Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture

Ingo Wald*

Abstract— We investigate how to efficiently build bounding volume hierarchies (BVHs) with surface area heuristic (SAH) on the Intel Many Integrated Core (MIC) Architecture. To achieve maximum performance, we use four key concepts: progressive 10-bit quantization to reduce cache footprint with negligible loss in BVH quality; an AoSoA data layout that allows efficient streaming and SIMD processing; high-performance SIMD kernels for binning and partitioning; and a parallelization framework with several build-specific optimizations. The resulting system is more than an order of magnitude faster than today's high-end GPU builders for comparable BVHs; it is usually faster even than spatial median builders; it can build SAH BVHs almost as fast as existing GPUs and CPUs- and CPU-based approaches can build regular grids; and in aggregate "build+render" performance is significantly faster than the best published numbers for either of these systems, be it CPU or GPU, BVH, kd-tree, or grid.

Index Terms— Bounding Volume Hierarchies (BVHs), Parallel BVH Construction, Surface Area Heuristic (SAH), Intel MIC Architecture

1 Introduction

With both CPUs and GPUs becoming ever more powerful, achieving interactive ray tracing is now easily possible on a variety of consumer hardware platforms. While the first interactive ray tracers used mostly CPUs, today most researchers prefer high-throughput architectures like GPUs and Intel's Aubrey Isle/Knights Ferry [1], whose significantly higher compute power (as well as features like hardware texturing and direct access to a frame buffer) let them achieve significantly higher performance and/or better image quality.

However, while such high-throughput architectures excel at easy-to-parallelize and compute-intensive tasks like tracing rays and shading hit points, they have a much harder time competing with CPUs when it comes to control-intensive tasks like *building* ray tracing data structures. Rendering easily maps to many threads and wide SIMD units, but using those features in building is significantly harder.

For regular data structures like grids, GPU-based algorithms now achieve build times that are just as fast as those for multi-core CPUs—and at higher render times, this is a clear win over traditional CPUs. For more complex data structures like bounding volume hierarchies with surface area heuristic (SAH BVHs), however, CPUs still have an edge: while Lauterbach have recently demonstrated that building SAH BVHs on modern GPUs is indeed feasible [2], their build times are still significantly higher than those for CPUs [3], even on a nominally much more powerful architecture.

With high-performance algorithms becoming ever more hardware-dependent, fully understanding how algorithms and hardware interact becomes ever more important [4]. In this paper, we are going to investigate how to achieve high SAH BVH construction performance on the Intel Many Integrated Core (MIC) architecture [1, 5]. The MIC architecture is a multi-core x86 architecture that combines many of the advantages of GPUs and CPUs: it offers the same raw compute power as high-end GPUs (and, of course, hardware texturing etc), while still featuring CPU-like features like real caches, a consistent memory model, global atomics, etc. However, MIC is not simply a faster CPU: being optimized for throughput workloads it relies on the same features that make building hard on GPUs—wide SIMD, a low cache-per-compute ratio, lots of cores, and multiple threads per core—and achieving high build performance requires a carefully designed framework that takes those hardware-specifics into account. In this paper we will not introduce new high-level algorithms, but rather concentrate on existing algorithms, and investigate how to best map these to the underlying hardware by concentrating on three major points: efficient use of the cache/memory subsystem, high-performance SIMD kernels, and efficient multi-core parallelism/synchronization.

2 Related Work

Ray tracing acceleration structures generally fall into three categories: grids, kd-trees, and bounding volume hierarchies (BVHs). Various hybrids and variants of BVHs are still being actively researched (eg, [6, 7, 8]); however, for the purpose of this paper we only consider traditional binary BVHs.

*I.Wald is with Intel Labs, Intel Corp, Ingo.Wald@intel.com

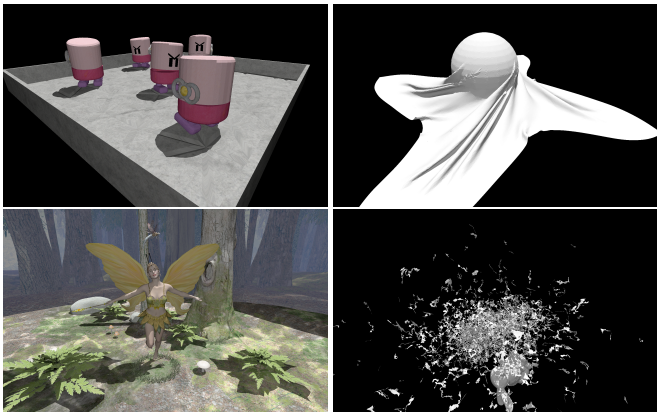


Fig. 1. Several sample frames from our system running on a prototype "Knights Ferry" board: Toasters (11k triangles), cloth (92k), Fairy Forest (174k), and UNC dragon/bunny animation (252k). All scenes are rebuilt from scratch every frame, using a full SAH BVH binning stage (with highest quality settings) for each BVH node. Normalized to a hypothetical 32-core 1GHz configuration these scenes cost 1.3ms, 12.5ms, 22.5ms, and 34.4ms, respectively, to build, and at 1920×1200 pixels render at 92, 44, 17, and 19 frames per second, respectively, including animation, rebuild, shading, texturing, transparency textures, and shadows).

Surface Area Heuristic. For kd-trees and BVHs, the best known method to build high-quality trees is the Surface Area Heuristic (SAH) as introduced by Goldsmith and Salmon [9]. Though originally proposed for iterative builds, best results are achieved using greedy top-down builds [10, 11]. In each recursion step, one considers different ways of splitting the triangles T into two groups L and R, and estimates each split's cost via

$$SAH(T \rightarrow (L,R)) = K_T + K_I\left(\frac{SA(L)}{SA(T)}N_L + \frac{SA(R)}{SA(T)}N_R\right), \qquad (1)$$

where $SA(X)$ is the surface area of the axis-aligned bounding box of X, N_X is the number of triangles in X, and K_I, K_T are implementation specific-constants to model intersection and traversal cost, respectively (in our case, $K_I = 1, K_T = .5$).

Fast SAH builds. To speed up SAH BVH build times, different authors have proposed *binning* techniques in which the SAH is evaluated only at some sample positions [12, 13]. These techniques have originally been proposed for kd-trees, but work even better for BVHs, where small numbers of bins produce BVHs nearly indistinguishable from sweep-based builds [3].

Given additional information, significantly faster builds can be achieved using from-hierarchy builds [14, 15] or refitting [16]. Though such techniques should be used where applicable, in this paper we will only consider fully general "from scratch" rebuilds.

Parallel SAH Builds. In [3], Wald proposed a framework that built SAH BVHs in parallel on a dual-Clovertown PC (8 cores total): Building is described as a sequence of build tasks that recursively spawn new tasks; tasks larger than a certain threshold are processed by all threads in parallel, those below the threshold are queued up for a later stage in which different threads build different sub-trees. Wald also proposed a variant in which triangles were initially binned into a regular grid that was then used to construct the top few levels of the BVH. This approach leads to spatial splits for the first few levels of the BVH, and thus trades BVH quality for build time.

Shevtsov et al. [17] proposed a similar framework for kd-trees: Initial parallel partitioning is done through object median splits until the number of parallel jobs equaled the number of cores; those are then built on different cores using SAH splits. Zhou et al [18] used a similar approach to realize the first GPU-based kd-tree builder; however, switching to SAH splits happens only for very small subtrees, and the majority of nodes is built using spatial median splits. An improved version of this approach—that also works for BVHs and that can handle significantly larger scenes—has recently been published in [19]. In [20], Choi et al. present two related approaches for parallelizing an SAH kd-tree build: Choi et al. also differentiate two distinct builds phases—breadth-first geometry-parallel and depth-first per-subtree parallel—but manage to parallelize both phases *without* having to rely on binning. Using a quad-core Xeon X7550 "Beckton" system (32 cores total) Choi et al achieve interactive rebuilds for a variety of scenes.

The first full SAH BVH builder[1] for GPUs was realized by Lauterbach et al. [2]. Lauterbach et al. also differentiate between large and small tasks: small tasks are built by a single warp each, and large tasks , tasks are processed by multiple warps in parallel; new tasks are queued up in work queues. To effectively use the GTX280's 32-wide SIMD , Lauterbach performed binning with 32 bins, using SIMD to update all bins in parallel. Lauterbach also proposed a much faster spatial-median build strategy—called LBVH—based on Z-order Morton Codes and fast GPU-based sorting algorithm, as well as a hybrid algorithm that used spatial median splits close to the root, and SAH splits deeper in the tree. Pantaleoni et al. [15] demonstrated a from-hierarchy variant of this algorithm that exploits spatial coherence already assumed to be present in the input meshes (possibly from a previous build step), thereby achieving significantly higher build times than those of Lauterbach et al. Pantaleoni also proposed a different way of using SAH information in his HLBVH build but unfortunately did not provide a "full" SAH variant of his algorithm.

Grids. With BVHs and kd-trees being costly to build, multiple authors have proposed to instead use cheaper-to-build grids. In [21], Reinhard demonstrated dynamic updates, but did not rebuild from scratch. Wald et al. [22] proposed a CPU-based approach that allowed interactive rebuilds, but used only reduced-resolution grids. Ize et al. [23] achieved highly-interactive rebuilds by parallelizing with a sort-middle approach. Finally, a GPU-based grid builder was introduced by Kalojanov et al. [24].

3 Algorithm Overview

At the most abstract level, our system uses the same ideas as described in [3] and [2], except that we never resort to spatial median splits: BVH construction is implemented by recursively partitioning a given set of primitives. Each such partition first bins all triangles into a set of bins (in our case, 16), then determining the bin with lowest SAH cost, and either produces a leaf node, or partitions the primitives into two halves, which are then processed recursively.

For parallel execution, this is implemented through a series of *tasks*. Each such build task contains a binning stage in which one or more threads bin the triangles. Upon completing the binning one then merges the different threads' binning information and evaluates the SAH. Based on this SAH evaluation, one then either creates a leaf, or issues a new *partitioning task* that performs the actual partitioning of the primitives and finally triggers two new tasks for the resulting two sub-trees. Tasks are usually processed by multiple threads until their size falls below a given threshold, at which time they are processed by specialized single-threaded code. On this high level our approach is exactly the same as what previous authors have done; the main difference is in how exactly those stages are mapped to the hardware.

[1] Under a "full" SAH builder we understand a build algorithm that uses an SAH cost function (possibly based on binning) for each node of the BVH.

4 Data Organization

The first thing we are going to concentrate on is how to efficiently use the memory and caching subsystem. At first glance, a (hypothetical) 32-core chip based on the MIC architecture would have a total of 8MBs of cache. This at first sounds ample, but isn't any more when considering this relative to functional units: For example, the Clovertown CPUs used in [3] have four 4-wide SIMD units working on the same physical cache size. Compared to 128 threads of 16-wide SIMD each this leaves 4x4=16 vs 128x16=2k active elements sharing the same cache—a difference of two orders of magnitude.

Like previous authors [3, 2] we compute an initial `(bounding box, primitive ID)` pair (called a *fragment*) for each primitive, and perform all build steps by re-ordering the fragment array. This ensures that irregular memory accesses to indexed vertices happen only in this initial stage, and that memory indirections are not required during building. In-place re-ordering would be preferable for maximum cache usage, but partitioning in parallel is easier if source and target regions don't overlap. We therefore use two fragment arrays—one to read from and one to write to—and alternate between them. Different sub-trees work on mutually exclusive fragment regions, so some threads can work from A to B while others work from B to A.

4.1 Data Reduction through Hierarchical Quantization

In addition to these techniques—which are identical to what other authors did—we minimize our fragments' cache footprint by quantizing all box coordinate in DX10 `unorm10` data format, which the MIC instruction set supports natively [25], and which allows to squeeze all 6 box coordinates into two dwords. Unorms have to be in the $[0..1]$ range, so we express all fragments relative to the scene bounding box. All our kernels operate exclusively on those local unit coordinates; the only time we transform to world space is when writing BVH nodes. Since this transformation also changes the surface areas of the bins, the SAH evaluation also has to correct for this distortion by properly scaling all bins' sides before evaluation.

Quantization to 10 bits implies a loss of accuracy from 24 to 10 bits. To avoid quantization artifacts during rendering, we *only* quantize the fragments, and never quantize the vertices used for intersection and rendering. To ensure that the quantized boxes (after rounding) still fully enclose the unquantized vertices, we also have to extend the box by half a bit's worth of mantissa before quantizing it. This slightly increases all fragments, which we will deal with later.

Re-quantization. At only 10 bits per dimension, the smallest region of space a BVH node can encode is the same as that of a regular grid of 1024^3 cells. This negatively affects the build quality in two different ways: first, it means that all BVH nodes have to be slightly enlarged to the next discrete value, increasing the SAH cost. Second, tiny triangles lying in the same cell will all end up having the same bounding box, in which case the builder cannot separate them.

This problem can be fixed by infrequently *re-quantizing* the fragments (relative to the current sub-tree's bounding box) whenever that sub-tree's bounding box becomes too small relative to the current quantization domain. Determining when to best perform re-quantization requires some form of heuristic. Ideally, this heuristic would perform re-quantization every time the number of relevant bits available for the current dimension drops below a given threshold. While using this ideal heuristic is certainly possible, for our system we have adopted an even simpler heuristic in re-quantizing only once—every time time

the number of primitives in a sub-tree falls below a certain threshold, in which case the system switches to a specialized "Local Build Job" code path, anyway (see Section 6.4). Performing quantization at that time made for a simpler system design, and at least for the scenes we tested has proven to be perfectly sufficient (Section 6.4). Also note that quantization *only* applies to the number of bits used for *binning*—not to the accuracy of bounding boxes coordinates—since those are afterwards re-fit at full floating point precision (Section 6.4).

4.2 AoSoA Data Layout

There are two choices for the data layout: array of structures (AoS), and structure of arrays (SoA). Unfortunately, both are problematic for our purposes: AoS is often considered the more natural, but since different fragments' same-type components (ie, their x values) are stored non-contiguously they do not lend well to SIMD processing.

SIMD architectures prefer SoA, in which same-type components are stored contiguously. In SoA, however, we need three different pointers for each fragment (without quantization we'd need 7!), and the partitioning kernel—which reads from one location and writes to two others—would have to operate on 9 different addresses in each iteration (plus whatever other pages that thread needs): That many "active" addresses require lots of registers and scalar operations to maintain them, and lead to cache pollution. Worse yet, for non-trivial data sets these addresses will end up in different virtual memory pages, and at order 10 pages per thread and four threads per core the TLB [26] will start to trash, requiring many costly TLB walks.

To avoid both data layouts' problems, we instead use a *structure-of-array-of-structures* (AoSoA, or ASA) data layout (Figure 2): we group all fragments into *chunks* of 16 fragments each, store each chunk in SoA layout, and the entire fragment array as an array of such SoA chunks (see Figure 2). Though awkward for scalar processing (each particular fragments' address depends on which chunk it is in) for SIMD processing it combines the best of both worlds: Like AoS, data is local, and can be addressed by a single pointer; like SoA, each chunk is in SIMD-friendly layout.

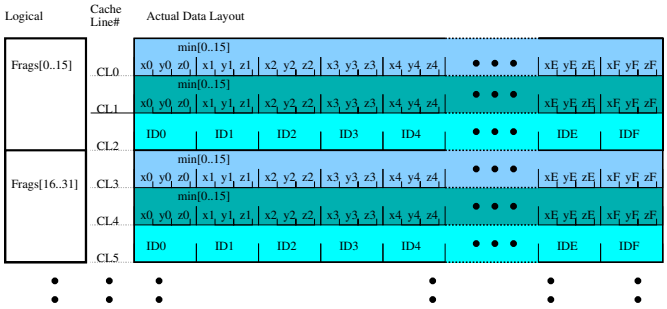


Fig. 2. Our AoSoA data layout with 10 bit quantization per coordinate. Each *chunk* of 16 fragments requires only three consecutive cachelines, and allows very efficient SIMD processing. Each [begin..end) region of chunks is contiguous, and can be addressed with a single scalar register.

5 Low-level Kernels

Before discussing the parallelization framework we first describe the low-level kernels that the build tasks call back upon. Though different threads may have to merge their respective kernels' outputs to complete a given task, the kernels themselves know nothing about parallel processing or synchronization, and only work on stack-local data.

5.1 Binning Kernel

Binning is the most compute-intensive kernel in our system. Like Lauterbach et al. [2], we utilize SIMD by updating all 16 bins in parallel. From an algorithmic standpoint, this is a bad idea: each primitive falls into only one bin, and updating all 16 bins—while achieving high SIMD utilization—effectively "wastes" work by updating more bins than required. Instead, it would be far more efficient to have each primitive update *only* "its" bin, and perform a min/max prefix/postfix operation across all 16 bins only once before SAH evaluation [3]. However, while this approach *is* more efficient on a scalar or 4-wide SIMD architecture, it does not map well to 16-(or more-)wide SIMD: for one element and one bin, there simply is not enough work to do in parallel; and trying to increase the amount of parallel work by operating on either the three different dimensions or 16 different fragments did not work out, either: operating on three dimensions could in theory occupy up to 9 (out of 16) vector elements, but has a very high overhead that made it very slow. Operating on 16 different fragments is more natural, but suffers from the fact that multiple of these fragments might project to the same bins, and the overhead to detect *and* resolve these conflicts has turned out to be far too high.

Consequently, we follow Lauterbach in having each fragment update all 16 bins in parallel. Though algorithmically sub-optimal, this approach leads to a very natural SoA layout for the 16 bins that induces a very efficient implementation: Assuming one dimension's bins are already loaded into registers, all the binning kernel has to do is SIMD load-and-broadcast a fragment's 6 coordinates, perform a SIMD comparison to get a mask, and do 12 masked SIMD min/max'es to update the 16 bins' left and right bounds (see [25] for how more information on masking and broadcasts).

The problem with this performance argument is that it assumes that all 16 bins are already in registers, which is not automatically the case. Each bin stores both bounding boxes (`lBounds,rBounds`) and primitive counts (`lCount,rCount`) for all primitives left resp right of that bin's partitioning plane. Even knowing that `rCount=N-lCount` this requires 13 values per bin, which for 16 boxes requires 13 (out of 32 [5]) vector registers for those bins alone—and three times as much for all three dimensions together. In a naïve implementation where each fragment updates all three dimensions' bins in turn the resulting kernel would have to spill through 39+ SIMD registers (3KB of data!) for *each* iteration (which might, in fact, happen in [2]) This spilling not only triples the instruction count (*each* min or max op becomes load-op-store), it also leads to instruction dependency chains and more I/O requests than the L1 can reasonably fulfill (almost two out of three cycles would demand either a load or a store).

To avoid this issue, we again exploit our AoSoA data layout: rather than iterating over fragments, we iterate over chunks; for each such chunk we first iterate over all 16 fragments in X dimension before doing them all again in Y and Z. This way, we can always load one set of bins into registers, re-use it 16 times, and only then switch to another dimension's bins. In addition, we can further reduce memory I/O by always loading multiple fragments into the same register, and using the instruction set's free broadcasts to then broadcast each such fragment to all 16 bins when required. Prefetching, too, is simple: Thanks to our data layout we only need two cache lines per chunk, and since each chunk performs plenty of work before needing its neighbor's cache lines there is ample time to prefetch those two lines. Taken together, the binning kernel works almost completely out of registers, and basically without any dependency-, cache miss-, TLB-, or any other kind of stalls. After loop unrolling, the code is almost devoid of load/stores and scalar ops, with the few remaining ones often "paired away" (ie, executed for free in the v-pipe while the u-pipe is doing more useful work (see [26] for a discussion of pairing).

Since the current range of fragments does not necessarily have to start or end at chunk boundaries, always operating on complete chunks means that compute a validity mask for the first and last chunk, and only bin elements whose mask bit is sent.

5.2 SIMD SAH Evaluation

Processing 16 bins in SoA layout also allows for efficiently evaluating all 16 bins' SAH in parallel. Since binning is performed in the local space used for quantization we must first correct for the resulting distortion; but, this only requires to properly scale all bounding box sides when computing their surface area. For all three dimensions, we then compute the 16 bin's SAHs in parallel, and compute the dimension and bin for which these values reach their minimum.

In addition, we can use a simple trick to compute the SAH termination criterion (almost) for free: Since the leftmost bin by definition has all primitives either in or right of it this bin's SA_r and N_r values corresponds exactly to the parent's area and primitive count, and since this bin can never produce a valid partition, anyway (lCount=0 is not a valid partition), all we have to do to compute the "no-split" cost is to mask our this bin when adding the K_T and SA_lN_L terms. Then, if the final horizontal reduction of the SAH cost indicates bin 0 as cheapest bin we know that this node should be made a leaf.

5.3 Partitioning Kernel

The partitioning kernel's job is to take a range of fragments in the input array, to compare each such fragment to the just computed partition plane, and to write those fragments to either of two regions (indicated by `lDest` and `rDest`) in the output array. As done for the binning kernel, we always operate on entire chunks. For each such chunk, we read first read the 16 fragments and perform a (free) unorm-to-float upconvert during the read. We then compute the respective 16 centroids, and use a vector compare with the partitioning plane to compute a mask indicating which of those fragments go to the left resp. right side.

When appending to the left respectively right "stream" of fragments our AoSoA layout requires some additional work, because the (compacted) fragments we are writing may start and end inside different chunks (and thus, in two different, non-neighboring cache lines). Fortunately, however, in our instruction set the vector compaction comes in two separate instructions—`packlo` for those elements that would go to the current cacheline and `packhi` for those for the next, respectively [25]—which is exactly what we need: For each left respectively right stream, we store a pointer to the current chunk and a 0–15 offset inside this chunk where the stream ends. Then, when appending a new chunk of fragments we first use `bitcount` to determine the number of fragments after compaction. If those fit into the current chunk, we perform three `packlo`s, and are done. If not, we first write the three lo-parts, then advance the pointer to the next chunk, and write the "overflows" using `packhi` (and, of course, prefetch the next chunk). Note that though we need to upconvert the fragments for computing the (float) centroids, for the actual compaction and writing we can directly write the three vector registers' worth of quantized data.

In addition to moving the fragments, the partitioning kernel also tracks the bounding boxes of the left respectively right centroids which we need for the next recursion step's binning. Since the partitioner itself needs only few registers, we can keep those in (12) SIMD registers, and cheaply update them with masked min/max'es based on the left and right mask, respectively. In combination, for each chunk of 16 fragments the resulting code contains only two conditionals (one for the loop, one to decide overflow or not), and otherwise contains almost only vector instructions (min,max,compare,and pack). The few remaining scalar instructions (conditionals, pointer and offset updates, prefetching) can mostly be paired way [26].

6 Parallelization Framework

Having described the low-level kernels, we now describe the parallelization framework that ties these together.

6.1 Tasking System

At the core of our framework is a fully general (collaborative) *tasking system* (in the spirit of CILK [27]) in which all work is described as a series of *tasks* that are inserted into a *task queue* from which worker threads pick them for execution. Tasks can optionally have dependencies, completions, and priorities. Once created, tasks get *scheduled* to the tasking system: tasks without dependencies are appended to the task queue; those with dependencies remain dormant until all their dependencies are fulfilled. Each task has one or more units of work called *jobs*; different jobs of the same task can be executed concurrently by different threads. To facilitate recursive algorithms (like building), tasks can also specify *completions*, which indicate that the task is fully done only if all its jobs are processed *and* its completions are done, too. Modifying the task queue (insertion, removal, and picking) requires locking a global mutex; all other task-specific values (next unclaimed job, number of remaining dependencies, completions, etc) are realized mutex-free via spin-loops. All tasks are reference counted, and automatically get destroyed once no longer needed.

Our system supports two levels of priorities by having two separate task queues; threads always pick high-priority tasks except if only low-priority tasks are available. Once a thread picks a task, it first checks if this task has any unclaimed jobs available, and removes it from the task queue if not; otherwise, it *joins* this task by calling its runfunction (the function that executes that task's code).

To avoid thread switching overhead, we use an *exact subscription* model in which there is a 1:1 correspondence between hardware threads and (affinitized) worker threads; all synchronization primitives are realized through atomics and spinning, making sure that no thread-switches occur except in the rare case that the OS preempts the entire process. Nevertheless, synchronization is costly, in particular when hundreds of threads want to modify the task queue at once.

6.2 Build-specific extensions

Most previous parallel build systems have made use of some sort of load-balancing "work queues" (eg. [2, 3]). Unlike an application-specific work queue system however our system is a fully general tasking system (like CILK) that not only drives the build, but also the rendering, the entire system, and various other tasks, if enabled.

Thanks to this generality there are several sources of overhead that would not be required for our particular application (eg, none of our tasks ever specifies a dependency, and core-local work queues would be much simpler for our application than for the general case). We have opted to keep full functionality even if this carries some overhead. We did, however, add some extensions particularly designed to improve efficiency of recursive build-style tasks.

Thread-local data. To enable threads to store thread-local data (at lower cost than using pthreads for this), we have made the runfunction aware of the executing thread's ID by passing this as a parameter.

Explicit next-task selection. Instead of a "schedule and forget" mechanism for newly generated child tasks, our tasking system allows each thread to, from within a runfunction, explicitly work on a given other task (usually, a child task) before returning to the task scheduler.

Persistent runfunctions. Traditionally, each of a task's sub-jobs would be executed in a different runfunction call, with the scheduler telling the runfunction which of the task's jobs to work on. Instead, our runfunctions get a pointer to the task itself, with which they can then autonomously claim and return as many jobs as they like: This allows the runfunction to amortize initialization/shutdown code (eg, bin initialization and merging) across multiple jobs. In addition, it also enables the runfunction to detect when no more jobs are available for this task, as well as which thread finished the task.

6.3 Standard Build Tasks

At the core of our system is a "standard" build task. To facilitate parallel processing, each task's range of fragments is logically subdivided into *blocks* of 512 fragments. Each of these blocks corresponds to one job; thus, such tasks can be processed by up to $\lceil$`numTriangles/512`$\rceil$ different threads. For each such job, the thread executing it first computes the range of fragments that this job corresponds to, and then performs binning using the binning kernel described in Section 5.1. Each thread accumulates partial results in its own, stack-local set of bins that, when done, it *merges* into the task's shared bin structure.

Deferred and Horizontal Merging of Bins. Before the SAH can be evaluated, the threads must first merge their respective local bins. In our initial implementation, we did this at the end of each job, which turned out to be too costly. Using our persistent runfunctions each

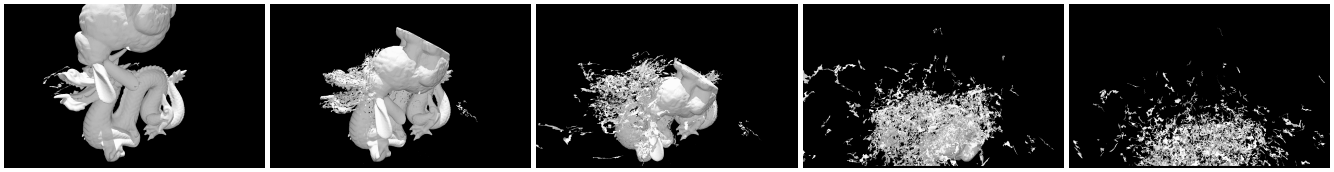


Fig. 3. Five frames from the "Bunny/Dragon" animations (252k triangles). Using a hypothetical 32-core 1GHz Knights Ferry board, we can fully rebuild this scene from scratch (at highest BVH quality settings) in 34ms per frame (43ms in sychronous mode). Including animation, rebuild, and rendering (at 1920×1280 pixels, including shadows) this animations runs at around 13–15 frames per second.

thread can now locally accumulate multiple jobs' bins, and defer all merging until no more jobs are available for this task.

Even then, serializing 128 threads for merging is a bad idea: Apart from having to spin for the shared mutex, merging requires reading and writing the task's shared set of bins—but since those have just been written by another core this operation guarantees 39 successive L2 misses that require fetching those cache lines from other cores, invalidating them on those core, evicting local cache lines to store them, etc. And since no other threads are available to hide the resulting latency (those are spinning on the mutex), this is expensive. In total, one simple merge operation can add up to thousands of cycles; serializing this for 128 threads produced a serious scalability problem.

Avoiding this serialization requires some sort of horizontal merging.

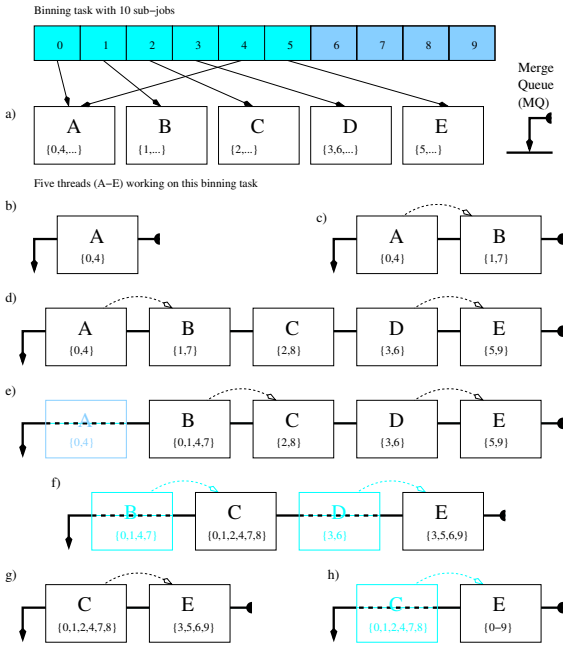


Fig. 4. Horizontal Merging: In this hypothetical but plausible example, five threads (A-E) are collaboratively binning a task consisting of 10 sub-jobs (0-9). a) each of the five threads works on different jobs, and accumulates its jobs' binning info in its own thread-local storage; the merge queue is empty b) thread A (having accumulated jobs 0 and 4) finishes first, enters the merge queue, has nobody to merge with, and waits to be consumed. c) thread B finishes, appends itself to the merge queue, tries to lock both itself and its predecessor (A), then starts merging A's binning data into its own. d) threads C, D, and E concurrently finish binning (while B is still merging A's data) and atomically enter the merge queue. C tries to lock B, but fails because B is busy; D and E try to lock C and D, respectively, producing a race as to who manages first to acquire the D's lock. Assuming E wins the locking race (the opposite case would have been valid, too), E starts merging in D's data while D spins for the C and D locks. e) after B finishes merging A's data it tells A that it is consumed; A then goes on doing other useful work. B is now at the head of the merge queue, releases its lock (thus allowing C to start merging its data), and spins until consumed. f) C and E finish merging B and D, respectively, and tell those to be consumed. g) E acquires C's lock, and starts merging C's data. h) E finishes merging C, tells C it is consumed, and realizes it now has all jobs' binning data, allowing it to proceed to SAH evaluation and creation of the partitioning task.

Our first attempt was to wait for all threads to finish, and then to hierarchically merge those threads' partial results. This however meant that all threads always had to wait for the slowest one to finish binning, which was just as bad.

Instead (also see Figure 4), we currently have each build task keep a linked list of all threads' partial bins: Any time a thread finishes, using an atomic exchange op and its thread-local storage (Section 6.2) it appends its local bins to this list. It then tries to lock both its own and its predecessor's data. This requires atomics, but only among those two threads, not globally. If successful, the thread merges its predecessor's bins into its own, and de-queues this predecessor, at which time the predecessor knows it is done with merging. Thus, multiple tasks finishing binning concurrently can start merging in parallel while other threads are still binning. If some threads take longer to bin than others those finishing early can leave the merging stage—and do something else—as soon as their data has been consumed.

SAH Evaluation and Node Processing. The thread doing the last merge operation for a task (which may or may not be the same that did the last bin) by definition has the final merged bins for the entire task, and can evaluate the SAH.

Leaf Generation. If the SAH's termination criterion suggest that a leaf should be created, the respective node is flagged as a leaf (its bounds are stored by the parent as described below). In our implementation, when making a leaf we also compute, for each triangle in this leaf, a 1-cache line structure used to accelerate ray/triangle intersection during rendering. Though not strictly part of the BVH builder, we nevertheless include this cost in our build cost.

Inner Node Generation and Partition Setup. If the SAH does indicate a valid split, the thread creates an inner node as well as a partition task to partition that task's fragments. To enable the build task to do this in parallel, we have to know in advance where each jobs' fragments have to go in the output array. Following [3], we first, during binning, store each job's bin-counters; then, before creating the partition task we know how many fragments each job will contribute to the left and right side, and accumulating these values yields the respective jobs' *target* offsets in the output array. Once these left and right targets are known for each block, the task computes the child nodes' bounds (by transforming the respective bin's bounds to world space), and creates the partitioning task.

Parallel Partitioning. Knowing where each job's left and right fragments belong in the output arrays, the different jobs can be processed in parallel without synchronization. Binning relative to the *centroid* bounds produces better BVH quality than binning relative to the subtree's fragment bounds [16]. Therefore, in addition to moving the fragments, the binning task also keeps track of the bounding boxes of the *centroids* of the fragments that go to the left resp right sides; these are needed to produce a tight binning domain for the eventual child tasks.

Once done partitioning the different threads have to horizontally merge their respective local centroid bounds (which works similar to merging bins). The thread that did the last merge then triggers recursion by creating two new build tasks for the left respectively right sub-tree.

6.4 "Local" Build-Jobs

Once a sub-tree become small enough it doesn't make sense to create and schedule a full task any more, and it is more efficient to build the

entire subtree recursively on the stack [3, 2]. In our implementation, we do this as soon as a build task would contain only a single job; ie, once the number of triangles drops to 512 or below.

These "local" build tasks are not "tasks" in the tasking system's sense; instead, the respective thread that would have created the task immediately builds the entire sub-tree—in depth-first order and on the stack—without any calls to the tasking system at all. To avoid the `atomic inc` for each node allocation we use a single atomic to pre-allocate the maximum number of nodes that the subtree could generate. Being processed by a single thread the resulting code is much simpler: for example, binning, SAH evaluation and binning can use the same stack-local data without any kind of synchronization, merging, or "glue" between those kernels. In particular, local build-jobs perform no atomics or synchronization at all, and never have to access the tasking system, which particularly important since such local build jobs produce lots of nodes, which would utterly swamp the tasking system if each such node would require a full task creation/scheduling.

Local re-fitting. As described above, "large" build-jobs compute their child nodes' bounds by transforming the respective bins' bounds into world space, but since those bins are built with quantized fragments they are slightly larger than they need to be. For local sub-trees, however, we can easily compute accurate bounds: In the leaves, when computing our triangle accel structure we already read all (unquantized) vertices, and can thus compute accurate bounds; and since local sub-trees are built entirely on the stack, after returning from building those children the thread can also compute accurate bounds for those inner nodes from their children. This not only improves accuracy, it also slightly improves performance because it is more SIMD-friendly than extracting 16 scalar values from an SoA bin structure.

6.5 Initial Quantization

So far, all tasks have operated on already-quantized chunks. Since these do not exist up front, we have one designated *root* task that works just like a standard build task except that it first generates these chunks by gathering every 16 triangles' vertices, computing their bounding box, and quantizing them with respect to the scene bounding box (which, like [2], we assume as given). Each such chunk is then binned just as done in a standard build task.

6.6 Improving Locality and Concurrency

Tasking systems usually operate in FIFO order, and there are good reasons for doing that. For recursive build algorithms, this implies that nodes are built in breadth-first order, which is what most previous systems did [3, 2]. This however has two serious problems. First, it destroys locality: a thread that finishes a build task will create two child tasks that work on the same data, but, when running in FIFO order, will not be able to capitalize on this locality. Even assuming that those child tasks would be executed by the same thread (which is unlikely), when running in FIFO order that thread would have executed other tasks before it even reached those tasks, which—in particular given our relatively small tasks—would have long evicted that data. Even worse, it means that all the threads that finish a job will all want to continue working with the same task—the one at the head of the queue. This in turn means that shared data like `nextJob` counters etc frequently bounces between all cores (creating stalls), that each threads' jobs are always scattered between as many cores as possible (maximizing cross-core communication and merging overhead), that all threads continually fight for the same task queue mutex, and that even if a task went through all the work of acquiring the head task some other threads have since processed all of its job.

To avoid this problem, we exploit our tasking system's *explicit next-task selection* feature that has been added for exactly this reason: the thread that created the child tasks first adds them to the task queue in (l,r) order, but then, rather than returning, first works on those two tasks, and does so in (r,l) order. Since both l and r are in the task queue other threads can still *join* in those tasks, but since they first have to exhaust their own stacks *and* all earlier tasks in the work queue before doing so this technique means that threads will very quickly diverge into different sub-trees. Though divergence is often a bad thing, in this case it is actually good, since most of the time the threads can build entire sub-trees without interference from—synchronization with—other threads. This maximizes cache-reuse, minimizes cross-core cache line sharing, and means that threads access the tasking system only when all "their" tasks are exhausted.

This reordering means that by the time a thread gets a new task from the tasking system there is a good chance that this task is already done by whoever task has originally generated it—but since much fewer threads ever access the tasking system the thread reaching those already-processed tasks usually has ample time to remove them before anybody else can even see them. In combination, for a block size of 512 triangles these techniques translate to a reduction in build time of roughly 2x (with even larger wins for smaller block sizes).

6.7 Block Size and Asynchronous Building

For any parallel system there is a conflict between job size (in our case, block size) and overhead: As a rule of thumb, smaller jobs sizes mean more parallel work, which allows for supplying more threads with work. However, more jobs also mean more accesses to the tasking system, more atomics, more per-job startup/shutdown cost, etc (eg, even at 512 triangles per block each job has only 32 frag-chunks).

Larger job sizes produce significantly fewer overhead, but often lead to threads running out of work, and having to spin idle until the last thread is done with its (large) piece of work. In rasterizers, this load balancing problem is often avoided by asynchronously working on multiple frames at the same time [5]. In our system, we achieve a similar effect by asynchronously overlapping two frames' tasks: we start the current frame's build while the previous frame is still rendering, and schedule all build tasks into the high-priority queue. As soon as a thread finishes a tile, it will—if any build task is available—immediately switch to building; and rather than idling, threads running out of build tasks simply render a tile. Since this "asynchronous" mode reduces idling it produces reduces "incremental" build times (the difference between "build+render" and "render only"), but since it also increases memory consumption this mode is optional.

6.8 Final Considerations

The most striking effect of this framework is that the entire system is (intentionally) rather chaotic, with threads picking whatever work makes most sense. This makes it nearly impossible to predict which thread is currently doing what at which time—some threads may be collaboratively working on a big job while other each work on different smaller tasks; and yet others may already be independently building entire sub-trees on their own, or even do something totally different like rendering tiles when no unclaimed build jobs are available.

At first, this loss of control seems like a bad idea in that different kinds of jobs could interfere with each other. In practice, however, at least as long as all threads behave reasonably we have not observed any negative effects. In fact, to some degree the opposite is true: a chaotic behavior means that a core's four threads usually execute a mix of different kernels with different properties—such as a bandwidth-intensive partitioning kernel and a compute-limited binning kernel—which allows the core to balance their individual bottlenecks. In particular, threads can almost always pick some useful work to do, while still being able to exploit locality and improve concurrency where possible.

7 Results

With the entire framework in place we can now evaluate its performance. The scenes used for our experiments are depicted in Figures 1, 3, and 5.

7.1 Experimental Setup

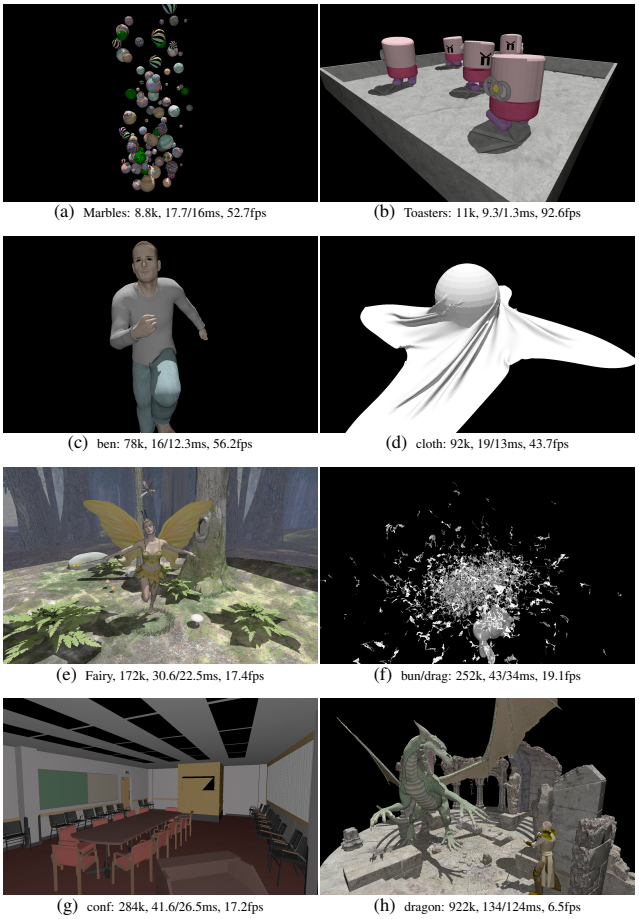
All experiments are run on actual hardware, using a "Knights Ferry" MIC archictecture prototype board [1]. While architectural details

like SIMD width, cache sizes and instruction set are already publicized [5, 25], actual core counts and clock frequency have not yet been disclosed. Though our prototype board's parameters may or may not vary from these settings, for the purpose of this paper we normalize all numbers to a round, and purely hypothetical, number of 32 cores and 1GHz (which would translate to a hypothetical peak FLOP rate of 1TFLOP/s). For the remainder of this paper, we omit the terms "normalized" and "hypothetical", always understanding that actual hardware parameters may be either higher or lower.

The BVHs are built using our parallel builder, using the default parameters as described above (ie, no per-scene parameter tuning). Rendering is done using a straightforward 16-ray packet tracer doing recursive packet tracing including texturing, alpha textures, and shadows from a directional light. Though none of our scenes features reflective surfaces, the alpha textures can require multiple rays to be traced for each pixel and/or shadow computation. Animation of the vertices is done on the host, and a full set of vertices is uploaded to the card every frame; building and rendering are performed entirely on the card.

7.2 Relative BVH Quality

Before measuring actual performance results, we first evaluate the quality of our generated BVHs. The only possible source of reduced BVH quality is through the 10-bit quantization. In Table 2, we compare our BVH quality—with and without re-quantization—to a (offline) reference builder without quantization. As can be seen, without re-quantization BVH quality is measurably lower. As it turned out, these lower-quality BVHs are much shallower than the reference BVHs (having roughly half as many nodes), which does indeed indicate that the builder ran out of bits to represent possible splits.



(a) Marbles: 8.8k, 17.7/16ms, 52.7fps (b) Toasters: 11k, 9.3/1.3ms, 92.6fps

(c) ben: 78k, 16/12.3ms, 56.2fps (d) cloth: 92k, 19/13ms, 43.7fps

(e) Fairy, 172k, 30.6/22.5ms, 17.4fps (f) bun/drag: 252k, 43/34ms, 19.1fps

(g) conf: 284k, 41.6/26.5ms, 17.2fps (h) dragon: 922k, 134/124ms, 6.5fps

Fig. 5. Sample frames from the scenes used in our experiments. Numbers below the scene indicate scene complexity (in #triangles), build time (synchronous/incremental, in ms), and aggregate build+render performance (in FPS, @1920×1200, including shading, shadows, and texturing).

Once enabling re-quantization and local sub-tree re-fitting as described above, however, this discrepancy mostly disappears, resulting in BVHs that are within a few percent of the reference BVHs (the only exception is the otherwise very simple *Toasters* scene, for which we do not currently know why it behaves differently). More complex scenes may require more than one re-quantization, we re-quantized only once, when switching to local build jobs.

Note that as long as build time is a dominant factor quantizing only once is actually *faster* in aggregate performance: running out of bits precludes some small splits that lead to shallower BVHs (on average, single-quantized BVHs have only half the number of nodes), and consequently build times that are about 30% faster. However, to facilitate comparisons with non-quantizing approaches we will, for the remainder of this paper, always use the (slower) re-quantizing variant.

scene	ref	single quantization		re-quant +local ref	
fairy	2456	2666	92%	2459	100%
conf	8120	9169	89%	8190	99%
dragbun	14.98	19.2	78%	15.4	97%
dragon	17553	23151	76%	17801	99%

Table 2. SAH quality produced by our builder vs a (offline) reference builder using $16 + \sqrt{N}$ bins and no quantization.

7.3 Scalability in Core Count and Scene Size

Scalability of our method is depicted in Figure 6. As can be seen, large enough scenes scale almost perfectly linearly, and scalability breaks only for "too simple" scenes for which a block size of 512 triangles produces too few parallel tasks to keep 128 threads busy. This could be alleviated by using a small block size for these scenes, but at higher overhead for the more interesting scenes. To allow easier comparisons, we intentionally did not do any per-scene parameter tweaking.

This lack of scalability for small scenes also means that there is a certain "minimum cost" below which our system can not drop. This effect can be seen, for example, by looking at the different scene' synchronous build times in Figure 5: while build time is roughly linear in scenes size for scenes of around 100k triangles and above, even trivial scenes will not drop below 10ms. Though to a much reduced degree, this effect can still happen even in asynchronous mode, since those trivially small scenes can sometimes render so fast that threads eventually run out of even low-priority render jobs.

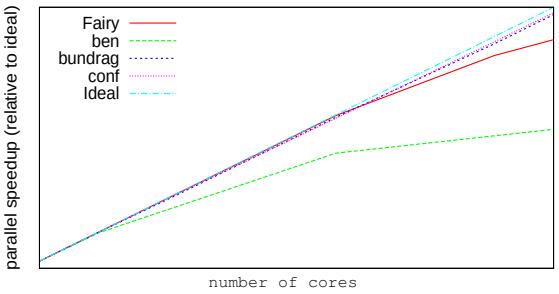


Fig. 6. Build performance relative to a single core. At 512 fragments per block, small scenes like ben do not generate enough parallel jobs to scale to all cores, but larger scenes like Fairy, conference and bundrag scale near-linearly. All scenes use the same set of default parameters.

7.4 Absolute Performance

Absolute performance data for both build and render—as well as comparable numbers from previously published approaches, where available—are given in Figures 1 and 5. The approaches we compare against use wildly different hardware platforms, data structures, quality thresholds, and render settings. To draw any conclusions from this comparison at all, we intentionally make the comparison as conservative as reasonably possible: in particular, we always use the highest-quality settings for our BVH builder, and never resort to shallower BVHs or spatial median splits, even though previous authors have

scene	CPU Grid	GPU Grid	CPU Grid	CPU kd-tree	GPU kd-tree	GPU kd-tree	GPU Hybrid	CPU BVH	CPU BVH	GPU SAH	MIC SAH
	2xXeon	GTX280	16x Opt.	4xXeon	8800GT	GTX280	GTX280	8x Xeon	8x Xeon	GTX280	MIC
SAH	n/a	n/a	n/a	partial	partial‡	partial‡◇	partial	partial	full	full	full
	Wald [22]	Kaloj. [24]	Ize [23]	Shevtsov [17]	Zhou [18]	Hou [19]	Lauterb. [2]	Wald [3]	Wald [3]	Lauterb. [2]	**[ours]**
Toasters	-	-	-	-	12ms	-	-	-	-	-	**11/1.3ms**
(11k)	9.4fps	-	-	-	55fps	-	-	-	-	-	**105**
	7.3fps	-	-	23.5fps	(33fps)	-	-	36fps	-	-	**93fps**
cloth	-	-	-	-	-	-	-	-	-	-	19/13ms
(92k)	-	-	-	-	-	-	-	-	-	-	**97**
	-	-	-	-	-	-	-	24fps	-	-	**44fps**
Fairy	68ms	24ms	-	-	77ms	58ms◇	124ms	**21ms**	70ms	488ms	31/23ms
(174k)	1.3fps	3.5fps	-	-	12.8fps	8fps◇	11.6fps	-	-	21.7fps	**29**
	1.1fps	3.2fps⋆	-	5.8fps	6.4fps⋆	5.4fps⋆◇	4.8fps⋆	11.5fps	-	1.8fps	**17fps**
BunDrag	-	**13ms**	-	-	-	-	66ms	20ms	70ms	403ms	43/34ms
(252k)	-	7.7fps	-	-	-	-	6.6fps	-	-	7.7fps	**55.7**
	-	7.3fps⋆	-	-	-	-	4.6fps⋆	14fps	-	1.9fps⋆	**19.1fps**
conf	89ms	27ms	**21ms**	-	-	-	105ms	26ms	96ms	477ms	42/37ms
(284k)	4.0fps	7.0fps	-	-	-	-	22.9fps	-	-	24.5fps	**46**
	2.9fps⋆	5.9fps⋆	-	6.2fps	-	-	6.8fps⋆	-	-	1.9fps⋆	**17fps**

Table 1. Comparison to other GPU- and CPU-based builders for various data structures and hardware platforms. For each entry, we report pure build time (top, in ms), pure render time excluding rebuild (middle, in fps), and aggregate render performance including rebuild (bottom, in fps). Best results are in bold. For papers that did not explicitly report aggregate performance data we that have computed this value based on individual build and render times (marked with ⋆). For some approaches (marked with ‡) exact tree quality metrics have not been fully specified in the paper. Note that due to wildly varying hardware as well as build- and render-parameters, all these comparisons are only partially conclusive. In particular, we always render at 1920×1200 with full shading and texturing, while other systems render at lower resolutions and/or simpler shading. ◇: Note that Hou et al. [19] can handle significantly larger scenes than either our or any other GPU approach. Hou et al. [19] also reports somewhat different performance data for the Zhou et al [18] method (in which building is faster but tracing is slower).

shown speedups of 3–4× when doing so [2, 3]. To adjust for different render settings, all our render timings include shading, texturing, transparency textures, and shadows. We also always render at the highest resolution possible on our system (1920×1200, vs 1024×1024 for most other systems), and never use any frustum traversal techniques even though some other approaches ([22] and [3]) do.

Even under these intentionally over-conservative conditions our approach consistently outperforms all competitors in aggregate render performance, and, somewhat more surprisingly, outperforms most even in build time. The only approaches that are at least sometimes faster in build time are—obviously—the grid-builders and, in one instance, Wald's CPU-based BVH builder, (and even then, only when using spatial median splits in the upper levels of the tree).

Numbers for Pantaleone's hybrid SAH/HLBVH build are unfortunately not available for any of the commonly used test scenes we used in our paper, but a back-of-the-envelope comparison with his numbers indicates that our performance is at least competitive with his numbers, despite us performing a full SAH build, and despite us not yet exploiting any possibly existing scene coherence; the same goes for the BVH build method described in [19].

7.5 Cost Break-down

Though a detailed cost break-down for all scenes and parameter settings is beyond the scope of this paper some initial profiling indicates that the lion's share of compute time is spent in the SIMD binning kernel: Thanks to caches and prefetching all that time is actually spent in "compute cycles", and the main reason this is so expensive is that—as argued before—updating 16 bins in parallel requires more FLOPs than updating only a single relevant bin (as a scalar or 4-wide architecture could have done [16]). In *synchronous* mode, the resulting load-imbalance at the end of the build process also translates to a significant number of cycles spent idling for work; in asynchronous mode, however, this is not the case, and the vast majority of time is spent in SIMD binning.

8 Summary and Conclusion

We have described a framework for the fast construction of SAH BVHs that is particularly designed for the Intel MIC architecture. In particular, we have used four key concepts: a hierarchically quantized data layout that exploits our architecture's support for DX data types and that minimizes cache footprint without compromising BVH quality; a AoSoA layout that combines AoS' data locality with SoA's SIMD-friendliness; highly optimized, low-level SIMD kernels for binning and partitioning; and a parallelization framework that has been particularly designed to maximize concurrency (tasking system, implicit stack-order, asynchronous building), minimize synchronization overhead (persistent runfunctions and deferred, horizontal merge), and maximize locality (implicit stack-order).

Using these techniques we achieve SAH BVH build performance that is competitive with or faster than previously published results, including those for significantly simpler data structures. In particular, normalized to a hypothetical 32-core 1 GHz Knights Ferry configuration we outperform even multi-socket multi-core CPU systems in build time (and even more so in aggregate render time); we outperform Lauterbach's GTX280-based approach by up to 15× even in synchronous mode (and up to 22+× in asynchronous mode), and even outperform its simpler spatial median builder by up to 4×; and we even come very close to the so-far best published build times for regular grids (which we significantly outperform in aggregate frame time). Though achieving this absolute performance is obviously at least partially due some architectural advantages of our underlying hardware architecture (eg, having real, large-enough, and coherent caches, support for stream compaction, ∼2x the FLOPs of a GTX 280, etc), it is only through our techniques that this potential is fully realized. While our particular implementation is explicitly geared towards the Intel MIC architecture, some of the ideas used in this approach (eg, reducing cache footprint through quantization, the AoSoA data layout, or the horizontal merging) might also benefit other highly parallel hardware architectures.

While this paper has intentionally focused only on best-quality SAH BVHs, an obvious next step is to also consider quality-vs-performance trade-offs like shallower BVHs and spatial median or (H)LBVH splits; with the given framework in place doing this should be rather straightforward. Similarly obvious is to even further improve render times by, for example, integrating Benthin's Multi-Frustum Traversal technique [28]. MFT prefers somewhat shallower BVHs [28], which we could accomodate by simply not doing the re-quantization; this would then produce a double win, improving both build times and render performance at the same time. Finally, in particular once those two features are added it would be interesting to use our technique as core ingredient for to build a complete ray tracing system in which the host application could then operate without any constraints at all in how it organizes and animates its geometry.

REFERENCES

[1] Intel, "Intel news release: Intel unveils new product plans for high-performance computing," http://www.intel.com/-pressroom/archive/releases/20100531comp.htm, May 2010.

[2] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH Construction on GPUs," *Computer Graphics Forum*, vol. 28, no. 2, pp. 375–384, 2009.

[3] I. Wald, "On fast Construction of SAH-based Bounding Volume Hierarchies," in *Proceedings of the 2007 IEEE/Eurographics Symposium on Interactive Ray Tracing*, 2007, pp. 33–40.

[4] T. Aila and S. Laine, "Understanding the Efficiency of Ray Traversal on GPUs," in *Proceedings of the 2009 ACM Symposium on High Performance Graphics*, 2009.

[5] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Transactions on Graphics (Proceedings of SIGGRAPH '08)*, vol. 27, no. 3, pp. 1–15, 2008.

[6] M. Ernst and G. Greiner, "Multi Bounding Volume Hierarchies," in *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing*, 2008, pp. 35–40.

[7] M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies," in *Proceedings of High Performance Graphics*, 2009.

[8] B. Fabianowski and J. Dingliana, "Compact BVH Storage for Ray Tracing and Photon Mapping," in *Proceedings of the 9th Eurographics Ireland Workshop*, 2009, pp. 1–8.

[9] J. Goldsmith and J. Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing," *IEEE Computer Graphics and Applications*, vol. 7, no. 5, pp. 14–20, 1987.

[10] K. R. Subramanian and D. S. Fussel, "Factors affecting performance of ray tracing hierarchies," The University of Texas at Austin, Tech. Rep. Tx 78712, July 1990.

[11] V. Havran, "Heuristic Ray Shooting Algorithms," Ph.D. dissertation, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[12] W. Hunt, G. Stoll, and W. Mark, "Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic," in *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, 2006.

[13] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek, "Experiences with Streaming Construction of SAH KD-Trees," in *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, 2006.

[14] W. Hunt, W. R. Mark, D. S. Fussell, and G. Stoll, "Fast and Lazy Build of Acceleration Structures from Scene Hierarchies," in *Proceedings of the 2007 IEEE/EG Symposium on Interactive Ray Tracing*, 2007.

[15] J. Pantaleoni and D. Luebke, "Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry," in *Proceedings of High Performance Graphics 2010*, 2010, pp. 87–95.

[16] I. Wald, S. Boulos, and P. Shirley, "Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies," *ACM Transactions on Graphics*, vol. 26, no. 1, pp. 1–18, 2007.

[17] M. Shevtsov, A. Soupikov, and A. Kapustin, "Fast and scalable kd-tree construction for interactively ray tracing dynamic scenes," *Computer Graphics Forum*, vol. 26, no. 3, 2007, (Proceedings of Eurographics).

[18] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," in *Proceedings of Siggraph Asia*, 2008.

[19] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, and D. Manocha, "Memory-Scalable GPU Spatial Hierarchy Construction," *IEEE Transactions on Visualization & Computer Graphics*, June 2010.

[20] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart, "Parallel SAH k-D Tree Construction," in *Proceedings of High Performance Graphics 2010*, 2010, pp. 77–86.

[21] E. Reinhard, B. Smits, and C. Hansen, "Dynamic acceleration structures for interactive ray tracing," in *Proceedings of the Eurographics Workshop on Rendering*, Brno, Czech Republic, June 2000, pp. 299–306.

[22] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker, "Ray Tracing Animated Scenes using Coherent Grid Traversal," *ACM Transactions on Graphics (Proceedings of SIGGRAPH '06)*, vol. 25, no. 3, pp. 485–493, 2006.

[23] T. Ize, I. Wald, C. Robertson, and S. G. Parker, "An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes," in *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, 2006, pp. 47–55.

[24] J. Kalojanov and P. Slusallek, "A Parallel Algorithm for Construction of Uniform Grids," in *Proceedings of High-Performance Graphics*, 2009.

[25] Intel LRBni, "C++ Larrabee Prototype Library," http://software.intel.com/en-us/articles/prototype-primitives-guide/, 2009.

[26] M. Abrash, *Graphics Programming Black Book*.

[27] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *ACM SigPlan Notices*, vol. 30, no. 8, p. 216, 1995.

[28] C. Benthin and I. Wald, "Efficient Ray Traced Soft Shadows using Multi-Frusta Tracing," in *Proceedings of the 2009 ACM Symposium on High Performance Graphics*, 2009.