Fast BVH Construction on GPUs

C. Lauterbach[1] and M. Garland[2] and S. Sengupta[3] and D. Luebke[2] and D. Manocha[1]

[1]University of North Carolina at Chapel Hill [2]NVIDIA
[3]University of California Davis

Abstract

We present two novel parallel algorithms for rapidly constructing bounding volume hierarchies on manycore GPUs. The first uses a linear ordering derived from spatial Morton codes to build hierarchies extremely quickly and with high parallel scalability. The second is a top-down approach that uses the surface area heuristic (SAH) to build hierarchies optimized for fast ray tracing. Both algorithms are combined into a hybrid algorithm that removes existing bottlenecks in the algorithm for GPU construction performance and scalability leading to significantly decreased build time. The resulting hierarchies are close in to optimized SAH hierarchies, but the construction process is substantially faster, leading to a significant net benefit when both construction and traversal cost are accounted for. Our preliminary results show that current GPU architectures can compete with CPU implementations of hierarchy construction running on multicore systems. In practice, we can construct hierarchies of models with up to several million triangles and use them for fast ray tracing or other applications.

1. Introduction

Bounding volume hierarchies (BVHs) are widely used to accelerate intersection computations for ray tracing, collision detection, visibility culling, and similar applications. There is an extensive literature on fast computation of BVHs optimized for these applications. Our overarching goal is to design real-time hierarchy construction methods for scenes with non-rigid models undergoing deformations or topological changes, thus enabling interactive applications to process such content.

As parallelism has become the main force driving increased processor performance, hierarchy construction methods must be parallel if they are to scale well on future architectures. However, nearly all prior work on BVH construction has focused on purely serial construction algorithms. In this paper, we develop efficient parallel algorithms for constructing BVHs on manycore processors supporting thousands of concurrent threads. Our approach is to design algorithms that expose substantial amounts of fine-grained parallelism, thus fully exploiting massively multi-threaded processors.

The first contribution of this paper is a novel algorithm using spatial Morton codes to reduce the BVH construction problem to a simple sorting problem. This Linear Bounding Volume Hierarchy (LBVH) algorithm is focused on minimizing the cost of construction, while still producing BVHs of good quality. Our second contribution is a breadth-first queue-based algorithm for constructing BVHs optimized for ray tracing using the Surface Area Heuristic (SAH). This incorporates a novel and very fast method for performing the initial splits for the hierarchy in parallel, thus removing many of the existing bottlenecks in the construction algorithm. The SAH construction is best suited for the case where traversal cost is at a premium and construction cost is relatively unimportant; our experiments demonstrate that LBVH is over an order of magnitude faster than SAH construction, but can reduce ray tracing performance by up to 85% in some cases. Finally, our third contribution is a hybrid construction algorithm that builds the highest levels of the tree with the LBVH algorithm and builds the rest with the SAH algorithm. We demonstrate that this attains roughly half the construction speed of LBVH while retaining essentially all the quality of the pure SAH-based construction.

To explore the performance of our algorithms, we have implemented our algorithms in CUDA [NBGS08] and benchmarked their performance on an NVIDIA GeForce 280 GTX GPU. In practice, our techniques can compute BVHs composed of axis-aligned bounding boxes (AABBs) on models with hundreds of thousands triangles in less than 100 milliseconds.

Figure 1: ***Flamenco benchmark:*** *Screenshots from dynamic scene with 49K triangles. Our algorithm can build the optimized BVH on a NVIDIA 280 GTX GPU in 25ms per frame, allowing full real-time ray tracing at 11 fps at* 1024^2 *pixels.*

2. Background

Bounding volume hierarchies (BVHs) have been extensively used for ray tracing [RW80] and collision detection [Eri04]. In interactive ray tracing, object hierarchy based approaches have recently regained interest with the adaptation of ray packet techniques from kd-tree traversal on both the CPU [GM03, WBS07, LYTM06, WK06] and GPU [GPSS07]. While earlier approaches used simple update techniques to support dynamic data sets, fast construction techniques for rebuilding BVHs were introduced as well [WK06] and parallel CPU implementations have been developed [Wal07].

2.1. Parallel computing model

We focus on implementing our algorithms using the CUDA programming model [NBGS08] and running them on modern NVIDIA GPUs [LNOM08]. A few basic attributes of this platform are relevant to our algorithmic discussions. CUDA programs consist of a sequential host program that can initiate parallel kernels on the GPU device. A given kernel executes a single program across many parallel threads. These threads are decomposed into *thread blocks*; threads within a given block may efficiently synchronize with each and have shared access to per-block on-chip memory. Consequently, three design principles are crucial for achieving scalable algorithms: (1) decomposing work into chunks suitable to be processed by each thread block, (2) exploiting the on-chip memory given to each block, and (3) exposing enough fine-grained parallelism to exploit the massively multi-threaded GPU hardware.

To analyze the complexity of parallel algorithms, we adopt the standard approach of analyzing both the *work* and *depth* complexity. Briefly, if a given computation is expressed as a dependence graph of parallel tasks, then its work complexity is the total amount of work performed by all tasks and its depth complexity is the length of the critical path in the graph. It is well-known that serial BVH construction has the same $O(n \log n)$ complexity as comparison-based sort. Therefore, an asymptotically efficient parallel BVH construction algorithm should perform at most $O(n \log n)$ with a depth of at most $O(\log n)$, which are the bounds on comparison-based parallel sort.

2.2. GPU traversal and construction

The massive parallelism of programmable GPUs naturally lends itself to inherently parallel problems such as ray tracing. Early implementations [CHH02, PBMH02] were constrained by architectural limitations, but modern approaches use hierarchical acceleration structures such as k-d trees [FS05, HSHH07] and BVHs [TS05, GPSS07]. While these approaches essentially implement techniques similar to those used in CPU ray tracing, other specialized techniques for stack traversal have also been developed [HSHH07, PGSS07]. Some support for scenes with deformable or dynamic geometry was introduced with geometry images [CHCH06] and a ray hierarchy based approach [RAH07]. Going beyond ray tracing, similar hierarchies have also been used for GPU-based collision detection [Eri04, GKJ*05].

Most of these approaches have tended to rely on serial algorithms running on the CPU to construct the necessary hierarchical acceleration structures. While once a necessary restriction due to architectural limitations, modern GPUs provide all the facilities necessary to implement hierarchy construction directly. Doing so should provide a strong benefit, as building structures directly on the GPU avoids the need for the relatively expensive latency introduced by frequently copying data structures between CPU and GPU memory spaces.

There has been some initial work on constructing hierarchies on multi-core CPU processors, although gains appear to decrease sharply even on CPUs with a low number of cores [PGSS06], mainly due to memory bandwidth limitations [Wal07]. Very recently, initial work has been shown on constructing spatial hierarchies on GPUs [ZHWG08, ZGHG08, AGCA08]. We present a more detailed comparison in section 6.

3. LBVH hierarchy construction

The simplest approach to parallelizing BVH construction is to reduce it to a sorting problem. This is the approach taken by our LBVH construction which begins by ordering all n input primitives along a space-filling curve, thus "linearizing" them into a single fixed sequence of length n. Given this sequence, we construct the tree by recursively splitting intervals of this sequence and creating corresponding nodes. The root corresponds to the interval $[0,n)$, its children to a partition $[0,m), [m,n)$ of the root interval, and so on. Thus every node in the BVH will correspond to a range of indices $[l_i, r_i)$ in the sequence of primitives.

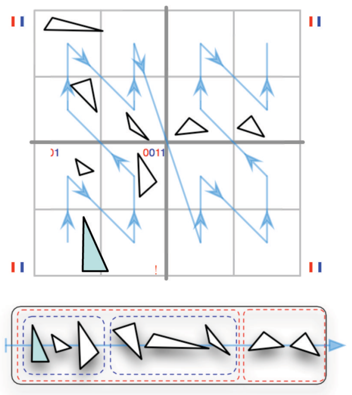


Figure 2: *Example 2-D Morton code ordering of triangles with the first two levels of the hierarchy. Blue and red bits indicate x and y axes, respectively.*

3.1. Tree Construction using Morton Codes

We use the well-known space-filling *Morton curve*—also known as the Lebesgue and z-order curve—for ordering primitives. This is because the Morton index code, which determines a primitive's order along the curve, can be computed directly from its geometric coordinates, whereas other space-filling curves require more expensive constructions to determine the ordering of primitives.

We assume that each input primitive is represented by an AABB and that the enclosing AABB of the entire input geometry is known. We take the barycenter of each primitive AABB as its representative point. By constructing a $2^k \times 2^k \times 2^k$ lattice within the enclosing AABB, we can quantize each of the 3 coordinates of the representative points into k-bit integers. The $3k$-bit *Morton code* for a point is constructed by interleaving the successive bits of these quantized coordinates. Figure 2 shows a 2-D example of this construction. Sorting the representative points in increasing order of their Morton codes will lay out these points in order along a Morton curve. Thus, it will also order the corresponding primitives in a spatially coherent way. Because of this, sorting geometric primitives according to their Morton code has been used for improving cache coherence in large geometric databases [PF01].

Once we have assigned a Morton code to every primitive, we can easily construct a BVH by bucketing primitives based on the bits of their Morton codes. We construct the first level split by examining the most significant bit of all codes, placing those with 0 and 1 bits in the first and second child, respectively, of the root. See Figure 2 for an example. We apply this bucketing procedure recursively in each child, looking at the second most significant bit. In each recursive step, we look at the next bit in the Morton code until all bits are consumed.

Note that this recursive partitioning is equivalent to the steps of an most-significant-bit radix-2 sort using the Morton codes as the keys. By generalizing this radix-2 algorithm to a radix-b algorithm—one which looks at $\log_2 b$ bits at a time rather than 1—we can also trivially produce trees with branching factor 2^b rather than simply binary trees. We note that for the radix-8 case, the recursive decomposition based on Morton codes is equivalent to an octree decomposition of the enclosing AABB.

3.2. Data-Parallel LBVH Construction

One of the main virtues of the LBVH construction is that it is inherently parallel. As noted above, the correct ordering of primitives can be computed simply by sorting them with their Morton codes as the sort keys. Furthermore, the Morton codes themselves encode all the necessary information about where in the tree a particular primitive will go.

Given a list of primitives, each with its associated Morton code, we sort them using an efficient parallel radix sort [SHG08]. Since the Morton codes may often have only zeroes in the most significant bits, it is more efficient to use radix-sort by least significant bit here. In the sorted sequence, consider two adjacent primitives at positions i and $i+1$. The bits of the Morton code uniquely encode the path that a point will take from the root to the leaf that contains only the primitive. Thus, we can determine their least common ancestor—the node farthest from the root whose subtree contains both—by determining the most significant bit in which their Morton codes differ. The bits above this point, shared by both primitives, uniquely encode the least common ancestor node.

To construct the LBVH from the sorted primitive sequence, we need to determine the interval dividing lines at which nodes begin and end. In other words, for primitives i and $i+1$, we want to know at what levels (if any) they should be split into separate nodes. We can construct these "split lists" for each adjacent pair in parallel, using the Morton codes. As we have just observed, the levels of the tree at which these two primitives are in separate sub-trees are those levels corresponding to the bit after which their Morton codes are no longer identical. If they first differ in bit position h, then we record the fact that a split must exist between them at each level from h to $3k$, since $3k$ is the maximum depth of the tree. We record this "split list" as a list of pairs $[(i,h),(i,h+1),\ldots,(i,3k)]$ containing the index of the triangle and the levels of the splits between the primitives. If we perform this computation for each adjacent pair and concatenate all such lists, we get a list of all splits in the tree sorted by the index of the triangles in the linear Morton order. Once we have this list, we resort it by level (i.e., the second item of each pair). Now we have a list where, for each level, we have a list of all the splits that should occur on this level. This tells us precisely what we want to know, namely the intervals in the triangle list spanned by the various nodes at this level.

One unwanted side effect of this parallel construction is that it can create chains of singleton nodes in the tree. To deal with this we must simply add an additional post-process that starts at the leaves of the tree and walks up to the root, rolling up singleton chains in the process.

As a final step, the representation of the BVH as splits by level needs to be converted in a form that is useable as a top-down hierarchy, mainly by computing the correct bounding boxes for each node, which is a similar process to a regular BVH refitting step and can be achieved in linear time. In addition, we also compute the explicit child pointers for top-down traversal which follow directly from the interval information in the splits.

Finally, note that both sorts required by the parallel LBVH construction can be implemented using a radix sort algorithm. The asymptotic work complexity of the LBVH construction is thus $O(n)$. Its asymptotic depth complexity is the same as parallel scan, which is $O(\log n)$. It is thus efficient, as it performs asymptotically the same amount of work as the corresponding sequential sort and has a logarithmic depth.

4. SAH hierarchy construction

The main disadvantage of the LBVH algorithm described in the previous section is that it does not build hierarchies that are optimized for performance in ray tracing since it uniformly subdivides space at the median. We now present an algorithm for GPU object hierarchy construction using the surface area heuristic (SAH). Before presenting the actual algorithm, we will first summarize how state-of-the-art implementations on the CPU build SAH-optimized hierarchies.

4.1. SAH hierarchy construction

The surface area heuristic [GS87, MB90, Hav00] method for building hierarchies can be applied to many types of hierarchical acceleration structures – among them object and spatial hierarchies – since it has been shown to be a good indicator of expected ray intersections. The time complexity for top-down construction of a hierarchy for n triangles is $O(n \log n)$ due to the equivalence to sorting, and research has shown [WH06] that SAH optimized construction can also be achieved in the same bound. Top-down builders proceed by recursively splitting the set of geometric primitives — usually into two parts per step, resulting in a binary tree. The manner in which the split is performed can have a large impact on the performance of a ray tracer or other application using the BVH. The SAH provides a cost function that allows to evaluate all possible split positions at a node and then pick the one with the lowest cost. Computing the cost function for each possible split position can be costly and requires a full sort, although a careful implementation needs to perform this sort only once for each of the 3 axes and then reuse the information [WH06]. Recently, new approximate SAH approaches have moved towards only sampling several split positions per split and basing the decision on this subset of split positions [PGSS06, HMS06, Wal07]. Results have shown that this only has a negligible impact on rendering performance, but usually is an order of magnitude faster during construction.

Other recent work has concentrated on parallel construction on multi-core CPUs in order to achieve interactive per-frame hierarchy rebuilds [PGSS06, SSK07, Wal07]. However, unlike ray tracing itself, hierarchy construction does not scale well with the number of processors. In particular, the main issues are that the first steps of the top-down algorithm for construction are not easily parallelized, and more importantly that the implementation may become bandwidth-limited. Since memory bandwidth has traditionally not increased by Moore's law like computing power, this remains a serious issue.

4.2. GPU SAH construction

The main challenge for a GPU algorithm for hierarchy construction is that it has to be sufficiently parallel so as to exploit the computational power of the processors. In a top-down construction algorithm, there are two main ways to introduce parallelism: a) process multiple splits in parallel and b) parallelize an actual split operation. In our approach, we use multiple cores to run as many node splits in parallel as possible and then use the data parallel computation units in the processors to accelerate the SAH evaluation as well as the actual split operation. In the further discussion we will use some terms and common parallel programming primitives from the PRAM model that can be easily implemented on vector architectures such as GPUs.

4.2.1. Breadth-first construction using work queues

We now present our algorithm for parallelizing the individual splits across all the cores. During top-down construction, each split results in new nodes that still need to be processed, i.e. split as well. Since each of the splits can be processed totally independently, processing all the open splits in parallel on multiple cores is an easy way to speed up construction. The simplest approach to manage all this work is to introduce a queue that stores all the open splits such that each processor can fetch work whenever it is finished working on a node, as well as put new splits on the work queue when finishing up. However, even though current GPU architectures support atomic operations that would make global synchronization possible, the problem is that the same architectures do not provide a memory consistency model that would ensure writes to the queue are seen in the correct order by other processors. Thus, implementing a global work queue is currently not practical yet. Instead, we opt for an iterative approach that processes all the currently open splits in parallel and writes out new splits, but then performs a queue mainte-

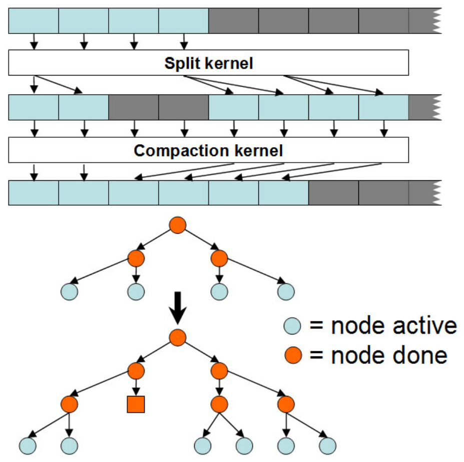


Figure 3: ***Work queue construction:*** *By using two work queues, we can run all active splits in parallel. Since the split kernel may not output new splits, a compaction step after each split level removes empty space in the queue and allows it to be used in the next step.*

nance step in between to provide a valid queue for the next step (similar in spirit to [Hor05]).

In a binary tree, each split can result in either 0, 1 or 2 new open splits as a result. Thus, if we have m open splits, we only need a work queue of at most size $2m$ to hold all potential new splits. By using two work queues, one as an input and one as an output, we can perform parallel splits without any explicit coordination between them. In our algorithm, each block i reads in its split item from the input queue and processes it (as described in the next section) and then either writes out new splits or a null split to the output queue at positions $2i$ and $2i+1$ (also see Fig. 3). After that step, the output work queue may have several null splits that need to be eliminated. Therefore, we run a compaction kernel (such as described in [SHG08]) on the queue and write the result into the input queue. We again run the split algorithm on the new input queue as long as there are still active splits left. Note that this means that we generate and process all the nodes in the tree in breadth-first order, whereas recursive or stack-based CPU approaches typically proceed in depth-first order (Fig. 3). Note that this form of work organization does not add work overhead and the the overall work complexity is still $O(n \log n)$.

4.2.2. Data-Parallel SAH split

In general, performing a SAH split for an object hierarchy consists of two steps:

1. Determine the best split position by evaluating the SAH
2. Reorder the primitives (or the indices to the primitives) such that the order in the global list corresponds to the new split

Note that unlike spatial partitioning approaches such as kd-trees each primitive can only be referenced on one side of the split (commonly determined by the location of the centroid of the primitive), which means that the reordering can be performed in-place and no additional memory is needed. All of these steps can be performed by exploiting data parallelism and using common parallel operations such as reductions and prefix sums, and each split is executed on one core.

We perform approximate SAH computation as described above and generate k uniformly sampled split candidates for each of the three axes, and then use $3k$ threads so that we test all the samples in parallel. Note that when k is larger or equal to the number of primitives in the split, we can instead use the positions of the primitives as the split candidates, in which case the algorithm produces exactly the same result as the full SAH evaluation. The algorithm loads each primitive in the split in turn while each thread tests its position in reference to its split position and updates the SAH information (such as bounding boxes for the left and right node) accordingly. Once all primitives have been tested, each thread computes the SAH cost for its split candidate. Finally, a parallel reduction using the minimum operator finds the split candidate sample with the lowest cost. We also test the SAH cost for not splitting the current node here and compare it to the computed minimal split cost. If it is lower, then we abort the split operation and make the node a leaf in the hierarchy.

Given the split coordinate as determined by the previous step, our algorithm then needs to sort the primitives into either the left or the right child node. To avoid copying the geometry information, we only reorder the indices referencing the primitives instead. To parallelize this reordering step, we go over all the indices in chunks of a size equal to the number of threads in the block. For each chunk, each thread reads in the index and set a flag to 1 if the respective primitive is to the left or 0 if on the right. We then perform a parallel prefix sum over the flags. Based on that, each thread can determine the new position to store its index. Finally, we store the bounding box information computed during SAH computation in both the child nodes and save back the new split information to the output split list for the next step if there is more than one primitive left in the respective node.

4.2.3. Small split optimizations

The algorithm as described above generates a hierarchy practically identical in quality to CPU-based approximate SAH techniques (see the results section). In practice, we can efficiently use more samples due to high data parallelism and the SAH quality may be even better since the main change has been to reorganize the split operation such that it uses parallelism at every stage. Fig. 4 a) illustrates the performance characteristics of the algorithm by showing the time

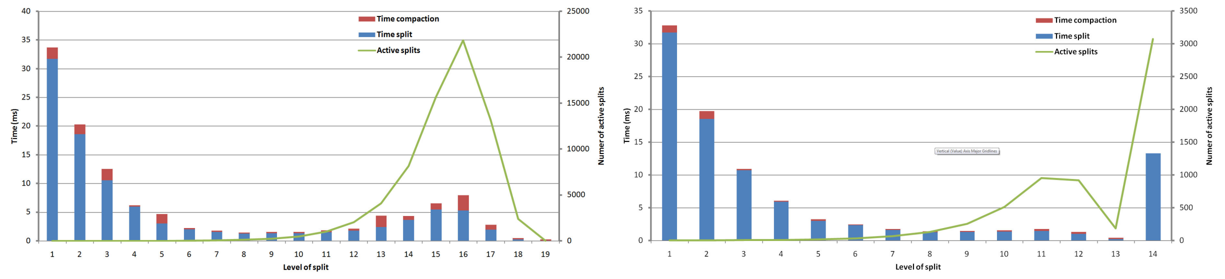


Figure 4: ***Construction timings per level:*** *In order to evaluate our SAH construction algorithm, we show the time taken to run the split as well as compaction kernels for the 69K Bunny model per level of the tree, starting with the one split on level 1. We also show how many splits are active at any level. Left: normal construction algorithm. Right: construction with small split kernel as described in the text.*

spent during construction by level of the split. There are two main bottlenecks in the algorithm: similar to CPU implementations, the initial splits at the top levels of the hierarchy are slow due to lack of processor parallelism and large numbers of very small splits at the end. All further splits are much faster even though there is much more work done per level because computational resources are more fully utilized and processors can hide memory latency by switching between active tasks. We present an improved scheme to improve the parallelism at the top levels in section 4.3, but we can also improve on the performance at the lower levels of the hierarchy. The main sources of overhead here are a) higher compaction costs since a high number of splits are generated during each step and b) low vector utilization due to only processing a few primitives per split. There is also some constant administrative overhead during each operation for reading in information about the split etc. that becomes more significant as the actual computational intensity decreases. We address these issues by using a different split kernel for all splits with sizes below a specified threshold and modify the compaction kernel such that these splits are automatically filtered from the main work queue into a small split queue. Once all the normal splits are performed, we can run the small split kernel on all the elements in that queue once.

The main idea about the new split kernel is to use each processor's local memory to maintain a local work queue for all the splits in the sub-tree defined by the input split, as well as to keep all the geometric primitives in the sub-tree in local memory as well. Since local memory is very fast, we can then run the complete sub-tree's construction entirely without waiting for memory accesses. Essentially, we are using this memory as an explicitly managed cache to reduce the memory bandwidth needed for construction. In addition, we use as few threads as possible in the kernel to maximize utilization of the vector operations. The threshold value of primitives for a split to be "small" depends on how much geometry data can be fit into local memory, i.e. the cache size. In our case, we have set both the thread count as well as the threshold to 32, which for our GPU architecture is the effective SIMD width and will also fill up most of local memory. Fig. 4 b) shows the impact of introducing this split kernel. As the normal construction is run, all splits smaller than the threshold are removed from the main work queue, so the number of splits falls much quicker than previously. At the very last split step, the small split kernel is invoked and complete processes all the remaining sub-trees in one run. Overall, we have found that this leads to a 15-20% speedup in our test cases due to practically full utilization of the computational resources and reduced overhead for queue compaction.

4.3. Hybrid GPU construction algorithm

It is also possible to combine both algorithms described in this paper into a hybrid builder. As was mentioned, the main bottleneck in the SAH construction algorithm is lack of parallelism in the initial splits when most processors are idle. The LBVH algorithm with radix sort gives us the possibility of using only a subset of the bits in the Morton code. In this case, our algorithm builds a very shallow hierarchy with large numbers of primitives at the leafs. We can then treat all the leafs in that hierarchy as still active splits and process them in parallel with the SAH construction algorithm. Essentially, this is similar to the bounding interval hierarchy [WK06] (BIH) and grid-assisted [Wal07] build methods, but in a highly parallel approach. Overall, this combines the speed and scalability of the LBVH algorithm with the ray tracing performance optimizations in the SAH algorithm.

5. Results and Analysis

We now highlight results from the algorithms described in the previous sections on several benchmark cases and scenarios. All algorithms are run on a Intel Xeon X5355 system at 2.66GHz running Microsoft Windows XP with a NVIDIA Geforce 280 GTX graphics card with 1 GB of memory and were implemented using the NVIDIA CUDA programming

Model	Tris	CPU SAH	GPU SAH	LBVH	Hybrid	Parallel SAH [Wal07]	Full SAH [Wal07]
Flamenco	49K	144ms 30fps/99%	85ms 30.3fps/100%	9.8ms 12.4fps/41%	17ms 29.9fps/99%	n/a n/a	n/a 100%
Sibenik	82K	231ms 21.4fps/97%	144ms 21.7fps/98%	10ms 3.5fps/16%	30ms 21.4fps/97%	n/a n/a	n/a 100%
Fairy	174K	661ms 11.5fps/98%	488ms 21.7fps/100%	10.3ms 1.8fps/15%	124ms 11.6fps/99%	21ms 93%	860ms 100%
Bunny/Dragon	252K	842ms 7.8fps/100%	403ms 7.75fps/100%	17ms 7.3fps/94%	66ms 7.6fps/98%	20ms 98%	1160ms 100%
Conference	284K	819ms 24.4fps/91%	477ms 24.5fps/91%	19ms 6.7fps/25%	105ms 22.9fps/85%	26ms 86%	1320ms 100%
Soda Hall	1.5M	6176ms 20.8fps/98%	2390ms 21.4fps/101%	66ms 3fps/14%	445ms 20.7fps/98%	n/a n/a	n/a 100%

Table 1: ***Construction timings and hierarchy quality:*** *First row for each scene: Timings (in ms) for complete hierarchy construction. Second row: relative and absolute ray tracing performance (in fps) on our GPU ray tracer compared to full SAH solution at* 1024^2 *resolution and primary visbility only. CPU SAH is our non-optimized approximate SAH implementation using just one core, GPU SAH is the algorithm as presented in section 4.2 and Hybrid the combined algorithm as presented in section 4.3. The parallel and full SAH are both from the grid-BVH build in [Wal07] and were generated on 8 and 1 core of an Intel Xeon system at 2.6 GHz, respectively.*

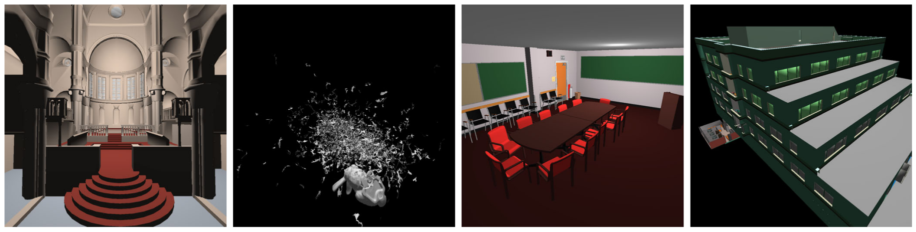


Figure 5: ***Benchmark models:*** *Our benchmark scenes used to generate results. From left to right: Sibenik cathedral (80K tris), Bunny/Dragon animation (252K tris), Conference room (284K tris), Soda Hall (2M tris).*

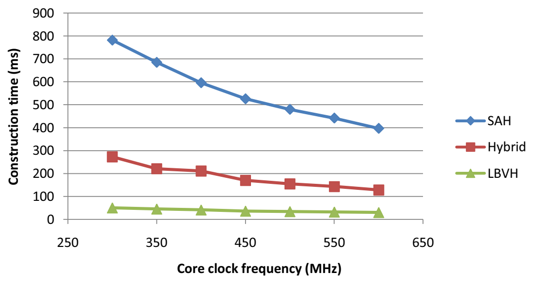
language [NBGS08]. We use several benchmark scenes chosen to both allow comparison to other published approaches as well as to cover several different model characteristics such as architectural and CAD scenes as well as scanned models (see Fig. 5). Note that all benchmark timings cover the full construction process starting with building the initial bounding boxes, but do not include any CPU-GPU copy of geometry. In our benchmark results for the hybrid algorithm, we used the LBVH algorithm for performing the first 6 levels of splits before switching to SAH construction. All performance results and images in the paper were produced with a BVH-based ray tracer running on the GPU. We implemented a relatively simple and unoptimized packet-based BVH ray tracer similar to [GPSS07] in CUDA and tested it on our benchmark scenes while rebuilding the hierarchy for each frame. In combination with the hierarchy construction, this allows interactive ray tracing of arbitrary dynamic scenes on the GPU.

To justify both the approximate SAH building algorithm as well as the hybrid approach, we first compare relative rendering speed of all the algorithms presented in this paper to a hierarchy built on the CPU with a full SAH construction algorithm such as described in [WBS07]. Rendering is performed using a standard CPU BVH ray tracer using ray packets [LYTM06, WBS07]. The results listed in table 1 demonstrate that for most scenes the approximate SAH has close to no impact compared to a full SAH build and that even the hybrid build is very close to the reference results. This backs results of similar splitting approaches in [WK06, Wal07]. Note that in some cases the approximate or hybrid algorithm is in fact faster than the reference implementation. The SAH is only a local heuristic and thus it is possible that a split that is non-optimal in the SAH sense may still result in better performance in the actual ray tracer. The efficiency of the LBVH construction is highly scene dependent. It can provide adequate quality for scenes with evenly distributed geometry such as the Dragon/Bunny model, but for architectural and CAD scenes the performance is clearly inferior. Because it uses a fixed number of bits per axis, its quality can degrade when very small details exist in a large environment (the classic *teapot in stadium*scenario). Thus, the Fairy scene (benchmark 3 in Table 1) with a complex model in the middle of a less complex environment is an example of this problem. In the hybrid algorithm, it is always possible to limit the impact by limiting the number of splits performed by LBVH. Furthermore, this problem is mitigated if the scene has some pre-existing structure, say from a scene graph that can assist the construction procedure. Having primitives with highly varying sizes can also have an impact on performance here since the classification according to Morton code does not take this into account. However, this is usually also a problem for other construction algorithms and the common solution would be to subdivide large primitives [EG07] before construction. Table 1 also shows timings for the actual construction with our different approaches and also lists published numbers from a fast parallel CPU BVH builder [Wal07] as well as our non-parallel CPU implementation of a sampling based SAH approach (similar to [HMS06, Wal07]) using 8 samples/split. Note that since the GPU ray tracer uses a higher number of samples (in our case 64), it can provide slightly better hierarchy quality compared to the CPU solution. The full LBVH is the fastest builder, but is dominated by kernel call and other overhead for small models.

Analysis: Current GPU architectures have several features that would make them suitable for hierarchy construction. First, thanks to special graphics memory they have significantly higher memory bandwidth. At the same time, even though the individual processing units do not have a general cache, they have explicitly managed fast local memory. Thus, if data for the current computation can be loaded from main memory and held in local memory, memory access and bandwidth are very fast. GPUs also have very high available parallelism both in terms of independent processors as well as data parallel units. However, exploiting this parallelism in the construction algorithm now becomes an harder challenge. Unlike the thread parallelism in CPU implementations, using data parallelism is both more important as well as difficult due to higher SIMD width.

The memory footprint of our construction algorithm is relatively low since object hierarchy construction does not require temporary memory and all elements can just be reordered in-place. We can size the main work queues conservatively since for n primitives there can be at most $n/2$ splits active at a time. Otherwise, the algorithm stores an AABB per primitive, the main index list as well as an array for holding the BVH nodes (which can be conservatively sized at the theoretical maximum of $2n-1$ nodes.) Overall, our algorithm uses up to 113 bytes/triangle during construction including the BVH, assuming the worst case of one triangle per leaf. This allows construction of hierarchies for multi-million triangle models on current GPUs without problems.

We also analyzed the current bottlenecks in our construction algorithms. Figure 6 shows the impact of independently changing core (i.e. processing) and memory clock for our construction methods. It is obvious that all three are currently bound by computation, and both the hybrid and



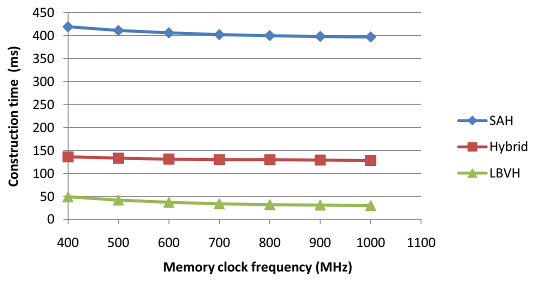


Figure 6: ***Bottleneck analysis:*** *Relative performance of the three construction algorithms when modifying the core processor clock (top) and the memory clock (bottom).*

SAH algorithms are not limited by memory bandwidth. The LBVH algorithm shows the highest dependence on memory speed: since GPU sort is known [GGKM06] to be bandwidth limited and construction must have the same lower bounds as sorting, we take this as a sign that the LBVH build is close to the limit of achievable scalability. Overall, this means that the construction algorithms will very likely scale well with added computational power in future architectures.

We also analyzed where time is spent in the construction. Figure 7 shows a break-up of relative time for each of our benchmark scenes into several components (as described in more detail in section 4): *SAH split* is the time for finding the optimal subdivision, *Reorder* is the time spent reordering the triangles according to the split, *Compaction* consists of the time for compacting and maintaining the work queues between splits and *LBVH* is the initial split time in the hybrid construction (for 6 levels). The remaining time (*Rest*) is spent reading in and writing back BVH node information, setting up splits and otherwise joining the rest of the steps together. Note that we did not include the time for computing the AABBs and Morton codes here as it makes up far less than 1% of the total time in all benchmarks. Overall, the results show that the full SAH build is clearly dominated by the cost of evaluating the SAH split information, while the hybrid build is more balanced. Computing the initial splits using the LBVH algorithm only takes a relatively small amount of time, but reduces both the overall construction time as well as some of the cost in work queue management.

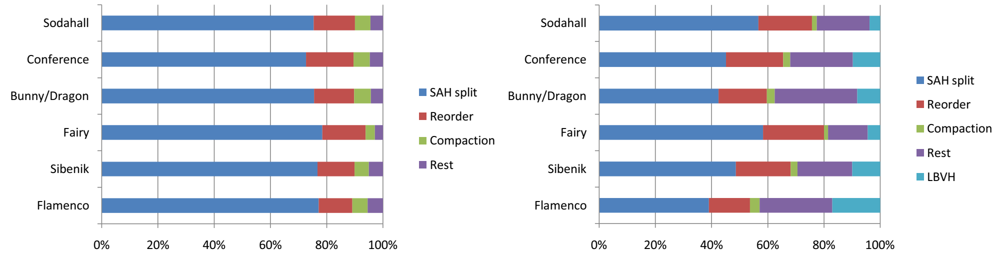


Figure 7: ***Time spent:*** *Split-up of the total time spent in construction into each part of the algorithm. Left: Full SAH build. Right: Hybrid build. The "rest" times consist of all operations not explicitly in other parts, such as reading and writing BVH nodes and setting up all the other components.*

Comparison: There has been some concurrent work on GPU hierarchy construction focused on building kd-trees [ZHWG08]. Our approach is similar in the organization of the splits by using a work queue, but the authors use spatial median splits after sorting primitives into chunks in order to speed up the top-level split computation. Despite the similarities in both approaches, we would like to point out some differences in BVH and kd-tree construction. In particular, the memory overhead in object hierarchy construction is very small, whereas dynamic allocation in kd-tree construction so far limits the GPU implementation — the current kd-tree algorithm would need about 1GB of memory to construct the kd-tree for a model with one million triangles, whereas we could support about ten times that. On the other hand, BVH splits are intrinsically more computationally intensive as the split kernel has to compute bounding boxes for each split candidate whereas kd-tree splits are directly given by the parent's bounding box and the split plane. Thus, the main inner loop need to load 3 times the data (all 6 bounds as opposed to 2) and a corresponding increase in computation. This difference may explain that our construction times are slightly higher than the kd-tree numbers in [ZHWG08] for the Fairy model (77ms compared to 124ms in our hybrid approach.)

A similar approach to our linear BVH construction has also been proposed, but for octrees of points [ZGHG08]. The authors use the Morton order in combination with a sort to build the hierarchy. However, the problem of building a spatial hierarchy on points is different than building an object hierarchy on AABBs as in our case, so the rest of the algorithm differs. We also observe that even though our algorithm has to perform more work during construction, our implementation is faster, presumably due to more efficient sort.

6. Conclusion and future work

In conclusion, we have presented algorithms for performing fast construction of optimized object hierarchies on current GPUs. We have shown that standard SAH techniques can be implemented while exploiting both processor as well as data parallelism with a reorganization of the construction process. We have also introduced a novel construction scheme based on fast radix sort that can be used both as a standalone builder as well as part of the SAH construction process and we show how the combination leads to the fastest implementation. We believe that our approach will also be practical on different highly parallel processors such as the CELL processor and on future upcoming architectures like Larrabee with a high number of cores and increased data parallelism. On current GPU architectures, there are still some limitations that could be removed in the future for better performance. For one, relatively static scheduling and lack of synchronization currently prevents the implementation of a truly shared work queue in main memory which would remove most of the overhead associated with maintaining the queue and repeatedly invoking the split kernel.

There are many opportunities for future work. It would be interesting to integrate the construction algorithm into a full rendering pipeline and combine a hybrid rasterization and ray tracing algorithm. In that context, we would also like to investigate using hierarchy construction directly on geometric primitives created on the GPU, e.g. using tessellation techniques. In additon, it would be interesting to further extent our algorithm to work on systems with multiple GPUs. Due to higher computational power GPUs should also be a good candidate for using other bounding volumes such as OBBs or k-DOPs in the construction process, and our algorithm show easily extend to using these. Finally, we would be interested to explore other applications of BVHs such as continuous collision detection and response computation, multi-agent navigation and more.

Acknowledgments

This work was supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134, 0429583 and 0404088, DARPA/RDECOM Contract N61339-04-C-0043, Disruptive Technology Office and Intel.

References

[AGCA08] Ajmera P., Goradia R., Chandran S., Aluru S.: Fast, parallel, gpu-based construction of space filling curves and octrees. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2008), ACM, pp. 1–1.

[CHCH06] Carr N. A., Hoberock J., Crane K., Hart J. C.: Fast GPU ray tracing of dynamic meshes using geometry images. In *GI '06: Proceedings of Graphics Interface 2006* (Toronto, Ont., Canada, Canada, 2006), Canadian Information Processing Society, pp. 203–209.

[CHH02] Carr N. A., Hall J. D., Hart J. C.: The Ray Engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Aire-la-Ville, Switzerland, Switzerland, 2002), Eurographics Association, pp. 37–46.

[EG07] Ernst M., Greiner G.: Early split clipping for bounding volume hierarchies. In *Proc. IEEE Symposium on Interactive Ray Tracing* (2007).

[Eri04] Ericson C.: *Real-Time Collision Detection*. Morgan Kaufmann, 2004.

[FS05] Foley T., Sugerman J.: KD-tree acceleration structures for a GPU raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (New York, NY, USA, 2005), ACM, pp. 15–22.

[GGKM06] Govindaraju N., Gray J., Kumar R., Manocha D.: GPUTeraSort: High performance graphics coprocessor sorting for large database management. *Proc. of ACM SIGMOD* (2006).

[GKJ*05] Govindaraju N., Knott D., Jain N., Kabal I., Tamstorf R., Gayle R., Lin M., Manocha D.: Collision detection between deformable models using chromatic decomposition. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH) 24*, 3 (2005), 991–999.

[GM03] Geimer M., Müller S.: A Cross-Platform Framework for Interactive Ray Tracing. In *Graphiktag im Rahmen der GI Jahrestagung* (Frankfurt am Main, 2003).

[GPSS07] Günther J., Popov S., Seidel H.-P., Slusallek P.: Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007* (2007), pp. 113Ű–118.

[GS87] Goldsmith J., Salmon J.: Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl. 7*, 5 (1987), 14–20.

[Hav00] Havran V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[HMS06] Hunt W., Mark W. R., Stoll G.: Fast kd-tree construction with an adaptive error-bounded heuristic. In *2006 IEEE Symposium on Interactive Ray Tracing* (September 2006), IEEE.

[Hor05] Horn D.: Stream reduction operations for gpgpu applications. *GPU Gems 2* (2005).

[HSHH07] Horn D. R., Sugerman J., Houston M., Hanrahan P.: Interactive k-d tree GPU raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2007), ACM, pp. 167–174.

[LNOM08] Lindholm E., Nickolls J., Oberman S., Montrym J.: NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro 28*, 2 (2008), 39–55.

[LYTM06] Lauterbach C., Yoon S., Tuft D., Manocha D.: RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. *IEEE Symposium on Interactive Ray Tracing* (2006).

[MB90] MacDonald J. D., Booth K. S.: Heuristics for ray tracing using space subdivision. *Visual Computer* (1990).

[NBGS08] Nickolls J., Buck I., Garland M., Skadron K.: Scalable parallel programming with CUDA. *Queue 6*, 2 (2008), 40–53.

[PBMH02] Purcell T. J., Buck I., Mark W. R., Hanrahan P.: Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), ACM Press, pp. 703–712.

[PF01] Pascucci V., Frank R. J.: Global static indexing for real-time exploration of very large regular grids. *Super Computing* (2001).

[PGSS06] Popov S., Günther J., Seidel H.-P., Slusallek P.: Experiences with streaming construction of SAH KD-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 89–94.

[PGSS07] Popov S., Günther J., Seidel H.-P., Slusallek P.: Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum (Proc. EUROGRAPHICS) 26*, 3 (2007), 415–424.

[RAH07] Roger D., Assarsson U., Holzschuch N.: Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU. In *Rendering Techniques 2007 (Proceedings of the Eurographics Symposium on Rendering)* (June 2007), Kautz J., Pattanaik S., (Eds.), Eurographics and ACM/SIGGRAPH, the Eurographics Association, pp. 99–110.

[RW80] Rubin S. M., Whitted T.: A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics 14*, 3 (July 1980), 110–116.

[SHG08] Satish N., Harris M., Garland M.: Designing efficient sorting algorithms for manycore GPUs. *Under review* (2008).

[SSK07] Shevtsov M., Soupikov A., Kapustin A.: Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum 26*, 3 (September 2007), 395–404.

[TS05] Thrane N., Simonsen L.-O.: *A comparison of acceleration structures for GPU assisted ray tracing*. Master's thesis, University of Aarhus, Aarhus, Denmark, 2005.

[Wal07] Wald I.: On fast Construction of SAH based Bounding Volume Hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing* (2007).

[WBS07] Wald I., Boulos S., Shirley P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1 (2007).

[WH06] Wald I., Havran V.: On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proc. of IEEE Symp. on Interactive Ray Tracing* (2006), pp. 61–69.

[WK06] Wächter C., Keller A.: Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques 2006: Eurographics Symposium on Rendering.* (2006).

[ZGHG08] Zhou K., Gong M., Huang X., Guo B.: *Highly Parallel Surface Reconstruction*. Tech. Rep. MSR-TR-2008-53, Microsoft Technical Report, April 2008.

[ZHWG08] Zhou K., Hou Q., Wang R., Guo B.: Real-time kd-tree construction on graphics hardware. In *Proc. SIGGRAPH Asia, to appear* (2008).