Technical Report Series of DCGI, Volume 2, Year 2012

Department of Computer Graphics and Interaction
Czech Technical University in Prague, CZ
Faculty of Electrical Engineering
Website: http://dcgi.fel.cvut.cz

Massively Parallel Hierarchical Scene Sorting with Applications in Rendering

Marek Vinkler, Michal Hapala,
Jiří Bittner, Vlastimil Havran

CS–TR–DCGI–2012–1 August 2012

Authors' Addresses

Marek Vinkler
Department of Computer Systems and Communications
Faculty of Informatics
Masaryk University
Czech Republic
E-mail: xvinkl AT mail DOT muni DOT cz
Web: `http://is.muni.cz/osoba/xvinkl`

Michal Hapala
Department of Computer Graphics and Interaction
Czech Technical University in Prague, FEE
Czech Republic

Jiří Bittner
Department of Computer Graphics and Interaction
Czech Technical University in Prague, FEE
Czech Republic

Vlastimil Havran
Department of Computer Graphics and Interaction
Czech Technical University in Prague, FEE
Czech Republic
E-mail: havran AT fel DOT cvut DOT cz

Funding Acknowledgements

Our research was supported by the Czech Science Foundation under research programs P202/11/1883 (Argie) and P202/12/2413 (Opalis), and the Grant Agency of the Czech Technical University in Prague, grant No. SGS10/289/OHK3/3T/13, supported by Ministry of Education of the Czech Republic.

Technical Report Series of DCGI, available at `http://dcgi.fel.cvut.cz/techreps`, Department of Computer Graphics and Interaction, Czech Technical University in Prague, Faculty of Electrical Engineering, Czech Republic.

ISSN 1805-6180

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 organized using a specialized work pool in which different blocks of processing units solve different functional blocks. We present an application of the proposed approach to two methods used in ray tracing: construction of the bounding volume hierarchies and the recently introduced divide-and-conquer ray tracing on the GPU. 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.

Keywords

GPU, ray tracing, bounding volume hierarchy (BVH), parallel sorting

Contents

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] 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. 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 as the first proposed algorithm 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: BVH construction and divide-and-conquer ray tracing. The results indicate that the implementations based on our method perform comparable to existing solutions, while they still provide a number of possibilities for performance improvement.

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] propose a simple non-blocking concurrent queue for FIFO processing for shared memory multiprocessor systems that utilizes compare-and-swap operations (CAS). With the availability of atomic operations on GPUs Cederman and Tsigas [CT08] compare four approaches for dynamic load balancing on GPUs and conclude that blocking queues perform the worst. Tzang et al. [TPO10] study efficiency of load balancing methods for irregular workloads on the GPU and they conclude that task-stealing and task-donation are the most efficient. Chen et al. [CVKG10] propose a task-based dynamic load balancing approach for single and multi GPU computer systems. They use a persistent kernel running on a *device(s)* (a GPU or several GPUs) where the task queue is generated on a *host* (CPU). The most recent work by Sundell et al. [SGPT11] propose a lock-free algorithm for distributing work on concurrent hardware without the restriction of work producers and consumers, this being the closest to our approach presented here. We also want to point out a recent paper by Lee et al. [Lee10] that rigorously analyze the performance of an NVIDIA GTX280 and an Intel Core i7 960 processor for fourteen different computational problems with carefully optimized implementations. They show 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 restrict our review here to the works concerning 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 method was followed by other papers which are surveyed by Wald et al. [WMG$^+$09]. The newer papers utilize specialized languages for the GPU like the CUDA API [NBGS08]. 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 and OpenCL.

Kd-trees. Zhou et al. [ZHWG08] present 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] study different possibilities to organize the traversal code on GPU architectures to get the highest performance. Pantaleoni and Luebke [PL10] present 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 present a hybrid algorithm with a two-level BVH, where top layers are built with an exact algorithm with a surface area heuristics (SAH) and bottom levels with a Morton curve based algorithm. Garanzha et al. [GPM11] simplify the HLBVH algorithm using binary search and work queues. They achieve both memory savings and lower build times for breadth-first-search tree layout using a parallel binning approach proposed by Wald [Wal07] for a multi-core CPU. Wald describes a parallel version of a BVH based builder with SAH using binning on a many-core architecture (MIC) [Wal12]. Building on their work, 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] present a parallel algorithm for building uniform grids, followed by another paper by Kalojanov et al. [KBS11] for hierarchical grids.

Without data structure approaches. Keller and Wächter [WK09] present an approach for ray tracing which simultaneously subdivides rays and triangles and can be computed without explicit spatial data structures that builds up on the approach of Bittner and Havran for kd-trees [BH07, BH09]. A similar approach was independently developed and implemented on a single-core CPU with SSE instructions by Mora [Mor11]. Mora also utilizes bounding cones for primary rays to improve the performance.

3 Hierarchical Scene Sorting

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. Note that 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 depends 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 prefix sum (PPS) computation used for subdividing scene primitives into subsets, 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 thus the smallest unit of work in our method. Note that while the task, 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). Typically the work chunk consists of 32 data items and thus 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 later in the paper.

Apart from the above defined terms we recall the basic terminology associated with CUDA: *kernel* is a program executed on the CUDA device, *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 information about its data interval (e.g. interval in the triangle index array) and the state of the task. The task also holds additional 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 several (at least two) subsets using the index array. After these phases the algorithm continues with the subsets of 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 computing bounding boxes. An example of the computational phases for the BVH construction is illustrated in Figure 3.1.

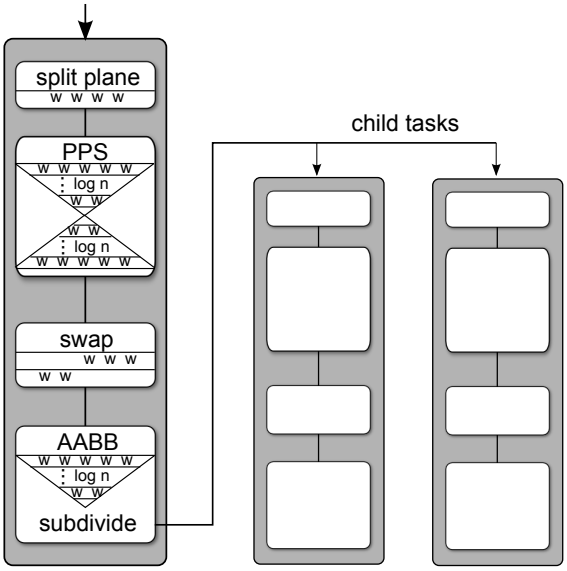


Figure 3.1: Overview of the task and its phases in an algorithm for BVH construction. The figure also shows the warps cooperating on solving the particular steps of the task phases. Symbol 'w' stands for a single warp.

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 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 parallelism). For example we want to determine a splitting plane for one node in the hierarchy using a particular 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 thus the data ranges of the associated child tasks will 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

Our algorithm is built on the concept of persistent warps [AL09]. We launch a single kernel with as many warps as can be run simultaneously on the GPU. There is no 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 scene geometry (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, as our algorithm is parallel, warps also participate in computation of the same task. Each warp takes a work chunk from an arbitrary active task based on the current distribution of work in the task pool. The algorithm can be described with a simple pseudo-code in listing 1.

```
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 work chunk of the task;
            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.

The algorithm shows that the warps are constantly searching for arbitrary work chunks that they can handle. When they succeed in acquiring 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 3.2. It consists of geometry data (for example triangles or rays), index array that allows to swap only indices instead of geometric data, and the task pool consisting of two separate arrays.

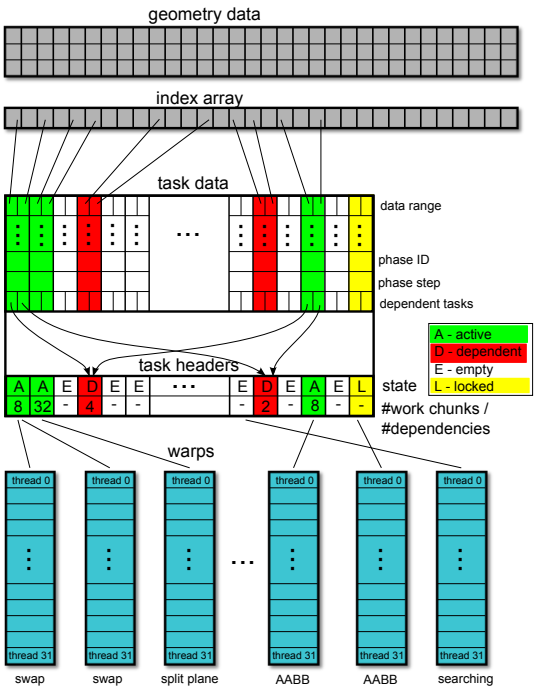


Figure 3.2: Overview of the main data structures used in the proposed method.

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 that represents the combination of a state and a counter. This gives a very small memory footprint that can easily fit into the cache on modern GPU architectures.

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 allows to use efficient mechanism to acquire work chunks and handle task dependencies that will be described below in this section.

The task can exhibit 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 given 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. Each thread in the warp then reads the state of one consecutive entry from the task header array. As multiple entries can be active, we have 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 i on the states of the entries within each warp and choose the i-th active task based on the warp ID. When the active tasks are chosen, the warps atomically decrement the task's 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 or acquiring the work chunk is not negligible. We use the following function for determining the number of work chunks retrieved by the warp:

$$N_w = max(\lceil \frac{S}{W} \rceil, K), \tag{3.1}$$

where S is the sum of work chunks in the task pool entries loaded by the warp searching for an active task, W is the number of warps launched, and K is a constant preventing the retrieval of too few work chunks for a given warp. The first term $\lceil \frac{S}{W} \rceil$ 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 deal with smaller amount of data. We have tested experimentally that $K = 14$ proved 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. 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 acquiring the task, i.e. the value returned by the atomic decrement was not positive (the value before the decrement operation is returned by atomic instruction), the same process is repeated.

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 for each task. 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.

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. This operation also unlocks it. Note that a memory fence operation must be issued before the task is unlocked.

Minimizing pool overhead. We use two improvements that accelerate the computation of tasks. They are both targeting steps with little parallelism. First, when some task 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 the global memory synchronization through the task pool (K is the constant used in Eq. 3.1). This is

often the case with the reduction in the *PPS* or *AABB* phases.

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. 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 can define four different phases:

1. *SplitPlane*: Splitting plane computation (spatial median or cost model with SAH).
2. *PPS*: Parallel prefix sum computation of the number of triangles right of the splitting plane.
3. *Swap*: Sorting of a triangle interval into the left and right sub-intervals.
4. *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 (*PPS*, *AABB*) and thus require a logarithmic number of steps to complete. The illustration of the phases is shown in Figure 3.1. Note that the figure also shows the 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. 3.1. 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 based one. Both of these cycle the splitting plane in the round-robin fashion, where for the first task the longest axis is chosen. Nevertheless, any strategy for choosing the planes is possible. For the SAH strategy we select 32 candidate planes and evaluate their cost using the SAH in parallel. Each warp processes up to 64 triangles 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 last warp that have finished its work loads these data from the global memory and chooses the best splitting plane. As there are exactly 32 candidates this works 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.

PPS. In this phase we determine the position of each triangle with respect to the selected splitting plane and perform parallel prefix sum to determine the target index of each triangle in the sorted array. When a centroid of the triangle is classified as lying on the left or straddling the splitting plane, its index is marked by zero, otherwise it is marked by one. The *PPS* then sums the number of triangles to the right of the plane for all smaller indices in the given range of triangles. We use the algorithm proposed by Harris [HSO07] but with our own implementation that allows for persistent threads. As the result of this operation the number of triangles to the left and straddling the splitting plane is known and it is used as a pivot index for the subsequent swap phase.

Swap. This phase consists of two steps that work with the triangle index array. It aims to swap the triangles so that they are sorted with respect to the splitting plane. In the first step the triangles with indices larger than the pivot index are processed. For the sorting we need a separate index array of the same size as the original triangle index array. When a triangle marked with zero is found in this interval its index is copied to the separate index array at the position given by the result of the *PPS* phase for this triangle. In the second step the triangles with indices smaller than the pivot index are processed. When a triangle marked with one is found in this interval it is swapped with the item in the separate index array at the corresponding position. The result of the swap phase is then stored in the original index array.

AABB. Segmented parallel reduction is computed on the interval in the triangle index array. The bounding boxes for the triangle intervals corresponding to left and right triangle subsets are computed by reusing the arrays allocated for the phase *Swap* described above. The computation requires $\log_2(\#tris)$ steps. This phase is only needed for the median splitting as the SAH evaluation gives us the exact bounding boxes. The complexity of this phase is higher than for a serial algorithm because the tournament-like parallel algorithm is used.

4.2 Handling Tasks

Note that since the algorithm subdivides the current data range into disjoint subsets there are no data dependencies among different tasks and thus 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 *Swap* 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 or a maximum depth is reached), 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] and Keller and Wächter [WK09]. 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 intervals 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-intervals at which an intersection of rays and triangles can happen. As we have three intervals for both rays and triangles we obtain nine pairs of different ray/triangle intervals to process. Out of the nine pairs for two pairs of intervals no intersection can happen: (1) triangles lying left from the splitting plane and rays lying right of the plane and (2) triangles lying right from the plane and rays lying left of the plane. For the remaining seven intervals we create child tasks and proceed with the computation. The subdivision into child tasks is illustrated in Figure 5.1.

There are clear computation dependencies among the child tasks shown in Figure 5.1: 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

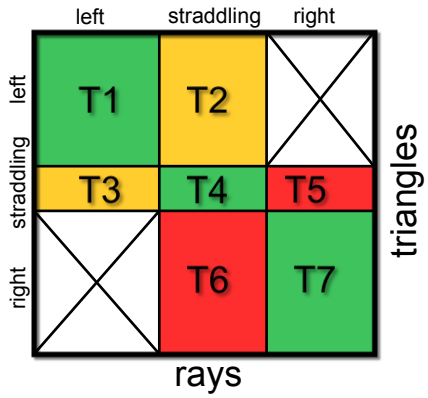


Figure 5.1: Matrix representing a subdivision of the task into its child tasks for the divide-and-conquer ray tracing.

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 six phases that are computed sequentially (see Figure 5.2). Some of these phases are only a minor modification of the phases described for the BVH construction in Section 4.1. Below we describe these phases in more detail.

SplitPlane. This phase is similar to the phase used for the BVH construction. The difference is that we also evaluate the position of rays with respect to the particular candidate splitting planes. 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 RDH cost model by Bittner and Havran [BH09]. The termination criteria are derived using this model, the intersection task is created when $\#tris \cdot \#rays \cdot C_{INTERS} < (\#tris + \#rays) \cdot C_{SORT}$, 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 triangle planes that cover all three axes. The number of candidates for each axis is proportional to the node's extent in that 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

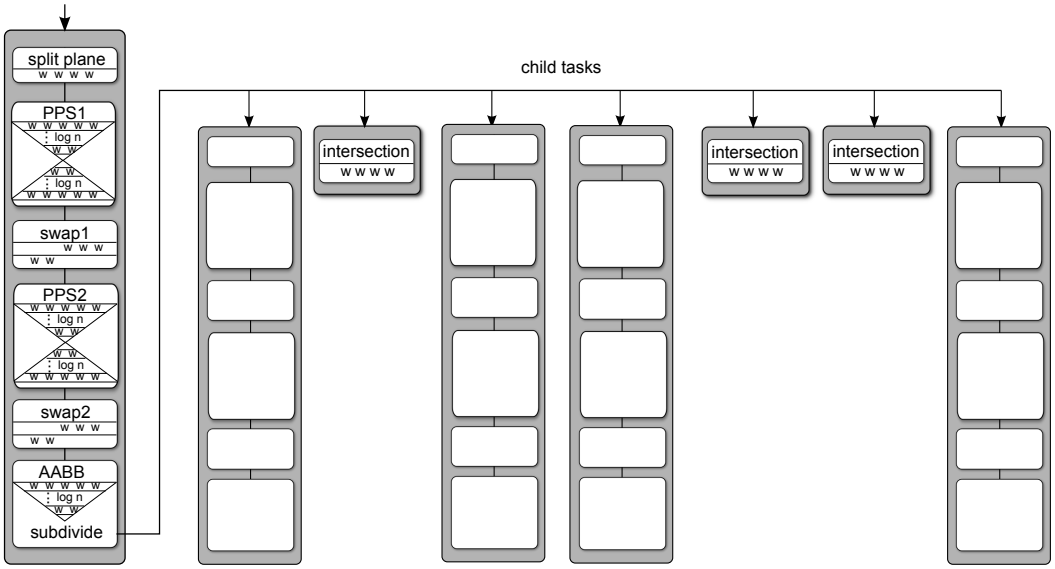


Figure 5.2: Overview of the task phases and steps for the divide-and-conquer ray tracing algorithm.

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.

PPS1. The parallel prefix sum is computed in parallel on both rays and triangles. 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 and the subsequent swap. Other than that this phase is exactly the same as the *PPS* phase in building BVH. The computation thus requires $2 \cdot \log_2(max(\#tris, \#rays))$ steps. As the result of this operation the number of the rays and triangles lying to the left or straddling the splitting plane and the number of rays and triangles to the right of the splitting plane is known.

Swap1. The swap phase reorganizes the triangle and ray array so that they are sorted according to the splitting plane. Similarly to the case of the BVH construction this phase requires two steps and a separate auxiliary index array. In the first step the rays and triangles lying at array indices to the right of the index of the splitting plane are processed. When a ray or triangle is classified as lying to the left or straddling the splitting plane, its index is copied to the separate index array. In the second step the array indices left of the index of the splitting plane are processed. When a ray or triangle classified as lying to the right of the splitting plane is found, it is swapped with the item in the separate index array. In both phases the indices where to write the result to and where to read from are computed from the prefix sum information for the processed ray or triangle.

PPS2. This phase is done the same way as *PPS1* but for two intervals in parallel. In the left interval created during the first split the prefix sum for the rays and triangles straddling the splitting plane is computed and in the right interval the prefix sum for clipped rays is computed.

Swap2. Similarly to *PPS2* two intervals are reorganized in parallel. After this phase the task's ray interval is fully sorted into left, straddling, right, and clipped rays and triangle interval into left, straddling, and right intervals 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 thus they cannot be processed fully independently. Certain groups of child tasks are, however, independent. In general 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 tasks need to inherit the activation pointers of the task being subdivided as some other tasks may depend on it.

As already discussed in Section 5.1, 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. Note that 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 decrements the task header array entry for the dependent tasks by the number of its children in the last dependency group minus one for itself (recall that the counter of dependencies is represented as a negative integer). If the task is a splitting task and has no children or it is an intersection task, the dependency counter in the dependent task may reach 0 after this operation. In this case the active task activates the dependent task by setting its entry in the header array to the number of its unfinished work chunks.

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 intervals (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. For testing we have used a PC with Intel Core i7-2600, 16GB of RAM and NVIDIA GeForce GTX 580 running on Windows 7 64-bit. We have used two types of scenes for testing, individual objects and more complex scenes with triangles sparsely distributed in space. The images for the test scenes are shown in Figure 6.1.

Figure 6.1: Snapshots for more complex architectural models: Conference, Sponza, Sibenik Cathedral, Powerplant section 9, Powerplant section 16, Fairy Forest, Sodahall, and for single geometric object models: Happy Buddha, Dragon, Armadillo, and Blade.

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 corresponding to the images in Figure 6.1, incoherent rays corresponding to ambient occlusion (AO) rays shot from the hit points of the primary rays (seven AO rays per primary ray), and random rays distributed uniformly in the scene bounding box. Although our algorithm is capable of using the SAH termination criteria, for easier comparison to other methods we have chosen to use simple termination criteria. A leaf is created when the number of triangles drops below a given threshold. The results are shown in Table 6.1 for the spatial median splits and in Table 6.2 for the SAH splits.

The build times range from 6.4ms (Sponza, spatial median, 32 triangles per leaf) to 381.6ms (Sodahall, SAH, 8 triangles per leaf). As expected the build times depend on the termination criteria – the deeper the tree, the more time it takes to construct it. For the spatial median we can mostly observe an increase of time complexity when building a deeper BVH (e.g. for the Conference with 32 triangles per leaf the build takes 17.5ms, whereas for 8 triangles per leaf it takes 34.2ms). Interestingly, for the SAH method the increase of build times when constructing a deeper tree is not that significant (e.g., for the Conference with 32 triangles per leaf the SAH build takes 37.4ms, whereas for 8 triangles per leaf it takes 51.2ms).

Figures 6.2 and 6.3 show the performance of our spatial median and SAH based BVH builders on six of our test scenes. The performance in MTris/s depends on the actual scene structure and interestingly it is higher on larger scenes. We expect that this is because for these scenes there is more exploitable parallelism present in the computation, which leads to better GPU utilization.

The ray tracing speed varies between 14.1 MRays/s (Happy Buddha, random rays, 32 triangles per leaf) and 1049.2 MRays/s (Powerplant section 9, ambient occlusion rays, 8 triangles per

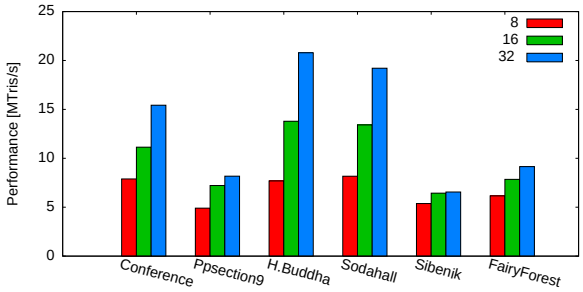


Figure 6.2: Performance in MTris/s for building the BVH with spatial median for varying termination criteria.

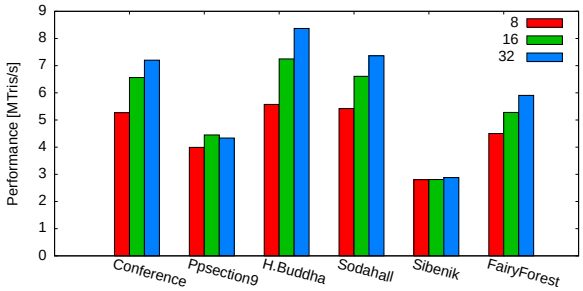


Figure 6.3: Performance in MTris/s for building the BVH with SAH for varying termination criteria.

leaf) for the spatial median splits. For the SAH splits the ray tracing speed ranges between 14.8 MRays/s (Happy Buddha, random rays, 32 triangles per leaf) and 1276.9 MRays/s (Powerplant section 9, ambient occlusion rays, 8 triangles per leaf). Constructing a deeper tree increases the ray tracing speed, and this increase is significant for most tested cases (e.g. Conference, spatial median, random rays, 22.2 MRays/s for 32 triangles per leaf and 41.6 MRays/s for 8 triangles per leaf). As expected the SAH based BVH provides higher ray tracing speed than the spatial median BVH while the difference is more significant for the architectural scenes such as Conference, Powerplant section 9 and 16, or Sodahall. The optimal choice of the building algorithm and the termination criteria depends on the scene as well as the target rendering algorithm and the actual rays which are cast for each frame. The more rays are cast per frame the more it pays off to construct a deeper tree using the more advanced SAH based splits.

The BVHs build up with our algorithm are comparable to other state-of-the-art methods such as [HSZ+11, PL10], but take more time than the fastest published method of Garanzha et al. [GPM11]. On the other hand, our method allows to easily implement a complex BVH building algorithm (such as SAH build for all inner nodes) and the resulting quality of the tree should pay off when more rays are being traced.

6.2 Divide-And-Conquer Ray Tracing

Second, we have tested the divide-and-conquer ray tracing described in Section 5 using the same scenes and the same types of rays as for the BVH evaluation. The results for the spatial median subdivision and for the cost model based on the RDH are given in Table 6.3. The constants

		Build [ms]			Trace performance [MRays/s]								
					Primary			Ambient Occlusion			Random		
Scene	#tris	8	16	32	8	16	32	8	16	32	8	16	32
Conference	283k	34.2	24.2	17.5	87.1	80.6	59.3	69.8	59.6	37.7	41.6	33.3	22.2
Sponza	76k	12.9	9.0	6.4	83.2	72.7	54.6	38.2	28.6	17.4	32.6	23.8	14.4
Ppsection 9	122k	23.7	16.1	14.2	156.3	147.9	112.6	1049.2	966.5	756.2	43.9	40.6	29.3
Ppsection 16	366k	46.7	31.4	22.6	36.1	33.7	26.8	115.1	108.6	89.6	44.5	36.7	25.1
H. Buddha	1,087k	135.0	75.2	49.9	216.5	161.8	109.7	243.6	185.1	121.9	30.4	22.2	14.1
Dragon	871k	104.0	55.8	39.4	187.8	139.2	94.4	175.6	136.0	90.4	32.5	23.9	15.2
Armadillo	307k	35.5	22.6	15.2	205.4	153.1	107.3	202.3	156.5	105.2	45.6	32.7	20.3
Blade	1.765k	207.1	120.4	77.8	207.8	157.9	106.6	340.6	275.0	190.8	37.8	28.1	17.9
Sodahall	2,169k	253.2	154.2	107.7	68.6	60.5	46.3	141.3	128.9	104.7	42.4	31.4	20.0
Sibenik	80k	14.3	11.9	11.7	61.9	56.9	41.8	40.1	31.2	19.3	61.4	46.9	31.4
Fairy Forest	174k	27.0	21.2	18.1	101.7	87.8	67.6	44.2	34.6	23.6	59.5	45.8	31.5
average	-	81.2	49.3	34.6	128.4	104.7	75.2	223.6	191.9	141.5	42.9	33.2	21.9

Table 6.1: Results for the spatial median BVH building algorithm. Build time and trace performance for test scenes for the leaf termination criteria either 8, 16, or 32 for the maximum number of triangles in a node. The image resolution (the number of primary rays) is 1024×1024 pixels. The number of ambient occlusion rays is 7 times higher than the number of primary rays, the number of random rays is the same as the number of primary rays.

		Build [ms]			Trace performance [MRays/s]								
					Primary			Ambient Occlusion			Random		
Scene	#tris	8	16	32	8	16	32	8	16	32	8	16	32
Conference	283k	51.2	41.1	37.4	143.4	133.2	91.4	98.6	84.1	52.1	54.6	43.5	27.8
Sponza	76k	21.8	19.6	19.3	132.6	116.4	88.1	64.8	50.1	30.4	50.5	36.5	22.6
Ppsection 9	122k	29.1	26.1	26.8	249.0	229.6	187.1	1276.9	1111.3	963.5	58.9	55.8	48.1
Ppsection 16	366k	70.2	55.2	48.4	54.2	51.6	41.6	172.2	164.7	139.0	65.7	59.7	43.5
H. Buddha	1,087k	186.2	143.1	124.0	230.0	172.8	117.2	260.8	197.3	128.2	32.5	23.6	14.8
Dragon	871k	147.3	115.5	100.8	208.2	156.4	105.0	189.1	145.7	96.4	35.0	25.5	16.1
Armadillo	307k	52.9	42.3	37.4	218.7	161.8	112.3	216.4	167.2	111.5	48.5	34.7	21.5
Blade	1.765k	303.4	234.4	205.1	228.0	171.0	111.9	358.1	285.3	193.4	39.9	29.8	18.9
Sodahall	2,169k	381.6	313.1	280.9	91.5	79.9	61.2	189.1	170.9	136.6	58.2	43.7	26.9
Sibenik	80k	27.4	27.3	26.6	100.7	88.9	63.5	59.0	44.9	27.9	83.6	63.1	40.8
Fairy Forest	174k	36.9	31.5	28.1	117.7	98.7	72.0	48.4	38.1	24.4	67.0	52.9	33.8
average	-	118.9	95.4	85.0	161.3	132.8	95.6	266.7	223.6	173.1	54.0	42.6	28.6

Table 6.2: Results for SAH-cost based BVH building algorithm with binning. The legend is the same as for Table 6.1.

		Spatial median, performance [MRays/s]						RDH performance, [MRays/s]					
		Primary		Amb. Occlusion		Random		Primary		Amb. Occlusion		Random	
Scene	#tris	512	1024	512	1024	512	1024	512	1024	512	1024	512	1024
Conference	283k	0.757	1.611	1.671	2.690	0.335	0.651	1.132	2.298	2.198	3.674	0.405	0.879
Sponza	76k	0.917	1.878	1.486	2.103	0.417	0.695	0.824	2.173	1.840	2.596	0.401	0.824
Ppsection 9	122k	1.396	2.785	31.120	40.194	0.731	1.392	2.097	4.077	39.932	73.387	1.118	2.359
Ppsection 16	366k	0.208	0.519	4.609	5.241	0.407	0.755	0.278	0.624	6.472	7.007	0.568	1.089
H. Buddha	1,087k	0.688	1.492	0.928	1.776	0.074	0.139	0.988	2.237	1.130	2.306	0.081	0.161
Dragon	871k	0.742	1.614	0.854	1.597	0.085	0.155	0.988	2.250	1.039	2.049	0.098	0.191
Armadillo	307k	0.616	1.168	1.604	2.783	0.187	0.320	0.814	1.322	1.994	3.609	0.215	0.361
Blade	1,765k	0.265	0.554	2.964	6.764	0.050	0.098	0.357	0.650	4.614	9.341	0.055	0.111
Sodahall	2,169k	0.313	0.706	4.471	5.391	0.062	0.132	0.359	0.832	6.499	7.810	0.080	0.176
Sibenik C.	80k	0.449	0.952	1.191	1.789	0.526	0.927	0.507	1.130	1.775	2.596	0.783	1.501
Fairy Forest	174k	0.255	0.566	0.455	0.804	0.286	0.544	0.794	1.555	0.896	1.444	0.504	1.020
average	-	0.601	1.259	4.668	6.467	0.287	0.528	0.831	1.741	6.217	10.529	0.392	0.788

Table 6.3: The results for divide-and-conquer ray tracing with spatial median subdivision and RDH cost-based model for two resolutions and three different kind of rays. The running performance in MRays/s is reported for the resolutions. The number of ambient occlusion rays is 7 times higher than for primary rays, the number of random rays is the same as the number of primary rays.

used in the formula deciding when to stop the subdivision and invoke the intersection tasks defined in Section 5.2 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, however these cases are rather rare as the intersection tasks are mostly invoked due to the above mentioned formula.

The ray tracing performance using the parallel implementation of the divide-and-conquer ray tracing ranges between 0.05 MRays/s (Blade, random rays, 512×512) and 73.4 MRays/s (Powerplant section 9, AO rays, 1024×1024). Note that the reported ray tracing speed also includes the time needed for the triangle sorting phase, which is similar to building an implicit BVH.

The table shows that the cost based RDH method is almost always faster than the spatial median splits and the improvement due to RDH ranges from few percent to a threefold speedup (Fairy Forest, primary rays, 512×512, 0.255 MRays/s versus 0.794 MRays/s).

We have studied the dependence of the rendering performance on the number of rays shot (see Figure 6.4). The figure shows that for the measured range of number of cast rays the ray tracing times grows almost linearly with the number of AO samples per pixel with a constant smaller than 1. This indicates that the method is better suited for tracing larger sets of rays where there is more coherence among the rays which can be exploited by the algorithm and where there is more parallelism.

6.3 Discussion

The resulting ray tracing performance of the divide-and-conquer algorithm is significantly lower than the performance of the BVH ray tracing once the BVH has been built. If we would include the BVH build times in the BVH ray tracing speed, the BVH based ray tracing would still exceed the speed of the divide-and-conquer ray tracing algorithm. This differs from the results of Mora [Mor11] who actually reported speedup of his CPU implementation of the method compared to the CPU BVH based ray tracers. When directly comparing the tracing times for our GPU implementation of the divide-and-conquer ray tracing with Mora's paper we observe either comparable or slower ray tracing performance for our GPU implementation. We expect that this is caused by several reasons: (1) the load balancing is problematic for the divide-

and-conquer ray tracing on many core architectures, (2) we did not yet implement the conic packet optimization used by Mora, (3) the current task subdivision used in our paper might probably be improved to achieve better computational efficiency. The last point addresses the fact that unlike Mora's CPU based method we subdivide the task into seven subtasks. The subdivision increases the amount of exploitable parallelism, but on the other hand it increases the number of sorting and intersection operations evaluated by our algorithm, since some of the created tasks might not be well balanced. This holds especially for tasks corresponding to rays or primitives straddling the splitting plane. A solution to this problem might be a strategy that would adaptively select the task subdivision pattern and the corresponding dependencies. Note, that such adaptive subdivision strategy could be easily handled in the framework proposed in our paper and we plan to study this possibility in future.

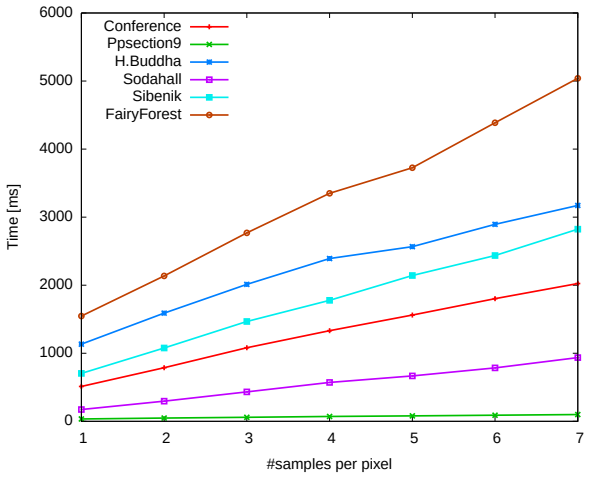


Figure 6.4: Dependence of rendering times on the number of rays shot in the divide-and-conquer ray tracer with RDH cost model on image with resolution of 1024×1024 pixels. Reported data are for ambient occlusion rays with varying number of samples per pixel on selected scenes.

6.4 GPU Utilization

There are several noteworthy facts caused by deviation of our method from the standard CUDA paradigm of launching separate kernels. First, we could use only functions operating on the warp level from libraries such as *CUDPP* or *Thrust*. As most logic in these libraries is done by calling sequences of different kernels we had to reimplement this logic in our framework. Thus, despite our effort, the code is probably not on the same level of optimization as the codes from these libraries. Second, the time of the complete build / trace kernel is reported for our method including the management of the computation, which is done on the CPU and usually not measured for other methods.

To judge the efficiency of our design we ran the CUDA Visual Profiler on the complex Sodahall model. Particularly interesting is the L1 Global Hit Rate, which is quite high on all of our kernels. For the BVH construction about 90% of global memory accesses are cached in L1 and for our divide-and-conquer ray tracer it is over 99%. Quite solid is also the caching behavior for the texture cache unit, which helps in loading of triangle data. The hit rate is about 80%

for all the kernels except for the BVH build with the median splitting strategy, for which it is 42%. We suspect this is because with the other methods the triangle data are loaded more times for each task. For SAH BVH the data are used for candidate plane evaluation, as well as for classification according to the chosen splitting plane. For the subdivision ray caster triangle data are used in the computation of bounding boxes as well as in ray-triangle intersection.

7 Conclusion

We proposed a novel method for massively parallel GPU sorting 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 phases, steps, and work chunks in order to map the algorithm to the proposed parallel framework. We show two applications of our method: construction of the BVH and divide-and-conquer ray tracing on the GPU. We proposed and evaluated two proof of concept applications, which indicate that the proposed approach has a good potential for massive parallelization of complex hierarchical problems.

In future we plan to further improve the particular phases of our applications in order to reduce the computational load and memory accesses required by our current implementation. We would also like to apply our method to other problems in computer graphics such as collision detection or kd-tree construction.

Acknowledgements

We want to thank Tero Karras, Timo Aila, and Samuli Laine for releasing their GPU ray tracing framework. Further, we want to thank all reviewers and our colleagues for their comments. This work was supported under research programs further by the Czech Science Foundation of the Czech Republic under research programs P202/10/1435, P202/12/2413, and P202/11/1883, and the Grant Agency of the Czech Technical University in Prague, grant No. SGS10/289/OHK3/3T/13.

Bibliography

[AL09] T. Aila and S. Laine. Understanding the efficiency of ray traversal on GPUs. In *Proc. of the Conference on High Performance Graphics 2009*, HPG'09, pages 145–149, New York, NY, USA, 2009. ACM.

[BH07] J. Bittner and V. Havran. RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures, September 2007. In poster section of IEEE Interactive Symposium on Ray Tracing.

[BH09] J. Bittner and V. Havran. RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures. In H. Hauser, editor, *25th Spring Conference on Computer Graphics (SCCG 2009)*, pages 61–67, Budmerice, Slovakia, May 2009.

[CKL+10] 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 *Proc. of the Conference on High Performance Graphics*, HPG'10, pages 77–86. Eurographics, 2010.

[CT08] D. Cederman and P. Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 57–64. Eurographics, 2008.

[CVKG10] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao. Dynamic Load Balancing on Single- and Multi-GPU systems. In *Proceeding of IPDPS'10 conference*, pages 1–12, 2010.

[DPS10] P. Danilewski, S. Popov, and P. Slusallek. Binned SAH Kd-Tree Construction on a GPU. Technical report, Computer Graphics Group, Saarland University, June 2010.

[GPM11] K. Garanzha, J. Pantaleoni, and D. McAllister. Simpler and faster HLBVH with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG'11, pages 59–64, New York, NY, USA, 2011. ACM.

[HSO07] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison Wesley, August 2007.

[HSZ+11] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, and D. Manocha. Memory-Scalable GPU Spatial Hierarchy Construction. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):466–474, April 2011.

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

[KS09] J. Kalojanov and P. Slusallek. A parallel algorithm for construction of uniform grids. In *HPG'09: Proceedings of the 1st ACM conference on High Performance Graphics*, pages 23–28, New York, NY, USA, 2009. ACM.

[Lee10] V. e. a. Lee. 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*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[LGS+08] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2008.

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

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

[PBMH02] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *Computer Graphics (SIGGRAPH '02 Proceedings)*, pages 703–712, 2002.

[PL10] J. Pantaleoni and D. Luebke. HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*, HPG'10, pages 87–95. Eurographics, 2010.

[SBU11] D. Sopin, D. Bogolepov, and D. Ulyanov. Real-Time SAH BVH Construction for Ray Tracing Dynamic Scenes. In *21th International Conference on Computer Graphics and Vision (GraphiCon)*, pages 74–77, 2011.

[SGPT11] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A lock-free algorithm for concurrent bags. In R. Rajaraman and F. Meyer auf der Heide, editors, *SPAA*, pages 335–344. ACM, 2011.

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

[TPO10] S. Tzeng, A. Patney, and J. D. Owens. Task management for irregular-parallel workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics*, HPG'10, pages 29–37. Eurographics, 2010.

[TZ01] P. Tsigas and Y. Zhang. 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*, pages 134–143. ACM, 2001.

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

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

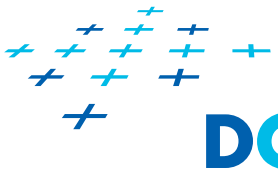
[WH06] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, pages 61–69, September 2006.

[WK09] C. Wächter and A. Keller. Efficient ray tracing without acceleration data structure, 2009. U.S. Patent Applications Publication No. US2009/02225081 A1.

[WMG+09] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the Art in Ray Tracing Animated Scenes. *Computer Graphics Forum*, 28(6):1691–1722, 2009.

[WZL11] Z. Wu, F. Zhao, and X. Liu. SAH KD-tree construction on GPU. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG'11, pages 71–78, New York, NY, USA, 2011. ACM.

[ZHWG08] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time KD-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):126:1–126:11, December 2008.



Technical Report Series of DCGI, Volume 2, Year 2012

Department of Computer Graphics and Interaction
Czech Technical University in Prague, CZ
Faculty of Electrical Engineering
Website: http://dcgi.fel.cvut.cz

Karlovo nám. 13
121 35 Praha 2
Czech Republic

Tel: (+420) 2 2435 7557
Fax: (+420) 2 2435 7556