Editing Compact Voxel Representations on the GPU

Mathijs Molenaar, Elmar Eisemann

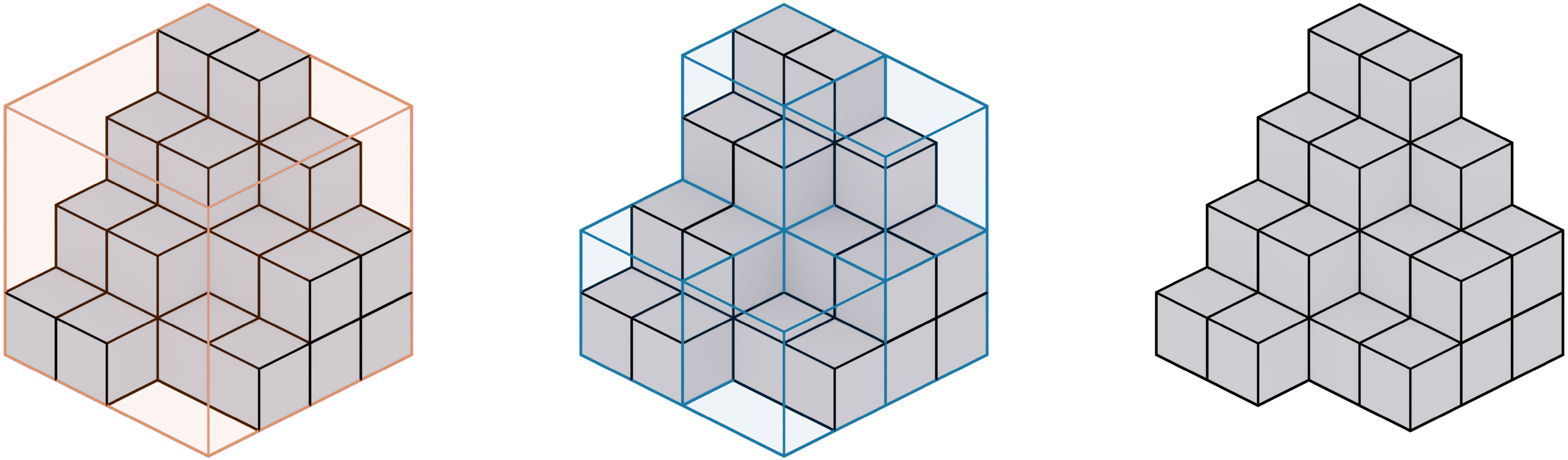
Delft University of Technology

Editing **Compact Voxel Representations** on the GPU

Sparse Voxel Octree

Voxel Octree Recursively subdivide space into evenly sized 2^3 regions

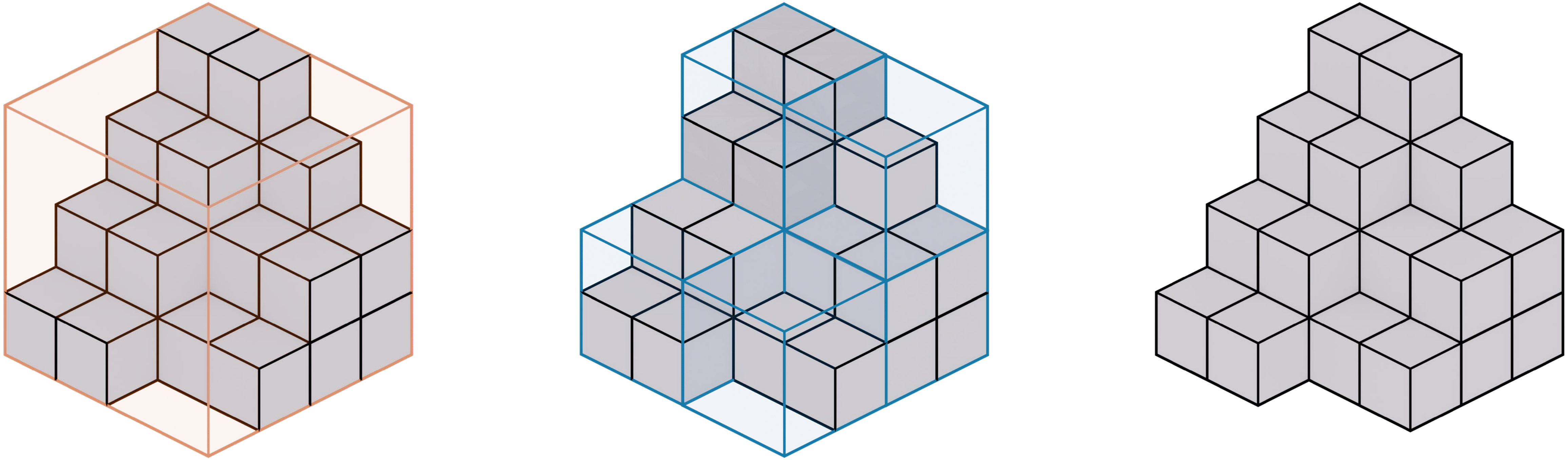
Sparse Don’t subdivide empty regions



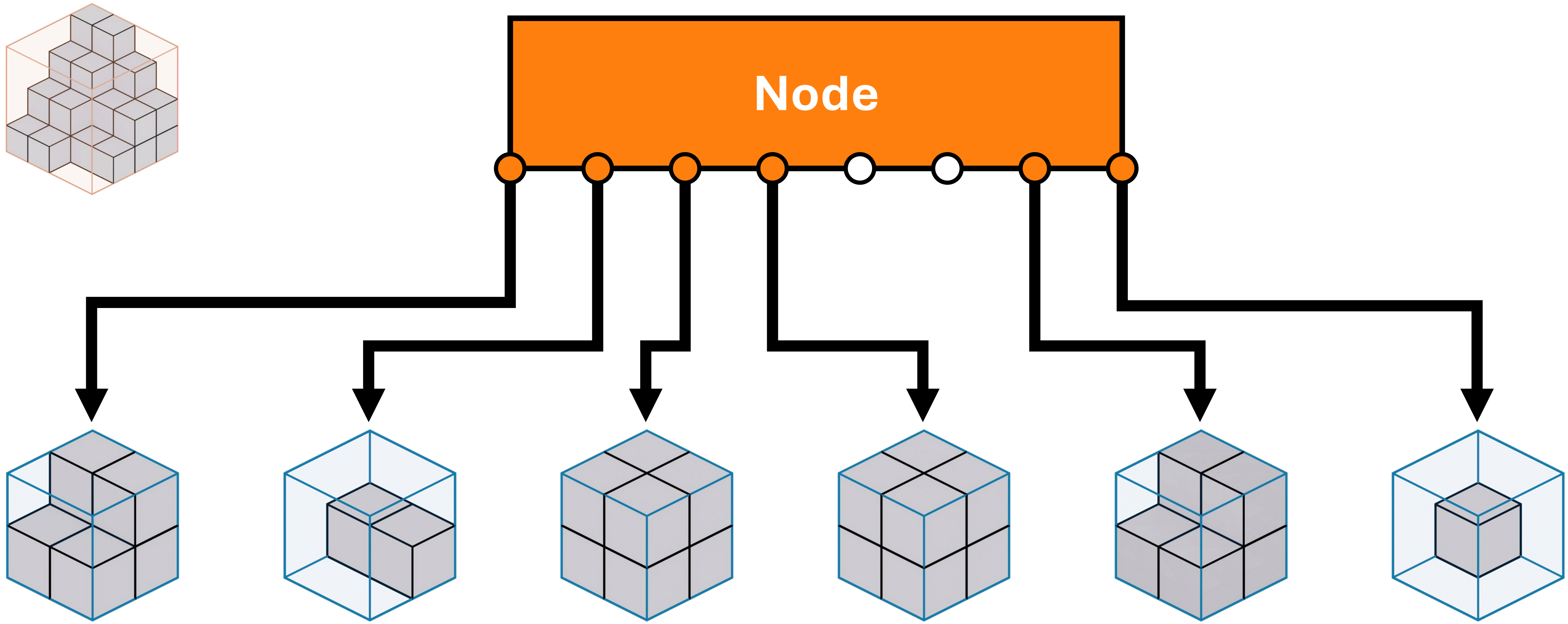
Sparse Voxel Octree

Voxel Octree Recursively subdivide space into evenly sized 2^3 regions

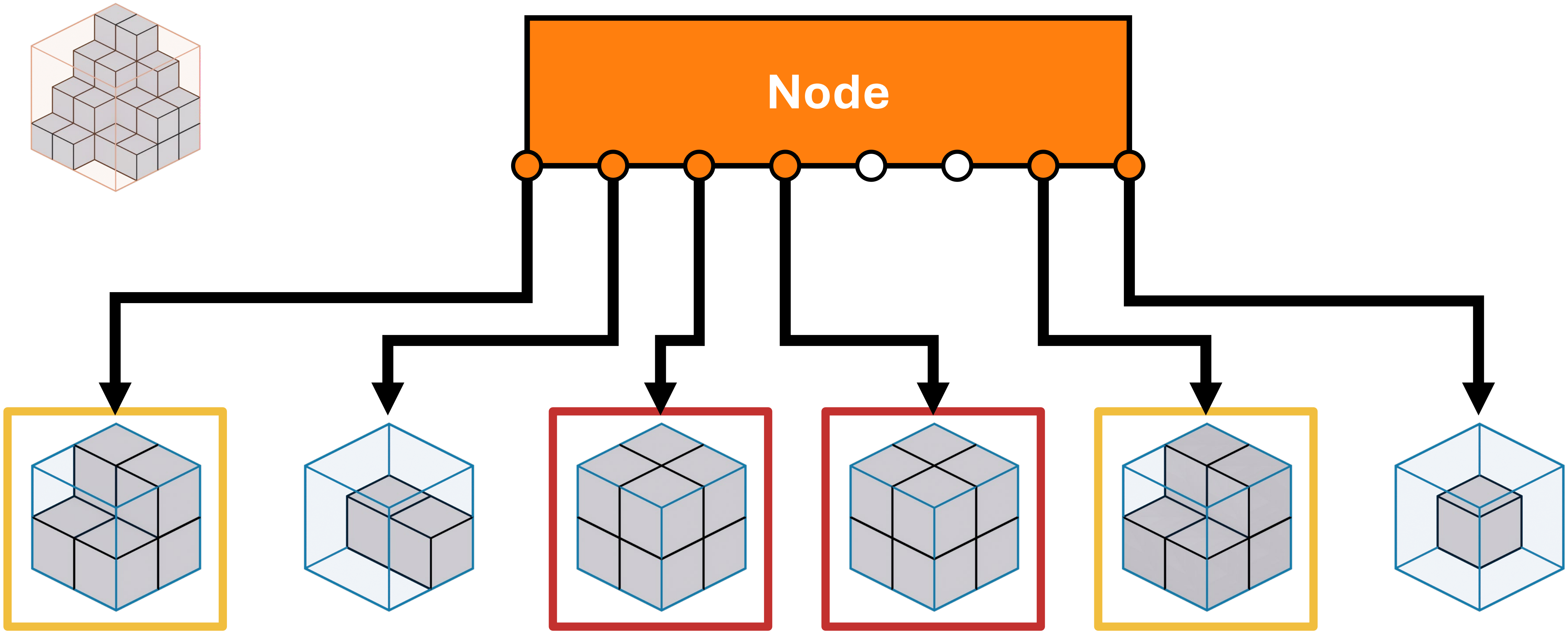
Sparse Don't subdivide empty regions



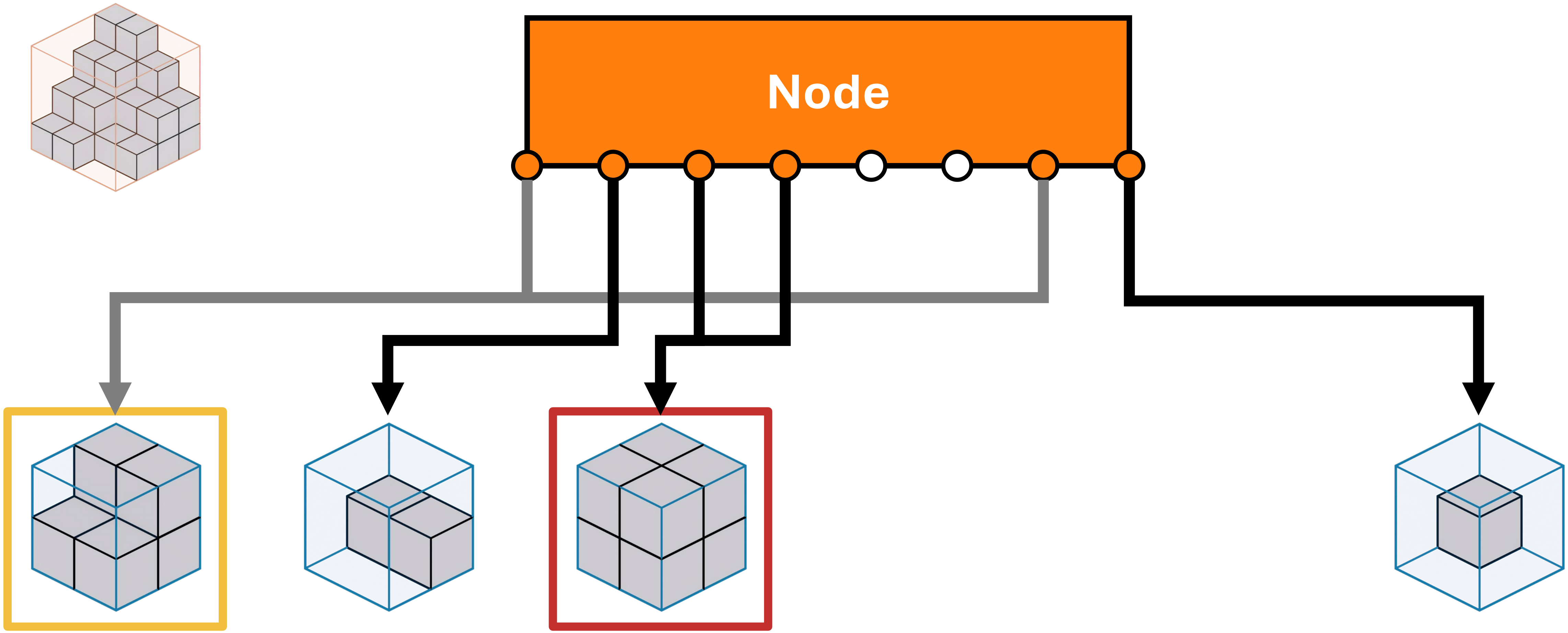
Sparse Voxel Octree (SVO)



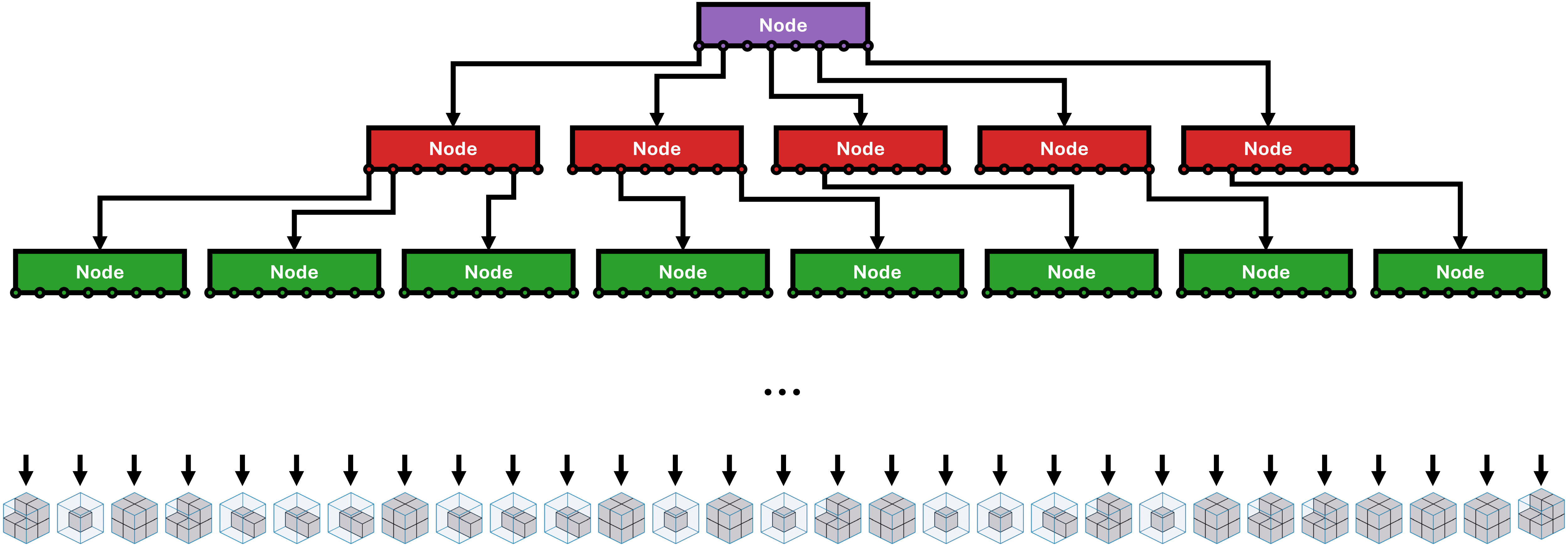
Sparse Voxel Directed Acyclic Graph (SVDAG) [KSA13]



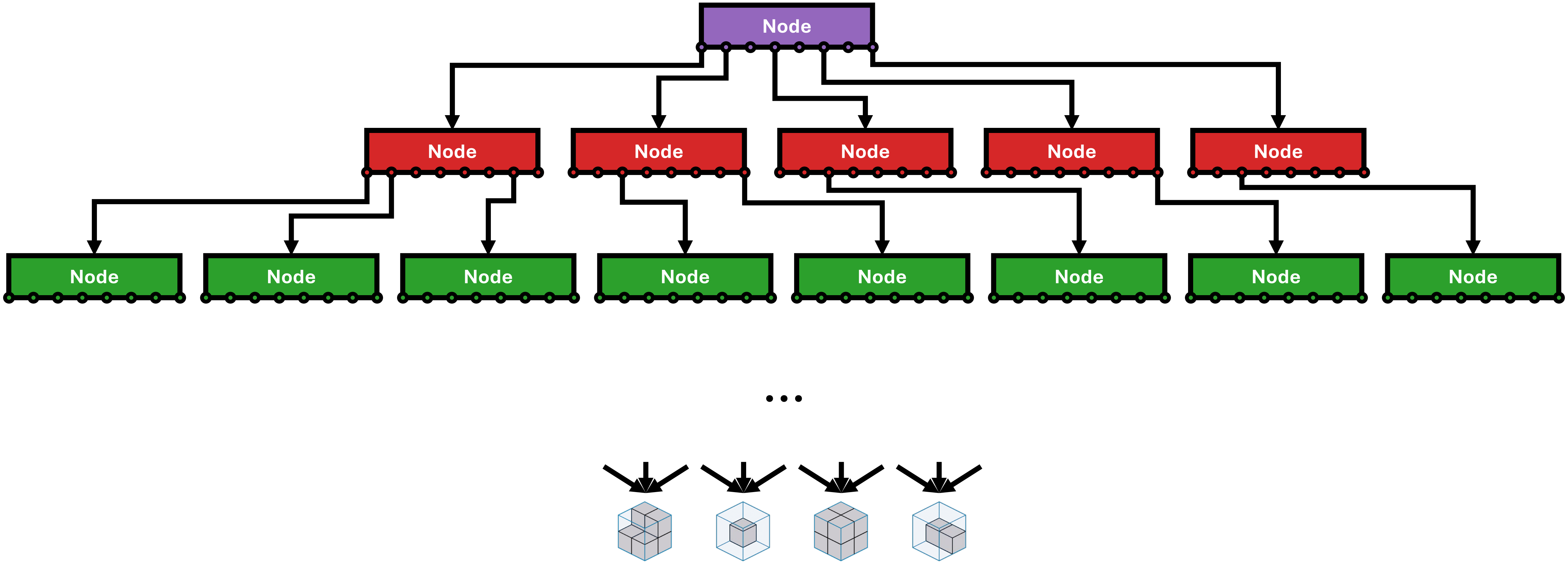
Sparse Voxel Directed Acyclic Graph (SVDAG) [KSA13]



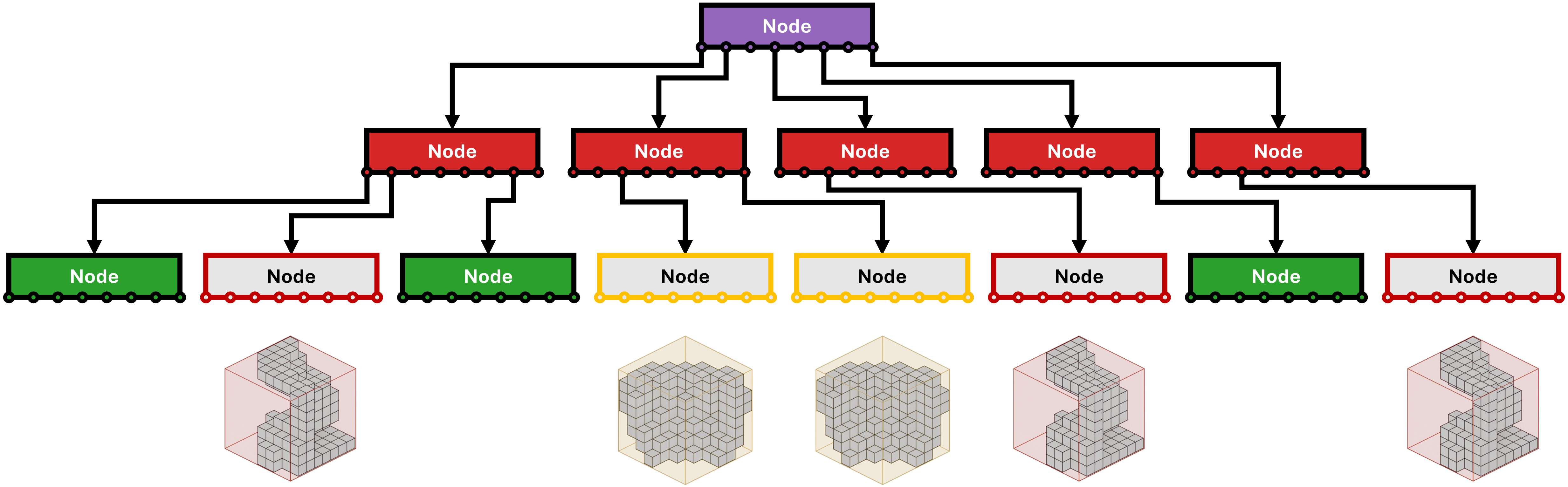
Sparse Voxel Directed Acyclic Graph (SVDAG) [KSA13]



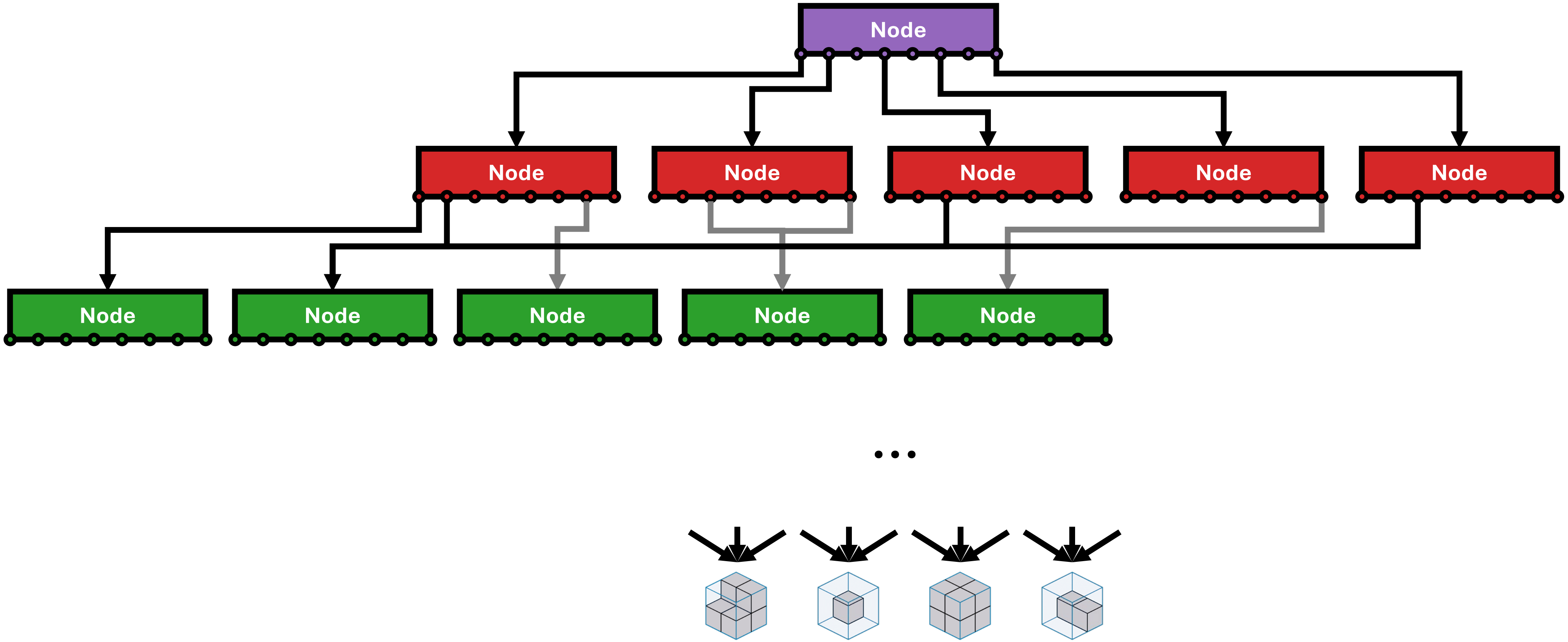
Sparse Voxel Directed Acyclic Graph (SVDAG) [KSA13]



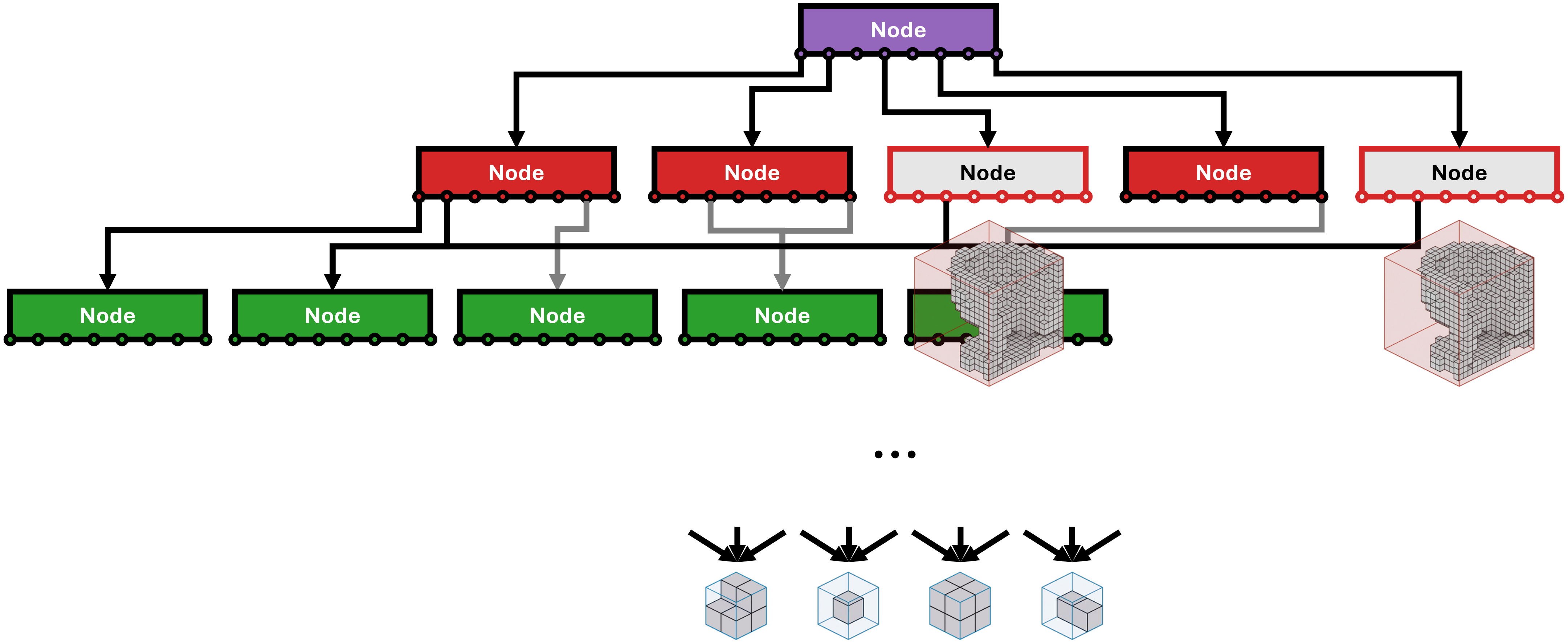
Sparse Voxel Directed Acyclic Graph (SVDAG) [KSA13]



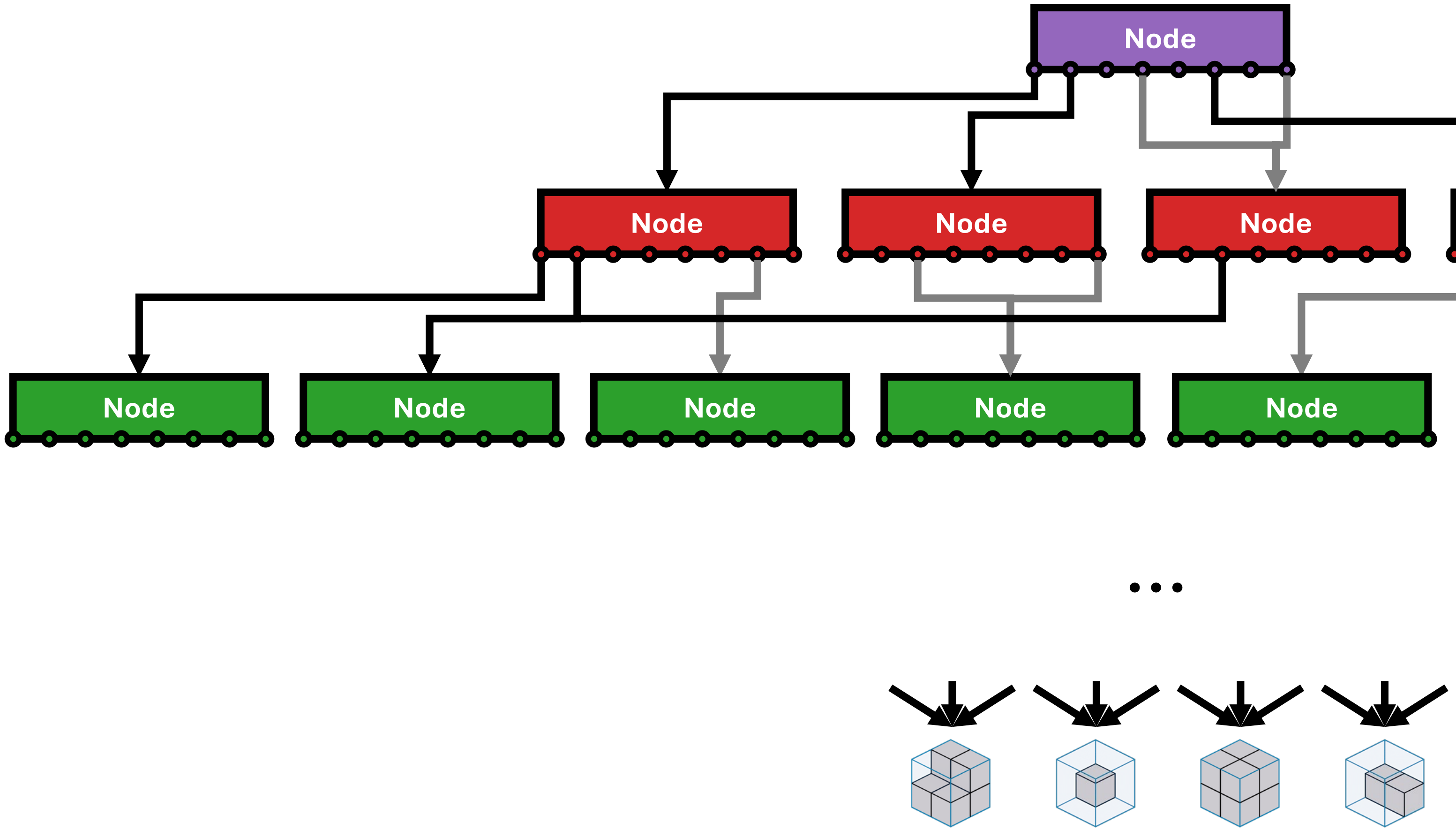
Sparse Voxel Directed Acyclic Graph (SVDAG) [KSA13]



Sparse Voxel Directed Acyclic Graph (SVDAG) [KSA13]



Sparse Voxel Directed Acyclic Graph (SVDAG) [KSA13]



Memory Usage

Scene		SVO	SVDAG [KSA13]	
Citadel $128K^3$ (No Materials) 13.7B occupied voxels		15,117 MiB	980 MiB	**15.4x**
Citadel $128K^3$ (4-bit Materials) 13.7B occupied voxels		22,516 MiB	5,997 MiB	**3.7x**
San Miguel $64K^3$ (4-bit Materials) 10.3B occupied voxels		14,865 MiB	2,929 MiB	**5.0x**

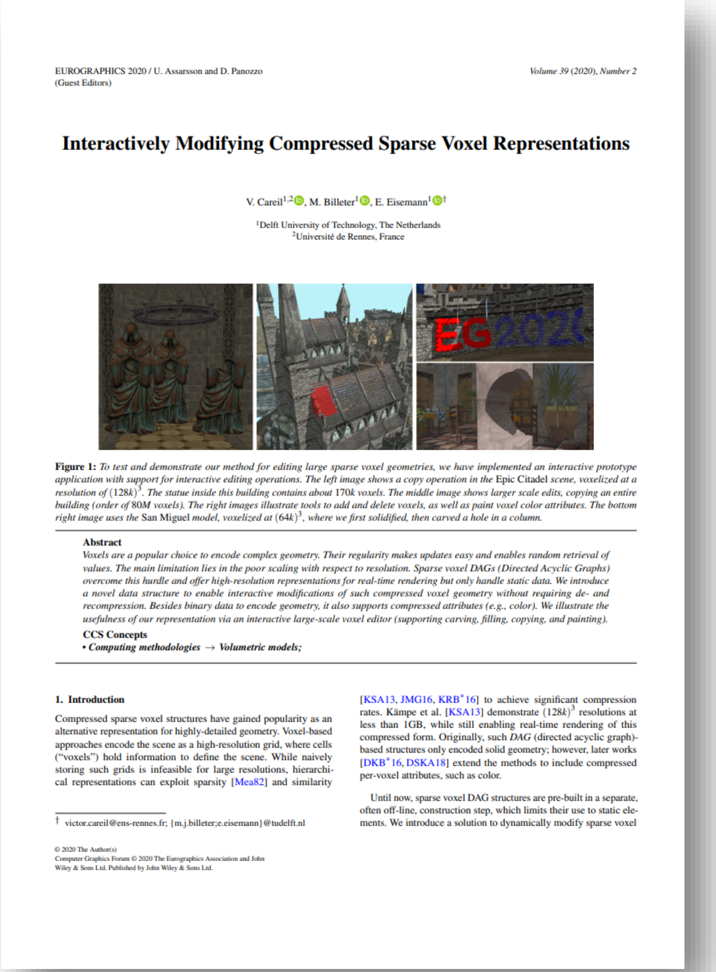
Editing Compact Voxel Representations on the GPU

HashDAG *[CBE20]*

Make modifications to existing SVDAG file

Performance considerations

- Editing **C**PU (Multi Threaded)
- Rendering **G**PU



EUROGRAPHICS 2020 / U. Assarsson and D. Panozzo (Guest Editors)

Volume 39 (2020), Number 2

Interactively Modifying Compressed Sparse Voxel Representations

V. Careil[1,2], M. Billeter[1], E. Eisemann[1]†

[1]Delft University of Technology, The Netherlands
[2]Université de Rennes, France

Figure 1: *To test and demonstrate our method for editing large sparse voxel geometries, we have implemented an interactive prototype application with support for interactive editing operations. The left image shows a copy operation in the* Epic Citadel *scene, voxelized at a resolution of* $(128k)^3$. *The statue inside this building contains about* 170*k voxels. The middle image shows larger scale edits, copying an entire building (order of* 80*M voxels). The right images illustrate tools to add and delete voxels, as well as paint voxel color attributes. The bottom right image uses the* San Miguel *model, voxelized at* $(64k)^3$, *where we first solidified, then carved a hole in a column.*

Abstract

Voxels are a popular choice to encode complex geometry. Their regularity makes updates easy and enables random retrieval of values. The main limitation lies in the poor scaling with respect to resolution. Sparse voxel DAGs (Directed Acyclic Graphs) overcome this hurdle and offer high-resolution representations for real-time rendering but only handle static data. We introduce a novel data structure to enable interactive modifications of such compressed voxel geometry without requiring de- and recompression. Besides binary data to encode geometry, it also supports compressed attributes (e.g., color). We illustrate the usefulness of our representation via an interactive large-scale voxel editor (supporting carving, filling, copying, and painting).

CCS Concepts

• ***Computing methodologies*** → ***Volumetric models;***

1. Introduction

Compressed sparse voxel structures have gained popularity as an alternative representation for highly-detailed geometry. Voxel-based approaches encode the scene as a high-resolution grid, where cells ("voxels") hold information to define the scene. While naively storing such grids is infeasible for large resolutions, hierarchical representations can exploit sparsity [Mea82] and similarity [KSA13, JMG16, KRB*16] to achieve significant compression rates. Kämpe et al. [KSA13] demonstrate $(128k)^3$ resolutions at less than 1GB, while still enabling real-time rendering of this compressed form. Originally, such *DAG* (directed acyclic graph)-based structures only encoded solid geometry; however, later works [DKB*16, DSKA18] extend the methods to include compressed per-voxel attributes, such as color.

Until now, sparse voxel DAG structures are pre-built in a separate, often off-line, construction step, which limits their use to static elements. We introduce a solution to dynamically modify sparse voxel

† victor.careil@ens-rennes.fr; {m.j.billeter;e.eisemann}@tudelft.nl

HashDAG *[CBE20]* – Fixed-Size Hash Table

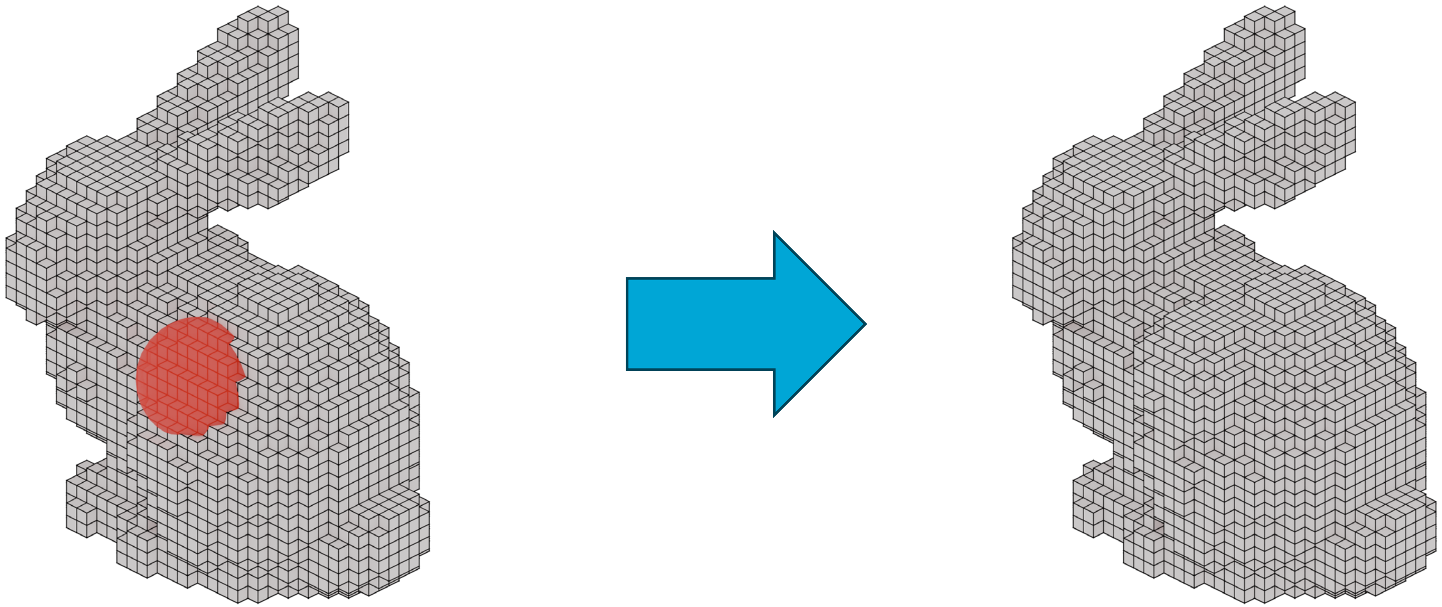
Hash table size heuristically determined at start-up:

- Number of buckets
- Maximum size of each bucket

Custom virtual memory system to store the hash table.

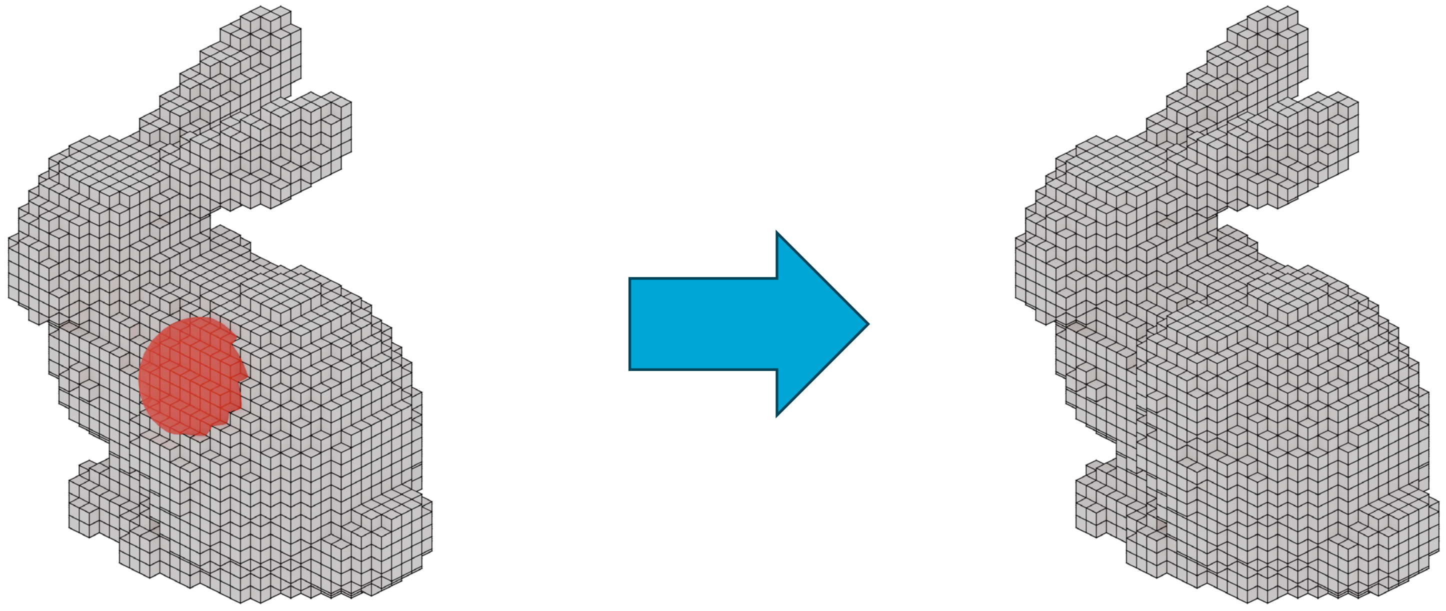
HashDAG *[CBE20]* – Limitations

- Limited scalability due to mutex locking
- CPU → GPU copy of scattered data
- Designed for local edits



Addressing These Limitations

- Limited scalability due to mutex locking
- CPU → GPU copy of scattered data
- Designed for local edits



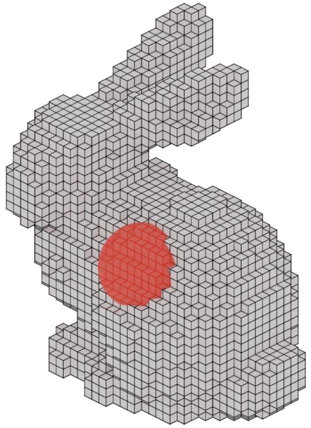
TU Delft

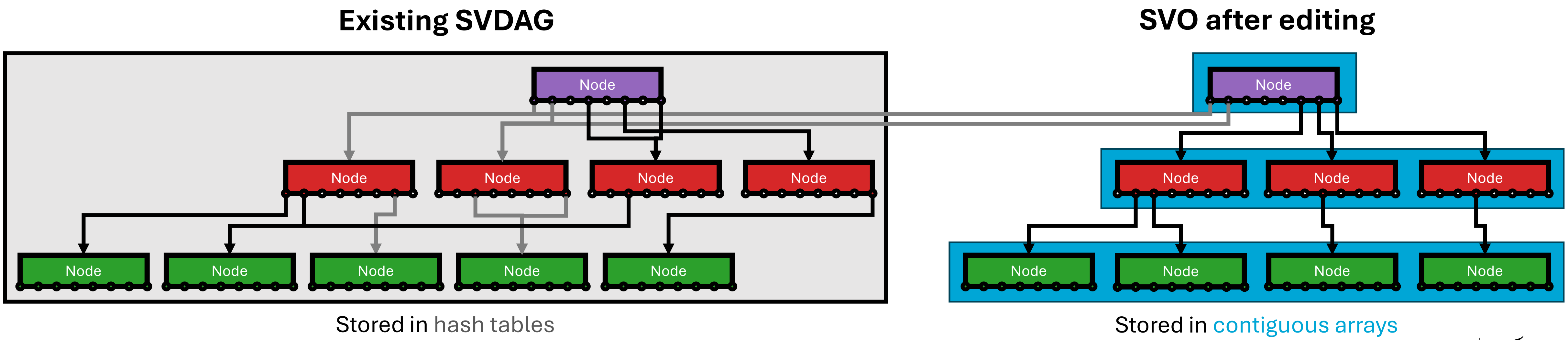
Our Work

Editing on the GPU

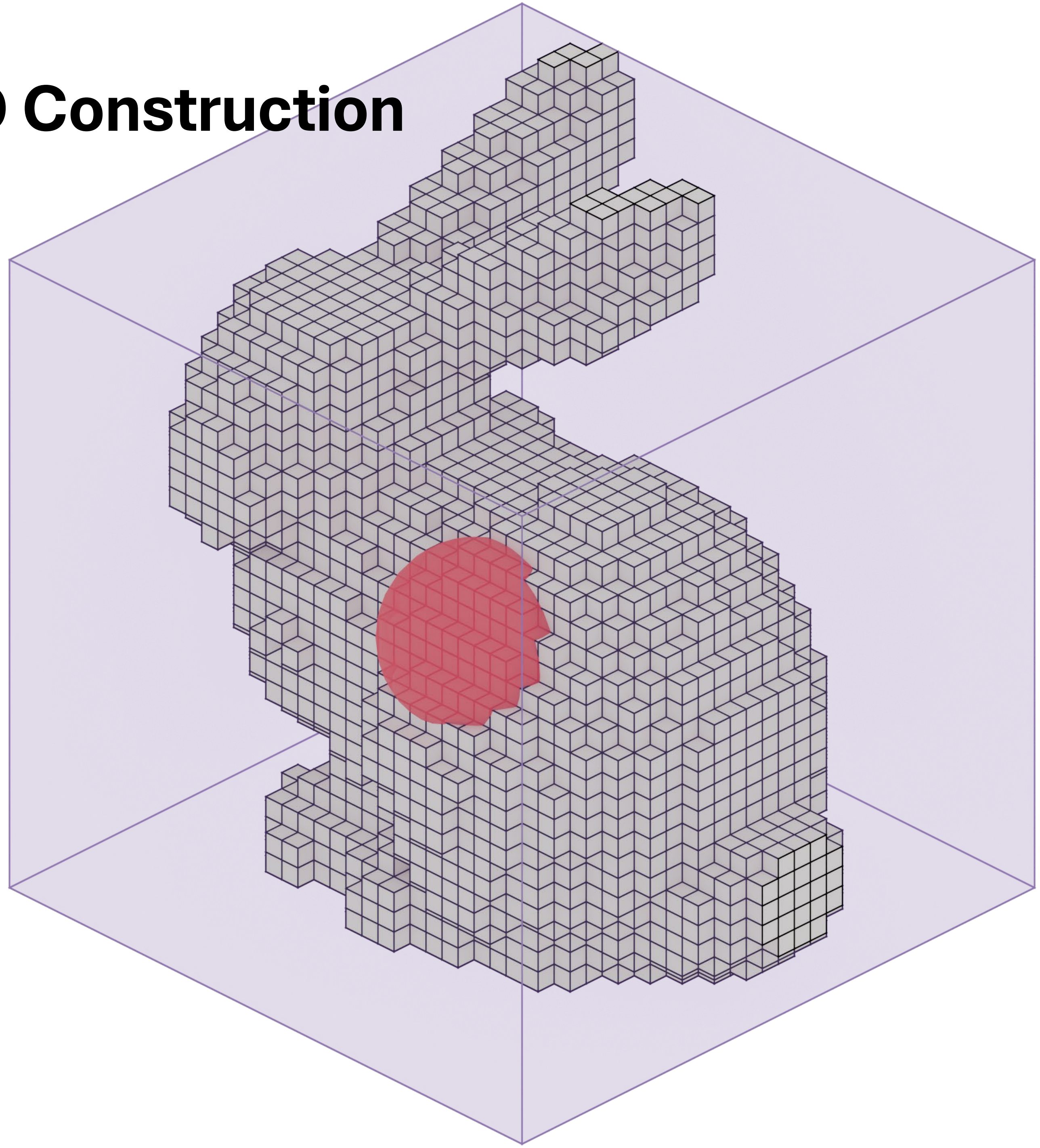
Conceptual Overview

1. Construct a Sparse Voxel "*Octree*" of the modified scene
   - Point to existing SVDAG for unchanged regions

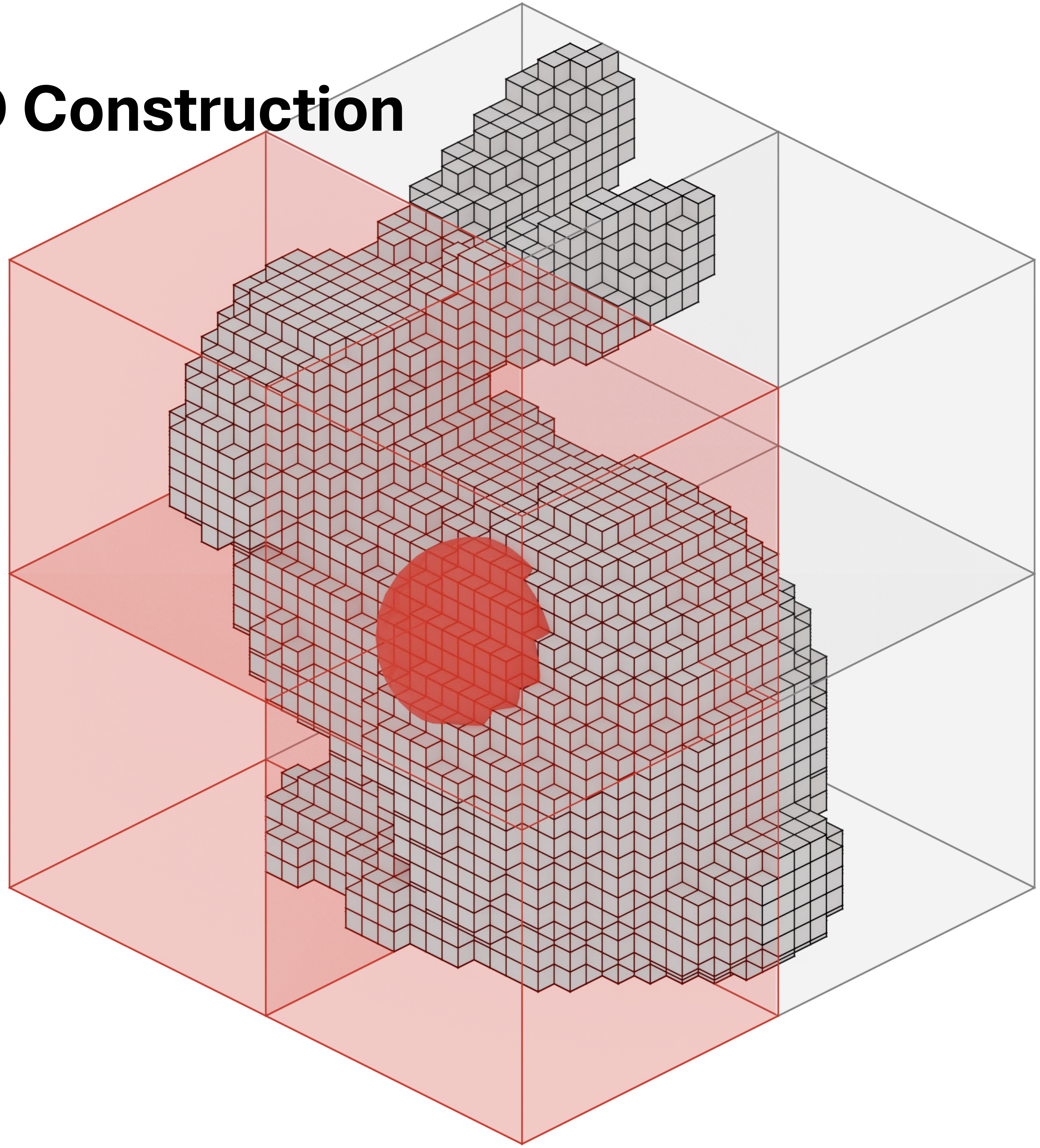




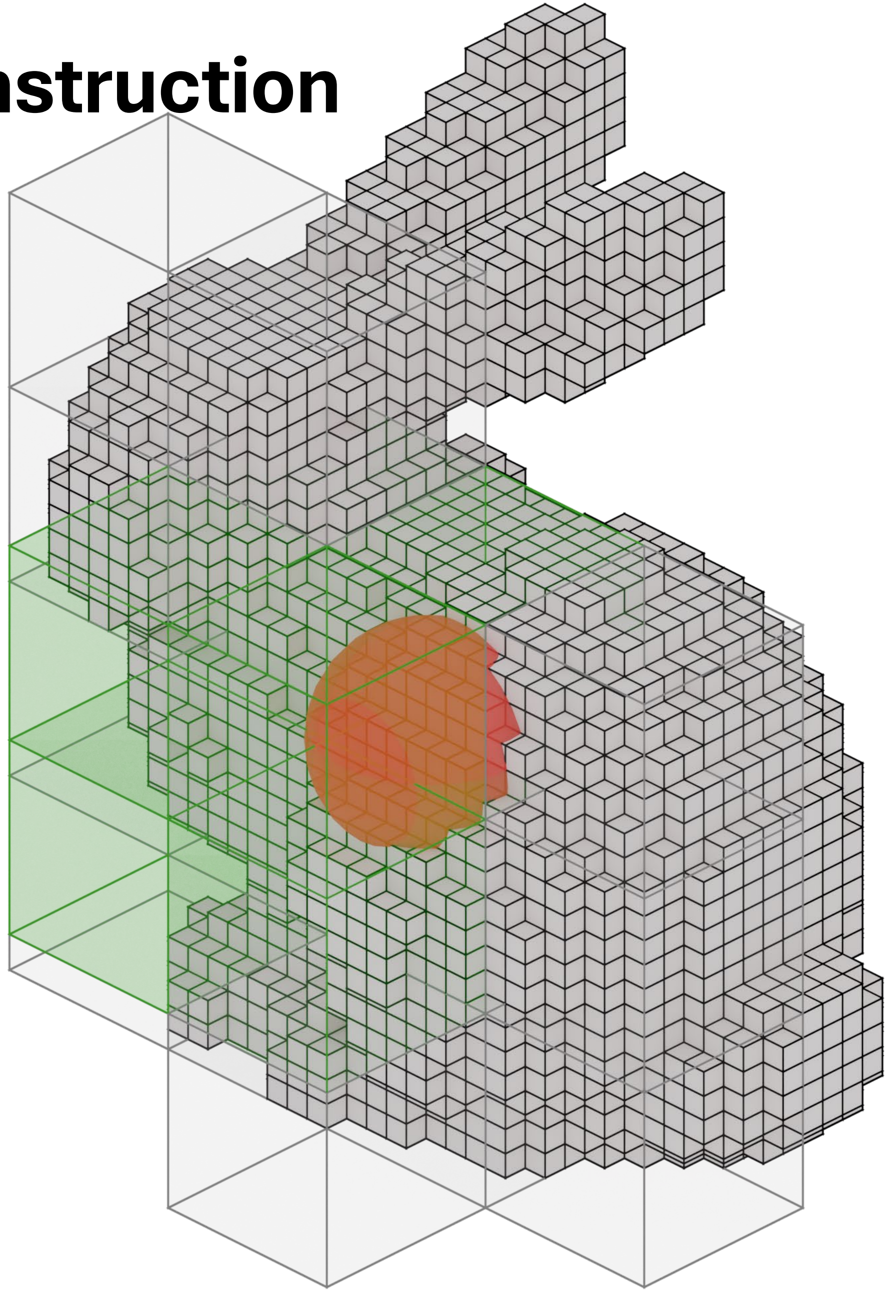
Top-Down SVO Construction



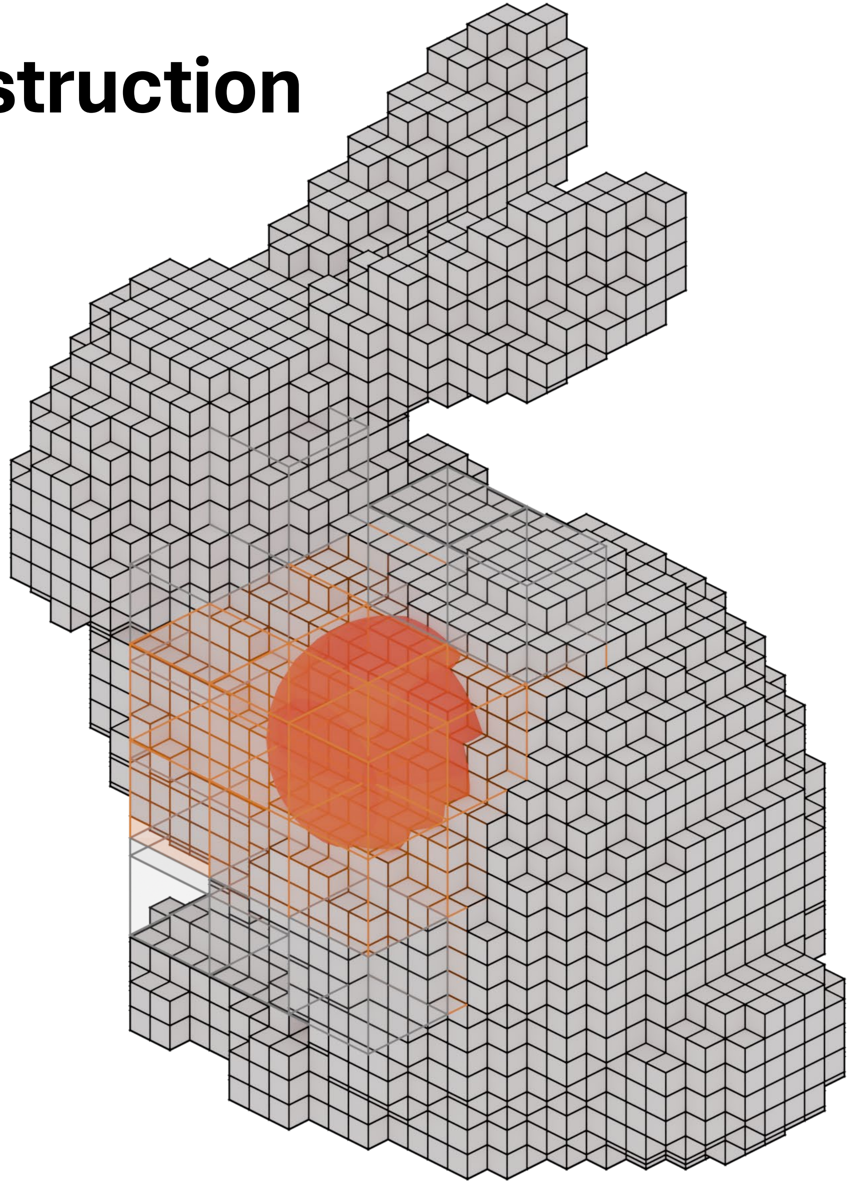
Top-Down SVO Construction



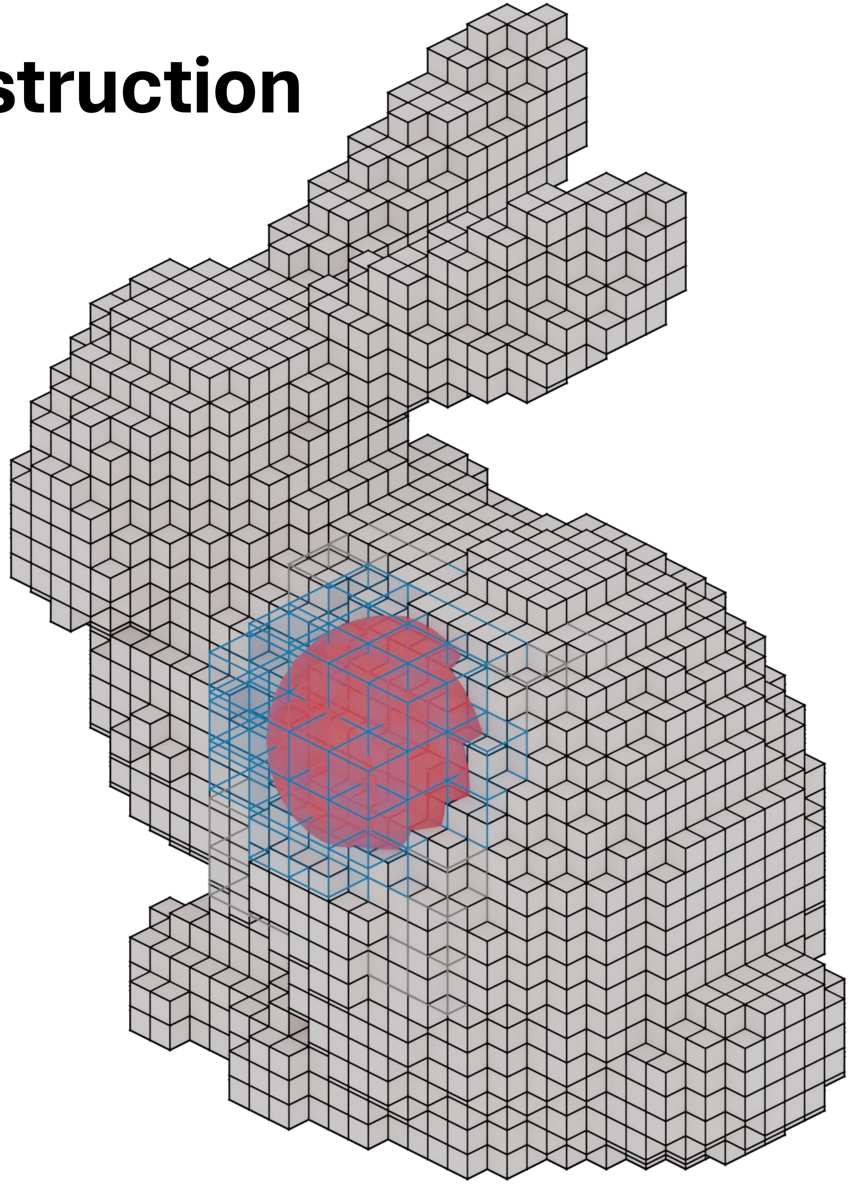
Top-Down SVO Construction



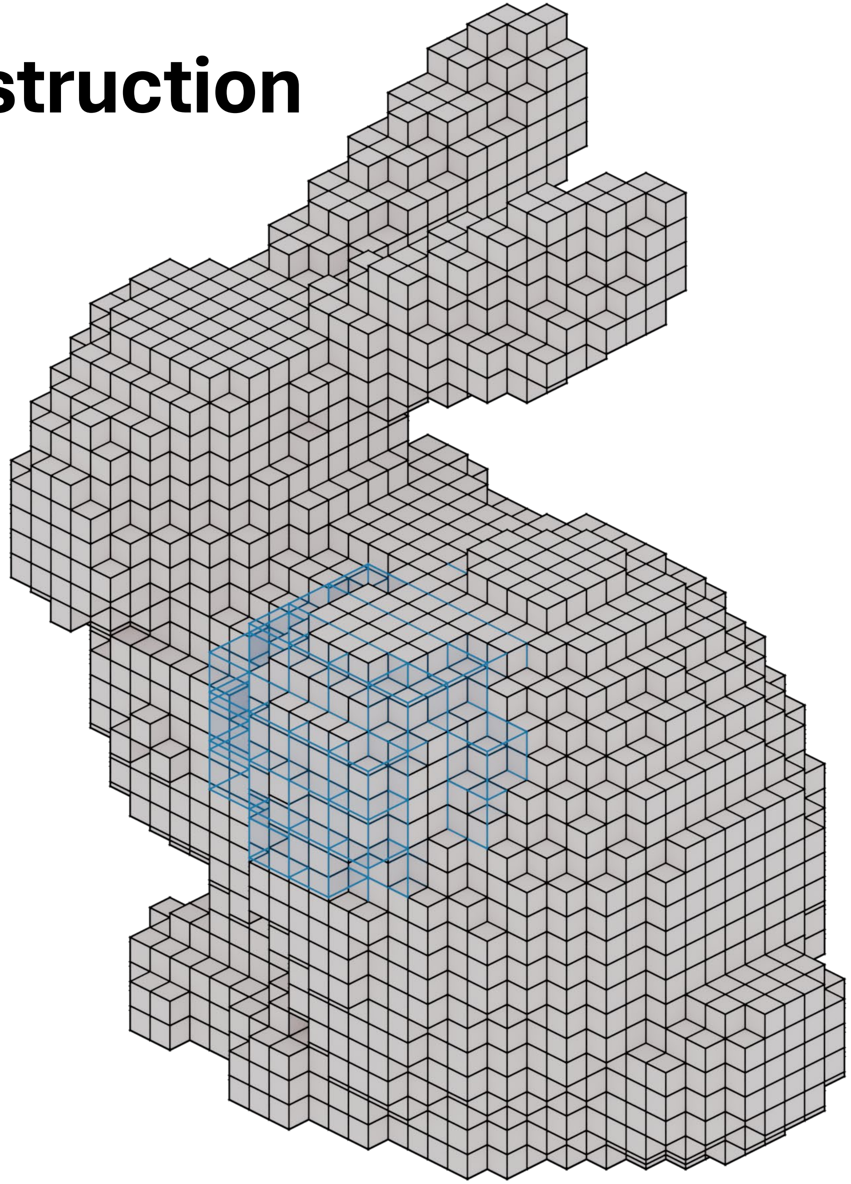
Top-Down SVO Construction



Top-Down SVO Construction

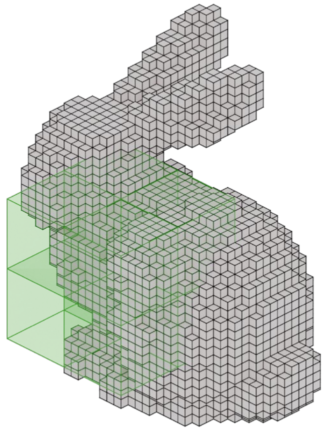


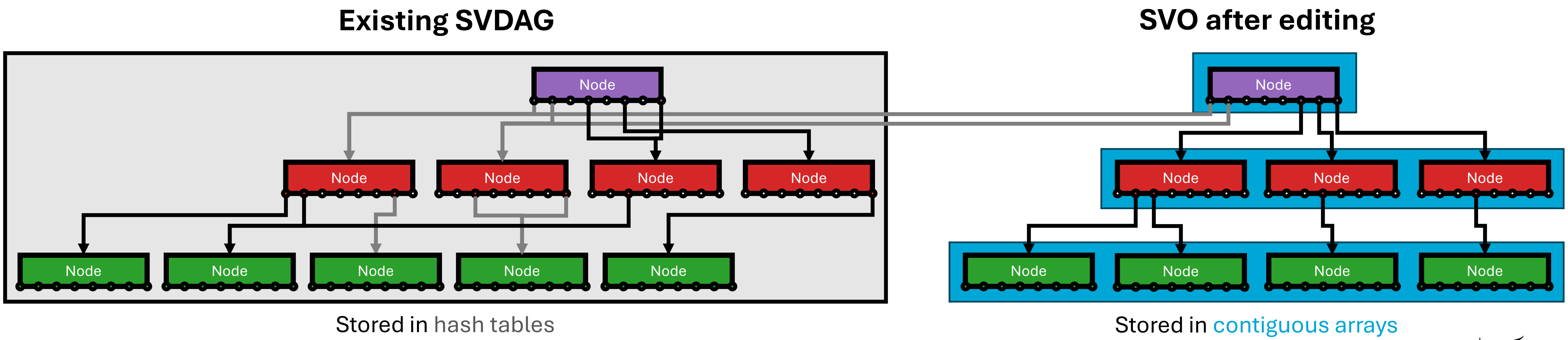
Top-Down SVO Construction



Conceptual Overview

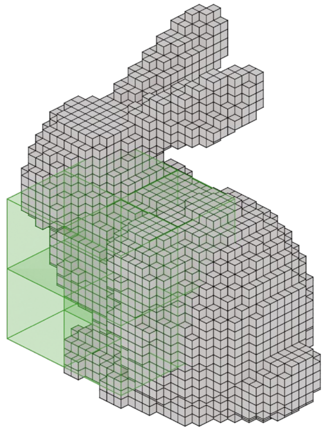
1. Construct a Sparse Voxel "*Octree*" of the modified region
2. Remove duplicates within this *Octree*
   - Use hash table to find duplicates



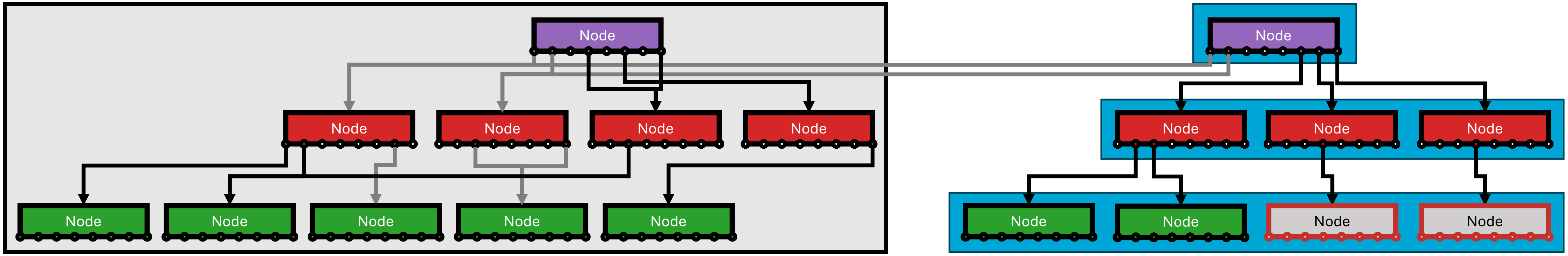


Conceptual Overview

1. Construct a Sparse Voxel "*Octree*" of the modified region
2. Remove duplicates within this *Octree*
    - Use hash table to find duplicates



Existing SVDAG **SVO after editing**

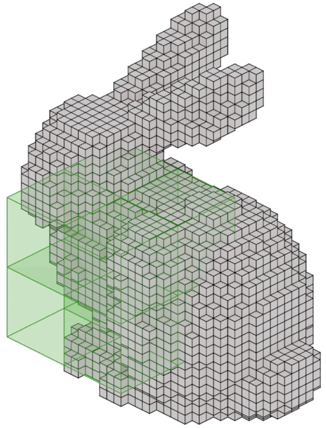


Stored in hash tables

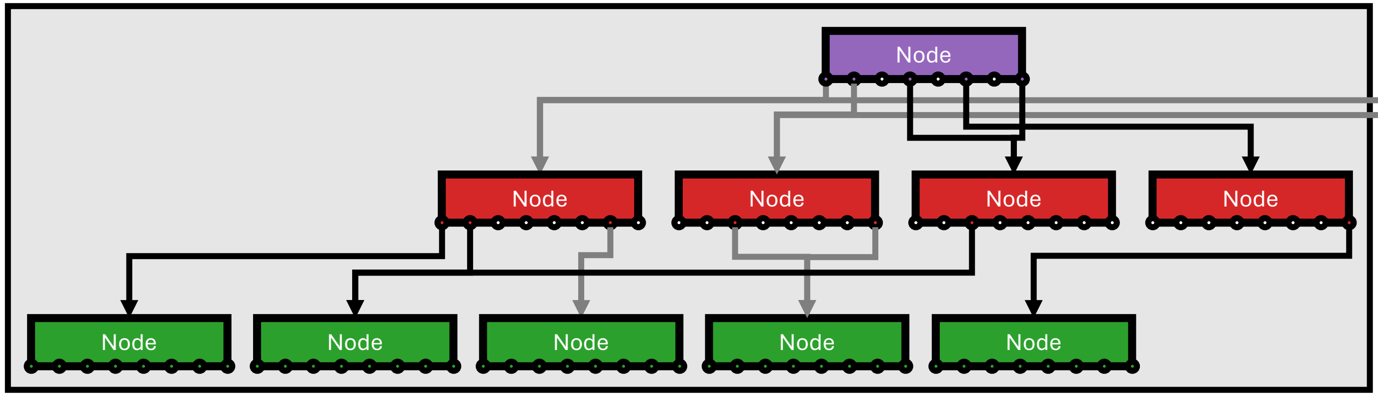
Stored in contiguous arrays

Conceptual Overview

1. Construct a Sparse Voxel "*Octree*" of the modified region
2. Remove duplicates within this *Octree*
   - Use hash table to find duplicates

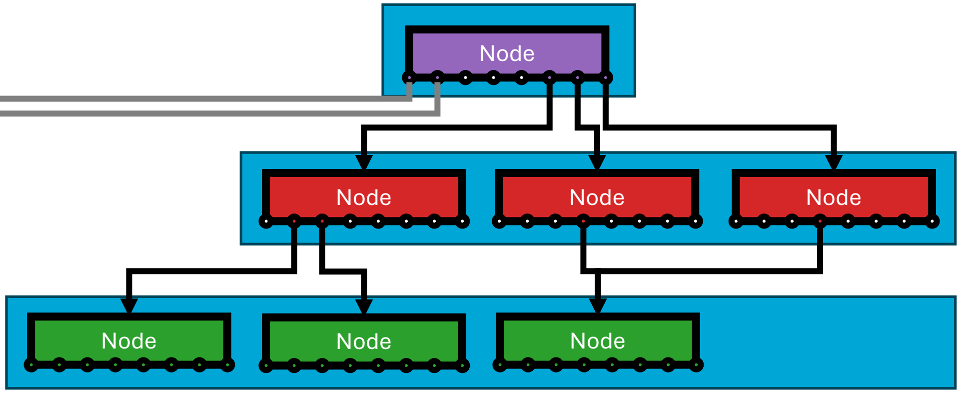


Existing SVDAG



Stored in hash tables

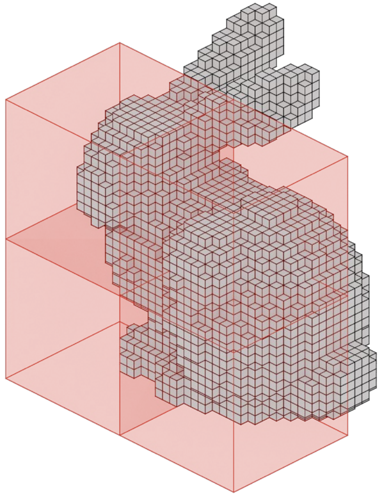
SVO after editing

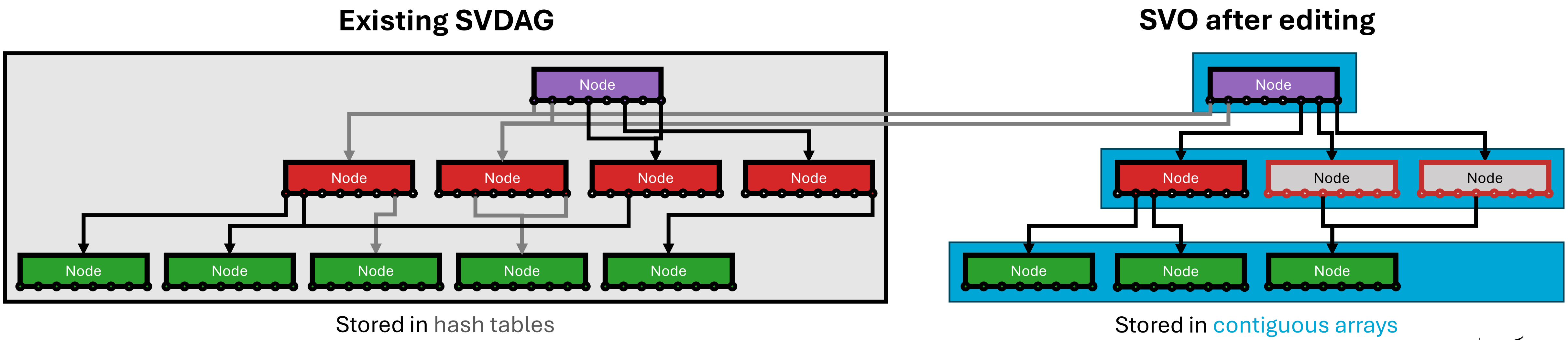


Stored in contiguous arrays

Conceptual Overview

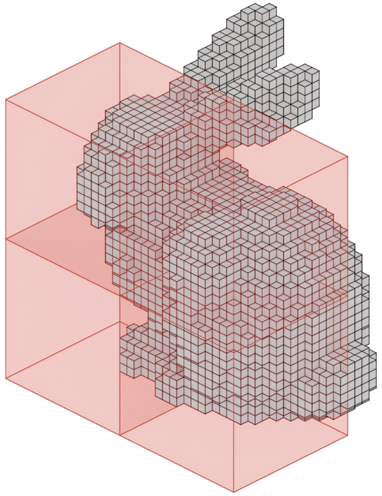
1. Construct a Sparse Voxel "*Octree*" of the modified region
2. Remove duplicates within this *Octree*
   - Use hash table to find duplicates

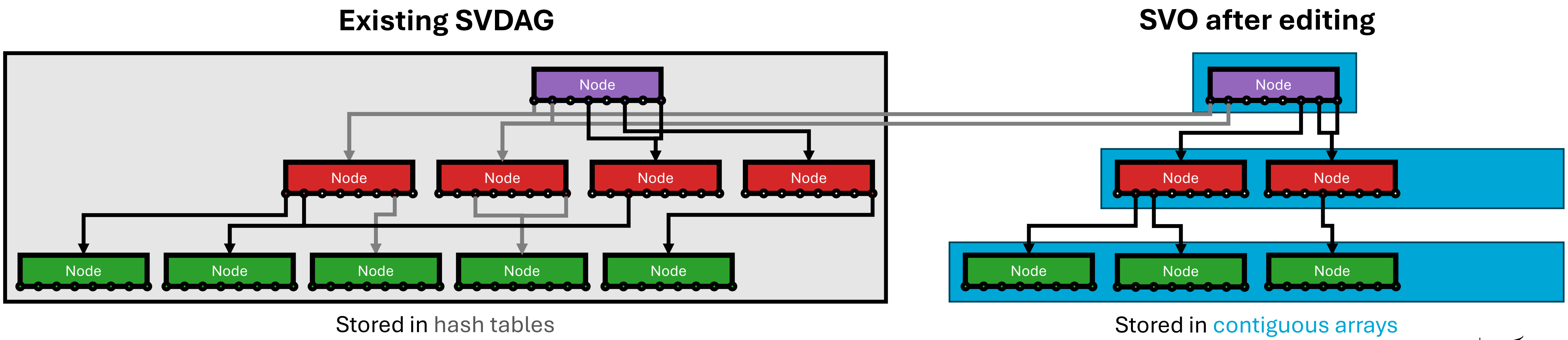




Conceptual Overview

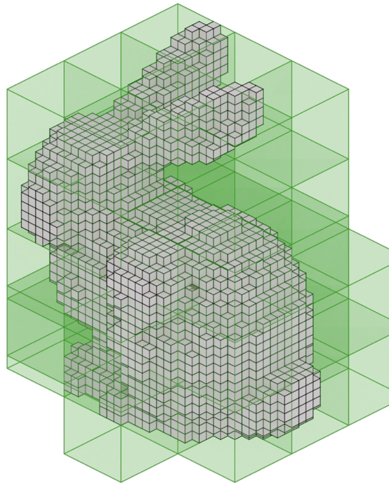
1. Construct a Sparse Voxel "*Octree*" of the modified region
2. Remove duplicates within this *Octree*
   - Use hash table to find duplicates

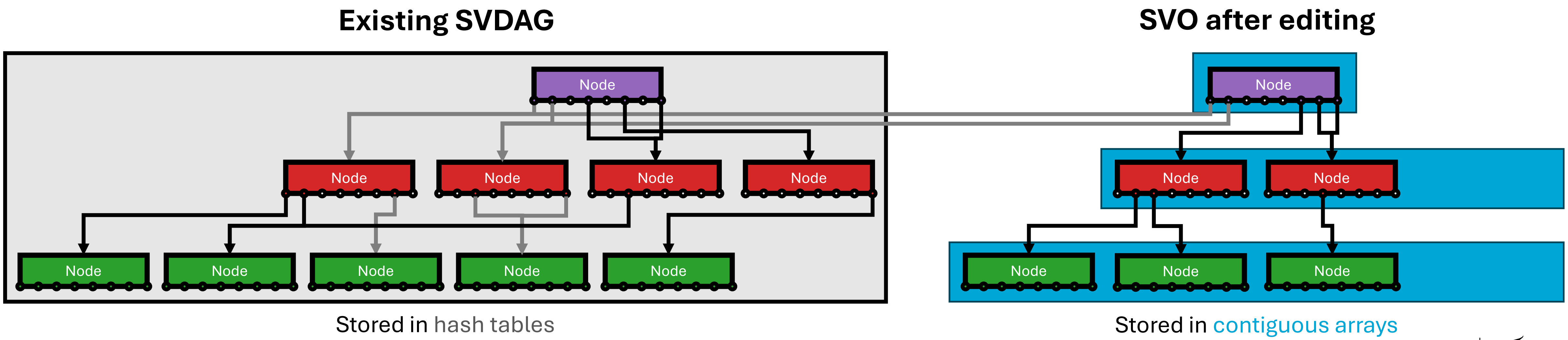




Conceptual Overview

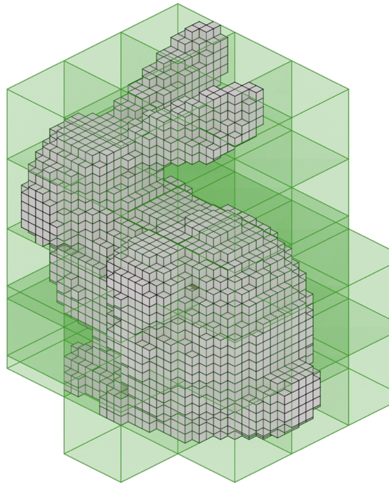
1. Construct a Sparse Voxel "*Octree*" of the modified region
2. Remove duplicates within this *Octree*
3. Merge into existing SVDAG

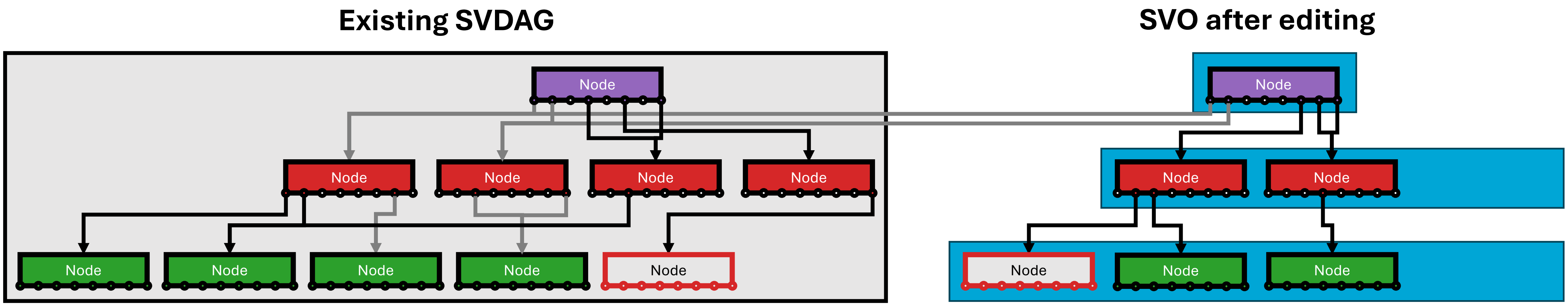




Conceptual Overview

1. Construct a Sparse Voxel "*Octree*" of the modified region
2. Remove duplicates within this *Octree*
3. Merge into existing SVDAG



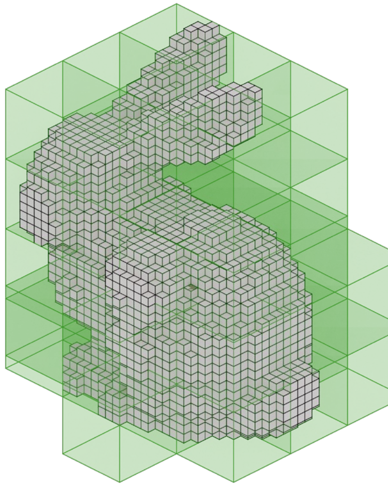


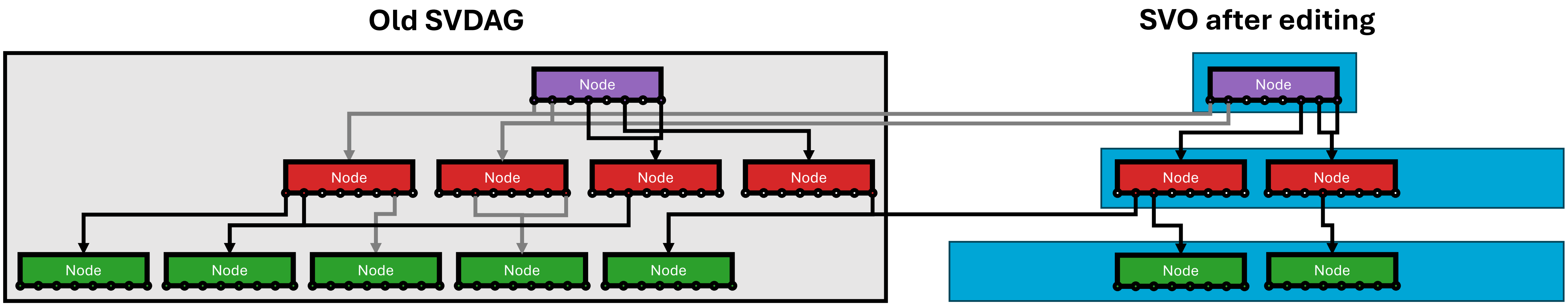
Stored in hash tables

Stored in contiguous arrays

Conceptual Overview

1. Construct a Sparse Voxel "*Octree*" of the modified region
2. Remove duplicates within this *Octree*
3. Merge into existing SVDAG



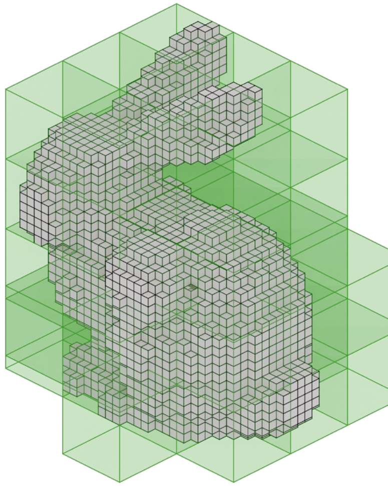


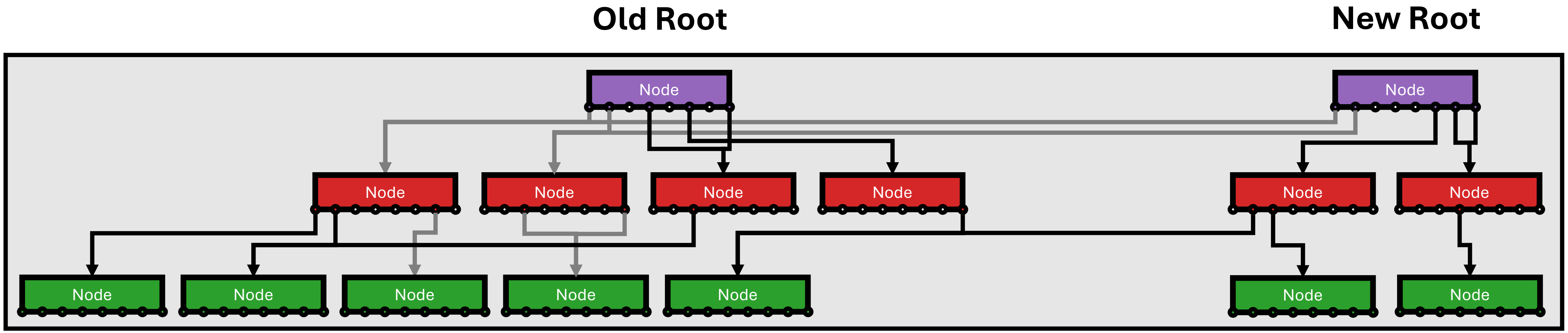
Stored in hash tables

Stored in contiguous arrays

Conceptual Overview

1. Construct a Sparse Voxel "*Octree*" of the modified region
2. Remove duplicates within this *Octree*
3. Merge into existing SVDAG





Stored in hash tables

GPU Hash Table

SlabHash *[AFCO18]*

Matches most of our requirements:

- Optimized for the GPU
- Pointer stability under insertion
- ~~Must support “large” items (>64 bits)~~

2018 IEEE International Parallel and Distributed Processing Symposium

A Dynamic Hash Table for the GPU

Saman Ashkiani
Electrical and Computer Engineering
University of California, Davis
sashkiani@ucdavis.edu

Martin Farach-Colton
Computer Science
Rutgers University
farach@cs.rutgers.edu

John D. Owens
Electrical and Computer Engineering
University of California, Davis
jowens@ece.ucdavis.edu

***Abstract*—We design and implement a fully concurrent dynamic hash table for GPUs with comparable performance to the state of the art static hash tables. We propose a warp-cooperative work sharing strategy that reduces branch divergence and provides an efficient alternative to the traditional way of per-thread (or per-warp) work assignment and processing. By using this strategy, we build a dynamic non-blocking concurrent linked list, the *slab list*, that supports asynchronous, concurrent updates (insertions and deletions) as well as search queries. We use the slab list to implement a dynamic hash table with chaining (the *slab hash*). On an NVIDIA Tesla K40c GPU, the slab hash performs updates with up to 512 M updates/s and processes search queries with up to 937 M queries/s. We also design a warp-synchronous dynamic memory allocator, *SlabAlloc*, that suits the high performance needs of the slab hash. SlabAlloc dynamically allocates memory at a rate of 600 M allocations/s, which is up to 37x faster than alternative methods in similar scenarios.**

I. Introduction

A key deficiency of the GPU ecosystem is its lack of *dynamic* data structures, which allow incremental updates (such as insertions and deletions). Instead, GPU data structures (e.g., cuckoo hash tables [1]) typically address incremental changes to a data structure by rebuilding the entire data structure from scratch. A few GPU data structures (e.g., the dynamic graph data structure in cuSTINGER [2]) implement phased updates, where updates occur in a different execution phase than lookups. In this work we describe the design and implementation of a hash table for GPUs that supports truly concurrent insertions and deletions that can execute together with lookups.

Supporting high-performance concurrent updates of data structures on GPUs represents a significant design challenge. Modern GPUs support tens of thousands of simultaneous resident threads, so traditional lock-based methods that enforce concurrency will suffer from substantial contention and will thus likely be inefficient. Non-blocking approaches offer more potential for such massively parallel frameworks, but most of the multi-core system literature (e.g., classic non-blocking linked lists [3]) neglects the sensitivity of GPUs to memory access patterns and branch divergence, which makes it inefficient to directly translate those ideas to the GPU.

In this paper, we present a new GPU hash table, the *slab hash*, that supports bulk and incremental builds. One might expect that supporting incremental insertions and deletions would result in significantly reduced query performance compared to static data structures. However, our hash table not only supports updates with high performance but also sustains build and query performance on par with static GPU hash tables. Our hash table is based on a novel linked list data structure, the *slab list*. Previous GPU implementations of linked lists [4], which operate on a thread granularity and contain a data element and pointer per linked list node, exhibit poor performance because they suffer from control and memory divergence and incur significant space overhead. The slab list instead operates on a warp granularity, with a width equal to the SIMD width of the underlying machine and contains many data elements per linked list node. Its design minimizes control and memory divergence and uses space efficiently. We then construct the slab hash from this slab list as its building block, with one slab list per hash bucket. Our contributions in this work are as follows:

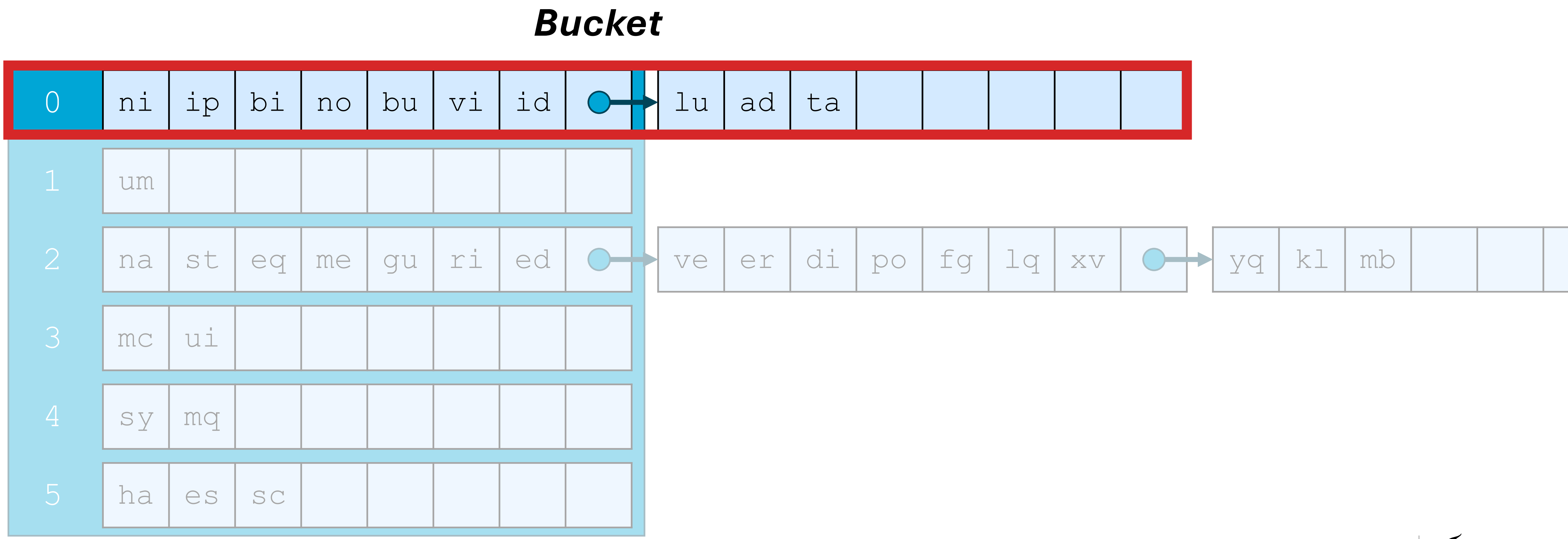
- The slab list is based on a node structure that closely matches the GPU’s hardware characteristics.
- The slab list implementation leverages a novel warp-cooperative work sharing strategy that minimizes branch divergence, using warp-synchronous programming and warp-wide communications.
- The slab hash, based on the slab list, supports concurrent operations with high performance.
- To allow concurrent updates, we design and implement a novel memory allocator that dynamically and efficiently allocates and deallocates memory in a way that is well-matched to our underlying warp-cooperative implementation.
- Our memory allocator is scalable, allowing us to support data structures up to 1 TB (far larger than the memory size of current GPUs) and without any CPU intervention.
- The slab hash’s bulk-build and search rates are comparable to those of static methods (e.g., GPU cuckoo hashing [1]), while additionally achieving efficient incremental updates.

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00052

IEEE computer society

TU Delft

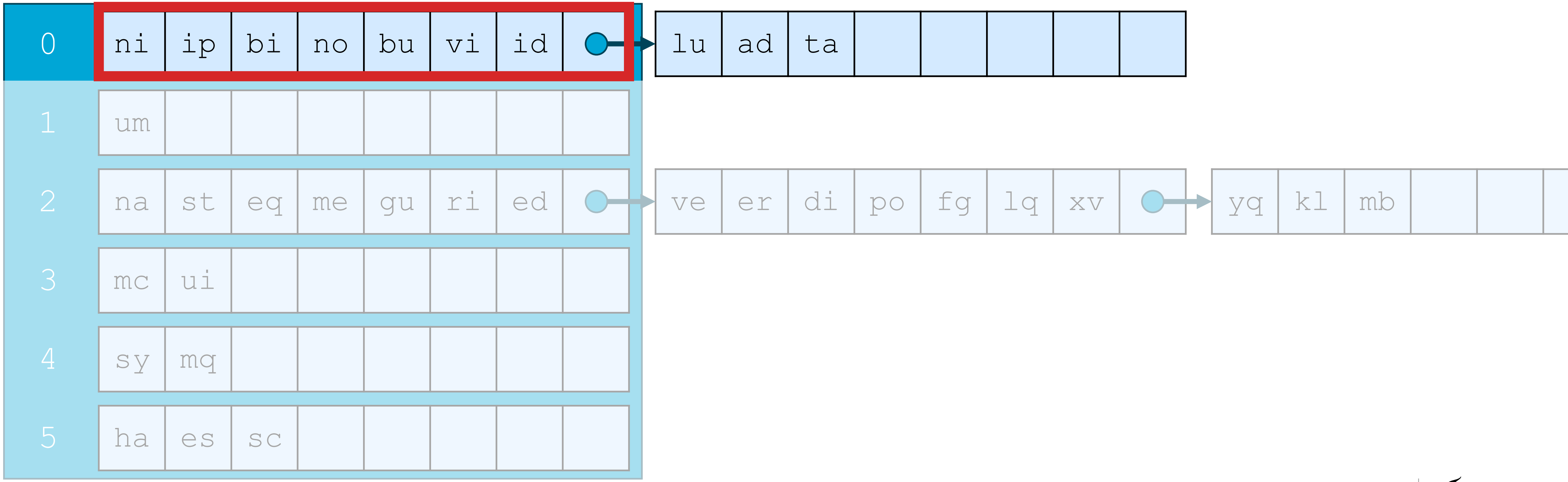
SlabHash *[AFCO18]* - Terminology



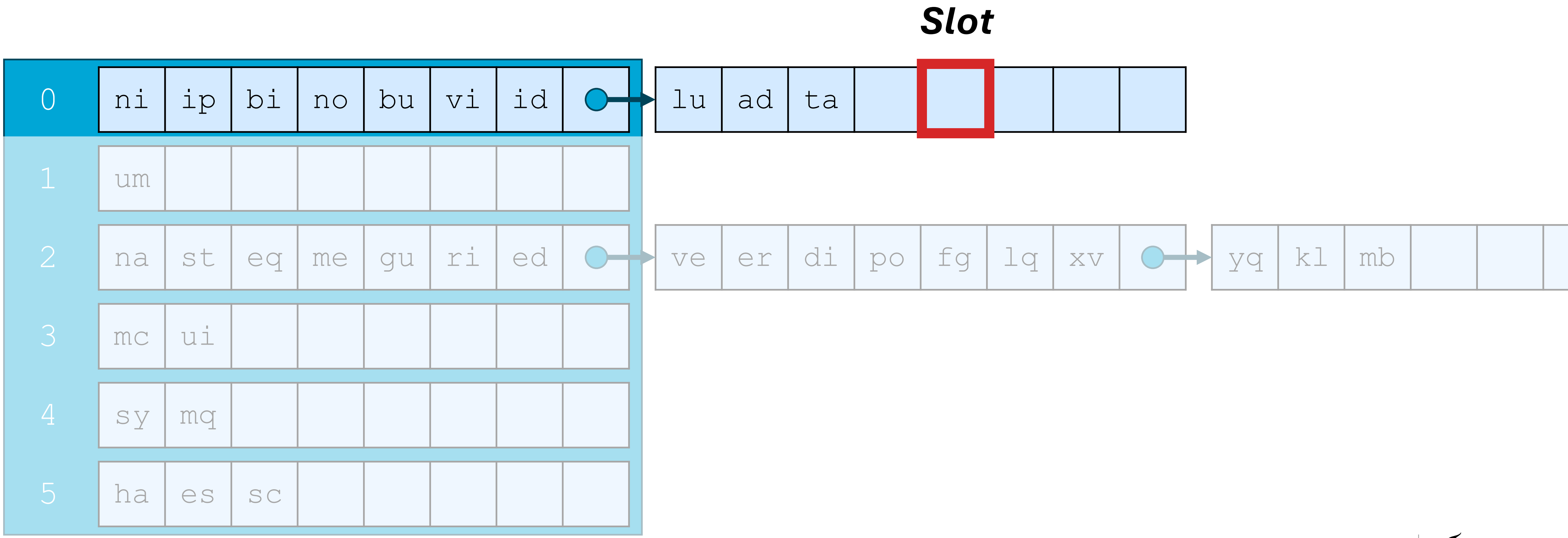
SlabHash *[AFCO18]* - Terminology

Linked list of 32-slot groups (aka ***slabs***)

Shown as slabs of 8 slots for brevity



SlabHash *[AFCO18]* - Terminology



Parallel Operations

Empty slots are indicated by a ***reserved special value*** (typically “0”).

Atomically swapped with the to-be inserted item.

Supports multiple operations in parallel:

- Insertion
- Search
- Removal

Extending SlabHash

Extending SlabHash

Extending SlabHash *[AFCO18]* to store large items (of uniform size)

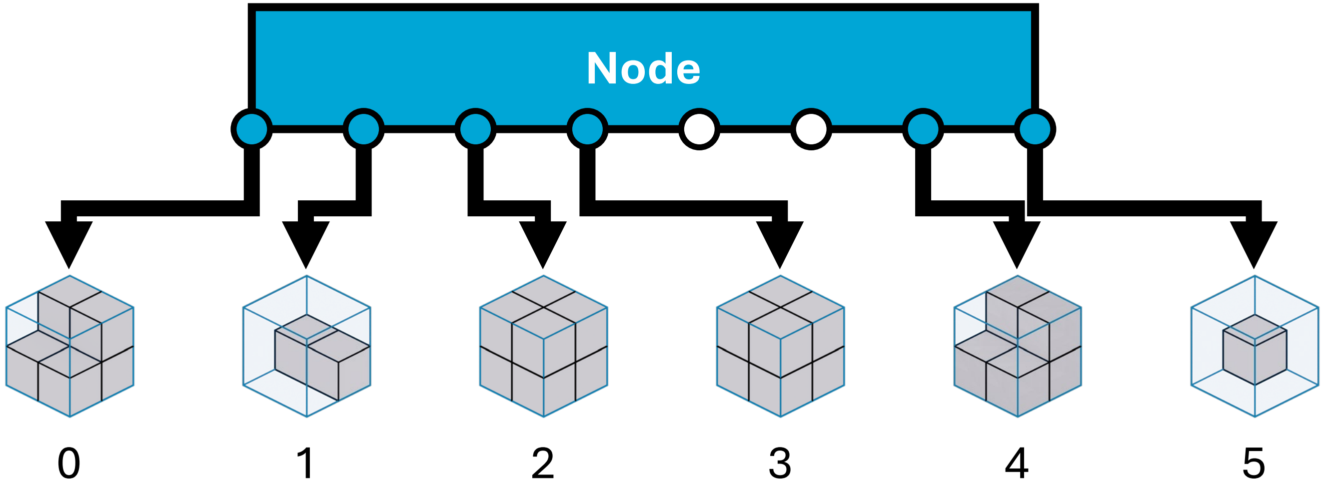
We propose three hash table designs:

1. **64-bit Atomic** Compare-and-Swap
2. **Ticket Board**
3. **Acceleration Hash** (both **8-bit** and **32-bit**)

Inner Node Encoding

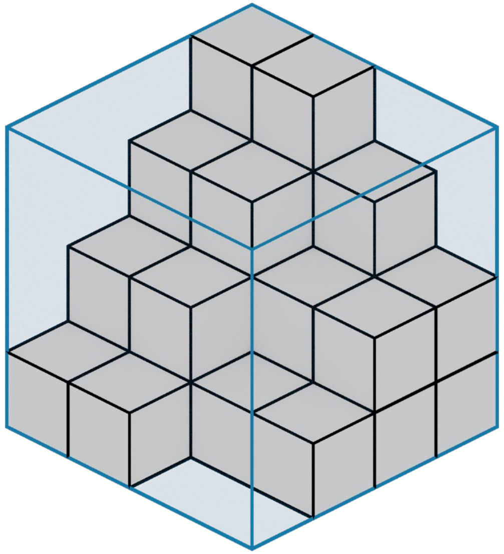
- 32-bit header (8-bit child mask)
- 32-bit pointers to non-empty children

11110011	0	1	2	3	4	5



Leaf Encoding (4^3 voxels)

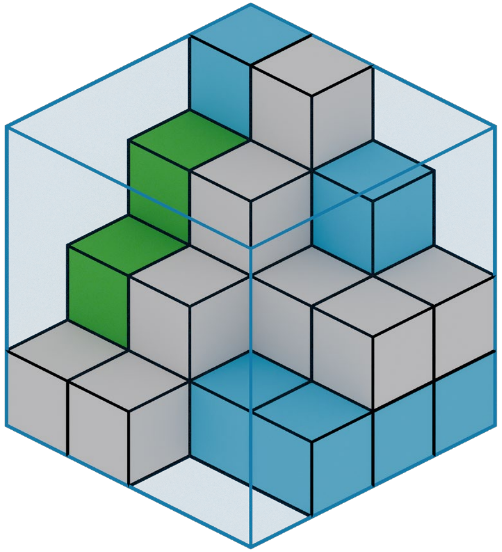
- 64-bit bitmask

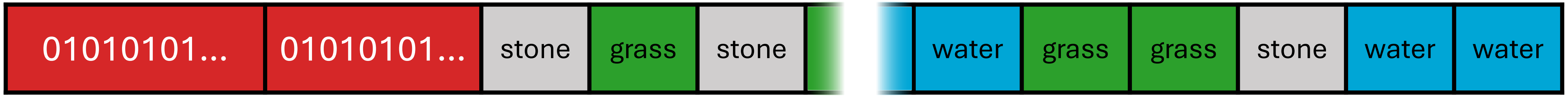


01010101…	01010101…

Leaf Encoding (4^3 voxels)

- 64-bit bitmask
- 4-bit material per occupied voxel





Hash Table Insertion and Removal

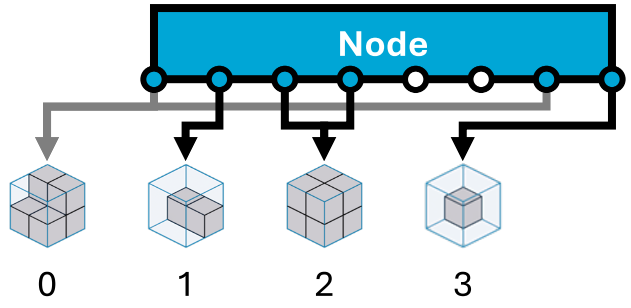
Note: items never start with 64-bit "0"

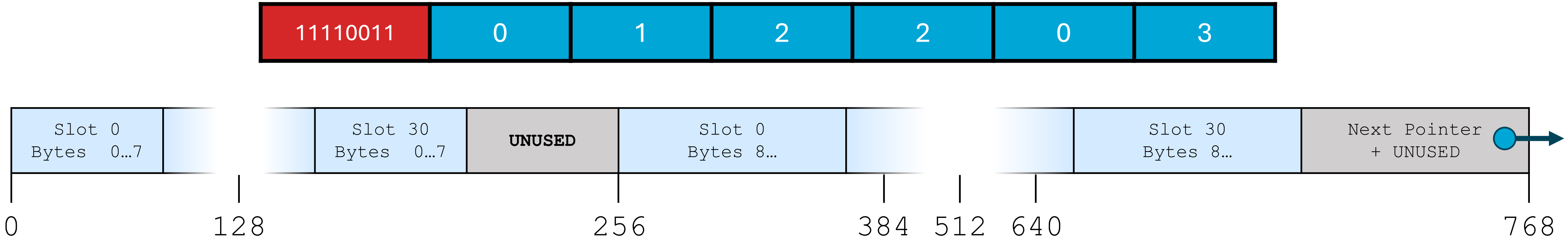
- Leaf must have at least 1 filled voxel
- Nodes must have at least 1 child

Atomically swap "0" to insert or remove

64-bit Atomic

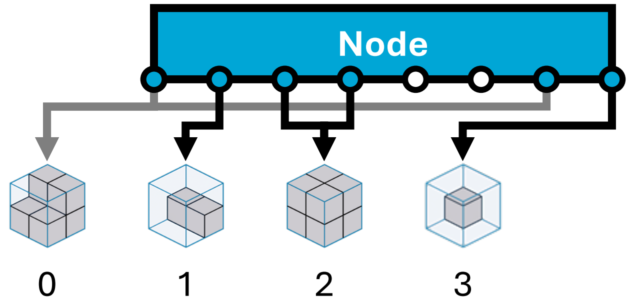
- Store first 64 bits (8 bytes) of each slot

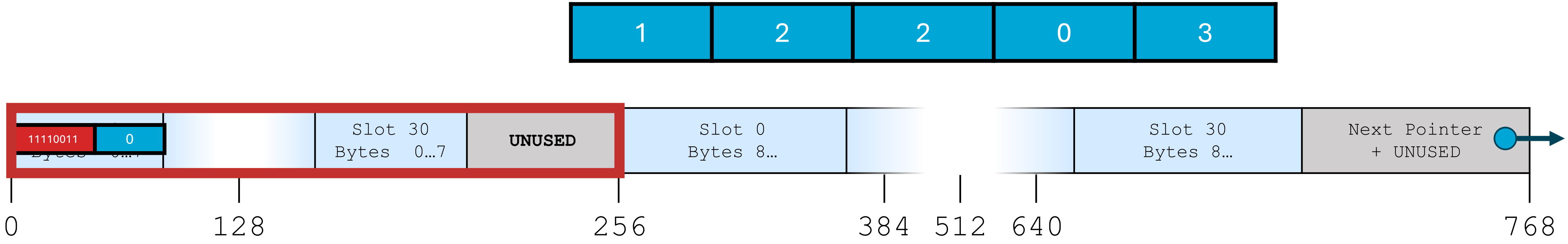




64-bit Atomic

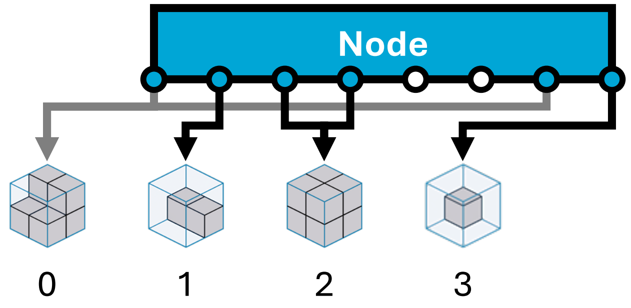
- Store first 64 bits (8 bytes) of each slot

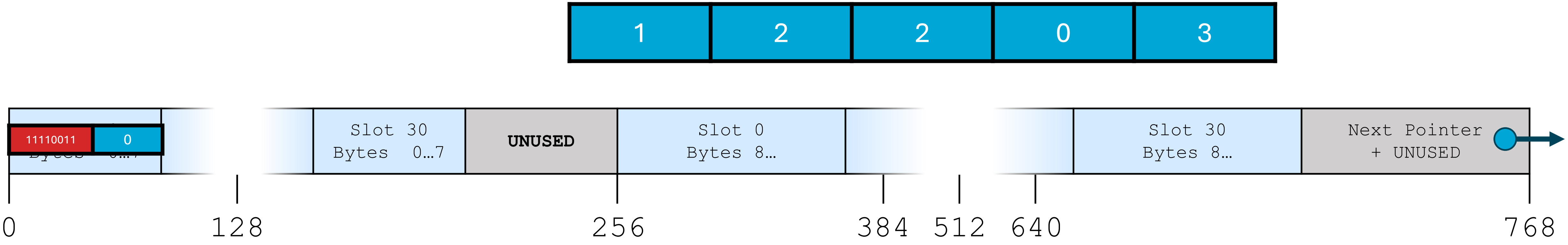




64-bit Atomic

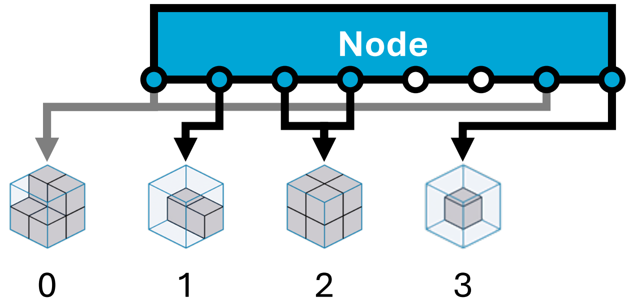
- Store first 64 bits (8 bytes) of each slot
- Store item remainders at the end of the slab

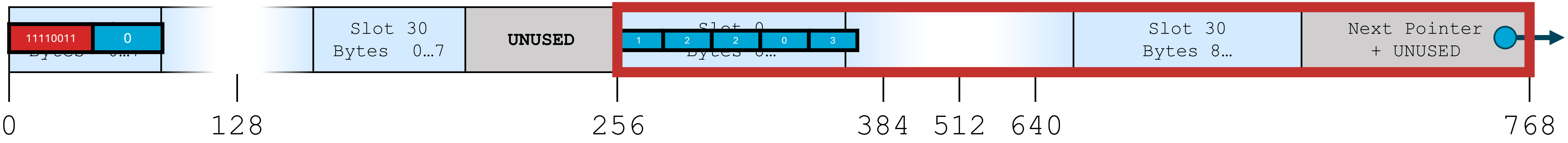




64-bit Atomic

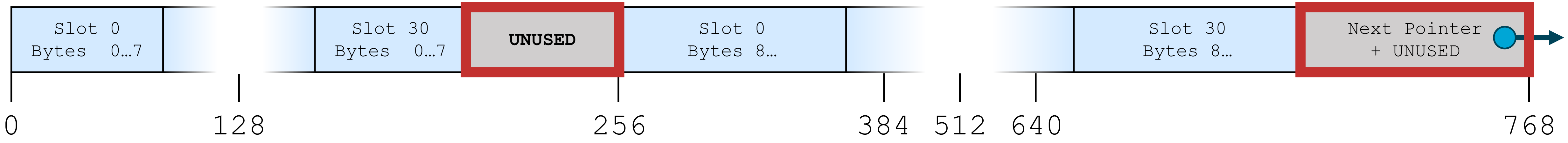
- Store first 64 bits (8 bytes) of each slot
- Store item remainders at the end of the slab





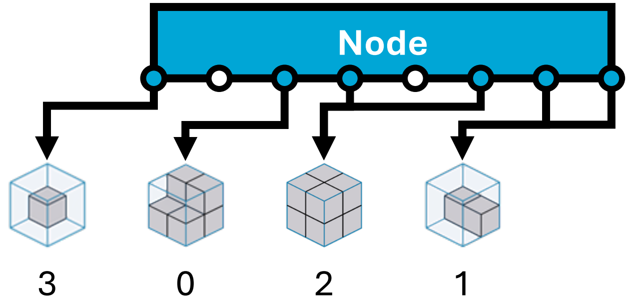
64-bit Atomic

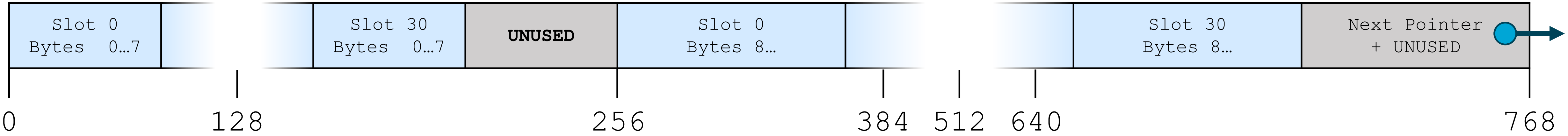
- Store first 64 bits (8 bytes) of each slot
- Store item remainders at the end of the slab
- 31 slots per slab to improve memory alignment



64-bit Atomic - Look-Up Efficiency

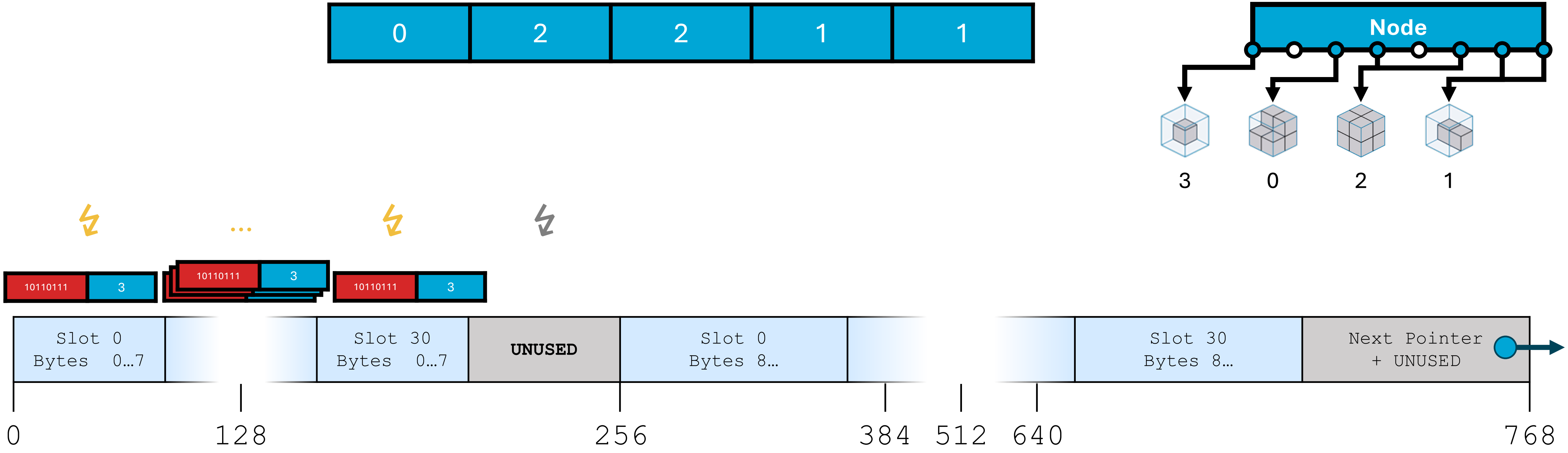
Search for the following node:





64-bit Atomic - Look-Up Efficiency

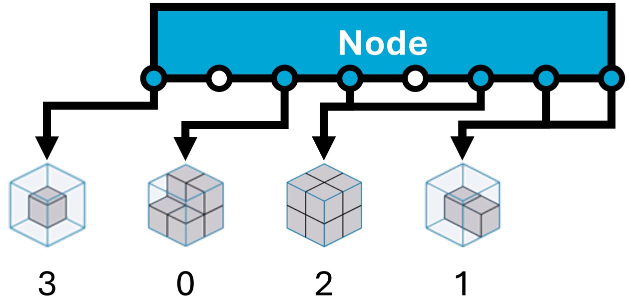
Warp of 32 threads collaboratively check first 64-bits of each item

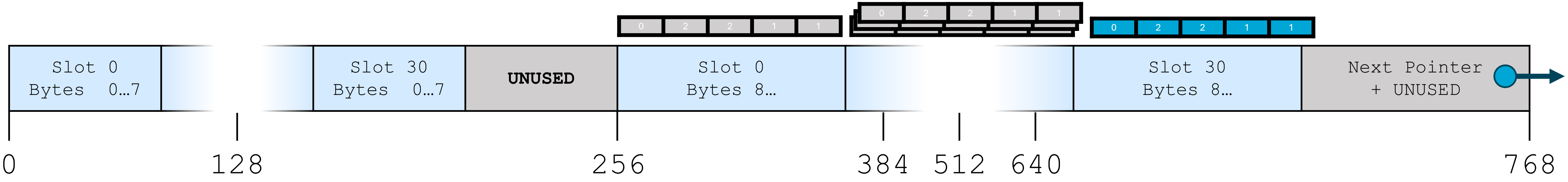


64-bit Atomic - Look-Up Efficiency

Warp of 32 ↯ threads ↯ collaboratively check first 64-bits of each item

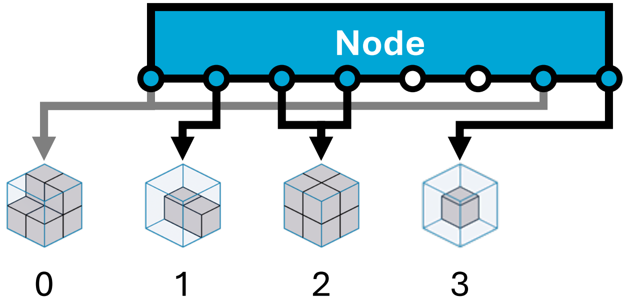
Only test ***remainder*** for potentially matching slots

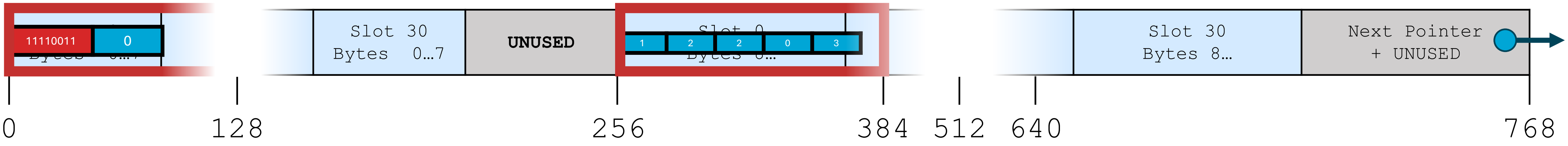




64-bit Atomic – SVDAG Traversal Performance

Visiting a node often requires loading two ***cache lines***

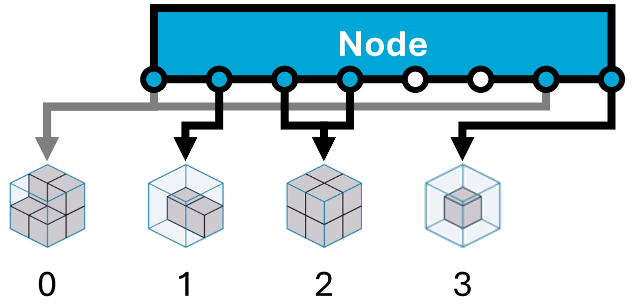


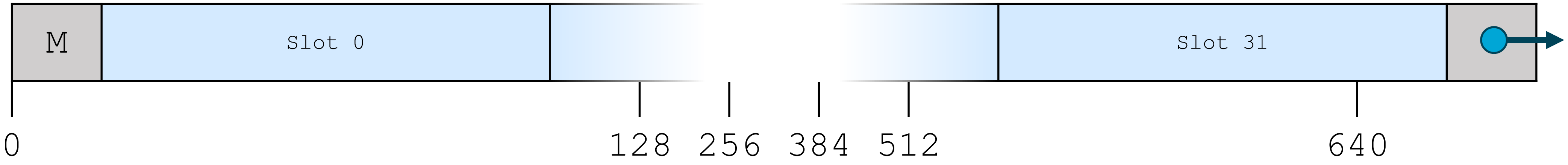


Ticket Board

Inspired by *Stadium Hashing [KBGB15]*

- 32-bit mask **M** to indicate slot usage
- No alignment padding

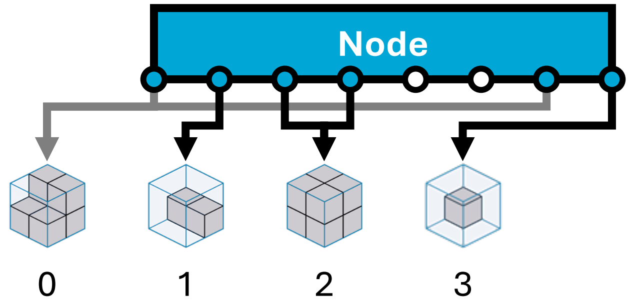


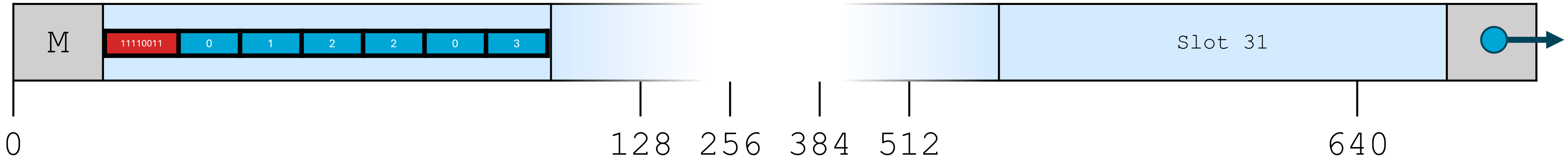


Ticket Board

Inspired by *Stadium Hashing [KBGB15]*

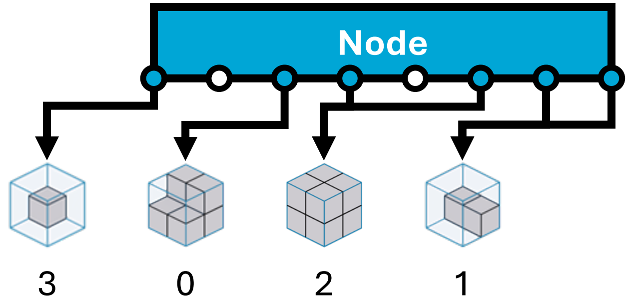
- 32-bit mask **M** to indicate slot usage
- No alignment padding

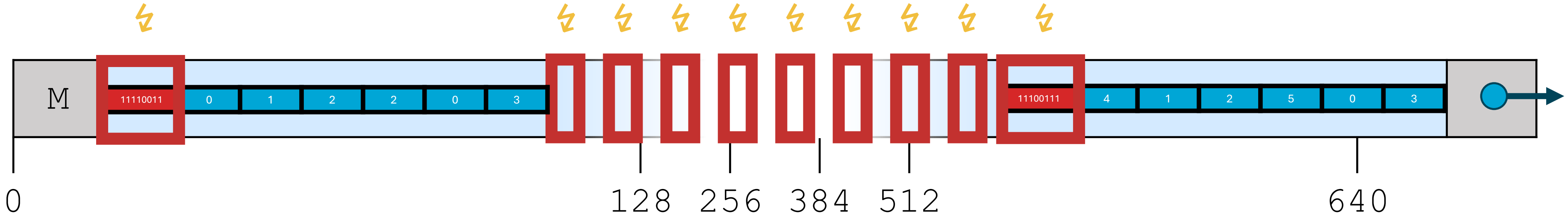




Ticket Board - Look-Up Efficiency

Look-ups require loading the entire slab from memory





TU Delft

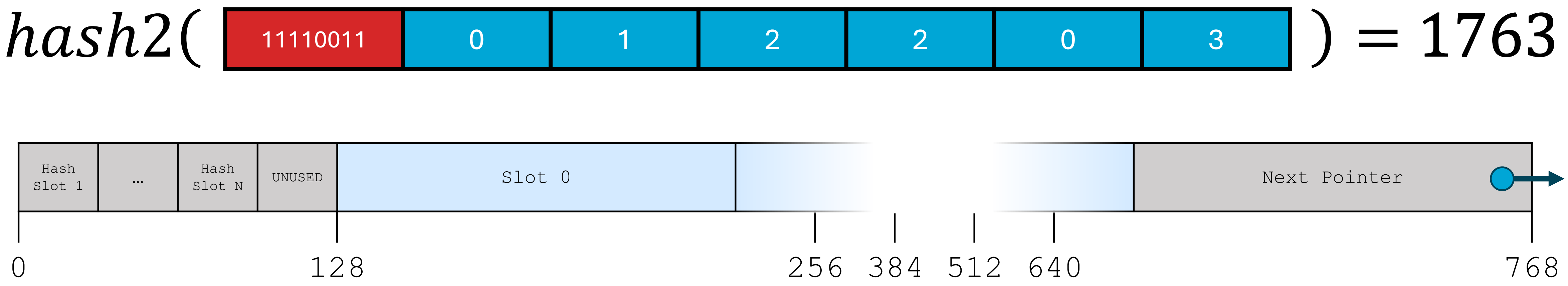
Best of Both Worlds

- Items are contiguous in memory (SVDAG traversal)
- Fast hash table look-ups
- Low memory overhead

Our solution: “acceleration hash”

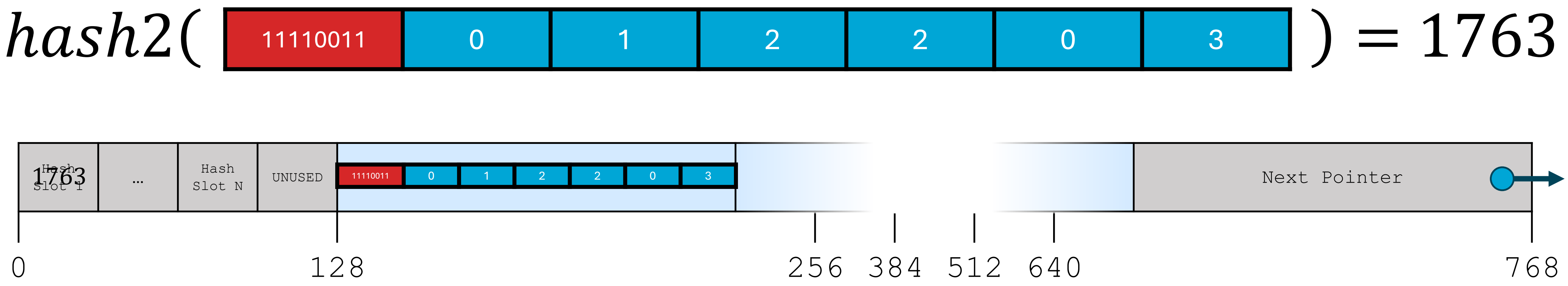
Acceleration Hash (32-bit)

- ***32-bit*** *secondary hash* of each slot
- Fast look-ups by first comparing *secondary hash*
- 31 slots per slab; aligned to cache line



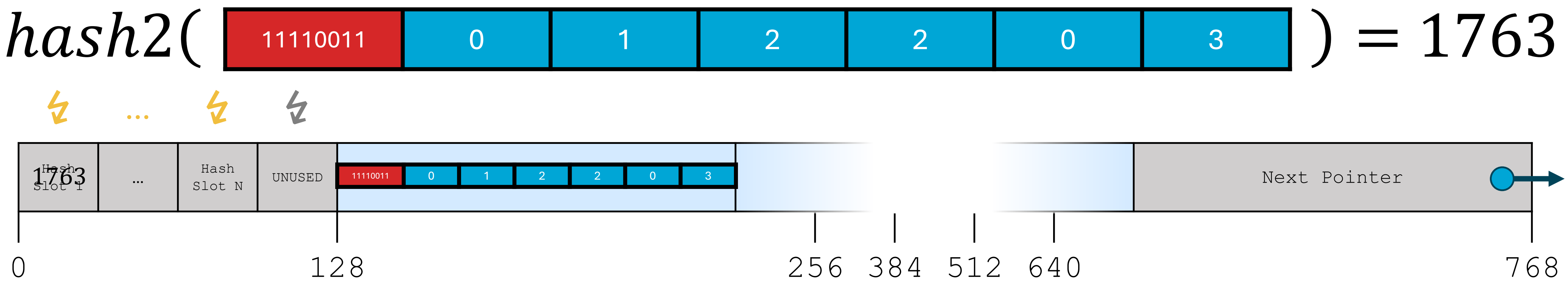
Acceleration Hash (32-bit)

- ***32-bit*** *secondary hash* of each slot
- Fast look-ups by first comparing *secondary hash*
- 31 slots per slab; aligned to cache line



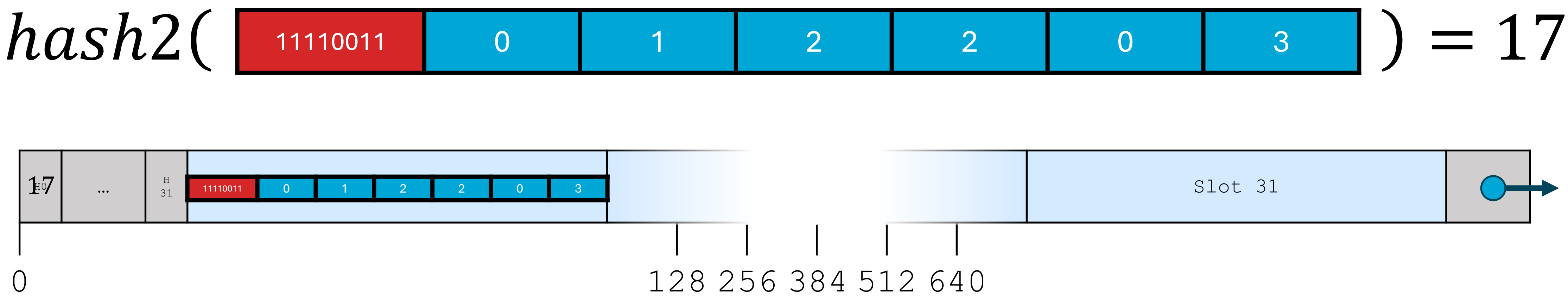
Acceleration Hash (32-bit)

- ***32-bit*** *secondary hash* of each slot
- Fast look-ups by first comparing *secondary hash*
- 31 slots per slab; aligned to cache line



Acceleration Hash (8-bit)

- ***8-bit*** *secondary hash* to ***reduce memory overhead***
- 32 slots per slab; no alignment padding



Evaluation

Evaluation

Implemented inside the HashDAG framework *[CBE20]*

Using the same test scenarios as previous work

Machine: Nvidia RTX4080, AMD 7950X3D, PopOS 22.04

Memory Usage

Memory Usage

Scene	Method	Memory
Citadel $128K^3$ (No Materials) SVO 15117 MiB SVDAG 980 MiB 	Atomic U64	1199 MiB (+22.3%)
Citadel $128K^3$ (With Materials) SVO 22516 MiB SVDAG 5997 MiB 	Atomic U64	7164 MiB (+19.5%)
San Miguel $64K^3$ (With Materials) SVO 14865 MiB SVDAG 2929 MiB 	Atomic U64	3509 MiB (+19.8%)

Memory Usage

Scene	Method	Memory
Citadel 128K³ (No Materials) SVO 15117 MiB SVDAG 980 MiB 	Atomic U64	1199 MiB (+22.3%)
	Ticket Board	1155 MiB (+17.9%)
Citadel 128K³ (With Materials) SVO 22516 MiB SVDAG 5997 MiB 	Atomic U64	7164 MiB (+19.5%)
	Ticket Board	7082 MiB (+18.1%)
San Miguel 64K³ (With Materials) SVO 14865 MiB SVDAG 2929 MiB 	Atomic U64	3509 MiB (+19.8%)
	Ticket Board	3461 MiB (+18.2%)

Memory Usage

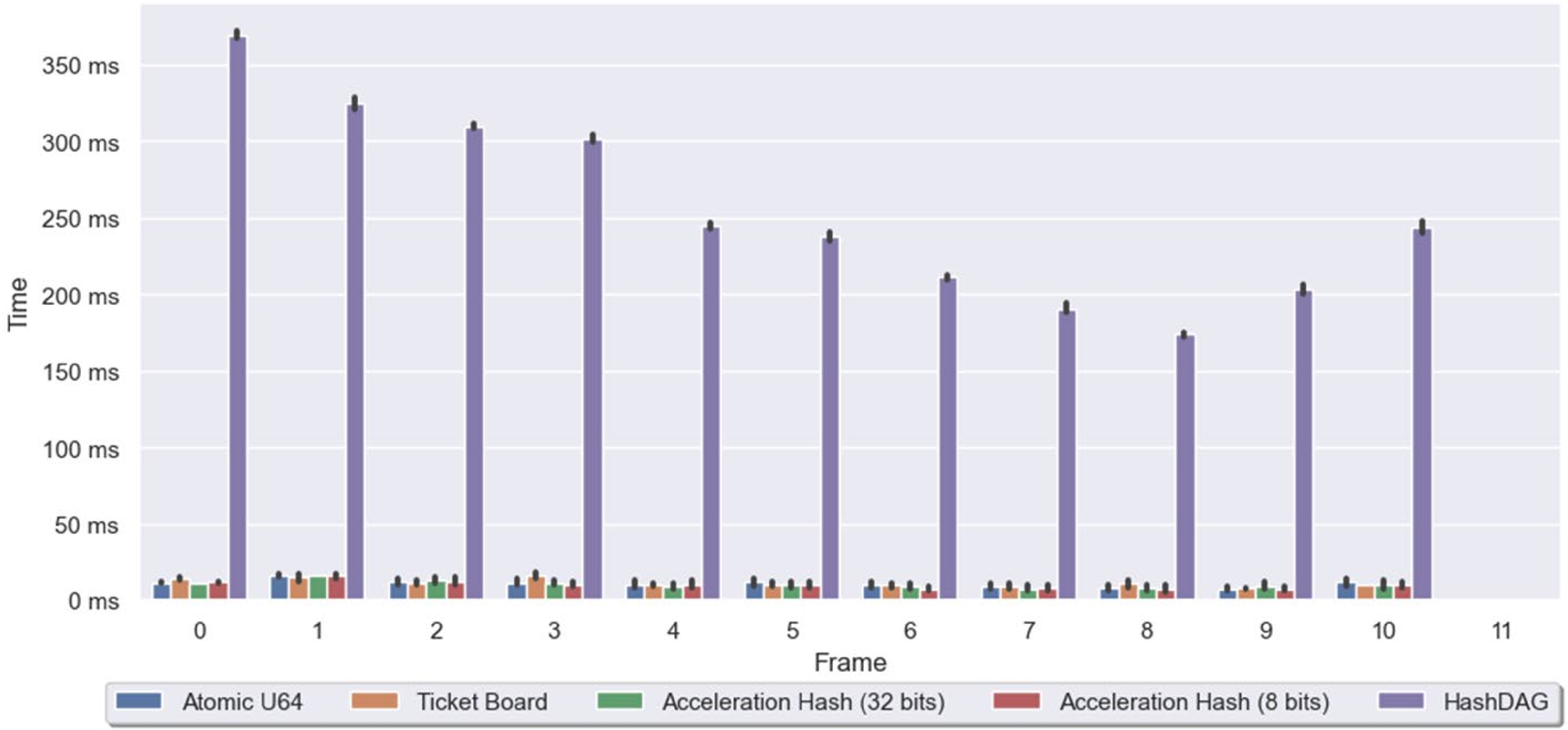
Scene	Method	Memory
Citadel 128K³ (No Materials) SVO 15117 MiB SVDAG 980 MiB 	Atomic U64	1199 MiB (+22.3%)
	Ticket Board	1155 MiB (+17.9%)
	Acceleration Hash (32-bit)	1400 MiB (+42.9%)
	Acceleration Hash (8-bit)	1203 MiB (+22.8%)
Citadel 128K³ (With Materials) SVO 22516 MiB SVDAG 5997 MiB 	Atomic U64	7164 MiB (+19.5%)
	Ticket Board	7082 MiB (+18.1%)
	Acceleration Hash (32-bit)	8685 MiB (+44.8%)
	Acceleration Hash (8-bit)	7404 MiB (+23.5%)
San Miguel 64K³ (With Materials) SVO 14865 MiB SVDAG 2929 MiB 	Atomic U64	3509 MiB (+19.8%)
	Ticket Board	3461 MiB (+18.2%)
	Acceleration Hash (32-bit)	4269 MiB (+45.7%)
	Acceleration Hash (8-bit)	3622 MiB (+23.7%)

Hash Table Trade-Offs

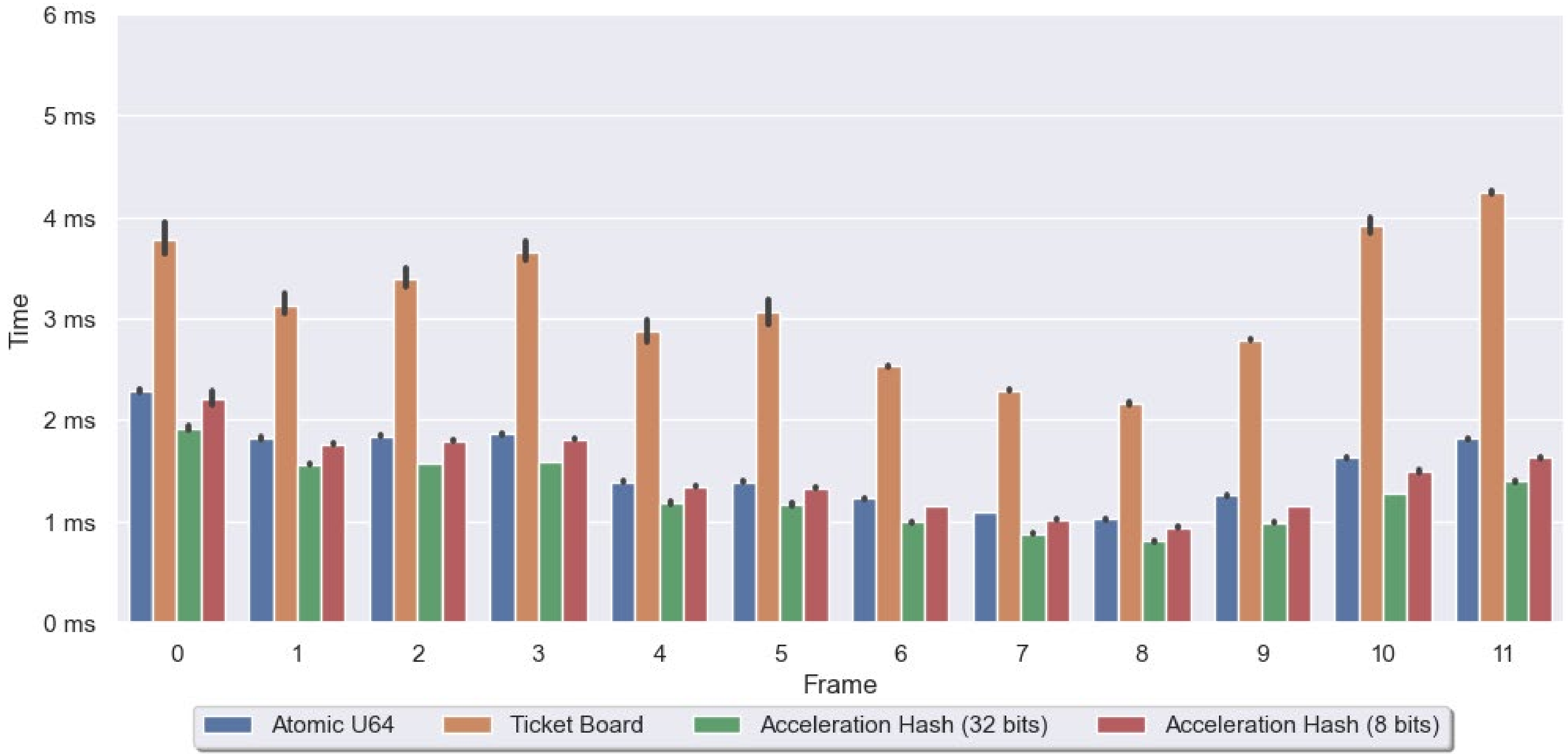
	Memory Usage
64-bit Atomic	+
Ticket Board	+
32-bit Acceleration Hash	-
8-bit Acceleration Hash	o

Editing Performance

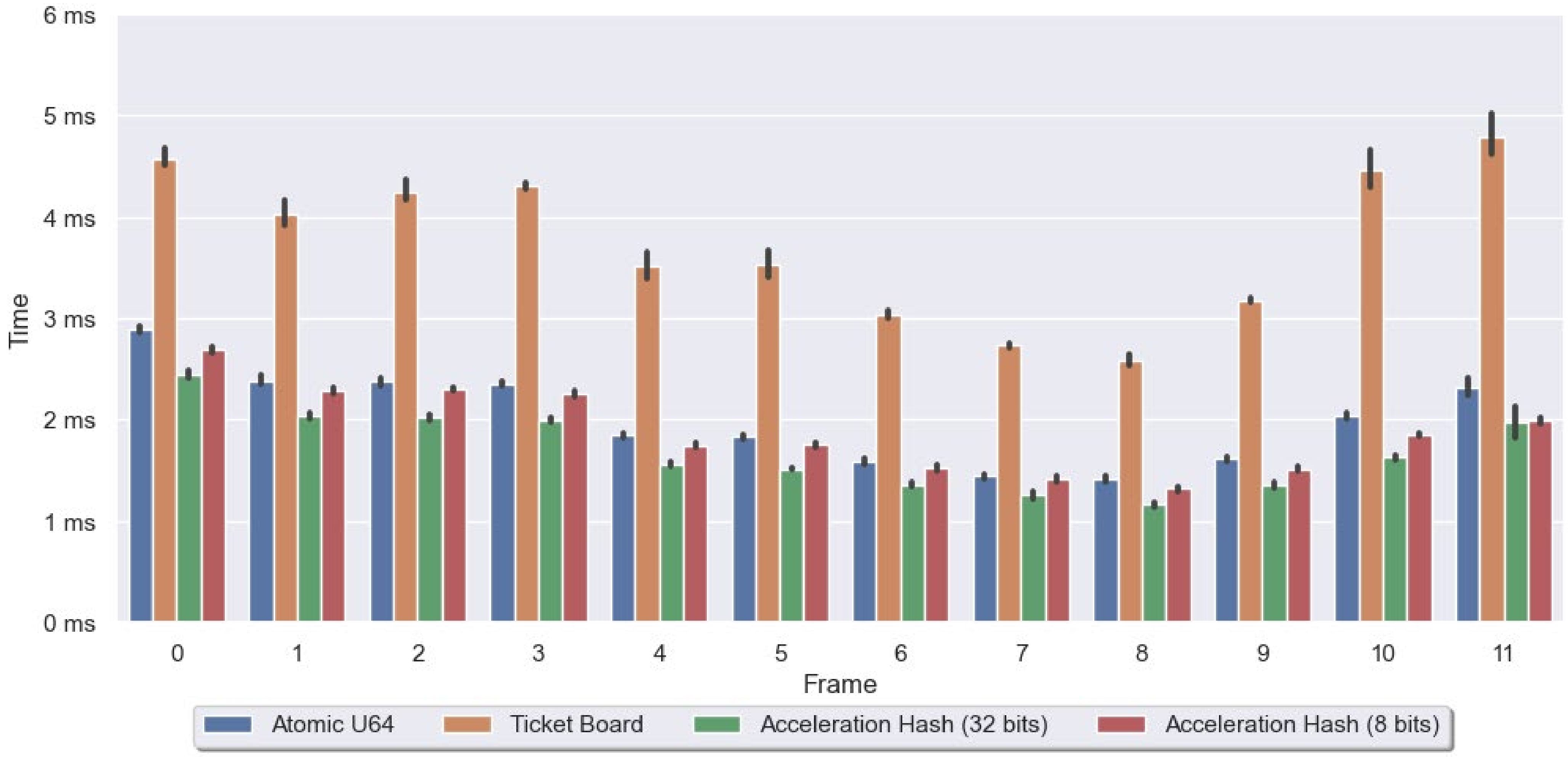
Frame Time (Without Materials)



Merge SVO with SVDAG (Without Materials)



Merge SVO with SVDAG (With Materials)



TU Delft

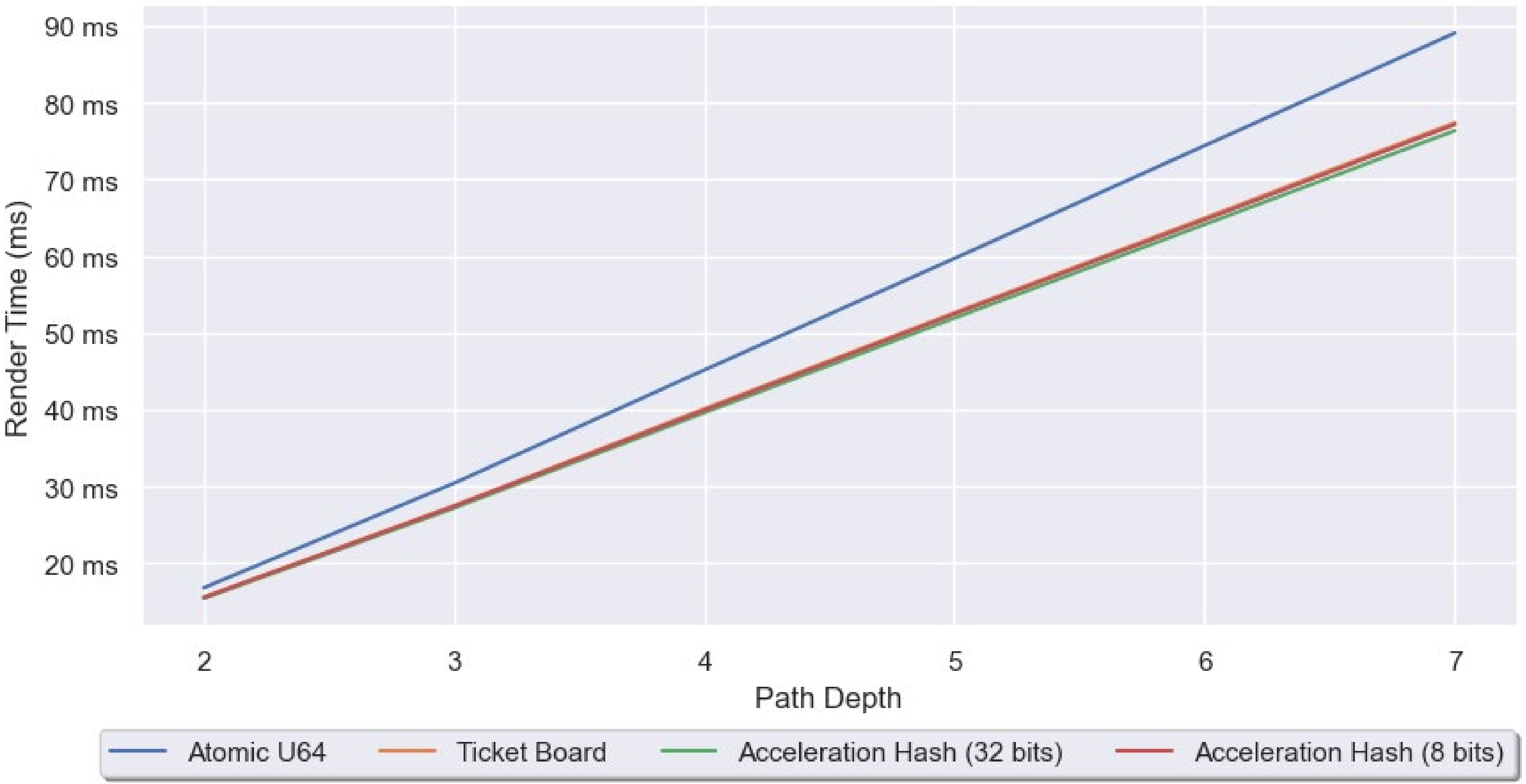
Hash Table Trade-Offs

	Memory Usage	Editing
64-bit Atomic	+	+
Ticket Board	+	-
32-bit Acceleration Hash	-	+
8-bit Acceleration Hash	o	+

Rendering Performance

Path Tracing – Inside of Stanford Bunny $16K^3$

Path Tracing – Interior of Stanford Bunny $16K^3$



Hash Table Trade-Offs

	Memory Usage	Editing	Ray Tracing
64-bit Atomic	+	+	o
Ticket Board	+	-	+
32-bit Acceleration Hash	-	+	+
8-bit Acceleration Hash	o	+	+

Hash Table Trade-Offs

Most compact with good editing performance

	sage	Editing	Ray Tracing
64-bit Atomic	+	+	o
Ticket Board	+	-	+
32-bit Acceleration Hash	-	+	+
8-bit Acceleration Hash	o	+	+

Fast rendering and editing, at the cost of memory usage.

Limitations & Future Work

- Using materials reduces SVDAG compression ratio significantly
- More advanced Garbage Collection

TU Delft

Closing Remarks

Details can be found in the paper & supplemental material

Planning to release the code on our website and GitHub:
https://publications.graphics.tudelft.nl/papers/13

https://github.com/mathijs727/