Eurographics Symposium on Parallel Graphics and Visualization (2019)
H. Childs, S. Frey (Editors)

Hybrid Online Autotuning for Parallel Ray Tracing

K. Herveau[1], P. Pfaffe[2], M. Tillmann[2], W. F. Tichy[2], and C. Dachsbacher[1]

[1]Computer Graphics Group
[2]Institute for Program Structures and Data Organization
Karlsruhe Institute of Technology, Germany

Abstract

Acceleration structures are key to high performance parallel ray tracing. Maximizing performance requires configuring the degrees of freedom (e.g., construction parameters) these data structures expose. Whether a parameter setting is optimal depends on the input (e.g., the scene and view parameters) and hardware. Manual selection is tedious, error prone, and is not portable. To automate the parameter selection task we use a hybrid of model-based prediction and online autotuning. The combination benefits from the best of both worlds: one-shot configuration selection when inputs are known or similar, effective exploration of the configuration space otherwise. Online tuning additionally serves to train the model on real inputs without requiring a-priori training samples.

Online autotuning outperforms best-practice configurations recommended by the literature, by up to 11% median. The model predictions achieve 95% of the online autotuning performance while reducing 90% of the autotuner overhead. Hybrid online autotuning thus enables always-on tuning of parallel ray tracing.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types

1. Introduction

Performance and image quality are the linchpin metrics of ray tracing. In its widespread application in movie production, architecture, or video games, its users are often forced to prioritize one over the other. The metrics are contradicting. Advances in both hardware and software however seek to accelerate the rendering performance while retaining quality as much as possible. Parallel processors and specialized hardware features such as NVIDIA's recent RTX technology [HA19] open new and exciting ways to accelerate ray tracing.

On the software side, heavily optimized ray tracing such as OSPray [WJA*17] benefits from spatial data structures, such as Bounding Volume Hierarchies (BVH) [KK86] which are the fundamental accelerators of ray tracing.

State-of-the-art acceleration structures offer a multitude of tunable parameters such as the leaf size or the depth of BVH trees. Finding the optimal configuration for these parameters is essential to achieve best performance. To date, this is a manual task left to the engineers who build the ray tracing application. Most of the time they simply use configurations recommended by experts. If the cost of the application is not dominated by ray tracing, using a standard configuration can well be an adequate choice. On the other hand when the application cost *is* heavily impacted by ray tracing, the standard configuration may turn out to be unsatisfactory [TPKT16]. This is not because the experts made poor guesses, but because the recommendation is *static*. The recommendations were made based on experiences or experiments on specific hardware, specific ray tracing applications, and specific inputs. The recommended configurations are optimal for those observed contexts. But they cannot be expected to generalize in the sense that they are optimal for different hardware, applications, or even inputs. These three aspects of the context all affect the quality of any particular configuration. A scene with small and large triangles next to each other would highly benefit from triangle splitting, but a scene with uniform triangle size will see its overall performance decrease because of the overhead of splitting. This example demonstrates that, to obtain best performance, configurations need to be selected *dynamically* and the context needs to be taken into account.

Autotuning is a method that deals with finding optimal parameter configurations. Its fundamental idea is to repeatedly sample the performance of configurations and employ an intelligent search scheme to traverse the configuration space. Doing this *online* (i.e., at application runtime) allows adapting to the context: Whenever the context changes tuning simply restarts. However there are several caveats. First and foremost, autotuning uses heuristic search algorithms. The configuration space is generally too large to be exhaustively explored. As a consequence, the configurations found by tuning are only locally optimal. Secondly, autotuning introduces overhead. Some autotuning methods have a notion of convergence, where the search algorithm eventually produces a single final point. Others simply search until a time or iteration budget is

DOI: 10.2312/pgv.20191110

exhausted. In either case the search will inevitably sample configurations whose performance is (substantially) worse than both the default and the global optimum. This opportunity cost accumulates during tuning and can introduce a significant overhead. Additionally, when used in a production environment application users may notice the decreased performance and changing frame rate.

Contributions. In this paper we integrate parallel ray tracing with a novel tuning technique we call Hybrid Online Autotuning. It combines classical search-based tuning with online learning. We use the search to *explore* the configuration space and to train a predictor simultaneously. The predictor maps inputs to configurations. We quantify the input through a set of metrics, which we refer to as *indicators*. Example indicators are the number of triangles, the total surface area, or the camera position. Upon changes in the input, hybrid tuning decides whether to search or to *exploit* the predictor. We show that classical search achieves a performance speedup of up to 11% median over the recommended default configuration. Moreover, the predictor trained during the search reduces the overhead introduced by the search by up to almost 90% while achieving up to 95% of the performance of the configuration found by searching.

We further focus on changing inputs in this paper because it is the most dynamic aspect of the changing context. However, our technique easily generalizes to include hardware- and application-specific indicators as well. Furthermore, although we implemented and evaluated our approach for parallel ray tracing only, we are confident that the same technique will be successful for sequential applications as well.

2. Related Work

Several works established the link between scene properties and acceleration structure quality. Such properties can be the distribution and overlap of triangles in a scene [SFD09] or the visibility [VHS12]. Dammertz et al. [DHK08] evaluated the impact of the number of triangles per leaf node of several BVH implementations and found substantial variations.

Recent work focuses on an incremental or iterative construction process. This process involves quickly building a low-quality data structure and then refining it over multiple iterations. Bittner et al. [BHH15] presented an incremental data structure where the number of modified nodes and the fraction of nodes to be updated are free parameters. Wodniok et al. [WG17] hand pick several values for their BVH's partitioning strategy. More recently, Meister et al. [MB18] expose a configurable parameter for the density of search nodes in their reinsertion strategy. These recent examples show that the work on acceleration structures is ongoing. Nevertheless, new approaches still expose performance-critical parameters for which a configuration must be selected.

In the following sections we first present approaches which aim at optimizing ray tracing and acceleration structures automatically for given inputs or hardware. Second, we report recent works in the wider field of autotuning which also approach context-sensitive tuning.

2.1. Optimizing Ray Tracing and Acceleration Structure Parameters

Although the importance of ray tracing and acceleration structure parameters is known, only few works aim at automating the search for configurations.

Ganestam et al. [GD12] employ a model-based online autotuner to optimize GPU ray tracing. They measure how the acceleration structure is queried by evaluating the number of cache hits and the number of hits per node of the BVH. To ensure a constant frame rate, image quality is adapted when the complexity of the scene increases. In their work, the quality is variable while our approach aims at keeping a constant quality and speeding up rendering.

Targeting the GPU, Weber et al. [WG14] employ autotuning to optimize the memory layout of the binning of kD-trees. They build a decision tree based on empirical measurements that halves construction time. The approach is orthogonal to ours. It could however be used in conjunction, as the authors only modify the memory layout by the acceleration structure and none of its exposed parameters.

A number of works established the benefits of machine learning for rendering. Vorba et al. [VKŠ*14] used Gaussian Mixture Models (GMM) to represent sampling distributions. They show improvements in convergence rate for path tracing, bidirectional path tracing, and Metropolis light transport. Similarly, Dahm et al. [DK17] improved the sampling scheme by using Q-learning, a model-free Reinforcement Learning (RL) technique. They perform training during rendering and learn to find light source paths. They successfully reduce the number of zero contribution paths and reduce the average path length. Our approach employs a concept related to GMMs, but uses it to optimize acceleration structure parameters without modifying the underlying image synthesis algorithm.

Search-based online tuning has successfully been used to optimize parallel acceleration structures for a ray tracing application by Tillmann et al. [TPKT16]. This work demonstrates that autotuning is effective in accelerating ray tracing performance beyond what expert-recommended parameter configurations offer. The authors report speedups of up to 1.96× over the standard configuration of an SAH kD-tree. However, they do not investigate techniques to reduce the autotuning overhead. Like in our approach, Tillmann et al. use the Nelder-Mead search algorithm, but use it to optimize a different acceleration structure.

2.2. Input-Sensitive Autotuning

Autotuning as a tool to optimize program parameters automatically has been around for two decades. It was made popular by the ATLAS library [WD98], which optimizes BLAS primitives during installations using training examples shipped with the software. Today, the most widely used general purpose autotuning tools are ActiveHarmony [ŢCH02] and OpenTuner [AKV*14]. Both use heuristic search to navigate the configuration search space and are oblivious to changing program inputs. ActiveHarmony uses the Nelder-Mead algorithm, whereas OpenTuner employs an ensemble of different search methods concurrently.

An idea similar to ours in design was developed by Bergstra et al. [BPC12]. They combine heuristic search and regression trees to optimize the parameters of a CUDA implementation of a numerical kernel. Their combination approach however operates in a different manner than the one we present: The regression tree model is built on training data and then serves as a surrogate for empirical search. Autotuning approaches that build and refine prediction models online are called "active learning" in the literature. The basic principle of active learning is to use the prediction model itself to recommend samples to refine the model [Set09]. A recent instance of this method was presented by Balaprakash et al. [BGW13]. They build a performance model using dynamic trees. As part of a compiler, the model is applied to loop optimization of numerical kernels and to predict the performance of MPI codes. The Nitro system [MSH*14] is an approach to select optimal code variants in an input-dependent manner. Programmer-defined features are computed from program inputs and are used to query an SVM model. The model is built during an offline-training phase from input data. Bao et al. [BHC*16] apply model-based tuning to minimize energy consumption by selecting optimal CPU frequency at program runtime. They use a compiler to determine relevant features for the program and create a set of profiling benchmarks to stress individual features. From the profiling benchmark their approach builds a decision tree.

These related works either build their models offline or online. In the former case, a-priori training data is required. In the latter case, the model is used to select training data. The performance of the configurations sampled by a still-training model are subpar. Our approach provides a target-oriented way to train the model and to optimize performance even during training.

3. Autotuning Parallel Raytracing

In this section we present the design and implementation of our parallel renderer, its acceleration structure, and its integration with the autotuner. We also discuss the various tuning parameters exposed by the acceleration structure and the indicators the renderer reports to the tuner.

3.1. Renderer

Ray tracing [Kaj86] and especially path tracing are now a standard of the movie industry [CFS*18] [BAC*18]. It is extensively used for interactive applications [NHD10] to help rasterization with global illumination [TO12]. Interactive ray tracing is also available on CPU thanks to OSPray [WJA*17] for instance, Intel's optimized ray-tracer. In real-time applications, it is common to use one sample per pixel (spp) and then process the image with denoising techniques.

The purpose of our renderer is to compare tuning parameters of acceleration structures. For our study we implement a Monte-Carlo progressive path tracer. To achieve real time performance we decompose the image into tiles and process them in parallel. Our path tracer supports Multiple Importance Sampling [Vea98] for area lights. We also support dielectrics, specular, diffuse and glossy materials. Glossy materials are implemented using anisotropic GGX distribution [WMLT07]. To stay within real time constraints we limit the path length to three bounces. A shorter path length is detrimental to the quality of the measurements, since it overemphasizes the importance of primary rays. Primary rays are coherent compared to subsequent rays in the path. Coherent rays query the acceleration structure from homogeneous directions. While it is possible to optimize for coherent rays, that is not the purpose of our study.

Name	Default Value	Description
quality	2	1: Morton codes 2: binned SAH 3: primitive split
branchingFactor	2	Max number of child nodes
maxDepth	32	Max depth of the BVH tree
sahBlockSize	1	Nb of considered nodes for SAH
minLeafSize	1	Min nb of triangles per leaf
maxLeafSize	32	Max nb of triangles per leaf
travcost	1	Cost of node traversal
intcost	1	Cost of triangle intersection
static	false	Optimize for static scene
compact	false	Optimize memory usage
robust	false	Robust intersection algorithms

Table 1: ***Embree parameters***. *Branching Factor has 2, 4 and 8 as possible values. Because traversal can be optimized with AVX, the branching factor is hardware dependent.* `intcost` *and* `travcost` *are SAH-related parameters and are real numbers between 0 and 20.*

3.2. Parameters and Indicators

In this section we discuss the choice of the acceleration structure and explain the parameters and indicators of our method.

We adopt Embree [WWB*14] as our acceleration structure framework. Embree is Intel's open source implementation of state-of-the-art BVHs. Embree targets high performance and provides a large set of parameters to tailor the BVH to any need. It includes algorithms for low, medium and high quality BVHs. Embree exposes eleven tunable parameters. We summarize them in Table 1. These parameters will affect the construction of the BVH and they are interdependent. For instance, reducing the number of triangle per leaf (min/max leaf size) is linked to the maximum depth. The tree will expand significantly more if the leaf size is small. The quality parameter selects the subdivision algorithm. It can either be Morton codes, binned SAH or binned SAH with primitive splitting, which allows for tighter bounding boxes. For Morton codes the `sahBlockSize`, `travcost` and `intcost` parameters are not used. These eleven parameters will be modified by the tuner to optimize the rendering or the building time. Embree supplies a recommended configuration for these parameters. For our analysis it is important to have a starting configuration. As we want to compare to expert knowledge, we use the recommended values of Embree. This standard configuration is versatile and adapts well to most scenes and most use cases.

We compute 17 indicators as listed in Table 2 to describe the input. We roughly estimate the complexity of the scene with the following indicators: the number of triangles, the number of meshes

Name	Range
Number of meshes	0 to 10^5
Number of triangles	0 to 10^7
Number of lights	0 to 10^5
Camera Position	-10 to 10 (x,y and z)
Target Position	-10 to 10 (x,y and z)
Vertical FoV	0 to 180
Diffuse Ratio	0 to 1
Extent	0 to 10^5
Total area	0 to 10
Area of lights	0 to 10
Area per Mesh	0 to 10
Mean of VarTriSize	0 to 10^{-2}
Variance of VarTriSize	0 to 10^{-2}

Table 2: ***Indicators***. *The camera uses the target position to extract the orientation relatively to an up vector* $(0,1,0)$. *VarTriSize is a list containing the variance of triangle sizes for each mesh.*

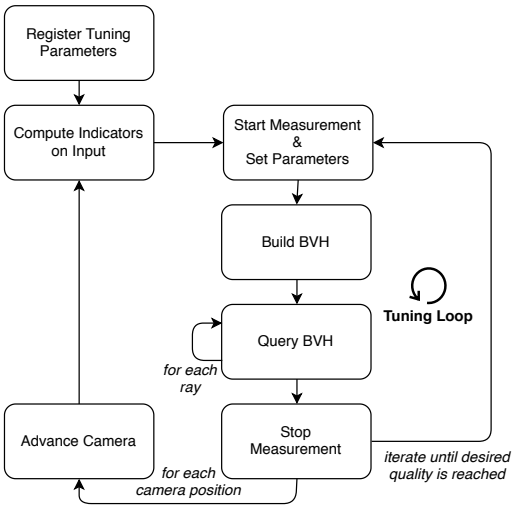

Figure 1: ***The tuning scenario***. *Integration of the tuner in the ray tracing workflow.*

(triangularly tessellated surfaces) and the number light sources. However, scenes with the same number of triangles can have different optimal configurations. Triangle size varies depending on the meshes in the scene, as walls for instance have large triangles while detailed objects have smaller ones. In other words, the triangle size vary among meshes. In addition, triangle size can vary within a mesh and BVHs are susceptible to this variation. To take this into account, we compute the variance of the triangle size for each mesh and name this set *VarTriSize*. As we cannot afford to have one indicator per mesh, we compute the average and variance of *VarTriSize*. These indicators are named Mean of *VarTriSize* and Variance of *VarTriSize*. Additionally, we computed the average area per mesh and the total area of the scene. For scaling reasons, we divided any surface-related value by the square of the diagonal of the scene's bounding box. This normalization is not accurate but sufficient for our needs. The bounding box diagonal is also used as an indicator for the scene's extent. The diffuse ratio indicator is a measure of the average specularity of the surfaces using material properties. The diffuse ratio is between zero and one except for transparent or mirror-like objects, where it is zero. We also added camera parameters: position, orientation and the vertical field of view to account for visibility.

3.3. Tuner integration

Integrating the tuner in the ray tracing code is straightforward. Figure 1 provides an overview of the tuning scenario. First we register the tuning parameters of Embree with the autotuner. Then we start iterating over each camera position. As the camera is part of our indicators we need to recompute them for each position. Then we start the tuning loop. The autotuner starts its measurement and sets the new values for the tuning parameters. The acceleration structure is built using the new parameters. After rendering, the autotuner completes its measurements and computes a new Configuration for the next iteration. We use the tuner to optimize either the rendering time or the total time, sum of rendering and acceleration structure building time. The choice of the time measurement affects the tuner behavior, as shown in Figure 2. When the search-based tuner is optimizing for rendering time, the resulting rendering time improves over the default configuration. The total time, however, increases. When total time is used as the measurement value, we see a smaller improvement to the resulting rendering time. As expected, the total time is now substantially smaller than what the default configuration achieves. The hybrid tuner will learn from this exploration to compute efficient configuration in exploitation mode. During exploitation no search is performed. Instead the learned model predicts a configuration based on the current indicators.

4. Hybrid Online Autotuning

The main obstacle to practical deployments of machine-learning or model-based methods is training. Pure offline techniques require massive amounts of samples a-priori to build the models. The quality of the models depends on the representativeness of the samples. Online approaches on the other hand learn from new data that actually occurs in the deployment context. The main question then becomes how to construct the initial model. Starting with an imprecise initial model means sub-optimal predictions and decreased performance. While offline training is certainly viable to seed an online approach, it still requires input samples, which can be hard to obtain in particular given the heterogeneity of 3D scenes. As a compromise, we propose *hybrid online autotuning*, combining classical search-based tuning with model-based prediction. The high-level tuning process is shown in Figure 3. The tuner observes the program and the system state through the indicators, which serves as an approximation to the true program and system state in the remainder of the paper. We therefore use the term "state" to refer to the current indicator values in the remainder of the paper. For every change in program or system state, the hybrid tuner chooses to either exploit the model or to explore the configuration space. In either mode, performance feedback from the application for every sampled configuration is used to update the model. Observed states, sampled configurations, and the performance feedback can

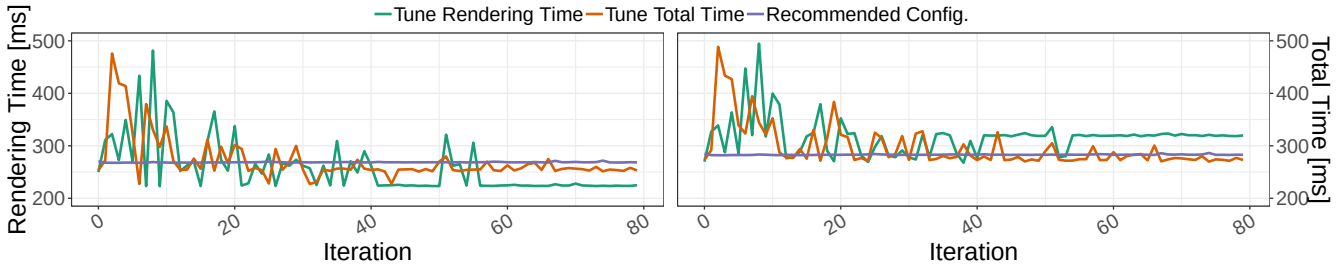

Figure 2: *Changing the feedback function drastically affects the tuning results. Red optimizes the rendering time only, whereas green targets the total time of rendering and building the acceleration structure. Blue is the configuration recommended by the literature.*

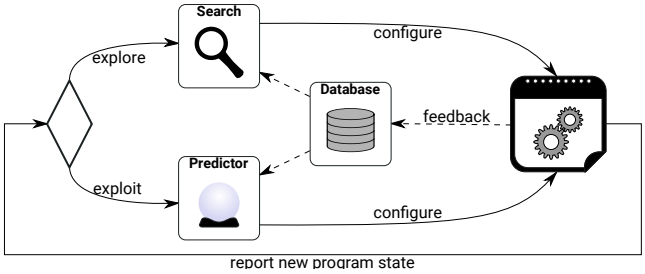

Figure 3: *The hybrid online tuning workflow.*

be stored in the tuning database to avoid sampling the same state-configuration pair twice. Choosing between exploration to gather new data and exploitation of known information is a central operation in Reinforcement Learning (RL) [SB18]. To implement the decision making we borrow an algorithm from that field, namely the ε-Greedy algorithm. The ε-Greedy algorithm is remarkably simple: With a probability of ε, choose to explore, otherwise greedily exploit.

A key difference between exploring and exploiting is that exploitation is one-shot. For a new state the autotuner produces exactly one configuration which once applied is kept until the next state change. Exploring on the other hand samples multiple configurations per state. That contrast is important because for ray tracing, updating the configuration incurs cost: Changing the BVH parameters requires rebuilding the data structure, which is an expensive operation. For exploration, we use a variant of the Nelder-Mead simplex algorithm [NM65]. Our version is similar to that of Chang [Cha12], using Latin Hypercube sampling [MBC79] as an initialization method.

We investigate two different types of models in this paper. The first one is a naive tabular approach and the second is based on a generalized reinforcement learning method. Both approaches are described in detail in the following two sections.

4.1. Nearest-Neighbor Prediction

The naive way to predict configurations is to record a table of previous states, configurations, and the observed runtime of those configurations. When the application enters a known state, the best known configuration can be selected from the table. Although this solution is obvious, it is also obviously limited: The state space is practically unbounded and too large to store let alone explore in its entirety. Even if it could be stored, we have to assume that for most states the table query would be unsuccessful, at least until the table is sufficiently complete. Therefore, a successful predictor must be able to service queries for unknown states. To realize such a predictor on the tabular model, we use a nearest neighbor approach: If the current state is unknown, respond with the best configuration for the state that is closest regarding Euclidean distance in the state space.

4.2. Prediction Through Function Approximation

Even when using nearest-neighbor predictions, achieving high accuracy may require enormous tables, which are expensive to store and query. Fortunately, that problem has been investigated in the past in the field of Reinforcement Learning. Generalized RL methods offer control (i.e., state-sensitive decision making) for large, both discrete and continuous state spaces through function approximation [SB18]. One more recent such algorithm is Greedy-GQ [MSBS10]. The algorithm is an off-policy gradient-based temporal difference learning method. It is relevant to our use-case because of a set of interesting properties: Off-policy learning enables learning from samples obtained using a different method than the predictor. In particular that allows us to learn from offline data (e.g., from past experiments or tuning searches) but also during online search, and even to eventually disable exploration entirely. Additionally, Greedy-GQ supports incremental online training and imposes no limits on the features we can use to represent the model inputs. Features encode properties of the model input in an application specific way. Both the memory and runtime complexity of the Greedy-GQ algorithm is linear in the number of features. The number of features is both constant and much smaller than the cardinality of the search space. Greedy-GQ is thus an improvement in terms of size over the tabular approach.

At its core, Greedy-GQ manages the linear value function $Q_\theta(s,a)$, which assigns every state s and "action" a a value. Intuitively, the value estimates the benefit of choosing the action a in state s. What is called an action in RL terminology corresponds to one of the configurations produced by the tuning search in the past. We thus use the terms interchangeably here. During exploitation, the predictor selects the action that maximizes Q_θ for the current state. The Greedy-GQ value function is defined as $Q_\theta(s,a) = \vec{\theta} \cdot \varphi(s,a)$, where $\varphi(s,a)$ is a vector of real-valued fea-

tures, and $\vec{\theta}$ are the coefficients to be learned. When a state-action pair s_t, a_t is evaluated, the coefficients are updated based on the "reward" R_{t+1} observed by the application, which is just the inverse of the runtime the configuration a_t produced.

The update equations for $\vec{\theta}$ are controlled by three hyper-parameters: α, β, which are learning rates, and γ, which is a discount factor governing the influence of expected future values. The latter parameter has a particularly interesting semantic in our use case. In the basic RL framework, actions taken in a given state influence the states that will be assumed in the future. However in our application, state changes are unaffected by the parameter configuration we pick. This means that we can set γ to zero, creating an algorithm that is sometimes called a "myopic" [SB18]. The update rule in our implementation of Greedy-GQ thus becomes $\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha(R_{t+1} - \vec{\theta}_t \cdot \varphi(s_t, a_t))\varphi(s_t, a_t)$.

As features we use radial basis functions [KA97]. A radial basis function (RBF) is a Gaussian curve $\varphi_i(x) = \exp(-\omega_i||x - c_i||)$, where $x = (s, a)$ is the concatenation of the state and action vectors. The RBF is centered at c_i with a width determined by ω_i. The hyper-parameters of our RBF model are the number of features and each feature's center and width. Based on preliminary experiments we chose 3.5 times the number of parameters and indicators as feature count, which results in 100 features. The widths are set to a fixed value of 1. The embedding of the indicator states and parameter configurations into the feature representation and the generation of the feature centers is described in the following.

To realize the feature embedding we must preprocess our model inputs. The scales of the indicators vary drastically. Because the distance metric is scale-variant, indicators spanning on larger scales will carry more weight in the distance computation than smaller ones. Before passing them into the RBF model, we thus need to normalize parameters and indicators to a comparable scale. For each of them we estimate their minimum and their maximum value. This is quite easy for the parameters since they are constrained by the design of the acceleration structure. For the indicators, we need to select a meaningful maximum (with the default minimum being zero) based on data. The biggest evaluation scene has 1.8 million triangles, we chose 10 millions as our upper bound. Regarding the extent, we chose 100000 and for the number of lights we chose 100000 even though most scenes have only the environment light. The maximum number of meshes was fixed to 100000. For the area, the biggest one is 5.83, so the area maximum was fixed to 10. For the field of view, it is logical to fix a maximum of 180, for the Diffuse Ratio a maximum of 1.0. For the camera positions a maximum of 10000 was used.

Finally, we need to define the centers of the features to form the RBF model. We produce the centers by quasi-randomly placing them in the down-scaled parameter and indicator space. Using Latin Hypercube Sampling again we obtain a space-spanning set of points within the space.

5. Evaluation

In this section we present the evaluation of our autotuned ray tracer. We compare hybrid autotuning to classical continuous online-autotuning. Because of the ε-Greedy policy that hybrid tuning uses to decide between exploration and exploitation, we are able to evaluate both modes in isolation. The ε-Greedy policy is purely stochastic. Therefore the tuner behavior in a production environment can be extrapolated from the per-mode results probabilistically.

We benchmark on seven scenes with differing complexities and characteristics. The scenes are `sponza`, `gallery`, `vokselia_spawn`, `conference`, `fireplace_room`, `buddha` [Mor17] and `bmw-m6` (which we call `car`) [PJH17]. These scenes have been chosen for their wide variety of geometry. The `vokselia_spawn` scene for instance is a rather large, open world but with a peculiar aspect: triangles are almost all of the same size because the world is composed of boxes. The `buddha` scene contains a highly triangulated mesh, but it only occupies a portion of the viewport at a time. The distance to the object is thus particularly important as fewer rays will intersect the object when the distance between camera and object increases. The `car` scene is quite similar to `buddha`, being highly tesselated, but it also features two reflective planes that bounce rays back to the car instead of ending in the environment map. The `gallery`, `conference`, `fireplace_room`, and `sponza` scenes enclose the cameras, so most of the lighting is indirect. Light paths are therefore occluded and rays tend to bounce more often causing the average path length to be higher.

We distinguish two ray tracing use-cases: progressive and real-time rendering. In the former, the acceleration data structure is constructed once and the rendering takes place as long as the scene/camera does not move. For real-time rendering, there is only one rendering step. This distinction affects the target function of our tuning scenario. The performance of progressive rendering is dominated by the rendering step, so the tuner should minimize rendering time. The target function of real-time rendering must additionally account for the time required to build the data structure. We therefore minimize the sum of construction and rendering time in this case.

5.1. Performance Results: Exploration Through Search-Based Online-Autotuning

First, we establish that online-autotuning improves ray tracing performance. We measure the speedup achieved by the autotuned configuration. Autotuning is initialized with the commonly recommended values, listed in Table 1. The recommended configuration is also used as the baseline for the speedup computations. We conducted two experiments, one optimizing rendering time only and one optimizing the total time (i.e., the sum of rendering time and acceleration structure building time).

Figure 4 shows speedups of converged autotuning runs on the seven scenes. The `sponza` scene benefits the most from autotuning and gains speedup of 11% (median). The other scenes show a moderate improvement, between 2 to 4% on average. Since the search algorithm starts on the recommended configuration, the converged configuration never falls below a speedup of 1. Figure 2 shows typical tuning runs. We observe the pattern of initial performance degradation during the first few iterations. After about 60 iterations the runtime converges to a better result than the recommended configuration. Additionally, we see how the choice of the

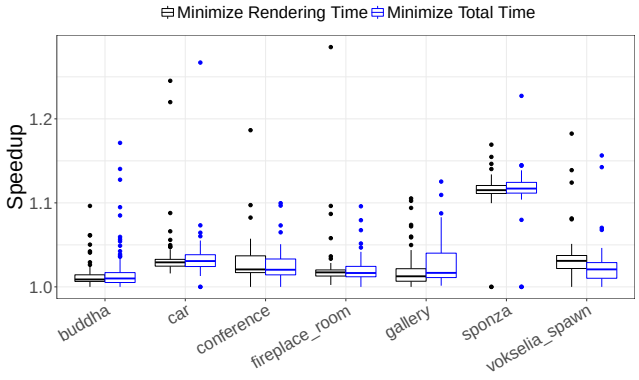

Figure 4: *Speedups of converged configurations relative to recommended configuration.*

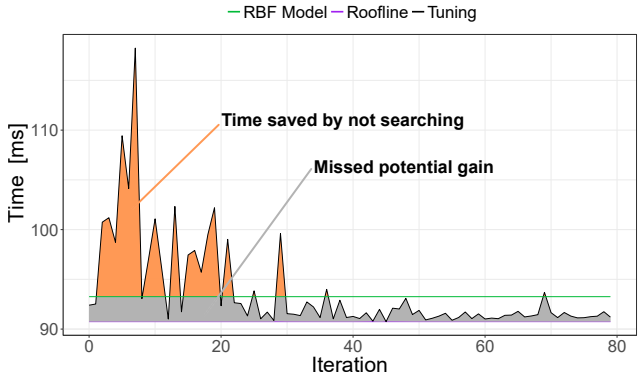

Figure 5: *Example: Tuning traces on the* `car` *scene. The area under the curves represents the time required to compute a fixed number of iterations. The colored areas represent the performance gain and missed potential of a predictor.*

target function (i.e., optimizing rendering time or total time), affects the configuration found. When tuning rendering time, the tuning result outperforms the default configuration by about 50 ms as shown in the left hand plot. Considering the total time on the other hand, the *same* tuning result shown in the right hand plot degrades performance by roughly 40 ms. However convergence time and wildly varying render times is still a major problem of online-autotuning.

5.2. Performance Results: Exploitation Through Predictive Online-Autotuning

In the following we report the performance results achieved by the prediction component of our hybrid tuner. We compare the performance of the predicted configurations to search based tuning. To understand the comparisons, consider the motivating example in Figure 5. We show in this figure the typical runtime behavior as observed during the tuning process. The jagged black curve shows the measured runtime during every iteration of a Nelder-Mead search. This is a tuning curve for one of the camera positions on the `car` scene. The purple bottom horizontal line represents the tuning curve (Roofline) of an ideal predictive autotuner: Using the best configuration from the first iteration. Note that this is not the *true* roofline, which we cannot know without exhaustively exploring the configuration space. We consider as roofline here the best configuration we found using the search. Lastly the green horizontal line exemplifies the result produced by our predictor. In general it produces a configuration that is worse than both the roofline and the tuning result. We quantify in the following sections how much worse the prediction result actually is. The primary aim of the predictor is not to produce a configuration as good as the roofline or the tuning search. Instead, its purpose is to reduce the total time spent searching, i.e., the time spent to complete a specific number of iterations. For progressive rendering this relates to the time required to compute a fixed number of frames. The time required to complete the 80 iterations in the example in Figure 5 is the area under the respective curves. The area highlighted in orange in the figure shows the performance gain achieved by using the predicted green configuration. The gray-colored area in turn shows the missed potential improvement. In the evaluation presented in this section we analyze our tuner with regard to both aspects: How much does our predictor improve the time required to complete a fixed number of iterations, and how close to the ideal performance does it get.

5.2.1. Generating Training and Validation Data

To vary the indicator state for the evaluation scenes in a controlled manner, we generate cameras moving through the scenes. The camera path is designed to pass through a variety of different configurations. For example, for the `buddha` scene the camera path will go through a closeup of the Buddha statue and then move further away, dramatically changing the amount of visible geometry. This is to ensure that the training set is as diverse as possible. We create these 100 cameras positions for every scene, interpolating the movement using a simple spline. To produce training and validation data, we first run the classic Nelder-Mead tuner for 80 iterations. The number is sufficient for the search to converge for all possible camera positions. Each search iteration builds the acceleration structure and renders one spp. This gives us to $80 \cdot 100 \cdot 7 = 56000$ data points per scene as a baseline. The evaluation machine features an i7-6700 at 3.40 GHz with 8 hardware threads, 8 MB of cache and 32 GB of RAM. The baseline data is recorded as mappings of indicator states to parameter configurations and timings.

From the baseline data we randomly pick 80 camera positions for training, and the remaining 20 for validation for each scene. We repeat the training and verification process 15 times. To train the nearest-neighbor predictor, we generate a lookup table mapping indicator states to optimal baseline configuration. The table for each training round contains $80 \cdot 7$ configurations. For an unknown state, the nearest-neighbor predictor returns the configuration that the table maps to the closest known indicator state in terms of Euclidean distance. The RBF model is trained on the full training set of $80 \cdot 80 \cdot 7 = 44700$ data points. Using the update rule described in Section 4.2 we obtain the linear combination of RBF features.

For the verification phase we evaluate the predictor on the 20 remaining cameras. We compare the performance of the configuration returned by the predictor with the baseline data. Note that this leaves room for error: The baseline data does not necessarily contain the globally optimal configuration. Only an exhaustive

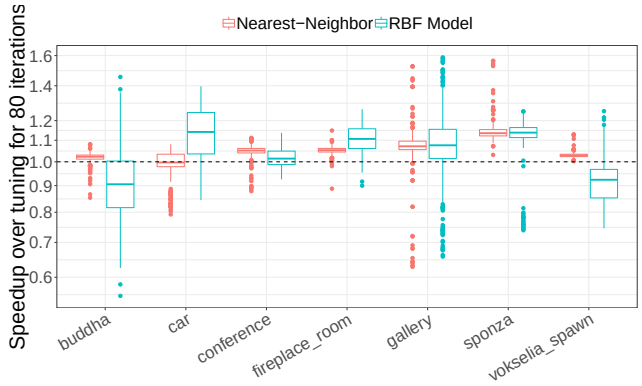

Figure 6: *Speedups of our predictor over search-based tuning. Both search and predictor minimize rendering time only.*

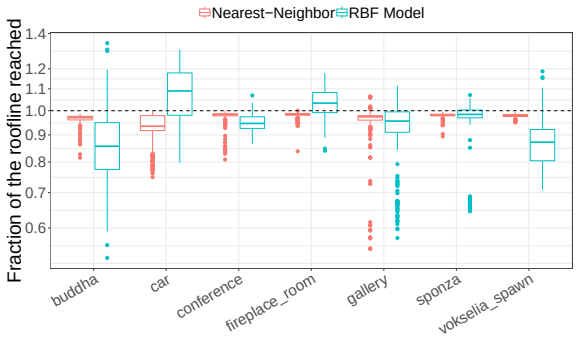

Figure 7: *Comparison of the predictors against the rooflines for every scene and camera position. Both baseline and predictor minimize rendering time only.*

and extremely time-consuming exploration of the parameter space could find that configuration. However, since we are only comparing search-based and predictive autotuning here, our comparison is sufficient.

5.2.2. Optimizing Render Time

We first analyze the behavior of our predictor for progressive rendering, which means rendering time only is to be minimized. In Figure 6 we show the speedup achieved by the predictors over the search baseline. The search baselines are the accumulated rendering times for 80 spp for the respective scenes and camera positions. In the figure, we compare the result of the table-based nearest neighbor predictor and the RBF model. We train the model as explained above. Although all observations are used to build the model, only those the Nelder-Mead algorithm deemed locally optimal are considered by the predictor. The speedup results shows that both the nearest neighbor and the RBF model predictor outperform the baseline in most cases. On average, using the geo-mean (geo-mean), they achieve a speedup of 1.05 and 1.04, respectively. However, we see two scenes where the RBF model does not find adequate configurations, namely `buddha` and `vokselia_spawn`.

Based on the speedup results we can also compare the overheads. We consider the overhead against a virtual baseline, which is the best known configuration for a camera and scene. Unlike for the roofline comparison done below, we choose the best among both the search and the predictions to make sure the overhead is non-negative. We achieve a geo-mean overhead reduction of up to 87.5% (for the `sponza` scene) using the nearest-neighbor predictor. The RBF predictor reduces only up to 79.2% of the overhead (for the `gallery` scene). Although the speedups appear to be small, the presentation here hides an important fact: We are comparing only render time. While that is the appropriate metric for progressive ray tracing it is not true for our baseline. The Nelder-Mead search samples different configurations at every iteration. Changing the configuration requires rebuilding the acceleration structure. In practice this incurs a substantial overhead for the baseline, but we exclude this in our comparison here for fairness.

In Figure 7 we compare our predictors against the roofline for every scene and camera position. In all cases both nearest neighbor and the RBF model are close to a speedup of one versus the rooflines. The geo-mean for both is above 95%. Although the speedup we achieve over the search appears small, we are close to what is actually achievable. Interestingly, the RBF model outperforms the roofline in several cases. This indicates that the baseline search has not found the globally optimal configuration for these scenes. This is a caveat of the Nelder-Mead search algorithm, which only converges against local optima. In Figures 6 and 7 we note outliers for the gallery and sponza scenes. These data points are consistent with the camera transitioning from non-occluded to highly occluded configuration.

Given the performance of only five percent one may ask: Why choose prediction over search? The question ignores the overhead in the baseline we are hiding. To answer the question, we present another view on the search and prediction comparison in Figure 8. The plot shows the number of tuning iterations and thus rendering samples required for the search to actually outperform the prediction. The data points the plot is based on represent the iteration number in which the curves of the cumulative times intersect. For the RBF model the mean break even point is 362, and for nearest neighbor it is 1771. That means, on average at least 362 rendering samples are necessary to observe a benefit from the configuration found by the search, even though that particular configuration is better than what the predictor provided. For visualization, we excluded 875 data points from the plot. That set includes 30 points for which the break-even point is greater than 10000. Those refer to predicted configurations which yielded similar performance to what the search produced, so the curves are nearly parallel. For 845 points the break even point is negative, which are cases where the predictor found a better configuration than the search. For the RBF model these stem predominantly from the `car` and `fireplace_room` scenes, where roughly 12% of the data points outperform the roofline.

5.2.3. Optimizing Render and Build Time

We evaluate the behavior of our predictors in a real-time context. In this use case the BVH has to be rebuilt every frame. The feedback function of the tuner measures the sum of rendering time and building time in this scenario. Figure 9 shows that the nearest neighbor approach still outperforms the baseline on most scenes. The only

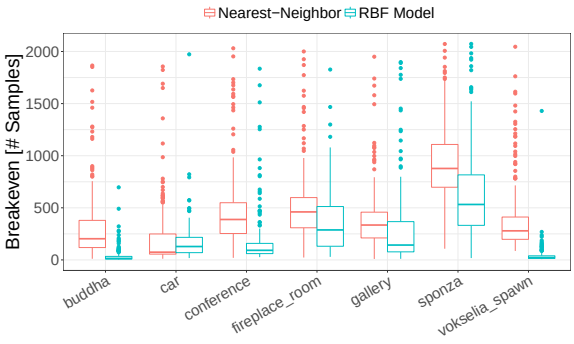

Figure 8: *Break-even points of the Nelder-Mead search. A substantial number of Nelder-Mead iterations is required to break even with predicted configurations.*

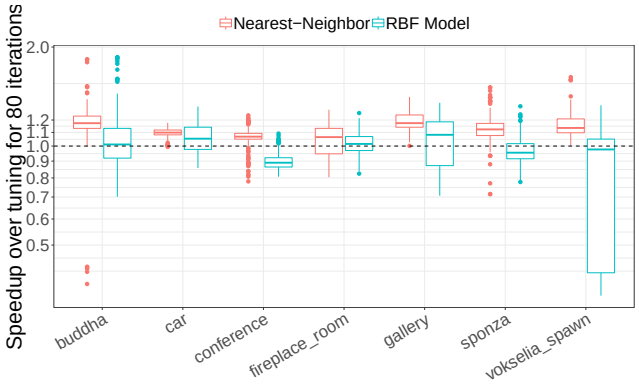

Figure 9: *Speedups of our predictor over search-based tuning. Both search and predictor minimize rendering time and building time.*

scene where more than 25% of the runs do not outperform the baseline is `fireplace_room`. On average, the nearest-neighbor approach achieves a speedup of 12%. The RBF model shows good behavior on `car` and `gallery` but the performance is underwhelming compared to the previous approach and does not accelerate rendering. When minimizing total time We achieve a geo-mean overhead reduction of up to 89.3% (for the `vokselia_spawn` scene) using the nearest-neighbor predictor. The RBF predictor reduces only up to 53.4% of the overhead (for the `gallery` scene). In Figure 10 we compare our predictors to the roofline for every scene and camera position. The roofline is again the best configuration found during search. The nearest-neighbor predictor is close to the roofline in most cases with an average of 92%. Compared to the nearest-neighbor predictor our RBF models shows greater variance and slower performing averages with 83%. We note that the RBF model tend to cross the roofline way more often than the nearest neighbor variant. This is due to the RBF model finding a better local optimum in configurations that do not satisfy the nearest-neighbor criterion.

6. Conclusion

In this paper we applied online autotuning to parallel raytracing. Autotuning is not widespread in ray tracing yet. Hard real-time

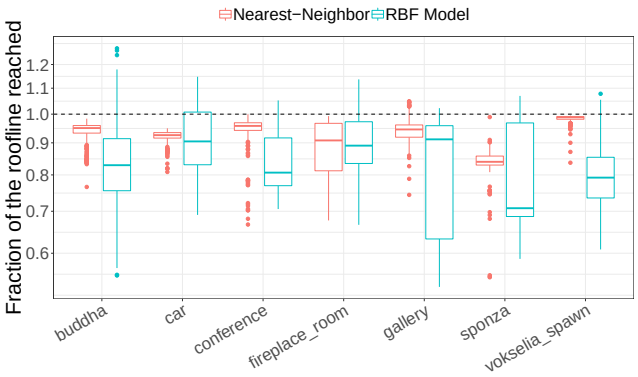

Figure 10: *Comparison of the predictors against the rooflines for every scene and camera position. Both baseline and predictor minimize rendering time and building time.*

requirements and the fact that updating parameter configurations is a costly operation render established autotuning techniques insufficient. To deal with these issues we introduce a novel hybrid online autotuning scheme. Borrowing methods from the field of reinforcement learning we combine search-based tuning with model-based prediction. Search-based tuning offers well-established exploration capabilities to find locally optimal parameter configurations for given inputs. Model-based prediction on the other hand produces configurations without sampling, eliminating the need to rebuild the acceleration structure for new parameter configurations. Combining autotuning and machine-learning, we successfully mitigated their respective downsides.

Our evaluation on seven scenes of varying complexity and properties shows that our predictors are able to compensate for the overhead of the search-based tuning. A simple nearest-neighbor achieves 95% of the performance offered by the search-based tuning while reducing the overhead by 5% of the rendering time. Because the nearest-neighbor predictor requires maintaining large input state tables, we also investigate a function approximation approach. We train a radial basis function model online during the search to provide the predictions. Although the space complexity of the model is constant with respect to the number of inputs, its performance is competitive. Our approach is also scalable to many-node parallelism. We can either use a predictor for each node or a predictor for the entire architecture, but future work in this direction is required.

Our results show that always-on online autotuning can be deployed with ray tracing, and provide substantial acceleration. Using our hybrid tuning technique the overhead of the search can be mitigated.

Acknowledgments

We thank Timo Kopf, Kevin Zerr and André Wengert whose Master's theses investigated early designs of Hybrid Tuning. We also would like to thank Hisanari Otsu for the Lightmetrica renderer and his comments on the draft of this paper. This project was funded by Deutsche Forschungsgemeinschaft (DFG, German Re-

search Foundation) – project 299215159 with Grants DA 1200/5-1 and TI 264/10-1.

References

[AKV*14] ANSEL J., KAMIL S., VEERAMACHANENI K., RAGAN-KELLEY J., BOSBOOM J., O'REILLY U.-M., AMARASINGHE S.: OpenTuner: An extensible framework for program autotuning. 2

[BAC*18] BURLEY B., ADLER D., CHIANG M. J.-Y., DRISKILL H., HABEL R., KELLY P., KUTZ P., LI Y. K., TEECE D.: The Design and Evolution of Disney's Hyperion Renderer. *ACM Transactions on Graphics 37* (2018). 3

[BGW13] BALAPRAKASH P., GRAMACY R. B., WILD S. M.: Active-learning-based surrogate models for empirical performance tuning. In *IEEE International Conference on Cluster Computing* (2013). 3

[BHC*16] BAO W., HONG C., CHUNDURI S., KRISHNAMOORTHY S., POUCHET L.-N., RASTELLO F., SADAYAPPAN P.: Static and Dynamic Frequency Scaling on Multicore CPUs. *ACM Transactions on Architecture and Code Optimization* (2016). 3

[BHH15] BITTNER J., HAPALA M., HAVRAN V.: Incremental BVH construction for ray tracing. *Computers & Graphics 47* (2015). 2

[BPC12] BERGSTRA J., PINTO N., COX D.: Machine learning for predictive auto-tuning with boosted regression trees. In *Innovative Parallel Computing* (2012). 3

[CFS*18] CHRISTENSEN P., FONG J., SHADE J., WOOTEN W., SCHUBERT B.: RenderMan: An Advanced Path-Tracing Architecture for Movie Rendering. *ACM Transactions on Graphics 37* (2018). 3

[Cha12] CHANG K.-H.: Stochastic Nelder-Mead Simplex Method - A New Globally Convergent Direct Search Method for Simulation Optimization. *European Journal of Operational Research* (2012). 5

[DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. In *Proceedings of the 19th Eurographics Conference on Rendering* (2008). 2

[DK17] DAHM K., KELLER A.: Learning Light Transport the Reinforced Way. In *ACM SIGGRAPH Talks* (2017). 2

[GD12] GANESTAM P., DOGGETT M.: Auto-tuning Interactive Ray Tracing Using an Analytical GPU Architecture Model. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units* (2012). 2

[HA19] HAINES E., AKENINE-MÖLLER T.: *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. 2019. 1

[KA97] KRETCHMAR R., ANDERSON C.: Comparison of CMACs and Radial Basis Functions for Local Function Approximators in Reinforcement Learning. In *Proceedings of International Conference on Neural Networks* (1997). 6

[Kaj86] KAJIYA J. T.: The Rendering Equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* (1986). 3

[KK86] KAY T. L., KAJIYA J. T.: Ray Tracing Complex Scenes. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* (1986). 1

[MB18] MEISTER D., BITTNER J.: Parallel Reinsertion for Bounding Volume Hierarchy Optimization. *Computer Graphics Forum 37* (2018). 2

[MBC79] MCKAY M. D., BECKMAN R. J., CONOVER W. J.: A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics* (1979). 5

[Mor17] MORGAN MCGUIRE: Computer Graphics Archive. http://casual-effects.com/data/, 2017. 6

[MSBS10] MAEI H. R., SZEPESVÁRI C., BHATNAGAR S., SUTTON R. S.: Toward Off-Policy Learning Control with Function Approximation. In *Proceedings of the 27th International Conference on Machine Learning* (2010). 5

[MSH*14] MURALIDHARAN S., SHANTHARAM M., HALL M., GARLAND M., CATANZARO B.: Nitro: A Framework for Adaptive Code Variant Tuning. In *Parallel and Distributed Processing Symposium, IEEE 28th International* (2014). 3

[NHD10] NOVÁK J., HAVRAN V., DACHSBACHER C.: Path Regeneration for Interactive Path Tracing. In *Eurographics* (2010). 3

[NM65] NELDER J. A., MEAD R.: A Simplex Method for Function Minimization. *The Computer Journal 7*, 4 (1965). 5

[PJH17] PHARR M., JAKOB W., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation*, third ed. 2017. 6

[SB18] SUTTON R. S., BARTO A. G.: *Reinforcement Learning: An Introduction*. 2018. 5, 6

[Set09] SETTLES B.: *Active Learning Literature Survey*. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009. 3

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *Proceedings of the 1st ACM Conference on High Performance Graphics* (2009). 2

[ŢCH02] ŢĂPUŞ C., CHUNG I.-H., HOLLINGSWORTH J. K.: Active Harmony: Towards Automated Performance Tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing* (2002). 2

[TO12] TOKUYOSHI Y., OGAKI S.: Real-time Bidirectional Path Tracing via Rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2012). 3

[TPKT16] TILLMANN M., PFAFFE P., KAAG C., TICHY W. F.: Online-Autotuning of Parallel SAH kD-Trees. In *IEEE Parallel and Distributed Processing Symposium* (2016). 1, 2

[Vea98] VEACH E.: *Robust Monte Carlo Methods for Light Transport Simulation*. PhD Thesis, Stanford University, 1998. 3

[VHS12] VINKLER M., HAVRAN V., SOCHOR J.: Visibility driven BVH build up algorithm for ray tracing. *Computers & Graphics 36* (2012). 2

[VKŠ*14] VORBA J., KARLÍK O., ŠIK M., RITSCHEL T., KŘIVÁNEK J.: On-line learning of parametric mixture models for light transport simulation. *ACM Transactions on Graphics 33* (2014). 2

[WD98] WHALEY R. C., DONGARRA J. J.: Automatically Tuned Linear Algebra Software. In *IEEE/ACM Conference on Supercomputing* (1998). 2

[WG14] WEBER N., GOESELE M.: Auto-tuning Complex Array Layouts for GPUs. In *Proceedings of the 14th Eurographics Symposium on Parallel Graphics and Visualization* (2014). 2

[WG17] WODNIOK D., GOESELE M.: Construction of bounding volume hierarchies with SAH cost approximation on temporary subtrees. *Computers & Graphics 62* (2017). 2

[WJA*17] WALD I., JOHNSON G., AMSTUTZ J., BROWNLEE C., KNOLL A., JEFFERS J.: OSPRay - A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics 23* (2017). 1, 3

[WMLT07] WALTER B., MARSCHNER S. R., LI H., TORRANCE K. E.: Microfacet Models for Refraction Through Rough Surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques* (2007). 3

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics 33* (2014). 3