Volume 0 (1981), Number 0 pp. 1–14 COMPUTER GRAPHICS *forum*

Massively Parallel Hierarchical Scene Processing with Applications in Rendering

Marek Vinkler[1] Jiří Bittner[2] Vlastimil Havran[2] Michal Hapala[2]

[1]Masaryk University, Brno
[2]Faculty of Electrical Engineering, Czech Technical University in Prague

Abstract

We present a novel method for massively parallel hierarchical scene processing on the GPU, which is based on sequential decomposition of the given hierarchical algorithm into small functional blocks. The computation is fully managed by the GPU using a specialized task pool which facilitates synchronization and communication of processing units. We present two applications of the proposed approach: construction of the bounding volume hierarchies and collision detection based on divide-and-conquer ray tracing. The results indicate that using our approach we achieve high utilization of the GPU even for complex hierarchical problems which pose a challenge for massive parallelization.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques—[Graphics data structures and data types] I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—[Ray tracing]

1. Introduction

Hierarchical algorithms and data structures are powerful tools for efficient processing of computationally intense problems. Hierarchical data structures such as bounding volume hierarchies or kd-trees have become standard methods for rendering acceleration particularly when targeting ray tracing based techniques. Apart from the established methods based on spatial hierarchies, some new techniques such as the divide-and-conquer ray tracing [WK09,Mor11,Áfr12] work with an implicit hierarchy stored in a simple index array. Such methods may become an interesting alternative for ray tracing highly dynamic scenes.

While the hierarchical techniques have their provable benefits in terms of algorithmic efficiency, the general drawback is their difficult mapping to the massively parallel computational model of the GPU. While a number of clever solutions for this mapping have already been designed, most of the proposed techniques rely on management of the computation from the CPU side, invoking specialized computational kernels at different stages of the computation. This is due to the fact that different computational stages of the hierarchical techniques exhibit different levels of parallelism and it is not easy to reflect this using the currently available frameworks for GPU computation such as CUDA or OpenCL. Therefore, the GPU may get underutilized if for any kernel there is not enough work for each processing unit. As a result the scalability of the CPU managed method might be reduced when targeting massively parallel systems with tens of thousands of processing units, which are likely to become available in the future.

In this paper we propose an innovative method which moves the whole computation to the GPU and it requires no management from the CPU side. The method handles all important aspects of hierarchical techniques as it is able to perform complex evaluation of the given task, spawn new tasks, and handle the dependencies among the tasks. We provide two applications that justify the concept for our method: Bounding Volume Hierarchy (BVH) construction and divide-and-conquer ray tracing. The results indicate that the implementations based on our method perform comparable or even superior to existing solutions.

Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

2. Related Work

We review here in short the relevant background knowledge in GPU algorithms and spatial sorting with focus on the building of hierarchical data structures on GPUs.

GPUs and Load Balancing. Load balancing and scheduling for GPU architectures is an active research area which relates to the method proposed in our paper. Tsigas and Zhang [TZ01] proposed a simple non-blocking concurrent queue for FIFO processing for shared memory multiprocessor systems that utilized compare-and-swap operations (CAS). With the availability of atomic operations on GPUs Cederman and Tsigas [CT08] compared four approaches for dynamic load balancing on GPUs and concluded that blocking queues perform the worst. Tzang et al. [TPO10] studied efficiency of load balancing methods for irregular workloads on the GPU and they concluded that task-stealing and task-donation are the most efficient. Chen et al. [CVKG10] proposed a task-based dynamic load balancing approach for single and multi GPU computer systems. They used a persistent kernel running on a *device(s)* (a GPU or several GPUs) where the task queue is generated on a *host* (CPU). The recent work by Sundell et al. [SGPT11] proposed a lock-free algorithm for distributing work on concurrent hardware without the restriction of work producers and consumers. Independently of our work Steinberger et al. [SKK*12] designed a flexible GPU framework, which also builds on the idea of persistent threads. Compared to their work our method is more specific to hierarchical scene processing and it provides dependencies among the tasks and better data level parallelism (more units can cooperate on solving the same task). We also want to point out a recent paper by Lee et al. [Lee10] who rigorously analyzed the performance of an NVIDIA GTX280 and an Intel Core i7 960 processor for fourteen different computational problems with carefully optimized implementations. They showed that the GPU-CPU performance gap narrows from the mythical 10-100 times to only 2.5 times on average.

GPU Rendering and Hierarchical Data Structures. There has been number of approaches dealing with the hierarchical data structures used in computer graphics for ray tracing, general visibility computations such as occlusion culling, collision detection etc. We focus our discussion on the bounding volume hierarchies (BVH) and kd-trees with the stress on the algorithms implemented on the GPU. In particular we focus not only on those that efficiently utilize the GPU for performing computations with the help of these data structures, but also on those that use the GPU for actually building these data structures. The first technique that used the GPU for ray tracing was proposed by Purcell et al. [PBMH02] who utilized a shading language and remapped a uniform grid into textures. This approach was followed by other methods which are surveyed by Wald et al. [WMG*09]. However, the data structures were typically prepared on the CPU and the memory footprint was transferred to the GPU to allow for parallel traversal operations. The building of data structures on the GPU have become possible with the introduction of CUDA [NBGS08] and OpenCL [SGS10].

Kd-trees. Zhou et al. [ZHWG08] presented an algorithm to build kd-trees on the GPU, restricting the approach to a spatial median and cutting off empty space. This approach was extended by Hou et al. [HSZ*11] using partial breadth-first-search to afford for limited memory consumption. Danilewski et al. [DPS10] presented a scalable GPU algorithm with binning for kd-trees that improves on the quality of constructed kd-trees following the method of Shevtsov et al. [SSK07]. Wu et al. [WZL11] proposed an algorithm running on the GPU as a sequence of kernels that construct kd-trees in a breadth-first search manner, but for all boundary positions in the fashion of the serial approach by Wald and Havran [WH06]. This algorithm was also parallelized for multi-core CPUs by Choi et al. [CKL*10].

Bounding Volume Hierarchies. Lauterbach et al. [LGS*08] presented an algorithm to build the Linear BVH (LBVH) using Morton codes, where the speed is moderately penalized by the quality of the built BVH. Aila and Laine [AL09] studied different possibilities to organize the traversal code on GPU architectures to get the highest performance. Pantaleoni and Luebke [PL10] presented a more efficient version of the LBVH algorithm with Morton codes and compress-sort-decompress strategy, together with improved memory management. They call it the Hierarchical LBVH (HLBVH). Further, they presented a hybrid algorithm with a two-level BVH, where top levels are built with an exact algorithm with a surface area heuristics (SAH) [Wal07] and bottom levels with a Morton curve based algorithm. Garanzha et al. [GPM11] simplified the HLBVH algorithm using binary search and work queues. They achieved both memory savings and lower build times than the paper by Pantaleoni and Luebke [PL10]. Wald described a parallel version of a BVH based builder with SAH using binning on a many-core architecture (MIC) [Wal12]. Sopin et al. [SBU11] studied binned SAH BVH construction on the GPU with focus on efficient division of data between computational units.

Grids. Kalojanov and Slusallek [KS09] presented a parallel algorithm for building uniform grids, followed by another paper by Kalojanov et al. [KBS11] for hierarchical grids.

Implicit Hierarchies. Wächter and Keller [WK09] presented an approach for ray tracing which simultaneously subdivides rays and triangles and can be computed without explicit spatial data structures. A similar approach was independently developed and implemented on a single-core CPU with SSE instructions by Mora [Mor11]. Mora also utilized bounding cones for primary rays to improve the performance.

3. Hierarchical Scene Processing

In this section we first present the terminology and an overview of our algorithm and then propose a novel general methodology of mapping a hierarchical algorithm to the GPU framework. We limit our discussion to CUDA [NBGS08] based implementations and use terminology and constants associated with the currently available CUDA platforms.

3.1. Terminology

Prior to introducing the algorithm we briefly define the basic terms used in the paper.

- *Task* is a computational job which is associated with the given range of scene data (geometry, ray queries, etc. stored in the linear array in contiguous block of memory). The whole computation is initiated using a single task associated with the whole scene. After a task is processed, it is either finished or spawns one or more child tasks. Every child task processes the associated range of data. The task is characterized by its state.
- *Phase* is a logical algorithmic block of the task, such as finding the splitting plane, sorting triangles, computing a tight bounding box, etc.
- *Step* is an algorithmic block of the phase. A phase might consist of a single step, but some phases need more steps. The number of required steps may depend on the size of the given data range. If the phase consists of more steps, the results of one step are processed by the further steps in order to compute an aggregated result of the whole phase. An example when more steps are needed is a parallel reduction computation used for computing the new AABB of child nodes, which requires a logarithmic number of steps.
- *Work chunk* is a data range associated with a particular step processed by a single warp. The work chunk is the smallest unit of work in our method. Note that while the phase and the step represent a subdivision into smaller algorithmic blocks (i.e. in the time domain), the work chunk represents a subdivision of the data associated with the step (i.e. in the contiguous block in memory address space). The work chunk consists of 32 data items and each thread in a warp processes a single item.
- *Task pool* is a data structure used for managing the execution of tasks. In our method the task pool is not working as a queue nor stack. This is implied by required computational efficiency as well as computational dependencies among the tasks. The details on the task pool will be given in section 3.4.

Apart from the above defined terms we recall the basic terminology associated with CUDA: *kernel* is a program executed on the CUDA device and *warp* is a group of 32 threads, which execute the same instruction at a time.

3.2. Algorithm Overview

Our algorithm follows the divide-and-conquer principle of hierarchical methods: when the current task is too large to be solved directly it is further subdivided until it is small enough to be terminated or solved in a trivial way.

Each task holds the information about its data range (e.g. interval in the triangle index array) and the state of the task. The task also holds information describing the current phase, the current step, the number of available work chunks, and auxiliary information such as the bounding box of the given geometry data.

A typical task dealing with 3D primitives can be divided into two major phases which are processed sequentially: determining a quicksort-like pivot for one phase of sorting (e.g. a splitting plane) and sorting the primitives according to this pivot into two parts using the index array. Note that this sorting can take place more times to create multiple subsets. After these phases the algorithm continues with subsets of these primitives in a given number of branches. The algorithm can also contain other phases, which evaluate data needed for further invocation of the algorithm such as bounding boxes. An example of the computational phases for the BVH construction is illustrated in Figure 1.

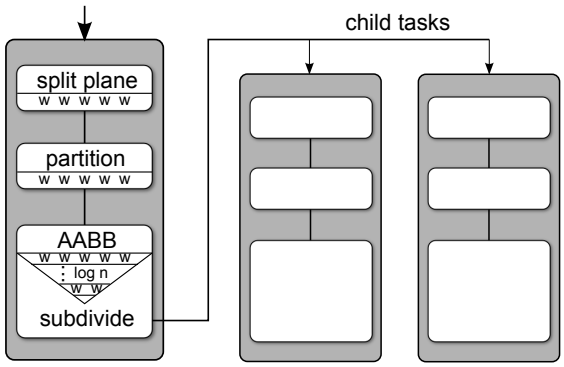


Figure 1: *Overview of the task and its phases in an algorithm for BVH construction. The figure also shows the relative number of warps cooperating on solving the particular steps of the task phases.*

Our algorithm is built on the concept of *persistent warps* [AL09]. Using persistent warps instead of threads or blocks has several advantages. First, it is easier to manage memory access coherency and branching as they are resolved on this level in hardware. Second, there are several horizontal warp-level functions that can accelerate the processing.

We aim at maximizing the parallelism of the computation on two different levels. First, we aim to process a given step of the computation using as many threads as possible (fine grain spatial parallelism), i.e. the number of threads working

on the given step corresponds to the size of the data range associated with the step. Second, we aim to compute different tasks in different computational phases in parallel (coarse grain temporal parallelism). For example, we want to determine a splitting plane for one node in the hierarchy using a number of warps and at the same time we perform sorting of the triangles in some other node using the remaining warps.

For some algorithmic problems it is possible that several tasks may need to work on the shared data range. For example when constructing a kd-tree, the subsets of triangles associated with the left and right children of a node generally overlap and the data ranges of the associated child tasks overlap as well. In such a case it is necessary to enforce an order on the task execution, and we mark some tasks as dependent on other active tasks. These dependent tasks must wait to be activated upon the completion of active tasks. A dependent task holds the counter on how many tasks have to finish before it is activated. An active task contains pointers (indices) to tasks it is responsible for activating.

3.3. Managing the parallel computation

We launch a single kernel with as many persistent warps as can be run simultaneously on the GPU. After launching this kernel there is no further management from the CPU and the work flow takes place completely on the GPU. The crucial component in our system is the *task pool* stored in the global memory: all warps are synchronized and take their work from the task pool. The task pool holds all the information about the current state of the computation.

When the kernel is launched the task pool is filled with a single task. This task encapsulates all the geometry (e.g. triangles). When this task is finished it can spawn its child task(s) until the whole task pool is empty, signalling that the computation is done.

Since warps are independent in CUDA, each warp can process a different task with a different state. However, in our method warps also participate in computation of the same task. This is in contrast with the previous approaches where communication and synchronization between warps was either limited or not possible at all. Each warp takes work chunks from an arbitrary active task based on the current distribution of work in the task pool. The pseudocode of the method is shown in Algorithm 1.

The pseudocode shows that the warps are constantly searching for arbitrary work chunks that they can handle. When they succeed in retrieving the work chunks they perform the work according to that particular task and its phase and step. The overview of the main data structures used in our method is depicted in Figure 2.

3.4. Task Pool

The task pool is represented by two arrays, one for holding all the information necessary for computing the task (*task data array*) and the second compact one for defining the amount of work to be done in the current step (*task header array*). Because of this decomposition the task header array contains a single integer for each task. This gives a very small memory footprint that can fit easily into the cache on modern GPU architectures.

```
In serial: Insert the first task into the task pool;
In parallel: while Task pool is not empty do
    if retrieve(taskIdx, work chunks) was successful
    then
        read task from task data array;
        switch task phase do
            repeat
                process_task (step, work chunk);
            until no more work chunks ;
            memory fence;
            advance the state or finish the task;
        end
    end
end
Computation is done;
```

Algorithm 1: Main loop of the algorithm.

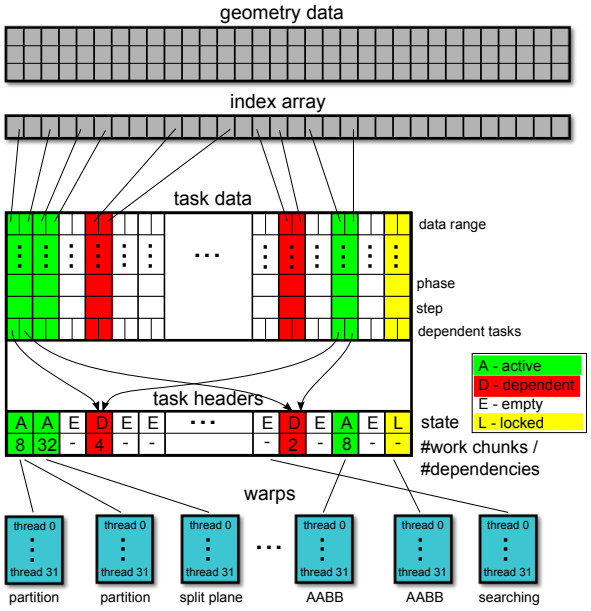


Figure 2: *Overview of the main data structures used in our method.*

For each task the header array encodes the task state in an integer value. Apart from the task state we also encode additional quantitative information in this integer value, the meaning of which depends on the task state. This additional information allows us to use efficient mechanism to retrieve work chunks and handle task dependencies. The task can ex-

hibit one of the following four states in its header by the integer value I:

- $I > 0$: *Active* state. The task is ready to be processed and there are I work chunks to be done on this task for the current phase and step. Note that the phase and the step is stored in the task data array. Below we call a task in active state an active task.
- $I < 0$: *Dependent* state. The task waits for $-I$ other tasks to finish before it is activated.
- $I = 0$: *Locked* state. The task is locked, which means that its data entry is just being created or modified.
- $I <= -BIG\_INT$: *Empty*. This entry in the task pool is currently not used and it can be populated with a new task.

Retrieving work. When warps are trying to find a work chunk to process they loop through the task header array searching for an active task. To promote parallelism each warp starts at a different index in the pool, based on its warp ID and the number of entries in the pool. Each thread in the warp then reads the state of one consecutive entry from the task header array. As multiple entries can be active, it has to choose one to take work from. To prevent all warps from choosing the same active task introducing conflicts of atomic operations, we compute the prefix sum on the states of the entries within each warp and choose the i-th active task, where i is based on the warp ID. When the active tasks are chosen, the warps atomically decrement the tasks' header. Each warp may decrement the value by any number i.e. retrieve as many work chunks from a single entry as it desires. It is often beneficial to retrieve multiple work chunks in one atomic operation because the overhead of retrieving the work chunk is not negligible. We use the following function for determining the number of work chunks retrieved by the warp:

$$N_w = max(\frac{S}{W} + 1, K), \quad (1)$$
$$S = \sum_{i=1}^{32} N_i,$$

where N_i is the number of work chunks corresponding to the task entry sampled by thread i of the warp; W is the number of warps launched on the GPU, and K is a constant preventing the retrieval of too few work chunks. Note that N_i represents the number of work chunks the current task step was created with and $N_i = 0$ for all inactive tasks.

The first term $\frac{S}{W} + 1$ aims to distribute the available work among other warps, while the constant K prevents the fragmentation of work and in turn it bounds the overhead connected with the task pool management, especially in the later stages of the computation when the processed tasks consist of smaller amount of data. Note that $K = 14$ was experimentally verified to be a reasonable choice in practice for contemporary GPU architectures.

It may happen that the value of the entry is decremented below the value representing the *Lock* state by multiple warps concurrently trying to retrieve work from the same task. This is not a problem as long as the counter is not decremented to the value representing an empty task. If the warp did not succeed in retrieving the work chunks the value returned by the atomic decrement is not positive. In that case the same process is repeated on a different task.

Finishing work. When warp finishes the retrieved work it has to communicate this fact to the other warps. In particular the last finished warp has to be aware that it is responsible for advancing the task state or issuing new tasks. To accomplish this we use another counter of unfinished work chunks stored in the task's data. This counter is atomically decremented by each finished warp. The warp that decrements it to zero is the last finished warp. This warp can then interpret the results of the step and progress the computation further to the next step or phase for the given task. We cannot use the value obtained from the task's header for this purpose since the warp that last retrieved work from the task need not be the warp that finishes it last.

Storing work. In order to create a new task the warp loops through the task header array searching for an empty entry. When it finds one it atomically compares-and-swaps its value with the value representing a lock. If it succeeds, it fills the corresponding entry in task data array with the child task. As the last step, it sets the header array entry with the number of work chunks required to process the first step of the first phase of the child task to unlock it, or it sets the entry with the number of tasks this task is dependent on to mark a dependent task. Note that a memory fence operation must be issued before the task is unlocked to make sure valid data are visible to other warps.

Handling dependent tasks. Working with dependent tasks is straightforward in our framework. Since their task header value is less than the *Lock* state they are ignored by the warps during the retrieving or storing of work. The active tasks that point to this dependent task increase the dependent task's header value upon their finish, eventually increasing it to the *Lock* state. This signals all dependencies are resolved and the header value can be overwritten by the number of work chunks in the task, signalling the *Active* state.

Minimizing pool overhead. We use two improvements that accelerate the computation of tasks. They are both targeting steps with little parallelism. First, when some phase requires zero work chunks to compute it is immediately skipped. Second, if some step requires less than K work chunks, this step is processed immediately by the given warp without writing the task into the pool (K is the constant used in Eq. 1). This is often the case with the reduction in the *AABB* phase.

3.5. Comparison to Standard Kernel Launching

Performing the entire computation and management on the GPU has several advantages compared to the standard method based on serial kernel launching and synchronization. First, the intermediate results need not be saved to global memory (on a GPU) between consecutive kernel launches or transferred over an even slower PCI-E bus to the main memory. Our approach is in an agreement with the GPU evolution which places more computation on the GPU side to limit the communication. Moreover, during these data transfers and kernel launch preparations the GPU is idle (if there is no concurrently running kernel). Also the kernel launch is a high overhead operation as stated by several authors [ZHWG08, GPM11].

Managing the computation on the GPU has other advantages besides limiting overhead. While the available parallelism is fixed for the kernel launching approach e.g. to a single level of the hierarchy (spatial parallelism), in our approach nodes from different levels can be processed simultaneously (temporal parallelism). This increases the available parallelism and limits the computation stalls due to underutilized GPU. For the kernel launching these stalls often happen when processing top levels of hierarchies where there is not enough data to process or when some warps have already finished their work and are waiting for other warps to terminate the kernel.

In the rest of the paper we discuss two applications of the proposed framework for parallelization of hierarchical algorithms: BVH construction and divide-and-conquer ray tracing.

4. Constructing Bounding Volume Hierarchies

Bounding volume hierarchies are common data structures used for rendering acceleration. They became particularly popular for ray tracing acceleration of dynamic scenes since they are relatively fast to construct and update, and have predictable memory footprint.

The algorithm for constructing a BVH can be easily mapped to our parallel framework as we describe in the next sections. The ease of mapping the BVH build comes mainly from the fact that each task is completely independent of other tasks. For the rest of the paper we assume that the scene consists of triangles although the method can generally handle other scene primitives as well.

4.1. Defining Phases and Steps

The computation starts with a single task associated with all scene triangles. Each task then needs to subdivide the given set of triangles into two disjoint subsets (assuming a binary hierarchy). The formation of these subsets is typically based on spatial criteria such as the spatial median or the more involved surface area heuristics (SAH). The subdivision can be easily implemented by sorting the triangle indices into two disjoint groups in the index array. If the given triangle subset is large enough, a new task is created. Otherwise, the current branch of the computation is terminated and a leaf is created.

For each task we define three different phases:

1. *SplitPlane*: Splitting plane computation (spatial median or cost model with SAH).
2. *Partition*: Partitioning of a triangle range into the left and right sub-ranges in the double buffered index array.
3. *AABB*: Computation of the two bounding boxes for the child tasks.

Note that some of these phases represent parallel divide-and-conquer algorithms on their own (*AABB*) and, therefore, require a logarithmic number of steps to complete. The illustration of the phases is shown in Figure 1. Note that the figure also shows the relative number of work chunks required by different steps of the tasks (indicated as w). The number of work chunks per step corresponds to the number of warps which perform the work according to Eq. 1. Below we describe the particular phases of the algorithm in more detail.

SplitPlane. Currently we support two splitting strategies: the spatial median and the SAH. The spatial median cycles the splitting plane in the round-robin fashion, where for the first task the longest axis is chosen. The SAH chooses the best plane from equal number of candidates in each axis, where the evaluation of the SAH cost is similar to [HHS06]. For the SAH strategy we select 32 candidate planes and evaluate their cost using the SAH in parallel. Each warp processes a distinct sub-range of a task's triangles from the index array and each thread computes their position with respect to one of the candidate planes. Then the number of triangles to the left and to the right of the splitting plane, as well as the bounding boxes are atomically updated in global memory. The warp that has finished its work last loads these data from the global memory and chooses the best splitting plane. As there are exactly 32 (warp size) candidates this is done in parallel as well. For the spatial median strategy the splitting plane is evaluated directly when the task is enqueued in the pool and this phase is skipped.

Partition. In this phase the triangles are divided into the left and right subsets based on the position of their centroid to the splitting plane. The method reads 32 consecutive triangles from the input index array and appends left triangles to the start of the output range and prepends right triangles to the end of the output range. Since the order of triangles in the left and right subsets is not important they can be written in arbitrary order, in our case the write offset is computed by *atomicAdd* to the start of the range and *atomicSub* to the end of the range. This atomic operation is done by a single thread in the warp and the returned value is used by all threads of the warp to compute their write offset using prefix scan. To prevent overwriting the input range, a new output array has to be used. We are using two triangle index arrays with each

task holding a pointer to either of the two arrays with the valid data.

AABB. Segmented parallel reduction is computed on the range in the triangle index array. The bounding boxes for the triangle ranges corresponding to left and right triangle subsets are computed using double-buffered array for storing the reduction tree. The computation requires $\log_2(\#tris)$ steps [HSO07]. This phase is only needed for the median splitting as the SAH evaluation already gives us the bounding boxes.

FullSAH. When the number of triangles in a task drops bellow the warp size it is possible to process the task more efficiently. In such a case we move it to a distinct phase that builds its subtree in one step using a single warp. The subtree is built using a full SAH computation that requires triangle sorting in all three axes. To make this operation efficient all the data are stored in registers and shared between the threads of a warp using the shuffle instruction introduced in the Kepler generation of NVIDIA GPUs.

4.2. Handling Tasks

Since the algorithm subdivides the current data range into disjoint subsets there are no data dependencies among different tasks and the tasks can be processed fully in parallel.

The two child tasks are created by the last phase, more precisely at the last step of the *AABB* phase for the spatial median splitting or *Partition* phase for the SAH based splitting. The algorithm first checks whether the termination criteria are met for the given subset of triangles. If this is the case (the number of triangles is below a threshold, a maximum depth is reached or the SAH termination takes place), a BVH leaf is created. Otherwise, a new task is stored to the task pool and it is initiated to the *SplitPlane* phase. When creating new tasks the method reuses the task pool entry for the current task and then it searches for an empty spot in the task pool to allocate the other child task.

5. Divide-And-Conquer Ray Tracing

In this section we describe the application of our method to the parallelization of the divide-and-conquer ray tracing algorithm proposed by Mora [Mor11], Keller and Wächter [WK09] and Áfra [Áfr12]. We first present a brief overview of this method and describe the phases and steps needed to cover the method in our framework.

5.1. Algorithm overview

The divide-and-conquer ray tracing is based on an idea of avoiding the explicit construction of a spatial data structure. Instead, the method performs a hierarchical computation in which an implicit spatial subdivision is used and maintained in an array of indices for both triangles and rays.

The method starts with all scene triangles and the set of rays to be cast. Then it picks up a splitting plane which subdivides the current bounding box into two smaller boxes. The method then sorts the triangle and ray arrays and recursively evaluates the triangles and rays intersecting one of the smaller boxes. When the recursion returns it resorts the rays and triangles to obtain those that intersect the other bounding box and performs recursion. The recursion is terminated if the number of rays or triangles is below a specified threshold. Then the intersections of rays and triangles are computed using a naive algorithm, computing the intersection among all pairs of rays and triangles.

While the recursive formulation of this method is simple, its parallelization is rather involved. The main problem is that the sets of rays and triangles intersecting the bounding boxes of the implicit spatial subdivision can overlap. We cannot evaluate all the child tasks in parallel using a single array of indices since different tasks would compete for sorting the ray and triangle ranges and storing the results. Therefore, we need to establish dependencies for the computation and handle them appropriately in the parallel version of the algorithm.

When a splitting plane is selected for the given bounding box all rays and triangles associated with the given task are classified as either lying left, right, or straddling the splitting plane. We aim to create child tasks which would cover all sub-ranges at which an intersection of rays and triangles can happen. As we have three ranges for both rays and triangles we obtain nine pairs of different ray/triangle ranges to process. Out of the nine pairs for two pairs of ranges no intersection can happen: (1) triangles lying left of the splitting plane and rays lying right of the plane and (2) triangles lying right of the plane and rays lying left of the plane. For the remaining seven range pairs we create child tasks and proceed with the computation. The subdivision into child tasks is illustrated in Figure 3.

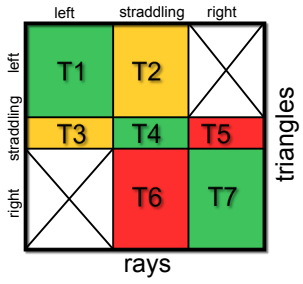


Figure 3: *Matrix representing a subdivision of the task into its child tasks for the divide-and-conquer ray tracing.*

There are clear computation dependencies among the child tasks shown in Figure 3: the tasks cannot be executed

simultaneously if they share some triangle or ray data (they are in the same row or column). There are several ways to execute and synchronize the tasks in order to avoid different tasks competing for the access to the same data. For example, the execution of the tasks can proceed as follows. We first activate three independent tasks T1, T4, and T7. Tasks T2 and T3 wait for execution as they depend on finalizing T1 and T4. Task T5 depends on T3 and T7 and task T6 depends on T2 and T7. More details about the task dependencies will be discussed in Section 5.3.

5.2. Defining Phases and Steps

For the divide-and-conquer ray tracing there are two types of tasks that can be created in the task pool: the intersection tasks and the subdivision tasks. The intersection task consists of one step with a number of work chunks which are set in a way that each warp processes 32 ray-triangle intersections in parallel (one intersection per thread). The closest intersection for each ray, if any, is then written to the global memory. Note that our implementation does not explicitly identify the type of the task. Instead for the intersection task we initiate it into a phase which implies a different task type (intersection phase).

The subdivision task is more complex. It consists of four phases that are computed sequentially (see Figure 4). Some of these phases are only a minor modification of the phases described for the BVH construction in Section 4.1. Since the rays are divided into four groups: left, straddling, right and clipped and triangles into three groups: left, straddling and right, the partition phase is executed twice, each time with a different pivot. Below we describe these phases in more detail.

SplitPlane. Again we support two splitting strategies: spatial median and cost model based splitting. A different cost model than SAH is used which is explained in this paragraph. Instead of using the SAH or a spatial median as proposed by Mora we use the Ray Distribution Heuristic (RDH) cost model by Bittner and Havran [BH09] which also takes the distribution of rays into account and achieves higher performance. The termination criteria are derived using this cost model; the intersection task is created when

$$\#tris \cdot \#rays \cdot C_{INTERS} < (\#tris + \#rays) \cdot C_{SORT}, \qquad (2)$$

where C_{INTERS} is the expected cost for one ray-triangle intersection and C_{SORT} is the expected cost for one sorting operation. We use 32 candidate planes that cover all three axes. The number of candidates is the same in each axis and the candidate positions are uniformly distributed. We do not use all triangles and rays associated with the given task for the evaluation of cost, but only their smaller subsets. The number of triangle samples N_T and ray samples N_R are computed as: $N_T = \sqrt{\#tris}$, $N_R = \sqrt{\#rays}$. The median splitting strategy is the same as for the BVH construction. Either strategy, this phase needs only one step.

Partition1. The partition is computed in parallel on both ray and triangle index arrays and runs in a single step. During the ray classification the rays that do not hit the bounding box of the current node are marked as clipped. These rays are treated as lying to the right of the splitting plane in this phase. Other than that this phase is exactly the same as the *Partition* phase in building BVH. As the result of this operation the rays are divided into two groups: left+straddling versus right+clipped and triangles are divided into left+straddling versus right. Also the sizes of the groups are known afterwards.

Partition2. This phase is done the same way as *Partition1* but for two ranges (the left+straddling and right[+clipped]) in parallel. After this phase the task's ray range is fully sorted into left, straddling, right, and clipped rays and triangle range into left, straddling, and right ranges with respect to the selected splitting plane.

AABB. This phase works similarly as for the BVH construction and requires $\log_2(\#tris)$ steps. The only difference is that we have to compute three new bounding boxes instead of two, since we compute the bounding box of the triangles straddling the splitting plane (tasks T2, T4, and T6).

5.3. Handling Tasks

As mentioned in Section 5.1 the child tasks resulting from the subdivision of a given task have certain dependencies and cannot be processed fully independently. Certain groups of child tasks are, however, independent. For the divide-and-conquer ray tracing we can formalize the dependencies among the tasks in a way that each task is responsible for activating at most two other dependent tasks. Initially we mark three child tasks as active and the remaining four tasks wait for being activated. Note that some of the child tasks need to inherit the activation pointers of the task being subdivided, as some other tasks may depend on it.

We propose to use the following subdivision into three independent task groups: (T1, T4, T7), (T2, T3), (T5, T6), which implies the following dependencies (TX→TY: TX activates TY, i.e. TY depends on TX):

- T1→T2, T1→T3,
- T4→T2, T4→T3,
- T7→T5, T7→T6,
- T3→T5,
- T2→T6,
- T5→P1, T5→P2,
- T6→P1, T6→P2,

where P1 and P2 are the dependencies inherited from the parent task. A task TY which should be activated by task TX has to be inserted first into the task pool. This is due to the fact that the task TX needs to know the index of the entry for the task TY in the task pool.

When an active task finishes, it updates the task header

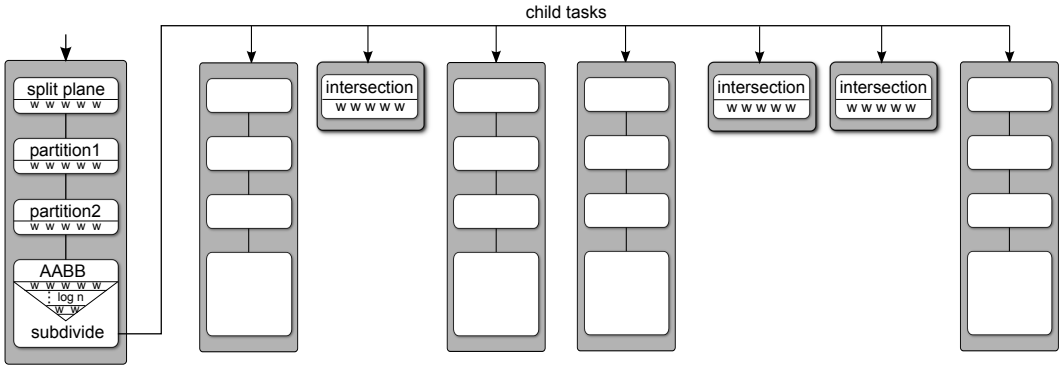


Figure 4: *Overview of the task phases and steps for the divide-and-conquer ray tracing algorithm.*

array for the dependent tasks. If this was the last dependence for that task the active task activates the dependent task by setting its entry in the header array to the number of its unfinished work chunks.

Often some of the child tasks T1-7 contain no rays or triangles in which case it is useless to add them into the pool. To prevent this the division of tasks into groups is defined by a look-up table. This table is queried when a parent task is divided into its child tasks. The index into the table is a binary array flag describing which ranges (left, right, straddling) are empty and which are nonempty. The table contains the number of child tasks to generate, number of child tasks in the last dependency group, the order in which the child tasks should be added into the task pool, the dependencies among the child tasks and the activity flag for each child task.

6. Results

We have implemented the proposed framework in C++ and CUDA [NBGS08]. For testing we have used a PC with Intel Core i7-2600, 16GB of RAM and NVIDIA GeForce GTX 680 running on Windows 7 64-bit. We have used two types of scenes for testing, individual objects and more complex architectural scenes. The images for the test scenes are shown in Figure 5.

6.1. Constructing BVHs

First, we tested the time for building BVHs using the parallel algorithm described in Section 4 and the traversal performance of these BVHs. We used three different ray distributions: primary rays, incoherent rays corresponding to ambient occlusion (AO) and diffuse rays shot from the hit points of the primary rays (seven AO or diffuse rays per primary ray). For reporting the SAH cost of the BVH we used the following traversal and intersection costs: $c_t = 3$, $c_i = 2$.

In order to show the quality of our method we compare it to our implementation of the state-of-the-art method of Garanzha et al. [GPM11]. According to the original paper we are using 30bit Morton codes of the triangle centroids, where $30 - 3k$ most significant bits are used for creating clusters of triangles falling inside the same voxel of the hypothetical grid. These clusters are the leaves of the top part of the tree built with SAH. The bottom part of the tree (each cluster) is built with fast HLBVH method using the least significant 3k bits of the Morton codes. This means that up to $3k$ bottom levels of the BVH are built using HLBVH (the constant k relates to constant m used in the original paper in this way: $k = 10 - m$). We denote the method with $k = 10$ as $HLBVH_M$ because it builds the entire tree with *median splitting*. Since the behaviour of the HLBVH method is strongly dependent on the number of bits used, we are always reporting the value of k. In the results we are using maximum of four triangles per leaf as termination criteria. The SAH termination cannot be used in this method as the SAH part of the tree is always built down to a single cluster per leaf and during the HLBVH construction bounding boxes are not known.

Our implementation of the HLBVH method does not feature multiple GPU queues mentioned in the original paper and has moderately slower build times. Nevertheless, the traversal performance of the HLBVH method should not be impacted and may be even superior as we are using the traversal kernels of Aila and Laine [AL09] with compact BVH layout and Woop triangle representation [Woo04].

For our method a leaf is created when the number of triangles in a node is four or less or when the SAH termination criteria are met. The building and traversal results are given in Table 1 where the HLBVH method with $k = 4$ (as proposed by Garanzha et al.) is compared to our method. The HLBVH build times start to be lower than the ones of our method for scene *Crytek Sponza*, for smaller scenes our method is in fact faster. This can be explained by the different complexity of the two algorithms: after sorting the Morton codes the complexity of HLBVH build is $O(N)$, while our method following the standard top-down scheme ex-

Figure 5: *Snapshots for more complex architectural models: Sponza, Sibenik Cathedral, Fairy Forest, Crytek Sponza, Conference, Soda Hall, San Miguel, Power Plant, and for single geometric object models: Happy Buddha, Blade, Hairball rendered using diffuse rays.*

Scene	$HLBVH_4$			$OurBVH$		
	T_{GPU} $[ms]$	T_{CPU} $[ms]$	R $[-]$	T_{GPU} $[ms]$	T_{CPU} $[ms]$	R $[-]$
Sponza	7.5	26.5	0.22	10.5	13.5	0.43
Sibenik Cathedral	6.9	27.3	0.20	12.9	13.7	0.48
Fairy Forest	11.2	27.0	0.29	19.7	14.6	0.57
Crytek Sponza	11.7	28.9	0.29	27.8	14.4	0.65
Conference	10.6	27.3	0.28	29.6	14.6	0.66
Happy Buddha	37.4	34.7	0.51	113.4	20.1	0.84
Blade	54.9	36.9	0.59	173.8	20.7	0.89
Soda Hall	56.0	37.8	0.59	228.7	20.5	0.91
Hairball	103.8	47.9	0.68	298.9	24.8	0.92
San Miguel	179.3	50.1	0.78	911.6	39.4	0.95
Power Plant	252.7	52.9	0.82	1452.1	49.0	0.96

Table 2: *Absolute GPU kernel time (T_{GPU}), CPU management time (T_{CPU}) and relative CPU idle time ($R = \frac{T_{GPU}}{T_{GPU}+T_{CPU}}$, higher is better).*

hibits $O(N \log N)$ complexity, but lower CPU management overhead. The traversal performance, on the other hand, is almost always higher for our method. This is not only because SAH splitting is used down to the leaves but also because the SAH evaluation in the top part of the tree is allowed to separate triangles that would fall into a single cluster for the HLBVH method.

Given that our method typically has slower build but faster traversal there is a crossover point where using our method leads to a lower rendering time. These points are evaluated in Table 3. Notice that on scenes with uniform size and distribution of triangles, such as Happy Buddha and Blade the tree quality cannot be improved much by the SAH and the crossover point lies very far ($> 100 MRays$). We believe the cases where the traversal for our method is slower are caused by the SAH being only approximate measure of performance.

The behaviour of the HLBVH method with varying number of bits used, given by the parameter k is shown in Table 4. Generally, the build times decrease with increasing value of k, while the traversal performance also decreases. This is in agreement with the increase of the SAH cost as more levels are built with the HLBVH. For smaller scenes the relations are not as straightforward, since the clusterization for low values of k may force creation of very small leaves, which hampers traversal performance.

Table 2 shows the build times on both the GPU and the CPU, and the relative GPU utility R. The value of R shows how much of the time needed for the data structure build is spent on the GPU in relation to the total build time. Note that value $(1-R)$ represents the time spent by copying the data to the GPU and managing the computation. With increasing build times, this management overhead is relatively less significant but still important when targeting realtime or interactive applications. Our method clearly features less CPU management overhead. Moreover, the CPU can perform some meaningful computations during the entire run of our kernel, while for the standard method the intervals between kernel launches are very short and the CPU must frequently interrupt its computation to keep the GPU busy.

6.2. Divide-And-Conquer Ray Tracing

Second, we have tested the divide-and-conquer ray tracing described in Section 5. We provide results for the spatial median subdivision and for the cost model based on the RDH compared to HLBVH with $k = 4$ and our BVH for tracing collision detection rays.

For the collision detection test we assume that the scene contains a number of moving agents (corresponding for example to characters in a game). The movement of the agents is given by randomly selected line segments within the bounding box of the scene. Next, we take 20 equidistant points on each segment that we use as the agents' positions in a simulation consisting of 20 frames. For each frame we shoot 128 ray segments into the sphere around the agents' position, where the length of the ray segment is the distance between the current position of the agent and the next one. The approximation of collision detection between moving agents and the scene is then computed as the intersections of the ray segments with the scene. Note, that since the line segments are random, the agent speed (corresponding to the length of the ray segments) varies among the agents.

Figure 7 shows the comparison of the four introduced methods for computing the intersections of the collision detection rays. The number of agents is denoted by #agents. While the computational times for the $HLBVH_4$ and $OurBVH$ are dominated by the build time, the computational time of the divide-and-conquer methods ($DACRT$) is dominated by the number and length of rays. RDH is usually faster than the median splitting and more so on complex scenes, which comes from the extra knowledge during

		SAH cost [-]		Build [ms]		Trace performance [MRays/s]					
						Primary		Ambient Occlusion		Diffuse	
Scene	#tris	$HLBVH_4$	$OurBVH$	$HLBVH_4$	$OurBVH$	$HLBVH_4$	$OurBVH$	$HLBVH_4$	$OurBVH$	$HLBVH_4$	$OurBVH$
Sponza	76k	188.6	200.8	34.0	24.0	306.5	259.5	182.4	173.4	54.4	51.2
Sibenik Cathedral	80k	80.6	75.7	34.2	26.6	203.0	231.2	167.7	197.2	42.0	43.2
Fairy Forest	174k	82.6	82.5	38.2	34.3	95.5	167.9	63.7	78.3	46.5	54.9
Crytek Sponza	262k	199.9	193.0	40.6	42.2	159.4	172.7	84.4	87.0	36.3	34.4
Conference	282k	122.3	117.5	37.9	44.2	274.1	303.4	134.0	150.4	63.6	70.2
Happy Buddha	1,087k	194.9	172.1	72.1	133.5	266.9	332.6	291.5	335.4	252.5	288.8
Blade	1,765k	222.6	201.7	91.8	194.5	266.5	323.8	326.9	365.4	243.9	272.3
Soda Hall	2,169k	212.9	203.9	93.8	249.2	279.5	343.5	299.1	324.2	88.3	99.8
Hairball	2,880k	1352.4	1145.1	151.7	323.7	38.8	53.6	36.8	47.3	30.4	39.5
San Miguel	7,880k	215.4	194.0	229.4	951.0	35.9	60.3	25.0	33.3	13.0	18.2
Power Plant	12,748k	171.3	123.4	305.6	1501.1	26.3	55.6	77.1	122.9	9.1	13.7
average	-	-	-	102.7	320.4	84.2	129.7	80.7	100.3	31.9	40.2

Table 1: *Results for HLBVH with $k = 4$ compared to our SAH BVH building algorithm. For the primary rays 1M rays are shot while for the Ambient Occlusion and Diffuse rays 7M rays are shot. The average MRays/s are computed from averaged ray tracing times.*

Scene	C_1		C_4		C_M	
	N_r [*MRays*]	R [−]	N_r [*MRays*]	R [−]	N_r [*MRays*]	R [−]
Sponza	+	1.19	10.08	0.95	2.12	1.42
Sibenik Cathedral	+	1.00	+	1.03	1.56	1.74
Fairy Forest	+	1.24	+	1.17	4.23	1.26
Crytek Sponza	74.99	0.98	-	0.95	2.76	1.40
Conference	+	1.18	5.61	1.10	3.37	1.65
Happy Buddha	+	1.07	141.17	1.13	150.19	1.18
Blade	+	1.02	235.86	1.10	230.27	1.13
Soda Hall	43.54	1.17	120.18	1.12	22.90	1.76
Hairball	+	1.28	28.67	1.29	37.02	1.30
San Miguel	26.12	1.43	36.13	1.39	18.07	1.80
Power Plant	56.04	1.29	35.54	1.49	17.31	2.03

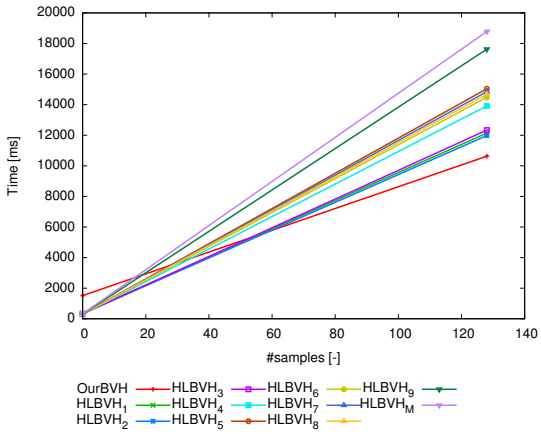


Table 3 & Figure 6: *The table shows the crossover of the time to image including build times for our BVH method and the HLBVH methods with various number of bits used for the HLBVH. The crossover point is computed as an intersection of lines given by two points: (0, T_b) and (128, T_i), where the first coordinate is the number of diffuse samples per pixel (for 1MPixel image), T_b is the build time of the BVH and T_i is the time to image (build time + primary rays time + diffuse rays time). C_1 gives the crossover point with $HLBVH_1$, similarly C_4 gives the crossover point with the method $k = 4$ and C_M is the crossover point with the fully median built $HLBVH$. N_r columns give the crossover points; the number of rays for which both methods have the same time to image. R columns give the ratio of traversal times for HLBVH and our method. The + sign is for cases where the time to image for our method is always lower than for the HLBVH method and the − sign vice versa. The right figure shows our BVH method compared to all of the HLBVH methods on the Power Plant model.*

Stat	$HLBVH_1$	$HLBVH_2$	$HLBVH_3$	$HLBVH_4$	$HLBVH_5$	$HLBVH_6$	$HLBVH_7$	$HLBVH_8$	$HLBVH_9$	$HLBVH_M$
SAH cost [-]	163.3	161.6	165.1	171.3	176.2	179.8	181.5	181.2	183.3	184.4
Build GPU [ms]	310.2	283.0	259.9	252.7	252.9	254.7	253.6	262.0	261.0	239.0
Build total [ms]	382.0	345.7	317.2	305.6	303.5	303.3	299.1	307.0	303.2	331.2
Primary [MRays/s]	36.5	40.7	29.5	26.3	23.9	26.7	22.4	23.1	14.9	13.9
Ambient Occlusion [MRays/s]	95.1	93.7	85.4	77.1	73.6	70.6	70.8	68.4	69.0	71.1
Diffuse [MRays/s]	10.5	10.7	10.4	9.1	8.5	8.8	8.6	8.7	7.2	6.8

Table 4: *Comparison of the HLBVH method based on the value of k for the Power Plant model. For the primary rays 1M rays are shot while for the Ambient Occlusion and Diffuse rays 7M rays are shot.*

the node splitting. The constants used in the formula deciding when to stop the subdivision and invoke the intersection tasks defined in Equation 2 were set as follows: $C_{INTERS} = 1$, $C_{SORT} = 80$. An intersection task is also invoked when the number of rays drops below 32 or the number of triangles drops below 16.

The collision detection rays were chosen because they have desirable properties for our parallel divide-and-conquer ray tracer: they are relatively short compared to the scene diagonal and relatively few rays are sufficient to compute the solution. Keeping the ray count low is important since the algorithm also needs to partition the rays, and global memory accesses are not cached in L1 in current generations of GPU, leading to excessive memory bus traffic. The ray length influences the dependencies between individual tasks. Since the introduction of dependent task limits the available parallelism and there are more dependencies with increasing depth of tasks the ray length directly influences the amount of parallelism and, thus, the performance of the method. For many infinitely long rays such as the diffuse rays the method is actually slower than when computed on the CPU.

7. Conclusion

We have proposed a novel method for massively parallel processing in the context of hierarchical algorithms dealing with 3D geometrical data. Our method runs entirely on the GPU and requires no management of the computation from the CPU side. We propose a methodology of subdividing a given hierarchical algorithm into tasks, phases, steps, and work chunks in order to map the algorithm to the parallel framework. We show two applications of our method: construction of the BVH and divide-and-conquer ray tracing on the GPU. We evaluated two proof of concept applications, which indicate that our approach has a good potential for massive parallelization of complex hierarchical problems.

In the future we would like to apply our method to other problems in computer graphics such as SBVH/kd-tree construction or GPU path tracing.

Acknowledgement

We would like to thank the contributors of the scenes used in our paper, Prof. C. Sequin for Soda Hall model, the University of North Carolina for the Power Plant model, Marko Dabrovic for the Sponza and Sibenik models, Ingo Wald for Fairy Forest, Greg Ward for the Conference model, Samuli Laine and Tero Karras for Hairball model, and Stanford repository for the other models. We would also like to thank Tero Karras, Timo Aila, and Samuli Laine for releasing their GPU ray tracing framework. Our research was supported by the Czech Science Foundation under research programs P202/11/1883 (Argie), P202/12/2413 (Opalis) and P202/10/1435.

References

[Áfr12] ÁFRA A. T.: Incoherent ray tracing without acceleration structures. In *Eurographics (Short Papers)* (2012), Andújar C., Puppo E., (Eds.), Eurographics Association, pp. 97–100. 1, 7

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG'09, ACM, pp. 145–149. 2, 3

[BH09] BITTNER J., HAVRAN V.: RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures. In *25th Spring Conference on Computer Graphics (SCCG 2009)* (Budmerice, Slovakia, May 2009), Hauser H., (Ed.), pp. 61–67. 8

[CKL*10] CHOI B., KOMURAVELLI R., LU V., SUNG H., BOCCHINO R. L., ADVE S. V., HART J. C.: Parallel SAH k-D tree construction. In *Proceedings of the Conference on High Performance Graphics* (2010), HPG'10, Eurographics, pp. 77–86. 2

[CT08] CEDERMAN D., TSIGAS P.: On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (2008), GH '08, Eurographics, pp. 57–64. 2

[CVKG10] CHEN L., VILLA O., KRISHNAMOORTHY S., GAO G. R.: Dynamic Load Balancing on Single- and Multi-GPU systems. In *Proceeding of IPDPS'10 conference* (2010), pp. 1–12. 2

[DPS10] DANILEWSKI P., POPOV S., SLUSALLEK P.: *Binned SAH Kd-Tree Construction on a GPU*. Tech. rep., Computer Graphics Group, Saarland University, June 2010. 2

[GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D.: Simpler and Faster HLBVH with Work Queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (New York, NY, USA, 2011), HPG'11, ACM, pp. 59–64. 2, 6, 9

[HHS06] HAVRAN V., HERZOG R., SEIDEL H.-P.: On the Fast Construction of Spatial Data Structures for Ray Tracing. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006* (Sept. 2006), pp. 71–80. 6

[HSO07] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, Nguyen H., (Ed.). Addison Wesley, August 2007, ch. 39, pp. 851–876. 7

[HSZ*11] HOU Q., SUN X., ZHOU K., LAUTERBACH C., MANOCHA D.: Memory-Scalable GPU Spatial Hierarchy Construction. *IEEE Transactions on Visualization and Computer Graphics 17*, 4 (April 2011), 466–474. 2

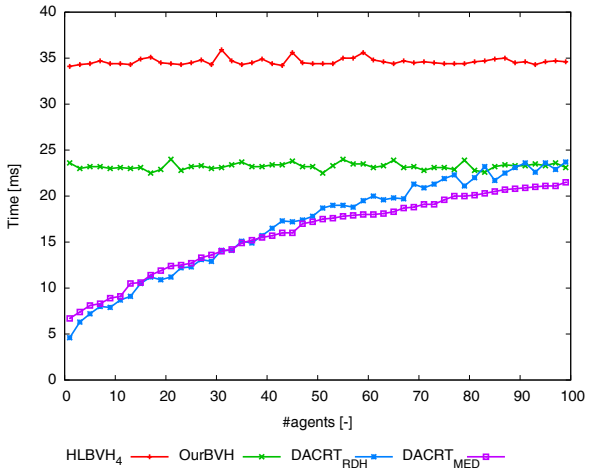
[KBS11] KALOJANOV J., BILLETER M., SLUSALLEK P.: Two-Level Grids for Ray Tracing on GPUs. *Computer Graphics Forum 30*, 2 (2011), 307–314. 2

[KS09] KALOJANOV J., SLUSALLEK P.: A Parallel Algorithm for Construction of Uniform Grids. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG'09, ACM, pp. 23–28. 2

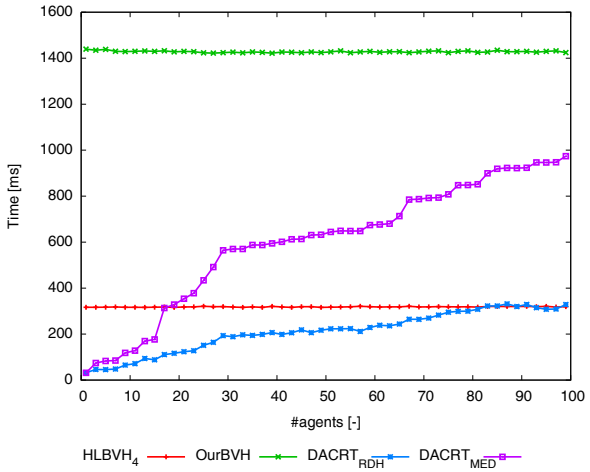
[Lee10] LEE V. E. A.: Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual international symposium on Computer architecture* (New York, NY, USA, 2010), ISCA '10, ACM, pp. 451–460. 2

[LGS*08] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH Construction on GPUs. *Computer Graphics Forum 28*, 2 (2008), 375–384. 2

[Mor11] MORA B.: Naive ray-tracing: A divide-and-conquer approach. *ACM Trans. Graph. 30*, 5 (Oct. 2011), 117:1–117:12. 1, 2, 7



(a) *Sponza*



(b) *Power Plant*

Figure 7: *Comparison of various methods for computing collision detection rays on the Sponza and Power Plant models in dependence on the number of moving agents. The computation times are for fully computing (build+trace) one batch of rays and are averaged over 20 batches simulating agent's movement in the scene. Each agent is checked for collision with the scene using 128 rays uniformly shot into the sphere at the agents' positions and the ray length is set as the distance between current position and the position in the next step.*

[NBGS08] NICKOLLS J., BUCK I., GARLAND M., SKADRON K.: Scalable Parallel Programming with CUDA. *Queue 6*, 2 (Mar. 2008), 40–53. 2, 3, 9

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray Tracing on Programmable Graphics Hardware. *ACM Trans. Graph. 21*, 3 (July 2002), 703–712. 2

[PL10] PANTALEONI J., LUEBKE D.: HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry. In *Proceedings of the Conference on High Performance Graphics* (2010), HPG'10, Eurographics, pp. 87–95. 2

[SBU11] SOPIN D., BOGOLEPOV D., ULYANOV D.: Real-Time SAH BVH Construction for Ray Tracing Dynamic Scenes. In *21th International Conference on Computer Graphics and Vision (GraphiCon)* (2011), pp. 74–77. 2

[SGPT11] SUNDELL H., GIDENSTAM A., PAPATRIANTAFILOU M., TSIGAS P.: A Lock-Free Algorithm for Concurrent Bags. In *SPAA* (2011), Rajaraman R., Meyer auf der Heide F., (Eds.), ACM, pp. 335–344. 2

[SGS10] STONE J. E., GOHARA D., SHI G.: OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test 12*, 3 (May 2010), 66–73. 2

[SKK*12] STEINBERGER M., KAINZ B., KERBL B., HAUSWIESNER S., KENZEL M., SCHMALSTIEG D.: Softshell: Dynamic Scheduling on GPUs. *ACM Trans. Graph. 31*, 6 (Nov. 2012), 161:1–161:11. 2

[SSK07] SHEVTSOV M., SOUPIKOV A., KAPUSTIN A.: Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of

Dynamic Scenes. *Computer Graphics Forum 26*, 3 (2007), 395–404. 2

[TPO10] TZENG S., PATNEY A., OWENS J. D.: Task Management for Irregular-Parallel Workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics* (2010), HPG'10, Eurographics, pp. 29–37. 2

[TZ01] TSIGAS P., ZHANG Y.: A Simple, Fast and Scalable Non-Blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems. In *Proc. of the 13th ACM Symposium on Parallel Algorithms and Architectures* (2001), ACM, pp. 134–143. 2

[Wal07] WALD I.: On fast Construction of SAH-based Bounding Volume Hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (Washington, DC, USA, 2007), RT '07, IEEE Computer Society, pp. 33–40. 2

[Wal12] WALD I.: Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture. *IEEE Transactions on Visualization and Computer Graphics 18*, 1 (January 2012), 47–57. 2

[WH06] WALD I., HAVRAN V.: On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006* (Sept. 2006), pp. 61–69. 2

[WK09] WÄCHTER C., KELLER A.: Efficient Ray Tracing Without Acceleration Data Structure, 2009. U.S. Patent Applications Publication No. US2009/02225081 A1. 1, 2, 7

[WMG*09] WALD I., MARK W. R., GÜNTHER J., BOULOS S., IZE T., HUNT W., PARKER S. G., SHIRLEY P.: State of the Art in Ray Tracing Animated Scenes. *Computer Graphics Forum 28*, 6 (2009), 1691–1722. 2

[Woo04] WOOP S.: *A Ray Tracing Hardware Architecture for Dynamic Scenes*. Tech. rep., Saarland University, 2004. 9

[WZL11] WU Z., ZHAO F., LIU X.: SAH KD-tree Construction on GPU. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (New York, NY, USA, 2011), HPG'11, ACM, pp. 71–78. 2

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time KD-tree Construction on Graphics Hardware. *ACM Trans. Graph. 27*, 5 (Dec. 2008), 126:1–126:11. 2, 6