Simpler and Faster HLBVH with Work Queues

Kirill Garanzha*
NVIDIA
Keldysh Institute of Applied Mathematics

Jacopo Pantaleoni*
NVIDIA Research

David McAllister*
NVIDIA

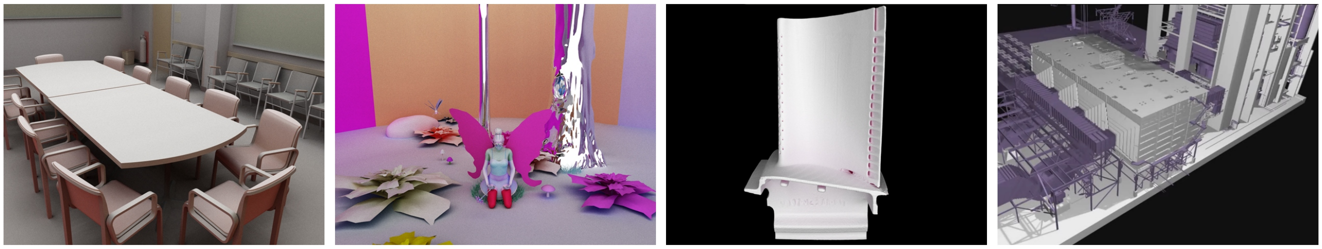


Figure 1: *Some of our test scenes, from left to right: Conference, Fairy Forest, Turbine Blade and Power Plant.*

Abstract

A recently developed algorithm called Hierachical Linear Bounding Volume Hierarchies (HLBVH) has demonstrated the feasibility of reconstructing the spatial index needed for ray tracing in real-time, even in the presence of millions of fully dynamic triangles. In this work we present a simpler and faster variant of HLBVH, where all the complex bookkeeping of prefix sums, compaction and partial breadth-first tree traversal needed for spatial partitioning has been replaced with an elegant pipeline built on top of efficient work queues and binary search. The new algorithm is both faster and more memory efficient, removing the need for temporary storage of geometry data for intermediate computations. Finally, the same pipeline has been extended to parallelize the construction of the top-level SAH optimized tree on the GPU, eliminating round-trips to the CPU, accelerating the overall construction speed by a factor of 5 to 10x.

CR Categories: I.3.2 [Graphics Systems C.2.1, C.2.4, C.3)]: Stand-alone systems—; I.3.7 [Three-Dimensional Graphics and Realism]: Color,shading,shadowing, and texture—Raytracing;

Keywords: ray tracing, real-time, spatial index

1 Introduction

For over 20 years, ray tracing has been considered an offline rendering technology. Historically, its need for a precomputed spatial index over the entire geometry dataset, such as a bounding volume hierarchy (BVH) or a k-d tree [Wald et al. 2007b], has been the highest barrier to overcome in making it usable at interactive rates. This barrier has started to fall down in the last decade, when a new wave of graphics research has started exploring construction of acceleration structures at interactive rates on both serial [Wächter and Keller 2006; Wald et al. 2007a] and parallel architectures [Popov et al. 2006; Shevtsov et al. 2007; Wald 2007; Lauterbach et al. 2009; Zhou et al. 2008]. Finally, recent developments in massively parallel computing allowed tapping into the domain of real-time rendering, showing that the construction of bounding volume hierarchies can be performed in a few milliseconds even in the presence of millions of fully dynamic triangles [Pantaleoni and Luebke 2010].

In this paper we extend the work of Pantaleoni and Luebke [2010] by introducing a novel variant of their Hierarchical Linear Bounding Volume Hierarchies (HLBVH) that is both simpler, faster and easier to generalize. Our first result is replacing their ad-hoc, complex mix of prefix-sums, compaction and partial breadth-first tree traversal primitives used to perform the actual object partitioning step with a single, elegant pipeline based on efficient work-queues. Besides greatly simplifying the original algorithm and offering superior speeds, the new pipeline also removes the need for all the additional temporary storage that was previously required. Our second result is the parallelization of the Surface Area Heuristic (SAH) optimized HLBVH hybrid. This combines the added flexibility of our task-based pipeline with the efficiency of a parallel binning scheme [Wald 2007]. Overall, this allows us to get a speedup factor of up to 10x over state-of-the-art. Beyond this speedup, by parallelizing the entire pipeline, it may all be run on the GPU, thereby eliminating the costly copies between the CPU and GPU memory spaces.

As with the original HLBVH, all our algorithms are implemented in CUDA [Nickolls et al. 2008], relying on freely available efficient sorting primitives [Merrill and Grimshaw 2010]. On an NVIDIA GTX 480 GPU, the resulting system can build a very high quality BVH in under 10.5 ms for a model with over 1.75 million polygons.

2 Background

LBVH and HLBVH: The basis of our work is, as for the previous version of HLBVH, the original idea introduced by Lauterbach et al. [2009], who provided a very novel BVH construction algorithm. The principle is very simple: the 3D extent of the scene is discretized using n bits per dimension, and each point is assigned a linear coordinate along

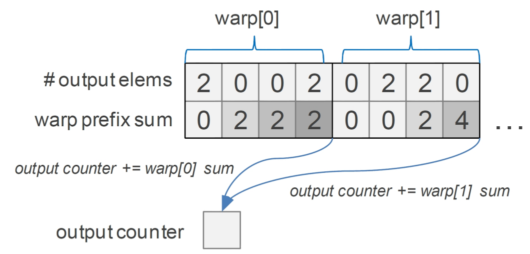
*e-mail:{kgaranzha,jpantaleoni,davemc}@nvidia.com a space-filling Morton curve of order n (which can can be computed by interleaving the binary digits of the discretized coordinates). Primitives are then sorted according to the Morton code of their centroid. Finally, the hierarchy is built by grouping the primitives in clusters with the same $3n$ bit code, then grouping the clusters with the same $3(n-1)$ high order bits, and so on, until a complete tree is built. As the $3m$ high order bits of a Morton code identify the parent voxel in a coarse grid with 2^m divisions per side, this process corresponds to splitting the primitives recursively in the spatial middle, from top to bottom.

HLBVH has improved on the basic algorithm in two ways: first, it provided a faster construction algorithm applying a *compress-sort-decompress* strategy to exploit spatial and temporal coherence in the input mesh. Second, it introduced a high-quality hybrid builder, in which the top of the hierarchy is built using a Surface Area Heuristic (SAH) [Goldsmith and Salmon 1987] sweep builder over the clusters defined by the voxelization at level m.

Ingo Wald [Wald 2010] reports a parallel binned-SAH BVH builder optimized for a prototype many core architecture called MIC. Similarly to what we do, he builds a custom scheduler based on task-queues to essentially implement a very light-weight threading model, avoiding the overheads of the builtin hardware threads support. In order to achieve maximum speed, he spends significant effort quantizing bounding box extents to minimize memory traffic and packing and accessing data in a ad-hoc *Array of Structures of Arrays* format. Moreover, to perform the binning step he uses substantial amount of local storage to keep duplicate bin statistics per-thread and avoid frequent use of atomic instructions on a shared set, requiring an additional parallel merging stage. All this seems to be unnecessary on our target architecture, most probably due to the much larger number of supported concurrent hardware threads and the consequent better latency hiding capabilities. Results are reported only for rather small scenes, and the achieved speed is significantly lower (4 to 5 times) than that offered by our hybrid SAH builder.

Uniform Grids: Kalojanov and Slusallek [2009] introduced real-time algorithms for the construction of uniform grids. While relatively fast, the construction times for this method are higher than those required by HLBVH, most notably due to the fact that spatial partitioning requires object duplication, a data amplification step that translates into higher bandwidth requirements. Moreover, the resulting spatial indices often lead to significantly lower traversal performance during ray tracing, as they don't allow efficient skipping of empty space. Hierarchical variants [Kalojanov et al. 2011] improve both construction times and rendering performance, but they generally remain slower.

3 Algorithm Overview



	warp[0]				warp[1]			
# output elems	2	0	0	2	0	2	2	0
warp prefix sum	0	2	2	2	0	0	2	4

Figure 2: *Efficient queuing with warp-wide reductions. Each thread computes how many elements to insert in the output queue independently, but the actual output slots are computed by a warp-synchronous reduction and a single per-warp atomic to a global counter. Different warps can proceed independently.*

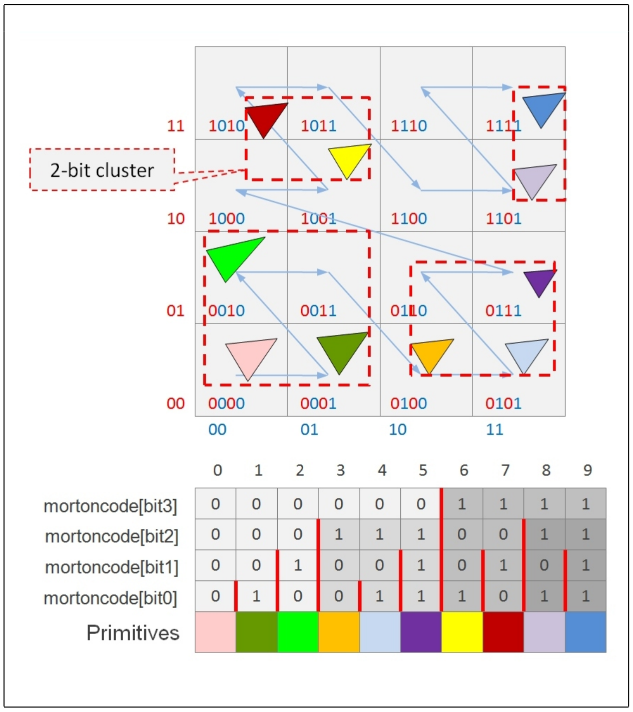
As with the original HLBVH [Pantaleoni and Luebke 2010], our algorithms can be used to create both a standard LBVH and a higher quality SAH hybrid. However, as the standard LBVH involves a subset of the work needed for creating the high quality variant, we will concentrate on the description of the latter. Similarly to the original LBVH [Lauterbach et al. 2009], our algorithm starts by sorting primitives along a 30-bit Morton curve spanning the scene's bounding box. Unlike Pantaleoni and Luebke [2010], who used *compress-sort-decompress* to accelerate sorting, we perform this stage of the algorithm relying on a brute force but more efficient least-significant-digit radix sorting algorithm that was not available at the time [Merrill and Grimshaw 2010]. However, following Pantaleoni and Luebke's observation that Morton codes define a hierarchical grid, where each $3n$ bit code identifies a unique voxel in a regular grid with 2^n entries per side, and where the first $3m$ bits of the code identify the parent voxel in a coarser grid with 2^m subdivisions per side, we then proceed to form coarse clusters of objects falling in each $3m$ bit bin. This is again an instance of a run-length encoding compression algorithm, and can be implemented with a single compaction operation.

After the clusters are identified, we partition all primitives inside each cluster using LBVH-style spatial middle splits, and then create a top-level tree, partitioning the clusters themselves with a binned SAH builder similar in spirit to the one described by Wald et al [2007]. Both the spatial middle split partitioning and the SAH builder rely on an efficient task-queue system to parallelize work over the individual nodes of the output hierarchies. The next sections describe the main parts of this system in detail.

3.1 Task Queues

The original LBVH performed partitioning in a breadth-first manner, creating the nodes of the final hierarchy level by level, starting from the root. As each input node could output either 0 or 2 nodes, each pass performed a prefix sum to compute the output position of the two children. HLBVH improved on this by performing a partial breadth first traversal, where several levels were processed at the same time outputting an entire treelet for each input node. The disadvantage of this approach is that processing each level or set of levels required several separate kernel launches, which is a relatively large latency operation.

We take an entirely different approach, relying on simple task queues, where an individual task corresponds to processing a single node. At run time, each warp (the physical SIMT unit of work on NVIDIA GPUs) continues fetching sets of tasks to process from the input queue, with each set containing one task per thread, using a single global memory atomic add per warp to update the queue head. After each thread in a warp has computed the number of output



	0	1	2	3	4	5	6	7	8	9
mortoncode[bit3]	0	0	0	0	0	0	1	1	1	1
mortoncode[bit2]	0	0	0	1	1	1	0	0	1	1
mortoncode[bit1]	0	0	1	0	0	1	0	1	0	1
mortoncode[bit0]	0	1	0	0	1	1	1	0	1	1
Primitives	■	■	■	■	■	■	■	■	■	■

Figure 3: *Assignment of the morton codes to the primitive centroids (a 2D projection and 4-bit morton codes are shown). Bottom: sorted sequence of the primitives where morton codes are used as keys (for every respective primitive the morton code bits are shown in separate rows; binary search partitions are shown with bold red lines).*

tasks it will generate, all threads in the warp participate in a warp-wide prefix sum to compute the offset of their output tasks relative to the warp's common base. Finally, the first thread in the warp performs a single global memory atomic add to compute the base address in the output queue. By using a separate queue per level, we can do all the processing inside a single kernel call, while at the same time producing a breadth-first tree layout. The process is summarized in Figure 2.

3.2 Middle Split Hierarchy Emission

Once again, we deviate from the previous approaches used to perform middle splits in LBVH and HLBVH, while generating exactly the same primitive partitioning. We recall that each node corresponded to a consecutive range of primitives sorted by their Morton codes, and that splitting a node required finding the first element in the range whose code differed from the preceding element.

Previously, this was achieved in a backwards fashion: rather than processing a single node per thread, with each thread looking for such a split point, each thread processed a single Morton code and tried to detect whether it differed from its predecessor.

If it did, it then elected the point as a split plane for its parent node. Unfortunately, locating the parent node first required precomputing a mapping between the primitives and the containing ranges, which in turn required another prefix sum.

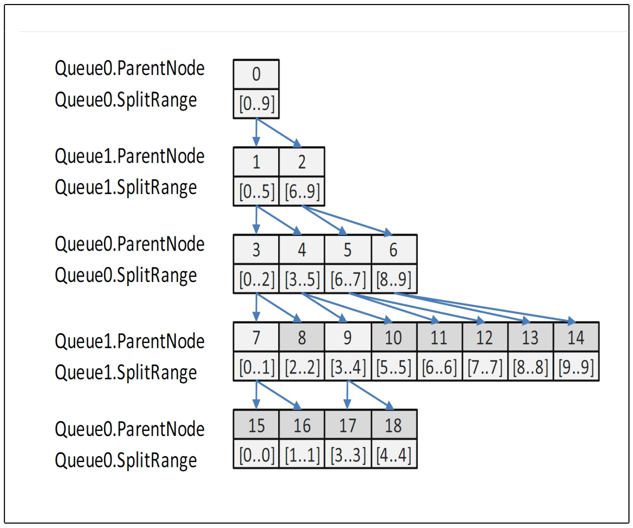


Figure 4: *The middle-split queues corresponding to Figure 3.*

We avoid this complex machinery by reverting to the standard ordering that would be used on a serial device: we map each node to a single thread, and let each thread find its own split plane. However, instead of simply looping through the entire range of primitives in the node, we observe that it is possible to reformulate the problem as a simple binary search. In fact, if the node is located at level l, we know that the Morton codes of its primitives will have the exact same set of high $l - 1$ bits. If we find the first bit $p \geq l$ by which the first and the last Morton code in the node's range differ, it is then sufficient to perform a binary search to locate the first Morton code that contains a 1 at bit p. The process is illustrated in Figure 3 and Figure 4.

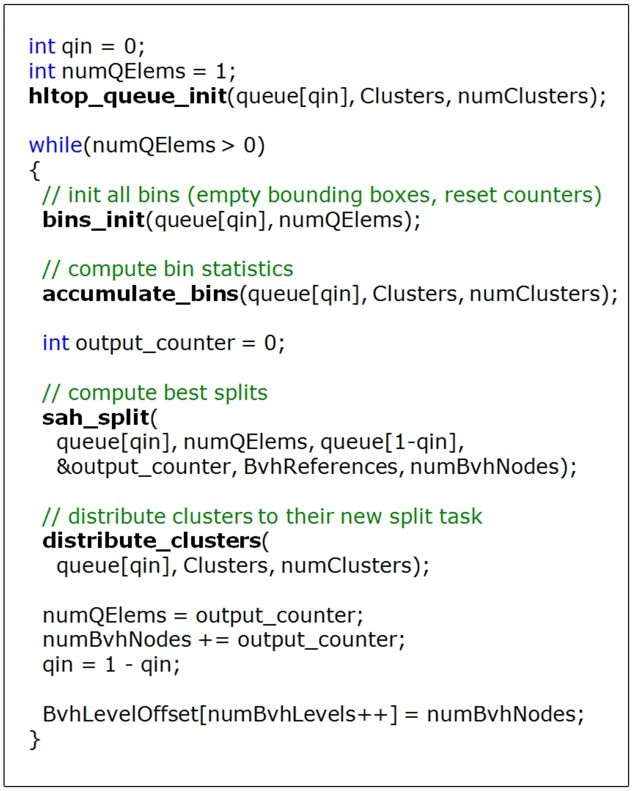
The reason this process is particularly efficient is that, for a node containing N primitives, it finds the split plane by touching only $O(log_2(N))$ memory cells. The original approach touched and processed the entire set of N Morton codes.

Large Leaf Refinement: Middle splits could sometimes fail, leading to occasional large leaves. When such a failure is detected, it's easy to split these leaves by the object-median.

Bounding Box Computation: After the topology of the BVH has been computed, we run a simple bottom-up refitting procedure to compute the bounding boxes of each node in the tree. The process is made particularly simple by the fact that our BVH is stored in breadth-first order. We use one kernel launch per tree level and one thread per node in the level.

3.3 Parallel Binned SAH Builder

As previously discussed, we run a SAH-optimized tree construction algorithm over the coarse clusters defined by the first $3m$ bits of our Morton curve, with m typically between 5 and 7. Before proceeding to the detailed description of the algorithm, we note that our algorithms run in a bounded memory footprint: if we process N_c clusters, we need to preallocate space only for $2N_c - 1$ nodes.



```
int qin = 0;
int numQElems = 1;
hltop_queue_init(queue[qin], Clusters, numClusters);

while(numQElems > 0)
{
  // init all bins (empty bounding boxes, reset counters)
  bins_init(queue[qin], numQElems);

  // compute bin statistics
  accumulate_bins(queue[qin], Clusters, numClusters);

  int output_counter = 0;

  // compute best splits
  sah_split(
    queue[qin], numQElems, queue[1-qin],
    &output_counter, BvhReferences, numBvhNodes);

  // distribute clusters to their new split task
  distribute_clusters(
    queue[qin], Clusters, numClusters);

  numQElems = output_counter;
  numBvhNodes += output_counter;
  qin = 1 - qin;

  BvhLevelOffset[numBvhLevels++] = numBvhNodes;
}
```

Figure 5: *Pseudo-code for the SAH binning procedure.*

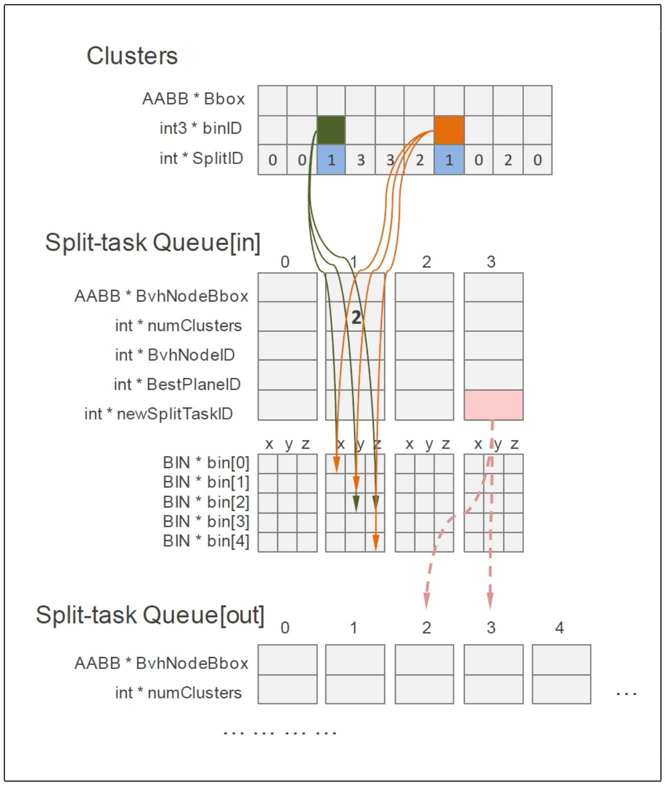


Figure 6: *Data-flow visualization of the SAH binning procedure. First, clusters (in orange and green) contribute to forming the bin statistics of their parent node. Finally, nodes in the input task queue are split, generating 2 entries into the output queue (in pink).*

Pseudo-code for the whole process is given in Figure 5.

In this pass we treat a cluster from the prior pass, with its aggregate bounding box, as a primitive. Once again, we split the computation into *split-tasks* organized in a single input queue and a single output queue. Each task corresponds to a node that needs to be split, and is described by three input fields, the node's bounding box, the number of clusters inside the node, and the node id. Two more fields are computed on-the-fly, the best split plane and the id of the first child split-task. As shown in Figure 6 we store these in structure-of-arrays (SOA) format, keeping five separate arrays indexed by task id. Additionally, we keep an array *cluster_split_id* that maps each cluster to the current node (i.e. split-task) it belongs to, which we update with every splitting operation.

The loop starts by assigning all clusters to the root node, forming split-task 0. Then, for each loop iteration, we perform the following 3 steps:

binning: a step analogous to that described by Wald et al [2007a], where each node's bounding box is split into M, typically eight, slab-shaped bins in each dimension. A bin stores an initially empty bounding box and a count. We accumulate each cluster's bounding box into the bin containing its centroid, and atomically increment the count of the number of clusters falling within the bin. This procedure is executed in parallel across the clusters: each thread looks at a single cluster and accumulates its bounding box into the corresponding bin within the corresponding split-task, using atomic min/max to grow the bins' bounding boxes.

SAH evaluation: for each split-task in the input queue, we evaluate the surface area metric for all the split planes in each dimension between the uniformly distributed bins and select the best one. If the split-task contains a single cluster, we stop the subdivision; otherwise, we create two output split-tasks, with bounding boxes corresponding to the left and right subspaces determined by the SAH split.

cluster distribution: the mapping between clusters and split-tasks is updated, mapping each cluster to one of the two output split-tasks generated by its previous owner. In order to determine the new split-task id, it is sufficient to compare the i-th cluster's bin id to the value stored in the *best split* field of the corresponding split-task:

```
int old_id   = cluster_split_id[i];
int bin_id   = cluster_bin_id[i];
int split_id = queue[in].best_split[ old_id ];
int new_id   = queue[in].new_task[ old_id ];
cluster_split_id[i] =
    new_id + (bin_id < split_id ? 0 : 1);
```

In conclusion, note that there is some flexibility in the order of the algorithm phases. For example, refitting can be performed separately for bottom-level and top-level phases to trade off cluster bounding box precision against parallelism.

Scene	# of Triangles	# of 15-bit Clusters	BVH Memory		Build Time	
			final	temp	new	old
`Fairy`	174k	2.4k	4 MB	33 MB	4.8 ms	23 ms
`Conference`	282k	2.5k	6.5 MB	36 MB	6.2 ms	45 ms
`Stanford Dragon`	871k	2.1k	20 MB	51 MB	8.1 ms	81 ms
`Turbine Blade`	1.76M	2.3k	42 MB	75 MB	10.5 ms	137 ms
`Power Plant`	12.7M	2.0k	290 MB	367 MB	62.0 ms	-

Table 1: *Build time and memory consumption statistics for various scenes. The Build Time column reports the building times for both our new binned SAH HLBVH implementation, and the one from Pantaleoni and Luebke. The Power Plant could not be built with the previous version of HLBVH due to its higher memory usage.*

Scene	HLBVH
`Morton code setup`	0.19 ms
`radix sort`	1.64 ms
`top level SAH build`	2.17 ms
`middle split emission`	3.1 ms
`AABB refitting`	1.0 ms

Table 2: *Timing breakdown for the Stanford Dragon.*

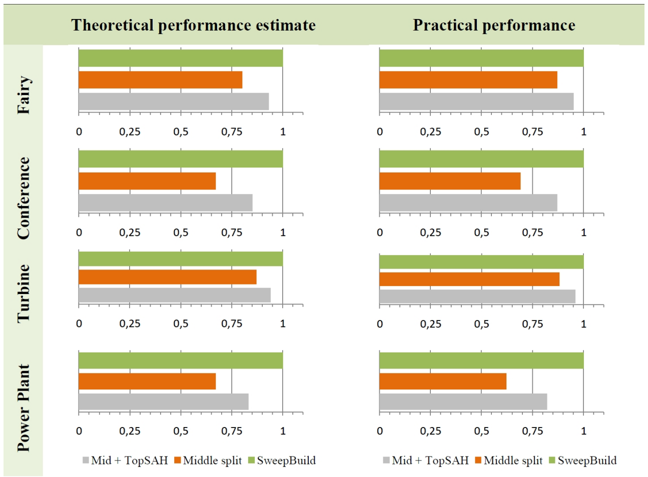


Figure 7: *Predicted and measured traversal performance of middle-split HLBVH, our binned SAH-based HLBVH and a classical SAH sweep builder. The performance measurements were done in the context of 3-bounce path tracing. On the left column, we report 1/SAH cost, on the right, actual traversal speed. All the values in the chart are normalized to 1 relative to the SAH sweep builder. Smaller values are better.*

4 Results

We have implemented all our algorithms using CUDA. The resulting source code will be freely available at *http://code.google.com/p/hlbvh/*.

We have run our algorithms on a variety of typical ray tracing scenes with various complexity: the Fairy Forest, the Conference, the Stanford Dragon, the Turbine Blade and the Power Plant (Figure 1). Our benchmark system uses a GeForce 480 GTX GPU with 1.5GB of GPU memory, and a 3GHz 4-core AMD Phenom with 16 GB DDR3 memory, running CUDA 3.2 and Windows XP-64.

In Table 1 we report absolute build times for our new implementations of the SAH-based HLBVH. In all cases, our trees used 4 primitives per leaf. Table 2 provides a more detailed breakdown of the timings of the individual components of our builders on one scene.

Besides measuring build performance, we measured the quality of the trees produced by both our algorithms. Table 7 shows both the traversal performance predicted using the SAH cost metric [Goldsmith and Salmon 1987], and the measured performance in the context of a 3-bounce GPU path tracer.

5 Summary and Discussion

First, we have presented a novel implementation of HLBVH based on generic task queues, demonstrating how this flexible paradigm of work dispatching can be used to build simple and fast parallel algorithms. Second, we used the same mechanism to implement a massively parallel binned SAH builder for the high quality HLBVH variant. Since this is implemented on the GPU the synchronization and memory copies between CPU and GPU are eliminated. When considering the elimination of these overheads the resulting builder is 5-10 times faster than the previous state-of-the-art. When considering just the kernel times alone it is up to 3 times faster. The new implementation can produce high quality bounding volume hierarchies in real-time even for moderately complex models.

The new algorithms are faster than the original HLBVH implementation despite the fact that they don't exploit any spatial or temporal coherence in the input meshes. This is possible thanks to the general simplification offered by the adoption of work queues, which allows significantly reducing the number of high latency kernel launches and reducing data transformation passes.

5.1 Future Work

As part of future work, we plan to integrate specialized builders for clusters of fine intricate geometry, such as hair, fur and foliage. Further on, this work can be easily integrated with triangle splitting strategies such as those described by Ernst and Greiner [2007]. Another interesting avenue is to try re-incorporating some of the original compress-sort-decompress techniques to exploit coherence internal to the mesh.

5.2 Aknowledgements

We thank the University of Utah for the Fairy scene, Anat Grynberg and Greg Ward for the Conference scene and the University of North Carolina for the PowerPlant model. We are grateful to the anonymous reviewers for their valuable comments.

References

Ernst, M., and Greiner, G. 2007. Early split clipping for bounding volume hierarchies. *Symposium on Interactive Ray Tracing 0*, 73–78.

Goldsmith, J., and Salmon, J. 1987. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7, 5*, 14–20.

Kalojanov, J., and Slusallek, P. 2009. A parallel algorithm for construction of uniform grids. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, HPG '09, 23–28.

Kalojanov, J., Billeter, M., and Slusallek, P. 2011. Two-level grids for ray tracing on GPUs. *Computer Graphics Forum* (4).

Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. 2009. Fast bvh construction on GPUs. *Comput. Graph. Forum 28*, 2, 375–384.

Merrill, D., and Grimshaw, A. 2010. Revisiting sorting for GPGPU stream architectures. Tech. Rep. CS2010-03, Department of Computer Science, University of Virginia, February.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. 2008. Scalable parallel programming with cuda. *ACM Queue 6*, 2, 40–53.

Pantaleoni, J., and Luebke, D. 2010. HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *High-Performance Graphics 2010, ACM Siggraph / Eurographics Symposium Proceedings*, Eurographics, 87–95.

Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. 2006. Experiences with streaming construction of SAH KD-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, IEEE Computer Society, 89–94.

Shevtsov, M., Soupikov, A., and Kapustin, E. 2007. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum 26, 3*, 395–404.

Wald, I., Boulos, S., and Shirley, P. 2007. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1, 485–493.

Wald, I., Mark, W. R., Günther, J., Boulos, S., Ize, T., Hunt, W., Parker, S. G., and Shirley, P. 2007. State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports*, Eurographics.

Wald, I. 2007. On fast Construction of SAH based Bounding Volume Hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*, Eurographics.

Wald, I. 2010. Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture. *IEEE Transactions on Visualization and Computer Graphics*. (to appear).

Wächter, C., and Keller, A. 2006. Instant ray tracing: The bounding interval hierarchy. In *In Rendering Techniques 2006 - Proceedings of the 17th Eurographics Symposium on Rendering*, Eurographics, 139–149.

Zhou, K., Hou, Q., Wang, R., and Guo, B. 2008. Realtime kd-tree construction on graphics hardware. *ACM Trans. Graph. 27*, 5, 1–11.