Analysis of Acceleration Structure Parameters and Hybrid Autotuning for Ray Tracing

Killian Herveau, Philip Pfaffe, Martin Tillmann, Walter F. Tichy, Carsten Dachsbacher

(Invited Paper)

Abstract—Finding optimal parameters for acceleration structures for raytracing is key to improved performance. Previous research has shown that a speedup of over 10% of rendering time is possible. Some parameters are interdependent which complicates the process of finding an optimal configuration. It is hence interesting to find them efficiently. Autotuning is an automatic optimization scheme able to search for optimal configurations and has been applied successfully to kD-trees in the past, which we apply today on BVHs.

The more parameters to optimize, the more difficult it is to find optimal solutions. In this paper, we analyze in detail the behavior of the parameters and their impact on acceleration structure building and rendering time. We show the interdependence and context sensitivity (i.e., scene, viewpoint) of the parameters. Based on the use case, this allows to target only crucial parameters.

Convergence speed towards an optimal configuration is essential. To find better parameters, the autotuner needs to build the acceleration structure over and over, changing parameters every time. We introduce a hybrid model-based prediction and online autotuning method to address this issue. The prediction model allows for both instantaneous near-optimal configurations when inputs are known or similar, and efficient search of the configuration space when inputs are completely new.

Online autotuning outperforms configurations recommended in literature by up to 11% median. The prediction model achieves 95% of the maximum speedup of the autotuner while reducing 90% of its overhead. Thus, hybrid online autonuning enables always-on tuning in ray tracing.

1 Introduction

Performance and image quality are the decisive metrics of ray tracing. In its widespread application in movie production, architecture, or starting in video games, its users are often forced to prioritize one over the other. Advances in both hardware and software however seek to accelerate the rendering performance while achieving ever better quality and fidelity. Parallelization and specialized hardware features such as NVIDIA's RTX technology [1] open new ways to accelerate ray tracing.

Acceleration structures such as Bounding Volume Hierarchies (BVH) [2] are always necessary and heavily optimized in many renderers. The quality of acceleration structures is crucial and yet there is still untapped potential even in state-of-the-art acceleration structures. The issue lies in the configuration of these structures. The default configuration can be enough when ray tracing is a small subset of the end application, where optimization would not be worth the cost. On the other hand, when a project relies heavily on ray tracing, the default is almost always suboptimal [3]. The recommended settings are static, based on experiences or experiments on specific hardware and specific inputs. In specific contexts, the configuration is likely to be near optimal, but we cannot expect it to generalize to the variety of hardware or inputs that ray tracing applications must deal with.

As an example, the SAH metric, used to measure the tree quality, introduces two parameters: intersection cost and traversal cost. The values recommended in the literature are, as we will see, not optimal. Other parameters such as the tree leaf size or its depth make the optimization more complex and error prone. A scene containing large and small triangles would likely benefit from triangle splitting [4] to prevent overly large nodes. These examples illustrate that dynamically selecting configurations according to the context is necessary to obtain the best performance.

Human decision is still involved in today's pipelines in the choice of many parameters, which makes the process unreliable especially with changing context. Autotuning is one method to find the best parameters. The underlying idea is simple: sample configurations one after the other using an intelligent search algorithm. During runtime, this approach allows the configuration to adapt to the context. The algorithm simply restarts upon context change.

This mechanism presents certain caveats. Autotuning uses heuristic search algorithms on a parameter space that is too large to be entirely explored. Consequently, the configurations found are only locally (in parameter space) optimal. The second caveat is the 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 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. When used in a production environment, users may notice the decreased performance and varying frame rate.

Contributions. We propose a study of the parameter space of the BVH acceleration structure in a ray tracing context. Using 800 000 random configuration samples, we ana-

- *K. Herveau and C. Dachsbacher of the Karlsruhe Institute of Technology (IVD), 76131 Germany*
- *P. Pfaffe, M. Tillmann and W. F. Tichy are with the Institute for Program Structures and Data Organization of the same university*

lyze the interconnections between the different parameters and their behavior for a variety of inputs. This analysis can provide relevant insight regarding acceleration structures independently of autotuning. Yet, it also helps to reduce configuration space by constraining the range of parameters, or selecting only the most crucial ones, improving reliability of our hybrid method.

We propose the Hybrid Online Autotuning technique to address the overhead of autotuning. It combines classical search-based tuning with online learning. The search explores the configuration space and trains a predictor simultaneously. The predictor maps from inputs to configurations. Inputs are represented through a set of metrics called *indicators*. Example indicators are the number of triangles, total surface area or camera position. Upon a change in the input, the hybrid tuning method decides whether to search the configuration space or to *exploit* the predictor. We show that classical search achieves a performance speedup of up to 11% median over the recommended default configuration. The predictor trained during the search reduces the overhead of the search by almost 90%. It, alone, achieves 95% of the performance of the configuration found by searching.

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 [5] or visibility [6]. Dammertz et al. [7] 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. [8] presented an incremental data structure where the number of modified nodes and the fraction of nodes to be updated each iteration are free parameters. Wodniok et al. [9] hand pick several parameter values for their BVH's partitioning strategy. More recently, Meister et al. [10] 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 present approaches aiming at optimizing ray tracing and acceleration structures automatically for given inputs or hardware. We then discuss recent works in the wider field of autotuning including context-sensitive tuning. Finally, we explore some applications related to the exploration of the parameter space in visualization.

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. [11] 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 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. [12] 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 of the acceleration structure and none of its exposed parameters.

Several works demonstrated the benefits of machine learning for rendering. Vorba et al. [13] and later Reibold et al. [14] used Gaussian Mixture Models (GMM) to represent sampling distributions. They show improvements in the convergence rate for path tracing, bidirectional path tracing, and Metropolis light transport. Similarly, Dahm et al. [15] 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 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 been used in a ray tracing application by Tillmann et al. [3]. They build a kD-tree by optimizing the splitting criteria, obtaining up to $1.96\times$ speedup over the standard SAH kD-tree. However, they do not investigate techniques to reduce the autotuning overhead.

2.2 Parameterization

The rise of machine learning methods requiring optimization renewed the interest in visualization methods to assist users in finding optimal configurations. Human-in-the-loop methods are still an active area of research. Recently, Liu et al. [16], [17] created interactive visualization recommendations that guide human-in-the-loop parameterisation. Many works deal with hyperparameterisation, especially for machine learning techniques sensitive to initializations. Hence, Angelos et al. [18] create a visualization tool for evolutionary optimization, guiding the user to find the best starting parameters for the model. HyperTuner [19] and more recently HyperTendril [20] help encapsulate the automatic optimization of hyperparameters and let the user guide the optimization by choosing initial parameters. These approaches are difficult to apply in ray tracing. They can be valuable for searching one optimal configuration, but will not adapt continuously as the scene changes, and have to be used offline. Our work focuses on automatic and continuous iterations, applying the optimization in an online context.

2.3 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 [21], which optimizes

BLAS primitives during installations using training examples shipped with the software. Currently, ActiveHarmony [22] and OpenTuner [23] are among the most widely used general purpose autotuning tools. Both use a 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.

Active learning refers to autotuning approaches that build and refine prediction models online. The basic principle of active learning is to use the prediction model itself to recommend samples to refine the model [24]. A recent occurrence of this method was presented by Balaprakash et al. [25]. 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.

Bao et al. [26] apply model-based tuning to minimize energy consumption by selecting an optimal CPU frequency at program runtime. They benchmark the system on individual relevant features, from which they build a decision tree.

These related works 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 while a model is still learning is subpar. Our approach provides a target-oriented way to train the model and to optimize performance even during training.

3 Ray Tracing

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

3.1 Renderer

Ray tracing [27] and especially path tracing are today a standard of the movie industry [28], [29]. It is increasingly used for interactive applications [30] to help rasterization with global illumination [31]. Interactive ray tracing is also available on CPU, e.g. with OSPRay [32]. In real-time applications, it is common to use one sample per pixel (spp) and then process the image with denoising techniques [33], [34], [35].

Embree [36] is Intel's open-source implementation of state-of-the-art BVHs, and is representative of optimized BVH implementations. As such, we use it for BVH building and traversal. Our findings are mostly renderer-independent and can be reproduced with an Embree implementation. This allows us to use our own renderer [37]. 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, subsequent rays are not. Coherent rays result in similar accesses and traversal orders of the acceleration structures. While it is possible to optimize for coherent rays, this is not the purpose of our study.

TABLE 1
Embree parameters. Branching Factor has 2, 4 and 8 as possible values and is hardware dependent because traversal can be optimized by AVX. `intcost` and `travcost` are SAH-related parameters and are real numbers between 0 and 10.

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

3.2 Parameters and Indicators

In this section we discuss the choice of the acceleration structure. We further detail its parameters and scene indicators which represent context specific information such as camera position or triangle count.

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 which we summarize in Table 1. These parameters affect the construction of the BVH, and they are interdependent. For instance, reducing the number of triangles per leaf (`min/max leaf size`) is linked to the maximum depth (`maxdepth`). 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 ignored.

The tuner modifies these eleven parameters to optimize a metric based on rendering and build time. For our analysis we further require a starting configuration. As we want to compare to expert knowledge, we use the configuration recommended by Embree.

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 (triangularly tessellated surfaces) and the number of light sources.

Scenes with the same number of triangles can have different optimal configurations. Triangle sizes vary depending on the meshes in the scene, as walls for instance typically have large triangles while detailed objects have smaller ones. In addition, triangle sizes can vary within a mesh and BVHs are susceptible to this variation. To take this into account, we compute the variance of the triangle sizes for each mesh and name this set VarTriSize. As we cannot afford to have one indicator per mesh (more indicators hinder model learning), we compute the average and variance of VarTriSize. Additionally, we compute the average area per

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

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}

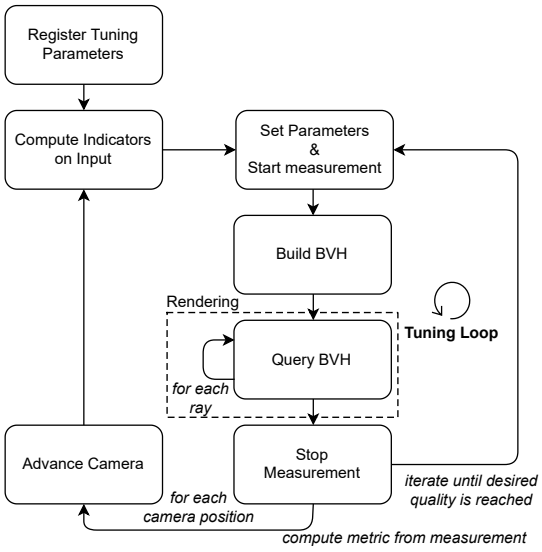

Fig. 1. **The tuning scenario**. Integration of the tuner in the ray tracing workflow.

mesh and the total area of the scene. For scaling reasons, we divide 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, hinting on potentially more coherent rays. We also added camera parameters: position, orientation, and the vertical field of view.

3.3 Tuner integration

Integrating any existing tuner in the ray tracing code is straightforward. It is a simple loop in which the tuner chooses parameters for the acceleration structure based on the feedback computed with rendering time and building time. Note that the classic tuning loop and the hybrid tuning loop differ only by the use of indicators. Figure 1 provides an overview of the tuning scenario. Before the loop, we measure the indicators and register the parameters in the autotuner. Every frame follows these four steps:

- Recording of the indicators,
- AS building with current parameters
- Rendering
- Recording of the optimization metrics as feedback,
- Updating current parameters with the values computed by the autotuner

The optimization metric is a lever to affect the tuner behavior, as shown on `sponza` in Figure 2. We first observe the initial performance degradation caused by rebuilding the BVH with subpar configurations 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 optimization metric affects the configuration found, and hence performance. When tuning rendering time, the tuning configuration outperforms the default one by 18% 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 14%, while the configuration found by optimizing for total time performs 4% better than recommended configuration. However, long convergence and wildly varying render time are still a major problem for online-autotuning.

This exploration data - the indicators, parameters, and corresponding performance - is learned by our hybrid tuner to compute a model giving us efficient configurations that we can then query at runtime, which we call exploitation. During exploitation, no search is performed. Instead, the learned model predicts a configuration based on the current indicators.

4 Parameter Analysis

The previously described parameters each have their influence on the rendering time or the acceleration structure build time. This section will first present each parameter's influence on the timings. We then describe the interaction between these parameters using the `sponza` scene as a support for the analysis. We finally compare the behavior of the building algorithms and especially how it handles more complex animated geometry setups.

Note that the analysis has been conducted on all scenes, see Figure 3, and the full results are presented in the supplemental. We provide the full data as well as the corresponding R scripts to generate graphs.

4.1 Correlation of random configurations

To explore the parameter space, we use a correlation matrix as a guide for our analysis, computed between `BVH` parameters and timings, the one for the `sponza` scene is shown in Figure 4. This matrix shows how the change in one parameter affects timings. Because we use random sampling, all parameters show 0 correlation with other parameters. Full correlation matrices for other scenes are available in the supplemental. The parameters `quality` and `compact` play an important role in the behavior of the acceleration structure according to their 57% and -58% of correlation with rendering time. We note that `min/maxLeafSize` have

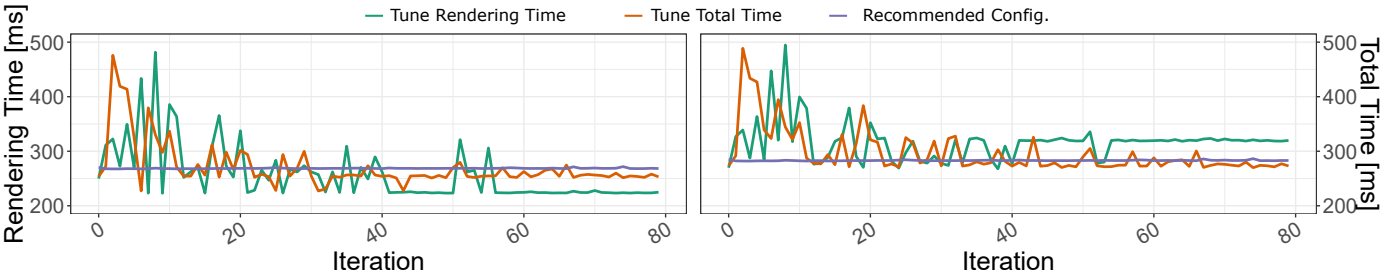

Fig. 2. Changing the feedback function drastically changes the tuning results. Green optimizes the rendering time only, whereas Red optimizes the total time of rendering and building. Violet is the recommended configuration by Embree. Scene: Sponza

Fig. 3. Image of the scenes: **fireplace_room, living_room, bmv, sponza, buddha, gallery, and vokselia**

Fig. 4. **Correlation matrix sponza (2/11 rows).** All non displayed rows are blank because of random sampling, except for `min-maxLeafSize` that are trivially correlated (min<max). `quality`, `min-maxLeafSize`, `compact` show high correlation. `sahBlockSize` and `intcost/travcost` present small correlation. (removed discussion about p-values)

decent impact with 29% of correlation with rendering time. We will see that these parameters have a non-negligible effect on the AS build time as well. `sahBlockSize` and `intcost/travcost` show small correlation with the rendering and AS build time, but we will show that the low correlation reflects their complex interaction.

The `compact` parameter entirely disables all the others. It is a specialized implementation optimized for memory consumption. As this is not the focus of the paper, we assume `compact` being disabled in the following.

The `quality` parameter enables or disables triangle splitting. It impacts AS build time and rendering in both value and variance. Triangle splitting creates a better AS, improving rendering time at the cost of build time. The efficiency of this feature depends on the application-specific rendering workload. To illustrate this, we compare two common screen resolutions on the `sponza` scene. The difference of rendering load between 480p and 720p is 432000 pixels. The total time speedup achieved by choosing to split

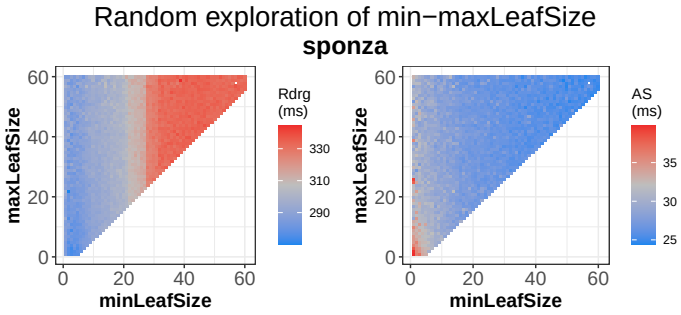

Fig. 5. **Min Max Leaf Size.** Performance distribution with splitting disabled. We observe a similar trend in every scene: small leaf size is preferable for rendering, big leaf size for AS build time.

triangles is -5.8% for 480p and $+8\%$ for 720p. It scales very rapidly as the rendering load increases.

Constraining the number of primitives per leaves in the BVH tree is possible using `min-maxLeafSize`. It impacts mostly rendering time, but its effect on AS build time in the case of complex scenes stays relevant. Because triangle splitting disables control over `minLeafSize` and `maxLeafSize`, it is disabled to study these two parameters. Figure 5 shows the distribution of average rendering time and average build time, with `minLeafSize` and `maxLeafSize` in the horizontal and vertical axis. The triangular shape results from `minLeafSize` $<$ `maxLeafSize`. AS build time and rendering time present a clear tradeoff: build time is faster with large leaves while rendering time is faster with small leaves. The optimal point is once again input dependent. Because the tradeoff is very clear despite the data being an average over many different configurations, we deduce that these two parameters offer consistent control in every circumstance.

The `sahBlockSize` parameter is related to hardware vectorization (SSE, AVX) optimizations. The API of Embree provides a range between 1 and 32, and we treat it as such even though only steps of powers of 2 impact the performance significantly. The parameter value matching the CPU specification, 4 in our case, tends to improve rendering time; and larger values tend to improve build time.

The SAH metric computation requires the weights defined by `intcost/travcost`. Their absolute correlation (between each other) is very low due to the need to find both parameters together in quite a small range of values to obtain a good configuration. They are interesting for tuning, as we will see in the next section.

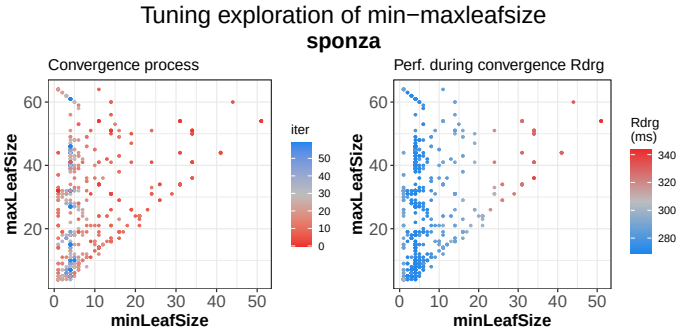

Fig. 6. **Optimization Process and Performance** *Left*: Tuner iterations. *Right*: Evolution of the performance across iterations for multiple runs. The tuner converged indeed toward the expected values from Figure 5. Looking at the area of the position of left side blue dots in the right plot gives the converged performance.

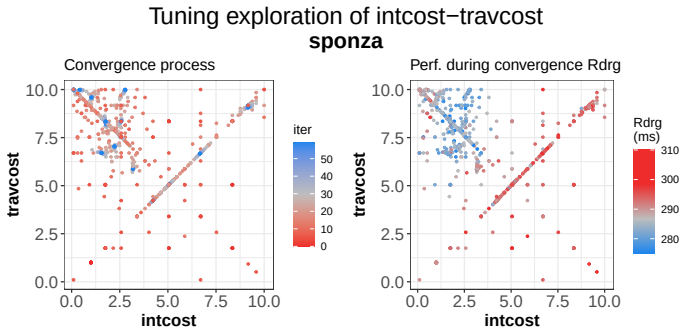

Fig. 7. **Optimization Process and Performance** *Left*: Iterations of the tuner. *Right*: Evolution of the performance accross iterations. Optima are bundled around top left corner, indicating shallow tree construction.

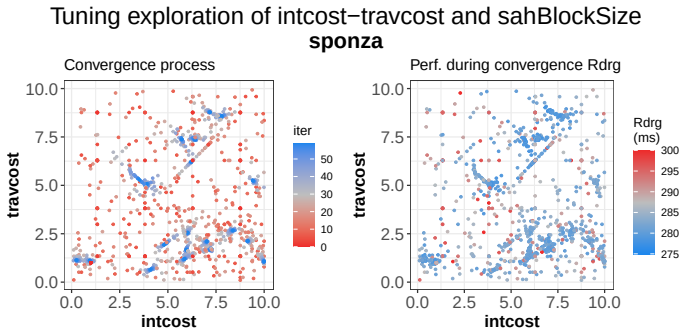

Fig. 8. **Optimization Process and Performance** *Left*: Iterations of the tuner. *Right*: Evolution of performance accross iterations. Optima are more scattered but consistently in the bottom right corner. Distribution is changed now that `sahBlockSize` is tuned as well (not represented).

4.2 Partial Tuning

Parameters can have non-trivial inter-relationships difficult to grasp using random sampling, even with high sample counts. By tuning a small subset of parameters only, we can better understand their coupling. In this section, we study the interplay of the following parameters:

- `minLeafSize` with `maxLeafSize`
- `intcost` with `travcost`
- `intcost` with `travcost` and `sahBlockSize`.

These parameters have been chosen because of their higher influence on performance and high degree of interdependency (see Figure 4). For this experiment, we allow the tuner to only tune the said parameters, the others are fixed to their default value.

Tuning only `min-maxLeafSize` improves rendering time up to 8% compared to the recommended configuration. The left side of Figure 6 shows the convergence process toward a small subset of points, indicating consistent behavior, and on the right the associated performance. For `sponza`, the optimum leaf size is 4 to 7, while it is between 25 and 30 for `vokselia`. The tuner appears to indeed converge toward the values found in Figure 5.

The SAH metric uses both `intcost/travcost`. Figure 7 shows the same data as previously, but with `intcost/travcost` instead of `min-maxLeafSize`. High traversal cost and small intersection cost seem to be favored. A high traversal cost makes the builder create shallow trees, which improves AS build performance but decreases rendering performance. The visible presence of diagonals shows the optimization path taken by the Nelder-Mead algorithm used by the tuner. Here, `sahBlockSize` is still set to 1. We will now see how this parameter changes the distribution.

The modification of the vectorization parameter influences the optimum of some other parameters, mostly the SAH costs. We study the triplet of parameters `intcost/travcost/sahBlockSize`. Figure 8 represents data the same way as the previous plots, but here `sahBlockSize` is tuned as well. `sahBlockSize` is not represented because its converged value is always the hardware optimum when the optimization metric is rendering time. It only changes the distribution of the other parameters. The same experiment yields different clusters of convergence for each scene. The density of samples in the cluster regions is very high, indicating a quick convergence, but more spread-out local optima. The scene `vokselia` shows more similarities with the previous graph than the other scenes. It seems more efficient for this scene to still construct a shallow tree. The construction of shallow trees (with leaves containing many primitives) is allowed by the default values of Embree: `minLeafSize` $= 1$ and `maxLeafSize` $= 32$. If these two values can change as well, the resulting performance and convergence is expected to change.

Up until now, the scenes were tested as is. Adding difficult geometry will challenge the building algorithm and provide insights on dynamic effects and limits of the AS itself.

4.3 Build time stability

The stability and reproducibility seemingly provided by triangle splitting will be put in perspective in this section. To expose the behavior of the building algorithm with complicated cases, we use a rotating cylinder containing long and thin triangles, intersecting the geometry of the scene. The cylinder is half as long as the diagonal of the scene. These triangles will put the building algorithm under heavy load, allowing us to evaluate how it handles hard cases. Figure 9 shows the performance during the rotation of the cylinder. The confidence interval at 99.99% is displayed as a greyed area around the average sampled value.

All scenes show improved rendering performance with splitting, but a significantly worse build performance: more than doubled for all scenes. The variability of the rendering time — or the vertical spread — is also very high as shown

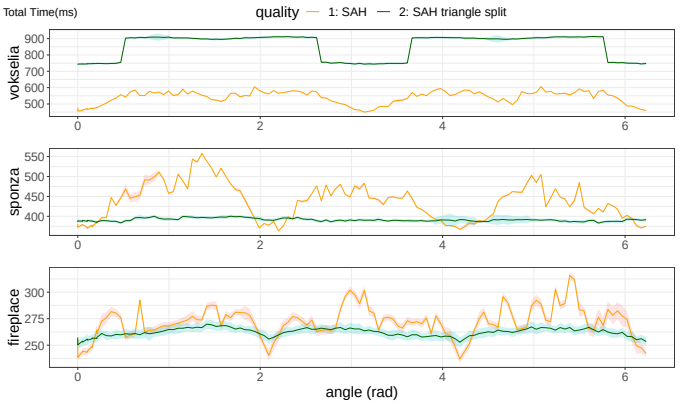

Fig. 9. **Evolution of performance depending on triangle splitting along a rotation of the cylinder.** Each graph is one of three scenarios: when splitting is *not* worth it: `vokselia`, when it is: `sponza`, and when it is about the same: `fireplace_room`. Note the difference in rendering time variance in each scene and among scenes.

particularly clearly by `sponza` where rendering time spans almost $175\,\mathrm{ms}$ which is 50% of variation around the average value.

`vokselia`'s build time in Figure 9 increases of 26%: from $560\,\mathrm{ms}$ without splitting to $710\,\mathrm{ms}$ with triangle splitting. The cylinder starts intersecting a significant portion of the underground geometry, causing a large amount of triangle splits, which is made apparent by the sharp increase in total time with triangle splits.

4.4 Observations

The analysis exposes certain behaviors of acceleration structures in general. This section summarizes the findings for different use cases.

When only rendering time is important, triangle splitting offers good performance and a more consistent average. Triangle splitting is tempting but is to be used only if the input is carefully controlled, without long, thin, and intersecting triangles because the cost can ramp up very quickly and not be amortized by the rendering.

When the build time is also important, disabling the triangle splitting is a significantly more versatile setting. It allows for fine adjustments on different trade-offs, i.e., smaller leaf size tends to reduce rendering time; $4-10$ seem optimal in our test cases. To reduce build time, `minLeafSize` can be increased. A value between 20 to 30 offers a good reduction in build time.

Unsurprisingly, `sahBlockSize` will perform best for rendering time at the corresponding hardware value. It is 2 for SSE, 4 for AVX2, 8 for AVX512, 32 for GPUs (warp size). Taking larger values than the one of the hardware will increase rendering time and decrease build time.

Thorough analysis of scenes is of course not common, and dynamic scenes require even more care. In the next sections, we expose how we create prediction models able to find near optimal configuration even in dynamic context.

5 From Analysis to Practical

Using the insights of Section 4, we can refine the autotuning process in several ways. We can set smaller intervals for the values of each parameters. These intervals can depend on the optimization target, i.e., rendering time, total time, or other relevant metrics. Reducing the range of the parameters allows a significant improvement of the worst configuration sampled during optimization. The exact improvement highly depends on the optimization method (for us: Nelder-Mead). Since this is an interchangeable part of our method, we refer to the literature on parameter sensitivity [38]. Another optimization is fixing certain parameters to values close to optimal values. This is easier for parameters related to hardware, such as `sahBlockSize`.

Applying the above suggestions helps in creating relevant indicators. Indeed, less but carefully chosen parameters imply a clearer relationship between them and indicators. For our hybrid online autotuning method, we will only restrict the parameters to algorithmically valid values and will keep all parameters including the most irrelevant. Indeed, these optimizations are not compulsory. We wish to keep a more general approach since optimizations are use-case specific. Our goal is to show that our method is robust enough to still produce encouraging results, and this shows that there is much to be gained with more efforts towards engineering indicators and reducing parameter space depending on the use case.

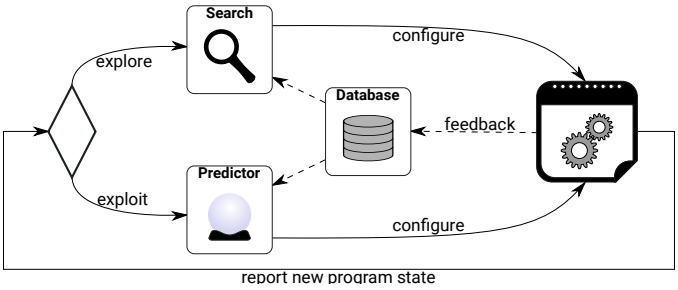

Fig. 10. The hybrid online tuning workflow.

6 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 is generated in the deployment context. The main question 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 10. The tuner observes the program and the system state through the indicators. They serve as an approximation of the program and system state in the remainder of the paper. We therefore use the term "state" to refer to the current indicator values. For every change in 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 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) [39]. To implement the decision process, we apply 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 [40]. Our version is similar to that of Chang [41], using Latin Hypercube sampling [42] as an initialization method.

We investigate two different types of models for the exploitation 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.

6.1 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. This solution is also quite limited: the state space is practically unbounded and too large to store let alone to explore it in its entirety. Even if it could be stored, we must assume that most states would not be in the table. 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 the distance metric (Euclidean in our case) in the state space.

6.2 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 [39]. A recent such algorithm is Greedy-GQ [43]. 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. That allows us to learn from offline data (e.g., from past experiments or tuning searches) but also during online search. Additionally, Greedy-GQ supports incremental online training and imposes no limits on the features we can use to represent the model inputs. Features are high-dimensional functions, in our case Gaussians, representing how parameters and indicators relate with performance. More features mean we can more finely describe this relation, but we also risk overfitting. Both the memory and runtime complexity of the Greedy-GQ algorithm are 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 $Q_\theta(s,a) = \vec{\theta} \cdot \varphi(s,a)$ where $\varphi(s,a)$ is a vector of real-valued features, and $\vec{\theta}$ are the coefficients to be learned. This function associates a scalar value with every state s and "action" a. Intuitively, this value estimates the benefit (higher is better) of choosing the action a in state s. An action corresponds to one of the configurations investigated by the tuning search. We thus use the terms interchangeably here. During exploitation, the predictor selects the action that maximizes Q_θ for the current state. When a state-action pair s_t, a_t is evaluated in search mode, the coefficients are updated based on the "reward" R_{t+1} observed by the application, which is the inverse of the runtime measure for the configuration a_t.

The update equations for $\vec{\theta}$ are controlled by three hyper-parameters: α and 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 possible future states. 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" [39]. 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 [44]. A radial basis function (RBF) has an N-dimensional Gaussian $\varphi_i(x) = \exp(-\omega_i||x - c_i||)$, 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 scales of the indicators vary drastically. Because the distance metric is scale-variant, indicators spanning larger scales will carry more weight in the distance computation than smaller ones. We thus need to normalize parameters and indicators to a comparable scale before passing them to the RBF model. 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 based on the data. This preprocess is necessary to compute our features' parameters, c_i in our case. We choose our upper bounds to be on the above order

of magnitude of the largest value found in our data set, e.g., the largest evaluation scene has 1.8 million triangles, we chose 10 million as our virtual upper bound.

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

7 Evaluation

In this section we present the evaluation of our method. We compare hybrid autotuning to classical continuous online-autotuning. The ϵ-Greedy policy used to choose between exploration and exploitation is purely stochastic. Therefore, we are able to evaluate the performance in production from the two behaviors in isolation.

We benchmark on seven scenes with differing complexities and characteristics. The scenes are `sponza`, `gallery`, `vokselia_spawn`, `conference`, `fireplace_room`, `buddha` [45] and `bmw-m6` (which we call `car`) [46]. These scenes have been chosen for their wide variety of geometry. Some of them can be seen in Figure 3.

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, the acceleration structure is used for a single rendering step only and rebuilt thereafter. Of course, real-time systems often have a complex datastructure management system, and this is a simplification. 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 minimize the sum of build and rendering time in this case.

7.1 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 shown in Figure 11: we show the typical runtime behavior as observed during the tuning process for one of the camera positions of the `car` scene. The jagged black curve shows the measured runtime during every iteration of a Nelder-Mead search. The purple dashed line represents the tuning curve (roofline) of an ideal predictive autotuner: using the best configuration from the baseline. 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. In the following sections, we quantify how much worse the prediction result is. The primary goal of the predictor is not

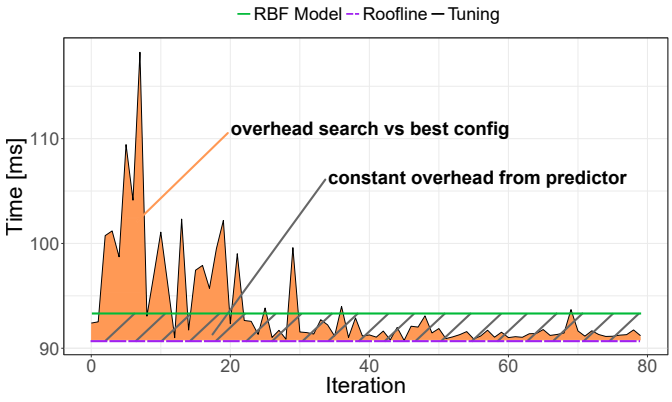

Fig. 11. Example: Tuning on the `car` scene. The orange area shows the total overhead of 80 iterations over the best configuration (purple dashed). The hashed area shows the difference between our predictor (green) and an ideal predictor (purple dashed).

to produce a configuration as good as the roofline or the tuning search. Instead, its purpose is to reduce the total time spent searching. Progressive rendering accumulates each subsequent iteration to improve quality. For our evaluation, we want to improve the time required to compute a fixed number of frames. The time required to complete 80 frames, or iterations, in the example in Figure 11 is the area under the respective curves. The area highlighted in orange shows the performance of the autotuner. The hashed area in turn shows the predictor performance. 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.

7.1.1 Generating Training and Validation Data

We want to generate the baseline in order to train and evaluate our model. We need a variety of indicator states for the evaluation scenes, for which we generate camera positions throughout the scenes. The camera path is designed on each scene to explore different amount of visible geometry and more or less complex lighting environments. We create 100 camera 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 sample per pixel. Given seven scenes, 100 cameras and 80 iterations, we obtain 56000 data samples.

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. It maps indicator states to the optimal configuration from the baseline found for this state. 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 6.2 we obtain the linear combination of RBF features.

For the verification phase we evaluate the predictor on the 20 remaining camera positions. 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 exploration of the parameter space could find that configuration. However, since we are only comparing search-based and predictive autotuning here, we believe that our comparison is sufficient.

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.

7.1.2 Optimizing Render Time

We first analyze the behavior of our predictor for progressive rendering, which means only rendering time is to be minimized. In Figure 12 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 the last iteration of each search is considered by the predictor, assuming the search converged.

The speedup results show that both the nearest neighbor and the RBF model predictor outperform the baseline in most cases. On average, using the 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 also compare the overheads against a virtual baseline, which is the best-known configuration for a camera and scene. Unlike for the roofline comparison done above, 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% (`sponza` scene) using the nearest-neighbor predictor. The RBF predictor reduces only up to 79.2% of the overhead (`gallery` scene). Although the speedups appear to be small, the presentation hides that we are comparing only render time. 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 for fairness.

In Figure 13 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

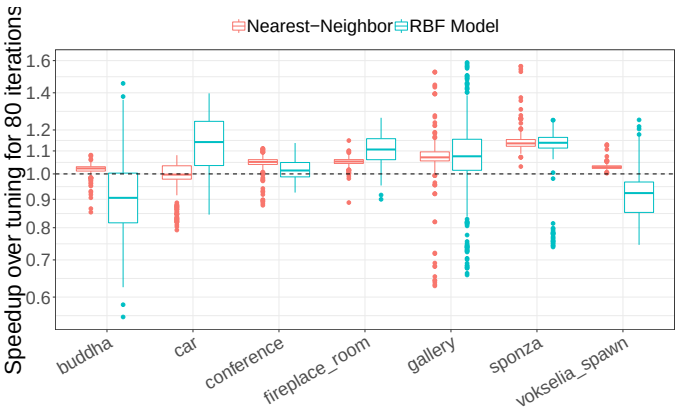

Fig. 12. Speedups of our predictor over search-based tuning. Both search and predictor minimize rendering time only.

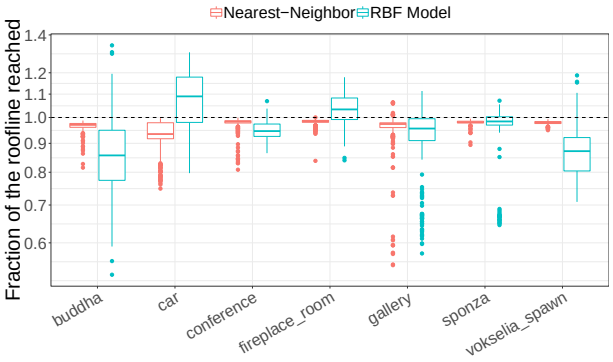

Fig. 13. Comparison of the predictors against the rooflines for every scene and camera position. Both baseline and predictor minimize rendering time only.

algorithm, which only converges to local optima. In Figures 12 and 13 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 increase of only five percent one may ask: Why choose prediction over search? The answer lies in the overhead of the baseline. We present another view on the search and prediction comparison in Figure 14. The graph shows the number of tuning iterations required for the search to outperform the prediction. The graph shows the number of iterations at which the cumulative search time and cumulative prediction time cross. For the RBF model the average break-even point is 362, and 1771 for nearest neighbor. On average at least 362 iterations are necessary to observe a benefit from the configuration found by the search. 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 yield 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.

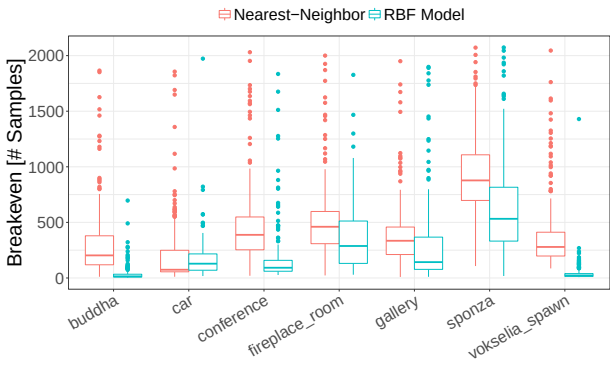

Fig. 14. Break-even points of the Nelder-Mead search. A substantial number of Nelder-Mead iterations is required to break-even with predicted configurations.

7.1.3 Optimizing Render and Build Time

Next, we evaluate the behavior of our predictors in a real-time raytracing context. In this use case the BVH must be rebuilt for every frame. The feedback function of the tuner measures the sum of rendering time and building time in this scenario. Figure 15 shows that the nearest neighbor approach still outperforms the baseline in most scenes. The only 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 16 we compare our predictors to the roofline for every scene and camera position. Recall that the roofline is 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 model shows greater variance and slower performing averages with 83%. We note that the RBF model tends to cross the roofline more often than the nearest neighbor variant. This is due to the RBF model finding a better local optimum in configurations that do not appear in the nearest neighbor search. In other words, the model sometimes outperforms the Nelder-Mead process.

8 Conclusion

In this paper, we addressed two major caveats encountered with acceleration structure autotuning. We explored the context sensitivity of their parameters and showed that while some parameters can be easily tweaked by an experienced engineer, some other parameters, such as the SAH costs, require automated optimization schemes. We showed that triangle splitting offers low variance for both rendering and building time. Yet this stability is only apparent. When under load, the performance drop in the case of animations can be considerable: 26% in our test scene.

Selecting or constraining the tuning parameters will reduce the parameter space and improve the convergence of the autotuner. The exploration, improved or not, stays

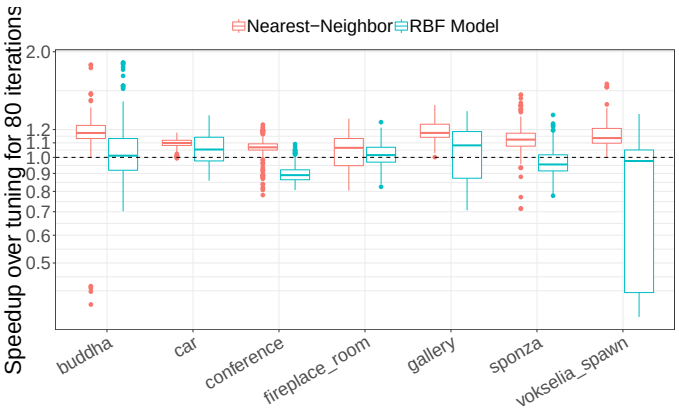

Fig. 15. Speedups of our predictor over search-based tuning. Both search and predictor minimize rendering time and building time.

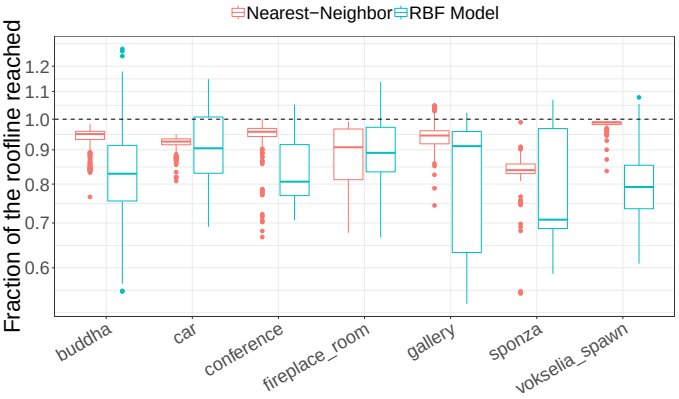

Fig. 16. Comparison of the predictors against the rooflines for every scene and camera position. Both baseline and predictor minimize rendering time and building time.

relatively expensive within the constraints of ray tracing applications, specifically real-time ones. The Hybrid Online Autotuning method further alleviates this caveat. Our model-based prediction does not need a sampling scheme, eliminating the need to rebuild acceleration structures. Combining both search and model approaches is a good balance to mitigate the downsides of each strategy.

Our evaluation on seven scenes of varying complexity shows that our predictors can compensate for the overhead of the search-based tuning. A nearest-neighbor method achieves 95% of the performance offered by the search-based tuning while reducing the overhead by 90% of the rendering time. Because the nearest-neighbor predictor requires maintaining large input state tables, we also investigated a function approximation approach. We trained a radial basis function model 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. Hierarchical optimization or per node optimization is promising and future work in this direction is required.

Acknowledgments

We thank Timo Kopf, Kevin Zerr and André Wengert whose Master's theses investigated early designs of Hybrid Tun-

ing. This project was funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project 299215159 with Grants DA 1200/5-1 and TI 264/10-1.

REFERENCES

[1] E. Haines and T. Akenine-Möller, *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, 2019.

[2] T. L. Kay and J. T. Kajiya, "Ray Tracing Complex Scenes," in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, 1986.

[3] M. Tillmann, P. Pfaffe, C. Kaag, and W. F. Tichy, "Online-Autotuning of Parallel SAH kD-Trees," in *IEEE Parallel and Distributed Processing Symposium*, 2016.

[4] V. Havran and J. Bittner, "On improving kd-trees for ray shooting," *Journal of WSCG*, vol. 10, pp. 209–216, 2002.

[5] M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies," in *Proceedings of the 1st ACM Conference on High Performance Graphics*, 2009.

[6] M. Vinkler, V. Havran, and J. Sochor, "Visibility driven BVH build up algorithm for ray tracing," *Computers & Graphics*, vol. 36, 2012.

[7] H. Dammertz, J. Hanika, and A. Keller, "Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays," in *Proceedings of the 19th Eurographics Conference on Rendering*, 2008.

[8] J. Bittner, M. Hapala, and V. Havran, "Incremental BVH construction for ray tracing," *Computers & Graphics*, vol. 47, pp. 135–144, 2015.

[9] D. Wodniok and M. Goesele, "Construction of bounding volume hierarchies with SAH cost approximation on temporary subtrees," *Computers & Graphics*, vol. 62, 2017.

[10] D. Meister and J. Bittner, "Parallel Reinsertion for Bounding Volume Hierarchy Optimization," *Computer Graphics Forum*, vol. 37, 2018.

[11] P. Ganestam and M. Doggett, "Auto-tuning Interactive Ray Tracing Using an Analytical GPU Architecture Model," in *Proceedings of the Annual Workshop on General Purpose Processing with Graphics Processing Units*, 2012.

[12] N. Weber and M. Goesele, "Auto-tuning Complex Array Layouts for GPUs," in *Proceedings of the 14th Eurographics Symposium on Parallel Graphics and Visualization*, 2014.

[13] J. Vorba, O. Karlík, M. Šik, T. Ritschel, and J. Křivánek, "Online learning of parametric mixture models for light transport simulation," *ACM Transactions on Graphics*, vol. 33, Jul. 2014.

[14] F. Simon, A. Jung, J. Hanika, and C. Dachsbacher, "Selective guided sampling with complete light transport paths," *Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, vol. 37, no. 6, Dec. 2018.

[15] K. Dahm and A. Keller, "Learning Light Transport the Reinforced Way," in *ACM SIGGRAPH Talks*, 2017.

[16] J. Liu, T. Dwyer, K. Marriott, J. Millar, and A. Haworth, "Understanding the Relationship Between Interactive Optimisation and Visual Analytics in the Context of Prostate Brachytherapy," *IEEE Trans Vis Comput Graph*, vol. 24, no. 1, pp. 319–329, Jan. 2018.

[17] J. Liu, T. Dwyer, G. Tack, S. Gratzl, and K. Marriott, "Supporting the problem-solving loop: Designing highly interactive optimisation systems," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 2, pp. 1764–1774, 2021.

[18] A. Chatzimparmpas, R. M. Martins, K. Kucher, and A. Kerren, "Visevol: Visual analytics to support hyperparameter search through evolutionary optimization," 2020.

[19] T. Li, G. Convertino, W. Wang, H. Most, T. Zajonc, and Y.-H. Tsai, "HyperTuner: Visual Analytics for Hyperparameter Tuning by Professionals," Oct. 2018.

[20] H. Park, Y. Nam, J.-H. Kim, and J. Choo, "Hypertendril: Visual analytics for user-driven hyperparameter optimization of deep neural networks," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 2, p. 1407–1416, Feb 2021. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2020.3030380

[21] R. C. Whaley and J. J. Dongarra, "Automatically Tuned Linear Algebra Software," in *IEEE/ACM Conference on Supercomputing*, 1998.

[22] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth, "Active Harmony: Towards Automated Performance Tuning," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, 2002.

[23] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: An extensible framework for program autotuning," in *Proceedings of the International Conference on Parallel Architectures and Compilation*. Association for Computing Machinery, 2014.

[24] B. Settles, "Active Learning Literature Survey," University of Wisconsin–Madison, Computer Sciences Technical Report 1648, 2009.

[25] P. Balaprakash, R. B. Gramacy, and S. M. Wild, "Active-learning-based surrogate models for empirical performance tuning," in *IEEE International Conference on Cluster Computing*, 2013.

[26] W. Bao, C. Hong, S. Chunduri, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, and P. Sadayappan, "Static and Dynamic Frequency Scaling on Multicore CPUs," *ACM Trans. Archit. Code Optim.*, 2016.

[27] J. T. Kajiya, "The Rendering Equation," in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, 1986.

[28] P. Christensen, J. Fong, J. Shade, W. Wooten, and B. Schubert, "RenderMan: An Advanced Path-Tracing Architecture for Movie Rendering," *ACM Transactions on Graphics*, vol. 37, 2018.

[29] B. Burley, D. Adler, M. J.-Y. Chiang, H. Driskill, R. Habel, P. Kelly, P. Kutz, Y. K. Li, and D. Teece, "The Design and Evolution of Disney's Hyperion Renderer," *ACM Transactions on Graphics*, vol. 37, 2018.

[30] J. Novák, V. Havran, and C. Dachsbacher, "Path Regeneration for Interactive Path Tracing," in *Eurographics*, 2010.

[31] Y. Tokuyoshi and S. Ogaki, "Real-time Bidirectional Path Tracing via Rasterization," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2012.

[32] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, and J. Jeffers, "OSPRay - A CPU Ray Tracing Framework for Scientific Visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, 2017.

[33] C. Schied, C. Peters, and C. Dachsbacher, "Gradient Estimation for Real-time Adaptive Temporal Filtering," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 2, pp. 1–16, 2018.

[34] C. Schied, M. Salvi, A. Kaplanyan, C. Wyman, A. Patney, C. R. A. Chaitanya, J. Burgess, S. Liu, C. Dachsbacher, and A. Lefohn, "Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination," in *Proceedings of High Performance Graphics on - HPG '17*. ACM Press, 2017, pp. 1–12.

[35] C. R. A. Chaitanya, A. S. Kaplanyan, C. Schied, M. Salvi, A. Lefohn, D. Nowrouzezahrai, and T. Aila, "Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder," *ACM Transactions on Graphics*, vol. 36, pp. 1–12, 2017.

[36] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst, "Embree: A Kernel Framework for Efficient CPU Ray Tracing," *ACM Transactions on Graphics*, vol. 33, 2014.

[37] H. Otsu. Lightmetrica3. [Online]. Available: https://github.com/lightmetrica/lightmetrica-v3

[38] P. Wang and T. Shoup, "Parameter sensitivity study of the Nelder–Mead Simplex Method," *Advances in Engineering Software*, vol. 42, pp. 529–533, Jul. 2011.

[39] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2018.

[40] J. A. Nelder and R. Mead, "A Simplex Method for Function Minimization," *The Computer Journal*, vol. 7, no. 4, 1965.

[41] K.-H. Chang, "Stochastic Nelder-Mead Simplex Method - A New Globally Convergent Direct Search Method for Simulation Optimization," *European Journal of Operational Research*, pp. 834–837, 2012.

[42] M. D. McKay, R. J. Beckman, and W. J. Conover, "A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code," *Technometrics*, vol. 21, no. 2, pp. 239–245, 1979.

[43] H. R. Maei, C. Szepesvári, S. Bhatnagar, and R. S. Sutton, "Toward Off-Policy Learning Control with Function Approximation," in *Proceedings of the 27th International Conference on Machine Learning*, 2010.

[44] R. Kretchmar and C. Anderson, "Comparison of CMACs and Radial Basis Functions for Local Function Approximators in Reinforcement Learning," in *Proceedings of International Conference on Neural Networks*, 1997.

[45] Morgan McGuire, "Computer Graphics Archive," 2017.

[46] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*, 3rd ed., 2017.