# Binned SAH Kd-Tree Construction on a GPU

Piotr Danilewski[1], Stefan Popov[1], Philipp Slusallek[1,2,3]
[1]Saarland University, Saarbrücken, Germany
[2]Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany
[3]Intel Visual Computing Institute, Saarbrücken, Germany

Contact: danilewski@cs.uni-saarland.de

June 2010

## Abstract

In ray tracing, kd-trees are often regarded as the best acceleration structures in the majority of cases. However, due to their large construction times they have been problematic for dynamic scenes. In this work, we try to overcome this obstacle by building the kd-tree in parallel on many cores of a GPU. A new algorithm ensures close to optimal parallelism during every stage of the build process. The approach uses the SAH and samples the cost function at discrete (bin) locations.

This approach constructs kd-trees faster than any other known GPU implementation, while maintaining competing quality compared to serial high-quality CPU builders. Further tests have shown that our scalability with respect to the number of cores is better than of other available GPU and CPU implementations.

## 1 Introduction

Ray tracing is a method of generating realistically looking images by mimicking light transport, which involves traversing a ray through the scene and finding the nearest intersection point with the scene triangles. A naive algorithm iterating through all triangles in the scene in order to find the intersection point would be prohibitively slow. That is why various acceleration structures have been designed: grids, octtrees, bounding volume hierarchies (BVH), and kd-trees to name a few. In most cases, kd-trees are the best in terms of ray tracing speedup, but are computationally expensive to construct.

To overcome this problem, we are interested in building the kd-tree in parallel on a GPU using the general-purpose CUDA programming language. A GPU is a powerful, highly parallel machine and in order to be used effectively, the program must spawn tens of thousands of threads and distribute the work evenly among them. That is the main challenge for the design of our algorithm.

Our kd-tree algorithm builds the tree in top-down, breadth-first fashion, processing all nodes of the same level in parallel with each thread handling one or more triangles from a node. The program consists of several stages each handling

nodes of different primitive counts and distributing work differently among all cores of the GPU in order to keep them all evenly occupied.

At all stages we are using the Surface Area Heuristic to find the splitting planes and never revert to simpler approaches, like splitting by the median object. While these simpler approaches perform faster and make it easier to distribute the work evenly between processors, they also produce trees of lower quality.

The result of our work is a highly-scalable implementation which can construct the kd-tree faster than any other currently existing program, while its quality remains comparable to the best non-parallel builders.

In the following sections we first focus on previous approaches for kd-tree construction. We describe the Surface Area Heuristic and how the kd-tree construction algorithm is mapped onto multiprocessor architectures. Later we present our algorithm, followed by more detailed focus on practical implementation on a GPU using CUDA environment with primary focus on work distribution, synchronisation, and communication between processors. Finally, we put our program to test and present the results both in terms of construction speed and quality.

# 2 Previous Work

Originally kd-trees were designed for fast searching of points in k-dimensional space [Ben75] (hence the name), but rapidly they found their way in the field of ray tracing [Kap85].

In this section we first describe the most common approach for kd-tree construction for ray tracing which uses the Surface Area Heuristic (SAH). Later we describe various attempts to parallelise the algorithm.

## 2.1 The Surface Area Heuristic

Kd-trees are typically built in a top-down manner. Each node is recursively split by an axis-aligned plane creating two child nodes. The position of the plane is crucial for the quality of the whole tree. MacDonald and Booth [MB90] defined a good estimation of the split quality by means of a cost function:

$$f(x) = C_t + C_i \cdot \frac{SA_L(x) \cdot N_L(x) + SA_R(x) \cdot N_R(x)}{SA_{parent}} \tag{1}$$

Where the particular components are:

- $C_t$ - cost of traversing an inner node,
- $C_i$ - cost of intersecting a triangle,
- $SA_L, SA_R$ - surface area of left/right bounding box
- $N_L, N_R$ - number of primitives on the left-/right side,
- $SA_{parent}$ - surface area of the parent bounding box

The above function is linear for all values of $x$ with an exception when $x$ is a lower or upper bound of a triangle. Therefore it suffices to compute the value of the cost function at those points to find the minimum. This approach is known as *exact Surface Area Heuristic* construction (*exact SAH*) and it builds kd-tree in $O(n \log n)$ on a sequential machine as shown by Wald and Havran [WH06].

A significantly faster approach is to compute the cost on a few sampled positions, which reduces quality only slightly. This *sampled SAH* approach has been first proposed by MacDonald and Booth [MB90]. A more extensive study has been done by Hunt et al. [HMS06]. Popov et

al. described an efficient algorithm for computing the cost using bins (hence *binned SAH*) in [PGSS06].

The SAH can be also used to determine when not to split the node anymore. The cost of a node which is not being split is:

$$f_0 = C_i \cdot N \tag{2}$$

If $f_0 < f(x)$ for all $x$ in the domain, we do not split. The values $C_i$ and $C_t$ influence the quality of the produced tree and its construction times. Optimum values depend heavily on the ray tracer, hardware, and algorithms used.

### 2.2 Towards Parallelism

In recent years we have been observing a major change in hardware development: the speed of a computing unit no longer grows as fast as in the past, instead we get more and more cores within the processor. This new architecture requires adjustments in the program to fully take advantages of the cores. The most extreme example is a GPU which is capable of running thousands of threads in parallel.

One notable approach to parallel kd-tree construction was done by Shevtsov et al. [SSK07] who implemented *binned SAH* kd-tree builder for multicore CPUs. In their work, the whole scene is first partitioned into $p$ clusters, where $p$ is number of cores. Each cluster has approximately the same number of primitives. Later on, each cluster is processed independently, in parallel to all other clusters.

Popov et al. [PGSS06] also introduced parallelism in their implementation but it is used only for deeper stages of the kd-tree construction, with each processor handling different subtree. At initial steps with only few nodes, the construction is done in sequence on a single CPU. On the other hand they use SAH on all levels of the tree. Choi at al. [CKL*09] gave completely parallel SAH algorithm which does include the initial steps as well.

The above approaches work well as long as the number of cores is relatively small. While they map well to multicore CPUs, they do not scale well enough to saturate a GPU.

The first and only known to us algorithm for GPU kd-tree construction was given by Zhou et al. [ZHWG08]. Their algorithm is divided into two stages: big-node and small-node. During the big-node stage either a part of the empty space is cut off, or a spatial median split is chosen. Only later, when there are less than 64 primitives per node, the exact SAH algorithm is used.

To sum it up, so far nearly every algorithm for kd-trees is either sequential at the beginning or sacrifices the tree quality by using median splits at top levels (or both). In order to obtain good kd-trees it is important to use the SAH at top levels of the tree as well. Using median splits instead may reduce rendering performance significantly[Hav00].

## 3 The Algorithm

We are constructing the kd-tree in top-down, breadth-first fashion. The algorithm works in *steps*, one for each level of the tree. Each step consists of 3 basic *phases* which are:

- *Compute-Cost* phase. Here we search for the optimal split plane using the sampled SAH for all active nodes (that is – for all leaf nodes that need to be processed). We also compute how many primitives are on each side of the plane and how many primitives must be duplicated.

- *Split* phase. As the name suggests, we then split the node and create two children. In this phase we allocate memory and rearrange triangles.
- *Triangle-splitter* phase. Some triangles may partially lie on both sides of the plane. In order to increase the quality of the subsequent splits, we compute the exact size of two bounding boxes, each encapsulating the respective part of the triangle (Figure 1). This is an important step which may increases resulting rendering performance by over 30% [Hav00].

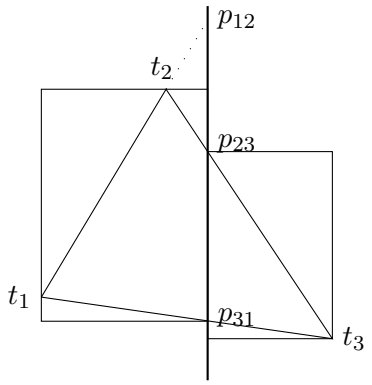


Figure 1: 2D example of a triangle split. Left bounding box is span over $\{t_1, t_2, p_{23}, p_{31}\}$, right bounding box over $\{p_{23}, t_3, p_{31}\}$

At the *Compute-Cost* phase and the *Split* phase we perform all operations on axis-aligned bounding boxes of the triangles only. *Triangle-splitter* phase is the only one where we access vertices directly. That is why we keep the triangle bounds in separate arrays and the first two phases work on those arrays only.

To keep a good utilisation of the GPU, the whole program is divided into several *stages*, each with different implementation of the phases, suited for different triangle counts in a node and different node count in the tree level. If given node does not meet the requirements for given stage it is scheduled for later processing. The implementation and differences between the stages are discussed in Section 4.

### 3.1 Compute-Cost

The first task in each step of the construction is to find an optimal split plane. We are using binned SAH approach with $b = 33$ bins and $c = b - 1 = 32$ split candidates. For each split candidate we need to compute the surface area cost as given by Equation 1 and to that end we need to know how many primitives are on each side of the plane. We are following the algorithm introduced by Popov et al. in [PGSS06].

We are interested in the position of the bounding boxes of all triangles along a given axis — the bounding box's lower and higher coordinate, which we refer to as lower and higher *events*.

We create 2 arrays of $b$ bins (See Figure 2 and 3, lines 6-7). One bin array ($L$) collects lower events and the other ($H$) collects higher events (lines 10-17). Bin $i$ stores an event occurring between the $i-1$-th and $i$-th split candidate.

The next step is to compute suffix sum on $L$ and prefix sum on $H$ arrays. After that, for a given split candidate $i$ we can immediately compute the number of primitives entirely on the left and right side and the number of primitives which are partially on both sides (lines 25-27) using the following formula:

$$N_{only\_left} = H[i]$$

$$N_{only\_right} = L[i+1]$$

$$N_{both} = N_{total} - L[i+1] - H[i]$$

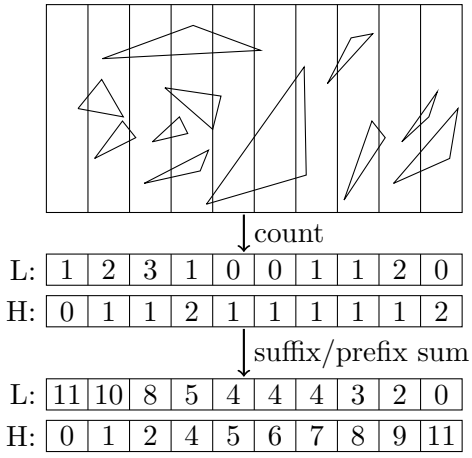


Figure 2: Example run of the binning algorithm: we collect lower and higher bounds of triangles in $L$ and $H$ arrays. This is followed by suffix sum on $H$ and prefix sum on $L$. At the end we can instantly get the number of primitives on each side of every split candidate.

In addition, at this phase we compute the no-split cost (Equation 2). We may also decide that the node contains too few triangles for given *stage* in which case we will process it only later, at one of the next *stages* (lines 2-4).

### 3.2 Split

At this point we know for every node if and where we want to split it and how many primitives are located on each side of the split plane. In this phase we want to rearrange the triangle bounding boxes so that all entries of a given node always occupy a single continuous part of memory. We do not rearrange triangle vertices as they are only accessed for a few primitives and only in the *Triangle-splitter* phase.

By performing a parallel prefix sum on the tri-

```
1 function computeCost(node)
2 if (node.triangleCount<minimumNodeSize)
3   result.{x,y,z}.cost:=infinity
4   return noSplit
5 for dim in {x,y,z} do
6   L[0..33]:=0
7   H[0..33]:=0
8   synchronize
9   triangleIdx:=node.trianglesBegin
        +threadIdx
10  while (triangleIdx<node.trianglesEnd)
11    lowBound:=BBox[triangleIdx].dim.low
12    binIdx:=chooseBin(lowBound)
13    atomic {L[binIdx]+=1}
14    highBound:=BBox[triangleIdx].dim.high
15    binIdx:=chooseBin(highBound)
16    atomic {H[binIdx]+=1}
17    triangleIdx+=threadCount
18  synchronize
19  suffixSum(L)
20  prefixSum(H)
21  cost[0..32]:=SAH(L,H,0..32) //Equation 1
22  i:=pick{i∈{0..32} : cost[i] is minimal}
23  result.dim.cost:=cost[i]
24  result.dim.splitPosition:=
        splitPosition(i)
25  result.dim.leftCount:=H[i]
26  result.dim.rightCount:=L[i+1]
27  result.dim.bothCount:=T–L[i+1]–H[i]
28 result.noSplit.cost:=SAH(noSplit) //Equation
       2
29 return pick{i∈{x,y,z,noSplit} :
       result.i.cost is minimal}
```

Figure 3: Compute-Cost phase

angle counts of each node we can allocate memory where we will move the entries. For each node three memory locations (*base pointers*) are computed — $B_L$ for left primitives, $B_R$ for right primitives, and $B_D$ for primitives falling into both child nodes. These references to primitives need to be duplicated and bounding boxes recomputed (Section 3.3)

If a node is not to be split, the data is moved to the *auxiliary array* to be processed in the next stage (Figure 4 lines 4-8). If we do split the node, we move data to the *destination array* using the *base pointers*.

```
1 function
      split(node,dim,splitPlane,B_A,B_L,B_R,B_B)
2 triangleIdx:=node.trianglesBegin+threadIdx
3 p:=splitPlane
4 if (dim = noSplit)
5   while (triangleIdx<T)
6     auxiliary[B_A+triangleIdx]
          :=source[triangleIdx]
7     triangleIdx+=threadCount
8   return
9 while(triangleIdx<node.trianglesEnd)
10  Off_L[0..threadCount-1]:=0;
11  Off_R[0..threadCount-1]:=0;
12  Off_D[0..threadCount-1]:=0;
13  low:=source[triangleIdx].dim.low;
14  high:=source[triangleIdx].dim.high;
15  if (low<p ∧ high<p) type:=L
16  if (low<p ∧ high≥p) type:=D
17  if (low≥p ∧ high≥p) type:=R
18  Off_type[threadIdx]:=1
19  synchronize
20  prefixSum(L)
21  prefixSum(R)
22  prefixSum(D)
23  synchronize
24  destination[B_type+Off_type-1]
        :=source[triangleIdx]
25  B_L+=Off_L[threadCount-1]
26  B_R+=Off_R[threadCount-1]
27  B_D+=Off_D[threadCount-1]
28  triangleIdx+=threadCount
```

Figure 4: Split phase

To correctly copy the primitives we introduce 3 offset arrays $Off_L$ (left), $Off_R$ (right) and $Off_D$ (duplicated) (Figure 4, lines 10-12). Each thread $t$ handles one primitive. Depending on the location of the primitive, one of cells $Off_L[t]$, $Off_R[t]$, or $Off_B[t]$ is set to 1, the other two are set to 0 (line 18). Finally we perform a prefix sum on the arrays (lines 20-22) and we move the bounding box data to the cell indexed by the *base pointer* incremented by the offset (line 24).

### 3.3 Triangle-Splitter

The last phase is dedicated to handling triangles that have been intersected by the split plane. Since only a small portion of triangles is being intersected (usually around $\sqrt{N}$), first we need first to assign threads to the triangles in question. This can be achieved by a single parallel prefix sum over the number of middle triangles of each active node, computed at *Compute-Cost* phase (Figure 3, line 27).

Consider the triangle ($t_1$, $t_2$, $t_3$) which is intersected by a plane (Figure 1). The algorithm projects a line through each pair of vertices $(t_1, t_2)$, $(t_2, t_3)$ and $(t_3, t_1)$ and finds the intersection points with the split plane ($p_{12}, p_{23}, p_{31}$ respectively). If an intersection point does not lie between the triangle's vertices it is discarded and not considered further. We then span two bounding boxes around the remaining intersection points and around vertices which are on the correct side of the split plane. Finally, the obtained boxes are intersected with the previous bounding box, which could be already smaller due to some previous splits of the triangle.

In some rare cases the resulting bounding box may not be the tightest box covering the fragment triangle, however our approach is less demanding on memory, simpler to compute and maps better to the CUDA architecture than more precise algorithms, e.g. Sutherland-Hodgeman algorithm [SH74].

## 4 Implementation details

So far we have discussed the general algorithm for our binned SAH kd-tree construction. Now we are focusing on its efficient implementation in CUDA. CUDA is a new C-like programming environment [NBGS08], the code of which can be compiled and run on all newest NVIDIA graphics hardware.

Execution of a CUDA code is split into *blocks*

with GPU usually capable of running several of them in parallel in MIMD fashion. Each block may run independently of all others, and communication between blocks is slow, handled through main GPU memory. Each block consists of several threads, all of which are executed on a single Stream Multiprocessor (SM) of the GPU. Communication and synchronisation between the threads of a single block is much faster thanks to additional on-chip memory and a barrier primitive. Threads of a block are bundled into 32-thread groups called *warps*. All threads of the same warp execute in a SIMD fashion.

In our implementation we always launch as many blocks as they can run in parallel on a GPU. Each node, after performing the task it was assigned to, checks if more nodes need to be processed, and if that is the case, it restarts with the new input data. This idea of *persistent blocks* follows the concept of *persistent threads* coined by Aila et al. in [AL09].

### 4.1 Program stages

The efficiency of the different parallel implementations of the algorithm's components depends on how many primitives are contained in the nodes and how many nodes are there to be processed. To reach maximum speed our program consists of five different implementations and performs the kd-tree construction in five stages: huge, big, medium, small, and tiny (see Table 1)

- *Huge* stage is used at beginning of the construction when very few nodes are processed, but these contain many triangles. At this stage several blocks handle a single node.
- *Big* stage is used when there are enough nodes to saturate the whole GPU with every block working on a different node.
- At *Medium* stage we launch smaller blocks handling smaller nodes. Because of reduced resource demand from the block more of them can run in parallel on a single multiprocessor.
- *Small* stage blocks process 4 nodes at once. Each node is handled by a single warp (32 threads).
- *Tiny* stage handles very small nodes consisting of only few primitives. At this stage an *exact SAH* approach is used and single blocks handle several nodes at a time.

### 4.2 Big stage

At the huge and big *stages* we are interested in nodes consisting of at least $\mathcal{M} = 512$ primitives (see Table 1). This particular value is the maximal number of threads that a block can hold. We first focus on the big stage which is simpler in its construct than the huge stage.

#### 4.2.1 Big Compute-Cost kernel

For the *Compute-Cost* kernel 3 blocks handle a single node with each block binning along only one of the axes, so that the main loop (Figure 3 line 5-27) is distributed between the blocks. Only the last instruction (line 29) requires combining the results and it is achieved by moving the task to a separate kernel.

A major performance hazard of the Compute-Cost algorithm, as it is described in Section 3.1, are the atomic operations (Figure 3, lines 13 and 16). In the worst case scenario, if all events fall into the same bin this may lead to a complete serialisation of the execution.

| Stage | Node size | th/bl | | $\frac{bl}{SM}$ | $\frac{N}{bl}$ | $\frac{N}{SM}$ |
|---|---|---|---|---|---|---|
| | | CC | Split | | | |
| Huge | >512 | 512 | 512 | 2 | <1 | $\leq 1$ |
| Big | >512 | 512 | 512 | 2 | 1 | 2 |
| Medium | 33-512 | 96 | 128 | 8 | 1 | 8 |
| Small | 33-512 | 128 | 128 | 8 | 4 | 32 |
| Tiny | $\leq 32$ | 512 | 512 | 2 | >14 | >28 |

Table 1: Configuration differences between the stages. "Node size" is the number of triangles that a node must contain. 'th/bl' is the number of threads in each block of CC (Compute-Cost) and Split phases. $\frac{bl}{SM}$ is the number of blocks that can run in parallel on a single Stream Multiprocessor (SM), $\frac{N}{bl}$ is the number of nodes a single block handles in parallel, and $\frac{N}{SM}$ — number of nodes processed in parallel by a single SM on GTX 285

To reduce the number of collisions we introduce an expanded version of the array, consisting of $\mathcal{W} = 32$ counters per bin, which is the size of a *warp* — the CUDA SIMD unit. With the help of the expanded bin array we are guaranteed not to have any collision between threads of the same warp.

Due to memory limitations on the multiprocessor chip, we keep only one expanded array. First we read all lower *events* (Section 3.1) in the loop (Figure 3 lines 11-13) and store them in the expanded array. Then we compress the array to the simple one-counter-per-bin representation, copy the results and reuse the expanded array for higher events (lines 14-16).

#### 4.2.2 Big Split kernel

For the big split phase we assign exactly one block per node consisting of $\mathcal{M} = 512$ threads. Because of the block size, each element of the offset arrays cannot exceed $\mathcal{M}$ after the prefix sum has been computed (Figure 4, after line 22). Having that in mind we can use a single offset array of 32-bit integers with every number holding three 10-bit integer values for each of the *Off* arrays.

This trick allows us not only to reduce memory requirements but also simplifies the code as only one prefix sum must be computed (Figure 5). The prefix sum call at line 15 computes the desired values for all 3 components at once without any interference between them.

### 4.3 Huge stage

There is one major difference between the otherwise similar big and huge stages. At the huge stage we have very few nodes to process, so in order to keep all multiprocessors busy we want to assign several blocks to a single active node. Let $P$ denote the number of blocks of huge stage kernels that can run in parallel at a time on a given GPU, taking into account their thread counts and resource usage (Table 1). Let $x := \frac{P}{N}$ where $N$ is the number of active nodes. To best utilise the capability of the GPU, we launch $C := \lceil \frac{x}{3} \rceil$ blocks per axis of a node for the *Compute-Cost* phase and $S := \lceil x \rceil$ blocks per node for the *Split* phase.

```
1 function bigSplit
      (node,dim,splitPlane,B_A,B_L,B_R,B_B)
2 triangleIdx:=node.trianglesBegin+threadIdx
3 p:=splitPlane
4 bitMask=2^10 − 1
5 if (dim = doNotSplit) [...]
6   return
7 while(triangleIdx<node.trianglesEnd)
8   low:=source[triangleIdx].dim.low
9   high:=source[triangleIdx].dim.high
10  if (low<p ∧ high<p) type:=L shift:=0
11  if (low<p ∧ high≥p) type:=D shift:=10
12  if (low≥p ∧ high≥p) type:=R shift:=20
13  offset[threadIdx]:=1 shl shift
14  synchronize
15  prefixSum(offset)
16  synchronize
17  addr:=(offset[threadIdx] shr shift) &
        bitMask
18  destination[B_type+addr−1]
        :=source[triangleIdx]
19  last:=offset[threadCount−1]
20  B_L+=last & bitMask
21  B_D+=(last shr 10) & bitMask
22  B_R+=(last shr 20) & bitMask
23  triangleIdx+=threadCount
```

Figure 5: Split phase with only one offset array

Since many blocks handle a single node, they need to communicate. Unfortunately, this process is expensive, so we want to minimise the amount of exchanged data.

#### 4.3.1 Huge Compute-Cost kernel

The main while loop of the Huge *Compute-Cost* kernel (Figure 3 lines 10-17) can work completely independently between blocks working on the same node. We just make sure that each triangle is accessed by exactly one thread from all blocks by correct indexing.

At the synchronisation bar (line 18) the contents of the L and H array are stored to global GPU memory and then all data is read by the last block to reach this point in the program. The prefix sums and following cost computation (lines 19-29) is performed by that single block.

#### 4.3.2 Huge Split kernel

In *Split* kernel we must ensure that two different blocks never copy triangle bounds to the same memory. To that end, the *base pointers* ($B_L$, $B_R$, $B_B$) (Figure 4) are stored in global memory of the GPU which can be accessed by all blocks. At each iteration of the main loop of the kernel, after the prefix sum is computed (right after line 22) the block atomically increment the global *base pointers*. As a result the correct amount of memory is reserved exclusively to this block.

### 4.4 Medium stage

It would be inefficient to use big stage blocks, consisting of 512 threads, for nodes having less than 512 triangles. That is why, at medium stage, we want to launch more, smaller blocks so that a single multiprocessor can handle more nodes in parallel (Table 1). This imposes harder memory constraints for each block.

As a result the *Compute-Cost* kernel cannot use extended bin arrays anymore (Section 4.2.1). Only 3 counters per bin are used, which only partially reduce atomic conflicts and may lead to higher execution serialisation. Our experiments have show this is a good tradeoff for being able to process more nodes in parallel.

Because the *Split* block consists now only of 128 threads (Table 1), that is the maximum sum value of the *Off* arrays. Therefore we are using 8-bit offset counters bundled into one 32-bit integer, similarly to what was used in the Big Split kernel (Figure 5) but we replace bitwise operations with type castings.

### 4.5 Small stage

After few iterations of medium stage, the active nodes become even smaller and we want to han-

dle even more of them at once in parallel, without having any threads being idle. That is why at the small stage we use a single *warp* per node. Due to hardware limitations we cannot launch more than 8 blocks on each Stream Multiprocessor, that is why the blocks still consists of 128 threads — 4 warps, but each of the warps work independently of all other warps.

In addition to higher parallelism, this allows us to get rid of all barriers in the kernel because all inter-thread communication is restricted to a single SIMD unit which is always implicitly synchronised.

### 4.6 Tiny stage

In the tiny stage we process nodes of size up to $\mathcal{W} = 32$ triangles. It would be a waste of resources to dedicate a whole *warp* to a single node, especially when these can contain only few primitives. That is why at tiny stage we try to pack as many nodes as possible to a single $\mathcal{M} = 512$-thread block regardless of the warp boundaries, while keeping one-to-one mapping between threads and triangles.

There are $\mathcal{M}$ triangles assigned to each block, one per thread, but we overlap the blocks with each other by $\mathcal{W}$ primitives (Figure 6). This configuration ensures that each node, consisting of at most $\mathcal{W}$ triangles, can be handled entirely by one of the blocks, with minimal impact on the overall performance.

Since there are only a few primitives in each node, using a binned algorithm would introduce a needlessly big overhead coming from the bins themselves. Secondly, as Hunt et al. in [HMS06] showed, cost function over the domain of those small nodes is much less smooth and the approximation less effective. That is why we use an *exact SAH* approach, similarly as it is done in [HMS06]. Each thread loads the triangle's lower and higher events and treats them as split candidates. Then, each thread counts the number of other events on each side of the split plane in linear time.

At the split phase, a single block handles several nodes, similarly as it was done in *Compute-Cost*. We use a single offset array for all handled triangles but the prefix sum algorithm is modified in such a way that values are not added up across node bounds (segmented sum).

## 5 Results

In order to evaluate the quality of the produced kd-tree versus the time needed for its construction, we measure the build time and the rendering time for four scenes using different kd-trees. We decided not to write our own GPU ray tracer, as its efficiency could influence the results. Instead we decided to integrate our builder into RTFact [GS08] and compare our resulting trees with others all in the same framework.

We are not interested in time to transfer the initial data from the RTFact structures to GPU and the construct tree back nor the overhead of initialising the CUDA context. We only measure the kd-tree construction time and the resulting CPU ray tracing performance of the scene using RTFact.

We have tested our algorithm against four scenes: Sponza, Conference, one frame of Fairy Forest, and Venice (Figure 7) running on an Intel Core2 Quad Q9400 2.66GHz computer with 4GB of RAM and an NVIDIA GeForce GTX 285 GPU.

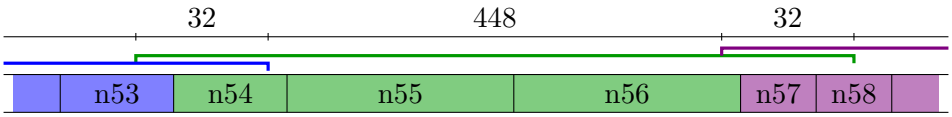


Figure 6: Work distribution at the tiny stage. Each block (e.g. green one) covers $\mathcal{M}$ triangles, but the first $\mathcal{W}$ triangles are also covered by the previous block (blue) and the last $\mathcal{W}$ triangles – by the next block (violet). Nodes 54, 55 and 56 are assigned to the green block. Node 57, although still in range of the green block, is already covered entirely by the violet one and it is assigned there.

Figure 7: Four scenes we tested our algorithm with: Sponza (67 000 triangles), Fairy Forest (174 000 triangles), Conference (283 000 triangles), and Venice (996 000 triangles)

### 5.1 Overall performance

The execution time and tree quality heavily depend on the chosen traverse-to-intersection cost ratio $\frac{C_t}{C_i}$. Throughout our program we set $C_i$ to 1 and modify only $C_t$. Depending on the number of rays in a packet, in our environment we found values of $C_t$ between 2 and 4 to be a good trade-off between tree quality and construction time (Table 2).

For three of the tested scenes and low $T_C$ values, our program performs 30-60 times faster than a full-quality CPU construction with only minor loss of rendering performance ($\leq$ 15%). In addition, construction time may be reduced a few times by using high traverse cost $T_C$, but as a result rendering takes about twice as much time.

So far the only successful kd-tree builder on GPU that we are aware of are the one of Zhou et al. [ZHWG08] and its memory-bound derivative of Hou et al. [HSZ*09]. As already described in Section 2.2 their algorithm uses spatial median splits or cuts off empty space on the top levels of the tree until they reach a threshold of 64 triangles in a node. This median split approach, especially for big scenes with unevenly distributed triangles may generally lead to a performance drop in ray tracing by a factor of 3 to 5 when compared to SAH[Hav00], but we have no explicit data to support that it is the case with this algorithm.

In [ZHWG08] their algorithm was tested on another NVIDIA card — a GeForce 8800 ULTRA. That card offers only around half the number of multiprocessors as GTX 285, but the multiprocessors are slightly faster. On the GeForce

| Stage | Full | Zhou | $C_t$ | | | |
|---|---|---|---|---|---|---|
| | | | 1.3 | 2.0 | 4.0 | 16.0 |
| Sponza | 1072 | | 37 | 29 | 21 | 11 |
| | 189 | | 204 | 212 | 244 | 384 |
| Fairy | 3596 | 77*/58 | 57 | 48 | 38 | 32 |
| | 244 | 245 | 238 | 243 | 277 | 434 |
| Conf. | 5816 | | 113 | 93 | 69 | 39 |
| | 196 | | 208 | 217 | 238 | 344 |
| Venice | 21150 | | N/A | N/A | 272 | 183 |
| | 250 | | | | 416 | 656 |

Table 2: Building performance (the top numbers of each row) and RTFact CPU rendering performance (the bottom numbers) depending on the used algorithm and the traverse cost $C_t$. All values are in milliseconds. For rendering — 3 arbitrarily chosen viewpoints per scene were tested, shooting only primary rays with no shading procedure, and the table reports the average results. The "Full" column represents the full quality SAH CPU builder used in RTFact. "Zhou" is a kd-tree from [ZHWG08] and [HSZ*09]. Their construction time was measured on a GeForce 8800 ULTRA(*) and GTX 280. For $C_T$ equal to 2.0 and 1.3, the Venice scene tree and duplicated data did not fit our GTX 285 so no construction times are available.

8800 ULTRA their program needs 77 miliseconds to build kd-tree for the Fairy Forest. Hou et al. [HSZ*09] test their GPU algorithm on a GTX 280, and produce the tree in 58 milliseconds.

As shown in Table 2 using our algorithm we can choose $C_t = 2.0$ and 25% faster construct a tree of slightly higher quality. By setting $C_t$ to 4 we can obtain the tree 60% faster but with reduced ray tracing performance by 13%.

### 5.2 Scalability

We tested our program with respect to parallel efficiency. The efficiency is defined as:

$$E(n) = \frac{t_1}{n \cdot t_n}$$

where $n$ is the number of processors running in parallel, and $t_x$ is the execution time of computation performed on $x$ processors. The CUDA environment and NVIDIA's GPUs provide two sources of parallelism: at the top level the GPU consists of several multiprocessors which work in MIMD fashion, and at the lower level each multiprocessor consists of several cores working in SIMD. We are interested in efficiency at the top level, that is — with respect to the number of multiprocessors.

Unfortunately we are unable to shut down just a part of the GPU. Instead we can carefully configure which blocks work and which stay idle, so that only the desired number of multiprocessors are working. We cannot control however the bandwidth of the memory controller which can bias down our results.

Kun Zhou et al. [ZHWG08] also analyse scalability of their implementation. As a comparison base they test the program on 16 cores and compare it with a 128-core run (which is 8 times

| $N$ | Sponza | Fairy | Conference |
|---|---|---|---|
| 1 | 289.0ms | 594.4ms | 1237.9ms |
| 2 | 0.96 (×1.9) | 0.95 (×1.9) | 0.97 (×1.9) |
| 4 | 0.89 (×3.6) | 0.89 (×3.6) | 0.92 (×3.7) |
| 8 | 0.77 (×6.2) | 0.80 (×6.4) | 0.86 (×6.9) |
| 16 | 0.61 (×9.7) | 0.65 (×10.4) | 0.73 (×11.7) |
| 30 | 0.46 (×13.8) | 0.52 (×15.6) | 0.60 (×17.9) |

Table 3: Parallel efficiency and speedup for a given a number of working multiprocessors $N$. For the value $N = 1$, an absolute run time is given and the efficiency is 1 by definition.

more, thus $N := 8$) of their GeForce 8800 ULTRA. With an exception of smaller scenes were their parallelism is less efficient, their speedup is between 4.9 – 6.1 (efficiency 61%-76%). Our algorithm for $N$=8 shows a significantly better speedup (6.2–6.9) and parallel efficiency (around 80%).

The parallel CPU SAH builder of Choi et al. [CKL*09] was tested only on few initial steps of the kd-tree construct. They report a speedups between 4.8 and 7.1 for 16 cores (hence efficiency 30%–44%) which is also much lower than our performance for $N = 16$.

Still, for higher $N$ values our efficiency is still far from unity. Main reason is that many control functions take too little data to be efficiently processed on multiple processors. Most control functions perform a prefix sum operation (e.g. computation of *base pointers* in Section 3.2). While there exist fast prefix sum algorithms (e.g. [BOA09]) they are optimised for several-million data sets, while in our case, input data consists of a few to a few thousand elements.

Tests have shown that for $N = 1$, on the Fairy forest, these control function take 18ms (3% of total work) while for $N = 30$ — 11ms (29% of total run time), becoming the bottleneck and the source of the loss of efficiency. As expected, the efficiency improves with the scene size, since there are more primitives to be processed in the main parallel code.

### 5.3 Memory usage

Since all memory allocation operations are prohibitively slow on GPU we preallocate memory before the algorithm starts. Although there exist memory-aware algorithms for kd-tree construction (e.g. [HSZ*09] [WK07]) we simply assume we never cross the predefined limits. If $N$ is the number of input triangles we assume the total tree size and number of triangle references never exceeds $4N$, and that there are never more than $2N$ active nodes at each step of the algorithm, to be processed in parallel.

Table 4 shows how much memory is needed for particular objects used by the program. Since triangle are only referenced, never duplicated, no additional memory needs to be allocated for their vertices. Bounding boxes keep a separate entry for each triangle instance. In addition, as shown on Figure 1, we are using three arrays which triples our memory demand.

Taking into account all additional structures needed by the algorithm, the total memory usage can be estimated to be around 0.85KB per triangle. This means that on a 1GB GPU, only scenes with up to 1.2 million triangles can be processed simultaneously.

## 6 Future work

Although the above basic algorithm is fast, it consumes a lot of memory. Several techniques

| Object | Element size | Element size | Venice |
|---|---|---|---|
| Triangle vertices | 48B | $N$ | 45MB |
| Triangle bounding boxes | 28B | $3 \times 4N$ | 320MB |
| Kd-tree data | 32B | $4N$ | 122MB |
| Huge stage bins | 260B | $8P$ | 61KB |
| Active nodes control arrays | 52B | $3 \times 2N$ | 297MB |
| Prefix sum arrays | 20B | $2N$ | 38MB |
| Total | | | 822MB |

Table 4: Memory usage of particular components of the algorithm. As an example memory usage for Venice test scene is given.

can be explored to significantly reduce the memory requirements. For example, after a few initial steps, a part of the scene could be scheduled for later processing, with the algorithm focusing only on another part. This would allow to significantly reduce the limits on the number of nodes and triangles being processed at a time with only a minimal impact on the overall performance.

Another option for handling big scenes would be to split the work between several GPUs. The split could be performed after a few iterations of the presented algorithm with very low limits, or — if that already uses too much memory — by performing a fast low-quality split on a CPU and uploading the data on separate GPUs.

So far we discussed a kd-tree builder as a separate entity which, in our case, is used with a CPU ray tracer. A much more efficient solution would be to closely integrate the builder and renderer, so that data would not have to be transferred or transformed from one representation to another. The close builder-renderer integration also opens possibility for lazy construction of the kd-tree which could speed up animations even further.

# 7 Conclusion

We have presented an efficient kd-tree construction algorithm. The builder uses SAH on all levels producing high-quality trees while outperforming all other known CPU and GPU implementations. The program is highly scalable and so we believe it will map well to future GPU architectures as well, taking advantage of most of their computing power.

# References


[AL09] Aila T., Laine S.: Understanding the efficiency of ray traversal on gpus. In *Proceedings of High-Performance Graphics 2009* (2009).

[Ben75] Bentley J. L.: Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517.

[BOA09] Billeter M., Olsson O., Assarsson U.: Efficient stream compaction on wide simd many-core architectures. In *Conference on High Performance Graphics* (2009), pp. 159–166.

[CKL*09] Choi B., Komuravelli R., Lu V., Sung H., Bocchino R. L., Adve S. V., Hart J. C.: *Parallel SAH*

*k-D Tree Construction for Fast Dynamic Scene Ray Tracing*. Tech. rep., University of Illinois, 2009.

[GS08] Georgiev I., Slusallek P.: RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In *IEEE/Eurographics Symposium on Interactive Ray Tracing* (Aug. 2008).

[Hav00] Havran V.: *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[HMS06] Hunt W., Mark W. R., Stoll G.: Fast kd-tree construction with an adaptive error-bounded heuristic. In *IEEE Symposium on Interactive Ray Tracing* (September 2006), pp. 81–88.

[HSZ*09] Hou Q., Sun X., Zhou K., Lauterbach C., Manocha D., Guo B.: *Memory-Scalable GPU Spatial Hierarchy Construction*. Tech. rep., Tsinghua University, 2009.

[Kap85] Kaplan W. N.: Space-tracing: A constant time ray-tracer. *Computer Graphics 19* (1985).

[MB90] MacDonald D. J., Booth K. S.: Heuristics for ray tracing using space subdivision. *The Visual Computer 6*, 3 (1990), 153–166.

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

[PGSS06] Popov S., Günther J., Seidel H.-P., Slusallek P.: Experiences with streaming construction of SAH KD-trees. In *IEEE Symposium on Interactive Ray Tracing* (September 2006), pp. 89–94.

[SH74] Sutherland I. E., Hodgman G. W.: Reentrant polygon clipping. *Communications of the ACM 17*, 1 (1974), 32–42.

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

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

[WK07] Wächter C., Keller A.: Terminating spatial hierarchies by a priori bounding memory. In *IEEE Symposium on Interactive Ray Tracing* (2007), pp. 41–46.

[ZHWG08] Zhou K., Hou Q., Wang R., Guo B.: Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 papers* (2008), ACM, pp. 1–11.