High Performance Graphics (2016)
Ulf Assarsson and Warren Hunt (Editors)

# Efficient Stackless Hierarchy Traversal on GPUs with Backtracking in Constant Time

Nikolaus Binder† and Alexander Keller‡

NVIDIA Corporation

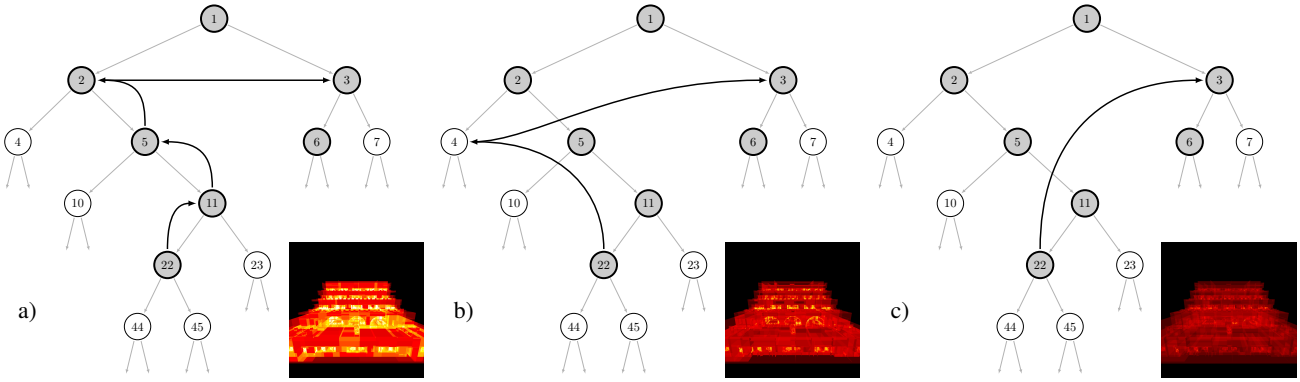



Figure 1: Backtracking from the current node with key 22 to the next node with key 3, which has been postponed most recently, by a) iterating along parent and sibling references, b) using references to uncle and grand uncle, c) using a stack, or as we propose, a perfect hash directly mapping the key for the next node to its address without using a stack. The heat maps visualize the reduction in the number of backtracking steps.

**Abstract**

*The fastest acceleration schemes for ray tracing rely on traversing a bounding volume hierarchy (BVH) for efficient culling and use backtracking, which in the worst case may expose cost proportional to the depth of the hierarchy in either time or state memory. We show that the next node in such a traversal actually can be determined in constant time and state memory. In fact, our newly proposed parallel software implementation requires only a few modifications of existing traversal methods and outperforms the fastest stack-based algorithms on GPUs. In addition, it reduces memory access during traversal, making it a very attractive building block for ray tracing hardware.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Ray Tracing

## 1. Introduction

Hierarchy traversal is one of the crucial building blocks for applications working on large data sets. As an example, accelerated ray tracing relies on traversing hierarchies in orders given by heuristics to efficiently find intersections of rays and scene geometry. In fact, bounding volume hierarchies are most prevalent in ray tracing due to a number of desirable properties such as flexibility, re-fitting, and simplicity of construction. The most efficient traversal order heuristics first visit the child node identified by the closer intersection of its bounding volume and the ray. For shadow rays it may pay off to first visit the bounding box with the larger surface area.

The conceptually most straightforward algorithm for binary hierarchy traversal is to traverse the tree top-down using a stack for backtracking: In each node the ray is intersected with the bound-

† e-mail:nbinder@nvidia.com
‡ e-mail:akeller@nvidia.com



DOI: 10.2312/hpg.20161191

ing volumes of both children. If both children are intersected, the traversal order heuristic decides which one of them is postponed and pushes it onto the stack. Whenever traversal ends in a leaf node or no intersection with children has been found, a node is popped from the stack.

While stack-based traversal on GPUs [AL09] is very efficient, the main drawback is the maintenance of the stack that comes with an overhead with regard to state memory and required bandwidth. As shown in Tab. 2, we improve upon the current state-of-the-art by introducing a new stackless traversal algorithm for GPUs that relies on

- important optimizations based on statistical evidence of what nodes are visited during backtracking (Sec. 2.1),
- a perfect hashing scheme to map keys of nodes of a binary hierarchy to their addresses in memory (Sec. 2.2.1), and
- its parallel construction (Sec. 2.2.2) at moderate cost in time and memory (see Tab. 1).

Furthermore, the early termination through the smart handling of disjoint traversal intervals (Sec. 2.3) and the simple pausing and resuming of hierarchy traversal (Sec. 2.4) are likely to deliver additional benefit in potential ray tracing hardware.

### 1.1. Previous Work

Smits [Smi98] proposed a stackless traversal of hierarchies with so-called *skip pointers* in every node. The tree is traversed by first intersecting the ray with the bounding volume of the current node. If an intersection is found, traversal continues to the first child; otherwise the subtree below the current node can be culled and traversal follows the skip pointer. As storing only one single skip pointer per node fixes the traversal order, arbitrary traversal orders require to store multiple skip pointers [BH06]. However, this comes with a major increase of required bandwidth due to larger node size and loading unused data. We evaluated a variant that processes rays sorted by the octant of their ray directions: For each octant, all skip pointers are replaced by the ones valid for the current octant. While this replacement has almost negligible run time and resolves the overfetching issue, rays must be partitioned or enqueued and performance may suffer from poor load balancing and decreased ray batch sizes. On current architectures, loading such nodes bears inefficiencies, as the nodes are small as compared to the cache lines size.

With the availability of GPUs, many approaches to stackless traversal of bounding volume hierarchies and $k$-d trees on the GPU [FS05, TS05, CHCH06, GPSS07, HSHH07, PGSS07] followed the original work of Smits. The most efficient stackless traversal methods perform iterative backtracking and therefore trade constant state memory for a non-constant effort to find the next postponed node: Laine [Lai10] marks tree levels in which a node was postponed in a *bit trail*. Backtracking then starts in the root node and follows the current traversal path until the last level with a bit set in the trail is found. Afterwards, traversal continues to the sibling of the previously taken child. The cost for the iterative backtracking, which is especially high in deep trees is ameliorated by also maintaining a short stack [HSHH07], effectively resulting in a hybrid of stackless and stack-based methods.

Hapala et al. [HDW*11] propose to perform backtracking with additional parent pointers. While the resulting bounding volume and triangle intersection tests are identical to a stack-based approach with the same traversal order and the number of iterations to get to close-by subtrees is dramatically reduced compared to restarting in the root, the method must re-evaluate the traversal order in every node during backtracking and therefore may suffer from the resulting restriction to cheap traversal order heuristics.

Áfra et al. [ÁSK14] and Barringer et al. [BAM14] introduce stackless traversal algorithms for arbitrary binary trees and dynamic traversal order without restarting from the root. Both methods store an additional pointer to the sibling in each node and in fact are very similar. However, Áfra et al. [ÁSK14] in addition test whether there are any remaining postponed nodes by checking if there are any bits set in the trail, which removes the need to return to the root node to terminate traversal and makes it the currently fastest stackless traversal method.

Yet, due to the overhead of iterative backtracking, all of the above methods are still not competitive with the fastest stack-based traversal kernels of Aila and Laine [AL09]. Combining the strengths of both classes of algorithms, we introduce a new tree traversal algorithm for ray tracing that determines the next node to be processed in constant time, i.e. without iteration, and in constant state memory.

Obviously, complete binary trees can be traversed with backtracking in constant time [MARG13, BAM14], because node addresses can be computed directly without chasing pointers. While there are applications where complete trees pay off, see e.g. Binder and Keller [BK15], the restriction to complete trees may result in a large performance penalty due to the lack of the ability to adapt to non-uniform geometry distributions.

Already Glassner [Gla84] had the idea of accessing nodes in a tree by computing a hash function of the path from the root towards a node in order to save memory for references and the stack. Today, comparing keys to resolve collisions and loops to identify existing nodes are too expensive to be competitive.

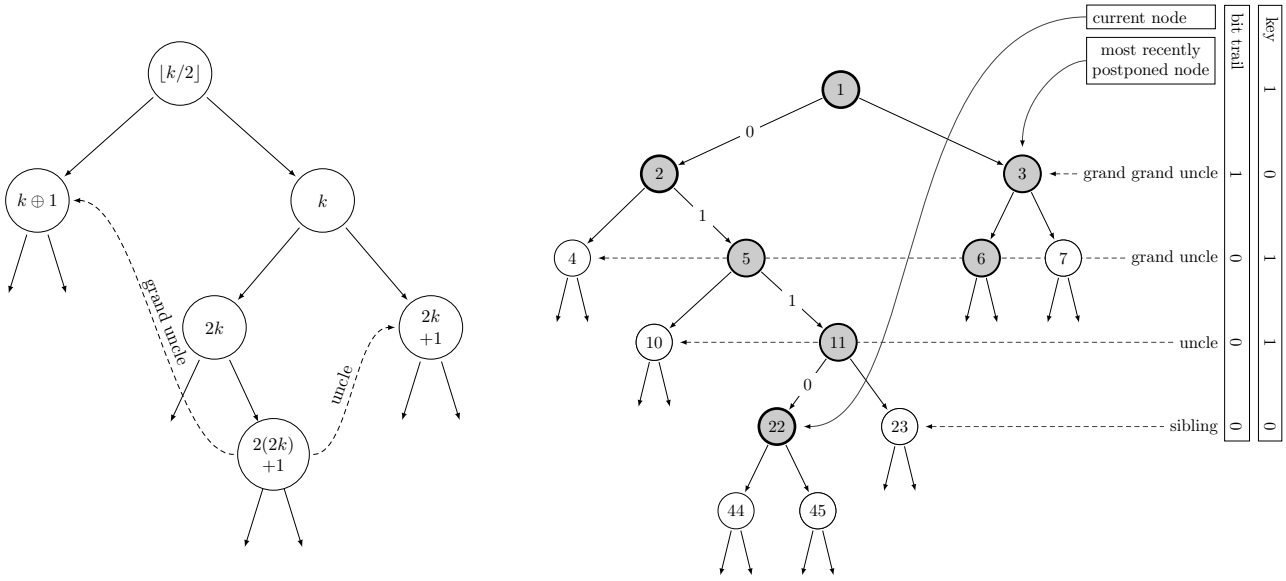
## 2. Efficient Stackless Traversal on GPUs

Alg. 1 shows the pseudocode of our traversal kernel that performs backtracking in constant time. In fact the majority of references to nodes to be processed next during hierarchy traversal can be retrieved using only one register instead of a stack (Sec. 2.1), while the remaining fraction of references can be retrieved in constant time using a perfect hash map (Sec. 2.2.1) that efficiently can be constructed in parallel (Sec. 2.2.2). In Sec. 2.3 and Sec. 2.4 techniques to further reduce the number of intersections are discussed.

### 2.1. Storing the Most Recently Postponed Node in a Register

First, the hierarchy (see the illustration in Fig. 2) is traversed down until either a leaf node is reached or both bounding boxes of the children are missed by the ray. Intersecting both children requires to postpone one child, whose reference is stored in a register as the *most recently postponed node*.

A second register is used as a bit trail [HL09, Lai10, ÁSK14] for




a) b)

Figure 2: a) Numbering convention for the keys of the nodes in the tree: The children of a node with key $k$ have the keys $2 \cdot k$ and $2 \cdot k + 1$, respectively. The parent of a node with key $k$ is $\lfloor k/2 \rfloor$, while the sibling has the key $k \oplus 1$, where $\oplus$ is the bit-wise `xor` operation. The dashed arrows from node $2 \cdot (2 \cdot k) + 1$ point to the grand uncle and the node's parent's sibling $\lfloor (2 \cdot (2 \cdot k) + 1)/2 \rfloor \oplus 1 = 2 \cdot k + 1$, which is the key of the uncle. b) An example snapshot of the traversal state: The gray nodes are the ones touched during traversal. On the way from the root with key 1 to the current node with key 22, the node with key 3 has been stored as the most recently postponed node, while the bounding boxes of the white nodes have not been intersected by the ray. As the nodes with keys 44 and 45 have not been intersected, traversal has to continue with backtracking - in this example - to the most recently postponed node with key 3. Note that the bit trail is relative to the current node and has a zero entry on each level where the siblings are not to be visited. The length of the bit trail is identical to the length of the path to the current node, which is indicated by the bits on the edges.

bookkeeping: Whenever descending a level in the hierarchy, the bit trail is shifted one bit to the left and the last bit is set, if and only if the ray intersected both bounding volumes of the children. Thus the number of trailing zeros indicates how many levels up in the hierarchy the next sibling has to be visited. For example, a set least significant bit indicates that the sibling of the current node needs to be visited, $10_2$ as the least significant two bits references the uncle, while $100_2$ refers to the grand uncle. We use the node data structure published in the source code of Aila and Laine [AL09], but in addition store siblings of ancestors like uncle, grand uncle, etc. in unused padding memory as illustrated in Fig. 4 and Fig. 5.

Loading the bounding boxes for intersection testing then also loads the references to the children, uncle, and grand uncle. So if no most recently postponed node has been stored, yet, the uncle reference is stored if its corresponding bit in the bit trail was set, or otherwise the reference to the grand uncle is stored if its corresponding bit in the bit trail was set.

When all threads of a warp either reached a leaf node or the descent could not be continued as none of the bounding boxes were intersected, traversal of nodes is paused. Then, all threads which have to intersect with data in leaf nodes perform the intersection test with the primitives it references. Using the same data structure as published in the source code of Aila and Laine [AL09], there are three unused 32-bit elements after the termination marker (see Fig. 5). We use the available space to store references to the uncle, grand uncle, and grand grand uncle of the current node. If the most recently postponed node is not set and the bit trail indicates that at least one of these three must be visited, it is set to the one on the lowest level which also has the corresponding bit set in the bit trail. Afterwards, all threads must perform backtracking to resume traversal.

### 2.2. Backtracking in Constant Time

Taking a look at the statistics in Fig. 3 reveals that in fact siblings, uncles, and grand uncles are the most likely backtracking targets. References to these nodes are always stored in the most recently postponed node register and consequently these targets are accessed in constant time. Unless overwritten, even targets higher up in the tree may have been stored in the most recently postponed node register. If not set, this register is updated whenever a node is loaded including siblings, uncles, grand uncles, and potentially grand grand uncles and thus extends the concept of a short stack [HSHH07, Lai10].

In order to enable backtracking in constant time for all remaining uncles, a numbering convention for the nodes in a binary tree is

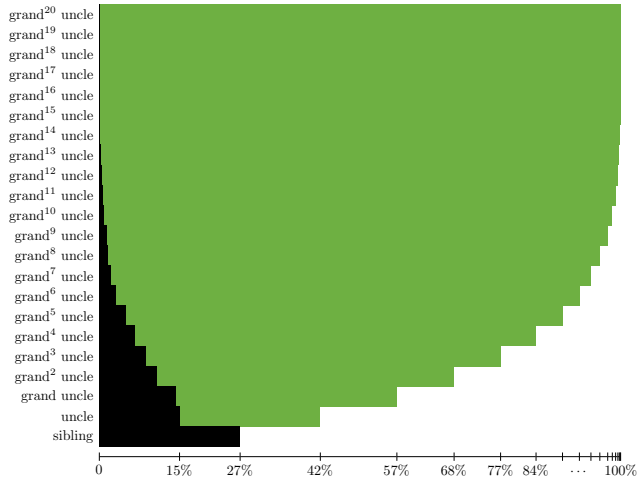



Figure 3: The histogram of the distances from the current node to the most recently postponed node during backtracking (black) and the cumulative number up to and including the ancestor (green), averaged over multiple test scenes and camera positions, reveals that backtracking to the sibling, uncle, and grand uncle are most common. Optimizing for short distances therefore pays off most.

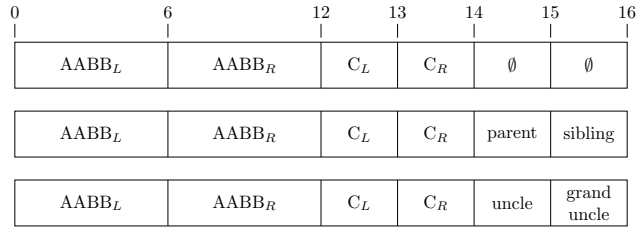



Figure 4: Node data structures for the traversal kernels: Stack-based traversal with padded elements due to memory alignment and cache efficiency (top), stackless traversal with parent and sibling references (middle), and uncle and grand uncle references (bottom) in the same memory footprint.

required: As illustrated in Fig. 2, the root node has the key 1, while each left child is double the key of its father, and its sibling, the right child, has the key twice of its father plus one. Note that the position of the leading one of the binary representation of the key of a node in fact is the length of the path from the root to this node, while the sequence of digits following the leading one identifies the branches taken along the path down the tree.

During traversal, we keep the key of the current node in an additional register. Updating the key is identical to updating the bit trail as described before: While descending the tree, the key is shifted one to the left. If traversal continues with the right child, the last bit in the key is set using binary xor/or with one.

The selected numbering convention allows for directly computing the key of the node that would be reached by the backtracking loop: Bits at identical positions in the key of the current node and the bit trail relate to the same level in the tree (see Fig. 2b). Therefore, going up in the hierarchy by shifting the bit trail to the right by the number of trailing zeros, the key of the resulting node is obtained by applying the identical shift operation to the key of the current node, i.e., just shortening its key. In the same way, toggling their least significant bits switches both the key and the bit trail to the other sibling. Note that the shift can be determined in constant time on the GPU using native instructions that reverse bits and count the leading zeros.

#### 2.2.1. Perfect Hashing

Using a perfect hash map, the updated key is mapped to the address of the next node in constant time. As a consequence, the code divergence due to a backtracking loop is completely removed. In fact, the hash tables may be considered a complementary acceleration data structure.

We chose a simple perfect hash method [TY79, FHCD92, LH06] to map a key $k$ to the index

$$h(k) := (k + d_{k \bmod D}) \bmod H$$

in the hash table of size $H$ that contains the address of the associated node. Selecting the size $D$ of the additional displacement table $d$ to be a power of two allows one to replace the inner modulo operation with a cheaper bit-wise and operation `& (D - 1)`. If furthermore $D < 2^{32}$, only the 32 least significant bits of the key need to be considered. As the table $d$ will require only a small fraction of the overall data, the above optimization is often desirable.

Although in principle hashing node keys is sufficient for traversal, storing uncle, grand uncle, and the most recently postponed node improves performance. As mentioned before, this does not increase the memory footprint, memory is loaded anyhow, and the cost is only one more register.

As a side note, storing uncle and grand uncle references in the nodes instead of parents and siblings [ÁSK14] allows for faster iterative backtracking: Then, backtracking may advance by two levels in one step (see Fig. 1b). As backtracking could also go over leaf nodes, it is then required to determine uncles and grand uncles of leaf nodes without having to search for the data beyond the termination marker. Fortunately, the used data structure has still empty space in the triangle identifier array, which can be used for this purpose. The evaluation of this approach reveals that on the average it is slightly faster than the method of Áfra et al. [ÁSK14]. Nevertheless, the bottleneck from iterative backtracking remains.

#### 2.2.2. Parallel Construction of the Hash Tables

The two required tables for perfect hashing can be constructed in time linear in the number of nodes to be hashed [FHCD92]. A *minimal* perfect hash would increase memory requirements by about 7% of the size of the BVH. However, taking into account the overall scene data including geometry, materials, and textures, it is affordable to use larger tables that in turn allow for a much faster construction. In addition, only keys of nodes whose references are not always stored in the most recently postponed node register need to be hashed: Keys are only hashed if their corresponding node has at least a grand nephew, which is an internal node, or at least a grand grand nephew that is a leaf node.

The size $D$ of the displacement table $d$ is set to the highest power

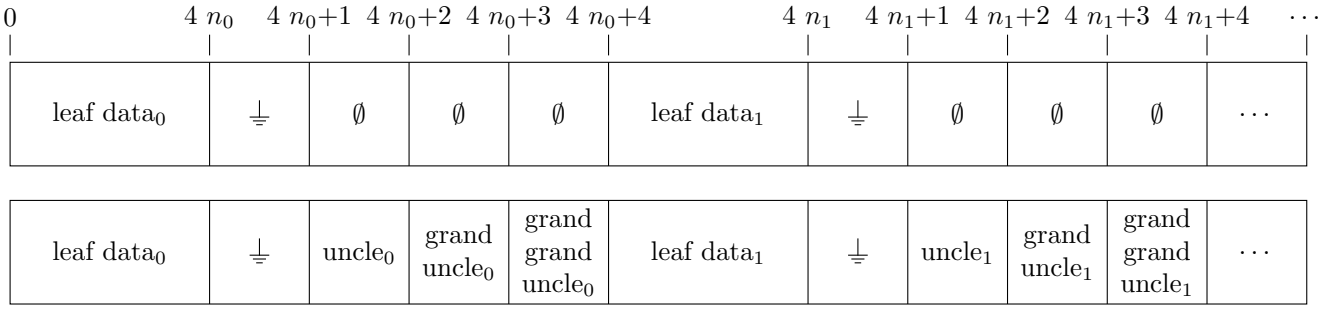



Figure 5: Original leaf data array for stack-based traversal (top) and layout with uncle and grand uncle references in place of padding memory (bottom) after the termination marker represented by the ground symbol. Note that marking the end of the leaf data arrays instead of storing their actual number $n_0, n_1, \ldots$ of primitives (like for example triangles) is more efficient on a GPU.

of two smaller than half the number of BVH nodes, and the size of the perfect hash table is selected as the smallest co-prime number greater than twice the number of BVH nodes. This means that obviously $H$ must be odd. If $D$ and $H$ were sharing a common factor $2^n \leq D$, according to the Chinese remainder theorem the number of collisions that can be resolved would be reduced by this factor, which is not desirable.

Observing the sequential construction algorithm for a perfect hash function [FHCD92], it turns out that the number of displacement values with the same number of dependencies is extremely large, which allows for a very simple and fast parallel construction method: We first map all keys $k$ to their associated displacement values with index $k$ mod $D$ and count the number of dependencies in parallel. Then, the displacement values are sorted in parallel by decreasing number of dependencies. Finally the displacement values are processed in parallel in batches with the same number of dependencies to resolve all conflicts in the hash table: Conflicts are resolved by first hashing all dependent keys of a displacement value, checking if the resulting cells in the hash map are empty. If this test fails for at least one of the keys, linear search is performed to find an appropriate displacement value that resolves the collisions for all keys simultaneously.

Given the BVH, this relatively simple parallel construction algorithm allows one to construct all required hash tables in less than 4 ms on an NVIDIA Geforce GTX Titan X (see Tab. 1).

As a simple optimization, we store the node address of displacement values with only one dependent key directly in the displacement value and mark these cases using the integer sign. As leaf nodes are also indicated by negative integers and can also be referenced, we shift the addresses to allow for a distinction. While this shift halves the number of primitives that can be referenced, it is still beyond available space for current hardware and the near future. This optimization not only improves construction performance, as it leaves more empty space in the hash table, but also removes the second dependent load operation for that case.

One indirection can also be removed by storing all data of internal nodes directly in the hash table. Then, all references to children and uncles in the node data must be updated to the new positions. However, there are two drawbacks: First, as we do not generate *minimal* perfect hash tables, the BVH size is increased. Second, the hash table must also reference leaf nodes, whose size is neither equivalent to internal nodes nor constant. In summary, the resulting special case together with efficient latency hiding of the load operation we just tried to omit eliminated the benefit in our evaluation.

### 2.3. Disjoint $t$-Intervals Mask

During the determination of the closest intersection of a ray and the scene boundary, traversal must continue after a first hit point has been found, because even though traversal is ordered along the ray, bounding volumes of adjacent subtrees may overlap, and therefore it cannot be guaranteed that all subtrees rooted by a postponed node are completely behind this point of intersection.

However, after an intersection has been found, postponed nodes for which $t_0$ is greater than the distance to the point of intersection can be pruned [Dam11]. Unfortunately, the overhead of loading and storing $t_0$ for every postponed node in practice often outweighs the benefit.

We therefore propose a cheaper, but at the same time less strict criterion: While disjoint bounding volumes of siblings guarantee that the entire subtree rooted by a postponed node is behind the one processed first, it is sufficient to check for each ray whether the $t$-intervals of the intersections with the bounding boxes of the two children are disjoint (see Fig. 6). This more general criterion is simple to check as all information is available during traversal, does not require any additional information in the hierarchy data, and not only covers the case of disjoint volumes (e.g. enforced by an SBVH builder), but also allows for pruning in cases similar to the upper example ray in Fig 6.

Using a second bit trail, we store one bit per level that indicates that the current $t$-intervals are *not* disjoint. After a point of intersection has been found inside the current bounding volume, this trail is used to mask out all subtrees that cannot be reached anymore: `bitTrail &= disjointIntervalsTrail`. Special care needs to be taken if the most recently postponed node reference is set: If the corresponding bit in the trail – the least significant bit – is masked out, the most recently postponed node reference must be unset. This test can be performed by checking `(bitTrail & -bitTrail) & disjointIntervalsTrail`.

Fig. 7 shows how the number of postponed nodes that need to

**Algorithm 1:** Pseudocode for efficient stackless traversal with backtracking in constant time.

```
nodeAddr ← rootNode;
nodeKey ← 1;
bitTrail ← 0;
mrpnAddr ← unknown; // address of most recently postponed node
while nodeAddr ≠ done do
    while nodeAddr is an internal node do
        (boxes, L, R, uncle, grandUncle) ← nodeData[nodeAddr];
        if mrpnAddr = unknown then
            if bitTrail & 0x6 then
                if bitTrail & 0x2 then mrpnAddr ← uncle;
                else mrpnAddr ← grandUncle;
        (hit_L, hit_R, closest) ← intersectBoxes(ray, boxes);
        if hit_L or hit_R then
            if ¬ hit_L or closest = R then nodeAddr ← R;
            else nodeAddr ← L;
            nodeKey ← nodeKey ≪ 1;
            bitTrail ← bitTrail ≪ 1;
            if hit_L and hit_R then
                bitTrail ← bitTrail ⊕ 1; // xor
                if closest = L then
                    mrpnAddr ← R
                else
                    mrpnAddr ← L
            if nodeAddr = R then
                nodeKey ← nodeKey ⊕ 1; // xor
        else
            break;

    if nodeAddr is a leaf then
        (primData, uncle, grandUncle) ← leafData[nodeAddr];
        intersect(ray, primData);
        if mrpnAddr = unknown then
            if bitTrail & 0x6 then
                if bitTrail & 0x2 then mrpnAddr ← uncle;
                else mrpnAddr ← grandUncle;

    if bitTrail = 0 then
        nodeAddr ← done;
    else
        // Backtracking
        numLevels ← ctz(bitTrail); // count trailing zeroes
        bitTrail ← (bitTrail ≫ numLevels) ⊕1; // xor
        nodeKey ← (nodeKey ≫ numLevels) ⊕1; // xor
        if mrpnAddr ≠ unknown then
            nodeAddr ← mrpnAddr;
            mrpnAddr ← unknown;
        else
            d ← displacement[nodeKey & (D - 1)];
            if d < 0 then
                nodeAddr ← d;
                if (unsigned int) nodeAddr < 0xc0000000u then
                    nodeAddr ← nodeAddr - 0x80000000u;
            else
                nodeAddr ← hashMap[(nodeKey + d) mod H];
```

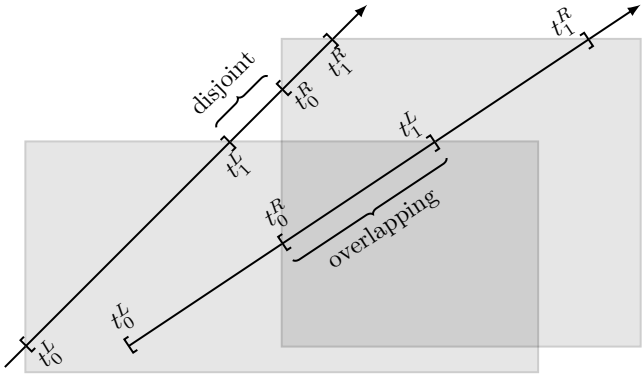



Figure 6: In ordered traversal, subtrees rooted by a postponed node can be pruned if an intersection has been found inside the current subtree and the $t$-intervals of the intersection of the ray with the postponed node and its sibling were disjoint (upper ray). While this is always the case for non-overlapping bounding volumes, in the general case there may be intersections of rays with the two bounding volumes that result in overlapping $t$-intervals (lower ray).

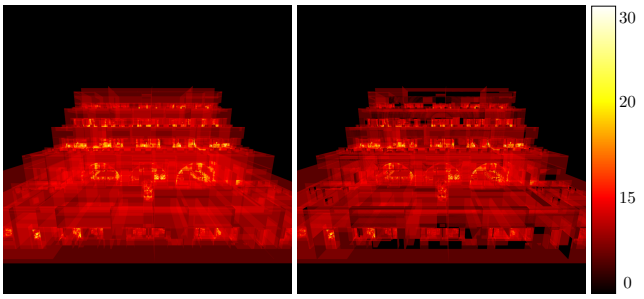



Figure 7: Accounting for disjoint $t$-intervals (right) reduces the number of postponed nodes that need to be checked again.

be visited is reduced when this mask is used in practice. However, we found that the bookkeeping overhead is slightly greater than the benefit for a an implementation in software on current generation hardware.

### 2.4. Pausing and Resuming

Stackless methods allow for efficiently pausing and resuming traversal, because only a small state of constant size must be saved. For example, traversal may be paused to perform intersection in separate kernels or to reorder rays for divergence reduction. Traversal may be resumed after an intersection has been found, which often is desirable for handling transparency, translucency, or for path tracing [MARG13, GMOR14].

With stackless traversal, as describe before, pausing and resuming can be realized by providing at least a value for `nodeKey` (see Alg. 1). Then, `nodeAddr` can be determined from the hash map and the bit trail is set to ones for all bits below the most significant one in the key. Unless the ray direction or origin have been changed, storing the bit trail in addition avoids revisiting uncles that already have been culled. The address `mrpnAddr` of the most


Figure 8: Heat map illustrating the reduction of the number of accessed nodes when continuing light transport paths from the node of intersection (right) instead of starting traversal from the root node (left). As close occlusions are found faster, the image on the right resembles an ambient occlusion rendering.

recently postponed node does not need to be stored, because one backtracking step needs to be performed before leaving the traversal loop as the current node address has already been processed. Then, `mrpnAddr` is either not set or `nodeAddr` is set to `mrpnAddr`.

Instances can be handled similarly: The original ray state is stored, while a new one is initialized for traversing the instance. On return, the previously stored state is used to resume traversal. Note that while it is simple to extend the number of bits in a bit trail to accommodate deep trees, instances also can be used to extend traversal depth at the cost of just a few registers.

Unless the ray direction changes, transparent surfaces can be handled as discussed. However, changing the origin or direction of the ray upon resuming in a point of intersection, as for example in path tracing, the bit trail becomes invalid and all uncles up to the root node need to be checked for intersection. While continuing paths right from the last intersection without restarting from the root notably reduces the number of box intersections as illustrated in Fig. 8, the performance of the software implementation did not improve. This may be attributed to worse cache hit rates, as memory access across SIMD units now is more divergent. A realization as latency hiding hardware, however, may turn out to be superior.

## 3. Discussion and Performance

Our traversal implementation is based on the code as published by Aila et al. [AL09] and uses their SBVH builder, which has been extended to write uncle and grand uncle, or parent and sibling references to node padding memory (see Fig. 4) and to the end of leaf data (see Fig. 5). Besides the original stack-based traversal kernel (the "`while-while`" code variant [AL09, Sec.3.2]), the previously fastest stackless traversal kernel by Áfra et al. [ÁSK14] using parent and sibling references has been added for comparison purposes. All measurements were performed on an NVIDIA Geforce GTX Titan X card and all kernels have been optimized for best performance by finding the optimal combination of cache qualifiers for the memory load operations.

For a fair comparison, we picked the best compiler version for each method: Stack-based traversal and stackless traversal with parent pointers achieved best speed with NVCC 6.5, while our new stackless approach showed almost the same performance for all compiler versions with a slight advantage for NVCC 7.5.

We use 32-bit data structures for scenes with a tree depth less or equal to 32 to maximize performance. Instead of using 64-bit types for deeper trees, it is beneficial to perform a custom overflow management: Each value that can overflow gets extra data in the shared memory of the GPU and an overflow is indicated by the most significant bit of the value. Only if this bit is set, shared memory is actually loaded and stored. Note that all operations except for the modulo operation of the current key do not require 64 bit operations. Still, a performance penalty can be observed when going beyond 32 bits.

As explained by Laine et al. [LKA13], a separation into specialized kernels often is favorable. We therefore evaluated the performance of isolated traversal kernels instead of measuring overall path tracing time using a megakernel. We averaged the throughput of several camera views in 100 passes after 50 warmup passes to avoid outliers. In each pass, 4 million rays were traced. Primary rays were enumerated in Morton order over the screen resulting in more coherent memory access. We also measured intersection performance for diffuse scattering, determining either closest or any intersections (shadow rays).

Dynamically fetching new rays and speculative traversal reduce divergence on SIMD architectures in stack-based traversal [AL09], especially in scenes with a highly varying number of internal nodes on the path to the first unculled leaf node within a warp, see e.g. the "Hairball" scene in Tab. 2. In stackless traversal, speculative traversal did not improve performance as already discovered by Áfra et al. [ÁSK14]. Furthermore, we observed that dynamically fetching rays is slower for primary rays in stackless traversal in all of the tested scenes. Depending on the implementation it either results in more instruction divergence or lower SIMD utilization during backtracking and therefore has been omitted in the implementation of the stackless traversal.

Note that all traversal methods included in the comparison enumerate the same nodes in the same order. The only exception is speculative traversal, which improved performance for the stack-based approach, but was slower for our stackless traversal method. If we disabled speculative traversal, the measurements would only point out the speed difference due to the different backtracking methods, but at the same time we would no longer compare against the state-of-the-art.

Besides operating in state memory of constant size, there are three fundamental differences between the presented final method using the perfect hash and the state-of-the-art method using a stack: First, the new method does not need to access global memory whenever nodes are postponed, but only requires operations on registers. Note that the same applies to the method of Áfra et al. [ÁSK14] and our briefly suggested variant which works with uncle references only. Second, the determination of the address of the next postponed node requires one or two potentially scattered memory fetches. However, memory access to the hash map is read only, wheres a stack-based approach also needs to store in global memory. Third, the extra memory for the hash tables is linear in the

number of nodes, whereas the extra stack memory for a stack-based approach is linear in the number of rays times stack size.

While in theory a stack-based approach for most cases generates more traffic to global memory, the cache hit rate for these requests is often very high and together with the latency hiding hardware results in an overall moderate penalty.

The number of read operations to find the next node in backtracking is the same if the node address is directly found in the displacement table. Otherwise another scattered load operation is required. On the other hand, postponing a node requires a write operation for stack-based approaches while stackless methods only operate on registers. As on GPUs a stack is not guaranteed to reside in the cache all the time, our measurements showed that the overall number of memory dependency stalls is smaller for our method.

We measured the performance for a number of freely available scenes with varying complexity. Tab. 1 summarizes their properties and shows the tiny overhead for setting up perfect hashing. Preliminary experiments using only a register for the most recently postponed node and the uncle references almost reached state-of-the-art performance. With the addition of perfect hashing, the measurements in Tab. 2 point out that the new stackless traversal method not only closes the gap between previous stackless algorithms and the state-of-the-art stack-based traversal kernels, but in fact outperforms all compared methods. The different performance speedups also show that the different behavior under varying conditions are sometimes advantageous for one method or the other: The overhead of bookkeeping and the divergent, scattered access to the hash tables are the main limiting factors for the new stackless method. On the other hand no global memory access is required when a node is postponed. Especially if traversal must only load very few to no postponed nodes, the new method is often superior to stack-based traversal. The speedup ranges between 2% and 35%. We observed that the largest speedups are likely achieved in scenes with a single object. However, even in closed scenes with many objects and a tree depth greater than 32 our method beats the-state-of-the-art by around 10 percent.

Pausing and resuming and the disjoint $t$-intervals mask reduce the number of nodes that need to be loaded as well as the number of postponed nodes at a very low overhead. However, the software implementation suffered from additional bookkeeping and register pressure on current generation GPUs, which only in rare cases improved performance. Obviously, this is an issue that would not be present in hardware. We speculate that different BVH construction methods with more aggressive split criteria might improve the situation and make the method worthwhile even in a software traversal kernel.

## 4. Conclusion and Future Work

We introduced a new stackless traversal method that outperforms the state-of-the-art stackless and stack-based traversal method due to its efficient backtracking in constant time. Besides the use of uncle references and storing the most recently postponed node in a register, perfect hashing is central to the approach as it completely eliminates one source of divergence on wide SIMD architectures. The only tiny extension of the ray state is especially interesting when huge numbers of rays are in flight during rendering [LKA13, ENSB13, McC14].

It is straightforward to apply the techniques to other hierarchy traversals such as many-light-hierarchies and occlusion culling. Finally, the read-only memory access during traversal and the simplicity of the algorithm lend themselves for an implementation in hardware which we plan to explore.

## Acknowledgements

This work has been dedicated to the memory of Brian Smits, ⋆1968-†2013, and his contributions to computer graphics.

## References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics* (2009), Eurographics and ACM, pp. 145–149. 2, 3, 7, 9

[ÁSK14] ÁFRA A., SZIRMAY-KALOS L.: Stackless multi-BVH traversal for CPU, MIC and GPU ray tracing. *Computer Graphics Forum 33*, 1 (2014), 129–140. 2, 4, 7, 9

[BAM14] BARRINGER R., AKENINE-MÖLLER T.: Dynamic ray stream traversal. *ACM Trans. Graph. 33*, 4 (July 2014), 151:1–151:9. 2

[BH06] BOULOS S., HAINES E.: Ray-box sorting. Ray Tracing News, Vol. 19, No. 1, 2006. 2

[BK15] BINDER N., KELLER A.: Stackless ray tracing of patches from feature-adaptive subdivision on GPUs. In *SIGGRAPH '15, Talks Proceedings* (2015), p. 22:1. 2

[CHCH06] CARR N., HOBEROCK J., CRANE K., HART J.: Fast GPU ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface* (2006), pp. 203–209. 2

[Dam11] DAMMERTZ H.: *Acceleration Methods for Ray Tracing based Global Illumination*. PhD thesis, Universität Ulm, 2011. 5

[ENSB13] EISENACHER C., NICHOLS G., SELLE A., BURLEY B.: Sorted deferred shading for production path tracing. In *Proceedings of the Eurographics Symposium on Rendering* (2013), pp. 125–132. 8

[FHCD92] FOX E., HEATH L., CHEN Q., DAOUD A.: Practical minimal perfect hash functions for large databases. *Commun. ACM 35*, 1 (Jan. 1992), 105–121. 4, 5

[FS05] FOLEY T., SUGERMAN J.: Kd-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (2005), pp. 15–22. 2

[Gla84] GLASSNER A.: Space subdivision for fast ray tracing. *IEEE Computer Graphics & Applications 4*, 10 (Oct. 1984), 15–22. 2

[GMOR14] GARCÍA A., MURGUIA S., OLIVARES U., RAMOS F.: Fast parallel construction of stack-less complete LBVH trees with efficient bit-trail traversal for ray tracing. In *Proceedings of the 13th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry* (2014), pp. 151–158. 6

[GPSS07] GUNTHER J., POPOV S., SEIDEL H.-P., SLUSALLEK P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing* (2007), pp. 113–118. 2

[HDW*11] HAPALA M., DAVIDOVIČ T., WALD I., HAVRAN V., SLUSALLEK P.: Efficient stack-less BVH traversal for ray tracing. In *Proceedings of the 27th Spring Conference on Computer Graphics* (2011), ACM, pp. 7–12. 2

[HL09] HUGHES D., LIM I.: Kd-jump: A path-preserving stackless traversal for faster isosurface raytracing on GPUs. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (Nov. 2009), 1555–1562. 2

Table 1: Test scene information, hash table size with relative increase in total memory (without textures and materials), and construction time of the hash tables.

| Scene | URL | k triangles | hash tables [MB] | | [ms] |
|---|---|---|---|---|---|
| Tears In The Rain | http://forums.cgsociety.org/forumdisplay.php?f=185 credit: Steven Stahlberg | 18 | 0.2 | +6% | 0.33 |
| Sibenik | http://graphics.cs.williams.edu/data/meshes.xml | 80 | 0.5 | +6% | 0.38 |
| Fairy-Forest | http://www.sci.utah.edu/ wald/animrep/ | 173 | 1.0 | +6% | 0.37 |
| Armadillo | http://graphics.stanford.edu/pub/3Dscanrep/ | 213 | 1.4 | +7% | 0.46 |
| CrytekSponza | http://graphics.cs.williams.edu/data/meshes.xml | 262 | 1.8 | +6% | 0.40 |
| Conference | http://graphics.cs.williams.edu/data/meshes.xml | 282 | 1.9 | +6% | 0.49 |
| Powerplant | http://graphics.cs.williams.edu/data/meshes.xml | 296 | 2.9 | +7% | 0.48 |
| Arabic-City | https://3dwarehouse.sketchup.com/model.html?id=809c028285022519b6b5161efde62bf9 | 407 | 3.1 | +7% | 0.54 |
| Classroom | https://www.blender.org/download/demo-files/ | 606 | 3.8 | +6% | 0.54 |
| Persian-City | https://3dwarehouse.sketchup.com/model.html?id=58f399c96e04552fde1c31612629d3dc | 761 | 5.9 | +7% | 0.77 |
| Dragon | http://graphics.cs.williams.edu/data/meshes.xml | 871 | 5.7 | +7% | 0.65 |
| Emily | http://gl.ict.usc.edu/Research/DigitalEmily2/ | 888 | 5.6 | +7% | 0.71 |
| Buddha | http://graphics.cs.williams.edu/data/meshes.xml | 1,088 | 6.7 | +7% | 0.66 |
| Levi | https://www.blender.org/download/demo-files/ | 1,710 | 11.8 | +7% | 1.09 |
| Bubs | http://www.cs.utah.edu/ rvance/cs6620/final/ | 1,888 | 11.9 | +7% | 1.24 |
| Soda Hall | http://www.cs.princeton.edu/ funk/walk.html | 2,169 | 13.3 | +7% | 1.25 |
| Hairball | http://graphics.cs.williams.edu/data/meshes.xml | 2,880 | 30.6 | +6% | 1.94 |
| Pipers Alley | http://forums.cgsociety.org/forumdisplay.php?f=185 credit: Clint Rodrigues | 4,053 | 24.0 | +7% | 2.08 |
| Enchanted Forest | http://forums.cgsociety.org/forumdisplay.php?f=185 | 7,521 | 48.4 | +7% | 3.58 |
| San-Miguel | http://graphics.cs.williams.edu/data/meshes.xml | 10,501 | 58.5 | +6% | 3.92 |

Table 2: Performance for primary rays (P), diffuse shadow rays (S) and diffuse closest hit rays (D) in M rays/s and relative to [AL09].

| Algorithm | [AL09] | | | [ÁSK14] | | | | | | our new algorithm | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | P | S | D | P | | S | | D | | P | | S | | D | |
| Tears In The Rain | 947 | 338 | 310 | 823 | 87% | 302 | 89% | 267 | 86% | 1042 | 110% | 431 | 128% | 393 | 127% |
| Sibenik | 790 | 292 | 231 | 628 | 79% | 247 | 85% | 169 | 73% | 803 | 102% | 316 | 108% | 237 | 103% |
| Fairy-Forest | 426 | 232 | 188 | 351 | 82% | 189 | 81% | 147 | 78% | 452 | 106% | 254 | 109% | 207 | 110% |
| Armadillo | 837 | 236 | 214 | 731 | 87% | 212 | 90% | 191 | 89% | 981 | 117% | 312 | 132% | 288 | 135% |
| CrytekSponza | 514 | 246 | 165 | 413 | 80% | 207 | 84% | 125 | 76% | 541 | 105% | 275 | 112% | 178 | 108% |
| Conference | 786 | 399 | 253 | 662 | 84% | 392 | 98% | 221 | 87% | 820 | 104% | 498 | 125% | 304 | 120% |
| Powerplant | 692 | 250 | 189 | 578 | 84% | 233 | 93% | 158 | 84% | 743 | 107% | 297 | 119% | 225 | 119% |
| Arabic-City | 673 | 332 | 261 | 543 | 81% | 279 | 84% | 206 | 79% | 698 | 104% | 384 | 116% | 300 | 115% |
| Classroom | 457 | 234 | 175 | 399 | 87% | 190 | 81% | 135 | 77% | 513 | 112% | 263 | 112% | 195 | 111% |
| Persian-City | 570 | 304 | 201 | 461 | 81% | 255 | 84% | 144 | 72% | 587 | 103% | 337 | 111% | 203 | 101% |
| Dragon | 743 | 212 | 194 | 624 | 84% | 184 | 87% | 164 | 85% | 866 | 117% | 280 | 132% | 254 | 131% |
| Emily | 676 | 254 | 234 | 542 | 80% | 224 | 88% | 201 | 86% | 737 | 109% | 320 | 126% | 293 | 125% |
| Buddha | 1237 | 210 | 185 | 1089 | 88% | 187 | 89% | 162 | 88% | 1419 | 115% | 281 | 134% | 245 | 132% |
| Veyron | 752 | 180 | 144 | 610 | 81% | 157 | 87% | 114 | 79% | 829 | 110% | 224 | 124% | 167 | 116% |
| Levi | 667 | 232 | 213 | 559 | 84% | 197 | 85% | 181 | 85% | 770 | 115% | 305 | 131% | 277 | 130% |
| Bubs | 709 | 335 | 237 | 586 | 83% | 262 | 78% | 179 | 76% | 797 | 112% | 358 | 107% | 261 | 110% |
| Soda Hall | 649 | 362 | 262 | 513 | 79% | 323 | 89% | 192 | 73% | 666 | 103% | 414 | 114% | 266 | 102% |
| Hairball | 190 | 77 | 65 | 147 | 77% | 72 | 94% | 57 | 88% | 192 | 101% | 96 | 125% | 79 | 122% |
| Pipers Alley | 558 | 199 | 164 | 464 | 83% | 175 | 88% | 134 | 82% | 580 | 104% | 232 | 117% | 186 | 113% |
| Enchanted Forest | 237 | 81 | 64 | 205 | 86% | 77 | 95% | 56 | 88% | 250 | 105% | 99 | 122% | 76 | 119% |
| San-Miguel | 246 | 149 | 81 | 198 | 80% | 138 | 93% | 65 | 80% | 257 | 104% | 184 | 123% | 89 | 110% |
| Average | | | | | 83% | | 88% | | 81% | | 108% | | 120% | | 117% |

[HSHH07] HORN D., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree GPU raytracing. In *Proceedings of the Symposium on Interactive 3D Graphics and Games* (2007), ACM, pp. 167–174. 2, 3

[Lai10] LAINE S.: Restart trail for stackless BVH traversal. In *Proceedings of High Performance Graphics* (2010), Eurographics and ACM, pp. 107–111. 2, 3

[LH06] LEFEBVRE S., HOPPE H.: Perfect spatial hashing. In *ACM SIGGRAPH Papers* (2006), pp. 579–588. 4

[LKA13] LAINE S., KARRAS T., AILA T.: Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proceedings of High-Performance Graphics* (2013). 7, 8

[MARG13] MURGUÍA S., ÁVILA F., REYES L., GARCÍA A.: Bit-trail traversal for stackless LBVH on DirectCompute. In *GPU Pro 4*. AK Peters/CRC Press, 2013, pp. 319–336. 2, 6

[McC14] MCCOMBE J.: Introduction to PowerVR ray tracing. GDC Vault Presentation Recording, 2014. 8

[PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless kd-tree traversal for high performance GPU ray tracing. *Com-*

*puter Graphics Forum 26*, 3 (Sept. 2007), 415–424. (Proceedings of Eurographics). 2

[Smi98] SMITS B.: Efficiency issues for ray tracing. *Journal of Graphics Tools 3*, 2 (1998), 1–14. 2

[TS05] THRANE N., SIMONSEN L.: *A Comparison of Acceleration Structures for GPU Assisted Ray Tracing*. Master's thesis, Aarhus Universitet, Denmark, Aug. 2005. 2

[TY79] TARJAN R., YAO A.: Storing a sparse table. *Commun. ACM 22*, 11 (Nov. 1979), 606–611. 4