Karlsruhe Institute of Technology

Incremental BVH Optimization on the GPU

Bachelor's Thesis of

Daniel Mensinger

at the Department of Informatics
Institute for computer graphics

Reviewer:	Prof. Dr.-Ing. Carsten Dachsbacher
Second reviewer:	Prof. Dr. Hartmut Prautzsch
Advisor:	M.Sc. Daniel Opitz

11. May 2018 – 10. September 2018

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 6th September 2018

. . . . . .

(Daniel Mensinger)

Zusammenfassung

Thema dieser Arbeit ist ein Algorithmus für die parallele Optimierung von Bounding Volume Hierarchies (BVHs). Der hierfür entwickelte Algorithmus basiert auf dem bereits existierenden, sequenziellen Algorithmus von Bittner u. a. [BHH13]. Dieser Ausgangsalgorithmus wurde parallelisiert und optimiert und sowohl auf der CPU als auch auf der GPU implementiert. Hierbei ist die GPU-Implementierung bis zu 17 mal schneller, und die CPU-Implementierung ist bis zu 2 mal schneller als der sequenziellen Ausgangsalgorithmus von Bittner u. a. [BHH13]. Durch diese Geschwindigkeitsoptimierung kann die GPU-Implementierung des hier präsentierten Algorithmus auch im Bereich des Echtzeit-Raytracings verwendet werden.

Abstract

The topic of this thesis is an algorithm for the parallel optimization of Bounding Volume Hierarchies (BVHs). The algorithm developed for this is based on the existing sequential algorithm from Bittner et al. [BHH13]. This initial algorithm has been parallelized and optimized and implemented on both the CPU and the GPU. The GPU implementation is up to 17 times faster, and the CPU implementation is up to 2 times faster than the sequential output algorithm of Bittner et al. [BHH13]. Through this speed optimization, the GPU implementation of the algorithm presented here can also be used in the area of real-time raytracing.

Contents

List of Figures

1. Introduction

Ray tracing algorithms are capable of rendering photorealistic images from a three-dimensional scene. They are, however, inherently expensive because these algorithms have to intersect millions of rays with millions of triangles in the rendering process.

Acceleration structures can alleviate this problem by significantly decreasing the expected number of ray-triangle intersections required for each ray. There are multiple acceleration structures such as Bounding Volume Hierarchies (BVHs), k-d-trees, uniform grids, and octrees.

However, these acceleration structures are not limited to ray tracing. They are also useful for collision detection and any other procedures where geometry has to be intersected. Thus these applications can also benefit from the results presented in this thesis.

As Thrane et al. [TSØ05] have shown, BVHs, as well as k-d-trees, usually outperform other acceleration structures. Vinkler et al. [VHB16] further investigated the performance of BVHs and k-d-trees. They concluded that BVHs outperform k-d-trees for small to medium-sized objects. Thus this thesis focuses only on BVHs.

However, there are many valid BVHs for any given scene, which can significantly differ in quality. The quality of a BVH generally determines how good it is at accelerating an algorithm. A high-quality BVH can enable a ray tracer, for instance, to compute more ray-triangle intersections per second than a low-quality BVH. Thus high-quality BVHs are crucial for fast ray tracing.

There are already multiple algorithms for creating a BVH directly from a 3D scene (see section 1.1 on the following page). These algorithms differ in the quality of the generated BVH and the time needed to create them. Although the goal of every BVH builder is to create a high-quality BVH in as little time as possible, there is generally a tradeoff between the BVH quality and the runtime performance of the BVH builder algorithm.

A different approach for high-quality BVH creation is to first generate a low-quality BVH with a fast algorithm. This BVH is then further optimized.

Such a parallel BVH optimization algorithm is the primary topic of this thesis. The algorithm presented here is based on the already existing BVH optimization algorithm proposed by Bittner et al. [BHH13]. Through optimization and parallelization of this original algorithm, the algorithm introduced in this thesis can optimize any given BVH to the same degree as the original algorithm in a significantly reduced amount of time.

1.1. Related work

As mentioned before, the use of acceleration structures in ray tracing has a long history. Already in 1980 Rubin and Whitted [RW80] manually created BVHs to accelerate the intersection test required for ray tracing. Weghorst et al. [WHG84] discussed different types of bounding volumes and methods to create them more optimally. Later Kay and Kajiya [KK86] created a top-down BVH creation algorithm using a median cut.

Goldsmith and Salmon [GS87] proposed a cost measure, currently known as the *surface area heuristic (SAH)*, which predicts the quality of a given BVH even during its construction. They used a greedy insertion based algorithm for BVH creation.

Later MacDonald and Booth [MB90] further investigated the SAH cost function and were able to create new SAH based algorithms, which produce higher quality BVHs.

Wald et al. [WBS07] described an optimal sequential top-down BVH builder. This algorithm calculates the optimal split plane by first sorting the primitives along the three coordinate axes. Then the algorithm performs coordinate sweeps to calculate the optimal split plane.

However, as Kensler [Ken08] has shown, BVHs can be further improved by iteratively optimizing the tree after its construction. Kensler used local tree rotations resulting in a further improved BVH, constructed by the best known top-down BVH construction algorithm.

Later Bittner et al. [BHH13] proposed a faster insertion based approach. Here the algorithm sequentially removes nodes from the BVH tree and inserts them back at a different location to minimize the SAH cost of the tree. In the same year Karras and Aila [KA13] also introduced tree optimization algorithm. Their massively parallel approach is based on local tree updates rather than the global tree updates used by Bittner et al. [BHH13]. Their parallel algorithm is faster than the sequential algorithm while reaching nearly the same level of BVH quality.

The optimization algorithm developed by Gu et al. [GHB15] first shoots a small number of sample rays to gather statistical information. Then a fast optimization is applied to increase the quality of the BVH significantly.

Another approach for increasing ray tracing speed is to optimize the memory layout of a BVH. Mahovsky and Wyvill [MW06] introduced a memory efficient layout for BVHs which trades of memory requirements with additional processing overhead.

Later Kim et al. [Kim+10] were able to further decrease the memory footprint by a ratio of 12:1 with their RACBVH. Meanwhile Yoon and Manocha [YM06] presented an algorithm to optimize the cache utilization of the ray tracer by modifying the memory layout of a BVH.

A major issue of SAH is that the construction of SAH based BVHs can be relatively expensive. Wald et al. [WBS07] calculated a time complexity of $O(n * log(n))$ for an optimal SAH builder.

In Wald [Wal07], Wald [Wal12] and Wald et al. [WIP08] a faster, binned SAH builder is introduced. Here only a few split planes are considered at each node. This approach reduces the quality of the generated BVH but also reduces its construction time.

Lauterbach et al. [Lau+09] were able to parallelize BVH construction by sorting all primitives along a space-filling curve, thus creating a linear bounding volume hierarchie (LBVH). This approach is an order of magnitude faster than other BVH construction methods but at the cost of the BVH quality.

The HLBVH algorithm developed by Pantaleoni and Luebke [PL10] improves upon the LBVH algorithm resulting in lower build times and the generation of better acceleration structures. Garanzha et al. [GPM11] improved this algorithm by using work queues and binary search. Karras [Kar12] was able to further parallelize the construction of the HLBVH tree by using binary radix trees.

Vinkler et al. [Vin+13] has shown that BVH construction can be parallelized on the GPU, by subdividing the algorithm into junks and parallelizing them on the GPU.

There are also other approaches for fast BVH creation:
In 2015 Ganestam et al. [Gan+15] developed Bonsai, a fast BVH construction algorithm which reaches 92% to 105% more ray tracing performance. Bittner et al. [BHH15] used a parallel insertion based algorithm to create high-quality BVHs. Until then it was thought to be impossible to create such an algorithm that produces a decent quality BVH.

The quality of a BVH is usually evaluated by calculating the SAH of the entire tree and measuring the time it takes to render an image. Aila et al. [AKL13] have shown that while the SAH predicts the actual runtime performance rather well the SAH alone can not explain the performance of a BVH. They introduced the new end-point overlap (EPO) and Leaf count variability (LCV) metrics which combined with the SAH predict the actual runtime performance of a BVH better.

Based on the work of Aila et al. [AKL13] Wodniok and Goesele [WG17] created the RBVH RSBVH and RSSBVH builders, which significantly reduces SAH as well as EPO.

BVH construction on specialized hardware also offers significant speed improvements. Doyle et al. [DFM13] introduced a hardware BVH constructor that computes a BVH roughly ten times faster than software-based binned construction algorithms. Later Viitanen et al. [Vii+17b] created MergeTree, a hardware based HLBVH builder. Viitanen et al. [Vii+17a] also proposed a hardware accelerator for constructing and refitting BVHs.

With the first programmable GPUs research was also done to parallelize the ray tracing algorithms on the GPU. Purcell et al. [Pur+05] and Carr et al. [CHH02] created ray tracing algorithms that utilizing the many-core architecture of the GPU. Later Weiskopf et al. [WSE04] created a nonlinear ray tracer running entirely on the GPU. Popov et al. [Pop+07] and Áfra and Szirmay-Kalos [ÁS14] have shown, that it is possible to increase the raytracing performance of the GPU further when using stackless algorithms.

1.2. Bounding Volume Hierarchies

Bounding Volume Hierarchies (BVHs) are tree data structures of geometric primitives. Each primitive is wrapped in an axially aligned bounding box (AABB) and represented as a leaf node in the BVH tree. Two or more of these nodes are then grouped to generate a tree node. Each tree nodes AABB encloses all AABBs of its children.

Thus a tree structure is generated where the AABB of the root node encloses the entire scene and represents all geometric primitives. Each child node, in turn, represents fewer primitives and has a smaller AABB.

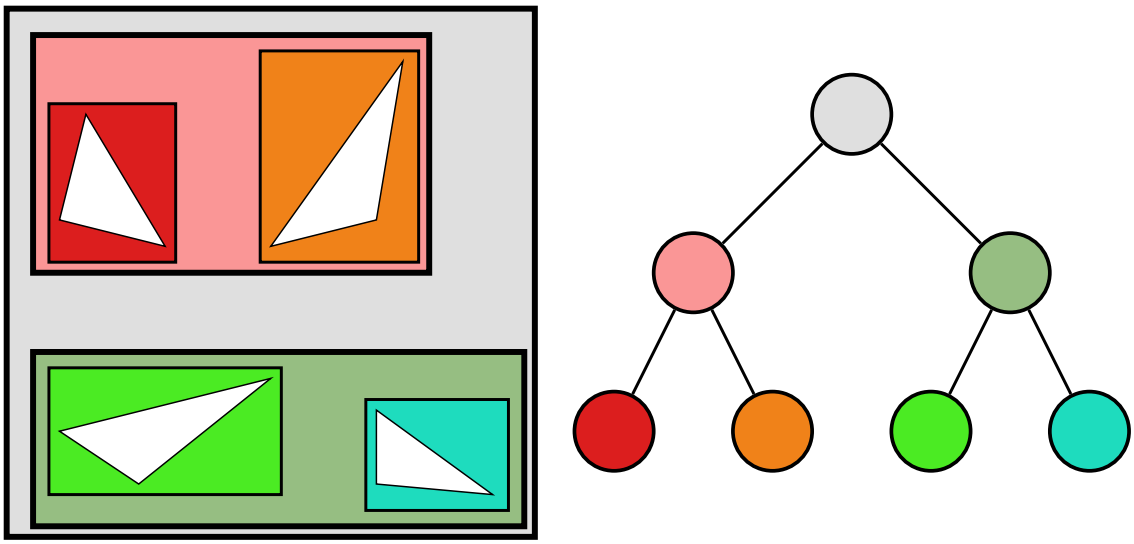


Figure 1.1.: BVH Visualisation

Figure 1.1 visualizes such a data structure. In the image, each color represents a bounding volume. Here, the white triangles are the geometric primitives of the scene.

The bounding boxes are visualized in the left section of the figure. In the right section the tree structure is visualized, to show how the bounding boxes on the left form the BVH.

1.3. Surface area heuristic

The *Surface area heuristic* (SAH) introduced by Goldsmith and Salmon [GS87] is a cost model used to predict the ray tracing performance of a given BVH.

Given a complete BVH tree, the SAH of this entire tree can be calculated with the following formula:

$$C_{SAH} = \frac{1}{S(n_r)}\left[C_i * \sum_{n \in N_i} S(n) + C_l * \sum_{n \in N_l} S(n)\right] \tag{1.1}$$

Here $S(x)$ is defined as the surface area of the node x, n_r is the root node of the tree, C_i is the cost for intersecting an inner node, and C_l is the cost for a leaf node.
N is the entire set of nodes in the tree. N_i and N_l are subsets of N, containing all **i**nner and all **l**eaf nodes.

The SAH cost model makes two general assumptions:

1. The Ray distribution is uniform
2. The Rays are unoccluded (Ray traversal continues when a primitive is hit)

Although these conditions often are not met in practice, Havran [Hav00] has shown that the SAH correlates surprisingly well with the real-life ray tracing performance. This correlation, however, requires finetuning of the parameters C_i and C_l. Later Aila et al. [AKL13] have shown that the SAH alone can not entirely predict the ray tracing performance across different BVH builders.

Nonetheless, the SAH can still be used to compare BVHs built or optimized with the same algorithm. This is the case in this thesis because different versions of fundamentally the same algorithm are discussed.

Since the primary focus of this thesis is the runtime optimization of the existing algorithm from Bittner et al. [BHH13], the SAH is only used to determine the BVH quality compared to the original algorithm. Furthermore, it is shown in chapter 3 on page 27 that the optimized algorithms can produce BVHs with the same SAH value as the original version. A more detailed analysis of the ray-tracing performance was done in the paper of the original algorithm.

Thus, a strong correlation between SAH and ray tracing performance is not required here and the finetuning of C_i and C_l can be skipped. In this thesis, C_i and C_l were both set to 1 for simplicity. Equation (1.1) on the preceding page can then be simplified to

$$C_{SAH} = \frac{1}{S(n_r)} \sum_{n \in N} S(n) \tag{1.2}$$

for $C_i = C_l = 1$. This adjustment only affects the SAH values of the test results, presented in chapter 3 on page 27. Choosing different values for C_i and C_l will result in a shift in the SAH of all plots. However, this is not a problem, since the relative differences in the graphs do not change.

2. Implementation

The algorithm presented in this thesis is fundamentally an optimized and parallelized version of the algorithm proposed by Bittner et al. [BHH13]. Thus the basic BVH optimization principle remains the same for all algorithms:

1. Identify problematic nodes
2. Remove them
3. Reinsert them in a better location

Furthermore, all algorithms were successfully implemented in C++ and CUDA. It should, however, be possible to implement them in any programming language.

It should be noted that the goal of the optimizations presented in this thesis is not to further improve the quality of the generated BVHs. Instead, the primary goal is to significantly increase the runtime performance while retaining the same BVH quality.

2.1. BVH Layout

Since all algorithms utilize the same BVH layout, it is presented here first. Furthermore, only binary BVH trees were used in this thesis.

This layout is introduced as an example to improve the readability of the pseudocode in the following sections. Since this layout is only presented for clarification and reproducibility, it should thus be possible to use different memory layouts. In theory, all algorithms can also be modified to work with another tree layout instead of the binary tree layout used in this thesis.

Figure 2.1 on the next page shows an example BVH tree while figure 2.2 illustrates how this tree is laid out in memory: With this layout, the entire BVH tree is tightly packed into a single array. The root node is furthermore always located at array index 0. Each node explicitly stores the array index of its two children and the index of its parent (the parent of the root node is the root node). The parent reference is neccessary since the tree is traversed in both directions in the algorithms presented here.

Since explicit references between nodes are used, the array locations of all nodes, besides the root node, are irrelevant. This has the advantage that the array can be tightly packed at the cost of additional storage in each node for the references.

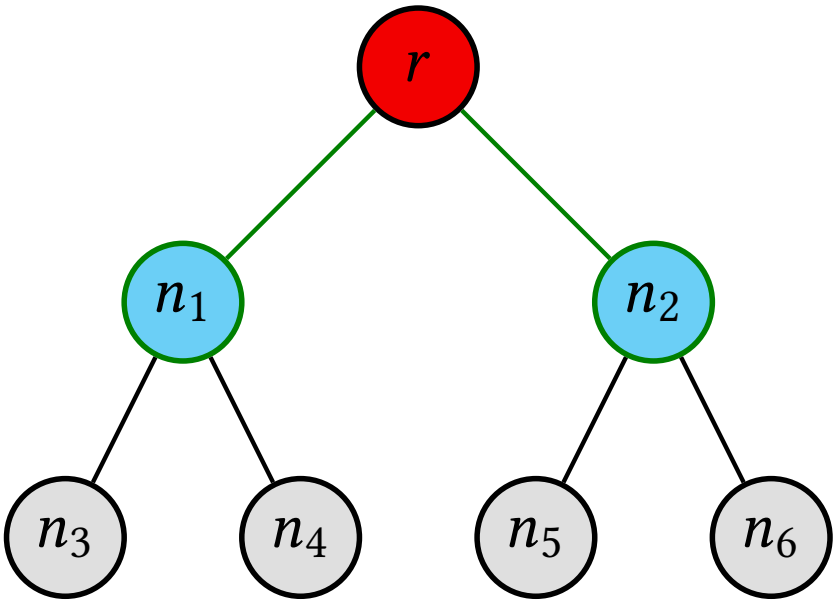


Figure 2.1.: Binary tree example

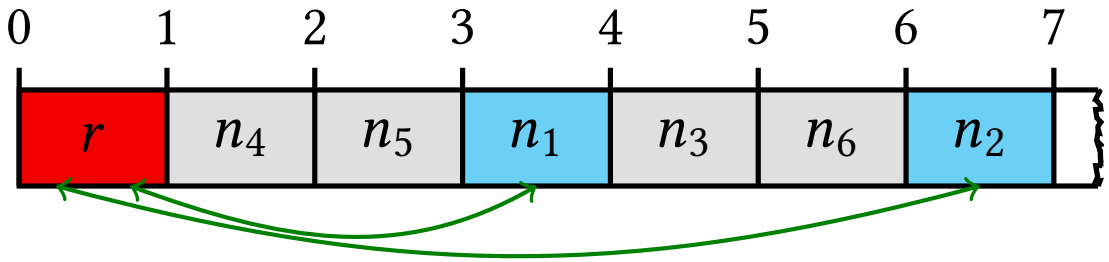


Figure 2.2.: Memory layout of the BVH (one cell represents one node)

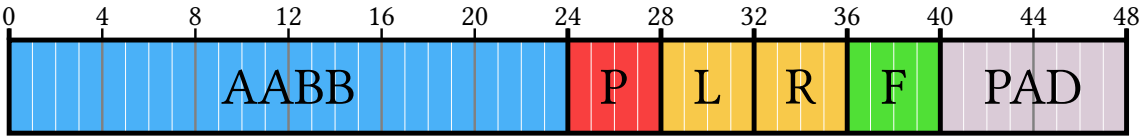


Figure 2.3.: Memory layout of the BVHNode structure

The nodes in the BVH tree are defined as shown in figure 2.3 on the facing page. The BVHNode data structure, shown in this figure, is 48 bytes wide. Here, the first 24 bytes are used to store the axis aligned bounding box (*AABB*).

The *AABB* stores the minimum and maximum coordinates of the bounding box. Since each coordinate consists of three floating point variables, the entire AABB requires $2 * 3 * 4 = 24$ bytes of storage. The size of the *AABB* can theoretically be reduced to $2 * 3 * 2 = 12$ bytes by using half-precision floating point variables. Doing so also reduces the precision of the AABB, which can decrease the performance of the ray tracer (see Mahovsky and Wyvill [MW06]).

In the following 12 bytes of the BVHNode structure, the references to the parent (*P*) and child nodes (*L* and *R*) are recorded. While *P* always points to the parent node, the definition of *L* and *R* changes depending on the node type. When the node is an inner node, *L* and *R* store the reference to the children of the current node. However, when the node is a leaf node *L* references the first triangle and *R* records the number of triangles represented by the node.

Bytes 36 to 40 (*F*) are reserved for binary flags that describe the node. Here is stored whether the node is a leaf node and if the node is the left or right child of the parent.

The remaining 8 bytes (*PAD*) are not relevant for the structure of the tree itself. They are used as padding to ensure a 16 byte alignment of the BVHNode. They can be used to store additional information like the tree level of the node and the precomputed surface area of the AABB.

2.2. Original algorithm

As mentioned before, the fundamental principle of the algorithm is the removing and reinsertion of nodes in the BVH tree. These tree updates are done in such a way that the total sum of all surface areas $\sum_{n \in N} S(n)$ in the tree is reduced. This automatically reduces the SAH (see equation (1.2) on page 5) of the BVH.

To achieve this, nodes which cause a SAH cost overhead have to be selected first. This can be done randomly, but better results can be obtained by selecting nodes with a cost overhead function (see section 2.2.1 on the following page).

When a node is found it is then removed from the BVH tree as explained in section 2.2.2 on page 12. Then a better position in the tree for the node is searched (see section 2.2.3 on page 13). Finally, the node is reinserted at that position in the tree (see section 2.2.4 on page 14) and the whole process is repeated.

In practice, this process is done in steps rather than one node at a time. Here a batch of `batchSize` nodes is first selected. Each node is then sequentially processed as described above.

This can be best illustrated with the following pseudocode:

```
void doStep(BVHNode  *nodes, uint32_t batchSize) {
  // First select a batch of nodes: section 2.2.1
  uint32_t *toProcess = selectNodes(nodes, batchSize);

  for (uint32_t i = 0; i < batchSize; ++i) {
    // Remove the current node from the tree: section 2.2.2 on page 12
    removeNode(toProcess[i], nodes);

    // Find the new position: section 2.2.3 on page 13
    auto pos = findReinsertionPosition(toProcess[i], nodes);

    // Reinsert nodes at pos: section 2.2.4 on page 14
    reinsertNode(toProcess[i], pos, nodes);
  }
}
```

These algorithm steps are executed until a termination criterion is met. As shown in section 3.1.2 on page 30 the SAH is significantly reduced in the first few steps and converges in the subsequent steps. The algorithm could thus be terminated when no significant decrease in the SAH value is detected after a certain number of steps.

Usually, a percentage of the total number of nodes in the tree is used for the `batchSize`. The effect of different such batch percentages can be seen in section 3.1.2 on page 30.

There are multiple reasons why this batch processing is done. First of all, this approach increases the performance of the algorithm because the cost overhead does not have to be calculated after every tree update. This step would be required otherwise, since one reinsertion procedure changes the cost measure of many nodes in the tree.

Secondly, the batch processing is more robust than processing one node after another. This is because the cost overhead measure (see next section) only *predicts* the cost overhead of a node. It can thus happen that the highest cost is always assigned to the same node. This permanently traps the algorithm in an infinite loop, where the SAH is no longer reduced.

It should be noted that this is not an issue when the nodes are randomly selected. An infinite loop is practically impossible with this approach.

2.2.1. Selecting nodes

Nodes are selected by first computing their SAH cost overhead. This is done with the node inefficiency measure proposed by Bittner et al. [BHH13]. This function is a combination of the following three different measures:

The first measure estimates the relative surface area increase of the node relative to its children.

$$M_{SUM}(n) = \frac{S(n)}{\frac{1}{|N_c(n)|} * \sum_{x \in N_c(n)} S(x)}$$

Here $S(n)$ is defined as the surface area of the node n and $N_c(n)$ is the set of child nodes of n. The next measure M_{MIN} identifies nodes, where one child node is significantly bigger than the other.

$$M_{MIN}(n) = \frac{S(n)}{MIN_{x \in N_c(n)} S(x)}$$

The last inefficiency measure corresponds to the surface area of the node:

$$M_{AREA}(n) = S(n)$$

The final measure is a combination of these three measures. It is defined as:

$$\begin{aligned} M(n) &= M_{SUM}(n) * M_{MIN}(n) * M_{AREA}(n) \\ &= \frac{S(n)}{\frac{1}{|N_c(n)|} * \sum_{x \in N_c(n)} S(x)} * \frac{S(n)}{MIN_{x \in N_c(n)} S(x)} * S(n) \\ &= \frac{S(n)^3 * |N_c(n)|}{\sum_{x \in N_c(n)} (S(x)) * MIN_{x \in (N_c(n))} (S(x))} \end{aligned} \tag{2.1}$$

This measure is computed for each BVH node. The result is stored with the index of the node in a separate array with the same length as the BVH node array (see figure 2.4).

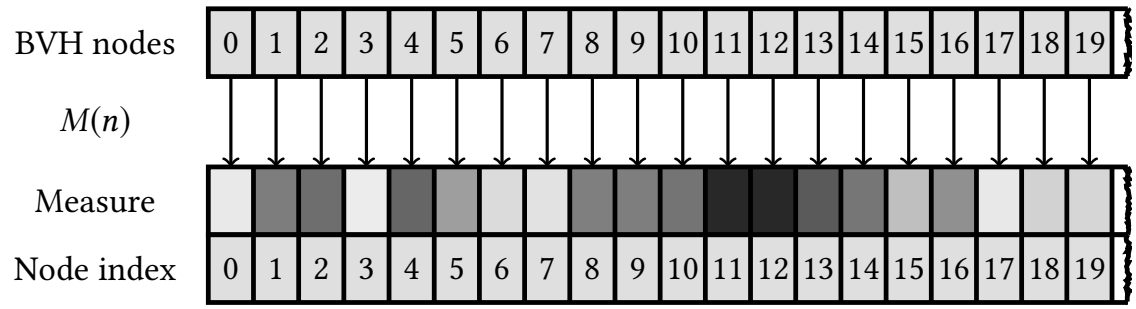


Figure 2.4.: Measure calculation

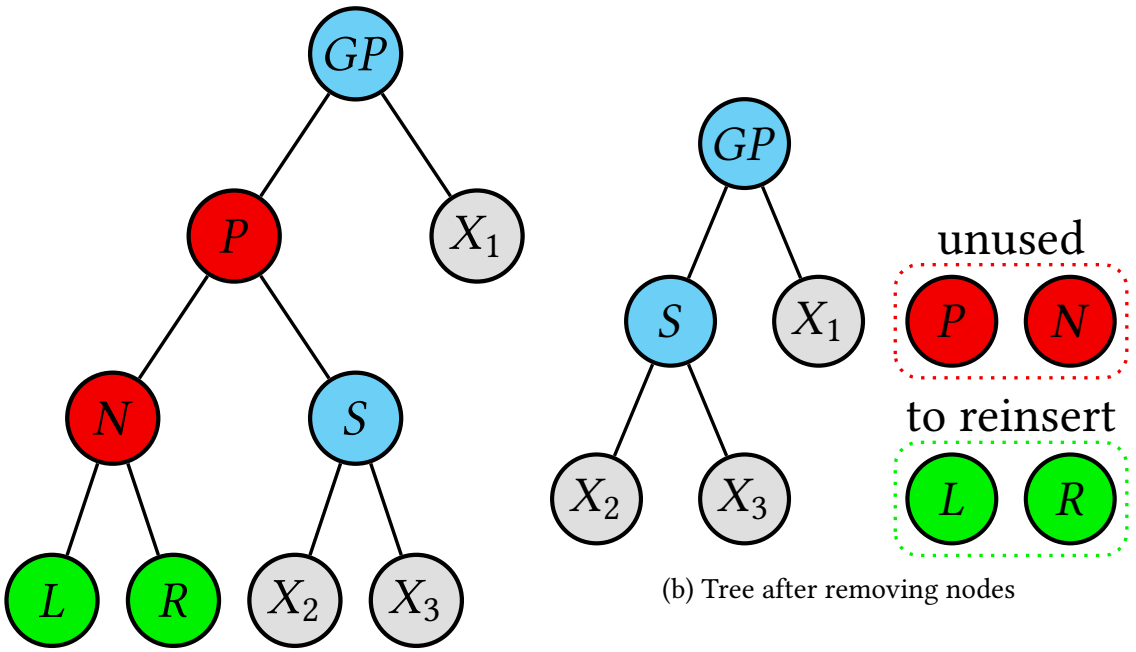
The nodes for the batch can then be selected by sorting this array by the measures and taking the first `batchSize` elements.

Alternatively a partial sorting algorithm like `std::nth_element` can also be used. This algorithm rearranges the elements in such a way that the element at position `batchSize` is identical to the element which would occur at this position if the whole array was sorted. It is furthermore guaranteed that all elements in the range $[0, \texttt{batchSize})$ are greater than or equal to the element at position `batchSize`.

Experimental testing in section 3.1.3 on page 32 has shown that there is practically no difference between a complete and a partial sort.

2.2.2. Removing nodes

Once a node is selected, it can be removed from the BVH tree. This is done by first removing both children from the node and fixing the tree accordingly. This process is illustrated by figure 2.5a and figure 2.5b:



(a) Original tree

(b) Tree after removing nodes

Figure 2.5.: Removing nodes

Node N represents here the selected node to be removed. P is the parent and GP the grandparent node of N. L and R are the child nodes of N which will be reinserted later in the tree. Both L and R are being kept intact, only their parent edges to N are removed. The subtrees of both child nodes are not touched. Because of that, the entire subtrees of the nodes are moved when they are later reinserted.

N and P, on the other hand, are entirely reset. They are later used for the reinsertion of L and R (see section 2.2.4 on page 14). The sibling S is used to fill the void left behind by P. Thus the BVH tree remains valid.

Before the next step of the algorithm can be executed, the bounding boxes have to be updated to reflect the changes to the tree. This can be done by walking from GP to the root of the BVH and updating the bounding boxes along the way.

2.2.3. Finding insertion position

Once the nodes L and R are removed, a new position is calculated for them. Here the node with the biggest surface area is processed first. Once the new position is found the node is immediately reinserted (see section 2.2.4 on the next page).

Only when this process is finished is the node with the lesser surface area processed. Using this approach, the SAH is decreased faster compared to processing the node with the lower SAH first.

The new positions are found with the branch and bound algorithm from Bittner et al. [BHH13]. Their algorithm always starts the search at the root of the tree. It uses the *direct* and *induced* cost proposed in their paper and a priority queue to find the best position for the node in the tree.

The following listing shows a possible implementation of their algorithm:

```
1  uint32_t findNode(uint32_t n, BVHNodes *nodes) {
2    float           bestCost      = numeric_limits<float>::infinity();
3    uint32_t        bestNodeIndex = UINT32_MAX; // no node found
4    BVHNode const * node          = &nodes[n];
5
6    std::priority_queue<Entry> PQ; // pq of node/cost pairs. Lowest cost is on top.
7    PQ.push({0, 0.0f});            // 0 is always the root.
8
9    while (!lPQ.empty()) {
10     Entry          curr     = PQ.top(); PQ.pop();
11     BVHNode const *currNode = nodes[curr.node];
12
13     if ((curr.cost + node->bbox.surfaceArea()) >= bestCost) { break; }
14
15     AABB lMerge = node->bbox;                // Calculate the direct cost:
16     lMerge.mergeWith(currNode->bbox);        // C_direct = S(n_toInsert ∪ n_currNode)
17     float directCost = lMerge.surfaceArea();
18     float totalCost  = directCost + curr.cost;
19
20     if (totalCost < bestCost) { bestCost = totalCost; bestNodeIndex = curr.node; }
21
22     float newInduced = totalCost - currNode->bbox.surfaceArea();
23     if ((newInduced + node->bbox.surfaceArea()) < bestCost && !currNode->isLeaf) {
24       PQ.push({currNode->left,  newInduced});
25       PQ.push({currNode->right, newInduced});
26     }
27   }
28   return bestNodeIndex;
29 }
```

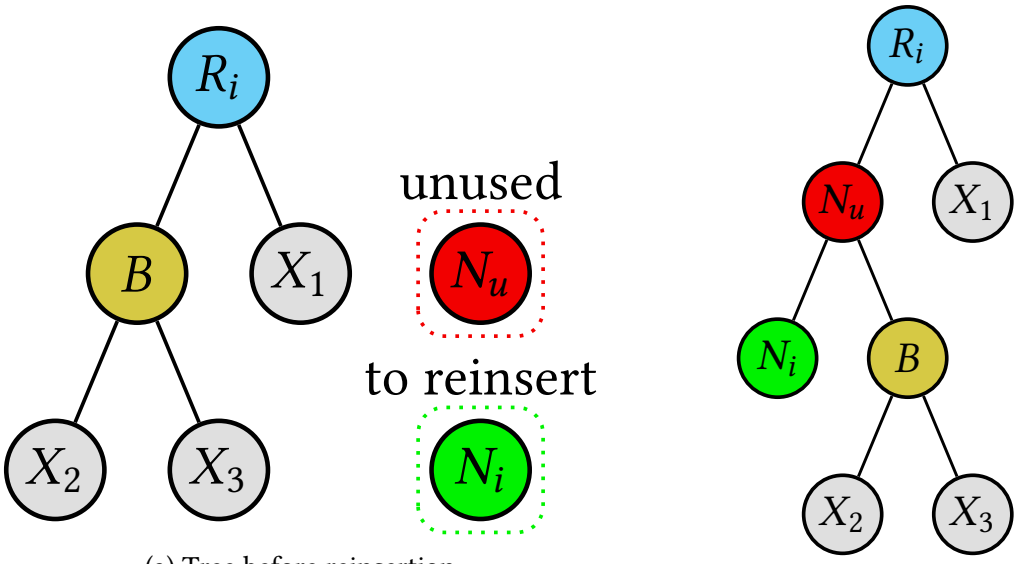
This algorithm is the central part of this BVH optimization approach. It is, however, inherently sequential and cannot be efficiently parallelized. This is because it implements a branch and bound algorithm with a priority queue.

Since the focus of this thesis is the *runtime* optimization of the algorithm by parallelization, this algorithm is not explained further.

The full explanation of this algorithm can be found in the original paper.

2.2.4. Reinserting nodes

With the new position found the node is reinserted into the tree. Figure 2.6a and figure 2.6b illustrate this process.



(a) Tree before reinsertion

(b) Tree after reinsertion

Figure 2.6.: Reinserting nodes

B is the node which currently resides on the position calculated by section 2.2.3 on the preceding page. R_i is its the parent. N_u represents one of the unused nodes N or P from section 2.2.2 on page 12. N_i is the node that is currently beeing processed (either L or R).

The algorithm essentially removes B form the tree and replaces it with N_u. B is then reinserted as a child of N_u alongside the currently processed node N_i.

After the reinsertion, the bounding boxes have to be updated. This is done by traversing the tree from N_u to the root and updating all bounding boxes on the path.

2.3. Parallel Implementation

The algorithm described in the previous section is inherently sequential. While the node selection can be trivially parallelized, the tree update, however, requires significant changes to the algorithm.

The parallelization approach presented here works, like the original algorithm, with a batch of selected nodes. First, this batch is determined as described in section 2.3.3 on page 19. Then the selected nodes are removed and reinserted in parallel.

However, as mentioned before the tree update algorithm is inherently sequential. This is because the *remove* → *findNode* → *reinsert* operations cannot feasibly be executed simultaneously on the same tree. Doing so may cause some of these operations to modify the same nodes of the tree. Such a conflict can introduce loops into the tree, and entire subtrees may become permanently unreachable as a result.

It might be possible to avoid these conflicts and race conditions with thread synchronization and conflict detection algorithms. However, such an approach would require complex logic to ensure the validity of the entire tree after every operation. This would, of course, result in additional processing and synchronization overhead. The complexity of this problem would also increase for every operation run in parallel. This overhead is not acceptable on high core count CPUs and many-core architectures like GPUs.

It is also not clear how the findNode algorithm (see section 2.2.3 on page 13) would function in such a scenario because it is based on a global search of the *entire* tree. Calculating the new position of a node with a tree that is simultaneously modified in another thread is likely to *increase* the SAH rather than reducing it.

2.3.1. Parallel algorithm overview

Because of all these problems, a different approach is used in this thesis. Instead of removing and reinserting nodes in parallel on the same tree, each *remove* → *findNode* → *reinsert* operation is done on a "copy" of the initial BVH tree.

Of course, not the entire tree is copied in this algorithm. Instead, lightweight BVH Patches are used to create the illusion of a full tree copy (see section 2.3.2 on page 17) for the findNode algorithm.

When all parallel operations are finished, the tree copies are checked for conflicts. When a conflict between two trees is detected, one of these trees is discarded (see section 2.3.5 on page 20). Next, the trees are merged back together in parallel. Finally, the bounding boxes in the updated tree are fixed as described in section 2.3.6 on page 20.

To achieve this, the algorithm is broken down into parts that can be easily parallelized:

First, the nodes for the batch are selected in parallel, as described in section 2.3.3 on page 19. Then the *remove* → *findNode* → *reinsert* operation is execute in parallel. Finally, the tree "copies" are checked for conflicts and merged (in parallel) into the original tree.

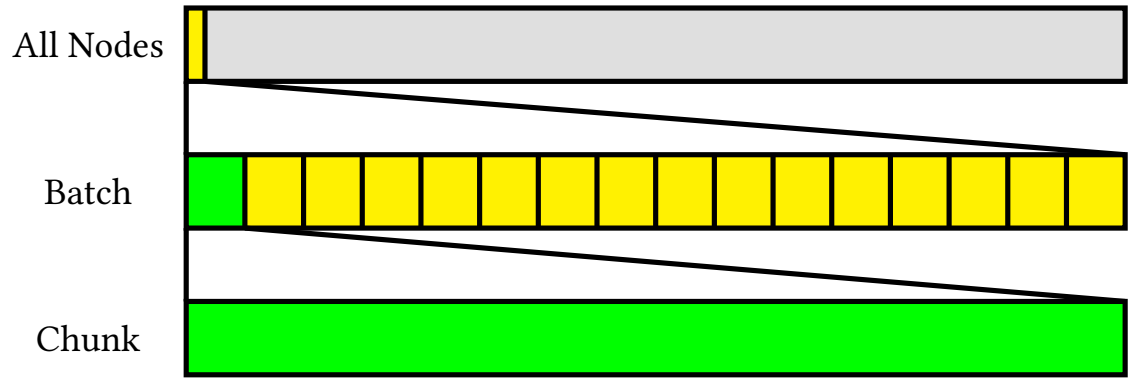


Figure 2.7.: Parallel process visualisation

Additionally not the entire batch is processed at once. Instead, the batch is further subdivided into equally sized chunks (see figure 2.7). The size of one such chunk is defined as `chunkSize`.

Only the nodes in each chunk are processed in parallel. Once all nodes in one chunk are completely processed, the tree "copies" are merged into the original tree. Only then is the next chunk processed. Here the updated BVH tree from the previous chunk is used.

The following pseudocode illustrates this algorithm:

```
1  void doStepPar(BVHNode *nodes, uint32_t batchSize, uint32_t chunkSize) {
2    auto *batch  = parallelSelectNodes(nodes, batchSize);
3    auto *chunks = parallelGenChunks(batch, chunkSize)
4
5    for (uint32_t i = 0; i < (batchSize / chunkSize); ++i) { // For each chunk
6      // Generate chunkSize patches (BVH "copies") from the current BVH
7      auto *patches = parallelGenPatches(chunkSize, nodes);
8
9      // Now execute the original algorithm for each node in the chunk in parallel
10     parallelRemoveFindAndReinsertNodes(patches, chunkSize);
11
12     parallelCheckConflicts(patches, chunkSize);
13     parallelApplyPatchesToBVH(nodes, patches, chunkSize);
14     parallelFixTree(nodes);
15   }
16 }
```

This separation into chunks is done to minimize the total amount of conflicts in the batch. In section 3.2 on page 33 the correlation between the number of conflicts and the number of chunks is analyzed.

However, as a result of this change, only a maximum of `chunkSize` nodes are processed at any given time. Thus a balance between the `chunkSize` and the number of conflicts should be found to maximize the hardware utilization while minimizing the wasted amount of work (conflicts).

Furthermore, it is important how these chunks are laid out. Tests in section 3.2 on page 33 have shown that besides the number of chunks, the chunk layout also significantly impacts the number of conflicts. Figure 2.8 visualizes the two, main layouts used in this thesis.

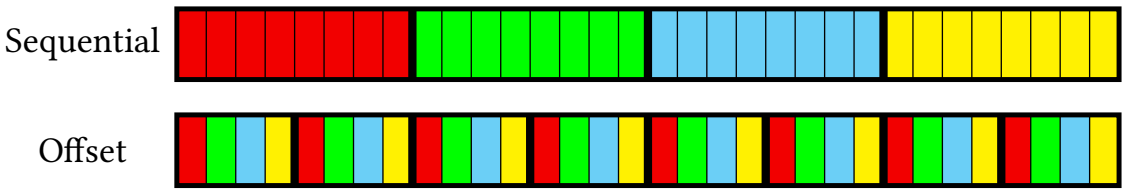


Figure 2.8.: Chunk access visualisation

This example shows a batch of 32 nodes. Each color corresponds to one of the 4 chunks with a `chunkSize` of 8. The first block represents the sequential chunk layout. Here a continuous sequence of 8 nodes is assigned to each chunk. In the offset layout, every fourth node is assigned to a chunk.

Tests in section 3.2 on page 33 have shown that the offset chunk layout produces far fewer conflicts than the sequential layout. This is due to the fact that in the (partially) sorted array conflicting nodes are near each other. Thus thinly spreading the chunk over the entire batch reduces the number of conflicts. The same effect can be achieved by shuffling the chunk, but this would introduce another processing step which decreases the overall performance.

It should also be noted that the sequential layout is more cache friendly than the offset layout. However, the slightly better access pattern does not outweigh the number of conflicts or the cost for shuffling the chunk. Furthermore, the amount of time spent reading from the chunk in the algorithm is insignificant compared to the time spent in other algorithm parts.

2.3.2. BVH patch

One of the key aspects of this algorithm is to compute the original tree update algorithm in parallel on a separate BVH copy. Copying the complete BVH tree, admittedly, wastes much memory since only a few nodes would be modified in the copy.

A better solution is to simulate a full BVH copy with a lightweight `BVHPatch` structure which only stores the modified nodes. This approach also simplifies the “merge” process of the trees: Here the patched nodes in the original tree are replaced with the ones stored in the patch.

The BVHPatch interface is defined as follows: When the algorithm modifies a node, the BVHPatch first creates a copy of the original node. All changes by the algorithm are then only applied to this copy. When the algorithm wants to access a node without modifying it, the BVHPatch first checks if the requested node is already patched. If it is patched it returns the patched node, if it is not patched, the node from the original tree is returned.

As a result, the original BVH tree is not modified in the entire algorithm, and no memory is wasted. However, this approach has a slight drawback: Every time a node is accessed the BVHPatch has to check if the requested node is already patched first. This introduces a slight overhead to every BVH access.

It should be noted that these patches only store changes to the tree layout. The bounding boxes are not stored in these node patches, but in a separate array described below.

Patching Nodes

One BVHPatch structure only has to store a maximum of 10 nodes. This upper bound is a direct result of the remove (as described in section 2.2.2 on page 12) and reinsert process (see section 2.2.4 on page 14). During node removal, the 4 nodes L, R, N, and P are removed from the tree. S and GP are also modified to keep the tree topology intact. During reinsertion the 2 nodes R_i and B are modified. Since the reinsert process is done twice (once for L and once for R) a total of 4 nodes are changed during this process. Thus the total amount of modified nodes never exceeds 10.

The indices of the patched nodes are also stored in the BVHPatch. Checking if a node is patched can then be trivially done by comparing the index of the requested node with all stored indices. As a result, 10 integer comparisons have to be made for each BVH access. This additional overhead does affect the performance of the findNode algorithm since it accesses many nodes.

Reducing the number of comparisons will thus increase the performance of the algorithm. Since the algorithm never traverses the tree backward, nodes that only had their parent pointer changed need not be checked. The same is true for removed nodes since they can never be accessed.

As a result, every node index only has to be compared to the index of the GP node, for the first run of findNode. Here the removed nodes can never be accessed, and the S node only had its parent pointer changed. During the second run of findNode, one R_i and either N or P (depending on which was reinserted) have to be checked in addition to GP.

This optimization measurably improves the performance of the algorithm.

Patching bounding boxes

Besides the nodes itself, the BVHPatch also has to store AABB changes which occur after node removal and node reinsertion. Here AABBs on a path from the modified node to the

root need to be updated. These updates are essential to ensure that the findNode algorithm (see section 2.2.3 on page 13) produces correct results. Two such paths have to be stored: The first path is generated after the nodes are removed. The second path is computed after the first node is reinserted. A third path is not required since the finde node algorithm is not executed again, after the second node is reinserted.

Accessing AABBs through the `BVHPatch` is implemented similarly to accessing nodes. First, the patch structure checks if the requested AABB is in a stored path. Then either a stored AABB or the original AABB is returned.

Furthermore, tests have shown that it is not required to store the entire AABB path to the root for the find node algorithm to work correctly. Tests have shown that limiting the AABB path length to 2 entries does not impact the optimization process for the tested scenes and BVH builders. This approximation works because the changes to the AABBs become less significant the farther away the nodes are from the modification source.

2.3.3. Parallel node selection

As mentioned before the node selection as described in section 2.2.1 on page 10 is the easiest part to parallelize. Each measure calculation can be executed in parallel without any thread synchronization.

This is possible because this calculation does not modify the BVH. Because of that, it does not matter if two or more threads access a node at the same time. Storing the calculated result also does not require synchronization because each array entry is only written to once by only one thread.

Any parallel sorting algorithm can then sort the array. In this implementation, the sorting of the array is not parallelized on the CPU. On the GPU a parallel radix sort algorithm is used.

Randomized node selection can theoretically also be parallelized. This, however, was not implemented since random node selection results in worse SAH reduction (see section 3.1.3 on page 32) and thus should not be used.

2.3.4. Parallel node reinsertion

Once the batch of nodes is determined, the central part of the algorithm is executed. As described in section 2.3.1 on page 15 these updates are done in chunks. At the beginning of each chunk `chunkSize` patches are created. Then the original algorithm from section 2.2.2 on page 12 to section 2.2.4 on page 14 is executed in parallel for each node in the chunk.

Besides the possible usage of the alternative find node algorithm, described in section 2.5 on page 23, and the usage of a `BVHPatch`, there are no changes to the algorithm.

2.3.5. Conflict detection

Once all nodes are removed and reinserted, the generated patches are checked for conflicts. A conflict between two patches occurs when they modify the same node. Such a conflict is resolved by discarding one of the patches.

This algorithm is implemented with an array of atomics. Every element in the array corresponds to one node in the BVH tree, and all elements in the array are initialized with 0. The patches are then checked for conflicts as follows:

For every node stored in the `BVHPatch` the corresponding atomic variable in the array is locked. When all nodes of a patch successfully acquire the locks, the patch passes. However, once one node fails to acquire the lock all previously locked nodes are unlocked and the entire `BVHPatch` is discarded.

The number of discarded patches could be reduced with a more sophisticated algorithm. This would, however, require more complexity and thus more time. Furthermore, tests in section 3.2 on page 33 have shown that this effect is insignificant compared to the speed of this conflict detection algorithm.

2.3.6. Parallel tree update

For the last step, the remaining patches are then applied to the original BVH tree. This is done by overwriting the nodes in the BVH tree with the stored nodes in the patch. This process can be fully parallelized without any thread synchronization since the previously discussed algorithm ensures that no node is written to more than once by multiple patches.

This step only updates the tree structure. The bounding boxes still have to be adjusted to reflect these changes. Updating the bounding boxes can also be parallelized with the atomic array introduced in section 2.3.5.

There are 3 nodes per `BVHPatch` from which the bounding boxes have to be updated: GP, N and P. All three nodes have in common that one or more of their child nodes changed. As a result, their bounding boxes are no longer correct and have to be fixed. Since this update also affects the parent nodes, all nodes on the path from the node to the root of the tree have to be processed.

This naive algorithm can be implemented, by locking the currently processed node and its children with a mutex. Thus only one thread can access one node, and the bounding boxes of its children are up to date. This approach works well with a CPU implementation, where there are a relatively few parallel threads. This reduces the probability that two threads try to access the same node. However, this is not the case on SIMD architectures like CUDA. Furthermore, locking cannot easily be implemented because of hardware restrictions.

With this algorithm, it is also unavoidable that nodes are processed more than once. The root node, in fact, will be updated 3 times per patch. As a result, the thread synchronization is unavoidable to prevent race conditions.

Thus another approach is used: Here the idea is to split the parallel tree traversal into two passes: The first pass is used as a setup phase to count how many threads will try to process each node. Here the tree itself is not changed. In the second the bounding boxes are updated. The results of the first pass are used to determine when to terminate a thread.

In the first pass every thread traverses the tree from the start node to the root. At each node, a fetchAdd operation is performed on the atomic variable of the node. If the previously stored value was 0, the thread continues. If this was not the case, the thread terminates.

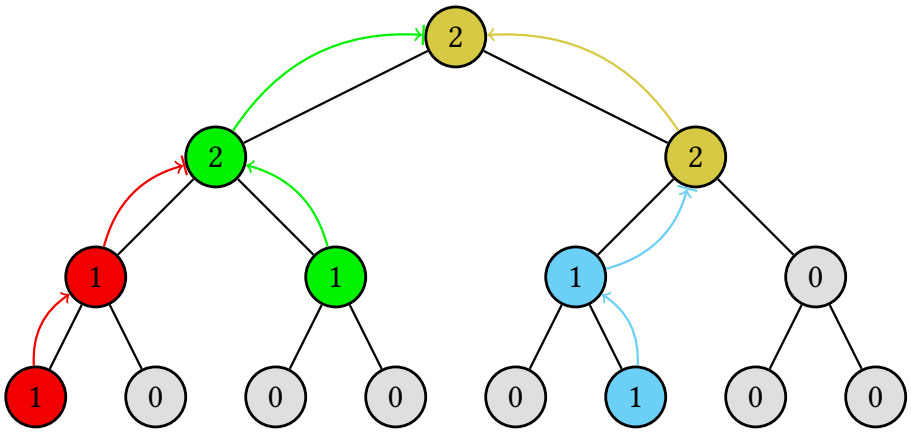


Figure 2.9.: Pass 1

Figure 2.9 visualizes this process. Here every color represents one thread with its start node. The colored arrows indicate the traversal path of each thread. The numbers in the nodes represent the values of the atomic variables after the first pass is executed. It should be noted that this is only one possible outcome of the algorithm. Although the values of the atomics will always be the same, which threads process which nodes can change due to different timings.

The second pass uses the atomic values set up in the first pass. Like in the first pass, every thread traverses the tree from its start node to the root. This time an atomic fetchSubtact action is performed, and the thread only continues the traversal if the previously stored value is 1.

As a result, all atomic variables are automatically set back to 0 after this pass finishes. This setup also ensures that the bounding boxes of all children are updated before the current node is updated. Thus no bounding box is updated more than once.

This second pass is visualised by figure 2.10 on the next page. This is also only one possible outcome of the second pass. Like in the first pass different threads could have terminated sooner, and others could have run for longer due to different timings.

This two-pass approach is implemented without thread synchronization and can thus be easily used on a GPU. Compared to the naive locking implementation, this algorithm only updates each bounding box once, which also improves performance. However, the tree has to be traversed twice to compute the initial atomic variables. Despite that, it has been confirmed through tests that this implementation is faster than the naive locking one.

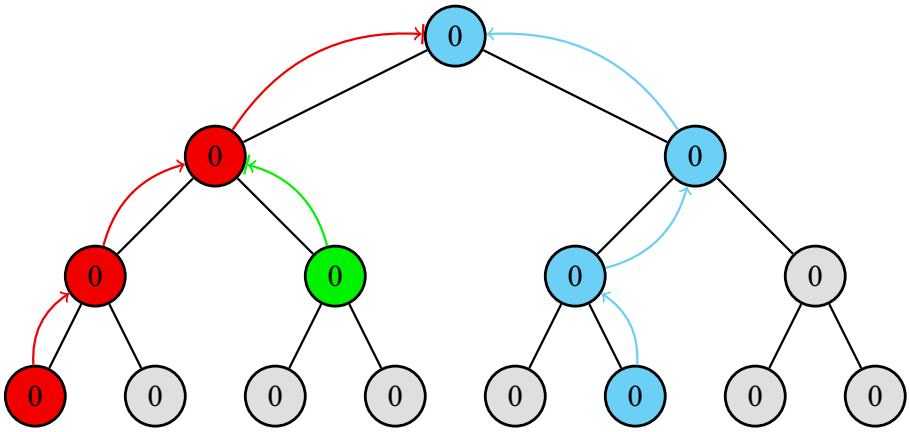


Figure 2.10.: Pass 2

The setup phase in this algorithm is necessary because not the entire tree is traversed. Without it, woukd not be known in the second pass how many threads process a node. However, this knowlege is essential for the second pass to function.

Another approach would be to recalculate the bounding boxes of the entire tree with a bottom up algorithm (start at all leaf nodes → fetchAdd atomic → terminate when $old = 0$). This can be realized in a single pass, since the entire tree is processed. However, test have shown that the two-pass algorithm presented here, is significantly faster than rebuilding the entire tree. This is due to the fact that not the entire tree has to be updated after after applying the patches.

2.4. GPU implementation with CUDA

In the parallel algorithm described in section 2.3 on page 15 several hundred nodes are updated in parallel in each chunk. The algorithm is furthermore completely lock-free and only uses one array of atomic variables for thread synchronization. Thus the algorithm can be implemented in CUDA without any changes.

Since all parts of the algorithm are entirely parallel, the *entire* algorithm can be efficiently implemented on the GPU. As a result, no data has to be exchanged between the device memory and the host memory.

There are however some implementation details which affect the performance of the algorithm on the GPU. The most significant difference between the CPU and the GPU regarding performance is the CUDA memory model.

The fastest memory in CUDA are the registers. Each thread can utilize a few registers for local thread storage. The second fastest memory is the shared memory. This memory is (usually) used for data that is shared across multiple threads in a thread block. Accessing the shared memory is almost as fast as accessing the reqisters, but its size is also limited (around $64bytes$ for each thread). The last and largest memory is the global memory. It

is also the slowest of all three memory types and stores large data structures such as textures and BVH trees. When one thread uses more memory than fits into its registers, the global memory is used for the remaining memory. This can significantly decrease the performance of a CUDA program.

Thus the amount of memory being processed in one thread significantly affects the performance. Because of that, the amount of memory being used should be reduced as much as possible so that most of the data is stored in the fast registers and not the global memory.

Unfortunately, this can not be easily achieved in the central reinsertion algorithm described in section 2.3.4 on page 19. The `BVHPatch` structure for instance stores 10 nodes. This is $10 * 48bytes = 480bytes$ alone for the node patches.

This memory requirement can be reduced to $16bytes$ for one patched node by only storing the information required for the patch. Although the overal performance is increased by this step, this still is not enough to store the entire `BVHPatch` in the registers.

Another trick used in this implementation is the utilization of the relatively fast shared memory. Although the shared memory is usually used to share data between threads, it can also be (ab)used to store thread-local data which would otherwise be stored in the far slower global memory.

The thread local memory was furthermore reduced by using the alternative find node algorithm proposed in section 2.5 which requires far less memory.

The optimizations described here have improved the performance of the CUDA implementation most notably. There are also many smaller optimizations in the source code which can not be listed all here. The results of the final, heavily optimized algorithm can be seen in section 3.3 on page 39.

2.5. Alternative finde node algorithm

One of the main issues with the GPU implementation was the high memory requirement of the priority queue in the findNode algorithm. In this section a modified version of the original algorithm (see section 2.2.3 on page 13) is presented. In this version of the algorithm, the priority queue is approximated with a fixed size array.

Like the original algorithm this “priority array” stores node indices and their cost. Separately the “priority array” index of the current lowest and highest cost is stored.

The priority array is first initialized with the cost of each node set to infinity. Then the root node with a cost of 0 is added.

In each iteration of the algorithm, the current node with the lowest cost is “popped” from the array. When nodes are added to the array, the first node is stored at the old location of the lowest cost node. The second node is stored at the location of the highest cost node.

At the end of the loop, the new positions of the lowest and highest cost nodes are calculated. The algorithm terminates if the cost of all nodes is infinity (the “priority array” is empty).

This approach effectively approximates the priority queue from the original algorithm. Because the size of the “priority array” is limited, this version of the algorithm will not always find the most optimal reinsertion position.

However, the accuracy of the algorithm can be adjusted with the size of the “priority array”. Tests in section 3.4 on page 45 have shown that an array size of 16 is sufficient to achieve nearly the same level of accuracy as the original algorithm.

The following pseudocode illustrates this algorithm:

```
struct FindHelper { uint32_t node; float cost; };

uint32_t findNode(uint32_t n, BVHNodes *nodes) {
  // ... Some other variables ...
  FindHelper PQ[PQ_ARRAY_SIZE]; // The priority array
  float      min      = 0.0f;     // The cost of the root is 0.0f
  float      max      = INFINITY; // Not set
  uint32_t   minIndex = 0;        // The root node is at index 0 in the beginning
  uint32_t   maxIndex = 1;        // Any index besides 0 would work

  for (uint32_t i = 0; i < PQ_ARRAY_SIZE; ++i) { PQ[i].cost = INFINITY; }
  PQ[0] = {0, 0.0f}; // 0 is the BVH root

  while (min < INFINITY) {
    FindHelper curr   = PQ[minIndex];   // Fetch the min node
    PQ[minIndex].cost = INFINITY;       // Remove the node

    // ... Code from the original algorithm ...

    if (pushChildNodes == true) {
      PQ[minIndex] = {currNode->left,  newInduced}; // Save at the empty minIndex
      PQ[maxIndex] = {currNode->right, newInduced}; // Override the highest cost node
    }

    // Calculate the new minIndex and maxIndex
    min = INFINITY; max = 0.0f;
    for (uint32_t i = 0; i < vAltFNQSize; ++i) {
      if (PQ[i].cost < min) { min = lPQ[i].cost; minIndex = i; }
      if (PQ[i].cost > max) { max = lPQ[i].cost; maxIndex = i; }
    }
  }
  return bestNodeIndex;
}
```

In this example, the "priority array" `PQ` and other variables are defined in lines 5 to 9. Next `PQ` is cleared by setting all costs to ∞. Also, the root node 0 with a cost of 0 is added to the array.

Subsequently, the main loop of the algorithm is executed. Here the node with the least cost is first "popped" from `PQ` in line 15 and 16. The node is also "removed" from the array by setting its cost to ∞.

After some calculations from the original algorithm where executed (see section 2.2.3 on page 13), the child nodes are "pushed" to `PQ` (see lines 21 and 22). This is implemented by storing the first child node at the location of the current node (which was removed from the array in line 16).

The second node is stored at the location of the node with the highest cost. This step may override an already stored node if `PQ` is already full.

Finally, the new minimum and maximum nodes are determined (see lines 26 to 30). This is implemented by looping over the entire array to find the new locations.

Although the entire array has to be searched in every iteration, it has been confirmed experimentally that this simple loop is faster than other approaches, like binary heaps. However, this is only true for relatively small arrays with up to 32 elements.

2.6. Realtime applications of this algorithm

Due to the speed of this parallel algorithm, it can be used in real-time applications. More specifically, the CUDA implementation of this algorithm (see section 2.4 on page 22) can be used to optimize the BVH of a dynamic scene in real-time.

Such a scene update procedure can be implemented as follows: First, the vertex positions of the dynamic scene are updated. Then the bounding boxes of the BVH tree are adjusted (see section 2.3.6 on page 20) to reflect these changes. Finally, the parallel optimization algorithm is executed to reduce the SAH.

However, in most scenes, it is not required to process the entire algorithm, since the frame to frame differences in a scene is usually relatively small. Thus a full algorithm step would do more work than required to keep the BVH at the same quality. One solution to this problem is to do just enough work per frame to keep the BVH at the desired quality.

The easiest method for reducing the work per frame is to spread one algorithm step over multiple frames. Since the algorithm is already split into chunks, this spreading can be done by processing only one or two chunks per frame.

This approach for handling dynamic scenes can be applied in all algorithms where a BVH is used. It is thus not limited to ray-tracing and can be used in other applications like collision detection.

2.6.1. Realtime raytracing example

This parallel optimization algorithm was implemented in CUDA to ray-trace dynamic scenes in real-time. The scene update procedure was implemented as described above: After each frame, the vertices of the scene are updated, and the bounding boxes of the BVH are adjusted. Next, the BVH is optimized by processing one chunk. Tests in section 3.6 on page 51 have shown that processing one chunk per frame is sufficient to keep the SAH low.

Furthermore, a stackless ray tracer (see Áfra and Szirmay-Kalos [ÁS14]), implemented in CUDA, was used to render each frame.

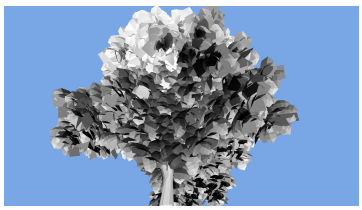
This approach for ray tracing dynamic scenes is faster than a full LBVH rebuild. Only $1ms$ to $2ms$ of additional processing overhead was added each frame. Furthermore, the parallel optimization algorithm produces higher quality BVHs for each frame. The test results are presented in section 3.6 on page 51.

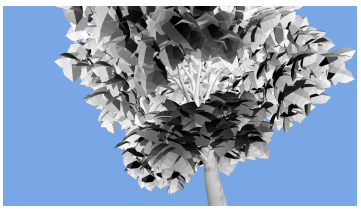
3. Evaluation

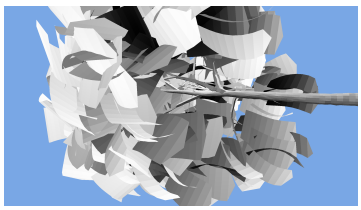
In this chapter the algorithms presented in chapter 2 on page 7 are evaluated. Here the original optimization algorithm from Bittner et al. [BHH13] is compared with the parallel algorithms introduced in this thesis.

All optimization algorithms were tested with two different starting BVHs. The first starting BVH was generated with the partition sweep algorithm introduced by Wald et al. [WBS07]. The second initial BVH was created with the LBVH builder proposed by Lauterbach et al. [Lau+09] with the radix tree optimization from Karras [Kar12]. The number of triangles referenced by each leaf node was limited to 1 for all BVH trees.

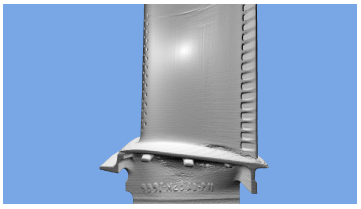
All algorithms were tested with the following 31 scenes (sources in section A.2 on page 62):

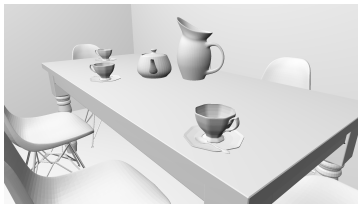

AL05a (316880 tris)


AL05m (168000 tris)

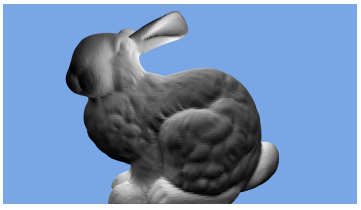

AL05y (38253 tris)

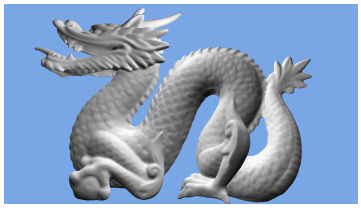

armadillo (345944 tris)


blade (1765388 tris)

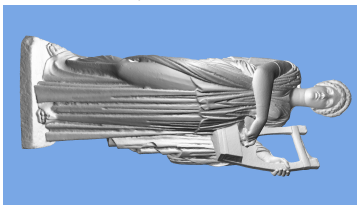

breakfast_room (1347076 tris)

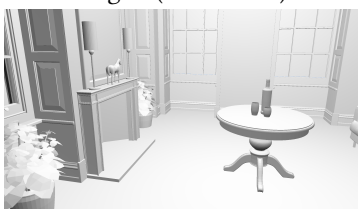

bubsBig (1881542 tris)


bunny (69451 tris)

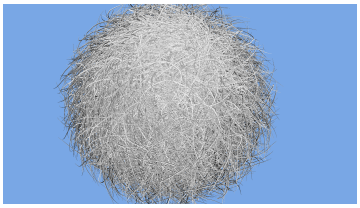

dragon (871306 tris)


exterior (2837209 tris)

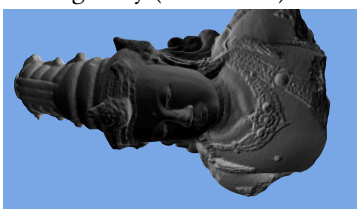

erato (412669 tris)

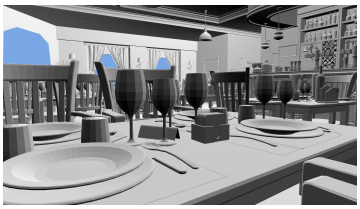

fireplace_room (143173 tris)

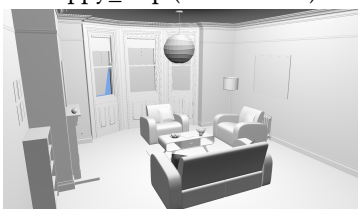

gallery (998941 tris)

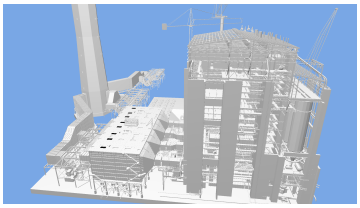

hairball (2880000 tris)


happy_vrip (1087716 tris)

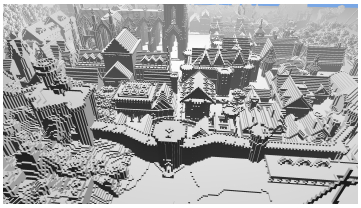

Indonesian_statue (589920 tris)


interior (1020907 tris)


living_room (580535 tris)

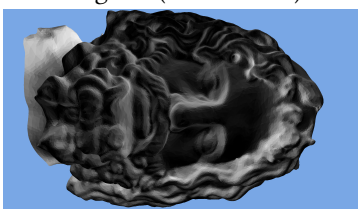

powerplant (12759246 tris)


roadBike (1677520 tris)


rungholt (6704264 tris)

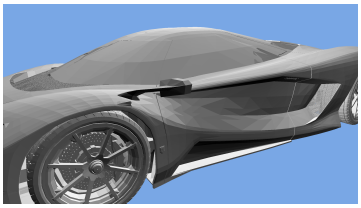

salle_de_bain (1231030 tris)

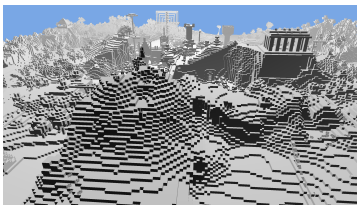

san-miguel (9980698 tris)

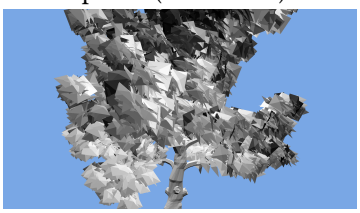

serapis (88040 tris)


sibenik (75284 tris)


sponza (262267 tris)


sportsCar (300603 tris)


vokselia_spawn (1875632 tris)


white_oak (36760 tris)


xyzrgb_dragon (7219045 tris)


xyzrgb_statuette (10000000 tris)

For each scene, the initial BVH is first generated by one of the BVH builders mentioned above. Next, the optimization algorithms are executed to generate the evaluation data. Here the SAH after each algorithm pass (processing one batch) and the execution time is recorded. A total of 32 passes were processed for each scene and algorithm configuration.

All algorithms tested here are implemented in C++17 (for the CPU) and CUDA (for the GPU). The build process for the source code is documented in section A.1 on page 61.

The following table shows the system used to generate the test results:

Compiler	GCC	8.1.0
	CUDA (nvcc)	V9.2.148
System	CPU	Intel(R) Xeon(R) CPU E3-1231 v3 @ 3.40GHz
	GPU	NVIDIA GeForce GTX 1060 6GB
	Linux	4.17.11

Unless explicitly stated otherwise, all tests were done with a batch size of 1% of all nodes, which are selected with the cost measure described in section 2.2.1 on page 10. Furthermore, the LBVH algorithm generally is used to generate the initial BVH. Also, only the average test results for the 31 scenes are presented, since the results of the individual scenes are similar. More substantial differences between scenes are pointed out when they occur.

In the firs part of this chapter (section 3.1 to section 3.4) the impact of the different configuration options for the algorithms are evaluated to determine the best parameter combination for all algorithms. In section 3.5 on page 48 the CPU and GPU implementations of the parallel algorithm (see section 2.3 on page 15) are compared to the original algorithm form Bittner et al. [BHH13].

3.1. Original algorithm

In this section, the original optimization algorithm (see Bittner et al. [BHH13]) is first tested to establish a baseline to compare against. Here the impact of the starting BVH on the optimization process is also examined. Furthermore, the effects of other options like the batch size and sorting are examined.

3.1.1. Initial BVH impact

First, the impact of the original BVH on the entire optimization process is examined. Figure 3.1 on page 31 shows the SAH value after each pass for each initial BVH. In figure 3.2 on page 31 the amount of time each pass uses is presented.

These two diagrams clearly show that the starting BVH impacts the resulting SAH as well the time each pass uses. The BVH created by the partition sweep algorithm from Wald et al.

[WBS07] is already optimal for some scenes and can thus no longer be further optimized. This is the case for the bunny and dragon scenes (the blue and red plots), for instance. In other scenes, where the BVH is not already optimal, the BVH is further optimized, and SAH decreases as a result. The SAH in all tested scenes decreases by 18% on average (see the black plot which was generated by averaging the data from all 31 scenes).

However, when a BVH built with LBVH is used for initially, the SAH is reduced by 63% on average. This is because this initial BVH has a much higher SAH value compared to a BVH built with the partition sweep algorithm. As shown in Figure 3.1 on the next page, both BVHs are improved to nearly the same level. However, the SAH of the partition sweep BVH was always slightly lower than the SAH of the BVH initially built with LBVH. On average, the SAH value for the LBVH is only 6.53% worse compared to the other builder (see the black plots). Furthermore, an optimized LBVH is (on average with the tested scenes) has a higher quality than an unoptimized BVH built with the partition sweep algorithm.

When comparing the time of an optimization pass (see figure 3.2 on the facing page), it is noticeable that optimizing an LBVH is slightly slower than optimizing a partition sweep BVH. Furthermore, the time required for each pass seems to be correlated to the current SAH of the BVH tree. Thus the first few passes, where the SAH is still high, are also slower than the last passes. The find node algorithm may cause this time difference. Here this algorithm does more work in passes with a higher SAH than in passes with a lower SAH.

Because a BVH generated by the LBVH builder has a greater SAH decrease when optimized, this BVH builder is better suited for evaluating the optimization algorithms. Furthermore, building an LBVH is significantly faster than building a BVH with the partition sweep algorithm. The difference in built time is so significant that it is faster to build an LBVH and then optimize it rather than directly building a high-quality BVH. This is true even for the original, sequential algorithm. Also, as can be seen in figure 3.2 on the next page, an optimized LBVH has, on average, a lower SAH than an unoptimized BVH built with Wald et al. [WBS07].

Thus, unless explicitly stated otherwise, a BVH built by LBVH algorithm is used for the initial BVH in the remainder of this chapter.

3.1.2. Batch size

In this subsection, the impact of the batch size on the optimization algorithm is evaluated. The plots in figure 3.3 on page 32 show the difference in SAH values for different batch sizes (*bs* from 0.25% to 3%). In this figure, the SAH values of the first few passes are cut off to focus on the SAH difference in the later passes.

Figure 3.4 on page 33 visualizes the total time the optimization algorithm requires for different batch sizes.

Figure 3.1.: Initial BVH impact: Wald07 vs LBVH

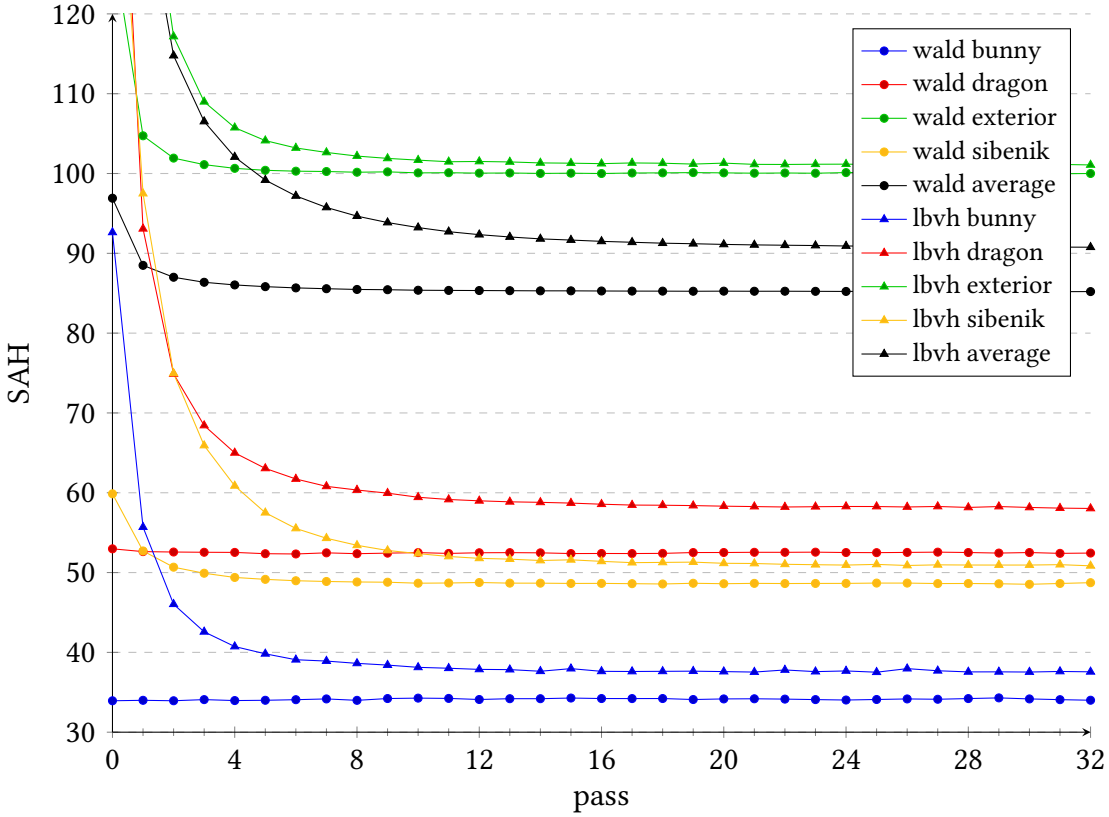


Figure 3.2.: Initial BVH impact: Wald07 vs LBVH (time)

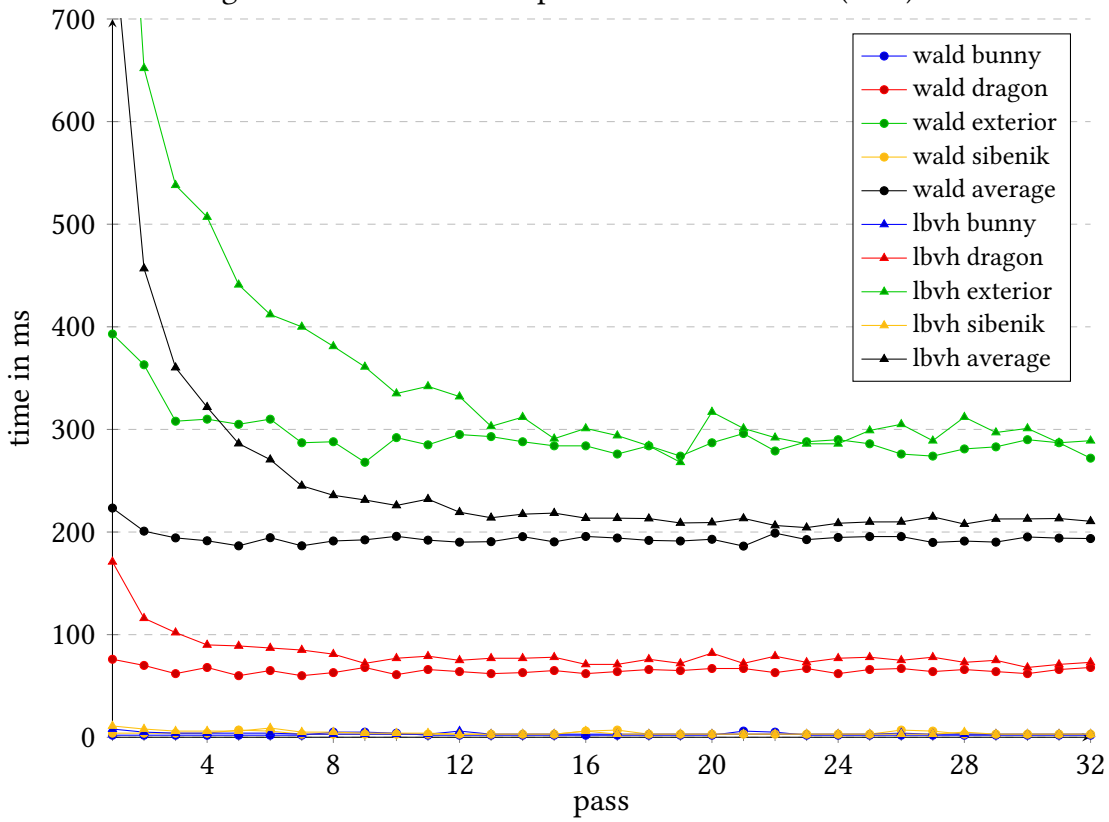
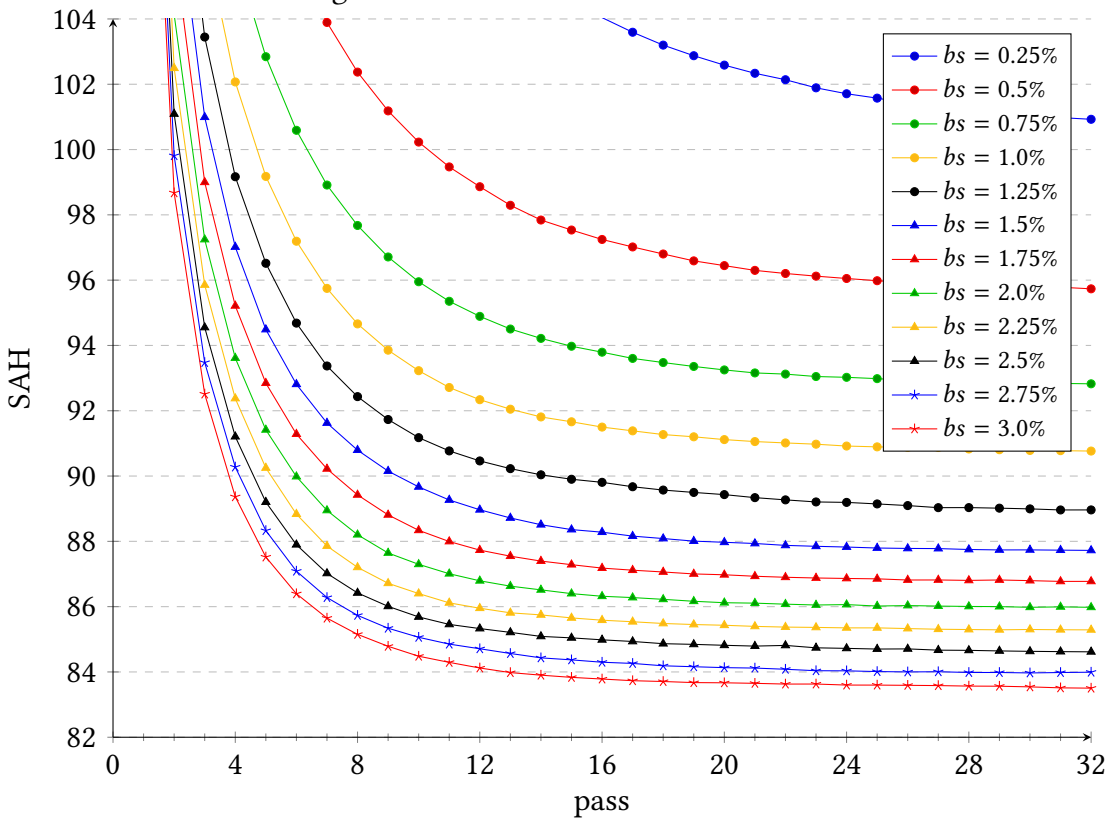


Figure 3.3.: Batch size SAH differences



The first figure clearly shows that the batch size affects how much a BVH can be optimized. Generally larger batch sizes result in lower final SAH values. The batch size also determines how many passes are required for tue SAH to converge. This effect can be observed when comparing the plots for the batch sizes 0.25% and 3.0%.

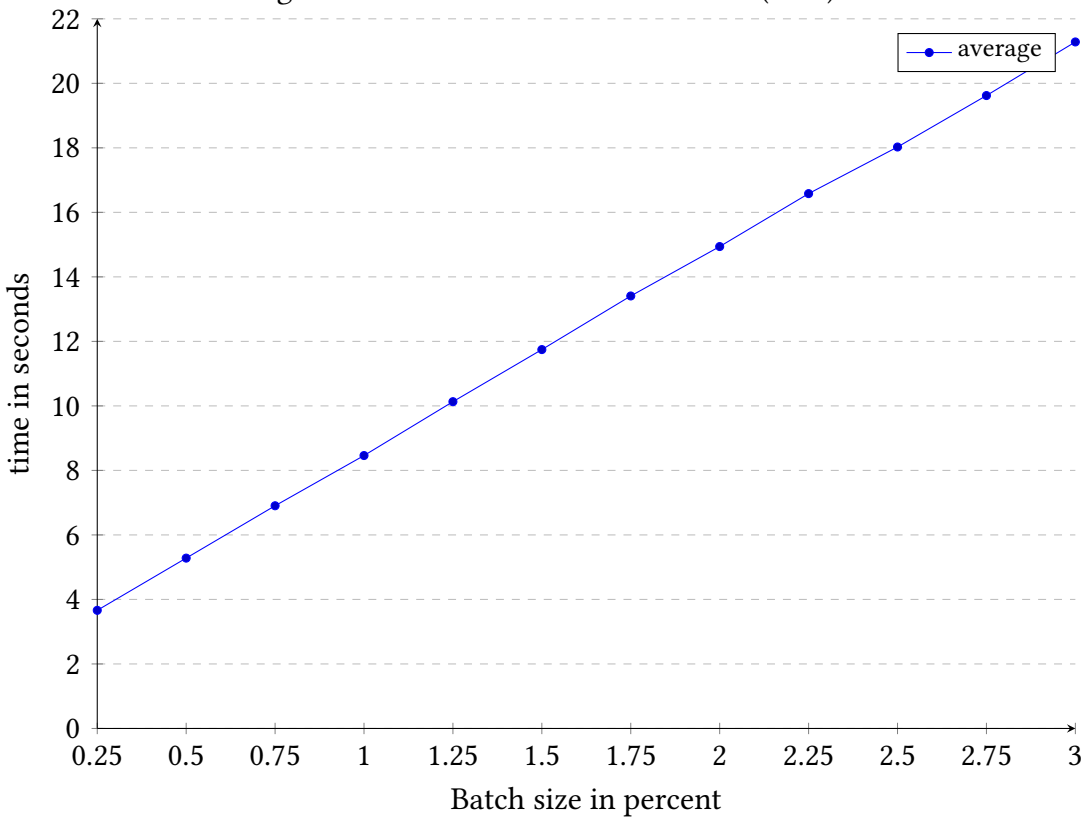
However, this effect is less significant when comparing larger batches. For instance, the gap in SAH values between 2.75% and 3.0% is only $\frac{1}{10}$ of the gap between 0.25% and 0.5%. Furthermore, the time required for processing one batch increases linearly, as shown in figure 3.4 on the facing page. Thus increasing the batch size too much will result in worse overall performance. Tests have shown that the best tradeoff between SAH value and processing time is at a batch size of around 1%.

For this reason, all tests in this chapter are done with a batch size of 1%.

3.1.3. Cost measure

Bittner et al. [BHH13] have already shown that the SAH is reduced faster when using a cost measure for the node selection. Figure 3.5 on page 34 confirms these results. Here the random node selection (green) is compared with the cost measure node selection (blue and red), presented in section 2.2.1 on page 10.

Figure 3.4.: Batch size SAH differences (time)



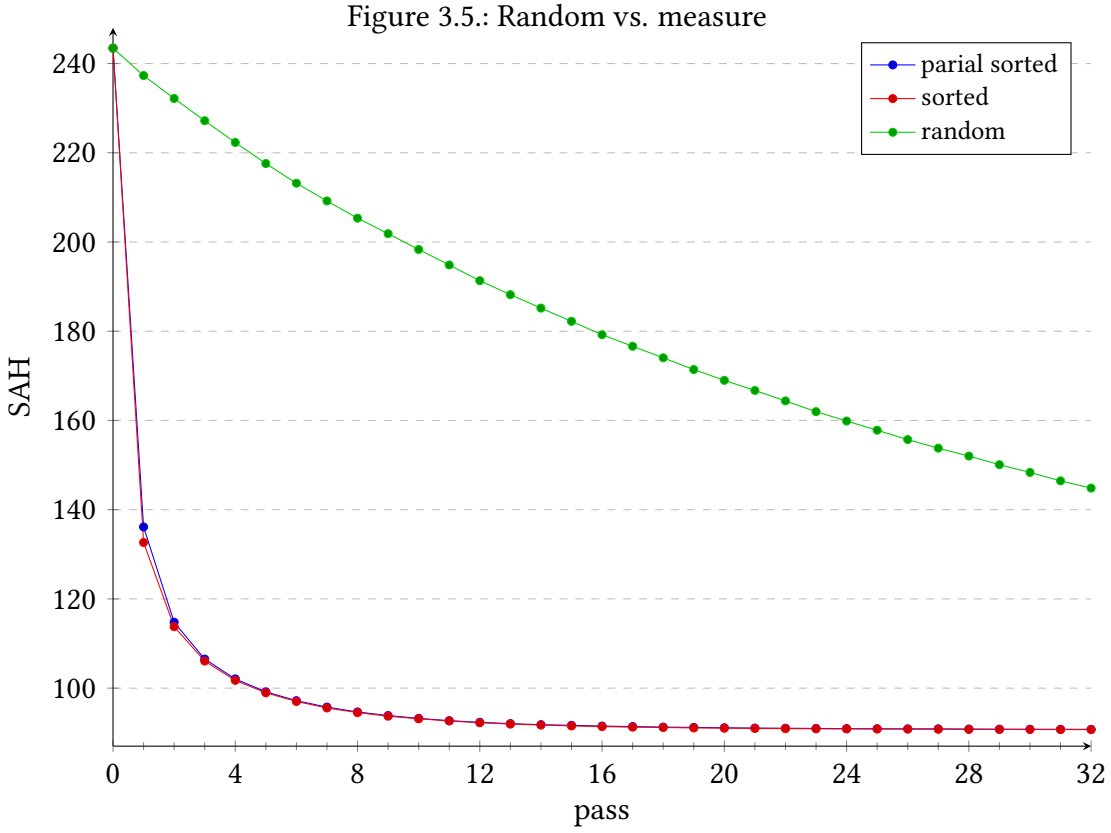
In this figure, the difference between thoroughly sorting (red) and doing a partial sort (blue) is also examined. Here it is noticeable that the SAH decreases slightly faster initially when doing a full sort. This effect is, however, only noticeable in the first three passes. Thus the partial sort is in the vast majority of scenarios the better option since a full sort is more time-consuming than a partial sort.

3.2. Chunk configurations

In this section the parallel optimization algorithm from section 2.3 on page 15 is evaluated. Here only the effects of the different chunk layout options (see section 2.3.1 on page 15) for the algorithm are tested. Thus the original findNode algorithm was used in all tests in this section. These test results are then used to determine the best chunk layout configuration for most use cases.

There are multiple configuration options for chunk layouts, which affect the parallel optimization process. These effects are most noticeable in the number of patches that need to be discarded due to conflicts. The chunk layout also impacts the SAH decrease during optimization and the time required to process one batch.

Figure 3.5.: Random vs. measure



3.2.1. Optimization process overview

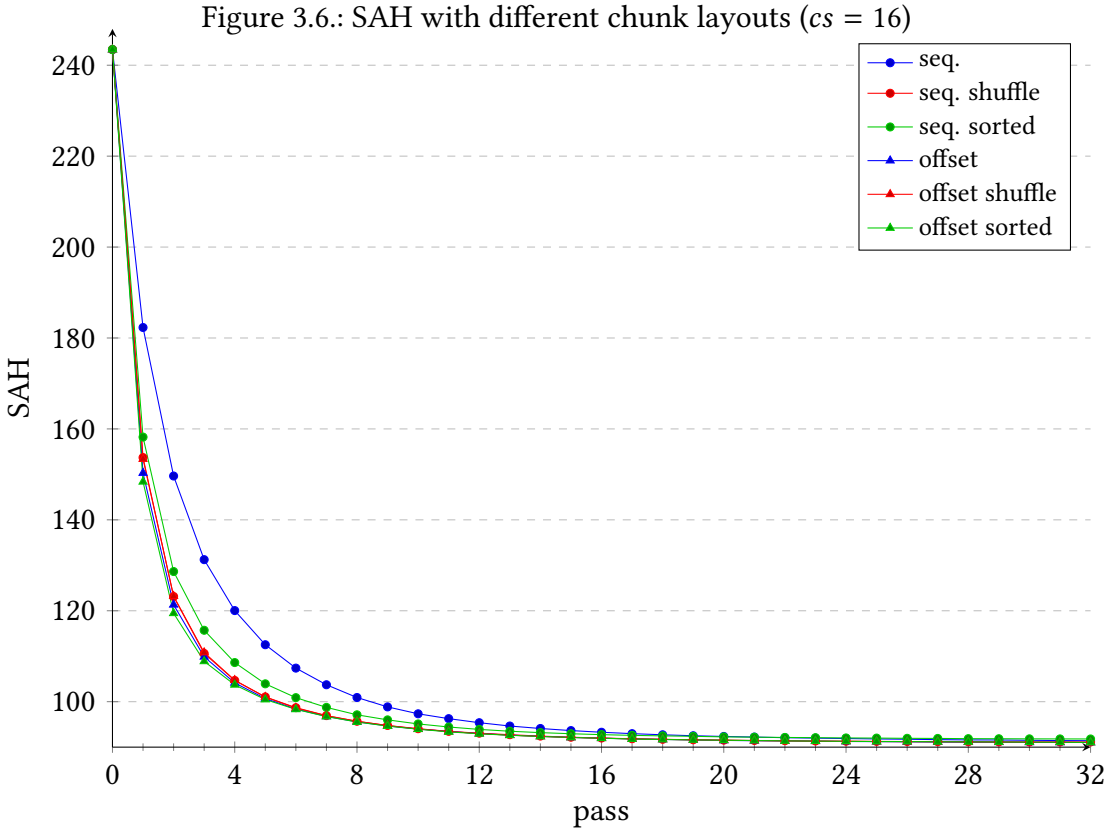
First, the impact of these different layout options on the entire optimization process is examined. Here, figure 3.7 on page 36 visualizes the optimization process for different number of chunks (= *nc*) and in figure 3.6 on the facing page the other layout options are examined at a fixed number of chunks of $nc = 16$.

In this figure, the first three plots (*seq., seq. shuffle, seq. sorted*) are all generated with the following chunk layout. In the first sequential plot (*seq.*), the batch was selected with a partial sorting algorithm. In the next plot (*seq. shuffle*) the nodes in the batch were shuffled after the initial selection. For the last sequential plot (*seq. sorted*), the batch was generated by doing a full sort instead of a partial sort. In the next three plots (*offset, offset shuffle, offset sorted*) the offset layout was used. Here the batch selection was performed as described for the sequential layouts.

Both figures clearly show that there is a difference in the SAH value initially, but it also shows that this difference becomes less significant in later passes. In fact, there is no significant difference in the SAH value after the $16th$ pass. At this point, the maximum observed difference in SAH was 3.58%, and after the $32nd$ pass the maximum difference shrunk further to 1.92%. This difference is insignificant compared to the total SAH reduction.

Although there is virtually no difference between the chunk layouts in later passes, there is still a noticeable difference initially. Thus the chunk layout primarily affects how *fast*

Figure 3.6.: SAH with different chunk layouts ($cs = 16$)



the SAH is reduced. Such "fast" layouts are generally preferable because they reach the desired optimization level of a BVH in fewer passes than "slower" chunk layouts.

3.2.2. Detailed comparison

The next three figures are used to differentiate the chunk layouts better and find the best layout for most use cases. All three figures evaluate the different chunk layout types (see description for figure 3.6) at different numbers of chunks.

In figure 3.8 on the following page the SAH value after the first pass is compared for the different chunk layouts. The goal of this graph is to compare how fast the SAH is reduced in the different layouts. The first pass was chosen because the differences between layouts are most noticeable after this pass.

In figure 3.9 on page 37 the percentage of discarded patches is evaluated for the different layouts. This percentage was generated by counting the number of discarded patches over the entire optimization algorithm (32 passes).

The first figure shows that the difference in the SAH value is noticeable for most layouts after the first pass. Here the sequential layout (*seq.*) produces the worst results for all number of chunks. Shuffeling (*seq. shuffle*) or sorting (*seq. sorted*) the batch significantly increases the SAH improvement for the sequential layout. However, this figure also shows that the offset layouts are generally better than the sequential layouts.

Figure 3.7.: SAH with different number of chunks (offset layout)

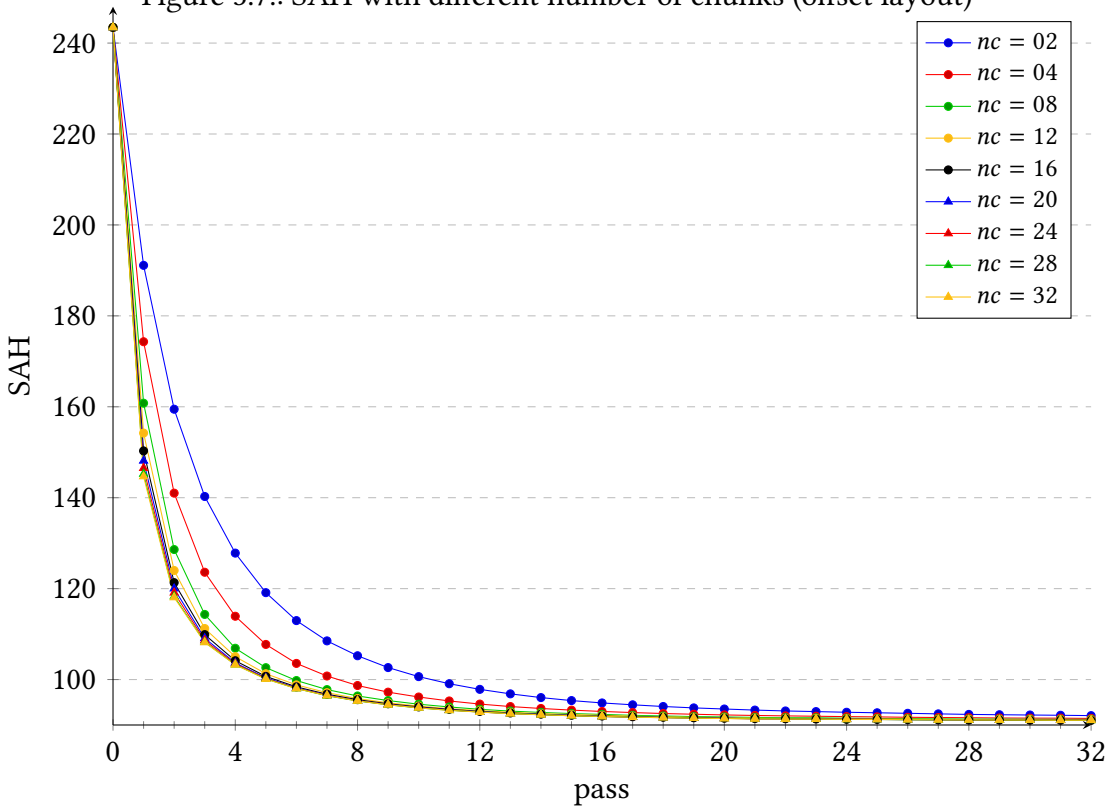


Figure 3.8.: SAH after the first pass for different number of chunks

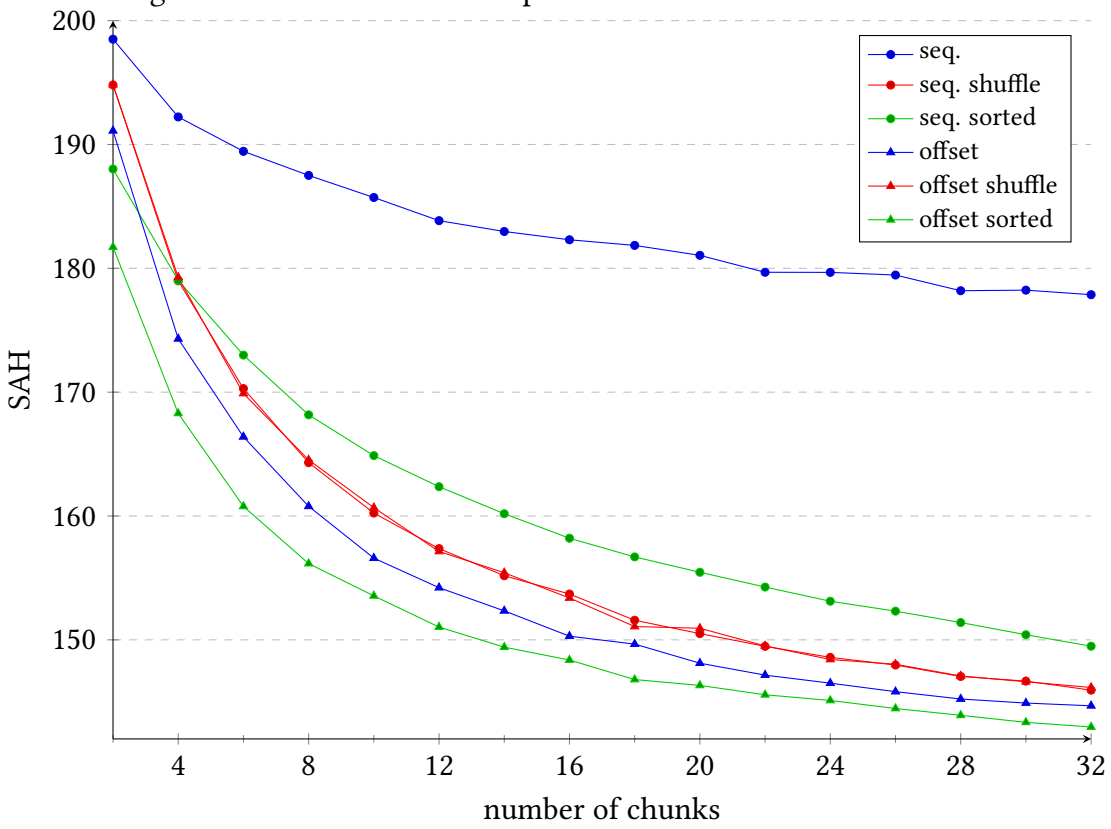


Figure 3.9.: Percentage of discarded patches at different number of chunks

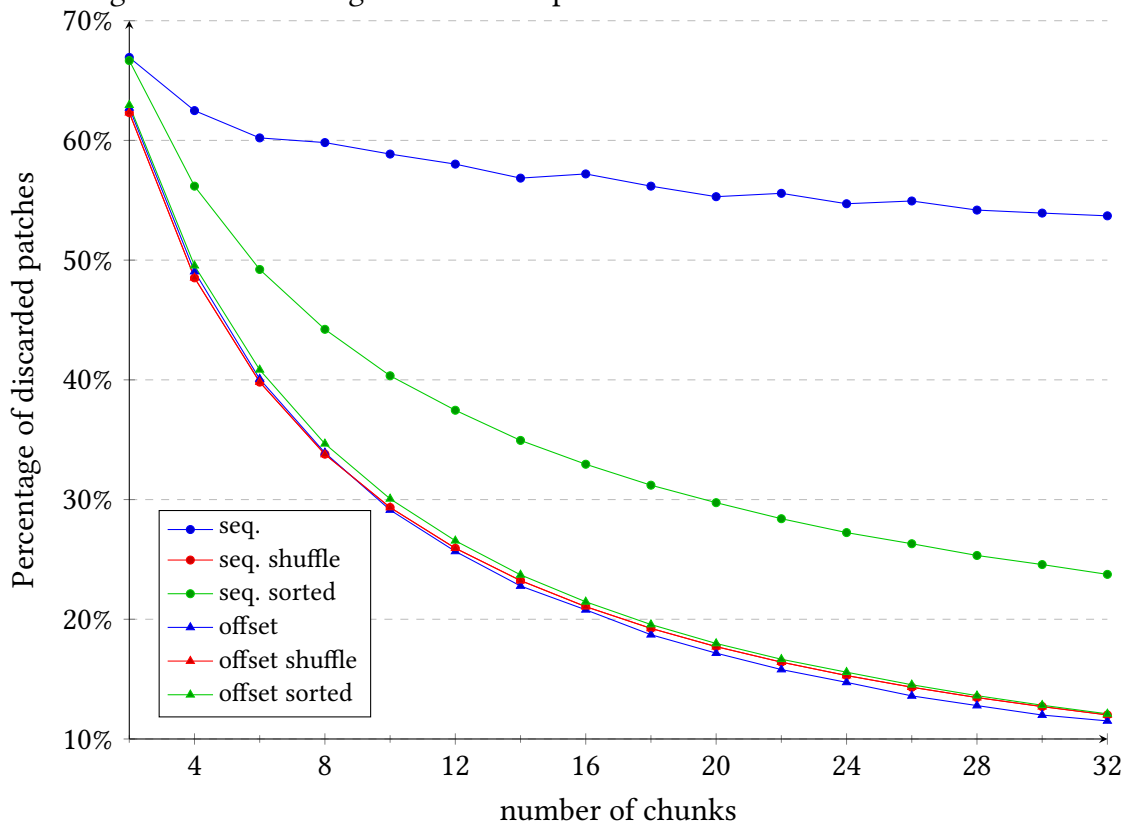
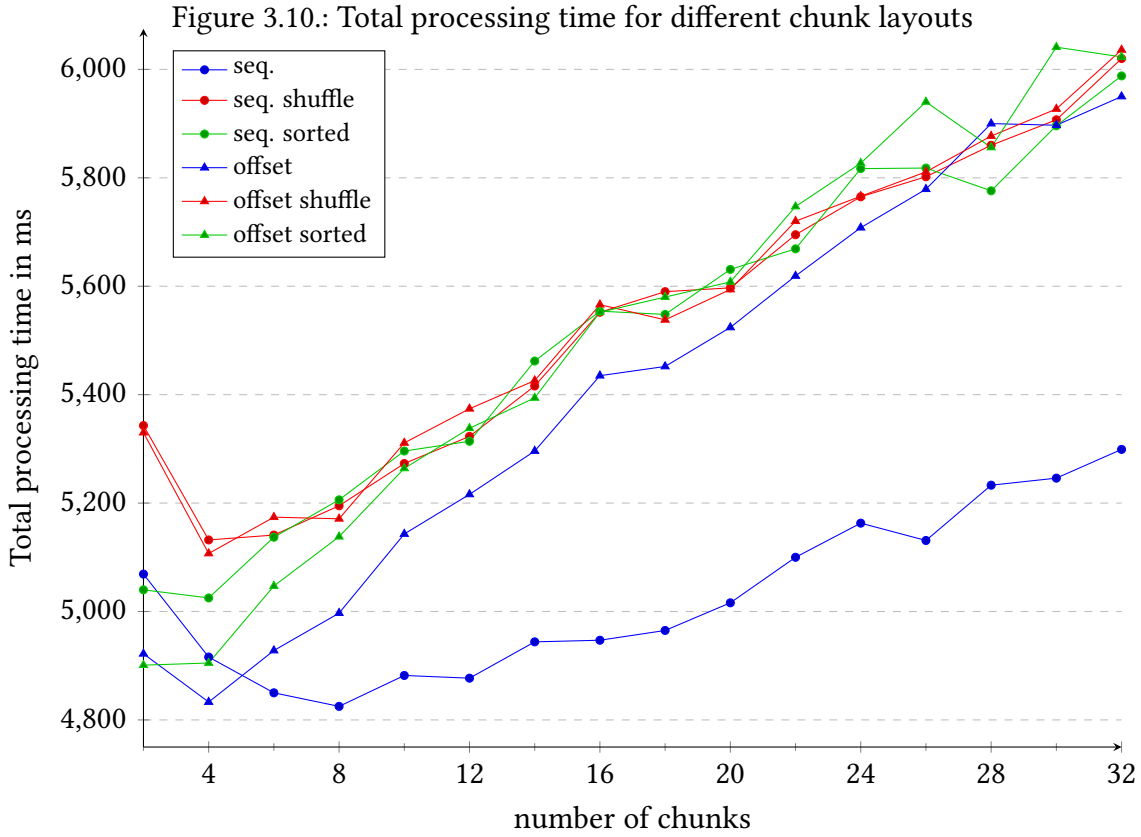


Figure 3.10.: Total processing time for different chunk layouts



These SAH values also roughly correlate to the number of discarded patches (see figure 3.9 on the preceding page). There are, however, some differences: The relative gap between the percentage of discarded patches between the sequential sorted and the offset plots is significantly bigger than the gap in SAH values. There is, furthermore, virtually no difference in discarded patches for the offset plots, while the difference in SAH values is noticeable.

When comparing the layouts, it is noticeable that there is virtually no difference between the sequential and offset layout with a shuffled batch (*seq. shuffle and offset shuffle*). This is not surprising since it should not make a difference how the randomized array is accessed. The nodes are randomly distributed over the chunks in either case.

However, There is also a surprising result: The offset layout (*offset*) reduces the SAH faster than the shuffled layouts (see figure 3.8 on page 36). The SAH is also further improved when the batch is sorted (*offset sorted*). This is despite the fact that all mentioned layouts produce a virtually identical amount of discarded patches.

The partial sorting algorithm can explain these results. This type of sorting algorithm mostly preserves the order of the original input. Thus nodes, which are relatively close to each other in the BVH tree, are also relatively close in the selected batch. As a result, more patches in a chunk conflict with a sequential layout, because of this spatial proximity. The offset layout breaks this spatial proximity. Thus the amount of discarded patches is drastically reduced.

Furthermore, when using a sorted batch, the nodes with the highest cost measure are processed first, and their patches are thus less likely to be discarded. These nodes are also responsible for the highest SAH decrease. Thus the SAH is reduced more when the batch is (partially) sorted.

3.2.3. Runtime comparison

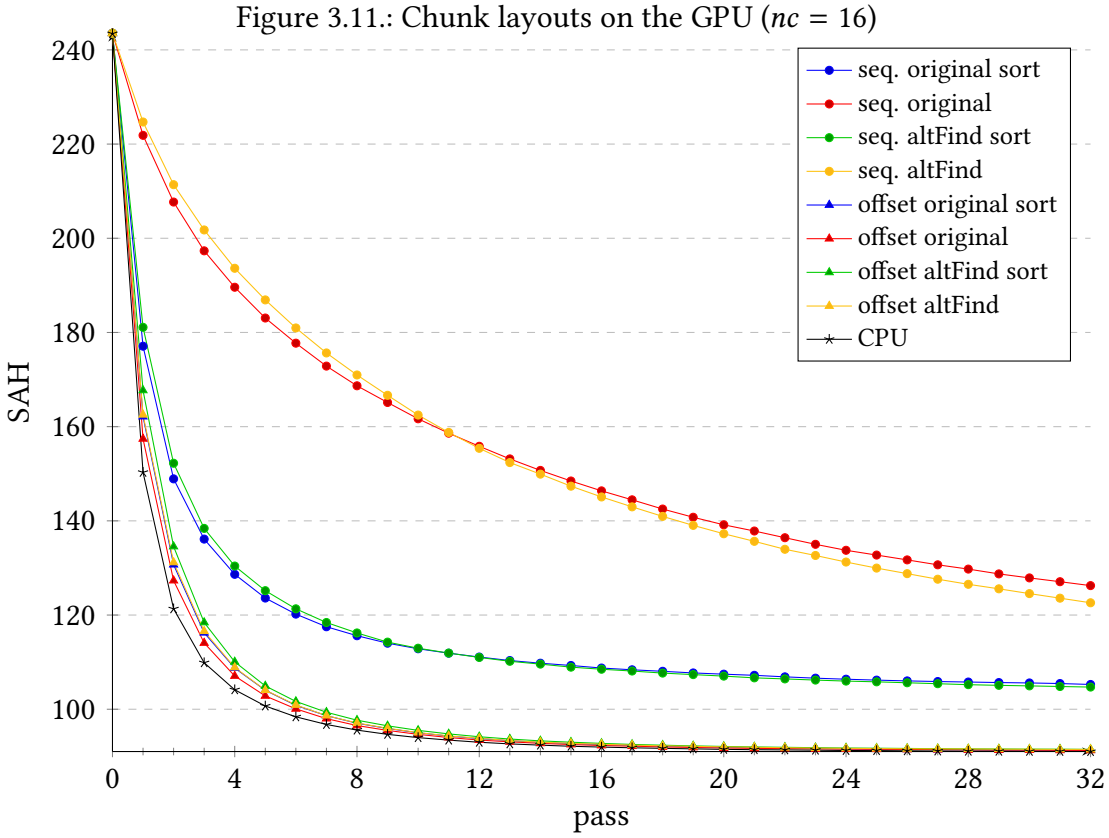
However, there is a difference in processing speed for the different chunk layouts, as shown in figure 3.10 on the preceding page. In this figure, the time for the entire optimization algorithm (32 passes) was measured.

All of the plots, except the one for the sequential layout (*seq.*), seem to follow a linear trend. Here the total time spent increases with the number of chunks. The figure also shows that there are differences in speed for the different layouts.

From these chunk layouts, the offset layout (*offset*) is the fastest. This is due to the fact that no further processing has to be done on the batch. All other layouts require either sorting or shuffling the nodes in the batch. This step adds further processing overhead, which results in the higher overall times shown in this figure.

The sequential layout (*seq.*) was ignored in this comparison because it has a high number of discarded chunks, which directly impact the processing time of the algorithm. This makes the comparison to the other layouts not feasible.

Figure 3.11.: Chunk layouts on the GPU ($nc = 16$)



3.2.4. Conclusion

Thus the offset layout with 16 chunks seems to be a good choice for most scenarios. It provides the second highest SAH decrease and a relatively low percentage of discarded patches (around 20%). It is also faster than other comparable chunk layouts since the batch is not processed further (through shuffling or sorting). Also, 16 chunks seem to provide a good tradeoff between the decrease in SAH and the required processing time.

3.3. CUDA implementation

In this section, the different algorithm configurations for the CUDA implementation are evaluated. Because the alternative findNode algorithm (short altFindNode, see section 2.5 on page 23) significantly increase the performance on the GPU, it is also included in this evaluation. The size of the "priority array" of the altFindNode algorithm was set to 16 in all tests. This value was chosen because of the test results in section 3.4 on page 45.

Figure 3.12.: Number of chunks on the GPU (offset, altFindNode, no sort)

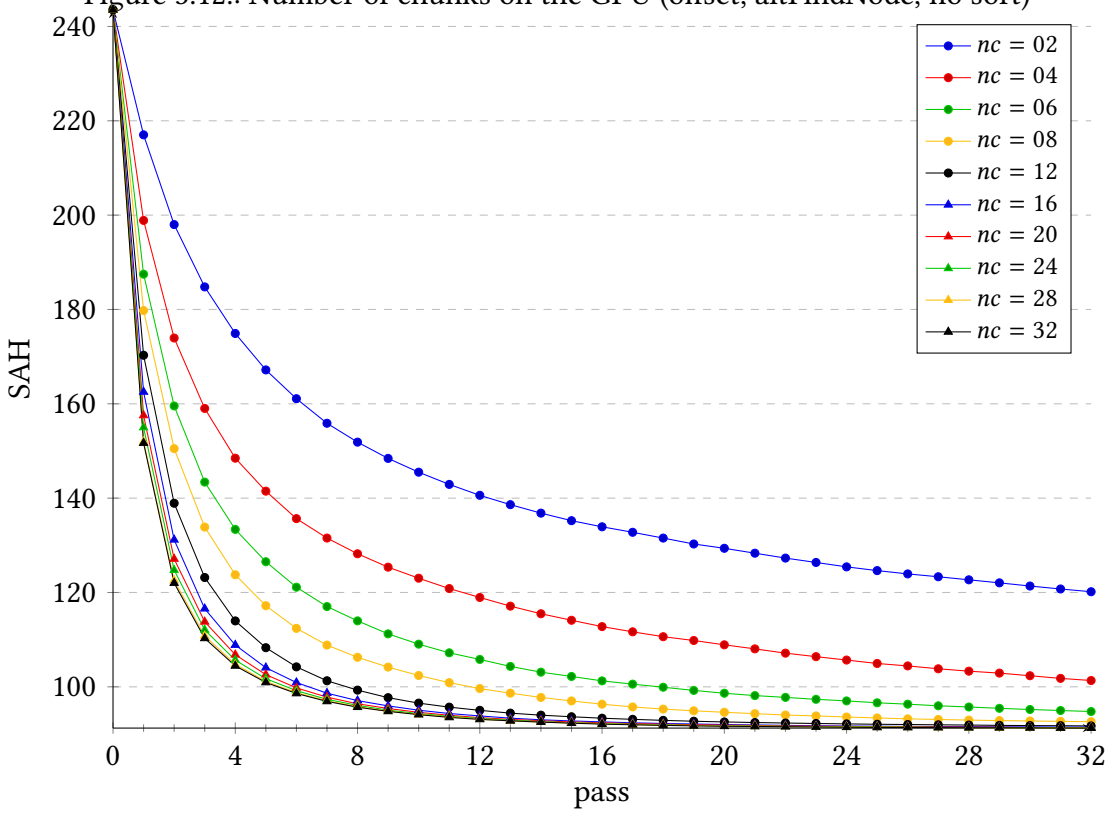


Figure 3.13.: SAH after the first pass

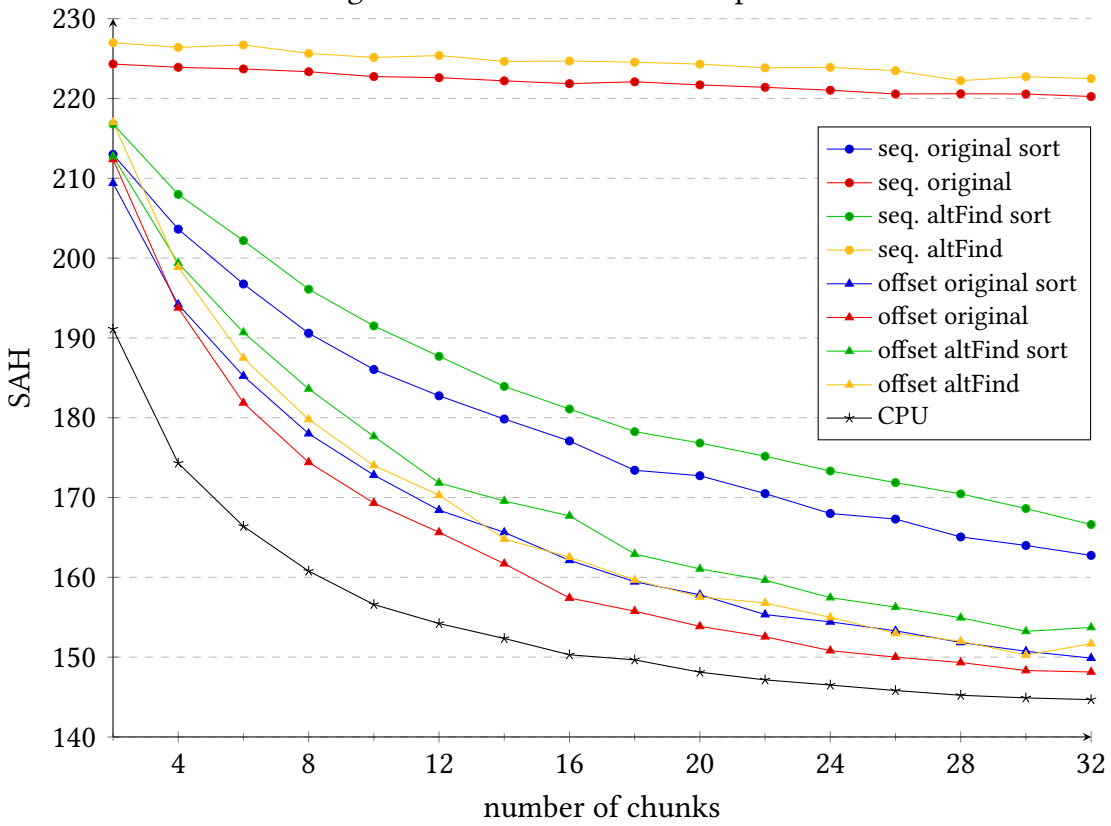


Figure 3.14.: Percentage of discarded patches

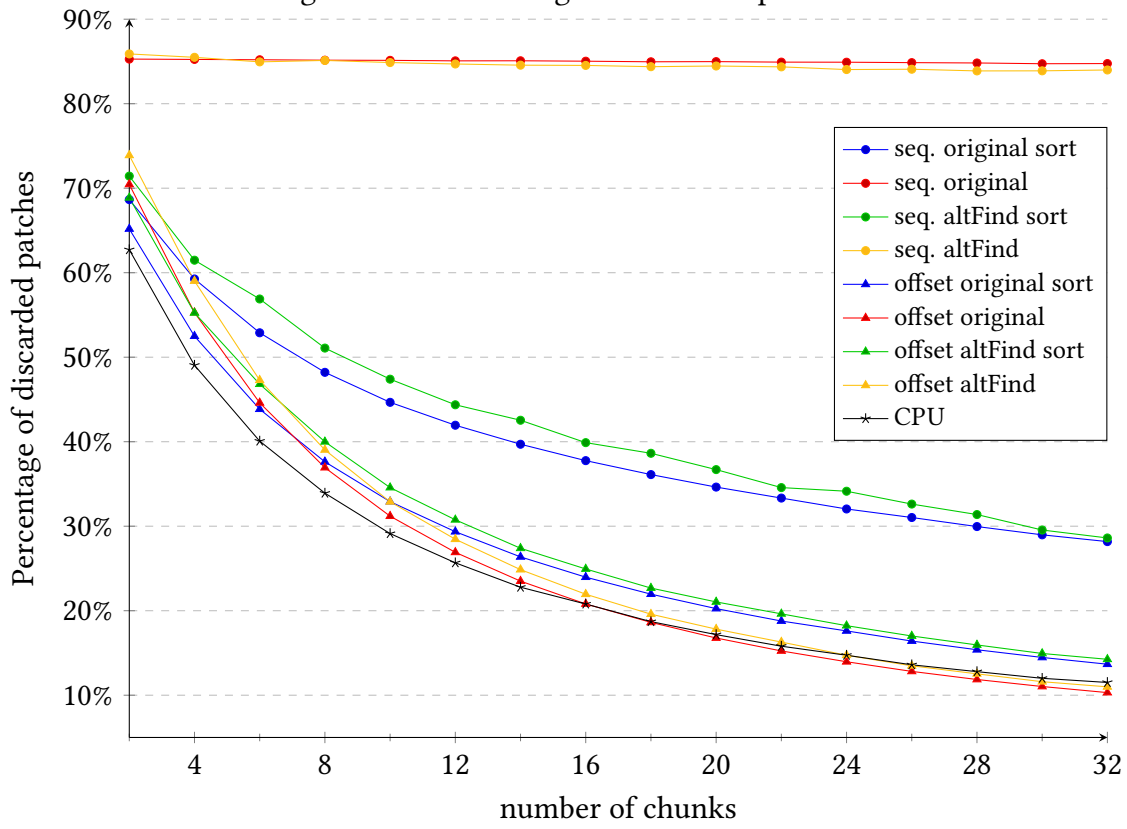


Figure 3.15.: Processing time for the different number of chunks

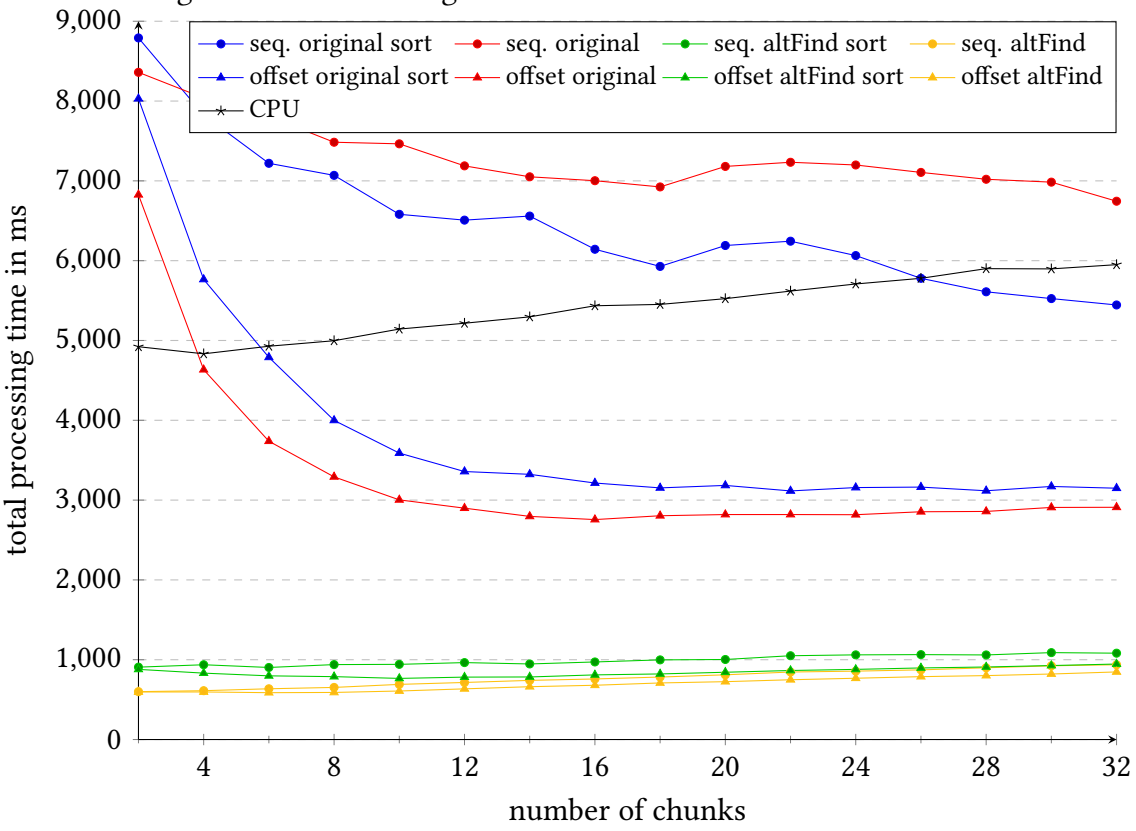
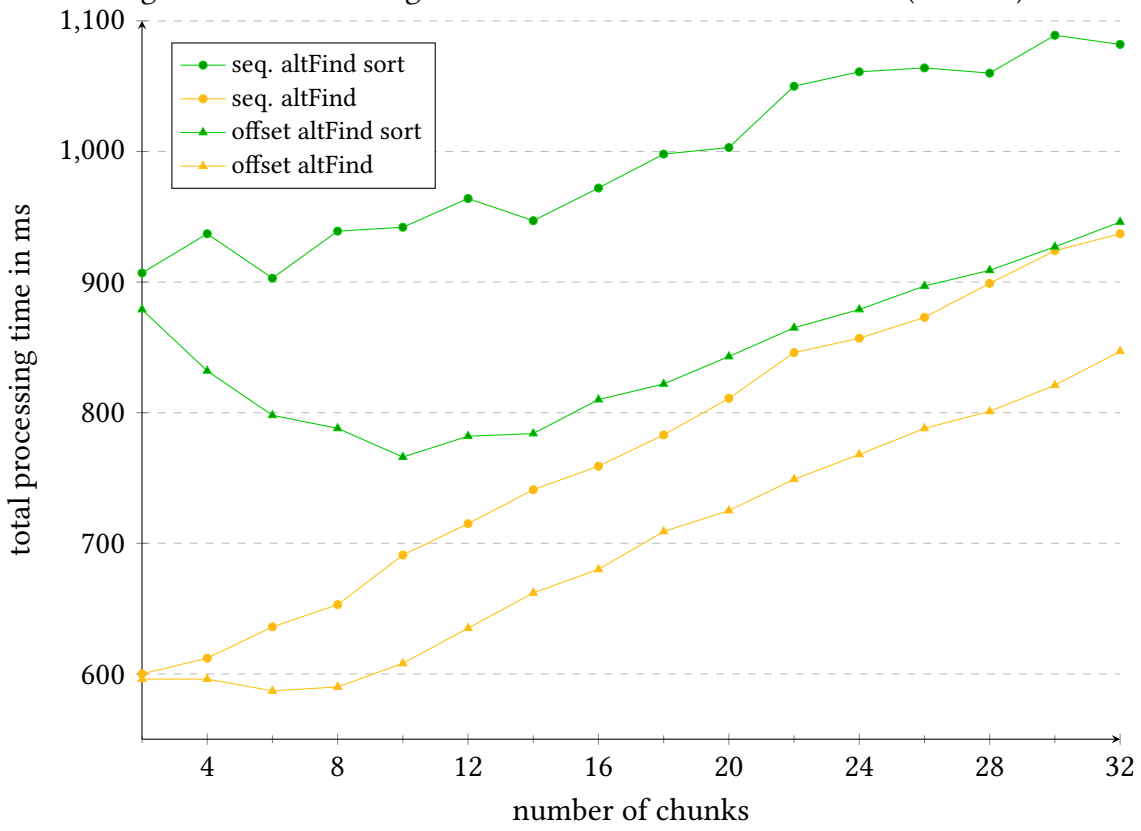


Figure 3.16.: Processing time for different number of chunks (zoomed)



3.3.1. Optimization process overview

When comparing the CPU and CUDA version of the algorithm, differences in SAH reduction between the implementations are apparent: Figure 3.11 on page 39 shows the SAH reduction for multiple chunk layout configurations on the GPU. The results for the offset layout with neither sorting or shuffling, generated on the CPU is also included for reference (*CPU*). These results were generated for 16 chunks per batch.

Here the first four plots (*seq.*) were generated with the sequential layout (see section 2.3.1 on page 15). The offset layout was used in the next four layouts (*offset*). Whether the batch was sorted or only partially sorted is indicated by the *sort* keyword at the end of the plot description. The *original* and *altFind* keywords indicate which algorithm was used to find the reinsertion position.

This figure shows that the results for all sequential layouts (*seq.*) are significantly worse on the GPU compared to the CPU version (compare figure 3.6 on page 35). It is furthermore noticeable that the plots for the sequential layouts do not converge to the same level as the other tested chunk layouts. Sorting the initial batch (*sort*) improves the SAH reduction for these layouts. However, even with this improvement, the SAH is not reduced to the same level as the same level as the offset layouts. Thus all (nonrandomized) sequential layouts are worse than offset layouts in the vast majority of scenarios.

Another difference between the CPU and GPU version can be observed in the SAH reduction for the different number of chunks. Figure 3.12 on page 40 visualizes the optimization process on the GPU for the offset layout with the altFindNode algorithm (*offset altFind*). This graph also shows different convergence behavior, compared to the CPU implementation (see figure 3.7 on page 36).

In this figure, the gaps between the plots for $nc \leq 16$ are noticeable wider, compared to the CPU version. It should also be noted that the plots for $nc \leq 6$ do not converge during the tested period of 32 passes. However, the plots for $nc \geq 16$ are virtually identical and perform similarly to the CPU implementation. Thus, the number of chunks has a more significant impact on the GPU than on the CPU.

3.3.2. Detailed comparison

This difference is further highlighted in figure 3.13 on page 40. Here the SAH after the first pass for the different layouts is evaluated. This figure also shows that all sequential layouts (*seq.*) perform worse than the offset layouts (*offset*). However, also the offset layouts never reach the same level as the reference CPU plot (*CPU*).

Though, this is not a major downside of the GPU implementation, since the SAH for all offset layouts converges to the same value (see figure 3.11 on page 39). Thus the CPU and GPU implementation only differ in the "speed" (relative number of passes) of the SAH reduction. As a result, the GPU implementation can reach the same SAH value when one or two additional passes compared to the CPU implementation are executed.

Figure 3.13 on page 40 also shows that, compared to the original, the altFindNode algorithm (*altFind*) performs slightly worse for all algorithms. Furthermore, sorting the batch (*sort*) also negatively affects the SAH for the offset layouts. Thus, the offset layout with the original algorithm decreases the SAH the "fastest". However, the altFindNode algorithm offers better overall performance, since it is significantly faster than the original (see below).

The number of discarded patches can mostly explain the difference in SAH reduction between the GPU and CPU implementations: Figure 3.14 on page 41 shows the total percentage of discarded patches for the different layouts. Here roughly 85% of the patches are discarded with the nonsorted sequential layouts. Although sorting decreases this percentage, it is still significantly higher compared to all offset layouts (for $nc \geq 6$).

The number of discarded patches for the offset layouts also correlate well with their SAH reduction performance. Here the sorted layouts (*sort*) have a slightly higher percentage than the partially sorted layouts. The original algorithm also slightly outperforms the altFindNode algorithm. However, the impact of the sorting algorithm is significantly higher compared to the impact of the findNode algorithm.

Similar to the parallel CPU implementation, this behavior can be explained with the used batch layout: When a partial sorting algorithm is used, the order of the original, unsorted

array is mostly retained. Thus nodes that are relatively close to each other in the BVH tree are also relatively close to each other in the selected batch array.

With the sequential chunk layout, most nodes in the chunks are thus relatively close to each other in the tree. This significantly increases the probability of a conflict between patches. Sorting the batch breaks this spatial locality, and fewer nodes are discarded as a result. However, as shown in figure 3.14 on page 41, sorting is still less effective compared to the offset layout.

3.3.3. Runtime comparison

In figure 3.15 on page 41 the total processing time for all 32 passes of the GPU implementation is evaluated. Since the bottom four plots are cramped together, a zoomed version of the plot is provided in figure 3.16 on page 42.

Both figures show that the altFindNode algorithm is significantly faster on the GPU compared to the original. They are also 5 to 7 times faster than the parallel CPU implementation (*CPU*).

As shown in the zoomed figure, the time required for the optimization processes rises mostly linearly with the number of chunks (for $nc \geq 8$). This linear trend is similar to the linear trend observed with the CPU implementation. However, this is not true for the original findNode algorithm (*original*). Here the time spent varies significantly compared to the altFindNode algorithm (*altFind*). Furthermore, the processing time is falling, rather than increasing in some cases with the number of chunks.

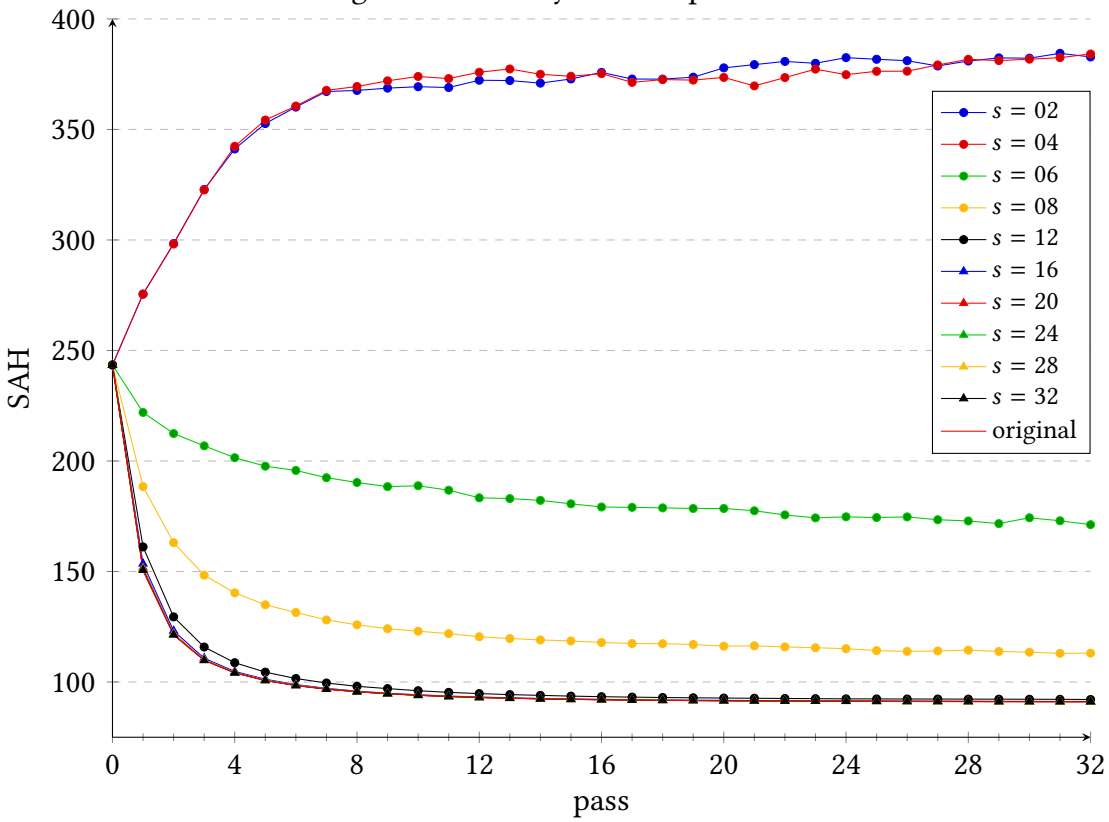
These results for the original algorithm (*original*) cannot be explained by just one part of the entire optimization procedure. Since the only change between these plot groups is the findNode algorithm, the top four plots should be offset by a fixed amount compared to the bottom four plots. This is, however, not the case. Thus, it is more likely that multiple factors influence the processing time at once.

It should also be noted that the processing time with the original algorithm does not provide a significant enough improvement in processing time compared to the parallel CPU implementation. In fact, both nonsorted layouts perform worse on the GPU compared to the CPU.

When comparing the layouts with the altFindNode algorithm (*altFind*), it is apparent that sorting the batch is slower than performing a partial sort. The offset layout (*offset*) is furthermore faster than the sequential layout (*seq.*). Thus the partially sorted offset layout with the altFindNode algorithm (*offset altFind*) is the fastest chunk layout configuration on the GPU.

This layout also achieves the same SAH reduction as the reference CPU implementation, as shown above. It thus offers the best combination of runtime performance and SAH reduction of all tested layouts. Similar to the CPU implementation, around 16 chunks per batch also provide the best tradeoff between SAH reduction “speed” and runtime performance.

Figure 3.17.: Array size comparison



3.4. Alternative findNode array size

In this section, the alternative findNode algorithm (see section 2.5 on page 23) is evaluated. All tests were run on the CPU with the offset layout (without sorting or shuffling) and 16 chunks per batch.

First, the SAH reduction of this optimized algorithm is evaluated. Figure 3.17 shows the SAH after each pass for multiple array sizes ($s = x$) of the "priority array". The original algorithm is also included in this figure for comparison.

This figure shows that the size of the "priority array" does affect the results of the alternative findNode algorithm and that it has thus an impact on the entire optimization process. Small values for the array size ($s \leq 12$) decrease the amount the SAH improves in each pass. If the size is to small ($s \leq 4$), the SAH can even increase.

However, the alternative find node algorithm produces virtually the same results as the original algorithm once the array size is big enough. The tests presented here, have shown that this point is reached for $s = 16$. Increasing the array size any further does not further improve the SAH. The plots in figure 3.17 illustrate this fact. Here the plots for $s \geq 16$ are directly on top of each other, with virtually no SAH difference. They are, furthermore, never better than the original algorithm.

Figure 3.18.: SAH for different queue sizes

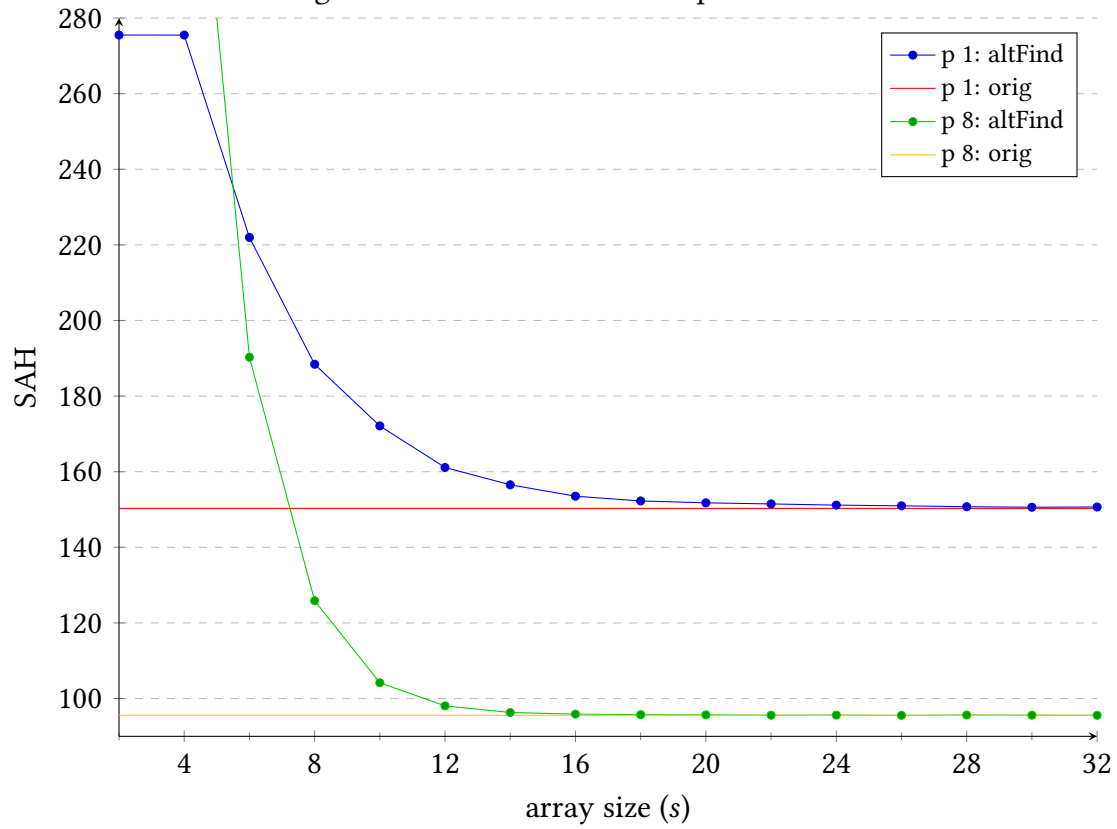
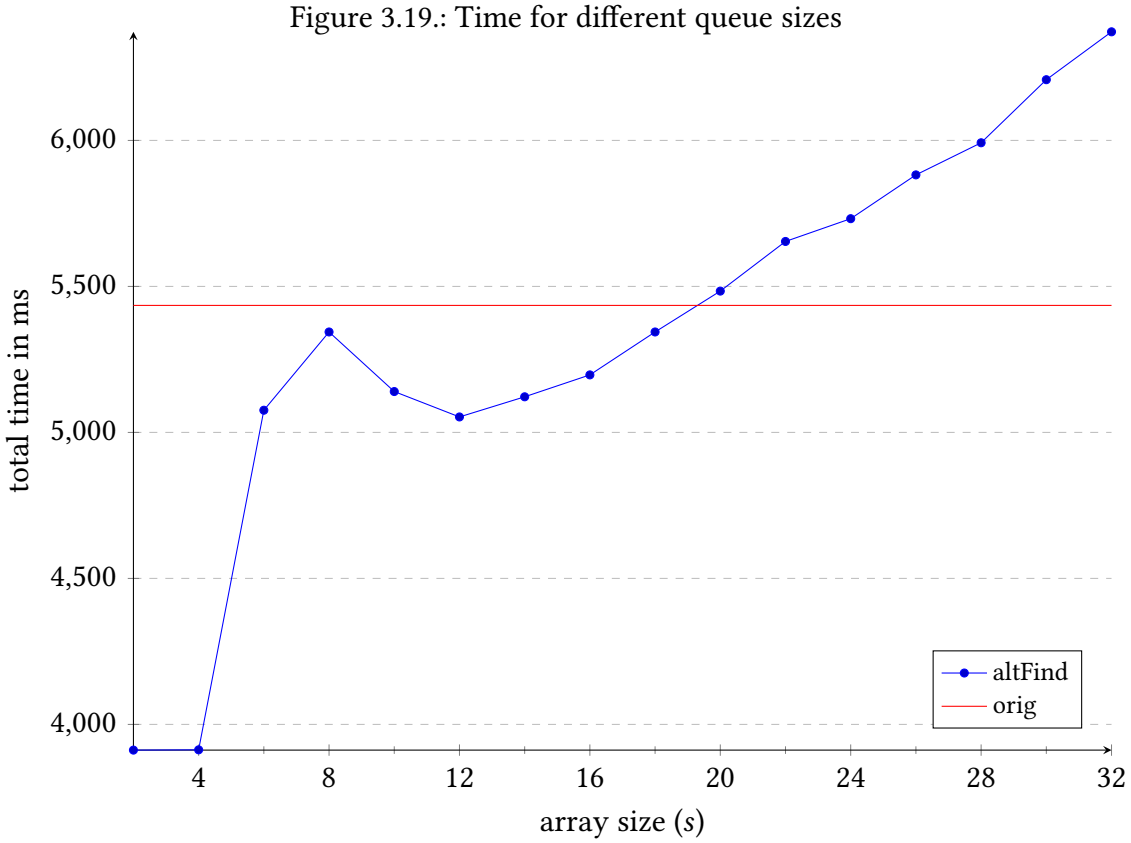


Figure 3.19.: Time for different queue sizes



This is due to the nature of the changes in the alternative findNode algorithm: Here the full priority queue of the original is replaced with a fixed sized array, that only approximates the priority queue. Other parts of the algorithm were not changed. Thus, the new algorithm can not produce better results than the original algorithm.

This is further illustrated in figure 3.18 on the facing page. In this figure, the alternative findNode algorithm (altFind) is compared to the original algorithm (orig). Here the SAH values for different array sizes were recorded after the first (p 1:) and eight (p 8:) pass. The SAH value of the original algorithm is visualized as a line since it does not have a fixed sized array.

Here the SAH values of the alternative algorithm converge to the SAH value of the original algorithm for $s \longrightarrow \infty$. Furthermore, the plot converges faster in later passes, compared to the first pass (see the relative difference in SAH at $s = 12$).

There is, furthermore, virtually no difference in SAH between the original algorithm and the alternative algorithm for $s \geq 16$, in pass 8. Thus, the alternative findNode algorithm is a good approximation of the original algorithm for $s \geq 16$. However, this value discovered here is not necessarily universal, even though it worked for all tested scenes. It is possible that this value depends on the scene and the BVH that is currently processed. This, however, was not observed in any tested scene.

Figure 3.19 on the preceding page shows the total time of the algorithm for different array sizes. Again, the original algorithm is visualized as a line, since it has no fixed size array.

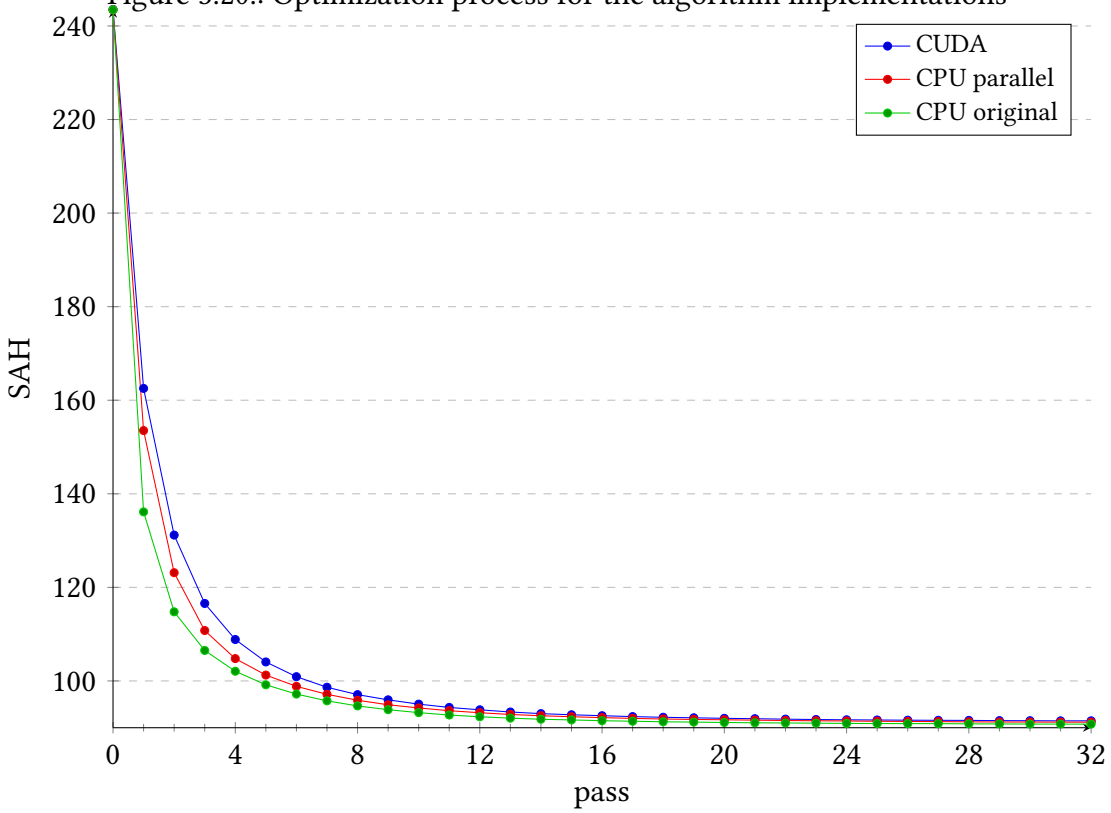
This figure shows that the runtime of the alternative findNode algorithm (altFind) is linked to the size of its "priority array". The total processing time rises linearly for $s \geq 12$. $s = 4$ and $s = 8$ do not follow this linear trend.

The relatively low value for $s = 4$ is linked to the bad "optimization" performance at this array size. Here, the algorithm terminates early because of the limited size of the "priority array". It thus does not find the correct reinsertion position and. The early termination of the algorithm is then responsible for the relatively low total time.

However, the spike at $s = 8$ cannot be explained this easily. Furthermore, this spike in processing time was not observed in all scenes. For instance, the time recorded for $s = 8$ in the *interior* scene follows the linear trend of the remaining values. Thus this time spike is most likely linked the initial BVH of specific scenes. Here, due to the insufficiently sized "priority array", the alternative findNode algorithm may take longer because the tree is modified in a way that negatively affects the runtime of the entire algorithm.

Despite that, it is noteworthy that the alternative findNode algorithm is consistently faster for $s \leq 18$ than the original. It is thus faster on the CPU (for $s \leq 18$), even though it was primarily developed to improve the GPU performance. Since there is virtually no downside in using this algorithm with $s = 16$ over the original algorithm, this version of the findNode can be used in all scenarios tested in this thesis. Furthermore, this algorithm offers far better performance on the GPU compared to the original (see section 3.3 on page 39).

Figure 3.20.: Optimization process for the algorithm implementations



3.5. Algorithm comparisions

In this section, the three optimization algorithms are compared. Here the best configurations, for each algorithm are used. These configurations were determined in the previous sections of this chapter. The LBVH algorithm was used to generate the initial BVH for all tests presented here. Furthermore, the batch size was set to 1% of all nodes. The batch was determined with the cost measure presented in section 2.2.1 on page 10. A partial sorting algorithm was then used to select the nodes with the highest cost measure. The batch was not further sorted or shuffled in either test presented in this section.

The batch was divided into 16 chunks for both parallel algorithms (*CPU parallel* and *CUDA*). Additionally, the offset layout was used to reduce the amount of discarded patches. In both cases, the reinsertion positions for the removed nodes were calculated with the alternative findNode algorithm (see section 2.5 on page 23). Here the "priority array" size was set to 16.

Figure 3.20 shows the average SAH reduction for all 31 tested scenes. Here the original algorithm (*CPU original*) performs significantly better than both parallel implementations. However, this difference in SAH becomes less significant in later passes, since all plots converge to the same value. After pass 8, the difference in SAH between the CUDA and the original algorithm is down to 2.41 and after pass 16 the difference is further reduced to 1.07. This difference is insignificant compared to the total SAH reduction.

Figure 3.21.: Average time comparison of the optimization algorithms

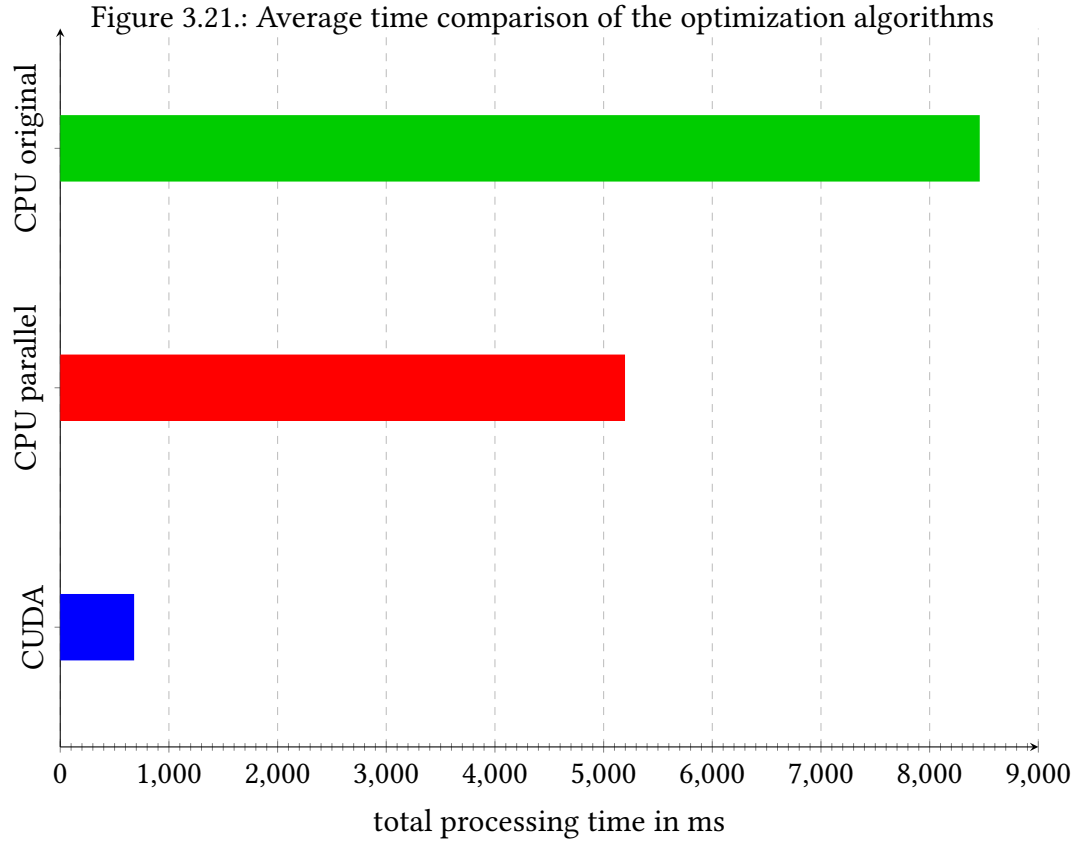
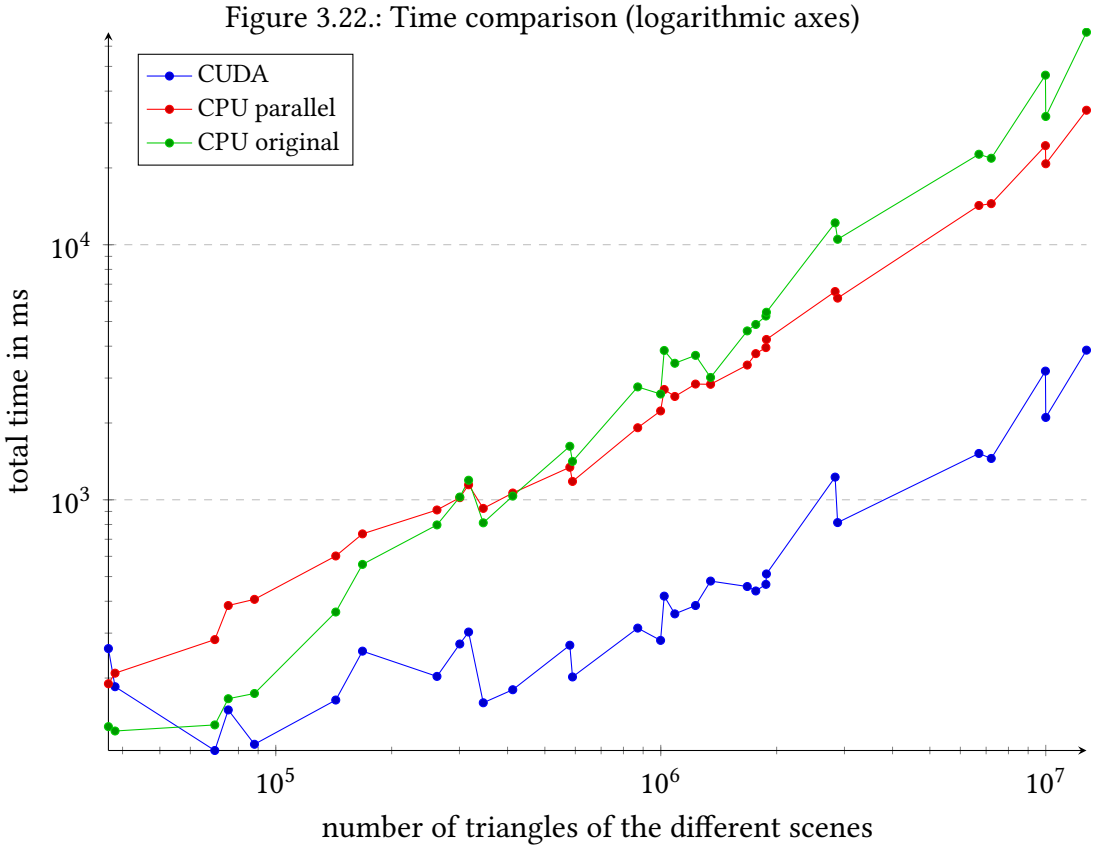


Figure 3.22.: Time comparison (logarithmic axes)



However, there are significant differences between the algorithms visible, when comparing the runtime performance. Figure 3.21 on the preceding page shows the average processing time for all 31 scenes. Here the total time for all 32 passes was recorded. This figure shows that the parallel CPU implementation reduces the average processing time by roughly $\frac{1}{3}$ while the CUDA implementation only requires 8% of the original processing time.

However, this speedup can vary depending on the scene. Figure 3.22 on the previous page visualizes the processing time for all scenes (represented by their triangle count). It should be noted that both axes are scaled logarithmically.

This figure shows the processing time generally increases with the number of triangles for the different scenes. There are, however, some noticeable differences in processing time for scenes with a similar amount of triangles. This is due to the fact that the processing time also depends on the layout of the scene in addition to the total amount of triangles. Despite this, a general correlation between the processing time and the number of triangles can still be observed.

This is not surprising since more work has to be done in scenes with a higher triangle count than in scenes with a lower triangle count. The figure also shows that the speedup of the parallel algorithms is not consistent across a wide range of scenes with different triangle counts.

In fact, the parallel CPU implementation performs worse compared to the original algorithm for the first third of the tested scenes. This implementation performs consistently worse than the single threaded version for scenes with a low triangle count. However, the performance of the parallel implementation increases, compared to the sequential version, for scenes with more triangles. For scenes with more than 1.000.000 triangles, the parallel version is consistently better than the sequential implementation.

This is due to the additional processing overhead of the parallel implementations. Here, additional steps are introduced to parallelize the algorithm. As a result of the parallelization only pays off for more complex scenes, where the additional overhead becomes less significant.

It should be noted that the tests were executed on a system with 4 cores and 8 threads. This currently limits the amount of parallel work on the current system. Thus, the results may improve on systems with higher core count CPUs, since more work can be parallelized here.

The results for the GPU implementation (*CUDA*) seem to confirm these assumptions. Here the amount of parallelization is significantly increased compared to a CPU. This implementation is already consistently faster for scenes with more than 80.000 triangles, as a result. Here the speedup also increases with the number of triangles. The GPU implementation is roughly 9 to 11 times faster than the 1.000.000 triangles and 16 to 18 times faster for scenes with more than 10.000.000 triangles.

Tests have shown that the memory speed of the GPU mostly bottlenecks the GPU implementation. As a result, more recent GPUs (like the upcoming RTX 20 series from NVidia) with faster memory are likely to increase the performance further.

Since these measurements were only performed on one system, the speedup of the parallel algorithms can vary depending on the hardware configuration. As mentioned above, a higher core count CPU is likely to increase the performance of the parallel CPU implementation. Furthermore, the speed of a CPU an a GPU based implementation was compared in this section. Swapping the GPU in this system can drastically change the relative performance of the tested implementations, as a result. Thus, the relative speedup is mostly dependant on the CPU and GPU combination.

3.6. Real time BVH optimization

In this section, the use of the GPU based optimization algorithm (see section 2.4 on page 22) for real-time BVH optimization is evaluated. More specifically, this algorithm is used to optimize the SAH for each frame. Each frame was also rendered with a CUDA ray-tracer based on Popov et al. [Pop+07] and Áfra and Szirmay-Kalos [ÁS14].

The real-time optimization algorithm was implemented with the approach presented in section 2.6 on page 25. Here one chunk was processed in each frame.

The dynamic scene of this test was generated as follows: The *interior* model (2837209 triangles) was used as a static background scene. Then the *dragon* model (871306 triangles) was moved along a fixed path through this scene. Here the position and orientation for the model were saved for each frame to ensure that the dragon would always follow the same path. The movement of the *dragon* was realized by first updating all vertex positions of the object. Then the bounding boxes of the BVH were updated to reflect these changes (the tree structure itself is not changed during this step).

The tested animation is 3121 frames long, and all frames were generated with a fixed camera position and a 1920×1080 resolution. Furthermore, the SAH value and the frame time were recorded for each frame. Figure 3.23 on page 53 and figure 3.24 on page 53 visualize these results with and without the BVH optimization enabled.

The GPU optimization algorithm used in this section was configured as follows: It uses the cost measure presented in section 2.2.1 on page 10 and a partial sorting algorithm to select the nodes for the batch. Like in all other tests, the batch size was set to 1% of all nodes in the BVH tree. Furthermore, the offset layout with 16 chunks was used. The reinsertion positions for the removed nodes were computed with the alternative findNode algorithm with an array size of 16.

The optimization algorithm is executed after the scene is updated (as described above) for the next frame of the animation.

The first figure shows that the SAH is consistently lower when the optimization algorithm is enabled. Furthermore, the used optimization approach can keep the SAH value roughly 101.2 after the initial 400 frames. Without optimization, on the other hand, the SAH rises and falls depending on the current position of the dragon (the SAH is generally higher when the object is farther away from the starting positio).

It should be noted that the maximum observed difference in SAH was only 1.47% with the BVH optimization enabled and disabled. Thus the animation only has a negligible impact in this specific scene.

Despite this, the GPU optimization algorithm is still able to keep the BVH at a consistent quality, espacially around frame $2,000$ where the difference in SAH is greatest. Thus, it should be possible for the optimization algorithm to also keep the SAH at a consistent level for more dynamic scenes. Further tests have to be done to confirm this.

In the second figure, the frame times for each frame are visualized. It is not immediately visible which of the versions performs better. Both plots fluctuate depending on the current scene and other factors. There are furthermore significant spikes in both plots that appear in regular intervals. The cause of these spikes is currently unknown.

The only observable difference between the to plots is the noticeably greater oscillation of the animation with optimization enabled. Furthermore, both plots oscillate around $60ms$ mark, with no apparent correlation to the SAH or the currently visible scene. Thus, the impact of the optimization algorithm in this scene is only minimal.

Furthermore, the average frame times for the test with and without optimization was $61.23ms$ and $60.99ms$. Thus, the version with optimization is $0.24ms$ slower than the version without optimization. However, the expected overhead of the GPU BVH optimization (see section 3.5 on page 48) is higher. Here, the average total processing time for 32 passes is $680ms$. Since each pass is further divided into 16 chunks, the expected additional overhead for each frame is $\frac{680ms}{32*16} = 1.32ms$. Thus, the time spent in other parts of the frame rendering process (most likely the ray-tracer) was reduced by $1.08ms$.

In addition to the tests with and without optimization, the third test with a full LBVH rebuild for each frame was run. Here the average frame time was $234.62ms$ and the average SAH for each frame was 302.3. These results were not included in the figures since they are considerably worse than the results from the other tests.

These test results show that the optimization approach for real-time ray tracing of dynamic scenes is a viable alternative to rebuilding the BVH from scratch for each frame. This algorithm is 3.7 times faster than rebuilding approach. Additionally, the quality of the BVH is considerably better. For this scene, the SAH value for rebuilding the BVH each frame was almost 3 times higher compared to the optimization approach.

Despite all these advantages, further tests with more scenes have to be made to determine the limitations of this approach.

Figure 3.23.: SAH after each frame

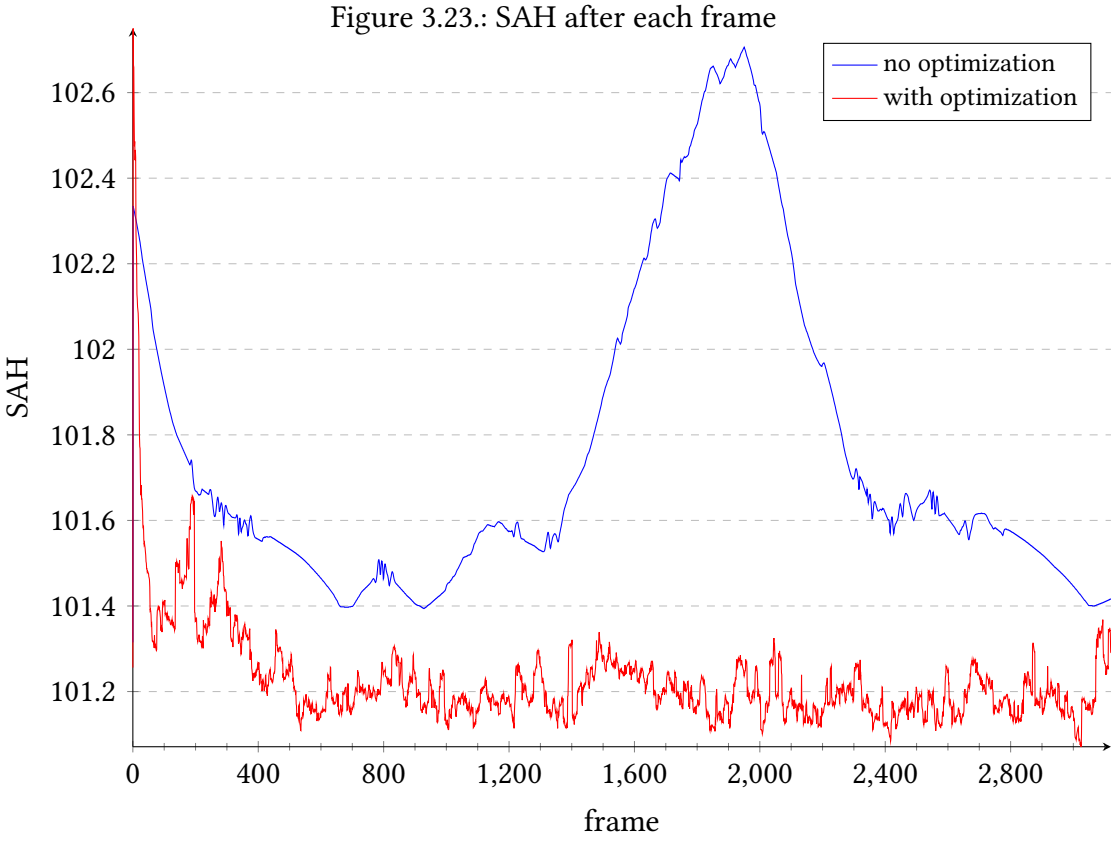
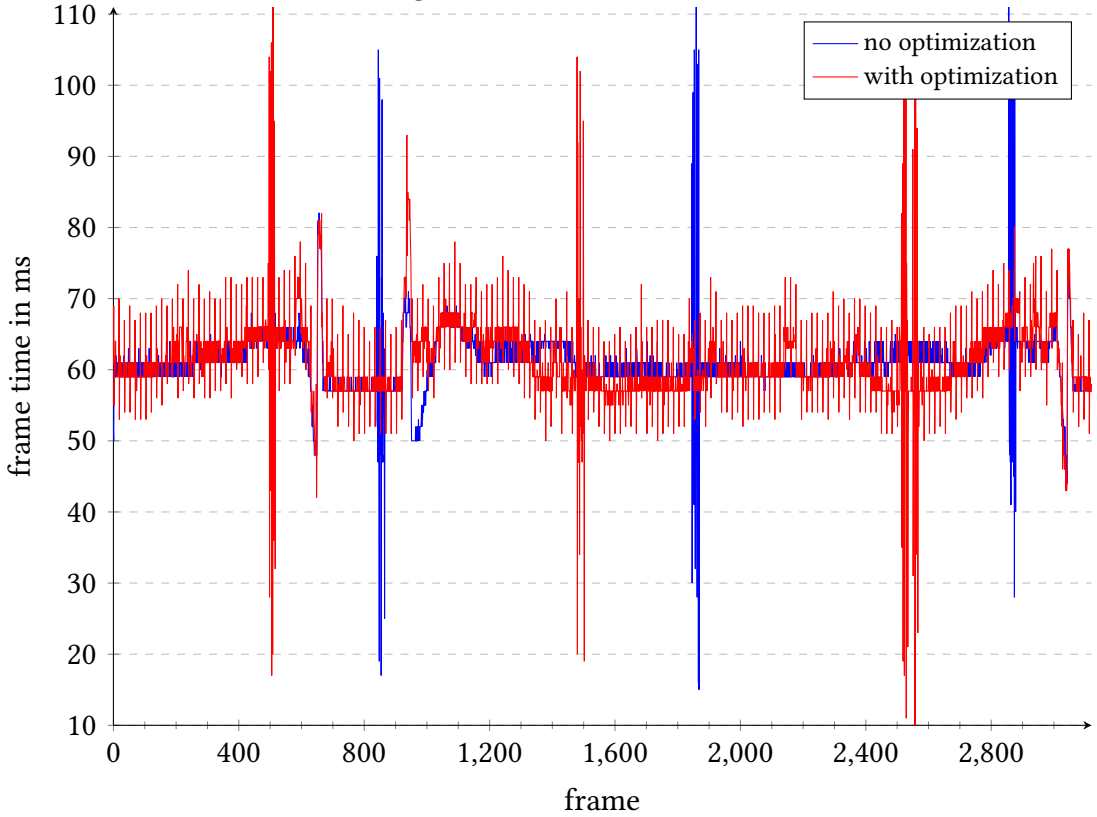


Figure 3.24.: Frame time



4. Conclusion

The main contribution of this thesis is the parallel parallelized version of the BVH optimization algorithm first introduced by Bittner et al. [BHH13]. However, multiple adjustments were necessary, to achieve the parallelization of this inherently sequential algorithm.

In the parallel algorithm, the batch of nodes is first divided into equally sized chunks, which are then each processed in parallel. Here each node in the chunk is removed and reinserted with the original algorithm to optimize the BVH. The fundamental idea with this approach is that the nodes can be processed in parallel, with the original algorithm, when each node has its own, virtual copy of the BVH tree. With this approach, the entire chunk of nodes can be processed independently and entirely in parallel.

Once this step is complete, the virtual BVH copies are checked for conflicts and merged together. Conflicting BVH copies are discarded in this version of the algorithm.

This algorithm can be implemented both on the CPU and the GPU. Here the GPU implementation has shown to be consistently faster (up to 17 times) than the original sequential implementation. The relative speedup generally increases with the size of the scene. The parallel CPU implementation, on the other hand, does not always offer a performance improvement over the original algorithm. With the hardware specified in chapter 3 on page 27, this implementation only offers better performance for scenes with more than 1.000.000 triangles.

Furthermore, both parallel implementations can optimize a given BVH to nearly the same degree as the original algorithm from Bittner et al. [BHH13]. As a result, both algorithms offer the same level of SAH reduction as the original with increased performance for large enough scenes.

A slightly modified version of GPU implementation can also be used in real-time scenarios. With the release of the RTX 20 series from NVidia, real-time ray-tracing becomes more and more relevant. This is because these video cards support ray-tracing at a hardware level, resulting in higher ray-tracing performance compared to older models.

In addition to the parallel algorithm, this thesis also introduces a modified version of the findNode algorithm. Here, the priority queue of the original algorithm is approximated with a fixed sized “priority array”. This approach offers a significant performance increase for the GPU implementation, due to better memory utilization. This modified algorithm can furthermore provide the same level of SAH reduction as the original with a reasonably sized “priority array”.

4.1. Future Work

Both parallel optimization algorithms provide many possibilities for future work.

First, the conflict detection algorithm in this thesis uses a rudimentary first come first served approach. As a result, it offers good runtime performance, at the cost of lower quality conflict detection. Here, more sophisticated algorithms may improve the SAH faster, by preferring patches which likely improve the SAH the most. Additionally, it may be possible to avoid discarding patches entirely by resolving the conflicts.

Furthermore, as shown in section 3.6 on page 51, the GPU implementation in particular shows promising results in the real-time ray-tracing sector. Here more research has to be done how precisely the algorithm can be integrated into real-time applications.

More specifically, it has to be determined how much work in each frame needs to be done to keep the tree quality at the desired level. It is also not clear whether or not it may become necessary to rebuild the BVH when a scene changes too much completely. The cache locality of the BVH may also degrade over time, which can also negatively impact the performance.

The chunk layouts used in both parallel implementations have also been shown to impact the performance, as well as the SAH reduction of both cases. It may thus be possible to further improve the SAH reduction “speed”, as well as the runtime performance with different layouts.

Bibliography

[AKL13] Timo Aila, Tero Karras, and Samuli Laine. "On Quality Metrics of Bounding Volume Hierarchies". In: *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: ACM, 2013, pp. 101–107. ISBN: 978-1-4503-2135-8. DOI: 10.1145/2492045.2492056. URL: http://doi.acm.org/10.1145/2492045.2492056.

[ÁS14] Attila T. Áfra and László Szirmay-Kalos. "Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing". In: *Computer Graphics Forum* 33.1 (2014), pp. 129–140. DOI: 10.1111/cgf.12259. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12259. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12259.

[BHH13] Jiří Bittner, Michal Hapala, and Vlastimil Havran. "Fast Insertion-Based Optimization of Bounding Volume Hierarchies". In: *Computer Graphics Forum* 32.1 (2013), pp. 85–100. DOI: 10.1111/cgf.12000. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12000. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12000.

[BHH15] Jiří Bittner, Michal Hapala, and Vlastimil Havran. "Incremental BVH Construction for Ray Tracing". In: *Computers & Graphics* 47.1 (2015), pp. 135–144.

[CHH02] Nathan A. Carr, Jesse D. Hall, and John C. Hart. "The Ray Engine". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '02. Saarbrucken, Germany: Eurographics Association, 2002, pp. 37–46. ISBN: 1-58113-580-7. URL: http://dl.acm.org/citation.cfm?id=569046.569052.

[DFM13] Michael J. Doyle, Colin Fowler, and Michael Manzke. "A Hardware Unit for Fast SAH-optimised BVH Construction". In: *ACM Trans. Graph.* 32.4 (July 2013), 139:1–139:10. ISSN: 0730-0301. DOI: 10.1145/2461912.2462025. URL: http://doi.acm.org/10.1145/2461912.2462025.

[Gan+15] P. Ganestam, R. Barringer, M. Doggett, and T. Akenine-Möller. "Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees". In: *Journal of Computer Graphics Techniques (JCGT)* 4.3 (Sept. 2015), pp. 23–42. ISSN: 2331-7418. URL: http://jcgt.org/published/0004/03/02/.

[GHB15] Yan Gu, Yong He, and Guy E. Blelloch. "Ray Specialized Contraction on Bounding Volume Hierarchies". In: *Computer Graphics Forum* 34.7 (2015), pp. 309–318. DOI: 10.1111/cgf.12769. eprint: https://onlinelibrary.wiley.com/

doi/pdf/10.1111/cgf.12769. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12769.

[GPM11] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. "Simpler and Faster HLBVH with Work Queues". In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. HPG '11. Vancouver, British Columbia, Canada: ACM, 2011, pp. 59–64. ISBN: 978-1-4503-0896-0. DOI: 10.1145/2018323.2018333. URL: http://doi.acm.org/10.1145/2018323.2018333.

[GS87] J. Goldsmith and J. Salmon. "Automatic Creation of Object Hierarchies for Ray Tracing". In: *IEEE Computer Graphics and Applications* 7.5 (May 1987), pp. 14–20. ISSN: 0272-1716. DOI: 10.1109/MCG.1987.276983.

[Hav00] Vlastimil Havran. "Heuristic Ray Shooting Algorithms". PhD thesis. the Faculty of Electrical Engineering, Czech Technical University, Prague, Nov. 2000.

[KA13] Tero Karras and Timo Aila. "Fast Parallel Construction of High-quality Bounding Volume Hierarchies". In: *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: ACM, 2013, pp. 89–99. ISBN: 978-1-4503-2135-8. DOI: 10.1145/2492045.2492055. URL: http://doi.acm.org/10.1145/2492045.2492055.

[Kar12] Tero Karras. "Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees". In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*. EGGH-HPG'12. Paris, France: Eurographics Association, 2012, pp. 33–37. ISBN: 978-3-905674-41-5. DOI: 10.2312/EGGH/HPG12/033-037. URL: https://doi.org/10.2312/EGGH/HPG12/033-037.

[Ken08] A. Kensler. "Tree rotations for improving bounding volume hierarchies". In: *2008 IEEE Symposium on Interactive Ray Tracing*. Aug. 2008, pp. 73–76. DOI: 10.1109/RT.2008.4634624.

[Kim+10] T. J. Kim, B. Moon, D. Kim, and S. E. Yoon. "RACBVHs: Random-Accessible Compressed Bounding Volume Hierarchies". In: *IEEE Transactions on Visualization and Computer Graphics* 16.2 (Mar. 2010), pp. 273–286. ISSN: 1077-2626. DOI: 10.1109/TVCG.2009.71.

[KK86] Timothy L. Kay and James T. Kajiya. "Ray Tracing Complex Scenes". In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), pp. 269–278. ISSN: 0097-8930. DOI: 10.1145/15886.15916. URL: http://doi.acm.org/10.1145/15886.15916.

[Lau+09] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. "Fast BVH Construction on GPUs". In: *Computer Graphics Forum* 28.2 (2009), pp. 375–384. DOI: 10.1111/j.1467-8659.2009.01377.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2009.01377.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2009.01377.x.

[MB90] J. David MacDonald and Kellogg S. Booth. "Heuristics for ray tracing using space subdivision". In: *The Visual Computer* 6.3 (May 1990), pp. 153–166. ISSN: 1432-2315. DOI: 10.1007/BF01911006. URL: https://doi.org/10.1007/BF01911006.

[MW06] J. Mahovsky and B. Wyvill. “Memory-Conserving Bounding Volume Hierarchies with Coherent Raytracing”. In: *Computer Graphics Forum* 25.2 (2006), pp. 173–182. DOI: 10.1111/j.1467-8659.2006.00933.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2006.00933.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2006.00933.x.

[PL10] J. Pantaleoni and D. Luebke. “HLBVH: Hierarchical LBVH Construction for Real-time Ray Tracing of Dynamic Geometry”. In: *Proceedings of the Conference on High Performance Graphics*. HPG ’10. Saarbrucken, Germany: Eurographics Association, 2010, pp. 87–95. URL: http://dl.acm.org/citation.cfm?id=1921479.1921493.

[Pop+07] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. “Stackless KD-Tree Traversal for High Performance GPU Ray Tracing”. In: *Computer Graphics Forum* 26.3 (2007), pp. 415–424. DOI: 10.1111/j.1467-8659.2007.01064.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2007.01064.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2007.01064.x.

[Pur+05] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. “Ray Tracing on Programmable Graphics Hardware”. In: *ACM SIGGRAPH 2005 Courses*. SIGGRAPH ’05. Los Angeles, California: ACM, 2005. DOI: 10.1145/1198555.1198798. URL: http://doi.acm.org/10.1145/1198555.1198798.

[RW80] Steven M. Rubin and Turner Whitted. “A 3-dimensional representation for fast rendering of complex scenes”. In: *SIGGRAPH*. Vol. 14. 1980, pp. 110–116.

[TSØ05] Niels Thrane, Lars Ole Simonsen, and Advisor Peter Ørbæk. *A comparison of acceleration structures for GPU assisted ray tracing*. Tech. rep. Aug. 2005.

[VHB16] Marek Vinkler, Vlastimil Havran, and Jiří Bittner. “Performance Comparison of Bounding Volume Hierarchies and Kd-Trees for GPU Ray Tracing”. In: *Computer Graphics Forum* 35.8 (2016), pp. 68–79. DOI: 10.1111/cgf.12776. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12776. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12776.

[Vii+17a] T. Viitanen, M. Koskela, P. Jääskeläinen, K. Immonen, and J. Takala. “Fast Hardware Construction and Refitting of Quantized Bounding Volume Hierarchies”. In: *Computer Graphics Forum* 36.4 (2017), pp. 167–178. DOI: 10.1111/cgf.13233. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13233. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13233.

[Vii+17b] Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, Heikki Kultala, and Jarmo Takala. “MergeTree: A Fast Hardware HLBVH Constructor for Animated Ray Tracing”. In: *ACM Trans. Graph.* 36.5 (Oct. 2017), 169:1–169:14. ISSN: 0730-0301. DOI: 10.1145/3132702. URL: http://doi.acm.org/10.1145/3132702.

[Vin+13] Marek Vinkler, Jiří Bittner, Vlastimil Havran, and Michal Hapala. "Massively Parallel Hierarchical Scene Processing with Applications in Rendering". In: *Computer Graphics Forum* 32.8 (2013), pp. 13–25. DOI: 10.1111/cgf.12140. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12140. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12140.

[Wal07] I. Wald. "On fast Construction of SAH-based Bounding Volume Hierarchies". In: *2007 IEEE Symposium on Interactive Ray Tracing*. Sept. 2007, pp. 33–40. DOI: 10.1109/RT.2007.4342588.

[Wal12] I. Wald. "Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture". In: *IEEE Transactions on Visualization and Computer Graphics* 18.1 (Jan. 2012), pp. 47–57. ISSN: 1077-2626. DOI: 10.1109/TVCG.2010.251.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. "Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies". In: *ACM Trans. Graph.* 26.1 (Jan. 2007). ISSN: 0730-0301. DOI: 10.1145/1189762.1206075. URL: http://doi.acm.org/10.1145/1189762.1206075.

[WG17] Dominik Wodniok and Michael Goesele. "Construction of bounding volume hierarchies with SAH cost approximation on temporary subtrees". In: *Computers & Graphics* 62 (2017), pp. 41–52. ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2016.12.003. URL: http://www.sciencedirect.com/science/article/pii/S0097849316301376.

[WHG84] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. "Improved Computational Methods for Ray Tracing". In: *ACM Trans. Graph.* 3.1 (Jan. 1984), pp. 52–69. ISSN: 0730-0301. DOI: 10.1145/357332.357335. URL: http://doi.acm.org/10.1145/357332.357335.

[WIP08] Ingo Wald, Thiago Ize, and Steven G. Parker. "Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes". In: *Computers & Graphics* 32.1 (2008), pp. 3–13. ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2007.11.004. URL: http://www.sciencedirect.com/science/article/pii/S0097849307002014.

[WSE04] Daniel Weiskopf, Tobias Schafhitzel, and Thomas Ertl. "GPU-Based Nonlinear Ray Tracing". In: *Computer Graphics Forum* 23.3 (2004), pp. 625–633. DOI: 10.1111/j.1467-8659.2004.00794.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2004.00794.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2004.00794.x.

[YM06] Sung-Eui Yoon and Dinesh Manocha. "Cache-Efficient Layouts of Bounding Volume Hierarchies". In: *Computer Graphics Forum* 25.3 (2006), pp. 507–516. DOI: 10.1111/j.1467-8659.2006.00970.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2006.00970.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2006.00970.x.

A. Appendix

A.1. Building the source code

The complete source code used for this thesis can be found on GitHub at `https://github.com/mensinda/BVHTest`. The build instructions presented here were tested on Ubuntu 18.04 and Arch Linux.

Requirements

A C++17 compatible compiler is required to build the source code. The additional requirements are listed in the table below.

meson	>= 0.45	`https://mesonbuild.com`
ninja	>= 1.7.2	`https://ninja-build.org`
CUDA	>= 9	`https://developer.nvidia.com/cuda-zone`
assimp	>= 3.2.0	`http://assimp.org`
glfw	>= 3.1.0	`https://www.glfw.org`
OpenGL	>= 3.3	`https://www.opengl.org`

Building the source code

The source code can be downloaded and build with the following commands:

```
git clone --recursive https://github.com/mensinda/BVHTest
cd BVHTest
meson build
ninja -C build
```

After all commands are finished, the final executable will be in `./build/src/bvhTest`.

A.2. Test scenes

The sources of the test scenes used in chapter 3 on page 27 are presented in the table below. Most models were downloaded from the *Stanford 3D Scanning Repository* and the *McGuire Computer Graphics Archive*.

Scene	URL
armadillo bunny dragon happy_vrip xyzrgb_dragon xyzrgb_statuette	`http://graphics.stanford.edu/data/3Dscanrep/`
AL05a AL05m AL05y breakfast_room exterior erato fireplace_room gallery hairball Indonesian_statue interior living_room powerplant roadBike rungholt salle_de_bain san-miguel serapis sibenik sponza sportsCar vokselia_spawn white_oak	`http://casual-effects.com/data/`
blade	`https://www.cc.gatech.edu/projects/large_models/`
bubaBig	`http://www.cs.utah.edu/~rvance/cs6620/final/`