MASTER'S THESIS | LUND UNIVERSITY 2016

Massively Parallel BVH Construction Using Mini-trees

Erik Nossborn

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-34

Massively Parallel BVH Construction Using Mini-trees

Erik Nossborn
erik@nossborn.se

June 19, 2016

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Per Ganestam

Examiner: Michael Doggett, michael.doggett@cs.lth.se

Abstract

The Bounding Volume Hierarchy is an acceleration data structure used in ray tracing, allowing for faster intersection tests between rays and triangles. The goal of this thesis was to take one algorithm for rapidly generating such data structures, the Bonsai algorithm, which was initially implemented on a CPU, and instead implement it on a graphics processing unit. Many parts of the algorithm go very well together with the massive parallelism of a graphics card. However, the resulting algorithm was considerably slower than the original.

Keywords: BVH, computer graphics, GPU, CUDA, ray tracing

Contents

1 Introduction 3

2 GPU computing concepts 4
- 2.1 CUDA 4
- 2.2 The GPU 4
- 2.3 Warps and blocks 4
- 2.4 Parallel reduction and segmented prefix sums 4
- 2.5 SIMD lanes 5
- 2.6 Unrolling 6
- 2.7 Ballot and popcount for memory calculation 6
- 2.8 Atomic functions and mutexes 7
- 2.9 Thrust 7

3 SweepSAH 8

4 The Bonsai Algorithm 10
- 4.1 Mini tree selection 10
- 4.2 Mini tree construction 10
- 4.3 Mini tree pruning 10
- 4.4 Top tree construction 11

5 GPU implementation 12
- 5.1 Mini tree selection 12
- 5.2 Mini tree construction 13
- 5.3 Mini tree pruning 15
- 5.4 Top tree construction 16

6 Results 17

7 Conclusions 18

1 Introduction

Ray tracing is a common method of generating computer graphics. To calculate the color of a specific pixel, one or more rays are traced through the scene, going backwards along the path light would have taken. Because of this emulation of physical photons, ray tracing makes some effects, like reflections and refractions, much easier to accomplish compared to other popular computer graphics techniques.

While tracing a ray, the scene is searched through, to find which triangles the ray intersects. Because this search is independent for every ray, the process can be massively parallelized.

Searching for intersections between the triangles and the rays can be very time consuming for larger scenes with many triangles, so data structures that allow for faster intersection tests are needed. This is especially relevant for ray tracing in real time, since all calculations have to happen between frames. One such data structure is the *bounding volume hierarchy* (BVH).

A BVH is a tree structure, where the inner nodes describe volumes, and the leaf nodes contain the triangles. The root node describes a volume that contains all the triangles in the scene. The volume of every inner child node is completely contained inside the volume of its parent. The triangles of the leaf nodes are also all inside the volume of the parent.

The advantage of utilizing a BVH is that a large number of triangles can easily be dismissed early on during intersection tests. If a ray is found not to pass through the volume of a particular node, it also cannot go through the volumes of any of that node's children, since the volumes of every child is contained inside the volume of the parent. This, in turn, means the ray cannot intersect any of the triangles at the leaves - they are also all inside the volume. As such, the whole sub-tree can be safely skipped.

If a scene is dynamic, meaning triangles move around from frame to frame, the BVH must be reconstructed at every frame. Otherwise, triangles may move out of the volumes where the BVH expects them to be, making traced rays unable to find proper intersections. As such, a fast BVH building algorithm is necessary. The Bonsai algorithm[1] was created for precisely this situation.

However, the Bonsai algorithm was designed for, and implemented on, a CPU, and there are many advantages to having an algorithm instead run on a GPU.

Many parts of the algorithm can benefit from the massive parallelism of a GPU, and although the original version utilized some parallelism on the CPU through extensions, the parallel capabilities of the GPU are much greater.

If the actual ray tracing takes place on the GPU, having the algorithm on the GPU as well can decrease bandwidth use - the amount of data flowing between the CPU and the GPU. If the BVH is built on the CPU, it must be sent to the GPU for ray tracing every frame. If it is built directly on the GPU however, only updates to the moving parts of the scene have to be transferred on a frame to frame basis, before the BVH is reconstructed.

However, there are also many parts of the algorithm that lend themselves towards sequential processing.

2 GPU computing concepts

2.1 CUDA

CUDA is a language for GPU programming on NVIDIA devices. It mostly uses C/C++ syntax.

2.2 The GPU

A GPU is built to handle thousands of active threads at once. That does not mean that all of them are *executing* at the same time, however. While all the GPU's *streaming multiprocessors* do execute many instructions in parallel, much of the processing power of the GPU comes from the ability to hide latency. The GPU will make context switches regularly, for example while waiting for data transfers and after calls to synchronize threads. This allows some threads to execute while others are unable to, hopefully always keeping the GPU executing relevant instructions. Avoiding unnecessary memory latency is still important, however. Contexts take memory, and with too many slow memory requests, there might not be enough threads available for execution. This will stall execution.

2.3 Warps and blocks

Threads in the GPU are organized into *warps* (CUDA uses 32 threads per warp). Within a warp, the same code is always run for every thread, and the instructions are executed in lock-step (meaning, right after each other).

This means that if different threads within a warp evaluate a condition branch, such as an if-statement, differently, a *divergence* will happen. Both condition branches will run for the whole warp, but any threads not involved in the current branch will pause, and just do nothing when it is their turn to execute. If possible, code should be written so that branch conditions within a warp evaluate the same for all of the threads, or are avoided entirely.

Blocks are collections of warps that execute on the same streaming multiprocessor. Memory within a block can be synchronized between threads relatively easily - sharing memory between blocks is much more difficult, and slow. Sharing memory within a warp is the easiest, requiring the least amount of synchronization.

2.4 Parallel reduction and segmented prefix sums

In the implementation of this algorithm, it is often necessary to search through an array for a specific value, such as the minimum or maximum. On a GPU, *parallel reduction* is used for this purpose. By utilizing the parallel threads of the GPU, several values are simultaneously compared with each other, storing the desired values back into the array. This process is repeated, reducing the number of values by half each iteration, until there

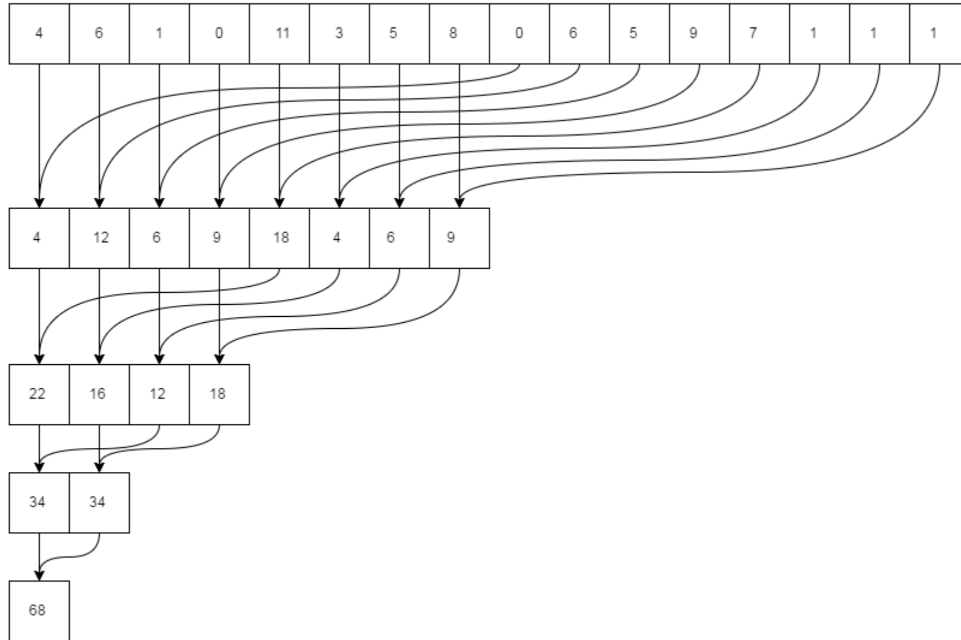

Figure 2.1: A reduction sum flowchart. Memory on a GPU is specialized in handling sequential data as shown in the diagram, so the order is purposeful.

is only one value left. If summing is used instead of comparing, this process will give the sum of all values in the array. (In fact, reduction works for any associative operation.) An example of a reduction sum is shown in Figure 2.1.

The *prefix sum* y of a sequence of numbers x is a sequence where $y_n = \Sigma_{k=0}^{n} x_k$. In other words, it is a running sum - the value at each position in sequence y is the sum of all the previous values in x. When calculating this on a GPU, the numbers can be added as shown in Picture 2.2, but there are also other, more sophisticated methods for larger numbers of threads. As shown by Harris and Garland[2], the special properties of the warp can be utilized to improve segmented sum calculations.

A prefix sum is also known as a *cumulative sum* or a *scan*. Prefix sums can also be *exclusive* rather than *inclusive*, meaning it excludes the current value in the array, only counting up those before it. That is of particular use when calculating memory indices, since the first thread tends to want memory location zero.

2.5 SIMD lanes

SIMD stands for *Single Instruction Multiple Data*. GPUs are specialized at making these kinds of calculations, where many threads execute the exact same operations, but with different values for their operands. A warp is a 32-wide SIMD group.

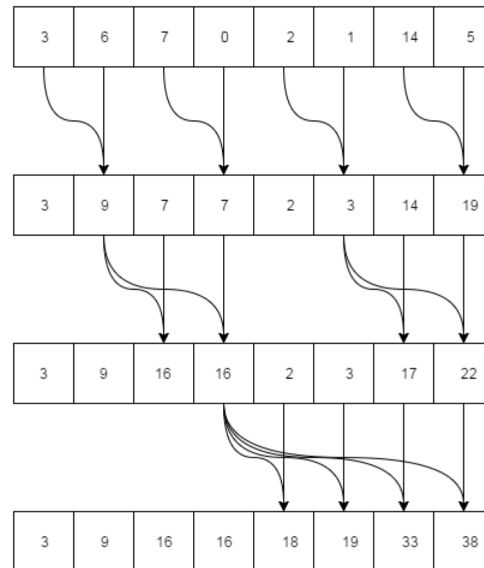

Figure 2.2: Flowchart of a prefix sum. In this chart, four threads can work at the same time, each adding two numbers together. For larger arrays, there are better algorithms.

2.6 Unrolling

When optimizing reductions in CUDA, it is often good to 'unroll' the last iterations of the process; writing them out explicitly instead of including them in a loop. More specifically, this is effective when all the remaining threads are in the same warp. One reason for this is because no block-wide synchronization is needed within a warp and so can be omitted.

2.7 Ballot and popcount for memory calculation

CUDA has a function called the *ballot function*. It takes a boolean expression and returns a 32 bit integer, where the bit at every position `i` is the boolean value of thread `i` within the current thread's warp. This allows all threads in a warp to communicate the results of a particular boolean expression between each other. The *popcount* function simply counts the number of ones in the binary representation of a variable. These two, in combination with bitwise operators, can be used to determine memory locations based on predicates[2], as explained below. This method is used extensively in this implementation.

If threads in a warp want to store a certain value in an array, but only if certain conditions are true, the relevant threads must find out which memory positions they are to write to. The position cannot be the same as any other thread that also wants to write to the array, but it should not leave any gaps between data either. To solve this within a warp, the warp is balloted for the predicate. Then, all bits higher than the index of the current thread are masked out using bitwise and. Finally, the number of ones in the resulting integer is calculated. The result for every thread becomes the number of threads with lower or equal

indices for which the boolean expression evaluated as true. This number is the same as how many threads with lower (and equal) indices want to store something in that array. This lets every thread place their data safely. This is a simplified (only ones and zeros) prefix sum, called a *binary prefix sum*.

2.8 Atomic functions and mutexes

Atomic functions are functions that are guaranteed to be performed in one piece, without reads, writes or cachings from other threads to potentially mess with them. They are used for global synchronisation, but they are very slow.

A mutex, or a *mutually exclusive semaphore*, simply speaking allows you to close off sections of code, perhaps data accesses that shouldn't be simultaneously handled by several blocks. Once a mutex has been taken (using an atomic function), no other thread can take it - and therefore pass into the sensitive code section - until it is released. In CUDA, however, it is up to the programmer to make sure it is always the same thread that locks and unlocks a mutex, and that the code sections really are locked off by the mutex - there are no checks.

Mutexes are even slower than most other atomic functions, since they are blocking - stopping other threads from executing - by design. There are also other problems, for example that mutexes don't stop cached and delayed reads and writes to the data they protect - you still need to synchronise properly in other ways. These things, of course, make mutexes - and global synchronising in general - a last resort.

2.9 Thrust

Thrust is an open source parallel computing library, giving access to several common algorithms that can execute on a GPU. This includes sorting and partitioning.

3 SweepSAH

SweepSAH is a BVH building algorithm that tries to maximize the tree quality by recursively splitting the triangles into groups that together minimize the total *Surface Area Heuristic* (SAH). Tree quality in this case means how good they are for ray tracing - how fast they can be traversed by a ray intersection finder.

Every triangle has an *axis aligned bounding box* (AABB), the smallest axis aligned box that holds the whole triangle within it. For every triangle in the partition we are making a tree out of, we calculate the midpoint of the AABB, and consider the result of that computation the midpoint of the triangle itself. Then, we create three lists of references to all the triangles in the partition. These lists are each sorted by the midpoint coordinates of the different dimensions. The result is three sorted lists of triangles, each along a different dimension.

The recursion starts at the root node. Then, for every node, when we look for a good partitioning, we try to find a low SAH cost by *sweeping* across the triangles in that node. Sweeping means that, for the three lists described above, all partitionings of triangles, as shown in Figure 3.1, are considered. First one triangle on one side, and the rest on the other. Then two triangles on one side, and the rest on the other, and so on. For all of these partitionings, we calculate an approximation of the SAH cost as described below, by calculating the bounding boxes for the two parts. The lowest cost we find is then of the chosen partitions. However, it is possible that not splitting the node at all would give a lower SAH cost. If that is the case, the node is not split, and the recursion ends. The node in question becomes a leaf. If splitting is preferable, however, two child nodes are created. During this partitioning into two child nodes, the sorted orders of the three triangle reference lists are kept intact within the new partitions, so they do not need to be re-sorted. This does however require a stable partitioning. In fact, only two of the three lists need to be sorted, since the split was made along the dimension of one of them. That one is already partitioned. After this, the process is repeated for the new children.

The SAH cost of a node is given by

$$C(n) = \begin{cases} C_I A(n) + C(n_l) + C(n_r), & n \in I, \\ C_T A(n) N(n), & n \in L, \end{cases}$$

where C is the SAH cost, A is the AABB area of a (suggested) node, C_I is the traversal cost of an inner node, C_T is the cost of intersecting a triangle, and N is the number of triangles in a (suggested) node. I and L are inner and leaf nodes, respectively.

Since this definition is recursive, it is impossible to use it as a guide for building the tree. We instead use a non-recursive formula that has been found to approximate the SAH well enough:

$$C(n) = \begin{cases} C_I + C_T \dfrac{A(n_l)N(n_l) + A(n_r)N(n_r)}{A(\text{root})}, & n \in I, \\ C_T N(n), & n \in L, \end{cases}$$

using the same notation as above.

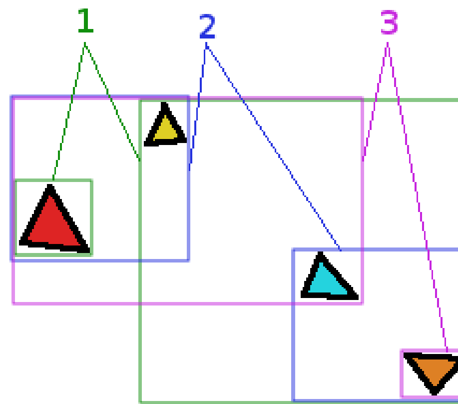

Figure 3.1: Sweep of four triangles from left to right. The total surface area spanned by the axis aligned bounding box of each possible pair of partitions along an axis is calculated during the sweep, repeated for all dimensions. In practice, the right sweep and left sweep are calculated separately, by iteratively adding triangles to the appropriate side. The SweepSAH algorithm uses the two surface areas in every partition to calculate a *surface area heuristic* (SAH) cost, and the lowest score across all partitions in all dimensions becomes the chosen partition.

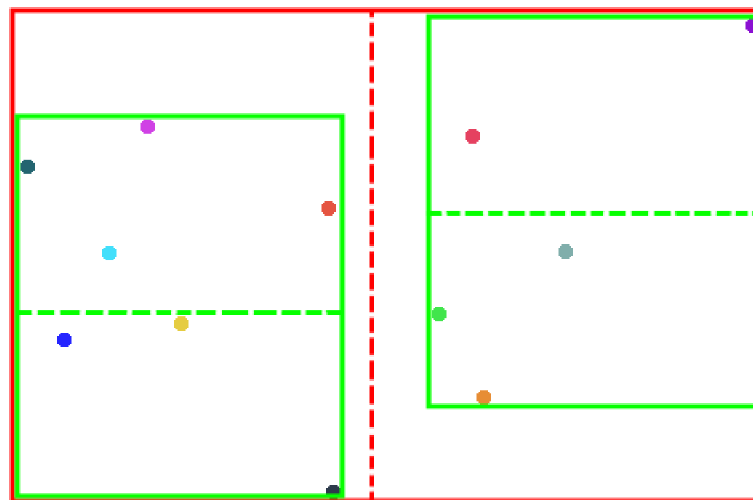

Figure 3.2: Two levels of the segmented partitioning for selecting mini trees, as described in section 4.1. The AABB midpoints of the triangles, represented by the dots in the figure, are used instead of the actual triangles. In each step, the partition is split along the longest axis of the axis aligned bounding box. The process is then repeated for the newly created partitions.

4 The Bonsai Algorithm

In the Bonsai algorithm, the triangles are first quickly partitioned into groups, based on a user-defined size.

Then, every individual group of triangles is built into a *mini tree*, using a more sophisticated - and therefore time consuming - BVH building algorithm, SweepSAH.

When all mini trees have been constructed, a process called *pruning* is performed on all mini trees, in an attempt to improve tree quality (which may have suffered because of how roughly the mini trees were selected).

Lastly, the top tree is constructed with SweepSAH, using the mini trees as leaf nodes.

4.1 Mini tree selection

Like in SweepSAH described above, we calculate the axis aligned bounding boxes of all triangles, and get their midpoints. In this part of the algorithm, the midpoint for every triangle is used as an approximation of the whole triangle. At every split, the total AABB of the partition - based on the midpoints described above - is found. Then, the partition is split in the dimension that has the biggest difference between minimum and maximum values. The splitting plane is chosen to be straight through the middle of the bounding box in this dimension. This process is repeated recursively for the new partitions, until they go below a user-defined number of triangles. This is shown in Figure 3.2 The maximum number of triangles in a finished partition in this implementation was chosen to be 4096. This was one of the values used in the original Bonsai paper.[1] Larger mini trees make the mini tree selection take less time, but increases the time taken for mini tree construction.

4.2 Mini tree construction

The mini trees are built using the SweepSAH algorithm described in section 3.

4.3 Mini tree pruning

In Bonsai pruning, we split mini trees that are deemed unbalanced.

The average AABB surface area for all mini trees is calculated.

Nodes are then recursively checked, starting at the root node. The surface area of each sub tree is compared to the average surface area of all trees. If the area of a node is greater than some user defined fraction T of the average area, the children are also checked. When a sufficiently small node - or a leaf - is found, that node becomes a new root node for a mini tree.

A value of $T = 0.1$ was used in this implementation. This was one of the values chosen in the original paper on Bonsai.[1] Performance-wise, a lower value mostly affects the top tree construction time, as it creates more mini trees to combine.

4.4 Top tree construction

The top tree is also built using the SweepSAH algorithm, with a few differences, since the tree uses mini trees as end points. For example, the recursion only stops when two or fewer nodes remain.

5 GPU implementation

5.1 Mini tree selection

First, the midpoints of all triangles are calculated. The midpoint in this case is defined as the midpoint of the AABB. Using the midpoints means that there is only one reference point to describe each triangle, instead of three. That simplifies many calculations, such as computing the bounding box of the tree. It is the midpoints that are being moved around in this step, and every midpoint has a reference to its original triangle as a fourth value in addition to the three coordinates.

Since global synchronization between blocks is difficult and slow, work is delegated fairly statically, and most of the synchronization is done by returning from kernel. Then, a second kernel can combine the results from several blocks. In fact, for large enough scenes, three levels might be preferred in one of the steps.

All partitions that need further partitioning are handled in parallel. After one such set of partitions has been calculated, the process is repeated for the new set of partitions, until all partitions are complete. The number of blocks used for every partition depends on the number of partitions in the current set of partitions - the more partitions being handled in parallel, the fewer blocks are assigned to every individual partition. This is in order to fill up the GPU, while still trying to avoid assigning too many blocks that will be redundant. A block size of 1024 threads is used in this implementation, since partitions will generally have a large number of triangles. However, depending on hardware, it might be worth considering using 512 threads per block as well, as that size may give a higher occupancy in the streaming multiprocessor. (That is, how full the streaming multiprocessor is. A higher occupancy often means more transactions being processed at the same time, which is good. However, a lower occupancy can often be good enough.)

For every partition step, we must find the AABB of all midpoints in the current partition. This is done through a reduction of bounding boxes. Every bounding box is eight float values. Three coordinates and one padding value each for the minimum and maximum of the bounding box. The maximum values are stored as their negatives, so that all threads can use the minimum function to determine their optimal value. This avoids divergence, since we do not have to conditionally use maximum and minimum evaluations. Eight threads can 'combine' the coordinates of two boxes at the same time, meaning one warp deals with four pairs of boxes at a time. Since the number of coordinates is often high, a second kernel is launched to collect the results, and sometimes even a third after that if multiple blocks are needed to gather the results.

Once the partition-wide bounding box has been calculated, it is time for the actual partitioning. The split is done along the longest axis of the bounding box. The mid-plane of that axis is determined, and triangles are chosen to get partitioned based on which side of the plane their midpoint belongs to. The partitioning happens in two steps: First, the midpoints are partitioned block-wise, using a segmented sum. The ballot-popcount method described in section 2.7 is used. The resulting partitions are stored in a buffer. The cut-off indices between the partitions are also stored. Then, another segmented sum is calculated, this time of every block's cut-off point. Using this sum, and the cut-off points, the midpoints are stored back into the original array, now partitioned. See Figure 5.1. Once the partitions are done, they are ready for another potential split.

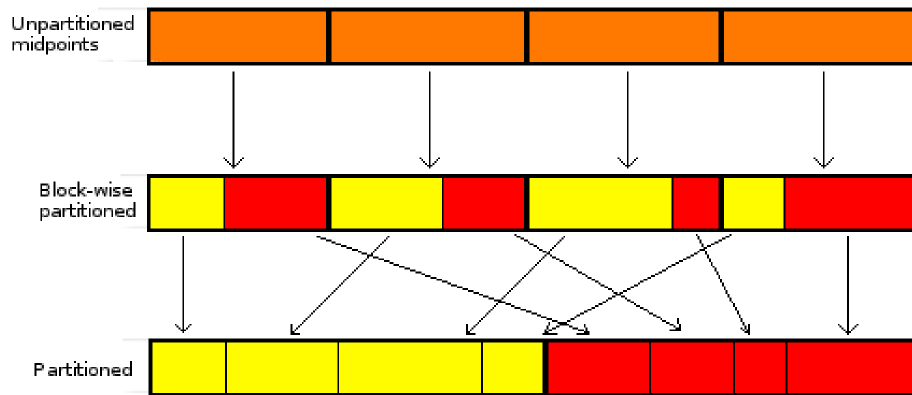

Figure 5.1: The partitioning implementation. In a first pass, a block-wide partitioning is done individually for every block, using segmented sums to determine new locations for all values. Then, a new segmented sum, this time for the partitioned blocks, is calculated, and this is used to transfer the values back to the first array, partitioned.

When a partition reaches a user-defined number of triangles, it will no longer split. In this implementation, a maximum tree size of 4096 was used.

5.2 Mini tree construction

The SweepSAH algorithm needs to have access to the triangles in the mini tree, sorted by their AABB midpoints, for all three dimensions. As such, we have three vectors, one for each dimension, with references to the triangles. These vectors are sorted for each tree. This is done using the Thrust library. Unfortunately, this requires sending some data (in particular, the locations of the different mini trees in memory) between the GPU and the CPU. With a well constructed sorting algorithm called from the kernel itself this overhead would not occur. However, Thrust is a very powerful library, so the overhead might very well be preferred in many cases.

Every tree is initially built independently by a block of 8 warps (although other powers of two are possible). This number gives a decent work load distribution. These initial nodes are stored statically, based on branch and depth of recursion. This is to avoid unnecessary communication between blocks.

As the tree branches, the warps are designated their own sub trees. As such, eventually each warp has its own part of the tree, which it has full independent responsibility for. At this point, the warp uses a queue to handle the nodes dynamically. This allows for less communication between warps. This way, the work load can be more evenly distributed, provided enough sub trees have been generated. After the initial static recursions, when the sizes of all sub trees are known, they are given storage proportional to the original size of the tree. This ensures that the bigger sub trees of unbalanced trees will not cause overflow into memory used by different sub trees as they add more nodes to their branch.

The actual node vector is in fact the queue as well - new nodes are added on at the end, and every node is processed in turn, from oldest to newest. The process stops when no more nodes are available.

The sweep is performed for each dimension consecutively. It would be possible to parallelize the sweeps of different dimensions, since they do not affect each other until the final SAH value has to be decided. This was however not done in this implementation. The sweep in every dimension is also done in two steps. First, the sweep goes one way, and stores the calculated areas for every triangle partitioning from one side. Then the sweep goes the other way, calculating the areas for the other side of the partitioning. It is also during this backsweep that the SAH costs are calculated using the formula in Section 3, since it is at that point the areas of both partitions are known.

During the sweep we want to calculate the SAH cost for every partitioning along the axes. Therefore, the bounding boxes are calculated using the same technique as segmented prefix sums, except for combining bounding boxes instead of summing up values. This will give us the cumulative boxes for every partitioning. Since warps have 32 threads and a bounding box has eight values (minimum and maximum for x, y, z, and a padding w), four boxes can be 'added' with four other ones in one warp-wide operation. Even though not having any padding would allow for ten boxes to be handled at once, since five is just above a multiple of two, no time would really be saved- there would be an 'extra' box that would have to be handled while the other threads are waiting. (To see this, imagine there were two more numbers in the flowchart from Figure 2.2. They would always have to be tacked on at the end, or somewhere in the middle.) Furthermore, properly aligned shared data is faster to access - four boxes fit perfectly into 32 consecutive four-byte words.

As was done in the mini tree selection, the maximum values are stored as their negatives, for the same reasons - avoiding divergence.

After a set of eight bounding boxes have been fully calculated, their surface areas are calculated and stored. A new set of bounding boxes are loaded in, and the last bounding box of the previous iteration is added to the first bounding box in the current one, so that the chain continues. Then, the process described above repeats, until the full prefix bounding is complete.

During the backsweep, the same process occurs, except in reverse. After each set of eight bounding boxes have been calculated, the SAH values for these eight partitions can be calculated, using the current data and the stored areas from the other sweep direction. Eight threads keep one SAH value each - together with the corresponding pivot index and dimension, and save any lower values they come across. After the sweep, the final eight values are reduced and the final SAH value, together with the dimension and pivot index, is determined.

At this point, the SAH cost of not splitting the node is calculated. If not splitting is better, and provided that the resulting node would not be too big (64 triangles was used as the minimum for a tree in this implementation), it is declared a leaf node together with all the triangles it contains, and marked as such. During the static portion, this means the branch will not continue, and it (and potential children) should be marked, to avoid trying to calculate a node that does not exist. In the dynamic portion, having no children simply means that no new nodes will be added to the queue by this node.

Now the new partitions have been decided, so we move on to the actual partitioning. During this process, a boolean vector is used to mark which triangles should be 'moved'

where. The triangles are in fact not moved during this stage, the partitioning is of the three sorted vectors that in turn point at the triangles. One vector, S, is already partitioned, on account of being of the dimension the split happened in. It is this vector that is used to fill the marking vector - any elements pointed at from S_i where $i < pivot$ end up in one partition, and the rest end up in the other. Just like during the mini tree selection, the ballot function can be used to calculate memory locations. Unlike in the mini tree selection, the partition needs to be stable - the vectors must keep their sorted order in each of the new partitions - but luckily we also know the exact sizes of both partitions beforehand. Also, since every warp is working independently, there is no thread communication needed outside of the ballot function during this stage. Like sweeping, the two dimensions can be calculated in parallel using two or more warps, but this is not done in this implementation.

At this point, the two child nodes are created, based on the information from the parent together with the pivot. The nodes are either inserted in the proper static position, or added to the queue for the dynamic part. Once a node gets children, the variables that used to describe the range of triangles under their care now instead point to the two children. The sizes of the new nodes (which eventually will mean the total number of triangles in all the leaf nodes beneath them) are also saved at this point. Node sizes are used during the top tree construction, and this happens to be a convenient time to compute them.

5.3 Mini tree pruning

Pruning consists of two simple kernel calls - calculating the average area of all original mini trees, and the actual pruning.

Calculating the average tree area is a simple addition reduction of calculated surface areas (all nodes already contain their own bounding box, including the mini tree roots) followed by a division by the total number of trees.

Only one warp per tree is used for pruning. More isn't really needed - one thread can check a node by itself, so you get minimal benefit unless you have over 32 pending nodes to check at the same time. Even then, it would require more synchronisation.

A stack consisting of indices to nodes is used for storing pending work, and nodes are handled up to 32 at a time. If their surface area is too big, and if they aren't already leaves, they should be pruned. This means adding their two children to the stack, effectively removing the current node from the system. If they should *not* be pruned, they are deemed a new mini tree root node, which means they should be added to a global list of roots. The surface area threshold is some constant T times the average tree area calculated above. $T = 0.1$ is used in this implementation.

Once again, warp balloting is used for calculating memory positions. We are helped by the fact that a pruned node always leaves two children in the stack - never just a single one - making the memory allocation fairly regular. The non-pruned nodes are initially stored in a shared vector. At the end, an atomic add operation is used to reserve space in the global array, after which the block local roots are copied to the reserved space.

5.4 Top tree construction

The top tree construction is very similar to the mini tree one, except it uses the newly pruned mini trees as 'leaves'. One difference is that the recursion only stops when there are two or fewer nodes, since inner nodes have exactly two children. Sometimes during this process, a node will end up only having one mini tree root left to process, but in this case, the current node just copies all the information from the only child, effectively "becoming" the child while still remaining connected to their own parent, effectively collapsing the two into one. Another difference is that a cumulative sum of the total number of triangles is calculated with the backsweep. This, of course, is because the SAH approximation uses the number of triangles in every partitioning. Using the number of nodes (that is, the number of mini trees) instead leads to imbalanced trees, since higher density nodes get undervalued.

One last difference is that since there is only one top tree, we can throw as many resources as we can and need on just building this tree.

6 Results

The implementation was done using an NVIDIA GeForce GTX580, with Compute Capability 2.0 and Fermi architecture. It has 16 Streaming Multiprocessors and 32 cores handling one SIMD group on every processor.

The comparisons were made towards the equivalent algorithm implemented on the CPU by Ganestam et al.[1].

The test scene was *Conference Room*[3], with 331179 triangles. Conference Room is a very common scene for testing ray tracing.

Midpoint calculation	0.169 ms
Partitioning	16.3 ms
Sorting mini trees	417 ms
Tree sweep 1	43.3 ms
Tree sweep 2	52.6 ms
Average tree area	0.010 ms
Pruning	0.032 ms
Top midpoints	0,010 ms
Sorting top trees	3.08 ms
Top tree sweep	7.60 ms

The implemented algorithm was significantly slower than the original CPU version, which finished the whole equivalent task in less than 20 ms.

The sorting of the mini trees is especially slow. However, that sorting is done through several calls to the Thrust library, which uses generic sorting, and the different calls may not be well parallelized. A specialized sorting, or an optimization of the thrust execution, would result in a significantly lower time.

It is possible that the algorithm uses too many global memory accesses that are spread out, resolving to separate requests rather than being coalesced. This could result in the GPU stalling, waiting for data from many locations to arrive while being unable to process anything else in the meantime.

7 Conclusions

While some parts of the SweepSAH algorithm lend themselves to parallelism, other parts do not. It is possible that for a GPU implementation, other BVH algorithms might be better matches to combine with partitioning and pruning. Those two processes are relatively separate from the rest of the algorithm, and can therefore be applied in other situations.

Since this implementation was made on slightly older hardware, it is possible that newer hardware would do much better. In particular, the algorithm does much work on individual warps, and often requires communication using shared memory. In NVIDIA's newer GPUs, it is possible to use the *thread shuffling* intrinsic, which allows threads within the same warp to exchange data very effectively.

It would possibly be beneficial to create the sorted arrays used during the mini tree construction before the mini tree selection, and then only using stable partitions to maintain the order. That way, during mini tree selection, calculating the AABB at every recursion is an $O(1)$ operation, because the first and last values in the partition of a sorted array define the bound for its corresponding direction. It would be fast to find the exact sizes of the new partitions. Sorting would not be required before mini tree construction, as the proper order has been preserved. For this implementation in particular, this means that no data has to exchanged between the GPU and CPU at that point in the algorithm. The downsides are that there are more arrays to partition, and that a stable partitioning is possibly more time consuming to perform. Also, it is generally slower to sort one long array than it is to sort many short ones.

Bibliography

[1] Per Ganestam, Rasmus Barringer, Michael Doggett and Tomas Akekike-Möller, *Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees*, 2015

[2] Mark Harris and Michael Garland, *Optimizing Parallel Prefix Operations for the Fermi Architecture*, 2012

[3] *Conference Room*. Original scene by Anat Grynberg and Greg Ward, recreated by Kenzie Lamar. Fetched 2015-08-18 from http://graphics.cs.williams.edu/data/meshes.xml

EXAMENSARBETE Massively parallel BVH construction using mini-trees

STUDENT Erik Nossborn
HANDLEDARE Per Ganestam (LTH)
EXAMINATOR Michael Doggett (LTH)

Parallelliserat byggande av datastruktur för avancerad datorgrafik i realtid

POPULÄRVETENSKAPLIG SAMMANFATTNING **Erik Nossborn**

Med mer kraftfulla grafikkort har möjligheterna utökats för vilka grafikmetoder som kan utföras i realtid. För en sådan metod, raytracing, används speciella datastrukturer, och för att kunna användas i realtid behöver dessa byggas upp snabbt.

I *raytracing*, eller *strålföljning*, skapas bilder genom simulering av ljusstrålar. Genom att ta reda på varifrån de ljusstrålar som man tänker sig har skapat bilden kommer ifrån, kan man räkna ut vilka färger ljusstrålarna - och därmed pixlarna i bilden - borde ha. Denna metod är speciellt bra på att skapa trovärdiga reflektioner, refraktioner, och skuggor.

Metoden kräver dock en stor mängd uträkningar för varje bild. Raytracing används därför vanligtvis endast i till exempel animerade filmer, där tiden för uträkningarna inte är kritisk. På senare tid har det dock gjorts framsteg när det gäller att kunna använda metoden i bilder som ska räknas ut i realtid, till exempel i datorspel.

För att snabbt kunna hitta exakt vilka objekt en ljusstråle har träffat, används speciella datastrukturer. En BVH (Bounding Volume Hierarchy) är en vanlig sådan. Det är en trädstruktur, där varje nod beskriver en del av volymen som utgör scenen. Varje barns volym är helt innesluten i förälderns volym, så att ju längre ner i trädet man kommer, desto mindre blir volymerna. Trädets löv består av trianglarna själva - även dessa helt inneslutna i förälderns volym.

Detta gör det möjligt att snabbt söka genom ett stort antal trianglar. Om en ljusstråle inte passerar genom en viss nods volym, så kommer den inte heller passera genom någon av barnens volymer, och därför inte heller genom någon av de trianglarna längre ner i den delen av trädet. Hela grenen kan därför ignoreras utan problem.

I detta arbete har jag tagit en LTH-utvecklad algoritm, *Bonsai*-algoritmen, som på en vanlig processor snabbt bygger BVHer av hög kvalitet, och implementerat den på ett grafikkort. Grafikkort är gjorda för att kunna behandla stora mängder data parallellt, vilket kommer till nytta under många delar av algoritmen. Bonsai-algorithmen delar till exempel upp trianglarna i flera grupper som oberoende av varandra kan byggas upp till små träd (en av anledningarna till att algoritmen kallas för "Bonsai"), och även varje individuell gren av desa träd är separat från varje annan gren. Eftersom själva raytracingen oftast utförs på grafikkortet, är det även en fördel om BVHn inte behöver skickas över till grafikkortet inför varje bilduppritning.

Det finns dock många delar av algoritmen som inte lika lätt lämpar sig för parallellisering, och slutresultatet gav i helhet ingen förbättring i tidsanvändning.