Karlsruhe Institute of Technology

GPU Ray Tracer for Solar Tower Power Plants

Bachelorarbeit
Mathematik

Juli 2019

Vorgelegt von Presented by	Carlos Schmidt Muniz carlos.muniz@student.kit.edu
Erstprüfer First examiner	Prof. Dr. rer. nat. Martin Frank Steinbuch Centre for Computing Karlsruhe Institute of Technology
Zweitprüfer Second examiner	PD Dr. rer. nat. Gudrun Thäter Department of Mathematics Karlsruhe Institute of Technology
Koreferent Co-Supervisor	Dr. rer. nat. Pascal Richter Steinbuch Centre for Computing Karlsruhe Institute of Technology

Contents

Nomenclature

μ_r	mirror reflectivity
i	$i \in \mathbb{N}_0$ with $i = 0$ being the receiver
P_i	parametric form of heliostat i
v_i, w_i	directions of heliostat i
n_i	normal of heliostat i
$\|\cdot\|$	standard euclidean norm
φ	azimuth of the sun
θ	altitude of the sun
s	sun's direction
d	direction of ray
η	intersection of ray with receiver
o	origin of ray
μ_{aa}	atmospheric attenuation
μ_c	cosine effect
DNI	direct normal irradiance
p_r	power value of ray
(r, α, β)	spherical coordinates
n_r	normal of ray
host	central processing unit CPU
device	graphics processing unit GPU
S	potential speed-up of a task
U	processing units working in parallel
f	fraction of task which can be parallelized

1 Introduction

Concentrating Solar Power (CSP) plants are becoming increasingly important as demand for energy from renewable sources rises. One key differentiator of this technology is the possibility to use high capacity storage at moderate cost and with a total loss of only 3%. The heat of the concentrated solar radiation can be stored in big molten salt tanks, where it changes the temperature of the salt. At times of high demand, the heat can be brought back into the cycle to generate electricity. The storage of thermal energy is as of now not only cheaper but also more environmentally friendly than the storage of electricity. This constitutes an enormous advantage for the ability to supply electricity on demand overcoming the stochastic fluctuation of solar and wind energy availability. This CSP plants combined with adequate storage capacity may play an important role in the transformation of the electricity system from a fosile-fuel based to a renewable source based one.
The maximization of the revenue of a CSP plant requires the consideration and optimization of a number of different elements such as output estimation or design of the plant. The position of the mirrors as well as the receiver(s) play an especially important role. A fast simulation of the CSP plant is essential because building it would be very costly. Given the simulation results it is for example possible to change the design of the plant before it is built. Several programs already exist which estimate the exact power output during a selected time period. There are different approaches on how to model the CSP plants and the simulation of the sun rays being reflected on the mirrors. The most common approach is a Monte Carlo simulation of the outgoing rays of the sun. The nature of this simulation is probabilistic which means that the more rays are generated and traced the more accurate the solution gets. Each ray is traced independently since they do not affect each other or interfere with their trajectory. Part of the calculations are deterministic which reduces overall computation times. This leads to a great potential for parallelization which could be used to reduce calculation time.
Even though the CPUs' performance are improving every year they are still not fast enough to trace millions of rays in a matter of seconds. The main problem is the lack of possible parallel threads. The solution for this lack of computing power is the use of a GPU. Even though each core is less powerful, the overall parallel computing ability is superb. There are several options to let programs run on a GPU. The programming language C++ is a very strong contender in order to maximize the performance of a CUDA enabled graphics card and will be therefore chosen for this thesis. CUDA is a parallel computing platform exclusively built for NVIDIA GPUs. It is usually accessible through C, C++ and Fortran but third party wrappers do also exist (e.g. Python, MATLAB among others).
Two Monte-Carlo ray tracers will be implemented on a given GPU. One will use standard data types given by C++ and the other will use specialized data types for the GPU. Furthermore, we will establish a ray tracer which runs on a CPU in order to compare its runtime with the GPU. The key aim of this work will be to achieve a significant performance improvement with the GPU in comparison to the CPU by reducing the necessary computation time. Additionally, an improvement should be seen with the

specialized data types versus the ”standard” data types on the GPU. At the end it has to be decided if the GPU ray tracer is fast enough and sufficiently accurate to substitute the established and widely used CPU ray tracers.

2 Solar Tower Power Plant

A solar tower power plant belongs to the group of concentrated solar power (CSP) plants. The radiation coming from the sun is concentrated on a receiver mounted on a tower. A receiver absorbs the energy of the sun rays and passes it to the heat transfer media. The mirrors which can be flat or curved are called heliostats and redirect the incoming rays from the sun to the receiver. Usually a CSP plant has thousands of heliostats which are positioned close to the tower to minimize tracking errors. Different heat transfer media (molten salt, water, air among other) can be used to absorb and transport the heat from the receiver to the turbine in order to generate electricity.
All heat transfer fluids are then used to first generate mechanical energy with the help of a steam turbine and at the end the energy is transformed into electricity by a generator. Afterwards the heat transfer fluid has to be cooled down in order to keep the efficiency levels of the turbine at a constant level. Dry cooling is sometimes the option of choice as CSP plants are usually located in dry areas and water availability is a limiting factor. Although it leads to a lower overall efficiency of the plant the environmental aspects of using less water in desert regions must be considered as well.
In the following Figure 1 you can see an exemplary scheme of solar tower power plant without a heat storage:

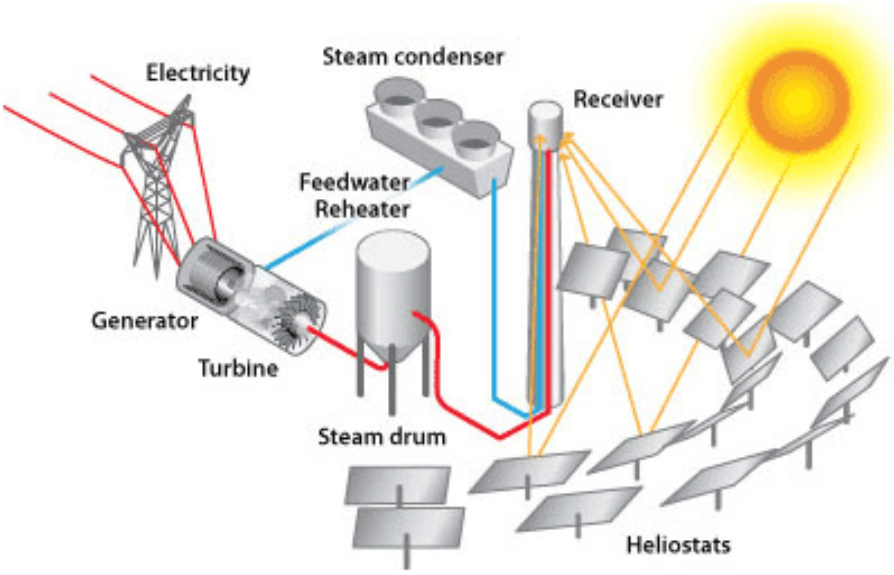

Figure 1: The reflected sun rays heat up the water which is then separated into steam and water in the steam drum. The superheated steam is then used to power the turbine and subsequently the generator.
https://climatekids.nasa.gov/concentrating-solar/

Heat storages are becoming increasingly important if the CSP plants are to replace coal-fired or nuclear power plants. Although the volatility of the weather is minimized by building in the desert, one still has to consider that electricity can only be produced during the day. This is a big problem because the demand for electricity during the

night is still significant. Molten salt tanks as heat storages partly solve this problem. They store surplus heat in order to generate electricity when it is needed or after sunset. In such CSP plants molten salt is usually used as a heat transfer fluid because it can be stored directly with very little losses. One disadvantage is the high minimum temperature which is required for molten salt to stay liquid. It is around 150°C ultimately depending on the exact mixture of salt being used. If it falls below that threshold the salt gets solid and can't be pumped anymore which potentially renders the CSP plant unusable.

Comparison to Conventional Plants

The first commercial solar tower power plant called "Planta Solar 10" (PS10) was built in 2006 close to Seville in southern Spain. It has a potential power output of 11 MW. In 2009 another plant called "Planta Solar 20" (PS20) was constructed on the same site with a electricity generating capacity of nearly 20 MW. Two years later Gemasolar, a tower plant near Seville with 19.9 MW, started operations and was the first one with molten salt as heat transfer fluid. Furthermore, it is able to store roughly 299 MWh which is equivalent to 15 hours of peak performance.
In October 2010 the construction of the biggest (as of now) solar tower power began at Ivanpah Dry Lake California. It finished in December 2013 and was fully operational in early 2014. The costs of building three identical plants seen in Figure 2 were approximately 2.2 Billion USD according to [2].

Figure 2: One of the three Ivanpah power plants is seen which are built next to each other. All three combined have 173.500 heliostats with 2 mirrors per heliostat. *https://dreambigfilm.com/stories/image-29-ivanpah/*

The net output of the plant is estimated to be 377 MW as stated by BrightSource Energy [4]. It also claims to produce enough electricity for more than 140.000 homes in California. For the purpose of understanding the actual size and power output of a solar tower power (e.g. Ivanpah power plant) a comparison with conventional plants in Germany will be made.
In Germany there are still 7 nuclear power plants active with an average net power output of 1.359 MW per power plant (see [5]). Furthermore, Germany still has a substantial amount of coal-fired power plants which have a electricity generating capacity of 44.915 MW all together. To substitute these conventional power plants, we would need at least 144 Ivanpah power plants. This calculation would only work if the solar tower power would generate electricity 24 hours a day with its maximum power output. In 2016 it had an average power output of 80 MW (according to [3]) compared to the peak performance of 377 MW. This leads us to an actual number of 678 Ivanpah power plants needed for substitution. We can say with high confidence, that these plants will not be built in Germany because sun radiation and space availability are not optimal. Obviously, the future does not lie in conventional power plants because the resources are either limited (e.g. coal) or just too dangerous to handle or store (e.g. uranium).

3 State of the Art Ray Tracing Tools

In this section I want to present existing ray tracing tools specifically implemented for CSP plants and highlight some of their features.

SolTrace

SolTrace was implemented in C++ and runs on a CPU. The development began in 2003 (see [6]), and it is available for download on the National Renewable Energy Laboratory web site[1]. According to [7] it uses the Monte-Carlo ray-tracing methodology. As user you can input the target geometry and determine how many rays should be traced. It is able to model parabolic collectors, linear Fresnel lens systems and solar tower power geometries. The output is given by scatter plots and flux maps which then can be saved or used for further analysis and processing. In [6] two example flux maps are created and compared with measured flux data. It does look similar, but they did not provide any error analysis or data. Thus, it is not obvious how precise SolTrace works and moreover no data concerning the time of computation is given.

SOLFAST

The second tool is called *SOLFAST* and also runs on a CPU. The paper describing *SOLFAST* was published in 2012 and thus far the program is not easily available or downloadable. The paper describes a new Monte-Carlo algorithm using an integral formulation with variance reduction techniques (see [8]). It states that it has a greater efficiency than other available programs (Soltrace and Tonatiuh). Furthermore, it compared its performance by simulating a small solar tower power plant (THEMIS around 1MW).

STERG

The third ray tracer also written in C++ was developed by *STERG* (see [9]) in 2013. It features a new approach which, additionally to the standard Monte-Carlo ray-tracing method, utilises statistical techniques to calculate the rays needed for an acceptable solution. Basically, it simulates a small number of rays to estimate how big the standard deviation is, and afterwards it calculates the optimum number of rays to be traced in order to reach the acceptable error (given by user). This method seems to be applicable to any ray tracer since it does only change how you utilise the ray tracing tool. The only downside is that if nearly exact solutions are needed (low tolerable error), this algorithm loses its utility (see [9]). In the paper they did test their new algorithm with two CSP plants (Eurodish and PS10) and compared the results with the publicly available ray tracing program *Tonatiuh*. The prediction did work satisfactorily given a low resolution and a significant error margin.

TieSOL

Tietronix Software Inc. located in Houston Texas developed the GPU-based Monte Carlo

[1] NREL `https://www.nrel.gov/csp/soltrace-download.html`

ray tracer *TieSOL* in 2012 [10]. Their requirements were to provide accurate shading and blocking percentages and to calculate the optical losses while returning a heat map of the incident flux on the receiver. Furthermore, they made use of different models for the sun shape, mirror surface irregularities and mirror tracking errors. The main goal was to reduce computation time. It is highly specialized for CSP plants and works with double precision. *TieSOL* consists of different programs, each serving a different purpose. The annual performance can be simulated as well as the heliostat field (graphically). All in all they achieved stunning performance-related results [10] without losing accuracy.

Nvidia RTX

In August 2018 NVIDIA announced new GeForce RTX gaming GPUs which are based on NVIDIA's Turing architecture[2]. RTX is a platform developed by NVIDIA which enables real-time ray tracing in games. It is included in several ray tracing APIs: NVIDIA OPTIX, Microsoft DirectX Raytracing and Vulkan. The new graphic cards have purpose-built ray tracing cores (RT cores) and their main task is to track rays through a scenery. They are able to track significantly more rays than older graphic cards[3]. Later in 2019 the RTX platform was brought to older GPUs (with a minimum of 6 GB memory) which enables real-time ray tracing for games. The RTX platform and the RT cores have the potential to be a promising topic for future research.

As a conclusion, there is a number of ray tracing tools using the CPU (except for *TieSOL*) for parallelization instead of the GPU which in theory should be faster. Technically it is a rather small step to transform a C++ program into a CUDA program since the language is partly the same with few exceptions. Moreover, the standard Monte-Carlo ray tracing technique is very convenient to parallelize since every ray has an independent trajectory and thus no information about other rays is needed.

[2]Nvidia RTX Platform `https://www.nvidia.com/en-us/geforce/news/nvidia-rtx-developer-support/`

[3]Accelerating The Real-Time Ray Tracing `https://www.nvidia.com/en-us/geforce/news/geforce-gtx-ray-tracing-coming-soon/?cjevent=381413aa9b1b11e983c0022e0a180511`

4 Model

The possible outcome of a solar tower power depends greatly on how efficient the shafts of sunlight can be gathered and transformed into electric energy. The following properties and crucial factors are important to optimize the output:

- **Mirror Reflectivity:** Since heliostats are no perfect mirrors, a certain percentage of rays are not reflected. Usually the reflection parameter is between 88% and 97%. This also depends on how often the heliostats are cleaned. The higher the reflectivity is, the better for the output.
- **Atmospheric Attenuation:** A small part of the radiation scatters in the atmosphere or is absorbed. On the one hand this effect depends on the turbidity factor of the atmosphere and on the other hand it is affected by the length of the path a ray has to "travel". It occurs when the ray of sunlight is reflected on the heliostat and on its way to the receiver.
- **Cosine Effect:** Since the rays of the sun don't always hit the heliostats perpendicular the actual received rays are significantly less depending on the angle. It is calculated by taking the cosine of the angle between incoming sun light and the heliostat normal. If the sun is vertically positioned the angle amounts to 90° which leads to a Cosine Effect of 100%.
- **Shading:** When the sun is low on the horizon some heliostats shade each other which prevents them from receiving sun light. This effect is also produced by the shadow of the tower where the receiver is situated. The bigger the distances between the heliostats the less shading one can observe.
- **Blocking:** When heliostats reflect parts of the incoming light onto the backside of neighbouring heliostats instead of the receiver, it is called blocking. This can also be avoided by increasing the distance between the mirrors. First this leads to higher tracking computation due to the higher gap to the receiver and second a larger ground is needed to host the same number of heliostats.
- **Spillage:** A fraction of the reflected light does not reach the receiver because of optical errors on the mirror surface or a shortcoming of tracking precision. Furthermore, acute angles (of the incoming sun rays) cause spillage because the reflected area of the heliostat is bigger than the receiver. For very acute angles the heliostat can not reflect the sun rays to the receiver and thus every ray misses.
- **Type of heliostat:** Heliostats can be flat or curved mirrors. Curved heliostats have a focal length which describes at what distance perpendicular rays are focused.

All these effects have a significant impact in a simulation of a solar tower power plant thus, they will be included in the GPU ray tracing program. Some of these factors are close to constant during the day (e.g. mirror reflectivity) and some vary a lot depending on the sun's position (e.g. cosine effect).

4.1 Simplified Model

The ray tracer programmed will be slightly different than others (see section 3) because the tracing does not start at the sun but at each heliostat. Usually rays are sent out from the sun following a random distribution. This method is inefficient because only a small fraction of the rays hit the heliostats. By creating the rays at the mirrors we omit the rays who missed the mirrors. Thus, in the beginning each heliostat is discretised depending on how many rays are to be traced. Afterwards the rays are created at their origin on the heliostat with each heliostat getting the same number of rays. First the light shafts are then traced to the sun and thereafter the reflection on the heliostat is calculated which takes place in the point of origin. Lastly the program has to calculate if the ray actually hits the receiver plane. This method ensures that less rays have to be traced in order to reach an acceptable accuracy.
Now it is time to establish the working ground for a simplified model. Since we only want to trace the sun light without any additional effects the following assumptions are necessary:

- The mirror will be **flat** in order to make the implementation easier
- There will be only **one** heliostat in order to avoid shading and blocking in the beginning
- No optical errors occur while reflecting rays
- Reflectivity μ_r of the mirror is set to 0.97 thus the ray loses 0.03 of his total energy. This number is specific to the mirrors of the given solar tower power plant.
- The sun will emit **parallel** rays whose direction can be calculated with φ and θ.
- At certain hours of the day the tower may shade a small part of the mirrors. Since this phenomenon only occurs in a short period of time it will be ignored.
- Normally a heliostat consists of several equally-sized facets (little mirrors) but we will simplify it by treating them as one big mirror/heliostat.

The last four simplifications will be kept throughout this work.

Calculations of the Simplified Model

The geometry of the heliostats and the receiver is given by four three-dimensional points, the vertices. First, they have to be transformed into the parametric form of plane in order to be able to use them properly. With $A_i, B_i, C_i, D_i \in \mathbb{R}^3$ and $i \in \mathbb{N}_0$:

$$P_i = A_i + r \cdot \underbrace{\overrightarrow{A_iB_i}}_{v_i} + s \cdot \underbrace{\overrightarrow{A_iD_i}}_{w_i} \text{ with } s, r \in \mathbb{R} \tag{4.1}$$

Furthermore, since the receiver and heliostats are rectangles:

$$v_i^\intercal w_i = 0 \tag{4.2}$$

If $i = 0$ we have the parametric form of the receiver and for $i > 0$ we have the parametric form of heliostat i. This parametric form will be transformed into a plane equation:

$$n_i^\intercal x = n_i^\intercal A_i \quad \text{with} \quad n_i = \frac{v_i \times w_i}{\|v_i \times w_i\|} \tag{4.3}$$

where $x \in \mathbb{R}^3$ and $\|\cdot\|$ is the standard euclidean norm. So only n_i (n_0) and p_i (p_0) are saved with each heliostat (receiver). We now have to calculate the reflection of the ray on the heliostat. The sun's direction called s can be derived from the sun's azimuth φ and altitude θ:

$$s = \begin{pmatrix} \cos(\theta)\sin(\varphi) \\ \cos(\theta)\cos(\varphi) \\ \sin(\theta) \end{pmatrix} \quad \text{and } \varphi \in [0, 2\pi), \theta \in [0, \pi] \tag{4.4}$$

With s calculated it is now possible to reflect the ray on the heliostat:

$$d = -s - 2 \cdot n_i n_i^\intercal (-s) \tag{4.5}$$

Given the new direction d we are able to calculate the intersection between ray and receiver plane. The intersection $\eta \in \mathbb{R}^2$ describes the position on the receiver plane by taking A_0 as new origin of the coordinate system. The idea is to treat the receiver plane as its own coordinate system. With the help of orthographic projection and (4.2) and $o \in \mathbb{R}^3$ being the origin of the ray we now follow:

$$\text{With} \quad z = o + t \cdot d - A_0 \quad \text{and} \quad t = \frac{n_0^\intercal (A_0 - o)}{n_0^\intercal d} \implies \eta = \begin{pmatrix} \frac{v_0^\intercal z}{\|v_0\|^2} & \frac{w_0^\intercal z}{\|w_0\|^2} \end{pmatrix}^\intercal \tag{4.6}$$

Lastly we have to check if $\eta \in [0, 1]^2$ because then the ray hit the receiver. η can also be used for plotting a heat map (see Figure 11 and 12) which shows where the rays landed on the receiver.

Atmospheric Attenuation

To correctly calculate the incoming power on the receiver the atmospheric attenuation μ_{aa} is of importance. It was calculated in this work by using the formula from [11]:

$$\mu_{aa} = \begin{cases} 0.99321 - 0.0001176x + 1.97 \cdot 10^{-8} x^2, & \text{if x} \leq 1000 \\ \exp(-0.0001106x), & \text{if x} > 1000 \end{cases} \tag{4.7}$$

Here $x \in \mathbb{R}$ describes the distance between a certain heliostat and the receiver. Thus, the atmospheric attenuation is different for every heliostat and has to be calculated in advance.

Cosine Effect

The incoming power DNI from the sun is measured in W/m^2. It can only be fully received if the shafts of sunlight hit the heliostat perpendicular. To calculate the cosine effect, we have to get the angle between s and n_i with the following scheme:

$$\mu_c = |n_i^\intercal s| \quad \text{because} \quad \|n_i\| = \|s\| = 1 \tag{4.8}$$

μ_c is the cosine effect which we have to multiply with the power value p_r of each ray.

Optical power

After we have calculated the cosine effect and the atmospheric attenuation, we are able to predict the incoming optical power of the receiver. Ever ray has a power value p_r which changes during the algorithm. In the beginning every ray has the initial power p of:

$$p = \frac{DNI \cdot H_{area}}{N \cdot N} \tag{4.9}$$

where H_{area} is the area of one heliostat assuming they all have the same size. Furthermore N characterizes the lattice of each mirror. If for example $N = 5$ is given, we divide the heliostat into a 5×5 lattice. Now that we have the initial power value p we have to consider the atmospheric attenuation μ_{aa}, cosine effect μ_c and the mirror reflectivity μ_r to calculate p_r:

$$p_r = p \cdot \mu_{aa} \cdot \mu_c \cdot \mu_r \tag{4.10}$$

This power value p_r is added to the total optical power if the ray hits the receiver.

4.2 Model Improvements

The following model improvements have to be included in the ray tracer in order to achieve accurate results. In contrast to section 4.1 we will include the shading and blocking effect and we will try to model tracking errors and mirror surface irregularities. At last the heliostats will be modelled as curved mirrors.

Shading and Blocking

As explained in the beginning of section 4 shading and blocking (see Figure 3) plays an important role in regards to optical losses. The goal is to calculate these losses with the GPU ray tracer. Every ray traced will be inspected if it was shaded or blocked by other heliostats. With increasing number of rays, the accuracy of shading and blocking increases as well. We do not want to compare the ray with every heliostat because it would be very expensive. Thus, we have to select a subset of the mirrors to reduce

computation costs.

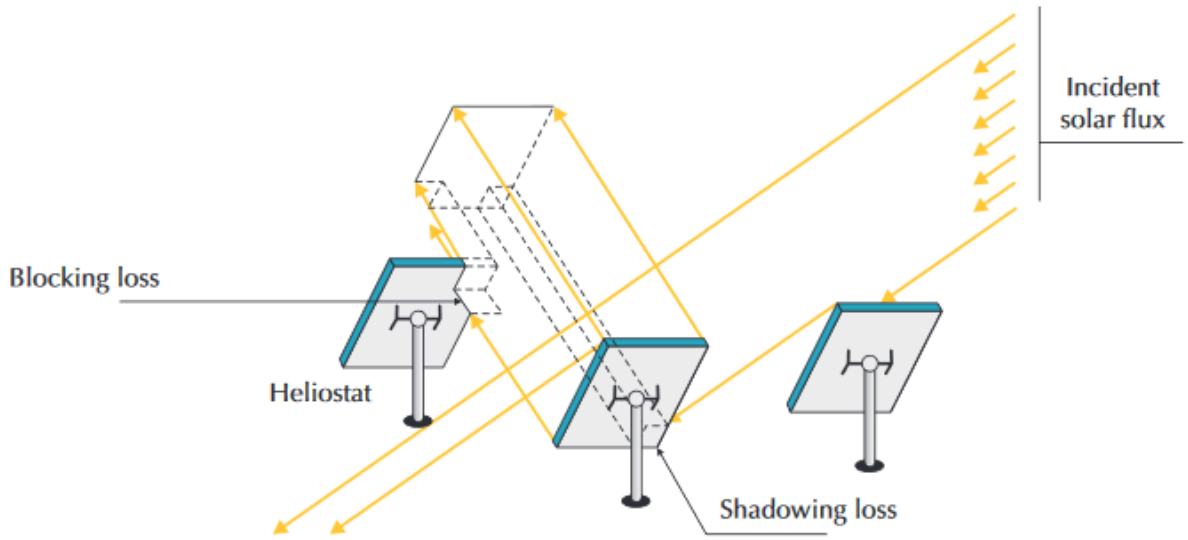

Figure 3: The heliostat in the middle is blocked by the left and shaded by the right one. The goal is to estimate these optical losses which are caused by the geometry of the heliostat field.
`https://www.iea.org/publications/freepublications/publication/Solar_Energy_Perspectives2011.pdf`

This subset will consist of a certain number of neighbours which are close to the heliostat of origin (of the ray). Additionally, they should stand in the ray's way to the sun. At midday this subset is minimal and in the afternoon/morning the subset is maximal. One possible solution to find the set of mirrors is to investigate the distances between them and choose the closest ones. Depending on the altitude angle we take more mirrors or less mirrors into our subset. This will be subsequently done for every heliostat and thus we reduce the number of calculations made significantly. Moreover, we could exclude the rays which have their origin at the centre of the mirror from the shading calculation since it is unlikely they are shaded. This depends again on the altitude angle of the sun. After discovering a ray is shaded all further calculations are aborted since we know it does not reach the receiver.
The blocking effect will be treated similarly by choosing a subset of neighbours which may stand in the ray's way to the receiver. Preselection will be done again with their distances to other heliostats.
The calculation of the distances between all heliostats will be done by calculating the length of the vector which goes from one centre point to the other. Then we choose the m closest mirrors with m being smaller or bigger depending on the altitude angle θ. At the end we save this subset to every mirror which can be accessed later by the device's main program.

Surface Errors

In section 4.1 we treated the mirrors as perfect mirrors and excluded any mirror surface irregularities or tracking errors. To simulate these effects we will need random numbers from a Gaussian distribution where the mean is always 0 and the standard deviation is given. Afterwards we will create an error cone around the previous direction and "select" a new direction from this cone. This is done by fist transforming the vector d into spherical coordinates (r, α, β):

$$\begin{aligned} r &= \|d\| \\ \alpha &= \arccos(\frac{d_3}{\|d\|}) \\ \beta &= \begin{cases} \arctan(\frac{d_2}{d_1}), & \text{if x} > 0 \\ \operatorname{sgn}(d_2)\frac{\pi}{2}, & \text{if x} = 0 \\ \arctan(\frac{d_2}{d_1}) + \pi, & \text{if x} < 0 \text{ and y} \geq 0 \\ \arctan(\frac{d_2}{d_1}) - \pi, & \text{if x} < 0 \text{ and y} < 0 \end{cases} \end{aligned} \tag{4.11}$$

Now that we have our spherical coordinates, we are able to create a small perturbation:

$$\begin{aligned} &\text{With} \quad X_1, X_2 \sim \mathcal{N}(0, \sigma^2) : \\ &d = \begin{pmatrix} r \cdot \sin(\alpha + X_1)\cos(\beta + X_2) \\ r \cdot \sin(\alpha + X_1)\sin(\beta + X_2) \\ r \cdot \cos(\alpha + X_1) \end{pmatrix} \end{aligned} \tag{4.12}$$

This small back and forth transformation happens directly after (4.5) and the calculations just continue with the "new" d.

Curved Mirrors

In (4.5) we calculated the reflection with the heliostat normal n_i as we approximated the mirror with a flat surface. The truth is that the surface is usually slightly curved and is characterized by the focal length of each heliostat. This enables a higher percentage of rays hitting the receiver and thus a higher optical power. After slicing the mirror one can describe the form as a parabola. We assume that heliostat i is curved alongside w_i with f_{cl} being the focal length and thus

$$\begin{aligned} &g(x) = a(x-b)^2 + c \quad \text{with:} \\ &a = \frac{1}{4f_{cl}}, \quad b = \frac{\|w_i\|}{2}, \quad c = -\frac{\|w_i\|^2}{16 f_{cl}} \end{aligned} \tag{4.13}$$

$g(x)$ describes the form of the sliced mirror.
In the following Figure 4 the sliced plane is pictured with axis $x \cdot w_i$ and $y \cdot n_i$. Hereby

it should not be forgotten that n_i is normalized and w_i isn't. Thus from $(0,0)$ to $(1,0)$ we exactly go $1 \cdot w_i$ which is the segment from A_i to D_i. The third axis v_i is the depth of the figure which is not shown because it does not change the calculations.

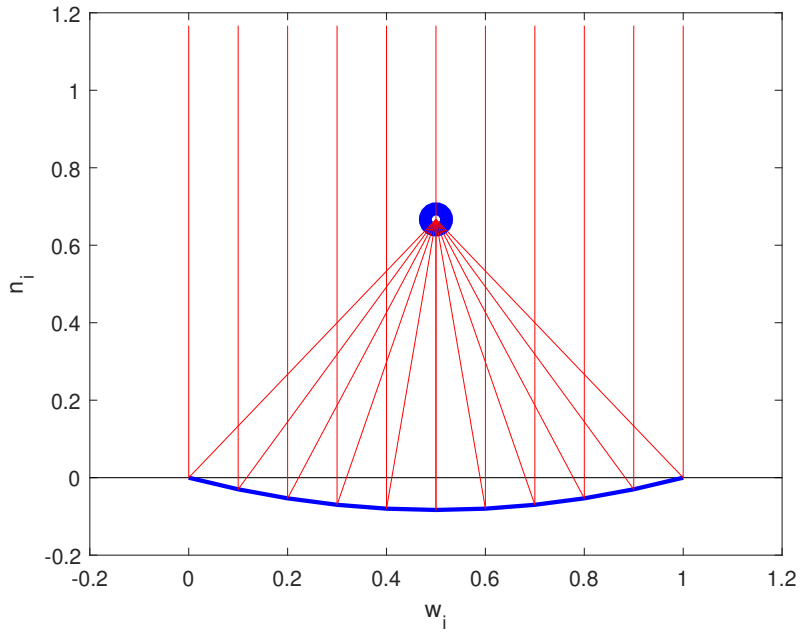

Figure 4: Exemplary curved mirror in blue and mirror plane in black. Perpendicular incoming rays are reflected towards the blue focal point. In this case the focal length is 0.75.

Every point on the heliostat can be described with w_i and v_i if we subtract A_i from it. The reason why this can be done, is (4.1) with P_i being the parametric form. But only the "amount" x_0 of w_i is important because the heliostat is curved alongside w_i. From the function $g(x)$ we can derive the normal n_r of every ray by calculating the following:

$$\begin{aligned} &\text{With} \quad g(x)' = 2a(x-b): \\ &n_r = \frac{w_i}{\|w_i\|} - \frac{1}{g(x_0)'} \cdot n_i = \frac{w_i}{\|w_i\|} - \frac{2f_{cl}}{x_0 - \frac{\|w_i\|}{2}} \cdot n_i \end{aligned} \tag{4.14}$$

Afterwards we continue with equation 4.5 but instead of n_i we use the specific normal of each ray n_r.

5 Ray Tracer

5.1 Goals

The first goal of this thesis will be the implementation of a simplified (with the model of section 4.1) ray tracer. This is necessary so that the program can run on the GPU without crashes. Afterwards we will do the following:

1. Test performance and accuracy so we already get an idea of what may happen in a more complex and detailed ray tracer. After successful testing the program can be reused.

2. Write a small program which takes a .csv file and returns the geometry of the particular solar tower power plant. The .csv file is called shapefile. It also includes the sun's azimuth φ, its altitude θ and the direct normal irradiance DNI.

3. Add important model improvements (see section 4.2).

4. Two additional ray tracers are added (three in total):
   - GPU_1 is the first (standard) GPU ray tracer which uses standard C++ variables.
   - GPU_2 is the second GPU ray tracer which uses CUDA specific variables (e.g. `float3`). They are vectors up to a length of four which are implemented as a struct. This is the **only** difference to GPU_1
   - CPU_1 has the exact **same** code as GPU_1 but it runs in parallel with the template library `OpenMP`[4] on the CPU.

5. Test performance and accuracy of the three ray tracers (see section 6.1 and 6.2).

The idea to implement GPU_2 came when an example[5] showed up which indicated better calculation times with these specific structs.

Since the ray tracer should run on a GPU, the program has to support as much parallel processing as possible. The strength of the GPU lies in its higher number (than CPU) of cores and its parallelization potential. Moreover, the theoretical operations per second of the GPU (green curves in Figure 5) is higher than on the CPU (blue curves in Figure 5). In order to exploit this advantage every ray will be traced independently with as little memory used as possible. At the end we want to trace millions of rays and because of the very limited storage capacity of a GPU every ray has to be "light" in terms of memory usage. For a start only single precision will be used instead of double precision. A single precision number (also called `float`) needs 32 bits (=4 bytes) as storage where double precision (also called `double`) needs 64 bits (=8 bytes). On the one hand `float` has the

[4] OpenMP `https://www.openmp.org/`

[5] Efficiency of CUDA vector types `https://stackoverflow.com/questions/26676806/efficiency-of-cuda-vector-types-float2-float3-float4`

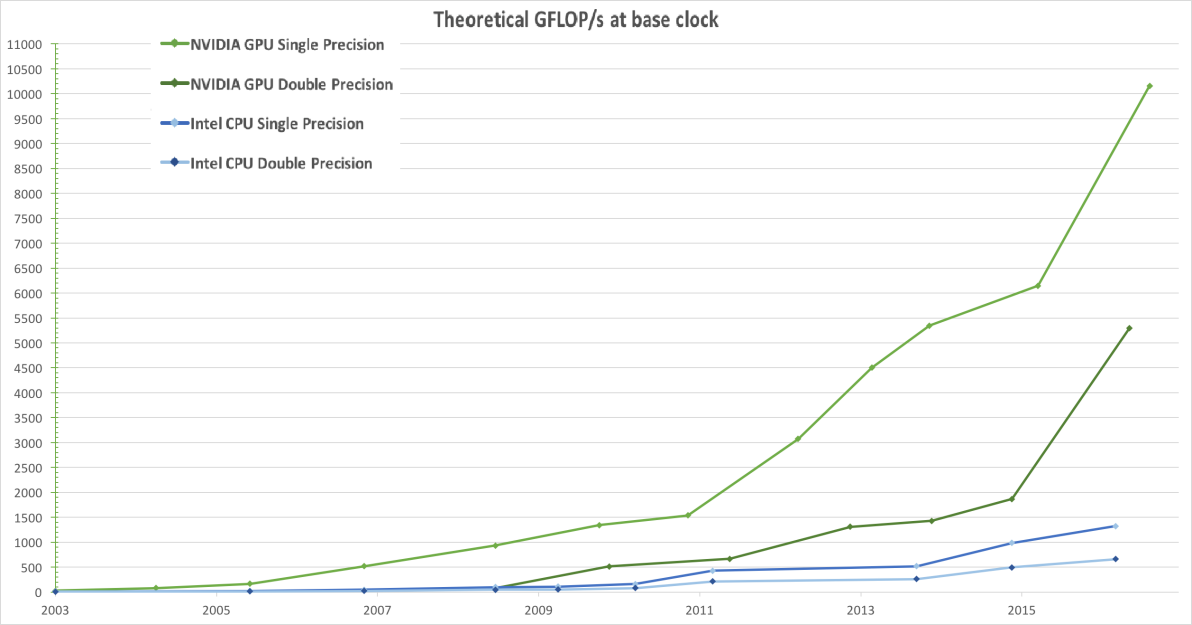

Figure 5: The theoretical speed in 10^9 floating point operations per second (GFLOP/s) from 2003 to 2016 for NVIDIA GPUs and Intel CPUs. Single precision is much faster than double precision. Green curve stands for GPU and blue curve stands for CPU.
`https://docs.nvidia.com/cuda/cuda-c-programming-guide/gr-phics/floating-point-operations-per-second.png`

advantage of being faster (see Figure 5) but on the other hand it has significantly less accuracy than `double`. This can be partly improved by selecting the most appropriate algorithms[6]. It may have a positive impact on accuracy if the best possible algorithm is chosen.
Once the simplified ray tracer has been implemented, the first model improvement which is "shading and blocking" has to be added. The challenge is how to write a function which computes if the specific ray of the current core is shaded or blocked. The difficulty lies in making the computation efficient because it is unnecessary to calculate intersections with every heliostat. The idea is to preselect those which shade or block the ray. For shading a small group of the closest neighbouring heliostats can be selected since they are the only ones who possible shade the ray. The same method will be applied to the blocking procedure.
To simulate mirror irregularities and tracking errors we have to generate random numbers which interfere with our calculated direction d of the rays. The clue is to let the device generate the random numbers in parallel instead of generating them beforehand on the host. A Gaussian distribution is used where the user enters the variance with the mean always being zero. There are different generators which can be chosen: pseudo random

[6]Dot product `https://docs.nvidia.com/cuda/floating-point/index.html#dot-product-accuracy-example`

or quasi random. Pseudo random numbers are created by a deterministic algorithm and the sequence of numbers is determined by setting the seed and offset. The same seed always produces the same sequence of random numbers. A quasi random number generator produces a low-discrepancy sequence which often is an advantage in Monte-Carlo methods. They are neither random nor pseudo random numbers. See [12] for further explanations on the generator types and their properties.
Lastly the modelling of the heliostats needs to be improved because usually they are slightly curved. This leads to less rays missing the receiver and thus a bigger optical power. The curved mirrors are characterized by the focal length which indicates at what distance the sun light is focused. This is a significant change because beforehand the heliostat was treated as a flat mirror and now we have a slightly curved mirror which can concentrate the sun light on a certain point. Furthermore, the focal length varies in a heliostat field because depending on their distance to the receiver the focus point has to be adjusted.

5.2 Basic Algorithm

Algorithm 1 Basic Algorithm

$ID \leftarrow$ unique ID in $[0, maxThreads]$
$N_{\text{total}} \leftarrow$ rays to be traced (input by user)
$ListRays \leftarrow$ list of all rays to be traced
for $ID < N_{total}$ **do**
  $Ray \leftarrow ListRays[ID]$ get Ray we are actually dealing with
  Trace Ray to the sun
  Calculate reflection of Ray on heliostat
  Trace Ray to the receiver
  if $Ray \in receiver$ **then**
    save intersection with receiver
  end if
  $ID \leftarrow ID + maxThreads$
end for

Now the basic algorithm is presented which is the part of the program which runs on the GPU and thus the only part where we can improve the computation time in comparison to the CPU program. Algorithm 1 works no matter how many threads on the GPU are called. It is the foundation of all following and improved ray tracers on the GPU. The user is able to control how many threads/blocks are opened and how many rays N_{total} we want to trace. In the beginning a personal and unique thread ID is created for every opened thread. They are essential because via the ID the work is distributed and due to their uniqueness, every ray is only calculated once. This particular thread calculates various rays in its lifetime until its ID is above N_{total}. Thus a `for` loop is needed with the constraint $ID < N_{total}$. The thread takes element ID of $ListRays$ and first calculates its trajectory to the sun. Afterwards we can derive the ray's new direction

by computing the reflection on the heliostat with the information where the ray comes from (sun). Lastly, we have to trace the ray to the receiver and check if it actually hits the receiver. After the thread finishes with one ray his ID is raised by the maximum number of threads. In the end the thread goes to the beginning of the `for` loop and starts the same process again with a different ray. The explanations of the calculations made in Algorithm 1 can be found in section 4.

5.3 Implementation

Every CUDA enabled GPU has a certain number of streaming multiprocessors (SM), a maximum number of threads per multiprocessor and a maximum number of threads per block. For example, a NVIDIA GTX 1050ti has 6 SMs with a maximum of 2048 threads each (1024 per block). Now the question arises how the user addresses these threads. First, he gives a number of blocks (can be any positive number) and second he inserts the number of threads per block (which has to be divisible by 32). Then the GPU distributes these blocks onto the SMs. For example, with the execution configuration $<<< 12, 1024 >>>$ (blocks, threads) each SM gets 2 blocks with 1024 threads. The number of threads has to be divisible by 32 because the GPU always executes in group of 32 threads also called warps. If it is not divisible by 32 the GPU has some inactive threads in the last warp which is inefficient.
The implementation of this program was done in Visual Studio 2017 with the CUDA Toolkit 10.1 (see [13]) installed. In further explanations the CPU will be called `host` and the GPU will be called `device`. Usually the code in C++ is executed on the host but a few small changes and tweaks enable the code to run on the device. The following basic steps are essential in order to be successful:

- Add the specifier `__global__` to the function you want to call on the device. This signals the CUDA C++ compiler that the function is callable from the host and executed on the device. The `__global__` functions are also called (GPU) kernels.

- The variables/memory initialized and used by the host have to be made accessible for the device. This is done by using a managed memory space (also called Unified Memory) where all processes see a coherent image of the memory. Basically, this memory space is accessible by the host and by the device. Alternatively one could copy the host variables to the device by first allocating the memory and then transferring the data into the allocated space. This has two significant disadvantages with the first being that the code is far more cramped and far less readable. The second one is linked to the memory access times which have been optimized in the Unified Memory space. Nevertheless, this does not necessarily lead to less runtime.

- The `__global__ function(input)` is called by the device with `function` $\lll$`a,b`$\ggg$`(input)` where a is the number of blocks and b is the number of threads per block.

- CUDA files have the file extension `.cu`.
- Lastly the host has to wait until the device is done with the computation thus the command `cudaDeviceSynchronize()` is necessary after the device is called.

Additionally, to the CUDA file, where the main program resides, two specific header files are written. Usually header files contain structs, classes and functions which are used by the main program. A struct `Ray` was created with as few member variables as possible because the memory occupied by one ray is later multiplied by a million (or more). Furthermore a class `Heliostat` was implemented which hosts some functions which are important during the initialization. At last a significant number of functions, which simplify the main GPU program, are included in the header file. It has to be pointed out that Microsoft Visual Studio 2017 always creates a debug version of the program which is significantly slower than the release version. This debug version though is needed for debugging and improving the code. Later, when the program is finished, all the performance tests will be made with the release version.

5.4 Sort Algorithm

In section 4.2 ("Shading and Blocking") we have to find the closest m neighbours of a specific heliostat. Because the number of heliostats is not known beforehand the sorting algorithm will be implemented on the device to accelerate the process. However very few sorting algorithms do exist for CUDA and they are not very customizable. The goal is then to implement a highly customized sorting algorithm which is faster than the given functions by the template library `Thrust`[7]. There are two main reasons why this is necessary:

- The standard sort algorithm sorts the entire array which is not needed because we only need the closest m mirrors. The order of the other neighbours is just not important.
- When an array is sorted, the indices in the beginning get lost during the process. It could be avoided by defining a second array where the indices are stored. However it occupies additional memory space which is naturally scarce on the device.

The following Algorithm 2 shows how the array of distances is sorted while keeping the indices in order to identify the closest neighbours:

The idea is to look for the shortest distance and save the index of the element to the list of neighbours. Afterwards we set this minimal distance to -1 so it is not found again in the next loop. The less closest neighbours we want, the better the speedup compared to the template library `Thrust`.
After the list of neighbours is filled with heliostats which possibly shade or block the given ray, we have to calculate the intersection between the ray and mirrors. This is showcased in (4.6). Additionally, t (of (4.6)) has to be positive because the intersections

[7] Thrust `https://thrust.github.io/`

Algorithm 2 Sort Algorithm

$neighbours \leftarrow$ empty list of neighbours for given heliostat
$buffer \leftarrow$ empty variable
$m \leftarrow$ number of wanted neighbours
$distance \leftarrow$ list with calculated distances to other heliostats
$i, j \leftarrow 0$ count variables
for $i < m$ **do**
  $buffer \leftarrow \infty$ Set $buffer$ to a high number so it is always bigger than the first distance $distance[0]$
  for $j < length(distance)$ **do**
    if $buffer > distance[j] > 0$ **then**
      $buffer \leftarrow distance[j]$
      $neighbour[i] \leftarrow j$
    end if
    $j \leftarrow j + 1$
  end for
  Set shortest found distance $distance[neighbour[i]]$ to -1 so it is not found again
  $i \leftarrow i + 1$
end for

cannot lie behind the origin of the ray. They have to be in the positive direction of d. If $t < 0$ is accepted, a ray would be shaded or blocked by a heliostat which is behind his mirror of origin and this is in reality not possible.

6 Case Study

We will do the case study with the "Planta Solar 10" in Seville. It has 624 heliostats and a receiver which is located on a 115 m high tower. The geometry given by the two shapefiles is identical but the focal lengths corresponding to each heliostat are different. On the one hand we have realistic focal lengths which differ depending on the distance to the receiver and on the other hand we have a constant and rather big focal length. Basically, we have one file with curved heliostats and one file with flat heliostats. Moreover, results for both shapefiles were given which then can be compared to the results of this work.
The goal of this section is to verify the accuracy of the results and to compare the speed of the three ray tracers (see section 5.1). Furthermore, a big gap in computation time between CPU_1 and GPU_1/GPU_2 is to be expected. Also a small gap between GPU_1 and GPU_2 will emerge because the CUDA specific variables are significantly faster.

6.1 Accuracy

First we want to analyse how accurate the programmed ray tracers work in comparison to the given data. GPU_2 was chosen to yield the results because all three ray tracers are identical concerning the calculations. The given optical power was calculated without taking into consideration the tower shading, atmospheric attenuation μ_{aa} and reflectivity μ_r of the mirror. Furthermore we set σ from section 4.2 ("Surface Errors") to 0.002 which is equal to 2 mrad since we perturb spherical coordinates. The following table shows the given data compared to calculated data in the case of flat heliostats:

Flat Heliostats	Given Results	Calculated Results	Deviation
Mean cosine	0.887413	0.888675	0.142211 %
Mean atmospheric attenuation	0.949964	0.950753	0.083056 %
Mean blocking efficiency	0.996478	0.996508	0.00301 %
Mean shading efficiency	0.946575	0.946469	0.011198 %
Optical power	$3.23452 \cdot 10^7$	$3.05607 \cdot 10^7$	5.517047 %

As the table shows, all deviation percentages are pretty good except for the optical power which deviates slightly more than 5 %. The deviations may come from differences in the model of the ray tracer. It could also be possible that the variable type `float` is not sufficiently accurate.
These results were produced by tracing more than five million rays to ensure accurate findings. The mean cosine effect only includes the effects of rays which hit the receiver. Since rays who do not reach the receiver have the power value $p_r = 0$ the effect isn't calculated. Next, we want to look at the following table which shows the given results compared to the calculated results in the case of curved heliostats:

Curved Heliostats	Given Results	Calculated Results	Deviation
Mean cosine	0.890849	0.888022	0.317338 %
Mean atmospheric attenuation	0.949722	0.950753	0.108558 %
Mean blocking efficiency	0.99726	0.99696	0.030082 %
Mean shading efficiency	0.946575	0.946469	0.011198 %
Optical power	$3.83738 \cdot 10^7$	$3.77956 \cdot 10^7$	1.506757 %

The realistic case with curved mirrors results in a higher and closer optical power which now only deviates 1.5 %. A total increase of 18-24 % of the optical power is significant and thus justifies the additional costs of curved mirrors. All the other parameters are again very close to the given results. The atmospheric attenuation is equal to beforehand because the calculations only include the distance between mirror and receiver and thus do not change if the mirror surface changes. Moreover the tool, where the results were produced, has a different algorithm and implementation than this GPU ray tracer. So even though no information about the implementation of the tool was given, the results only deviate slightly.

An important question is whether the GPU ray tracer produces accurate results with far less rays and thus far less computation time. In Figure 6 the trend can be seen for the shading and blocking effect in the case of flat and curved heliostats. The grid sizes for the heliostats range from 1×1 to 25×25. In the beginning when only one ray is traced per heliostat (Grid 1×1) the effects differ a lot from the solution. Nevertheless, after tracing only 100 rays (Grid 10×10) per heliostat the results are very accurate and don't change a lot if more rays are traced. This would imply a total number of 62400 rays for a reasonable accuracy. It is important to mention that the shading effect does not differ between curved and flat heliostat in Figure 6. The "Flat" curve is on top of the "Curved" curve thus the color is the same. It is only logical because the shading effect occurs from the sun to the mirror and it does not matter if the mirror is flat or curved. The vertices of the shapefiles are the same for curved and flat. In contrast to that the blocking effect does vary because the rays are reflected differently on curved or flat heliostats. With a new direction the ray may be blocked more or less likely.

The positive findings in Figure 6 are due to a sufficient count of neighbours (see section 4.2) being stored to every heliostat. The question arises how this optimum number is found. In Figure 7 the development of the shading and blocking effect is shown with an increasing count of saved neighbours. We always traced 1.56 million rays in order to not falsify the results. After increasing the number of neighbours to six the effects stay constant even though more neighbours are added to each heliostat. More neighbours in the list implies a bigger computation because more intersection between ray and neighbours have to be calculated. This optimal number is unique because of the geometry of the heliostat field and may change if a different CSP plant is simulated.

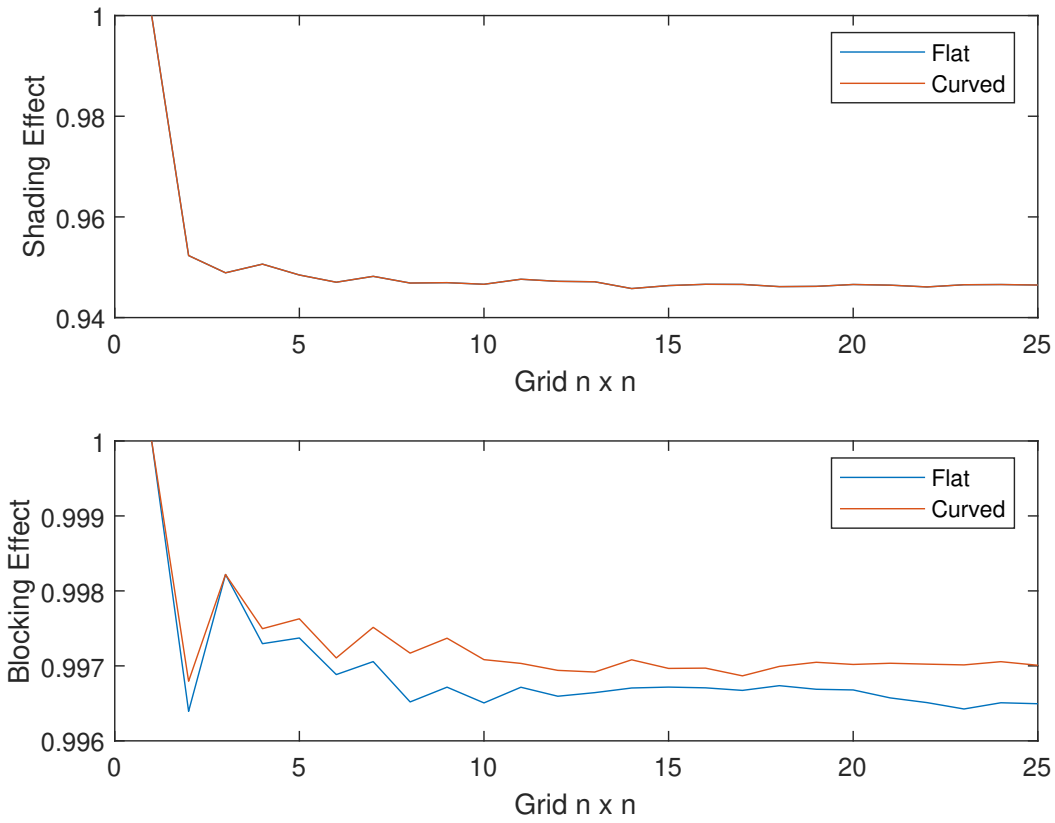

Figure 6: Shading and blocking effect for different grid sizes with flat and curved heliostats

Nevertheless for the purpose of securing reasonable results the number of neighbours is raised to ten in all the simulations.

Lastly we want to turn our attention to the Figure 8 where the cosine effect of both cases is shown. There is a consistent small gap between flat and curved with the curved mirrors having less cosine effect. Again we can see that after a certain point the lattice does not need more refinement because the results are close to constant. This behaviour reinforces the theory of just needing approximately 100 rays per heliostat to get accurate results.

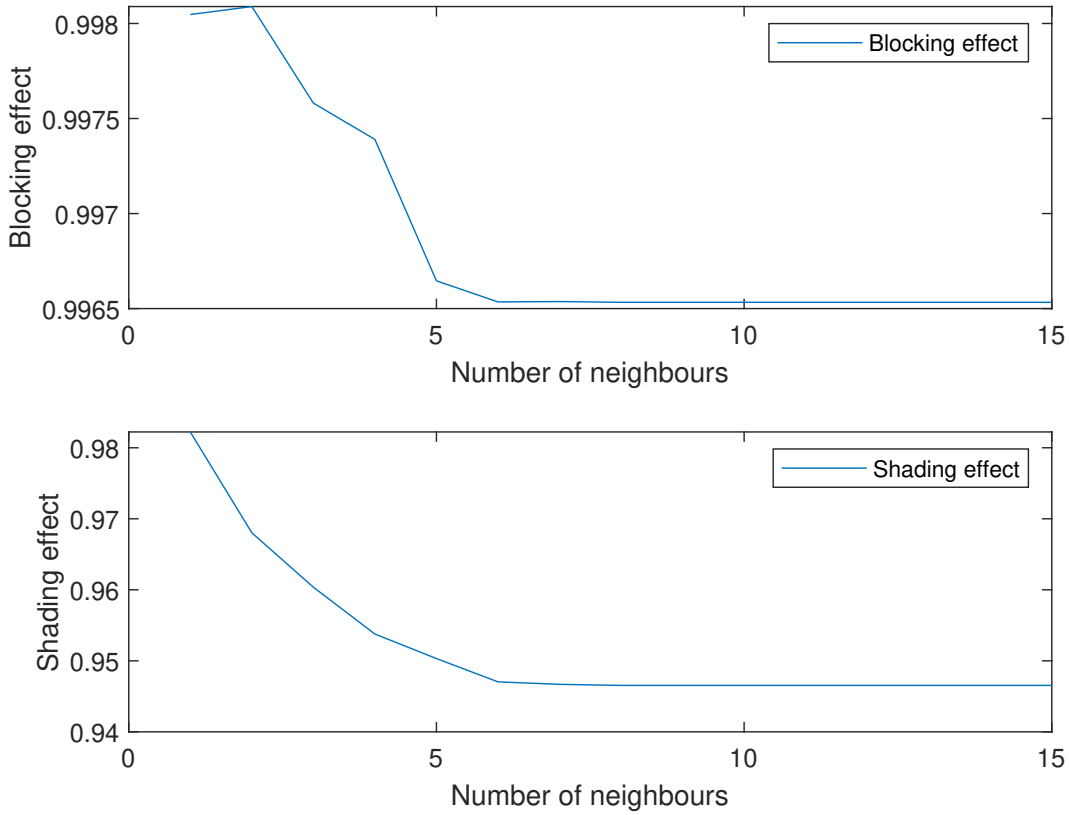

Figure 7: Blocking and shading effect for different number of neighbours. Done with tracing 1.56 million rays

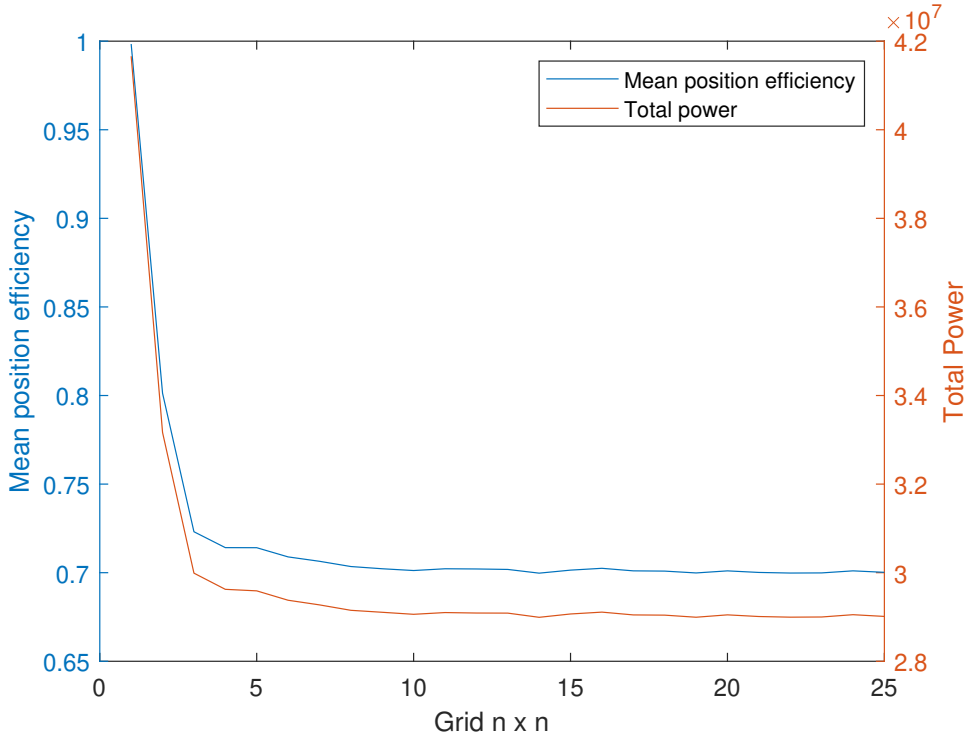

Figure 9: Mean position efficiency and total power for flat mirrors

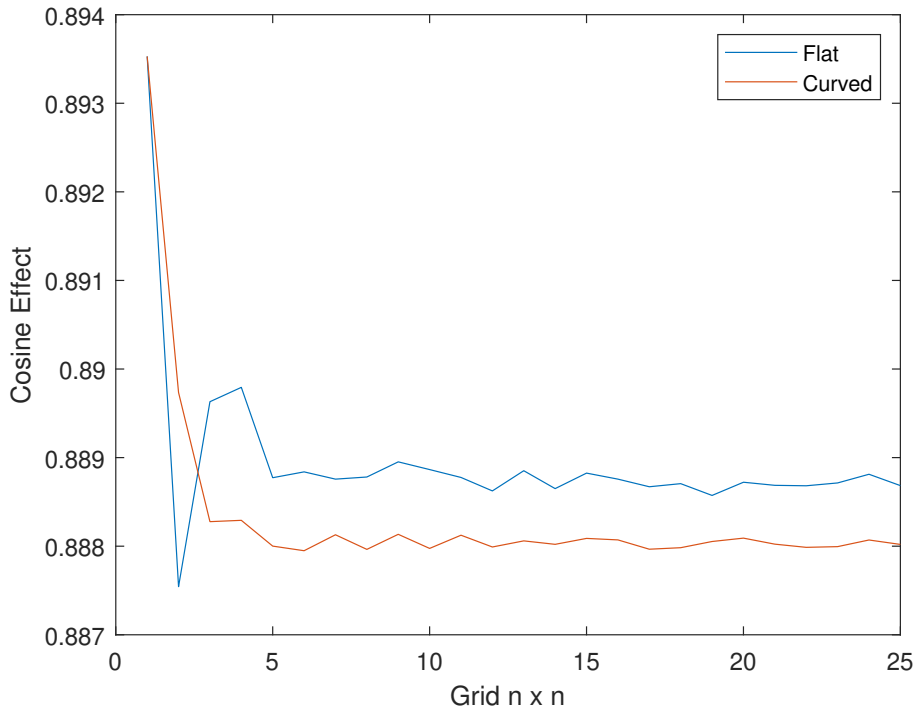

Figure 8: Cosine effect for different grid sizes with flat and curved heliostats

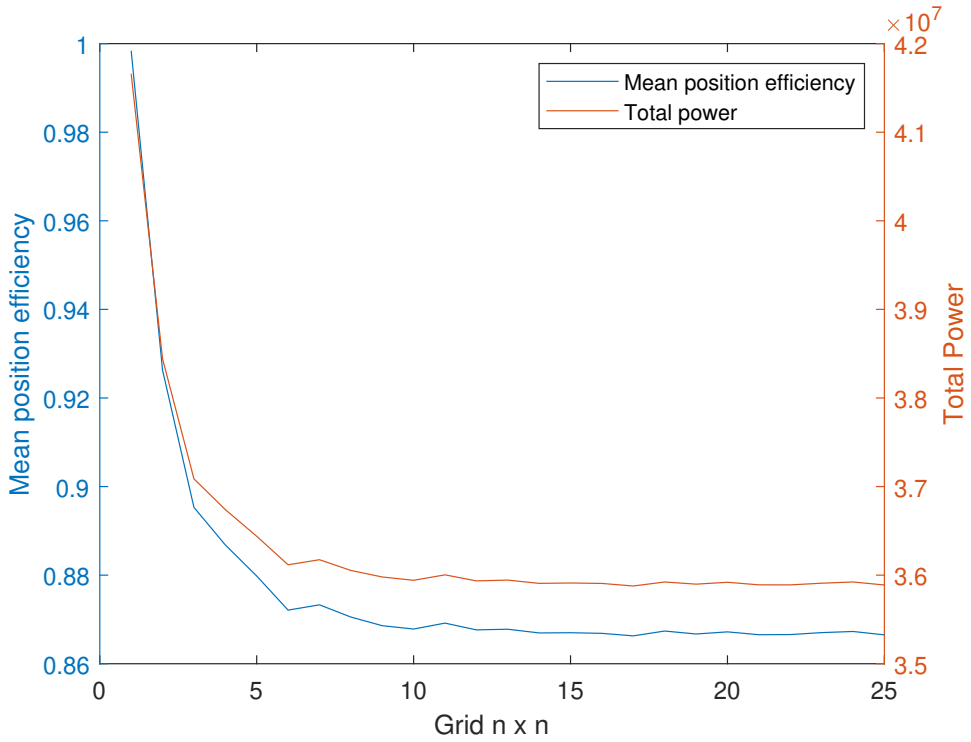

Figure 10: Mean position efficiency and total power for curved mirrors

In Figure 9 the mean position efficiency and the total power is shown for different grid sizes. The following effects are included in the mean position efficiency: spillage, surface errors, shading and blocking effect. It gives us a mean percentage about how many rays reach the receiver. We just have to multiply this efficiency with the average energy level of the rays and we get a good estimation for the optical power. Furthermore we have an indicator for the qualitative position of the mirrors. Not only the position but also the shape and quality of the heliostats is assessed. In the beginning the mean position efficiency sharply drops until it reaches a lower limit and afterwards it just changes slightly even though we increase the number of rays traced to almost 400000. The same goes for the total power which in the beginning is at its maximum because every ray traced per heliostat (one) hits the receiver. Therefore we derive that all heliostats are correctly aligned because the center ray always reaches the receiver. Afterwards the total power fluctuates very slightly with increasing grid sizes. This behaviour continues for increasing number of rays but it never leaves the area of $2.9 \cdot 10^7$. It may be strange that it does not converge to a certain value but it is evident why this happens. With further grid refinement and more rays we have more normally distributed numbers which perturb the rays and thus we always have a bit of "randomness" in the results.

Figure 10 is similar to Figure 9 in many ways. Initially we have sharp drop and after a few more grid refinements the curve stabilises and stays at a certain level. These levels however are different to those in Figure 9 because the mean position efficiency and the total power are significantly higher. Observing this spread justifies the additional costs of curved mirrors instead of flat mirrors.
At the end of this section Figure 11 and 12 represent the heat maps of the respective type of heliostat. On the right hand side of the figures is the map legend which describes how many rays hit a certain quadrant on the receiver. The length of one x is the width of the receiver and the length of one y is height of the receiver. In addition to that the red rectangle represents the receiver. Everything which is outside this rectangle missed the receiver.
Observing Figure 11 one can notice that the rays are relatively evenly distributed. On the top edge of the receiver we have a lot of rays who are missing the receiver. On the other hand the bottom edge is rather empty with just few rays reaching the receiver.
If we compare Figure 12 with Figure 11 a big difference in the distribution of the rays is seen. In Figure 12 we have much more concentrated points on the receiver. This is reinforced by the color of the points where yellow stands for $10 \cdot 10^4$ rays and light blue for $4 \cdot 10^4$ rays. The rays in Figure 12 are far less evenly distributed and most of the rays reach the top half of the receiver. Furthermore the maximum distance (deviation) from the red rectangle is less than in the heat map of the flat heliostats. Thus we can safely say that the curved mirrors are better suited to reflect the incoming sun light than the flat mirrors.

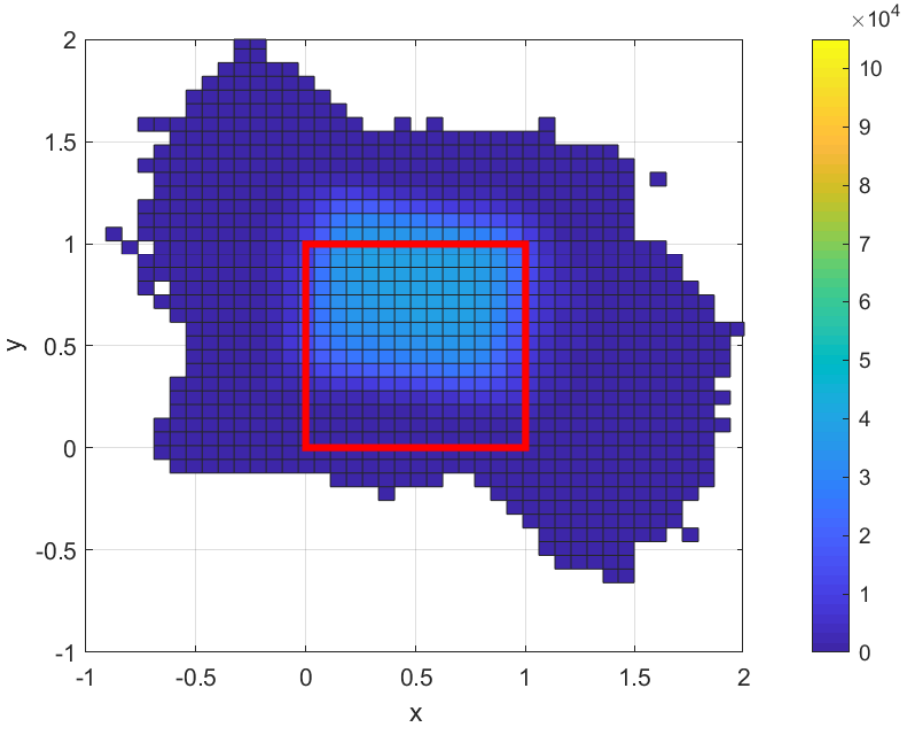

Figure 11: Heat map in the case of flat heliostats

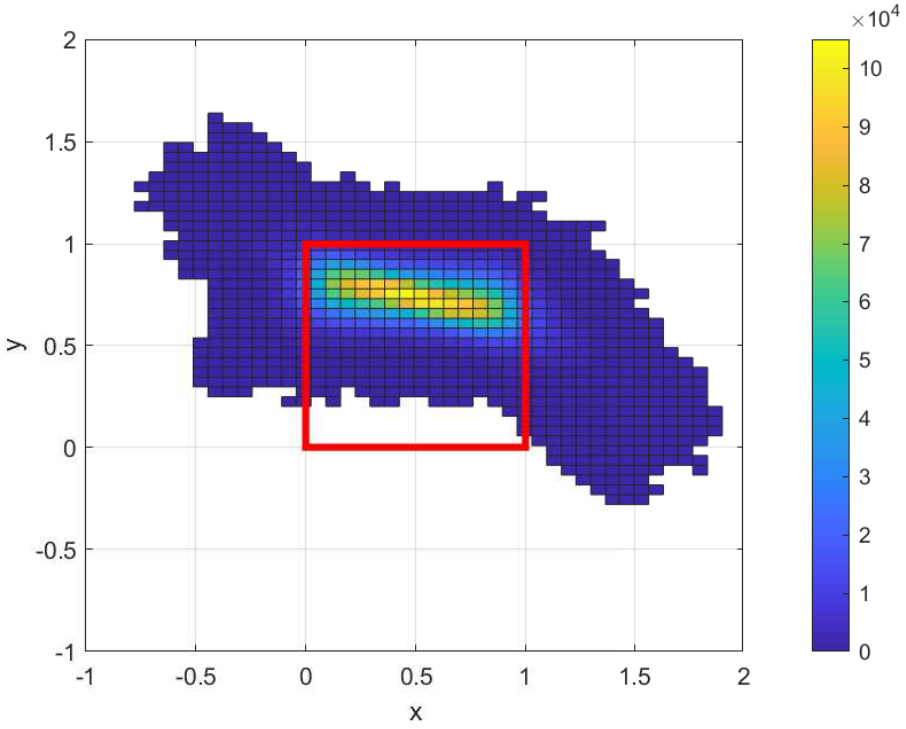

Figure 12: Heat map in the case of curved heliostats

6.2 Speed

We want to measure the speed-up of the GPU program and thus its ability of becoming faster if more CUDA threads are used. First the potential theoretical speed-up is estimated and afterwards we compare it to the results of the GPU. This can be done by applying Amdahl's and Gustafson's law (see [14]) which state:

$$\text{Amdahl: } S(U)_A = \frac{1}{1 - f + \frac{f}{U}}, \quad \text{Gustafson: } S(U)_G = (1 - f) + U \cdot f \tag{6.1}$$

where S is the total possible speed-up with U being the processing units which can work in parallel. f is the fraction of the task which is able to be executed in parallel. Usually these laws are applied to programs which run on a CPU but it is possible and has been widely discussed to apply them to CUDA software. Both theories rely heavily on a big fraction f to even get a good speed-up. f is calculated by letting the program run with one core available and afterwards we measure how big the sequential and parallel parts are. Since we implemented the ray tracer on a GPU we have very slow cores in comparison to the CPU. This leads us to a fraction f which is very close to 1 and thus we expect very good speed-up with more cores and threads in both functions.
One condition for Amdahl's law is that the problem size has to be kept constant while increasing the number of processing units. This is called strong scaling. Furthermore one can derive an upper limit to the possible speed-up by observing the behaviour of $S(U)_A$ for $U \to \infty$. In our case we have an initial f of 0.9994. Thus the upper limit of the speed-up is:

$$\lim_{U \to \infty} S(U)_A = \frac{1}{1 - f} = \frac{5000}{3} \approx 1666.7 \tag{6.2}$$

In Figure 13 the plot shows that Amdahl's law tries to reach the upper limit while significantly increasing U. In the beginning the speed-up increases rapidly but later on we have to add a lot of processing units to still observe a significant speed-up. The blue curve in the plot shows how the speed-up for the ray tracer GPU_2 evolves. It does follow Amdahl's law but slightly starts to fall of after $\sim$ 1200 threads. All in all GPU_2 is a highly parallelizable code which greatly benefits from high number of threads.

Usually we do not have a fixed problem size but rather an increasing one. Therefore we have to analyse how the program copes with an increasing problem size and number of threads. This is called weak scaling. The idea behind Gustafson's law is that large problems are solved with more computation power than smaller problems. Furthermore the sequential part of the code is expected to be constant during the increase of the problem size. Hence we have a linear function with a slope smaller than one which describes the scaled speed-up. In theory the following should happen: if we double the problem size and double the available processing units the computation time should stay the same. In Figure 14 the scaled speed-up is seen for the theory and for GPU_2. Because the computation time of the sequential part of GPU_2 increases linearly we have a deviation to Gustafson's law in Figure 14. Nevertheless we still have a linear increase

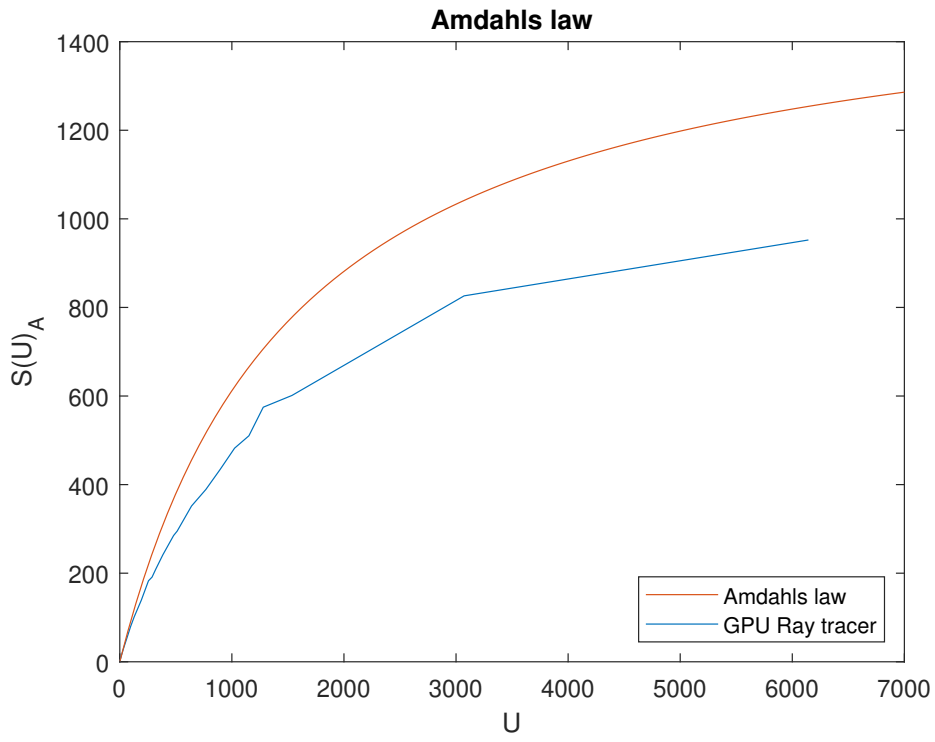

Figure 13

until $\sim$ 1000 threads. Another reason for the deviation may be that the GPU's threads don't work fully in parallel after a certain threshold. In this example we traced 22464 rays in the first run with 2 threads and 92 million rays in the last run with 8192 threads. In summary it proves that GPU_2 has the potential of solving small and big problems with different thread counts in a timely manner. We also have a thread threshold where the GPU stops working 100% in parallel. It depends on the implementation, type of program, and problem size.
Now we come to the speed plots which show the raw computation time of each program including the sequential part. Figure 15 plots the computation time against the problem size with 40 million rays as maximum. On this specific GPU (NVIDIA GTX 1050ti) we can roughly trace 100 million rays until the GPU runs out of memory (4 GB). The CPU_1 program ran on an over clocked 6-core CPU (AMD Ryzen 5 1600) with Hyper-threading (thus 12 threads). When the CPU and GPU were bought they had the same price tag and thus we assume they are comparable. In Figure 15 CPU_1 is clearly the slowest of the three while GPU_2 is even slightly faster than GPU_1.

A better estimation for the exact speed-up is shown in Figure 16 where the GPU program is 6 times faster than the CPU program even with a linear increase of the sequential part. At the edge we have an even higher difference between both programs. The blue curve is plotted by multiplying the computation times of the faster program with the speed-up (in this case 6).

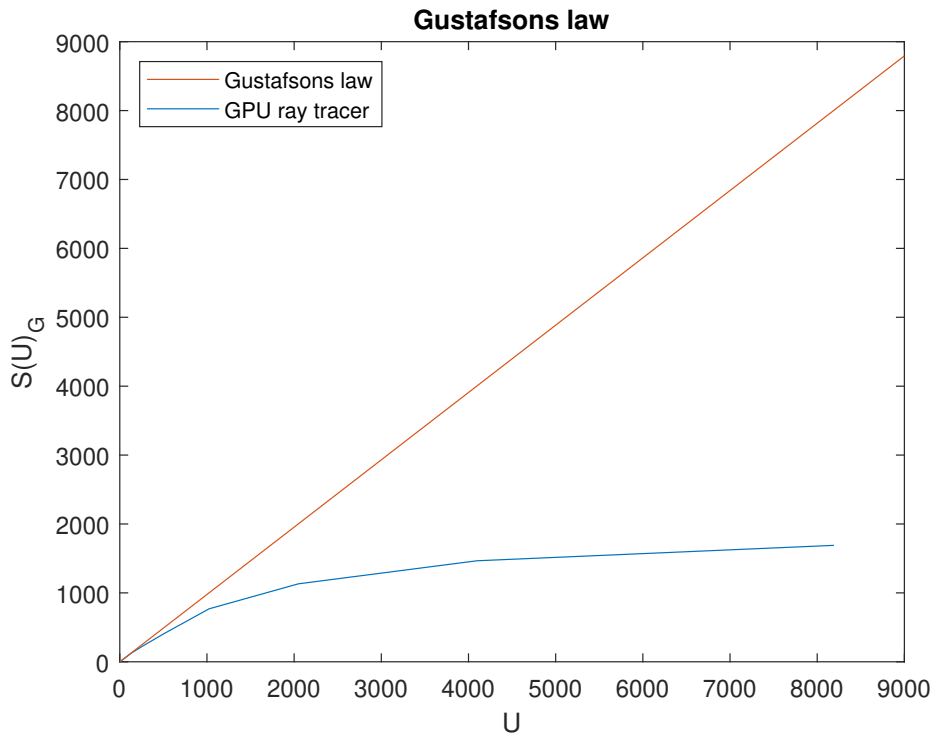

Figure 14

Further down Figure 17 reveals an even higher increase in speed for the GPU_2 program in comparison to the other program. This is the maximum speed-up which was achieved in this work. Finally even a significant difference between both GPU programs is seen in Figure 18. It was achieved by just changing variable types in the whole program. All in all the GPU programs have a considerable advantage over the CPU program concerning the computation times.

It was mentioned in section 4.2 that a new sort algorithm had to be implemented because the already available algorithm did not meet the requirements. Now we have to examine if the new algorithm is viable and faster than the template library `Thrust`. Figure 19 shows how both algorithms cope with an increasing number of to be sorted neighbours. The `Thrust` sort algorithm always sorts the entire array and thus the computation times should not change. However measuring such small computation times on a GPU inevitable lead to a certain variance. Because the GPU is never in idle we cannot deliver exact computation times.

However there is a significant difference between both algorithms in the beginning. In total we had 624 arrays with a length of 624 each which had to be partially (see section 4.2) sorted. The less sorted elements we want, the faster the self-developed sort algorithm is. The blue circle in Figure 19 represents the optimal number of 10 sorted neighbours which was proved in Figure 7. In short the self-developed algorithm is faster and requires less (∼half) memory to be executed

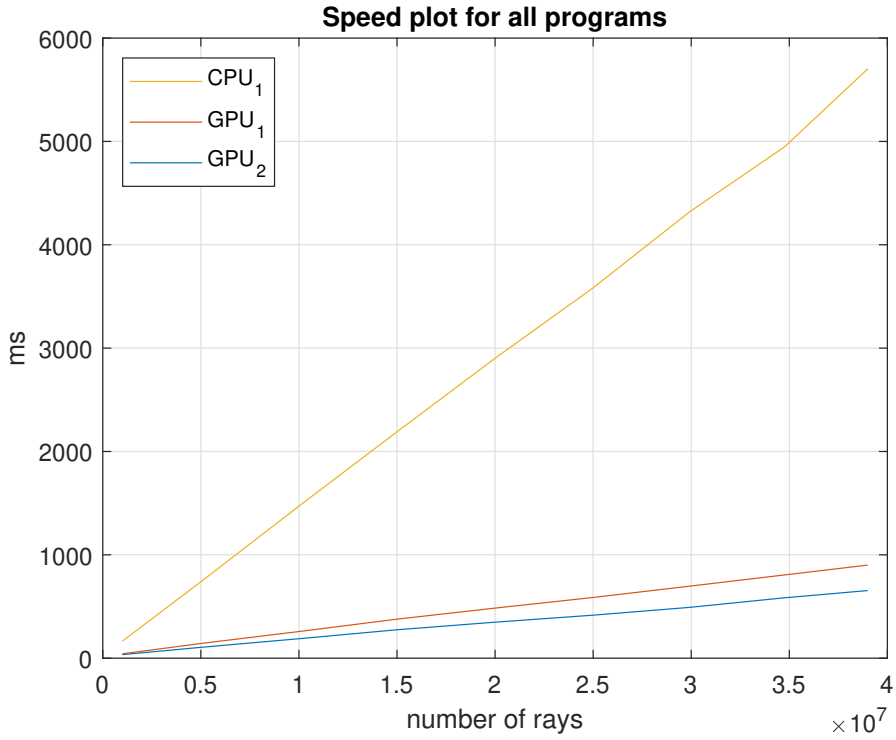

Figure 15

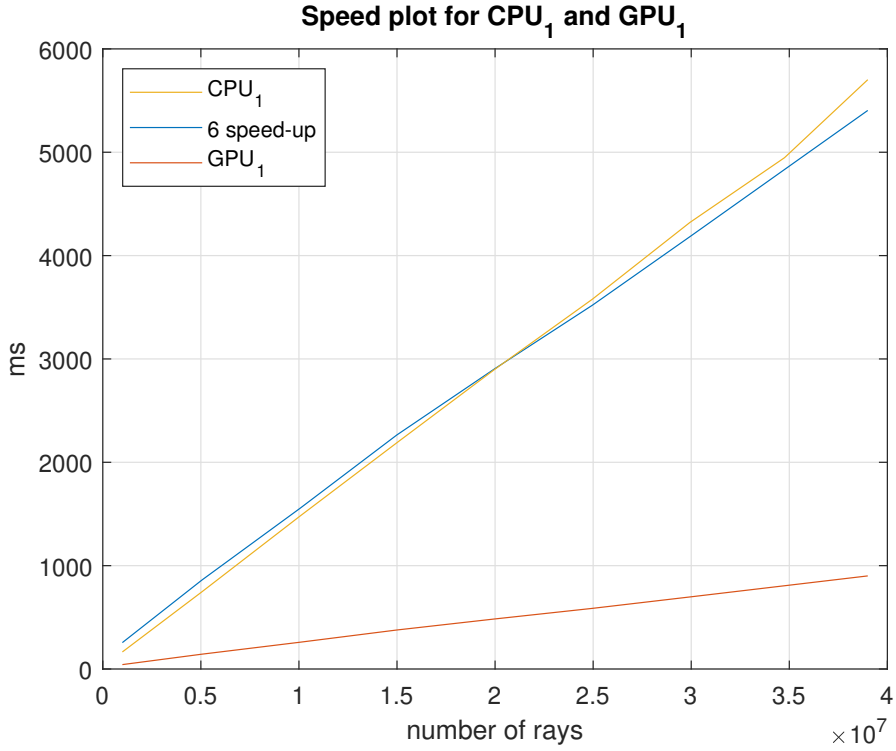

Figure 16

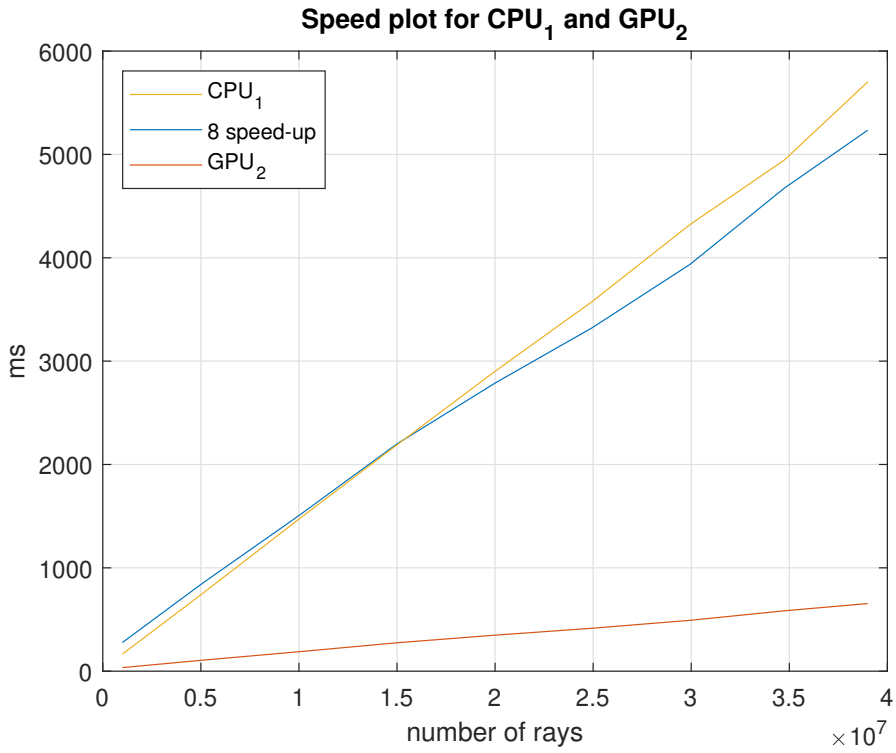

Figure 17

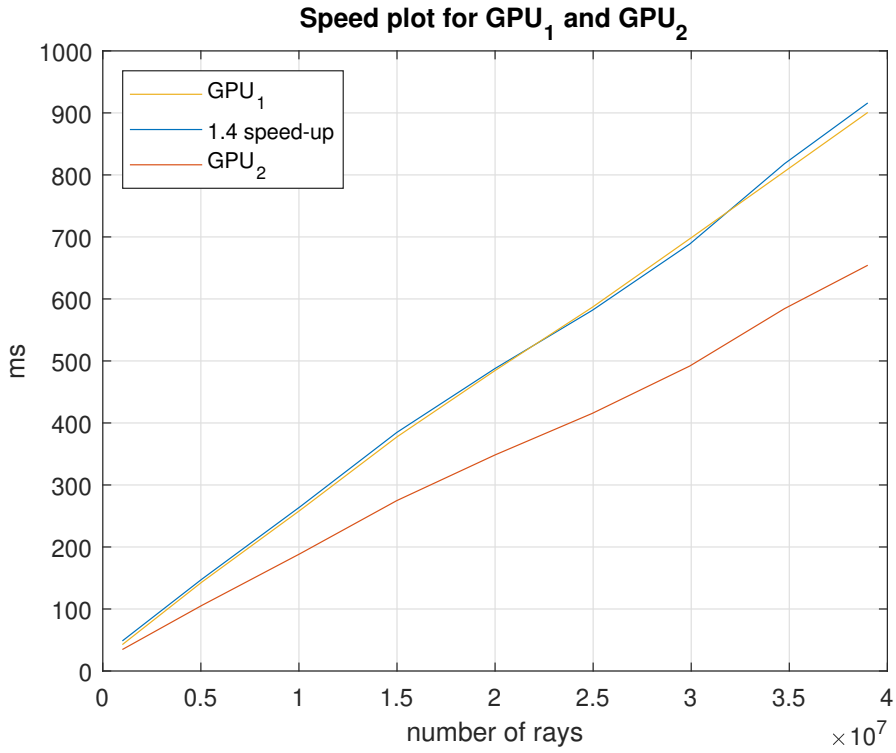

Figure 18

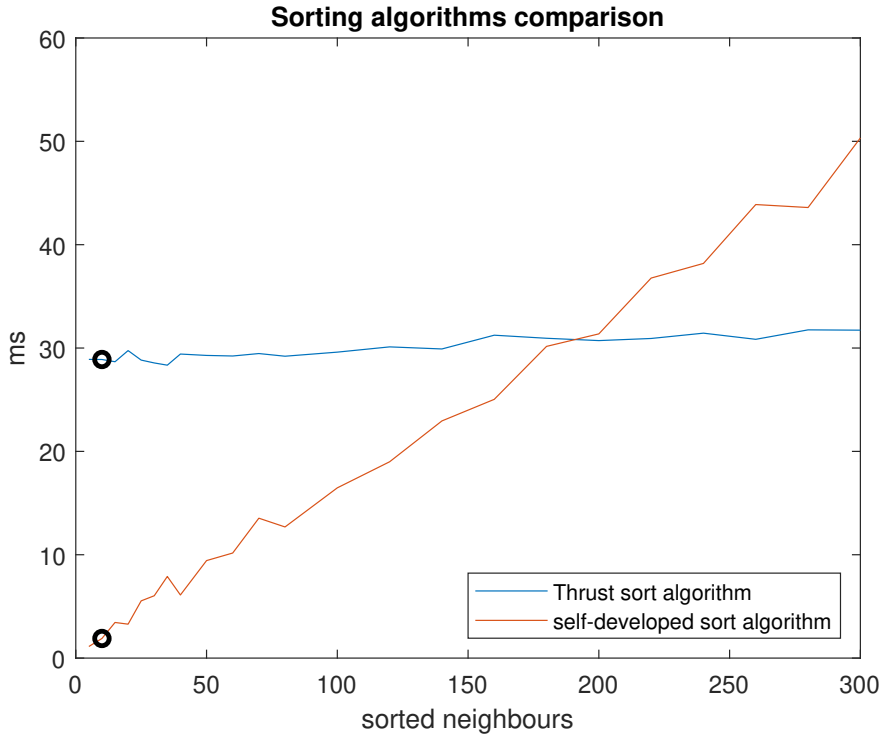
Sorting algorithms comparison
ms
sorted neighbours
Thrust sort algorithm
self-developed sort algorithm
0
10
20
30
40
50
60
0
50
100
150
200
250
300

Figure 19

6.3 Further Improvements

Further improvements are possible for the model which was used for the ray tracing and its implementation:

Global memory

Every time we want to calculate something on the GPU, we first have to give it access to the initial values it needs. This is done by transferring the data to the global memory (Unified Memory) where the GPU is able to access it. The memory bandwidth gives us the peak performance of how fast the GPU can read/write variables. Compared to the computational throughput (see Figure 5) the memory bandwidth is very slow. For the NVIDIA GTX 1050ti we have a bandwidth of 112 GB/s and a computational throughput of 2138 GFLOP/s. Thus it takes the same time reading or writing **one** variable (`float`) as making 76 standard calculations (e.g. multiplication). Essentially the GPU is faster computing than accessing memory. Furthermore the memory space is limited and thus we should only transfer data to global memory which is absolutely necessary for the calculations.
In the programs GPU_1 and GPU_2 we used the Unified Memory to pass the geometry data of the heliostat field to the GPU. But we already preprocessed the data so we do not have to do the same calculation more than once. Since we have so much computation power we should move these to the GPU and favour less memory instead of less calculations.
For very big problem sizes, like 92 million rays, it takes more time transferring the data to the GPU than doing the calculations. The algorithm 1 in section 4.1 requires a full list of rays so the work can be evenly distributed. This leads to allocating a list of 92 million copies of the struct ray. If we want to further improve the computation times of the ray tracer we have to find a different solution to distribute the work evenly.
To sum it up the Unified Memory space is a great way to give the GPU access to the data but it should be used carefully and only for essential data.

Sequential code

Before we can actually trace the rays we have to initialise them by calculating the origin of every ray. Even though it involves few calculations, we still have to iterate through the number of rays which is not efficient. This leads to a linear increase of the sequential code following the increase of rays traced and hindering parallelization. The improvement proposal would be to move these calculations to the GPU.
The bottom line is that the sequential code should not increase linearly with more rays and be as small as possible in order to achieve the best possible speed-up.

Shared memory

Shared memory is an option to optimize data transfers. It is much faster than Unified Memory (see [15]) but it can only be accessed by the threads in the same block. Furthermore the threads have to be synchronised if they pass data to each other.
In our case with GPU_1 and GPU_2 we could for example load the sun's direction into

shared memory because it has to be read a few times. Since one block consists of hundreds of threads and each thread has to read this specific variable three times per ray it would definitely make a difference. If we would trace 40 million rays with 1000 threads every thread would trace 40000 rays. This implies reading sun's direction 120000 times per thread in its lifetime. The `__shared__` instruction only has to be issued once in the beginning where the sun's direction is copied from the Unified Memory to the shared memory. Since the shared memory space is very limited it only makes sense for small variables which have to be read/written often.

Model

The model we used to simulate the CSP plant is sufficiently accurate as seen in section 6.1. We still omitted a few effects which occur in the real world. First we should try to calculate the tower shading. Depending on the height of the tower it may have an impact on the calculated optical power. Furthermore we assumed that the sun emits parallel light. The assumption could be replaced by implementing one of the various sun shape models. In [10] they modelled the sun as a disc where the sun's direction is randomly generated on that disc. The points closer to the center of the disc are more probable than the ones further away.
Moreover we only simulated a short moment of the CSP plant but normally we need at least a simulation of one day, week, month or year. This can easily be done by adding a `for` loop to the program and a list of θ and φ with the corresponding DNI. In every iteration we have to invoke the kernel with a different sun's position and DNI.

Double precision

In the beginning we motivated the use of the variable type `float` with the performance of the GPU (see section 5.1). But since our bottleneck is the memory bandwidth we should give it a try and change the standard variable type to `double`. It is an easy step to implement and the results and computation time can be directly compared to beforehand.

Arithmetic intensity

The arithmetic intensity (AI) measures the ratio between computation and communication cost. The bigger the AI the more calculations we do per read/write instruction. Since our GPU computes faster than it communicates we should aim at a large AI. If we want to improve the computation times of the entire ray tracer we have to go through every line and examine the specific AI and try to improve it. Let's look at a simple example:

```
for (int i = 0; i < M; i++) {
    A[i] = B[i] * C[i];
}
```

We have M floating point operations because of the multiplication. Additionally we read twice per iteration and write once per iteration. Thus we have $AI = \frac{1}{3}$. Overall we want to achieve a big AI ratio in order to fully take advantage of the GPU's architecture.

7 Conclusion

In this section we want to touch on the things which worked out and also on the things which still can be done.
First we can conclude that the overall goal of implementing a fast ray tracer on a GPU was achieved. The results were sufficiently accurate and the speed-up compared to the CPU was significant. The speed-up achieved definitely encourages to keep implementing with CUDA and to transfer certain C++ programs to CUDA. The only condition is that it can be executed in parallel with a large number of threads.
In the beginning the GPU program did not seem any faster than the CPU program because only a small portion was parallelised and the ray count was very small. Progress was made when the specialized CUDA variables (e.g. `float3`) were introduced and implemented. They made calculations easier and accelerated the whole ray tracer. The developed sort algorithm (see algorithm 2) took advantage of the special requirements of the ray tracer. Although the speed-up was impressive, (see Figure 19) the absolute decrease in computation time was too low. Especially since the sorting algorithm has to be executed only once. The next step was to decrease the used memory per ray so we can trace more rays. It also helped the performance because less memory had to be allocated on the device. When the surface error had to be implemented, the question arose about how and where the random numbers should be generated. After extensive testing and tryouts every CUDA thread ended up generating its own random numbers. Furthermore some crucial functions were improved by increasing the arithmetic intensity of the instructions. This also helped improve computation times and overall performance.
The ray tracer still has to be expanded so it can deliver yearly simulations and more accurate and varying effects. Moreover the use of memory has to be improved in order to trace a large number of rays in less time. We have to use more shared memory instead of Unified Memory because it is much faster. Hence the algorithm must change so a group of threads can work together and share information about the results. CUDA offers a lot of different analysing tools which may help improve the implementation and the use of memory.
Researching and developing a suitable model for a ray tracer was interesting and challenging. On the one hand was the theoretical part of how it should work and how the solution could be found. On the other hand you had to implement the formulas and test if the results matched the expectations.
For future work better GPUs and above all more GPUs should be used for the calculations. To integrate several devices will be a challenging assignment. Especially NVIDIA's ray tracing GPUs (see section 3) seem promising for further research. Additionally programming languages, which are available for CUDA, should be compared in order to find the fastest possible implementation.

References

[1] M. Kaltschmitt, W. Streicher, A. Wiese: *Erneuerbare Energien: Systemtechnik, Wirtschaftlichkeit, Umweltaspekte*, Springer, 2013, pp. 293-308

[2] National Renewable Energy Laboratory: *Ivanpah Solar Electric Generating System*
https://solarpaces.nrel.gov/ivanpah-solar-electric-generating-system. Retrieved 18th March, 2019.

[3] United States Energy Information Administration: *Ivanpah 1, 2 and 3 data*
https://www.eia.gov/electricity/data/browser/. Retrieved 25th March, 2019.

[4] BrightSource Energy: *Ivanpah Project Facts*
http://www.brightsourceenergy.com/stuff/contentmgr/files/0/8a69e55a233e0b7edfe14b9f77f5eb8d/folder/ivanpah_fact_sheet_3_26_14.pdf. Retrieved 18th March, 2019.

[5] Bundesnetzagentur: *Kraftwerksliste*
https://www.bundesnetzagentur.de/DE/Sachgebiete/ElektrizitaetundGas/Unternehmen_Institutionen/Versorgungssicherheit/Erzeugungskapazitaeten/Kraftwerksliste/kraftwerksliste-node.html. Retrieved 19th March, 2019.

[6] T. Wendelin, A. Dobos, A. Lewandowski: *SolTrace: A Ray-Tracing Code for Complex Solar Optical Systems*, National Renewable Energy Laboratory (NREL), October 2013.

[7] National Renewable Energy Laboratory: *SolTrace*
https://www.nrel.gov/csp/soltrace.html. Retrieved 26th March, 2019.

[8] J.P. Roccia, B. Piaud, C. Coustet, C. Caliot, E. Guillot, G. Flamant, J. Delatorre: *SOLFAST, a Ray-Tracing Monte-Carlo software for solar concentrating facilities*, Journal of Physics: Conference Series 369 (2012) 012029.

[9] S.J. Bode, P. Gauche, D. Griffith: *A novel approach to reduce ray tracing simulation times by predicting number of rays*, ScienceDirect, Energy Procedia 49 (2014), pp. 2454-2461.

[10] M Izygon, C. Nilsson, N. Vu, P. Armstrong: *GPU-based Monte Carlo Ray Tracing Simulation For Solar Power Plants*
http://on-demand.gputechconf.com/gtc/2012/presentations/S0321-GPU-Based-Monte-Carlo-Ray-Tracing-Sims-for-Solar-Power-Plants.pdf. Retrieved 27th March, 2019.

[11] M. Ewert, O. Navarro Fuentes: *Modelling and simulation of a solar tower power plant*
http://www.mathcces.rwth-aachen.de/_media/5people/frank/solartower.pdf. Retrieved 1st June, 2019.

[12] Nvidia: *CUDA Toolkit Documentation v10.1.168 cuRand*
https://docs.nvidia.com/cuda/curand/index.html. Retrieved 15th June, 2019.

[13] Nvidia: *CUDA Toolkit Documentation v10.1.168*
https://docs.nvidia.com/cuda/. Retrieved 27th June, 2019.

[14] J. Simon: *Paderborn Center for Parallel Computing APRS Lecture*
https://pc2.uni-paderborn.de/fileadmin/pc2/Vorlesung_APRS/VL_APRS-04.pdf. Retrieved 26th June, 2019.

[15] M. Harris: *Using Shared Memory in CUDA C/C++*
https://devblogs.nvidia.com/using-shared-memory-cuda-cc/. Retrieved 27th June, 2019.

Erklärung

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde, sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlrsuhe, den 4. Juli 2019