Eurographics Symposium on Parallel Graphics and Visualization (2016)
W. Bethel, E. Gobbetti (Editors)

Parallel Spatial Splits in Bounding Volume Hierarchies

V. Fuetterling[1,2], C. Lojewski[1], F.-J. Pfreundt[1] and A. Ebert[2]

[1]Fraunhofer ITWM
[2]TU Kaiserslautern

Abstract

Bounding volume hierarchies (BVH) are essential for efficient ray tracing. In time-constrained situations such as real-time or large model visualization, fast construction of BVHs usually compromises hierarchy quality, resulting in reduced rendering speed. We propose a parallel framework for the state-of-the-art BVH construction algorithm with spatial splits (SBVH) that provides highest quality hierarchies within a time frame competitive with low-quality builders optimized for construction speed. We leverage both data and task parallelism to employ threading and single instruction, multiple data (SIMD) capabilities of modern CPUs. Our key contribution is a lightweight memory management and load balancing scheme that maximizes parallel efficiency.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

Ray tracing is a versatile method for rendering, collision detection and related problems. Efficiency ray tracing relies on acceleration structures that organize the scene data in a hierarchical fashion. This reduces the algorithmic complexity of a search query from $O(N)$ to $O(\log N)$ where N is the number of primitives in the scene.

In recent years the bounding volume hierarchy (BVH) with axis-aligned boxes as bounding volumes has been established as the state-of-the-art acceleration structure for ray tracing applications [EG08, WBS07, FLPE15]. Thus high performance BVH construction is the key to reduce the time-to-image for dynamic and massive scenes.

A variety of construction algorithms exist that can be categorized as divisive top-down and agglomerative bottom-up types. The linear BVH (LBVH) [LGS*09] is one of the fastest implementations regarding construction speed and part of the agglomerative family. Its efficiency originates from a linear time complexity and straightforward parallelizability where the primitives are sorted into an implicit octree-like structure defined by the Morton space-filling curve, followed by a simple merging procedure to construct the hierarchy of bounding boxes. However, due to the predetermined structure this method does not adapt to the scene geometry and produces BVHs of low quality, resulting in inflated ray tracing times. Various extensions aim at improving LBVH quality, such as approximate agglomerative clustering (AAC) [GHFB13], post-process optimization [KA13, BHH13] or hybrid strategies [PL10, GPM11]. However, the quality remains inferior to divisive construction based on the surface area heuristic (SAH) [GS87, Hav01, AKL13]. Given a set of primitives, the SAH associates a cost metric for every partitioning proposal and thus steers the subdivision process through cost minimization.

Augmenting the SAH-based algorithm with the option to split primitives, if cost effective, leads to the split BVH (SBVH) [SFD09, PGDS09] which produces the highest quality BVHs of all known methods. The drawback of the divisive algorithms is the increased time complexity $O(N \log N)$ over the agglomerative approaches based on the LBVH and the increased difficulty for scalable parallelization. In particular, efficient memory management for primitive splits becomes an issue in the presence of multiple threads. While parallelization schemes have been proposed for BVH construction without splits [Wal12, Wal07, BHH15], an optimal solution for the SBVH is still missing. Thus the contribution of our work is a highly scalable parallelization framework for the SBVH that is lightweight and easy to implement.

Furthermore we demonstrate how to apply single instruction, multiple data (SIMD) instructions in the form of advanced vector extensions (AVX) [Int16] to accelerate performance critical parts of the algorithm, such as primitive split-

© The Eurographics Association 2016.

DOI: 10.2312/pgv.20161179

ting. The combined result of our contributions is a SBVH implementation significantly faster than the best available alternative of its kind, competing with the inferior quality agglomerative algorithms for moderate primitive counts.

2. The SBVH algorithm

This section introduces the SBVH algorithm first proposed by [SFD09] and establishes the terminology used throughout the paper.

The structure of the SBVH algorithm is similar to other divisive BVH builders. Initially a single set of primitives exists (the parent set) that is partitioned into two smaller sets (the child sets). Partitioning is repeated recursively until the sets are small enough to form leaf sets. Along with every new child set a node is created which is referenced by its parent node and holds the axis-aligned bounding box enclosing all the primitives in the set. Once a set is turned into a leaf the corresponding node (now a leaf node) references the remaining primitives directly. As the SBVH is SAH-based, determining the partitioning with minimum cost for a given set is required before the actual subdivision can be performed. Since finding the exact partitioning with minimum cost is not feasible, an approximate is computed by choosing a small number of samples and selecting the one with the lowest cost. The sampling is implemented in the form of binning, where the parent bounding box is subdivided into $n+1$ equally-sized bins by n equidistant axis-aligned planes. The binning is performed for each axis separately.

The SBVH algorithm distinguishes between object binning and spatial binning. During object binning primitives are only considered as point-like elements defined by the center of their bounding box, while spatial binning takes the full size of a primitive into account. Thus spatial binning requires a primitive to be split if it overlaps one or multiple of the planes. While object binning produces partitionings that have disjunct child sets which may have overlapping bounding boxes, spatial binning leads to partitionings with disjunct bounding boxes but possibly overlapping child sets. Which binning strategy will result in the lowest cost partitioning is dependent on the primitive constellation and cannot be foreseen. Thus the approach taken by the SBVH is to find the best object partitioning and if the corresponding child bounding boxes overlap by a certain amount try to lower the cost further by testing the spatial binning. This is a sensible compromise because splitting is an expensive operation and increases memory consumption, while it is unlikely to improve SAH cost if object binning yields spatially disjunct sets. The total amount of splits during hierarchy construction is bounded by the *split budget* parameter. Once the split budget is consumed spatial binning is disabled. The SBVH algorithm is summarized in the following pseudo code:

```
1:  stack[]
2:  task ← root
3:  loop
4:      loop
5:          leafCost ← CalculateLeafCost(task)
6:          objCost ← BestObj(task)
7:          if task.childBoxes overlap then
8:              spatialCost ← BestSpatial(task)
9:          end if
10:         if leafCost is best then
11:             createLeaf(task)
12:             break
13:         else if objCost is best then
14:             (left, right) ← PartitionObj(task)
15:         else
16:             (left, right) ← PartitionSpatial(task)
17:         end if
18:         createNodes(task)
19:         stack.push(right)
20:         task ← left
21:     end loop
22:     task ← stack.pop()
23:     if task is empty then
24:         break
25:     end if
26: end loop
```

A task contains all the information required to partition the corresponding set of primitives. After partitioning execution continues with one of the two resulting tasks, denoted left and right, while the other is pushed to the task stack. Once the task stack is popped in an empty state hierarchy construction is finished.

2.1. Primitive fragments

Instead of working directly with the primitives (triangles in our case), proxy elements called *fragments* are used. A fragment stores the axis-aligned bounding box and a reference to the primitive it represents. Thus fragment data is sufficient for the binning process and access to the full primitive structure is only required in the event of splitting. Also splitting does not result in duplication of the primitives, just copies of the corresponding fragments with refitted bounding boxes.

2.2. Binning

As mentioned previously, the parent bounding box is sliced into $n+1$ equally sized bins $b_i, i \in [0,n]$ separated by n equidistant axis-aligned planes $p_i, i \in [0, n-1]$. Each bin keeps track of the number and spatial extent of the fragments it is assigned. The *bin index* i corresponding to a particular coordinate c is computed as $i = (c - parent_{min}) / planeDistance$. In the case of object binning a fragment's bin index is derived from its bounding box centroid. Spatial binning requires two indices, i_{min} and i_{max}, calculated from the minimum and maximum of the fragment's bounding box respectively. If the indices differ

© The Eurographics Association 2016.

the fragment overlaps all bins $b_i, i \in [i_{min}, i_{max}]$ and requires splitting at every plane $p_i, i \in [i_{min}, i_{max}-1]$, resulting in $i_{max} - i_{min}$ new fragments. The bounding boxes of the fragments are updated to tightly fit the primitive they represent within their respective bin. After the binning procedure the SAH cost is evaluated for every pair of child partitions left and right to the planes p_i.

2.3. Partitioning

Performing the partitioning resulting from object binning is straightforward. For each fragment the bin index is computed again and compared to the best plane index i_{best}. If the bin index is smaller the fragment is moved to the left, otherwise to the right set. Since the left and right counts are known from the binning, memory offsets can be computed to store the fragments of both sets in a continuous array. In the case of spatial binning the procedure is slightly different. Minimum and maximum indices are computed again and compared to the best plane index i_{best}. If i_{min}/i_{max} is smaller/larger than i_{best} the fragment is moved to the left/right set. Otherwise the fragment intersects the split plane and requires insertion into both sets.

3. Parallelization

Divisive algorithms such as SBVH offer intrinsic task parallelism by producing two completely independent tasks from every subdivision. The tasks can be distributed among several threads, and by implementing a proper load balancing scheme the parallel efficiency is maximized. However, at the start of hierarchy construction only a single task exists (the root task) and with every subsequent level of subdivision the number of independent tasks doubles. Consequently, a maximum of 2^n tasks are available at level n. If the number of participating threads is high, this initial bottleneck can harm scalability considerably, especially in the case of BVH construction where the amount of work is approximately constant at every level. Removing this bottleneck requires the implementation of shared task parallelism, so that multiple threads can collaborate on the same task [Wal12, WWB*14]. However, it is desirable to minimize the number of shared tasks which are not able to achieve the same parallel efficiency as multiple exclusive tasks due to fine-grained synchronization points.

We propose a novel scheduling strategy based on dynamic thread pools, that employs shared tasks only initially and permanently switches to exclusive task execution as soon as possible. For the dynamic load-balancing of exclusive tasks we introduce a lightweight lock-free mechanism which allows on-demand sharing of tasks while maintaining the task topology. Both contributions are generally applicable to divide and conquer algorithms given a task cost estimation.

In addition to load-balancing, a second critical component for multi-threaded BVH construction is efficient memory management in order to avoid synchronization and excessive over-provisioning. This is especially true for spatial splits where the number of fragments increases recursively.

In the following we introduce a novel approach based on dynamic pre-allocation with reinjection to implement recursively growing fragment buffers. Our solution requires no synchronization, retains a small memory footprint and as a positive side effect keeps the split budget balanced over the entire hierarchy.

3.1. Memory management

A SBVH implementation requires two types of dynamic memory buffers. The temporary buffers containing the fragments need to support creation and shuffling of elements, whereas for the output buffers holding the BVH nodes and the primitive lists referenced by leaves it is sufficient to support only creation with the constrain that elements are packed as tightly as possible in memory.

Space allocation for the output elements is implemented by simple atomic counters that are shared among all threads. This is similar to previous approaches for BVH construction without spatial splits. In order to reduce the frequency of atomic operations threads always allocate entire chunks of elements and manage such a chunk with local counters. This mechanism is fast and lock-free, resulting in tightly packed elements where a small amount of fragmentation can only occur in the final chunk of every thread. The size of the output buffers can be conservatively estimated by considering the number of input primitives and the size of the split budget.

The presence of spatial splits complicates the management of the temporary fragment buffers considerably in a multi-threaded environment. The reason is that the fragments need to be partitioned recursively and due to the primitive splitting the combined size of the two child sets may be larger than the parent set. Thus our memory management needs to be significantly more flexible compared to previous approaches.

The key idea is to bind space in the fragment buffer to a task, and recursively distribute this space among the corresponding child tasks as visualized in Figure 1. Initially the entire fragment buffer is allocated to the root task, where the input fragments reside in the lower part of the buffer and the upper part provides free space for primitive splits. During the partitioning phase the left and right child sets are created adjacent to the lower and upper boundary respectively, growing towards the center with the free space in between. The remaining free space is distributed to the left and right tasks in proportion to the size of the respective child sets. Thus a task always includes the necessary resources and a thread acquiring one of the tasks can directly access these resources without any additional synchronization. As proposed previously [Wal12], the implementation of the frag-

© The Eurographics Association 2016.

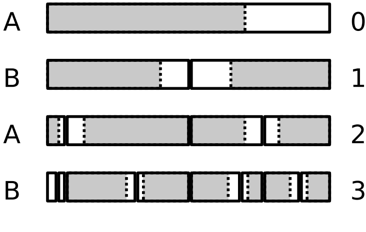

Figure 1: *Visualization of the fragment buffer management. A and B on the left mark the two individual buffers forming the fragment double buffer. The numbers on the right denote the hierarchy level. At the root level buffer A is partially filled with the initial set of fragments (shaded region), while the remaining free space can be consumed by splits. After the first subdivision the child sets are aligned to the left and right borders of buffer B and the free space in the middle is divided proportionally to the set size. This process is repeated recursively on the child sets while alternating buffers A and B.*

ment buffer features a double buffering technique, where the parent set resides in one buffer and the child sets are written to the other buffer. After a subdivision source and destination pointers are simply swapped. This way read/write dependencies that would exist in an in-place approach are eliminated, allowing all fragments to be partitioned in parallel.

As a side effect of our unique memory management the split budget is distributed evenly among the scene geometry, avoiding the situation where excessive splitting during the early part of the build process can drain the split budget for the later part. However, if a scene demands highly non-uniform split densities, the balanced distribution can be harmful. In this case the split budgets in low density regions go unused while in high density regions insufficient split budgets prevent optimal subdivisions.

To remedy this situation, we propose a mechanism to reinject unneeded split budgets back into the build process. Upon completion of a leaf task the remaining number of splits are added to the *reserve counter*. The reserve counter is a global state that is managed with atomic operations to allow sharing of the reserve splits among all the threads. However, to reduce frequency of the expensive atomics, each thread caches its reserve budget with a local counter and updates the global state only occasionally. If spatial binning produces a partitioning that exceeds the split budget provided by the corresponding task, a thread acquires the difference from the reserve counters, where the local counter has priority over the global counter. If the reserve budget is insufficient, the algorithm falls back to the best object partitioning. Once the split budget has been secured, the fragment buffer region bound to the current task is not large enough to hold the fragments for both child partitions, so that a new partition needs to be allocated for the smaller of the two child partitions. At the initial allocation of the fragment double buffer, a part of the space is set aside for this purpose, referred to as the *reserve buffer*. Allocations from the reserve buffer are performed with an atomic counter, and once all the reserve space has been used up the remaining tasks can no longer use the reserve mechanism. It would be possible to allocate additional space from system memory as the new buffer does not have to be continuous with respect to the initial buffer, though this would be rarely necessary. Since the per-fragment memory consumption related to the double buffer is marginal (less than 1%, see below), the reserve buffer can be large (e.g. twice the split budget).

Split budget balancing with reinjection combines the advantages of the purely balanced and first-come-first-served principle. Each part of a scene is guaranteed a relative amount of splits, while the unneeded budget can be shifted to high split density regions.

Finally, in order to reduce memory bandwidth demands and overall memory consumption, we replace the fragment double buffer with a fragment reference double buffer and keep the actual fragment data in a separate memory region managed by atomic counters in the same way as the output buffers. This is distinct from previous publications [SFD09, Wal12]. Since the fragment data structure is 32 bytes in size while a reference occupies only 4 bytes, a total of $2*32-(2*4+32)=24$ bytes is saved per fragment. The bandwidth balance is also positive since each task reads its fragments $2-3\times$ and writes once. With references this amounts up to $3*(4+32)+4=112$ bytes per fragment while using the fragments directly would result in $3*32+32=128$ bytes. In addition, significantly reducing the size of writes from 32 to 4 bytes has the advantage of reducing DRAM access because while reads are potentially serviced from the cache, writes need to be flushed to DRAM eventually. The drawback of this approach is increased access latency due to the reference indirection and inhibition of fragment hardware prefetching. However, we have measured experimentally that for working sets fitting into the L3 cache performance is equal for both buffering schemes, while for working sets larger than L3 a total run time reduction of up to 35% with references has been observed.

Interestingly, a very similar technique for recursively growing memory during spatial split partitioning has been developed in parallel by Ganestam et al. [GD16]. In contrast to our proposition they do not support reinjection and the layout of the memory buffer does not keep the free space centered, resulting in unnecessary memory movement. As future work they suggest to improve parallelization of the initial phase of partitioning which is addressed in the next section.

© The Eurographics Association 2016.

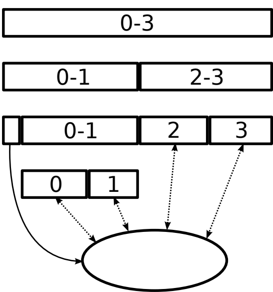

Figure 2: *Visualization of the thread management. The numbers represent thread identifiers. The root task is processed by all threads, and after the subdivision the thread pool is split into two in proportion to the left and right set size. The next subdivision performed by threads 0 and 1 yields one very small and one very large set on the left and on the right respectively, so the left task is inserted into the global task queue (implemented as a ring buffer) and both threads continue with the right task. Once a thread owns a task exclusively it switches to single-threaded execution. Dynamic load balancing is performed by exchanging tasks on the global task queue as indicated by the dashed arrows.*

3.2. Shared tasks

As mentioned previously, the goal of shared tasks is to allow fully multi-threaded execution from the start of hierarchy construction. At the same time it is desirable to minimize the number of threads processing the same task simultaneously and quickly reach the threshold where every thread can work on a single task exclusively. While the concept of shared tasks is not new itself [Wal07], we propose a novel scheduling mechanism optimizing the above constrain, permanently switching to exclusive tasks as soon as a proper load balancing is established with the help of dynamic thread pools. Dynamic thread pools prevent the inherent risk of a permanent switch once the number of independent task is equal to the number of threads, that the complexity of the individual tasks may vary widely, to the point where one thread has finished the entire child hierarchy belonging to its task, while another thread is still working on the first subdivision, thus stalling the fast thread due to the lack of more tasks.

The idea of dynamic thread pools is illustrated in Figure 2. For the root task all threads belong to a single pool. After the first subdivision the thread pool of size T is split into two, with the number of threads in each pool proportional to the number of fragments in the respective child tasks, according to the following equation:

$$T_l = \left\lfloor \frac{N_l}{N_l + N_r} T + 0.5 \right\rfloor, \; T_r = T - T_l,$$

where N_l and N_r are the number of fragments of the left and right child task respectively, and T_l and T_r the number of threads of the corresponding thread pools. Both pools can now operate independently. This procedure is repeated recursively, and once a thread finds itself to be the only one in the pool it permanently switches to exclusive task execution. If the subdivision of a shared task yields one child task with too few fragments to be assigned even a single thread the task is inserted into the global queue for exclusive tasks and the entire thread pool continues with the larger child task. As a result all threads will have tasks with roughly the same number of fragments upon switching from shared to exclusive task execution, and the demand for dynamic load balancing of exclusive tasks will be kept to a minimum.

3.3. Exclusive tasks

Exclusive tasks are processed by a single thread only, thus avoiding any kind of synchronization thanks to our memory management. However, dynamic load balancing requires that tasks produced by one thread can be consumed by another. In addition, the task topology should be maintained across thread boundaries so that post-order procedures can be applied to the BVH hierarchy, such as leaf pruning.

In contrast to previous approaches our algorithm does not classify tasks by the number of primitives to push them either to a strictly local stack or to a shared task pool. Instead tasks are always placed on the local stack and exchange of tasks is achieved with a global task queue storing *task pointers*, which is implemented as a lock-free atomic ring buffer. The *target size* defines the number of tasks that should be available from the task queue at any time, for which we have determined the base 2 logarithm of the thread count to be a good value. After subdivision of a task into two child tasks the thread continues with the child task containing the larger number of fragments and pushes the remaining child task onto the local stack. The thread checks the number of tasks in the global queue against the target size and inserts a task if necessary. Since the check is not atomic, it may happen that the number of tasks in the queue increases above the target size occasionally. Task insertion is always performed with a pointer to the bottom-most task on the local stack. Upon insertion the task is marked as *non-local*. As soon as post order traversal of the local stack pops a non-local task the traversal is terminated and a new task pointer is fetched from the global queue. A place holder containing the task pointer is pushed to the local stack. Once post-order traversal returns to the place holder, the corresponding pointer is used to write a completion notification to the original task. If a fetch operation is not successful because the queue is empty, the operation will block until a new task pointer is inserted by another thread. Once all threads have entered the blocking state hierarchy construction is almost finished and the threads are released with a null pointer. In the final step the remaining non-local and place holder tasks on the local stacks are processed until the post-order traversal reaches the root node.

© The Eurographics Association 2016.

The advantage of our approach compared to others [WWB*14] is that on the one hand task sharing happens only on demand increasing data locality, and on the other hand adapts the task size dynamically for optimal load balancing, with large task at the beginning and small tasks at the end of hierarchy construction.

4. SIMD

Single instruction multiple data (SIMD) allows an instruction to operate on multiple data elements gathered in a single vector register in parallel. On contemporary mainstream CPUs SIMD is available in the form of *advanced vector extensions* (AVX) with 256 bit registers accommodating 8 single precision floating point values (floats). Since arbitrary gather/scatter operations are not available or very slow, the layout of data structures should map naturally to the vector registers. Otherwise multiple loads and shuffle operations will diminish the performance potential of *SIMDfication*. Ideally the data has a *structure of arrays* (SoA) format so that operating on vectors is the same as operating on scalars, only multiple at a time. For various reasons this approach is seldom feasible in practice, so *arrays of structures* (AoS) are often used. Sometimes a combination of the two can be a good layout as well (e.g. AoSoA). For fragments AoS is a good fit as the bounding box requires 6 floats and one primitive index. By adding one padding element the fragment fits an AVX register exactly:

$$(x_{min} \;\; y_{min} \;\; z_{min} \;\; idx \;\; x_{max} \;\; y_{max} \;\; z_{max} \;\; pad)$$

This is similar to the layout proposed by [GBDAM15]. One of the most common operations during binning is the union of two bounding boxes. With the previous data structure this would require unpacking and a minimum/maximum instruction on the lower/upper part. In order to calculate the union of two fragments with a single instruction, we propose to use the convention to store the negatives of the minimum values:

$$(-x_{min} \;\; -y_{min} \;\; -z_{min} \;\; idx \;\; x_{max} \;\; y_{max} \;\; z_{max} \;\; pad)$$

Thus a single maximum instruction is sufficient, operating directly on the data structure.

Our SBVH implementation utilizes AVX instructions for all compute intensive parts of the algorithm, specifically for object/spatial binning/partitioning, primitive splitting and SAH calculation. In the following we discuss the high level SIMD design of the binning/partitioning kernels and separately primitive splitting, which has not been discussed in literature before. The implementation details are revealed by the source code provided as supplemental material.

4.1. Binning and partitioning

Both binning and partitioning require the calculation of bin indices as described in sections 2.2 and 2.3. Depending on the bin index of a fragment, the binning kernel updates the count and bounding box of the appropriate bin while the partitioning kernel appends the fragment index either to the left or right child partition. Thus operating on multiple fragments in parallel demands partly serialized scattered memory accesses. Since no hardware support is available for this kind of scattering mechanism, it is not obvious how to implement it efficiently in software. In fact, our first attempts barely improved performance upon the scalar code at all. Also previous work has struggled with this problem [Wal12], opting to utilize SIMD instructions inefficiently to parallelize over bins instead of fragments. Through experimentation we have established an efficient design pattern that works well for both binning and partitioning multiple fragments in parallel.

The basic idea is to divide the body of the loop over all fragments into a vectorized part for the bin index and a scalar part for the bin update. By interleaving the vectorized part for iteration $i+1$ and the scalar part for iteration i, both parts can be executed in parallel as they utilize different hardware resources of the CPU. Moving the first iteration of the vectorized part and the last iteration of the scalar part out of the loop yields an elegant implementation illustrated in the following snippet:

1: vector part *start*
2: **for** $i = start$ **to** *end* **do**
3:     vector part $i+1$
4:     scalar part i
5: **end for**
6: scalar part *end*

For both object and spatial binning two fragments are processed along all 3 axes simultaneously, utilizing 6 out of the 8 vector elements. In this case this is faster than working with 8 fragments, because the higher utilization would not compensate for the additional shuffle overhead. For the object partitioning only a single axis is of interest, so here the best approach is to process 8 fragments in parallel. Spatial partitioning only operates on one fragment at a time because the more complex control flow diminishes the advantage of multiple elements.

4.2. Primitive splitting

The primitives considered here are triangles, so primitive splitting requires a triangle-plane intersection test. Given an axis-aligned plane, the triangle-plane intersection is computed by choosing the two edges of the triangle overlapping the plane and calculating the corresponding line-plane intersection points. Processing the edges can be performed in parallel utilizing two vector elements of an AVX register. In order to profit from the remaining elements, multiple triangle-plane intersections are necessary.

The first option is to intersect one triangle with one plane in each dimension, filling only 6 of the 8 vector elements. Further elements are wasted because a triangle is not very likely to overlap binning planes in all three dimensions si-

© The Eurographics Association 2016.

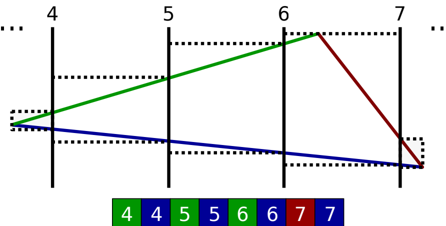

Figure 3: *Visualization of the triangle splitting. Numbers denote planes, colors denote triangle edges. The squares represent elements of a vector register and are colored and numerated according to the edge-plane combination they process. The dashed lines indicate the tight bounding boxes of the triangle within the respective bin.*

multaneously. The second option is to test 4 different triangles with 4 different planes. While this approach guarantees high utilization, the overhead of gathering the data from many scattered locations would be quite significant.

We propose a middle ground by performing intersection of one triangle with 4 consecutive planes along the same axis, as illustrated in Figure 3. This has the advantage of keeping data access coherent and allowing all vector elements to be utilized. Obviously, if the number of planes a triangle overlaps is not a multiple of 4 the AVX register is not fully occupied.

5. Results

The evaluation of our SBVH implementation focuses on three aspects: Overall performance, parallel efficiency and the SIMD advantage. We measure the overall performance by constructing BVHs for several test scenes and compare the timings to the parallel SBVH implementation of Embree [WWB*14] (version 2.7.1), a high-performance ray tracing library developed by Intel. Both implementations are configured to perform binning along all axes with 32 object bins and 16 spatial bins with the split budget set to 100% of the number of input triangles. Included in the timings are all computations required to obtain a ray tracing ready BVH, in particular the root bounding box calculation and triangle processing for accelerated ray-triangle intersection. Also both implementations output a 4-ary BVH because this is the preferred branching factor for ray tracing [FLPE15, Gut14]. This does not alter the SBVH algorithm except for the node layout. We determine the parallel efficiency of our algorithm by analyzing build times for varying thread counts and for varying scene sizes. Finally, we measure the performance advantage achieved through SIMDfication of our binning/partitioning kernels and triangle-plane intersection implementations. For all experiments the hardware platform is a dual socket Intel Xeon E5-2680v3 Haswell (24 cores / 48 threads total at 2.5GHz).

5.1. Overall performance

We test the build performance for 6 scenes commonly used in ray tracing benchmarks. The results are presented in Table 1. Our SBVH implementation demonstrates a significant speed-up over Embree for all scenes, ranging from $66k$ to $300M$ triangles in size. Especially for the smaller scenes below $1M$ triangles our algorithm is between $5-7\times$ faster. As we will show in the next section this is influenced to a large extent by the scalability of the two implementations.

Also for extremely large scenes such as the BOEING performance is high with respect to Embree. We attribute this observation in one part to the reduced parallel efficiency of Embree measured for large scenes (Figure 5) and in one part to our bandwidth conserving reference scheme, as we have observed the highest relative speed-up of about 35% for the BOEING compared to double buffering the fragments directly.

Compared to the performance achieved by LBVH based builders on a Nvidia Titan GPU for a moderately sized scene such as FAIRY, our high-quality SBVH implementation lies within the reported range of $2-9ms$ [KA13].

5.2. Parallel efficiency

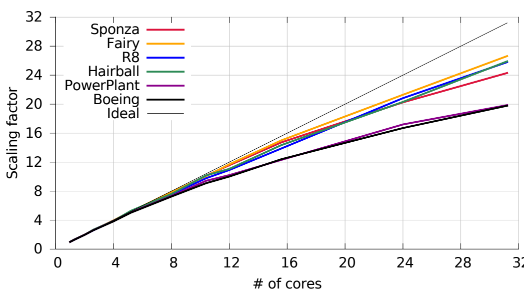

Figure 4: *Scaling factor as a function of core count with respect to the performance of a single thread. Results for all the test scenes from Figure 1 are provided. If hyperthreading is enabled (two threads per core), the core count is multiplied by* 1.3.

We analyze the parallel efficiency of our SBVH implementation in two ways, once by keeping the primitive count fixed and scaling the number of threads, and once by scaling the primitive count with all of the 48 threads active. Since our test platform has only 24 cores but 48 threads we multiply the core count by 1.3 if hyperthreading (HT) is enabled. This multiplier has been determined experimentally by comparing performance for one thread and for two threads pinned to a single core.

Figure 4 depicts the scaling factor as a function of thread

© The Eurographics Association 2016.

Table 1: *Overall performance with 48 threads for several scenes, comparing our SBVH implementation and Embree. The* splits *row indicates the increase in triangle count due to splitting for our implementation.*

	SPONZA	FAIRY	R8	HAIRBALL	POWERPLANT	BOEING
# triangles	66k	174k	795k	2.9M	12.8M	300M
Splits	30%	17%	10%	89%	16%	10%
Our	**3.6 ms**	**7.7 ms**	**24.7 ms**	**362 ms**	**608 ms**	**13.5 s**
Embree	26.0 ms	41.9 ms	162.7 ms	1266 ms	2785 ms	131.2 s
Speed-up	**7.2**×	**5.4**×	**6.6**×	**3.5**×	**4.6**×	**9.7**×

count for all the test scenes together with the ideal curve. Up to about 10 threads (or 8 cores + HT) all scenes exhibit ideal scaling.After this point parallel efficiency diverges from the ideal curve and the graphs separate into two bundles. The smaller scenes including HAIRBALL scale up to 26×, while POWERPLANT and BOEING achieve around 20× for all threads active. This behavior indicates that our SBVH implementation is memory bandwidth limited since all the small scenes fit (almost) entirely into the large L3 cache.

The situation becomes clearer by analyzing Figure 5. Here we scale the number of triangles and keep the thread count at 48. At 10k triangles the problem size is too small for our algorithm to scale above 15× (total run-time is about 0.7ms). The plateau of highest parallel efficiency (around 26×) is reached with slightly less than 100k triangles and extends until about 2M. After that scalability rapidly decreases towards a steady state of 20×. This cliff is where the L3 cache looses its effectiveness and fits perfectly with the data from Figure 4.

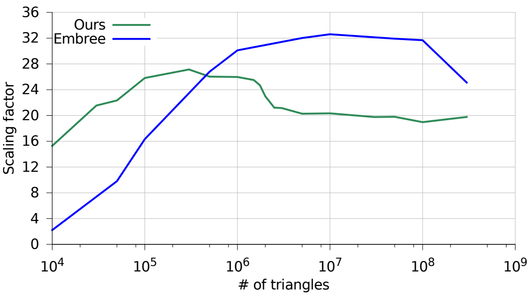

Figure 5: *Scaling factor (1 vs. 48 threads) as a function of triangle count based on the* BOEING *scene. Results for both Embree and our implementation are provided.*

The scalability of Embree exhibits a different behavior. For small triangle counts parallel efficiency is significantly worse compared to our implementation, but improves for larger triangle counts until catching up at about 750k triangles. From there however scaling continues up to the ideal of 32×. Contrary to our algorithm there is no cliff once the scene size exceeds the L3 cache. This indicates that Embree is not limited by memory bandwidth constrains, but rather by computation and/or memory and thread management.

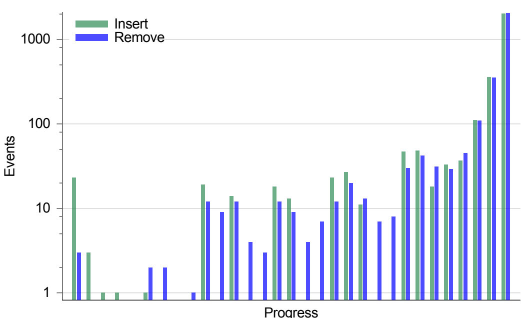

Figure 6: *Exchange events on the global task queue (logarithmic scale) as a function of progress bins for the construction of the* POWERPLANT *scene with 48 threads. Progress is measured as the number of finished nodes at the occurrence of a particular event. The last bin accounts for 72% of all events.*

In order to illustrate the load balancing characteristic of our parallelization framework (Sections 3.2 and 3.3) Figure 6 shows the exchange events on the global task queue for the construction of the POWERPLANT scene. For the largest part insert and remove events are very sparse, with only about 10% (300 total / 6 per thread) exchanges until 90% of the BVH is completed. For the last 10% of BVH construction the event rate increases exponentially due to the continued decrease in average number of fragments per task. Hence the load balancing works as expected: The dynamic thread pool mechanism leaves each thread with a similar initial task size upon switching from shared to exclusive task execution, reducing the demand for task exchange. Only when the tasks

© The Eurographics Association 2016.

become small at the end of BVH construction fine grained load balancing takes over to keep all threads busy.

5.3. SIMD advantage

Table 2: *Speed-up due to our AVX implementation (Section 4) with respect to scalar code. The results for the binning/partitioning kernels and the triangle-plane intersection test are reported separately. For the intersection test the average utilization of the vector registers is indicated (4 would be 100%).*

	Kernels	Intersection	
	Speed-up	Speed-up	Utilization
SPONZA	1.3×	2.6×	2.3
FAIRY	1.4×	1.8×	1.6
R8	1.6×	1.2×	1.2
HAIRBALL	1.2×	3.6×	2.8
POWERPLANT	1.4×	1.4×	2.7
BOEING	1.5×	1.1×	1.6

For the evaluation of our AVX implementations described in Section 4 we divide the results in triangle intersection test and binning/partitioning kernels for both spatial and object variants. The speed-ups reported in Table 2 are relative to a scalar implementation for either the intersection test or the kernels respectively and include the full build process. For the kernels the AVX version improves between 20% to 60% upon the scalar variant. The spread depends on the ratio of object to spatial binning, since for spatial binning most of the time is usually spent in triangle intersection and not in the binning itself. For triangle intersection the results vary considerably from scene to scene, from a significant 3.6× for HAIRBALL to a mediocre 1.1× for BOEING. This is in line with our expectations since the HAIRBALL geometry is predestined for excessive splitting while the BOEING and also the R8 have high object/spatial ratios.

Noting that an AVX register has 8 elements that theoretically allow an 8× speed-up, the question is if a more efficient vectorization compared to ours is possible. Since the binning kernels are very compact, this would most likely require hardware assistance for the gather/scatter operations. However the triangle intersection leaves some room for further improvement since we have not explored all the strategies to efficiently saturate the vector registers.

6. Conclusion

We have introduced an efficient parallelization framework for the SBVH algorithm. This includes thread and memory management, AVX accelerated binning kernels and triangle splitting, and small but important details like fast bounding box calculations and memory bandwidth savings. Adding up all the optimizations, our SBVH implementation substantially outperforms the best available alternative and rivals the speed of fast low-quality BVH algorithms. Our contribution enables full quality interactive BVH construction for scenes up to 1M triangles and considerably improves the workflow for large CAD models. Future work should focus on further lowering the bandwidth demand of our SBVH implementation and on exploring a NUMA-aware design for multi-socket platforms.

Acknowledgements

This research was supported by the German Research Foundation (DFG) as part of the the IRTG 2057 “Physical Modeling for Virtual Manufacturing Systems and Processes”.

References

[AKL13] AILA T., KARRAS T., LAINE S.: On quality metrics of bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG ’13, ACM, pp. 101–107. doi:10.1145/2492045.2492056. 1

[BHH13] BITTNER J., HAPALA M., HAVRAN V.: Fast insertion-based optimization of bounding volume hierarchies. *Computer Graphics Forum 32*, 1 (2013), 85–100. doi:10.1111/cgf.12000. 1

[BHH15] BITTNER J., HAPALA M., HAVRAN V.: Incremental BVH construction for ray tracing. *Computers and Graphics (Pergamon) 47* (2015), 135–144. doi:10.1016/j.cag.2014.12.001. 1

[EG08] ERNST M., GREINER G.: Multi bounding volume hierarchies. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing* (2008), RT ’08, IEEE Computer Society, pp. 35–40. doi:10.1109/RT.2008.4634618. 1

[FLPE15] FUETTERLING V., LOJEWSKI C., PFREUNDT F.-J., EBERT A.: Efficient Ray Tracing Kernels for Modern CPU Architectures. *Journal of Computer Graphics Techniques (JCGT) 4*, 4 (2015), 89–109. URL: http://jcgt.org/published/0004/04/05/. 1, 7

[GBDAM15] GANESTAM P., BARRINGER R., DOGGETT M., AKENINE-MÖLLER T.: Bonsai : Rapid Bounding Volume Hierarchy Generation using Mini Trees. *Journal of Computer Graphics Techniques (JCGT) 4*, 3 (2015), 23–42. URL: http://jcgt.org/published/0004/03/02/. 6

[GD16] GANESTAM P., DOGGETT M.: SAH guided spatial split partitioning for fast BVH construction. *Computer Graphics Forum (Proceedings of Eurographics) 35*, 2 (2016). 4

[GHFB13] GU Y., HE Y., FATAHALIAN K., BLELLOCH G.: Efficient bvh construction via approximate agglomerative clustering. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG ’13, ACM, pp. 81–88. doi:10.1145/2492045.2492054. 1

[GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D.: Simpler and faster hlbvh with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (New York, NY, USA, 2011), HPG ’11, ACM, pp. 59–64. doi:10.1145/2018323.2018333. 1

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl. 7*, 5 (May 1987), 14–20. doi:10.1109/MCG.1987.276983. 1

© The Eurographics Association 2016.

[Gut14] GUTHE M.: Latency Considerations of Depth-first GPU Ray Tracing. In *Eurographics 2014 - Short Papers* (2014), Galin E., Wand M., (Eds.). doi:10.2312/egsh.20141013. 7

[Hav01] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University Prague, 2001. 1

[Int16] Intel Architecture Instruction Set Extensions Programming Reference, 2016. URL: https://software.intel.com/en-us/isa-extensions. 1

[KA13] KARRAS T., AILA T.: Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 89–99. doi:10.1145/2492045.2492055. 1, 7

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH Construction on GPUs. *Computer Graphics Forum 28*, 2 (2009), 375–384. doi:10.1111/j.1467-8659.2009.01377.x. 1

[PGDS09] POPOV S., GEORGIEV I., DIMOV R., SLUSALLEK P.: Object Partitioning Considered Harmful: Space Subdivision for BVHs. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 15–22. doi:10.1145/1572769.1572772. 1

[PL10] PANTALEONI J., LUEBKE D.: HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry. In *High Performance Graphics* (2010), Doggett M., Laine S., Hunt W., (Eds.), The Eurographics Association. doi:10.2312/EGGH/HPG10/087-095. 1

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 7–13. doi:10.1145/1572769.1572771. 1, 2, 4

[Wal07] WALD I.: On fast construction of sah-based bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (2007), RT '07, IEEE Computer Society, pp. 33–40. doi:10.1109/RT.2007.4342588. 1, 5

[Wal12] WALD I.: Fast construction of SAH BVHs on the Intel Many Integrated Core (MIC) architecture. *IEEE Transactions on Visualization and Computer Graphics 18*, Mic (2012), 47–57. doi:10.1109/TVCG.2010.251. 1, 3, 4, 6

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (2007), 1–10. doi:10.1145/1186644.1186650. 1

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G., ERNST M.: Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics 33*, 4 (2014). doi:10.1145/2601097.2601199. 3, 6, 7

© The Eurographics Association 2016.