# Online-Autotuning of Parallel SAH kD-Trees

Martin Tillmann, Philip Pfaffe, Christopher Kaag, Walter F. Tichy
*Institute of Program Structures & Data Organization*
*Karlsruhe Institute of Technology*
*Karlsruhe, Germany*
*Email: {martin.tillmann, philip.pfaffe, walter.tichy}@kit.edu, christopher.kaag@partner.kit.edu*

***Abstract*—We explore the benefits of using online-autotuning to find an optimal configuration for the parallel construction of Surface Area Heuristic (SAH) kD-trees. Using a quickly converging autotuning mechanism, we achieve a significant performance improvement of up to 1.96x.**

**The SAH kD-tree is a spatial data structure and a fundamental tool in the domain of computer graphics and simulations. The parallel construction of these trees is influenced by several parameters, controlling various aspects of the algorithm. However, the parameter configurations advocated in the literature are hardly ever portable. To boost portability, we apply online-autotuning to four state-of-the-art variants of parallel kD-tree construction. We show that speedups over the variants' standard configurations are possible with low programmer effort. We further demonstrate the performance portability of our approach by evaluating performance on varying multicore platforms and both static and dynamic geometries.**

***Keywords*-spatial data structures; online-autotuning; parallel performance optimization;**

## I. Introduction

Spatial data structures are a fundamental tool in the domain of computer graphics and simulations, supporting fast range or nearest neighbor queries on multidimensional data. While accelerating many applications in this domain, the most prominent one may be ray tracing. Ray tracing is a technique for synthesizing photo-realistic images, but is also applied in radio or acoustic simulations. In its essence, this technique casts a ray onto a given scene and simulates its interaction with the environment, represented as a set of (geometric) primitives.

There is a large collection of data structures that have been proposed to accelerate the ray-primitive intersection computation. One of the most popular among them is the Surface Area Heuristic (SAH) kD-tree, which permits efficient parallel construction, a mandatory trait for interactive or real-time ray tracing. In recent years, several highly parallel algorithms for constructing kD-trees have been developed, both for multi- and many-core platforms (cf. e.g. [1]–[3]). While these works report on great performance gains through parallel processing, they share a common difficulty. The SAH – the heuristic controlling how geometric primitives are assigned kD-tree nodes – as well as the parallel implementation itself come with a set of parameters, whose optimal configuration is not only application and platform dependent but also dependent on input data and user interaction. Although the authors usually propose a "good" configuration for a given application, this suggestion is generally not portable and is often only an educated guess, such as using the number of hardware threads as the number of spawned parallel tasks. Hence, an optimal configuration needs to be identified manually to achieve maximum performance in constructing kD-trees, which is a tedious task. The problem is even exacerbated by the fact that the optimal configuration does not only depend on the hardware and the application, but also on input data and interactive user inputs, such as scenes of varying complexity or camera movement.

In this paper, we propose using online-autotuning to address this problem. Autotuning has proven to be a valuable tool to solve similar configuration problems across different domains [4]. Although originally invented for offline optimization, e.g. during install or deploy time, it has since also been applied in online tuning scenarios [5]–[7], which enables it to deal with the dynamics of application execution. We show that, using AtuneRT, which is an extension of the work in [5], we are able to find optimal parameter configurations for parallel SAH kD-tree construction with negligible programmer involvement and little runtime overhead. To demonstrate the effectiveness of the approach, we implement four state-of-the-art variants of parallel kD-tree construction from the relevant literature. Using AtuneRT, we then evaluate these implementations in a ray tracing scenario on different multicore platforms and on both static and dynamic scenes.

## II. Related Work

In the past four decades, kD-trees have been the subject of extensive research. Many methods exist for efficient construction of the tree, such as the $O(n \log n)$ algorithm by Wald and Havran [8] for ray tracing in static scenes or the method by Günther et al. [9] for animated scenes. Similar effort has been invested in inventing fast search strategies for answering queries on the data structure, for instance by Sample et al. [10], who introduce a fast search algorithm combining depth-first branch and bound search with an intelligent search ordering heuristic.

In more recent years, the focus has shifted towards exploiting the benefits of parallelism brought by ubiquitous parallel multicore processors. Prominent examples are Shevtsov et al. [1] and Choi et al. [2], who combine parallel precomputation of the upper tree levels with subsequent parallel processing of the lower levels via space subdivision. Following this trend, specialized parallel manycore platforms, such as GPUs, have gained a lot of attention. Wu et al. [3] and Danilewski et al. [11] compute kD-trees on GPUs in a top-down, breadth-first fashion. Roccia et al. [12] distribute work load across CPU and GPU depending on individual node sizes.

While all these works devise various optimizations for the aspects of using (SAH) kD-trees, they expose multiple parameters for which suitable values must be found. Although autotuning has proven to be a valuable tool to find optimal parametrizations in various domains (cf. e.g. [13]–[15]), the benefits of autotuning kD-trees have not been studied previously. Autotuning for spatial data structures has, to the best of our knowledge, only been subject to a single study: Ganestam and Dogget [16] employ model-based online-autotuning to optimize a GPU-based ray tracing application using the Bounding Volume Hierarchies (BVH) data structure towards a predefined performance goal. To that end, the rendering quality is adapted until a desired rendering performance is reached. This work differs from ours in two key aspects, as we maximize ray tracing rendering performance in a fixed quality setting, and find configurations using a search technique instead of a machine model, thus avoiding a machine model's inherent inaccuracy.

## III. Fundamentals

### A. Online-Autotuning with AtuneRT

First, we specify the autotuning problem. The different kD-tree implementations each come with a set of tunable parameters. A given algorithm $a$ contains several tuning parameters $\tau_{a,j}$. We define a *tuning parameter* $\tau_{a,j}$ as the set of valid values for this tunable parameter. In practice most tuning parameters are closed integer intervals. We can now define the search space $T_a$ for an algorithm $a$.

$$T_a = \tau_{a,0} \times \cdots \times \tau_{a,J}$$

A *configuration* $C_a$ is a valid assignment of the tuning parameters and corresponds to a point in the search space $T_a$. The execution time of an algorithm using a configuration $C_a$ is measured by the timing function $m_a$. In general the execution time is influenced by the employed hardware and the input data. We call these factors the measuring context $K$. To minimize the execution time we need to find the optimal configuration $C_{opt,a}$:

$$C_{opt,a} = \arg\min_{C_a} m_a(C_a, K)$$

```
1  void DoThings() {
2    unsigned N;
3    unsigned min = 1;
4    unsigned max = 32;
5    unsigned step = 1;
6    Tuner T;
7    T.RegisterParameter(&N, min, max, step);
8
9    while(thingsToDo) {
10     T.Start(); // Start Measurement
11
12     #pragma omp parallel for num_threads(N)
13     for (...) { DoThing(); }
14
15     T.Stop(); // Stop Measurement
16   }
17 }
```

Figure 1: Example implementation using *AtuneRT*

Online-Autotuning searches at runtime for $C_{opt,a}$, this means that each evaluation of $m_a$ on a non-optimal configuration has to amortize itself. In conjunction with the sizeable extent of the search space $T_a$, a fast convergence of the search algorithm used to find $C_{opt,a}$ is essential.

In this work, we use the general-purpose online-autotuner *AtuneRT* (also used in [15]) which is the successor of *Atune* [17], an offline-autotuner. The search algorithm of *AtuneRT* samples the search space at random points to seed a Nelder-Mead search [18]. This approach performs well in cases where good starting configurations are unknown, and converges quickly even in high dimensional search spaces. *AtuneRT* is an application agnostic tool, and interoperates with client applications merely via shared memory and a small API. The client application tuning workflow is outlined in figure 1. Here, we wish to optimize the number of threads used in a parallel loop within a given range. More formally speaking, we wish to find an optimal configuration $C_{DoThings} \in T_{DoThings} = \tau_N$, with $\tau_N = [\texttt{min}, \texttt{max}]$ for the program variable `N`. Interaction with the tuner is implemented using its core API functions:

- `RegisterParameter(&N, min, max, step)` Register a variable `N` in memory for tuning within the range $[\texttt{min}, \texttt{max}]$ using a stride of `step`.
- `Start()` Start a new measurement cycle.
- `Stop()` End a measurement cycle and apply new configuration.

Integrating *AtuneRT* with client code is thus usually extremely simple, given that the desired pieces of the software that are to be tuned already provide explicit tuning parameters.

### B. SAH kD-Trees

Recursive space subdivision is the core concept of constructing kD-trees. The k-dimensional geometric primitives

are divided into two sets by a hyperplane, the resulting sets are then further subdivided until an end condition is reached. In the three dimensional use case the initial axis-aligned bounding box of the geometry is split by an axis-aligned plane. This method has two degrees of freedom: the selection of the plane and the end condition for stopping the subdivision. The Surface Area Heuristic (SAH) formalizes this problem and provides an approximation to construct *optimal* kD-trees. Optimal kD-trees minimize the time needed to intersect a ray with the geometry.

By subdividing a bounding box $b$ into two cuboids $l$ and $r$ we can compute the surface areas ($A(l)$, $A(r)$) and the number of geometric primitives contained in each cuboid ($N_l$, $N_r$). Assuming uniform distributions of rays, the probability $p$ of a ray, that intersects the original cuboid $b$, also intersecting a subdivision $b_{sub}$ is given by the ratio of their surface areas [8].

$$p(b_{sub}, b) = \frac{A(b_{sub})}{A(b)}$$

Given a bounding box $b$ and a subdividing plane $h$ we can now compute the cost of querying $b$.

$$cost(h, b) = C_T + p(l, b) \cdot cost(l) + p(r, b) \cdot cost(r)$$

Here $C_T$ is the fixed cost of traversing a tree node and $cost(l)$ and $cost(r)$ are the costs of finding an intersection in the corresponding sub-cuboids.

However geometric primitives commonly used in computer graphics such as triangles can end up intersecting the plane $h$. These triangles have to be duplicated and included in both subsets. This special case increases the number of triangles that are stored in the kD-tree which leads to additional costs. The SAH represents duplication by introducing the parameter $C_B$, the cost of duplicating a single primitive.

The cost functions of the subdivisions $l$ and $r$ can be over approximated by assuming that each contained primitive must be tested. If the cost to intersect a single primitive is $C_I$, we can write the SAH cost function as:

$$\begin{aligned} SAH(h, b) = C_T &+ p(l, b) \cdot N_l \cdot C_I \\ &+ p(r, b) \cdot N_r \cdot C_I \\ &+ (N_l + N_r - N_b) \cdot C_B \end{aligned} \tag{1}$$

By solving $\arg\min_h(SAH(h, b))$ we can now find the optimal hyperplane.

Additionally, this tells us when a bounding box $b$ should not be further subdivided. If the cost of intersecting all primitives is less than the optimal split cost further subdivision is not profitable.

$$\textbf{if } N_b \cdot C_I < \min_h(SAH_{split}(h, b)) \textbf{ stop} \tag{2}$$

As proposed the SAH relies on three parameters to perform its computations: the cost factors $C_T$, $C_I$ and $C_B$.

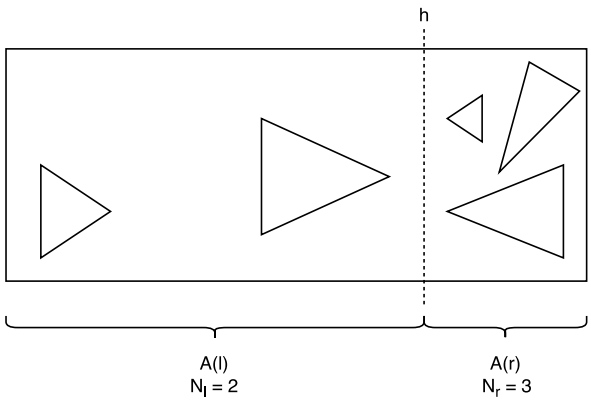



Figure 2: Two dimensional case of the Surface Area Heuristic (SAH). Triangles are separated into two subsets by line $h$. The relative surface area of the two rectangles $l$ and $r$ are used as probabilities for accessing the corresponding tree nodes. The cost of accessing a tree node is proportional to the number of contained triangles.

| *Parameter* | *Semantics* |
|---|---|
| $C_I$ | Cost for intersecting a triangle |
| $C_B$ | Cost for duplication of a primitive |
| $S$ | Max. number of subtrees per thread |

(a) Parameters of the node-level, nested and in-place algorithm implementation

| *Parameter* | *Semantics* |
|---|---|
| $C_I$ | Cost for intersecting a triangle |
| $C_B$ | Cost for duplication of a primitive |
| $S$ | Max. number of subtrees per thread |
| $R$ | minimal resolution of a node |

(b) Parameters of the lazy construction implementation

Table I: Tunable parameters of the implementations for parallel SAH kD-tree construction

Modern implementations and literature rely on expert knowledge and experience to choose these parameters. However as we show in section V-D different geometry or hardware produces different optimal values for these parameters.

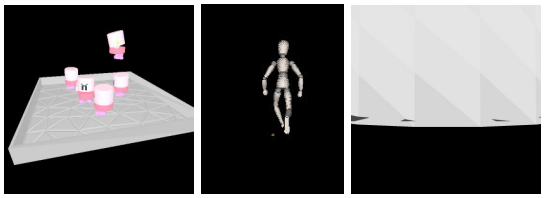
## IV. Implementation

Next we describe our parallel implementation of four state-of-the-art algorithms for kD-tree construction. Our implementations are based closely on the works of [2] and [8].

### A. Node-level Parallel Algorithm

In [8] Wald and Havran describe the best known sequential algorithm for constructing an SAH kD-tree. Although sequential, this algorithm allows for naïve parallelization, as for any inner tree node, the subtrees for its children may be constructed independently. We implement a parallel version

(a) Static scenes: Bunny, Sponza and Sibenik Cathedral



(b) Dynamic scenes: Toasters, Wood Doll and Fairy Forest (with most of the geometry covered)

Figure 3: Evaluation scenes

of this algorithm by spawning OpenMP tasks for every recursive call in the tree construction, up to a maximum depth.

The maximum depth is determined based on a configurable parameter $S$, defining the maximum number of subtrees to be expanded per thread. Together with the SAH parameters this procedure yields three tunable parameters for this implementation, listed in Table Ia. $C_I$ and $C_B$ are parameters of the SAH, estimating the cost of intersecting a triangle and a ray, respectively. As the configuration of $C_I$ and $C_B$ is only meaningful in relation to $C_T$ – the cost of visiting an inner node – we fix $C_T$ to an arbitrary value of 10.

### B. Nested Parallel Algorithm

The nested parallel algorithm presented in [2] is a more complex version of the node-level parallel version we described in section IV-A. In addition to handling independent subtrees in parallel in the same way as above, Choi et al. also parallelize the processing of geometry primitives inside individual nodes. To this end, this list of primitives is split into a number of chunks which are distributed across threads and then processed in what is essentially a sequence of parallel prefix operations. However, the interaction between prefix operations is in fact serialized. Because of this fork-join pattern, our implementation of the algorithm uses the OpenMP `parallel for` loop for parallelization.

As this algorithm is an extension of the node-level parallel version, it inherits its existing tuning parameters resulting in the set of parameters listed in table Ia.

### C. In-Place Parallel Algorithm

The in-place parallel algorithm is an alternative approach to constructing an SAH kD-tree [2]. Instead of creating nodes containing sets of triangles in a depth-first order, this algorithm analyzes the primitives in breadth-first fashion, keeping track of the nodes each triangle belongs to. This way, each level of the tree is constructed at once. The algorithm consists mainly of two operations. First, the maximum SAH value needs to be determined for every node of the current level and its associated primitives, which is implemented as a parallel prefix operation. We implement this as before, using OpenMP `parallel for`. Second, the next level of the tree needs to be created by assigning each triangle to its containing nodes. In this step, every triangle can be handled independently. Therefore, we again use OpenMP `parallel for` in our implementation.

This algorithm is controlled by the same parameters as the previous two, shown in table Ia.

### D. Lazy Construction Algorithm

To cope with the high memory consumption and strict response time requirements of kD-tree construction, state-of-the-art ray tracers and kD-tree builders oftentimes resort to a "lazy" or "on demand" construction strategy (cf. e.g. [19], [20]). This is especially beneficial when, for instance, rendering a scene with a high degree of occlusion, preventing a lot of primitives from ever being considered for intersection with a ray. For such situations it makes sense to only construct the spatial data structure to a certain level of detail, and only compute higher resolutions when and where needed.

We implement a lazy variation of the in-place parallel algorithm by introducing a parameter $R$ describing the minimal tree node resolution. The tree is constructed as before, parallelized across the primitives in the top-level nodes and across subtrees in the lower levels, but creation of subtrees is halted once the node contains less than $R$ primitives. A node will first be fully expanded once it is being hit by a ray during ray tracing. Because individual rays may be traced in parallel, we protect the deferred processing of a node with an OpenMP critical section.

Our implementation of the lazy construction algorithm inherits the parameters of the in-place algorithm, adding only the $R$ parameter. The full list is shown in table Ib.

## V. Evaluation

To test the performance of our kD-tree implementations, we evaluate them in the context of a ray tracing scenario, rendering six different 3D scenes. Because we wish to concentrate on the performance of the spatial data structure instead of the actual renderer, we chose a simplistic ray tracing technique, most commonly referred to as ray casting. Before showing and discussing our evaluation results, we briefly introduce our ray casting implementation and the evaluated rendering scenes.

### A. Ray Casting

Invented in 1968 [21], ray casting is a simple method of ray tracing image rendering. For every pixel in the synthesized image, a ray is sent towards the scene, finding the first intersecting geometry primitive. From this intersection point a shadow ray is cast to the light sources to determine the lights contribution to the resulting color. The rendered result is then the shaded color of the primitive. To find the first intersecting geometry primitive we query the kD-tree using a traversal implementation based on [22, pp. 319-321]. As the tree can be traversed independently for every ray, we parallelize intersection testing across different rays.

To optimize ray casting, our goal is to minimize the time to compute a given frame $t$. To compute a frame we have to build the kD-tree data structure from the input geometry and query this data structure for each ray cast in the rendering algorithm. Therefore, time $t$ is the sum of the construction time $t_c$ of the kD-tree and the rendering time $t_r$. Hence, the execution time function is simply:

$$m_a(C_a, K) = t_c(C_a, K) + t_r(C_a, K)$$

We use the tuning workflow seen in figure 4. After tuning parameters are registered we start measuring the execution time. We build the kD-tree according to the current configuration of the tuning parameters. The ray tracing algorithm queries the kD-tree for ray-triangle intersections. We stop measuring the execution time when the ray tracing is finished and advance the animation frame. An animation changes the geometry and necessitates a rebuild of the kD-tree from the previous frame. Most animations retain the general distribution of the geometry and therefore the optimal configuration only changes in small steps. Our tuning algorithm can react to these small shifts. This continued tuning has to amortize itself over the duration of the program. We evaluate static geometry with the same workflow. Although in static scenes it is not required to rebuilt the kD-tree for each frame, we still tune the construction instead of relying on offline-tuning. Camera positioning, system load and other environment effects all influence the optimal configuration and can differ each time the program is run.

### B. Rendering Scenes

Using ray casting, we evaluate the performance of the autotuning kD-tree implementations on six scenes, shown in figure 3. The set of scenes consists of three static scenes and three dynamic scenes. The static scenes (figure 3a) include Bunny (69666 triangles), Sponza (66450 triangles) and Sibenik Cathedral (75284 triangles). The dynamic scenes(3b) consist of Toasters (11141 triangles, 246 frames), Wood Doll (6658 triangles, 29 frames) and Fairy Forest (174117 triangles, 21 frames). The camera in the Fairy Forest scene is positioned up close to an object, most of the geometry in the scene is occluded. The cast rays intersect only with a tiny fraction of the scenes triangles, this constitutes a corner case for querying kD-trees.

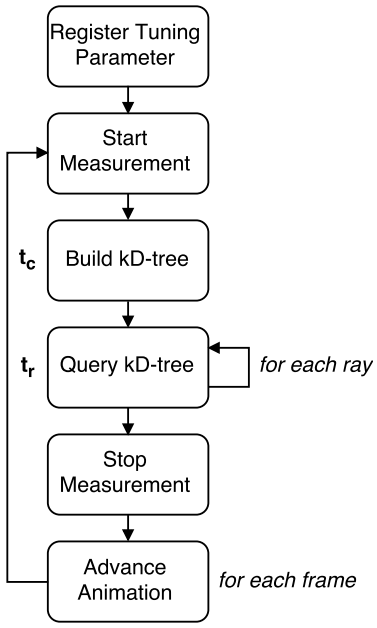



Figure 4: Ray Caster Tuning Workflow

| *Parameter* | *Range* |
|---|---|
| $\tau_{C_I}$ | $[3, 101]$ |
| $\tau_{C_B}$ | $[0, 60]$ |
| $\tau_S$ | $[1, 8]$ |
| $\tau_R$ | $[16, 8192]$ (limited to powers of 2) |

Table II: Tuning Parameters

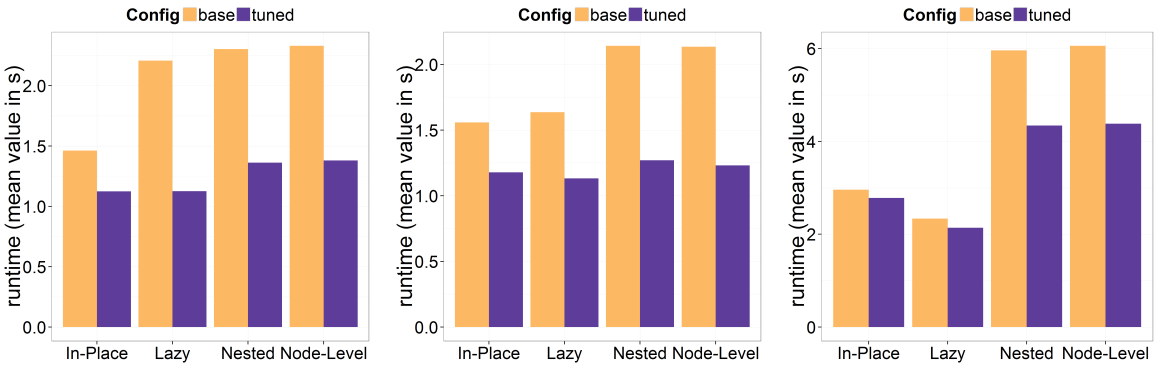
### C. Benchmark

We evaluate the ray tracing benchmark on an AMD Opteron 6168 24-core system (1.9 GHz). To assess the performance portability of our approach, we furthermore evaluated the benchmark for the In-Place algorithm on the Sibenik scene on three additional systems: An Intel Xeon E5-1620 quadcore (3.7 GHz, 8 threads), an Intel i7-4770K quad-core (3.5 GHz, 8 threads) and a mobile AMD A8-4500M quad-core (1.9 GHz).

A single experiment consists of constructing the kD-tree for each frame of a scene with the current parameter configuration and then rendering the image, using the autotuner to measure the total time required and to determine the next configuration. The tuning parameters are defined in table II. The search space $T$ is thus:

$$T = \tau_{C_I} \times \tau_{C_B} \times \tau_S \times \tau_R$$

Speedups presented in the following discussion are in relation to a manually crafted base configuration $C_{base} =$




(a) Sibenik (b) Sponza (c) Fairy Forest

Figure 5: Absolute execution time with and without tuning. Hardware: dual AMD Opteron 12-core system (1.9 GHz)

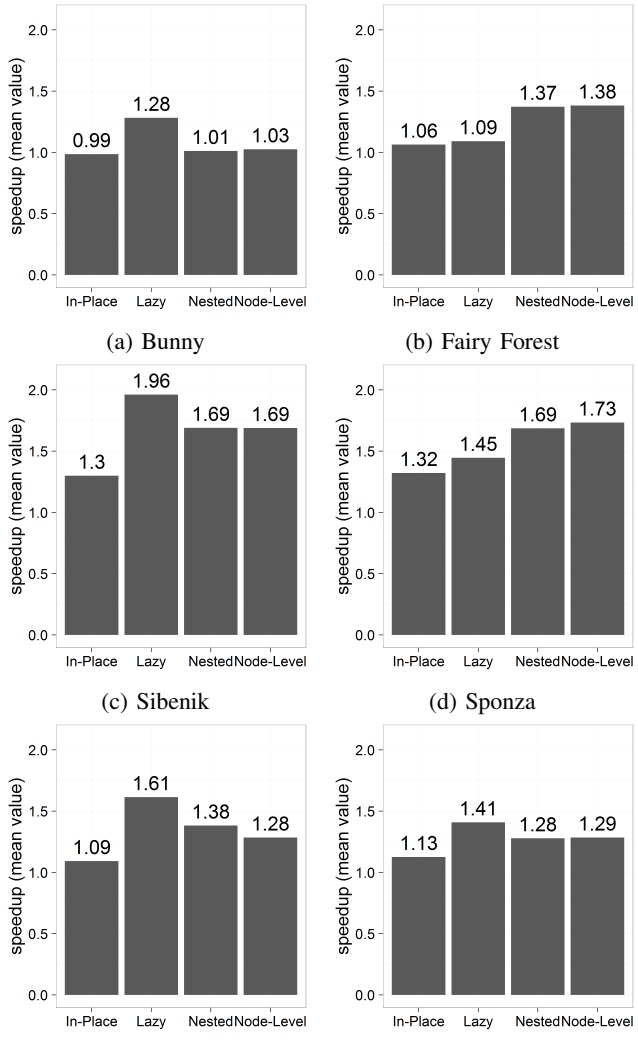
$(17, 10, 3, 2^{12}) \in T$, which is based on best practices and recommendations from literature, such as [2], [11].

In order to achieve optimal speedups, it is mandatory for the autotuner to reach convergence. Thus, for the static scenes, we repeat both steps until autotuning convergence has been reached. For the dynamic scenes on the other hand, the autotuner sometimes takes too long to reach a converged state before the frame sequence ends, which is why we artificially extend the sequence by repeating every frame 5 times. This experiment is repeated 15 times for every scene.

### D. Results

*1) Speedup Through Autotuning:* In figure 6 we see the speedup of the four tuned algorithms over all scenes on the Opteron dual 12-core platform. The speedup compares the execution time of the tuned configuration found by our autotuner $m_a(C_{tun,a}, K)$ to the base configuration $m_a(C_{base}, K)$. For the Bunny scene the in-place parallel algorithm (IV-C) is slower when tuned, resulting in a speedup of 0.99. On the Toasters and Fairy Forest scene the in-place parallel algorithm reaches a speedup of up to 1.09. Tuning the lazy construction algorithm (IV-D) on the Fairy Forest scene results in the same marginal speedup of 1.09. Both the node-level parallel algorithm (IV-A) and the nested parallel algorithm (IV-B) cannot be improved beyond a 1.03 speedup on the Bunny scene. All other combinations achieve satisfying speedups, the highest peaking at 1.96 with the lazy algorithm on Sibenik. Figure 5 details the absolute execution times of the four kD-tree implementations on three scenes. For comparison the execution time with the base configuration is shown next to the execution time with the tuned configuration found by our autotuner.

While the overall impact of tuning our selection of parameters is evident for most scene and algorithm combinations, local minima are a danger for the Nelder-Mead search. The tuning of the in-place parallel algorithm on the Bunny scene terminates in such a local minimum, in combination




(a) Bunny (b) Fairy Forest

(c) Sibenik (d) Sponza

(e) Toasters (f) Wood Doll

Figure 6: Speedup of the four tuned algorithms compared to their base configurations. Hardware: dual AMD Opteron 12-core system (1.9 GHz)

with the overhead from the online-autotuning this results in a slight slowdown. If the base configuration is already close to the optimal configuration we can see only marginal improvements. This is the case for the in-place parallel and lazy construction algorithms on Fairy Forest, and the node-level and nested parallel algorithms on the Bunny scene.

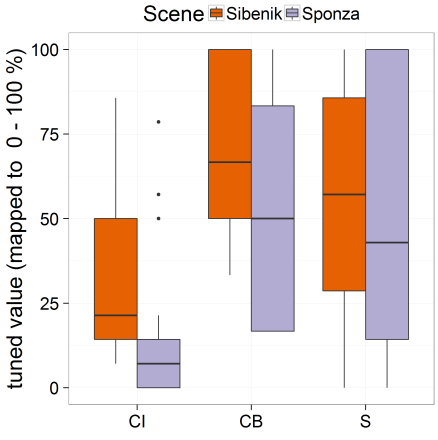
As seen in figure 5, the Lazy Construction algorithm performs well on the Fairy Forest scene where most of the geometry is occluded. Most tree nodes will not be expanded.

*2) Portability of Configurations:* To assess the ability of the online-autotuning approach to provide portable performance in the face of varying input data, figures 7a and 7b show a comparison of tuned parameters for the in-place algorithm on the Opteron platform for pairs of static and dynamic scenes. For improved legibility, parameter ranges have been normalized to $[0, 100]$. Outliers are shown as black dots. The diagrams show boxplots for the tuned values of all parameters and it is clearly visible that tuned configurations for different scenes fall into rather different ranges. We see that, for different input, tuned configurations are strikingly different and are thus not portable.

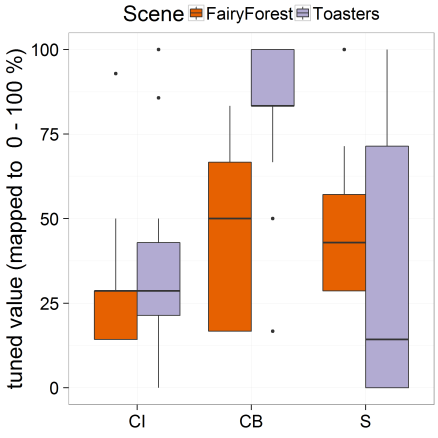
Figure 7c shows a similar comparison for four different hardware platforms for the in-place algorithm on the Sibenik scene. We again see a great difference for the varying platforms, indicating that once again tuned configurations are not portable for different systems.

It is important to note that there is no such thing as "the" optimal configuration. Instead, there may in fact be multiple distinct configurations that lead to equally optimal performance, which is the reason for the vertical extent of the boxes in figure 7. Therefore and because there usually exists a complex relationship between parameters, it is not possible to just pick a random configuration that satisfies the statistical properties displayed in figure 7 and expect optimal performance. Only specific configurations achieve this goal, making autotuning a valuable tool to solve the configuration problem.

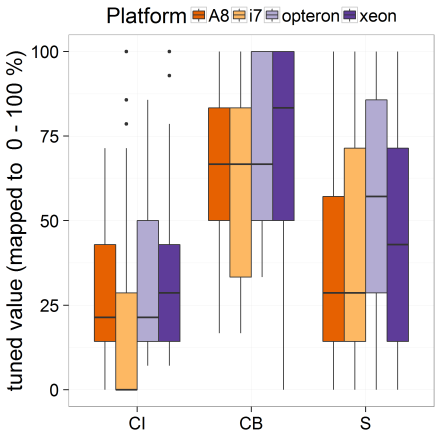
*3) Convergence:* Figure 8 shows the convergence properties of the autotuning process for the Sponza and Wood Doll scenes as representatives for the static and dynamic input data. In both scenes, the autotuner reaches a relatively stable state after just about 40 iterations. However, there is a clear difference between the static and dynamic scenes. For the static scenes, once a stable state is reached, there is little jitter in the speedup. We attribute this jitter to measurement and system load noise. For the dynamic scenes, however, we see a variance of much greater magnitude. This shows the effect of input data on an optimal configuration: A configuration that is optimal for one frame, may not be as good for the subsequent one, which the autotuner needs to correct.




(a) Static Scenes

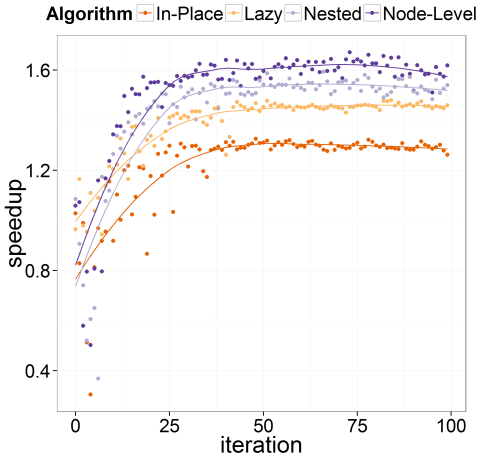



(b) Dynamic Scenes

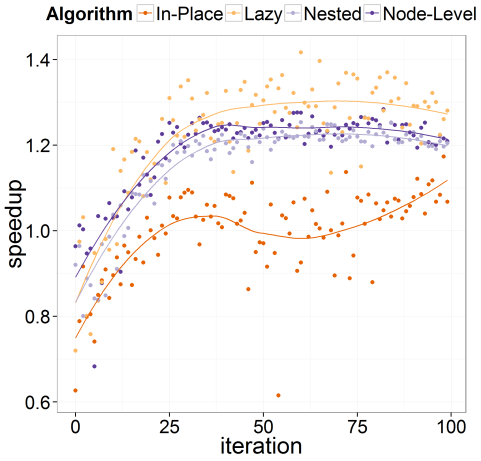



(c) Sibenik scene on four different hardware platforms

Figure 7: Distribution of tuned configurations on the In-Place Algorithm.




(a) Sponza



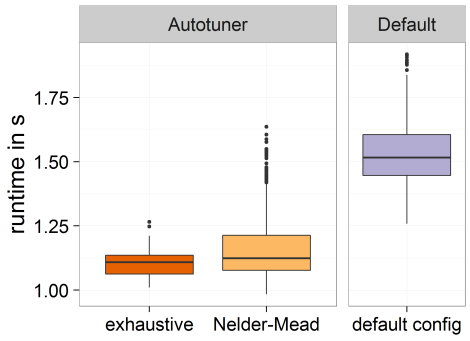

(b) Wood Doll

Figure 8: Mean speedup over time

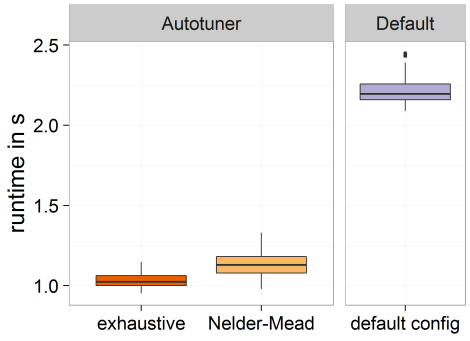
*4) Comparison to exhaustive search:* Being a global optimization technique, the Nelder-Mead search is vulnerable to local minima. To determine the effects of this threat in our experiments, we compare the Nelder-Mead optimization results with the results of an exhaustive search. Figure 9 shows the absolute runtime for the Sibenik scene and all algorithms, using the configurations found by exhaustive and Nelder-Mead search, as well as the default configuration. The underlying measurements have been repeated 150 times.

For the nested, in-place and node-level algorithms, the median performance achieved by the Nelder-Mead search is within 2% of the minimal achievable runtime. However, we can observe a small number of outliers with a speedup of $\sim$ 1. These point to optimization runs where the Nelder-Mead search converges to local minima.

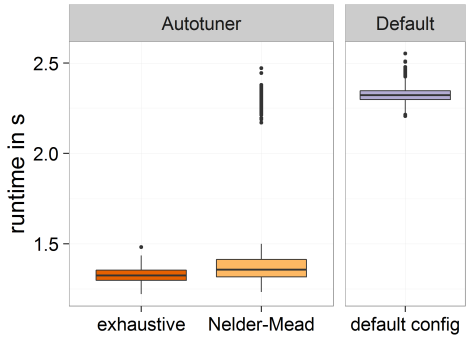
For the lazy algorithm, the median runtime of the Nelder-Mead optimization results is within 10% of the median



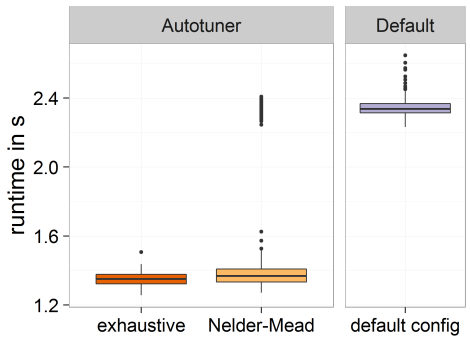

(a) In-Place




(b) Lazy




(c) Nested




(d) Node-Level

Figure 9: Comparison between Nelder-Mead, exhaustive search and the default configurations on the Sibenik scene. Hardware: dual AMD Opteron 12-core system (1.9 GHz)

optimal runtime. Although there are no outliers with significantly worse performance in this experiment, the lack thereof does not disprove the existence of local minima. Instead, we attribute the larger disparity between median runtimes to local extrema, however with less adverse effects as is visible in the other experiments.

In summary, this data suggests that the Nelder-Mead search yields close to optimal performance. Although the results point to rare cases in which we do not achieve a runtime improvement through autotuning, these cases are not worse than the default configuration. Because we employ online autotuning, these cases can in practice be countered by repeating the optimization as needed during application runtime.

## VI. Conclusion

We have used online-autotuning to optimize kD-tree data structures. We tuned algorithm specific and parallelization parameters of four different SAH kD-tree implementations and evaluated the speedup on three hardware platforms and six scenes. The resulting speedups of up to 1.96x over the base configuration prove that optimizing these parameters is necessary to achieve optimal performance. Finding the optimal configurations on alternating scenes and hardware with differing multi-threading capabilities showed that configurations are hardly ever portable. However our online-autotuning technique finds the best configuration independent of the current context. Low programmer effort is needed to use our simple online-autotuner API.

We believe that tuning parallelization parameters in conjunction with algorithm parameters is especially important as multi-threaded applications become the norm and the diversity of hardware platforms increases. However, the experiments we ran left one degree of freedom unexamined, namely the question of which algorithm creates the best performance for a given scene and given hardware. It is hard for current autotuning techniques to provide an answer for this question, because the elaborate employed search techniques are generally based on the notions of "distance" and "direction" in the search space, both of which are undefined for an unordered set of algorithms. It is thus an interesting open research question if autotuning in this context can be improved beyond optimizing one algorithm after another and then picking the best, especially because having more than one such "nominal" parameter will quickly lead to combinatorial explosion.

## Acknowledgments

We thank the Stanford University Computer Graphics Laboratory for the Bunny model, M. Dabrovic for the Sponza and Sibenik models as well as the Utah 3D Animation Repository for the Fairy Forest, Toasters and Wood Doll models.

## References

[1] M. Shevtsov, A. Soupikov, and A. Kapustin, "Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes," in *Computer Graphics Forum*. Wiley Online Library, 2007.

[2] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart, "Parallel SAH k-d tree construction," in *Proceedings of the Conference on High Performance Graphics*. Eurographics Association, 2010.

[3] Z. Wu, F. Zhao, and X. Liu, "SAH KD-tree construction on GPU," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. ACM, 2011.

[4] C. Tapus, I.-H. Chung, and J. K. Hollingsworth, "Active harmony: Towards automated performance tuning," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, 2002.

[5] T. Karcher and V. Pankratius, "Run-time automatic performance tuning for multicore applications," in *Euro-Par Parallel Processing*. Springer Berlin Heidelberg, 2011.

[6] 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 23rd International Conference on Parallel Architectures and Compilation*. ACM, 2014.

[7] H. Jordan, P. Thoman, J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A multi-objective autotuning framework for parallel codes," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[8] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in O(N log N)," in *IEEE Symposium on Interactive Ray Tracing*, 2006.

[9] J. Günther, H. Friedrich, I. Wald, H.-P. Seidel, and P. Slusallek, "Ray tracing animated scenes using motion decomposition," in *Computer Graphics Forum*. Wiley Online Library, 2006.

[10] N. Sample, M. Haines, M. Arnold, and T. Purcell, "Optimizing search strategies in kd trees," in *Fifth WSES/IEEE World Multiconference on Circuits, Systems, Communications & Computers (CSCC)*, 2001.

[11] P. Danilewski, S. Popov, and P. Slusallek, "Binned SAH Kd-tree construction on a GPU," *Saarland University*, 2010.

[12] J.-P. Roccia, M. Paulin, and C. Coustet, "Hybrid CPU/GPU KD-Tree Construction for Versatile Ray Tracing," in *Eurographics - Short Papers*. Eurographics Association, 2012.

[13] N. Weber and M. Goesele, "Auto-tuning complex array layouts for GPUs," in *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*. Eurographics Association, 2014.

[14] D. Gadioli, G. Palermo, and C. Silvano, "Application autotuning to support runtime adaptivity in multicore architectures," in *International Conference on Embedded Computer Systems: Architectures, Modelling and Simulation*. IEEE, 2015.

[15] M. Tillmann, T. Karcher, C. Dachsbacher, and W. F. Tichy, "Application-independent autotuning for GPUs." in *Parallel Computing: Accelerating Computational Science and Engineering*. IOS Press, 2013.

[16] P. Ganestam and M. Doggett, "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*. ACM, 2012.

[17] C. A. Schaefer, V. Pankratius, and W. F. Tichy, "Atune-IL: An instrumentation language for auto-tuning parallel applications," in *Euro-Par Parallel Processing*. Springer Berlin Heidelberg, 2009.

[18] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The computer journal*, no. 4, 1965.

[19] W. Hunt, W. Mark, and D. Fussell, "Fast and lazy build of acceleration structures from scene hierarchies," in *IEEE Symposium on Interactive Ray Tracing*, 2007.

[20] P. Djeu, W. Hunt, R. Wang, I. Elhassan, G. Stoll, and W. R. Mark, "Razor: An architecture for dynamic multiresolution ray tracing," *ACM Transactions on Graphics*, no. 5, 2011.

[21] A. Appel, "Some techniques for shading machine renderings of solids," in *Proceedings of the 1968, spring joint computer conference*. ACM, 1968.

[22] C. Ericson, *Real-Time Collision Detection*, ser. The Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann Publishers Inc., 2004.