HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
INSTITUT FÜR INFORMATIK

Extending and Accelerating a GPU Ray-Tracing Algorithm for Photon Simulation in Beamlines

Bachelor Thesis

Thesis submitted for the degree
Bachelor of Science (B. Sc.)

submitted by: Enrico Philip Ahlers
born on: 11.07.1998
born in: Berlin

Reviewer: Prof. Dr. Henning Meyerhenke
Prof. Dr.-Ing. Peter Eisert

submitted on: .... defense on: ....

To efficiently design beamlines for an electron storage ring, a physically correct ray tracing simulation is needed. This work improves and extends the ray tracing software RAY-X, which is being developed by Helmholtz-Zentrum Berlin. The calculation of the intersection point between rays and the optical elements in a beamline is optimized, improving performance by up to 37%. Furthermore, an extension which enables dynamic ordering of optical elements was developed, enabling a more realistic simulation. Lastly, the initialization overhead is reduced and the GPU usage is increased by tracing multiple beamlines simultaneously. Depending on the beamline configuration this can improve the performance by multiple magnitudes.

Contents

1 Introduction **4**
- 1.1 Goals and Challenges . . . 4

2 Fundamentals **6**
- 2.1 RAY-UI and RAY-X . . . 6
- 2.2 Synchrotron Radiation and Beamlines . . . 6
- 2.3 GPU Computing . . . 6
  - 2.3.1 Nvidia Ampere GPU Architecture . . . 7
  - 2.3.2 Nvidia RTX 3090 . . . 9
- 2.4 Efficient Parallelization for GPUs . . . 9
- 2.5 VULKAN . . . 10
  - 2.5.1 Compute Shader . . . 10
  - 2.5.2 Buffers . . . 11
- 2.6 Ray Tracing . . . 11
- 2.7 Nvidia Nsight Graphics . . . 12
- 2.8 The current state of RAY-X . . . 14

3 New Approaches **17**
- 3.1 Eliminating Conditional Branching in the intersectionPoint-Method . . . 17
  - 3.1.1 Algebraic Method . . . 19
- 3.2 Dynamic Ordering of Optical Elements . . . 22
- 3.3 Multiple Beamlines . . . 27

4 Implementation Details **32**

5 Experiments **34**
- 5.1 Identifying Bottlenecks . . . 34
- 5.2 Performance of the Improved intersectionPoint-Method . . . 35
- 5.3 Performance of Dynamic Ordering of Optical Elements . . . 38
- 5.4 Performance of Tracing Multiple Beamlines . . . 40

6 Discussion **42**

1 Introduction

Simulating beamlines of electron storage rings is a fundamental requirement when designing beamlines. Since only a limited amount of physical experiments can be done within a time frame, there is no room for error. With physically correct simulation of beamlines, the amount of errors can be reduced drastically. For this, the software RAY [4] developed by Helmholtz-Zentrum Berlin is an industry standard. It uses a highly realistic ray tracing algorithm to simulate photons in beamlines. To optimize the performance of the ray simulation the project RAY-X has been started, which makes use of state-of-the-art libraries and APIs like VULKAN. This enables the use of parallel computation using Graphics Processing Units (GPUs) to accelerate the ray tracing process. For this, the ray tracing code used in RAY, which is written in FORTRAN, has been ported to GLSL, a shader language that enables offloading calculations to the GPU. However, to fully exhaust the GPU's parallel computing abilities, the algorithm must be optimized for parallel execution. The original FORTRAN code relies on CPUs for execution, and thus does not include parallel processing optimizations.

1.1 Goals and Challenges

The goal of this work is to optimize the tracing procedure to use the GPU more efficiently. Furthermore, two new features will be developed. The implemented features and optimizations are then tested and compared.

Eliminating Conditional Branching in the Computation of Intersection Points Finding the intersection point between a ray and an object is one of the fundamental parts of a ray tracing procedure. In RAY, quadrics are used to represent optical elements. Compared to ray-triangle intersections, which are heavily used in ray tracing applications, the method for finding an intersection point between a ray and a quadric is more complicated. Furthermore, the procedure contains if-conditions, which leads to branching. Branching can result in a significant reduction in performance when using a GPU. To counter this behaviour, this work removes branching from the intersectionPoint-method in section 3.1 to improve the ray tracing performance.

Using this approach, an increase in performance by up to 37% is achieved.

Dynamic Ordering of Optical Elements In RAY, beamlines are represented in a static order. Rays that pass through the beamlines hit the optical elements sequentially in a pre-defined order. While this procedure is sufficient for most beamlines, it limits the physical accuracy of the ray tracing simulation. Rays can only hit each element once and rays that do not pass through all optical elements are ignored. In a typical beamline this does not hurt accuracy much, as all rays follow a similar path and should hit every element only once. However, when designing a beamline errors can cause rays to leak through gaps and produce interference on the generated footprint. This behaviour can go undetected when tracing the beamline sequentially. To conquer this problem, a ray tracing method with dynamic ordering of optical elements is developed in section 3.2.

The implemented algorithm enables more realistic beamline simulation by sacrificing performance depending on the complexity of the beamline.

Tracing Multiple Beamlines in Parallel In RAY-X only one beamline is traced at the same time. Because of the time spent initializing VULKAN this can result in a significant overhead, especially when tracing beamlines with a low number of rays. Moreover, the GPU is capable of heavy parallel computation, but has less than ideal sequential performance. When tracing beamlines with small numbers of rays, the GPU might not be fully utilized, resulting in a bottleneck. Tracing multiple beamlines in parallel, which is detailed in section 3.3, aims to improve the GPU utilization. Furthermore, the time spent initializing VULKAN is minimized.

Especially when tracing thousands of beamlines with a low number of rays each, the removed initialization overhead results in multiple magnitudes better performance. Moreover, the GPU utilization for practically used beamlines is increased by up to 104%.

2 Fundamentals

2.1 RAY-UI and RAY-X

RAY-UI is a ray tracing software developed by Helmholtz-Zentrum Berlin, which builds on the command-line-based software RAY. The development of RAY started in 1984 at BESSY with the intention to calculate VUV- (Vacuum Ultraviolet) and soft X-ray optical schemes [4].

RAY enables the user to simulate photons in the beamlines of an electron storage ring. This information can be used to design new beamlines for electron storage rings like BESSY II. For this, the software takes advantage of multi-core processing to speed up the simulations.

RAY-X is a new project by Helmholtz-Zentrum Berlin with the aim to make use of state-of-the-art software APIs and the multi-processing capabilities of GPUs to speed up the tracing process even further. The graphics and computing API VULKAN [1, 2] by Khronos Group is used to develop universal compute shaders, which can be run on a variety of GPUs and CPUs, making RAY-X not only faster, but also more versatile.

2.2 Synchrotron Radiation and Beamlines

Charged particles, usually electrons or ions, that are moving at speeds close to the speed of light, emit synchrotron radiation when their path is altered [5]. The spectrum of this radiation is broadband from microwave to x-ray spectral regions [6].

A beamline is a part of an electron storage ring consisting of multiple optical elements. Synchrotron radiation from a source passes through the beamline and its optical elements and may be reflected, refracted, focused and filtered until it arrives on an image plane, e.g. a CCD camera sensor [7]. The optical elements used depend on the experiment and include, but are not limited to planar mirrors, gratings, slits and reflection zone plates.

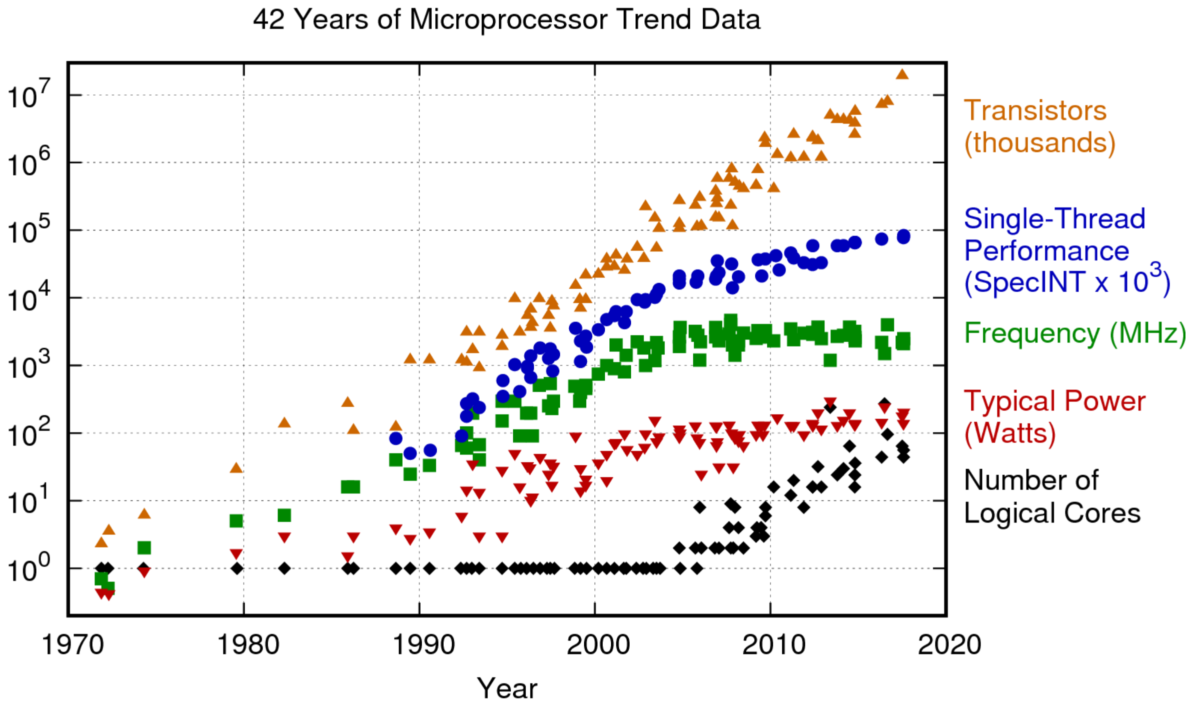
Applications for beamlines include producing highly magnified images of living cells, creating diffraction patterns of crystallized proteins to determine their structure or investigating meteorites and archaeological finds [8, 7].

2.3 GPU Computing

In recent years graphics processing units (GPUs) have become increasingly more popular. For a long time Central Processing Units (CPUs) have been the main tool for computation. This is mainly because CPUs are very versatile and offer exceptional sequential performance. As seen in Figure 1, the single thread performance has been stagnating in the last 15 years. This is a problem, since the demand for processing power increased over the years. To counter the stagnation, parallel computing has become more common to make use of more than one CPU at a time.

Compared to a modern CPU, which can have from a single core to more than 64 cores [9], a GPU consists of thousands of cores [15]. While each core is slower and less versatile than a CPU core, the GPU makes use of the great number of cores to efficiently handle parallelized workloads.

Specific Applications For applications like ray tracing and neural network interference, GPUs can be multiple times faster than CPUs. Moreover, applications like video games have become much more popular [10], causing companies that develop GPUs to grow heavily [11]. Because of this, more funds are available for the development of GPUs, causing them to become even faster. Furthermore, machine vision and artificial intelligence are becoming more integrated into daily life, powering applications like autonomous cars, medical research, social networks, video and photo manipulation and many more [12]. All of these benefit greatly from parallel processing done by GPUs.


42 Years of Microprocessor Trend Data

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Figure 1: Microprocessor Trend Data, from [13]

2.3.1 Nvidia Ampere GPU Architecture

While RAY-X is designed to be compatible with many GPUs and CPUs, the graphics card used for the development and testing in this work, an Nvidia RTX 3090, is based on the Nvidia Ampere GPU Architecture.

An Nvidia Ampere GPU consists of multiple GPU processing clusters (GPCs), texture processing clusters (TPCs), streaming multiprocessors (SMs) and memory controllers, as well as RT (ray tracing) cores and Tensor Cores [15].

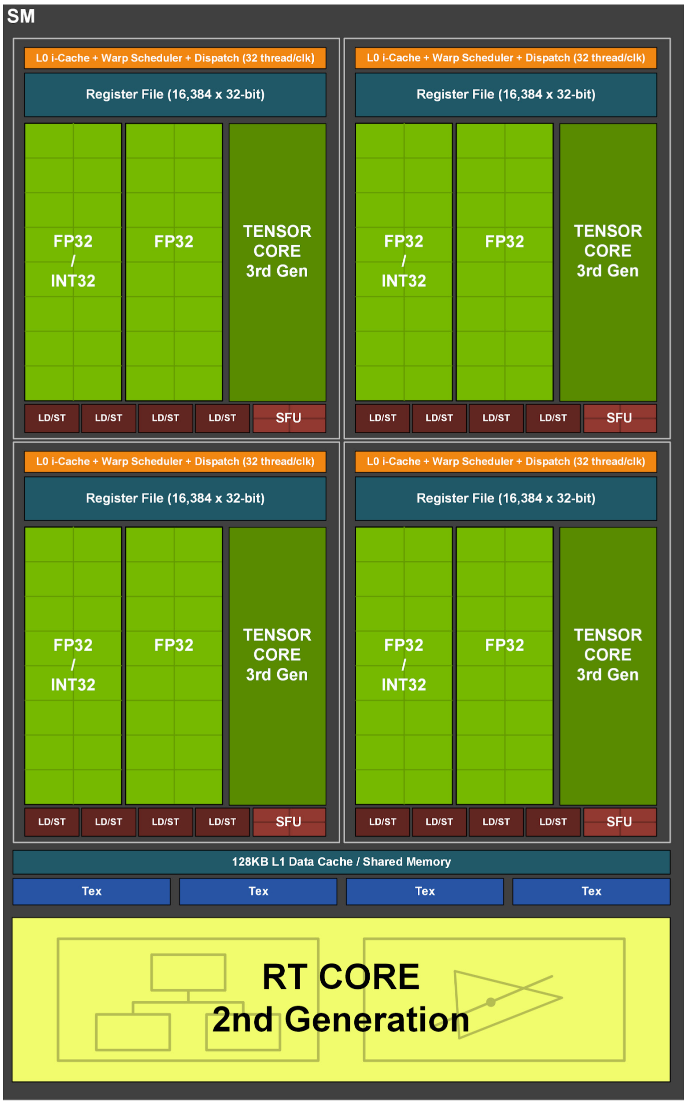

SM
L0 i-Cache + Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
FP32 / INT32
FP32
TENSOR CORE 3rd Gen
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
RT CORE 2nd Generation

Figure 2: GA10x Streaming Multiprocessor (SM), from [15]

As visualized in Figure 2, each SM contains 128 Compute Unified Device Architecture (CUDA) Cores, multiple Tensor Cores, Texture Units, a Ray Tracing Core and L1 shared Memory. Each SM is partitioned into four processing blocks, which each have one warp scheduler and 32 CUDA cores. RT cores accelerate ray/triangle intersection, resulting in 58 RT TFLOPS of processing power. While this is particularly useful for real-time ray tracing applications in video games or animation, the RT Cores can not be used in RAY-X. This is because triangles are not used to represent optical elements. While triangles are sufficient for video games and animation, beamline simulation requires curved surfaces, which can only be approximated using triangles. The resulting precision would be too low for the experiments done using RAY-X. This means that the calculations need to be done using CUDA cores.

CUDA cores are Nvidias GPU equivalent of CPU cores [16]. In contrast to RT or

Tensor cores, CUDA cores are responsible for fast general purpose calculations.

Tensor cores are execution units designed to accelerate tensor/matrix operations, which are heavily used in deep learning applications [15].

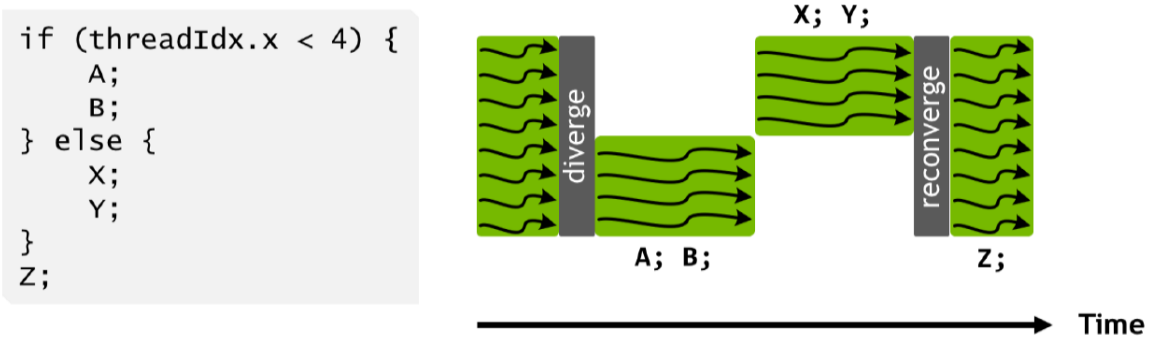
2.3.2 Nvidia RTX 3090

The GPU used to perform the experiments in this thesis is an Nvidia RTX 3090 graphics card, which uses an Nvidia GA102 GPU. The graphics card has 82 SMs, 10496 CUDA Cores and 24 GB of GDDR6X Memory [15]. The RTX 3090 has a 384-bit memory interface and GDDR6X runs at a data rate of 19.5 Gbps, resulting in a memory bandwidth of 936 GB/s. The base and boost clock of the GPU are 1395 MHz and 1695 MHz respectively. It is connected to the mainboard using 16 PCIe4 lanes, enabling a bandwidth of 31.51 GB/s [46]. While the RTX 3090 has a peak of 35.6 FP32 TFLOPS, RAY-X relies on FP64 calculations. The FP64 TFLOP rate is 1/64th the TFLOP rate of FP32 operations, resulting in a peak of 556 GFLOPS [17]. Furthermore, each of the 82 SMs has 128 KB of L1 cache.

The Nvidia RTX 3090 is a consumer GPU, while enterprise GPUs are more popular in a professional environment. A consumer GPU was chosen because RAY-X will regularly be run on PCs instead of servers. Furthermore, the optimization for consumer GPUs enables users to run the software offline without the need to connect to a server. Moreover, enterprise GPUs can still be used if necessary.

2.4 Efficient Parallelization for GPUs

GPUs are based on a Single Instruction, Multiple Threads (SIMT) system. A SIMT system corresponds to a tiled Single Instruction, Multiple Data (SIMD) architecture, in which every SIMD processor emulates multiple threads using masking [20]. Even though SIMT devices have a high number of threads, the threads are grouped into blocks. As the GA102 GPUs SMs have 4 warps with 32 cores each, the instruction is run on the 32 cores simultaneously.



```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

Figure 3: Thread divergence in GPUs, from [19]

Every core executes the same instruction with different data. This can be a problem when using diverging control flow, which happens when two cores in the same warp have to execute different instructions as a result of data-dependent conditional branches, e.g. if-conditions. This is realised using masking, because instructions are pipelined. Unlike CPU cores, the instructions are issued in order and there is no branch prediction or speculative execution [18]. Some threads in a single warp execute the instruction of one path, while the others evaluate the other path. While one path is executed, the other threads are disabled, as seen in Figure 3. It is also possible that both paths are evaluated before the condition is evaluated. This is especially wasteful since results that are computed are not used.

Branching can result in significantly longer execution times, as the parallel computation capabilities of the GPU are not fully used. When developing algorithms for the GPU, this property needs to be respected and ignoring it can result in degradation of performance. Full efficiency is achieved when all 32 threads have the same execution path. Branch divergence however occurs only within a single warp.

Memory Access The SIMD processors used to emulate SIMT are designed to use data from a single cache line. If threads in a SIMD processor need to access different cache lines, the performance drops. This is the result of divergent memory accesses, which require multiple memory cycles to resolve the memory access. If however all threads in a SIMD core access the same cache lines, the memory access can be united, which improves the performance.

2.5 VULKAN

VULKAN is a state-of-the-art graphics and computation API developed by Khronos Group. Sometimes referred to as the "next generation OpenGL initiative, VULKAN aims to offer less overhead and higher performance than OpenGL [22]. It enables the user to efficiently use GPUs for computation while not being restricted to specific GPU models like the CUDA API, which relies on GPUs made by Nvidia. In contrast to OpenGL, VULKAN offers explicit low level control. The developer has to manage things like memory, resource updates, batching and scheduling [23]. Furthermore, VULKAN is designed to take advantage of multi-core systems, which improves the performance.

Code written using the VULKAN-API can easily be executed on a variety of CPU and GPU combinations while only needing a minimum of system specific adjustments. This makes the final product very easy to be utilized by end-users, as application specific workstations or servers are not needed.

2.5.1 Compute Shader

VULKAN supports different shader stages natively, including, but not limited to a Vertex Shader, a Geometry Shader, a Fragment Shader and a Compute Shader [24]. While shaders like Vertex Shaders and Fragment Shaders are commonly used in applications with a graphical output, RAY-X only depends on calculations. This is why only a Compute Shader, which is responsible for the computation, is used.

Compute shaders are mainly used for calculations not directly related to drawing triangles and pixels. While other shaders like a vertex shader might be executed multiple times in a second (e.g. for real time rendering), the Compute Shader in this work is only executed once per beamline simulation. All calculations like the computation of the intersection between a ray and a optical element are executed in a single shader call.

The shader is written using the OpenGL Shading Language (GLSL), a high level shading language based on the C programming language [3]. The instructions in the compute shader are executed for every ray. This means that a thread for every ray is created, which is then executed by a core on the GPU.

2.5.2 Buffers

Buffers are objects used by VULKAN to store the data needed by the cores of the GPU. The GA102 GPU has different kinds of memory, which each have their own unique properties. RAY-X uses a staging buffer, which serves the purpose of transferring data from the system RAM (random access memory) to the GPU memory. For this, host-visible memory is needed, which is GPU memory that can be accessed by the CPU. One disadvantage is the size, as there is only 128 MB of host visible memory, which is visible in the left part of the GPU memory in Figure 4. When using the more optimal device local memory shown on the right side of the GPU memory, the full speed and memory size can be used, i.e. 24 GB for the RTX 3090 [25]. This concludes that it is important to choose the right kind of memory for each task.

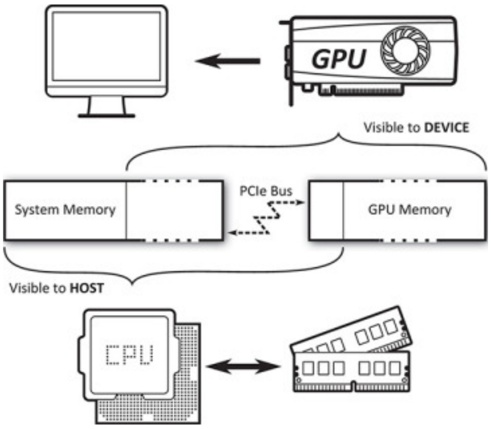


Figure 4: Host-Visible and Device-Local memory, from [27]

2.6 Ray Tracing

Ray Tracing is a technique to compute realistic paths of light. It is mostly used in computer graphics as a lighting technique because it simulates how light in the real world could behave. This results in more realistic lighting than other lighting techniques can achieve.

Ray Tracing works by simulating each ray, which generally has a position and a direction in 3-dimensional space. In computer graphics, 3D objects are typically represented using polygons, which usually are triangles. The intersection point between the polygon and the line represented by the position and direction of the ray is calculated. Then, the position and direction of the ray are updated depending on the material of the object. The newly calculated parameters are used for the next iteration. The path of the ray is calculated by iterating this procedure.

While this method is simple, the intersection point between the ray and every polygon in the scene has to be calculated. Furthermore, it has to be determined which polygon is hit first. This concludes a single iteration, which can already take a long time. The procedure is repeated until a maximum number of iterations is done or a light source is hit. While the basic ray tracing algorithm is simple, it takes a lot of processing power to compute all the iterations for each ray. The algorithm however is easy to parallelize, which makes it a perfect use case for GPU computation.

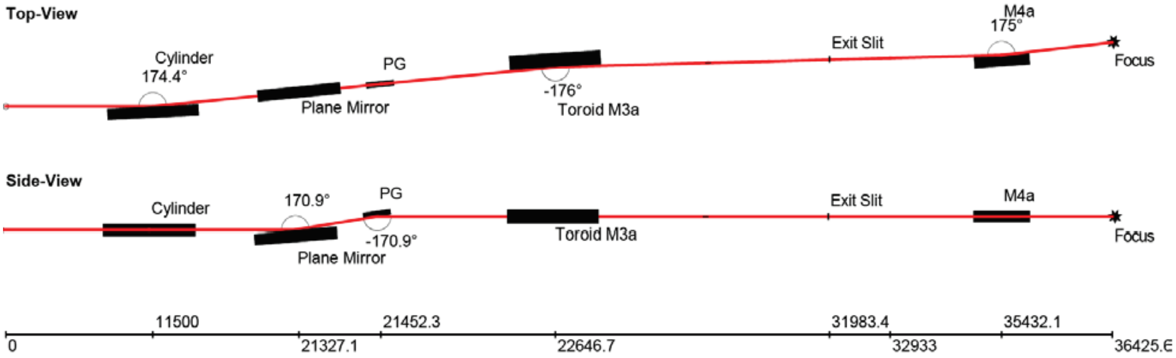


Figure 5: Visualization of a beamline, from [31]

Lighting Techniques There are multiple techniques for ray tracing, e.g. Ray Casting, Forward Ray Tracing, Backward Ray Tracing and Path Tracing. While casting rays through each pixel of the camera is often used in computer graphics, RAY-X uses backward ray tracing [29] where rays are sent from the light source. This is visualized in Figure 5. The rays are generated on the left side and sent through the beamline, where they pass through multiple optical elements. They ultimately arrive at the image plane (referred to as ”focus” in the visualization) where the intersection between the ray and the image plane is used to generate a footprint [31]. Figure 6 shows a series of footprints generated by RAY. Depending on the position of the image plane, the rays are focused differently [30].

2.7 Nvidia Nsight Graphics

Nvidia Nsight Graphics is a performance analysis tool used to determine GPU utilization. It enables the user to debug, profile and export frames built with Direct3D, VULKAN,

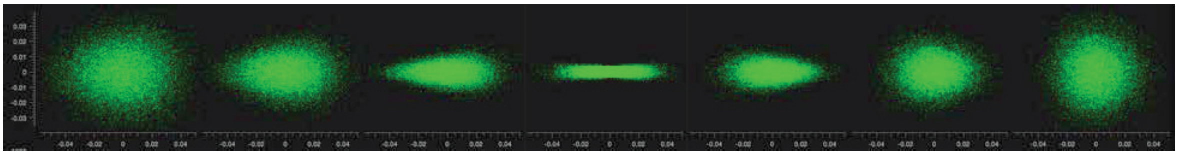
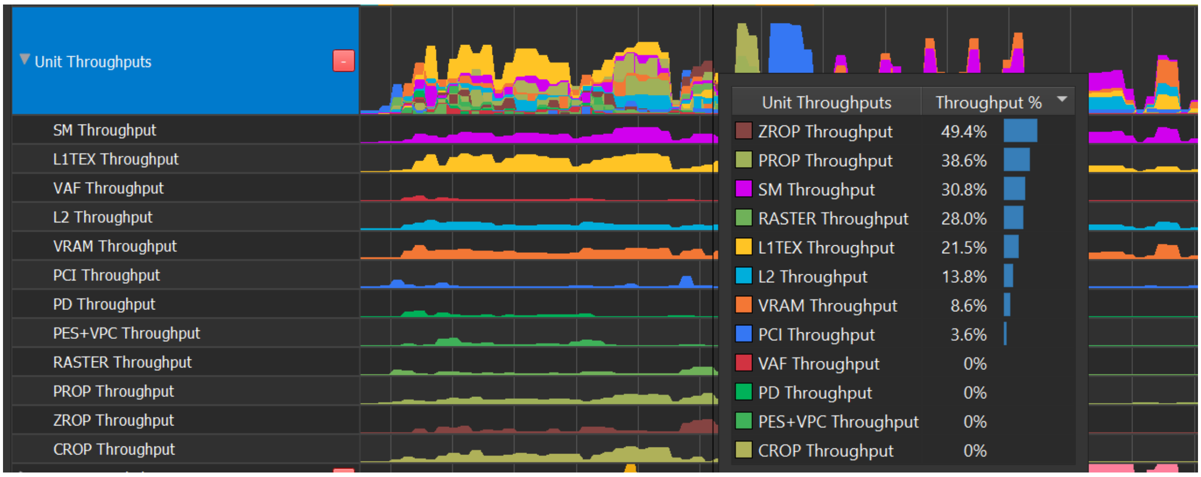


Figure 6: Footprints generated in RAY, from [30]

OpenGL and others [33]. It assists in finding bottlenecks to make software use the hardware more efficiently.

GPU Trace Tracing a VULKAN compute shader call is realized using the GPU Trace functionality of NVIDIA Nsight. It is a graphics profiler which uses periodic sampling to gather metrics associated with different GPU hardware units [34].



Unit Throughputs	Throughput %
🟫 ZROP Throughput	49.4%
🟨 PROP Throughput	38.6%
🟪 SM Throughput	30.8%
🟩 RASTER Throughput	28.0%
🟨 L1TEX Throughput	21.5%
🟦 L2 Throughput	13.8%
🟧 VRAM Throughput	8.6%
🟦 PCI Throughput	3.6%
🟥 VAF Throughput	0%
🟩 PD Throughput	0%
🟩 PES+VPC Throughput	0%
🟨 CROP Throughput	0%

Figure 7: Unit Throughputs, from [34]

Figure 7 shows the %-of-max-throughput of different hardware units in the GPU. The SM, L1TEX, L2, VRAM and PCI throughput are of special interest when profiling RAY-X, while other metrics e.g. PD are not as important. As shown in table 1, the primitive distributor (PD) is responsible for sending triangles to the vertex shader. Because RAY-X does not use a vertex shader, this metric can be ignored.

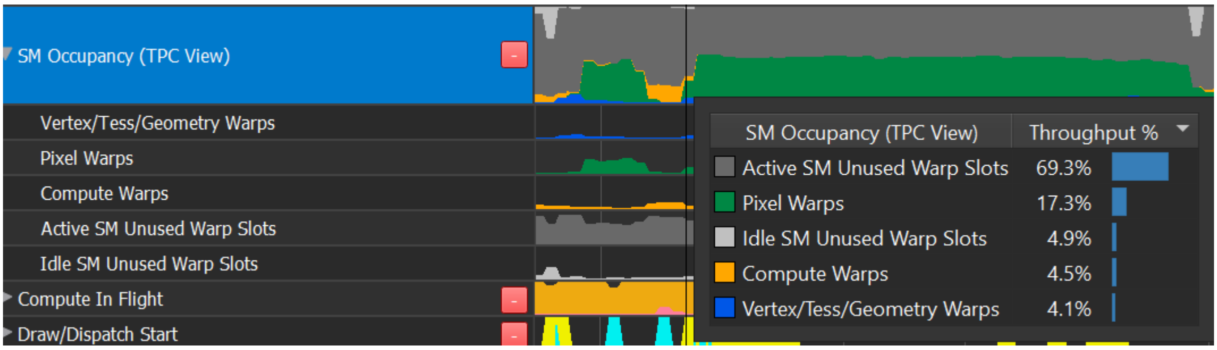
When profiling RAY-X, the metric SM occupancy is especially important. Because code optimized for a CPU has been ported to the GPU, conditional branching is a major issue in RAY-X, causing threads to stall. This can be monitored using the metric shown in Figure 8.

Each SM has a limited number of warp slots. For efficient usage of the GPU, as many SMs as possible should be occupied by Compute Warps. However, experiments will show that when tracing a beamline most of the SMs are occupied by:

1. Idle SM unused warp slots: The unused warp slot is a member of an idle SM
2. Active SM unused warp slots: The unused warp slot is a member of an active SM

Unit	Pipeline Area	Description
SM	Shader	The Streaming Multiprocessor executes shader code.
L1TEX	Memory	The L1TEX unit contains the L1 data cache for the SM, and two parallel pipelines: the LSU or load/store unit, and TEX for texture lookups and filtering.
L2	Memory	The L2 cache serves all units on the GPU, and is a central point of coherency.
VRAM	Memory	GDDR6X
PD	World Pipe	The Primitive Distributor fetches indices from the index buffer, and sends triangles to the vertex shader.
VAF	World Pipe	The Vertex Attribute Fetch unit reads attribute values from memory and sends them to the vertex shader. VAF is part of the Primitive Engine.
PES+VPC	World Pipe	The Primitive Engine orchestrates the flow of primitive and attribute data across all world pipe shader stages (Vertex, Tessellation, Geometry). PES contains the stream (transform feedback) unit. The VPC unit performs clip and cull.
RASTER	Screen Pipe	The Raster units receives primitives from the world pipe, and outputs pixels (fragments) and samples (coverage masks) for the PROP, Pixel Shader, and ROP to process.
PROP	Screen Pipe	The Pre-ROP unit orchestrates the flow of depth and color pixels (fragments) and samples, for final output. PROP enforces the API ordering of pixel shading, depth testing, and color blending. Early-Z and Late-Z modes are handled in PROP.
ZROP	Screen Pipe	The Depth Raster Operation unit performs depth tests, stencil tests, and depth/stencil buffer updates
CROP	Screen Pipe	The Color Raster Operation unit performs the final color blend and render-target updates. CROP implements the “advanced blend equation”

Table 1: Explanation of unit throughput metrics, from [34]



SM Occupancy (TPC View)	Throughput %
⬛ Active SM Unused Warp Slots	69.3%
🟩 Pixel Warps	17.3%
⬜ Idle SM Unused Warp Slots	4.9%
🟧 Compute Warps	4.5%
🟦 Vertex/Tess/Geometry Warps	4.1%

Figure 8: SM Occupancy, from [34]

2.8 The current state of RAY-X

The original RAY software uses a CPU based ray tracing algorithm. Rays are generated by a source, which can be a matrix source, a point source or others implemented in RAY. The rays are then traced through the beamline. The intersection point between a ray and a optical element is calculated. Furthermore, other properties like reflection and refraction can be computed too. The algorithm iterates through all the optical elements defined in the beamline in a fixed order. Usually the algorithm finishes by calculating the intersection between a ray and an image plane, producing a footprint, which can be viewed as an image.

Quadrics There are different kinds of optical elements in RAY, most of which base on the mathematical representation of a quadric. RAY-X mirrors the basic calculations and representations of RAY, including how optical elements are represented. In Ray-X, a quadric is defined by 16 parameters, which define a surface in 3 dimensional space using the general equation for second order surfaces shown in equation 1.

$$\begin{aligned} F(x,y,z) = a_{11}x^2 + a_{22}y^2 + a_{33}z^2 + 2a_{12}xy + 2a_{13}xz \\ + 2a_{23}yz + 2a_{14}x + 2a_{24}y + 2a_{34}z + a_{44} = 0 \end{aligned} \tag{1}$$

This way, curved elements can be represented which would not have been possible with the widely used approach of representing objects using polygons. The increased accuracy gained by using a quadric representation is important to compute realistic ray paths and footprints. This accuracy however comes at a cost, as the function needed to compute the intersection point is significantly more complicated and computationally expensive.

If-conditions Moreover, the calculations depend on if-conditions, as determining the intersection between a ray and the quadric is calculated according to the direction of the ray. The direction vector consists of the values x, y and z. x, y or z is "dominant", if its value is larger than the values of the other two, e.g. the direction vector $(1, 2, 1)$ is y-dominant. Algorithm 1 shows if-conditions similar to the if-conditions in the intersection point function. This is a disadvantage compared to ray tracing using triangles. It is not a problem when doing the calculations on a CPU, as the data is processed sequentially. When using a GPU however, data is processed using a SIMT approach. Using different instructions depending on the data interferes with SIMT and can lead to conditional branching, possibly slowing down the calculations significantly. This problem will be assessed in this work.

Algorithm 1: If-conditions depending on the ray direction

```
if x-direction is dominant then
    funcA(ray)
else if y-direction is dominant then
    funcB(ray)
else
    funcC(ray)
end if
```

Sequential Tracing Similarly to RAY, optical elements are represented sequentially. This means that each optical element is defined relative to the preceding element. Optical elements, including the image plane, have a strict order which is defined at the creation of the beamline. As a result, the optical elements are also traced sequentially. Since most of the rays are bundled, meaning they are close together and have a relatively similar direction, the majority of rays are traced correctly. This way of representing the beamlines however has its limitations. Rays that would pass a optical element without hitting it, but then hitting another element in the beamline are ignored.

Initialization Overhead One of the applications for RAY is the generation of data for machine learning. For this, a significant amount of data is needed. At the current state many instances of RAY are run on a server, utilizing tens to hundreds of CPU cores to generate data more quickly. This however can lead to a significant overhead, as the entire program must be initialized for every beamline. RAY-X presents the same behavior, which is amplified, since the initialization of VULKAN takes more time. This results in even greater overhead, slowing down the generation of the ray tracing data. One specific machine learning problem for which RAY is currently used depends on a beamline which is traced using 2 million rays. In the current state of RAY-X, a single RAY uses 64 bytes of data, resulting in 128 megabytes (MB) of data. An RTX 3090 possesses 24 gigabytes (GB) of data and has a bandwidth of 936.2 GB per second, of which only a fraction is used when tracing 128 MB. This indicates that there is sufficient headroom, which can be used to process more data using a single initialization. This is until one of the resources is exhausted, resulting in a bottleneck.

3 New Approaches

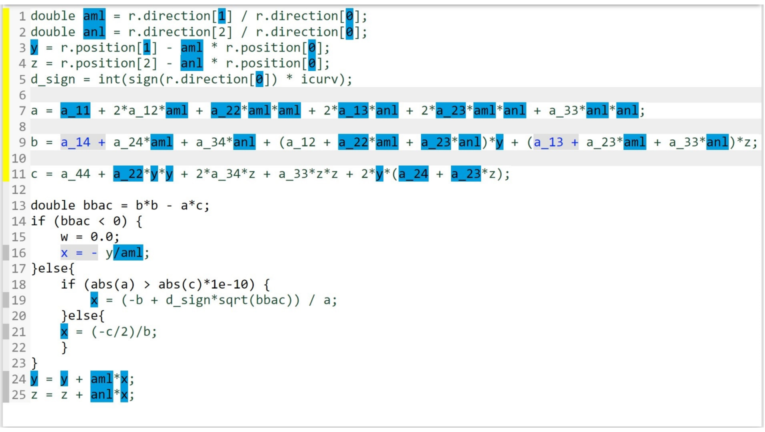
This work consists of three extensions for the existing RAY-X software with the aim of improving the execution time and adding new features. This includes optimization for a high throughput of beamlines.

3.1 Eliminating Conditional Branching in the intersectionPoint-Method

As described in chapter 2.9, the tracing algorithm used by RAY-X is ported from CPU-optimized Fortran code to a GLSL-based compute shader. As a result, the algorithm is not optimized for GPU computing. While the standard calculation of the intersection point does not include significant branching, the optimizations done to improve the accuracy do. The standard approach consists of inserting the coordinates of the ray into the general equation for second order surfaces [4, 36]. When the obtained quadratic expression is solved, the intersection point between the ray and the quadric is found. However, this can be optimized. In particular, the ray is translated towards the origin of the coordinate system until the y-z-plane is hit. This results in a normalized ray, which then simplifies the equation. Because of this a higher accuracy is achieved, which is needed because of the limited precision of double precision numbers. The optimized method calculating the intersection point between a ray and a quadric depends on the direction of the ray. Because of this, the function makes use of if-conditions to split the calculations into three cases. Depending on which of the x, y or z values of the direction of a ray is dominant, different instructions are executed.

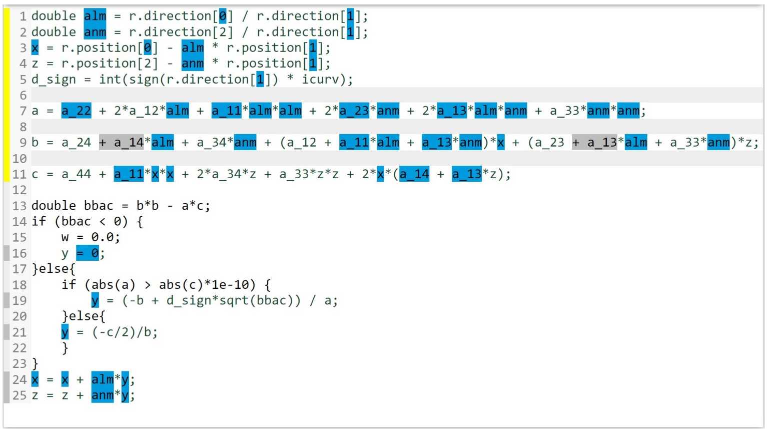
On a GPU, which is based on SIMT, this can lead to conditional branching [26]. This happens, when two threads in a single warp need to execute different instructions. In this case, threads in the same warp must wait until the other thread finishes the calculations in its branch. If the main direction of a ray A is different than the direction of a ray B and the threads of both A and B are in the same warp, conditional branching happens. In a warp with a size of 32 threads, only one thread has to take a different branch than the others to invoke conditional branching.

Issues of the intersectionPoint-Method In the case of the intersection point function, there are three branches. It is possible that all three branches need to be executed by a warp, which can lead to three times the computational cost, thus resulting in significantly worse execution times. While there are three different branches, the calculations done in each branch are very similar to the other two branches. As seen in Figure 9, the only difference between the branches is the data used to do the calculation, while the basic instructions are the same. The differences are marked in blue. This makes the algorithm a candidate for optimization using algebraic expressions.



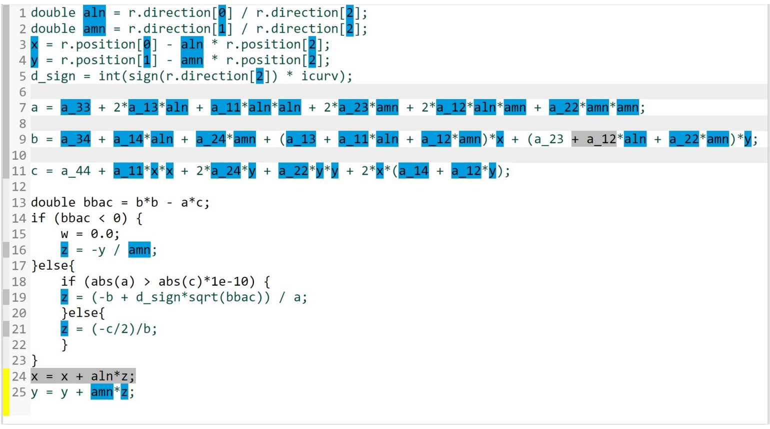
```
double aml = r.direction[1] / r.direction[0];
double anl = r.direction[2] / r.direction[0];
y = r.position[1] - aml * r.position[0];
z = r.position[2] - anl * r.position[0];
d_sign = int(sign(r.direction[0]) * icurv);

a = a_11 + 2*a_12*aml + a_22*aml*aml + 2*a_13*anl + 2*a_23*aml*anl + a_33*anl*anl;

b = a_14 + a_24*aml + a_34*anl + (a_12 + a_22*aml + a_23*anl)*y + (a_13 + a_23*aml + a_33*anl)*z;

c = a_44 + a_22*y*y + 2*a_34*z + a_33*z*z + 2*y*(a_24 + a_23*z);

double bbac = b*b - a*c;
if (bbac < 0) {
    w = 0.0;
    x = - y/aml;
}else{
    if (abs(a) > abs(c)*1e-10) {
        x = (-b + d_sign*sqrt(bbac)) / a;
    }else{
    x = (-c/2)/b;
    }
}
y = y + aml*x;
z = z + anl*x;
```

(a) Case = 1



```
double alm = r.direction[0] / r.direction[1];
double anm = r.direction[2] / r.direction[1];
x = r.position[0] - alm * r.position[1];
z = r.position[2] - anm * r.position[1];
d_sign = int(sign(r.direction[1]) * icurv);

a = a_22 + 2*a_12*alm + a_11*alm*alm + 2*a_23*anm + 2*a_13*alm*anm + a_33*anm*anm;

b = a_24 + a_14*alm + a_34*anm + (a_12 + a_11*alm + a_13*anm)*x + (a_23 + a_13*alm + a_33*anm)*z;

c = a_44 + a_11*x*x + 2*a_34*z + a_33*z*z + 2*x*(a_14 + a_13*z);

double bbac = b*b - a*c;
if (bbac < 0) {
    w = 0.0;
    y = 0;
}else{
    if (abs(a) > abs(c)*1e-10) {
        y = (-b + d_sign*sqrt(bbac)) / a;
    }else{
    y = (-c/2)/b;
    }
}
x = x + alm*y;
z = z + anm*y;
```

(b) Case = 2



```
double aln = r.direction[0] / r.direction[2];
double amn = r.direction[1] / r.direction[2];
x = r.position[0] - aln * r.position[2];
y = r.position[1] - amn * r.position[2];
d_sign = int(sign(r.direction[2]) * icurv);

a = a_33 + 2*a_13*aln + a_11*aln*aln + 2*a_23*amn + 2*a_12*aln*amn + a_22*amn*amn;

b = a_34 + a_14*aln + a_24*amn + (a_13 + a_11*aln + a_12*amn)*x + (a_23 + a_12*aln + a_22*amn)*y;

c = a_44 + a_11*x*x + 2*a_24*y + a_22*y*y + 2*x*(a_14 + a_12*y);

double bbac = b*b - a*c;
if (bbac < 0) {
    w = 0.0;
    z = -y / amn;
}else{
    if (abs(a) > abs(c)*1e-10) {
    z = (-b + d_sign*sqrt(bbac)) / a;
    }else{
    z = (-c/2)/b;
    }
}
x = x + aln*z;
y = y + amn*z;
```

(c) Case = 3

Figure 9: Comparison of branches. r is the ray and $a11$ to $a44$ are the coefficients of the quadric.

3.1.1 Algebraic Method

One of the goals of this work is removing the If-conditions by using algebraic expressions. Depending on the case which is currently processed, different values of the quadrics anchor points are used. Furthermore, the branches not only have different r-values, but also different l-values, e.g. in Figure 9, line 3, y is calculated for case 1, while x is calculated for case 2.

The algebraic method combines the code of all three branches shown in Figure 9 into one algorithm.

Algorithm 2: Determining the *case* the algorithm needs to process

```
Data: Ray, Quadric
Result: X, Y and Z coordinates of the intersection between the ray and the
        quadric
$case \leftarrow 0$;
if $abs(r.direction.y) > abs(r.direction.x)$ and
  $abs(r.direction.y) > abs(r.direction.z)$ then
    $case \leftarrow 1$
else if $abs(r.direction.z) > abs(r.direction.x)$ and
  $abs(r.direction.z) > abs(r.direction.y)$ then
    $case \leftarrow 2$
end
```

Case Index While the old intersectionPoint-method uses the case indices 1,2 and 3, the new approach uses 0, 1 and 2 for easier calculation. Algorithm 2 shows how the *case* index is calculated: 0 if X is dominant, 1 if Y is dominant and 2 if Z is dominant. These indices are used to calculate which values are processed in the final algorithm. Figure 9a shows the code executed if the x-direction of the vector is dominant. In line 1 the value for *aml* is calculated by dividing the y-direction by the x-direction of the ray. In code, the direction is represented by a 3-dimensional vector where $r.direction[0]$ = x-direction of the ray, $r.direction[1]$ = y-direction of the ray and $r.direction[2]$ = z-direction of the ray. Using this representation, we can construct a mathematical function which maps the case index to the corresponding index needed for this case.

Algorithm 3: Calculating *alpha* using if-conditions

```
if case is 0 then
    $alpha \leftarrow \frac{dir[1]}{dir[0]}$
else if case is 1 then
    $alpha \leftarrow \frac{dir[0]}{dir[1]}$
else if case is 2 then
    $alpha \leftarrow \frac{dir[0]}{dir[2]}$
end if
```

Example: Calculation of alpha The terms *aml*, *alm* & *aln* are used in case 1, 2 and 3 respectively, which can be seen in Figure 9, line 1. For the algebraic method, they are grouped as *alpha*. Furthermore, the terms *anl*, *anm* & *amn* in line 2, also used in case 1, 2 and 3 respectively, are grouped into *beta*. Algorithm 3 shows how *alpha* is calculated using if-conditions. To eliminate the if-conditions, algebraic expressions are used, which dynamically choose the values needed to calculate alpha. For this, the numeric value of *case* is used. Let *case* be 0, then *alpha* (which corresponds to *aml* if *case* is 0) is calculated by $\frac{dir[1]}{dir[0]}$. For $case = 1$, *alpha* is calculated by $\frac{dir[0]}{dir[1]}$ and for $case = 2$, *alpha* is $\frac{dir[0]}{dir[2]}$. To remove the if-conditions, *case* is used to calculate the indices for *dir*. In the counter of Algorithm 3 the following map is needed: $f(0) = 1, f(1) = 0$ and $f(2) = 0$.

To construct such a map, following strategy might be used. Firstly, identify values which map to the same result. In this case, 1 and 2 both need to map to the same value. This can be achieved by adding 1 to the input, then dividing by 2 and using the floor function. This results in $f_{partial}(x) = \lfloor \frac{x+1}{2} \rfloor$ which maps 0 to 0, 1 to 1 and 2 to 1. The next step is to finalize f, so the exact needed values are calculated. In this case, the *modulo* operator can be used. Adding 1 to the output of $f_{partial}$ and using *modulo* 2, the resulting function f_{100} shown in Algorithm 5 maps 0 to 1, 1 to 0 and 2 to 0.

Using f_{100}, the index of *alpha* can be calculated dynamically using the *case* index. The dynamic calculation of *alpha* can be seen in Algorithm 4.

Algorithm 4: Dynamically calculating *alpha*

```
$alpha \leftarrow \frac{r.direction[f_{100}(case)]}{r.direction[case]}$;
```

If case is 0, then $alpha \leftarrow \frac{r.direction[f_{100}(0)]}{r.direction[0]}$ evaluates to $\frac{r.direction[1]}{r.direction[0]}$, which is the same result of the method shown in Algorithm 3. This calculation however only outputs the right value if *case* is 0,1 or 2, meaning that assertions should be used when implementing the algorithm.

Algorithm 5: Maps used in the intersectionPoint-method

```
$f_{001}(x) = \lfloor \frac{x}{2} \rfloor$
$f_{010}(x) = x \bmod 2$
$f_{011}(x) = \lfloor \frac{x+1}{2} \rfloor$
$f_{0x1}(x) = \lfloor \frac{x}{2} \rfloor$
$f_{100}(x) = (\lfloor \frac{x+1}{2} \rfloor + 1) \bmod 2$
$f_{101}(x) = ((x \bmod 2) + 1) \bmod 2$
$f_{110}(x) = (\lfloor \frac{x}{2} \rfloor + 1) \bmod 2$
$f_{112}(x) = \lfloor \frac{x}{2} \rfloor + 1$
$f_{131}(x) = ((x \bmod 2) \cdot 2) + 1)$
$f_{220}(x) = (\lfloor \frac{x}{2} \rfloor + 1) \bmod 2 \cdot 2$
$f_{221}(x) = ((\lfloor \frac{x}{2} \rfloor + 1) \bmod 2) + 1$
```

Naming Scheme The name f_{100} of this function is chosen according to the output of the function. The input values 0, 1, 2 result in the values $f_{100}(0) = 1$, $f_{100}(1) = 0$, $f_{100}(2) = 0$. One exception to this rule is the function f_{0x1}, which maps 0 to 0, and 2 to 1. The output of the function for $x = 1$ is irrelevant for the particular part of the algorithm in which f_{0x1} is used. This is because for $case = 1$ the mathematical equation resolves to 0 regardless of the value of f. To completely remove the If-conditions, more functions are needed. The functions used in the updated intersectionPoint-method are detailed in Algorithm 5.

New intersectionPoint-Method Algorithm 6 shows the updated method for finding the intersection point between a ray and a quadric. It makes use of the algebraic functions defined in Algorithm 5 to eliminate the branching present in the old intersectionPoint-method. The algorithm includes the optimizations needed for the increased accuracy. Algorithm 6 combines the 3 branches of the old intersectionPoint-method into a single code segment. This removes the need to calculate values more than once, which can happen when using branching. The remaining if-conditions can not be replaced by the algebraic method.

Algorithm 6: Code replacing the if-conditions

```
**Data:** Ray r, Quadric q, Case cs
**Result:** X, Y and Z coordinates of the intersection between the ray and the quadric
$alpha \leftarrow \frac{r.direction[f_{100}(cs)]}{r.direction[cs]}$;
$beta \leftarrow \frac{r.direction[f_{221}(cs)]}{r.direction[cs]}$;
$params \leftarrow (alpha, beta)$;
$xyz[f_{100}(cs)] \leftarrow r.position[f_{100}(cs)] - alpha \cdot r.position[cs]$;
$xyz[f_{221}(cs)] \leftarrow r.position[f_{221}(cs)] - beta \cdot r.position[cs]$;
$xyz[3] \leftarrow 0$;
$d_{sign} \leftarrow \lfloor sign(r.position[cs]) \cdot icurv \rfloor$;
$a \leftarrow q[cs][cs] + 2 \cdot q[0][f_{112}(cs)] \cdot alpha + q[f_{100}(cs)][f_{100}(cs)] \cdot alpha \cdot alpha + 2 \cdot q[f_{011}(cs)][2] \cdot beta + 2 \cdot q[f_{100}(cs)][f_{221}(cs)] \cdot alpha \cdot beta + q[f_{221}(cs)][f_{221}(cs)] \cdot beta \cdot beta$;
$b \leftarrow q[cs][3] + q[f_{100}(cs)][3] \cdot alpha + q[f_{221}(cs)][3] \cdot beta + (q[0][f_{112}(cs)] + q[f_{100}(cs)][f_{100}(cs)] \cdot alpha + q[f_{100}(cs)][f_{221}(cs)] \cdot beta) \cdot xyz[f_{100}(cs)] + (q[f_{011}(cs)][2] + q[f_{100}(cs)][f_{221}(cs)] \cdot alpha + q[f_{221}(cs)][f_{221}(cs)] \cdot beta) \cdot xyz[f_{221}(cs)]$;
$c \leftarrow q[3][3] + q[f_{100}(cs)][f_{100}(cs)] \cdot xyz[f_{100}(cs)] \cdot xyz[f_{100}(cs)] + 2 \cdot q[f_{221}(cs)][3] \cdot xyz[f_{221}(cs)] + q[f_{221}(cs)][f_{221}(cs)] \cdot xyz[f_{221}(cs)] \cdot xyz[f_{220}(cs)] + 2 \cdot xyz[f_{100}(cs)] \cdot (q[f_{100}(cs)][3] + q[f_{100}(cs)][f_{221}(cs)] \cdot xyz[f_{221}(cs)])$;
$bbac \leftarrow b^2 - a \cdot c$;
**if** $bbac < 0$ **then**
    $ray.weight \leftarrow 0$;
    $xyz[cs] \leftarrow -\frac{xyz[f_{131}(cs)]}{params[f_{0x1}(cs)]}$;
**else**
    **if** $abs(a) > 1e - 10$ **then**
        $xyz[cs] \leftarrow \frac{(-b + d_{sign} \cdot \sqrt{bbac})}{a}$;
    **else**
        $xyz[cs] \leftarrow (\frac{-c}{2 \cdot b})$;
    **end**
**end**
$xyz[f_{100}(cs)] \leftarrow xyz[f_{100}(cs)] + alpha \cdot xyz[cs]$;
$xyz[f_{221}(cs)] \leftarrow xyz[f_{221}(cs)] + beta \cdot xyz[cs]$;
```

3.2 Dynamic Ordering of Optical Elements

Motivation The beamline representation of RAY, where optical elements are in a strict order, has its limitations. Only rays which hit all optical elements in the beamline are considered when generating the footprint on the image plane. For the majority of beamlines this representation is sufficient, as almost all of the rays are traveling through each optical element. When designing new beamlines, mistakes like an incorrect alignment of optical elements could cause gaps in a beamline, allowing rays to pass optical elements. These rays can then falsify the footprint on the image plane. Figure 10 shows two rays traveling from a light source to an image plane. The blue ray takes the desired path and hits the plane mirror before arriving at the image plane. Ideally all rays should take this path, but depending on the configuration of the beamline, some rays might hit the image plane without hitting all optical elements first. This behaviour can not be simulated using a static beamline representation.

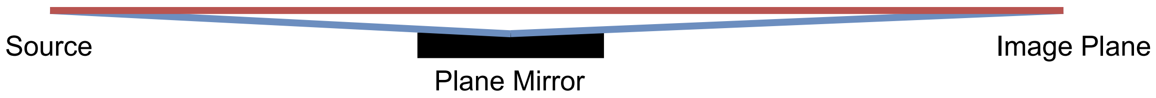
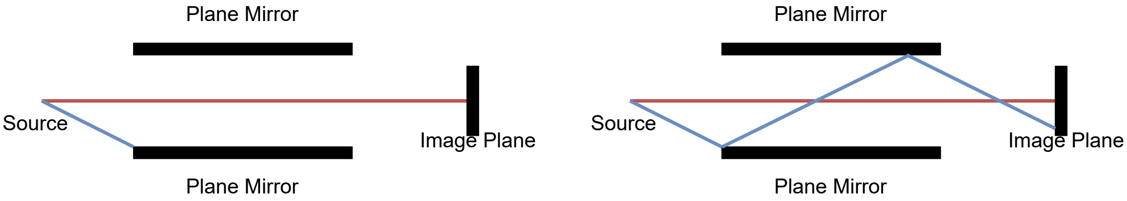


Figure 10: Rays traveling from Source to Image Plane

Bounces In the current implementation of beamlines, leaking rays can go undetected, as the rays are simply ignored. To improve the realism of the light simulation, a ray tracing algorithm using dynamic ordering of optical elements has been developed. In contrast to the old technique, the optical elements are not iterated in a fixed order to calculate the path of the rays. Instead, the parameter *bounces* has been introduced. Similar to the real world, a ray interacts with the first surface it hits. Depending on the surface, the direction of the ray may be altered. Using this representation, a ray is not guaranteed to hit the image plane and might bounce between different surfaces indefinitely. This can also cause some rays to reach the image plane in few iterations, while other rays might take many iterations or never reach it. Because of this such a simulation can take a lot of time or possibly never finish. To address this behaviour, the number of bounces each ray can make is limited. The parameter *bounces* determines the maximum number of interactions of each ray that are calculated.



(a) Rays traveling 1 Bounce

(b) Rays traveling 3 Bounces

Figure 11: Rays traveling through a beamline

Computational Complexity The main challenge of using dynamic ordering of optical elements is the determination of which element is hit first. For this, the intersection between the ray and each element has to be calculated. Compared to the original approach, this technique is more computationally expensive, as the intersection has to be calculated more often. Instead of $numberOfRays \cdot numberOfOpticalElements$ intersections, $numberOfRays \cdot numberOfOpticalElements \cdot bounces$ intersections have to be calculated. The value of $bounces$ needs to be chosen by the user of the program. A higher value for $bounces$ results in a more realistic behaviour. This is visualized in Figure 11a and 11b. When using a higher number of bounces, more rays can reach the image plane. In an usual beamline most rays hit every optical element once. This is why the number of bounces should to be at least as high as the number of optical elements so that the rays can reach the image plane. Using the minimum needed number of bounces, the computational complexity of the approach is $numberOfRays \cdot numberOfOpticalElements^2$. When using a high number of bounces or optical elements, the calculation of the intersection points can become the main bottleneck of the algorithm.

Possible Optimizations Many ray tracing implementations, especially those used for rendering realistic scenes for movies or video games, employ optimization techniques like bounding boxes [28] to reduce the time spent calculating intersections. These scenes however have far more objects the ray can intersect with, reaching hundreds of millions to billions of polygons [44]. In a beamline the number of optical elements rarely exceeds 10. This removes the need for the usual optimizations done in ray tracing programs.

Finding the Closest Intersection The intersection point between each ray and each optical element is calculated for every bounce. Figure 12 shows an example of a beamline where the rays pass through multiple optical elements. To determine the closest intersection point, the intersection between the ray and each optical element is calculated. Because the optical elements in a beamline often lie behind one another, multiple intersection points are found. The distances between the ray origin and the intersection points are calculated and compared. The optical element which has the shortest distance from the ray origin is chosen. Depending on the type of the optical element, optical effects like reflection and refraction are computed to generate the new direction of the r. Using the new position and direction, the next bounce can be calculated.

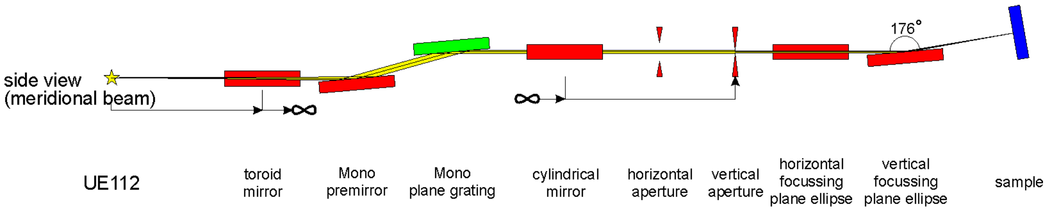


Figure 12: UE112_PGM-1 beamline in BESSY II, from [45]

Self Intersection Due to the limitations of double-precision numbers, the computed intersection point is usually not fully accurate. This can lead to problems like self intersection [32], where the same optical element is hit again. Self intersection is a problem because the newly calculated intersection point between the ray and the optical element that was hit in the previous iteration is at the same coordinates as the origin of the ray. This causes the distance between the closest intersection point and the position of the ray to be 0. As a result, the same optical element is chosen again, which repeats for every bounce. The approach to this issue detailed in [32] is not applicable to ray-quadric intersections as quadrics are not necessarily planar surfaces.

Because of the nature of curved surfaces it is possible for a ray to intersect with a quadric in 2 points, meaning the previous quadric can not simply be ignored in the determination of the closest intersection. Figure 13 shows a ray being reflected onto the same curved surface in a 2-dimensional room. In this example 2 bounces are needed to traverse the optical element. The function for determining the intersection point is capable of determining both intersection points given the ray intersects the quadric at two points. When calculating the intersection point between the ray and the quadric that was previously hit, this behaviour is exploited to ignore the closest intersection, which would be at a distance of 0. If there is a second intersection point, it will be considered in the selection of the closest intersection point.

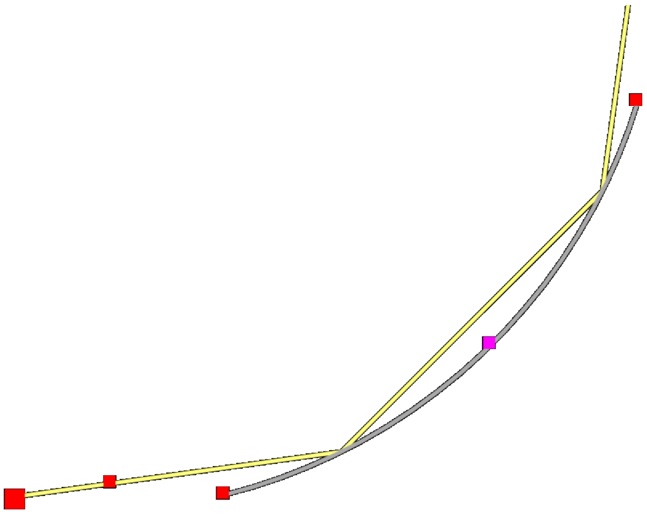


Figure 13: Example of a ray hitting a curved surface multiple times in a row

Dynamic Tracing Procedure Algorithm 7 shows the tracing procedure, which is executed in parallel for every ray. The variables *bounces*, which was detailed before, *ray* and $numberOfOpticalElements$ are passed as an argument. The variable $closestOpticalElementIndex$ stores the optical element with the current closest intersection point and $whichIntersection$ is used to decide whether the first or second intersection point between a ray and a given optical element is used. The algorithm iterates *bounces* times and executes the code in the outer for-loop if the image plane has not been hit yet. In the loop, the variable *lastClosestOpticalElementIndex* is used to store which optical element was chosen in the last iteration. This is important to avoid self intersection later. In the second for loop, which executes for each optical element in the beamline, the closest optical element is determined.

The function *getOpticalElementIntersection*() is called using the parameters *ray*, the current index of the optical element and information about whether the first or second intersection point should be determined. If *whichIntersection* is 1, the first intersection between the optical element and the ray is calculated. Using -1, the second intersection is determined. The value of *whichIntersection* is set according to the last chosen optical element. If *opticalElementIndex* is the same as the last chosen optical element (*lastClosestOpticalElementIndex*), the second intersection needs to be determined and *whichIntersection* is set to -1.

The method *getOpticalElementIntersection*() returns the intersection point between the ray and the optical element. The euclidean distance between the returned value and the position of *ray* is calculated and compared to the current shortest distance, which is the distance between *closestHitpoint* and the position of the ray. If the returned intersection point has a shorter distance to *ray.position*, *closestHitpoint* is set to the value in *currentIntersection*. *closestOpticalElementIndex* is also set to *opticalElementIndex*.

If *closestOpticalElementIndex* is still -1 after the calculations, the algorithm returns, preventing unnecessary computation. After the closest intersection point is determined, the function *traceOpticalElement*() is used to determine the new position, direction and weight of the ray. If the traced optical element is an image plane, the algorithm returns, stopping further calculations for this ray.

Algorithm 7: Algorithm for tracing a beamline dynamically. This is run in parallel for each ray

```
Data: ray, bounces, numberOfOpticalElements
$closestOpticalElementIndex \leftarrow -1$;
$whichIntersection \leftarrow 1$;
for $i \leftarrow 0$ to $bounces$ do
    $lastClosestOpticalElementIndex \leftarrow closestOpticalElementIndex$;
    $closestHitpoint \leftarrow$ (infinity, infinity, infinity);
    for $opticalElementIndex \leftarrow 0$ to $numberOfOpticalElements$ do
        if $opticalElementIndex == lastClosestOpticalElementIndex$ then
            $whichIntersection \leftarrow -1$;
        else
            $whichIntersection \leftarrow 1$;
        end
        // returns infinity if no intersection is found or no
           quadric with the given index is found
        $currentIntersection \leftarrow$ getOpticalElementIntersection(ray,
          opticalElementIndex, whichIntersection);
        if distance(ray.position, currentIntersection) < distance(ray.position,
          closestHitpoint) then
            $closestHitpoint \leftarrow currentIntersection$;
            $closestOpticalElementIndex \leftarrow opticalElementIndex$;
        end
    end
    if $closestOpticalElementIndex == -1$ then
        $ray.weight \leftarrow 0$;
        return;
    end
    // updates ray
    traceOpticalElement(ray, closestOpticalElementIndex, closestHitpoint);
    if typeOfElement(closestOpticalElementIndex) == imagePlane then
        return;
    end
end
```

3.3 Multiple Beamlines

Motivation Experiments have shown that initializing VULKAN for every beamline results in a significant overhead. While the execution time of 148 ms for the initialization is not noticeable while designing a beamline, it does make a great difference when tracing many beamlines. This is particularly important for the generation of data for machine learning. When tracing multiple hundred thousands of beamlines, the initialization time adds up, slowing the process down. To address this problem, multiple beamlines have to be traced following a single initialization step. Depending on the number of rays traced per beamline, the GPU might not be fully utilized. Tracing one beamline after the other while not fully utilizing the GPU results in a bad efficiency. To fully exhaust the GPUs parallel computing abilities, multiple beamlines can be traced at the same time.

Memory Requirements In the current version of RAY-X, a single ray is represented by 8 double precision values, using 8 bytes of data each, resulting in 64 bytes of data per ray. Figure 14a shows how rays are represented in RAY-X.

```
Ray {
  dvec3 position;
  double weight;
  dvec3 direction;
  double energy;
}
```

(a) Ray representation

```
Quadric {
  dmat4 anchorPoints;
  dmat4 inTranslation;
  dmat4 outTranslation;
  dmat4 misalignment;
  dmat4 inverseMisalignment;
  dmat4 objectParameters;
  dmat4 elementParameters;
}
```

(b) Quadric representation

Figure 14: Representation of Ray and Quadric in RAY-X. A dvec3 consists of 3 doubles: x,y,z. A dmat4 consists of 16 doubles values.

Furthermore, an additional 32 bytes per ray are used for the calculation of the intersection point when tracing the beamline using the updated method for finding the intersection point. An optical element is defined by 7 double precision 4x4 matrices, as shown in Figure 14b. This results in 112 double values, or 896 bytes of data per optical element. Even when using a low number of rays, such as 20,000, the storage required for the rays exceeds the storage required for the optical elements significantly. 20,000 rays use 1028 kilobytes, while 10 optical elements only use 8.96 KB. The difference increases even further when using more rays per beamline, making the amount data used for optical elements negligible.

Limitations RAY-X is designed to work on multiple operating systems, including Microsoft Windows. Due to the Windows timeout detection, tasks that take more than 2 seconds to respond are marked as frozen, causing the task to be cancelled [38]. Because of this the compute shader should always finish in less than 2 seconds. The beamline specified in section 5 takes 28.6 ms using the new intersectionPoint-method. If it is assumed that the execution time scales linearly with the number of beamlines traced, a maximum number of 69 beamlines can be traced. Tracing 69 beamlines results in 128 million rays, which take up around 8.8 gigabytes of VRAM. This is well within the maximum of 24 GB that is available on the RTX 3090. However, other consumer graphics cards with less VRAM might be limited by the size of the memory.

Memory Layout When tracing multiple beamlines, the data for the different beamlines is consecutively in memory. As Figure 15 shows, rays are consecutively in the ray buffer and the quadrics are consecutively in the buffer for the optical elements. This brings the advantage of better cache locality, as rays that are in the same thread group usually interact with optical elements in the same beamline. Figure 16 shows how the memory is divided when tracing multiple beamlines. Furthermore, the xyz-Buffer, which is detailed in chapter 4, is only needed when using the updated method for finding the intersection point.

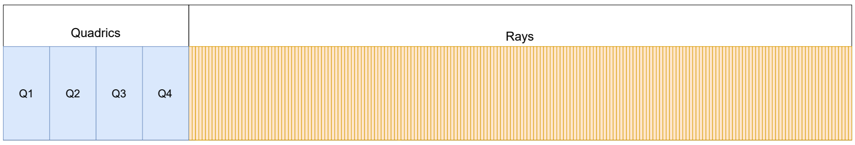


Figure 15: RAY-X Memory Layout with static optical element ordering, the old intersectionPoint-method, 4 Quadrics and 200 rays

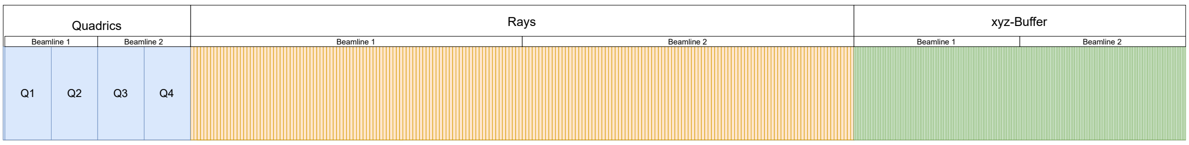


Figure 16: RAY-X Memory Layout with dynamic ordering of optical elements, the new intersectionPoint-method and 2 beamlines with 2 Quadrics and 100 rays each

Because of this layout, parameters are transferred to the shader containing information about the number of beamlines to be traced. Currently it is only possible to trace multiple beamlines which have the same number of rays and the same number of optical elements. This is because for machine learning, only small adjustments to the source and optical elements are done, while the basic structure of the beamline stays the same. However, support for tracing different beamlines simultaneously can be added easily in the future if needed. For this, information about the number of quadrics in each beamline is needed. The header of the buffer could be extended to include this information.

Algorithm 8: Algorithm for tracing multiple beamlines dynamically. This is run in parallel for each ray

```
Data: ray, bounces, numberOfOpticalElements
$closestOpticalElementIndex \leftarrow -1$;
$whichIntersection \leftarrow 1$;
$beamlineIndex \leftarrow \lfloor threadID/numberOfRaysPerBeamline \rfloor$;
for $i \leftarrow 0$ to $bounces$ do
    $lastClosestOpticalElementIndex \leftarrow closestOpticalElementIndex$;
    $closestHitpoint \leftarrow$ (infinity, infinity, infinity);
    for $opticalElementIndex \leftarrow beamlineIndex * numberOfOpticalElementsPerBeamline$ to $numberOfOpticalElementsPerBeamline$ do
        if $opticalElementIndex == lastClosestOpticalElementIndex$ then
            $whichIntersection \leftarrow -1$;
        else
            $whichIntersection \leftarrow 1$;
        end
        // returns infinity if no intersection is found or no
           quadric with the given index is found
        $currentIntersection \leftarrow$ getOpticalElementIntersection(ray, opticalElementIndex, whichIntersection);
        if $distance(ray.position, currentIntersection) < distance(ray.position, closestHitpoint)$ then
            $closestHitpoint \leftarrow currentIntersection$;
            $closestOpticalElementIndex \leftarrow opticalElementIndex$;
        end
    end
    if $closestOpticalElementIndex == -1$ then
        $ray.weight \leftarrow 0$;
        return;
    end
    // updates ray
    traceOpticalElement(ray, closestOpticalElementIndex, closestHitpoint);
    if $typeOfElement(closestOpticalElementIndex) == imagePlane$ then
        return;
    end
end
```

Implementation Tracing multiple beamlines simultaneously is implemented as an extension of dynamic ordering of optical elements. The memory layout is used to select which quadrics the ray can intersect with. As Algorithm 8, line 3 shows, the beamline index for each ray is calculated by dividing the thread ID by the number of rays per beamline. The beamline index is then used in line 7 to calculate an offset for the quadric buffer. In the example in Figure 16, $beamlineIndex$ can be 0 or 1. If it is 0, the for-loop starts at $Q1$ and runs to $Q2$. If $beamlineIndex$ is 1, the offset makes the for-loop start at $Q3$ and end at $Q4$. Using the offset, only the optical elements in the corresponding beamline are considered.

4 Implementation Details

While implementing the features there have been some issues which had to be dealt with. Especially the new intersectionPoint-method had to be adjusted to work on the GPU properly. To implement the algebraic expressions of the new intersectionPoint-method, dynamic indexing was needed.

Dynamic Indexing To use the algebraic expressions, the x, y and z values were represented using an array. Depending on the *case* index in the shader, the x, y or z value was changed, e.g. when *case* is 0, x might be changed. When *case* is 1, the value for y might be changed. To summarize, depending on the index, not only the value for the variable is dynamically chosen, but also which variable is to be changed. Because the value for *case* is not known when compiling, it is also not known which variable is to be changed. The old intersectionPoint-method used 3 arguments: the ray r, the anchor points of the optical element q and the parameter $icurv$. The ray r is the ray which intersects with a optical element, while q contains the parameters used for the intersection point calculation. The parameter $icurv$ is used to determine whether the first or second intersection point between the ray and the optical element should be calculated. Originally, the intersectionPoint-method uses the GLSL parameter qualifier *inout* to change the position of the ray. The functions return value is the normal of the intersection. Due to conditional branching, this structure could not be used.

The compute shader, while written in GLSL, is compiled to SPIR-V (Standard Portable Intermediate Representation). SPIR-V is a binary intermediate language for parallel compute and graphics [40, 41]. It is for representing shader stages and compute kernels for e.g. VULKAN, OpenGL and OpenCL. The Nvidia GPU driver uses their NVVM compiler stack for SPIR-V, which makes use of LLVM IR after being translated for SPIR-V [42]. Due to an interaction between these modules, dynamic indexing in GLSL results in runtime LLVM errors when initializing VULKAN.

To counter this behaviour, a Shader Storage Buffer Object (SSBO) is used. SSBOs are used to store and retrieve data within GLSL [43]. While this enables the usage of the algebraic expressions, it is not a perfect solution. SSBOs have the advantage of being much larger. The specification guarantees up to 128 megabytes of storage, while most implementations enable sizes up the limit of the GPU memory. This however also brings disadvantages, as the buffer is slow, resulting in more stalling of threads while waiting for data.

An SSBO is being used to create the xyz-buffer in the updated intersectionPoint-method, as mentioned in chapter 3.3. While tests have shown that the new intersectionPoint-method is more performant using the algebraic expressions, it is most likely being slowed down by the memory accesses.

Splitting of the intersectionPoint-Method The original implementation of the method for finding the intersection point between a ray and a optical element calculates the intersection and the normal of the intersection. For dynamic ordering of optical elements, the intersection point needs to be determined more often. As specified before, for each bounce, the intersections between the ray and all optical elements have to be determined. This results in $bounces \cdot numberOfOpticalElements$ times the intersection points needs to be calculated for each ray to find the closest optical element. Because the normal is not needed for this, there are more calculations done than necessary. Furthermore, the original $intersectionPoint$-method changes the position of the ray, which must not be done when determining the closest optical element. To combat these problems, the functions were split into $setIntersectionPoint$ and $getIntersectionPoint$. The latter is a cut down version of the original method, that does not calculate the normal vector. Moreover, it does not change the position of the ray, but returns a 3-dimensional vector containing the hit point. The function $setIntersectionPoint$ gets the intersection point calculated in $getIntersectionPoint$ as an argument and calculates the normal, while also setting the new ray coordinates. Combining the functions results in the behaviour of the old intersectionPoint-method.

5 Experiments

RAY-X has been tested thoroughly using multiple beamline configurations. All newly implemented features and optimizations have been evaluated and compared. The beamlines used for the evaluation are partly based on specific applications of RAY-X, while some beamlines used for testing have minor changes to make them more suitable for testing. The most important beamline used for testing is being used for the generation of data for a machine learning model. This beamline consists of a point source, which generates the rays used in the tracing process. Furthermore, the beamline contains one reflection zone plate and an image plane. The machine learning model requires the beamline to be traced using 2,000,000 rays.

Testing Hardware The machine on which the performance has been evaluated on has an AMD Ryzen 3600 processor running at 4.1 GHz, a B550 motherboard, 32 GB of RAM running at 3600 MHz and an RTX 3090 with 24 GB of VRAM, connected to the PC via 16 PCIe4 lanes.

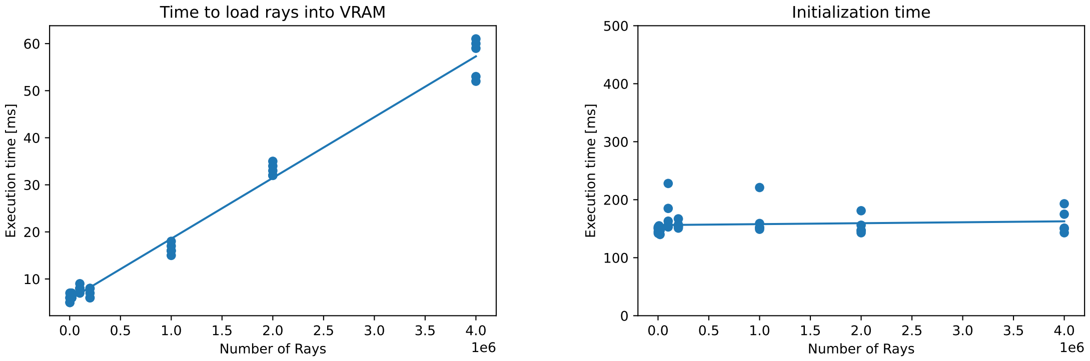
5.1 Identifying Bottlenecks

VULKAN Initialization Initial experiments showed that there are two main bottlenecks when tracing a beamline. Firstly, initializing VULKAN presented a significant overhead, especially when tracing beamlines with a small amount of rays. As shown in table 2, running a beamline with 2 million rays takes on average 371 milliseconds when using dynamic ordering of optical elements. Of these 371 ms, roughly 150 ms is spent initializing VULKAN, i.e. creating the buffers, creating a compute queue, initializing the physical device and more. This means that more than 42% of the processing time is spent setting up the environment. Figure 17b shows, that the initialization time is constant and is not influenced by the amount of rays. When tracing beamlines using smaller amounts of rays, e.g. 20,000, the time spent initializing VULKAN can be magnitudes higher than the time spent for the actual tracing process.

The overhead of around 150 milliseconds is not noticeable when designing beamlines, as only a single beamline needs to be traced. However, it makes a great difference when generating data for machine learning. Using the unoptimized algorithm which initializes VULKAN for every beamline, the cumulative initialization time of 1 million tracing passes exceeds 41 hours.

Number of Rays	Initialize VULKAN	Load rays into VRAM	Run Shader	Sum
2,000,000	154 ms	33 ms	184 ms	371 ms
200,000	156 ms	5 ms	24 ms	185 ms
20,000	147 ms	3 ms	5 ms	155 ms

Table 2: Average computation time for a beamline with 1 point source, 1 reflection zone plate, 1 image plane, using dynamic optical elements



(a) Loading rays into VRAM in linear time

(b) Initialization time is constant

Figure 17: Beamline with 1 point source, 1 reflection zone plate, 1 image plane, using static optical element ordering

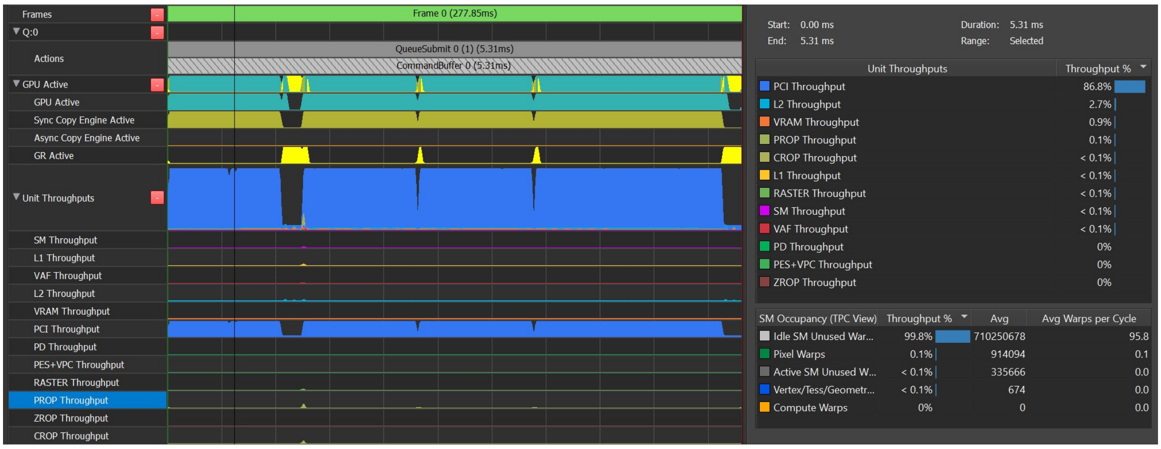
Optimizing the Shader For larger amounts of rays like 2,000,000, the time spent running the actual shader becomes more relevant. When tracing the beamline used for the generation of table 2, the time spent running the tracing algorithm is around 50%. When using a higher number of optical elements or *bounces*, even more time is spent running the shader. Using the feature implemented in section 3.3, the overhead caused by the initialization of VULKAN becomes increasingly insignificant. Ignoring this overhead, running the shader takes up 85% of the computation time. This is a significant fraction, which means that improvements in this section would also vastly speed up the entire process.

Data Transfer Copying the rays to the VRAM only takes up 15% of the time and scales linearly with the number of rays, as seen in Figure 17a. Moreover, tests show that the PCIe throughput while transferring the rays is on average at 28 GB/s. The RTX 3090 is connected via 16 PCIe 4.0 lanes, which enables a bandwidth of 31.51 GB/s [46]. This means that the rays are transferred using 89% of the maximum available bandwidth, leaving little room for improvement.

Figure 18 shows typical Unit Throughputs when transferring 2,000,000 rays to the GPU. In this example, the average PCIe Throughput is at 86.8%, while it peaks at 94.1%. Furthermore, the graphics/compute-engine (GR Active) [35] has a throughput of 100% while the PCIe throughput drops at 1 ms, meaning that the GPU is fully utilized at all times.

5.2 Performance of the Improved intersectionPoint-Method

Figure 19 shows the execution time of the shader without any modifications. The shader is using the original method of calculating the intersection point. Furthermore, only one beamline is traced at the same time and dynamic ordering of optical elements is not used, i.e. the optical elements are in a set order.



Start: 0.00 ms
End: 5.31 ms
Duration: 5.31 ms
Range: Selected

Unit Throughputs	Throughput %
PCI Throughput	86.8%
L2 Throughput	2.7%
VRAM Throughput	0.9%
PROP Throughput	0.1%
CROP Throughput	< 0.1%
L1 Throughput	< 0.1%
RASTER Throughput	< 0.1%
SM Throughput	< 0.1%
VAF Throughput	< 0.1%
PD Throughput	0%
PES+VPC Throughput	0%
ZROP Throughput	0%

SM Occupancy (TPC View)	Throughput %	Avg	Avg Warps per Cycle
Idle SM Unused War...	99.8%	710250678	95.8
Pixel Warps	0.1%	914094	0.1
Active SM Unused W...	< 0.1%	335666	0.0
Vertex/Tess/Geometr...	< 0.1%	674	0.0
Compute Warps	0%	0	0.0

Figure 18: Unit Throughputs while loading rays into GPU memory

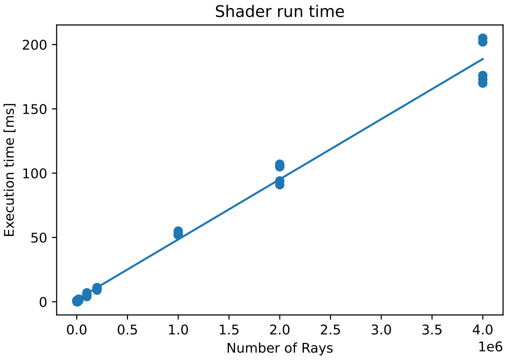
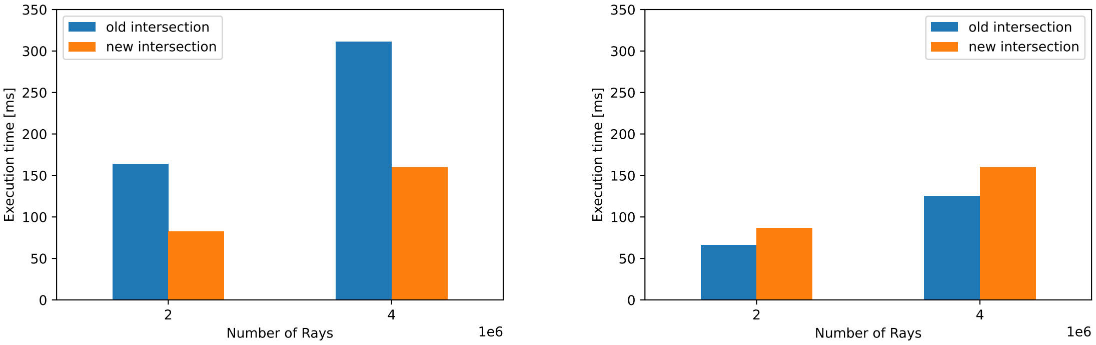


Figure 19: Time spent running the original shader

Isolated Testing of the intersectionPoint-Method To compare the updated method for finding the intersection point, multiple experiments have been conducted. For direct testing of the intersectionPoint-method, the code has been run isolated on the shader without any other calculations. However, due to branching problems and compiler optimizations this test is not sufficient to evaluate the performance. The methods are compared using mixed and bundled rays. "Bundled rays" means that rays in the simulation have a high change of pointing into the same direction. This regularly happens when a light source has a low dispersion. However, depending on the source and the optical elements this can change, causing the rays to point into different directions. The "mixed rays" for this evaluation have been created using a source which sends rays in all directions. As Figure 20a shows, the updated intersectionPoint-method performs better than the original method when using mixed rays. When using bundled rays, the old method performs slightly better. This is due to the nature of the original method for finding the intersection point. The method is split up into 3 branches, which interferes with the parallel computing abilities of the GPU. In the case that all rays in a warp point into the same direction, the GPU can ignore the branches that are not needed. This results in a fast computation of the intersection point.

However, when using mixed rays, multiple branches need to be evaluated, causing threads in a warp to stall, resulting in worse performance. Figure 20a shows that the new intersectionPoint-method is 99% faster than the old method when using mixed rays. As seen in Figure 20b, the new method is outperformed when using bundled rays. In this experiment, the old intersectionPoint-method was 31% faster than the new method. Comparing the performance of the old method using mixed rays with the performance of the old method using bundled rays shows, that using mixed rays results in a slowdown of 148%. The new intersectionPoint-method is much more stable, taking the same amount of time to compute the intersection using mixed and bundled rays. Although testing shows a speedup of 99% when using mixed rays, the execution time per 2,000,000 intersections is only 1.63 milliseconds for the old shader and 0.82 seconds for the new shader. Further experiments will show that the calculation of the intersection points will take longer when running the code in combination with the other calculations.

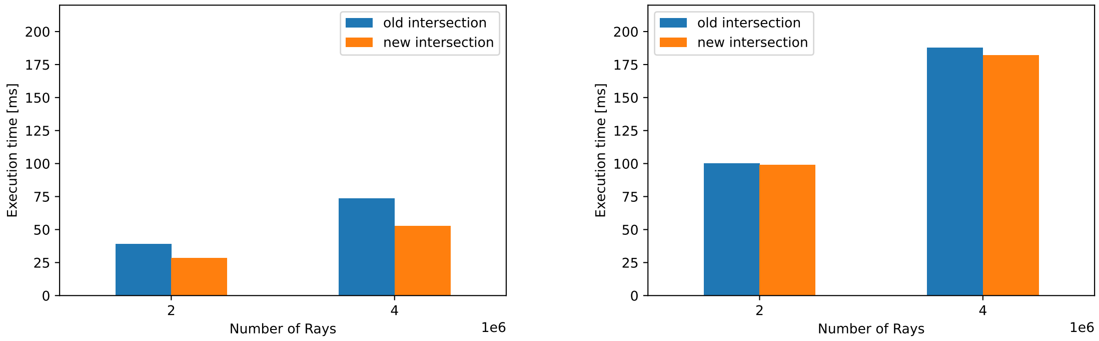


(a) Mixed rays

(b) Bundled rays

Figure 20: Execution time of the old and new shader running 100 intersections. Lower is better

Testing the intersectionPoint-Method with Beamlines The results obtained by testing the intersectionPoint-method using beamlines differ from the results shown earlier. The beamlines in this test are traced sequentially, i.e. dynamic ordering of optical elements is not used. As the beamline consists of one point source, one reflection zone plate and one image plane, only 2 intersection points need to be calculated. The data in Figure 20a and 20b indicated that the calculation of the intersection points is not time consuming, taking only 0.82 ms to 1.62 ms for mixed rays. Nevertheless, Figure 21a shows that using the new intersectionPoint-method does result in a significant speedup. Using the old intersectionPoint-method, the average execution time is 39.2 ms and 73.6 ms for 2 million and 4 million rays respectively. The new intersectionPoint-method achieves execution times of 28.6 ms and 52.8 ms, resulting in a speedup of 10.6 ms and 19.8 ms. For a beamline with mixed rays, the new intersectionPoint-method is 37% faster on average. Using bundled rays the execution times are longer than using mixed rays. This is because more rays pass through the optical elements, meaning that more calculations, e.g. intersections, reflections and refractions have to be done. Despite the expected slowdown indicated by the tests illustrated in Figure 21b, the old and new intersectionPoint-methods have similar execution times when using bundled rays. It can be concluded that the new intersectionPoint-method is up to 37% faster than the old method in some applications and equally fast in others.



(a) Mixed rays

(b) Bundled rays

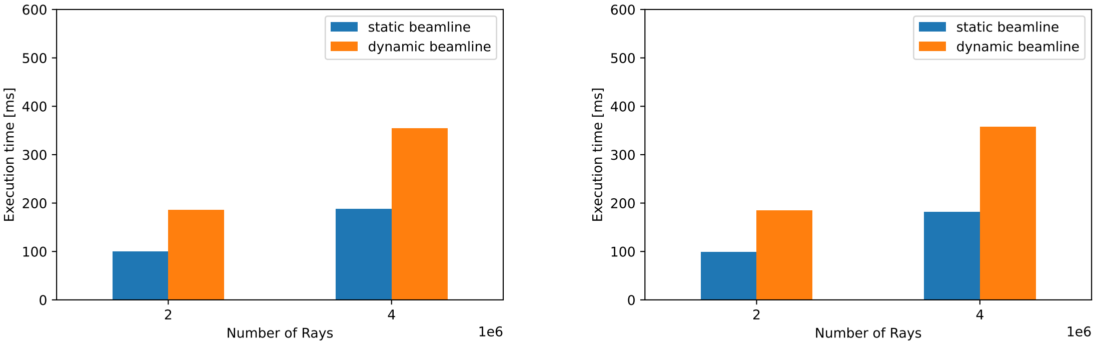
Figure 21: Execution time of old and new shader running a beamline. Lower is better

5.3 Performance of Dynamic Ordering of Optical Elements

Dynamic ordering of optical elements introduces a new feature, for which more calculations need to be done. This feature does not directly compete with the original static method concerning execution times, as for some tasks the new features are a necessity. If however the performance of dynamic ordering of optical elements is not much worse, it could replace static element ordering as the standard procedure.

Test Results Figure 22a shows the execution times of tracing a static beamline versus tracing a dynamic beamline. For this, the old intersectionPoint-method is used. Using dynamic ordering of optical elements results in significantly worse execution times. Tracing 2 million rays dynamically using 2 bounces takes on average 185.8 ms when using the old intersectionPoint-method. Compared to tracing a static beamline using the old intersection point, which takes 100 ms on average, dynamic tracing takes 85% longer. Using the new intersectionPoint-method, the execution time is 184.4 ms on average, which can be seen in Figure 22b. This is in line with the results obtained in chapter 5.2. Furthermore, it shows that both methods for finding the intersection points can be used as usual.

The beamline used for the experiment consists of 2 optical elements, as described at the beginning of the chapter. The execution time for dynamic tracing is 85% longer than the static execution time when tracing a beamline with 2 optical elements. For each bounce, the intersection point between each ray and each optical element is calculated, resulting in 4 calculations of intersection points in this example. When using a static beamline, only two intersection points need to be calculated, thus a 100% longer execution time is expected. The execution time of the dynamic tracing algorithm using the old and new intersection point being 85% longer is slightly lower than expected.



(a) Old intersectionPoint-method

(b) New intersectionPoint-method

Figure 22: Execution time of static and dynamic element ordering (2 bounces) using bundled rays. Lower is better

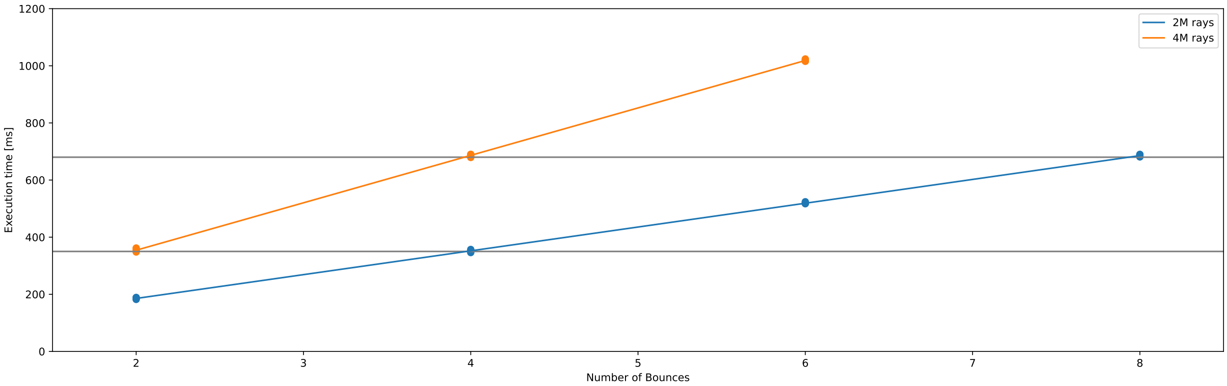


Figure 23: Execution Time Scaling with Number of Bounces

Figure 23 shows that the execution time of tracing using dynamic ordering of optical elements scales linearly with the number of bounces used. Furthermore, the data suggests that calculating one bounce for 2 million rays takes 90 ms. Figure 23 also suggests that calculating one bounce takes 180 ms for 4 million rays. It can be concluded that the computation time per ray is roughly the same for 2 million and 4 million rays. Moreover, the net number of intersections calculated seems to define the computation time, as it is not important whether the intersections are calculated because of the number of rays, or the number of bounces. This can be seen using the gray lines.

Calculating 2 bounces for 4 million rays takes roughly the same time as calculating 4 bounces for 2 million rays. The same behaviour can be observed when comparing 4 bounces with 4 million rays and 8 bounces with 2 million rays.

5.4 Performance of Tracing Multiple Beamlines

Tracing multiple beamlines in parallel promised multiple advantages. Firstly, the time spent initializing VULKAN is reduced. Figure 24 shows the execution time of beamlines traced with one initialization per trace and the execution time of multiple beamlines traced in parallel. The amount of beamlines that can be traced simultaneously is limited by the GPU's resources and most importantly the Windows Timeout Detection.

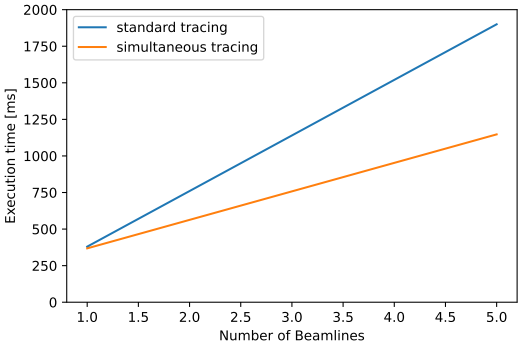
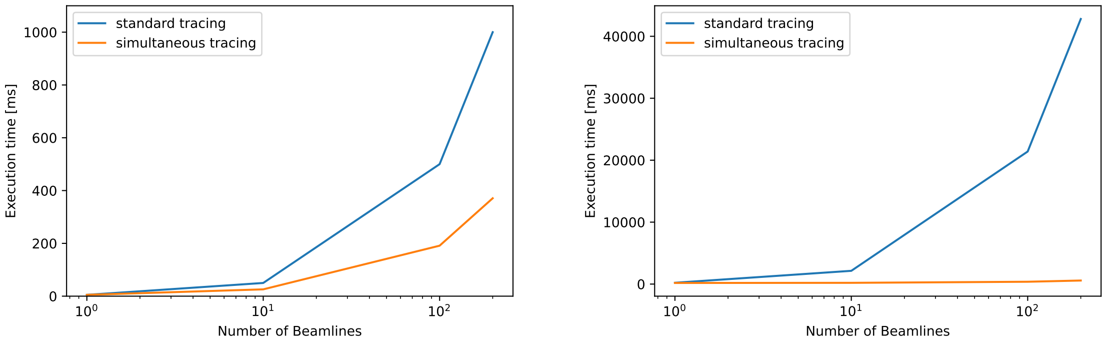


Figure 24: Tracing beamlines in parallel



(a) Shader execution time

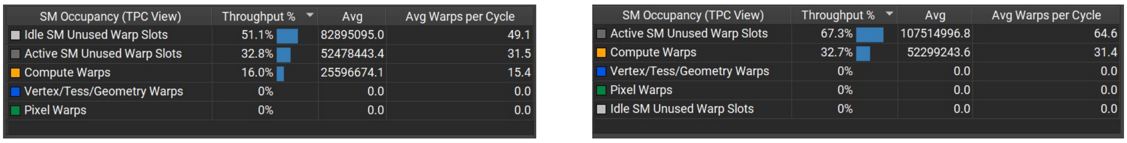
(b) Shader execution time + initialization time

Figure 25: Beamline with 20 thousand rays and two optical elements

The second advantage of tracing multiple beamlines in parallel is that the GPU's resources can be used more efficiently. Especially when tracing beamlines with a low number of rays, the GPU might not by fully used. Tracing beamlines in parallel provides the GPU with more work. As Figure 25a shows, tracing multiple beamlines with a low number of rays like 20 thousand does not use the GPU fully. A single run takes on average 5 ms (without initialization). If it were using the GPU fully, we would expect tracing 10 beamlines to take 50 ms. However, experiments show that tracing 10 beamlines in parallel takes only 26 ms on average.

Figure 25b shows the execution time of the shader and the initialization per beamline. When taking the initialization time into account, tracing only one beamline per initialization takes multiple magnitudes longer than the parallel tracing method.

The difference can be explained by the SM occupancy. Figure 26a shows the SM occupancy when tracing a single beamline with 20 thousand rays. This results in 51% of the warp slots belonging to idle SMs, while only 16% of the slots are occupied by compute warps. When tracing multiple beamlines in parallel, all SMs are active. Figure 26b shows that 32.7% of the warp slots are being occupied by compute warps when tracing 10 beamlines in parallel. The reason why 67.3% of the warp slots are unused is most likely because the SM's resources are fully used, thus preventing more warp slots to be occupied.



SM Occupancy (TPC View)	Throughput %	Avg	Avg Warps per Cycle
⬜ Idle SM Unused Warp Slots	51.1%	82895095.0	49.1
⬛ Active SM Unused Warp Slots	32.8%	52478443.4	31.5
🟧 Compute Warps	16.0%	25596674.1	15.4
🟦 Vertex/Tess/Geometry Warps	0%	0.0	0.0
🟩 Pixel Warps	0%	0.0	0.0

(a) SM Occupancy for one beamline

SM Occupancy (TPC View)	Throughput %	Avg	Avg Warps per Cycle
⬛ Active SM Unused Warp Slots	67.3%	107514996.8	64.6
🟧 Compute Warps	32.7%	52299243.6	31.4
🟦 Vertex/Tess/Geometry Warps	0%	0.0	0.0
🟩 Pixel Warps	0%	0.0	0.0
⬜ Idle SM Unused Warp Slots	0%	0.0	0.0

(b) SM Occupancy for 10 beamlines in parallel

Figure 26: Nvidia Nsight Graphics SM Occupancy statistics

6 Discussion

Improving the intersectionPoint-Method Initially the main focus of this work was to improve the method for calculating the intersection point. The extensive branching present in the method promised substantial performance gains. Out of multiple approaches, e.g. using sorting algorithms to prevent branching, the algebraic method seemed to be the best option. Using algebraic expressions to exploit the benefits of the SIMD/SIMT architecture appeared to be favourable.

However, implementing the method turned out to be much more challenging than initially expected. The interactions between GLSL, VULKAN, the SPIR-V compiler and LLVM when using e.g. dynamic indexing proved to be problematic. Furthermore, troubleshooting the problems was time consuming as there was no documentation available for some errors. This caused many workarounds to be required, which complicated the code, slowing down the method. We conjecture that the need to use SSBOs instead of simple arrays caused the performance of the algorithm to suffer.

Nevertheless, the new intersectionPoint-method improves the performance by up to 37%.

Dynamic Ordering of Optical Elements In contrast to improving an existing feature in section 3.1, dynamic ordering of optical elements is a completely new feature. The main focus was to develop an efficient method for tracing a beamline dynamically. The implemented method fulfils the requirements and allows simulating leaking rays and self-intersections. While the number of bounces is usually not much higher than the number of optical elements in the beamline, using a high number of bounces can result in longer than desired execution times.

To counter this a bounding box system could be implemented in the future. This allows using simpler methods for finding the closest optical element, thus reducing the execution time.

Tracing Multiple Beamlines Simultaneously Eliminating most of the initializations reduces the execution time noticeably. However, to reduce the time spent for initialization, tracing beamlines simultaneously is not the only option. It would also be possible to initialize VULKAN and then run multiple compute shaders consecutively, without the need to initialize VULKAN again. However, this approach does not solve the problem that the GPU is not fully utilized when tracing beamlines with a low amount of rays. In this case, simultaneous tracing presents a noticeable increase in performance, even when not taking the initialization time into account.

Nevertheless, the method has its limitations. Especially on the Windows operating system, the Windows timeout detection limits the execution time of a single shader call to two seconds. This limits the number of simultaneous tracing procedures. To counter this problem, a combination of simultaneous tracing and using multiple compute shader calls could be used.

Conclusion Depending on the use case, the improved intersectionPoint-method will replace the previous method. With more optimization it will most likely replace the previous method completely.

Dynamic ordering of optical elements is an important new feature that will help detect errors when designing a beamline. Furthermore, it enables simultaneous tracing, which depends on the dynamic ordering of optical elements. The combination will likely replace the static tracing procedure.

References

[1] *VULKAN.* (2021/07/29)
`https://www.vulkan.org/`

[2] *VULKAN Registry.* (2021/07/29)
`https://www.khronos.org/registry/vulkan/`

[3] *GLSL.* (2021/07/29)
`https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language`

[4] *Schäfers, F., 2008. The BESSY Raytrace Program Ray. Springer Berlin Heidelberg.*

[5] *Synchrotron Radiation.* (2021/06/27)
`http://abyss.uoregon.edu/~js/glossary/synchrotron_radiation.html`

[6] *What is synchrotron radiation?.* (2021/06/27)
`https://www.nist.gov/pml/sensor-science/what-synchrotron-radiation`

[7] *What the Heck is a Beamline (and why is it important)?.* (2021/06/27)
`https://www2.lbl.gov/Science-Articles/Archive/tabl-wonder.html`

[8] *Electron Storage Ring BESSY II.* (2021/06/27)
`https://www.helmholtz-berlin.de/forschung/quellen/bessy/index_en.html`

[9] *AMD EPYC 7763 Specification.* (2021/07/04)
`https://www.amd.com/en/products/cpu/amd-epyc-7763`

[10] *Video Game Industry Statistics.* (2021/07/04)
`https://www.wepc.com/news/video-game-statistics/`

[11] *Nvidia Revenue Trend.* (2021/07/04)
`https://news.alphastreet.com/nvidia-earnings-revenue-hit-record-highs-in-q1-beat-estimates/`

[12] *Sarker, I.H., 2021. Machine learning: Algorithms, real-world applications and research directions. SN Computer Science, 2(3), pp.1-21..*

[13] *42 years of Microprocessor Trend Data, Rupp, K, 2018.* (2021/08/04)
`https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`

[14] *Topics of the GTC 2021 Keynote.* (2021/07/04)
`https://www.nvidia.com/en-us/gtc/topics/`

[15] *Nvidia Ampere GA102 GPU Architecture.* (2021/06/27)
`https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf`

[16] *What are NVIDIA CUDA Cores?, Stuart, S.* (2021/08/04)
`https://www.gamingscan.com/what-are-nvidia-cuda-cores/`

[17] *Nvidia RTX 3090 Specification.* (2021/07/05)
`https://www.techpowerup.com/gpu-specs/geforce-rtx-3090.c3622`

[18] *CUDA C Programming Guide.* (2021/07/05)
`https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation`

[19] *Using CUDA Warp-Level Primitives, Lin, Y., Grover, V..* (2021/08/04)
`https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/`

[20] *McCool, M., Reinders, J. and Robison, A., 2012. Structured parallel programming: patterns for efficient computation. Elsevier.*

[21] *A Note on Branching Within a Shader.* (2021/07/25)
`https://www.peterstefek.me/shader-branch.html`

[22] *VULKAN annnouncement.* (2021/07/05)
`https://www.anandtech.com/show/8363/khronos-announces-next-generation-opengl-initiative`

[23] *Kenwright, B., 2017. Getting started with computer graphics and the vulkan API. In SIGGRAPH Asia 2017 Courses (pp. 1-86).*

[24] *VULKAN Registry: Shaders.* (2021/08/04)
`https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/chap9.html`

[25] *Overvoorde, A., 2019. Vulkan Tutorial. Vulkan Tutorial.*

[26] *Halladay, K., 2019. Practical Shader Development: Vertex and Fragment Shaders for Game Developers. Apress.*

[27] *Sellers, G. and Kessenich, J., 2016. Vulkan programming guide: The official guide to learning vulkan. Addison-Wesley Professional.*

[28] *Smits, B., 1998. Efficiency issues for ray tracing. Journal of Graphics Tools, 3(2), pp.1-14.*

[29] *Arvo, J., 1986, August. Backward ray tracing. In Developments in Ray Tracing, Computer Graphics, Proc. of ACM SIGGRAPH 86 Course Notes (pp. 259-263).*

[30] *Baumgärtel, P., Witt, M., Baensch, J., Fabarius, M., Erko, A., Schäfers, F. and Schirmacher, H., 2016, July. RAY-UI: A powerful and extensible user interface for RAY. In AIP Conference Proceedings (Vol. 1741, No. 1, p. 040016). AIP Publishing LLC.*

[31] *Baumgärtel, P., Grundmann, P., Zeschke, T., Erko, A., Viefhaus, J., Schäfers, F. and Schirmacher, H., 2019, January. RAY-UI: new features and extensions. In AIP Conference Proceedings (Vol. 2054, No. 1, p. 060034). AIP Publishing LLC.*

[32] *Dammertz, H. and Keller, A., 2006, September. Improving ray tracing precision by object space intersection computation. In 2006 IEEE Symposium on Interactive Ray Tracing (pp. 25-31). IEEE.*

[33] *NVIDIA Nsight Graphics.* (2021/07/25)
`https://developer.nvidia.com/nsight-graphics`

[34] *Nsight Graphics Activities - Advanced Learning.* (2021/07/25)
`https://docs.nvidia.com/nsight-graphics/AdvancedLearning/index.html`

[35] *Nsight Systems User Guide.* (2021/07/30)
`https://docs.nvidia.com/nsight-systems/pdf/UserGuide.pdf`

[36] *Lindley, C.A., 1992. Practical ray tracing in C. John Wiley & Sons, Inc..*

[37] *Ray Optics Simulation.* (2021/06/27)
`https://ricktu288.github.io/ray-optics/`

[38] *Windows Timeout Detection.* (2021/07/05)
`https://docs.microsoft.com/en-us/windows-hardware/drivers/display/timeout-detection-and-recovery`

[39] *Understanding GPU Caches.* (2021/07/05)
`https://rastergrid.com/blog/gpu-tech/2021/01/understanding-gpu-caches/`

[40] *The Industry Open Standard Intermediate Language for Parallel Compute and Graphics.* (2021/07/07)
`https://www.khronos.org/spir/`

[41] *SPIR-V, Extended Instruction Set, and Extension Specifications.* (2021/07/07)
`https://www.khronos.org/registry/spir-v/`

[42] *LLVM Is Playing A Big Role With Vulkan/SPIR-V Compilers.* (2021/07/07)
`https://www.phoronix.com/scan.php?page=news_item&px=LLVM-Vulkan-SPIR-V-Popular/`

[43] *Shader Storage Buffer Object.* (2021/07/07)
`https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object`

[44] *A first look at Unreal Engine 5.* (2021/06/27)
`https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5`

[45] *UE112_PGM-1 Beamline at BESSY II.* (2021/06/27)
`https://www.helmholtz-berlin.de/pubbin/igama_output?modus=einzel&gid=1649&sprache=en`

[46] *What is PCIe 4.0?.* (2021/06/27)
`https://business.kioxia.com/content/dam/kioxia/ncsa/en-us/business/asset/KIOXIA_CM6-CD6_FAQ.pdf`

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den August 10, 2021 ....