FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Accelerating Ray Tracing Using FPGAs

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Alexander Reznicek, BSc
Matrikelnummer 1125076

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Hiroyuki Sakai

Wien, 15. Juni 2020

Alexander Reznicek	Michael Wimmer

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Accelerating Ray Tracing Using FPGAs

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Alexander Reznicek, BSc
Registration Number 1125076

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Hiroyuki Sakai

Vienna, 15th June, 2020

Alexander Reznicek

Michael Wimmer

Erklärung zur Verfassung der Arbeit

Alexander Reznicek, BSc
Huttengasse 49/12
1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. Juni 2020

Alexander Reznicek

Danksagung

Bei dieser Diplomarbeit gilt es einigen im Umfeld der Universität besonderen Dank auszusprechen, da jeder dieser Beiträge diese Arbeit erst ermöglicht hat. Zu allererst möchte ich mich beim ECS Institut im Allgemeinen bedanken, die mir einen FPGA für die ersten Schritte zur Verfügung gestellt haben und bei so manchen kniffligen Problemen gerade am Anfang unterstützt haben. Der TU Wien selbst, respektive dem Dekanat, gilt allerdings genauso mein Dank. Der Erhalt des Forschungsstipendiums hat es erst ermöglicht, den in der Arbeit erwähnten FPGA auf einer PCIe-Karte für den vorgesehenen Ansatz einzusetzen.

Vom Institut für Visual Computing & Human-Centered Technology möchte ich Hiroyuki Sakai gleich in zweifacher Hinsicht danken. Zum einen, da er der Thematik mit großem Interesse begegnet ist und mich unterstützt hat in der Arbeit weiter zu gehen bzw. diese überhaupt zu starten. Zum anderen, weil er viel Unterstützung nicht nur beim Schreiben gegeben hat, sondern auch rundherum, bspw. kurzfristig einen Labor-PC zu organisieren nachdem mein Heim-PC überraschend kaputt ging. Ich möchte auch Michael Wimmer danken dafür, dass er die doch etwas außergewöhnliche Thematik ermöglicht hat, trotz der vielen möglichen Ungewissheiten die eine solche Arbeit mit sich bringt. Ich möchte an der Stelle beiden dafür danken, dass ich gerade am Ende schnelles Feedback zur Arbeit selbst sonntags erhalten habe, was ich beileibe nicht als selbstverständlich empfinde.

Im Freundes- und Familienkreis möchte ich zum einen meinen Eltern für die Geduld danken, was in Anbetracht der Dauer dieser Arbeit durchaus als bemerkenswert zu sehen ist. Auch wenn mir noch einige Freunde einfallen denen mein Dank gilt, möchte ich hier namentlich vor allem Michael Agbontaen nennen, der unter Einsatz seines PCs einige zusätzliche Tests durchführte, damit ich einige CPU und GPU Benchmarks durch eine zweite “Meinung” verifizieren konnte. Zuletzt ist auch das in der Arbeit verwendete Stromverbrauchsmessgerät eine Leihgabe von ihm und die Idee die Messung derart durchzuführen gemeinsam mit ihm entstanden.

Acknowledgements

For this thesis, I have to thank many people in the surroundings of the university, as each of these contributions only made the thesis possible. First, I want to thank the ECS institute, which has provided me an FPGA for the first steps and gave support for some tricky problems especially in the initial phase. I also want to thank the TU Wien, respectively the deanery, for the receiving of the "Forschungsstipendium", which just made this thesis possible by allowing to use the mentioned FPGA on the PCIe card for the designated approach.

From the Institute of Visual Computing & Human-Centered Technology I want to thank Hiroyuki Sakai on two counts. First, because he encountered the topic with much interest and supported me to go further on the thesis respectively begin its writing. Secondly, he not just supported me at the writing much, but also around that, e.g., by organizing a Lab-PC quickly after my own PC got broken surprisingly. I also want to thank Michael Wimmer, who made this somewhat extraordinary topic possible, even with respect to the various uncertainties this topic brings on. I want to thank at this point both of them explicitly that I got especially at the end feedbacks fast and even on sundays, what I do not see as implicitness.

In the circle of family and friends I want to thank my parents for the patience, which is, considering the duration of this thesis, notable. Even if I could name some friends whom I further have to thank, I want to name especially Michael Agbontaen for testing some CPU and GPU benchmarks to verify my own tests. Lastly, it was also his power meter which was used for the tests and the idea of using it for the tests was born together.

Kurzfassung

Die Synthese eines Bildes aus einer im Computer gespeicherten Szene ist das sogenannte Rendering, das beispielsweise mit einigen Vertretern der Klasse der Raytracing-Algorithmen fotorealistische Ergebnisse liefern kann. Diese Varianten (beispielsweise das Path Tracing) zeichnen sich allerdings durch eine stochastische Charakteristik aus, welche in einem hohen Rechenaufwand resultieren. Dies liegt in der Natur stochastischer Algorithmen, die durch eine hohe Anzahl an Stichproben ein Ergebnis berechnen–im Falle des Ray Tracing durch eine hohe Anzahl an Strahlen, die zur vollständigen Bildsynthese nötig sind.

Eine Möglichkeit um das Ray Tracing, sowohl in den stochastischen als auch in den simpleren Formen, zu beschleunigen ist der Einsatz von spezialisierter Hardware. FPGRay ist ein solcher Ansatz, der dabei die Verwendung von spezialisierter Hardware mit der Software auf einem handelsüblichen PC kombiniert um eine Hybridlösung zu bilden. Dadurch soll die höhere Effizienz spezialisierter Hardware genutzt werden und zeitgleich eine Zukunftsfähigkeit im Falle sich ändernder Algorithmen erreicht werden.

Die Ergebnisse deuten darauf hin, dass eine solche Effizienzverbesserung möglich ist. Allerdings war dies im Rahmen der Arbeit nicht realisierbar und die konkrete Implementation zeigte eine niedrigere Effizienz als reine Softwarelösungen. Die Möglichkeit der Erreichung einer höheren Effizienz durch diesen Ansatz konnte allerdings durch das Aufzeigen von FPGRays Potential sichtbar gemacht werden.

Abstract

The synthesis of an image from a scene stored on a computer is called rendering, which is able to deliver photo-realistic results, e.g., by using specific variants of the class of ray tracing algorithms. However, these variants (e.g., path tracing) possess a stochastic characteristic which results in a high computational expense. This is explained by the nature of stochastic algorithms, which use a high number of samples to compute a result—in case of ray tracing, these samples manifest in a high number of rays needed for a complete rendering.

One possibility to accelerate ray tracing—no matter if using a stochastic or simpler variants—is the use of customized hardware. FPGRay is such an approach, which combines the use of customized hardware with the software of an off-the-shelf PC to a hybrid solution. This allows increasing the efficiency by specialized hardware and delivers a sustainability in case of changing algorithms at the same time.

The results point towards a possible efficiency gain. Unfortunately, in the scope of this thesis this was not realizable and the specific implementation showed a lower efficiency compared to the software implementation. Nevertheless, the possibility to achieve a higher efficiency with this approach by indicating FPGRay's potential could be shown.

Contents

CHAPTER 1

Introduction

Rendering is the task of synthesizing an image from a model. Concretely, it refers to producing an image based on a three-dimensional scene which is stored on a computer in order to visualize the scene from a certain position and perspective (described by the so-called camera). This task is essential for many applications. Among the most famous examples are computer games and computer-generated imagenery (CGI) in movies. Another application is the prototyping and development of a variety of products ranging from cars to mobile phones in order to make the development process much cheaper. In medicine, rendered visualizations can help medical practitioners to detect diseases or even prevent them. Architects can render visualizations of houses and their interiors. The advent of virtual reality (VR) makes visualizations even more attractive for future applications, e.g., visualizing a new apartment with the desired furniture. Some of these applications aim for highly accurate results, some require fast results, and some even require both.

Nowadays, fast results can be generated using rasterization. Techniques that are based on ray tracing, on the other hand, can produce highly realistic results that are generally much slower. Ray tracing is a class of algorithms which rely on the propagation of light rays through the scene. Physically correct rendering is one of the reasons to use ray tracing instead of rasterization. In contrast to a rasterizer, a ray tracer enables physically based rendering, as the propagation of light rays is an abstraction of light transport in reality. Because of this, to accurately simulate lighting, no special cases need to be implemented for realistic rendering. Ultimately, this results in less complex code of a ray tracer compared to rasterization. For example, for mirrored surfaces, a rasterizer needs to compute a rendering of the mirrored objects to use it as a map which is used on the surface of the mirror to generate the final image. In contrast, a ray tracer simply reflects a ray if it hits a mirror. As the evaluation of the material on a hit point including possibly spawning a new ray is always done, no additional code is needed; the effect is taken care of implicitly. Algorithms such as path tracing based on ray tracing use stochastic methods

to generate photo-realistic images. This is not possible for all ray-tracing algorithms, as effects such as global illumination are not considered in basic algorithms of this class. Because of its photo-realism, path tracing is nowadays widely used for creating realistic CGI in movies and for many visualization tasks. But the stochastic approach of a path tracer comes with a significant drawback: A high number of rays is needed to render an image, thus leading to a high computational expense.

One crucial part of the rendering process is intersection: The task of finding the nearest object a ray hits along its way. The naive way of finding the first object that hit the ray is just testing all shapes in a scene for intersection with the ray by evaluating their respective intersection function. It is obvious that the complexity is linear in the number of objects. For scenes used in practice with millions of triangles, this is very slow. An option to make this more efficient is the use of space partitioning: Instead of testing every object, only a part of the scene is tested. Only if the ray intersects with this part of the scene, the containing objects are further evaluated. By recursively building partitions, this leads to a logarithmic complexity. Intersection acceleration makes intersections much faster, but it is still computationally expensive: When we compared the computation times of different operations for multiple scenes, intersections amounted to about 50% of the time.

Challenges. One way to accelerate the rendering process is the use of specialized hardware. Such systems consist of a custom-built design on a chip or protoyping platform to perform the ray-tracing computations. The hardware receives the scene data and renders the desired frame on its own. The key advantage is the increased energy efficiency and saving of logic resources (i.e., transistors on the chip) which is characteristic for tailored hardware designs. Furthermore, the throughput is often increased in comparison to the software-based approach. But all current hardware designs limit the rendering system such that it supports the rendering techniques which are pre-built only. As ray tracing is a topic of ongoing research, new techniques such as more accurate material models, filtering algorithms for the final image, or noise-reduction approaches are still under development. Traditionally, a new technique requires a new hardware design. Current hardware designs become obsolete fast if complex algorithms are superseded by newer alternatives.

Contributions. FPGRay is our approach for accelerating intersection by using a customized hardware design. In contrast to prior work, it was developed from the beginning to perform only parts of the rendering task. The use of a software renderer on a Central Processing Unit (CPU) is accompanied by FPGRay to accelerate specific operations only. The idea is to support only basic and often used operations to improve the efficiency of the rendering process. This also allows using fewer hardware resources by avoiding the implementation of rarely used operations. As the rendering still relies on a software renderer, such operations can be computed in software in case the frequency of usage does not make a more efficient hardware implementation appropriate. Moreover, if a new technique should be introduced to further optimize the rendering or achieve a

more accurate result, it can be done in software while the renderer can still make use of the hardware design. Our approach is similar to Graphics Processing Units (GPUs), which accelerate rasterization.

As a consequence, we also need a similar interface for data transfer as GPUs in terms of latency, throughput, and accessibility with interaction to a host Personal Computer (PC). Therefore, FPGRay is built on a Peripheral Component Interconnect Express (PCIe) card. The FPGRay card communicates via PCIe to an off-the-shelf PC, which can use the intersection function through an Application Programming Interface (API). The API can be used to easily add FPGRay's functionality to existing renderers. As an example, pbrt-v3 was extended with our functionality. pbrt-v3 also forms the foundation of FPGRay's algorithms.

Before we describe our approach in the following sections, we provide some background information on relevant aspects, in particular, about ray tracing in general, k-d trees, Field Programmable Gate Arrays (FPGAs), and hardware design in general. Afterwards, related works are briefly discussed. In the main section, the details about the implementation are presented. Finally, results, conclusions, and possible avenues for future work are described.

CHAPTER 2

Background

2.1 Ray Tracing Explained

Ray tracing is the task of tracing a ray through the scene while testing for intersections along its way and possibly spawning new rays on hit points of intersected objects. A traversal with multiple consecutive rays is called a path. The depth of a path is the number of rays that constitute the path and the number of bounces indicates the number of objects that were hit during traversal. Simple ray tracing algorithms use paths with a depth of only one. More complex algorithms such as recursive ray tracing [FW80] use paths with higher depth only for a few materials like glass. This omits some physical effects and leads to a lack of realism.

Even more complex algorithms based on path tracing facilitate physically based renderings that can not be distinguished from real photos. These algorithms are stochastic algorithms, so they are highly expensive in terms of computation. The stochastic approach is used in path tracing for numerically computing the rendering equation.

The Rendering Equation. Kajiya [Kaj86] developed a solid basis for physically based rendering—the so-called rendering equation. This equation generalizes the foundations of many rendering algorithms. As Kajiya stated, the idea of the rendering equation itself is not new. But other forms invented before, such as the radiosity equation or recursive ray tracing, use specialized forms which do not consider similarities between them. In the following, we do not focus on the equation as it is described in the original paper with rays going from one point to another. Instead, we focus on the commonly used formula described in terms of ray directions for path tracing, as it directly shows the behavior of the lightning simulation at objects.

The equation is given by

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i)(\omega_i \times \mathbf{n})\, \mathrm{d}\omega_i,$$

where

$L_o(\mathbf{x}, \omega_o)$ is the radiance[1] from position $\mathbf{x}$[2] in direction ω_o,

$L_e(\mathbf{x}, \omega_o)$ is the emitted radiance at $\mathbf{x}$ in direction ω_o,

$L_i(\mathbf{x}, \omega_i)$ is the radiance coming towards $\mathbf{x}$ from direction ω_i,

$f_r(\mathbf{x}, \omega_i, \omega_o)$ is the bidirectional scattering distribution function (BSDF) at the point $\mathbf{x}$,

$(\omega_i \times \mathbf{n})$ is the attenuation based on the incident angle of light. Because of the use of unit vectors this can be written as $\cos(\theta_i)$, where θ_i is the angle between the two directions. Finally,

$\int_\Omega ... \, \mathrm{d}\omega_i$ denotes the integration over the unit hemisphere Ω.

To properly calculate the light at any surface point, this formula needs to be evaluated. This means that the complete hemisphere Ω needs to be evaluated (see Figure 2.1). But this is not possible analytically due to the mutual dependencies of surface points. See Figure 2.2 for an example: To compute the radiance coming from the blue hit point, the incoming radiance from the red hit point to the blue hit point must be known. But to be able to compute the radiance coming from the red hit point, the incoming radiance from the blue hit point needs to be known in turn, as the blue hit point can deliver incoming radiance to the red one.

Bidirectional Scattering Distribution Function. The BSDF describes the scattering characteristics of a material. It is a function delivering the energy of light of an outgoing direction for an incoming light direction for a point $\mathbf{x}$. In practice, for each surface material an object can have, a distinct BSDF is used. In renderers, the outgoing direction is found by sampling a probability density function which approximates the BSDF. For example, diffuse surfaces equally distribute light over the complete hemisphere so the function chooses one of these directions. In contrast, functions for mirrored surfaces yield only one outgoing direction for one incoming direction.

[1] The radiance is the energy in watts coming through a unit area m^2 per unit angle sr (steradian)— $W/(m^2 \times sr)$

[2] For path tracers, $\mathbf{x}$ is a position on the surface of an object which is hit.

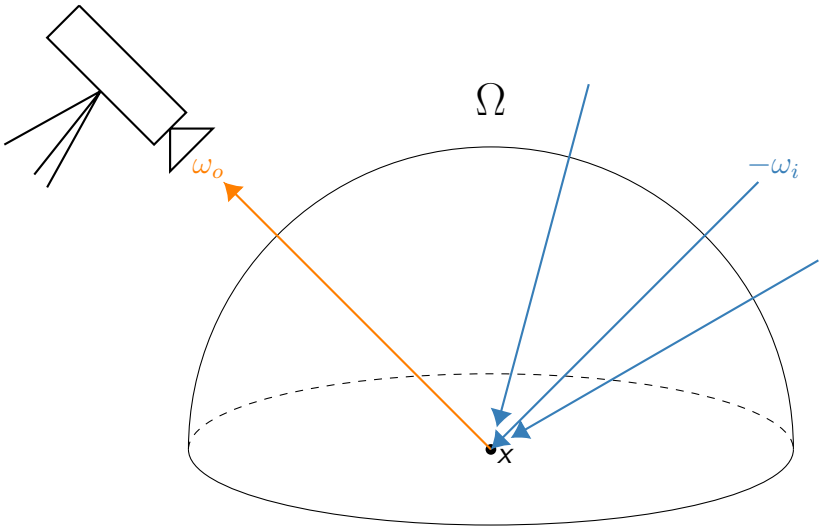


Figure 2.1: The hemisphere over which the rendering equation needs to be integrated. The output vector points towards the camera from the hit point where the incoming radiances from all directions irradiate (depicted by the incoming vectors).

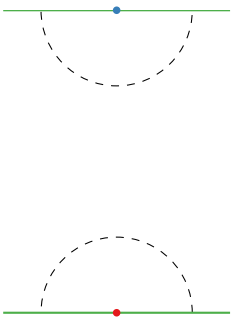


Figure 2.2: The two green objects surfaces are diffuse. If the integral of the blue hit point should be computed, one direction of the hemisphere comes from the red hit point, whose value is needed. The integral of this hit point in turn requires the integral of the blue hit point's integral. The integrals are mutually dependent.

Monte Carlo Rendering. The rendering equation can not be solved analytically. A widely used approach to solve integrals such as the rendering equation is by using a numeric integration algorithm which is known as the Monte Carlo method. It is also used by multiple ray tracing algorithms. When the rendering equation is solved by the Monte Carlo method, a high number of rays are shot from the hit point in arbitrary directions. This delivers enough information to compute the integral of the rendering equation with sufficient accuracy.

Kajiya [Kaj86] came up with an approach different to the idea of shooting a high number of rays from a hit point in arbitrary directions. Instead of branching and generating a

tree for each camera ray by spawning multiple rays at each bounce, only one direction is evaluated. This is referred to as a path (or a tree with branching factor 1). By shooting multiple paths from the same origin considering the probabilities of the BSDFs, different ways through the scene are evaluated by each path. This approach lowers the number of rays that are shot in later bounces and, differently to the straightforward Monte Carlo method, even shoots more camera rays (rays before the first bounce) than any bounce afterwards. This is owed to the fact that every path starts with a camera ray but can terminate at the first bounce. The approach of only generating up to one ray per bounce considers the fact that the first ray contributes most to the reduction of the variance of the pixel's integral.

Paths can be considered as samples in the numerical integration process. If the number of samples per pixel (spp) is high enough, the integrals are accurate enough such that the image is rendered satisfactory. Such an image is called converged. If the number of spps is too low, objectionable noise manifests in the rendering that reduces the quality of the result. Mathematically this is the result of using of too few samples for computing the rendering equation. The required number of spp is highly dependent on the scene and in practice, rendering a single frame can take up to days. Increasing the efficiency of this computationally expensive algorithm is an extensive topic in computer graphics, which is not constrained to improved sampling algorithms such as bidirectional path tracing (BDPT) and Metropolis light transport (MLT): Besides such improved sampling methods, there is also the group of filtering methods to make noise less prominent. Other works focus on the use of specialized hardware to gain greater efficiency. Increasing the efficiency is still subject to much ongoing research.

Path Tracing. For basic path tracing (without any improvements), the traversal of one sample can be described as follows (see Figure 2.3 depicting the elements of the explanation below).

A path starts at the camera with a ray shot into the scene. The direction of the ray from the camera is determined by the position of the corresponding pixel on the image plane deviated with a slight offset. This unique offset for every sample allows to get an evaluation of the whole pixel area. At the end, the samples are averaged to calculate the resulting pixel value. For the shot ray an intersection test is performed, i.e., every object in the scene is tested for intersection with the ray and the first intersected object along its hit point is returned. For the hit object, the BSDF of the surface material has to be evaluated. The evaluation may return with a different direction or output point than the incoming ray, so another ray is shot from the evaluated position and direction against the scene. This ray is intersected and evaluated exactly as the ray before (increasing the depth of the path). This is continued until a light source is hit or a stopping criterion is met, e.g., the maximum allowed depth is reached.

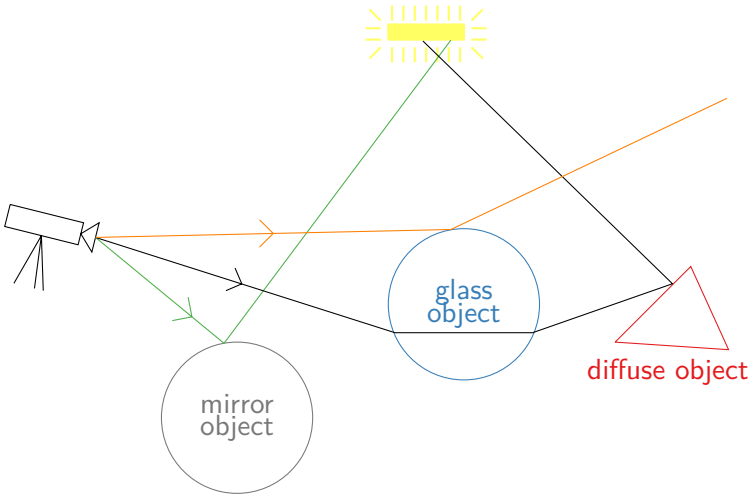


Figure 2.3: The basic parts of a path tracer. Three samples are shown. The first path (green) has one bounce at the mirror surface and then hits the light source. The second path (black) goes through a glass object (first and second bounce) and changes its direction towards the light at the last bounce (diffuse object). The third path (orange) does not contribute to the image as no light source is reached.

2.2 Intersection Acceleration With k-d Trees

Multiple algorithms for space partitioning exist. For ray tracing algorithms in general, the k-d tree and Bounding Volume Hierarchy (BVH) are the most widely used. BVH performs the same on average but the heavily used traversal routine is less complex for the k-d tree. We therefore chose the k-d tree for FPGRay (see Figure 2.4). In the following, we describe pbrt's [PJH16] k-d tree implementation, as FPGRay is based on it.

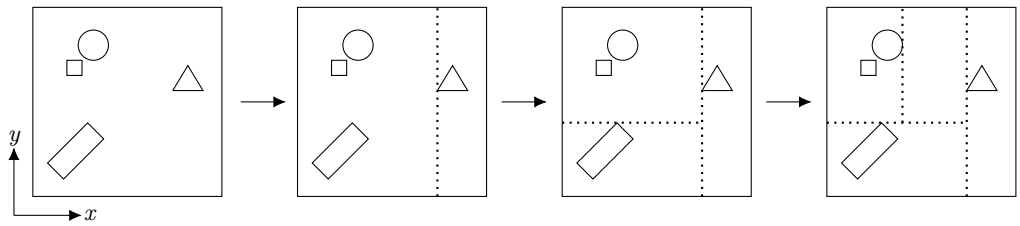


Figure 2.4: A k-d tree splits the scene recursively, ultimately enabling an efficient intersection of a ray with the scene. In each iteration, a particular axis is chosen and the scene is subdivided by a plane determined by the split position. Because of the partitioning, the tree is traversed in such a way that the nearest primitives along the ray's direction are intersected first.

The k-d Tree. As any tree in informatics, the k-d tree contains inner nodes and leaf nodes. The first node is the root node. All inner nodes are descendants of the root.

Nodes without any children are called leaf nodes. In rendering, the k-d tree contains the whole scene, which is recursively partitioned. The root node represents the whole scene, which is split into parts represented by child nodes. The splitting is performed recursively until the space in a child node is partitioned in such a way, that the maximum number of allowed primitives is not exceeded. If the maximum specified depth of the tree is reached, a leaf is generated instead of further evaluating the tree. In a k-d tree only one split per node is done, which means that the tree is a binary tree. A split is done along one of the coordinate axes at a certain position. Depending on the implementation, it is possible that one leaf can contain no objects at all for building a more efficiently partitioned tree. Emtpy leaf nodes are also used in pbrt [PJH16].

Construction. There are multiple ways to represent a scene with a k-d tree and the optimal representation cannot be found efficiently. Therefore, several heuristics have been developed for this purpose. The most famous one is the Surface Area Heuristic (SAH). It delivers a good approximation of the optimal solution. The heuristic makes use of the bounding boxes of primitives and inner nodes containing them to compute the surface area the bounding box of the complete inner node has.

The construction for each node's children is done by splitting all primitives in it into multiple bisections as candidates for possible split positions. After evaluating a cost function for each of the bounding boxes, the SAH uses the candidate with the lowest cost as child nodes. As this is done independently for every node, an optimal segmentation (i.e., over multiple nodes) may not be found (this makes the SAH to a greedy algorithms).

The costs are computed for the cases of splitting up a node at different chosen positions along an axis or to not split it up any further. The cost of not splitting a node any further is $\sum_{i=1}^{N} t_{\text{isect}}(i)$ where $t_{\text{isect}}(i)$ is the cost of the intersection of primitive i and N the number of primitives that need to be intersected in that node. The cost to split at a certain position is defined as $t_{\text{trav}} + (1 - b_{\text{e}})(p_{\text{A}} \sum_{i=1}^{N_{\text{A}}} t_{\text{isect}}(a_i) + p_{\text{B}} \sum_{i=1}^{N_{\text{B}}} t_{\text{isect}}(b_i))$. t_{trav} denotes the cost of a traversal, $t_{\text{isect}}(a_i)$ and $t_{\text{isect}}(a_i)$ are the costs of the intersection of primitive a_i and b_i in the corresponding child, p_{A} and p_{B} denote the probabilities that a traversal ends up in child A or B, N_{A} and N_{B} are the number of primitives of the corresponding children, and b_{e} is a factor for accounting children with no primitives in it.

The costs of the constants $t_{\text{isect}}(i)$ and t_{trav} depend on the complexity of one of these operations. Furthermore, pbrt does not use different costs for intersections but rather fix it to the constant t_{isect}, using precisely $t_{\text{isect}} = 80$ and $t_{\text{trav}} = 1$. This is done as the relative difference is needed only, so one constant can always be set to 1. The parameter valuesare optimized for pbrt's software implementation, especially for the high number of method calls. As the authors of pbrt state, most implementations use values closer together because of the use of lower call depths.

Our implementation of the intersection algorithm in FPGRay would also need t_{isect} and

t_{trav} values that are much closer to each other for optimal use of the heuristic. Scene trees tested with FPGRay are nevertheless generated with the same parameters if not stated otherwise. This also applies to the decision of using a constant t_{isect} cost. This facilitates a meaningful comparison between hardware and software implementation. The value b_{e} is 0, or if at least one child has no primitives in it, 0.5. By computing the surface area of the primitives of both children and dividing through the surface area of the complete node, the probabilities p_{A} and p_{B} can be calculated directly [PJH16].

Based on the costs, the SAH can easily determine the split position by choosing the split among a certain number of candidate positions depending on the lowest cost. It is sufficient to just use positions at edges of bounding boxes (see Figure 2.5) as candidates for splits. This is due to the fact that the algorithm should bisect primitives, for which it does not matter if the bisection is done directly at an edge or in between two edges. In cases where the algorithm is allowed to use empty leaf nodes, it can even increase the efficiency as the space between primitives is maximized. So the algorithm splits all candidates at each edge along an axis in two partitions, which easily gives a finite number of possible splits. The split with the smallest cost is used. The smallest cost of the heuristic results in a good bisection, even for later iterations. Note that the axis where the candidates are computed is chosen such that it is the one with the maximum extent. It is however possible for the k-d tree algorithm to check the other axes afterwards, if the cost function yields values that are close to the values that would have resulted without splitting.

The generation of child nodes is done the same way recursively until the maximum depth or the maximum number of primitives per leaf is reached. This results in leaf nodes which contain only up to the maximum number of allowed primitives. As mentioned above, leaf nodes with no primitives in it at all are also possible depending on the concrete implementation.

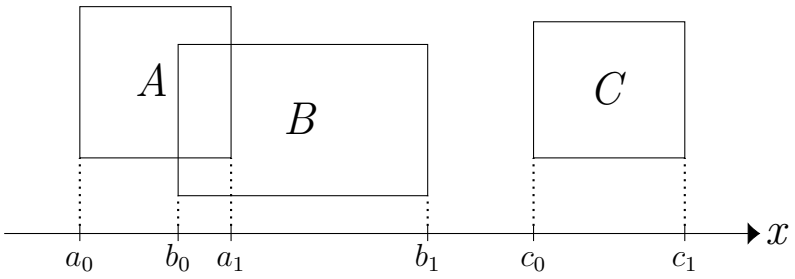


Figure 2.5: The SAH considers edges of the bounding boxes that are along the axis with the maximum extent. Since the computation of the cost function with a split along one of those edges results in the minimum cost achievable, intermediate positions do not need to be considered. With this approach, a finite number of split candidates can be easily and efficiently generated.

Traversal. Figure 2.6 illustrates the traversal algorithm. Note that the `ray finished?` decision checks if the stack is empty or if the next node on the stack has a minimum extent larger than the current `rtmax`. If one of those conditions applies, the intersection operation is finalized.

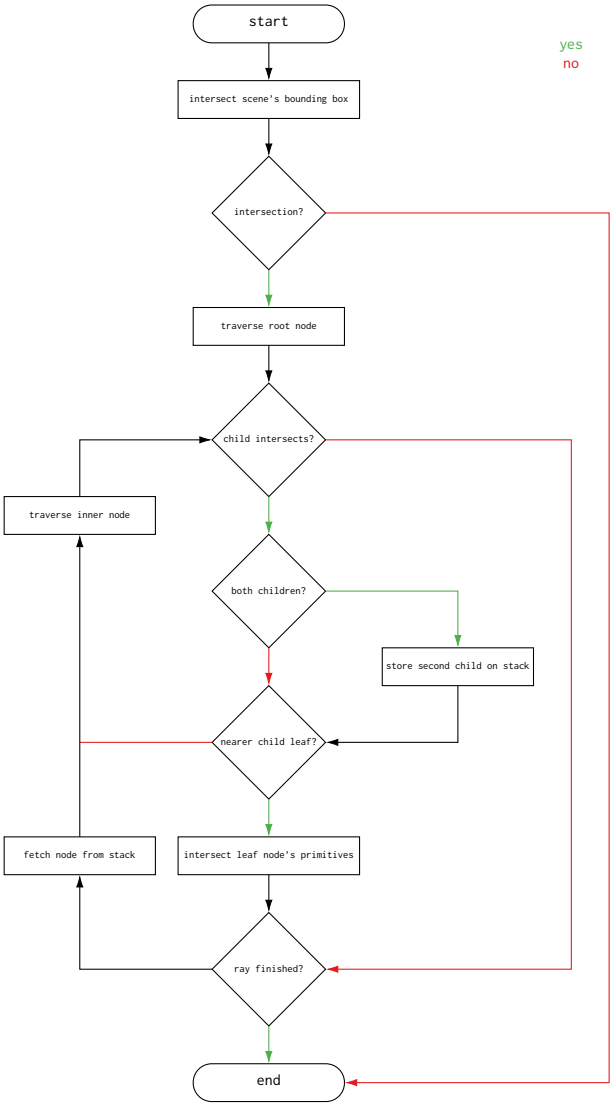


Figure 2.6: The intersection of a ray against a k-d tree.

2.3 FPGAs

For this thesis, hardware acceleration is used to establish a proof of concept that demonstrates a way to speed up the rendering process. The difference between software, which runs on hardware, or hardware alone is exactly the absence of software. A CPU

needs a program to properly compute the data while a dedicated hardware block only needs the data because the algorithm is implemented in the silicon.

Example: Dot Product. To compute a dot product in software, a program is needed which consists of multiplications and additions that are executed by a CPU by transferring the data to the arithmetic logic of it. But for every multiplication and addition the command has to be loaded, fetched, and the data needs to be loaded from RAM before it is executed. The operations also can not be parallelized well, so before the next dot product is performed, the actual one has to be finished first. Hardware, on the other hand, can pipeline the complete operation and has exactly the needed logic to give a proper throughput: One data block after the next can be handed over as input and the data is directly computed and routed from logic unit to logic unit until the result is computed. The significant efficiency advantage of hardware implementations is already exploited for high bandwidth applications such as video decoding.

ASICs vs. FPGAs. General purpose hardware such as CPUs or dedicated accelerating hardware such as GPUs are so-called Application-specific Integrated Circuits (ASICs). These are produced chips that consist of the designed logic built in silicon. The production of such an ASIC is costly and requires much time, which only pays off for mass production. This makes the use of ASICs infeasible for this thesis. Instead, we focus on using an FPGA.

The FPGA is a chip which can be programmed by the user to build up any desired hardware. In contrast to a program on a CPU, the loaded design consists of logic cells that run as the design instead of computing the program with general purpose logic. In practice, this is not done by a lithographic process on an empty silicon wafer such as for ASICs. Instead, the FPGA itself is an ASIC with many basic logic cells. The cells can be connected to the input pins or other cells via a complex routing scheme. This is also the reason why an FPGA is slower than an ASIC of the same manufacturing process running the same design. The overhead of this routing and the use of basic logic cells make FPGAs around a factor of 2-4 slower compared to ASICs. Furthermore, they also suffer from a 9-12 times higher power consumption and require a 20-40 times larger area [Ian06].

The performance increase by changing from an FPGA to an ASIC is always possible. Furthermore, modern FPGAs contain additional logic elements, such as multipliers, which deliver performance similar to that of an ASIC for the respective operation.

Source Code. The design of an FPGA is written in a hardware description language that looks similar to programming languages. There are two languages that are often used: Verilog and Very High Speed Integrated Circuit Hardware Description Language (VHDL), which is also used for this thesis. Because the thesis' design is primary using VHDL, further explanations about the program constructs focus on VHDL. It should be noted, that Verilog and VHDL can be used together in a design. This is also done in

FPGRay by using Intellectual Properties (IPs) from Intel which internally make use of Verilog.

An IP is a design made by others that can be used in an own hardware design. This is similar to a library in software engineering. Examples are trivial buffers such as the First in - First out (buffer) (FIFO), but also complex IPs such as complete CPUs. For FPGAs, basic IPs from the vendor should be used, as specialized logic elements, such as on-chip memory or multiplier units, can be accessed with them. This is not guaranteed when using plain code. Another speciality of FPGAs is the so-called HardIP. A HardIP is a logic element that is not built from basic logic elements, but an element providing a complex functionality, such as a complete Random Access Memory (RAM) controller.

Similarly to the compilation of a program, a hardware design gets synthesized out of VHDL or Verilog code. For writing and synthesizing FPGA designs, a so-called Electronic Design Automation (EDA) from the FPGA's manufacturer is used, which can be seen as the equivalent of an Integrated Development Environment (IDE) in software design. In case of the Arria V used in this thesis, the EDA is Intel's Quartus Prime [qua20]. It also delivers the needed IPs for this thesis.

Qsys. This thesis also makes use of Qsys, the system integration tool shipped with Quartus. It is used to automatize the integration by generating interconnect functionality between hardware components. With this tool, a complete computer system can be built on the FPGA. The Qsys system, built from hardware components itself, constitutes a hardware component that can be integrated in VHDL or Verilog code in the same manner as an IP or any hardware component written in plain source code. The advantage is that the interconnection between components inside of a Qsys system is done transparently. This means that by using standardized hardware interfaces—well defined combinations of signalling conduits facilitating data transfer—at hardware components, the system adds translation or synchronisation logic (called adapters by Qsys) between them if needed.

The standardized interface between the components that is used in this thesis is the so-called Avalon interface from Intel. It has two variants, one for connecting memory-mapped and one for streaming interfaces. In the memory-mapped variant, an interface designated as master can read and write data to and from a slave interface at a specific address, which can be used for, e.g., RAM access. The streaming interface variant is not using addresses and directly forwards data from the source to the sink interface when there is any ready. Both variants of interfaces allow many parameters which change the behavior of the interfaces, e.g., if an interface should be using a conduit indicating when it is ready for new data.

In this thesis, a Qsys system is created by combining the PCIe controllers to FPGRay's logic. Figure 2.7 shows the Qsys main window with FPGRay's Qsys system as an example.

To make use of the Qsys system with any hardware language, it needs to expose input and output conduits. Furthermore, to access the memory-mapped Avalon interface from

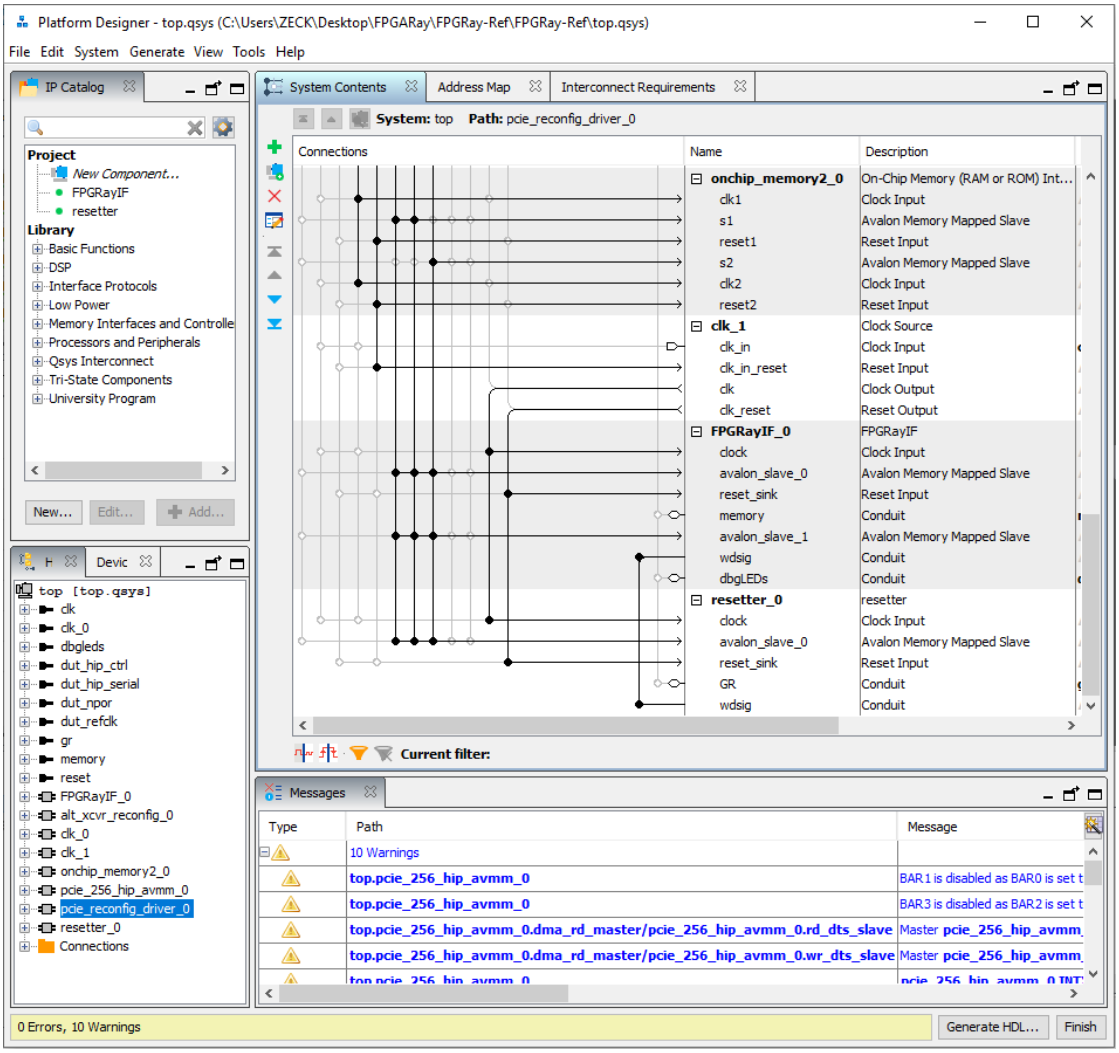


Figure 2.7: The Qsys Main window showing the actual Qsys system of FPGRay.

hardware in the software program, an address range is needed.

Hardware Synthesis. The synthesis of a hardware design for an FPGA consists of the following parts:

- Analysis and Synthesis: The VHDL or Verilog code is interpreted. The EDA tries to build the needed logic by mapping it to matching functional units. A schematic of the design is generated. Errors in code such as syntax errors or more complex errors such as wrong bit widths between connections or multiple assignments to the same conduit are found here. As VHDL is not only used for designing hardware,

some constructs are part of VHDL but not possible to build in hardware. Such non-synthesizable code is also caught in this stage.

- Fitting: The schematic is placed to specific units on the FPGA and routed together. The errors that can occur here are problems fitting the design properly. This can happen due to the use of special functions that are not available, connecting logic to the wrong pins of the FPGA, or just because there are too little basic logic units, memory, additional logic elements, or insufficient routing capacities.

- Assembler: Generates a programming file from the completed hardware design. This file is used to program the FPGA.

- Timing Analysis: The generated hardware design is analyzed and checked with respect to the desired speed (which is determined by the clock speed). For this, it is measured whether all routings between two registers (i.e., the logic elements that hold their output value for a clock cycle and only change them at the rising edge of a cycle) are short enough. This will not result in an error for the design. But if the design is too slow on the FPGA, the results can be delayed, ultimately generating wrong outputs.

Terms and Definitions. As a hardware design has different requirements and thus terms compared to a software design, a few definitions need to be explained.

A hardware component or logic block—analogous to a class in object-oriented programming—programmed in VHDL is called a **design entity**. It contains the **entity** defining the interface (the input and output conduits) and the **architecture** defining its behavior (the actual implementation). Both can be in the same file or split up in two distinct files. To expose a design entity such that it can be used as hardware component inside of other design entities, a **component** needs to be declared. It describes the interface of a design entity exactly as an entity, but is written in a scope that is accessible for other design entities. This is analogous to the declaration of a function in C such that other functions have access to it. To collect multiple components or separate them from the rest of the code–similarly to a header file in C–a **package** file is used.

If a design entity uses another design entity in its own code, it is called **instancing**. An IP can also be instanced exactly as a design entity, as long as it is made visible through a component. A QSys system can also be instanced, similarly to an IP or design entity. It should be mentioned that instances can also be a hardware component written in Verilog[3].

A conduit in VHDL is called **signal**. It is possible to merge multiple conduits to a **signal vector**. These vectors can be used to represent data flowing through logic. It can be roughly described as variables in software, but differently to variables, signals form logic that can be processed. Furthermore, as they are real conduits, a combination of signals

[3]In reverse direction Verilog is also able to instantiate VHDL design entities, IPs, or QSys systems

(and signal vectors) can build up an interface such as the already mentioned Avalon interfaces. Similarly to some programming languages using a keyword such as “var” for all variables, both signals and signal vectors are referred with the same keyword: In VHDL it is always called “signal”. The actual data type indicates whether only one signal or a signal vector is present, e.g., a signal containing a floating point value can be used.

Within the thesis, the term **port** is also used several times. A port is an interface, which concretely means that it consists of multiple signals combined together for data exchange between design entities. The important difference between the terms port and interface is that a port refers to a non-standardized interface (no Avalon interface variant or similar). Every time the term is used, the signals and with it the communication protocol (e.g., streaming or memory-mapping) are tailored to the connected design entities. In fact, the signals are not combined in any way in the source code: The port just identifies a logical combination which makes it easier to refer to belonging signals, e.g., all input signals of a design entity computing data in a streaming manner can be a port.

The top design file, also called main project file, has a special purpose. It is a VHDL or Verilog source file combining the whole logic. All design entities used in a hardware design need to be instanced or coded in this file. From a technical perspective, this is a normal source file, which could be used as an instance in a bigger project. For the EDA, it is the starting point of the synthesis. The input and output ports of the design entity determine the pins of the FPGA to connect peripherals. It should be noted that plain source files (both VHDL and Verilog) do not contain the pin mappings to identify which pins of the FPGA should be connected to which signals. This information is dependent on the EDA and for Quartus, stored in so-called `.tcl` files.

An architecture can be coded by either making use of structural coding, i.e., by connecting design entities together—which means they are instantiated inside the architecture of another design entity—or through behavioral coding. It is possible to mix both in one architecture to combine already implemented functions with new logic. Behavioral coding refers to the writing of logic in form of algorithms in so-called **processes**. The instructions in a process are executed sequentially, as it is done in a function in software. But in contrast to software, every process is working in parallel with all other processes. In the final hardware, each process shows up as dedicated logic besides other logic from other processes. Processes and instances of design entities can be connected together arbitrarily by just using signals between them (which again shows that signals are in fact conduits and not imaginary variables as in software).

This project also makes use of **constants**, **generics**, and **generate** statements. They are used for making the hardware design parameterizable, similarly to preprocessor directives in programming languages. All of them are evaluated during the synthesis process of an EDA program to customize the final design. Constants are used to set the number of buffer elements the design should have or—in combination with generics—to change the synthesized functionality, for instance. Generics are constants that can be used to uniquely parameterize each instantiation of a design entity. They are specified besides port mappings (the input and output signals described as the interface in an entity),

during the instantiation. The generate statements are the safe option to synthesize different hardware depending on the constants and generics values. With the "if generate" and "for generate" statements, instances can be generated only if they are needed and the generation of multiple instances is also possible. In practice, EDA programs optimize unnecessary logic out, which applies even to code which would be still present by not optimally using the generate statements. In this thesis and by the EDA, this is referred to as **synthesized away**. But the amount of code synthesized away is not standardized amongst EDAs. Depending on the EDA, code might not be synthesized away at all, even if possible. This results in a higher number of logic units that are used than actually necessary.

Synchronous Designs. A special type of process is the synchronization process. Different to other processes which always compute an output depending on the input, a synchronization process only changes its output once every clock cycle, often at the rising edge of the clock signal. Synchronization processes are used to introduce the clock and build up a synchronous design. Nowadays, most hardware is synchronous (e.g., a clock frequency of 4 GHz of a CPU means that the chip is processing the hardware logic driven by that clock 4 billion times in a second, or in other words the logic computes 4 billion output values from 4 billion input values) but on a chip it is possible to use different clocks and also exchange data between those clock domains (e.g., the cores of a CPU may operate at 4GHz while the on-chip GPU only runs at 1GHz).

The data flow of a synchronous design is a combination of synchronous processes and processes which do not trigger their processing on a clock signal. For every clock cycle, the synchronous process changes its output signals only once, presenting the following stage the signal values the previous stage computed over the last cycle. This means the computational processes which take a synchronous signal as input to return its output as input for a synchronous process has exactly the time of the clock cycle to completely process its logic. If it does not compute the value in time, the wrong value would be fetched by the synchronous process. Despite this drawback, synchronous designs are the common case nowadays as their design flow is well known and common tools are optimized for such designs.

For synchronous designs, the **latency** is the number of clock cycles a design entity needs to perform such that it can present valid output signals. So if a result can be computed in one cycle, the latency of the entity is one. The **throughput** on the other hand defines how often new data can be forwarded to the input. If a design entity accepts a new input date every cycle, its throughput is one. This means different to the latency, the throughput can be less than one. If the throughput should not decrease but the needed processing logic is too complex to be processed in one cycle, pipelining is used to split up the logic into more processes. With this change, the throughput will be the same in exchange for a higher latency. Each process in such a multi-latency design entity is then called stage; the data will be forwarded from one stage to the next via the synchronization process.

Porting Hardware Designs. VHDL code is portable with respect to the target hardware. If a design such as FPGRay should be used on a different FPGA, there are important steps needed to properly port the design, nonetheless. The main reason is the use of IPs. For the same FPGA vendor, they can often be reused and are regenerated by the EDA, otherwise they need to be changed to work. This means that the IPs need to be changed to other IPs having the same specifications (e.g., the same throughput). It should be noted that besides the VHDL code, there are additional files such as pin mappings that need to be adapted if necessary. VHDL is also used for ASICs, but any information regarding that, including porting from an FPGA, is out of the scope of this thesis.

Design Entity vs. Entity. Please note that in practice, the term **entity** is used as a synonym for the whole design entity and does not describe the interface definition. This is sufficient in practical use, as for the discussion about design entities and its functionality it is not necessary to distinguish between the entity and architecture. For the sake of simplicity, we adhere to this tradition in this thesis.

CHAPTER 3

Related Works

In the past, some other works already focused on the idea to use hardware to accelerate ray tracing. One of the first approaches was SaarCOR [SWS02]. It uses a k-d tree for accelerating ray tracing. The aim of the project was to use the hardware design for rendering a computer game in real time. This was possible by using the computationally less expensive and non-probabilistic recursive ray tracing algorithm [FW80]. In contrast to FPGRay, it was only simulated, so they used no board and all benchmarks were based on timing-accurate simulations.

Woop et al. developed RPU [WSS05], which also only accelerates recursive ray tracing. The used acceleration structure was a k-d tree and it was implemented on an FPGA. In contrast to these two implementations, FPGRay aims at accelerated intersection tests tailored for the needed precision and high number of intersections in photo-realistic rendering, i.e., path tracing instead of recursive ray tracing.

The T&I-Engine [NPP+11] was the first approach using Single Instruction, Single Data (SISD) to compute each ray independently. Similarly to SaarCOR, it only relied on a simulation for evaluating the design and accelerated the intersections using a k-d tree. Although its aim was real-time rendering based on recursive ray tracing, the design can be used for basic path tracing too. To support those class of algorithms, techniques such as sampling and path termination are additionally needed.

SGRT [LLN+12] is based on both a simulation and an FPGA for benchmarks and is able to perform path tracing although it was mostly used to benchmark recursive ray tracing. In contrast to all other mentioned works, including FPGRay, it uses a BVH to accelerate the intersection function. SGRT uses a combination of a simplified processor for ray generation and performs shading on an ARM CPU of a mobile device for simulation. The FPGA design handles the scene transfer via Universal Serial Bus (USB) to the prototype.

RayCore by Nah et al. [NKK+14] is the last paper presented here. It also supports path tracing and was tested with it, but its primary aim was the use of real-time recursive ray

tracing for mobile devices with low power consumption. An interesting comparison was done by not using only an FPGA for benchmarking, but also an ASIC, which gives a good approximation of how much performance gain is possible with an ASIC compared to an FPGA.

In contrast to all papers presented here, FPGRay is not aimed at real-time rendering. The focus lies on a high number of rays per scene as it is needed for photo-realistic rendering. FPGRay is designed as a hybrid solution where not all functionality (i.e., shading, ray generation) is computed in hardware. Its intended use is the acceleration of a software renderer running on the CPU of a PC. In contrast to that, all presented papers use a dedicated solution for shading and ray generation too. Some use a fixed-function design (SaarCOR and RayCore), some use a simplified processor for it (RPU and T&I-Engine). This enables them to render a complete frame on chip, which is not possible with FPGRay. It should be mentioned that the hardware design of FPGRay is the only work which is intended to be flexible, allowing to build hardware designs with different number of computation instances.

Furthermore, it should be noted that NVIDIA released the NVIDIA RTX cards in 2018. In contrast to all other works described in this section, they are consumer products and not scientific research projects. This also manifests itself in limited information about the specifications. What is known is the fact that the eponymous RTX cores are fixed-function hardware for computing the intersection of a scene saved in a BVH. Benchmarks and other comparisons are omitted here as at the writing of this thesis, the RTX cards were too new. RTX cards also aim at recursive ray tracing for improving effects of normal rasterized computer games. It is also not known how much of the needed work for path tracing can be done on such a card and how it can be used without the need of using DirectX or Vulkan.

CHAPTER 4

FPGRay

FPGRay is the complete implementation of a hardware able to accelerate ray tracing computations. Besides the hardware design, the implementation consists of corresponding software to access the hardware from a PC. FPGRay can be used by software renderer to let operations of the ray tracing algorithm be computed in hardware. This implies a hybrid approach using a combination of specialized hardware and software. In contrast to previous work using hardware acceleration (or the software-only approach), multiple advantages speak for the use of a hybrid system:

- Flexibility: If new functionality is needed, e.g., a new material model, this can be easily added in software without changing the hardware. In contrast to a hardware-only solution, this allows a sustainable approach which distributes the initial implementation costs over a longer period.
- Efficiency: Using hardware acceleration for higher efficiency is a good idea, but it does not make sense to implement all functionality as a hardware design. Computing inherently sequential operations in software also saves expensive hardware resources. However, the implementation of all needed functionality for rendering in hardware was needed in previous works, as in case of the hardware-only approach this was the only possibility to enable the rendering at all.
- Utilization: Nowadays, CPUs with many cores combined with much system memory are common. If hardware acceleration is used, these resources have reserves that can be used for other computational tasks the hardware is not able to perform.

In light of these points, we identify intersecting the ray with the scene using an acceleration data structure (in our case, a kd-tree) as the most important operation for ray tracing. In a ray tracing system, they are always required, regardless of the actual rendering algorithm. They are needed for every ray, which makes them a heavily used operation.

As already mentioned, intersections amounted to almost 50% of the overall rendering time for the tested scenes—despite the use of acceleration structures (k-d trees and BVHs). The only (negligible) drawback is that intersection is not a well predictable operation as the number of iterations needed to traverse the acceleration structure completely is highly dependent on the scene and the particular ray.

The combination of the flexibility and efficiency advantage over previous works is a direct consequence of FPGRay's design. Instead of facilitating real-time rendering with ray tracers, FPGRay aims at photo-realistic rendering, which cannot be computed in real-time as of now. However, the use of new techniques is crucial to achieve appropriate photo realism. The only viable option is to implement such techniques in software assisted by hardware as the needed hardware implementation to support those would be too laborious.

A key characteristic of the hardware design is the possibility to change the number of processing units to tailor the design to different FPGAs. This degree of flexibility should not be seen as an advantage over current approaches. It is more the result of the development which needs this parameterization for utilizing the FPGA resources the best way. The possibility to even add further functionality to the hardware design without needing to change the actual hardware (i.e., the FPGA card) is a property of the FPGA itself and would apply to all previous FPGA-based works. The only difference is that previous works do not seem to be adaptable with the same ease by just changing constants, but this is independent of the mentioned contributions.

To achieve higher efficiency by making use of parallelization, multiple rays at once can be sent as a batch. The implementations of intersection and traversal algorithms are based on pbrt-v3, which manifests itself in many similarities between pbrt and FPGRay on the algorithmic side.

Overview

Figure 4.1 depicts a rough overview of FPGRay's composition. If a program (`rendering SW`, such as pbrt; not part of FPGRay) wants to use FPGRay, it has to establish a connection via the C++ `API` (Section 4.6). The API abstracts the data transfer and connection handling to circumvent the direct use of the driver. The `PCIe driver` (Section 4.5) is responsible for transferring the data and commands at kernel level to the hardware. The complete communication between FPGRay and the software on a PC is done via the `PCIe interface`. On the hardware side, after the low level communication, the data ends up at `FPGRayIF` (Section 4.1). In this entity, all instances used for computation of FPGRay's functions are present. The data is computed here and then kept available to be read back from software side.

With the depicted configuration, the hardware design can be used for intersection tasks. In the default parameterization, FPGRay allows ray-scene intersections against a scene stored in a k-d tree, which is its intended main operation. By changing the

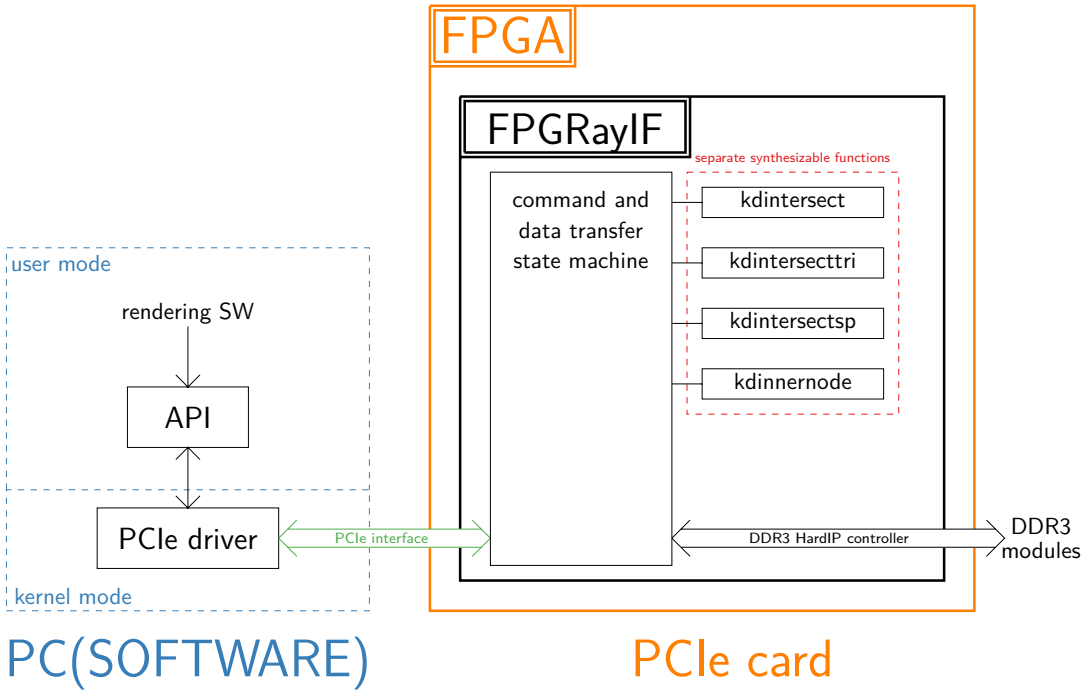


Figure 4.1: FPGRay's schematic

parameters before synthesis, parts of the intersection procedure can be done alone too. The intersection operation for a scene stored in a k-d tree is done in three steps:

1. Load the tree's data to the RAM of the FPGA.
2. Configure the intersection instance by specifying the tree information and the memory locations of the tree transferred before.
3. Intersect rays with the loaded scene.

From the API side, a user program can use FPGRay's functions by calling the corresponding API function. The data is handed over as a continuous memory space.

As basic functionality, the FPGRay hardware design always supports the communication via the PCIe interface and the data transfer to and from the on-board DDR3 RAM. An actual hardware design on an FPGA can support the intersection of a k-d tree (`kdintersect`, Section 4.2; synthesized by default) or optionally only a part of the k-d tree intersection, if synthesized with the corresponding parameters[1]. The optional operations are legacy operations which are partly still used by `kdintersect`, namely the intersection of a triangle (`kdintersecttri`, Section 4.3.2), intersecting a sphere

[1]Precisely, by setting VHDL constants located in FPGRayIF.

(`kdintersectsp`, not used any more by `kdintersect`), or the traversal of a node of a k-d tree (`kdinnernode`, Section 4.3.1).

Protocol. The communication protocol is very basic. FPGRay just awaits a command including the number of data blocks that should be computed. After that, the data blocks are sent as raw data without separators or terminators.

Lower Level. To access the PCIe interface on the software side, a Linux kernel module [ALK, LKM] as a driver is needed. On the hardware side, to make use of the PCIe interface, the FPGRayIF entity is supplemented by IPs from Altera. The actual composition of this hardware design, which is abstracted in the (Figure 4.1) by the green arrow, is further explained in Section 4.7.1.

The communication on software side is done by writing and reading to a special memory location, exactly as it is done when accessing RAM. This can be done because the driver is mapping this location to the PCIe interface, ultimately transferring data to this interface instead of accessing the RAM. To get a higher throughput, a Direct Memory Access (DMA) is used for transfer. This means, instead of transferring data word-wise, the PC's hardware copies a block of data at once.

The next sections contain a more detailed overview over the different components of the hardware design. After that, the PCIe driver and the API are explained in more detail. At the end, some implementation details are also discussed.

4.1 FPGRayIF

FPGRayIF (FPGRay InterFace) is the entity that accepts data from the PCIe interface on the hardware side and forwards the data to the demanded entity. For being able to determine the correct entity which should be fed with the data, a state machine is used. The state machine gets to know its state by the first data word of a data transfer, which is containing the command. The state machine then forwards the amount of data which was announced by the command. In the final step, the state machine returns to its idle state to await the next command.

As already mentioned, some operations can be chosen to be not synthesized in the final hardware design. This helps to save resources, but if such an operation would be initiated, the PC could freeze. To overcome this problem, the state machine automatically mimics the data flow of unsynthesized operations and generates data blocks with dummy values to ensure a safe operation.

Operations which are finished save their data into an output buffer within FPGRayIF. The software takes the data from this buffer via the PCIe interface. A special purpose of the output buffer is to ensure that the data is always in-order. This means, the order of the rays sent from software should be kept when the results are received. As a k-d tree

intersection needs a different number of tree traversals and intersections for each ray, the runtimes can vary greatly. FPGRay's implementation kdintersect (Section 4.2) outputs a finalized ray without taking care of the order. To overcome this problem, the output buffer has a resorting functionality, automatically storing the ray in the correct position and not in FIFO manner.

4.2 kdintersect

The main part of this thesis is the implementation of the intersection routine for a k-d tree. The concrete implementation is based on pbrt's `KdTreeAccel` class and the `Triangle` class. The decision for using pbrt was a practical one. Both the code basis and the documentation (i.e., the book [PJH16]) are intended to be educational. Thus the technical details are extensively discussed, including the problems of computational errors due to the finite precision of floating point numbers. pbrt is also using a simple code base. In contrast to that, many renderers, such as Mitsuba [Jak10], are highly optimized. Because of the different workflow needed for FPGRay compared to a software-only approach, those optimizations are a disadvantage. Another advantage is the relation of pbrt with LuxCoreRender [lux20]. LuxCoreRender is based on pbrt-v1 and uses the same class structure as the pbrt-v3 version used by this thesis. This makes another renderer besides pbrt, which is more optimized, easily extendable for the use with FPGRay.

kdintersect only implements the intersection (the traversal of the k-d tree and the intersection of the contained primitives) and is not able to construct a k-d tree scene from primitives. The absence of a hardware-accelerated tree construction operation is due to the fact that the used SAH algorithm is not parallelizable and sufficiently fast in software. The reason for this is that the tree construction is only done once per scene, but the intersection is performed multiple million, possibly billion, times. So the possible acceleration of the construction is negligible especially in comparison to the hardware resources that can be better used for the actual intersection. It should be noted that other algorithms to construct k-d trees in real time exist [ZHWG08, CKl$^+$10], which could further improve the rendering efficiency. Moreover, for dynamic scenes (e.g., when using animations) this eliminates a possible bottleneck in case of highly varying scenes.

Basic Characteristics. kdintersect is the complete implementation needed to intersect a ray against a scene's k-d tree. It is therefore instancing many entities to achieve a better structure, which will be explained further in the following sections. The kdintersect entity performs initial computations directly in its entity, which is further described in Section 4.2.1. The initial computations are done to intersect the bounding box of the scene's k-d tree to check whether the ray is intersecting anything at all.

kdintersect takes a ray consisting of the origin, direction, a unique ray identification number (`rayid`), and the `rtmax` value of the ray as input. The usage of the `rtmax` value originates from pbrt, where it defines the maximum distance at which a ray can get intersected (the range of the ray). Initially, it has the value of infinity and is used during

intersection to save the distance of the nearest intersected primitive. `rtmax` is one of the output signals alongside the `primitiveid`, the identification number of the intersected primitive. Furthermore, the ray's origin, direction and `rayid` will also be forwarded to the output. The `rayid` is crucial as the order of a ray is not retained by this entity and the resorting at the end will be done based on it.

Structure. kdintersect has three types of entities which do the actual computations necessary for intersection. All of them are derived from parts of pbrt's `KdTreeAccel` intersection functionality and the triangle intersection function. The two entities performing the operations of intersecting a triangle (kdintersecttri, Section 4.3.2) and traversing through an inner node of the k-d tree (kdinnernode, Section 4.3.1) are distinct entities which can also be used multiple times to increase the throughput of the kdintersect entity. These distinct entities are not directly instanced in kdintersect, but rather in the kdscheduler entity (Section 4.3). The third part is not a distinct entity but rather kdintersect's preparation part, which performs the initial computations needed for traversing through the scene. As the check of the scene's bounding box against a ray needs to be done only once, this computational logic can reside as a singleton in kdintersect. Figure 4.2 shows the schematic of kdintersect. Besides the preparation being computational logic directly written in kdintersect, only one entity is directly instanced: kdscheduler, which encapsulates all logic needed for the actual traversal and intersection.

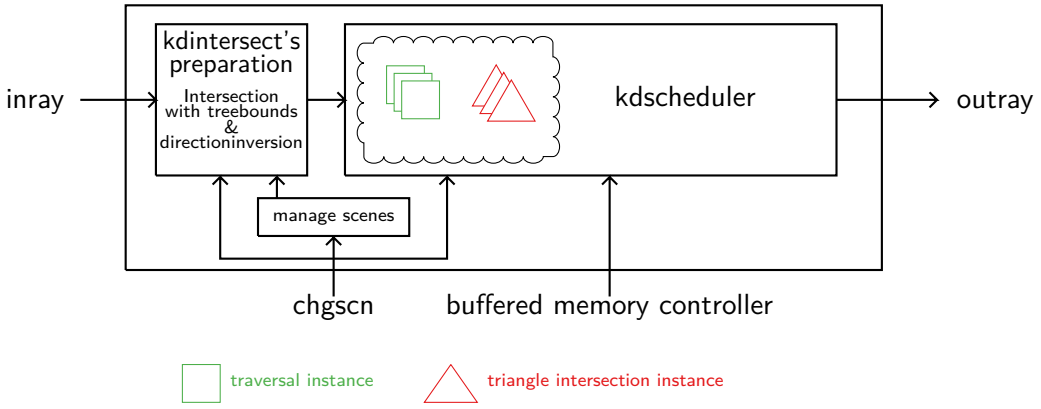


Figure 4.2: The schematic of kdintersect.

Restrictions. In comparison to pbrt's intersection routine, there are a few constraints owing to technical reasons:

- The k-d tree is the only intersection routine that is supported. BVH or direct intersection of a triangle via kdintersect is not possible.[2]

[2]However, a triangle could be intersected directly by using FPGRay's own kdintersecttri entity (Section 4.3.2).

- Only one k-d tree containing the whole scene is supported. Nesting of multiple k-d trees is not supported.
- Other shapes than triangles are not supported.[3]
- Alpha textures are not supported in any primitive.

Specialties. kdintersect offers a high degree of flexibility with regard to resource usage. It allows to tailor the design depending on the targeted scenes and the FPGA's resources using various parameters. As an example, the number of intersection and traversal instances can be parameterized.

kdintersect is able to process rays of multiple scenes in parallel. The support for multiple scenes in parallel is important for animations, where a scene can change from one frame to another. For identifying the scene a ray should be intersected, the `sceneid` of the ray is used, which is set by the "change scene" signal. This functionality is implemented to ensure a proper utilization even in case of a scene change. Without it, a ray of a new scene would need to wait for entering kdintersect until the last ray of the scene before was completely intersected. As an arbitrary ray can take longer than other rays, only one ray might be intersected for a long time, lowering the throughput.

The following sections describe the used entities inside kdintersect in more detail.

4.2.1 kdintersect's Preparation

Before entering the tree, the bounding box of the k-d tree is checked for intersection. Only if this intersection is present, an intersection of the ray with the scene is possible. The logic for computing this bounding box intersection is located in kdintersect as preparation logic before feeding the data into the kdscheduler (Section 4.3). See Figure 4.3 for the concrete logic. The kdscheduler gets the result as a bit signal (i.e., the output of `isnt_false_already`). The inverted directions of the ray (`rinvdx`, `rinvdy`, and `rinvdz`) are additionally forwarded to the kdscheduler to the input, as these values are needed for further computations. The computation of the bounding box intersection is done using pbrt's geometry class implementation for bounding boxes.

4.3 kdscheduler

The kdscheduler is the entity connecting all entities together to properly intersect a ray with a scene stored in a k-d tree, except the test if a ray intersects the scene at all (which is located in kdintersect, Section 4.2). This includes entities for traversing inner nodes, fetching nodes and primitives, and intersecting primitives.

[3] However, a sphere could be intersected with FPGRay's own kdintersectsp entity.

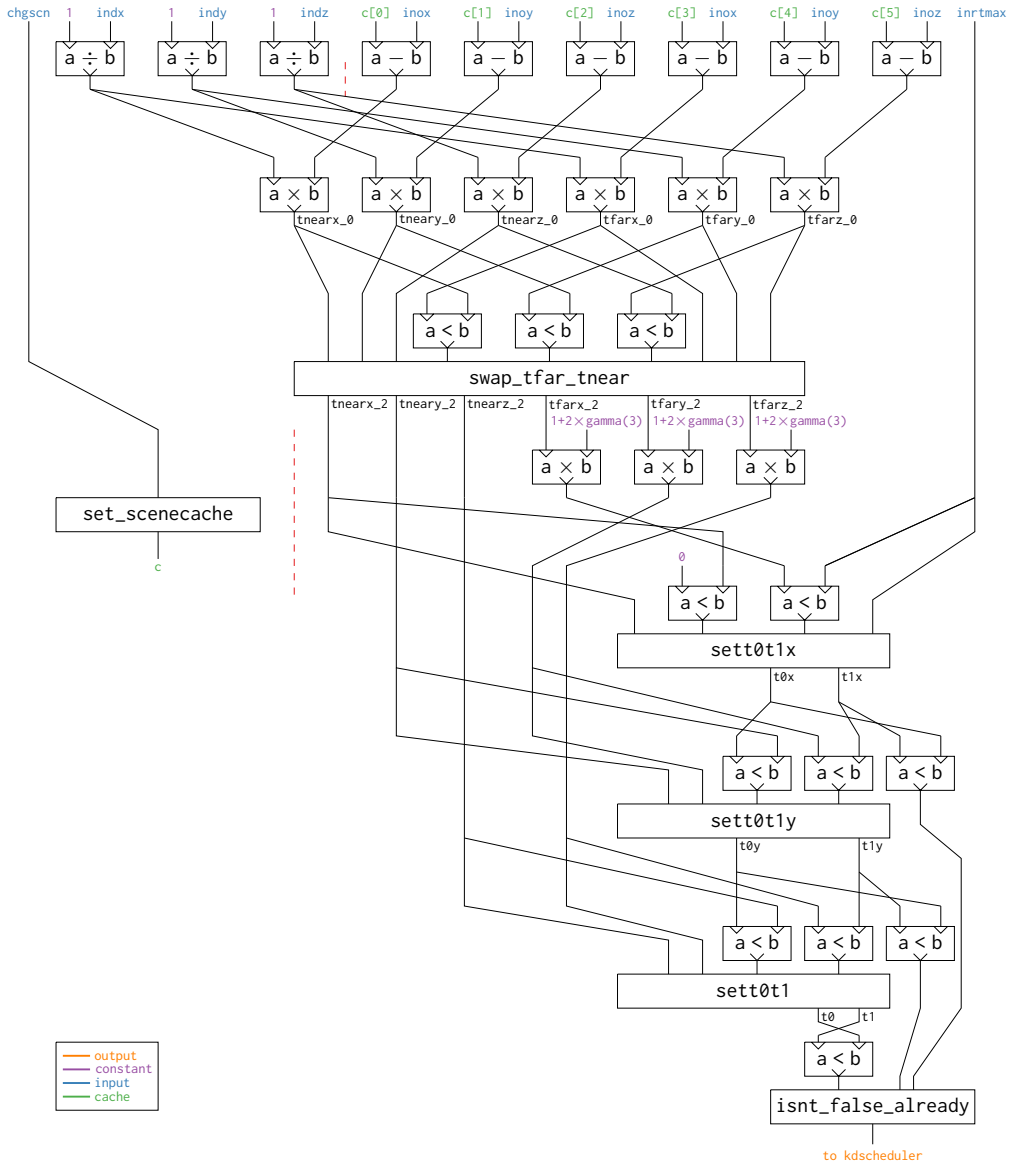


Figure 4.3: The logic needed to compute an intersection of a bounding box with a ray. Note that any conduit at the input or output of an operation or process on the same height as other conduits is in the same cycle (i.e., synchronized). The only exceptions for this are the two partitions through the red lines.

Most of the parameterizable parts of kdintersect apply to this entity. The most notable parameters are the number of traversal instances and intersection instances. These settings do not only affect the concrete structure of kdscheduler: Many entities adapt to these parameters to deliver enough ports or use different versions of buffers to prevent overflows, for instance. This results in many possible data flows and also alters parts

of the code used for the synthesized design. This variability is also the reason for the name: All computation and data fetching instances are connected and controlled here. See Figure 4.4 for the schematic.

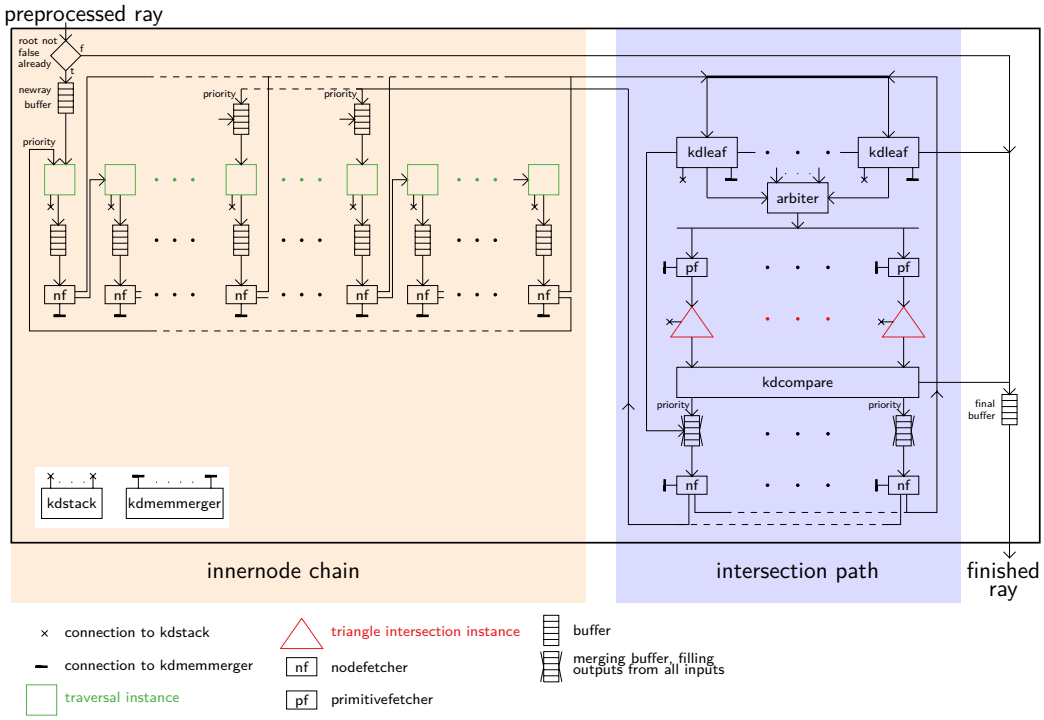


Figure 4.4: The schematic of kdscheduler. The number of the green kdinnernode instances, the red kdintersecttri instances and the kdleaf instances is determined by parameters. Although it does not seem so, kdintersect can be parameterized to do the reinsertion of the intersection path to the innernode chain on all positions. This includes the first position, where also the input data is taken.

The kdscheduler entity can be roughly split into two computation blocks which interact with each other. The `innernode chain` is responsible for traversing through the inner nodes of the k-d tree while the `intersection path` performs the decoding of leaf nodes and the actual intersection of triangles. At the end of an intersection, the ray is written to a `final buffer` to be ultimately forwarded to the output. It should be noted that this output is the output of kdintersect, which means that there is no further processing needed.

Besides the ray data going through the data flow, a ray also occupies a stack for storing not yet processed tree nodes. Instances in both computational blocks need access to these ray stacks. For this, the instances of entities needing access to the stack are connected to the `kdstack` instance (Section 4.3.5) inside of kdscheduler, which stores and manages all stacks. More precisely, the `kdstack` depicted in Figure 4.4 is not a single instance. It is possible that the kdscheduler uses multiple `kdstack` instances. These instances are

abstracted for the other entities in the kdscheduler such that the instances behave like a single `kdstack` instance. This design decision is due to optimization reasons; more details about this can be found in Section 4.3.5.

Some entities in both computational blocks also need access to data from RAM, which get this access by a connection to the `kdmemmerger` instance (Section 4.3.4), also instanced in kdscheduler. The `kdmemmerger` routes the instances needing memory access to the FPGA's RAM, arbitrating the data transfer. `kdmemmerger` is therefore the only instance inside of kdintersect which has a direct connection to the RAM.

Each instance needing access to a stack or RAM has an own port to it, as depicted in Figure 4.4. For instance, each traversal instance has its own port at the `kdstack`.

The kdscheduler begins its operation with the `newray buffer`, where all precomputed rays from kdintersect's preparation arrive. This buffer is responsible for applying a flow control to the input data to avoid an overflow inside of kdscheduler. If there is space left in the innernode chain, a new ray can be inserted into the chain every cycle.

The Innernode Chain. The innernode chain is basically a combination of elements that is able to traverse an inner node of a k-d tree. An element consists of a few entities. Multiple of those elements can be chained together multiple times to traverse rays with larger throughput. Each element's end is connected to the beginning of the next element. With this approach, a ray inserted into the first element can travel along the chain as it traverses into the depth of the k-d tree. This enables not only the next ray to be inserted in the next cycle. It is also possible to insert a new ray even if an older ray is needing another traversal through an element of the innernode chain, which increases the throughput. Only in case the depth of a k-d tree is greater than the number of elements, the insertion of a new ray is blocked. In that case, the last element forwards the ray back to the first element, which goes through the chain again. As the number of elements can be parameterized (with the number of traversal instances), the achievable throughput depends on the actual synthesized hardware design.

One element of the innernode chain consists of an instance of a kdinnernode entity (Section 4.3.1) followed by a buffer to prevent an overflow in the following kdnodefetcher instance (Section 4.3.3), which fetches the next node from RAM. The initial insertion of a ray starts at the first kdinnernode instance. The output of a kdnodefetcher leaves the innernode chain when it identifies a leaf node. The ray is then forwarded to the intersection path. The intersection path itself can fetch an inner node through the kdnodefetcher instances at the bottom of the intersection path. In case of such a node, the ray gets reinserted to the innernode chain again (and this is the only possibility for a ray coming back to the innernode chain). As both, each element of the innernode chain's end and the returning path from the intersection path return an inner node which is already fetched and decoded by the kdnodefetcher entity, it can be forwarded to the kdinnernode instance on the beginning of an element directly.

Please note that, in contrast to the depiction, the reinsertion from the intersection path can also be done at the first element of the innernode chain. In that case, the ray from the intersection path will be prioritized over the rays from the `newray buffer` and those coming from the last element of the chain.

The Intersection Path. The intersection path performs the actual intersections of primitives by using the kdintersecttri entity (Section 4.3.2). Before intersecting primitives, they must be fetched first, which is done by the kdprimitivefetcher (Section 4.3.3). Furthermore, as leaf nodes coming from the innernode chain may not contain the primitives data directly, some logic is needed to decode the leaf node to identify the primitives to load. This decoding is done by kdleaf (Section 4.3.6). It delivers the `primitiveid`s with which the primitives can be fetched and intersected. Lastly, to find the nearest intersection of a leaf node, the kdcompare entity (Section 4.3.7) compares all intersection results of the node. As a finished leaf node results in either a finished ray or another traverse through the k-d tree, additional kdnodefetchers are used at the end of the intersection path. If a ray is not completely traversed, these fetchers either forward the ray to the beginning of the intersection path in case of another leaf node being found or route them back to the innernode chain when an inner node was encountered.

To increase the throughput, the intersection path may use multiple kdleaf instances. The number of instances is automatically configured, depending on the number of traversal and intersection instances that are parameterized. To deal with any configuration, an arbiter is used to merge the data for the following primitive fetching. When only one kdleaf instance is used, the data is directly forwarded to the primitive fetchers. In case of multiple instances, the arbiter works in a different version. In this mode, the arbiter lets one kdleaf instance exclusively finish to forward a leaf node's result to not split it in the middle. Only after a full leaf node was forwarded, the arbiter takes the data of a different kdleaf.

The number of kdintersecttri and kdprimitivefetcher instances depend on the number of intersection instances that were parameterized. The number of kdnodefetcher instances in the intersection path are set automatically depending on the intersection instances and may be higher than the number of intersection instances in order to avoid lowering the throughput.

Finishing an Intersection. If an intersection is evaluated completely, it is forwarded to the `final buffer`. The direct way from the input of kdscheduler to the `final buffer` is also possible. This path is taken for all rays which do not intersect the bounding box of the scene at all (i.e., rays for which `root not false already` is `false`, which are rejected by kdintersect's preparation ,Section 4.2.1, already). The `final buffer` outputs the ray (consisting of `rayid`, origin, direction, and `rtmax`) besides the `primitiveid`.

Latency and Throughput. Most entities used in kdintersect have a certain latency but because of full pipelining ability, they can sustain a throughput of one. This means,

every cycle a new ray can be processed. This also applies to the data flow of kdscheduler. The only exception is kdleaf, where multiple possible inputs are merged to one to optimize resource usage. In practice, the only possible decrease of throughput comes from the fetchers. This is owed to the fact that the RAM is too slow to be accessed by all fetchers at once, forcing some of them to stall the data until the RAM returns their requests. Furthermore, a decrease can also occur due to the caches which may not be able to process all requests in time if the RAM latency is too high.

It should be mentioned that a low number of intersection and traversal instances lowers the throughput of the kdscheduler entity depending on the complexity of the scene it intersects. This is due to the mentioned fact that the innernode chain returns nodes from the end of the chain to the beginning, if the depth of the scene's k-d tree is too high. So even if the distinct instances used for the innernode chain do not suffer from a throughput penalty, the fact that a ray is processed twice by the same instances—which technically still sustains the throughput—implies that the throughput in terms of rays per cycle decreases.

4.3.1 kdinnernode

The kdinnernode entity implements the computation of a traversal along an inner node in a k-d tree. It takes the ray and the data of the node, consisting of the split axis, split position along this axis, and the references to the child nodes, as input. The output consists of the ray including the references of the children of the node. It traverses through the inner node evaluating which of the two children should be evaluated next. This child is always forwarded. The other child is only forwarded to the output if its space is traversed by the ray. There are two differences to the implementation of pbrt-v3 that only concern the saving and loading of the next children. Instead of loading the next child in kdinnernode, the output consists of the index which is forwarded to the kdnodefetcher (Section 4.3.3) in FPGRay. Also, FPGRay does not save a possible second child of the k-d tree in the stack, this data is directly sent as output to the kdstack (Section 4.3.5), where the data is put onto the stack.

We omit a description of the implementation details and only provide a high-level overview of the needed logic (see Figure 4.5), as the implementation is almost identical to the one in pbrt-v3 (namely the code of `KdTreeAccel::Intersect` inside the `!node->IsLeaf()` branch).

4.3.2 kdintersecttri

The kdintersecttri entity implements the logic to intersect a triangle against a ray. The triangle corner points are the inputs which have to be provided in the cycle in which the ray is forwarded. The implementation of this entity is directly taken from pbrt-v3's `Triangle::Intersect` method from the beginning of it until the part commented with `Compute $\delta_t$ term for triangle $t$ error bounds and check _t_` (including this part). It takes the ray and the primitive to intersect including its `primitiveid` as

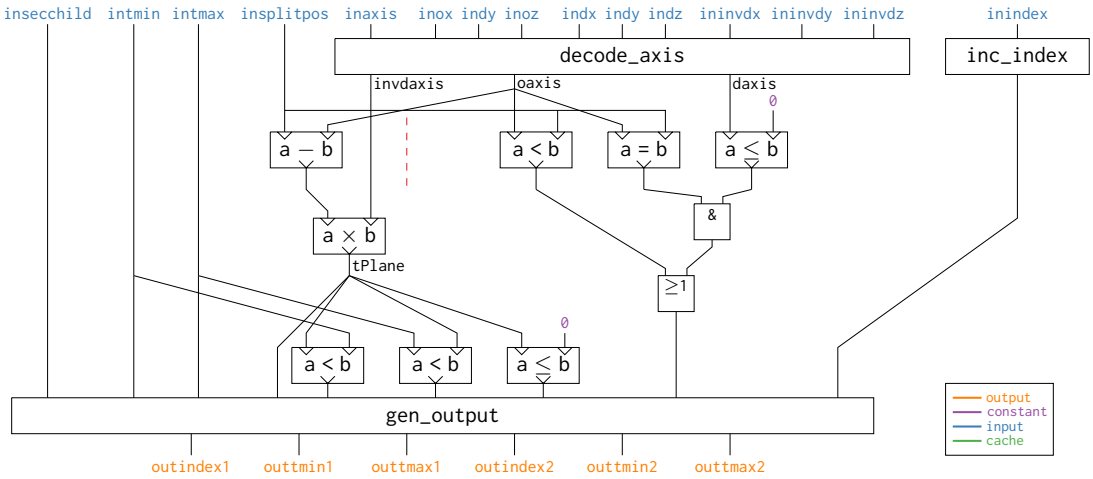


Figure 4.5: The logic to evaluate the children of an inner node and forward them in the right order. Note that conduits of inputs and outputs (applies also to the inputs and outputs to the instances) drawn in the same height are synchronous, i.e., in the same cycle of their computation. The only exception is the splitting depicted by the red line.

input. kdintersecttri checks at the end of a valid intersection if the distance to it (the `t` value) is smaller than the ray's `rtmax`. This indicates a closer intersection than the one found previously. In this case, the ray's `rtmax` value and the `primitiveid` are updated accordingly to form the new output. Otherwise, the output of this entity equals the input. The implementation of the logic as it is done in FPGRay is seen in Figure 4.6. kdintersecttri forwards the stack index to the stacking logic to fetch the next stack element 4 cycles before the finished computation is forwarded to the output. This is done to avoid any latency due to the stacking logic after the primitive intersection.

Because FPGRay only has access to the scene data but nothing else such as textures, alpha textures can not be tested against intersection. Alpha textures give the ability to additionally define areas on a shape that do not lead to an intersection even if the shape is intersected. As FPGRay does not test against alpha textures but pbrt does in the `Triangle::Intersect` method, using alpha textures would lead to different results of the intersection functions. For this reason alpha textures are not supported in any scene using FPGRay.

4.3.3 Data Fetcher

kdintersect (Section 4.2) needs three types of fetcher entities for delivering the needed data from RAM. One type is used for fetching nodes (kdnodefetcher), one for fetching primitives (kdprimitivefetcher), and one for fetching the indices of primitives that should be intersected in one node (kdprimindicesfetcher). All three entities have only small differences and use the same underlying kdcachedfetcher for the actual data fetching and caching.

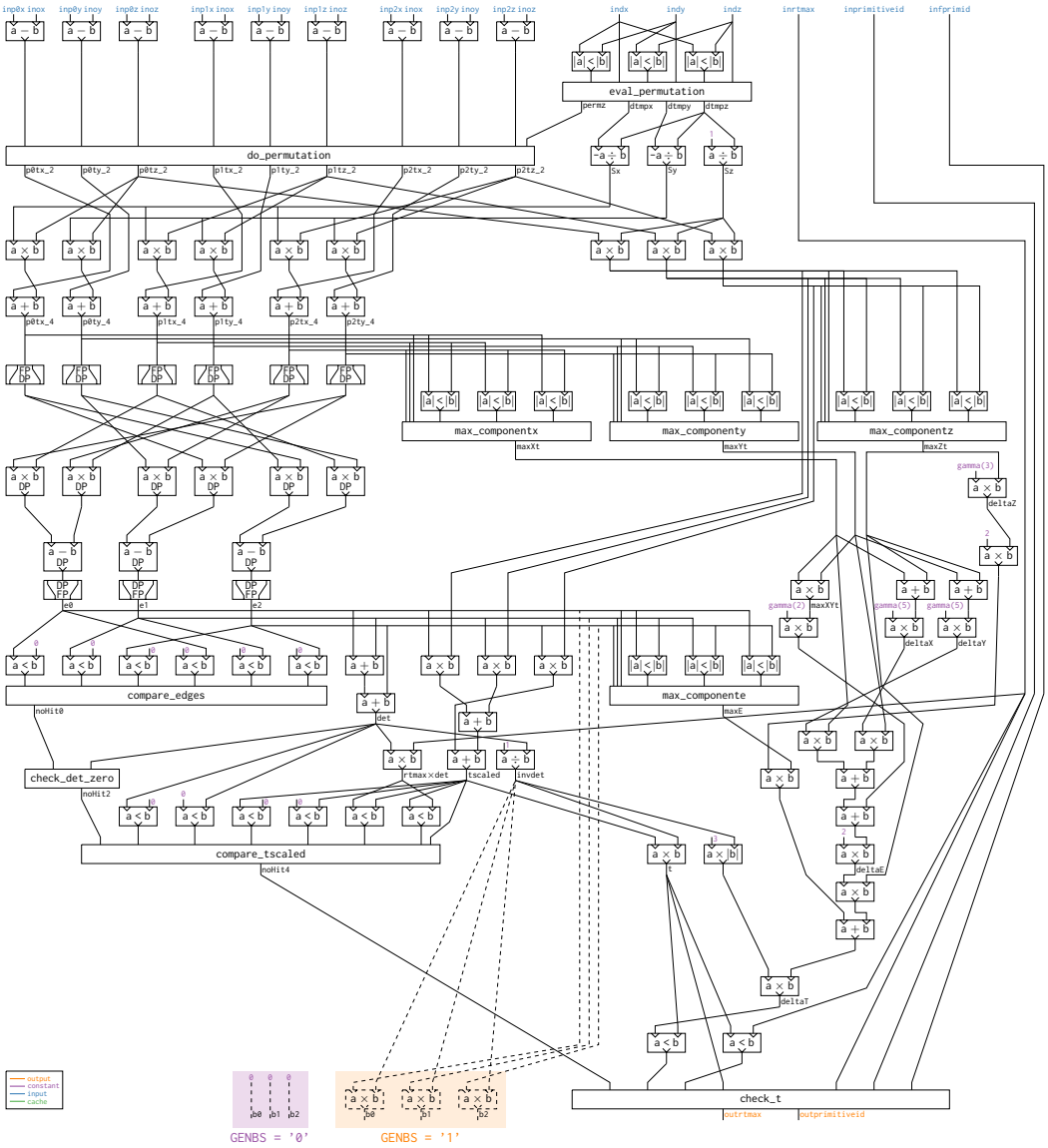


Figure 4.6: The logic to intersect a triangle against a ray. Notice that this entity is the only one making explicitly use of double precision floating point operations (operations marked with DP). It should be mentioned that only the input and output signals are in the same cycle but conduits of other operations or processes may not be on the same cycle if they are at the same height.

The distinct types only instance kdcachedfetcher and split the different types of signals for final presentation or processing. This allows to save much code. The kdnodefetcher and kdprimitivefetcher entities can be seen in Figure 4.4 labeled as nf and pf. The kdprimindicesfetcher is contained exactly once in each kdleaf instance (Section 4.3.6).

kdcachedfetcher

The kdcachedfetcher implements the cache and the interface to the RAM. It is able to fetch data with an arbitrary bit width and can also save data of arbitrary bit width (that is different to the bit width of the fetched data) in a wait queue during fetching. With two operation modes, it is built to deliver the base logic for all needed fetchers in this thesis.

Each kdcachedfetcher is connected to the kdmemmerger (Section 4.3.4), as there is no connection from a fetcher to the RAM directly. The kdcachedfetcher is fetching data via the kdmemmerger and caches it. The fetcher is abstract as it only uses the index of the loading object—in FPGRay this is either a node, a primitive, or primitive indices—and the `sceneid` to identify the address of the data in RAM. Data which is needed after the fetcher but not during fetching (e.g., the ray's data) is passed through, which means the data is stored in a wait queue and forwarded at the output besides the fetched data. If multiple signals need to be passed through, they need to be concatenated and split afterwards. Besides the passed-through data, the fetched data (either from RAM or cache) is returned.

The fetcher outputs a ready signal (named `rdy`) at the input port—which consists additionally of the mentioned `sceneid`, index to fetch and the signal for data that should be passed-through—such that a flow control can be used. If any queue inside the fetcher is full, the signal `rdy` is set to `'0'`. In practice, this often happens due to the latency and limited throughput of the RAM. As a result, the fetchers are the only instances in kdintersect (Section 4.2) that use flow control, giving them a variable throughput.

Two Operation Modes. The fetcher is parameterizable by generics. The operation mode can be changed to out-of-order by setting `ALLOW_REORDER` to `'1'`. In this mode, the fetcher forwards data when it is available, i.e., when it is present in cache. Data that is fetched from the RAM is forwarded via the main output port and cached data is forwarded via a second port. This enables to maximize the throughput by forwarding data when it is available. Furthermore, the latency suffers from less penalty. If `ALLOW_REORDER` is set to `'0'`, the fetcher is working in-order. In this mode, all data at the output has the same order as the data fed into the fetcher. All data is forwarded via the main output port no matter if cached or from RAM. The second output always outputs zeroes. This mode suffers from a throughput and latency penalty due to the stalling of already available data. Nevertheless, when the order of the data needs to be preserved, this penalty must be accepted.

To be able to satisfy the needed properties for the concrete types of fetchers, the main output port itself uses a flow control similar to a ready signal. With this, it is able to stall readily available data while still presenting it on the output port until it should be forwarded (to present the next finished date). The second port does not exhibit this special behavior and just forwards data when it is ready.

kdnodefetcher

The kdnodefetcher is used to fetch all nodes of the k-d tree. This applies to inner nodes as well to leaf nodes. The fetcher is able to extract the type of the fetched nodes (that indicate whether a node is an inner node or a leaf node). This allows the fetcher to forward the data to the primitive intersection instances or the traversal instances depending on the node type.

The input ray values including the actual bounding values of the k-d tree are passed-through by an instance of kdcachedfetcher. The id of the requested node together with the `sceneid` is issued in the corresponding input signals. The kdcachedfetcher is used in out-of-order mode as the order of nodes does not matter. As an advantage, already cached data can be forwarded before data already waiting in the queue for the RAM interface, allowing shorter latency. Additionally, this increases the throughput because the next data fetching can be started earlier as the queues inside the kdcachedfetcher are emptied earlier.

Each kdnodefetcher also includes two kdnodedecode instances. Each instance of this entity evaluates the data of an already fetched node. Its input therefore consists of a ray with a raw fetched node (i.e., the output of the kdcachedfetcher), which is evaluated for whether it is an inner node or a leaf node. Depending on the result, the data is forwarded either to the traversal output port or the intersection output port. The node's data is properly formatted for one of the outputs (leaf node output or inner node output) of the kdnodefetcher i.e., the values are extracted from the raw data of the fetched node. The entity decodes in one cycle, which can not be changed.

The connection scheme inside a kdnodefetcher entity is illustrated in Figure 4.7. An interesting point about the fetcher is the fact that the innernode outputs of the kdnodedecode instances are buffered whereas both outputs for leaf nodes are directly forwarded.

kdprimindicesfetcher

In FPGRay, primitives are stored in a contiguous RAM space. As each primitive is stored only once, leaf nodes with multiple primitives can not use just a range of this space. A good example that demonstrates the problem would be two leaf nodes having a common first primitive with all the others being different. To find the correct primitives, a third RAM memory space, besides those for nodes and primitives, which stores the primitive indices, is used. These hold the pointers to the primitives for each leaf node with more than one primitive in a consecutive manner. The kdprimindicesfetcher is able to fetch and forward up to four of these indices in one cycle. This is owed to the fact that one data block from RAM holds 4 primitive indices.

This entity uses one kdcachedfetcher instance for fetching. Besides the ray data, the fetcher takes the index of the data block containing the desired indices, the number of primitives that should be fetched, the offset of the first index in the first data block, the `sceneid`, and a final flag as input. The data block index and the `sceneid` are used as

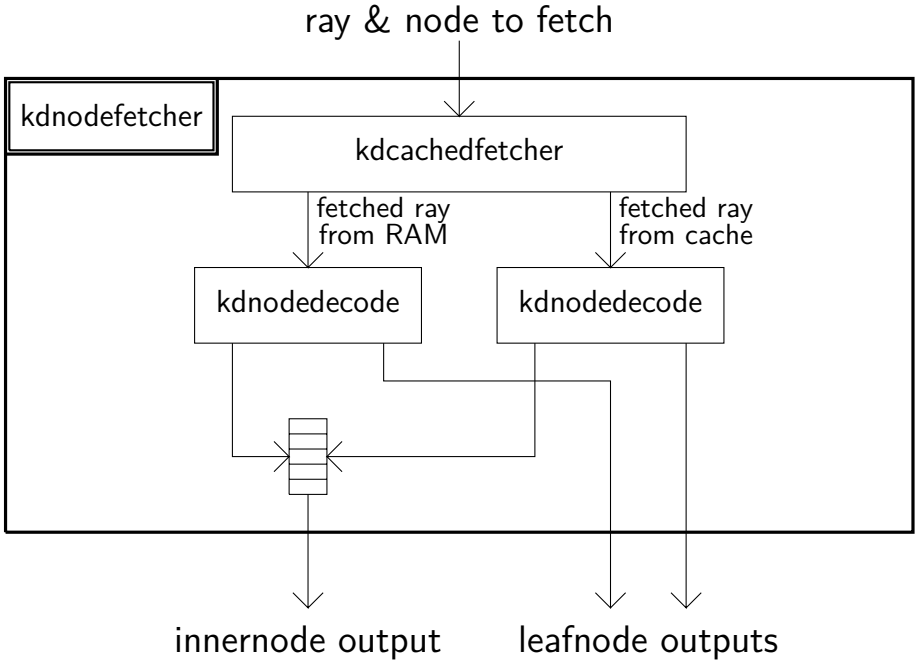


Figure 4.7: A kdnodefetcher instance consists of a kdcachedfetcher and two kdnodedecode instances. A buffer for inner nodes is buffering data if two inner nodes have finished fetching at the same cycle.

input for the fetching address. All other signals are passed through. The kdcachedfetcher is used in in-order mode, as blocks need to stay in order. This is important because a fetched block is the index of a primitive of a leaf node which needs to be fetched afterwards. A leaf node can contain multiple primitives and to allow an independent (i.e., parallel) intersection for each of them, the complete ray's data is copied for every primitive. But to be able to correctly separate different rays, the last ray copy is flagged. If the ray copy marking the final copy of this node—indicated by the mentioned final flag—is not the last copy in the data flow, the node can be split up further, corrupting multiple rays.

The kdcachedfetcher's output, besides the passed through data, is a block consisting of four indices. This block is obtained from a memory space that stores all indices for all leaf nodes consecutively. Because not all leaf nodes contain exactly a multiple of four primitives, the first primitive belonging to a node may not be the first in the block. kdpriminidcesfetcher thus computes the correct starting position (and analogously the end position) to forward only the indices belonging to the desired node.

For improving the throughput, up to four output ports can be configured to forward all four indices in parallel in the same cycle (or less if not all indices are needed by a leaf node). This configuration is done automatically when the number of triangle intersection instances is greater than one. If less than four output ports are used, the output of

kdcachedfetcher's main output port is stalled until all indices of the block are forwarded.

The fetcher also alters the final flag of the input. At the input, it indicates the last indices block of a ray. The fetcher issues the final flag at the output only for the last primitive of this block, ultimately marking the last primitive of the leaf node of a ray.

kdprimitivefetcher

The kdprimitivefetcher fetches the primitive data from RAM and delivers the 3 points of a triangle.

This entity uses one kdcachedfetcher instance for fetching. Besides the ray data, the `primitiveid`, a final flag, and the number of the cluster this ray is contained in are passed through. The cluster number is used to identify a ray without the need of the `rayid`. Cluster numbers are used later on by the kdcompare entity (Section 4.3.7) for more efficient comparisons. The address of the requested primitive and the `sceneid` is used for kdcachedfetcher's address inputs. Similarly to the kdprimindicesfetcher entity, the kdprimitivefetcher needs to be in in-order mode because a split ray copy of a leaf node could be interchanged with one of another leaf node, ending up in not being able to merge the ray back properly.

The main output port from the kdcachedfetcher is stalled if the primitive fetcher is waiting to forward the output. Thus the primitive fetcher just controls when the kdcachedfetcher should forward the output and does not have an additional buffer.

The primitive fetcher has two modes for outputting the primitive data including the ray data. In normal mode, which is configured by setting the generic `SYNC_OUTPUT` to `'0'`, each output is forwarded when the data is ready.

If the generic is set to `'1'`, the primitive fetcher is doing its work in combination with other primitive fetchers, syncing the outputs. This mode ensures that primitives intersected in parallel that are intended to be in the same cycle are kept so by stalling all fetched primitives as long as all of a cycle can be forwarded. For this synchronization, the kdprimitivefetcher uses additional input and output signals. All kdprimitivefetchers exchange a few outputs with the other instances before the actual forwarding of the outputs is performed. The exchanged values are the `rayid`, the `primitiveid` and the validity signals of every fetcher. These outputs are compared with the reference values handed over at the input to identify the readiness to return the desired values. Only if the expected and current values are equal, all kdprimitivefetchers forward the data in sync (which is ensured as all expected values are the same for all fetchers). The synchronization is done to be able to use multiple fetchers and triangle intersection instances in parallel because in such a case it has to be ensured that split up rays cannot be mixed up and bring the design into an undefined state.

4.3.4 kdmemmerger

The connection from the RAM interface to all fetchers is established by one instance of the kdmemmerger entity. All three types of data fetchers are connected to this merger such that the actual transfer with the RAM is centralized. Figure 4.8 shows the use of the three port types and the merging to two ports for RAM access. It should be noted that this number is fixed to two ports to optimize for the actual FPGA used.

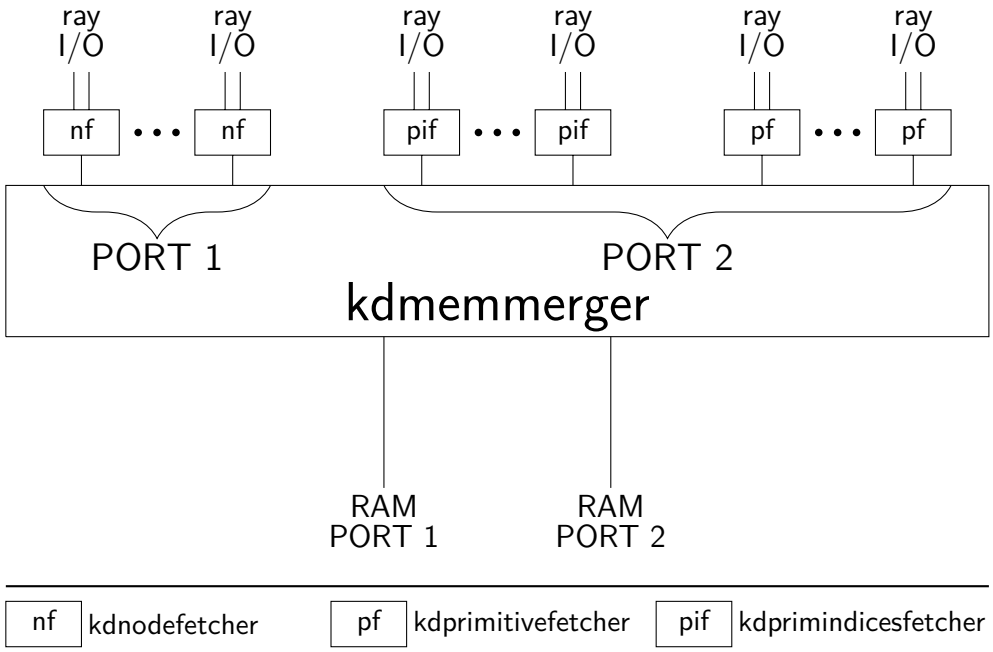


Figure 4.8: The connection scheme of kdmemmerger. Each type of fetcher has its own array of ports. Depending on the number of fetchers, different numbers of ports per array are used. Only the two ports to the RAM are fixed.

The kdmemmerger has a distinct buffer for every port to stall requests if the RAM ports are busy. Although this can absorb a sizeable amount of requests for a few cycles, it is assumed that the number of RAM ports—and of course its overall throughput—is sufficient to deliver enough data on average. The kdmemmerger issues no ready signal to prevent an overflow if the number of RAM ports is not sufficient for the number of fetchers used. Because of the big buffers used, this is only an issue if the RAM or the interface to it is too slow. If FPGRay would issue too many requests, this could only be prevented by reducing the throughput of FPGRay itself (e.g., by lowering the number of stacks).

4.3.5 kdstack

Each ray needs its distinct stack for saving nodes that are candidates for future evaluation. As kdintersect (Section 4.2) needs to compute many rays in parallel for a good utilization,

a high number of stacks is needed. To use multiple stacks, the kdstack entity provides a parameterizable number of stacks. The abstraction mentioned in kdscheduler (Section 4.3) that mimics only one instance of the kdstack even in case of multiple is technically implemented in kdscheduler and kdstack together. Other optimizations of the stacking logic that are described further, however, are entirely implemented inside of kdstack and are only configured by other entities instancing it.

The stack consists of three entities:

- kdstack: Main entity containing the whole logic needed for stack allocation. It also includes the instances of the other two entities.
- stackcommunication: Entity connecting multiple stacks together.
- stack: The basic entity containing one stack and the needed logic for maintaining the stack pointer.

Depending on the parameterized latencies of the basic floating point operations and the number of kdintersecttri (Section 4.3.2) and kdinnernode (Section 4.3.1) instances used, many stacks can be needed.

kdstack. The main entity is responsible for arbiting all read and write accesses to the stack that happen in any entity used in kdintersect (Section 4.2). As every kdinnernode instance (Section 4.3.1) needs to write to a stack, each instance has a distinct port to kdstack. It is used to write the second child of the node onto the stack if it needs to be evaluated during tree traversal. Besides the node index, the minimum and maximum extent of that node and the stack index are handed over by kdinnernode as input. For every kdnodefetcher, 2 read ports are needed. The reason is that every kdnodefetcher can output 2 leaf nodes in parallel. If these nodes contain no primitives at all, all rays would need to read the next node from stack to traverse the k-d tree further. Additionally, for every kdintersecttri instance, an own port to read from the stack is needed. For each reading port, the stack index is needed as input and the output consists of the node index, minimum and maximum extent, and a zero bit signalling an empty stack. In case of an empty stack, the values must be ignored. See Figure 4.9 for the connections between the other entities of kdintersect.

kdstack also provides reset ports. For every possible finished intersection, one port is provided. Additionally, for every kdnodefetcher two ports are needed. They are needed in case a ray fetches an empty leaf node that is also the last node of the traversal (resulting in the end of the traversal and thus release of the stack). If a ray has finished its evaluation through kdintersect, the stack index of it gets handed over to properly reset the concerning stack. This is done by setting the stack pointer of this stack to zero. The input of every reset is only the stack index and there is no returning value.

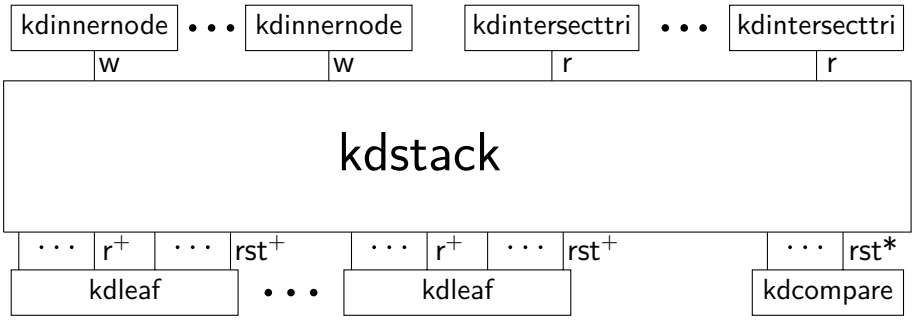


Figure 4.9: The connection scheme of kdstack, showing ports of read (r), write (w), and reset (rst) access type. * is the number of ports of the comparison instance (kdintersecttri +1 or one if kdintersecttri = 1), $^+$ is the number of ports one kdleaf instance (Section 4.3.6) has (of each of the two needed types): It is for full kdleafs 8 or 16 (2 ports from every kdinnernode instance, where 4 or 8 are connected to each full kdleaf, depending on the configuration) and for the last kdleaf 2×(remaining kdinnernode instances)+1.

Stack Allocation. kdstack maintains the number of available stacks as well as the allocation and releasing of stacks. For signalling the availability of a free stack, an output bit signal is provided including an index pointing to the next free stack. If the free stack bit is false, the indicated free stack is in fact not available and using it would result in a collision. The allocation of a stack must be indicated by an own signal. The release of a stack is done by issuing a reset through one of the reset ports, which also resets the stack pointer. Differently to the reset, an allocation can be done only once per cycle. Precisely, it is issued only by the `newray buffer` in kdscheduler, when a ray leaves the buffer towards the innernode chain (see Figure 4.4).

Multiple kdstack Instances. The mentioned configuration where multiple kdstack instances mimic a single instance is technically used to optimize the routing further. Each kdstack instance gets an equal number of stacks it exclusively maintains and instances inside of it. The conversion of the stack addresses is then done in kdscheduler. Outside of the conversion, a single instance is mimicked and each kdstack instance is treated as it is the only instance. The connection scheme seen by other instances is therefore illustrated in Figure 4.9.

A Stack Element. If a read or write operation occurs, kdstack uses 3 cycles to perform the operation on the stack. This is due to technical reasons to save resources on the FPGA (see Section 4.7.8 for further information). Every variable is written in an own cycle resulting in an increase or decrease of 3 entries on the stack for one stack operation. Figure 4.10 shows how the variables are stored on stack.

kdstack does not directly perform the stack operations and instancing the memory logic.

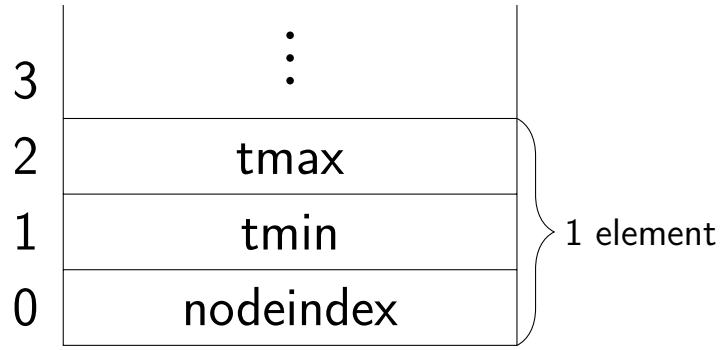


Figure 4.10: This is how a stack element on the stack actually looks like.

For this, it instances the stackcommunication entity where the read, write, and reset operations are processed.

For technical reasons, each stack has a maximum depth (i.e., number of elements) of 85, which means a deeper k-d tree is not supported. As this is even more than pbrt's maximum of 64 entries, a bottleneck should not be expected in this regard.

stackcommunication

This entity is responsible for generating instances of the stack entities. It connects them together in parallel, distributing access to them via the same connections. If data needs to be returned, the stackcommunication traces which stack needs to return data to the kdstack. At the cycle the returned data is available, the corresponding stack's output is routed to the kdstack.

As many stacks can be needed, an efficient routing between the stacks is crucial. Due to this, the communication is encapsulated in a distinct entity. This enables easy code changes of this connection to optimize timing or resource usage without changing the actual stack logic.

stack

The stack entity is the actual implementation of the stack memory with its stack pointer. It should be noted that this memory is not part of the RAM but rather on-chip memory, exactly like the buffers. In the context of the kdstack, one element requires three entries, i.e., if kdstack writes one element to the stack, three entries are written—for the stack these are distinct elements. This means the stack can be used with a different number of entries per stack element.

4.3.6 kdleaf

This entity summarizes the entire logic needed to evaluate a leaf node and decides where to hand over the processed node further. It is possible to have more than one instance of

kdleaf in a kdintersect instance (Section 4.2). To learn more about how the number of instances is obtained or how to influence it, see Section 4.7.9.

Figure 4.11 depicts kdleaf's entities and own provided logic and the internal connections of them. Depending on the number of primitives a leaf node possesses, different paths are used through the kdleaf entity. One kdprimindicesfetcher instance (Section 4.3.3) is needed to fetch the indices of multiple primitives in case a leaf node holds multiple primitives to intersect.

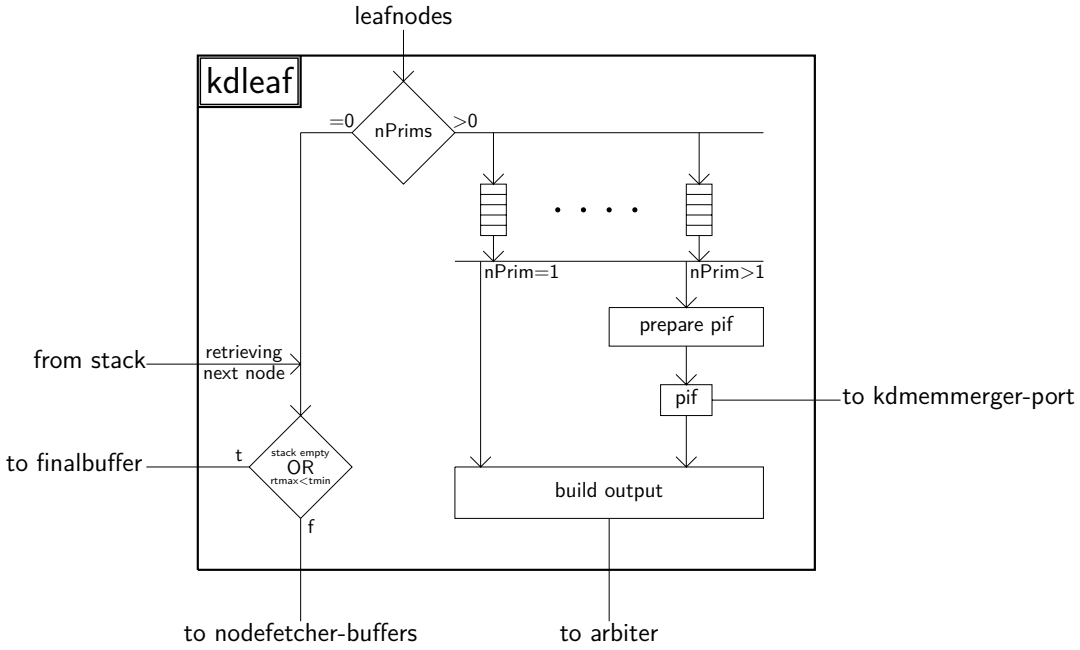


Figure 4.11: The internal scheme of kdleaf. Depending on the number of primitives a node has, three paths can be taken. The path on the left can be taken by nodes with no primitives at all, the right directly from the buffers to `build output` if a node has only one primitive. The complete run through the rightmost path with a kdprimindicesfetcher is only done for leaf nodes with more than one primitive.

Path One: No Primitives. kdintersect's (Section 4.2) k-d tree can have leaf nodes with no primitives at all (left branch of Figure 4.11). This means the node is completely evaluated already and the next node of the ray's traversal can be fetched (by first fetching its node index from the stack). If there are no further nodes on the stack or the `rtmax` value of the ray is already lower as the node's minimum extent `tmin`, the ray gets forwarded to the final buffer. Otherwise, the ray is forwarded to the kdnodefetcher entities to load the next node from RAM.

Path Two: A Single Primitive. If a node has exactly one primitive in it, it is forwarded from the input buffers to the output process `build output` and is ready to be

processed by the arbiter in kdscheduler (Section 4.3), which ultimately forwards the data to the kdprimitivefetchers (Section 4.3.3). This is possible because the leaf node stores the address of the single primitive directly instead of making use of the primitive indices array.

Path Three: Multiple Primitives. If a node has more than one primitive in it, additional computations need to be done to be ready for the output processes. Concretely, one kdprimindicesfetcher is needed to be able to fetch primitive addresses of nodes with multiple primitives in it. If a node with multiple primitives in it occurs, it hands over the index of the first primitive's index in the primindices array including the number of primitives that should be intersected. The primitives indices are stored in the primindices array one after another after the first index. As for many primitives, multiple index pages—each page (i.e., one data block fetched from the RAM) consisting of 4 primitive indices—need to be fetched, the `prepare pif` process is able to copy a ray multiple times to generate multiple fetching requests. It also computes the starting index inside a page, as multiple node indices could be stored in one index page. Independently of the possible copying of a ray, `prepare pif` marks the last copy (or original if no copies were made) of a ray with a final flag. This signals the end of a ray to the kdcompare entity (Section 4.3.7), such that the comparison between intersection results can be finished. It should be noted that with one primitive (path two), no comparison is possible: Each ray is automatically marked with the final flag.

The data from `prepare pif` is forwarded to the kdprimindicesfetcher, which fetches the actual primitive addresses which are stored at the primitive indices. Similar to `prepare pif`, the ray can get copied again, as each index of the page can belong to the current node. If the page is the last of a node, the final flag marker is only set at the last primitive. The primitive addresses which are fetched are forwarded to the output process, where they take the same way as those of path two.

The output process `build output` at the end has two versions, depending on the corresponding arbiter type kdscheduler is using. One version is the simple process that forwards data directly and the other is the complex version that requires a buffer to be able to properly operate with multiple kdleaf instances in parallel. For further explanations about that please refer to Section 4.7.9.

Cluster Numbers. The more complex version of the `build output` process generates so-called cluster numbers. Each forwarded primitive gets this number, which is unique for the node (but not primitive) in each cycle. The number indicates the group of primitives for which comparisons must be performed. This also means that in one cycle, comparisons of different rays with different leaf nodes can be done independently. The numbering and primitive position is in order, so only neighboring primitives can be of the same node.

The simple buffer does not need such cluster numbers, as it is only synthesized for the case of one primitive being forwarded per cycle. So there is no need for distinction in the comparison stage.

4.3.7 kdcompare

The kdcompare entity is used to extract the `rtmax` value of a ray after its intersection with primitives of a node. For this task, the entity has to find the nearest intersecting primitive of the node saving its `primitiveid` and `rtmax` value as the ray's data after the intersection. It should be noted that a comparison of the intersection of interest with intersections that are already done is not necessary here. kdintersecttri (Section 4.3.2) already gets two `primitiveid`s (the current nearest intersection and the one which is intersected) as input besides the old `rtmax` value and outputs the nearer of the two. So the only comparison that must be done here is between multiple primitive intersections within a node. The kdcompare entity gets the ray with its `primitiveid`, the stack index, a cluster number, a final flag, and the data from the stack for fetching the next node as input. This input port is replicated for each kdintersecttri instance, which makes it possible to process multiple rays per cycle. For each of these ports there are two output ports. One is forwarding to the final buffer (Figure 4.4) and the other to the kdnodefetcher (Section 4.3.3) for further evaluation.

Plain Comparison. For designs using one kdintersecttri instance, an optimized plain comparison function is used. The complete comparison is always finished with a fixed latency of 3 cycles, beginning at the cycle where the last primitive of a leaf node was intersected. If a comparison is not finished, the function is able to store the intermediate result for multiple cycles where no new data appears.

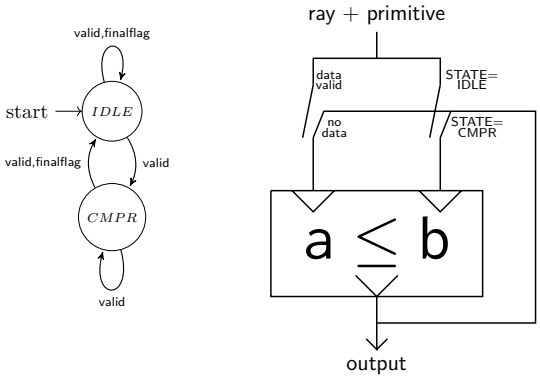


Figure 4.12: The logic for comparing primitives over multiple cycles, including the state machine that is used. Note that only the feedback functionality is depicted here for simplicity.

The functionality of the feedback loop is depicted in Figure 4.12. The loop is built such that every possible state (i.e., single primitive node, begin, middle, or end of a multiple

primitive node) of the data flow can be forwarded through it. In case of leaf nodes with only one primitive, the data is transferred through both inputs of the comparison function in "IDLE" state. If a leaf node contains multiple primitives, the loop's state changes to "CMPR" after the first primitive of the node—so it also forwards its data to both inputs of the comparison function—and switches back only if the final primitive was forwarded. In the "CMPR" state, one input of the comparison function always gets the result of the comparison of the last iteration as input. If no new data arrives, the other input also gets the comparison result as input to "store" the intermediate value in the feedback loop. When new data arrives, the newly finished primitive's intersection will be compared with the stored comparison value. After the last primitive exited the feedback loop (bringing it again back in "CMPR" state if necessary), the plain comparison evaluates the final comparison result with the stack's data. The output is then forwarded to the final buffer (Figure 4.4) if the stack is empty or if the current ray's `rtmax` value is smaller than the node's minimum extent `tmin`. Otherwise, it is put back to the kdnodefetcher's input (of the intersection path, as seen in Figure 4.4). Please note that in practice, the functionality is performed over more than one clock cycle and the state machine is performing only with the `rtmax` value and the rest of the data is forwarded at the end. The plain comparison does not need cluster numbers and thus ignores it.

Complex Comparison. If multiple kdintersecttri instances are used, the complex comparison function has to be used. It uses comparison stages, each able to compare pairs of two primitives and its comparison results. By comparing a pair and merging two `rtmax` values to the lower value, the number of `rtmax` values can be halved with each stage. Ultimately, this results in the need of a logarithmic number of comparison stages in terms of kdintersecttri instances. It does not matter whether the primitives forwarded to this function in the same cycle are of the same node or different nodes or any mixture of them. Similarly to the plain comparison, the intermediate comparison value can be stored if primitives still need to be compared in future cycles (or if the entity needs to wait for new data).

It would be possible to use this version for any number of intersection instances, even in case of one kdintersecttri instance. However, the plain comparison should be preferred in such situations as it provides higher efficiency. This should only be seen as side note, as the comparison function will be chosen automatically depending on the set parameters.

Please refer to Section 4.7.10 for more information about the complex comparison function.

4.4 Parameterization Recommendations

When FPGRay should be adapted (e.g., to fit on a different FPGA model properly), a few recommendations ease the proper parameterization of kdscheduler (Section 4.3):

- Computing instances: The number of kdinnernode instances (Section 4.3.1) should be greater than the number of kdintersecttri instances (Section 4.3.2). This is due

to the fact that a practical k-d tree uses a depth that is greater than the number of primitives in one node. This means that in order to intersect one primitive, multiple inner nodes need to be traversed. The design is therefore more limited by the number of kdinnernode instances than the number of kdintersecttri instances.

- Cache sizes: The number of cache elements should be increased as much as possible (as the timing is more limiting than the number of logic resources). Experiments also showed that the RAM access is so slow that bigger caches can even almost facilitate the same throughput as an additional computing instance would in case of low hardware resources. For more in-depth information about the cache refer to the implementation details of the cache (Section 4.7.6).

- Stacks: As explained before, a low number of kdinnernode instances limit the throughput more than the number of kdintersecttri instances. As a result, providing enough stacks to fully utilize the kdinnernode instances is sufficient. As a rule of thumb it is sufficient to use 2 times the number of stacks than the latency of the whole kdinnernode entity, multiplied with the number of used instances.

- Tradeoffs: When combining all of the recommendations above, the available resources of the FPGA should be utilized by first setting the number of computing instances and with it the number of stacks. Then, the size of the caches should be increased with the remaining resources. Experiments showed that the design with two or three kdinnernode instances need at least 64 elements in order to avoid an over-utilization of the RAM. If this number of cache elements can not be synthesized on the FPGA or when scenes with many primitives are used, it may be better to sacrifice a computing instance for more cache elements. Such a design can be faster because the data access from cache increases the throughput more than one additional computing instance which is in turn waiting longer for data.

4.5 PCIe Driver

The driver accessing the PCIe interface is a Linux kernel module driver [ALK, LKM] and as such, it is written in C code. Because of this, the data transfer between the API (Section 4.6)—which provides FPGRay's functionality to user programs—and the driver is done by making use of plain C structs. The driver gets pointers to arrays of such structs to gain access to the user program's data. As C structs store data plainly, they can be directly used as data block, as the variables lie in consecutive memory locations. Accordingly, the DMA can copy the data one by one to the FPGA. This is in contrast to the classes and structs typically used by renderers. pbrt, for example, uses C++ classes for rays, which must be changed when using FPGRay.

The driver running in kernel mode takes the pointers to data from a struct the API hands over. This is done every time a function call to the driver is performed. The supported functions are:

- Configure a transfer: Sets the number of data units to transfer, the operation that should be performed by FPGRay and the delays to use.
- Reset the FPGA: Sends a reset request to the `resetter` entity in the FPGA. After sending the acknowledge, the FPGA shortly waits to safely finish the acknowledge and needs time to properly reset. Thus to safely be ready again, the function waits 500 ms before returning.
- Read a register: Reads a register of one of the reserved addresses in FPGRay. These registers deliver debug and profiling information about FPGRay. Note that this function could read output data of FPGRay or reset the FPGA but should not be used for that.

The actual data transfer and the desired operations are done by a different system call. This call just starts the operation and needs to be called after the configuration of the transfer. All calls from a user program are blocking, i.e., the driver stops the program's execution until the data is computed and transferred back from the FPGA.

To configure a transfer, the following parameters can be used:

- command: The 32-bit command for FPGRay containing the operation and the number of units (e.g., rays, memory lines) to process. The command is the value which is used by FPGRay's state machine (FPGRayIF, Section 4.1) and is directly handed over.
- read and write pattern: The number of data blocks a unit contains (one 128-bit data block corresponds to one line of the PCIe interface or RAM) in sending direction to the FPGA and receiving direction. In the case of different numbers of data blocks per direction, the transfer needs to be adjusted to ensure maximum throughput and simultaneously avoid overflows in the overall system. This is done by using so-called patterns. The bigger data block is transferred continuously, while the smaller data block is transferred with pauses in between. For a k-d tree intersection (kdintersect, Section 4.2), two 128 bit blocks are needed as input to the FPGA and one as output. This would result in a continuous read and a write only every second data transfer period.
- offset parameter: This parameter defines how many DMA operations (i.e., data transfer periods) should be read only until the first write operation should be started (which may then run simultaneously, depending on the read and write pattern and total number of transfer periods needed). Because each transfer period consists of the transfer of 1024 data blocks (read, write, or both simultaneously), the write back starts after the read of the $1024 \times$`(offset parameter)` th data block.
- source and destination pointer: Marks the beginning of the memory spaces a user program wants to get data taken from or written to.

Depending on the function and command with which FPGRay is used, not all parameters may be used. If a pointer is not needed, the null pointer can be used.

4.6 API

This project does not stop at the driver to make the usage of FPGRay more convenient. The API is used to abstract the hardware and also the hardware-near implementation of the driver. It provides a simple interface for using FPGRay in software by instancing a C++ class. The API itself is written in C++ but makes use of the data structures specified in the C driver. This is owed to the fact that the driver is programmed in C and the data structures should be handed over efficiently. All functions of the driver can be accessed through the API.

Here is a list of the functions that are available:

- `bool connected()`: Indicates whether the FPGA card is properly connected via the API.
- `int resetFPGA(bool output)`: Resets the whole card including the RAM. Returns `0` on success, `1` if not. Set `output` to `true` to print the `resetter` instance's response (`0x19E5E7` will be sent by FPGRay if the reset was issued correctly).
- `int readRegister(int addr)`: Returns the register's content at address `addr`. Only the offset address is needed as the function starts at the base address of the FPGRayIF instance (Section 4.1).
- `int loadScene(int *src, int numunits)`: Writes `numunits` 128-bit wide data blocks beginning from `src+1` to the RAM of the FPGA. The starting target address is the content of `src` plus one, which is incremented for every data block. This is the function that should be used for loading scene data to the RAM. The increment of the base address is done because FPGRay interprets the first (offset-)address 0 as null pointer.
- `int storeRAM(int *dest, int address, int numunits)`: Reads `numunits` 128-bit wide data blocks from the FPGA's RAM to `dest`. Begins at the position `address` with the copy operation. As this function is for debug/raw reading, the `address` is not incremented by one as in `loadScene()`.
- `int processRaw(int *dest, int *src)`: Transfers data beginning at location `src+1` to FPGRay and the resulting data returned from the FPGA after processing to `dest`. As this function uses the raw mode of the underlying driver function, the command needs to be present at the first position of the data field `src`.
- `int processFunc(int *dest, int *src, int numunits, uint8_t func, uint8_t read_pattern, uint8_t write_pattern, uint8_t writeafterreadoffset)`: Performs the DMA transfer and lets the operation `func` compute on the FPGA. The

parameters that are used are explained in Section 4.5. More in-depth information can be found in Section 4.7.11. This method is used for nearly all of FPGRays method calls, which just call this function with properly set parameters.

- `traverseNode/intersectPrimitive/intersectTriangle(int *dest, int *src, int numunits)`: Use the instanced kdinnernode (Section 4.3.1), kdintersectsp, or kdintersecttri (Section 4.3.2) for intersecting a ray. Essentially, they all only call `processFunc()` with proper parameters and do not check if the operation is synthesized by design (this needs to be done manually by reading the function register of FPGRay).
- `int kdIntersectChgScn(...)`: This function is used to set the k-d tree scene's root node and the addresses of RAM locations for the kdintersect instance (Section 4.2). This is done by a struct which is handed over to `processFunc()`.
- `int kdIntersect(int *dest, int *src, int numunits, int readafter)`: This function uses FPGRay's kdintersect to intersect rays found in `src` together with the loaded scene in the RAM of the FPGA. With the `readafter` parameter, the offset parameter of the PCIe driver can be configured. Similarly to `kdIntersectChgScn()`, it does not check if kdintersect is synthesized in the current design.

The memory blocks handed over via `src` and `dest` need to be arrays of plain C structs which are provided in the API.

The usage of FPGRay via API is simple, which is exemplified with this code:

```
1  #include "FPGRay.h"
2  /*
3   * The data of the scene needs to be present in those blocks already (so
4   * these variables need to be defined and filled outside of FPGRay's API):
5   * primmemoryblock, nodememoryblock, primindicesmemoryblock
6   * Furthermore, the following variables need correct values and also need
7   * to be defined outside of FPGRay's API and this example already:
8   * num_primitives, num_nodes, num_primitiveindices, worldspminx, worldspminy,
9   * worldspminz, worldspmaxx, worldspmaxy, worldspmaxz, rootnodeIndex,
10  * rootnodeAboveChild, primbaseaddr, SplitPos, primindicesbase, SplitAxis
11  */
12 int main(){
13    FPGRay *hw = new FPGRay();    //connect to FPGRay
14    if(!hw->connected()) return -1;     //check if connected
15    hw->resetFPGA(true);       //reset the Hardware
16
17    //load the scene into FPGRay's RAM
18    hw->loadScene((int*)primmemoryblock,num_primitives);
19    hw->loadScene((int*)nodememoryblock,num_nodes);
20    hw->loadScene((int*)primindicesmemoryblock,num_primitiveindices);
21    //set the scene information for kdintersect
22    hw->kdIntersectChgScn(worldspminx, worldspminy, worldspminz, worldspmaxx,
23          worldspmaxy, worldspmaxz, rootnodeIndex, rootnodeAboveChild,
24          primbaseaddr, SplitPos, primindicesbase, SplitAxis);
```

```
25
26    //allocate the memory for the number of rays an intersection should be done
27    struct inIntersectRay *input= (struct inIntersectRay*)
28          malloc(sizeof(struct inIntersectRay)*num_units);
29    input[0].ox = 21.42;//And so on to fill in the data of the rays
30    //allocate the memory for the output
31    struct outIntersectRay *output = (struct outIntersectRay*)
32          malloc(sizeof(struct outIntersectRay)*num_units);
33
34    //perform the operation
35    hw->kdIntersect((int*)output,(int*)input,num_units,2);
36
37    cout << "first␣rays␣rtmax␣is␣" << output[0].rtmax;
38
39    delete hw;      //disconnect from FPGRay
40    return 0;
41 }
```

Lets take a closer look at a few lines:

```
FPGRay *hw = new FPGRay();     //connect to FPGRay
if(!hw->connected()) return -1;      //check if connected
delete hw;      //disconnect from FPGRay
```

After creating a new instance of the FPGRay class from the API, the PCIe card should be connected. With the method `connected()`, this can be checked. Likely causes for not getting a connection are the absence of the hardware design on the FPGA (trying to connect to the "empty" card with no design loaded), the card not being connected, or the missing installation of the driver. To close the connection, the class needs to be destroyed to properly disconnect from the card. If no dynamic memory allocation is desired (e.g., in a small method calling FPGRay), it is possible to instance the card in the scope with `FPGRay hw;` like every other C++ class.

```
hw->loadScene((int*)primmemoryblock,num_primitives);

hw->kdIntersectChgScn(worldspminx, worldspminy, worldspminz,
      worldspmaxx, worldspmaxy, worldspmaxz,
      rootnodeIndex, rootnodeAboveChild, primbaseaddr,
      SplitPos, primindicesbase, SplitAxis);
```

With the `loadScene()` method, arbitrary data can be loaded to the RAM of the FPGA. The method is used to load the scene data. One call for the primitives, one for the nodes and the third one for the indices of the primitives are needed. The loading of data can of course be split further such that these three types can be loaded in parts. For bigger scenes this would be necessary, as at maximum `268,435,440` Bytes can be loaded at once.[4] The `kdIntersectChgScn(...)` method is used to forward the additional

[4] In practice, this is no problem at the moment, as the PCIe card used for the thesis only has 256 MiB RAM anyway.

parameters computed during the scene generation beforehand (outside of this example and the FPGRay API) to the `kdintersect` entity. `rootnodeIndex`, `primbaseaddr`, and `primindicesbase`, identify the memory locations of the scene data.

```
struct inIntersectRay *input= (struct inIntersectRay*)
        malloc(sizeof(struct inIntersectRay)*num_units);
```

FPGRay uses C structs (see `FPGRay.h` for exact definitions) to send rays and receive the computed results. Multiple rays can be sent at once by using an array of those structs. This can be done with static or dynamic memory allocation. For more than a few rays, the code above (using dynamic memory allocation) must be used to avoid a stack overflow. The struct that should be used changes depending on the performed operation and on the direction (in/out). The structs can be filled with data and read like any other struct to easily set the data for FPGRay and fetch the result back:

```
input[0].ox = 21.42;
input[1].rayid = 14;

cout << "first␣rays␣rtmax␣is␣" << output[0].rtmax;
cout << "second␣rays␣rayid␣is␣" << output[1].rayid;
```

The actual operation is done by calling the corresponding method. The needed parameters are the starting addresses to the source and destination memory blocks as pointers and the number of units on which the computation should be performed. Note the cast to `int*` for the arrays of structs:

```
hw->kdIntersect((int*)output,(int*)input,num_units,2);
```

After the call of this method, the results are stored in the output memory block. Note that the memory blocks have to be allocated beforehand to avoid any errors[5]

We omit the explanation of the extraction of a scene from pbrt but show an actual snippet for performing a ray-intersection test:

```
1  bool KdTreeAccel::Intersect(const Ray &ray, SurfaceInteraction *isect) const {
2      ProfilePhase p(Prof::AccelIntersect);
3     FPGRay *hw = new FPGRay();//connect to FPGRay
4     //Set values from ray
5     struct inIntersectRay input;
6     input.rayid = 1;
7     input.ox = ray.o.x;
8     input.oy = ray.o.y;
9     input.oz = ray.o.z;
10    input.dx = ray.d.x;
11    input.dy = ray.d.y;
12    input.dz = ray.d.z;
13    input.rtmax = ray.tMax;
14    //prepare memoryblock
```

[5]Similar to a segmentation fault, a crash of the program will happen without proper allocation.

```
15      struct outIntersectRay memblock;
16      //let FPGA do the computation
17      hw->kdIntersect((int*) &memblock, (int*) &input, 1, 1);
18      if(memblock.intersected == 0) return false;
19
20      //intersection = true -> compute surface interaction
21      const std::shared_ptr<Primitive> &prim = primitives[memblock.primitiveid];
22      prim->Intersect(ray, isect);
23      return true;
24  }
```

This intersection method is the alternative version of the original `KdTreeAccel::Intersect(...)` method that is accelerated by FPGRay. The code is similar to the example above and much smaller than the original intersection method. It should be noted that the resulting primitive, found by FPGRay, is intersected again by the pbrt method to properly populate pbrt's intersection object.

4.7 Implementation Details

In this section, we provide a deeper look into FPGRay. We discuss details about how the components work to justify some design decisions. We also give additional information about some key parameters and how they change the hardware design. As the details refer to the information of the higher-level discussions in the previous sections, reading the previous sections before this one is highly recommended.

4.7.1 Hardware Design

As stated, FPGRayIF (Section 4.1) does not handle the physical and protocol-specific communication of the PCIe interface. For this, the HardIP of the FPGA is used, together with additional IPs, such as an on-chip memory for storing DMA requests. Besides FPGRayIF (which is the top entity, encapsulating all functionality with respect to ray tracing), there is another entity used for interfacing with software, the so called `resetter`.

The QSys integration tool provided by the Quartus EDA from Altera is used to properly connect all those components. See Figure 4.13 and Figure 4.14 for a view of the used Qsys system. Two reasons speak for the Qsys approach. First, with Qsys, we do not need to build the arbitration between multiple entities connected to the same bus. There is no need for connecting interfaces together by plain VHDL and therefore running the risk of needing additional translation code to properly connect different entities. Instead, all needed translation is done implicitly by the bus used in Qsys. The second reason for Qsys is the saved work by building the design upon the reference design. The used Qsys system is based on a demo for showing the DMA operation between the PC and the FPGA's RAM. So the only modification needed on the hardware side was the removal of the RAM controller and the inclusion of FPGRayIF and the `resetter` in its place. The connection between the PCIe IPs, FPGRayIF, and the `resetter` is done inside the

Qsys system by using the Avalon memory-mapped interface with one bus connecting all entities.

As the interface only needs to perform basic data transfer, no experiments were carried out by using additional features of the PCIe interface (e.g., interrupts).

The complete hardware design consists of the top design file instancing the mentioned Qsys system (Figure 4.13 and Figure 4.14) and the DDR3 RAM controller, which is instanced outside of the Qsys system. It should be noted that there is no arbitration or adapter used between the Qsys system and the DDR3 controller, as the only entity using it is FPGRayIF (i.e., this connection is an exported conduit from Qsys connected to the controller's conduit). Besides the internals of the used IPs, the top design file (taken from the reference design) is the only code written in Verilog. All other code is written in VHDL.

To have debugging capabilities even without a PCIe connection, additional code was added in the top design file of the project to make use of the FPGA card's Light-emitting Diodes (LEDs). The design was also extended to allow the reset of the card from within the Qsys project. This enables the user to reset the card via software through the `resetter` entity. As the reset button is located on the PCIe card and therefore difficult to access, this is the only viable possibility for performing a reset.

All IPs used in this thesis are coming from Altera. Additionally, all computational logic, including logic written on our own, have a throughput of one. Entities which rely on RAM access times—which is the case for all fetching entities—can have a throughput lower than one. In contrast to the throughput, the latency depends on the entity (and set parameters).

Accessing the Main Entities. To get access to FPGRayIF and the `resetter` via software, they have been assigned to an exclusive address space (starting at the "Base" address specified by the columns in Figure 4.13 and Figure 4.14). This means that if the PCIe interface gets data for a specific address, the data is sent to either FPGRayIF or the `resetter`, depending on the applicable address space. Writing or reading to an address mapped to one of the spaces, transfers data from and to the entities instead of transferring data to the PC's RAM. Because of this behavior, such an interface is called memory-mapped. FPGRay should actually get one ray after the other and return the performed intersection results in the same order. This is the behavior of a streaming interface, but the use of a memory-mapped interface instead of this desired interface has a simple reason: The provided IPs for PCIe only allow the use of the memory-mapped mode. Making use of the streaming behavior would need much more work. In particular, it would require extensive knowledge about the PCIe specification.

The `resetter` is an entity listening to the PCIe interface's bus for a reset command from software. Additionally, a distinct port directly (i.e., without a conversion from Qsys) connected to FPGRayIF exists. Over this port, the `resetter` receives an alive signal. This signal is used to feed a timer inside the `resetter`, which automatically resets the

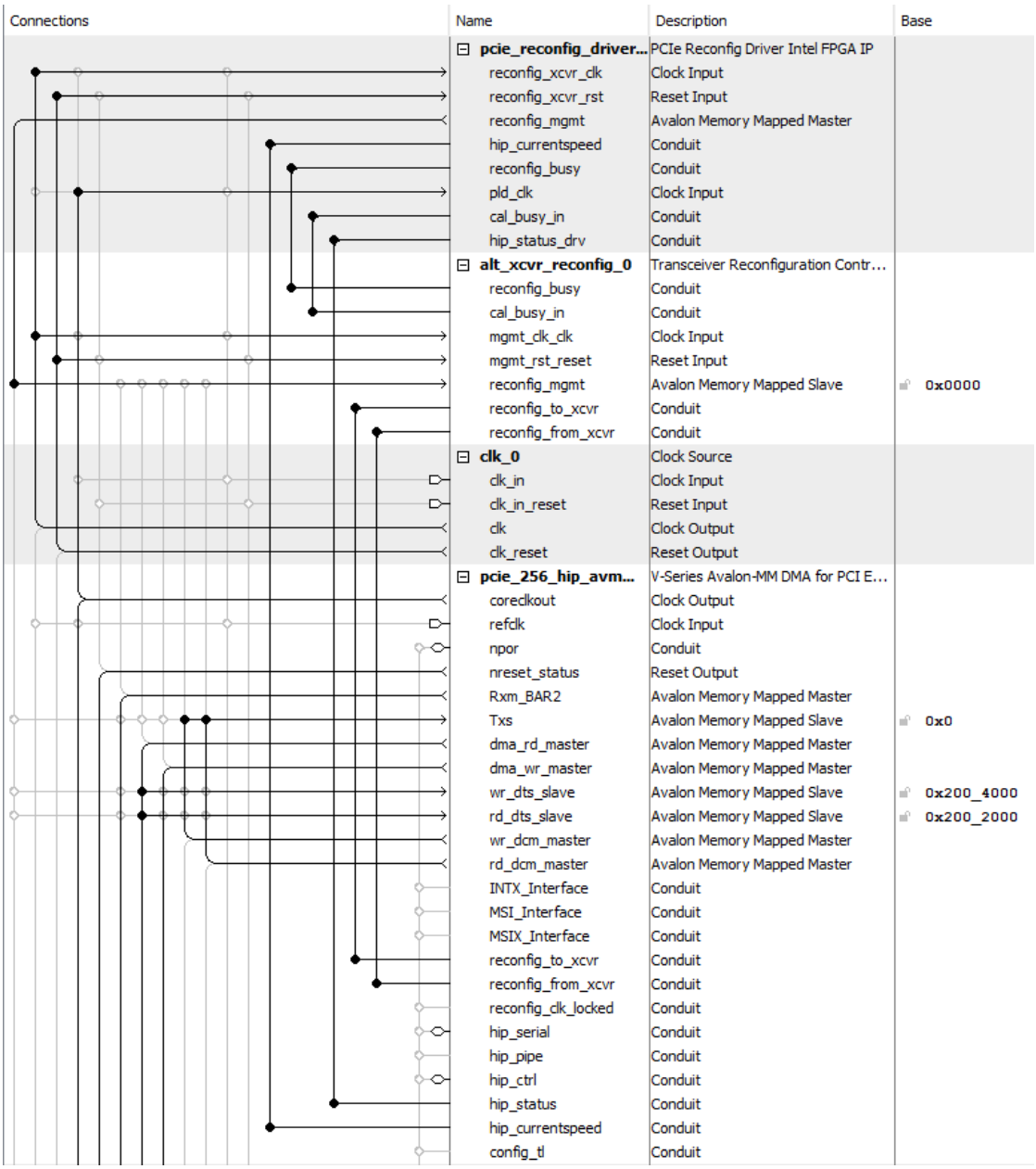


Figure 4.13: The part of the Qsys system FPGRay is composed of which is needed for the PCIe communication. `clk_0` is a clock source using via exported conduit a pin of the FPGA. For configuration of the PCIe-controller HardIP `pcie_256_hip_avmm_0`, the two components on the top are needed. Note the connections which are done via the Avalon bus ("Avalon Memory Mapped Master/Slave" in the "Description"). They are indicated by the lines in the "Connections" column and connect the data transfer between all of FPGRays components.

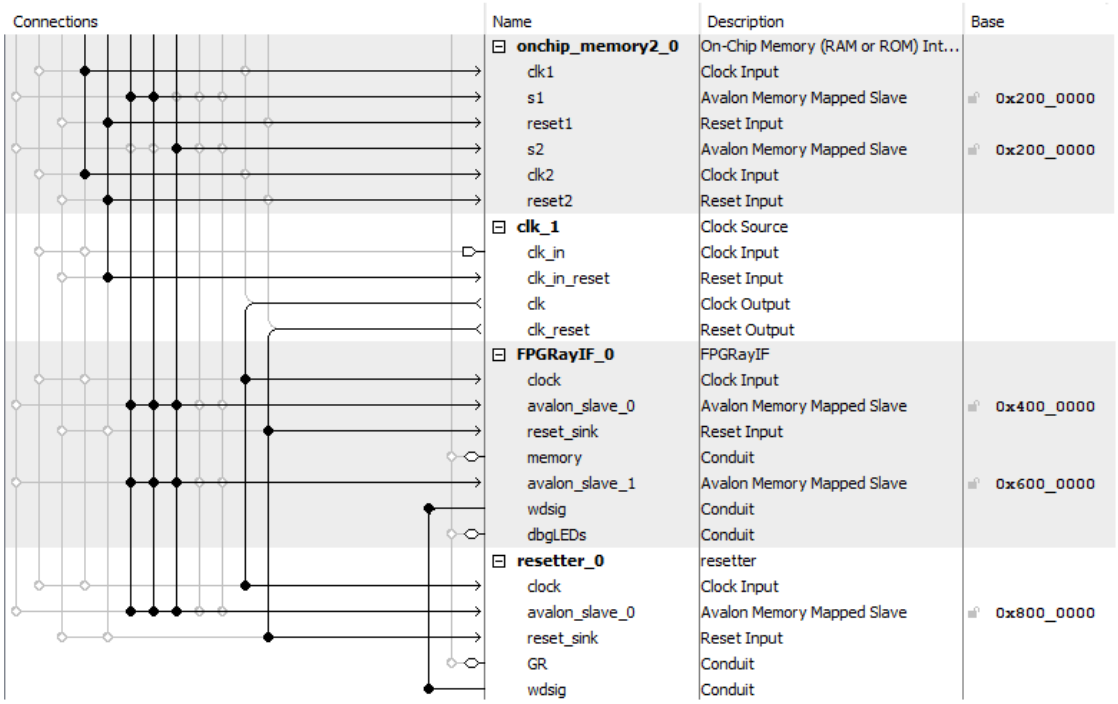


Figure 4.14: The part of the Qsys system FPGRay is composed of which is containing the FPGRay specific components `FPGRayIF_0` and `resetter_0`. Additionally, a small on-chip memory (to store DMA requests) is seen at the top, including a second clock source `clk_1` FPGRays components are using. Note the connections which connect their Avalon bus interfaces to those of the PCIe components shown in Figure 4.13.

FPGA after getting no alive signal for about 10 seconds. This feature is generally known as watchdog timer. It is used for resetting the FPGA if the PCIe interface is blocking (e.g., because of a corrupted DMA transfer) and a reset can not be sent from software.

It should be noted that FPGRayIF and the `resetter` always return with the empty word (all bits set to `'1'`) on read access when no data is available (which is always the case for the `resetter`). Only when a function is processing data at the moment, the read is blocking until the accessed address' data is available. An exception from this rule are special registers listening on reserved addresses that always forward the register's content on access. The reset command at the `resetter` is such a special register (and besides the activation or deactivation of the watchdog via another special register the `resetter`'s only functionality). When no function is performed, both entities ignore any received data with no command (i.e., all bits of a data block are set to `'0'`). This behavior ensures that reading and writing can always be done safely, even when no operation should be performed on the transferred data. This enables the driver to use data blocks of fixed size (1024 per DMA operation) and still allows an arbitrary number of data blocks with actual content, which facilitates simpler code for the driver. As the driver writes and

reads empty data, accepting and returning it is crucial in order to avoid blocking a read, which can crash the host PC.

4.7.2 FPGRayIF

FPGRayIF (basic explanations in Section 4.1) holds the state machine for correctly routing input data to the entities that perform the ray-tracing-related operations. It also instances not only those entities but also contains the output buffer for providing finished data to be read back by software and logic to ensure the data is kept in order.

Resorting. To translate between the memory-mapping interface and the actual desired streaming behavior, FPGRayIF possesses some extra logic.

On the input side, an additional buffer for all data transfers—except for those belonging to kdintersect's (Section 4.2) operation[6]—is used. It is resorting data arriving out-of-order from the PCIe interface transparently. Out-of-order data only occurs seldomly and the magnitude of the displacements tends to be small, so a very small buffer can be used here. The reason for the resorting is due to the PCIe interface specification, which does not require to preserve the order of requests (i.e., data transfers). But as the DMA sends requests in order, the nearly in-order data transfer is noticeable.

At the output side of FPGRayIF, results are stored in a buffer. As it allows random access, the PCIe interface can read data out-of-order directly without the need of an additional buffer. The resulting data is written into the output buffer in-order. The only exception is kdintersect, which uses a resorting logic to fill the output buffer out-of-order. This needs to be done as individual rays of a k-d tree intersection can get shuffled due to the different number of operations an intersection of a ray can take. The resorting logic therefore tracks the ray order and returns each result at the correct position in the output buffer. As FPGRay uses its own internal ray IDs for sorting, the used `rayid`s can be chosen arbitrarily.

DDR3 Controller. FPGRayIF directly connects to the DDR3 RAM controller in the top design file. The connection is done via the fabric interface provided by the HardIP. It enables the usage of two Avalon memory-mapped interfaces with the frequency at which FPGRayIF operates, transparently transferring between the clock domains of FPGRayIF and the DDR3 interface. The DDR3 controller runs at 533 MHz, which is the maximum the controller supports.[7] The interfaces are connected to kdintersect and additionally, the first interface of the RAM has a second connection to the input state machine of FPGRayIF on the request side and a connection to the output buffer on the response side. This connection is used to load scene data from the PC to the RAM. Note that requests to the RAM are buffered.

[6] As kdintersect returns rays out-of-order a resorting needs to be done anyway. kdintersect therefore marks the rearrangement at the input and resorts these changed positions at the end only.

[7] FPGRayIF (and with it all computational logic concerning ray-tracing operations) runs at 100 MHz.

I/O. The part of FPGRayIF providing the output logic is completely independent of the input state machine and does not use a state machine. By designing the entity like this, it is possible to perform operations in arbitrary order. For example, if a scene has been loaded and intersection operations have been performed, further intersections can be done or a new scene can be loaded. It is still possible to read the data of the first operation even if the input state machine is already performing another operation. Additionally, further operations could be easily added by adding a new case to the state machine and just routing a new operation to a new entity—the output logic stays as is and only needs to know the number of data blocks that should be awaited as for any other operation.

The communication protocol of the state machine is as follows. The first data block, containing 32 bits of data, encodes the command on the upper 8 bits. The lower 24 bits determine the number of data units on which the operation should be performed. Then, without any termination symbol, the data units are sent one after another. After receiving a data block, the state machine is in the state named after the operation that should be performed, routing any input to the corresponding function. Notice that for all operations, one data unit needs to be a multiple of 128 bits[8]. If no data needs to be sent (for just reading data from RAM), only the command is sent.

As in practice the usage scenario only consists of performing one operation at a time, it is not supported to do multiple operations. Every operation has to be finished before starting the next one (but its result does not need to be read from the output buffer). The only exception is the issuing of the same operation again. This is crucial due to the fact that no arbitration between multiple simultaneous operations is done on the output side. With this approach hardware resources are saved. It is also not a problem when using the provided driver, as it never performs multiple operations simultaneously.

Supported Operations. Per default, FPGRay is always able to perform the direct passthrough or inversion of input data (operations `NOP`, `INV`), write data (`LOAD`) to the card's RAM, or read from it (`STORE`). All other operations that can be used, can be synthesized into the final hardware through constants in the FPGRayIF entity. Apart from the passthrough (not needing many resources) or the RAM access (basic operation), not synthesizing all operations can save hardware resources for other operations. As an example, the final hardware design for this thesis only contains the k-d tree intersection function `KdINT` (which additionally provides the function `CHGSCN` to load the k-d tree parameters into kdintersect) to spend all available hardware resources for it. All other operations[9] are not present. However, except kdintersectsp, all operations are used inside of kdintersect; the distinct use of those operations was implemented for testing purposes.

If an operation is not synthesized in the current design, FPGRayIF returns the same amount of data as the called operation, filled with a "not synthesized" pattern. This is

[8] This bit width for one data block is the bit width of the PCIe interface on hardware side.

[9] `INTERS` – intersect a sphere (kdintersectsp), `INTERTRI` – intersect a triangle (kdintersecttri, Section 4.3.2), `TRAV` – traverse an inner node of a k-d tree (kdinnernode, Section 4.3.1)

done to avoid freezes.

Besides the normal operation using the state machine and the output logic, FPGRayIF possesses additional registers residing on specific addresses in the upper end of the address range assigned to FPGRayIF. Because the addresses are on the upper part of the address range, they will not be accessed mistakenly during normal operation. The registers provide profiling and debugging information. For example, the operations synthesized and thus usable in the current design can be read through a register. Another example is a register tracking buffer overflows throughout the used instances. For a more detailed view about the synthesizable operations, reserved registers, and parameters, see the parameter list in Section 4.7.12.

It should be noted that the optionally synthesizable operation `INTERS` is the only synthesizable operation which is not part of kdintersect. kdintersectsp was the first primitive intersection entity, used to intersect a sphere instead of a triangle. It is not used any more and only available as a distinct operation. It is also the only logic which needs a fixed latency for the floating point operations (i.e., 1 cycle latency for comparisons and 10 cycles for all other operations); this does not apply to other entities as explained further in the following section.

4.7.3 kdintersect

kdintersect, as the entity containing all intersection-related entities, possesses some characteristics that are not discussed in the main section (Section 4.2).

If no primitive intersection was found for a ray, its `rtmax` value will be kept and the `primitiveid` with index `0` will be returned. Furthermore, indices such as the `primitiveid` and the `stackid` are shifted inside kdintersect to reserve `0` as null pointer. For the `primitiveid` this convention is kept up to the the output, so FPGRayIF (Section 4.1) converts it before forwarding it to the output buffer.

In detail, there are two possible reasons for the reordering of rays inside of kdintersect. First, each ray can need a different number of nodes to traverse and the number of primitive intersections is also not the same. Secondly, the rays can also get reordered when requesting nodes. This is due to the cache, which is using the out-of-order operation mode for nodes, returning data from the cache faster than from the RAM.

Dealing With Multiple Scenes. In all hardware designs used for this thesis, kdintersect can intersect rays against 16 different scenes in parallel. As in practice only two different scenes are intersected in parallel—at the point where the next animation frame is intersected—this number is sufficient. The use of a different scene for all following rays is done when the `CHGSCN` function is called from software. To allow parallel operation, the `rayid` needs to be unique over multiple scenes[10]. The bookkeeping of scenes is managed

[10]Note that FPGRayIF generates for proper resorting own `rayids` which is the reason why this is always fulfilled for kdintersect even when the software does not send unique `rayids`.

by the process `set_scenecache` (see Figure 4.3).

kdintersect can also use a different scene for every ray. In that operation mode (`USE_DEDICATED_SCNID`), each ray must supply the index of the scene against which the intersection should be done.

Changing IPs. For each floating point operation used inside kdintersect, different IPs can be used by changing only the implementation in `FPGRay_pkg`. The latency can be any natural number (in clock cycles) and can be also changed there. This enables the easy change of IPs to adapt the design for other FPGAs. Note that only one delay value can be used for all comparison functions.

The commonly used 32-bit `IEEE754` standard is used to represent floating point values. But like many software ray tracers, the bit width can be changed. In this case, additional steps are needed: The IPs for floating point operations and all constants must be adapted. The used constants are also defined in `FPGRay_pkg` and are basic constants such as `0` as well as precomputed values for the gamma function used for intersecting triangles. Furthermore, FPGRayIF would need adaptions because many bit width values are hardcoded for allowing a simpler PCIe interfacing logic.

4.7.4 kdscheduler

kdscheduler has a special path for supporting k-d trees with primitives only (i.e., the root node is a leaf), which is not depicted in the overview diagram (Figure 4.4). The path is forwarding data before entering the `newray buffer` to a special input port of a kdleaf instance (refer to Section 4.3.6; the last instance if multiple exist).

The stack plays a central role for the flow control of kdintersect (Section 4.2). As a ray can only enter the computation from the `newray buffer` of the kdscheduler when a free stack is available, the number of stacks control the number of concurrently computable rays. If there is only one stack, each ray can only start to traverse if the complete intersection of the previous ray is done. An exception is the use of primitive-only scenes.

kdscheduler is the entity that is responsible for the flow control of the whole kdintersect entity: The `ready` signal of kdintersect depends on the occupancy of the `newray buffer`. The entity does not ignore data on the input port if it is not ready, which means it must be ensured that no data is fed while the ready bit signals zero to prevent overflows.

In contrast to the PCIe interface, the RAM must answer data in the same order as requested.

4.7.5 kdintersecttri

The intersection function of a triangle against a ray conforms to pbrt's implementation with one exception: The values `e0`, `e1`, and `e2` are always computed with double precision while pbrt is doing it only as a fallback if the accuracy of single precision is not sufficient.

kdintersecttri is able to compute the values `b0`, `b1`, and `b2`, which can be found in pbrt's code below the comment `Compute barycentric coordinates and $t$ value for triangle intersection`. All computations below this comment are not needed in kdintersect (Section 4.2), as this is the code which is not required for the actual task of finding the nearest primitive. Therefore, the values `b0`, `b1`, and `b2` are synthesized out per default. They can be included if the computations performed on the FPGA should be further used in software.

4.7.6 kdcachedfetcher

The kdcachedfetcher can be used in in-order and out-of-order mode. Both modes use the main output port to provide fetched data for the following entities in producer-consumer manner. This means that when data is available, it is presented at the output and the following entity has to acknowledge its use. With that behavior, the other fetcher types can stall the output without the need for a distinct buffer. In out-of-order mode, a second output port is delivering data that is found in cache (thus the main output port delivers only data fetched from RAM) without producer-consumer behavior, i.e., it just forwards the data.

Implementation Details. Internally, the fetcher stores all requests in a `waitbuffer`. A second buffer, the `uniquewaitbuffer`, is used to store only those requests that are sent to RAM. Those requests are unique as only the first request of an address which is not in cache is forwarded to the RAM. The response may therefore be requested by multiple entries in the `waitbuffer`. As in general, fetched data has a high locality, the same data is requested often which allows the use of a much smaller `uniquewaitbuffer` compared to a `waitbuffer`.

For storing passed through data, an on-chip FIFO is instanced. As the `waitbuffer` contains only a few elements, the minimum possible 256 elements for the FIFO are used. The cache itself is an on-chip RAM with a size that is determined by a generic (i.e., the number of cache elements of the fetcher).

Other aspects that are parameterizable by generics are the number of elements the `waitbuffer` and the `uniquewaitbuffer` should have; and the bit widths of the memory index, the `sceneid`, the data that should be fetched, and the passed-through data.

The memory index and `sceneid` combination is used as tag to identify the origin of a cache entry in RAM. This tag is checked to identify either a cache hit (element is in cache) or a cache miss (not found; fetch the element from RAM). As on scene change a `sceneid` can refer to a new scene, the fetcher is notified for scene changes and flushes the entire cache—by deleting the tag array and resetting the clock algorithm explained below—in such a case[11].

[11] It has a performance impact on the tag array to find and delete only those entries with the obsolete scene, but this array is a timing critical part of the cache already. Therefore, the cache is flushed completely

Clock Algorithm. kdcachedfetcher uses the clock algorithm [TR01] to replace cache entries with new data. The algorithm uses a pointer to a cache entry, which traverses all entries in a clockwise manner and an array of bits indicating an access for each entry. See Figure 4.15 for an example. Initially, all bits are set to 0. If an entry was accessed (i.e., read or write from cache) the bit of that entry is set to 1.

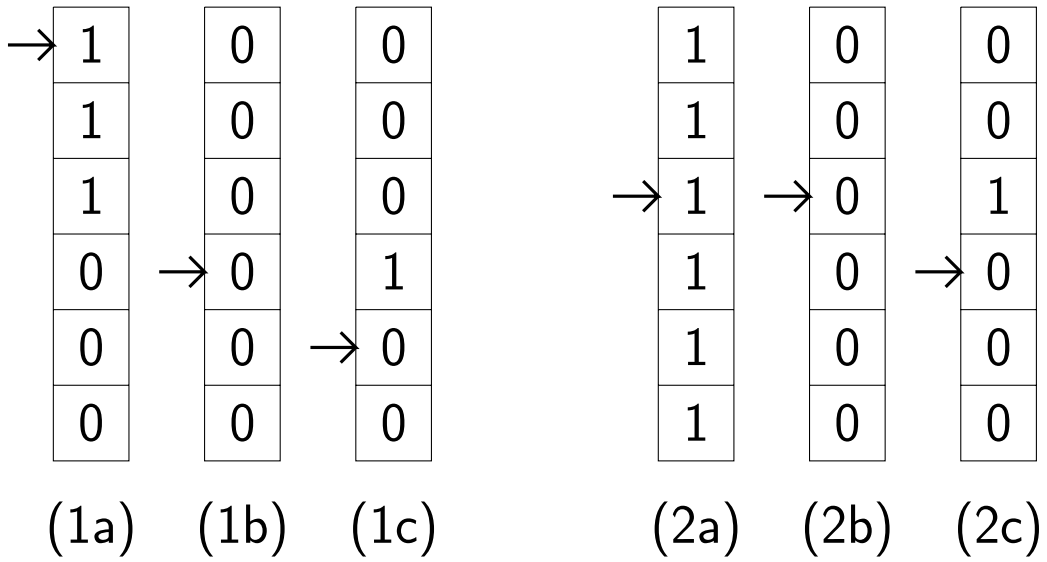


Figure 4.15: The function of the clock algorithm.

Given a cache with an access pattern as illustrated in Figure 4.15 (1a), the insertion of a new date is as follows. The pointer walks from its last position along the cache entries, finding the next cache entry which has its bit set to 0. On its way, all bits of value 1 are reset (Figure 4.15 (1b)). The bit of the entry which stores the new data is set to 1 and the pointer increments its position (Figure 4.15 (1c)). If all the bits in the access pattern are set (Figure 4.15 (2a)), the pointer walks around all entries once, ultimately finding the entry it was at the beginning with its bit set to 0 (Figure 4.15 (2b)). The fetched data is inserted in the found entry and the pointer is incremented by one position (Figure 4.15 (2c)).

4.7.7 kdmemmerger

For every cycle, the kdmemmerger forwards one request from a fetcher to one of the RAM interfaces. The requests can be buffered for each fetcher and processed in a round-robin manner to avoid overflows and starvation. Two additional buffers store requests already sent to RAM to be able to properly route them back to the correct fetcher in a non-buffered manner.

This entity needs RAM interfaces with a streaming behavior, whereas the Altera HardIP DDR3 controller uses a memory-mapped interface. To overcome this, the input from the

kdmemmerger to the RAM interface is buffered. On the response side, no buffering is needed as the streaming and memory-mapped interfaces are the same for this side.

The kdmemmerger stores the base addresses (i.e., the starting address of a memory space) for all scenes. But besides the `sceneid` of a request, the base address also depends on the type of fetcher that has sent it (i.e., node, primitive, or indices fetcher). If a fetcher sends a request, the base address will be added to the request's address resulting in the absolute address of the RAM lines. With this separation, the fetchers use only abstract array indices without the knowledge of the actual memory locations. For primitive fetchers, the kdmemmerger additionally splits a request into three RAM requests, as a triangle's data is constituted by three RAM lines.

4.7.8 kdstack

The stack is needed to store a child node for further evaluation when another child node is evaluated first. This applies to every ray, so to be able to compute multiple rays in parallel, multiple stacks are needed. kdstack therefore supplies a parameterizable number of stacks and maintains them.

It is crucial that each stack is only accessed once until the operation is completely finished, as the operations require multiple cycles. Violating this rule results in data corruption, as the stacks are not secured against parallel access. FPGRay waits the minimum needed number of cycles for avoiding collisions. This is ensured as the fetchers which are in between stack accesses have a sufficiently high latency and furthermore—in the case of a split ray—only a ray copy that is flagged as final accesses its stack.

Stack. The stack entity uses an on-chip RAM block as memory. As explained previously, one stack element needs 3 entries in the stack memory. This is done for saving resources as on the used Arria V FPGA, one memory block has at least 256 entries but provides only up to 40 bits of data per entry. The split of the access into three cycles of operation does not add much latency as most of the accesses can be anticipated and thus started earlier than needed.

The stack pointer is also located in the stack entity, which means it has to indicate an empty stack for other entities. This is done by an empty signal. To preserve a constant latency for an empty stack as well, kdstack always performs three read accesses and does not check for the empty bit. The stack has an underflow protection to allow such reads, but as the stack depth is very high in practice, an overflow protection is not used.

4.7.9 kdleaf

One kdleaf entity is instanced for every 8 kdnodefetcher instances (Section 4.3.3), processing only the leaf nodes coming from its 8 fetchers. By setting the `DINDS` (double indices) parameter to `'1'`, the number of kdleaf instances is doubled, making each instance processing the outputs of only 4 fetchers. The last instance, which may not process 8

(resp. 4) fetchers if there are not a multiple of 8 (resp. 4) fetchers instanced, additionally processes all root-only k-d trees (i.e., scenes where the root node is a leaf node). Those nodes are processed the same as the other inputs with the exception of being routed to the final output buffer of kdscheduler (Section 4.3) in any case as there is no other node which could be evaluated afterwards. Please note that such root-only k-d trees are just a node containing the whole scene by referencing all primitives in it; such trees are corner cases which should not be used in practice.

Single-primitive nodes are stored at the input buffer of the corresponding input port. They are kept on these buffers until it is requested by the merging output buffer at the end of kdleaf (`build output` in Figure 4.11).

Simple and Complex Output Buffers. The output buffer in a kdleaf instance (`build output` in Figure 4.11) communicates with the arbiter in kdscheduler (Section 4.3, see Figure 4.4). For the two versions of the output buffer, a matching arbiter as a counterpart is available. The simpler output is optimized for one primitive intersection instance and one kdleaf instance. The more complex version needs to be used when more than one primitive intersection instance or more than one kdleaf instance is used (but could be always used). However, the simpler buffer should be used if possible because the complex buffer uses much more logic resources. This is due to the fact that the on-chip memory buffer does not support more than one memory access per cycle; but for the complex buffer multiple accesses are needed.

The simple buffer just forwards one primitive of the kdprimindicesfetcher (Section 4.3.3) each cycle as long as the fetcher has not finished the output of all primitives of a node. Otherwise, a single node primitive is forwarded from an input buffer. This is done in round-robin manner to avoid starvation.

The complex buffer needs a distinct buffer to prepare a certain number of primitives which are presented as output to the arbiter. The data flow is done in a producer-consumer manner: Only if the arbiter acknowledges the output, the next primitives from the buffer are presented. This needs to be done as the arbiter might choose a different kdleaf instance to take data from. The number of primitives that are presented to the arbiter is the number of intersection instances used. In contrast to the simple buffer, the complex buffer fills its buffer such that single-node primitives can occur before and after primitives from the kdprimindicesfetcher, but never in between of those primitives. It is also ensured that the buffer has enough empty slots to store primitives from the kdprimindicesfetcher to avoid a data loss. An overflow protection is not used as the buffer's size is chosen such that the arbiter takes primitives from it often and fast enough.

Note that the arbiter also ensures that a node is not split in case of multiple kdleaf instances, i.e., the arbiter only takes data from another kdleaf instance if the current instance returned a completed node. Furthermore, a round-robin fetching of data from kdleaf instances to avoid starvation is used as well.

4.7.10 kdcompare

The complex comparison can be used for any number of synthesized primitive intersection instances. To efficiently compare multiple primitives which finished their intersection in parallel, comparisons are also done in parallel.

The Helper Entity. The merging of resulting intersections is done by multiple instances of the `complexcmp_stage` entity. It merges intersections as depicted in Figure 4.16. To be able to perform this operation, the entity needs the `primitiveid`, `rtmax`, the final flag, the cluster number, and the stack index position for each input port. The stack index position is the index of the port of the `complexcmp_stage` entity from which the stack's data should be used as stack data for the own port. `complexcmp_stage` returns with the same number of output ports but the number of outputs forwarded are less than inputs as only merged outputs are forwarded. Note that even if primitives of different nodes can be merged in parallel they must be consecutive, i.e., all primitives of node 1 and after that all primitives of node 2; mixed primitives are not supported. `complexcmp_stage` can merge n inputs to $\frac{n}{2}$ outputs ($\frac{n}{2} + 1$ when n is odd) in the best case by comparing all primitives in pairs.

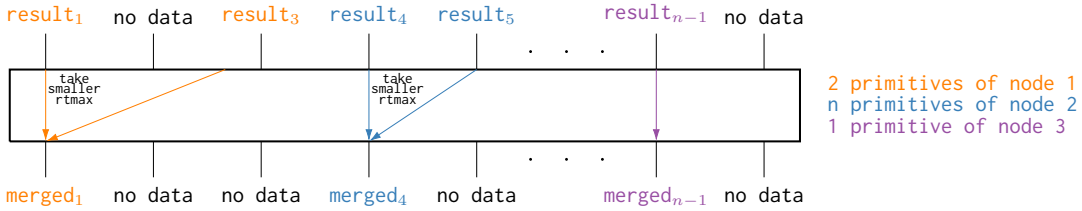


Figure 4.16: One instance of the `complexcmp_stage` checks for inputs with the same cluster number (i.e., are intersection results of the same node) and merges them in a pairwise manner. The entity can merge pairs of multiple nodes independently in the same cycle. Merged inputs are returned at the left of the two possible output ports which is desired for the final comparison step. The merged result at the output is the `primitiveid` with the smaller `rtmax` value. Note that the stack index is always taken from the right input, as the last primitive of a node shows up at the rightmost position and is the only primitive which contains valid stack data.

Note that all comparisons between `rtmax` values in the simple and complex comparison are done with the `fpgray_cmpaleb` comparison function. It always uses a latency of zero (which is needed for the loopback comparison of the comparison state machine at the end) and automatically adapts to any given floating point specification set in `FPGRay_pkg`. Note that `fpgray_cmpaleb` is independent from FPGRay and can be used standalone. Each `complexcmp_stage` instance uses a different number of `fpgray_cmpaleb` instances to merge inputs together. The number is set automatically and depends on the number of inputs which have to be merged.

Complex Comparison. The complex comparison provides an input port for each kdintersecttri instance and the same number of output port pairs plus an additional one (each pair consists of a port to fetch the next node and one to finalize a ray's intersection completely). This is done to forward finalized intersections without bottleneck. With this configuration, each intersection and a collected intersection result over multiple cycles can be forwarded in the same cycle.

The comparison starts by merging all intersection results of a cycle together to have only one result per leaf node. This is done by concatenating a certain (automatically computed) number of `complexcmp_stage` instances together. At the end, the final comparison uses the same state machine as the simple comparison (Figure 4.12) to merge intersection results over multiple cycles if a node is not finished within a cycle. The same state machine can be used as it is ensured that only one comparison per cycle can be unfinished. In parallel to the multi-cycle state machine, the stack's data is checked if the ray is finished or the tree needs to be further traversed. This check is also done for all rays finished in this cycle. Note that this check can only be done at the end, as the stack's data is only loaded by the ray copy with the final flag, i.e., the last flag.

It should be noted that the number of stages and as such, the number of kdintersecttri instances can be arbitrary. In practice, there is a limit nonetheless. This is due to the fact that the overall intersection path (Figure 4.4) has a high number of interconnections between its entities. This results in a very limited number of kdintersecttri instances compared to the high number of elements possible in the innernode chain. As the number of traversal instances should be more than those of intersection instances anyway, this is not a big issue.

4.7.11 PCIe Driver

The communication is done via the PCIe interface on the hardware level and needs a driver on the software side to be able to use the hardware in programs. To use the FPGA card, it has to be plugged into a PCIe port before booting up. After booting up a Linux distribution, the so-called driver has to be installed. As this is a Linux kernel module driver [ALK, LKM], this has to be done after every restart of the PC. Furthermore, the Linux headers need to be installed such that the module can get compiled for the used kernel. The compilation is done automatically during installation.

The driver is based on the driver used in Altera's reference design for testing the PCIe communication. The driver is provided as source code and can be installed via the make file named `install`. This process requires root access. All other operations for FPGRay can be run by a non-root user.

An important note on changing the driver: When compiling and reinstalling the driver (simply by using the `install` make file with no parameters), it is necessary to reset FPGRay before reinstalling the new driver to avoid freezing the PC.

Basic Properties. The DMA transfer is viewed from the FPGA's perspective, which means a read transfer is the operation of sending data to the FPGA and a write transfer describes the reception of data.

The driver is only able to properly handle one FPGA card on a PC, but the same card can be accessed by multiple driver handles in parallel. As all driver handles use the same DMA descriptors for controlling the transfer, a simultaneous use is only recommended when a second driver instance is used for accessing reserved registers (e.g., to read statistics or reset the device in case of a failure).

The configuration of a DMA transfer (as discussed in the higher level section), the access to reserved registers, or the reset is done by the function `altera_dma_exec_cmd()`. The actual DMA transfer is performed by the function `transfer_DMA_simul()` afterwards. Not configuring the transfer properly before can even crash the PC as the driver is running in kernel mode.

Note that it would be technically possible to do the whole data transfer without using a DMA, but since experiments indicated an unacceptable throughput (only a few MB/s), a function using direct access was discarded. In contrast, the access to the reserved registers is only a transfer of a few bytes, so using the DMA, which would add additional overhead, is not practical in this case. Furthermore, an undisturbed register access during DMA transfer is possible. For data transfer of much data, such as intersection payloads, the DMA overhead pays off, so the DMA is always used for such transfers.

transfer_DMA_simul()

The function `transfer_DMA_simul()` configures the DMA such that it always transfers 1024 data blocks (each being 128-bit wide, the bit width of the PCIe interface). The function awaits the end of the transfer and triggers further transfers if more data should be transferred. As DMA descriptors for configuring write and read access in parallel exist, the function transfers in full-duplex mode when needed. Besides being faster, this also prevents overflows of the FPGA's output buffer by writing data back before the end of the read transfer. Note that a transfer of less than 1024 data blocks (or more by chaining multiple DMA descriptors together) is possible, but `transfer_DMA_simul()` never makes use of this. This makes the code simpler, and in practice the used settings have proven to be a good choice with regard to performance.

If a transfer requires less than 1024 data blocks, the function pads the empty data blocks before reading and crops the unused data after writing back. The only limitation is that a multiple of 16 bytes (i.e., 128-bit, one data block) needs to be transferred. As in such a case padding is transferred, which reduces efficiency, it should not be used too extensively (e.g., by using only 1 data block per call). The function allows the use of 1 to 16,777,215 data units. Note that a data unit (e.g., one ray) can use more than one data block (i.e., more than 128 bit), which means one function call can transfer more than $16{,}777{,}215 \times 16$ bytes of data.

`transfer_DMA_simul()` supports the configuration of the transfer by the struct specified in the higher level section of the API (Section 4.5). With the three parameters "Offset", "Read Pattern", and "Write pattern", any operation FPGRay supports can be performed. As the values are handed over from user space, the driver can handle new functions in future releases without change; only the API needs to be adapted by calling the driver with different parameters. The "Offset" determines how many DMA transfers should be read only before starting with the full-duplex operation. The "Read Pattern" and "Write Pattern" are needed to specify the number of 128-bit data blocks one unit of the operation has in each direction—`transfer_DMA_simul()` uses them to properly synchronize the transfer for a different number of data blocks per unit.

4.7.12 Parameter List

As mentioned, FPGRay is able to be tailored to different target FPGAs. The needed changes can be done mostly by changing parameters, which are technically constants or generics of different entities. All constants regarding ray tracing are collected in one package file (`FPGRay_pkg.vhd`), which also contains the declarations of all components used in FPGRay. Furthermore, in combination with the instantiations of the used floating point and on-chip IPs, all configurations and changes of any IP can be done in this package.

The constants presented here are often used to control the number (which can also be zero) of instances by using the `for generate` and `if generate` program constructs defined in VHDL. FPGRay uses these constructs to synthesize away logic that is not needed for the parameterized design. The constructs are not used if the additionally generated logic does not need many resources (in case the EDA does not find the unnecessary code) and using such constructs would lower the code readability.

The types seen in the following are data types of VHDL signals. Besides numbers (`integer` and `natural` as an `integer` with range from `0` to `max(integer)`), there are the `std_logic` and `std_logic_vector` types. `std_logic` is a bit signal such as a `valid` signal, but as it is able to model electric lines there are additional values besides `'0'` and `'1'` (in fact, there are 9 values). In case of the specified constants below, they should be always set to either `'0'` or `'1'` to represent `false` or `true`. A `std_logic_vector` is a vector of `std_logic` signals. Such vectors are needed for parallel data lines, e.g., any floating point value is in fact an `std_logic_vector` of `BIT_WIDTH_FP` `std_logic` elements.

The following list gives a complete overview over all constants that can be configured:

Constant	Type	Default	Description
Bit widths			
`BIT_WIDTH_FP`	`natural`	`32`	The bit width of a floating point value
`BIT_WIDTH_EXP`	`natural`	`8`	The bit width of the exponent of the floating point value. Per default this is 8 bit for IEEE754 single precision. This constant together with `BIT_WIDTH_MANTISSA` is needed for `fpgray_cmpaleb` (Section 4.7.10) to know the used floating point format

BIT_WIDTH_MANTISSA	natural	23	The bit width of the mantissa of the floating point value
BIT_WIDTH_PRIMIDS	natural	32	The bit width of primitive identification numbers (primitiveids, equals the primitive's index in memory), also used for primitive indices used together with the indices memory space
BIT_WIDTH_RAYIDS	natural	32	The bit width of ray identification numbers (rayids)
BIT_WIDTH_STACKIDS	natural	8	The bit width of stack identification numbers (stackids)
BIT_WIDTH_ALTS	natural	8	The bit width of scene identification numbers (sceneids)
BIT_WIDTH_NUMPRIMS	natural	8	The bit width of the number of primitives a leaf node can contain
BIT_WIDTH_NODEIDS	natural	32	The bit width of node identification numbers (the index of a node in memory)
Parameterization of kdintersect			
NUM_SCENES	natural	16	The number of scenes kdintersect can store in parallel to use them for rays
NUM_INNERNODES	natural	1	The number of kdinnernode instances that are used
NUM_INTERSECTTRI	natural	1	The number of kdintersecttri instances that are used
DINDS	natural	0	DoubleINDices parameter (allowed range 0-1): if 1, one kdleaf instance for every 4 kdnodefetchers is used, else 1 every 8
NUM_STACKS	natural	32	The number of stacks the kdintersect instance has in total
NUM_CACHE_ELEM_PRIMITIVES	natural	32	The number of cache elements each kdprimitivefetcher instance should have
NUM_CACHE_ELEM_PRINDICIES	natural	32	The number of cache elements each kdprimindicesfetcher instance should have
NUM_CACHE_ELEM_INNERNODES	natural	32	The number of cache elements each kdnodefetcher instance should have
NUM_BUFFERELEM_INNERNODEUAD	natural	4	The number of elements the uniquewaitbuffer of each kdnodefetcher should have
NUM_BUFFERELEM_INNERNODEAD	natural	4	The number of elements the waitbuffer of each kdnodefetcher should have
NUM_BUFFERELEM_PRIMITIVEUA	natural	4	The number of elements the uniquewaitbuffer of each kdprimitivefetcher should have
NUM_BUFFERELEM_PRIMITIVEAD	natural	4	The number of elements the waitbuffer of each kdprimitivefetcher should have
NUM_BUFFERELEM_INDICESUA	natural	4	The number of elements the uniquewaitbuffer of each kdprimindicesfetcher should have
NUM_BUFFERELEM_INDICESAD	natural	4	The number of elements the waitbuffer of each kdprimindicesfetcher should have

NUM_BUFFERELEM_INNERNODEOB	natural	8	The number of elements the buffer merging the inner node outputs of both kdnodedecode instances should have (the buffer depicted in Figure 4.7). Please note that this is no on-chip memory.
NUM_BUFFERELEM_NEWRAYBUFFER	natural	256	How many entries the newray buffer of kdscheduler (Figure 4.4) should have
NUM_BUFFERELEM_FINALBUFFERS	natural	256	How many entries each final buffer (Figure 4.4) should have (there is one for each output which is forwarding rays to the output of kdscheduler)
NUM_BUFFERELEM_LEAFOUTPUTBUFFER	natural	256	How many buffer elements the complex version of the build output buffer (Figure 4.11) of a kdleaf instance should have (note that no on-chip memory is used). Is ignored if the simple version buffer is used
BIT_WIDTH_DDR3ADDR	natural	25	The bit width of the address line to the DDR3 RAM controller. Only subject to change when using a different FPGA board with less or more RAM
DELAY_*	natural	-	The constants used to specifiy the latency of each floating point operation. Constants exist for: FPCMP (one for all comparison functions), FPADD, FPSUB, FPMULT, FPDIV, FPSQRT, FPCONVERTTODP (for conversion function converting standard floating point value to double precision), FPCONVERTFROMDP (for converting back double precision to the standard used representation), DPFPMULT, and DPFPSUB

Additionally, there are also constants whose values are computed automatically based on the constants set above (but it may sometimes be useful to manually change them too):

Constant	Type	Default	Description
Bit widths			
NUM_STACKBLOCKS	natural	1	The number of blocks in which each kdstack instance should be (equally) divided, used for allowing a better timing by lowering the path lengths (this is the setting choosing how many stackcommunication instances one kdstack should use)
NUM_KDSTACK_UNITS	natural	1	The number of kdstack instances in one kdintersect instance should be used. This is the setting defining if a single kdstack instance should be used or the abstracted version where the kdscheduler mimics multiple kdstack instances that they exist alone

BEGINRETURNINDX	natural	NUM_INNERNODES/2	The index of the innernode chain element, where the reinsertion of rays from the intersection path should start (Figure 4.4), assuming the first element getting data from the newray buffer is defined as elment index 0. May be changed to a manual index for very small or very big designs to optimize the data flow to normally encountered scenes, for which the simple default computation is not viable

It should be noted that FPGRayIF (Section 4.1) relies on additional constants that are only used internally. These parameters cannot be found in FPGRay_pkg.vhd as FPGRayIF, as the interfacing entity, is exceptional, e.g., by not using bit width constants for all signals (but rather use fixed bit widths for optimized data transfer between the PCIe interface). Because of this, changing constants for bit widths in FPGRay_pkg.vhd does not affect all signals in FPGRayIF and makes manual changes necessary. All constants tailored to the current PCIe interface are therefore collectively stored in FPGRayIF.vhd. In particular, this applies to the following constants:

Constant	Type	Default	Description
GENINTERS	std_logic	'0'	Determines if a dedicated kdintersectsp instance to be used as a function should be synthesized
GENINTERTRI	std_logic	'0'	Determines if a dedicated kdintersecttri instance to be used as a function should be synthesized
GENTRAV	std_logic	'0'	Determines if a dedicated kdinnernode instance to be used as a function should be synthesized
GENkdINT	std_logic	'1'	Determines if a dedicated kdintersect instance to be used as a function should be synthesized
INTERTRI_GENBS	std_logic	'0'	Determines if the b values (Figure 4.6) of the dedicated kdintersecttri instance should be computed and thus written to software. This is not supported for the instances inside of kdintersect as the values got not space reserved in the instance to be able to store them up through the data flow until it could be forwarded to software
NUM_BLOCKS	natural	1024	The number of PCIe data blocks that are sent with one DMA transfer
NUM_BUFFERELEM_OUTPUTBUFFER	natural	8192	The number of elements the output buffer of FPGRayIF should have

Lastly, the profiling and debug registers of FPGRayIF, which can be read from software, are always 32-bit wide values. In detail these are:

Address	Description
131,072	Returns the number of readily computed data blocks the output buffer of FPGRayIF contains. As data may not be returned in-order, the data blocks which contain the data do not have to be consecutive in the buffer
131,073	Returns the functions which are synthesized at the current FPGRayIF instance. From Least Significant Bit (LSB) beginning, the 4 bits indicate `INTERS` (intersect a sphere with kdintersectsp), `TRAV` (traverse an inner node of a k-d tree with kdinnernode, `INTERTRI` (intersect a triangle with kdintersecttri), and `kdINT` (intersect against a k-d tree scene with kdintersect
131,074	Returns the number of resets which were performed since power up

CHAPTER 5

Results

The evaluation of FPGRay showed that it is in the current form not usable for the intended use as hybrid solution in combination with a PC. Nevertheless, after a closer look at some specific characteristics of FPGRay, one can see that this goal can be reached by removing some of the found bottlenecks. But as during the writing of this thesis, the first GPUs supporting ray tracing operations with fixed-function hardware have been introduced and are now already available, such hardware is the better solution for most of the target audience. This is due to the much lower costs by saving the development of custom hardware. How and under which circumstances a further use of FPGRay or similar custom hardware designs are still a viable option is elaborated in more detail in the last section of this chapter.

A variety of tests have been performed on FPGRay. The basic evaluation was performed by synthesizing multiple designs for the Arria V GX[1] FPGA. The FPGA is contained on the "Arria V GX Starter Kit" from Intel, which is a PCIe card made for development and evaluation purposes. The PCIe card was plugged into the PC which was used for the software-related benchmarks too.

Either the number of basic logic elements or the routing capabilities of the FPGA were the factors that limited a further increase of FPGRay's parameters. The hardware multiplying units, registers (i.e., the logic elements that introduce the clock and thus separating stages), and the on-chip memory were never the limiting resources. For more details on the FPGA's intrinsics mentioned in this paragraph, please refer to Section 2.3. Furthermore, the HardIPs for the PCIe interface and the DDR3 RAM controller were used. Note that the PCIe interface supports up to 2GB/s, either via 8 lanes at Gen1 or 4 lanes via Gen2, and the RAM controller supports up to DDR3-533. As the card's two 128 MiB DDR3-1066 chips are fast enough, the DDR3-533 specification is used. Although the

[1]The specific model is the Arria V 5AGXFB3H4F35C4

card can load a design on power up from the onboard memory, each design was loaded via the PC.

The host PC consists of an AMD Ryzen 3700X with 32 GB DDR4-3200 RAM and runs Manjaro Linux KDE with kernel 5.4.

5.1 Test Scenes and Designs

Multiple designs were synthesized or simulated to evaluate the characteristics of FPGRay. All hardware designs shared the following basic characteristics:

- The buffer sizes to stall data and prevent overflows are set to 256 entries, which is the minimum possible size for on-chip buffers. The only exceptions are the stalling buffers before the kdprimitivefetcher (Section 4.3.3) and kdprimindicesfetcher (Section 4.3.3) instances (which use 512 entries).
- The delays of the floating point operations are set to the minimum possible values for these IPs.
- FPGRayIF (Section 4.1, and with it all ray-tracing-related operations) run at a clock speed of 100 MHz.
- kdintersect (Section 4.2) and FPGRayIF both use the default parameters as stated in Section 4.7.12.

Based on these constants, the following designs were used:

Design	No. instances	No. entries	No. stacks	statistics
1I	1/1	32	32	No
1Is	1/1	32	32	Yes
1ILs	1/1	64	32	Yes
1IXLs	1/1	128	32	Yes
1IXLSLs	1/1	64	128	Yes
1ILSLs	1/1	64	96	Yes
1IXLSXLs	1/1	128	128	Yes
2Is	2/1	64	64	Yes
3I	3/1	64	96	No
1ILc	1/1	64	32	-
1IXXL	1/1	256	32	-
4I	4/1	256	128	-
8I	8/1	256	256	-
8I2	8/2	256	320	-

The “No. instances” indicate the number of traversal and the number of intersection instances, the “No. entries” the number of entries per cache, and “statistics” stands for

the ability of the design to collect additional statistics such as latencies. The blue colored designs are synthesized and usable on the FPGA, the orange colored are only available for simulation as they do not fit on the used FPGA. This was necessary as the limit for the FPGA was the 3I design, which already had a basic logic resource usage at around 90%.

The naming conventions of the design should be read as follows: All digits are used to indicate the number of the computational instances. The digit on the left of the separator "I" is for the number of traversal instances (i.e., kdinnernode, Section 4.3.1). The number on the right of "I" indicates the number of intersection instances (i.e., kdintersecttri, Section 4.3.2); a missing number stands for a single instance. As for 1I, multiple designs were used, the size identifiers L, XL, and XXL mark parameters with a setting higher than the default 1I design. If the size identifier is followed by "S", it is identifying a design with a higher number of stacks. If the identifier is followed by "s", the number of cache entries are increased. The "s" at the end of a design identifies the presence of the mentioned statistics functionality.

Note that the simulation-specific designs contain only the kdintersect instance, i.e., they were not simulated with its PCIe connection via FPGRayIF. Furthermore, they do not include the statistics functionality because all desired information can be taken from the simulation anyway. To allow a direct comparison between these different designs used for the FPGA and simulation tests, the designs 1ILs and 1ILc (where c is the abbreviation for "comparison") use the same parameters.

To test the designs, five scenes were used: Two from Benedikt Bitterli's rendering resources page [Bit16] and three from the official test scenes of pbrt-v3 [pbr20]. In detail, these were:

- "Pontiac GTO 67": referred as "car2", using 1600x900@512spp for the PC benchmarks, 160x90@1spp with FPGRay
- "Japanese Classroom": referred as "classroom", using 800x450@512spp for the PC benchmarks, 80x45@2spp with FPGRay
- "bathroom": using 1200x760@512spp for the PC benchmarks, 120x76@1spp with FPGRay
- "coffee-splash": referred as "coffee", using 1000x800@512spp for the PC benchmarks, 100x80@2spp with FPGRay
- "head": using 1280x720@512spp for the PC benchmarks, 128x72@1spp with FPGRay

All of these scenes were modified to use the k-d tree accelerator, different resolutions, and different spps. See Figure 5.1 and Figure 5.2 for a rendering of the test scenes.

Note that all scenes use a path tracer, the bathroom scene uses BDPT.

Figure 5.1: The test scenes from Benedikt Bitterli's page. "Pontiac GTO 67" is seen on the left and "Japanese Classroom" on the right.

Figure 5.2: The test scenes from the official pbrt-v3 page. "bathroom" is seen on the left, "coffee-splash" in the middle, and "head" on the right side.

5.2 Synthetic Tests

The evaluation of the synthesized designs was done with a synthetic test program. This was done as a renderer supporting the desired batched call was not implemented, as such an implementation would have exceeded the scope of this thesis[2]. To benefit from the parallelism FPGRay provides, multiple rays per function call, via batches, need to be transferred.

The synthetic test program uses test dumps containing the scene's data and a large number of rays which should be intersected. The test dumps can be produced by rendering a scene with k-d tree with our modified pbrt-v3 version called pbrt-fpgray. These dumps additionally contain pbrt's intersection results such that the test program can verify FPGRay's outputs. It should be noted that the verification is done by accepting small differences in the `rtmax` values between FPGRay and pbrt as the implementations differ slightly[3]. Furthermore, because of these differences the obtained `primitiveids` are accepted too when the `rtmax` values differ within the acceptable margin. For each benchmark run, such occurrences are additionally collected for manual inspection.

The tests performed with the synthetic test program load the scene dump to FPGRay

[2]pbrt-fpgray's intersection method using FPGRay (presented in the API section, Section 4.6) supports the call with only 1 ray/Batch.

[3]The differences of `rtmax` values lie within the magnitude of 10^{-6}.

and send multiple ray batches to simulate the use in a renderer. For each benchmark, three runs over 10,000 batches were performed. Each run starts at the first ray of the test dump and fills the desired batch size with subsequent rays. Each of the 10,000 batches starts with the second ray of the previous batch as the new first. Initially, this overlapping was done to test different rays without requiring many different rays. However, it was kept to potentially break the locality of neighboring rays from time to time; as seen in later tests, this works, as the caches are not large enough to have the data of old rays still present. The reported runtime is the average of all batches over three runs. In the benchmarks, the used batch sizes can be identified by the rays/batch (r/b) values. For comparison, benchmarks without FPGRay were performed with pbrt-fpgray. For these tests, the default settings of pbrt-v3 were used, with the measurement of the intersection throughput as only activated extension. As this means that pbrt-fpgray is using the same code as pbrt-v3 during the actual rendering, we denote tests performed with pbrt-fpgray as pbrt-v3 to express the use of the software-only renderer.

The time measurements take only the API call to FPGRay into account, so the extraction of rays from the test dump and the verification afterwards is unaccounted for. The measurements were done by using the C++ `std::chrono::high_resolution_clock`. The measurements of the rays/second (r/s) are done by pbrt-fpgray itself by evaluating the collected statistics of the profiling system from pbrt-v3 at the end of the run. Only the calls tagged “ray intersections” are taken into account, “shadow ray intersections” are not included. This resembles the rays the test dump is containing as only calls for ray intersections are used to dump rays.

As stated before, the resolutions and spps of the test scenes were changed. When using any test on FPGRay, a smaller resolution and spp setting of a scene was used. These changes were necessary to test a big part of the rendered frame with the small test dumps in order to avoid a throughput distortion by rendering only over a small part with high locality or beneficial frame contents. Unfortunately, using test dumps generated from the renderings using the settings for the software renderer was not feasible. This would have resulted in long runtimes and large files sizes for the test dumps for the high number of scenes and designs that were tested. As an example, the fastest scene for pbrt-v3 was the head scene with around 30 seconds; running the synthetic test program over the corresponding test dump for the fastest 3I design only would have taken approximately 17 hours. The bathroom scene as worst case with 14 minutes with pbrt-v3 would have taken 14 days with the 3I design. These extreme runtimes would have been caused by the runs with small batch sizes as seen later on.

Note that the test dumps are generated as plain text files to allow an easier debugging. However, even if optimally sized binary dumps would have been used, the dump files for the amount of rays needed for the high resolution and spp settings would have taken around 120 GB for all five scenes.

On the other hand, pbrt-v3 needs a higher number of rays to obtain valid benchmark

results[4], therefore the higher resolutions and spps were necessary for the software rendering.

To verify the assumption that this benchmark procedure was not beneficial for FPGRay, a few examples were tested. The classroom, bathroom, head and car2 scenes were tested with the first part of the test dump that is generated by the scenes using the settings from pbrt-v3's runs. The same tests as in this section were performed with different starting rays for the first batch (i.e., rays 0, 100k, 200k, 500k, 1M) to get a higher accuracy of the achievable throughput for the whole test dump. Furthermore, a test with no overlapping (i.e., a following batch uses the first ray of the dump not being in the previous batch) was performed for 10,000 batches (again averaging over three of those runs). For these tests, only a batch size of 1024 was tested to save time. For the classroom and bathroom scenes, the assumption holds and slightly higher throughputs from 1% to 10% were seen. For the car2 scene, the 100k, 200k, and non-overlapping results achieved a lower throughput of 7% and only for the other tests a comparable to 1% higher throughput was achieved. The head scene was even much faster than in the following tests, from around 60% to a factor of 10. As another test enlightens, the FPGA would achieve much narrower throughputs for the head scene for those different settings, but the PCIe interface limits the smaller test dump. Using the throughputs measured without the overhead of the interface, the head scene is, similar to the classroom and bathroom scenes, only slightly faster by 10%. As a conclusion it can be stated that the different scenes settings are not beneficial for FPGRay.

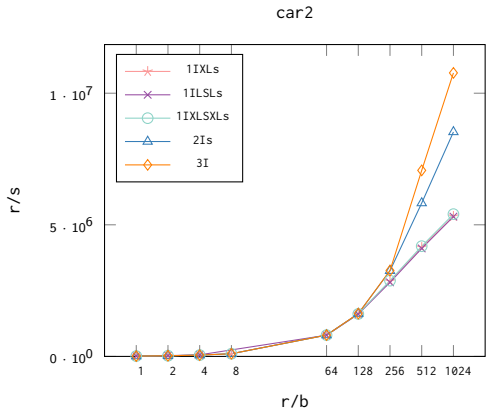
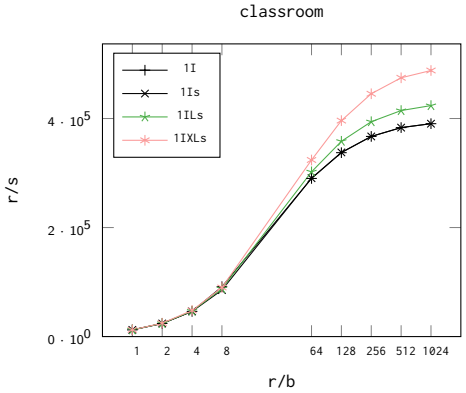


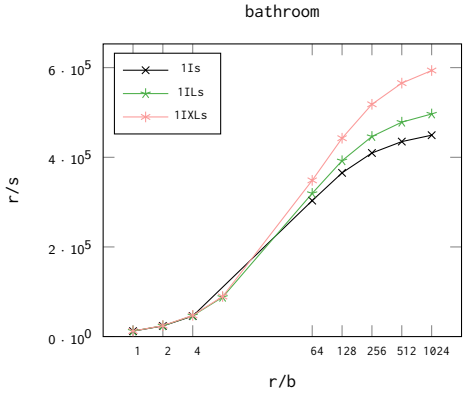
Figure 5.3: Synthetic test run results performed on the car2 scene.

Results. Figure 5.3, Figure 5.4 and Figure 5.5 depict the results of the test runs. The car2 scene was merged to a single plot as it was the only one which showed a very different behavior not being affected by most parameter changes. To verify that designs supporting the collection of statistical data, which are marked with an "s" in their name,

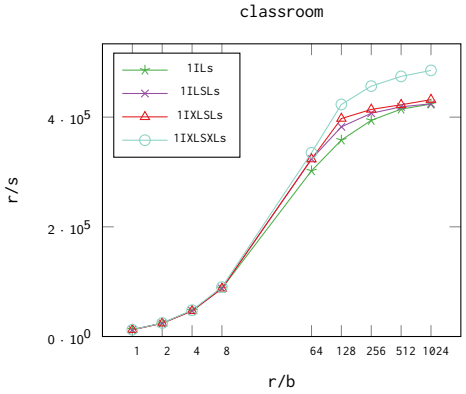
[4]pbrt-v3 measures profiling information on 0.01 second basis, so a short rendering results in a high distortion of the throughput



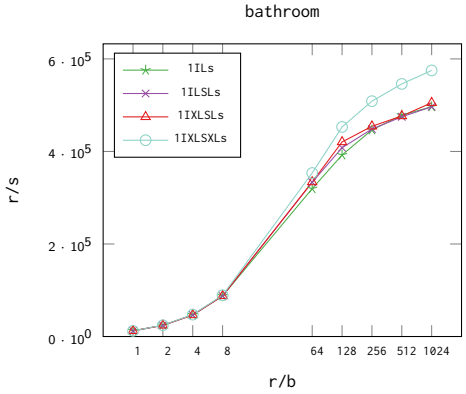
(a) Different cache sizes



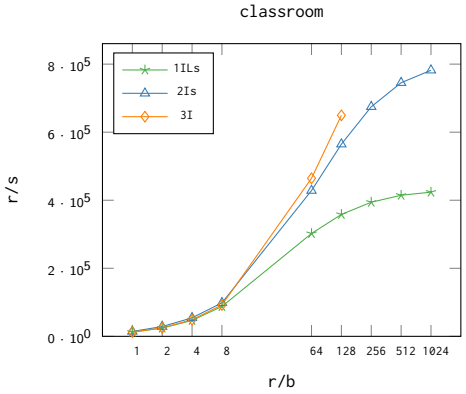
(b) Different cache sizes



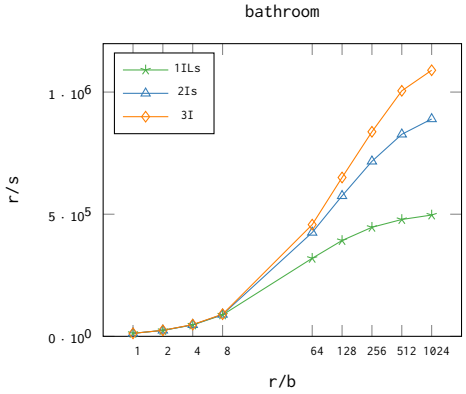
(c) Different stack (and cache) sizes



(d) Different stack (and cache) sizes

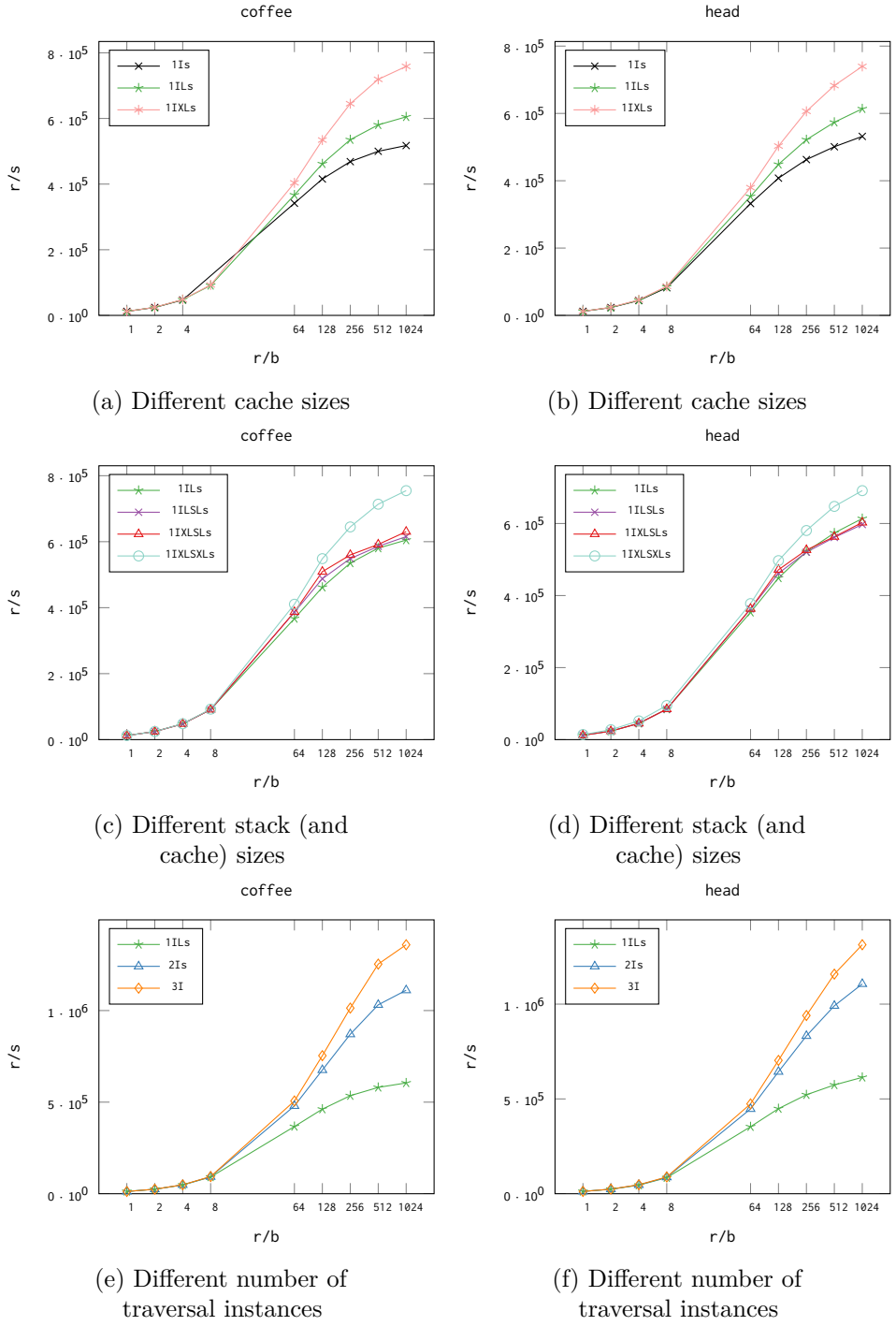


(e) Different number of traversal instances



(f) Different number of traversal instances

Figure 5.4: Synthetic test run results performed on the classroom and bathroom scenes.



(a) Different cache sizes

(b) Different cache sizes

(c) Different stack (and cache) sizes

(d) Different stack (and cache) sizes

(e) Different number of traversal instances

(f) Different number of traversal instances

Figure 5.5: Synthetic test run results performed on the coffee and head scenes.

do not suffer from a throughput penalty, one run with a design differing only in the statistical functionality was performed with the classroom scene. The 1I and 1Is designs depicted in Figure 5.4 (a) deliver exactly the same throughput. Differently to software, where debugging information causes a performance penalty, this is possible as additional hardware is used, which does not alter the rest of the design. For this reason, each design supports the collection of statistics if enough hardware resources are available. Note that the batch sizes are not limited to powers of two: Any batch size can be used without detrimental effects on performance. However, the batch size is still limited to a maximum value of 1024 rays, as FPGRayIF's output buffer uses 2048 entries, which provides enough space for one finalized batch and a second ongoing. Using more than 1024 r/b can cause data loss. Note that because of a problem which is circumscribed to FPGRayIF, but not properly identified further, the tests with 16 and 32 r/b are omitted, as in this range a freeze of the interface has often been experienced.

Analyzing the Results. At the first look, for all designs and scenes tested, a small batch size yields in low throughput. The bad utilization of the hardware's pipeline is apparent. On the other hand, for all designs except the car2 scene (and much less prominent, the head scene), when increasing the batch size, at some point the linear throughput gain flattens. The design is fully utilized and larger batches only allow marginal gains. When a bigger design is used, the point the throughput curve flattens is delayed. For the bathroom and coffee scenes when using the 3I design, this is seen relatively late at 512 r/b, but as a significant flattening. Such gains after full utilization are possible as the design is not fully utilized at the beginning and end of the batch were not all possibly computable rays are processed; a higher batch size raises the ratio between full utilization and the beginning and end parts. As all plots concerning the stack sizes show, for 1I designs, the stack size is chosen well enough to not be the bottleneck. In contrast, the cache sizes and the number of traversal instances limit the throughput for all except the car2 scene, which is only limited by the number of traversal instances. When choosing caches that are four times larger, the throughput of the 1I design increases between 25% for the classroom scene to 47% for the coffee scene for 1024 r/b. When using a second traversal instance but leaving the number of cache entries per fetcher the same, all scenes except the car2 scene showed a 79 - 85% higher throughput for 1024 r/b.

The car2 scene behaved completely different compared to all other scenes on the used designs when increasing the batch size. It is not limited by cache sizes as well as stacks for the 1I designs; all deliver the same throughput. On the other hand when using more traversal instances, the throughput increases linearly even up to 1024 r/b. From one to two instances 60%, and from one to three instances 102% higher throughput was observed.

The head scene behaves for most designs like the others. When increasing the number of traversal instances, the flattening throughput gain is much less prominent than for other scenes; the curvature is similar to that of the car2 scene.

The measurements for all except the car2 scene are as expected: Increasing the batch size,

a linear throughput gain can be reached until the number of computational instances limit it. Additionally, a limit of the caches can be seen. As we elaborate in more detail later, the memory and with it the caches are highly utilized. The car2 scene, which has a lower utilization for the memory subsystem on average, is therefore not memory bound. Furthermore, the head scene is also much less memory bound, which is another similarity to the car2 scene. For all scenes, the similar progression suggests that for bigger designs, significant throughput gains can be reached by using higher batch sizes. They also show that if FPGRay should be used without loosing too much performance, a high batch size is important. Note that for the computational instances to be the limiting factor, the stack sizes need to be set properly. As seen, this is in fact the case for the 1I designs but may not be the case for bigger designs, especially for 3I, which is near the limit of the FPGA's resources. The number of stacks is the primary limiting factor and even more limiting than the number of computational instances. The reason is the number limits the concurrently computable rays and thus prevents proper utilization; the linear gain for increasing batch sizes would flatten at the point the number of stacks equals the batch size.

At first glance, these results seem to be in discrepancy to the expectations when knowing the design's intrinsics. The design is able to retain a throughput of one as long as the fetchers do not stall due to high memory latency. But to deliver this throughput, the design must contain enough computational instances to not reroute the same ray multiple times through the same instance. If a reroute happens, the ray is processed by the same instance (e.g., a traversal instance) a second time, which would stall the insertion of a new ray. This stall would lower the throughput only marginally. But if this happens too many times, a severe throughput penalty can be observed. This relation between throughput and the number of computational instances is seen for nearly all scenes, which explains the limiting factor being the number of instances. The car2 and head scenes, on the other hand, does not suffer as much from this relation, as many rays end after a low number of bounces (or even without entering kdscheduler's computational parts, as they do not intersect the scene at all). This explains why the RAM is not that highly utilized for the car2 and head scenes, as many rays do not even require any data. The reason is the scene itself, which contains no background, i.e., every ray that does not intersect the car or the floor, respectively the head for the head scene, is not intersecting anything at all. The scenes are therefore facilitating a throughput near to the optimum that is attainable with the current design, where no computational instance lowers it by being used two times for the same ray. For the other scenes, this optimum can not be reached by only 3 traversal instances (and one intersection instance). Note that the maximum throughput for these designs would be 50 Mrays/s, as the FPGRayIF interface needs two cycles for one ray and runs at 100 MHz.

Hardware vs. Software. The following table lists the highest observed throughputs for the scenes tested with FPGRay compared to the software rendering by pbrt-v3:

in kr/s	pbrt-v3	FPGRay	factor
classroom	10 014.5	782	12.8
car2	18 527.3	10 771.1	1.7
bathroom	8954.1	1089.3	8.2
coffee	9726.6	1359.2	7.2
head	13 131.6	1313	10

When compared to a pure software implementation, it can be seen that the software is much faster. The well-performing car2 scene is still by a factor of 2 times faster with CPU-only rendering and the much slower classroom scene is one magnitude (a factor of 13) slower on the FPGA. The software implementation yields an 8 to 10 times higher throughput for all scenes. This means that the presented design can not be used for the proposed hybrid approach on modern PCs as it would slow down the rendering. The utilization of both specialized hardware and a PC is therefore not achievable with our setup.

Nevertheless, some bottlenecks have been identified during testing which point to possible improvements or extensions to be able to reach that goal. A more detailed description of those bottlenecks follows in the next section.

5.3 Special Tests

In this section, we present a few tests that focus on a few key aspects in order to explore the problems that lead to a low throughput. Furthermore, some predictions of the performance increase due to particular improvements are presented. Lastly, we present more in-depth comparisons between the throughput of CPU-based vs. GPU-based renderers, as well as the power consumptions of all shown configurations.

5.3.1 RAM And Cache Latencies

Of the designs presented in Section 5.2, the ones designated with “s” are capable of tracking statistics, such as cache and RAM latencies. With these measures, the hit rate of the caches and thus their efficiency can be evaluated. When the hit rate is high, only a low number of the data requests are forwarded to the RAM, i.e., the cache is big enough to store data which is often requested.

By investigating these statistics, some interesting conclusions can be made. For example, if we look at the hit rate of the caches, the efficiency of the caching can be seen. The following table lists the average hit rates of the fetchers (using the last 2048 fetching requests as data):

		classroom	bathroom	coffee
	r/b	Hit rate (%)	Hit rate (%)	Hit rate (%)
1Is	64	19	22	59
	1024	25	27	35
1ILs	64	26	27	48
	1024	34	38	46
1IXLs	64	35	37	59
	1024	44	49	56
2Is	64	30	31	53
	1024	38	42	49

The test scenes are split into two groups. The first group with the head and car2 scene has a hit rate of 100%. This is another advantage which allows those two scenes to achieve a higher throughput without encountering a flattening in throughput increase at specific r/b.

The other group consists of the remaining scenes that, with the exception of the coffee scene reaching 59% at its best, barely reached a 50% hit rate. This means that the majority of data accesses is directly served by the RAM. It can also be seen that the 1I designs can nearly double the hit rate by using a cache that is four times larger compared to the other ones. 2Is achieves a higher hit rate than 1ILs despite using 64 entries per fetcher for both designs. But 2Is has slightly more cache entries, as it has one cache more than the 1I designs due to the additional traversal instance. The fluctuations between 64 r/b and 1024 r/b show the locality of data which is preserved longer for 1024 r/b. The reason is that inside of a batch, subsequent rays are spatially close (i.e., they have a high locality). But when switching between batches, the last ray has a low locality to the first ray of the following batch.

Note that the high hit rate of 100% for the two scenes (which would mean no RAM access at all) can be reached as the observed data is just collected from the last 2048 rays during rendering, after the initial requests were fetched already. This is also the case for the other scenes, so the significant difference of the hit rates between those two groups of scenes are assumed to be representative.

The tracked latencies for fetching from the cache or the RAM reveals significant differences, namely 3-4 cycles against 17-41, with an overall latency of 14-21 cycles, which is inversely proportional to the cache sizes for the group of scenes with a low hit rate. These latencies are similar to the access latency from kdintersect to the RAM alone (18 cycles on average). Because of the high hit rates for the car2 and the head scene, a much lower average of 4 cycles had been observed. Note that the latency for accessing the RAM increased over time, which indicates that it is at the limit.

These findings lead to two conclusions:

- The number of cache entries are too low. By quadrupling their number, the hit rate nearly doubles. It is still low with around 50%, as hit rates of over 90% are

often seen in practice.

- The locality of the data allows a higher cache efficiency. Using a higher batch size and thus more locality in the data, the hit rate increased between 25-32%. Increasing the locality can be done independently to the cache sizes with different sampling patterns in the software (e.g., by using tiled rendering).

As shown, the hit rate can be increased. This not only lowers the utilization of the RAM but further increases the throughput due to much lower fetching latency and additionally causes less stalling in the buffers before them. Furthermore, a lower latency for each ray results in more rays being able to be processed at the same time.

5.3.2 FPGA Runtimes

Another statistical variable which is taken by the "s" variants of hardware designs is the runtime of kdintersect alone, i.e., the raw intersection time without the overhead of the data transfer from hardware to the API in software. In fact, two values are taken: the raw time (being only the latency of a ray through kdintersect) and the theoretical output time. The theoretical output time is adding the latency caused by not reading each finished ray instantly but rather waiting until it is in turn. This means, the consecutive access of the DMA is simulated to assess the moment the results could be read back. This moment is compared to the time instant forwarding the ray to kdintersect for computing the latency. The table shows the averaged r/s for the last 2048 rays:

	rays/Batch	kr/s	kr/s (theoretical)	kr/s (software)	factor
classroom					
1ILs	64	352.2	351.9	302.2	1.16
	1024	401.4	401.4	423.7	0.95
2Is	64	538.4	537.4	428.4	1.25
	1024	739.4	739.4	782	0.95
bathroom					
1ILs	64	341	340.8	319.5	1.07
	1024	474.3	474.3	496.6	0.95
2Is	64	550.2	548.5	425.1	1.29
	1024	892.8	892.7	889.8	1.00
coffee					
1ILs	64	410.2	409.3	367	1.12
	1024	607.6	607.5	604.9	1.00
2Is	64	646.9	643.3	479.5	1.34
	1024	1132	1131.9	1110.9	1.02

	rays/Batch	kr/s	kr/s (theoretical)	kr/s (software)	factor
head					
1ILs	64	7960.8	7950.9	353.4	22.50
	1024	9502.6	9501.7	613.7	15.48
2Is	64	12 191.9	12 168.7	447.3	27.21
	1024	16 664	16 661.2	1106.4	15.06
car2					
2Is	64	5358.0	5165.1	811.8	6.36
	1024	11 282.5	11 117.7	8527.4	1.3

When comparing these latencies with the actual r/s measured by software (as done for the synthetic tests), it shows that kdintersect respectively the hardware-accelerated intersection gets slowed down by the data transfer between software when using a small number of r/b. The impact on the speed from the software r/s to the theoretically achievable r/s ranges from a factor of nearly zero up to 27. The impact is gone for most scenes when using 1024 r/b. For the classroom and bathroom scenes, the throughput with the overhead of the data transfer is even higher than the estimated theoretical limit (theoretical r/s). But as the FPGA runtimes show a few rays only and the software was benchmarked over multiple thousands, this discrepancy can be explained by fluctuations in the runtimes. Nevertheless, some scenes are even limited by the interface when using 1024 r/b. For the car2 scene, the overhead is acceptable, but the head scene is still heavily limited. The current data transfer approach is therefore not optimal. The scenes reaching the highest theoretical throughput and suffering from the used data transfer dramatically indicate that the transfer with the most scenes is less problematic because of the low throughput that is needed to utilize FPGRay on the current hardware. It also indicates, besides the high hit rate, that the head scene is small enough to avoid too many rays going through the same instances inside of kdintersect multiple times just as the car2 scene, respectively also has a high number of rays not intersecting the scene at all.

5.3.3 Simulated Tests

As the used FPGA is too small for big designs, the designs 1ILc, 1IXXL, 4I, 8I, and 8I2 have been tested through simulations. This is done by using the Modelsim software, included in Quartus. The simulation is done with kdintersect only and the test bench feeding it with data is faking the RAM with an on-chip memory in the test bench (this is done as it dramatically increases the simulation speed). As the on-chip memory is limited in size, the classroom scene, which has a low number of primitives, was used. Because it is the worst performing scene, it can be assumed that the other scenes would achieve at least the same throughput. The `statisticsbuffer` entity used in FPGRay on hardware designs to measure the runtime of operations in clock cycles is also used here to measure the runtime of rays (which is used for computing the rays/second).

As the benchmark is simulating only kdintersect, the overhead of the PCIe interface is avoided and thus the results can be compared best with the FPGA runtimes of the section before. However, the simulations should be seen as approximations only. This is because the simulation is only done at behavioral level, so there is no proof that the simulated design runs with the desired timing on a real FPGA.

The test bench (i.e., a VHDL code file which is not synthesizable but made for simulation only) uses the scene's and ray's data from the same test dumps that were used for the real benchmarks, so the parameters for the classroom scene are also 80x45@2spp. In contrast to the synthetic tests (Section 5.2), each simulation is only performed once (as the runs are deterministic) and always start with the first ray of a test dump. Furthermore, only one batch per batch size is tested and not multiple runs (as the simulation is very time consuming). The latency of the simulated RAM interface is set to 18, which is the actual measured average latency observed during the tests on the FPGA. The designs introduced in Section 5.1 were using the same parameters as the synthesized designs with the exception of using no resource optimization for the stacks.

Figure 5.6 shows the results of the simulations. Note that the missing results for higher batch sizes are due to problems with the designs. The investigations showed that an overflow happens in the design when intersecting too many rays (for 1024 r/b with 8I it was identified at the buffer stalling primitives before the fetcher). This issue could not be fixed even when using bigger caches, a faster RAM access would be necessary.

Note that, 1ILc was around 20% faster than its synthesized equivalent 1ILs in the synthetic tests, even though both designs utilize the same design parameters. But as the simulation only computes one run and always uses 18 cycles for RAM access, these benchmarks can not be compared exactly. Lastly, in comparison to 1ILc consisting only of kdintersect, 1ILs is a synthesized design that uses FPGRayIF and the PCIe interface; the results depicted come from the synthetic tests which already include any overhead of the data transfer.

The results show that increasing the number of computational units allows a higher throughput. E.g., if we compare 1IXXL with 8I2 (using the same cache sizes), the throughput for 128 r/b is roughly 4 times higher. But even for this highest obtainable throughput, the software-only rendering is slightly over 4 times faster. Nevertheless, by using a bigger design on a hardware with more resources (and a faster RAM), the throughput can be increased further. The tests allow the conclusion that the bottleneck is in fact the amount of resources on the FPGA and not the design. Furthermore, the throughput increase with more computational units is not only seen in the simulation: when comparing the synthetic test results of the classroom scene with the 1ILs and 2Is designs (using the same number of cache entries), a nearly doubled throughput was observed.

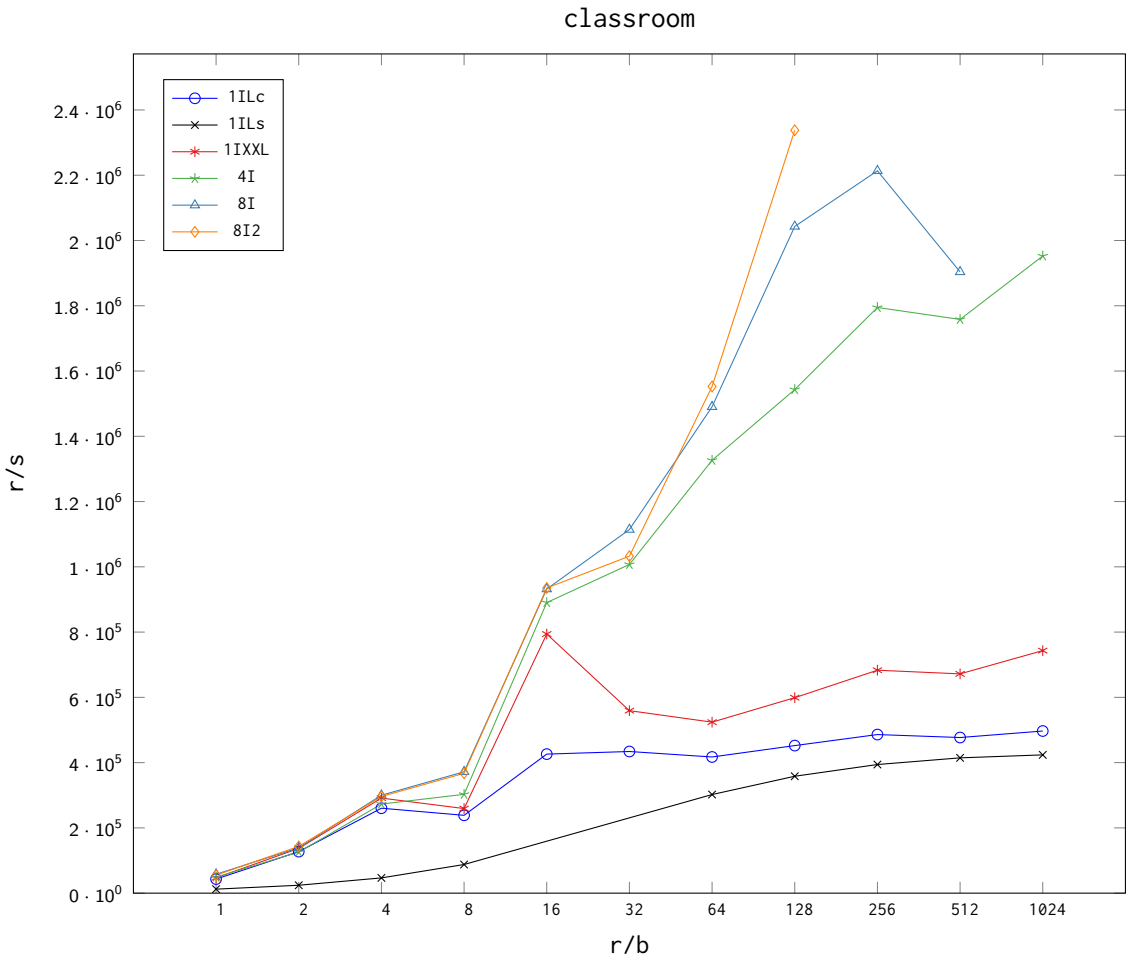


Figure 5.6: The results of the simulated intersection operation of kdintersect for the classroom scene. Note that 1Is is the synthetic test's result for reference.

5.3.4 Latency

In this thesis the term latency was often used. Whilst performing the benchmarks of the FPGA runtimes (Section 5.3.2) and simulations (Section 5.3.3), the actual latencies were directly observed: The runtimes the "s" designs deliver are the latencies of each ray in the unit of cycles and the simulated tests can be depicted as timing diagrams (Figure 5.7). The runtimes show for the slowest and fastest renderings an average latency of 772 cycles for the car2 scene and 10721 for the classroom scene. This represents the throughput, which is depicted in the plots of the synthetic tests and shows that the number of computational instances each ray in the classroom scene traverses through is much higher (around 10 times when taking into account that the lower hit rates of the fetchers additionally increase the latency by the fetchers). The other scenes' rays were intersected in this range of cycles, which means that a ray in these scenes traversed less

computational instances than those of the classroom scene on average.

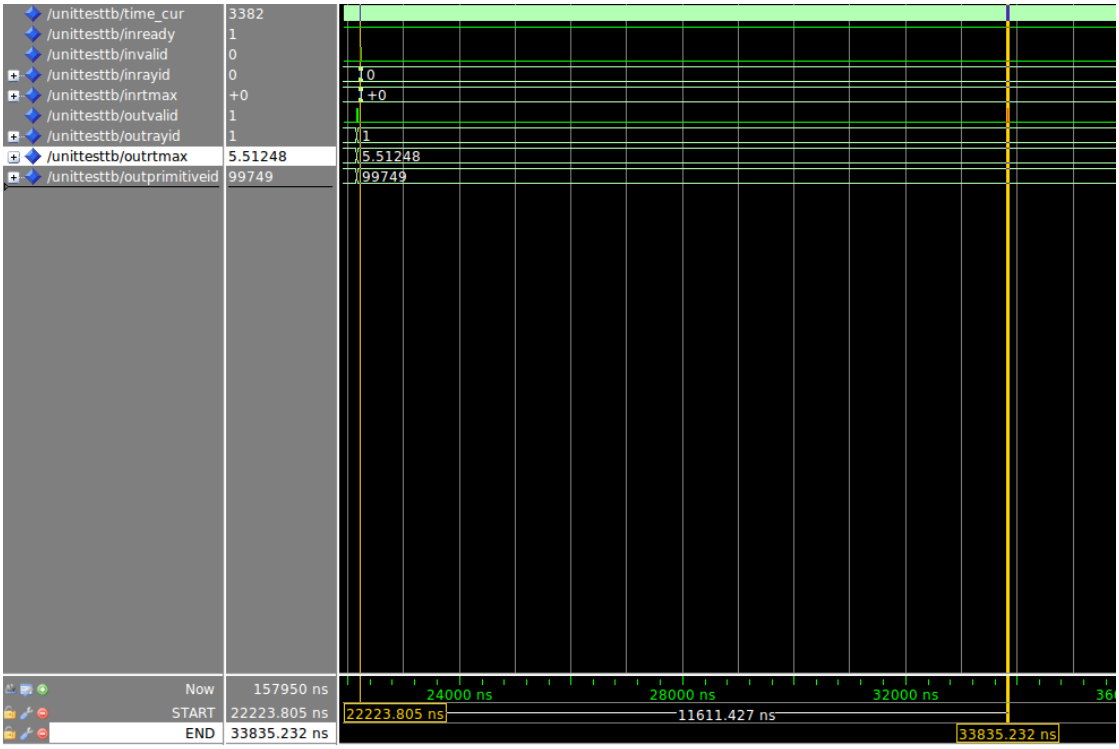


Figure 5.7: The timing diagram of a simulation of 4I with the classroom scene showing the input and output of a ray from kdintersect.

For a more detailed view, one simulated ray intersection during the simulated test benchmark of the classroom scene is taken and its latencies are further explored: The 128th ray of the 256 rays batch is taken, as it is the first ray of this batch size which was not intersected by a preceding batch already. Furthermore, this batch size is high enough to utilize the cache and thus give a good example of an arbitrary ray in the middle of the intersection batches where some data still needs to be fetched from RAM. This ray has a total runtime and thus latency of 30173 cycles. Independent from the scene, the current hardware design has fixed latencies of 14 cycles for a node traversal and 77 cycles for an intersection. Furthermore, kdintersect's preparation for the initial bounding box intersection test has a latency of 27 cycles. The following table shows the number of traversal (t), intersection (i), node fetcher of the innernode chain (nf), node fetcher of the intersection path (nfp), primitive fetcher (pf), kdleaf (l, Section 4.3.6), and number of kdcompare (c, Section 4.3.7) uses of the observed 128th ray. The used simulated design was 1Ic, but the number of calls are the same regardless of the design; only the instance which works on a specific call may vary on bigger designs.

t	i	nf	nfp	pf	l	c
46	4	46	9	4	10	1

The measured overall latency is not the sum of the latencies of these calls. The reason is that the latencies of stalling before the fetchers is not taken into account in the presented measurements. Furthermore, some operations are processed in parallel, which lessens the detrimental effects of individual latencies.

5.3.5 Power Consumption

Unfortunately, the power consumption of the FPGA card can only be read from a PC with the Windows driver. But as FPGRay uses a Linux driver for operation, the on-board consumption measures can not be performed. To still deliver a rough approximation for the power consumption and thus compute the efficiency of the design, a power meter was used. The concrete model[5] was plugged between the PC and the power plug and measured with a tolerance of 1%. As the measurements were done for the complete PC, some fluctuations need to be considered. The following table lists the measured consumptions:

	consumption (Watt, W)
PC idle (with FPGA card plugged in)	76 - 79 (80 - 86)
PC rendering classroom (with FPGA card plugged in)	160 (181)
PC rendering car2 (with FPGA card plugged in)	162 - 164 (174 - 176)
PC rendering bathroom (with FPGA card plugged in)	169 (180)
PC rendering coffee (with FPGA card plugged in)	172 (175)
PC rendering head (with FPGA card plugged in)	176 (182)
PC stress 1T	112-117
PC rendering with FPGRay (any scene)	110 - 120
FPGA card with default design (and fan on) FPGA card with 3IL (and fan on)	7.4 (9.6) 9.5 (11.8)

Note that the measurements named “FPGA card” used the card’s dedicated power supply plugged into the power meter, i.e., without measuring the power consumption of the PC. Only the 3IL design was used for this measurement to maximize the resource usage on the FPGA and is, as 3I, using three traversal instances. The only difference to 3I is the use of 128 cache entries for the primitive fetcher.

All measurements show that the FPGA card consumes between 7 and 12 W, depending on the loaded design. For the same design, the consumption exhibited no difference outside the tolerance. This behavior is typical for FPGAs, as they only have basic power saving features which are not used in FPGRay. The measured power consumption is therefore, apart from a negligible difference (due to the power needed for switching between ‘1’ and ‘0’ or keeping the conduit’s power level), the same for idle and full load.

PCs, in particular modern CPUs, possess advanced power saving features which resulted in a difference of around 100% between idle and rendering with full CPU load. For the

[5] TS Electronic EMG-1

used CPU, the highest allowed consumption is defined by the Package Power Tracking (PPT) at 88 W [ryz20].

When comparing the overall consumption delta between FPGRay and pbrt-fpgray not using the FPGA, the software rendering has a delta of around 100 W (which is more than the CPU is allowed to consume [ryz20]). In contrast, FPGRay exhibits a delta of roughly 40 W when compared to not having the card plugged in at all. As seen in the card-only measurements, most of this consumption delta is coming from the PC. In detail, it is the single CPU thread being utilized nearly 100% by the synthetic test program. To verify this, a comparable consumption could be seen by just using the Linux program "stress" for applying a high load to the CPU [str20b] at 1 thread.

Additionally, by modifying the synthetic test program to show the consumption of FPGRay alone without the consumption of reading and verifying the test dump, it was shown that the consumption decreased to 106-110 W but the throughput did not increase. This leads to the conclusion that the single-threaded API use is not a bottleneck. Furthermore, a different data transfer to the FPGA (e.g., by interrupts) may be able to reduce the consumption.

5.3.6 CPUs vs. GPUs

Besides using only CPUs for rendering, GPUs are supported by multiple renderers nowadays. Unfortunately, FPGRay, pbrt-fpgray, and pbrt-v3 do not support GPUs as rendering target, which makes a direct comparison with the same program impossible. Furthermore, the related LuxCoreRender [lux20] would support GPUs, but does not support k-d trees or pbrt scenes any more.

This means that only a rough approximation for the throughput comparison of a GPU compared to a CPU based on different scenes and renderers can be made. For more variety, Blenders' Cycles [ble20b] path tracer and through the offical Blender add-on, LuxCoreRender were used. Both were fed by the same Blender scenes. All tests were done with the default rendering settings for the path tracing algorithm and tiled rendering was activated. For LuxCoreRender, CPU tests were performed with the C++ renderer and GPU tests with the OpenCL renderer with the GPU being the only activated device. The tests were performed on the PC that was used for the synthetic tests consisting of a Ryzen 3700X, 32 GB DDR4-3200 RAM and an AMD Radeon RX480[6]. The consumption measurements were done in the same way as in the previous section. As the FPGA card was plugged in, the same idle consumption of around 80 - 86 W was measured.

Four test scenes were used: Two pbrt-v3 test scenes [pbr20], bathroom and coffee, for which the Blender scenes are available and two from the official Blender homepage [ble20a] in order to have scenes similar to the car2 and classroom scenes. For the classroom scene, the equally named "Class room" scene was used to show a closed room with various objects in it (Figure 5.8). For the car2 scene, the "Car Demo" (Figure 5.9) was modified

[6]Having 8 GB GDDR5 RAM and using as boost clock for the GPU 1290MHz.

Figure 5.8: The test scene “Class room”. The left image depicts the intended outcome as rendered by Cycles, the right how it is processed by LuxCoreRender.

Figure 5.9: The test scene “Car Demo”. The left image depicts the intended outcome as rendered by Cycles, the right image shows how it is rendered by LuxCoreRender.

Figure 5.10: The three different renderings of the bathroom scene. The left image is the result of pbrt-v3, the middle image from Cycles, and the right image was rendered by LuxCoreRender.

by removing the second car and the floor/wall combination. The “Car Demo” scene provided a scene file for CPU and one for GPU rendering, which was used with the corresponding renderers.

With Cycles, each test scene was completely rendered as specified by the default settings. The time was taken together with the measured power consumption. The preparation times (e.g., building of the BVH or compiling rendering kernels) were subtracted for the reported runtimes. With LuxCoreRender, the r/s were taken from the metrics samples/s

Figure 5.11: The three different renderings of the coffee scene. The left image is the result of pbrt-v3, the middle image from Cycles, and the right image was rendered by LuxCoreRender.

and rays/sample after the rendering held these values for multiple seconds, which was done to avoid registering misleading values during the rendering startup. In contrast to Cycles, LuxCoreRender renders the scene until it is fully converged. As not all material models of the test scenes are supported by LuxCoreRender, the materials were converted by the "Use Cycles settings" function in Blender. The resulting renderings therefore differ from those of Cycles, see Figure 5.8 and Figure 5.9 as an example. Furthermore, as different algorithms (e.g., BVH vs. k-d tree for intersection acceleration), material models, and different camera perspectives were used, the bathroom and coffee scenes are not directly comparable to pbrt-v3 too, as seen in Figure 5.10 and Figure 5.11. Those differences make the comparability between Cycles and LuxCoreRender (or pbrt) impossible but for each renderer, the performance differences between CPU and GPU rendering is a valid indicator for the general performance increase by using GPUs. Note that differently to the other tests, the complete rendering was measured and not the intersection time. Please note that the GPU engines perform the complete rendering on the device and not only the intersection operation, so they are not hybrid approaches.

Cycles Renderer								
scene	Car Demo		Class room		bathroom		coffee	
	t [s]	P [W]	t [s]	P [W]	t [s]	P [W]	t [s]	P [W]
CPU	52.15	186	481.88	194	4085.8	196	384.52	188
GPU	93.35	190	1156.98	190	7918.03	191	503.04	191

LuxCoreRender								
scene	Car Demo		Class room		bathroom		coffee	
	Mr/s	P [W]	Mr/s	P [W]	Mr/s	P [W]	Mr/s	P [W]
CPU	7.82	190	6	197	8.8	199	10.92	192
GPU	32.30	180	16.25	179	18	184	27.54	184

As the results show, the GPU is surprisingly slower than the CPU path tracer when using Cycles. Moreover, the same consumption could be measured and the resulting images, although not being the same, are for both versions noisy in a similar way. The CPU is therefore 70 - 140% faster.

For LuxCoreRender, the results showed the expected throughput increase when using the GPU for rendering. The scenes achieved a 2 - 4 times higher throughput.

As final note, it should be mentioned that the used renderers are built with the OpenCL [ope20] API. There is also the proprietary CUDA [cud20] environment for implementing such general purpose computations on GPUs. As CUDA is a common standard in comparison to OpenCL, it may be possible that optimizations have been done already for renderers using this platform to achieve an even higher throughput. But as CUDA is NVIDIA exclusive, it could not be used on the benchmarking PC that had an AMD card installed.

5.4 Efficiency of FPGRay

The throughput of FPGRay is inferior to the CPU implementation. But when it comes to the efficiency, FPGRay can be more efficient, depending on the design and scene. Using the deltas respective the observed consumption for the FPGA card and compare them with the achievable throughputs of the synthetic tests (Section 5.2) to estimate the total efficiency[7], the FPGA card alone shows a competitive efficiency to the PC. For the car2 scene, FPGRay even wins the duel clearly with 898 krays/W against 211 krays/W. A much lower efficiency was observed for the coffee scene with 113 kr/W vs. 111 kr/W for pbrt-v3. The bathroom scene achieved a comparable efficiency with a small advantage for pbrt-v3 with 91 kr/W vs. 102 kr/W. For the head scene, the PC took the lead with 149 kr/W against the 109 kr/W of the FPGA. The classroom scene for which FPGRay performed worst is also the scene with the biggest lead for pbrt-v3: 114 kr/W stand against the 65 kr/W of FPGRay. It should be noted that by increasing the cache sizes, a significant throughput and thus efficiency increase would be possible by realizing larger caches. Furthermore, for the head scene it would be sufficient to improve the PCIe interface, as the achieved throughput on the FPGA was by far higher than in software. To summarize, while FPGRay in combination with the used FPGA is not inefficient, it is too slow for a proper use with a modern PC.

The deltas of the power consumption measurements of the complete PC for the CPU implementation with pbrt-v3 against the synthetic tests of FPGRay lead to similar results with a more obvious advantage for the software-only approach. FPGRay still performs well for the car2 scene with 269 krays/W against 211 krays/W. But for all other scenes, the PC takes over the lead with a significant difference.

Figure 5.12 lists all efficiency computations collected for CPU, GPU, and FPGA rendering. The CPU-based renderers pbrt-v3 and LuxCoreRender achieved different results for the similar scenes. While the GPU version of LuxCoreRender was able to achieve a higher efficiency for all scenes, FPGRay only performs better than the CPU-based renderers for the car2 scene, which illustrates that FPGRay does not scale well for different scenes,

[7] 40 W for FPGRay, 88 W for the PC and 12 W for the FPGA card alone was used. The throughputs used were the highest achievable results observed during the synthetic tests.

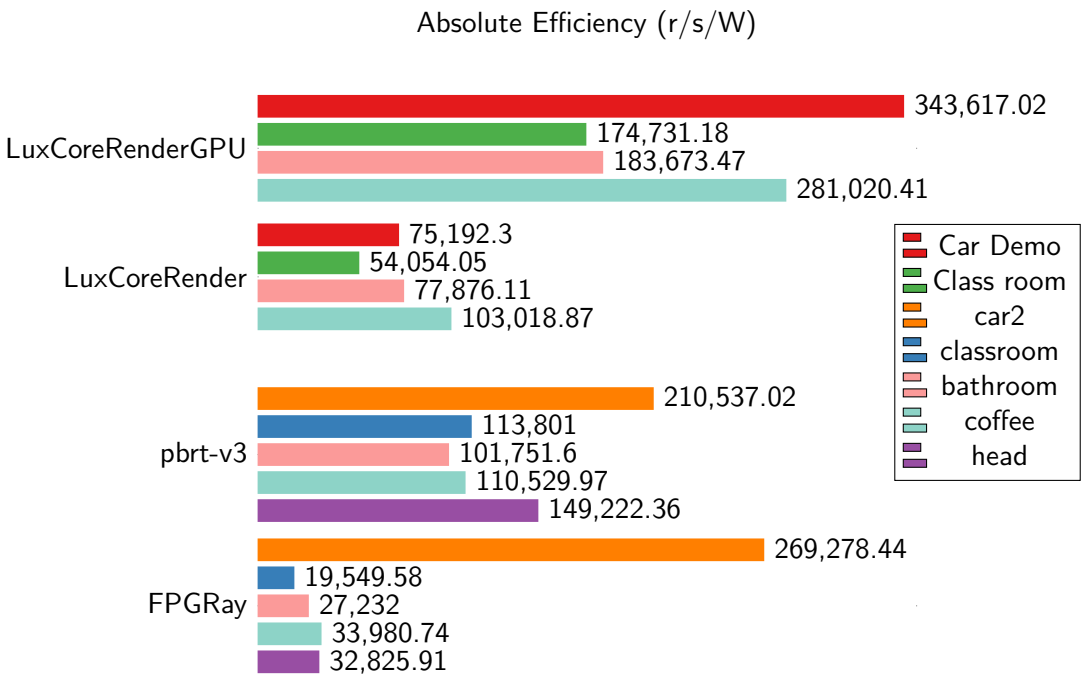


Figure 5.12: The absolute efficency of the renderers supporting special hardware.

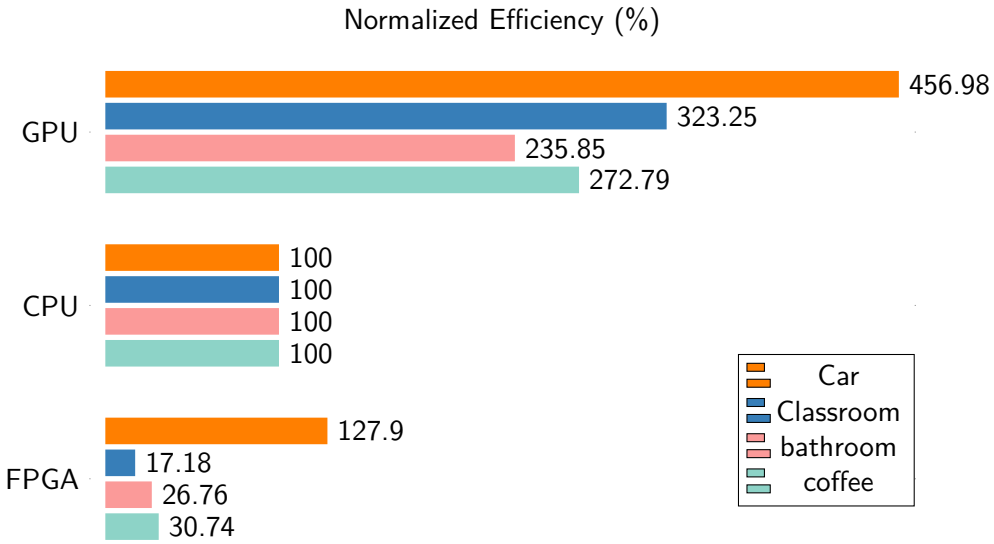


Figure 5.13: When normalizing the CPU benchmarks of the similar scenes of both renderers, an approximate comparison between FPGRay and a GPU implementation can be made.

which is possible with the GPU implementation. The results lead to the conclusion that the GPU renderers are superior to the CPU renderers, which is not the case for FPGRay.

Lastly, for Figure 5.13, we have aggregated the results of both CPU renderers and normalized the result to 100% in order to provide an approximate comparison between the FPGA and the GPU approach. This was done by mapping both CPU implementations (i.e., LuxCoreRender and pbrt-v3 of Figure 5.12) and their similar scenes (e.g., LuxCoreRender's Car Demo with pbrt-v3's car2 scene) together to the normalized 100%. Therefore, the GPU (i.e., LuxCoreRenderGPU) and FPGA (i.e., synthetic test results of FPGRay) implementations are depicted in relation to its CPU-based counterpart. Unsurprisingly, in the current implementation, FPGRay achieved a higher efficiency in comparison to the CPU for the car2 scene and a lower efficiency for the other scenes. Furthermore, the GPU renderer is superior in terms of efficiency.

5.5 Analyzing the Results

The previous sections not only provided comparisons but also showed different aspects of FPGRay that could not be investigated by simple tests on demo scenes to expose unseen potential. The main benefit of FPGRay should be the efficiency gain compared to a CPU-only solution. Unfortunately, the efficiency measures led to an underwhelming outcome. The most important question emerging from the presented tests is how the results should be interpreted.

FPGRay is in its current state not usable for the aimed scenario. But the use of specialized hardware to increase the efficiency and even accelerate the rendering is still a viable approach. This can be exemplified with the first GPUs containing such hardware instances for real-time graphics. As the manufacturers of these GPUs are substantial companies with enough resources to develop advanced hardware designs, it is enough for most of the target audience to use them and build their renderers on it instead of developing a new and costly design.

However, if new or more advanced techniques should be elaborated which are not likely to be implemented in new GPUs, at least for some time, FPGRay can be used by extending its functionality. In other words, if hardware designs should be used in the field of research, FPGRay is a viable option. For production rendering, visualizations, or development of new products, GPUs provide the highest efficiency with low implementation effort.

In any case, it should be noted that FPGRay has to be improved to allow a proper usage. The special tests section (Section 5.3) showed already some bottlenecks of the presented design. In the following, we address some of them and explore the impact of possible solutions.

Cache Sizes. Quadrupling the cache sizes increased the hit rate to around 70% and achieved a 20 - 40% higher throughput (even for the head scene with an already high hit rate, which increased its throughput by 39%). Taking the (even for such cache sizes

low) hit rate of around 40 - 50% into account, an improvement of the hit rate of at least a factor 2 is possible. This means, by additionally quadrupling or better 8-folding the cache sizes relative to the biggest size used, the throughput could increase again by 40%, increasing the hit rate to around 90%. The assumable gain makes the increase of cache sizes desirable, even if it would result in a higher latency for cache accessess. This throughput gain would not increase the power consumption significantly as long as the design fits on the current or a bigger FPGA of the same series. This would result in an efficiency gain equal to the throughput increase of around 40%.

Improving the Interface. As the FPGA runtimes (Section 5.3.2) exposed, the PCIe interface can limit the throughput already for small scenes. When improving the throughput on the FPGA, the interface could prove to be a serious bottleneck. Furthermore, the measured power consumption of FPGRay's distinct units revealed that most of the power was consumed by the PC: of the measured 40 W, only 12 W were consumed by the FPGA. So an improved interface would not only avoid a bottleneck at the PCIe interface, it would also reduce the needed power. Although the power consumption from the PC might be decreased further, we assume that the delta can be halved, which would in turn double the efficiency. As the PC only needs to call the API, initiating the driver transferring the rays to and from the FPGA, such a power reduction is possible. It should be noted that we did not take the fact that the ray data needs to be transferred to appropriate memory structures for use with FPGRay into account. This assumes that a renderer is built using such structures anyway to avoid a translation between different structures for the PC and FPGRay.

An improved interface would also allow to use fewer resources on the FPGA, which in turn could be used for other improvements mentioned in this section. As a side effect, more than 1024 r/b could be used stably.

Number of Computational Instances. By using two instead of one traversal instance, the throughput nearly doubled. For 4 instances at 1024 r/b, an increase of 2.6, using one instance as a baseline, was observed. As the synthetic tests (Section 5.2) and FPGA runtimes indicated, the bigger designs are able to increase the throughput for higher batch sizes more than for smaller ones. It is therefore likely that using more computational instances and a higher batch size would increase the throughput linearly as seen for the smaller designs. Extrapolating from the used three traversal instances to 8, this would even, with sublinear scaling, allow to double the throughput. This would not fit on the current FPGA, so the efficiency would not double when using a bigger, more power-consuming FPGA. Nevertheless, when using 8 traversal and a second intersection instance or even using more than 8 traversal instances to fill up a bigger FPGA, a doubling of the efficiency could be posssible. Note that as stated before, to be able to retrieve this gain through higher batch sizes, an improved interface would be necessary.

Changing the FPGA. As already mentioned, a bigger FPGA could be used to increase the throughput. But a different FPGA can also deliver a higher efficiency by just using an improved design or new special function blocks itself. Furthermore, the efficiency can be improved by using an FPGA with a newer manufacturing process. Chip manufacturing companies state that by using a newer manufacturing node, the efficiency increases by around 30%. The AMD Radeon Vega 56, Vega 64 and Vega VII prove this by technically using the same architecture for all designs which was produced in Samsung/Globalfoundry's 14 nm process node for the Vega 56/64 and TSMC's 7 nm process node for the Vega VII. The Vega VII delivers either 30% higher performance for the same consumption or delivers a 30% higher efficiency through higher performance while consuming less [veg20b, veg20c, veg20a]. Note that the used CPU for the thesis' tests also uses the TSMC 7 nm process, while the FPGA uses the TSMC 28 nm process [arr20a]. This means that those two chips are apart by at least 4 manufacturing nodes, which results in at least a 2.5 times higher efficiency for the CPU. Furthermore, the currently used Arria V FPGA can be changed to an Arria 10 using the 20 nm node, which delivers 20% more efficiency without changing the design at all [arr20c].

Using a bigger FPGA can improve the throughput without much higher power consumption. As the device table [arr20b] shows, the currently used Arria FPGA could be changed to a model with 40% more of the limiting basic logic resources. This model could accommodate the 4I design with ease and should even allow up to 6 traversal instances. The mentioned Arria 10 series allows, besides the newer processing node, to use around 3 times more logic resources. Furthermore, the biggest and most recent FPGA series from Intel, the Stratix [str20a] and Agilex [agi20] series provide 20 respectively 4 times more resources and use the 14 respectively 10 nm process node. This gives a big potential for accommodating bigger designs if the high prices can be afforded.

FPGAs vs. ASICs. As mentioned in the background chapter, an ASIC is the best option for hardware designs. Unfortunately, they are prohibitively expensive to develop and prepare for production. But it should be stated that a significant efficiency gain could be achieved by using an ASIC for the design. As Ian Kuon and Jonathan Rose [Ian06] found out, an ASIC has a 9 - 12 times lower power consumption than a comparable FPGA of the same process node. Furthermore, the FPGA is still slower by a factor of 2 - 4. This enables the ASIC to get a at least 18 times higher efficiency.

When accumulating the extrapolated efficiency increases that are possible from the existing design, it can be assumed that the efficiency can be 2.8 times higher when just improving the caches and interface. If the saved space from the improvements is used for a fourth traversal instance, those three improvements would allow a gain of the factor 3.6. This would suffice already to be on par in terms of efficiency with the CPU renderer for all except the classroom scene without changing the FPGA.

Furthermore, if a newer FPGA of the same Arria series is used, the design could reach an even higher efficiency for every tested scene. Moreover, when using the available

resources of a bigger FPGA to also improve the throughput, it would be fast enough to allow its intended use besides the CPU while not being the bottleneck. This is based on the assumption that 8 instead of 4 traversal instances can be accommodated and achieve a doubled throughput, which delivers for the worst scene classroom an efficiency at least 20% higher than the CPU renderer. Note that for the newer and bigger Arria, the same power consumption is assumed, as more hardware resources increase the power consumption subproportionally to the throughput increase and a more advanced manufacturing node is used to compensate a higher consumption. Furthermore, for achieving or even exceed these predicted increases, the RAM respectively its interface to the FPGA needs to be scaled in terms of throughput appropriately.

An even bigger FPGA of the high-end series or an ASIC is therefore not even necessary to outperform current CPUs when extrapolating the results. Nevertheless, to achieve a 4 times higher efficiency than the CPU to even outperform GPUs, such an FPGA or even an ASIC could be needed.

CHAPTER 6

Future Work

After having discussed the impact of possible improvements in the previous section, ideas of how to implement them and various future work are presented below.

6.1 Improvements

6.1.1 PCIe Interface

As seen in the FPGA runtimes benchmark (Section 5.3.2), the transfer via the PCIe interface needs to be improved to avoid a bottleneck. Furthermore, the efficiency is subpar with the current approach. Technically, the interface is implemented as DMA, which is defined as a dedicated controller that has direct access to the PC's RAM. But the DMA is used in a simple in-order mode, copying data consecutively. This is the standard behavior of the PCIe HardIP adopted from Altera's reference design for FPGRay. For the feeding of data to kdintersect, this is no problem, but for the resulting rays which are returned out-of-order, a random-access scheme would be preferred. Using a random scheme would make the output buffer inside of FPGRayIF unnecessary, which would save resources. Furthermore, as rays are sent to the PC's RAM instead of waiting for a specific order, the throughput is maximized.

It should be noted that PCIe supports the use of interrupts, which can be used to implement callbacks for asynchronuous function calls and thus avoid the currently used polling behavior. Nevertheless, interrupts were not considered for this thesis and it is unclear if an improvement is possible by using them, so this would be another aspect which could be investigated further.

6.1.2 Caching and Memory Interface

The tests showed that even a cache size of 128 entries per fetcher is not sufficient for a good hit rate of the caches. But the cache was optimized for latency, which only allows

the use of 128 entries to meet the timing requirements. By changing the cache to support a higher number of entries by sacrifying latency, the hit rate could be improved and with it, the overall throughput of FPGRay.

The memory interface is built to support two ports to the RAM. But this limits the throughput to 2 accesses per cycle. If a faster memory controller is used, the kdmemmerger (Section 4.3.4) would be the bottleneck. To avoid this, the merger needs to be extended to split memory accesses to multiple ports.

6.2 Extensions

6.2.1 Optimized Algorithms

The entities FPGRay uses are taken from pbrt-v3. Compared to the papers discussed in the related works (Chapter 3), the number of operations a triangle intersection needs is much higher. By using optimized algorithms, the latency of a computational instance decreases, which also results in a lower number of stacks needed. Finally, the design either needs fewer resources or a bigger design can be used on the same FPGA, which increases the throughput.

6.2.2 New Operations

The intersection operation is an essential and heavily used operation for every ray tracing algorithm. But FPGRayIF could instance additional functions to support more parts of the rendering process. This may even be extended to use multiple operations consecutively before returning the data back to software.

6.2.3 pbrt-fpgray Part 2

Currently, a pbrt version extending pbrt-v3 with the use of FPGRay is provided. But this extension only interchanges the computation of a k-d tree intersection with a method that makes use of FPGRay. The function therefore intersects each ray with its own batch, which is inefficient. To make use of the parallelism of the FPGA, pbrt (or any other renderer) needs to be rewritten to render with batches. The optimal solution should send multiple samples and use tiles to optimally make use of the locality of the rays. When performing path tracing, the paths should be split into distinct rays such that for each bounce, the locality of the rays is maximized. The renderer should support collecting a high number of rays for a single batch—at least 128—to reach the limit of the FPGA throughput.

Only the usage of a real renderer instead of the synthetic test program makes a proper comparison of the hybrid approach vs. the software approach possible. The current tests do not enlight possible bottlenecks when rendering and communicating with the same data, i.e., if the synchronization between the rendering threads and the communication thread to the FPGA is working without long waiting times.

6.2.4 Ray Storage

The rays' data should be stored in structs rather than classes in ray tracers. As the batches are arrays of structs, this avoids the conversion and decreases the overhead of the data transfer between the PC and FPGRay. Depending on the PC's platform, it should be considered to use the low-level allocation function `dma_alloc_coherent()` of Linux' DMA-API. Additionally to using structs, this may omit the copy operation between the renderer's allocated memory and the memory space used for DMA, but is not supported on all platforms.

CHAPTER 7

Conclusion

FPGRay is an approach to accelerate path tracing with the use of an FPGA. This is done by providing a PCIe card plugged into a PC and appropriate drivers to use this hardware for acceleration. With such a hybrid approach between hardware and software, the advantages of both can be combined. The customized hardware design enables an increase in the overall efficiency but still saves hardware resources because not all computations needed for ray tracing need to be built in silicon. Such computations are done in software to utilize the resources of a modern PC. The approach thus results in the possibility to concentrate on heavily used tasks of the rendering algorithm, which should be done by the hardware design to increase the efficiency. As ideal candidate for such an operation, the intersection of a ray against a k-d tree scene was chosen to be implemented in hardware.

The flexibility the software part of FPGRay delivers allows adding new techniques to the rendering process without changing the hardware design. The efficiency improvement is still present as long as the hardware-accelerated k-d tree intersection routine can be used. This sustainability of FPGRay is unique compared to all previous work, which required a new hardware design in case of just one changed aspect inside of the ray-tracing algorithm.

To make use of FPGRay from software, a driver for Linux-based PCs and a C++ API were implemented. Furthermore, a short demo implementation of pbrt's k-d intersection method for FPGRay showed the functionality of the system. Because of the naive implementation, which just replaces the original method and does not consider the needed batch sizes for efficient hardware acceleration, further work to properly extend a renderer with FPGRay's functionality is necessary.

For allowing realistic benchmarks in the needed—batched—data flow, a synthetic test program was implemented that makes use of scene and ray data generated through the rendering of a scene with pbrt-v3. As FPGRay's hardware design is flexible and

configurable (e.g., in the number of traversal instances), the synthetic test program used the generated test dumps to evaluate the throughput of different designs synthesized on the FPGA. The results showed—although working—bad throughput in comparison to the host PC of FPGRay rendering the scenes in software alone. Depending on the tested scene, the throughput was from a factor of 2 to around a magnitude worse than using the software renderer. As a direct comparison was not feasible, renderers with CPU as well as GPU renderers were fed with similar scenes to provide an approximate comparison between GPU rendering and FPGRay. Unsurprisingly, as GPU rendering achieves a higher throughput compared to its CPU counterparts, FPGRay is also having a lower throughput in comparison.

After testing different parts of FPGRay to find the reason for its bad performance, some key problems in the hardware design were found. The memory subsystem adds a high throughput penalty to FPGRay because the caches are too small to buffer the RAM access of the PCIe card properly. This stresses the RAM interface in a way that it easily gets over-utilized because it is working at the limit. Another problem is the interface between the PCIe card and the PC, which is not only a problematic part of the hardware design operating at the limit but rather uses a non-optimal data transfer. Moreover, the currently used FPGA was too small to fit a design with enough computational units to process the PC's data fast enough. These main drawbacks of the current implementation do not allow FPGRay to unfold its full potential.

The final measurements of the power consumption nevertheless showed a beneficial efficiency of FPGRay. However, the absolute throughput is too low for current PCs. Therefore, it acts as bottleneck of the overall system and makes a use in the intended hybrid form not viable. Unfortunately, FPGRay could not surpass the efficiency of GPU renderers. In its current design, the efficiency is therefore not improved over exisiting solutions.

Lastly, the tests indicated that FPGRay has much untapped potential to improve the efficiency and with it, the throughput, which in turn may allow surpassing existing approaches. Unfortunately, the indicated possibilities for improvements could not be further explored in the scope of this thesis.

In the scope of the thesis, FPGRay was not able to reach its goal. Nevertheless, on its basis, multiple improvements were presented that show how and for which expected impact the goals can be reached. However, as dedicated ray-tracing hardware was introduced to GPUs recently, the efforts to implement those improvements are only feasible for a few specific purposes, in particular, all topics that focus on the investigation or implementation of any new or optimized hardware-accelerated functionality in the field of ray tracing.

List of Figures

List of Tables

List of Algorithms

Acronyms

API Application Programming Interface. 3, 24–26, 51–54, 70, 78, 79, 87, 93, 96, 99, 107

ASIC Application-specific Integrated Circuit. 13, 19, 22, 100, 101

BDPT bidirectional path tracing. 8, 77

BSDF bidirectional scattering distribution function. 6, 8

BVH Bounding Volume Hierarchy. 9, 21, 22, 24, 28, 94, 95

CGI computer-generated imagenery. 1, 2

CPU Central Processing Unit. 2, 12–14, 18, 21–23, 85, 92–98, 100, 101, 108, 112

DMA Direct Memory Access. 26, 49–51, 55, 58, 59, 69, 70, 73, 87, 103, 105, 111

EDA Electronic Design Automation. 14, 15, 17–19, 55, 70

FIFO First in - First out (buffer). 14, 27, 63

FPGA Field Programmable Gate Array. 3, 13–17, 19, 21, 22, 24, 25, 29, 32, 41, 43, 48–53, 55–58, 62, 63, 65, 68–70, 72, 75, 77, 80, 84, 85, 88–90, 92, 93, 96, 98–101, 103, 104, 107, 108, 111

GPU Graphics Processing Unit. 3, 13, 18, 75, 85, 93–98, 101, 108, 112

IDE Integrated Development Environment. 14

IP Intellectual Property. 14, 16, 19, 26, 55–57, 59, 62, 64, 70, 75, 76, 103, 111

LED Light-emitting Diode. 56

LSB Least Significant Bit. 74

MLT Metropolis light transport. 8

PC Personal Computer. 3, 24, 26, 55, 56, 59, 68, 69, 75–77, 85, 92, 93, 96, 99, 103, 105, 107, 108

PCIe Peripheral Component Interconnect Express. 3, 14, 24–26, 49, 50, 52, 53, 55–60, 62, 68, 69, 73, 75, 77, 80, 89, 96, 99, 103, 107, 108, 111

PPT Package Power Tracking. 93

RAM Random Access Memory. 14, 25, 26, 32, 34, 35, 37, 38, 40, 41, 44–46, 49–53, 55, 56, 59–65, 72, 75, 76, 84–89, 91, 93, 101, 103, 104, 108, 110

SAH Surface Area Heuristic. 10, 11, 27, 109

SISD Single Instruction, Single Data. 21

spp samples per pixel. 8, 77, 79, 80

USB Universal Serial Bus. 21

VHDL Very High Speed Integrated Circuit Hardware Description Language. 13–17, 19, 25, 55, 56, 70, 89

VR virtual reality. 1

Bibliography

[agi20] intel® Agilex™ I-Series SoC FPGA Product Table, accessed on 08.06.2020. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/intel-agilex-i-series-product-table.pdf.

[ALK] Linux Loadable Kernel Module. https://wiki.archlinux.org/index.php/Kernel_module.

[arr20a] Arria V Device Overview, accessed on 08.06.2020. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-v/av_51001.pdf.

[arr20b] Arria V FPGA and SoC Features, accessed on 08.06.2020. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/arria-v-product-table.pdf.

[arr20c] Intel® Arria® 10 FPGAs, accessed on 08.06.2020. https://www.intel.com/content/www/us/en/products/programmable/fpga/arria-10.html.

[Bit16] Benedikt Bitterli. Rendering resources, 2016. https://benedikt-bitterli.me/resources/.

[ble20a] Blender: Cycles render demo scenes, accessed on 01.06.2020. https://www.blender.org/download/demo-files/#cycles.

[ble20b] Blender: Open source 3D creation, accessed on 01.06.2020. https://www.blender.org.

[CKl+10] Byn Choi, Rakesh Komuravelli, Victor lu, Hyojin Sung, Robert Jr, Sarita Adve, and John Hart. Parallel sah k-d tree construction. pages 77–86, 06 2010.

[cud20] NVIDIA's CUDA parallel computing platform, accessed on 01.06.2020. https://developer.nvidia.com/cuda-zone.

[FW80] J. D. Foley and Turner Whitted. An Improved Illumination Model for Shaded Display, 1980.

[Ian06] Ian Kuon and Jonathan Rose. Measuring the Gap between FPGAs and ASICs. In *in ACM International Symposium on Field Programmable Gate Arrays*, 2006.

[Jak10] Wenzel Jakob. Mitsuba renderer, 2010. http://www.mitsuba-renderer.org.

[Kaj86] James T. Kajiya. The Rendering Equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM. http://doi.acm.org/10.1145/15922.15902.

[LKM] Linux Loadable Kernel Module introduction. http://tldp.org/HOWTO/Module-HOWTO/x73.html.

[LLN+12] Won-Jong Lee, Shi-Hwa Lee, Jae-Ho Nah, Jin-Woo Kim, Youngsam Shin, Jaedon Lee, and Seok-Yoon Jung. SGRT: A Scalable Mobile GPU Architecture Based on Ray Tracing. In *ACM SIGGRAPH 2012 Posters*, SIGGRAPH '12, pages 44:1–44:1, New York, NY, USA, 2012. ACM. http://doi.acm.org/10.1145/2342896.2342953.

[lux20] LuxCoreRenderer: Open Source Physically Based Renderer, accessed on 22.03.2020. https://luxcorerender.org.

[NKK+14] Jae-Ho Nah, Hyuck-Joo Kwon, Dong-Seok Kim, Cheol-Ho Jeong, Jinhong Park, Tack-Don Han, Dinesh Manocha, and Woo-Chan Park. RayCore: A Ray-Tracing Hardware Architecture for Mobile Devices. *ACM Trans. Graph.*, 33(5):162:1–162:15, September 2014. http://doi.acm.org/10.1145/2629634.

[NPP+11] Jae-Ho Nah, Jeong-Soo Park, Chanmin Park, Jin-Woo Kim, Yun-Hye Jung, Woo-Chan Park, and Tack-Don Han. T&I Engine: Traversal and Intersection Engine for Hardware Accelerated Ray Tracing. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, SA '11, pages 160:1–160:10, New York, NY, USA, 2011. ACM. http://doi.acm.org/10.1145/2024156.2024194.

[ope20] OpenCL: OPEN STANDARD FOR PARALLEL PROGRAMMING OF HETEROGENEOUS SYSTEMS, accessed on 01.06.2020. https://www.khronos.org/opencl/.

[pbr20] Scenes for pbrt-v3, accessed on 05.06.2020. https://https://pbrt.org/scenes-v3.html.

[PJH16] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2016.

[qua20] Intel Quartus Prime Software Suite, accessed on 20.02.2020. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html.

[ryz20] AMD Ryzen 3700X Power Consumption and Limits, accessed on 02.05.2020. https://www.anandtech.com/show/14605/the-and-ryzen-3700x-3900x-review-raising-the-bar/19.

[str20a] intel® Stratix® 10 GX/SX Product Table, accessed on 08.06.2020. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/stratix-10-product-table.pdf.

[str20b] stress - tool to impose load on and stress test systems, accessed on 31.05.2020. https://linux.die.net/man/1/stress.

[SWS02] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR: A Hardware Architecture for Ray Tracing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, pages 27–36, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association. http://dl.acm.org/citation.cfm?id=569046.569051.

[TR01] Andrew S. Tanenbaum and Robbert Van Renesse. Modern Operating Systems, Second Edition, 2001.

[veg20a] ComputerBase: AMD Radeon Vega VII vs. Vega 64, accessed on 08.06.2020. https://www.computerbase.de/2019-02/amd-radeon-vii-test/5/#abschnitt_radeon_vii_vs_rx_vega_64_bei_gleicher_leistungsaufnahme.

[veg20b] Radeon Vega VII Test, accessed on 08.06.2020. https://www.igorslab.de/heisses-eisen-im-test-amd-radeon-vii-mit-viel-anlauf-und-wind-auf-augenhoehe-zur-geforce-rtx-2080/16/.

[veg20c] The AMD Radeon VII Review, accessed on 08.06.2020. https://www.anandtech.com/show/13923/the-amd-radeon-vii-review/19.

[WSS05] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 434–444, New York, NY, USA, 2005. ACM. http://doi.acm.org/10.1145/1186822.1073211.

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. Technical Report MSR-TR-2008-52, April 2008.