Volume xx (20yy), Number z, pp. 1–12

Editing Compact Voxel Representations on the GPU

M. Molenaar[†1] and E. Eisemann[1]

[1]Delft University of Technology, The Netherlands

Figure 1: *The Citadel scene at a voxel resolution of* $128K^3$ *in which we place several large spheres in real-time using our editing framework. The radius of the spheres varies between 1140 and 1820 voxels. Each voxel stores a 4-bit material ID which is textured accordingly.*

Abstract

A Sparse Voxel Directed Acyclic Graph (SVDAG) is an efficient representation to display and store a highly-detailed voxel representation in a very compact data structure. Yet, editing such a high-resolution scene in real-time is challenging. Existing solutions are hybrid, involving the CPU, and are restricted to small local modifications. In this work, we address this bottleneck and propose a solution to perform edits fully on the graphics card, enabled by dynamic GPU hash tables. Our framework makes large editing operations possible, such as 3D painting, at real-time frame rates.

CCS Concepts

• ***Computing methodologies*** → ***Volumetric models;*** *Ray tracing; Graphics processors;* ***Concurrent algorithms;***

1. Introduction

Three-dimensional data can be represented in various forms, but in this work we focus exclusively on voxels. Voxels are (cubic) cells of a regular 3-dimensional grid; the 3D equivalent of pixels. Since voxels represent volume, and not surfaces, they are a good fit for representing volumetric data.

Voxels have many uses besides object representations. They find applications in physics simulations [LHN*05, Hoe16], rendering [CNS*11], 3D painting [DGPR02, LHN*05, KKK18], and 3D printing [Kuž21]. In these latter cases, voxel data is not static and requires updates. Supporting these operations for very high-resolution voxel scenes requires a compact, yet easy-to-modify and fast-to-render structure.

A voxel representation that has received a lot of attention in recent years is Sparse Voxel Directed Acyclic Graphs (SVDAGs). This representation involves a sparse compressed voxel encoding, where repeating information is only stored once. Thus, the typically high memory consumption of voxel representations could be significantly reduced. One major downside of this representation is

† Chairman Eurographics Publications Board

that being compressed means that editing is not as straightforward, precluding the use of this representation in many voxel contexts.

In this paper, we present a method for storing, editing, and rendering high resolution voxel models using SVDAGs. Our work is closely related to (and builds on top of) the HashDAG [CBE20]. Yet, while HashDAG implements editing on the CPU and achieves only interactive, not real-time frame rates, we present a fully GPU-based method to make full use of the highly parallel hardware. Our contributions include a novel GPU hash-table structure to encode the DAG, an efficient implementation for graphics hardware, and an extensive evaluation of various implementation variants.

2. Related Work

A regular grid is the simplest method of storing voxel data and indexing is a constant time operation. However, memory usage quickly becomes a problem due to the cubic memory consumption. Further, in most use cases, voxel models are sparse, containing large regions of either empty or homogeneous space. Examples include voxelization of 3D models, as used in video games, or 3D painting in VR. Here, memory usage can be significantly reduced by introducing one, or multiple, levels of indirection.

2.1. Spatial Hashing

Spatial hashing [GG98, ASA*09] attempts to reduce memory usage by inserting only non-empty voxels into a hash table using 3D positions as keys. This changes the memory usage to scale linearly with the number of non-empty voxels. Accessing voxels becomes slightly more computationally-expensive but remains at constant expected time. However, in GPU processing, worst-case computation time is typically more relevant than expected time as all threads (in a wave) have to wait for the slowest thread. A slow look-up is caused by hash collisions. When a bucket in the table is already occupied, a new key needs to be inserted into another location. When performing a look-up with a key, the searching thread may, thus, need to visit multiple locations in the hash table. Perfect Spatial Hashing [LH06] aims to alleviate this issue. A perfect hash function guarantees that two voxels never map to the same hash-table bucket. Computing such a hash function is very costly, making this extension ill-suited for real-time editing.

2.2. Sparse Voxel Octrees

Another popular representation for voxel data is the Sparse Voxel Octree (SVO). The voxel space is recursively divided into 2x2x2 non-overlapping regions of equal size. Empty regions are omitted from the data structure and not subdivided any further. This ensures that *sparse* regions of the scene occupy little space. SVOs not only improve memory usage but can also act as an acceleration structure for ray tracing.

Since empty regions are likely to occur, it is expected that most nodes will have empty children. It has thus become standard practice to store these nodes using a variable-rate encoding scheme [LK10, KSA13]. In the most simple case, each node stores a bitmask indicating which child regions are occupied, followed by the pointers to these children. More advanced pointer compression schemes exist [DKB*16, VMG16] but are not suitable for real-time editing.

Sparse Voxel Octrees can be constructed in various ways. The parallel approach of [Kar12] utilizes the fact that the octree structure follows a Morton space filling curve. Tree construction is a trivial $O(N)$ process given a set of sorted voxels. Sorting can be efficiently parallelized using radix sort. Similar techniques are used in Bounding Volume Hierarchy construction [PL10, Kar12]. Modifying an existing octree, while conceptually simple, becomes challenging in practice due to memory management and thread safety.

2.3. Sparse Voxel Directed Acyclic Graphs

Aligned repeating voxel patterns cause duplicate subtrees in an SVO. By merging the subtrees, we turn an SVO into a Sparse Voxel Directed Acyclic Graph [KSA13] (SVDAG). Depending on the geometric complexity of the scene, this can lead to large memory savings. Additional re-use can be achieved by considering symmetry relations [VMG16] (i.e., mirrored geometry) or by allowing approximate matches with a controllable level of error [vdLSE20].

Editing of SVDAGs is difficult compared to modifying octrees. Adding a new subtree requires eliminating redundancy in its interior and the existing DAG structure. Similar to the construction algorithm in [KSA13], this operation can be implemented efficiently using a bottom-up approach, which only requires node-node comparisons, rather than tree-tree comparisons. A more detailed explanation is provided in Section 3.

Careil et al. [CBE20] use this algorithm to modify high-resolution SVDAGs at interactive frame rates. Nodes and leaves of the SVDAG are stored in a hash table to accelerate the search for duplicates. This hash table, which doubles as a memory allocator, is replicated across both the CPU and GPU. Editing is performed on the CPU using multi-threading, after which changes to the hash table are uploaded to the GPU for rendering. These memory transfers do not scale with core count and limit performance.

2.4. SVDAG Materials

One of the most challenging parts in encoding and editing SVDAGs is maintaining a compact representation of voxel attributes, such as material descriptions. These values typically require much more data per voxel than binary occupancy. Various compression algorithms have been proposed [DKB*16, DSKA17, ME23] to reduce their memory usage. These algorithms assume that attributes are stored in a single continuous 1D array by collecting all voxels along a space-filling curve (e.g., Morton-order [Mor66]).

Updating any of these representations requires inserting values into a sorted array, which is an expensive operation. The HashDAG [CBE20] reduces this cost by splitting the single colour array into smaller sub-arrays. During editing, new (or modified) colours are not compressed, but rather directly embedded into the initially compressed representation. For very large scenes, we found that this approach remains a big performance bottleneck due to many tiny allocations (small sub-arrays) or the large amount of memory bandwidth required to maintain contiguous sorted arrays (large sub-arrays).

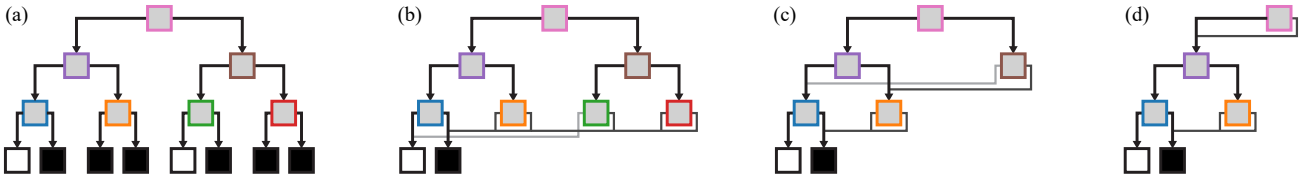

Figure 2: *1D example of converting a binary SVO (a) to an SVDAG (d). Duplicate leaves are eliminated and replaced by a single instance (b), followed by inner nodes at the level above (c). This process is repeated until the root node is reached (d).*

Our approach assumes that each scene contains a limited set of unique attributes, such as a material type. These attributes are stored in the leaves of the SVDAG, rather than a separate array, and are compressed alongside the geometry.

2.5. Hash Tables

To perform SVDAG editing on the graphics card, we require an efficient GPU hash table. Early works on GPU hash tables often focused on static construction [ASA*09, AVS*12, GLHL11]. While useful for many computer-graphics applications [LH06], they are unsuitable for our use case.

Various dynamic GPU hash tables have been developed recently, which differ in their collision handling (two keys mapping to the same bucket). In *open addressing* schemes, collisions are handled by moving a key/value (KV) pair to a different bucket. Searching then requires iterating over multiple buckets until the item is found or we can guarantee that the key is not in the hash table.

Stadium Hashing [KBGB15] and WarpDrive [JHS18] use probing schemes. Starting from the initial bucket, pointed to by the hash function, they iterate over buckets in a deterministic pattern until an empty slot is found.

Cuckoo hashing [PR04], as applied in [ZWY*15, LZL*21], assigns each key/value pair to two buckets, using two different hash functions. When encountering a collision, the new item is swapped with any potential item currently residing in that bucket. If an item was swapped out, it is subsequently inserted into the hash table using the second hash function, following the same process. Since items may only reside in either of the two buckets, look-ups have a constant worst-case time complexity. Insertions can be computationally expensive due to the potential for unbounded recursion.

In *closed addressing* hashing schemes, such as SlabHash [AFCO18], collisions are handled by creating a linked list of key/value pairs in each bucket. Compared to open-addressing this can make a table grow indefinitely, although the look-up performance tends towards a linear search as the number of items grows to infinity. The same concept also applies to the HashDAG [CBE20], where the virtual memory pages create a list.

Developing hash tables for the GPU introduces additional challenges, as a lot of performance can be gained by designing memory layouts that better match the GPU hardware capabilities. A common approach in most existing work [ZWY*15, JHS18, AFCO18, LZL*21] is to assign insertion or search operations to warps (currently a group of 32 threads) instead of individual threads. Each bucket of the table stores multiple slots, typically a multiple of 32. The threads inside a warp cooperate to check consecutive slots to either find a search key or an empty slot in the case of insertion. This results in fast coalesced memory accesses, which maximises memory bandwidth and, thus, performance.

3. Background

3.1. Duplicate Elimination

Turning a Sparse Voxel Octree (SVO) into a Sparse Voxel Directed Acylcic Graph (SVDAG) requires us to find and eliminate redundant leaf- and interior nodes. A naive solution would consider comparing all subtrees of equal depth, but this is computationally expensive. However, the bottom-up approach, popularized by [KSA13], only requires constant-size comparisons (Figure 2).

Assume an SVO with binary leaves at level $l = 0$. We can trivially find and eliminate all duplicate leaves (Figure 2a), producing the SVDAG leaves. Given these SVDAG leaves, we update the child pointers of inner nodes situated at the parent level $l = 1$ (Figure 2b). Now, when processing level $l = 1$, we know that the subtrees represented by two children are equivalent if and only if they have the same pointer. Thus, comparing inner nodes only requires us to look at the child pointers, rather than the attached subtrees. With this knowledge, we eliminate the duplicates and update the child pointers at the parent level $l + 1$. We can now repeat this process for $l + 1$ until we reach the root node.

3.2. HashDAG

Careil et al. [CBE20] propose a HashDAG for displaying and editing SVDAGs. At the core of this framework lies a hash table, which stores all SVDAG nodes and leaves. This hash table (created at start-up) is sized to be large enough to contain any nodes/leaves that are created during editing. Allocating all of this memory is likely to exceed the GPU capacity, which is addressed using virtual memory.

The hash table is replicated in CPU and GPU memory (for editing and rendering, respectively). Editing is performed by traversing the SVDAG, and deciding at each node whether to modify it, or keep it as-is. A recursive algorithm descends into the to-be modified children until the leaf level is reached. At this point, the updated leaf nodes are constructed and added to the SVDAG hash table. This returns a pointer to the new leaf, which is used to construct an updated node at the parent level. It is effectively a recursive implementation of the the bottom-up algorithm (Section 3.1). Multi-threading is achieved by spawning tasks for the first levels of recursion. To prevent race conditions, each bucket of the hash table is protected by a mutex.

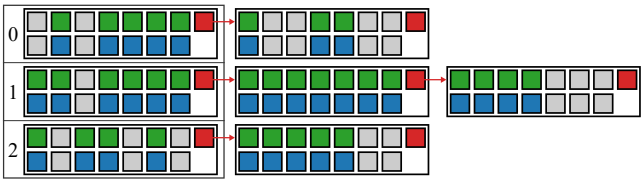

Figure 3: *Illustration of SlabHash [AFCO18] with four buckets. For illustrative purposes each slab contains 10 slots, consisting of a key (green) and a value (blue). Gray indicates an empty slot. The last slot (red) is used to store a pointer to the next slab.*

3.3. Slab Hash

Building on the same concepts as the HashDAG [CBE20], our work requires a GPU hash table which remains efficient under a highly parallel workload. The SlabHash [AFCO18] will be the basis of our hash-table implementation.

SlabHash consists of an array of N buckets. Keys are transformed into an arbitrary integer H by a hash function and inserted into bucket $B = H \bmod N$. By deterministically assigning items to a bucket, one reduces the search space from the entire data set to just a single bucket. Each bucket consists of a linked list of slabs, allowing it to grow arbitrarily (Figure 3). Each slab has 31 slots storing key/value pairs, and a final slot, which is used to store a pointer to the next slab. This memory layout ensures that the slabs are aligned to a GPU cache line (128 bytes).

Searching the hash table is performed by one warp (32 threads) per item. After computing the hash, the warp collaboratively iterates over the linked list of slabs in the selected bucket. For each slab, thread i loads keys$[i]$ from memory, and compares it to the item being searched for. If a match is detected, all threads in the warp collectively return the value at that slot. Empty slots are indicated by key $= 0$, making the insertion process similar to searching for 0 and atomically swapping it with the desired key.

4. Our Method

In this chapter, we provide an overview of our method. We start with a general overview of the Sparse Voxel Directed Acyclic Graph (SVDAG), followed by our unique GPU hash table design. Finally, we describe how editing is implemented in practice using various different compute kernels.

4.1. SVDAG Representation

Our Sparse Voxel Directed Acyclic Graph is inspired by the HashDAG [CBE20]. Rendering is performed using ray tracing for both primary visibility and shadows.

Leaves in our SVDAG encode regions of 4^3 voxels, using a 64-bit mask followed by 4-bit material indices for each of the occupied voxels. The leaves are padded to align to 32 bits, which is the fundamental *WORD* size of our SVDAG. Inner nodes of the DAG are stored in a similar fashion, starting with a bitmask indicating which of the 2^3 child regions are occupied, followed by 32-bit child pointers - one for each child that is present. We additionally reserve some bits to store whether a region is homogeneous (fully filled with a single material), and if so, with what material. This information is used to accelerate some of the editing tools, which benefit from knowing if a region is fully filled.

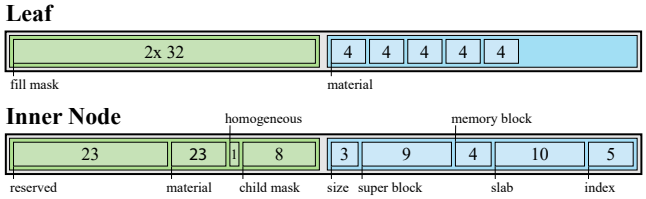

Figure 4: *Memory layout of SVDAG nodes and leaves. A leaf starts with a 64 bit mask (single green block for brevity) followed by up to 64 material IDs (blue). A node starts with a 32-bit header (green) followed by up to 8 child pointers (blue).*

Material Representation

There are various real-world scenarios in which a user might want to store not only occupancy but also a material (or some other numeric value) in each voxel. Previous work stores attributes outside the SVDAG in a separate array. While this may improve compression ratios, it significantly complicates and slows down editing. We instead store numeric values inside SVDAG leaves. This automatically compresses the attributes along with the geometry. The method's downside is that the SVDAG compression is negatively affected, since subtrees are merged only if shape and material match.

In this work, we store a material ID for each occupied voxel using 4 bits, which allows for 16 unique materials. The number of bits can easily be adjusted if more precision is required. Furthermore, one may use these values to encode an index into a position-dependent palette to allow for more variety.

4.2. SVDAG Encoding

We rely on a hash table to encode the SVDAG nodes and leaves, which allows us to search for duplicates quickly. However, it introduces some additional requirements to the GPU hash-table design:

Pointers must be stable. Inserting new items should not change the pointers of items that already reside in the hash table. If a pointer were to change, all SVDAG nodes pointing to that item would have to be updated. This would modify their respective hash keys, moving them to a different location in the hash table and, thus, invalidating their pointers as well. Due to this recursion, moving a single leaf would require a complete scan over the entire SVDAG.

Large items Since our nodes and leaves are stored in a compressed format, their size may vary. For simplicity, we create separate hash tables for each possible size. This size typically exceeds the largest basic data type (8 bytes), eliminating the possibility of atomically swapping items as in some related works [GLHL11, AVS*12, ZWY*15, JHS18, AFCO18, LZL*21].

Optimized for SVDAG traversal SVDAG traversal is performed many times during both editing and rendering. We optimize this operation - even if it comes at a (slight) cost; a lower insertion

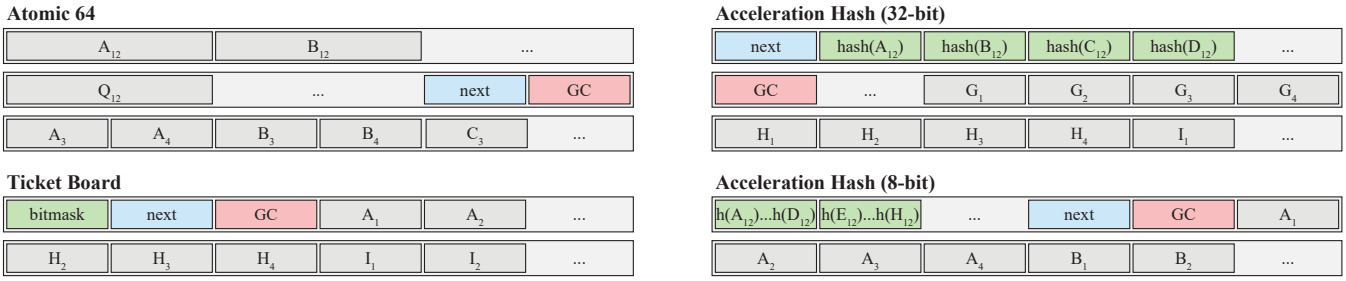

Figure 5: *The proposed slab memory layouts displayed as 128-byte cache lines, storing items consisting of four WORDs (16 bytes) each. X_i indicates the i'th WORD of item X. Repeating patterns are indicated by "...". Each method contains a pointer to the next slab (blue), and a garbage collection bitmask (red). The bottom methods are tightly packed and may not always start at the beginning of a cache line.*

or search performance. In practical terms, this means that we try to avoid splitting items into different memory locations.

Based on these requirements, we conclude that open addressing schemes are unsuitable for our use case. Probing insertion requires low load factors (thus high memory usage) and replacement strategies, such as Robin Hood and Cuckoo hashing, do not provide pointer stability. Furthermore, open addressing requires rehashing when the table becomes full. A closed addressing scheme, such as the SlabHash [AFCO18] is more suitable, even though it may perform slightly worse than open addressing [LZL*21].

SlabHash [AFCO18] stores a linked list of slabs. A slab is an array of 31 4-byte items plus a 4-byte pointer to the next slab. Empty slots are indicated by a sentinel value of zero, which is atomically swapped with the desired key when inserting. Combining multiple items in a single slab reduces stress on the memory allocator and allows for efficient processing on the GPU.

Extending SlabHash to support larger keys is challenging as GPU hardware does not provide atomic operations for values larger than 8 bytes. The size of our SVDAG nodes and leaves range from 2 to 10 *WORDs*, making atomic swapping of entire items impossible. However, if we do not support simultaneous insertions and look-ups, the first 8 bytes are enough to indicate an empty slot. Since a node will always have at least one child, the children bitmask is non-zero. Similarly, the bitmask of a leaf must contain at least one filled voxel. Therefore, in both cases, the first 8 bytes are never zero. Consequently, it is safe to use an 8-byte value of zero to indicate empty slots.

Hash-Table Variants

Here, we will discuss different variations for the hash table, which we tested for our application case; Atomic64, Ticket Board and two Acceleration Hash variants.

Atomic 64 is a straightforward extension of SlabHash storing the first 8 bytes (64 bits) of all items in a contiguous array at the start of the slab (Figure 5). These slots are followed by the remainder of the items (in case the items are larger than 8 bytes). They are stored in an Array-Of-Structures (AoS) layout, such that accessing an item typically requires loading two cache lines. Like SlabHash, the 32nd slot is reserved for the next pointer and a garbage collection bitmask. This way we can ensure that slabs are always aligned to a cache line (128 bytes on our hardware) for optimal memory throughput.

In line with previous work, insertion and search operations are performed at a warp level using 32 threads at a time (Algorithms 1 and 2). Each thread reads the first 8 bytes of its corresponding slot and compares it to the desired value. Subsequent insertion or comparison of the remainder of the item is also performed by the entire warp.

This memory layout is compact, but it does split nodes and leaves into multiple cache lines (first 8 bytes and the rest). This causes the most common operation, traversing the SVDAG, to be more expensive. While we found this to have little impact on primary visibility rays, where traversal is coherent, it does reduces path-tracing performance. Placing the camera inside a voxelization of the watertight Stanford bunny, we found that rendering performance decreased by 9% at 1 indirect bounce and over 16% at 6 indirect bounces.

Ticket Board, inspired by Stadium hashing [KBGB15], stores items contiguously in memory, making traversal more efficient. A bitmask at the start of the slab indicates whether each of the 32 slots is occupied. A design downside is that the search becomes less efficient; the entire slab needs to be loaded from memory, independent of whether it contains the item being searched for. Thus, there is no reason to strive for cache-line alignment, and we forgo padding.

Acceleration Hash is a novel variant, aiming to address the search performance of *ticket board*. Instead of a single bit, we store an array with a second hash of the item at each corresponding slot. The hash-value range starts at 1 such that a value of 0 indicates an empty slot. The 32 threads in a warp load their corresponding hash values, which are contiguous in memory, and compare them to the hash of the item being searched for. This allows warps to efficiently skip slabs with no matching items.

We consider two acceleration-hash versions. The first optimizes memory throughput by storing 32-bit (1 *WORD*) hash values. We again only use 31 slots in order to store the next pointer, while padding the slabs to the size of a cache line. The second version stores the hash in only 8-bits, reducing memory overhead from 4 bytes to just a single byte per slot. For compactness, we do not perform any memory padding and use 32 slots per slab (Figure 5).

Memory Allocation

When all slabs in a bucket are filled, we need to efficiently link a new slab. We use the SlabHash's *SlabAlloc* [AFCO18], to allocate

these (fixed-size) slabs. Here, we give a brief overview. Details are in the supplemental material (or the original work [AFCO18]).

The allocator maintains *memory blocks*, which contiguously store 1024 slabs, along with a bitmask to indicate the allocated slabs. Allocating from a *memory block* is performed by a warp of 32 threads. The bitmask is loaded in a single memory transaction; with each thread searching 32-bits of the mask for zero bits.

To efficiently allocate *memory blocks*, we do so in groups of 16, called *super blocks*. The amount of super-block allocations is determined by comparing the currently available memory to a conservative estimate during the edit operation. We then allocate super blocks if needed to ensure that memory is available for the edits. When adding nodes/leaves to the SVDAG, we track the actual amount of used slab entries through an atomic counter. Hereby, we can know the still available empty slabs for future edits.

To find an empty slab, when needed, a warp selects a *super block* based on its warp index. To distribute warps evenly, each warp selects one *memory block* at random within the chosen *super block*. If this selected *memory block* is full, the warp iterates the process with the next *super block* until it finds an empty location.

4.3. Editing

In this section, we discuss the implementation of editing operations. Our implementation provides a small number of editing tools such as 3D painting (placing new voxels), recolouring (changing the attributes), and a tool that copies a small region of the scene to a new location. We chose these operations, as they can be considered representative of a variety of operations - additional tools could easily be added.

4.3.1. SVO Construction

The first step of editing consists of constructing a Sparse Voxel *Octree* (SVO) of the scene as it is after the editing operation, which actually is a graph and not a tree, since unmodified regions of the scene still refer to the SVDAG (Figure 6a). We reserve a couple of bits of the node header to indicate for each child pointer, whether it points into the SVO or the main SVDAG (*reserved* bits in Figure 4).

The SVO is constructed following a top-down breadth-first traversal of the SVDAG. At each level, we collect the following information about the current nodes: their location, size, and whether they are homogeneous (fully filled with just a single material). This information is sent to the editing tool, which decides whether to either subdivide a region, fill it, or keep it as-is.

Simple editing tools like the 3D paint brush do not interact with the existing scene; they simply overwrite it. The copy tool however must traverse source and target regions. To do so efficiently, each editing tool can maintain a state during traversal, which was not supported by the original HashDAG framework. In the case of the copy tool, it can perform a dual traversal of the source- and target copy regions.

The traversal process is implemented using two GPU kernels: one for inner nodes and one for the leaves. Inner nodes are processed by groups of 8 threads, each determining whether one of the 8 child regions needs to be visited. These are appended to a work-queue using a combination of warp intrinsics and an atomic counter. After each level the counter is copied to the CPU in order to determine how many items the next level may at most produce. Since leaves represent regions of $4^3 = 64$ voxels, we spawn 64 thread workgroups. Hence, each thread processes a single voxel. The leaves are then constructed in shared memory before being copied to global memory.

4.3.2. Merging into the existing SVDAG

After the intermediate SVO is constructed, it needs to be merged into the existing SVDAG. This process is performed level by level, starting from the bottom, which allows for efficient detecting of duplicate subtrees (Section 3.1).

We split the process of duplicate elimination into two parts: duplicates within the SVO, and duplicates between the SVO and the existing SVDAG (Figures 6b and 6c, respectively). In a single threaded application, one could simply iterate over the SVO nodes/leaves, trying to find them in the SVDAG, and inserting them if they are not present yet. Running the same algorithm in a multi-threaded environment requires that search and subsequent insertion are a single atomic operation. This requires locking, which is expensive, especially with many duplicate items (which map to the same hash table buckets).

We work around this problem by eliminating duplicates before we search and insert them into the SVDAG. We have experimented with duplicate detection through sorting, but the performance cost dominated the editing time. Instead, we construct a second hash table into which all nodes/leaves at the current level are inserted, along with a pointer to their location in the SVO. Note that, in this case, we intentionally allow for occasional duplicate inserting. A subsequent search of all items identifies, for each group of duplicates, a single "leader". This is achieved by ensuring that the search operation always returns the first matching instance in a bucket.

After performing the previous operations, we are left with two SVDAGs (Figure 6b). To merge them, we check whether each node/leaf in the new SVDAG was already present in the original SVDAG. If this was not the case, then they are inserted into the original SVDAG.

Finally, child pointers of the parent SVO nodes are updated to represent these changes. During this step, we also check whether regions now represented by the updated parents are fully-empty or fully-filled regions. The former information may be used to accelerate the editing tools and is stored inside the node's header(Figure 4). The entire process is repeated for the parent levels until we reach the root node.

5. Implementation

In this section, we will discuss some of the implementation details regarding the hash tables and the SVDAG.

5.1. Hash Tables

The implementation of our SVDAG consists of multiple hash tables, each storing items of one specific size. Nodes and leaves share

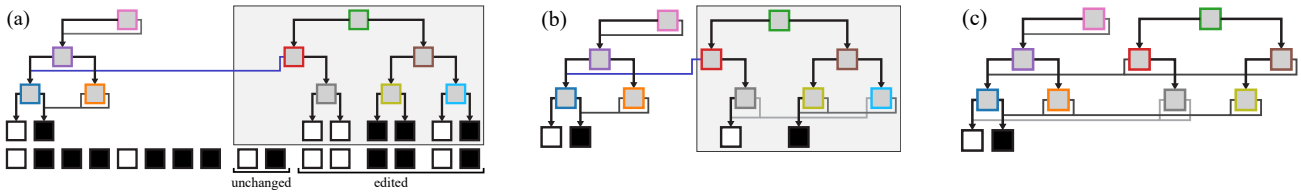

Figure 6: *Constructing an SVDAG while editing. Starting with the original SVDAG (a, left), we construct an SVO according to the editing tool (a, right). This tree points into the SVDAG for regions with no change (blue line). The SVO is converted into a SVDAG (b) using the bottom-up algorithm (Figure 2). The two are merged into a single DAG with multiple root nodes (d).*

the same hash table, which means that a single piece of memory may store both a node and a leaf at the same time when their bit pattern matches.

In line with existing research, all of the proposed hash tables perform insertions and searches using one warp per item. After computing to which bucket the item belongs, a warp iterates through the linked list of slabs, with each thread i checking the i'th slot. Depending on the hash table method (Section 4.2), this means either comparing the first 8 bytes, or a secondary hash. If one or more threads find a potential match, the remaining bytes of the item are checked to confirm. The pseudo-code for both operations can be found in the Appendix. For a line-by-line explanation of the algorithm, we refer the interested reader to the supplemental material.

5.2. Finding Duplicates in the SVO

As discussed in Section 4.3.2, we separate the process of eliminating duplicates within the intermediate SVO from eliminating duplicates in the SVDAG. All nodes and leaves in the SVO are padded, such that they are the same size in memory, and stored in a single contiguous array per level. To eliminate duplicates within these arrays, we initially experimented with sorting, but we found that using a specialized hash table provides better performance.

The requirements for this duplicate-detection hash table are more relaxed compared to the SVDAG hash tables (Section 4.2). While this does open the door for open addressing schemes, we stick with the slab approach for simplicity. We use a very similar memory layout to the 64-bit atomic scheme (Figure 5) with some notable change. Since the hash table needs to store both keys and values (SVO pointers), we store an additional 32-bit value for each slot at the end of the slab rather than just the keys.

With many duplicates, we run the risk of creating very long slab lists for some buckets. To reduce this issue, we check, when an item is inserted, whether it is already present in the first slab. If the first slab already contains the key, then the insertion is canceled. If not, we insert it anyways. This is not optimal but delivers a better trade-off between insertion and search performance in practice. We use the value (of the key/value pair) of a slot as a spinlock, to prevent threads from reading partially written items.

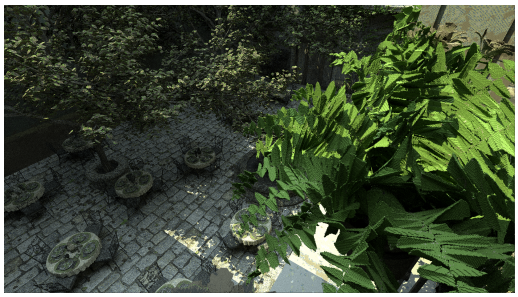

Figure 7: *Vegetation in the San Miguel scene.*

5.3. Garbage Collection

Since nodes and leaves may have multiple parents, releasing them from memory requires either reference counting or garbage collection. Reference counting adds an additional memory overhead, so we opted for garbage collection. Each slab of the hash table contains a 32-bit mask, which indicates for each slot, whether it is still being referenced. When the garbage collector is invoked, these masks are initialized to zero. We then traverse the SVDAG from top to bottom, using one kernel invocation per level, to activate the respective bits. Finally, we iterate over the slabs in the hash table, and set the inactive slots to empty. If an entire slab is empty, it is removed from the bucket and returned to the memory allocator.

Our garbage collection is a proof-of-concept implementation. It is triggered by a button in the user interface. Currently, the implementation has not been optimized, thus causing a momentary pause. However, this could be alleviated by running the garbage collection asynchronously. Both SVDAG traversal and hash-table iteration can be split into smaller steps, which can be interleaved with rendering. For example, one may traverse only a part of the SVDAG each frame, or iterate over a subset of the hash-table buckets. Alternatively, reference counting may deliver more consistent performance at the cost of increased memory usage.

6. Results

To evaluate our solution, we tested the proposed hash-table schemes separately, comparing also to existing work on GPU hash tables.We then test our entire SVDAG editing solution. All tests were performed on a machine containing an AMD 7950X3D (16 cores / 32 threads), 64GB RAM and an NVIDIA RTX4080 graphics card (PCIe4 x8) running PopOS 22.04 LTS.

6.1. Hash Tables

We evaluate four different hash-table schemes: *SlabHash* extended using 64-bit atomics and support for larger items, *ticket board*, and *acceleration hash using 32- or 8 bits* (Section 4.2). To test these hash tables, we create a simple test scenario that simulates editing behaviour. We initially fill each hash table with $2^{25} \approx 33M$ items.

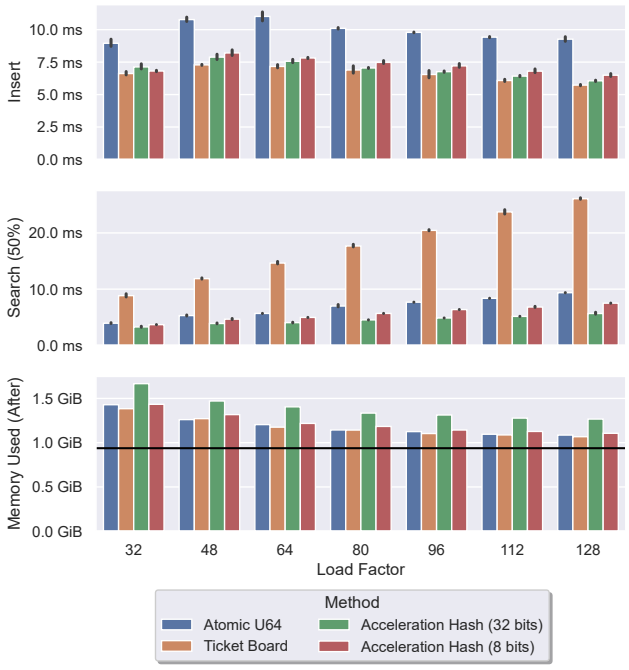

Figure 8: *Search and insertion performance as well as memory usage for different load factors. We search and insert 8M items into a table initially filled with 33M items of 6 WORDs (24 bytes) each. Memory usage is measured in slabs and thus includes unused slots. The black line indicates the size of the input data.*

We perform both a search- and insertion operation which mimics finding and subsequently inserting SVDAG nodes and leaves. Both operations are performed with $2^{23} \approx 8M$ million items.

Figure 8 shows the results of the benchmark at different load factors for an item size of 6 *WORDs* (24 bytes). Looking at insertion performance, we find that high load factors (lower number of buckets) result in improved performance despite an increase in thread contention. This is explained by an increase in the L1 and L2 cache hit rate. The opposite is true for search performance, which decreases, as it needs to iterate over a longer list of buckets. Note that this is not the case for insertions, as there we typically find an empty slot in the first slab.

We notice that both insertion and search performance are correlated with the number of bytes that need to be loaded from memory. This explains why the 64-bit method trails in search performance as it checks more bytes before deciding whether an item might match. The 32-bit acceleration hash leads to the best search performance as it initially loads only half the amount of memory. Reducing the size of the acceleration hash did not improve performance, which we can attribute to various factors. For example, the slabs are not aligned anymore, resulting in reduced memory bandwidth (due to memory coalescing). Additionally, by allowing only 255 unique hash codes, we increase the chance of hash collisions. We found that in practice, we check up to an additional 0.21 slots for each query. An interesting observation is that the search performance may increase as the hit rate (percentage of successful searches) goes down. When an item is not present, then the search operation needs to visit all slabs and cannot "early out" (when an item is found). However, comparing an item after a potential match is expensive due to non-coalesced memory access and branching. Especially for larger items, traversing all slabs during an unsuccessful search can be faster than having to compare an item.

Scene	Method	Memory
San Miguel 64K 4-bit Materials	Nodes/Leaves Only	2929 MiB
	Atomic U64	3509 MiB
	Ticket Board	**3461 MiB**
	Acceleration Hash (32 bits)	4269 MiB
	Acceleration Hash (8 bits)	3622 MiB
Citadel 128K No Materials	Nodes/Leaves Only	980 MiB
	Atomic U64	1199 MiB
	Ticket Board	**1155 MiB**
	Acceleration Hash (32 bits)	1400 MiB
	Acceleration Hash (8 bits)	1203 MiB
Citadel 128K 4-bit Materials	Nodes/Leaves Only	5997 MiB
	Atomic U64	7164 MiB
	Ticket Board	**7082 MiB**
	Acceleration Hash (32 bits)	8685 MiB
	Acceleration Hash (8 bits)	7404 MiB

Table 1: *Memory usage of the tested scenes both with (4-bit) and without (N/A) materials. This includes memory that is allocated but not currently used (partially filled slabs).*

Considering memory usage, we see that the additional 32-bits of acceleration hash cause significant overhead compared to the other three methods. The 64-bit atomic method uses additional memory for padding, whereas the other methods (additionally) spend some memory on acceleration hashes/ticket boards. Note that the results in Figure 8 include memory used by empty slots, i.e., the hash tables could potentially fit more items without growing in size.

Table 2 shows the results of the same benchmark for different item sizes. We include both SlabHash [AFCO18] and DyCuckoo [LZL*21] for comparison. As expected, insertion and search performance both scale with item size. Interestingly, we found that item sizes consisting of an even number of *WORDs* typically perform slightly better than item sizes with an odd number. We suspect that this is due to those items straddling cache lines more often. Comparing our methods to SlabHash, on which they are based, we see that insertion performance is comparable, while search performance is reduced. One of the reasons for this deficit is that, in order to support dynamic memory growth, we require an extra level of pointer indirection to access a slab. Although this is mentioned in the SlabHash paper, it is not implemented in the published code. Our methods also require more memory bandwidth (64-bit atomics) or additional computation (acceleration hash), which introduces some overhead.

6.2. SVDAG Editing

We will now evaluate the performance of our SVDAG editing system and compare it to the CPU-based HashDAG [CBE20]. For both methods, we target a load factor of 96 directly after the scene has been loaded. We use the same test scenes as in [CBE20], but

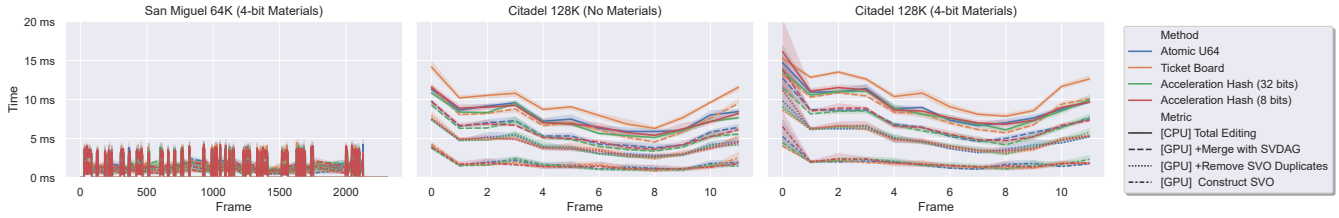

Figure 9: *Breaking down editing performance in San Miguel* $64K^3$ *and Epic Citadel* $128K^3$*, as measured on the CPU, into the most costly GPU kernels. The remaining time is spent on smaller GPU kernels, memory allocation, synchronisation and other CUDA related overhead.*

we convert the colors into a palette of 16 unique materials with similarly colored textures, mimicking popular voxel-based games. The used test scenes are Citadel (Figure 1) and San Miguel (Figure 7), which are voxelizations of textured triangle meshes. Citadel is stored at a voxel resolution of $128K^3$, whereas San Miguel uses a resolution of $64K^3$. The reason for this difference is the elongated shape of the Citadel scene, which results in a higher level of sparsity.

Table 1 shows the memory usage of the tested scenes. Including materials in the SVDAG significantly increases memory usage due to a combination of larger leaves and a reduction in repetition (as subtrees only match if shape and material are identical). Materials are assigned based on the diffuse textures of the original triangle mesh. This creates a high level of local material detail, which may not be present in other datasets. Despite this, storing the same scenes in a Sparse Voxel Octree (SVO) would have required roughly 4x more memory (22*GiB* Citadel, 14*GiB* San Miguel) compared to our SVDAG. When only storing voxel occupancy, without materials, the SVDAG uses only 6.5% of the memory compared to an SVO.

We test our method using two different tools. We start with a *sphere tool*, which acts as a three-dimensional paint brush, placing voxels inside the spherical brush radius (Figure 1). This creates a stress-test for duplicate detection, as the surface of the sphere has many repeating patterns. Note that filling the inside of the spheres is cheap, as filling large homogeneous regions of space is a special case, well supported by our solution. For the second test, we use the copy tool to gradually make a copy of some of the vegetation in the San Miguel scene. This tool copies a cubic region around the mouse cursor. To copy a large object the user drags the cursor along the surface as it progressively copies.

The results of both tests are shown in Figure 9. Placing spheres in the Citadel scene takes between 15ms and 18ms depending on whether materials are enabled. The radii of the spheres vary between 1140 and 1820 voxels per frame, which is reflected in the frame times. The largest ones occupy roughly 25 billion voxels, which creates an intermediate Sparse Voxel *Octree* of 114MiB when materials are enabled. Please note that this octree does not subdivide the inside of the spheres. SVDAG compression reduces this to 23MiB for the first sphere and merely 15MiB for subsequent spheres as geometric repetition increases. Copying is distributed over many frames and operates within a couple of milliseconds, which is well within the realm of real-time frame rates.

Figure 9 breaks down the total editing time (CPU wall clock) into the most costly GPU kernels (GPU time) (please see the supplementary material for a more detailed break down). Considering the Citadel scene without materials; traversing the editing region and constructing the temporary Sparse Voxel *Octree* takes 1.7*ms* on average. Removing duplicate nodes and leaves within this octree (Section 5.2) is a relatively costly operation at just over 2.5*ms*. Finally, finding duplicates between the SVO and SVDAG and inserting the unique items takes between 1.3*ms* and 3.2*ms* depending on the hash table. These timings increase by around 30% when enabling materials due to an increase in SVDAG leaf sizes. The remaining time is spent on various other GPU kernels (updating parent pointers, detecting homogeneous regions, etc.) and some CPU/GPU synchronisation. We found that this synchronisation is negatively influenced by memory allocation calls, such as *cudaMalloc* or *cudaMallocAsync*, which is why we replace all memory allocations by the Vulkan Memory Allocator [AMD23].

From the four proposed hash-table implementations, the two acceleration hash methods perform best, followed by the 64-bit atomics method. This is exclusively due to a difference in search performance with 0.94*ms* for the *atomics* method versus 0.87*ms* and 0.70*ms* for the *8-bit and 32-bit acceleration hash* methods, respectively. The *ticket board* method performs significantly worse at 2.51*ms*. The cost of inserting new nodes and leaves into the hash tables is comparable for all four methods at roughly 0.6*ms*. Based on both run-time performance and memory usage we conclude that the *atomics* and the *8-bit acceleration hash* methods are most suitable for our use case.

To compare against existing work, we perform the same editing operations using the HashDAG [CBE20]. Some of the improvements we made, such as faster GPU memory allocation, were integrated in the HashDAG. To ensure a fair comparison, we disable voxel colours, as they are not directly comparable to our material system. It takes the HashDAG 52.6*ms* to place the spheres in the Citadel scene using 32 CPU threads, which is roughly 5 times slower than our GPU-based method. Additionally, the HashDAG requires 192.8*ms* to copy these changes to the GPU, which is necessary to render the updated scene. CPU editing performance has improved with the availability of high core-count processors, yet the cost of CPU-to-GPU memory transfer remains a bottleneck.

7. Conclusion

Sparse Voxel Directed Acylcic Graphs (SVDAG) have proven to be a memory efficient data structure for storing dynamic sparse voxel data. In this work, we have shown that it is possible to edit SVDAGs entirely on the graphics card, improving vastly the editing performance compared to previous CPU methods and circumventing the

expensive CPU/GPU memory copies. The main building block is a GPU-based dynamic hash table, which can store large items and provides pointer stability. We propose four different implementations inspired by SlabHash [AFCO18]. Despite our additional requirements, stemming from the SVDAG support, the performance of our solution remains close to the state-of-the-art.

Our GPU-driven SVDAG editing pipeline consists of almost a dozen compute kernels and an additional hash table optimized to deal with duplicate removal. We enable large SVDAG modifications in real-time, while reducing memory usage by several factors compared to an octree.

Our application supports simple voxel attributes stored inside the SVDAG. While this works well for small and coherent attributes, additional memory savings may be achieved by separating geometry from attributes and compressing them separately. Current methods [DKB*16, DSKA17, ME23] are not optimized for many fragmented modifications however. We believe this to be an interesting avenue for future work.

References

[AFCO18] ASHKIANI S., FARACH-COLTON M., OWENS J. D.: A dynamic hash table for the gpu. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2018), IEEE, pp. 419–429. 3, 4, 5, 6, 8, 10

[AMD23] AMD GPUOPEN: Vulkan memory allocator. https://gpuopen.com/vulkan-memory-allocator/, 2023. 9

[ASA*09] ALCANTARA D. A., SHARF A., ABBASINEJAD F., SENGUPTA S., MITZENMACHER M., OWENS J. D., AMENTA N.: Real-time parallel hashing on the gpu. In *ACM SIGGRAPH Asia 2009 Papers* (New York, NY, USA, 2009), SIGGRAPH Asia '09, Association for Computing Machinery. doi:10.1145/1661412.1618500. 2, 3

[AVS*12] ALCANTARA D. A., VOLKOV V., SENGUPTA S., MITZENMACHER M., OWENS J. D., AMENTA N.: Building an efficient hash table on the gpu. In *GPU Computing Gems Jade Edition*. Elsevier, 2012, pp. 39–53. 3, 4

[CBE20] CAREIL V., BILLETER M., EISEMANN E.: Interactively Modifying Compressed Sparse Voxel Representations. *Computer Graphics Forum* (2020). doi:10.1111/cgf.13916. 2, 3, 4, 8, 9

[CNS*11] CRASSIN C., NEYRET F., SAINZ M., GREEN S., EISEMANN E.: Interactive indirect illumination using voxel cone tracing: a preview. In *Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2011), I3D '11, Association for Computing Machinery, p. 207. URL: https://doi.org/10.1145/1944745.1944787, doi:10.1145/1944745.1944787. 1

[DGPR02] DEBRY D., GIBBS J., PETTY D. D., ROBINS N.: Painting and rendering textures on unparameterized models. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (2002), pp. 763–768. 1

[DKB*16] DADO B., KOL T. R., BAUSZAT P., THIERY J.-M., EISEMANN E.: Geometry and attribute compression for voxel scenes. *Computer Graphics Forum 35*, 2 (2016), 397–407. doi:https://doi.org/10.1111/cgf.12841. 2, 10

[DSKA17] DOLONIUS D., SINTORN E., KÄMPE V., ASSARSSON U.: Compressing color data for voxelized surface geometry. *IEEE transactions on visualization and computer graphics 25*, 2 (2017), 1270–1282. 2, 10

[GG98] GAEDE V., GÜNTHER O.: Multidimensional access methods. *ACM Computing Surveys (CSUR) 30*, 2 (1998), 170–231. 2

[GLHL11] GARCÍA I., LEFEBVRE S., HORNUS S., LASRAM A.: Coherent parallel hashing. *ACM Transactions on Graphics (TOG) 30*, 6 (2011), 1–8. 3, 4

[Hoe16] HOETZLEIN R. K.: GVDB: Raytracing Sparse Voxel Database Structures on the GPU. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics* (2016), Assarsson U., Hunt W., (Eds.), The Eurographics Association. doi:10.2312/hpg.20161197. 1

[JHS18] JÜNGER D., HUNDT C., SCHMIDT B.: Warpdrive: Massively parallel hashing on multi-gpu nodes. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2018), IEEE, pp. 441–450. 3, 4

[Kar12] KARRAS T.: Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics* (2012), pp. 33–37. 2

[KBGB15] KHORASANI F., BELVIRANLI M. E., GUPTA R., BHUYAN L. N.: Stadium hashing: Scalable and flexible hashing on gpus. In *2015 International Conference on Parallel Architecture and Compilation (PACT)* (2015), IEEE, pp. 63–74. 3, 5

[KKK18] KIM Y., KIM B., KIM Y. J.: Dynamic deep octree for high-resolution volumetric painting in virtual reality. *Computer Graphics Forum 37*, 7 (2018), 179–190. doi:https://doi.org/10.1111/cgf.13558. 1

[KSA13] KÄMPE V., SINTORN E., ASSARSSON U.: High resolution sparse voxel dags. *ACM Transactions on Graphics (TOG) 32*, 4 (2013), 1–13. 2, 3

[Kuž21] KUŽEL V.: Real-time voxel visualization and editing for 3d printing. 1

[LH06] LEFEBVRE S., HOPPE H.: Perfect spatial hashing. *ACM Transactions on Graphics (TOG) 25*, 3 (2006), 579–588. 2, 3

[LHN*05] LEFEBVRE S., HORNUS S., NEYRET F., ET AL.: Octree textures on the gpu. *GPU gems 2* (2005), 595–613. 1

[LK10] LAINE S., KARRAS T.: Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics 17*, 8 (2010), 1048–1059. 2

[LZL*21] LI Y., ZHU Q., LYU Z., HUANG Z., SUN J.: Dycuckoo: dynamic hash tables on gpus. In *2021 IEEE 37th international conference on data engineering (ICDE)* (2021), IEEE, pp. 744–755. 3, 4, 5, 8

[ME23] MOLENAAR M., EISEMANN E.: Editing compressed high-resolution voxel scenes with attributes. *Computer Graphics Forum 42*, 2 (2023), 235–243. doi:https://doi.org/10.1111/cgf.14757. 2, 10

[Mor66] MORTON G. M.: A computer oriented geodetic data base and a new technique in file sequencing. 2

[PL10] PANTALEONI J., LUEBKE D.: Hlbvh: hierarchical lbvh construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics* (2010), pp. 87–95. 2

[PR04] PAGH R., RODLER F. F.: Cuckoo hashing. *Journal of Algorithms 51*, 2 (2004), 122–144. 3

[vdLSE20] VAN DER LAAN R., SCANDOLO L., EISEMANN E.: Lossy geometry compression for high resolution voxel scenes. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 3*, 1 (2020), 1–13. 2

[VMG16] VILLANUEVA A. J., MARTON F., GOBBETTI E.: Ssvdags: Symmetry-aware sparse voxel dags. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2016), pp. 7–14. 2

[ZWY*15] ZHANG K., WANG K., YUAN Y., GUO L., LEE R., ZHANG X.: Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment 8*, 11 (2015), 1226–1237. 3, 4

Item Size (U32)	Method	Insert	Search (25%)	Search (50%)	Search (75%)	Memory Used (Initial)
1	SlabHash	5.8 ms	**3.1 ms**	**2.8 ms**	**2.4 ms**	-
	DyCuckoo	**4.7 ms**	3.1 ms	2.8 ms	2.4 ms	-
2	DyCuckoo	5.5 ms	**3.5 ms**	**3.1 ms**	**2.7 ms**	-
	Atomic U64	6.9 ms	7.6 ms	6.9 ms	6.2 ms	**318.7 MiB**
	Ticket Board	**5.4 ms**	11.0 ms	9.9 ms	8.9 ms	328.4 MiB
	Acceleration Hash (32 bits)	5.8 ms	4.0 ms	4.1 ms	4.3 ms	478.1 MiB
	Acceleration Hash (8 bits)	5.9 ms	5.6 ms	5.6 ms	5.4 ms	362.7 MiB
3	Atomic U64	9.6 ms	7.9 ms	7.6 ms	7.3 ms	**478.1 MiB**
	Ticket Board	**5.7 ms**	13.3 ms	13.4 ms	11.0 ms	485.1 MiB
	Acceleration Hash (32 bits)	6.2 ms	**4.1 ms**	**4.4 ms**	**4.8 ms**	637.3 MiB
	Acceleration Hash (8 bits)	6.4 ms	5.9 ms	5.9 ms	5.9 ms	519.4 MiB
4	Atomic U64	9.5 ms	7.9 ms	7.5 ms	7.1 ms	**637.4 MiB**
	Ticket Board	**6.0 ms**	15.8 ms	14.5 ms	13.2 ms	642.0 MiB
	Acceleration Hash (32 bits)	6.2 ms	**4.0 ms**	**4.3 ms**	**4.5 ms**	796.7 MiB
	Acceleration Hash (8 bits)	6.6 ms	5.9 ms	6.0 ms	6.0 ms	676.3 MiB
5	Atomic U64	10.0 ms	8.2 ms	7.9 ms	7.7 ms	**796.2 MiB**
	Ticket Board	**6.2 ms**	18.8 ms	17.3 ms	15.8 ms	798.6 MiB
	Acceleration Hash (32 bits)	6.6 ms	**4.3 ms**	**4.8 ms**	**5.2 ms**	955.6 MiB
	Acceleration Hash (8 bits)	6.9 ms	6.1 ms	6.3 ms	6.4 ms	832.9 MiB
6	Atomic U64	9.8 ms	8.1 ms	7.7 ms	7.3 ms	956.1 MiB
	Ticket Board	**6.4 ms**	22.7 ms	20.4 ms	19.2 ms	**955.7 MiB**
	Acceleration Hash (32 bits)	6.8 ms	**4.4 ms**	**4.9 ms**	**5.4 ms**	1115.4 MiB
	Acceleration Hash (8 bits)	7.1 ms	6.2 ms	6.4 ms	6.6 ms	990.1 MiB
7	Atomic U64	10.4 ms	8.4 ms	8.3 ms	8.2 ms	1115.1 MiB
	Ticket Board	**6.6 ms**	25.9 ms	23.7 ms	21.5 ms	**1112.4 MiB**
	Acceleration Hash (32 bits)	7.0 ms	**4.6 ms**	**5.2 ms**	**5.7 ms**	1274.6 MiB
	Acceleration Hash (8 bits)	7.3 ms	6.4 ms	6.7 ms	7.0 ms	1146.7 MiB
8	Atomic U64	10.3 ms	8.3 ms	8.1 ms	8.0 ms	1274.3 MiB
	Ticket Board	6.8 ms	29.2 ms	26.8 ms	24.7 ms	**1269.2 MiB**
	Acceleration Hash (32 bits)	**6.7 ms**	**4.3 ms**	**4.5 ms**	**4.8 ms**	1433.9 MiB
	Acceleration Hash (8 bits)	7.5 ms	6.4 ms	6.7 ms	7.0 ms	1303.8 MiB
9	Atomic U64	10.7 ms	8.6 ms	8.6 ms	8.7 ms	1433.5 MiB
	Ticket Board	**7.0 ms**	29.3 ms	27.3 ms	24.9 ms	**1425.8 MiB**
	Acceleration Hash (32 bits)	7.5 ms	**4.8 ms**	**5.5 ms**	**6.1 ms**	1592.6 MiB
	Acceleration Hash (8 bits)	7.7 ms	6.6 ms	7.0 ms	7.4 ms	1459.9 MiB
10	Atomic U64	10.3 ms	8.4 ms	8.0 ms	7.6 ms	1593.2 MiB
	Ticket Board	**7.2 ms**	30.3 ms	27.4 ms	25.6 ms	**1582.6 MiB**
	Acceleration Hash (32 bits)	7.5 ms	**4.9 ms**	**5.5 ms**	**6.2 ms**	1752.2 MiB
	Acceleration Hash (8 bits)	7.8 ms	6.7 ms	7.1 ms	7.5 ms	1616.6 MiB

Table 2: *Results for searching 8M Items and subsequently inserting 8M Items into a hash table initially storing 33M Items targeting a load factor (after insertion) of 96. Search performance is evaulated for various hit rates (number of search operations that succeed).*

Appendix A: Search and insertion algorithms

Algorithm 1 Warp-centric search operation

```
1: threadMask ← 0x7FFFFFFF
2: hash ← HashFunction(needle)
3: search64 ← U64(needle)
4:
5: slab ← table[hash % numBuckets]
6: while slab ≠ end do
7:     comp64 ← U64(slab[2 * threadIdx])
8:     match ← comp64 = search64
9:     if warp.ballot(match) & threadMask ≠ 0 then
10:        if match then
11:            for i = 0 to itemSize − 2 do
12:                comp32 ← slab[64 + threadIdx × (itemSize − 2) + i]
13:                if needle[2 + i] ≠ comp32 then
14:                    match ← 0
15:                    break
16:                end if
17:            end for
18:        end if
19:
20:        activeMask ← warp.ballot(match) & threadMask
21:        if activeMask ≠ 0 then
22:            outThreadIdx ← bitscan(activeMask)
23:            return encodePointer(slab, outThreadIdx)
24:        end if
25:    end if
26:    slab = slab[62]
27: end while
```

Algorithm 2 Pseudocode to insert an item into the atomic64 hash table. Both the item to be inserted (*needle*) and the slabs are pointers to 32-bit *WORDs*. We have omitted memory fences for brievity.

```
1: threadMask ← 0x7FFFFFFF
2: hash ← HashFunction(needle)
3: insert64 ← U64(needle)
4:
5: slab ← table[hash % numBuckets]
6: loop
7:     slab = warp.shfl(slab)
8:     if slab = end then                          ▷ Add new slab to bucket
9:         newSlab ← allocateAsWarp()
10:        newSlab[threadIdx] ← 0
11:        newSlab[32 + threadIdx] ← 0
12:        if threadIdx = 30 then
13:            newSlab[62] ← table[bucket]
14:            h ← atomicCAS(table[bucket], slab[62], newSlab)
15:            if h = slab[62] then
16:                slab ← newSlab
17:            else
18:                free(newSlab)
19:                slab ← h
20:            end if
21:        end if
22:    end if
23:                                        ▷ Attempt inserting first 64 bits
24:    comp64 ← slab[threadIdx]
25:    activeMask ← warp.ballot(comp64 = 0) & threadMask
26:    if activeMask ≠ 0 then
27:        outThreadIdx ← bitscan(activeMask)
28:        if threadIdx = outThreadIdx then
29:            prev ← atomicCAS(slab[2 * threadIdx], 0, insert64)
30:            if prev = 0 then
31:                inserted ← 1
32:            end if
33:        end if
34:                                        ▷ Insert remaining bytes
35:        if warp.shfl(inserted, outThreadIdx) then
36:            if threadIdx ≤ itemSize − 2 then
37:                slab[64 + outThreadIdx * (itemSize − 2) + threadIdx] ← needle[2 + threadIdx]
38:            end if
39:            return encodePointer(slab, outThreadIdx)
40:        end if
41:    else
42:        slab ← slab[62]
43:    end if
44: end loop
```