating potential blocking and shading heliostats. The heliostat center, bounding sphere and aiming point are invariant to different sun positions, hence the potential blocking heliostats for each heliostat do *not* need to be recomputed. However, this does not hold for the blocking candidates for each facet, as the facet center has moved after alignment. Furthermore shading candidates have to be computed again on both heliostat and facet level.
A new discretization of the heliostat into cells is not necessary as they are defined in the local coordinate system of the heliostat, which is relative to the invariant heliostat center. Global cell position is calculated for each moment during raytracing and uses the updated heliostat axes.

Following the optimization of serial execution, the upcoming Section will discuss the parallelization of the optical simulation.

5 Parallelization

After several refinements to the serial execution of both a moment simulation and an entire annual simulation, this Section will tackle the parallelization of the simulation. We start by discussing the implementation of a baseline CPU parallelization using OpenMP [11] and will debate some steps to increase its throughput.
Following the CPU parallelization we take a look at the GPU architecture before describing how the code is adapted to be compatible with CUDA and to make it portable to GPU. Finally a number of performance considerations for the GPU implementation are investigated.

5.1 CPU parallelization

First we parallelize the initial setup of each heliostat before the first simulation moment by distributing heliostats among threads. It includes heliostat discretization, aimpoint calculation and heliostat facet canting. Performance analysis with Intel VTune Profiler shows that on our system, a typical desktop computer with a single 8-core CPU, using more than two threads for these tasks will impede performance, likely due to their memory bound nature. This also holds for the subsequent setup steps of heliostat alignment and AABB tree creation which are done before every simulation moment during an annual simulation, as explained in Section 4.2.

The parallelization of the actual raytracing in other research is done by distributing rays or chunks of rays among the threads as the enormous amount of rays allows perfect work balancing on any system. We instead choose to again distribute the work on the heliostat level to avoid a synchronization problem. This means each thread handles all rays that are generated from a specific heliostat, or from a specific set of heliostats. As one objective of a SunFlower simulation is to collect information on the efficiency of each heliostat, it counts for each heliostat how many of its corresponding rays hit the receiver, are blocked or shaded, and how significant the cosine effect is. Parallelizing over rays independently from heliostats would mean synchronization is necessary for the aggregation of these results, as multiple threads could simultaneously try to update the values of a specific heliostat. As the size of modern heliostat fields is increasing, simulations are commonly done with several thousands or even tens of thousands of heliostats. Thus the workload can still be efficiently distributed on a CPU with only a handful of simultaneous threads. Furthermore when each thread continuously traces rays for a single heliostat we profit from spatial data locality and can reuse certain data, such as the heliostat aiming point, its position and coordinate system. In this way the data access is more efficient, no synchronization is needed for heliostat efficiency results and we do not significantly handicap load balancing.

To achieve additional throughput on CPU, one should further consider vectorization and fused operations. Where possible, these techniques are typically applied by the compiler, but in many cases the developer first has to revise the code to expose these possibilities, especially vectorization, to the compiler.
In case of our ray tracing there are multiple pieces of code that could be vectorized by restructuring the execution, however the most costly items in intersection checks, random number generation and most of the integration cannot easily be vectorized.

In general additional optimizations of the CPU parallelization are estimated to be very labor intensive compared to their expected impact. Consequently we at this point refrain from further optimizations, as the goal of the CPU parallelization is merely to establish a baseline for comparison with the later GPU implementation. Thus we now shift our focus to the GPU parallelization.

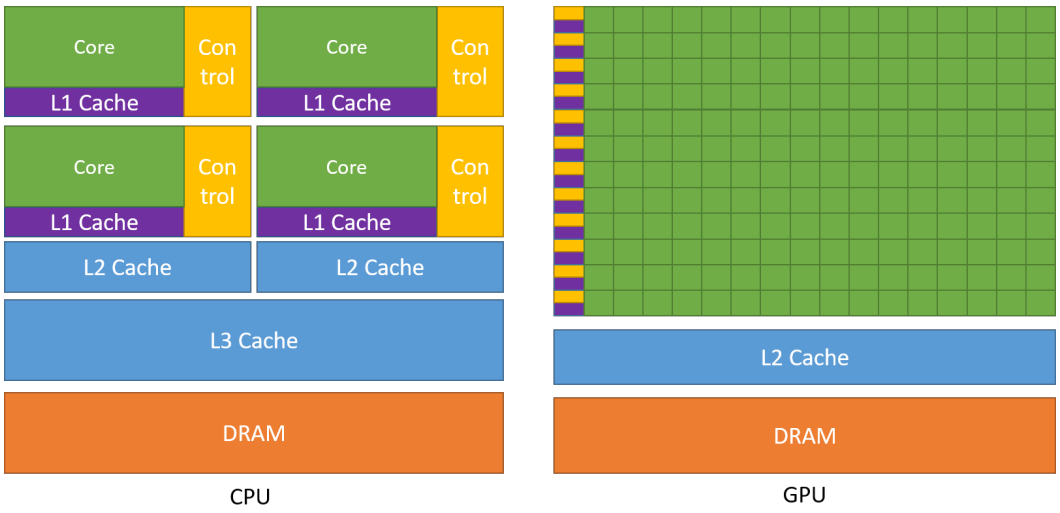


Figure 13: Schematic comparison of CPU and GPU architectures [36]

5.2 GPU parallelization

Before we lay out the adaption of the software for GPU parallelization, we first take a look at the key characteristics of the GPU architecture and the corresponding programming model.

5.2.1 GPU architecture

We start with a brief comparison of GPU architecture to the traditional multi-core CPU architecture. Figure 13 shows a schematic comparison of both architectures. The CPU is a general purpose architecture and consists of few cores with dedicated, comparably large on-chip caches as well as advanced on-chip control units. This architecture is suited to efficiently execute any code in serial due to low latencies and high level control logic for branch prediction and out-of-order execution. Each core can work independently on a different task and cache coherency is ensured through various protocols. Additional parallelism on each core is possible for some operations on vectorized data, called Single Instruction Multiple Data (SIMD), through special registers.

In comparison the GPU is a more specialised architecture, inherently catered towards data parallelism. Instead of a small number of independent cores, the GPU is composed of a large number of weaker cores which are packed as groups. In case of NVIDIA GPUs these are called Streaming Multiprocessors (SM). They share a cache, which can also be used as shared memory. Each thread has a limited amount of registers available and also has private memory inside the global memory. There is no dedicated cache for each parallel processor.

The CUDA programming model lets the developer distribute the work on GPU in a

CPU	vs.	**GPU**
Few powerful cores		Many weaker cores
Task parallelism		Data parallelism
optional SIMD		inherent SIMT
Low latency		High throughput
Efficient serial execution		Effective parallel execution
Large coherent caches		Small incoherent caches
High level control logic		Basic control logic

Table 3: Key differences of CPU and GPU architectures

hierarchy. The developer sets the grid size for the start of a GPU function (kernel), translating to the number of thread blocks that are created. Furthermore the developer establishes how many threads should be executed in each thread block. The thread blocks are then distributed to the SMs similarly to threads to cores on the CPU. Each SM may be able to run multiple thread blocks concurrently if enough blocks are started.
The execution model on SMs is a more general form of SIMD which NVIDIA calls Single Instruction Multiple Thread (SIMT). Tasks are executed in warps of 32 threads with the entire warp working in sync. Thus branches that would diverge the execution path of threads within a single warp have to be serialised. Table 3 summarizes the key differences of both architectures.

The following Sections will discuss how we adapt the SunFlower tool for the CUDA C++ programming language and the GPU architecture as well as the steps we took to further optimize the initial GPU implementation.

5.2.2 Adapting SunFlower for GPU

The CUDA C and C++ programming language essentially comprises the C programming language extended by a subset of C++ features and functionality necessary to run kernels on GPU. The GPU inherently does not support the use of any C++ libraries such as the Standard Library (std), boost or CGAL. Data structures have to mostly oblige C conventions and CUDA offers its own libraries for advanced mathematical functions.

Throughout the entire implementation of the SunFlower tool there are many C++ libraries used. Especially the CGAL library is used extensively for the optical simulation. The geometry of the optical model is saved in CGAL data structures and CGAL functions are used for a multitude of computations, such as directions, angles, distances or intersections.
Additionally std library container classes and smart pointers are used to easily allocate memory, store and handle data.

Therefore, and to ease the integration of the GPU into the workflow of the SunFlower tool, we re-implement a lightweight version of the optical model. The goal is not only to adhere to the limitations of the CUDA C++ language, but also to strip the model off any unneccessary clutter.
First we implement the basic geometry data structures and functionality in CUDA C++ code. Next the entire optical model and the raytracers are added. They are restricted to the new geometry and C style arrays and pointers for data storage.

We further implement a new Translator module that will act as interface between host (CPU) and device (GPU) code. To setup the new model in GPU memory the settings are read by the already existing host code at the start of the simulation. Afterwards the Translator module allocates the required memory on host and device for all optical entities. The data is initialized on host and finally copied to the device. Additional memory is allocated on the device for data that will be produced during the simulation.

As we want to avoid dynamic memory allocation on GPU, we pre-allocate necessary memory. For most data we know very accurately how much space will be required, but for the arrays that store the indices of blocking and shading candidates according to the methods introduced in Section 3.3 and 4, we can only estimate the required memory space. The worst case would be that every heliostat is a candidate for every facet, which would mean we need to allocate
$4\text{Bytes} \times \#Heliostats \times \#FacetsPerHeliostat \times \#Heliostats$ of memory for blocking and shading each. For a field of ten thousand heliostats with 30 facets per heliostat this equates to 24 Gigabytes of memory.
As this is an immense overestimation and also infeasible, we need to find a smarter way to store the candidates.
After testing different sun positions and heliostat fields we determine that the sum of all candidates for the entire heliostat field, denoted as $\sum N_\mathrm{C}$, in a realistic setup is always well below N':

$$\sum N_\mathrm{C} \leq N' = \#Heliostats \times \#FacetsPerHeliostat \times \sqrt{\#Heliostats}. \tag{23}$$

Using this estimation, we could limit the allocated memory to only 240 Megabytes for the given example. However, this bound holds only for the total number of candidates for the entire field, not for each facet. If we dedicate an equal amount of memory to each facet, this estimation would cap the amount of candidates each facet can save to $\sqrt{\#Heliostats}$, or 100 in the given example. Some facets may exceed this number during low sun positions, so we share the memory across a heliostat, giving each heliostat space for $\#FacetsPerHeliostat \times \sqrt{\#Heliostats}$ candidates. This suffices to reliably save all candidates for all facets. For this we implement a simplified compressed sparse row storage (CSR) which is used for sparse matrices [48]. We only need one indexing array each for blocking and shading that saves the starting position of a facet's candidates.

To complete the setup process a small kernel is started before the first moment simulation that computes facet canting, heliostat expansion, atmospheric attenuation and the AABB tree of the receiver directly on the device.

This entire setup is necessary only once before the first moment simulation. All data on GPU can be re-used for further moment simulations of an annual simulation. Essential manipulations of the optical model for these subsequent simulations, such as heliostat alignment and AABB tree creation (see Section 4.2), are implemented in device code, in part to avoid unneccessary host-to-device data migration.
After data initialization all moment simulations on GPU are started by the host by handing the calculated sun direction to the Translator module which then executes a number of kernels:

1. **Heliostat setup kernel:** each thread handles one heliostat. It aligns the heliostat according to the sun vector, creates the heliostat AABB tree and calculates potential blocking and shading candidates first on heliostat and then on facet level.

2. **Raytracing kernel:** analogously to the CPU parallelization, each thread again handles one heliostat. Rays are generated and evaluated according to the chosen raytracing approach and the methodology discussed in Section 3. The flux of hitting rays is added to the receiver flux map using *atomicAdd*. Synchronization for heliostat efficiencies is not necessary (see Section 5.1).

3. **Result kernels:**
   - **Flux map reduction kernel:** efficiently sums (reduces) the receiver flux map to get the total flux on the receiver. Initially each thread sums a small part of the flux map in shared memory, then summation continues in a tree structure.
   - **Heliostat field efficiency aggregation kernel:** aggregates the efficiency values of all heliostats to evaluate the overall efficiency of the heliostat field.

After starting each kernel the host has to wait for the device to completely synchronize before starting the next kernel. Once the result kernels have finished, the desired result data is migrated to the host and is integrated back into the existing SunFlower workflow.

Random number generation with cuRAND

As mentioned in Section 4, CUDA offers its own library for random number generation with cuRAND [36, 35]. Besides multiple high quality random number generators it also offers output in different distributions.
The cuRAND library also implements a XORWOW RNG, which works similarly to the xoshiro256+ generator we used for our CPU implementation. However, according

to NVIDIA's own testing on a high-end GPU, the counter-based Philox pseudo-random generator [44] is the fastest in cuRAND at generating normal-distributed single precision numbers on GPU [35].
For our implementation of the Monte-Carlo raytracing on GPU we tested both the XORWOW and the Philox RNG. As we could not determine noticable differences, we choose to use the Philox generator.

With this basic GPU implementation in place, we now take a look at the additional steps we took to optimize it's throughput on GPU.

5.2.3 Execution divergence

A major performance concern for any code executed on GPU is execution divergence. As explained in Section 5.2.1, the GPU groups threads in warps which work synchronously and therefore has to serialize code that diverges the execution path of threads within a warp. The typical scenario for this would be a large conditional branch that is entered only by some of the threads of a warp.
We take a look at two parts of the execution where we can alter the code to mitigate this problem.

AABB tree intersection check

First is the AABB tree intersection check. In Section 4 we briefly discussed the tree traversal of our AABB trees and decided to use a basic recursive traversal on CPU. However, this recursive traversal leads to high execution divergence on GPU as each thread independently decides whether to recurse on or to skip a tree node. If two threads decide differently, they may go out of sync [28].
As NVIDIA developer Karras [28] showcases, a carefully managed iterative traversal minimizes this effect as all threads repeat the same loop, mostly independent of specific traversal decisions.
We adopt the proposed iterative traversal method for our AABB tree. Although this will not fully eliminate the problem as nearby threads may do a different number of iterations, it greatly reduces its impact.

Bivariate polygon integration

The polygon integration necessary for the Convolution method (see Section 3.1.2 and Algorithm 1) branches multiple times to handle four different angular cases [25]. To ensure that threads do not go out of sync during the computation of this integral, we re-write the code such that we avoid these branches.
As the branches largely do the same computations, only altering the parameters and signs, we just have to manipulate the parameters and signs according to the case we encounter without branching. For this we make use of the boolean arithmetics of the C programming language. The in some cases undesired calls to the complementary

error function (*erfc*) can also be handled by multiplication with a boolean. Case dependent positive or negative signs are analogously produced with boolean arithmetic.
Algorithm 4 shows the revised version of the integration. All cases are still handled correctly, however now all threads are guaranteed to take the exact same execution path for the computation.

As next step we discuss our use of floating point data types as it heavily affects the instruction throughput of the GPU architecture.

5.2.4 Floating point data types

The SunFlower tool previously worked exclusively on double precision floating point numbers to guarantee high accuracies. For many calculations this level of precision is not required, but on modern CPU architecture the performance implications are negligible. In contrast, the GPU is an architecture optimized for 32-bit floating point operations, so our 64-bit floating point calculations will not achieve optimal instruction throughput. Depending on the exact GPU chip involved, the instruction throughput of double precision operations is up to 32x lower than that of single precision [36].
However, replacing all double precision variables and computation with respective single precision ones significantly impacts the accuracy of our results. Therefore we need to carefully evaluate where double precision calculations are necessary to ensure high accuracy and where single precision is sufficient.

Incrementally stepping through the code, we determine that the main source of inaccuracies with single precision values comes from the repeated summation of miniscule values onto the receiver flux map. Especially as the flux accumulates to larger magnitudes, further addition of very small single precision floats results in significant rounding errors.
Similarly we detect rounding errors that occur during the accumulation of heliostat efficiency values due to the same problem with numbers of varying magnitude.
We keep the corresponding variables as double precision, change all other to single precision and make sure that all computations use the correct data type such that type casting is only necessary for the summation of result values.
Finally it was found that some sensitive projection calculations of the Convolution raytracer also introduce slight inaccuarcies on single precision numbers. In this case the problem could not be solved by keeping a small set of values and computations as double precision. For that reason and as deviations in results are within 0.02%, we decide to continue with single precision. As part of the case study in Section 6, we will validate the accuracy against the established CPU version.

Besides utilizing the extreme advantage of single precision floating point operations on GPU another way to maximize the instruction througput is to employ GPU

Algorithm 4 Branchless Bivariate Polygon Integration

function integrateGaussianPolygon2(D, σ_{hor}, σ_{ver})
  $D' :=$ scaleToCircular$(D, \sigma_{hor}, \sigma_{ver})$
  $res := 0$
  for each vertex V_i in D' **do**
    $v_1 := (V_i - \text{predecessor}(V_i))/|V_i - \text{predecessor}(V_i)|$
    $v_2 := (\text{successor}(V_i) - V_i)/|\text{successor}(V_i) - V_i|$
    $R^2 := V_i \cdot V_i$
    $R := \sqrt{R^2}$
    $res \mathrel{+}= (R == 0) \cdot \arccos((v_1 \cdot v_2)/(2\pi))$
    $l := V_i/R$
    $\theta_1 := \text{sign}(\det(v_1, l)) \cdot \arccos(v_1 \cdot l)$
    $\theta_2 := \text{sign}(\det(v_2, l)) \cdot \arccos(v_2 \cdot l)$
    $RSQRT2 := R \cdot \sqrt{2}$
    $g_1 := (v_1 \cdot l) \cdot RSQRT2$
    $g_2 := (v_2 \cdot l) \cdot RSQRT2$
    $h_1 := \sin(\theta_1) \cdot RSQRT2$
    $h_2 := \sin(\theta_2) \cdot RSQRT2$
    $gg := ((g1 \geq 0) \text{ and } (g2 \geq 0))$ ▷ Booleans for branchless computation
    $gs := ((g1 \geq 0) \text{ and } !(g2 \geq 0))$
    $sg := (!(g1 \geq 0) \text{ and } (g2 \geq 0))$
    $ss := (!(g1 \geq 0) \text{ and } !(g2 \geq 0))$
    $\theta_1 := \theta_1 + (sg + ss) \cdot \pi \cdot (1 - 2 \cdot (\theta_1 > 0))$
    $\theta_2 := \theta_2 + (gs + ss) \cdot \pi \cdot (1 - 2 \cdot (\theta_2 > 0))$
    $\theta_1' := (ss + gg) \cdot \theta_1 + (gs + sg) \cdot \theta_2$ ▷ Branchless parameter manipulation
    $\theta_2' := (ss + gg) \cdot \theta_2 + (gs + sg) \cdot \theta_1$
    $g_1' := (gg - ss) \cdot g_1 + (sg - gs) \cdot g_2$
    $g_2' := (gg - ss) \cdot g_2 + (gs - sg) \cdot g_1$
    $h_1' := (gg - ss) \cdot h_1 + (sg - gs) \cdot h_2$
    $h_2' := (gg - ss) \cdot h_2 + (gs - sg) \cdot h_1$
    $res \mathrel{+}= (!gg \cdot \text{erfc}((gs + ss) \cdot h_2 + sg \cdot -h_1) - ss \cdot \text{erfc}(h_1))/2$
      $+(-1 + 2 \cdot (gg + ss))$
      $\cdot$ integrateAngularRegion2$(R^2/2, \theta_1', \theta_2', g_1', g_2', h_1', h_2')$
  return $1 - res$ ▷ $\equiv P_{\text{int}}(D)$

intrinsic functions.

5.2.5 Intrinsic functions

CUDA offers a number of low throughput mathematical functions as so called intrinsic functions, which are tailored to increase throughput. These functions are only available on the device and therefore cannot be used in host code.
Intrinsic functions execute fewer native instructions compared to their standard counterparts and subsequently are faster, but they also are slightly less accurate [36]. The intrinsics for single precision floating point numbers include a number of operations that SunFlower regularly requires for its simulation:

- trigonometric functions
- exponential functions
- logarithmic functions
- power function
- square root and reciprocal square root
- fused multiply-add
- division and reciprocal

To inform the compiler to replace standard mathematical functions with their intrinsic alternatives, CUDA offers the *-use_fast_math* compiler option. Due to accuracy concerns with intrinsic functions we instead take the same approach as for the floating point data types and step through the computationally intensive code regions incrementally. For each mathematical function that is replaced we confirm that the introduced inaccuracy has no notable effect on the simulation result. During this process the deviation only raised problems with some projection and angular calculations of the Convolution raytracer. Hence these calculations are done with their standard functions. Again, the accuracy will be validated as part of the case study in Section 6.

In the following we will investigate the performance of the raytracing kernel with the Nsight Compute profiler to determine our next steps.

5.2.6 Nsight Compute performance analysis

To profile the GPU implementation, we need a larger heliostat field to have a large enough data set for its computing power. Therefore we use a heliostat field of 8600 heliostats, called AbengoaCRS. We profile a single moment simulation of this heliostat field on a NVIDIA GeForce RTX 2070 SUPER graphics card using the Nsight Compute profiler [37].

Kernel	Time spent [ms]
Raytracing kernel	946.78
Heliostat setup kernel	14.94
Result kernels	0.25

Table 4: Time spent in GPU kernels during a moment simulation of the 8600 heliostat field AbengoaCRS on a NVIDIA GeForce RTX 2070 GPU.

Table 4 lists the reported time spent in the GPU kernels we implemented (see Section 5.2.2). As expected the raytracing kernel is by far the computational hotspot of the simulation.

Figure 14 shows the by Nsight calculated floating point operation roofline model of the raytracing kernel. This model relates the theoretical limits of the hardware to code performance depending on the code's arithmetic intensity. The arithmetic intensity of a code describes how many bytes of memory the code has to load (or store) relative to the number of floating point operations (FLOPs) it conducts. In terms of hardware limitations, there are two types of computations: memory-bound and compute-bound computations. When the arithmetic intensity of a code is low, it will be limited by the memory bandwidth of the hardware and is considered memory-bound. In contrast a code that is not limited by memory bandwidth due to its high arithmetic intensity is called compute-bound.
Nsight reports a maximum memory bandwidth of about 430 GByte/s and about 8.3 TFLOP/s peak performance for single precision for the RTX 2070 graphics card. This is close to the values NVIDIA claims for this architecture [34].
The in Figure 14 depicted roofline model shows that computations on this hardware are memory-bound for single precision computations with arithmetic intensity of less than 19 FLOPs per byte. The yellow circle indicates the actual performance and arithmetic intensity of single precision computations in the raytracing kernel.
This confirms that the kernel is not making efficient use of the GPU's resources, as the performance is far off from hardware limitations. It also illustrates that the code in its current form is memory-bound.

More careful inspection of the profile reveals a possible reason for the deficient performance. As the raytracing kernel is started with one thread for each heliostat, we even for this large heliostat field start too few threads for the GPU architecture. The achieved occupancy, which is a measure of how many warps are actually active in comparison to how many warps the hardware can support, was at only 12.2%.

5.2.7 Occupancy

On the employed graphics card each SM can have up to 32 warps active simultaneously, or a total of 1024 threads. The four warp schedulers will issue instructions for warps that have all dependencies fulfilled while other warps will be

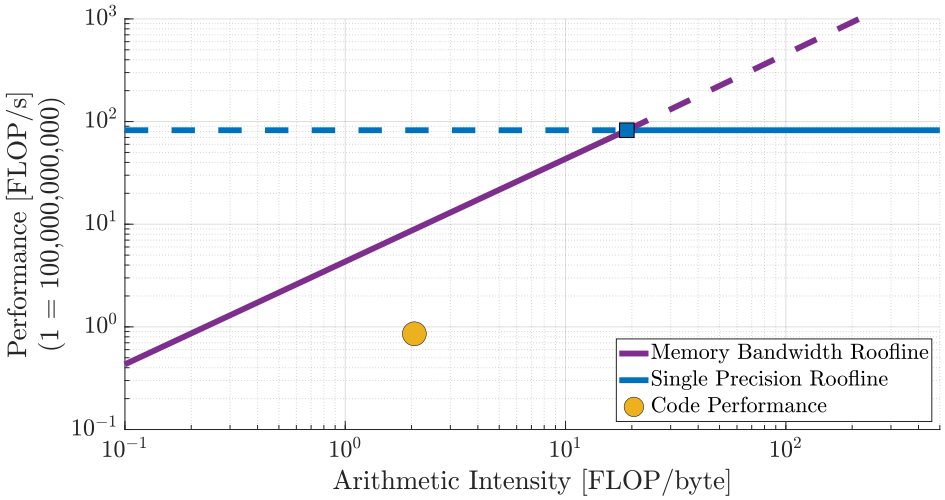


Figure 14: Floating point roofline model of the Monte-Carlo raytracing kernel during a moment simulation of a 8600 heliostat field on a NVIDIA GeForce RTX 2070 GPU. Calculated by NVIDIA Nsight Compute.

stalled. In this way the hardware can keep the processors busy and hides latencies. However, our raytracing kernel is not able to exploit this advantage of the hardware, as only one thread per heliostat is started. Even for the large heliostat field this leaves us with only close to nine thousand threads. To maximize the occupancy on the RTX 2070 GPU we need to start at least 1024 threads on all 40 SMs, or a total of 40960 threads.

Consequently we rework the parallelization of the kernel. Instead of starting one thread per heliostat, we start one thread block per heliostat. The rays generated on the heliostat are equally distributed among the threads. With this distribution each thread can evaluate it's own heliostat efficiency values in registers and at the end it adds it to the heliostat with the *atomicAdd* function. Alternative storage and reduction in shared memory was tested, but did not offer better performance.
Another considerable advantage of this parallelization is the memory access pattern. With each thread handling it's own heliostat, the data that was required by a warp was far apart and with that memory access was mostly uncoalesced. Now threads in a block access the same or nearby located data.
Using for example 256 threads per block this approach results in over two million threads for the AbengoaCRS heliostat field. This gives the GPU more than enough threads even on small heliostat fields.

A profiling of the new parallelization shows significant improvement: the kernel runtime is down by a factor of 4.5x to about 200ms.
Inspite of this progress, the achieved occupancy for this kernel merely increased to

Bounds [TpB, BpSM]	Registers per thread	Occupancy	Runtime
128, 2	239	24.09%	183.5ms
128, 3	168	35.81%	181.1ms
128, 4	128	47.89%	187.4ms
128, 8	64	95.10%	197.1ms
256, 1	239	24.10%	184.8ms
256, 2	128	47.98%	189.7ms
256, 4	64	95.13%	199.8ms

Table 5: Nsight profiling results of the raytracing kernel with varying launch boundaries specified to the compiler. The launch boundaries specify the maximum threads per block (TpB) and the minimum blocks per SM (BpSM) the kernel will be executed with.

23%. The discrepancy to the improvement in runtime affirms that this parallelization approach is advantageous not only due to higher occupancy.
Nsight Compute hints at the number of registers used per thread to be the limiting factor for the number of active warps. Every SM on the GeForce RTX 2070 card supports a total of 65536 registers, or 64 registers for each of 1024 threads. The raytracing kernel used 202 registers per thread, which makes the hardware unable to have more than 324 threads active per SM. As only full blocks are run, this is further reduced to 256 active threads per SM.

While limiting the number of registers per thread is possible, it will likely lead to register spilling, meaning data that would normally be kept in registers will repeatedly be stored to and loaded from the slow local memory. However, due to the higher occupancy the SM may be able to hide the latencies. As this tradeoff is hard to predict, we test different settings.
With the *__launch_bounds__* directive for the kernel, we can inform the compiler how many threads will be in a thread block at maximum and how many blocks the kernel will execute on each SM at minimum. The compiler will use this information to predict the optimal settings for the kernel, including the number of registers each thread is allowed to use.
Table 5 shows the profiled kernel runtime for a small set of different settings. The tested settings include cases that maximize the theoretical occupancy as well as the cases that do not limit register usage. The kernel runtimes indicate that the downside of register spillage slightly overtakes the advantage of high occupancy for more than 384 threads (128 threads per block, 3 blocks).

Figure 15 shows the roofline model Nsight Compute provides for the kernel executed with 8 blocks of 128 threads on each SM, illustrating that the register spillage results in less arithmetic intensity and the application almost reaching the maximum memory bandwidth.

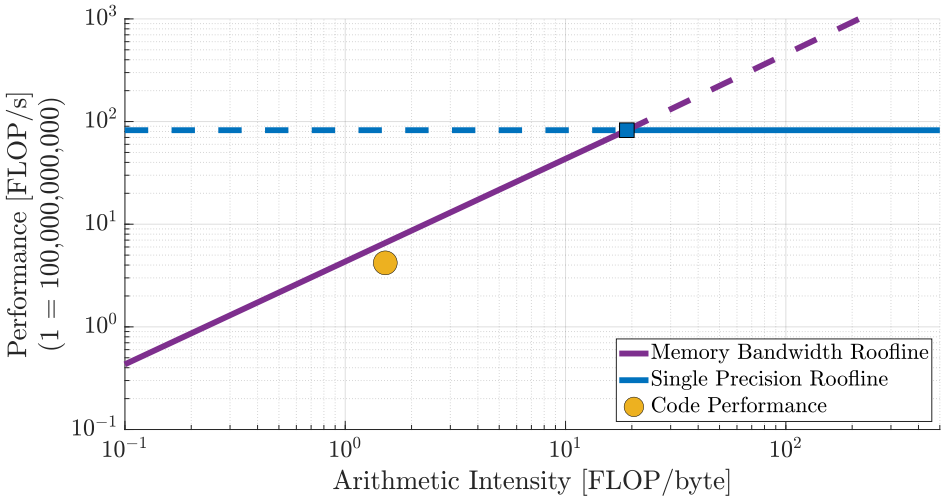


Figure 15: Roofline model of the raytracing kernel with maximized theoretical occupancy (128 threads per block, 8 blocks per SM) during a moment simulation of a 8600 heliostat field on a NVIDIA GeForce RTX 2070 GPU.

Additional information in the profile of the 128 threads per block and 3 blocks per SM launch boundary kernel verifies that this setup is also limited by memory access. Nsight Compute reports warps to be predominantly stalled due to *Long Scoreboard* dependencies, which correspond to access of data that resides outside of the SM [37]. Accordingly, further optimizations should be primarily targeted towards the memory access patterns of the application. Even though our new parallelization approach increases the data locality as neighbouring threads work on the same heliostat and access nearby data, there likely is room for improvement.

The use of shared memory was tested for frequently used data such as facet pieces or the receiver AABB tree, but no considerable advantage could be identified and space limitations raised concerns with higher discretization resolutions. Likewise it is infeasible to store all heliostat AABB trees in shared memory for large or medium sized heliostat fields.

Additionally we investigated re-ordering of some data structures. For example a pre-order storage of the AABB trees was implemented to allow efficient pre-order traversal during intersection checks. However, the performance gain was negligible. At this point no impactful and in the scope of this work feasible pathways were found to further improve the implementation in regards to memory access. However, Section 7.2 will touch on ideas that could be investigated in future research.

For that reason we conclude the optimization of the GPU implementation and

investigate its accuracy and performance in the following case study.

6 Case study

To thoroughly validate the accuracy of the new implementation and to obtain a comprehensive picture of its performance we simulate three central receiver systems: PS10, Gemasolar and AbengoaCRS.
The PS10 power plant has a small south facing heliostat field and uses a cylindric cavity receiver. Gemasolar is a medium sized concentrated solar power plant with heliostats arranged around the external cylindric receiver. AbengoaCRS is a hypothetical power plant based on data from Abengoa. It shares many of the concepts of the Gemasolar power plant but employs a much larger heliostat field. Table 6 summarizes for each power plant the key parameters we use in our simulation and Figure 16 shows the field layouts.

Parameter	PS10	Gemasolar	AbengoaCRS
Number of heliostats	624	2650	8600
Heliostat width	12.84 m	11 m	10.71 m
Heliostat height	9.45 m	10 m	12.95 m
Heliostat facet reflectance	88 %	93 %	91.96 %
Receiver shape	Cylindric cavity	External cylindrical	External cylindrical
Receiver panels	4	18	16
Receiver panel width	3.445 m	1.476 m	3.158 m
Receiver height	12 m	16 m	18.5 m
Receiver top height	115 m	126.5 m	229.5 m
Sun error (Gaussian)	2.35 mrad	2.35 mrad	2.35 mrad
Slope error	2.6 mrad	2.6 mrad	2.6 mrad
Tracking error	1.3 mrad	1.3 mrad	1.3 mrad

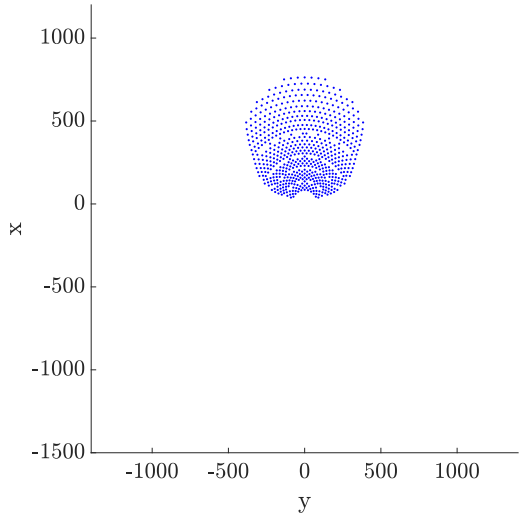
Table 6: Configuration parameters of the PS10 and the Gemasolar central receiver systems, adopted from Schöttl et al. [47], and for the hypothetical AbengoaCRS plant.

With these three power plant configurations we cover the two most common receiver types as well as a wide range of heliostat field sizes. The first goal of this study is to validate the simulation results of the new GPU implementation for these three plant configurations.

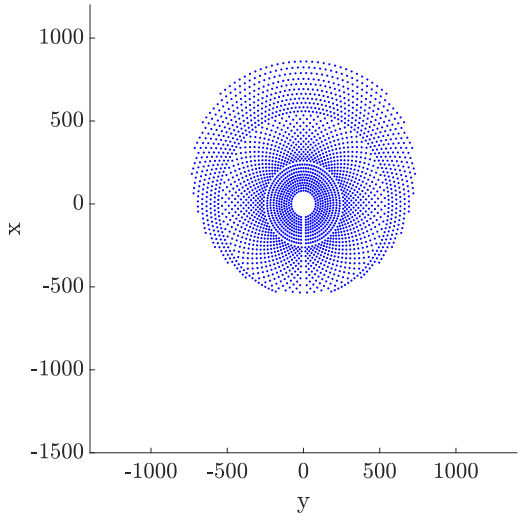
6.1 Validation

The new GPU parallelization involves a number of adaptions and optimizations with possible accuracy implications, as discussed in Section 5.2. Therefore we validate its results against the established CPU-based SunFlower code, which previously was validated against other state-of-the-art raytracing tools [43].

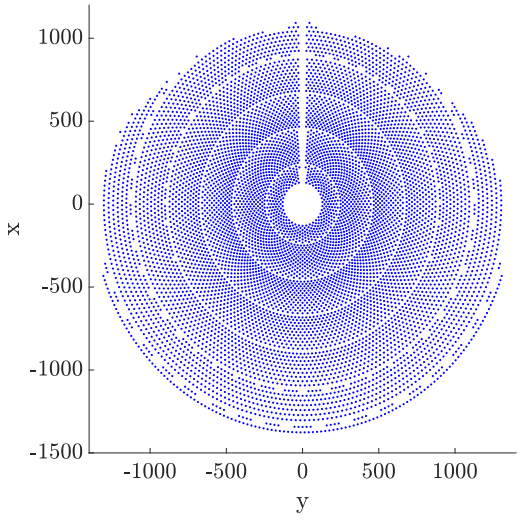
First the Monte-Carlo raytracer results are scrutinized. We simulate two moments of the 21st of June, one with low solar altitude (8AM), and one at solar noon (12PM).



(a) PS10



(b) Gemasolar



(c) AbengoaCRS

Figure 16: Heliostat field layout of the three power plants: PS10 (a), Gemasolar (b) and AbengoaCRS (c).

Each moment is simulated with varying ray densities and every simulation is repeated thirty times to account for the fluctuations of random number generation. As reference we simulate both moments with the CPU-based Monte-Carlo raytracer with 100 rays per square meter thirty times and take the mean. Figure 17 shows the maximum deviations in simulated optical power from the reference for the varying ray densities for all three power plants. In all three cases the GPU implementation yields results with an error margin of less than 0.05% even for as little as 35 rays per square meter. However, simulations with the same 100 rays per square meter discretization as the reference illustrate that the GPU raytracer results deviate by up to 0.04% in a direct comparison.

The Convolution raytracer is subjected to a similar test series. Here the simulation results are deterministic, so each simulation is only done once. The mean of the CPU Monte-Carlo raytracer results is again used as reference. The results are depicted in Figure 18. The error is, similarly to the Monte-Carlo raytracer, greater with the larger power plants Gemasolar and AbengoaCRS, but for 50 rays per square meter or more, the error margin is again less than 0.05% for all configurations.

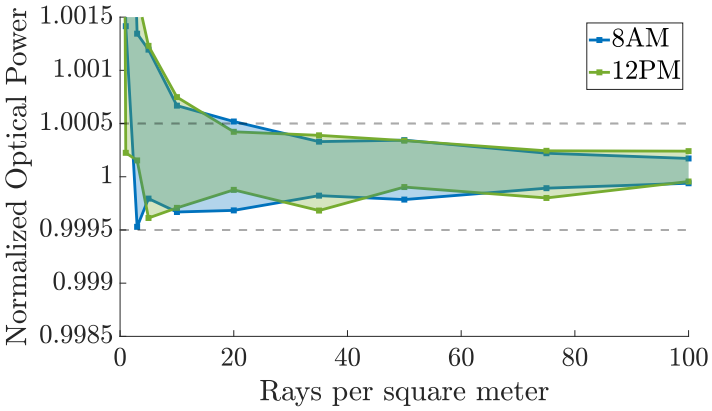
As both GPU raytracers deviate from CPU results by at most 0.05% for the three tested configurations and sensible ray discretization, we conclude that the newly implemented GPU raytracers deliver high accuracy despite any concerns raised in Section 5.2.
The next step is to investigate the performance of each raytracer.

6.2 Performance

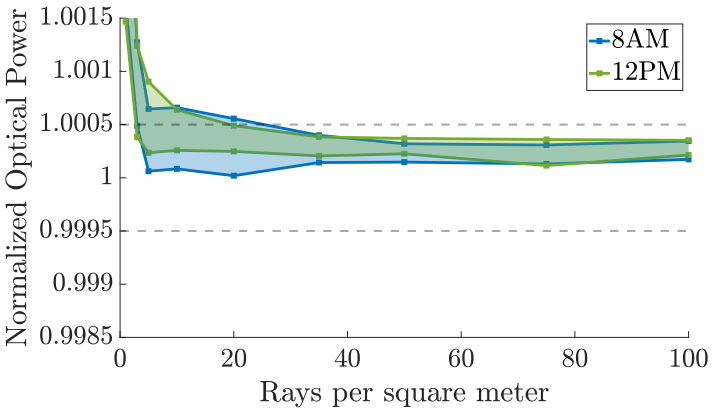
All performance tests were run on an Ubuntu 18.04LTS operating system using a NVIDIA GeForce RTX 2070 SUPER graphics card along with a Ryzen 7x 3700x CPU and 32GB of RAM.

To test the performance of each raytracer, we want to mimic common use cases rather than maximizing potential throughput. Therefore we first discuss the setup of the raytracers.
For the Convolution raytracer the setup boils down to the choice of rays per square meter. Common users will want to simulate with high accuracy of about 99.95%. Making use of the previous results, we therefore set 50 rays per square meter for the PS10 power plant, 35 rays per square meter for Gemasolar and 25 rays per square meter for AbengoaCRS to achieve this.

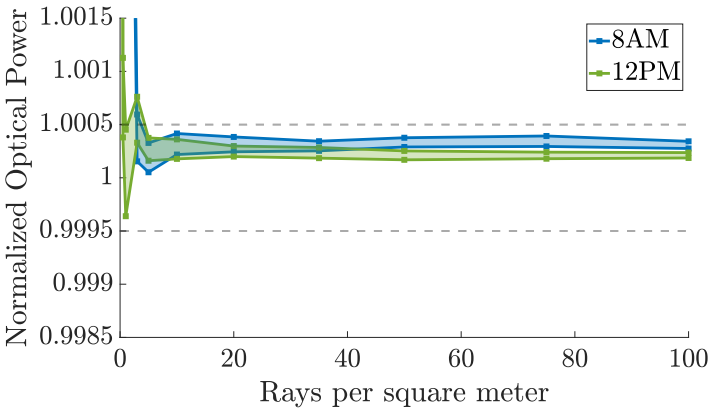
In case of the Monte-Carlo raytracer the setup involves the additional choice of the number of repeated samplings of each ray. Choosing to repeatedly sample each ray reduces the fluctuations in results, but also increases the runtime of the simulation. To gauge the impact on result fluctuation, a set of different configurations is tested. We vary the number of repeated samplings M and the rays per square meter and measure



(a) PS10

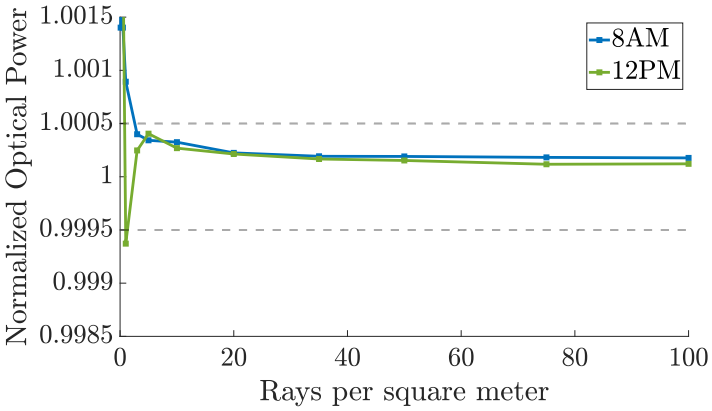


(b) Gemasolar

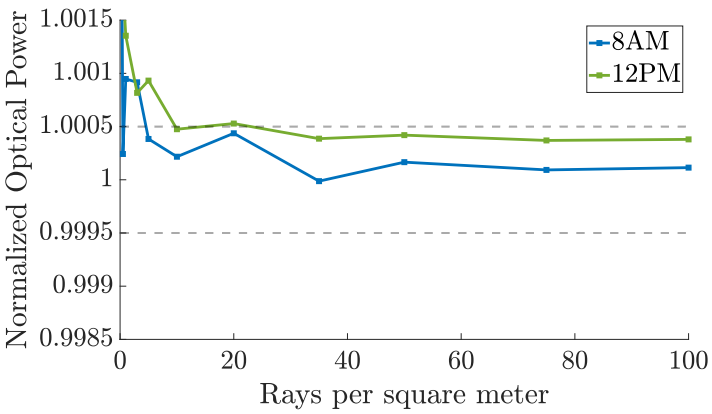


(c) AbengoaCRS

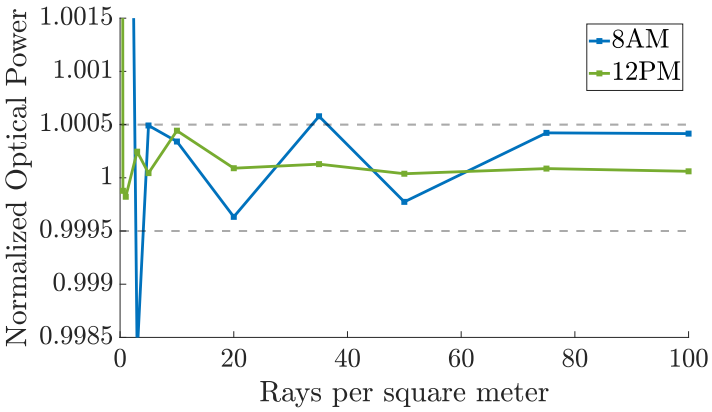
Figure 17: Maximum deviations of the optical power for varying ray densities of the GPU Monte-Carlo raytracer for two representative moments of the 21st of June. The reference result is obtained by simulation with the established CPU-based SunFlower Monte-Carlo raytracer with 100 rays per square meter.



(a) PS10



(b) Gemasolar



(c) AbengoaCRS

Figure 18: Maximum deviations of the optical power for varying ray densities of the GPU Convolution raytracer for two representative moments of the 21st of June. The reference result is obtained by simulation with the established CPU-based SunFlower Monte-Carlo raytracer with 100 rays per square meter.

both the maximum deviation in optical power and the average runtime across thirty simulations. The same reference as for the validation tests is used. Figure 19 shows the runtime in relation to the maximum deviation for increasing ray densities and six different number of samplings M.

These results illustrate that the impact of the repeated sampling of rays on accuracy is negligible for all our test cases. In fact, if one wants to reduce fluctuation and increase accuracy, this indicates that it is more efficient to simply increase the number of rays per square meter.

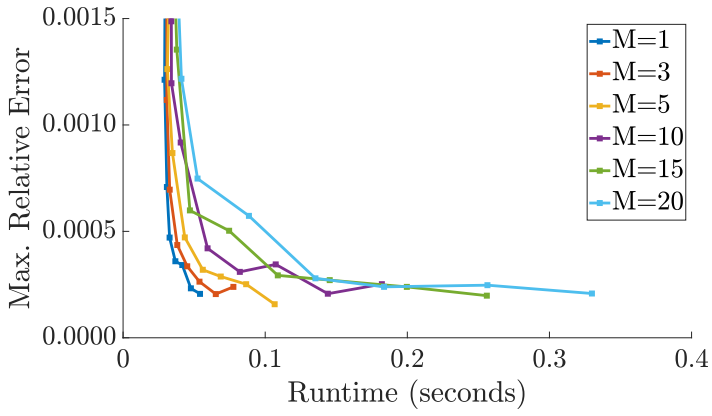
Thus we set $M = 1$ and again use the results of our validation tests to set up the rays per square meter. In short: the same ray discretization is used for the Monte-Carlo raytracer as for the Convolution raytracer to mimic typical simulations with accuracies of about 99.95%.

Apart from accurate raytracing for each moment, a prominent simulation goal is to compute the annual energy production to a similarly high degree of accuracy. While SunFlower offers various configurations for the annual simulation and integration, the resulting optical simulation only differs in the number and choice of simulated moments. For instance the state-of-the-art *monthly nearest neighbour approximation* [39] is implemented and will simulate about 80 to 100 distinct moments (depending on power plant location) to approximate the annual energy production to accuracies of typically above 99.5% [41]. As it is one of the most common methods for annual energy calculation, we will use this configuration for our performance tests.

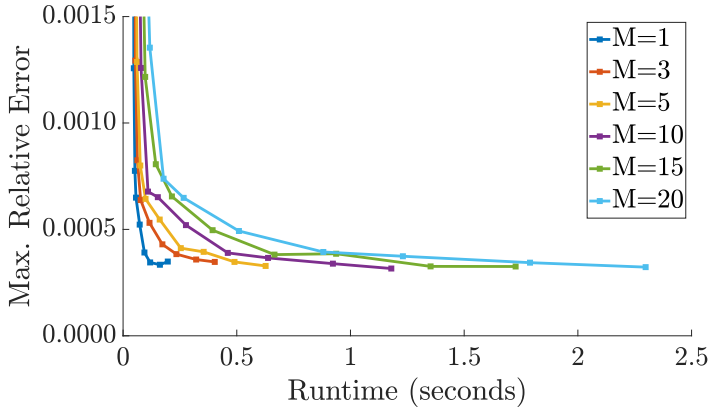
The first objective is to investigate the impact of the implemented optimizations and to compare the performance of the new CPU and GPU parallelizations. Accordingly the tests are run with four different code iterations:

1. **CB-Serial**: CB being short for *codebase*, this refers to the initial code version unaltered and without parallelization.
2. **OPT-Serial**: this is the according to Section 4 revised (*OPT*imized) code without parallelization.
3. **OPT-OpenMP**: the revised code with basic OpenMP parallelization, see Section 5.1.
4. **OPT-CUDA**: the revised, in CUDA C re-implemented and on GPU parallelized code, see Section 5.2.

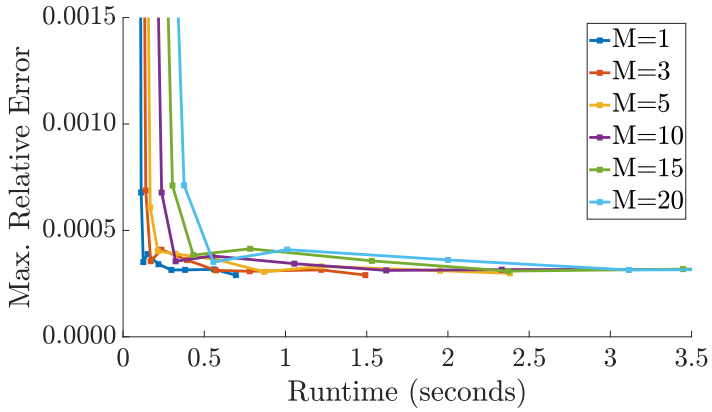
To summarize: for each of the four code versions we measure the runtime of a simulation of the annual energy production with the aforementioned raytracer configurations and the *monthly nearest neighbor* method on each of the three power plants.



(a) PS10



(b) Gemasolar



(c) AbengoaCRS

Figure 19: Maximum deviation of the optical power for increasing ray densities of the Monte-Carlo raytracer in relation to the runtime. Six different number of repeated samplings M are displayed.

	Monte-Carlo		Convolution	
Code iteration	Runtime	Speedup	Runtime	Speedup
CB-Serial	6699.3s	1.00x	42417.6s	1.00x
OPT-Serial	1518.3s	4.41x	11187.8s	3.79x
OPT-OpenMP	374.8s	17.87x	1409.2s	30.10x
OPT-CUDA	**5.4s**	**1248.53x**	**19.4s**	**2184.02x**

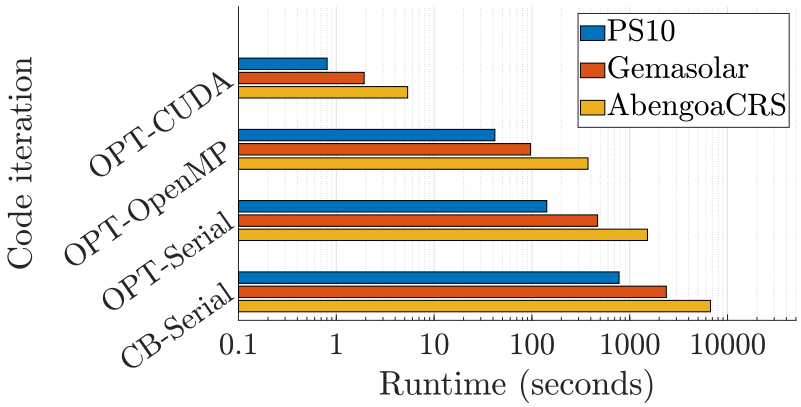
Table 7: Runtime and corresponding speedup of the AbengoaCRS simulation with the Monte-Carlo and Convolution based raytracer for four different code iterations.

Figure 20a gives a comparison of the runtime of the Monte-Carlo raytracer for the different code versions and the three power plants.
While the initial code *CB-Serial* required 778 seconds to execute the annual simulation of PS10 and exceeded 6600 seconds for AbengoaCRS, these runtimes are down to about 140 seconds and 1500 seconds respectively with the optimized serial code *OPT-Serial*. Basic parallelization on CPU with OpenMP further reduced the runtime to 42 seconds for PS10 and 375 seconds for AbengoaCRS. The final parallelization on GPU *OPT-CUDA* in comparison finished the annual simulation of PS10 in 800 milliseconds and AbengoaCRS in 5.36 seconds.

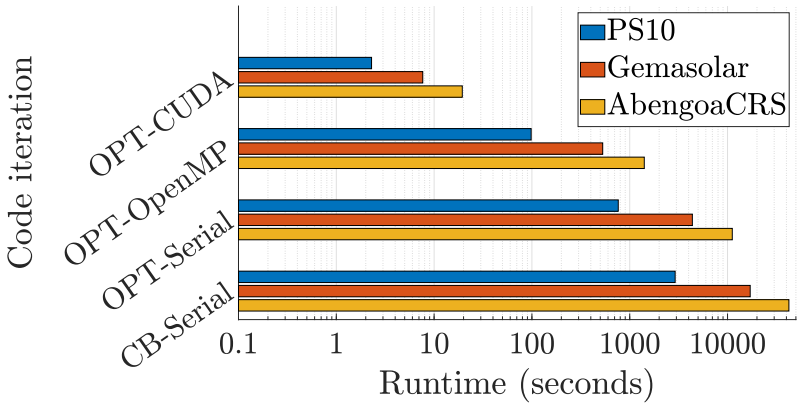
Figure 20b shows the corresponding results for the Convolution raytracer. The annual simulation of AbengoaCRS with the codebase *CB-Serial* in this case took close to 12 hours or 42400 seconds. The optimizations of Section 4 reduce this by almost factor four to 11000 seconds with the *OPT-Serial* code. While OpenMP parallelization significantly improves this even further to 1400 seconds, the CUDA parallelization on GPU in comparison squashes the runtime to only 19.5 seconds.

To get a clearer picture of the relative improvements, Figure 21 depicts the speedup that the revised code iterations achieve in comparison to the codebase *CB-Serial* on each power plant. Speedup here refers to the factor by which the runtime is reduced. Optimizations to the serial code accelerate annual simulations by about factor five for the Monte-Carlo and about factor three for the Convolution raytracer. Adding OpenMP parallelization, the initial execution times are reduced by factors of roughly 20 and 30 for the Monte-Carlo and Convolution raytracer respectively. The on GPU parallelized raytracers achieve overall speedups of over 1000 and close to 2000.
An overview of these results for the AbengoaCRS simulations is given in Table 7.

After inspection of the accomplished performance gains, the next step is to compare the performance of the new GPU parallelized raytracers to that of other state-of-the-art tools.

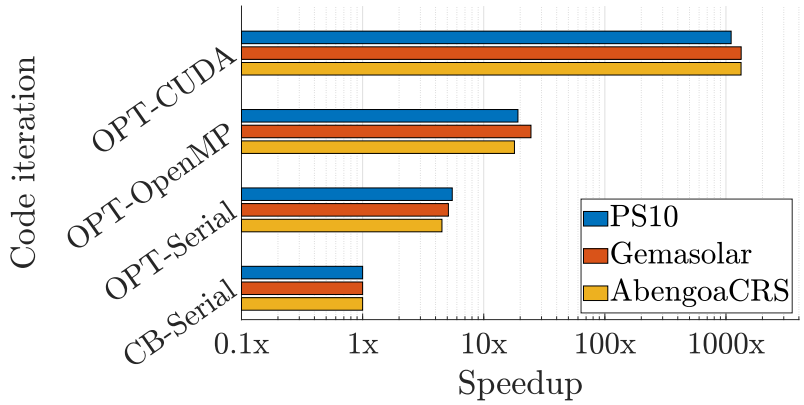


(a) Monte-Carlo raytracer

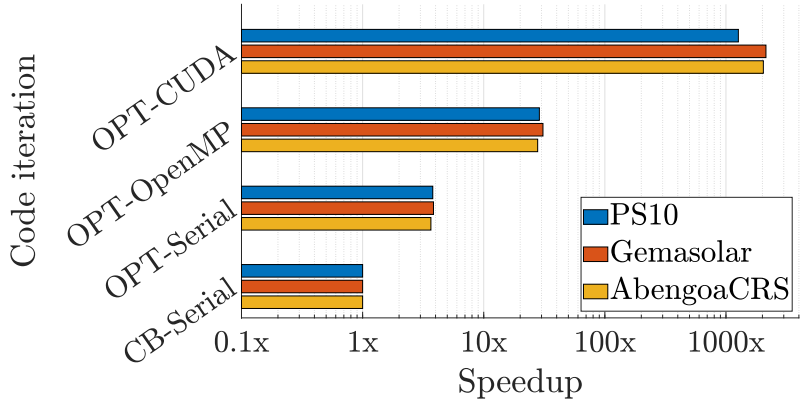


(b) Convolution Raytracer

Figure 20: Comparison of the runtime of annual simulations with four different code iterations and three power plants for the Monte-Carlo (a) and Convolution (b) raytracers.



(a) Monte-Carlo raytracer



(b) Convolution Raytracer

Figure 21: Speedup of the revised code iterations compared to the codebase for the Monte-Carlo (a) and Convolution (b) raytracers.

6.3 Performance comparison to state-of-the-art tools

Other high-performance central receiver system raytracers could not be tested by us directly as they are only commercially or not at all available. Thus we have to formulate a superficial comparison with the performance that is reported by the developers. We were able to determine approximate performance magnitudes of three state-of-the-art GPU parallelized Monte-Carlo raytracers:

- **TieSOL** is an established commercial raytracer that reports tracing 100 million rays in 887 milliseconds on three GTX 570 GPUs [26].
- **QMCRT** is a Quasi-Monte-Carlo raytracer recently developed by a chinese research team which reports tracing 200 million rays in 1.5 seconds [15].
- **sbpRAY** is a very fast commercial Monte-Carlo raytracer and optimization tool that claims to produce converged results for a heliostat field of 25000 Stellio heliostats in one second and close to half that for subsequent simulations [19]. Stellio heliostats have $47.5m^2$ of surface area [38]. We approximate that using 20 rays per square meter should produce a converged result. This would correspond to 237.5 million rays per second or close to 470 million rays per second when considering subsequent simulations.

The annual simulations with SunFlower's GPU Monte-Carlo raytracer traced on average 534.5 million rays per second on the PS10 power plant, 576 million rays per second on Gemasolar and 564 million rays per second on AbengoaCRS. Figure 22 illustrates the reported performances of each tool in comparison to that of SunFlower's GPU Monte-Carlo raytracer.

Comparing the Convolution raytracer is more difficult, as we could find only two reports of non-commercial GPU parallelized analytical raytracers [22, 12].
Developed by the same team as QMCRT, the HFLCAL-adopted [46] analytical raytracer by He et al. [22] is fast but involves significant inaccuracies. He et al. [23] recently published another research which proposes a more accurate version of their analytical raytracer, called *iHFLCAL*, but did not report usable performance metrics. Therefore we will use the values reported in the initial study for comparsion with SunFlower.
As this analytical raytracer does only trace a single ray per heliostat, the research team takes the number of traced rays and multiplies it with the receiver discretization to give a performance comparison to other raytracers. As computational workload and accuracy of analytical raytracers is tightly bound to receiver discretization, we find the proposed estimation to be reasonable to directly compare the performance of analytical raytracers. For that we will denote the product of traced rays and number of receiver pieces as $R \times P$. He et al. [22] report tracing 624 rays for PS10 against $600 \times 268 \times 4$ receiver pieces in 67.1 milliseconds. This equates to 401 million $R \times P$ in 67.1 milliseconds or about 6 billion $R \times P$ per second. SunFlower's Convolution

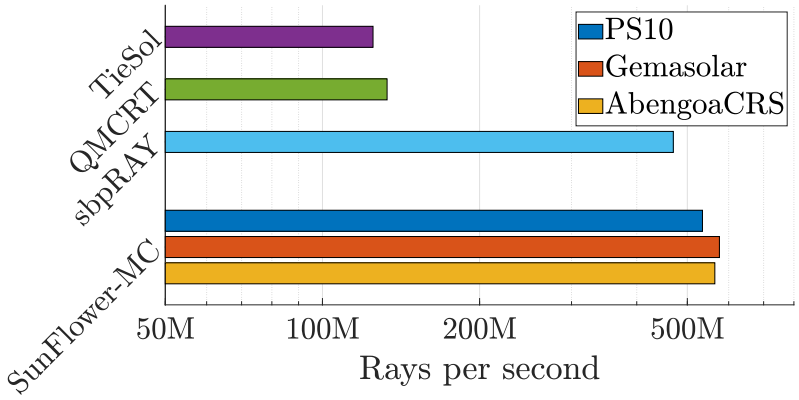


Figure 22: Performance comparison of SunFlower's Monte-Carlo raytracer with three state-of-the-art GPU Monte-Carlo raytracers: TieSOL [26], QMCRT [15] and sbpRAY[19].

raytracer on PS10 traces 7.2 million rays per second against $20 \times 20 \times 4$ receiver pieces, equating to 11.5 billion $R \times P$ per second according to the proposed calculation. Furthermore they claim the maximum number of rays to be reached for 30000 heliostats and a $65 \times 65 \times 8 \times 8$ receiver resolution. The corresponding 8.11 billion $R \times P$ are evaluated in 618 milliseconds, resulting in 13.12 billion $R \times P$ per second. In comparison SunFlower's maximum throughput in the test cases was on the Gemasolar power plant with 18.47 billion $R \times P$ per second.

Chiesi et al. [12] developed an analytical raytracer with GPU parallelization that can be compared in the same way. It is reported that 12000 heliostats of $1m^2$ are discretized into 20×20 cells and traced against 40×40 receiver pieces in 4.33 seconds. This results in 1.77 billion $R \times P$ per second. Figure 23 gives a visual comparison of the performance of all three analytical raytracers.

6.4 Discussion of the results

The validation study verifies that both GPU parallelized raytracers are able to deliver highly accurate results inspite of various adaptions. The error margin was found to be within 0.05% for both raytracers and all three simulated power plants, see Figure 17 and 18. The validation study indicates that simulation results are of high quality for any configuration of receiver and heliostats as well as both small and large heliostat fields.

Besides accuracy, the results in Section 6.2 exemplify that the performance in total was improved by atleast three orders of magnitude in all cases, as Figure 21 illustrates. Optimizations to the serial execution sped up simulations of the Monte-Carlo

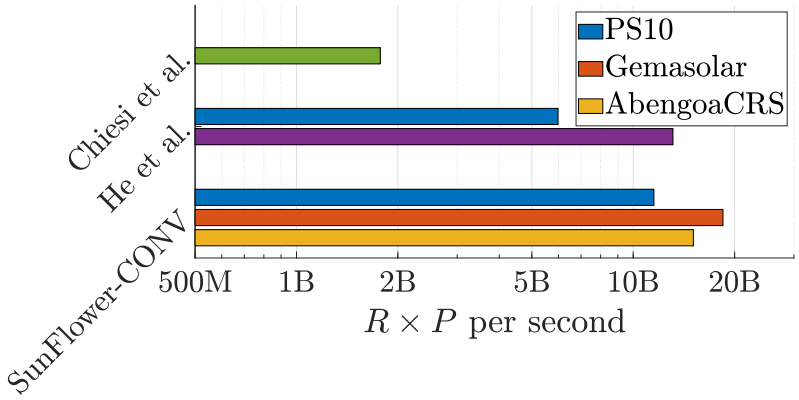


Figure 23: Performance comparison of SunFlower's Convolution raytracer with two analytical raytracers on GPU: He et al. [22] and Chiesi et al. [12].

raytracer already by factor five and the Convolution raytracer by factor three. With this improvement as starting point, the GPU parallelization was able to again reduce runtimes by well over factor 200. Furthermore the GPU implementation represents a speedup of above factor 50 compared to the OpenMP based CPU parallelization. Figure 20 also shows that runtime scales roughly linearly with the size of the heliostat field.

As a main task of heliostat field optimization is to repeatedly evaluate annual energy production, the implications of this study are substantial. For example a typical heliostat field optimization could be done on two parameters with each 50 levels, resulting in a total of 2500 combinations to test. For each combination the annual energy production has to be evaluated. With the previous version of SunFlower raytracers, this was only feasible for small heliostat fields and mediocre accuracy and still involved immense runtimes of multiple days. The revised GPU raytracers presented in this work allow to complete this task with high accuracy in acceptable time frames, even for large heliostat fields of ten thousand and more heliostats.

In case of the largest tested heliostat field, AbengoaCRS, the runtime of an annual simulation with the Monte-Carlo raytracer is down from close to two hours to merely 5.3 seconds. This can further be reduced by about factor two to three by choosing a smarter approximation method for the annual energy production [41]. The annual simulations conducted in this case study used the *monthly nearest neighbor* approach and simulated 98 moments. SunFlower offers to instead approximate the annual energy production to similar accuracy with a Lagrange approximation using only 30 to 50 simulation moments. As the workload scales linearly with the number of simulated moments, this means an annual simulation of a field with 8600 heliostats and 99.95% accurate results can be run in about 2 seconds. Accordingly, a heliostat field optimization with 2500 annual simulations would take just close to 1.5 hours.

Comparison to other state-of-the-art tools showed that the performance of both the

Monte-Carlo and the Convolution raytracer can easily compete with other GPU parallelized raytracers. Our Monte-Carlo raytracer achieved several times the throughput of the well respected TieSOL [26]. Figure 22 shows that even the recent QMCRT [15] and sbpRAY [19] raytracers are outperformed by SunFlower in all test cases. Similarly, our Convolution raytracer beats out the performance of both He et al. [22] and Chiesi et al. [12] in all test cases, see Figure 23.

7 Conclusion and Outlook

7.1 Conclusion

This work presents how raytracers for solar central receiver systems can be optimized and efficiently parallelized on GPU. The Monte-Carlo based raytracing method as well as the analytical Convolution raytracing approach are considered. Both previously in the SunFlower tool implemented raytracers for the respective approaches are investigated. Based on profiling information gathered with Intel VTune compuational hotspots are identified and optimizations to the serial execution are applied. The filtering of blocking and shading heliostats is enhanced and intersection checks with AABB trees are improved by careful handling of the bounding volume hierarchy. Along with increasing the efficiency of existing code and utilization of the simulation structure of annual simulations, the serial execution is sped up by factor three to five for both raytracers.

The entire optical model, basic geometry code and both raytracers of SunFlower are re-implemented in the CUDA C programming language. Multiple GPU kernels are introduced and several performance considerations are discussed for the GPU parallelization. Optimizations are layed out and implemented accordingly. To identify remaining performance bottlenecks the kernel performance is profiled and analysed with the help of NVIDIA Nsight Compute. Final improvements are applied based on this analysis.

The comprehensive case study on three different plant configurations on one hand confirms that the accuracy of the GPU implemented raytracers remains within 0.05% of that of the previous, against SolTrace [49] and Tonatiuh [7] validated CPU raytracer. On the other hand it shows that the performance in total is improved by factors of up to 2000 when compared to the initial serial code. Moreover the GPU parallelization represents a speedup of above factor 50 in direct comparison to an OpenMP parallelization on CPU.
In summary, the revised GPU raytracers allow highly accurate and immensely quick simulation of any heliostat field. Most importantly this tremendously speeds up optimization of heliostat field layouts, as annual simulation runtimes are reduced from multiple hours to at worst a few seconds. This makes accurate optimization possible even for extremely large heliostat fields, which previously was infeasible due to enor-

mous runtimes of days or weeks.
Additionally it is illustrated that the implemented GPU parallelized raytracers outperform the published performance of other GPU parallelized central receiver system raytracing tools in all of our test cases.

7.2 Outlook

Although the new GPU parallelization yields impressive performance improvements, the kernel profiling presented in Section 5.2 indicated that the raytracing kernel is roughly one order of magnitude below peak FLOP throughput of the employed graphics card and memory-bound. Accordingly the memory access patterns of the raytracing kernel should be targeted to further improve performance.

For example, each thread block handles exactly one heliostat, thus the threads in each block share heliostat-level blocking and shading candidates. As this typically amounts to just a handful of candidates it may be feasible to prefetch the AABB trees and triangle mesh of blocking and shading candidates into shared memory to eliminate the otherwise inevitable uncoalesced access of global memory during blocking and shading checks.

A more intricate but possibly impactful approach would be to divide the raytracing kernel into multiple smaller kernels that each ensure well optimized memory access. Currently each thread of the raytracing kernel evaluates multiple rays from their generation through to storing their final result. Instead it might prove more effective to decouple this process into stages, temporarily storing results of each stage in shared memory. This might allow to reduce execution and data divergence by adding a small step inbetween stages that sorts the work such that nearby threads can work efficiently.

Aside from directly improving code efficiency, the heliostat field optimization can also be sped up further by sharing the workload on multiple systems. While the workload of a single annual simulation can be efficiently parallelized on a single graphics card, as this work showcases, the large number of annual simulations necessary for layout optimizations could be distributed among multiple GPUs. Depending on the optimization approach, communication of results across systems may be necessary only sporadically or not at all. This could reduce optimization runtimes for large heliostat fields to just a few minutes.

References

[1] Nils Ahlbrink, Boris Belhomme, Robert Flesch, Daniel Maldonado Quinto, Amadeus Rong, and Peter Schwarzbözl. STRAL: Fast ray tracing software with tool coupling capabilities for high-precision simulations of solar thermal power plants. In *Proceedings of the SolarPACES 2012 conference*, 2012.

[2] Viorel Badescu. *Modeling Solar Radiation at the Earth's Surface*, volume 1. Springer, 2014.

[3] Tavian Barnes. Fast, branchless ray/bounding box intersections, 2011. URL `https://tavianator.com/2011/ray_box.html`. Last visited: June 2021.

[4] Omar Behar, Abdallah Khellaf, and Kamal Mohammedi. A review of studies on central receiver solar thermal power plants. *Renewable and sustainable energy reviews*, 23, 2013.

[5] F. Biggs and C. N. Vittitoe. HELIOS: A computational model for solar concentrators. Technical report, Sandia National Laboratories, 1977.

[6] David Blackman and Sebastiano Vigna. Scrambled Linear Pseudorandom Number Generators. *arXiv*, 2018.

[7] Manuel J. Blanco, Juana M. Amieva, and Azael Mancillas. The tonatiuh software development project: An open source approach to the simulation of solar concentrating systems, 2005.

[8] Sebastian-James Bode and Paul Gauché. Review of optical software for use in concentrating solar power systems. In *Proceedings of South African Solar Energy Conference*, 2012.

[9] Juan Ignacio Burgaleta, Santiago Arias, and Diego Ramirez. Gemasolar, the first tower thermosolar commercial plant with molten salt storage. In *Proceedings of the SolarPACES 2011 conference*, 2011.

[10] Russel E. Caflisch. Monte carlo and quasi-monte carlo methods. *Acta numerica*, 1998.

[11] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers, 2001.

[12] Matteo Chiesi, Luca Vanzolini, Eleonora Franchi Scarselli, and Roberto Guerrieri. Accurate optical model for design and analysis of solar fields based on heterogeneous multicore systems. *Renewable Energy*, 55:241–251, 2013. doi: 10.1016/j.renene.2012.12.025.

[13] N.C. Cruz, J.L. Redondo, M. Berenguel, J.D. Álvarez, and P.M. Ortigosa. Review of software for optical analyzing and optimizing heliostat fields. *Renewable and Sustainable Energy Reviews*, 72:1001–1018, 2017. doi: 10.1016/j.rser.2017.01.032.

[14] A. R. Didonato, Jr. M. P. Jarnagin, and R. K. Hageman. Computation of the integral of the bivariate normal distribution over convex polygons. *SIAM Journal on Scientific and Statistical Computing*, 1(2):179–186, 1980. doi: 10.1137/0901010.

[15] Xiaoyue Duan, Caitou He, Xiaoxia Lin, Yuhong Zhao, and Jieqing Feng. Quasi-monte carlo ray tracing algorithm for radiative flux distribution simulation. *Solar Energy*, 211:167–182, 2020. doi: 10.1016/j.solener.2020.09.061.

[16] Linus Franke. Modelling and optimization of large scale solar tower power plants modellierung und optimierung von solarturm-kraftwerken. Master thesis, RWTH Aachen University, 2018.

[17] L. García, M. Burisch, and M. Sanchez. Spillage estimation in a heliostats field for solar field optimization. *Energy Procedia*, 69:1269–1276, 2015. doi: 10.1016/j.egypro.2015.03.156.

[18] Pierre Garcia, Alain Ferriere, and Jean-Jacques Bezian. Codes for solar flux calculation dedicated to central receiver system applications: A comparative review. *Solar Energy*, 82(3):189–197, 2008. doi: 10.1016/j.solener.2007.08.004.

[19] Daniel Gebreiter, Gerhard Weinrebe, Markus Wöhrbach, Florian Arbes, Fabian Gross, and Willem Landman. sbpRAY – a fast and versatile tool for the simulation of large scale CSP plants. In *Proceedings of the SolarPACES 2018 conference*, 2019. doi: 10.1063/1.5117674.

[20] Michael Geyer and William B. Stine. Power From The Sun, 2001. URL `www.powerfromthesun.net/book.html`. Last visited: June 2021.

[21] E. R. Golder and J. G. Settle. The box-muller method for generating pseudo-random normal deviates. *Applied Statistics*, 25(1):12, 1976. doi: 10.2307/2346513.

[22] Caitou He, Jieqing Feng, and Yuhong Zhao. Fast flux density distribution simulation of central receiver system on GPU. *Solar Energy*, 144:424–435, 2017. doi: 10.1016/j.solener.2017.01.025.

[23] Caitou He, Xiaoyue Duan, Yuhong Zhao, and Jieqing Feng. An analytical flux density distribution model with a closed-form expression for a flat heliostat. *Applied Energy*, 251:113310, 2019. doi: 10.1016/j.apenergy.2019.113310.

[24] Caitou He, Yuhong Zhao, and Jieqing Feng. An improved flux density distribution model for a flat heliostat (iHFLCAL) compared with HFLCAL. *Solar Energy*, 189: 116239, 2019. doi: 10.1016/j.energy.2019.116239.

[25] Florian Hoevelmann. Accelerated Raytracer for Solar Tower Power Plants. Bachelor thesis, RWTH Aachen University, 2019.

[26] Michel Izygon, Peter Armstrong, Claus Nilsson, and Ngoc Vu. TieSOL – a GPU-based suite of software for central receiver solar power plants. *Proceedings of the SolarPACES 2011 conference*, 2011.

[27] David Jafrancesco, Joao P. Cardoso, Amaia Mutuberria, Erminia Leonardi, Iñigo Les, Paola Sansoni, Franco Francini, and Daniela Fontani. Optical simulation of a central receiver system: Comparison of different software tools. *Renewable and Sustainable Energy Reviews*, 94:792–803, 2018. doi: 10.1016/j.rser.2018.06.028.

[28] Tero Karras. Tree traversal on the gpu, 2012. URL `developer.nvidia.com/blog/thinking-parallel-part-ii-tree-traversal-gpu/`. Last visited: June 2021.

[29] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques - SIGGRAPH '86*. ACM Press, 1986. doi: 10.1145/15922.15916.

[30] P. L. Leary and J. D. Hankins. User's guide for MIRVAL: a computer code for comparing designs of heliostat-receiver optics for central receiver solar power plants. Technical report, Sandia National Laboratories, 1979.

[31] Lifeng Li, Joe Coventry, Roman Bader, John Pye, and Wojciech Lipiński. Optics of solar central receiver systems: a review. *Optics Express*, 24(14):A985, 2016. doi: 10.1364/oe.24.00a985.

[32] Tomas Möller and Ben Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997. doi: 10.1080/10867651.1997.10487468.

[33] Corey J. Noone, Manuel Torrilhon, and Alexander Mitsos. Heliostat field optimization: A new computationally efficient model and biomimetic layout. *Solar Energy*, 86(2):792–803, 2012. doi: 10.1016/j.solener.2011.12.007.

[34] NVIDIA. Geforce RTX 2070 SUPER, 2020. URL `www.nvidia.com/en-us/geforce/graphics-cards/rtx-2070-super/`. Last visited: June 2021.

[35] NVIDIA. Random Number Generation on NVIDIA GPUs, 2021. URL `developer.nvidia.com/curand`. Last visited: June 2021.

[36] NVIDIA. CUDA Toolkit v11.3.0, 2021. URL `docs.nvidia.com/cuda/`. Last visited: June 2021.

[37] NVIDIA. Nsight Compute, 2021. URL `developer.nvidia.com/nsight-compute`. Last visited: June 2021.

[38] Schleich Bergermann Partner. Stellio Heliostat, 2015. URL `https://www.sbp.de/en/project/stellio-heliostat/`. Last visited: June 2021.

[39] Robert Pitz-Paal, Nicolas Bayer Botero, and Aldo Steinfeld. Heliostat field layout optimization for high-temperature solar thermochemical processing. *Solar Energy*, 85(2):334–343, 2011. doi: 10.1016/j.solener.2010.11.018.

[40] Ari Rabl. *Active Solar Collectors and Their Applications*. Oxford University Press, 1985.

[41] P. Richter, J. Tinnes, and L. Aldenhoff. Accurate interpolation methods for the annual simulation of solar central receiver systems using celestial coordinate system. *Solar Energy*, 213:328–338, 2021. doi: 10.1016/j.solener.2020.10.087.

[42] Pascal Richter. Simulation and Optimization of Solar Thermal Power Plants. Ph.d. thesis, RWTH Aachen University, 2017.

[43] Pascal Richter, Gregor Heiming, Nils Lukas, and Martin Frank. SunFlower: A new solar tower simulation method for use in field layout optimization. In *Proceedings of the SolarPACES 2017 conference*, 2018. doi: 10.1063/1.5067217.

[44] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*. ACM Press, 2011. doi: 10.1145/2063384.2063405.

[45] Mark Schmitz, Peter Schwarzbözl, Reiner Buck, and Robert Pitz-Paal. Assessment of the potential improvement due to multiple apertures in central receiver systems with secondary concentrators. *Solar Energy*, 80(1):111–120, 2006. doi: 10.1016/j.solener.2005.02.012.

[46] Peter Schwarzbözl, Robert Pitz-Paal, and Mark Schmitz. Visual HFLCAL - A software tool for layout and optimisation of heliostat fields. In *Proceedings of the SolarPACES 2009 conference*, 2009.

[47] Peter Schöttl, Karolina Ordóñez Moreno, De Wet van Rooyen, Gregor Bern, and Peter Nitz. Novel sky discretization method for optical annual assessment of solar tower plants. *Solar Energy*, 138:36–46, 2016. doi: 10.1016/j.solener.2016.08.049.

[48] Fethulah Smailbegovic, Georgi N. Gaydadjiev, and Stamatis Vassiliadis. Sparse Matrix Storage Format. In *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing*, pages 445–448, 2005.

[49] Tim Wendelin. SolTrace: A New Optical Modeling Tool for Concentrating Solar Optics. In *Solar Energy*. ASMEDC, jan 2003. doi: 10.1115/isec2003-44090.

[50] Tim Wendelin, Aron Dobos, and Allan Lewandowski. SolTrace: A Ray-Tracing code for complex solar optical systems. Technical report, 2013.

[51] Amy Williams, Steve Barrus, R. Keith Morley, and Peter Shirley. An efficient and robust ray-box intersection algorithm. In *ACM SIGGRAPH 2005 Courses on - SIGGRAPH '05*. ACM Press, 2005. doi: 10.1145/1198555.1198748.