"Improving cache efficiency through parallel ray batching"

Victor Veldstra

June 21, 2019

Supervisors: Dr. ing. J. Bikker, Dr. F. Staals
Utrecht University, Faculty of Science, Dept. of Computer Science

ICA-3956865

Abstract

Ray tracing is a rendering technique with an incoherent memory access pattern. In order to increase coherence and benefit more from caches in modern processors the technique of ray batching was created. This involves pausing ray traversal and resuming it later, when more cache hits can be expected. Ray batching proved successful but the technique was thus far limited to one processor core. This research aims to extend the technique of ray batching to work on multiple processor cores, in order to fully leverage the power of modern CPUs. A technique is presented to perform ray batching across multiple processor cores, however no performance increase is achieved over the current state-of-the-art techniques. Despite not achieving better performance than the current standard, insights are gained in what challenges need to be overcome for future work to improve upon the current standard. This thesis exposes the impact of inter-core communication overhead and shows promising results for batching in future work when that overhead is eliminated.

Utrecht University

Contents

1 Introduction

Ray tracing [Whi79] is regarded to be an excellent rendering technique to produce realistic-looking images. However, its memory access patterns are incoherent, reducing effectiveness of the available caching systems in modern CPUs. [WSBW01]. In an effort to increase coherence, and so speed up ray tracing, Gasparian and Bikker created a technique called ray batching. [Gas15]. Ray batching proves to be an effective technique, increasing performance of intersection queries between the rays and the scene. Ray batching so far only works for one processor core, leaving potential computational power of modern multi-core CPUs unused. This research aims to extend ray batching to leverage all available cores in a CPU. First related preliminary knowledge is presented in Chapter 2, followed by previous research in this field in Chapter 3. An overview of the used software tools is given in Chapter 4. Then a description of the parallel batching system is provided in Chapters 5 and 6. The proposed system is then evaluated with a series of experiments in Chapters 7 and 8. Finally the results and contributions to future work are discussed in Chapters 9 and 10.

2 Preliminaries

This chapter will present knowledge that is related to this research, elaborating on the topics of ray tracing principles, the Bounding Volume Hierarchy acceleration structure, Single Instruction Multiple Data, and hardware.

2.1 Ray properties

In ray tracing a distinction can be made between different rays. Understanding the properties of these rays helps optimizing resource usage for the different types of rays.

2.1.1 Primary Rays

Primary rays are generated from the camera into a scene, based on the viewing plane. They benefit from a high coherence caused by their common origin and closely related directions. Their coherent nature makes them a good candidate for packet tracing, a technique first mentioned by Wald et al. [WSBW01]. In packet tracing, Single Instruction Multiple Data (SIMD) operations are used to intersect multiple rays against a scene in parallel. The cost of fetching scene geometry from the memory is amortized over the traced rays in a packet, thus hiding latency. This subject is further described in Section 2.3.

2.1.2 Diffuse Rays

Depending on the material of the hit surface, a (primary) ray may bounce into the scene. Coherence is lost very quickly after bouncing, decreasing SIMD efficiency for packet traversal, since SIMD does not perform well in situations with diverging calculations. Some work has been done to increase SIMD utilization for incoherent rays, with techniques such as stream compaction and ray reordering. [vA11, ORM08, BWB08, GR08]

2.1.3 Ambient Occlusion Rays

Ambient occlusion is defined by the ratio of rays that hit or miss geometry within a certain distance from the point of impact of the primary ray. This can be used as a useful approximation of global illumination. By randomly sampling a number of rays over the hemisphere of the hit point of the primary this ratio can be calculated.

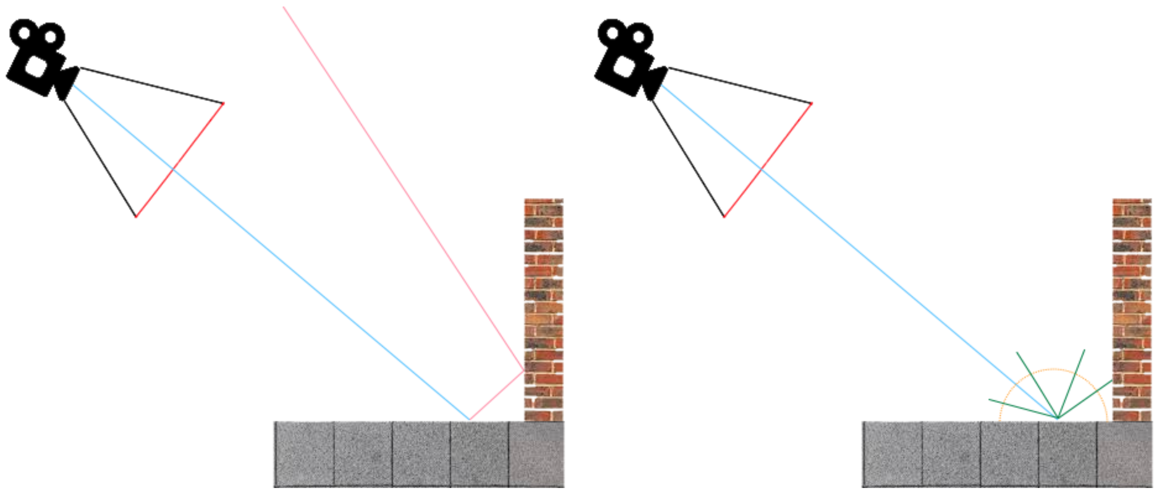


Figure 1: **Left:** An illustration of a primary ray (light blue) and two diffuse ray bounces (red). **Right:** An illustration of a primary ray (light blue), the ambient occlusion hemisphere (yellow) and several ambient occlusion rays (green).

2.2 (Multi) Bounding Volume Hierarchy

In order to speed up intersection queries with a scene some acceleration structure is desirable. This principle was first applied by Whitted [Whi79], who surrounded objects in the scene with bounding spheres. Kay and Kajiya then improved upon this idea by building a hierarchy of bounding volumes [KK86]. When a ray does not intersect a bounding volume, the entirety of the hierarchical structure inside that bounding volume can safely be ignored. This reduces intersection time from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$ where n is the number of objects in the scene.

In order to maximize usage of the SIMD instructions that are present in modern day CPUs for incoherent rays, Manfred et al. [EG08] and Wald et al. [WBB08] introduced the Multi Bounding Volume Hierarchy. This acceleration structure improves SIMD utilization for incoherent rays by intersecting a single ray against four or eight bounding boxes simultaneously for four-wide and eight-wide SIMD architectures respectively. This acceleration structure has been adopted into the Embree Framework [WWB$^+$14], which supports both the 4-wide and 8-wide MBVH. Henceforth an n-wide BVH will be referred to as BVHn.

2.3 Single Instruction Multiple Data

Single **I**nstruction **M**ultiple **D**ata (**SIMD**) is a concept where one instruction is executed on multiple values in parallel. Since there is only one instruction being executed at a time, there is parallelism, but no concurrency (**M**ultiple **I**nstruction **M**ultiple **D**ata). This term applies to a broad range of approaches. For example, one could store four byte values (0 .. 256) in a 32 bit integer value and then execute operations on this value. Most commodity hardware supports separate SIMD instructions and registers, such as AVX and AVX2, which feature 256 bit registers, able to perform operations on 8 32-bit values or 4 64-bit values. Since the use of SIMD instructions increases the amount of data being processed while keeping the number of instructions the same, near-linear speedups can be achieved, when used correctly. In order to achieve such speedups related data must be packed together in memory. The term *data locality* is used do describe how well data is placed in the memory, referring to related data being close to each other.

2.4 Hardware and Caching

For over thirty years a gap has been growing between computational speed and memory latency [HPAD, PAC$^+$97]. This means a lot of CPU cycles are wasted waiting for memory to be retrieved when no countermeasures are taken. This problem will likely continue to grow, since the development trends for memory favor storage capacity over latency improvements. One of such countermeasures is caching. With this technique data is retrieved from DRAM and stored in a smaller memory that is on the CPU core, so that when data needs to be accessed it is already available, and can be processed with very little latency. Modern systems, such as the target hardware for this research, feature a cache hierarchy for optimal performance. Data is retrieved per *cacheline*, which is a block of data of 64 or 128 bytes. When a memory address is requested that is in the same cacheline as previous requests, this value can be retrieved at optimal speed. This means an algorithm will benefit from reading data that is close to previously read data. This principle is called **data locality**.

Prefetching is the act of loading data into a cache before it is used. This means that data will be present when the operations require it, increasing the number of cache hits. The Intel i7 processor family is equipped with two kinds of hardware prefetching mechanisms:

Data Cache Unit prefetcher This prefetching system is triggered when data is accessed in an ascending manner. The prefetcher assumes the next data that will be required can be found in the next cache line, and prefetches this line.

Instruction pointer based strided prefetcher This prefetcher detects regularities in load instructions. When a regular stride is detected in the load instructions, this prefetcher fetches the next address based on the current cache content and the detected stride. This prefetcher can prefetch forward and backwards, with strides up to 2KB (half of a 4KB memory page).

The i7 processor family also supports software-controlled prefetching of data. (See [noa], chapter 2.4.4.2). Software controlled prefetching allows the programmer to load data into caches before it will actually be used by the instructions. The effectiveness of any prefetching operation is bound by various factors, such as ongoing memory usage by other applications, the locality of the data that is being requested and the time between the prefetch request and the moment the data is actually first used.

3 Previous Work

This chapter elaborates on research that is done prior to this work, upon which this thesis is built.

3.1 Ray Batching

Batching rays is a concept that is originally described by Pharr et al. [PKGH97]. In their paper they describe a system where batching and caching are used to render scenes that are too large to fit in main memory (out-of-core rendering). The scene is split up into equal sized voxels (called "geometry voxels"). A geometry cache holds a limited number of these voxels in RAM, the rest of the scene resides on a disk. Rays are not traced directly, but fed into a scheduler. This scheduler processes rays in an order that optimizes cache usage. To achieve this, the scene is split up into voxels (called "scheduling voxels"). These scheduling voxels may differ from the geometry voxels. Rays are batched inside the scheduling voxels. An algorithm then picks a voxel to process based on a cost estimate for tracing that voxel. The cost estimate is calculated based on the geometry that is currently in the cache, minimizing expected cache misses. When all voxels are processed the algorithm terminates.

Ray batching can also be observed in hardware implementations. Aila et al. [AK10] proposed a hardware architecture that operates on subdivisions of the acceleration structure. These subdivisions, called treelets, are built heuristically in a dynamic-programming approach. Kopta et al. [KSS$^+$13] proposed a hardware rendering system where a BVH is split up into treelets. Each treelet is exactly the size of L1 cache, so that cache hits are guaranteed after the first miss. Keely et al. [Kee14] showed, using a hardware simulator, that bandwidth is a limiting factor in GPU ray tracing and by reducing the precision of hardware, a substantial speedup can be obtained. They also used a scheduling algorithm on treelets to maximize cache utilization.

3.2 Stackless Traversal

In classical ray tracing the acceleration structure is traversed in a recursive manner (see Algorithm 1). When the algorithm enters recursion (line 8 and 9), the information about the traversal state on the program stack is updated by the compiler.

In order to facilitate pausing and resuming traversal the state must be stored. Storing the entire call stack for a single ray would leave a big memory footprint, so a "stackless" traversal scheme must be exploited to efficiently pause the ray traversal. The task of maintaining a traversal stack is taken away from the compiler (by eliminating recursion), and given to the user.

Algorithm 1 Pseudocode for a classical BVH traversal algorithm

```
 1: function TraverseNode(BVHNode node, Ray ray)
 2:     if !Ray.Intersect(node.BoundingVolume) then
 3:         return
 4:     end if
 5:     if node.IsLeaf( ) then
 6:         node.IntersectPrimitives(ray)
 7:     else
 8:         TraverseNode(node.LeftChild)
 9:         TraverseNode(node.RightChild)
10:     end if
11: end function
```

Stackless traversal for the BVH acceleration structure was first introduced by Laine [Lai10]. The algorithm proposed by Laine stores one bit per BVH depth level, and only works for binary BVH's. When there are insufficient bits available to store the traversal stack, restarting is used instead of backtracking. Hapala et al. [HDW$^+$13] use parent pointers to allow for stackless traversal, eliminating the need for restarting. This method however requires the traversal order to be recalculated when revisiting a node, meaning expensive calculations (such as calculating the ray/boundingbox intersection distances) slow down the traversal. Consequently the traversal order in this approach is limited to a simple heuristic. Barringer et al. [BAM13] propose a method for stackless traversal that allows for dynamically ordered traversal of the leaf children called Dynamic Ray Stream Traversal (DRST). Fuetterling et al. expand upon this work by lifting some of its limitations, namely the limited number of traversal orders and high bookkeeping overhead [FLPE15].

3.3 Cache Oblivious Algorithms

In many cases of software design very few assumptions can be made about the target hardware that the software will run on. The programmer cannot know the properties of cache sizes, memory layout and other hardware-specific details. Even if the programmer does know, it is undesirable to perform optimizations for every possible combination of the properties. To address this problem a class of algorithms called **Cache Oblivious Algorithms** exist. These are algorithms that perform well in terms of cache utilization without knowing anything about the hardware they are executed on. Cache Oblivious Algorithms should not be confused with **Cache Aware Algorithms**, which take properties about the cache size as parameters. [FLPR97].

3.3.1 COA in the context of Ray Tracing

Moon et al. [MBK+10] used a cache oblivious approach to ray reordering to improve cache utilization. First, they trace rays into a very low-polygon version of the scene, which is generated off-line based on the actual scene data. This results in an approximation of the hit points for each ray, by which the rays are sorted in a cache oblivious optimal order (such as a Z-curve). Then the rays are traced into the high-quality version of the scene. Since the rays are sorted by their hit points, many geometry cache hits are expected for sequential traces, since a lot of the same (or nearby) geometry will be hit.

Yoon et al. [YLPM05] improved access times for mesh-related algorithms (ray tracing, collision detection, etc) by optimizing the data layout of meshes according to estimated cache misses when accessing the mesh data. Yoon et al. later further improved upon this by storing a BVH in such a way that cache misses are reduced [YM06]. Both of these algorithms and the heuristics used to estimate cache performance are cache oblivious.

4 Software Tools

This chapter describes software tools and frameworks, as well as software design and synchronization principles, that are related to the system presented in this research.

4.1 Synchronisation tools and techniques

In an ideal parallel execution every thread may work independently. In more complicated algorithms this is often not the case, and some synchronization or communication between threads is required. Some tools and techniques that are available to the programmer to achieve this are described in this chapter.

4.1.1 Mutexes

In parallel programming a region of the program instructions that may cause issues when simultaneously executed by multiple threads is called a **critical section**. To prevent problems, only one processor may access that piece of code. To achieve this programmers can use the concept of **Mutual Exclusion**, or **Mutex** in short. Mutexes exist in different forms, and the best choice depends on the use case. There are a multiple different properties that define a mutex flavor. [Int19]

Scalability A mutex is called scalable when using it does *not* degrade performance further than serial execution. Some mutexes consume heavy processor power or memory bandwidth when waiting, making performance worse than serial code. Under heavy contention, non-scalable mutexes are often faster than their scalable counterparts.

Fairness A fair mutex lets threads pass in the order they arrive in. Because of this, fair mutexes avoid starving threads. However, unfair mutexes may be faster, because they could let a thread pass that is already running, avoiding the cost of starting up the thread that is waiting next in line.

Recursive A recursive mutex lets a thread that already holds a lock obtain another lock. This is useful for recursive algorithms, but generally adds some overhead to the lock implementation, making it slower.

Yield or block When waiting a long time, two options exist. A hardware thread may either yield, keeping the thread awake, but informing the scheduler that time may be spent on another thread. A hardware thread may also block, putting it to sleep so the scheduler doesn't have to allocate time for it. A sleeping hardware thread needs to be woken up when the mutex is allowed to pass. For short waiting times, a yielding mutex is often faster than its waiting counterpart.

4.1.2 Atomics

In contrast to regular operations, atomics are considered single operations even though they may perform complex tasks. As an example, let us consider the Fetch And Increment operation on an integer type. A regular increment operatation consists of fetching the value from memory, adding one to the value, and writing the outcome back to memory. This means that problems can arise when two threads execute the regular add operation simultaneously. Let's say two threads, thread A and B, want to increase a shared integer a of value 10. The expected outcome for $a = 12$. Thread A reads the register and loads value 10, thread B reads the register and loads value 10. Both threads increment the value by one and now have value 11. Both threads write back the value to the register, which now incorrectly holds the value of 11.

This issue can be solved by using atomic operations, which do not consist of multiple separate steps. One thread sees an atomic operation on another thread as happening instantaneously. For our example this means that two different threads incrementing $a = 10$ will correctly result in the expected outcome of $a = 12$.

One drawback of atomics is that their set of possible operations is quite limited, and often not enough to express complex behavior. The available atomic operations for x86/x64 family architectures are listed in Table 1

Operation	**Result**
= x	read the value of x
x=	write the value of x and return it.
x.fetch_and_store(y)	do x=y and return the old value of x
x.fetch_and_add(y)	do x+=y and return the old value of x
x.compare_and_swap(y,z)	if x == z, then do x = y. In either case return the old value of x.

Table 1: Atomic operations and their outcome upon execution.

4.1.3 Intel Threading Building Blocks

Intel Threading Building Blocks (often abbreviated to TBB) are, as the name suggests, a set of tools that can ease the task of parallel programming, while striving for high performance. By offering high-level abstractions over complicated concepts, parallel execution can easily be obtained. For expert users low-level control can be maintained for maximum speed when requirements are known. TBB offers a collection of thread-safe containers, such as a concurrent vector, a concurrent queue, concurrent set and a concurrent hashtable. Instead of trying to fit the most general requirements as libraries often do, TBB empathizes on speed. A concrete example is that the concurrent vector does not implement an operation to remove an element, since that cannot be achieved in a lock-free manner. Instead of opting for the most general API and implementing a locking remove operation, TBB simply omits this operation. TBB also ships a highly customizable task scheduler, abstracting away hardware threads when using the high-level API.

4.2 Embree

The Embree framework [WWB$^+$14] is a highly optimized set of open source ray tracing kernels specifically designed for x86/x64 family CPU architectures. It takes away some of the most important core tasks of ray tracing and executes these with high efficiency. Embree features include acceleration structure maintenance and intersecting rays with a scene. This means the functionality of Embree is limited to a low level, giving the user full control over the way rays are generated and shading techniques that are applied to achieve the intended graphical output. Since Embree operates on the core tasks at such a low level, it is widely used in production systems, for example Dreamworks' MoonRay [LGXT17], where developers desire full control of the input and outcome of the render. Embree features a flexible API, allowing scene and ray data to be input in various formats. Since Embree does not have control over the entire rendering pipeline, optimal performance cannot be achieved by Embree alone. Some degree of cooperation between the user and the framework is required to achieve optimal performance. An example of this is the user having to provide Embree with a hint about the coherence of rays that are handed to the API, enabling Embree to pick the fastest traversal algorithm.

5 Batching

This chapter describes the original batching scheme by Gasparian and Bikker [Gas15], and the changes made for the purpose of this research. A description on how parallelism is achieved in this research is given in Chapter 6.

5.1 Motivation

In Section 2.4 caching and its benefits were discussed. The batching algorithm operates on the premise that executing a task with many cache hits is faster than executing the same task with fewer cache hits. Classical BVH traversal, as shown in algorithm 1 of section 3.2, fetches many nodes for each ray, resulting in a scattered memory access pattern when executed on a large quantity of rays. By batching rays that have a similar direction and origin, we make the path rays take through the BVH more coherent, increasing cache efficiency, which contributes to the primary goal of reducing time spent on ray/scene intersection query calculations.

In order to harness the full power of modern CPU's the algorithm needs to operate on multiple cores in parallel. A naive approach, where the entire workload is copied for each core, is hardly scalable and faces a massive memory requirement. A more sophisticated solution is required to keep all cores busy while preventing unnecessary copying of data.

Additionally, by batching the rays based on their location and direction we can divide work geographically. This could be interesting for applications in distributed computing. Each node in a distributed computing network then only needs to hold a portion of the scene, instead of all nodes requiring the full scene data, as would be the case when dividing the work on a per-pixel basis. While this may be an interesting observation for future work, it will not be elaborated further upon in this research.

5.2 One core

The batching scheme works by taking a regular 4-wide BVH (also known as a BVH4) and splitting it horizontally into two layers. All rays traverse the top layer until they reach a **treelet**, a top-most node of the bottom layer. Instead of continuing traversal as they classically would, rays are put in a container on the root of the treelet. This is referred to as **batching**. The traversal state of the ray for the top layer is saved. The technique for saving the traversal state of a ray will be explained in detail later. When all rays have traversed the top layer of the BVH, treelets that have rays in their associated containers are handled and every ray that was queued up in the container resumes its path through the treelet. After traversing a treelet two scenarios can occur. The ray may have undiscovered branches left in the top layer of the BVH, as determined by its saved traversal state. In that case traversal is resumed in the top layer of the BVH and the process of batching starts again. The ray may have no unfinished work left to do, its traversal state is empty, in which case the ray is terminated. The scheme repeats the cycle of traversing the top layer of the BVH and handling all treelets until all rays are terminated, as seen in Figure 2.

The horizontal splitting of the BVH happens when any of the following conditions are true. A node is marked as treelet when either one of its children is a leaf-node, its depth in the BVH is 15, or the total number of nodes below it is smaller than a pre-set threshold. This threshold is empirically picked so that a treelet fits nicely into the processor cache. The depth restriction of 16 layers, of the top-BVH is due to the way the traversal stack is saved, which will be elaborated on later.

To maximize cache hits it is best to pick a treelet with a large number of batched rays at each step. To achieve this a lightweight bookkeeping solution is required. The collection of rays for a treelet is made of fixed size **buckets**. When the amount of batched rays in a treelet exceeds the number of rays that fit in a bucket, an extra bucket is reserved and associated with that treelet. In order to facilitate picking a treelet with lots of batched rays three collections are distinguished between. A collection of treelets that have multiple full buckets, a collection of treelets that have one full bucket (and optionally another bucket which is partially filled), and a collection of treelets that have only one bucket which is only partially filled. When deciding which treelet to handle first, the algorithm prefers

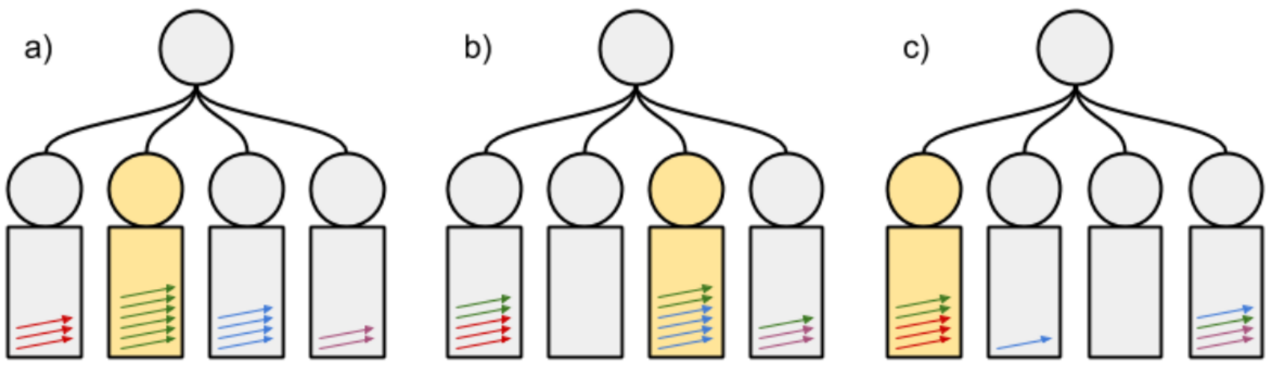


Figure 2: An illustration of the batching process. A top-BVH with 5 nodes, of which 4 are treelets, is shown. All rays are traversed through the top-BVH and are distributed among the treelets. Image **a.)** shows how a treelet with a high number of rays is selected (highlighted in yellow). Its rays are traced through the treelet and distributed over the top-BVH, resulting in image **b.)**. This process is repeated until all rays are terminated.

to take one from the collection with multiple full buckets, then from the second or third collections respectively. This solution is simple enough to avoid expensive sorting calculations, while keeping a high workload.

5.2.1 Saving the ray state

The batching step requires a lightweight way to store the state of a ray during traversal of the top-BVH, in order to resume traversal at a later moment. To solve this Gasparian and Bikker [Gas15] added a 64 bit integer field to every ray instance, called the **bitstack**. This field serves as a stack during the traversal, by setting certain bits to indicate unfinished work. Since every node in the BVH has four children, we can use four bits to represent a node. This means the top-BVH is limited to a depth of 16. The four least significant bits in the bitstack represent the root node of the top-BVH. For every child node that is hit a bit is set to 1. These four bits are called the **interestMask**. The nearest intersecting child node is selected and its bit is set to zero to make sure it will not be visited again. The next four bits, corresponding to the depth of the child nodes are set to one, which indicates the next level needs to be traversed. A new intersection test for every child node is done, resulting in the **hitmask**, bits corresponding to child nodes that do not intersect the ray are set to zero through a bitwise AND operation, and the process is repeated. An illustration of this process can be seen in figure 3.

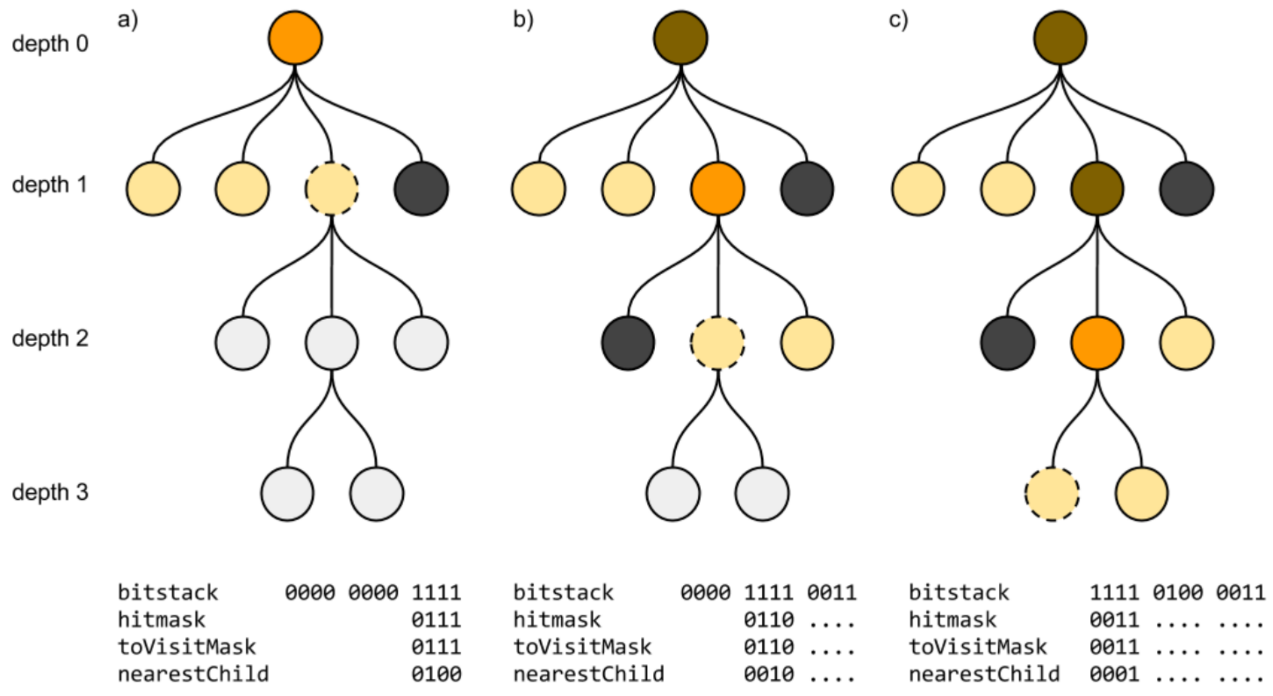
Unwinding the traversal stack requires each node in the top-BVH to have a collection of pointers to their ancestors. These ancestor-pointers are used to jump up through the BVH in constant time, as is illustrated in figure 4.

For the treelet traversal any algorithm can be supplied. Gasparian and Bikker recommend a traversal scheme with low starting overhead. For their own research and validation, they used ORST [FLPE15] to handle treelets.

For further technical details, pseudocode and accompanying C++ source code of the single-core batching system, please refer to the masters thesis of Gasparian [Gas15].

5.3 Differences from the singlecore implementation

In the implementation by Gasparian and Bikker ORST is used as the traversal scheme for the treelets. This research uses Embree to handle the traversal through treelets, in order to be more comparable to the ground truth and similar research. Besides traversal, Embree provides the benefit of a production-quality BVH builder. The system uses the BVH provided by Embree as a basis for the top-level BVH as well as the treelets. The bookkeeping of the treelets is somewhat simplified. The implementation



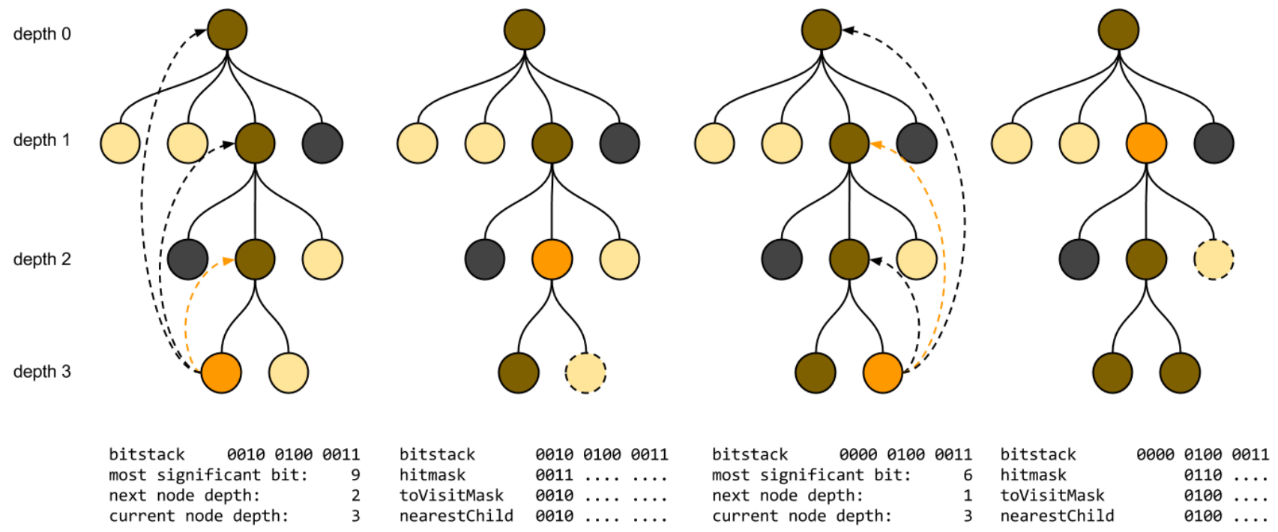
```
bitstack       0000 0000 1111
hitmask                  0111
toVisitMask              0111
nearestChild             0100
```

```
bitstack       0000 1111 0011
hitmask             0110 ....
toVisitMask         0110 ....
nearestChild        0010 ....
```

```
bitstack       1111 0100 0011
hitmask        0011 .... ....
toVisitMask    0011 .... ....
nearestChild   0001 .... ....
```

Figure 3: An illustration of the traversal process and its bitstack. The orange node indicates the current node being considered. Dark grey nodes do not intersect the ray. Yellow nodes intersect the ray. A dashed outline indicates the nearest node and brown nodes have already been visited. **a.)** The stack is initialised by setting the four least significant digits to 1. The hitmask is constructed by intersecting the children. The toVisitMask is constructed from a bitwise AND of the interestMask and the hitmask. We then select the nearest child, and update the bitstack with the toVisitMask. The process is then repeated for the nearest selected child. **b.)** and **c.)** show the next two interations.

of this research only distinguishes between treelets that have one or more full buckets, and treelets that have exactly one partially filled bucket, as opposed to the three collections in the original work, as described in Section 5.2.



```
bitstack      0010 0100 0011
most significant bit:      9
next node depth:           2
current node depth:        3
```

```
bitstack      0010 0100 0011
hitmask       0011 .... ....
toVisitMask   0010 .... ....
nearestChild  0010 .... ....
```

```
bitstack      0000 0100 0011
most significant bit:      6
next node depth:           1
current node depth:        3
```

```
bitstack      0000 0100 0011
hitmask            0110 ....
toVisitMask        0100 ....
nearestChild       0100 ....
```

Figure 4: An illustration of the backtracing process of the bitstack. For a legend of the node colors see figure 3. The dashed arrows indicate ancestor pointers of the current node. When backtracing is required, for example when there are no more children to visit, we find the index of the most significant digit of the bitstack. The depth of the next node to visit can be found by dividing that index by four. Using this depth we follow the corresponding ancestor pointer and resume traversal.

6 Description of the parallel batching algorithm

Ray tracing is generally considered embarrassingly parallel, meaning turning a serial implementation into a parallel one is trivially easy. The batching scheme however is not. This chapter describes how synchronization techniques such as mutexes and atomics can be used to turn the batching scheme by Gasparian and Bikker into a parallel scheme.

6.1 Preparation

The algorithm starts by preprocessing the given scene. After feeding the scene data to Embree a BVH4 can be retrieved from Embree. The top-BVH is constructed serially, since that is merely a one-time operation. Each node in the top-BVH is given a thread-safe container for its buckets, and a read-write mutex called the **bucketLock**, along with the regular node data such as bounding box information, and the data required by the batching scheme as explained in Section 5.2. Furthermore each top-BVH node is given a reference to the node it was constructed from in Embrees BVH.

6.2 Core routine

The entrypoint for the intersection algorithm starts at *IntersectRaysBatching*, which is described in Algorithm 2. Just as with the serial version, we start by traversing all rays through the BVH. Our Top-BVH traversal algorithm will be discussed in chapter 6.3. The top-BVH traversal algorithm has been modified to allow parallel execution, so to benefit from this the for-loop on line 2 is executed in parallel. When all rays have traversed the top-BVH the treelet collections, named *NodesWithFullBuckets* and *NodesWithPartiallyFilledBuckets*, will repeatedly be emptied. Unlike the serial implementation, we can empty these lists in parallel. In order to avoid concurrency conflicts, we copy both collections, and clear their originals (lines 9-12). This means treelets can safely be handled without any chance of hindering or corrupting the work of the previous iteration, much like a **double-buffer** approach. Copying these containers is not overly expensive since they merely contain references, not the treelets themselves. Then, we call the *HandleNodeList* method, which will handle all the treelets in the container in parallel. For bookkeeping purposes, we must point it to our created *taskList*, so that it may append tasks to it. The inner workings of the HandleNodeList method will be discussed later, in chapter 6.4. Before starting the next iteration of the loop we must wait for all created parallel tasks to be finished. This can be achieved by looping over the task list that was filled during the treelet handling, and calling the join operation, lines 15-17. Afterwards the task list must be emptied to prepare for the next iteration.

Algorithm 2 Pseudocode for the core batching system

```
1:  function IntersectRaysBatching(BVH4 bvh, Ray[] rays)
2:      for all $ray \leftarrow rays$ in parallel do
3:          $ray.traversalStack \leftarrow 15$
4:          TraverseTopBVH($bvh.root$, $ray$)
5:      end for
6:      $taskList \leftarrow [\,]$
7:
8:      while $NodesWithFullBuckets$ has items $\vee NodesWithPartialBuckets$ has items do
9:          $copyFullNodes \leftarrow NodesWithFullBuckets$.Copy( )
10:         $copyPartialNodes \leftarrow NodesWithPartiallyFilledBuckets$.Copy( )
11:         $NodesWithFullBuckets$.Clear( )
12:         $NodesWithPartiallyFilledBuckets$.Clear( )
13:         HandleNodeList($copyFullNodes$, $taskList$)
14:         HandleNodeList($copyPartialNodes$, $taskList$)
15:         for all $task \leftarrow taskList$ do
16:             $task$.join( )
17:         end for
18:         $taskList$.clear( )
19:     end while
20: end function
```

6.3 Top BVH traversal

In order to allow parallel traversal of the top-BVH the flow of the algorithm must be altered to remove any serial constraints. The traversal algorithm by Gasparian and Bikker assumes serial execution, and results in incorrect behaviour when executed in parallel. The following chapter shows how synchronization techniques such as atomics and mutexes, together with known properties and constraints on the thread-safe containers, can facilitate a parallel-friendly traversal scheme. The pseudocode provided in Algorithm 3 shows the traversal of the top-BVH. Lines 3-12 show the traversal of a ray though the interior nodes of the top-BVH. Instead of recursion, as in a classical traversal algorithm, the stack management technique as described in chapter 5.2 is used in conjunction with a while loop. Lines 39-42 show the backtracking through ancestors when no child collision is found. The stack is used to retrieve the correct ancestor to resume traversal.

Line 14 shows that a mutex is set in *read mode* for the node. The mutex being set is of type **ReadWriteMutex**, or a **RWMutex** in short. A RWMutex has the following interesting properties that we will exploit in the batching scheme.

1. An infinite number of Read Locks can be acquired at any time.
2. Only one Write Lock can be acquired at any time.
3. Requesting a Write Lock will block a thread until **all** other locks have been released.
4. Requesting a Read Lock will block a thread until no write locks are being held.
5. In regards to the mutex properties as described in chapter 4.1.1, the RWMutex implementation used in this research is not scalable, not fair, not recursive and yielding.

Because of property 1 of the RWMutex, multiple threads can safely pass this lock request on line 14, and no blocking will occur, meaning the traversal code is concurrency-friendly, despite the lock.

The lock is released from read mode on line 37, when traversal is completely finished for that ray. Since no write locks occur in the traversal algorithm, the lock may appear redundant. This is not the case. A write lock is acquired elsewhere, and will be discussed later.

On line 15 a unique index for the ray is set based on an atomic Fetch And Increment operation, see Chapter 4.1.2 for more information about atomic operations and their properties. This index is guaranteed to be unique for the ray in relation to the node, even under parallel execution. Adding new buckets to a node is done on lines 16-22. A ray number that is a multiple of the bucket size means that either no buckets exist yet, or the last bucket is now full. If the former is the case, the node has now become a node with a partially filled bucket, and is thus added to the collection (line 19).

A bucket index is decided for the ray on line 24 based on the ray index and the maximum number of rays that can fit in a bucket. Due to the nature of parallel execution, a thread with a ray number of 0 *may still be busy* adding a new bucket, while a thread with a ray number > 0 may pass beyond line 24. For this reason we have to wait a bit until the other thread, which is in charge of adding a new bucket, has completed the if statement of lines 16-22. The while-loop on lines 25-27 takes cares of this, and simply **yields** the thread, signaling the scheduler that computation time can best be spent on other threads. This wait time is unfortunate on the rare occasion that waiting here is required, but luckily very short in practice. Caution is advised: this method requires the container for the buckets to be thread-safe, and puts certain requirements on its API. For example, the containers' "count" method must not return incorrect results when called simultaneously with the "add" method.

When the first bucket is full, which can be detected easily by checking for equality between the ray index and the bucket capacity, the node must be moved to the container of full nodes. Line 29-32 show this check.

Finally the ray must be put in a bucket. The correct bucket can be obtained from the bucket index that we calculated earlier, and the desired place within that bucket is derived from the ray index. This can be seen on lines 34-36.

6.4 Handling the treelet lists

Rays now need to be traced into the treelets. The pseudocode in Algorithm 4 shows the method of handling a node list, which will be called from the core loop, with either *NodesWithPartiallyFilledBuckets* or *NodesWithFullBuckets* and the *taskList* that it needs to append to as its arguments.

We start by looping over the treelets in the supplied collection. The mutex called *bucketsLock*, for which we have so far only acquired a read lock, is then locked in write mode (line 3). If we recall the RWMutex properties from Chapter 6.3, we see that property 3 causes this request to block the algorithm as long as there are other open locks (either read or write). If any rays are still traversing the top-BVH through that treelet, and thus holding a read lock for it, we must wait. Luckily this is rarely the case in practice, and waiting time caused by acquiring the write lock is negligible. A more serious problem is caused by property 4 of the RWMutex. As long as this write lock is open, no read locks may be acquired. In other words, no top-BVH traversal through this treelet can be done as long as this mutex is locked in write mode. It is therefor of vital importance for our performance that we keep this write lock as short as possible. Immediately after the critical operation of clearing the buckets is finished, we release the write lock. The need for this write lock is caused by the specifications of the thread-safe buckets container. The implementation uses the Intel TBB concurrent_vector[1], whose documentation states that insertion and iteration may be done in parallel, clearing the container may not. So in order to maintain thread safety, we must be sure that no other threads are inserting or iterating over the collection of buckets. Again a double-buffer style approach is used. The collection of buckets is copied and the original is cleared. The copy is used for further processing. This is possible because once again, the collection only consists of references to the buckets, not the buckets themselves. By clearing the buckets the top-BVH traversal can freely work without danger of corrupting, overwriting or otherwise invalidating data.

Then the algorithm iterates over the collection of buckets and spawns a parallel task for every bucket (line 9). It is important to note that this method does *not* block until the HandleBucket

[1]`https://software.intel.com/en-us/node/506203`

Algorithm 3 Pseudocode for the parallel top-BVH traversal algorithm

```
function TraverseTopBVH(BVH4Node node, Ray ray)
    $stack \leftarrow ray.traversalStack \& ((1 << node.depth + 4) - 1)$
    while $node$ is not empty do
        $hitMask \leftarrow$ IntersectChildBoxes($node.children, ray$)
        if $hitMask$ is not empty then
            $closestChildIndex \leftarrow$ GetClosestChildIntersection($node.children, ray, hitMask$)
            $child \leftarrow node.children[closestChildIndex]$
            $stack \leftarrow stack + (15 << (node.depth + 4) + (hitMask - (1 << closestChildIndex)))$
            if $child$ is not a root of a treelet then
                $node \leftarrow child$
                continue
            end if

            MutexLockReadMode($child.bucketlock$)
            $myRayIndex \leftarrow$ AtomicFetchAndIncrement($child.RaysInNode$)
            if $myRayIndex$ mod $BucketSize \equiv 0$ then
                $newEmptyBucket \leftarrow$ GetEmptyBucket()
                if $child.buckets$.Count( )$\equiv 0$ then
                    $NodesWithPartiallyFilledBuckets$.Add(child)
                end if
                $child.buckets$.Add($newEmptyBucket$)
            end if

            $myBucketIndex \leftarrow myRayIndex \div BucketSize$
            while $child.buckets$.Count( ) $\leq myBucketIndex$ do
                YieldThread( )
            end while

            if $myRayIndex \equiv BucketSize$ then
                $NodesWithPartiallyFilledBuckets$.Remove($child$)
                $NodesWithFullBuckets$.Add($child$)
            end if

            $bucket \leftarrow child.buckets[myBucketIndex]$
            $bucket.rays[myRayIndex \bmod BucketSize] \leftarrow ray$
            $bucket.numberOfRays \leftarrow bucket.numberOfRays + 1$
            MutexUnlock( $child.bucketlock$ )
        end if
        if $stack \& ((1 << node.depth) - 1) \neq 0$ then
            $ancestorIndex \leftarrow$ GetMostSignificantBitIndex( $stack \& ((1 << node.depth) - 1)$ )
            $node \leftarrow Node$.GetAncestor( $ancestorIndex \div 4$ )
        end if
    end while
end function
```

Algorithm 4 Pseudocode for the node handling algorithm

```
 1: function HandleNodeList(NodeList treelets, TaskList taskList)
 2:     for all $treelet \leftarrow treelets$ do
 3:         MutexLockWriteMode($treelet.bucketlock$)
 4:         $bucketsCopy \leftarrow treelet.buckets.$copy( )
 5:         $treelet.buckets.$Clear( )
 6:         $treelet.raysInNode \leftarrow 0$
 7:         MutexUnlock($treelet.bucketlock$)
 8:         for all $bucket \leftarrow bucketsCopy$ do
 9:             $spawnedTask \leftarrow$SpawnParallelTask(HandleBucket(bucket, treelet))
10:             $taskList.$Add($spawnedTask$)
11:         end for
12:     end for
13: end function
14:
15: function HandleBucket(Bucket bucket, TopBVHNode treelet)
16:     EmbreeTraceRays($bucket.rays$, $treelet.embreeBVHNode$)
17:     for all $ray \leftarrow bucket.rays$ do
18:         if $ray.traversalStack \& ((1 << node.depth) - 1) \neq 0$ then
19:             $ancestorIndex \leftarrow$ GetMostSignificantBitIndex($stack \& ((1 << node.depth) - 1)$ )
20:             $ancestorNode \leftarrow treelet.$GetAncestor($ancestorIndex \div 4$)
21:             TraverseTopBVH( $ancestorNode$ )
22:         end if
23:     end for
24: end function
```

method is complete, instead it immediately starts execution of the supplied task, and returns a handle to it. This handle is saved in the task list, and can later be used to wait for the work to be complete, as seen in Algorithm 2, line 15-17. The task scheduler tools shipped with Intel TBB (see chapter 4.1.3) is used to achieve this behavior.

The method that is executed by a parallel task, *HandleBucket*, starts by tracing the rays through the treelet (Algorithm 4, line 16). Embree is given the collection of rays it needs to trace, and the BVH node at which traversal must start. Since we saved references to the Embree BVH during the construction of our top-BVH, we can retrieve the starting point for the traversal by accessing *treelet.embreeBVHNode*. When Embree has finished tracing the rays through the treelet we must check if they have any work on the traversal stack left. Just like during the top-BVH traversal, we find the most significant bit in the ray's stack, and extract the depth and correct ancestor from it (lines 18 to 22). If we have any work left to do, we repeat the process of traversing the top-BVH. Since this traversal call is executed from the spawned task, we again benefit from parallel execution during the top-BVH traversal.

For cache coherence reasons, spawning a task for every *node* instead of every *bucket* might seem an obvious choice. However this proved to degrade performance significantly, due to processor cores having little to no work to do at times, while waiting for other cores to finish their work. The reason for this uneven distribution of work lies in the nature of ray tracing. Some nodes in the BVH may contain up to a thousand times more rays that others. Consequently, parallelization at a node level creates a very uneven workload. The big inequality of the number of rays that visit each treelet is illustrated for an example scene in figure 5. This problem can hardly be adressed by modifying the BVH, since imbalance in the tree is in the nature of a BVH. A more suitable approach is to spawn tasks per bucket instead of per node. Spawning tasks per bucket has a big advantage in terms of predictability. Buckets are of fixed size, meaning the workload of every spawned task is roughly equal. This allows the task scheduler, which is in charge of distributing the work of parallel tasks amongst cores, to evenly distribute the work. Even distribution of the work means all processor cores are kept busy with calculations, regardless of the BVH's quality and the distribution of geometry in the scene.

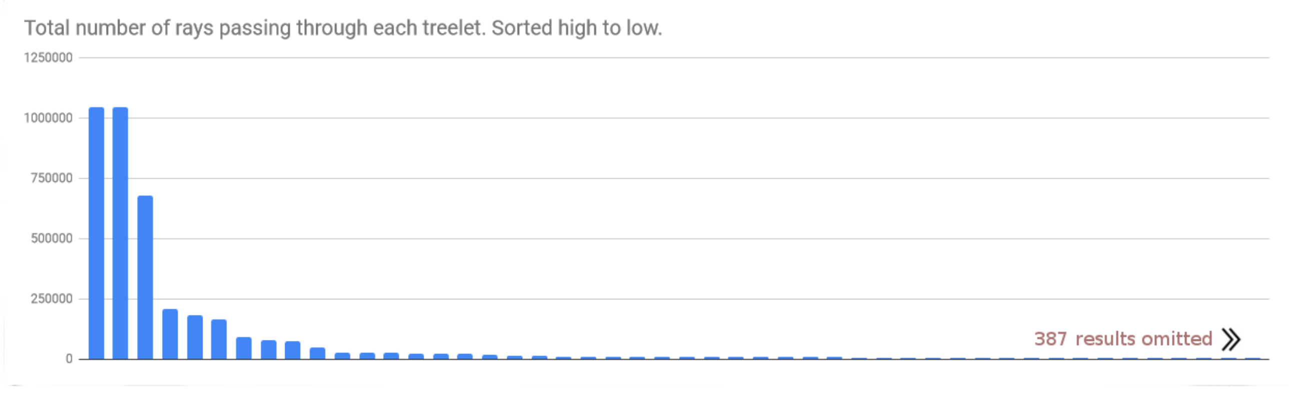


Figure 5: Graph showing the number of rays that pass through a node, sorted from high to low. 10 nodes contain high amounts of rays, surpassing 10k rays. Three nodes even contain over 500k rays. All the other nodes contain fewer than 10k rays. Over 300 nodes have been omitted to the right to make the graph readable. The total sums up to 4.2M rays. Scene used is crytek sponza. Results are expected to vary for different scenes. This graph only serves as an illustration of *an* uneven distribution, not as a general truth for every possible scene.

7 Experiments

In order to generate work for the batching scheme a simple Ambient Occlusion (AO) renderer has been created which first traces primary rays through the scene, followed by an adjustable number of AO samples for each hit. To provide a sensible ground truth for comparisons, the renderer is equipped with an option to entirely disable the batching scheme. When batching is disabled, all intersection work is propagated to Embree. The AO shading code is shared between the batching and non-batching mode, enabling meaningful comparison between the two.

Other applications and operating system services will be running and using memory. That means the processor caches may contain data that is not related to our experiments (cold caches). Since the batching scheme is dealing with cache performance we need to **warm the caches** before any test that we do. To achieve this we run the desired experiment twice and discard the results. By discarding the first two experiments we minimize the impact of cold caches. Then any test will be executed multiple times and an average will be taken.

The batching scheme features two parameters for which an optimal value must be found in order to achieve maximum performance. These parameters are

TreeletThreshold An integer number describing the biggest size a node may be before being cut off from the top-BVH and becoming a treelet. It is one of the three conditions for a node to become a treelet, as described in chapter 5.2. Used exclusively during the building of the top-BVH.

BucketSize The number of rays that fit in a single bucket.

Since these parameters greatly influence performance we will first find suitable values for them and then assume them to be constant for the remainder of the tests. This reduces the required number of test runs and consequently also reduces the dimensionality of the collected data.

7.1 Scenes

The **Sponza Atrium** scene was chosen due to its popularity in the ray tracing community. It features an enclosed scene consisting of 67k triangles. Since the building modeled in this scene is fully closed the camera can be placed at any position inside the scene and it will always have a 100% hit rate for its primary rays. This makes it an excellent target to create big and predictable workloads.

The **Powerplant** scene was chosen due to its high complexity and primitive count. It features lots of pipes and other industrial objects resulting in a complex acceleration structure where lots of ray intersections are expected to barely miss the geometry. Combined with its high primitive count it makes for a good candidate to test acceleration structure usage. It consists of 12.7M triangles.

Figure 6: Left: Example image of Sponza. Right: Example image of Powerplant

7.2 Tests

To test the performance of the batching scheme the **render time** is measured. This includes shading and intersection calculations for a 1024x1024 pixels image. Since the shading calculations are shared between the batching and non-batching rendering scheme, any difference in performance can be attributed to the intersection code.

To test how well the parallel batching scheme scales to multiple cores we also test the **performance versus the available cores** when doing the batching intersections. This test aims to verify that the performance of the multi-threaded batching scheme scales linearly with the number of cores as expected. Since the parallel batching scheme involves some synchronization in the form of locks, we measure the time spent waiting for locks. To achieve this Intel VTune is used to observe the execution of the program. A *Threading Analysis*[2] can be used to observe how much time is spent within the program waiting for locks.

In order to test our expectation that the batching scheme will indeed improve cache utilization a test involving simulated caches is set up. The cache simulator is a piece of software that takes the processor instructions from a compiled program, and simulates the CPUs behaviour as closely as possible. The cache simulator is a tool from the open source DynamoRIO project, a set of tools for runtime code analysis and manipulation. From the output of the cache simulator we can see how many data accesses have hit the level one cache and how many have missed for each simulated core.

The cache simulator measures the cache behaviour of the entire program. In order to verify that cache behaviour for *only the intersection queries* is indeed improved, we use VTune to measure how much time is spent on function calls to Embree. The batching scheme is expected to have less execution time being spent on calls to Embree, since we benefit from cache coherence.

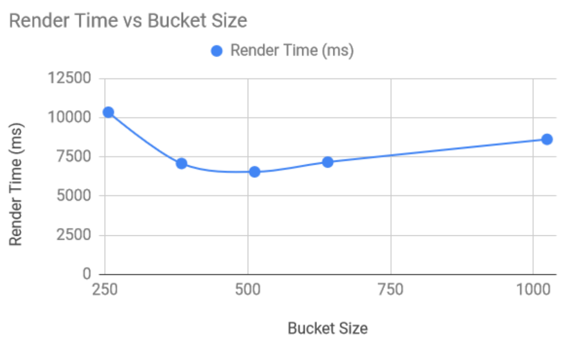
[2]`https://software.intel.com/en-us/vtune-amplifier-help-threading-analysis`

8 Results

This chapter describes and presents the results from the experiments described in chapter 7. The meaning and conclusions that can be drawn from the data will be discussed in chapter 9

8.1 Finetuning parameters

Before performing the experiments described in chapter 7, we need to find suitable values for the TreeletThreshold and Bucket Size. Figure 7 (left) shows the results of rendering the Powerplant scene with variable bucket sizes. After determining a good value for the bucket size parameter we can move to find a good value for the Treelet Threshold. Figure 7 (right) shows the results of rendering the powerplant scene with a bucket size of 512 and variable treelet thresholds. A value of 512 results in the best render time, so we will pick that for further experimentation.



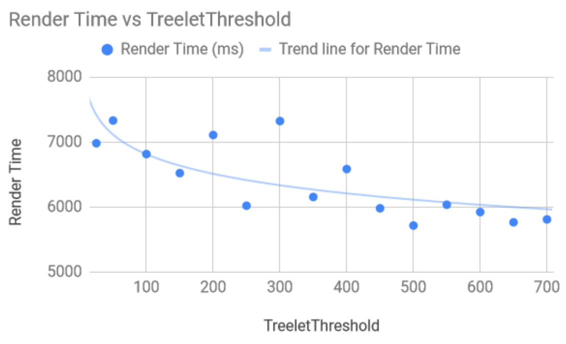
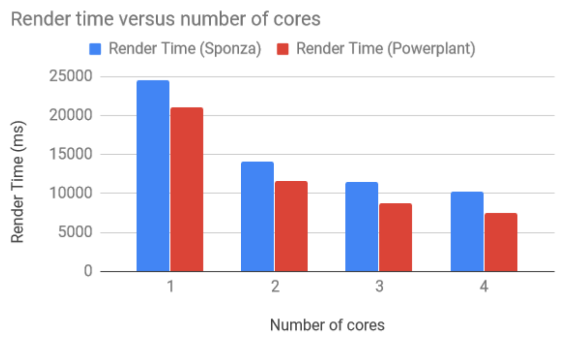


Figure 7: **Left**: Render time (y axis) of a 1024x1024 image of Powerplant with 16 AO samples and variable bucket size (x axis). **Right**: Render time (y axis) of Powerplant with 512 bucket size and variable TreeletThreshold (x axis)

8.2 Cores

To test our expectation that the batching scheme scales linearly with the number of available cores the number of cores was restricted and different scenes were rendered. The resulting render times can be seen in Figure 8 (left)



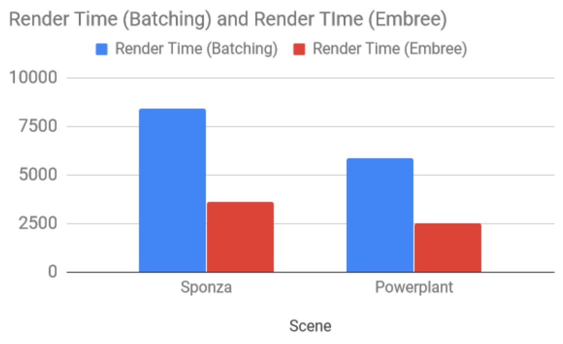


Figure 8: **Left**: Render time (y axis) of a 1024x1024 image with 16 AO samples and a variable number of processor cores (x axis). **Right**: Render time of different scenes rendered with the batching scheme and rendered with only Embree.

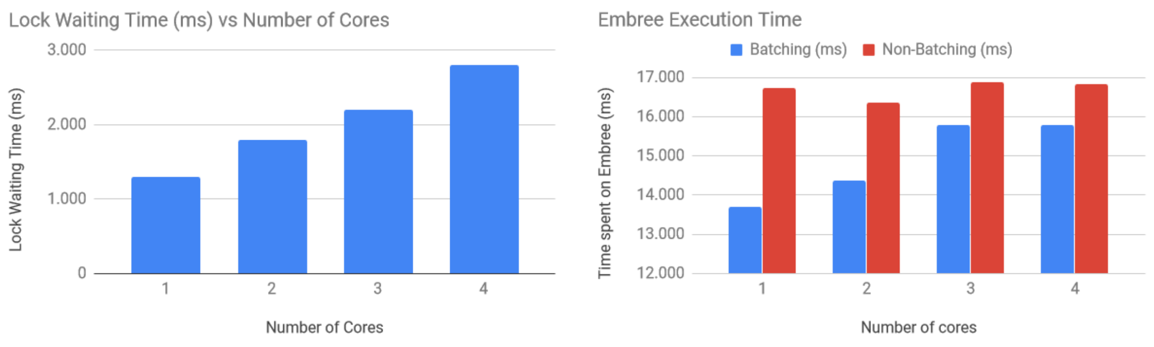


Figure 9: **Left**: Total time spent waiting for locks (y axis) when rendering a 1024x1024 image with 16 AO samples and a variable number of processor cores (x axis). **Right**: Time spent across all cores within function calls to the Embree intersector, as measured by Intel VTune when rendering the Sponza scene for a 1024x1024 image with 16 AO samples.

	Sponza			Powerplant		
	Hits	Misses	Miss Rate %	Hits	Misses	Miss Rate %
Core 0	164.676.932	3.859.509	2,29	6.171.288.642	170.186.670	2,68
Core 1	148.569.169	3.379.322	2,22	3.895.625.491	125.176.581	3,11
Core 2	79.819.695	2.144.995	2,62	3.959.992.803	121.282.007	2,97
Core 3	127.035.091	2.869.449	2,21	4.016.257.205	117.305.766	2,84
Averages	130.025.222	3.063.319	2,34	4.510.791.035	133.487.756	2,90

Table 2: Cache hits, misses and percentages for the batching scheme for Sponza and Powerplant

8.3 Batching versus Embree

We also test the speed of the multi-threaded batching scheme against the speed of Embree without batching. The render times of this comparison can be seen in Figure 8 (right)

8.4 Cache utilization

To verify our expectation that the batching scheme improves cache utilization we run the batching scheme and Embree using a cache simulator. The results of the cache simulator for the batching scheme can be seen in Table 2 and the results for the cache simulator for Embree can be seen in Table 3.

Figure 9 (right) shows time spent within intersection calls to Embree. Since the same scene is rendered for batching and non-batching, results between the two times can be attributed to cache performance.

	Sponza			Powerplant		
	Hits	Misses	Miss Rate %	Hits	Misses	Miss Rate %
Core 0	137.025.385	3.092.433	2,21	5.402.870.525	164.373.873	2,95
Core 1	94.933.133	2.085.586	2,15	3.927.808.080	118.598.391	2,93
Core 2	74.650.784	2.039.783	2,66	3.934.107.502	115.837.811	2,86
Core 3	115.755.329	2.214.774	1,88	3.954.874.608	117.707.738	2,89
Averages	105.591.158	2.358.144	2,23	4.304.915.179	129.129.453	2,91

Table 3: Cache hits, misses and percentages for Embree for Sponza and Powerplant

9 Discussion

From the render times in Figure 8 (right) we can see that Embree performs significantly better than the batching scheme to render the same scene. This can be explained by examining the performance scaling of the batching scheme with different cores in Figure 8 (left). An ideal algorithm scales linearly with the number of available physical cores, meaning that having 4 cores should ideally result in a 4x speedup in render time. This is not the case for the multi-threaded batching scheme. As the number of cores increases, the gain in performance decreases. These diminishing returns can be explained by the bookkeeping overhead the parallel implementation requires. Locks and atomic operations tend to collide more frequently when more parallel threads are active, reducing the effective work done. Figure 9 (left) confirms this, and shows increasing wait times for concurrency overhead with an increasing number of processor cores working in parallel. Since the implementation without batching spreads the work independently over the available cores, there is no communication involved between the cores, meaning they can speed up the calculations linearly. With this knowledge in mind, it is hardly surprising that the embarrassingly parallel Embree renderer outperforms the multi-threaded batching scheme.

Tables 2 and 3 show an increase of roughly 20% in memory fetches when using the batching scheme compared to using only Embree. As the number of cache hits increases when enabling the batching scheme, so does the number of cache misses, keeping the Miss Rate roughly equal between batching and non-batching. The cache miss rate is especially close to the embree base line for the more complex Powerplant scene, having only a 0.01 percent difference.

The overall increase in memory access for the batching scheme can be explained when looking at its implementation. The top level BHV that is used for the batching scheme holds more information per node than the BVH that Embree uses, such as data for locks, counters for the batched rays and containers for the buckets. Having a bigger data structure means more cache lines must be fetched to get all the required data to the core.

If we observe the chart in Figure 9 (right) we can see that total time spent within Embree across all cores is roughly equal for each number of available cores when intersecting without batching. This is entirely in line with our expectations, since classical ray tracing works independently and scales linearly with the number of cores. The time spent within Embree is less for the batching approach. Since the code that is being executed within Embree is exactly the same, the difference in performance must be attributed to the coherence of the workload that it is given. The total time spent in Embree across all cores changes with the number of cores. This can be explained by a difference in the amount of rays that are batched, which changes with the number of cores. With only one core available *all* the rays are distributed over the treelets before the treelets are traced, making for a very high cache coherence. With more cores available, the handling of the treelets starts earlier on a parallel thread while rays are still being distributed over the treelets. This means coherence is reduced.

Time spent within Embree being less for the batching scheme can also be explained by the fact that the batching scheme traces the top part of the BVH, and Embree only tracing the treelets. While this is expected to account for some difference, it is not enough to be the *only* difference. As shown by Bikker the top BVH traversal is typically only a small fraction of the BVH. [Bik12]. Especially as scenes become larger the top BVH becomes a smaller fraction of the total BVH, since its depth is restricted and therefor can not grow indefinitely.

The performance gains reported by Gasparian and Bikker for the batching scheme, which are roughly 50%, are not achieved in this research. This can be attributed to changes made to the algorithm in order to adapt it for parallel execution. These changes include processing buckets per thread instead of treelets in order to balance the workload better, as discussed in chapter 6.4. This means fewer cache hits can be expected for each treelet. Another factor is the simplification of treelet bookkeeping as discussed in chapter 5.3. Since the parallel batching scheme does not keep track of treelets with multiple full buckets any more, the benefit of picking treelets with a very high workload is lost, reducing cache utilization.

10 Conclusions and Future Work

The parallel batching scheme presented in this research is not faster for ray/scene intersection queries than Embree. This can mainly be attributed to the communication overhead causing the algorithm to scale poorly with the available processor cores. Furthermore, dividing the work over multiple cores implies fewer cache hits per core will be made. More time to optimize the implementation might have decreased the gap between the render times for batching and non-batching. However, in order to beat the render time of Embree a different technique, with better performance scaling, is likely required. Measurements indicate that batching does reduce computation time for intersection queries, however the presented technique poses too much overhead to outweigh the benefits.

Even though this implementation was unable to improve upon the ray/scene performance of Embree, a couple of insights have been gained that can prove as a foundation for future research. Looking into a lock-free approach to parallel batching will probably prove fruitful, since the presented approach faces diminishing returns for additional cores due to synchronization overhead. The importance of high, yet predictable, workloads should be pointed out. The findings in this research indicate that future work with a focus on maintaining high workloads even when the number of available cores increases and a low overhead cost is more likely to succeed in improving upon the current standard.

Acknowledgements

I would like to thank the following people for their support during the project. Firstly, my primary supervisor Dr. Bikker, for his honesty, advice and enthusiastic guidance. Nick Begg for the brainstorm sessions and the good talks over countless cups of 'korte cappuchino'. My second supervisor, Dr. Staals for the provided feedback. My girlfriend, for maintaining her cheerful attitude at times when the project took mine away. My friends and family for the caring support and showing honest interest in my work. And finally T. Gasparian for creating the images in Figure 2 , 3 and 4 which I have gratefully included in my own thesis.

References

[AK10] Timo Aila and Tero Karras. Architecture considerations for tracing incoherent rays. *Proceedings of the Conference on High Performance Graphics*, page 10, 2010.

[BAM13] Rasmus Barringer and Tomas Akenine-Möller. Dynamic Stackless Binary Tree Traversal. 2(1):12, 2013.

[Bik12] J. Bikker. *Ray Tracing for Real-time Games*. PhD thesis, TU Delft, November 2012.

[BWB08] Solomon Boulos, Ingo Wald, and Carsten Benthin. Adaptive ray packet reordering. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 131–138, Los Angeles, CA, USA, August 2008. IEEE.

[EG08] Manfred Ernst and Gunther Greiner. Multi bounding volume hierarchies. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 35–40, Los Angeles, CA, USA, August 2008. IEEE.

[FLPE15] Valentin Fuetterling, Carsten Lojewski, Franz-Josef Pfreundt, and Achim Ebert. Efficient Ray Tracing Kernels for Modern CPU Architectures. 4(4):21, 2015.

[FLPR97] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. page 13, October 1997.

[Gas15] Tigran Gasparian. *Fast Divergent Ray Traversal by Batching Rays in a BVH*. PhD thesis, Utrecht University, Utrecht, Netherlands, December 2015.

[GR08] Christiaan P. Gribble and Karthik Ramani. Coherent ray tracing via stream filtering. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 59–66, Los Angeles, CA, USA, August 2008. IEEE.

[HDW$^+$13] Michal Hapala, Tomáš Davidovič, Ingo Wald, Vlastimil Havran, and Philipp Slusallek. Efficient stack-less BVH traversal for ray tracing. In *Proceedings of the 27th Spring Conference on Computer Graphics - SCCG '11*, page 7, Vinin, Slovak Republic, 2013. ACM Press.

[HPAD] John L Hennessy, David A Patterson, and Andrea C Arpaci-Dusseau. Computer Architecture : A Quantitative Approach. page 705.

[Int19] Intel. Intel TBB Mutex Flavours, 2019. Find at (https://www.threadingbuildingblocks.org/docs/help/tbb_userguide/Mutex_Flavors.html).

[Kee14] Sean Keely. Reduced Precision for Hardware Ray Tracing in GPUs, 2014.

[KK86] L Kay and T Kajiya. Ray Tracing Complex Scenes. 20(4):10, 1986.

[KSS$^+$13] Daniel Kopta, Konstantin Shkurko, Josef Spjut, Erik Brunvand, and Al Davis. An energy and bandwidth efficient ray tracing architecture. In *Proceedings of the 5th High-Performance Graphics Conference on - HPG '13*, page 121, Anaheim, California, 2013. ACM Press.

[Lai10] Samuli Laine. Restart Trail for Stackless BVH Traversal. *High Performance Graphics*, page 5, 2010.

[LGXT17] Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. Vectorized production path tracing. In *Proceedings of High Performance Graphics on - HPG '17*, pages 1–11, Los Angeles, California, 2017. ACM Press.

[MBK$^+$10] Bochang Moon, Yongyoung Byun, Tae-Joon Kim, Pio Claudio, Hye-Sun Kim, Yun-Ji Ban, Seung Woo Nam, and Sung-Eui Yoon. Cache-oblivious ray reordering. *ACM Transactions on Graphics*, 29(3):1–10, June 2010.

[noa] Intel® 64 and IA-32 Architectures Optimization Reference Manual. page 672.

[ORM08] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark. Large ray packets for real-time Whitted ray tracing. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 41–48, Los Angeles, CA, USA, August 2008. IEEE.

[PAC$^+$97] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent RAM: IRAM. page 23, February 1997.

[PKGH97] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. pages 101–108. ACM Press, 1997.

[vA11] Dietger van Antwerpen. Improving SIMD efficiency for parallel Monte Carlo light transport on the GPU. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics - HPG '11*, page 41, Vancouver, British Columbia, Canada, 2011. ACM Press.

[WBB08] Ingo Wald, Carsten Benthin, and Solomon Boulos. Getting rid of packets - Efficient SIMD single-ray traversal using multi-branching BVHs -. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 49–57, Los Angeles, CA, USA, August 2008. IEEE.

[Whi79] Turner Whitted. An improved illumination model for shaded display. *ACM SIGGRAPH Computer Graphics*, 13(2):14–14, 1979.

[WSBW01] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–165, 2001.

[WWB$^+$14] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics*, 33(4):1–8, July 2014.

[YLPM05] Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha. Cache-Oblivious Mesh Layouts. page 8, 2005.

[YM06] Sung-Eui Yoon and Dinesh Manocha. Cache-Efficient Layouts of Bounding Volume Hierarchies. *Computer Graphics Forum*, 25(3):507–516, September 2006.