Parallelization of ray casting for solar irradiance calculations in urban environments

Patrick Eggers
2017

Student thesis, Advanced level (Master degree, one year), 15 HE
Geomatics
Master Programme in Geomatics

Supervisor: Stefan Seipel
Examiner: Bin Jiang
Assistant examiner: Ding Ma
ABSTRACT

The growing amount of photovoltaic systems in urban environments creates peaks of energy generation in local energy grids. These peaks can lead to unwanted instability in the electrical grid. By aligning solar panels differently, spikes could be avoided. Planning locations for solar panels in urban environments is very time-intensive as they require a high spatial and temporal resolution. The aim of this thesis is to investigate the decrease in runtime of planning applications by parallelizing ray-casting algorithms. This thesis includes a software tool for professionals and laymen, which has been developed in a user centered design process and shows ways to perform those calculations on a graphics processing unit.

After creating a computational concept and a concept of the software design, those concepts have been implemented starting with an implementation of the Möller-Trumbore ray-casting algorithm which has been run with Python on the central processing unit (CPU). Further the same test with the same algorithm and the same data has been performed on the graphics processing unit (GPU) by using PyCUDA, a Python wrapper for NVIDIA’s Compute Unified Device Architecture (CUDA). Both results were compared resulting in, that parallelizing, transferring and performing those calculations on the graphics processing unit can decrease the runtime of a software significantly. In the used system setup, the same calculations were 42 times faster on the Graphics Processing Unit than on the Central Processing Unit. It was also found, that other factors such as the time of the year, the location of the tested points in the data model, the test interval length and the algorithm design of the ray-casting algorithm have a major impact on the performance of such. In the test scenario the processing time for the same case, but just during another time of the year, increases by factor 4.

The findings of this thesis can be used in a wide range of software as it shows, that computationally intensive calculations can easily be sourced out from the Python code and executed on another platform. By doing so, the runtime can be significantly decreased and the whole software package can get an enormous speed boost.

Keywords: Solar Radiation, Parallelization, Simulation, Hardware Architecture, Ray-casting
ACKNOWLEDGMENTS

I would like to thank everyone who helped me writing this thesis and supported my exchange year in Sweden. Special thanks go to Professor Stefan Seipel. I owe him thankfulness for his advice and his patience with me throughout the whole thesis. Furthermore, I would like to thank the people involved in the whole process of writing the thesis and designing the software. In particular the person who proofread my thesis several times and did not ask for anything in return. I really appreciate the time that you dedicated to proofreading my thesis and giving me advice how to improve this piece more and more.

Also, I want to thank all people at the University of Applied Sciences Mainz who made my year abroad possible. First of all, to Prof. Dr.-Ing. Harmut Müller, who established the first contact to the University of Gävle and maintains a good relationship with the University. Another person without whom this year would not have been possible is Ms. Margaritha Vogt, who managed to shift exams for a whole semester to give me the opportunity to travel to Sweden early enough to participate in a Language Course. Additionally, I would like to thank Ms. Ulla Plate and her team from the International Office in Mainz for caring about me. Last but not least, I would like to thank the examination board of the Geoinformatics and Surveying division in the Faculty of Technics under the leadership of Prof. Dr.-Ing. Martin Schlüter, for honoring and acknowledging my work at the University of Gävle.

In addition to those, I would like to thank “Cusanuswerk”, a German sponsorship organization funded by the Federal Ministry of Education and Research and the Catholic church in Germany. Being one of their scholars has given me the financial and academic support to make this year abroad project possible and without their support of the exchange, I would not have been able to complete my research project to the best of my ability.

Another organization I wish to thank is the Sea Scouts from Gävle (Gävle Sjöscoutkår). Their numerous group activities throughout the week and during the weekend provided me with the right balance of fun outdoor activities in the Swedish countryside, engaging voluntary work in the local community and working at home or at University for my thesis project.

Finally, I would like to thank my parents and my brother for giving me advice and supporting me in all of my decisions, no matter what.
# TABLE OF CONTENTS

1 Introduction .................................................................................................................. 1  
1.1 Background of this study ........................................................................................... 1  
1.2 Motivation and problem statement .......................................................................... 1  
1.3 Aim of the study ....................................................................................................... 3  
1.4 Organization of the thesis .......................................................................................... 3  
2 Theoretical Background .................................................................................................. 4  
2.1 Physics Background ................................................................................................... 4  
   2.1.1 Sun Position ....................................................................................................... 4  
   2.1.2 Radiation .......................................................................................................... 4  
2.2 Computational Background ....................................................................................... 5  
   2.2.1 Parallel computing ............................................................................................. 5  
   2.2.2 Central Processing Unit vs. Graphics Processing Unit Architecture ..................... 6  
   2.2.3 Ray intersection algorithm (Möller-Trumbore) .................................................... 6  
   2.2.4 Python .............................................................................................................. 7  
   2.2.5 Compute Unified Device Architecture .................................................................. 7  
   2.2.6 Compute Unified Device Architecture in Python ............................................... 8  
   2.2.7 3D-Models and Stereolithography ...................................................................... 8  
3 Methods ........................................................................................................................... 10  
   3.1 Software Concept ................................................................................................... 10  
      3.1.1 Software Design Concept .................................................................................. 10  
      3.1.2 Computational Concept ................................................................................. 10  
   3.2 Implementation ....................................................................................................... 12  
      3.2.1 Graphical User Interface .................................................................................. 12  
      3.2.2 Computation in Python .................................................................................... 13  
      3.2.3 Computation in Compute Unified Device Architecture in Python ..................... 13  
3.3 Evaluation .................................................................................................................. 13  
   3.3.1 Evaluation-Environment ..................................................................................... 13  
   3.3.2 Hardware-Evaluation .......................................................................................... 14  
   3.3.3 Algorithm-Evaluation (Parameter) ...................................................................... 14  
4 Results ......................................................................................................................... 16  
   4.1 Visualization .......................................................................................................... 16  
   4.2 Implementation of the user interface and the algorithms ......................................... 17  
   4.3 Evaluation .............................................................................................................. 18  
      4.3.1 Hardware .......................................................................................................... 18  
      4.3.2 Comparison of results ...................................................................................... 18  
5 Discussion ....................................................................................................................... 22
5.1 Visualization................................................................................................................22
5.2 Computation..................................................................................................................22
5.3 Comparison of results....................................................................................................22
6 Conclusion and future work ............................................................................................24
  6.1 Conclusions..................................................................................................................24
  6.2 Future Work ................................................................................................................25
7 References .........................................................................................................................26
8 Appendices ........................................................................................................................29
  8.1 Appendix A – City and Suburb model in a GIS Environment ......................................29
  8.2 Appendix B – Structural description of the Möller-Trumbore algorithm ..................31
  8.3 Appendix C – Möller-Trumbore Algorithm in Python .................................................32
  8.4 Appendix D – Möller-Trumbore-Algorithm as a PyCUDA function .........................33
  8.5 Appendix E – Calculation of Solar Irradiance with the developed software ..............36
LIST OF FIGURES

Figure 1: The angle of incidence $\theta$ between the collector and the incoming solar beam (Masters, 2013)..........................................................................................................................................................................................................................................................5
Figure 2: Impact of the angle of incidence $\theta$ on the incoming beam (own work)..................................................5
Figure 3: CPU and GPU hardware design (Kirk & Hwu, 2010)..................................................................................6
Figure 4: Translation and change of base of the ray origin (Möller & Trumbore, 1997)..............................................7
Figure 5: CUDA thread organization (Kirk & Hwu, 2010).........................................................................................8
Figure 6: STL format description (University of California San Diego, 2017)..............................................................9
Figure 7: Computational Concept (own work)...............................................................................................................11
Figure 8: Array dimensions (own work, based on (BasicLogging, 2012)).................................................................11
Figure 9: The Möller-Trumbore algorithm in a Nassi-Shneidermann Diagram (own work)....................................12
Figure 10: Glaciärvägen (left) and downtown model (right) (own work).................................................................14
Figure 11: Test sites within Gävle (own work)............................................................................................................15
Figure 12: Final sketch (own work)..........................................................................................................................16
Figure 13: The final realization in Python (own work)...............................................................................................17
Figure 14: Speedup CPU vs. GPU (own work); Blue line: CPU runtime [s]; Orange line: GPU runtime [s]; Grey bars: speedup Factor ..................................................................................................................19
Figure 15: The two diagrams represent different time intervals from 1.6.2010 for 40 days. Left: 10 minutes interval; Right: 40 minutes interval (own work)........................................................................................................................................21

LIST OF TABLES

Table 1: Used hardware (own work)...........................................................................................................................18
Table 2: Runtime Comparison CPU/GPU; multiple points (own work).................................................................18
Table 3: Runtime Comparison CPU/GPU; one point (own work).............................................................................19
Table 4: Location of Point Comparison (GPU) (own work)......................................................................................20
Table 5: Time of Year comparison (GPU) (own work)..............................................................................................20
Table 6: Interval Comparison (GPU) (own work)......................................................................................................20
Table 7: Test area size comparison (own work)......................................................................................................21
1 INTRODUCTION

Energy became more and more important for wealthy communities in the last centuries. A world without electric power is hardly imaginable as it keeps our communities together. Therefore, many different power plants which run mostly on fossil fuels have been built during the last century. Nowadays, communities become more thoughtful and there is a change from unsustainable to sustainable, which means fossil to non-fossil driven power generation. This shift not only includes how power is generated, but also where power is generated. The focus goes from centralized power plants to decentralized power generators like block-type thermal power stations, windmills and photovoltaic systems (PV systems). The change of our mindset is based on multiple reasons. As anthropogenic greenhouse effects become more evident people are more sensitive on environmental questions regarding energy generation. For this reason, a reduction of carbon dioxide emissions is indispensable. As PV systems become cheaper, more efficient and the module costs of price per Watt decrease to 1 US-Dollar and less (Cengiz & Mamiş, 2015), it becomes more profitable to invest in those systems, which do unless conventional power plants, not require any fuel or produce nuclear waste. Furthermore, not a lot of maintenance work is needed. Even though, sunlight is not as strong as in equatorial regions, due to the lower prices per Watt, solar panels become more profitable in zones closer to the poles (Stridh, Yard, Larsson, & Karlsson, 2014).

1.1 BACKGROUND OF THIS STUDY

The consequence of the above mentioned is that more and more house owners install PV systems on their roofs and try to supply as much electrical current as possible. In old power-grids this leads, especially on very sunny and windy days to a challenging situation (Buttler, Dinkel, Franz, & Spliethoff, 2016). On those days, a lot of power is generated by windmills and photovoltaic systems, which leads to an overproduction of energy. Conventional power plants such as nuclear and gas power plants cannot throttle their energy generation accordingly and adjust to the overall power generation as it would be needed in those cases.

A big advantage of PV systems is that the energy is usually produced during peaks of energy consumption and in close proximity to where it is needed. This is usually a decentralized low-voltage grid and therefore, unlike to most windmills, the national transmission network does not have to be expanded (Wirth, 2017). Thus, less PV produced energy is lost due to grid resistance.

One of the major challenges with PV systems is, that most of them are aligned to the peak of the sun to generate as much power as possible. This leads to the earlier described problems of an overproduction of power during peak times. To solve this problem, two solutions could be possible. First, every household which has a PV system could install a battery and charge the battery during the day to use the energy during periods when its own power consumption is high. As Eusebio and Camus (2016) proved with example calculations, the prices for batteries are still too high to work profitable with this approach. A second solution would be to see all PV systems in a grid as a whole and align them, so that peaks in energy generation could be avoided. Then, some PV systems would perfectly produce power in the morning, some during noon and some in the afternoon. The PV systems would not produce as efficiently as possible, but the power would be generated more evenly, and high peaks could be avoided. With PV systems, which become more and more efficient, this would still be profitable.

1.2 MOTIVATION AND PROBLEM STATEMENT

Spatial decision making, which is needed to plan future locations of software panels, has a long history. One of the earliest and probably the most prominent example is more than 170 years old. In
1854 Dr. John Snow identified a water pump as one of the causes of a cholera outbreak in Soho, London. He plotted all locations of cholera deaths on a map and saw the relation between the pump at Broad Street and the many infected people who live close to that pump. By removing the handle of the pump, he ended the epidemic (McLeod, 2000). This story shows, that mapping and geographic representation of problems can be applied in various fields and can help finding solutions for different problems. Nowadays the two-dimensional approach, that has been used for quite a long time can be extended with a third dimension. That three-dimensional data which very often represents urban environments such as cities allows getting solutions for more complex problems and allows the user a three-dimensional exploration of the problem and possible solutions. It helps in planning processes in terms of communication and participation of the different stakeholders such as the city government or citizens (Eran, Hiroshi, Yeung, Underkoffler, & Ishii, 2001).

To plan the locations of solar panels software is needed, which calculates when the sun is at a specific spot and as well as if it shines or not. Especially in an urban environment, those calculations are quite extensive, as many different factors need to be considered. Shadowing effects of other buildings and structures play one of the biggest roles in finding the right spot to install such a system. Tools that achieve those tasks are by Goodchild’s definition part of the field of geographic information science. The described applications have to deal with data measurement, data modelling, algorithms and, depending on the functionality, display of data (Goodchild, 1992). The software needs to combine different components of geographical information science (Mark, Goodchild, & Worboys, 2003). The most important component is computational geometry, due to the calculation of shadowing effects and the time component as the calculations are dependent on the time. Other components, such as scale, are less important, as they are pre-set by the used environment model.

A lot of computational tools which support spatial planners already exist, but the majority have either a lack in the spatial or temporal domain (Horvat & Dubois, 2010). One of the main reasons for the lack of either high spatial or high temporal resolution is the computational intensity of those calculations. A higher degree of spatial resolution means a more detailed 3D-model which leads to more tests that need to be performed and thus to a longer computing time. A high temporal resolution in this context means very small intervals between two moments in which the test is performed. This thesis investigates if and how the temporal and spatial resolution can easily be improved by transferring intensive computations to another processing unit. This novel method of outsourcing computations should contribute to more efficient planning software for exploiting urban solar potential, as Seipel, Lingfors and Widén demand in their paper (Seipel, Lingfors, & Widén, 2013).
1.3 **AIM OF THE STUDY**

To solve the above described problems, the study has three aims which are built on top of each other.

Aim I: A sketch of a software which fulfills the requirements regarding temporal and spatial resolution should be drawn. The sketched software should be easily comprehended by laymen and professionals. Different visualizations should be implemented including diagrams and maps as well as 3D-Models for interactive exploration.

Aim II: The sketched software from Aim I should be transferred into a software written in Python. In that software it should be possible to import 3-dimensional city environments and points through a guided dialog. Furthermore, the software should calculate if the sun is visible at the specific imported points at predefined times. Therefore, a ray-intersection algorithm should be performed on the 3D-city model. All computations like ray casting, in this case collision detection between rays and buildings and calculations of radiation should be written in Python and calculated on the Central Processing Unit (CPU).

Aim III: The third aim should to evaluate the performance of geometric algorithms regarding sunlight calculations in high spatial and temporal resolution on two different processor platforms. Therefore, the most intense calculation of Aim II, the ray casting, should be calculated with support of Nvidia’s CUDA platform on the graphics processing unit (GPU). The aim is reached by comparing the performance of the same data processed with the same algorithm on both, CPU and GPU, as well as investigating factors affecting the runtime. Those are the location of the point in the 3D-Model, the season in which the tests are performed and the length of the interval between moments when tests are done.

1.4 **ORGANIZATION OF THE THESIS**

First, this thesis gives some background information on the physical and technological background. Then it conceptualizes a software design, analyzes the problem in Python and then as a fourth step transfers the most time-consuming task performed on the CPU to the GPU to analyze the differences in calculation times. All code will be written in Python to make it later available for other software to use it as a plug-in. Examples are GIS-Software such as QGIS or ESRI’s ArcGIS, as their Application Programming Interface (API) is written in Python.
2 THEORETICAL BACKGROUND

This chapter introduces the physical and computational background of the thesis. Major factors like the calculation of the relative sun position and the calculation of the irradiance on the earth surface will be covered. The second part focuses on the computational background. It introduces parallel computing, explains used algorithms and describes the difference between calculations on the CPU and on the GPU.

2.1 PHYSICS BACKGROUND

This section depicts the two main physical foundations that are necessary to understand this thesis. It gives a short introduction how the position of the sun, which is necessary to calculate a sun vector, is calculated and how the radiation at a specific point can be calculated. That calculation is necessary to get an understanding what the difference between the power output of the sun and the possible energy yield on earth is.

2.1.1 Sun Position

The first step, for very precise calculation whether the sun is shining or not, the sun position has to be determined. Various algorithms of different complexity exist in this field. One of the most recent developed calculation methods, which is widely used within science as its errors in the sun-angle are in a range up to 0.0003°, is used in this thesis. After predefining latitude, longitude and time of a specific place, it allows calculating the azimuth and the altitude angle for that specific time and position. The algorithm has been adopted by Ibrahim Reda and Afshin Andreas at the National Renewable Energy Laboratory, which is part of the Midwest Research Institute in Colorado, from a method that has been deployed in France in the 1980s (Andreas & Reda, 2008).

2.1.2 Radiation

Clear sky direct beam radiation is usually used to estimate the power output of photovoltaic systems. Other kinds of radiation such as diffuse and reflected radiation play just a minor role in calculating the power output. The radiation hitting the earth surface is usually indicated in Watts per square meter. To estimate the solar irradiance, the first step is to estimate the extraterrestrial solar insolation. This is an imaginary value which varies throughout the year as it depends on the distance between earth and sun. But as this factor is predictable, it can be eliminated. After the ray passes the imaginary barrier gases absorb some of the radiation and the beam gets scattered by molecules (Masters, 2013). On a clear day, there is a loss of about 30% of the extraterrestrial flux compared to the radiation on earth. The atmospheric factors can be estimated by calculating the air mass ratio and optical depth, which are both depending on the time of the day. After applying all given factors, the solar irradiance on earth can be approximated. The resulting direct insolation on a PV system, is calculated by applying a cosine-function of the normal vector of the collector and the angle of incidence on the direct radiation. If the absolute values of the normal vector of the collector and the angle of incidence are the same, there is no difference between the direct radiation and the insolation striking the collector (Figure 1 & 2) (Masters, 2013).
Figure 1: The angle of incidence $\theta$ between the collector and the incoming solar beam (Masters, 2013)

Figure 2: Impact of the angle of incidence $\theta$ on the incoming beam (own work)

2.2 COMPUTATIONAL BACKGROUND

The computational background section introduces the technological background of the thesis. The section begins with an introduction to parallel computing and differences in hardware architectures of CPUs and GPUs. That part is followed by an explanation of the used programming languages and how one language can be wrapped, so it is accessible in another one. Furthermore 3D-models and the STL-file format are introduced. Also, the calculation part on which this thesis focuses, the ray intersection algorithm is introduced. The algorithm is needed to check if a sun-beam reaches a predefined point or hits a building or another structure, that is predefined.

2.2.1 Parallel computing

Parallel computing is defined as “simultaneous processing by more than one processing unit on a single application” (Deng, 2012). The whole process of parallelization can be split into four different parts. Firstly, the given problem is divided into different parts that can be solved independently from each other. Secondly, each part is broken down into different instructions. Then all instructions are performed on different processors simultaneously and finally a mechanism that controls everything is applied (Barney, 2017). To improve a task by parallel computing, the given problem needs to be divisable into different tasks that are independent from each other.

As a lot of processes in the real world are parallel, parallelization of computational processes is obvious as it saves time and money. Compared to serial computing, it can also solve larger and more complex systems in less time. Nowadays, a lot of different projects that take advantage of parallel computing exist. Supercomputers, as the most powerful computers are called, do not necessary need to be local. Different tasks of a huge amount of calculations can also be distributed to different places in the world. Examples include the SETI@home project from Berkeley University and the Folding@home from Stanford University (Barney, 2017). Both projects have more than 1.5 million
users globally. Furthermore, non-scientists and owners of desktop-computers can take advantage of parallel computing. Nowadays, nearly all personal computers have central and graphics processing units that consist of up to a few hundred cores that can work simultaneously.

2.2.2 Central Processing Unit vs. Graphics Processing Unit Architecture

Central Processing Units are designed for tasks like word processing and web browsing, which are very complex and do not have a huge potential for parallelization (Klöckner, et al., 2012). Their design focuses more on a few cores with a lot of cache memory, which allows a very fast sequential processing of data (NVIDIA Corporation, 2017). GPUs in contrast have many more smaller cores, which focus on uniform, less complex and non-sequential operations on large datasets (Klöckner, et al., 2012). Depending on the complexity and the design of the task GPUs can perform calculations up to 100 times faster than CPUs (NVIDIA Corporation, 2017). Besides the different number and power of the cores, the data storage and supply model is quite different, too (Figure 3). Each system is just as strong as its weakest part and for CPUs nowadays, the bandwidth from data storage devices to the actual core is a problem. CPUs and other general-purpose processors need, in contrast to GPUs, to satisfy some requirements concerning applications and Input-Output devices, which GPUs do not need to meet. Hence, it is much easier to achieve higher memory bandwidth for graphic chips. The memory bandwidth is very dependent on the interaction between the memory and the processor. GPUs usually do not use the same memory modules as CPUs do, but the so-called Graphics Double Data Rate Random Access Memory (GDDR RAM), which is compared to the RAM on the motherboard a very-high bandwidth off-chip memory. It has a higher latency than normal RAM, but this is just a minor factor when it comes to parallelization (Kirk & Hwu, 2010).

![CPU and GPU hardware design](kirk_hwu_2010)

2.2.3 Ray intersection algorithm (Möller-Trumbore)

Urban 3D-models are created out of many triangles representing all surfaces of buildings and structures. To calculate, if the sun is visible from a certain point, it is essential to check, if a ray between the specific point and the sun hits an existing structure in an urban 3D-model. A very fast ray-triangle intersection algorithm has to be applied. Even though, it is difficult to compare different ray-triangle intersection algorithms as they depend on many different factors. The Möller-Trumbore algorithm is considered as one of the most performant ones. This algorithm is named after its designers Tomas Möller and Ben Trumbore who invented it in 1997 (scratchapixel, 2017).

Algorithms which have been developed prior to the Möller-Trumbore algorithm, have always checked first if there is an intersection between the ray and the plane containing the triangle and then if the intersection is within the edges of the triangle. The biggest advantage of the Möller-Trumbore
algorithm is that those computations do not have to be performed. By applying different kinds of transformations and translations, a vector which includes the distance between the intersection of the ray and the triangle is calculated (Figure 4). If this distance is zero, the ray intersects the triangle (Möller & Trumbore, 1997).

\[ \text{Figure 4: Translation and change of base of the ray origin (Möller & Trumbore, 1997)} \]

2.2.4 Python

Python is a widely-used programming language which has been first released by Guido von Rossum in 1991 (Venners, 2003). As it is a language which is very easy to learn, it is widely used in a lot of different applications in everyday life and science. Nowadays a lot of add-on libraries for various tasks have been developed and as the language is very intuitive, programmers can code up to approximately six times faster than programmers in low-level programming languages such as C or C++. Reasons for this speedup are for example that Python programmers do not have to deal with garbage collection, an automatic memory management, defining datatypes and memory allocation whereas C-programmers do (Scribblings, 2010). Python developers are aware of this lack in performance and describe Python often as a good “glue language”. It glues calculations which are performed in other languages together (Python Software Foundation, 1997). Python is nowadays used in two different versions. An old version (2.7) is still maintained until 2020, as many older applications are based on it and the code is sometimes not portable to the present version (3.X). In the TIOBE Index, an index that measures the popularity of programming languages, Python is currently (May 2017) ranked as the 4th most popular language (TIOBE - The Software Quality Company, 2017).

2.2.5 Compute Unified Device Architecture

CUDA (Compute Unified Device Architecture) is a platform by NVIDIA which allows parallel computing on the GPU (NVIDIA Corporation, 2017). It is implemented in every NVIDIA GPU since the G80-generation, which has been released in 2007. CUDA allows programmers to avoid having knowledge of OpenGL and DirectX as CUDA code can be written in C (Ruenda & Ortega, 2008). It makes it easier than before, for programmers to run general purpose computations on GPUs. There are just a few development tools for GPU programmers, as the focus has been on CPU programming. With those CPU based high-level languages the programmer does not have to consider hardware architecture (Klöckner et al., 2012). Those high-level languages and integrated development environments (IDEs) are not available for GPUs yet, so programmers still have to deal with redesigning algorithms so that they can efficiently run on GPUs, as their hardware designed for pixel and geometry calculations. Programmers need to keep in mind that changing GPU code is much more sensitive to small changes than CPU code, due to the hardware architecture. But still, if the calculations are parallelizable, high increases of the runtime can be reached (Kirk & Hwu, 2010).

Parallelization is possible as GPUs consist out of many more processor cores than CPUs. Each CUDA-program has a kernel function which runs synchronously with different data on the GPU. This technique is called SPMD (single program multiple data) and allows parallel computing. Whenever a
kernel is run on the GPU a grid of parallel threads, is run on the GPU. For better organization, all threads in a grid are clustered in different blocks (Figure 5). All blocks and threads are organized in a hierarchy of one, two or three dimensions. This division into small computations is necessary to distribute the tasks to different processors. (Kirk & Hwu, 2010)

Figure 5: CUDA thread organization (Kirk & Hwu, 2010)

2.2.6 Compute Unified Device Architecture in Python

PyCUDA is a wrapper for Python which gives the user access to Nvidia’s parallel computing API CUDA from Python. The wrapper is under development since 2009 and is led by Andreas Klöckner. The main idea behind it is to join the ease of use of Python and the very high advances of performance from parallelization on GPUs (Klöckner et al., 2012). In other words, to free the CPU from additional calculating and let the high-level scripting language just be responsible for control, communication and displaying of data streams. PyCUDA has the same performance as any other C-controlled program, but writing the communication between the different parts of the program in Python is much less effort for the programmer than in a low-level language such as C or C++ (Klöckner et al., 2012).

2.2.7 3D-Models and Stereolithography

To reach the goals of this study, a sufficient 3D-model is indispensable. Such models are generated through different kinds of data which includes GIS, Computer Aided Design and Building Information Modelling Data. Different storage models of data exist, but no matter which model is used, different Levels of Detail of the model can be defined (Biljecki, Ledoux, Stoter, & Zhao, 2014). To perform a solar irradiance simulation, a 3D-model of an urban environment needs to be used and different calculations have to be performed on it. Those models usually contain many polygons, which represent buildings and structures. To fulfill the aims of the thesis, a simple data-format is needed.

The STereoLithography (STL) file format has been developed and is used as an industry standard for data exchange of 3D-models between different design programs since 1987. Nowadays the file format is implemented in a lot of different software packages and mainly used for 3D-printing applications. The data format stores 3D-surfaces in an unordered list of triangles which consist out of three points with three floating point numbers and a 3-dimensional normal vector (Figure 6). Color or surface information is not saved in the format originally, even though there are some unofficial extensions nowadays. The simplicity of the data format is at the same time its biggest disadvantage. Every vertex which is shared by different triangles is saved several times. Due to rounding and inconsistency small leaks can occur. Also, naturally no physical units are defined. This information has to be saved in metadata files. Another disadvantage is saving the surface normal. As all triangles are by default defined by the right-hand rule, the normal vector could be calculated when needed, so this information is superfluous (Hiller & Lipson, 2009).
solid
...
facet normal 0.00 0.00 1.00
  outer loop
    vertex 2.00 2.00 0.00
    vertex -1.00 1.00 0.00
    vertex 0.00 -1.00 0.00
  endloop
endfacet
...
endsolid

Figure 6: STL format description (University of California San Diego, 2017)
3 METHODS

The first section of this chapter describes the concept of the software and the computational design. The following sections document the development of the software. First the Python implementation and programming of the graphical user interface and then the implementation of the CUDA-Code in the python software. Finally, the Evaluation is depicted. The different evaluation environments, as well as the hardware setup of the computer and the parameters of the algorithm are pointed out.

3.1 SOFTWARE CONCEPT

The software concept part is divided into two subsections. The first one deals with design issues of the software and explains the process how the software is designed. That subsection also emphasizes the methods how the requirements on the usability should be reached. The second one elaborates how, and in which order the computations are performed and on which computational platforms. Those processes are visualized in diagrams for a better understanding of the matter and give an impression how modulization of the different calculation parts has been achieved.

3.1.1 Software design Concept

Creating a software with a user-friendly environment, which can be understood by people with professional and non-professional backgrounds needs to follow different rules. As development in human computer interaction is quite fast and new interaction methods develop constantly, standards in this field relate to usability in general. ISO 9241-11 defines usability as “Usability: the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use” (Bevan, 1995).

To reach a high usability, the developed software has been created in a user centered design (UCD) process which is described in ISO 9241-210. This standard gives recommendations and requirements for human-centered design principles (International Organization for Standardization, 2010). The UCD process is divided into three different steps. The first one is called the “Analysis” part and during that stage, the needs and goals of the software are analyzed to give the programmer some understanding of the context the software will be used for. The user is asked which tasks should be performed and how the results should be visualized, and use-cases are developed.

The second stage is the “Design” stage in which conceptual models, wireframes and prototypes are presented. Together with the third stage, the evaluation of the design, it forms an iterative process, before the actual software is developed and all functionalities are implemented (Web Accessibility initiative, 2004).

The software is designed by applying the UCD process. Two laymen with no background in spatial planning, but in computer science and with knowledge of the UCD process were confronted with the problem set and their task was to design an application which can be understood by everyone. At the end of the process, the result was presented to a professional, whom is working with and doing research in the field of solar radiation in urban environments. After final suggestions, the software was realized.

3.1.2 Computational Concept

The computation is divided into different steps (Figure 7). In this way, each sub-result can be exported to other software and the input can also come from another software. This allows a high degree of separability of the different tasks. Different calculations can be excluded and performed by other software modules. This concept allows to compare the ray tracing and collision detection on the CPU and the GPU, as both can be calculated individually and compared afterwards.
The first step includes all preparing tasks for a calculation. After the startup of the software, the first operation that is done, is the import of the urban 3D-model. Then a point file, which contains the origin of the coordinate system in World coordinates as well as all points which are saved in local XYZ-coordinates, for which the possible radiation should be calculated in the process. Then the program sets initial calculation variables which can be modified in a later step.

In the second step, a two-dimensional array \((m\times d)\) is created (Figure 8). It contains datetime objects, which represent all moments for which the radiation of each point is calculated. The size of one
dimension (m) is as big as the number of moments which are checked per day. This means for example, if the radiation should be measured every 10 minutes, that one day has 144 moments (24 hours/10 minutes). The second dimension (d) is as big as the number of days for which the calculation should be performed. In the third step, an array of the same size as the first array (m*d) is created. For each moment a three-dimensional vector from the earth-surface to the sun is calculated. If the z-component of the vector is less than zero, a zero is written into the vector, as it means the sun did not rise at that moment. This vector array is not calculated for each point, as there is no difference between the vectors from each point to the sun.

In the fourth step, a collision analysis is performed by applying the Möller-Trumbore-Algorithm. This is done for every ray from every point with the whole urban 3D-model. For each point it is checked, if the ray from the point to the sun collides with one of the triangles that form the buildings (Figure 9). This calculation results in a three-dimensional array with boolean values. The first two dimensions of the array have the same size as before (m*d). The third dimension is the number of points, as this calculation has to be performed for each point.

![Ray-triangle-intersection diagram](image)

*Figure 9: The Möller-Trumbore algorithm in a Nassi-Shneidermann Diagram (own work)*

The resulting array with boolean values is used in the next step to calculate the radiation for each moment. First, it is checked if the value in the array from the fourth step is true, which means the sun is shining at that specific point and the ray between the sun and the point does not hit any triangle in the mesh. Then, the datetime object from the array, which has been generated in the second step, is used to calculate the radiation. All those calculations are parallelizable as they are neither interdependent nor consecutive.

### 3.2 Implementation

According to the aims, the first software approach is written in Python. This section gives an insight which libraries have been used and why. It is divided in a GUI and a computational part. The GUI part is used for both, the computational part in Python and in PyCUDA. As all required libraries are available in Python version 3.X, the implementation is realized in Python 3.X to ensure good accessibility in the future and easy use as a plug-in for future software projects.

#### 3.2.1 Graphical User Interface

The Graphical User Interface (GUI) is realized with the *tkinter* library which is the standard Python User Interface library. The toolkit can be used on most Unix, Mac OS X and Microsoft Windows platforms (Python Software Foundation, 2017). Besides setting up the frame, the GUI is
used to lead the user through the import process of the STL-File and the dates and times for displaying can be manipulated with it. To display different kinds of data in high quality figures Matplotlib, a multi-purpose plotting library is used. It is used to visualize the triangle mesh as a navigable 3D-Model and to visualize the radiation array as a diagram. With its already built-in functions for rotation and zoom and exporting the current view of the model as a .png-file, the library suits all the requirements which are set in the design concept.

3.2.2 Computation in Python
The computation is performed in different steps. All steps (Figure 7) have been realized in Python. To import store and use 3D-models stored in STL-Files the numpy-stl library has been used. It is developed to read, write and modify STL-files in Python. With support of further libraries such as re, a handler for regular expressions and numpy, a Python package for scientific computing which comes with build-in functions such as basic geometric matrix functions like the cross and dot product, all simple and non-penalizable computations have been performed.

3.2.3 Computation in Compute Unified Device Architecture in Python
When Using PyCUDA, different steps need to be done before CUDA code can be executed. First of all, the PyCUDA libraries are imported and initialized. As a next step, memory on the CUDA-capable device is allocated and the data that will be used is transferred onto the device. Usually, this data is stored in single precision numpy-arrays in Python. After transferring data, a Kernel is invoked, which is written in CUDA-C, a C-like low-level programming language. After executing the Kernel, the data is fetched from the GPU-memory again and output as a normal numpy-array in Python.

The implementation of the Möller-Trumbore algorithm in CUDA takes much more time than in Python. Geometric functions like the Cross and Dot Product, but also simple Vector-Subtractions are manually implemented and not predefined as in Python. As the code is written as a String in the IDE, debugging is not possible in traditional means. The simplest form of debugging, a printing out of the results after the calculations is applied. Just the third step in Figure 7, the calculation if the ray hits the building has been performed on the graphics card processor.

3.3 EVALUATION
The evaluation of the software is divided in three main parts. The first one evaluates the realized visualization concept. The second one evaluates the different hardware platforms and how the performance gain by changing this factor. The third part evaluates the used algorithms and how different parameters affect the runtime of the calculations.

3.3.1 Evaluation-Environment
All tests were performed in the city of Gävle, Sweden as a 3D-model of the city was available. Different test sites were selected (Figure 11). Two smaller datasets on a block level (Figure 10), one dataset comprises a whole neighborhood and one represents a whole city. The smaller models contain a building block downtown around the main square and consists out of 1592 triangles. The other block, called “Glaciärvägen” contains 752 triangles and is located in the suburb of “Sätra”.

The neighborhood-level dataset comprises the whole neighborhood of “Sätra” and has 56,346 triangles (Model “Suburb”). The city whole city model contains all buildings in the city of Gävle and consists of 347,236 triangles (Model “City”). In all scenarios, different tests were performed to evaluate the relevance of the computing hardware and the parameters.
3.3.2 Hardware-Evaluation
The hardware is evaluated by measuring the time when running the same algorithms in Python on the CPU and CUDA-C on the GPU. The time is saved in a very-high precision before initializing the function and after getting a result back. The times are subtracted, and the factor of improvement is calculated. To get comparable results no other parameters have been changed in the software. On the computer system, no other software ran at the time the tests have been done, so that they are not influenced.

3.3.3 Algorithm-Evaluation (Parameter)
To evaluate the algorithms, three different parameters are changed in the setup. The first parameter is the time of the year. That parameter checks if the time of the year has an effect on the calculation time. To get the best results, the weeks around the winter and summer solstices are used. The second parameter is the length of intervals. They varied between 10, 20 and 40 minutes. Giving then finally a different level of detail in the measurements. A third parameter is the size of the dataset. Three different sizes are evaluated. A dataset which includes a block of houses, a suburb and a whole city (Figure 11). All datasets are tested with two different time intervals to get independent results that can be compared with each other. A last parameter that is evaluated is the location of the tested points in the dataset. The test checks if it affects the calculation time whether the tested point is early or late in the 3D-model.
Figure 11: Test sites within Gävle (own work)
This chapter shows the results and evaluations of the different methods that are described in the preceding chapter. All tests were done with data from a 3D-Sketchup model of the city of Gävle, Sweden. Parts of the model were exported as an STL-file and then imported in the software. The points are chosen due to different factors which are explained later in this chapter.

4.1 Visualization

As already mentioned, the users were defined as both professionals and laymen. One example use case was that a user wants to install solar panels in his garden, so that they can power fans in his detached house during afternoon and late afternoon.

After going through the steps of the user centered design process, the final design of the software was as follows:

The final design for this project is a screen which is divided in two a left and a right part. The left part shows radiation-diagrams. Those are two dimensional diagrams, in which each pixel represents the possible energy yield at a specific time. The horizontal axis of the diagram represents the time of the day and the vertical axis of the diagram the day of the year. This visualization shows intuitively the variation and availability of solar irradiation throughout the day during the specified period. One such diagram exists for each 3D-point in the point file. The diagrams of the different points are organized in tabs. The right side is divided into an upper part, the control part, and a lower part, the 3D-visualization. In the upper part, it is possible to control the day on which the calculation period should begin, how long the intervals between the moments of calculation are and for how many days it should be computed. In the lower part, the imported 3D-model as well as the imported points and the rays between them and the sun are visualized. By clicking on a pixel on the diagram, the 3D-view is updated and displays the points and their rays between them and the sun for the moment which the clicked pixel represents (Figure 12).

![Figure 12: Final sketch (own work)](image-url)
4.2 IMPLEMENTATION OF THE USER INTERFACE AND THE ALGORITHMS

The Python implementation consists of two parts. The implementation of the graphical user interface and the computational part. The first part was a one-to-one transfer from the drawn sketch. The chosen libraries have been used. Some programming has been necessary to display the diagrams, implement different tabs and include the 3D-model (Figure 13). The logic in the software has been implemented by following the computation model (Figure 7). No changes to the initial model were made. An example implementation of the Möller-Trumbore algorithm can be found in the Appendix (
Appendix C – Möller-Trumbore Algorithm in Python).

Compared to the fast implementation of the Möller-Trumbore algorithm in Python, the implementation in CUDA-C was much more time consuming. Many functions that are built in in Python are not available in CUDA-C. Hence, they needed to be manually implemented. As no debugging was possible, the whole process took much longer. For comparison of the final algorithm in CUDA-C with the Python algorithm, an example of the function can be found in the appendix (Appendix D – Möller-Trumbore-Algorithm as a PyCUDA function).

Figure 13: The final realization in Python (own work)
4.3 Evaluation

This section describes the used hardware in all tests and the results of the comparisons between the influence of different parameters and performing hardware considering the time in which the calculations are done. By changing the number of points, number of triangles in a mesh, number of moments when measurements were taking place, location of tested points and time of the year of measuring, different results were achieved and compared.

4.3.1 Hardware

As the main focus of the Thesis is on hardware comparison, the used hardware is presented here. The hardware in the computer on which the calculations were performed, has a huge impact on the results. The configuration of the hardware is built in in a Lenovo laptop and has not been changed since the purchase. As the CPU and GPU have about the same release date and are in the same price range, they are comparable and suitable for the performed tests. In this case the measurements were performed on a Personal Computer running Windows 10 with the following specifications:

<table>
<thead>
<tr>
<th>Table 1: Used hardware (own work)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
</tr>
<tr>
<td>Model</td>
</tr>
<tr>
<td>Cores</td>
</tr>
<tr>
<td>Clock Speed</td>
</tr>
<tr>
<td>Main Memory</td>
</tr>
<tr>
<td>Release Date</td>
</tr>
</tbody>
</table>

4.3.2 Comparison of results

The tests that were described in Chapter 3.3 give different results. This subsection shows the achieved results and sets them in context. The analyses have different backgrounds and the changed parameters point to different strengths and weaknesses of the applied algorithms. The first test is the one, that this thesis focuses most on. The comparison between the CPU and the GPU. The following tests are not dependent on the used hardware, but just on the algorithms. No matter on which platform they are run.

4.3.2.1 CPU vs. GPU

The first tested scenario is located in north Gävle in the Suburb Sätra and deals with the “Glaciärvägen”-model. Therefore, a test with five points in the middle of a block of building was conducted. Starting on the 01.05.2010 with a measuring interval of 10 minutes for 1, 10, 25 and 50 days. All buildings in the block of buildings with which the Möller-Trumbore test is performed consist out of 752 triangles. The points have been randomly chosen and are in one line. The speedup factor from CPU to GPU calculations is always around 40 (Figure 14).

<table>
<thead>
<tr>
<th>Table 2: Runtime Comparison CPU/GPU: multiple points (own work)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of days</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>50</td>
</tr>
</tbody>
</table>
The second scenario, the “downtown model”, is located in the city center of Gävle and deals with one point in the middle of a big square. The calculations are done in time intervals 20 minutes. The calculation begins always on 1.1.2010 and lasts is for 1, 25, 50, 100, 200 and 365 days. The used 3D-model contains out of 1592 triangles. The point in the middle of the square has been chosen, as the results for that point are in other tests much more comprehensible for the different seasons than randomly chosen points.

Table 3: Runtime Comparison CPU/GPU; one point (own work)

<table>
<thead>
<tr>
<th>Number of days</th>
<th>Runtime on CPU [s]</th>
<th>Runtime on GPU [s]</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.369856</td>
<td>0.019012</td>
<td>12.45</td>
</tr>
<tr>
<td>25</td>
<td>12.328611</td>
<td>0.312785</td>
<td>39.42</td>
</tr>
<tr>
<td>50</td>
<td>35.422929</td>
<td>1.212365</td>
<td>29.19</td>
</tr>
<tr>
<td>100</td>
<td>112.207257</td>
<td>3.207257</td>
<td>35.16</td>
</tr>
<tr>
<td>200</td>
<td>425.636446</td>
<td>10.036754</td>
<td>42.41</td>
</tr>
<tr>
<td>365</td>
<td>647.824722</td>
<td>15.380454</td>
<td>42.12</td>
</tr>
</tbody>
</table>

In both tests, it is clearly visible that the speedup of the GPU compared with the CPU is with small deviations always the same. The bad speedup for just one day is explainable with the necessary transfer of the data to the memory of the GPU and back. That transfer takes the most time of the whole calculation. As that transfer time is nearly the same, no matter how big the transferred arrays are, the transfer-time does not play a role anymore.

4.3.2.2 Location of Point
The third scenario compares, within the “downtown model”, three different locations of points. After analyzing the STL-file, the points have been set in different parts of the 3D-model. The first one is located between buildings that consist of triangles which are in the top-part of the triangle mesh. The second and third point are located in an area between triangles which are in the last 30% of the mesh-file. The fourth point is located 40 meters above the main square, as no collisions should be detected.
there. The goal of this test is to evaluate the performance of the Möller-Trumbore algorithm and its dependence on the order of the triangles in the mesh. The tests were performed from 1.1.2010 every 10 minutes for 200 days. The final radiation arrays (fourth output in Figure 7) of the tested points which indicate if the sun is shining at the specific points are compared with the sun-vector array (second output in Figure 7) which indicates if there could be possible sunshine or not. The number of how often the point is in a shade of a building (at how many moments the ray hits at least one triangle) is calculated.

Table 4: Location of Point Comparison (GPU) (own work)

<table>
<thead>
<tr>
<th>Point number</th>
<th>Point location</th>
<th>Runtime on GPU [s]</th>
<th>Number of hits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Early in the STL-file</td>
<td>13,882210</td>
<td>12570</td>
</tr>
<tr>
<td>2</td>
<td>Late in the STL-file</td>
<td>18,241510</td>
<td>12441</td>
</tr>
<tr>
<td>3</td>
<td>Late in the STL-file</td>
<td>25,309790</td>
<td>9633</td>
</tr>
<tr>
<td>4</td>
<td>Above the square</td>
<td>41,719588</td>
<td>0</td>
</tr>
</tbody>
</table>

4.3.2.3 Time of year

The fourth scenario compares the same points as in the 4.3.2.2 in different times of the year to check the impact of different day lengths on the results. The tests have been performed with the four points of the Location of Point test in the same dataset. The tests have been done during the 20 and 40 days around the summer and winter solstice (21.06. and 21.12.) with an interval of 10 minutes. The results show, that the runtime is linear to the number of days at the same time of the year.

Table 5: Time of Year comparison (GPU) (own work)

<table>
<thead>
<tr>
<th>Start date</th>
<th>Days before and after</th>
<th>Runtime on GPU [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.12.2010</td>
<td>10</td>
<td>02,368458</td>
</tr>
<tr>
<td>21.12.2010</td>
<td>20</td>
<td>04,188777</td>
</tr>
<tr>
<td>21.06.2010</td>
<td>10</td>
<td>10,536986</td>
</tr>
<tr>
<td>21.06.2010</td>
<td>20</td>
<td>20,911863</td>
</tr>
</tbody>
</table>

4.3.2.4 Intervals

The aim of the fifth analysis is to analyze the impact of the length of intervals on the analysis with the same setup as the two-previous analysis and a length of 40 days each. Due to the different interval lengths, different levels of detail are visible (Figure 15). Smaller gaps appear in the visualization of the sunshine as shorter the interval is. The aim of this test was to show the dependency of the runtime on the interval length. At the same time of the year, the runtime is linear to the interval length.

Table 6: Interval Comparison (GPU) (own work)

<table>
<thead>
<tr>
<th>Start date</th>
<th>Interval length</th>
<th>Runtime on GPU [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.12.2010</td>
<td>10 minutes</td>
<td>04,188777</td>
</tr>
<tr>
<td>1.12.2010</td>
<td>20 minutes</td>
<td>02,075377</td>
</tr>
<tr>
<td>1.12.2010</td>
<td>40 minutes</td>
<td>01,058703</td>
</tr>
<tr>
<td>1.6.2010</td>
<td>10 minutes</td>
<td>20,911863</td>
</tr>
<tr>
<td>1.6.2010</td>
<td>20 minutes</td>
<td>10,454932</td>
</tr>
<tr>
<td>1.6.2010</td>
<td>40 minutes</td>
<td>05,231468</td>
</tr>
</tbody>
</table>
Figure 15: The two diagrams represent different time intervals from 1.6.2010 for 40 days. Left: 10 minutes interval; Right: 40 minutes interval (own work)

4.3.2.5 Dataset size
The sixth scenario compares different levels of dataset size. Calculation times between the block-level-“Glaciärvägen”-dataset are compared with the dataset in a suburb and a city level. All tests have been performed on the GPU and the analysis has been done for a period of 20 days. The triangles, that are hit during the day are after the first 30% of all triangles in the City and Suburb data. The aim of this test is to check if the calculations can also be performed on a bigger test area and if the change in the time of year has the same significant impact on a smaller test area.

Table 7: Test area size comparison (own work)

<table>
<thead>
<tr>
<th>Start date</th>
<th>Interval length</th>
<th>Glaciärvägen [s]</th>
<th>Suburb [s]</th>
<th>City [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.12.2010</td>
<td>60 minutes</td>
<td>0.296868</td>
<td>25,970127</td>
<td>209,729889</td>
</tr>
<tr>
<td>11.12.2010</td>
<td>30 minutes</td>
<td>0.625149</td>
<td>74,925857</td>
<td>372,932181</td>
</tr>
<tr>
<td>11.06.2010</td>
<td>60 minutes</td>
<td>1.296923</td>
<td>179,728308</td>
<td>909,795917</td>
</tr>
<tr>
<td>11.06.2010</td>
<td>30 minutes</td>
<td>2.593870</td>
<td>339,971201</td>
<td>1742,983331</td>
</tr>
</tbody>
</table>
5 Discussion

5.1 Visualization
The visualization has been done by following the pre-defined UCD process. An evaluation with a professional and two laymen has been done. A few explanations were necessary, which were then implemented as labels in the software. All in all, the visualization has been evaluated as sufficient, which does not mean, that there is no space for more enhancements. One suggestion which came up after realizing the project was to realize a function which makes it possible to animate the change of the rays during a day from sunrise to sunset.

5.2 Computation
Even though it is a very naive algorithm as it does not take care for geometric relationships, the Möller-Trumbore algorithm has been a good choice. As it is a very performant algorithm, easy to implement and parallelizable. One disadvantage is that many more cases are checked with the algorithm than necessary. It is for example also checked from which side the triangle has been hit. As those calculations are a precondition for the final check they are unavoidable. Besides parallelizing other tasks, it could be checked, as mentioned before, if an alternative to the Möller-Trumbore algorithm is more feasible for the ray-triangle intersection.

The realization of the computational concept has shown that it is good to modularize different tasks and not to do everything in one step. Data that has been produced at an earlier stage can easily be used for comparisons at a later stage. In general, the results of the calculations show the expected and proof that geometric calculations need much less time when processed on GPUs in low-level languages than on CPUs in high-level languages. The results show, that it can be useful to source out calculations even though writing the code can be time-consuming. It is inevitable to prepare the data in the high-level language, so that it can be used in a low-level language. Python for example casts its data in the most suitable data format and allocates memory automatically. CUDA C in contrast needs much more supervision, as memory has to be pre-allocated and datatypes have to be predefined. One advantage using a wrapper such as PyCUDA is that the user does not have to care about garbage collection, as an automatic memory management is nowadays implemented in most high-level programming languages. If it is worth the effort to transfer code to another platform differs from case to case. In this case for example it has been useful, as the calculations were highly parallelizable. For other tasks, there might be no benefit from transferring data to another platform.

5.3 Comparison of Results
The speedup of factor 40 is a good result and shows that, even with a not perfectly optimized algorithm, the outsourcing of parallelizable calculations is worth the time-consuming code writing. The speedup has two different factors which have not been individually examined. The first one is of course the platform. The second one is the language. Running the Möller-Trumbore algorithm in C on the CPU would very likely have been much faster too than in Python, but this speedup cannot be quantified with the software written in this thesis.

The third test shows that the location of the point in the STL-file plays a big role. Point 1 and 2 have approximately the same number of hits (about 1% difference), but the runtime differs by 30%. The point above the main square is an even better example, that the location of the point has a huge impact on the runtime of the whole algorithm. One solution, to improve this could be a subdivision into different areas of the whole mesh. That could be realized with a kd-search tree or simple tests, that would check if triangles are between the specific point and the sun or if triangles are below the point.
The fourth test shows that the calculations are dependent on the time of the year, as the times of the sunset and sunrise differ a lot. In the example of Gävle, the day has a length at the summer solstice of about 19:08 hours and at the winter solstice of about 5:38 hours. The length of a day during winter is just around a third as during summer and the calculations take therefore about a fourth of the time.

The fifth test shows, that the runtime of an interval change is not dependent on the time of year. A 10-minute interval takes double the amount of time compared to a 20-minute interval, no matter if the test is performed in the beginning of June or in the beginning of December. The same applies to the comparison of 20 and 40-minute intervals in June and December: the test reveals a clear linear performance gain.

The results of the last test scenario indicate that there is a relationship between intervals, times of the year and the number of the triangles. No matter how many triangles the test scenario has, the calculations take about 4.5 times longer in summer than in winter. Minor deviations can occur due to a different order of the triangles in the mesh file. It is also visible that the calculation time correlates with the number of triangles in the test. Six times more triangles also mean that the whole process takes six times longer. If changes in the interval time are made, the change in the calculation times also correlates with those. Dividing the interval by two means double the calculation time. Those effects are visible in all datasets.
6 Conclusion and Future Work

This thesis aims to improve ray casting for solar irradiance calculations in urban environments by parallelization. This study compares one method of ray casting on two different hardware and software platforms and draws conclusions on suitability in terms of performance and efficiency. The final product is a software which embeds the achieved conclusions. This chapter summarizes the outcomes of the thesis in two sections. Section 6.1 draws conclusions of the study and Section 0 gives suggestions on future work in this topic.

6.1 Conclusions

Graphical representation gives a better understanding of complex problems. This thesis designed a software which is accessible for laymen and professionals to easy understand the matter. The evaluation results of the software design showed that the UCD process was the right one to use. The respect of the temporal and spatial resolution were successfully implemented in the software. Final evaluations with both target groups, laymen and professionals prove that they both intuitively understood what was displayed on the screen and drew the right conclusions the whole process was a success.

The implementation of the sketch in a simple Python-application has been successful. The demands such as the import of city-environment models and point data, as well as a limitation on Python have been defined beforehand and have all been satisfactory implemented. The second software related aim was the performance evaluation of the algorithm on two different processor platforms. A wide range of different tests which evaluated a lot of different parameters, such as the size of the city model or the time of the year, have been made and the evaluation has provided a variety of different results. The decision to choose a quite naïve algorithm has been a good decision as it was rather easy to implement and produces comparable results. The results imply that a parallelized calculation on a much weaker hardware is a good choice if the calculations do not have a high complexity.

Generally, it can be said, that the written software is clearly an enhancement of the software approach that Seipel, Lingfors and Widén gave in their paper. The higher level of usability realized by a higher degree of interactivity gives the user a better understanding of changing irradiance in urban built environments. But of course, the results and the visualization can just be as good as the 3D-model that was used. The performance gain by using PyCUDA can have implications on future software projects. This thesis shows that, if calculations are parallelizable, it is relatively easy to source calculations out of a high-level programming language and get an enormous speed boost. Subsequently, this study can have a huge impact on future software in the field of solar irradiation calculation in urban built environments, as it gives suggestions as to how to improve the speed of calculations, enabling such software to attain a higher performance there and become more user-friendly. Why already existing software in this field did not use parallelization yet is just speculation. Reasons could be a lack of knowledge or parallelization technology has not been ready for the market at the time of development.

Besides technical aspects, this thesis can contribute to create a more sustainable habitat for men in cities by improving the planning process in urban built environments. The thesis project does not just improve the technical aspect in the planning process. It is also a good tool for community involvement as it visualizes the problems and thereby can establish a better comprehension for non-professionals. It is the same approach as Dr. John Snow chose in 1854. Visualizing the problem and then convincing people who are not familiar with the subject that chosen solution is the right one.
6.2 Future Work

This thesis is limited in some ways. The calculations for example can be more efficient. Now, just the ray casting calculations are parallelized. The algorithm that calculates the pre-conditions for the ray casting algorithm is not parallelized. By parallelizing the calculations of the exact times and dates, sun vectors and radiations, the overall runtime of the software would be shorter and gain a higher degree of usability. The gains would of course be less than the ones gained by parallelizing the ray casting calculations, but would still be appreciably.

The user interface of the developed software can also be improved in a further development. Right now, it is very plain and functional. The design is not really appealing even though it is clear what it displays. The colors could match a bit better and the design of the 3D-model could be a bit more individual as it is an out-of-the-box solution right now. The inputs could also be a bit more user friendly. In this state, they are not checking if the inputs are valid which can lead to errors if wrong values are entered.

In terms of realistic environments there are a lot more variables and data sources that could be implemented. Right now, the software calculates the radiation for perfect conditions. By taking a cloud model into consideration, the predictions would be much more realistic. Another point that is not considered, is the inclination of solar panels. At the moment, the calculations just output the theoretical power that could be generated with a solar panel with an efficiency of 100% which is perfectly aligned to the sun. That leads to two more values which should be implemented. The alignment of the panels and the efficiency of the used solar panels.

The results of this thesis can be used in different fields of science. Nowadays many scientific institutions such as the DLR German Aerospace Center or the European Organization for Nuclear Research (CERN) write their software in Python. The main reason for them to use Python is the speed in which software can be written and not the performance with which it is executed. Same applies for calculation-intense software such as most plugins for GIS software and other tools. This trend is understandable, but with the shown outsourcing, the performance of those can be significantly improved.

The outcomes of this thesis can be used for planning processes of photovoltaic systems in new and already existing urban environments. It can contribute to more autonomous energy grids in urban and suburban regions and lower the dependence on power plants that are driven by nuclear processes and fossil energy sources. This would lead to a more sustainable habitat for men and can, on a global view, make a positive impact on slowing the anthropogenic climate change down.
7 REFERENCES


https://www.w3.org/WAI/EO/2003/ucd

8 APPENDICES

8.1 APPENDIX A — CITY AND SUBURB MODEL IN A GIS ENVIRONMENT
This appendix shows the city model of Gävle and the suburb model which have been tested in 4.3.2.5. The city model is displayed in ArcScene. The model is aligned to north. The suburb of Sätra as well as the train tracks and the river can be seen clearly. For a better understanding compare with Figure 11.
The suburb model shows the suburb Sätra in Gävle, Sweden. The clear geometric structure of the suburb is visible. Park areas, as well as big streets are dimly imaginable.
8.2 Appendix B – Structural Description of the Möller-Trumbore Algorithm

This Nassi-Shneiderman diagram visualizes the structure of the Möller-Trumbore algorithm. This algorithm is used for Ray-Triangle intersection in this Thesis and source code examples for Python and PyCUDA are given in Appendix C and D.
This appendix includes the Möller-Trumbore ray-triangle algorithm that has been used in the Python implementation in the thesis. The structure of the source code is shown in 8.2.

```python
def ray_triangle_intersection_NEW(sunvector_array, points, mesh_array):
    sunshine_yesno = []
    print(len(sunvector_array))
    # loop all points that should be checked
    for point in points:
        sunshine_point = []
        # loop over all days that should be checked
        for sunvector_day in sunvector_array:
            sunshine_day = []
            # loop over all moments that should be checked
            for sunvector_moment in sunvector_day:
                hit_CPU = False
                # if the sun is risen, checks if z-vector is positive
                if (sunvector_moment[2] != None):
                    # check all triangles in the triangle-array
                    for triangle in mesh_array:
                        # the Möller-Trumbore algorithm is run
                        v1 = triangle[0]
                        v2 = triangle[1]
                        v3 = triangle[2]
                        eps = 0.0000001
                        edge1 = v2 - v1
                        edge2 = v3 - v1
                        pvec = np.cross(sunvector_moment, edge2)
                        det = edge1.dot(pvec)
                        if abs(det) < eps:
                            hit_CPU = False
                        else:
                            inv_det = 1. / det
                            tvec = point - v1
                            u = tvec.dot(pvec) * inv_det
                            if (u < 0. or u > 1.):
                                hit_CPU = False
                            else:
                                qvec = np.cross(tvec, edge1)
                                v = sunvector_moment.dot(qvec) * inv_det
                                if (v < 0. or u + v > 1.):
                                    hit_CPU = False
                                else:
                                    t = edge2.dot(qvec) * inv_det
                                    if (t > eps):
                                        hit_CPU = True
                                        # leave the loop if a ray hits one triangle
                                        if (hit_CPU):
                                            break
                            # if one triangle is hit, add a "false" to the
                            "sunshine_day"-array
                            if (hit_CPU):
                                sunshine_day.append(False)
                            else:
                                sunshine_day.append(True)
                        else:
                            sunshine_day.append(False)
                            sunshine_point.append(sunshine_day)
                            sunshine_yesno.append(sunshine_point)
    return sunshine_yesno
```

33
This appendix includes the Möller-Trumbore ray-triangle algorithm that has been used in the CUDA-C implementation in the thesis. The structure of the source code is shown in 8.2.

```python
import numpy as np
from pycuda import driver, compiler, gpuarray, tools

def GPU_intersect_NEW(sunvector_array, points, mesh_array):
    """ sunvector array is a 2D array with 1st dimension days, 2nd
dimension time during the day
    and the content of each cell is a 3D-vector (xyz) from the points
    that should be checked to the sun
    The vector from every point to the sun is the same, as the points
    are close to each other.
    points is an array of n-points that will be checked
    mesh_array is an array of triangles
    """

    mod = compiler.SourceModule(""
    __global__ void intersection(float *sunvectors, float *points, float
    *mesh, float *res_gpu)
    /*the res_gpu should have all true/false or 1/0 results in it in the
    end. it's dimensions are no_of_points*days*times_measured_during_day */
    {
        int no_of_points = res_gpu[0];
        int no_of_days = res_gpu[1];
        int no_of_moments = res_gpu[2];
        int no_of_triangles = res_gpu[3];

        for(unsigned int p = 0; p < no_of_points; p = p + 1){
            float origin[3] = {points[p*3],points[p*3+1],points[p*3+2]};
            for (unsigned int d = 0; d < no_of_days; d = d + 1){
                for (unsigned int m = 0; m < no_of_moments; m = m + 1){
                    float dir[3];
                    dir[0] = sunvectors[d*no_of_moments*3+m*3];
                    dir[1] = sunvectors[d*no_of_moments*3+m*3+1];
                    dir[2] = sunvectors[d*no_of_moments*3+m*3+2];
                    if(isnan(dir[0])){
                        res_gpu[p*no_of_days*no_of_moments+d*no_of_moments+m] = 0;
                    }else{
                        /*the sun is shining, check here if it hits one of
                        the triangle*/
                        unsigned int t = 0;
                        bool end_loop = false;
                        for (t = 0; t < no_of_triangles; t = t + 1){
                            float t0[3] = {mesh[t*9], mesh[t*9+1],
                            mesh[t*9+2]};
                            float t1[3] = {mesh[t*9+3], mesh[t*9+4],
                            mesh[t*9+5]};
                            float t2[3] = {mesh[t*9+6], mesh[t*9+7],
                            mesh[t*9+8]};
                            float orig[3];
                            orig[0] = origin[0];
                            orig[1] = origin[1];
                            orig[2] = origin[2];
                        }
                    }
                }
            }
        }
    }
    ""
```

8.4 APPENDIX D – MÖLLER-TRUMBORE-ALGORITHM AS A PYCUDA FUNCTION
float epsilon = 0.000001f;
float v;
float u;
float qvec[3];
float res[3];

/*t1 becomes t0t1*/
t1[0] = t1[0] - t0[0];
t1[1] = t1[1] - t0[1];
/*t2 becomes t0t2*/
t2[0] = t2[0] - t0[0];
t2[1] = t2[1] - t0[1];
/*res becomes pvec = dir X t0t2*/
float det = t1[0] * res[0] + t1[1] * res[1] +
t1[2] * res[2];
if(det<epsilon && det>-epsilon){
 /*ray lies in plane of triangle or ray is parallel to plane of triangle*/
 end_loop = false;
}else{
 float invDet = 1.0f/ det;
 /*orig becomes tvec*/
 orig[0] = orig[0] - t0[0];
 orig[1] = orig[1] - t0[1];
 u = (orig[0] * res[0] + orig[1] * res[1] +
 orig[2] * res[2]) * invDet;
 if (u < 0 || u > 1){
 /*not culling*/
 end_loop = false;
}else{
 /*t1 becomes qvec*/
t1[1];
t1[2];
t1[0];
 v = (dir[0] * qvec[0] + dir[1] *
 if (v < 0 || u + v > 1){
 /*The intersection lies outside of the triangle 01*/
 end_loop = false;
 }else{
 float t;
 t = (t2[0] * qvec[0] + t2[1] *
 if (t > epsilon){
 end_loop = true;
 }
 }
if (end_loop){
  break;
}
/*for triangles*/
if (end_loop == true){
  res_gpu[p*no_of_days*no_of_moments+d*no_of_moments+m] = 0;
}else{
  res_gpu[p*no_of_days*no_of_moments+d*no_of_moments+m] = 1;
  /*if nan else*/
  /*for moments*/
  /*for days*/
  /*for points*/
}

no_of_points = len(points)
no_of_days = len(sunvector_array)
no_of_moments = len(sunvector_array[0])
no_of_triangles = len(mesh_array)
res = np.zeros([no_of_points, no_of_days, no_of_moments]).astype(np.float32)
res[0][0][0] = no_of_points
res[0][0][1] = no_of_days
res[0][0][2] = no_of_moments
res[0][0][3] = no_of_triangles

sunvectors_gpu = gpuarray.to_gpu(sunvector_array)
points_gpu = gpuarray.to_gpu(points)
mesh_gpu = gpuarray.to_gpu(mesh_array)
res_gpu = gpuarray.to_gpu(res)

func = mod.get_function("intersection")

# call the kernel on the card
func(
    # inputs
    sunvectors_gpu, points_gpu, mesh_gpu,
    # output
    res_gpu,
    block = (4, 4, 1),
)
result = res_gpu.get()
sunvectors_gpu.gpudata.free()
points_gpu.gpudata.free()
mesh_gpu.gpudata.free()
res_gpu.gpudata.free()

return result
8.5 APPENDIX E — CALCULATION OF SOLAR IRRADIANCE WITH THE DEVELOPED SOFTWARE

This step-for-step tutorial explains how solar irradiance calculations can be performed with the developed software. As the design of the software is user-focused and the software is centered around performance and not on a wide range of tools, it is not very complicated to use the software.

Run simulation:

1. Start the software and choose an STL-file (*.stl).

2. Click Open (in the language of your operating system) (1)

3. After choosing an STL-file, open an Origin/Point-File which contains the origin of the local coordinate system and all files to be checked.

4. Click Open (in the language of your operating system) (2)
5. A new window opens. On the left side, different tabs (3) with the irradiation on different points, as well as the possible sunshine are available. On the right side, the loaded 3D-model and diagram preferences are available.

6. By clicking on “Update Graphs” after the day, month, year, interval and number of days (4) have been changed, a new calculation is performed. The diagrams change.

7. After changing day, month, year, hour and minute in the second preferences menu and clicking on “Update 3D” (5), the rays in the 3D-model change to the selected date and time. Clicking into the diagram (6) of any point or the possible sunshine tab has the same effect.