Dynamic GPU Sparse Grids and Voxel-sequential Processing in NanoVDB

EFTYCHIOS SIFAKIS, NVIDIA, USA and University of Wisconsin-Madison, USA
FRANCIS WILLIAMS, NVIDIA, USA
CHRISTOPHER HORVATH, NVIDIA, USA
ROHAN SAWHNEY, NVIDIA, USA
MARK HARRIS, NVIDIA, Australia
JONATHAN SWARTZ, NVIDIA, New Zealand
KEN MUSETH, NVIDIA, USA

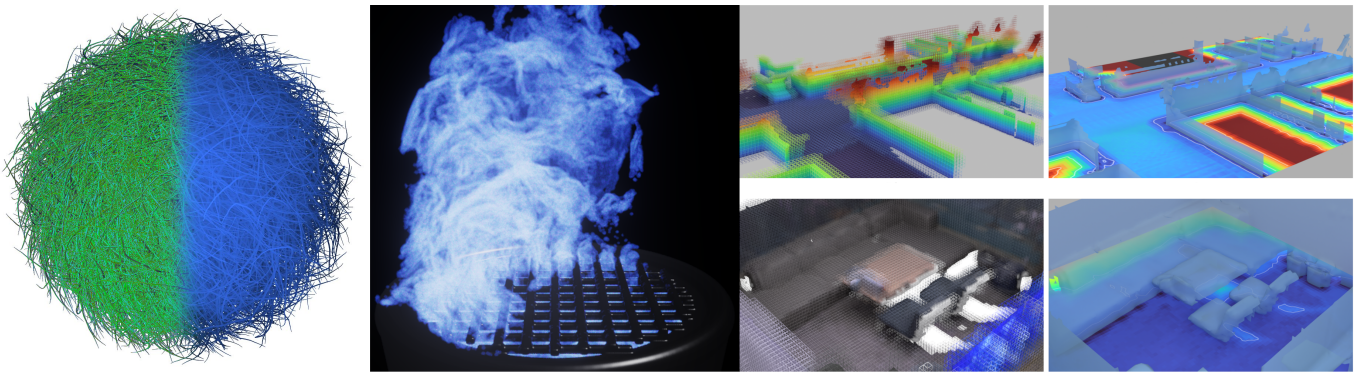


Fig. 1. We demonstrate our enhancements to NanoVDB in three diverse applications that rely on dynamic sparse grids. From left to right: **Large-Scale Voxelization** of a 2.8M-triangle hairball mesh [McGuire 2017] (left hemisphere in green), our `MeshToGrid` implementation generates a sparse index grid with 9.88B active voxels and UDF values stored in a sidecar (right hemisphere in blue) in 3.24s; **Fluid Simulation** of fire using the ghost fluid method [Hong et al. 2007; Nguyen et al. 2002, 2001] to track the combustion flame front using a truncated signed distance field; a **Reality Capture** processing pipeline which fuses a TSDF surface mesh, on the left, and derives a Euclidean signed distance field (ESDF), on the right, to facilitate collision checking and path planning from an input LiDAR sweep or a sequence of depth images. Hardware details in Appendix A.

We introduce two fundamental advances to the NanoVDB data structure: dynamic GPU-native topological updates, and a voxel-sequential access paradigm that dramatically accelerates point-wise and stencil-based operators on sparse grids. NanoVDB's linearized memory layout has delivered strong GPU performance, and the Index-Grid concept introduced in the fVDB framework [Williams et al. 2024] provided topology/data separation and optimal compactness of storage in sparse machine learning workloads. However, all these prior iterations of NanoVDB assume a static topology that has precluded use of this data structure in simulation and dynamic geometric processes. We eliminate this limitation by introducing a suite of dynamic topology operators – morphological dilation, grid merging, pruning, and cross-resolution coarsening and refinement. These operators use bit-parallel algorithms to reconstruct grids containing hundreds of millions of voxels in just a few milliseconds.

We also introduce a voxel-sequential access paradigm and data structure for GPU kernels operating on sparse data. We stream active voxels as a flat linear sequence, minimizing or eliminating the need for tree traversal in the process. This achieves compute density and regularity that allow even complex stencil-based operators to approach peak memory bandwidth, with minimal storage overhead. Together, these contributions remove the read-only constraint from NanoVDB, establishing it as a high-performance primary representation for dynamic GPU simulation and geometry manipulation. We evaluate the efficacy of our newly introduced operators in benchmarks sourced from applications such as 3D reconstruction, fluid simulation, and large-scale mesh voxelization.

CCS Concepts: • **Computing methodologies** → *Vector / streaming algorithms*; *Volumetric models*; *Graphics processors*; • **Information systems** → *Data structures.*

Additional Key Words and Phrases: Sparse volumetric data, GPU data structures, Dynamic topology, NanoVDB, Voxel-sequential access, Index-grids

ACM Reference Format:
Eftychios Sifakis, Francis Williams, Christopher Horvath, Rohan Sawhney, Mark Harris, Jonathan Swartz, and Ken Museth. 2026. Dynamic GPU Sparse Grids and Voxel-sequential Processing in NanoVDB *ACM Trans. Graph.* 45, 6, Article 225 (December 2026), 21 pages. https://doi.org/10.1145/3842564

Authors' Contact Information: Eftychios Sifakis, NVIDIA, USA, esifakis@nvidia.com and University of Wisconsin-Madison, USA, sifakis@cs.wisc.edu; Francis Williams, francis@fwilliams.info, NVIDIA, USA; Christopher Horvath, chorvath@nvidia.com, NVIDIA, USA; Rohan Sawhney, rsawhney@nvidia.com, NVIDIA, USA; Mark Harris, mharris@nvidia.com, NVIDIA, Australia; Jonathan Swartz, jswartz@nvidia.com, NVIDIA, New Zealand; Ken Museth, kmuseth@nvidia.com, NVIDIA, USA.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 1557-7368/2026/12-ART225
https://doi.org/10.1145/3842564

1 Introduction

The need to store and manipulate high-resolution, sparse volumetric data is ubiquitous across the spectrum of modern spatial computing. From high-fidelity fluid simulations in visual effects to massive datasets in medical imaging and neural radiance fields, a few recurring technical challenges emerge prominently: *efficiency* in memory utilization, *scalability* in computational throughput and accommodation of larger domain sizes, and the necessity of *sparsity* in representing irregular 3D domains. The OpenVDB data structure [Museth 2013] and its various derivatives were born out of these challenges, emerging as the industry standard for encoding sparse grid-embedded data. By providing a framework to represent virtually unbounded Cartesian domains with sparse occupancy, OpenVDB enjoys a rich ecosystem of operators and broad adoption in digital content creation tools. More recently, NanoVDB [Museth 2021] has extended this functionality to the GPU with emphasis on implicit geometry modeling and real-time rendering tasks, while the fVDB framework [Williams et al. 2024] further extended the scope of NanoVDB and demonstrated its efficacy in sparse machine learning workloads. NanoVDB has seen industry-wide adoption, powering tools such as Blender's Cycles rendering engine, Houdini's Pyro, Vellum and MPM solvers, Autodesk's Arnold, Chaos' V-Ray, and Pixar's RenderMan.

Despite its success, the VDB ecosystem – as originally materialized in OpenVDB – was fundamentally conceived for the CPU, relying on fine-grained heap allocation and pointer-heavy nodes, accelerated by means of a software cache implemented in *ValueAccessor* objects. Although this design allows for the effortless topological change required by dynamic simulation, it is not the most natural fit for the massively parallel, throughput-oriented architecture of the GPU. NanoVDB [Museth 2021] was introduced to facilitate the transition of VDB grids to GPU. Originally intended for read-only applications, e.g. random access tasks like ray-tracing and collision detection, NanoVDB forgoes heap allocation and pointer-based bookkeeping by adopting a linearized memory layout with relative addressing, specifically tailoring the structure for SIMT execution.

In a phenomenon that should not surprise us given the precedent of other high-throughput systems, NanoVDB's narrower scope – sacrificing OpenVDB's topological malleability for a compact, prebaked buffer – results in a structure that is not only more portable but significantly faster for streaming, sampling, and rendering tasks. Consequently, OpenVDB remains the tool of choice for dynamic, evolving topologies at the cost of peak performance, while NanoVDB serves as the expert subset for GPU-accelerated workloads, limited by its static, read-only topology. So far, no published research has seriously challenged this limitation, leaving developers with an untenable choice: the dynamic topological versatility of the CPU (OpenVDB) or the read-only speed of its GPU sibling (NanoVDB).

In this paper, we resolve this tension by introducing a framework for the dynamic evolution of NanoVDB sparse volumes directly on the GPU. We demonstrate that static topology is not an inherent trait of the NanoVDB data structure, but rather a limitation of the algorithms previously used to maintain it. Our work specifically aims to be synergistic with NanoVDB index-grids [Williams et al. 2024], the latest feature iteration of the data structure that explicitly separates the active voxel set topology from the data payload stored in it, and also offers the highest compactness of storage of all VDB variants. Our tools target the rapid reconstruction of the tree's hierarchical backbone independently of the underlying data payload. We adopt a bulk-synchronous manipulation paradigm where topological changes are enacted collectively in discrete epochs, and enable a suite of operators – including morphological dilation, grid merging, pruning, and multi-resolution refinement – that can reconstruct trees with hundreds of millions of voxels in a few milliseconds.

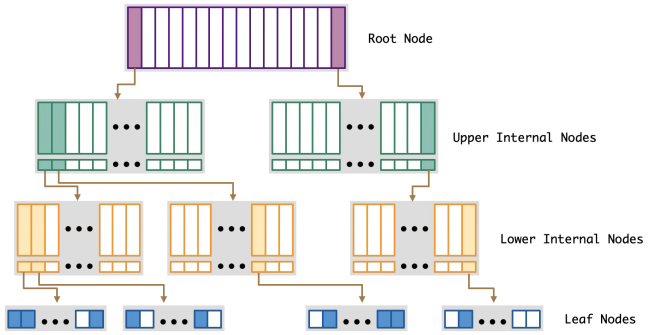
With topological manipulation now handled efficiently on the GPU, the remaining fundamental concern for sparse simulation codes is the performance of streaming and stencil operations. *Accessor* objects – originating in OpenVDB and carried into NanoVDB – are the standard paradigm for data access in the VDB ecosystem. They act as software caches that hide the cost of tree traversal, if the locations being accessed are spatially coherent. They are also available on the GPU (in NanoVDB) although their performance benefit may be hampered at times due to thread divergence when an accessor needs to update its local cache after a miss. Another challenge stems from the fact that on the GPU, the natural way to dispatch parallel threads is to assign a threadblock (e.g. 512 CUDA threads) to process all voxels of a leaf node in parallel, leading to load imbalance if threads are wasted being assigned to inactive voxels. Although accessor-based random access still has its place (especially in ray tracing and collision processing workloads), we propose an alternative data traversal paradigm, which we label *voxel-sequential access* that is optimized for stencil and streaming operations on the GPU. Within this paradigm, we stream active voxels as a flat linear sequence, regardless of the leaf nodes they straddle along this ordering. We facilitate this via a new *VoxelBlockManager* data structure that opens the path to streaming and stencil operations at optimal compute density, and can absorb the cost of tree traversal via enhanced regularity and warp-synchronous operation.

Our specific contributions are:

- **Bulk-Synchronous Topological Evolution:** A paradigm for GPU- and VDB-native topological changes—including merging, pruning, and refinement—on linearized index-grids. We argue this is the most fitting abstraction of *dynamic* topology evolution in GPU-focused workloads. We demonstrate bit-parallel algorithms that reconstruct complex topologies with hundred million voxels in milliseconds.
- **Injection and Sidecar Operators:** A suite of operators for the efficient integration and evolution of auxiliary *sidecar* payloads alongside the primary topology.
- **The Voxel-Sequential Access Paradigm:** The *VoxelBlockManager* (VBM) data structure, which replaces legacy hierarchical traversal with a linear streaming pattern. This eliminates thread divergence and load imbalance, significantly boosting efficiency on stencil-oriented workloads.
- **High-Performance Validation:** We demonstrate the efficacy of our framework additions in microbenchmarks sourced from real-world workloads, including 3D geometry reconstruction, interface tracking in fluids simulation, and high-resolution 3D mesh voxelization.

	OpenVDB	NanoVDB		
	[Museth '13]	Original [Museth '21]	IndexGrid (fVDB) [Williams et al '24]	Ours
Storage Sparsity	Moderate	Moderate	High	High
Memory Layout	Fragmented	Linear	Linear	Linear
Topology / Data Separation	No	No	Yes	Yes
Random Access	Yes (Read/Write)	Yes (Read only)	Yes (Read only)	Yes (Read only)
Data access	Fast	Very fast	Very fast	Very fast
Platform	CPU Only	CPU & GPU	CPU & GPU	CPU & GPU
Topology Adaptation	Dynamic (at voxel granularity)	Static (requires reinitialization)	Static (requires reinitialization)	Dynamic (bulk-synchronous)
Voxel-sequential Access	No	No	No	Yes (SIMD/SIMT optimized)
Designed For	General purpose Maximum flexibility Competent performance	Ray tracing (GPU) Implicit surfaces (static) High performance	General static topology and ML workloads. High performance/sparsity	General simulation, rendering, ML workloads (dynamic data) Optimal performance/sparsity

Fig. 2. **Feature comparison of VDB iterations** Our proposed enhancements combine the highly compact storage of NanoVDB Index-Grids, the GPU-compatibility of all NanoVDB iterations, support for GPU-native topology modification operations, and a new voxel-sequential access paradigm



Name	LOG2DIM	Branching	Voxel span
Root	—	n_{Tiles}	unbounded
Upper Internal	5	$32^3 = 32{,}768$	4096^3
Lower Internal	4	$16^3 = 4{,}096$	128^3
Leaf	3	$8^3 = 512$	8^3

Fig. 3. **Branching factor and voxel span** for each level of the NanoVDB tree (LOG2DIM = 5, 4, 3). Each node represents a span of $2^{3\cdot\text{LOG2DIM}}$ voxels, at the leaf level. The root holds an unbounded, hash-keyed list of n_{Tiles} tiles.

2 Background and Related Work

We will start with an overview of the evolution of the data structures in the VDB ecosystem, and the implications that the design decisions in every iteration brought about. Figure 2 highlights some of the key architectural differences, which are further detailed below.

OpenVDB [Museth 2013] introduced the core concept of the VDB tree structure: a hierarchical tree-based representation with fixed depth but aggressive and varying branching factors. At the coarsest scale, the sparse 3D voxel domain is covered by aligned macroscopic *root tiles* (nominally spanning 4096^3 voxels), which are maintained by the root node in a linear list or a hashtable.

If a root tile requires refinement rather than carrying a constant value, an *upper node* is associated with it, spanning the same area and holding the branch of the tree that refines it. Upper nodes are subdivided into voxel partitions of size 128^3 voxels (so called *lower nodes*) and those into smaller aligned partitions of 8^3 (*leaf nodes*).

A leaf node densely allocates a buffer (on the heap) sufficient to store all $8 \times 8 \times 8$ values within its span, and a bit-encoded mask indicating an active/inactive state for each voxel in its domain. OpenVDB strives to present a usage model that is as permissive as a hypothetical, infinite-extent dense grid would be: random read/write access is supported to arbitrary locations in the Cartesian domain, with the data structure performing on-demand allocation to create storage for any location that was not previously provisioned for.

The fundamental abstraction that OpenVDB uses to mitigate the (substantial) worst-case performance cost of random access is the concept of *accessors*, cf. `ValueAccessor` in [Museth 2013]. An accessor is a tool for read/write access at arbitrary coordinates, which seeks to accelerate access cost if the read/write requests are made for spatially coherent locations. Specifically, if repeated access requests are made for voxels that fall within the same leaf node, a software cache that the accessor incorporates will completely bypass tree traversal and provide fast access to the requested value. That is, an accessor allows for bottom-up vs top-down tree traversal, optimally avoiding traversal altogether. However, the highly permissive use model of OpenVDB limits its potential for aggressive optimization: the data structure is clearly targeting CPU-only usage, as accessors are prone to warp divergence (they need to modify their internal cache even if they access data for read purposes only), and the allocate-on-insertion model for topology updates is very challenging to reconcile with the typical GPU/CUDA programming paradigm.

NanoVDB [Museth 2021] was introduced as a narrowly-scoped subset of OpenVDB that was expressly aimed at maximizing portability, supporting use in both CPU and GPU, x86 and ARM platforms, and even within shading languages. NanoVDB froze the geometric sizes of the tree nodes from being tunable at compile-time in OpenVDB, to a standardized 4096/128/8 node size progression as illustrated in Figure 3, and replaced the *allocation-on-access* model of OpenVDB with a strict *allocation-on-initialization* without support for changing the grid topology after it was created. Leaf nodes and their data would be allocated in a contiguous, linear storage span, rather than the fragmented heap-based allocation that OpenVDB required. NanoVDB is more efficient in (spatially coherent) random access than OpenVDB on the CPU, due to the fact that NanoVDB uses linearized, cache- and prefetch-friendly storage. Of course, in this initial iteration of NanoVDB, its linear memory model came at the expense of no support for performing edits to the grid topology (modifying the active voxel set) without requiring rebuilding the grid from scratch.

Index-Grids [Williams et al. 2024] were a crucial addition to NanoVDB, introduced to facilitate support for machine learning workloads in the fVDB framework. In an Index-Grid, the value associated with each active voxel is not the actual data "payload", but an integer index into an auxiliary linear storage, referred to as a *sidecar*. This topology/data separation provided a number of substantial benefits. First, it is trivial to store scalar fields corresponding to distinct

quantities associated with each voxel (e.g. density, pressure, level set value, etc in a fluid simulation), while reusing the same topology, but distinct "sidecar". Second, it provides the maximum potential for sparsity, as the sidecar only needs to allocate space for *active* coordinates, at voxel granularity; in contrast, storage for the prior NanoVDB iterations had to be allocated for all 8^3 voxels of every leaf node , regardless of how many of such voxels are active. Finally, the actual linear index associated with every active voxel coordinate does not need to be explicitly stored, but computed on-the-fly, with minimal storage footprint (typically less than 2 bits/voxel for grids with approximately 50% leaf-level occupancy). Still, in this latest iteration as presented in fVDB [Williams et al. 2024] NanoVDB (including IndexGrids) remained a read-only structure after initialization. Our work closes exactly this feature gap, describing a set of GPU-native dynamic topology manipulation operators.

At this point, we should clarify (and defend) the characterization of such operators as "dynamic", in contrast with the context that this term is used elsewhere in the VDB ecosystem. OpenVDB supports a truly laissez-faire paradigm of topology updates, where an individual voxel can be introduced or removed from the active set at any time. We argue that such an extremely permissive model of ad-hoc topology manipulation is not how typical high-throughput (and at the same time thread-safe and deterministic) simulation and geometry processing workloads ought to operate. OpenVDB topology updates are not thread-safe, and must be guarded in use-specific ways to ensure that each execution thread doesn't attempt to consume voxels that were asynchronously created by another. We advocate, especially in light of GPU-oriented workloads, for a bulk-synchronous model of topological adaptation, where the topological change is prescribed holistically for the entire domain (e.g. a morphological dilation, or a topological union of two grids), and enacted as one synchronous operation. Within this framework, we defend the "dynamic" label for our update operators in the sense that (a) they are GPU-native and highly optimized, and (b) they operate natively within the VDB representation. The one concession we commit to is the need to retain in memory both the "before" and "after" version of the VDB topology, while the operation takes place, although the absolute storage cost of this is small, due to the topology/data separation of NanoVDB Index-Grids.

Sparse grids for level set methods and volumetric modeling. Sparse volumetric grids have a long history in computer graphics, particularly in applications such as implicit modeling, fluid simulation, and level set methods. Because this body of literature is too extensive to cover in its entirety, we restrict our focus to data structures designed for narrow-band level set simulations – a central application of the current work – and exclude VDB-based approaches discussed above. Existing methods generally fall into three categories: hierarchical, blocked, or compressed representations. Hierarchical methods typically utilize height-balanced octrees [Losasso et al. 2004; Min 2004; Strain 1999] to adaptively refine the grid near the interface; however, these structures often suffer from the computational overhead associated with deep tree traversals and pointer chasing. To address cache locality, blocked or tiled schemes, eg. [Lefohn et al. 2003], group voxels into contiguous memory blocks, leveraging hardware-friendly layouts such as those found in SPGrids [Setaluri et al. 2014] and the more recent Multiresolution Sparse Block Grids (MSBG) [Braun et al. 2025]. Beyond its memory layout, SPGrid also demonstrates that a pyramid of sparsely populated uniform grids can support adaptive, multi-resolution stencil work without explicit tree traversal – the same coupling of resolution change and stencil computation that our coarsening and refinement operators are intended to support. Conversely, compressed representations such as the Dynamic Tubular Grid (DT-Grid) [Nielsen and Museth 2006] and Hierarchical Run-Length Encoding (H-RLE) [Houston et al. 2006] generalize compressed-row-storage and RLE to 3D. While these formats provide exceptional memory efficiency for static volumes, they incur significant re-structuring costs when the topology is dynamic and requires frequent updates. Finally, out-of-core approaches [Nielsen et al. 2007] enable streaming computing of extremely large level sets, however their performance is fundamentally bottlenecked by I/O bandwidth and the latency of disk-to-CPU data transfers and as such cannot match the computational speeds of GPU approaches like ours.

Block-sparse hash grids for volumetric reconstruction. Outside the VDB ecosystem, the dominant sparse-volumetric representation in real-time 3D reconstruction is the *block-sparse hash grid*: integer block coordinates (typically 8^3 voxels per block) are inserted into a parallel hash map, and each occupied block heap-allocates a dense voxel payload. Newcombe et al. [2011]'s KinectFusion popularised dense volumetric TSDF fusion from depth streams, and Nießner et al. [2013] replaced the dense grid with a GPU hash map to remove the bounded-domain restriction. This block-hash recipe has since been adopted across the field: Voxblox [Oleynikova et al. 2017] extends it to incremental Euclidean signed distance fields for MAV planning, nvblox [Millane et al. 2024] ports the same recipe onto the GPU, and Open3D [Zhou et al. 2018] ships a block-hash TSDF backend built on ASH [Dong et al. 2023], a modern GPU spatial-hashing framework. Within the VDB family itself, VDBFusion [Vizzo et al. 2022] demonstrates that an OpenVDB-backed TSDF pipeline reaches comparable quality without committing to a specific block resolution at compile time. The octree alternative pioneered by OctoMap [Hornung et al. 2013] remains widely deployed in robotics for occupancy mapping. All of these systems are built around a static fusion-and-query loop — they grow topology as new observations arrive but do not support the kind of bulk topological rewriting (dilation, union, pruning, refinement) that fluid simulation, dynamic-scene mapping, and geometry processing demand. Section 5.1 compares directly against nvblox, VDBFusion, and Open3D on the same reconstruction benchmarks.

Sparse-voxel deep learning frameworks. A parallel line of work targets sparse 3D learning rather than reconstruction. Minkowski Engine [Choy et al. 2019] and spconv [Spconv Contributors 2022] encode active coordinates in a hash table and run sparse convolutions over the resulting key-value structure; TorchSparse [Tang et al. 2022] and its successor TorchSparse++ [Tang et al. 2023] extend this design with kernel-fusion and tile-sparsity optimisations. O-CNN [Wang et al. 2017] and Kaolin [Jatavallabhula et al. 2019] instead embed sparse data in an octree, while Taichi [Hu et al. 2019] provides a more general DSL for spatially-sparse compute that has been used both for simulation and for learning. NeuralVDB [Kim

et al. 2022] explores neural compression of VDB topology, complementary to the explicit dynamic operators we introduce here. Compared with these flat hash-keyed representations, the VDB hierarchy used by fVDB [Williams et al. 2024] retains spatial locality at every level of the tree, which our work exploits to make topological updates — bit-parallel at the leaf, hash-based at the root — both deterministic and fast.

Implicit and generative alternatives. Recent neural representations sidestep explicit sparse grids by encoding geometry in network weights and learned feature volumes. Instant-NGP [Müller et al. 2022] stores trainable features in a multiresolution hash table whose collisions are disambiguated by the downstream MLP — an approach that is highly efficient for fitting but does not expose an explicit topology that downstream geometric operators can act on. Closer to our setting, NKSR [Huang et al. 2023] demonstrates large-scale neural surface reconstruction from sparse oriented points using a VDB-backed kernel basis, and XCube [Ren et al. 2024] performs voxel-hierarchy diffusion directly on VDB grids to generate 1024^3 scenes in a single feed-forward pass. Both rely on fVDB for their voxel topology, and both would benefit directly from the dynamic topology operators we develop here: NKSR's basis support could be incrementally refined as new observations arrive, and XCube's coarse-to-fine generation could write its progressively finer levels through our refinement operator rather than rebuilding the grid from scratch.

Resolution-selection criteria. A complementary line of work focuses on how resolution should adapt. In volumetric reconstruction the criterion is typically derived from an observation model rather than from geometry alone: Vespa et al. [2019] select the integration scale within an octree from the effective sensor resolution at the observed depth, while De Rebotti et al. [2025] adapt voxel size to the local variance of SDF observations in a flat multi-resolution hash. Both evaluate the criterion per block and apply it incrementally, as observations arrive during fusion. Criteria of this kind are the natural drivers for the coarsening and refinement operators we introduce in section 3, which supply the corresponding mechanism on the GPU: such a predicate, evaluated over the active set, becomes the input mask of a coarsening or refinement epoch, which the operator then enacts as a single bulk rewrite of the topology. We deliberately keep decision and mechanism separable – the same primitives serve a sensor-driven, a variance-driven, or a purely geometric criterion without modification.

3 Dynamic Topology Evolution

We propose a bulk-synchronous model for evolving the topological domain of active voxels in a NanoVDB sparse grid. Topological adaptation follows three steps: (a) topology changes (where and by what rule) are *decided* collectively; (b) prescribed changes are *materialized* in bulk, resulting in a new, valid topology and data state; and (c) computation resumes on this *new epoch* of the grid.

We adopt this epoch-based paradigm as the foundation of our dynamic topology updates, providing five operators for evolving the hierarchical tree structure: (1) Morphological *dilation* expands the active voxel set into its immediate neighborhood; (2) *Pruning* removes voxels based on an input mask; (3) Merging computes a Boolean union of two Index-Grids; (4) Coarsening downsamples by a structured 2:1 ratio on each axis (8:1 volumetric reduction); and (5) Refinement, the adjoint of coarsening, upsamples by the same 2:1 ratio.

These operators compose into macroscopic, domain-specific workflows. For example, a typical level set narrow-band update is recast as three discrete steps: a speculative dilation to accommodate the anticipated motion of the interface; a fixed-topology numerical update of the level set values; and a pruning pass to re-establish the narrow-band limits and discard voxels that have moved beyond the region of interest. These operators are designed specifically for *Index-Grids* [Williams et al. 2024], which only store the topology structure of the grid, leaving the data payload to a separate linearized sidecar. We first describe the common skeleton (and individual details) for these operators as they pertain to the rebuilding of the topology-only grid itself; operators that update the sidecar storage in response to topological changes are discussed in section 3.4. The applications of section 5.1 additionally use the prior `voxelsToGrid` primitive of Williams et al. [2024], which rasterizes a set of integer coordinates into a fresh Index-Grid; we treat it as a black-box building block.

Figure 4 illustrates the conceptual pipeline that our topological operators follow. Several elements of this pipeline are common to all operators, while others are specialized per operation; we call these out explicitly in the description of fig. 4.

3.1 Topology Enumeration and Allocation

Index-Grid reconstruction follows a multi-phase pipeline designed to saturate GPU parallel resources while maintaining a linearized memory layout. It begins with *topology enumeration*, where we determine the hierarchical composition (node count per tree level) of the new grid before any final memory is allocated.

Root Tile Table Pre-processing. We first construct a *conservative envelope* of candidate root tiles for the target operation (fig. 4; Step 1). While a tight estimate is ideal, we prefer to over-provision and speculatively create tiles that may go unused, as these are removed in subsequent steps. The construction of this candidate set is operator-specific: for *merging*, we take the union of all input root tiles; for *pruning*, we retain the existing set (deferring removal of unused ones). For *dilation*, *coarsening*, and *refinement*, we analyze the index-space bounding box of each existing tile and instantiate candidate neighbors wherever the operation may trigger spatial expansion. These operations run on the CPU (e.g., via a hash table), as the root table is a minuscule fraction of the tree's total memory footprint.

Hierarchical Mask Marking. We next allocate two bit-encoded scratchpad structures to enumerate the result tree nodes precisely (fig. 4; Step 2). To visualize their encoding, consider as in fig. 5 a *conceptual dense NanoVDB tree* that starts at the candidate root tiles and branches out to a dense set of child nodes at every finer level, down to leaf nodes. In NanoVDB, each non-leaf node maintains a bitmap (`childMask`) marking all its geometric children that contain active voxels. Our scratchpads store, sequentially, all the `childMask` bitmaps at the two internal levels of the tree, as seen in fig. 5.

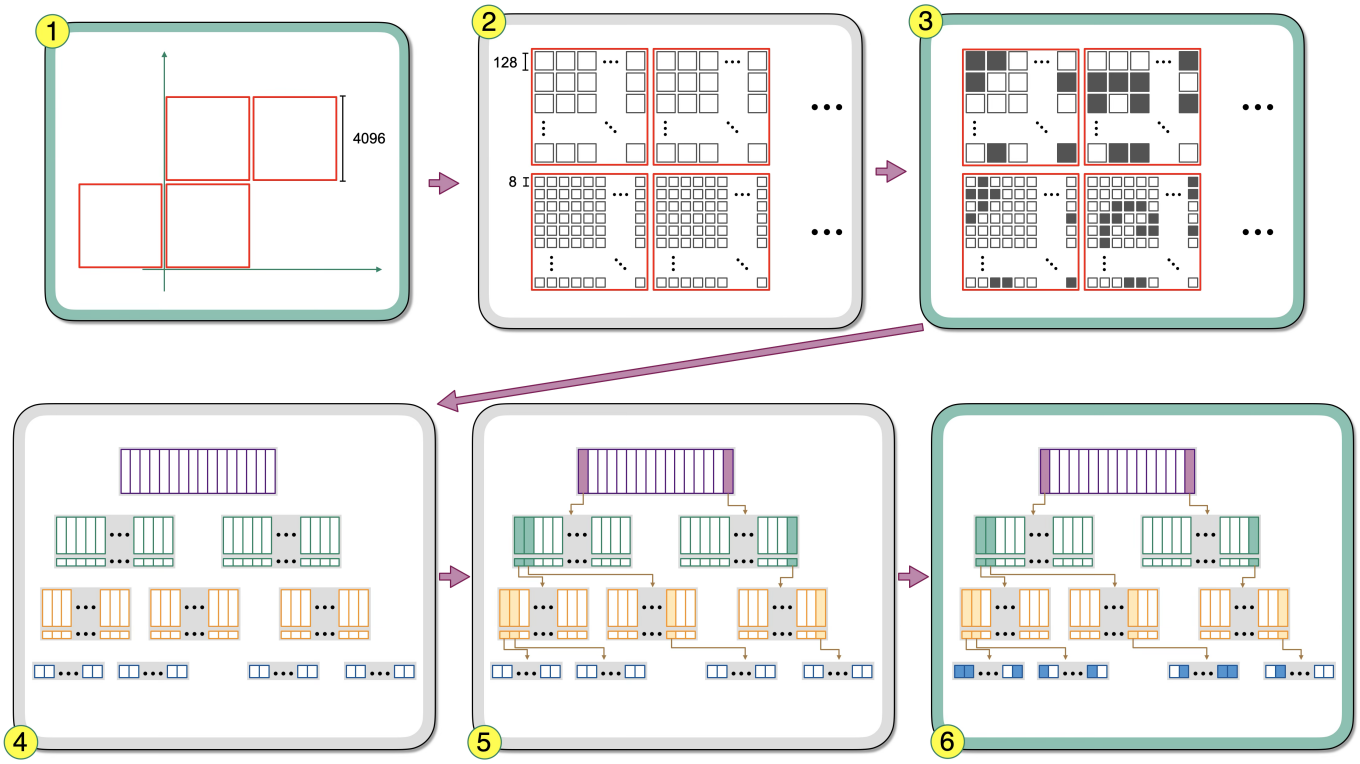


Fig. 4. **Process pipeline of topological operators.** For each step we indicate whether it is operation-specific (outlined in green) or generic (outlined in grey) across all operations. [Step 1; op-specific] A candidate set of root tiles (each spanning 4096^3 voxels) is introduced to conservatively cover the resulting topology. [Step 2; generic] For each candidate root tile, we allocate a dense bit map(representing the upper tree node) encoding the presence of each lower tree node (spanning 128^3 voxels each; top row) and each leaf node (8^3 voxels each; bottom row). [Step 3; op-specific] Depending on the operation, we mark in parallel all lower/leaf nodes that must be materialized, using the previously allocated bitmap scratchpads. [Step 4; generic] We enumerate all necessary tree nodes (pruning empty ones) and monolithically allocate storage for the resulting Index-Grid. [Step 5; generic] The connectivity of the tree is rebuilt in parallel from the lower/leaf bitmap scratchpads. [Step 6; op-specific] The previous stage leaves a properly allocated sparse bitmap that tightly covers the active voxel set, but with no voxels yet marked active; we populate the active voxel flags in an operation-dependent way to complete the operation. [Step 7; generic (not pictured)] A final bottom-up sweep updates any remaining metadata (voxel offsets, bounding boxes, checksum etc).

NanoVDB defines a templatized class `Mask<N>` for storing bitmaps of dimension $(2^N)^3$. At the upper level, these are stored in an array, `upperMasks`, of `Mask<5>` bitmaps (32^3 bits each). At the lower level, we allocate a 2D array, `lowerMasks`, consisting of 32^3 `Mask<4>` entries per root tile, indexed by the parent tile ID and the local linear offset of the child node. The storage footprint is remarkably modest: 4 KB for `upperMasks` and 16 MB for `lowerMasks` of each candidate root tile. Since a single root tile covers 4096^3 voxels, practical grids rarely exceed a few dozen tiles. Once a tile's occupancy exceeds a mere 0.01%, the footprint of a single scalar float channel already outweighs these temporary bookkeeping structures.

These structures serve as parallel scratchpads for capturing the target topology (fig. 4; Step 3). For each operator, we evaluate the input topology in parallel to determine which leaf nodes must exist in the output epoch. For *merging*, an active leaf in either input mandates inclusion; for *pruning*, a leaf persists if any of its active voxels remain after mask application. (The more intricate logic of *dilation* is addressed in section 3.3.) Necessary leaf nodes are directly marked in the scratchpad bitmaps along with the lower node parents. Because the scratchpads are dense, threads determine the target bit location statically via a simple root-tile table probe. We use `atomicOr` reductions to populate the bitmaps in parallel, giving high throughput without complex synchronization.

Prefix Sums and Monolithic Allocation. Once the scratchpad bitmasks are marked, we resolve the final hierarchical counts through a series of parallel prefix sums. We define three intermediate count arrays: `upperCounts`, `lowerCounts`, and `leafCounts`.

Array	Size	Definition (for index i)
`upperCounts`	nTiles	1 if i-th upper node has any active lower node children; else 0
`lowerCounts`	nTiles × 32K	1 if i-th lower node has any active leaf node children; else 0
`leafCounts`	nTiles × 32K	Number of leaf node children of the i-th lower node (`popcnt` of the lower node's `childMask`)

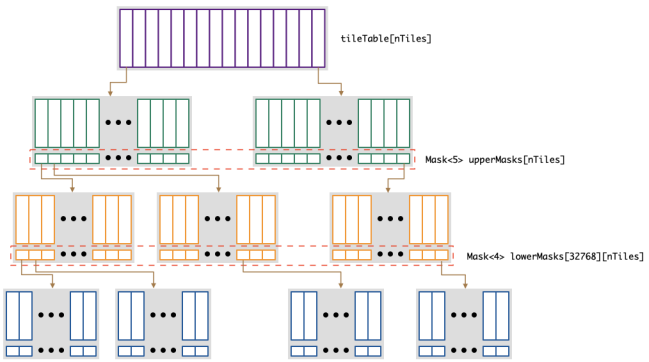


Fig. 5. **Topology Enumeration via Linearized Scratchpads.** To facilitate parallel reconstruction, we map the tree onto a *conceptually dense* intermediate representation of serialized bit-mask arrays. Each candidate root tile is associated with one `Mask<5>` in the `upperMasks` array and a block of 32^3 `Mask<4>` entries in the `lowerMasks` array. This layout resolves any coordinate to a specific bit via $O(1)$ spatial-to-index mapping, capturing the exact occupancy of the next epoch before the final NanoVDB buffer is materialized.

For each potential lower internal node i in the flattened sequence, we set `lowerCounts[i]` to 1 if the corresponding `Mask<4>` – which identifies the presence of its 16^3 potential *leaf* children – is non-empty, and `leafCounts[i]` to the bit population count (`popcnt`) of that same mask. `upperCounts` is populated analogously, from the occupancy of the respective `Mask<5>` bit-masks.

To resolve global memory addresses, we compute inclusive prefix sums for these count arrays, using the NVIDIA CUB library for the parallel scans and reductions. The terminal element of each resulting offset array provides the exact global sum: the total number of nodes at that level of the hierarchy. Knowing these topological requirements *a priori*, we perform a single *monolithic allocation* for the entire tree structure (fig. 4; Step 4). This eliminates fine-grained heap allocations and multiple synchronization points, and stores the new grid epoch in a perfectly packed, contiguous buffer.

3.2 Index-Grid Materialization and Voxel Population

With the destination buffer allocated and the hierarchical offset arrays computed, we proceed to materialize the final Index-Grid in three parallel stages: structural reconnection, voxel-level bit-mask population, and topological post-processing.

Structural Reconnection and Root Pruning. The first stage "wires" the parent-child relationships within the new buffer (fig. 4; Step 5). Each thread responsible for a node in the scratchpad uses its corresponding prefix sum from the `offsets` arrays to resolve its absolute address in the destination buffer. This phase also performs the final pruning at the root level. While the CPU pre-processing phase speculatively created a *conservative envelope* of candidate root tiles, the `upperOffsets` array now acts as a contraction map: a candidate tile i is written to the final `tileTable` only if its entry in the offset array indicates it is non-empty.

For every surviving tile and internal node, we update the relative child pointers by sampling the starting offset of its children's range from the next level's offset array (e.g., an internal node at index j retrieves its child pointer from `leafOffsets[j]`). At the end of this pass the tree is fully valid for traversal, although the leaf-level voxel masks remain empty.

Operator-Specific Voxel Population. With the skeletal structure established, we populate the leaf-level voxel masks (`valueMask`) in an operation-specific manner (fig. 4; Step 6). Separating structural reconnection from bit-mask filling is a key optimization, as the latter can be performed in-place within the already-allocated hierarchy. For operators such as *merging* and *pruning*, threads perform bitwise `OR` or `AND` operations between input grids and masks, writing directly to the new leaf nodes. For *coarsening* and *refinement*, we perform structured 2:1 bit-permutations to downsample or upsample the occupancy patterns. The logic for morphological *dilation* is more complex and is detailed in section 3.3.

Topological Post-Processing. The final stage of the pipeline performs global bookkeeping. We compute a final prefix sum across the newly populated leaf bit-masks to assign a unique *voxel offset* to every leaf node. This count provides the total active voxel population of the tree and defines the size of the linearized data sidecar discussed in section 4. Finally, a parallel reduction updates the index-space bounding boxes and computes the grid checksum, leaving the grid fully compliant with the NanoVDB specification.

3.3 Morphological Operators: Bit-Parallel Dilation

While merging and pruning follow a straightforward implementation path, morphological dilation presents a unique challenge: an active voxel at the boundary of a *leaf node* may require materializing adjacent neighbors to accommodate the dilation stencil's footprint. For a standard $3 \times 3 \times 3$ stencil, a single boundary voxel may instigate the creation of up to seven neighboring leaf nodes (at a leaf corner), while the collective occupancy of a leaf can influence all 26 neighbors. To resolve this without expensive tree traversals for neighbor access, we implement a *hierarchical bit-parallel dilation* that decouples topological expansion from voxel-level population.

Leaf-level Neighbor Influence Masks. Dilation begins with a parallel sweep in which each leaf node is processed by an individual CUDA thread, determining whether the dilated footprint of the leaf's active voxels intersects any of its 26 leaf neighbors and thus necessitates their materialization in the result tree.

We recall that the $8 \times 8 \times 8$ voxel occupancy mask is bit-packed into eight `uint64_t` variables (in `Mask<3>`). For a standard $3 \times 3 \times 3$ box dilation, we infer whether each leaf neighbor must materialize using simple bitwise operations on these packed words: combining slices via bitwise `OR` and isolating the "faces" of the 8^3 volume via bitwise `AND` with predetermined masks yields a 27-bit *influence mask* per leaf, indicating whether each of its leaf neighbors needs to be introduced. These masks accommodate even wider dilation scenarios; for any stencil with a radius $R \leq 8$, the influence of an 8^3 leaf node is strictly confined to its immediate 3^3 cluster of leaf neighbors. This leaf-level computation is the only step that inspects the dilation stencil itself; all subsequent steps depend only on the resulting mask of affected leaf neighbors.

Leaf-Major to Tap-Major Transposition. To avoid the high contention of irregular atomic updates, we transpose these influence masks in parallel in shared memory, shifting from a *leaf-major* "scatter" (each leaf updates 27 taps) to a *tap-major* "gather" (the lower node computes 27 uniform spatial shifts). The left of the following illustration shows the state prior to this operation: for each of the 16^3 leaf node children of a *lower* internal node, we have a 27-tap bit stencil indicating *based on the voxels within the leaf node that this stencil is centered on, which of the* $3 \times 3 \times 3$ *neighboring leaf nodes should be made active?*

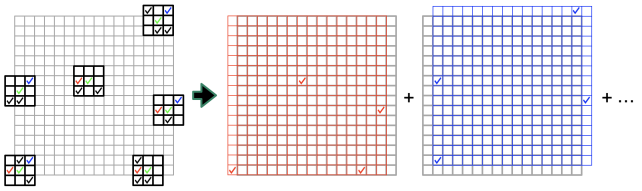


Fig. 6. **Leaf-major to tap-major transposition.** Per-leaf neighbor-influence stencils (left) are reorganized into one bitmask per stencil offset (right), enabling warp-collective processing.

We transpose this (leaf-major) grid-of-stencils into a tap-major stencil-of-grids (each grid encoded as a NanoVDB `Mask<4>`), as seen on the right side of the illustration above. The transposition at warp level, using `CUB::WarpReduce` to aggregate influence bits across lanes, tallying the "votes" of 32 leaves into a single bit-parallel write. The resulting `offsetMasks` represent the collective topological footprint of the lower internal node for every neighbor offset.

Warp-Collective Stencil Convolution. We finally combine the 27 "offset" `Mask<4>` bitmaps to compute the leaf nodes that should be marked active, both within the extent of this lower node and its $3 \times 3 \times 3$ neighbors. As a building block we implement a warp-level operator `MaskShift<di,dj,dk>`, a 3-dimensional bit shift on a `Mask<4>` with compile-time signed shift distances; this uses warp-level shuffle and rotation instructions to exploit bit-level parallelism. We accumulate the result by shifting each `offsetMask` by its own tap offset and combining the shifted masks with a bitwise `OR`: a bit set at leaf slot p of tap (di, dj, dk) indicates that slot $p + (di, dj, dk)$ must be materialized. Bits displaced past the 16^3 extent of the lower node are accumulated into the mask of the corresponding neighboring lower node, so the kernel yields 27 `resultMasks`: one for this lower node and one for each of its 26 neighbors.

Global Atomic Merging. The kernel concludes by committing the 27 resulting neighbor masks to the global scratchpads defined in section 3.1. For any neighbor mask containing active bits, the thread probes the `dilatedRoot` to find the target tile and uses `atomicOr` to update `upperMasks` and `lowerMasks`. This "blackboard" approach lets thousands of threads concurrently contribute to the new topology, which is then finalized via the prefix-sum pipeline.

Voxel Mask Reconstruction. One further dilation-specific computation is required. At Step 6 of fig. 4 the tree topology holding the dilation result exists, but no voxels have been set active. We perform this as a bit-parallel *gather*: every leaf node composes its `valueMask` (a bit-encoded `Mask<3>`) by fetching the `valueMasks` of neighboring leaf nodes from the source topology and combining them with bit-parallel shifts.

3.4 Sidecar Update Operators (Inject)

The operations described thus far affect only the *topology* (active voxel set) of a sparse Index-Grid, leaving the associated data payload (the "sidecar") unaffected. We deliberately keep the sidecar update as a separate operator, since with our GPU-optimized implementation the topological updates typically complete in less time than it takes to stream the data payload through memory. We introduce the following abstraction:

```
injectData(srcGrid, dstGrid, srcSidecar, dstSidecar)
```

This operation takes the source and target topologies (`srcGrid`, `dstGrid`) and their respective sidecars. It operates in accordance with the following semantics: any voxel that is present in the *intersection* of the two Index-Grids triggers a copy of the data from the location in the source sidecar to the location the same coordinate maps to in the destination sidecar; all other data in the destination sidecar is left untouched. This definition has proven fully adequate for all practical applications of the operators in this section.

Our implementation maximizes memory coalescing, especially where one topology is a proper subset of the other (in which case either reads or writes are coalesced). We assign each pair of source/destination leaf nodes with the same origin to an individual warp, leveraging the guarantee that data for active voxels within a single leaf node is sequential in the sidecar. We also provide variants of `inject` for strided sidecars and for sidecars holding multi-dimensional (vector-valued) features, as is common in fVDB machine learning workloads.

4 Voxel-sequential access and the VoxelBlockManager

A fundamental feature of Index-Grids [Williams et al. 2024] is the compactness of storage they enable: only values associated with *active voxels* need to be stored in the auxiliary sidecar, in contrast to prior iterations of NanoVDB, which quantize data allocation upwards to full $8 \times 8 \times 8$ blocks corresponding to the domain of each leaf node. It is natural to seek a matching compactness of computation, especially for streaming and stencil-oriented workloads. Ideally, a data-parallel operation such as applying a finite difference stencil to all active voxels would iterate over the linear sequence of active voxels, regardless of how they are distributed across leaf nodes or how sparsely occupied those leaf nodes are in different parts of the domain.

Implementing such an iteration paradigm, which we call *voxel-sequential access*, is significantly more challenging on the GPU. On the CPU, the typical parallelization pattern in both OpenVDB and NanoVDB distributes leaf nodes across threads and processes their active voxels in a single-threaded iteration. On the GPU, the canonical pattern launches kernels over leaf nodes, with an entire thread block to process all 8^3 voxels of a leaf in parallel, active or not. The load balance difficulty is clear: any warp mapped to a 32-tuple of

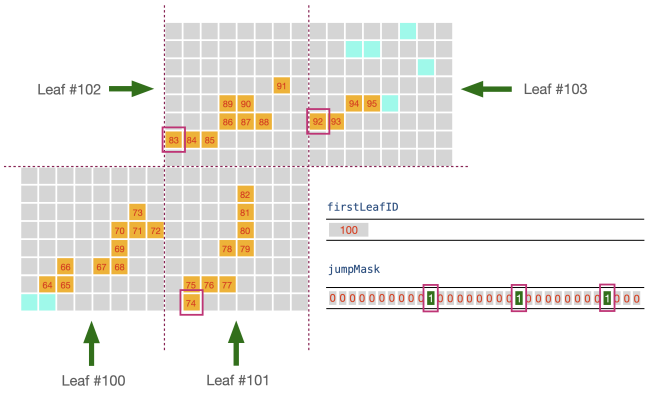


Fig. 7. **VBM metadata for a W=32 block.** The 32 active voxels at sequential indices 64 . . . 95 (orange) span four leaf nodes of the underlying Index-Grid; a few active voxels with indices outside this block appear in cyan. The metadata for the block reduces to a single `firstLeafID` = 100 (the leaf containing the first voxel in the block) plus a 32-bit `jumpMap` whose set bits at positions 10, 19, and 28 mark the intra-block offsets at which each subsequent leaf begins.

voxels that are not all active wastes SIMT throughput on the inactive fraction. The same argument applies to the CPU; the loss is simply more evident on the GPU, where SIMT computation is the default, whereas explicit SIMD on the CPU is typically an intentional, targeted optimization.

We introduce *voxel-sequential access*, a linear iteration paradigm over only active voxels, in the order that Index-Grid defines. An Index-Grid is functionally a *forward map* from a Cartesian voxel coordinate (i, j, k) to its sequential active-voxel index; voxel-sequential access requires the *inverse* of this map, which VDB has not previously provided.. We introduce a new NanoVDB acceleration structure, the *Voxel Block Manager* (VBM) to encode this inverse map: given the sequential index of an active voxel, it recovers the voxel's position in the tree—its containing leaf and in-leaf offset—and by extension its Cartesian coordinate. This section defines the structure, highlights its very low storage footprint, and illustrates its use in voxel-sequential streaming and stencil operations.

4.1 VoxelBlockManager metadata and initialization

The VBM is parameterized by a compile-time *block width* $W \geq 64$, taken to be a multiple of 64. Conceptually, W is the width of the voxel interval we operate on in a data-parallel fashion. On the GPU it is typically chosen to match the thread block size of the operation being implemented ($W = 128$ or $W = 256$ in most of our examples). We partition the sequence of active voxels into aligned intervals of size W, which we call VBM *blocks*. For each block, the VBM stores two metadata items (Figure 7):

- `firstLeafID`: a `uint32_t` per block, storing the index of the leaf node containing the block's first active voxel.
- `jumpMap`: one bit per active voxel, packed into `uint64_t` words ($W/64$ per block). Bit p ($p > 0$) is set if the active voxel at intra-block position p is the *first* active voxel of a new leaf node.

Together these consume at most 1.5 bits per active voxel, independent of grid size, payload type, or leaf occupancy. NanoVDB Index-Grids guarantee that the active voxels within a single leaf node occupy a contiguous index interval. The semantics of the VBM implicitly assume two further properties:

- No leaf nodes have zero active voxels (such a leaf node would be functionally irrelevant to the coordinate-to-index map the grid defines).
- Active voxel indexing is contiguous across leaf nodes: if K is the index of the last active voxel in leaf node L, the first active voxel in leaf node $L + 1$ has index $K + 1$.

Although neither property is stipulated by the core NanoVDB data structure, both are de facto enforced by all current NanoVDB tools that generate Index-Grids. Given their utility in enabling the VBM, we mandate them as invariants in applications that use it.

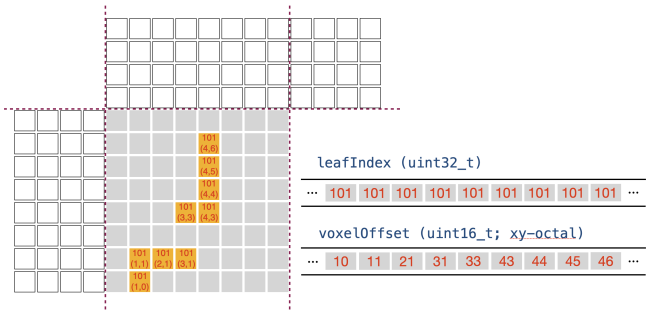
Finally, the `firstLeafID` and `jumpMap` arrays are computed in parallel. For each leaf we identify all overlapping *blocks*. Blocks overlapping a strict prefix of the leaf's voxel interval have the leaf's starting position marked in their `jumpMap` via `atomicOr`; blocks beginning inside the leaf have their `firstLeafID` set to its index. Construction cost is typically sub-millisecond for Index-Grids with up to 10^7 active voxels.

4.2 VBM Decoding in stencil-based kernels

A kernel using the VBM is launched with one W-thread CUDA block per VBM block. Every such kernel begins by decoding the VBM metadata at thread block granularity, consuming the block's `firstLeafID` entry and `jumpMap` row and writing two arrays to shared memory:

- `leafIndex[W]`: the index of the leaf containing each thread's active voxel.
- `voxelOffset[W]`: the local (Cartesian) offset of that voxel within its leaf, encoded as a `uint16_t` in xyz-octal form (three bits per axis).

A section of the decoded arrays, corresponding to Leaf #101 from Figure 7 is illustrated below:



Decoding operates as follows. The number of leaf nodes overlapping the current block is the bit population count of the `jumpMap`, plus one. For each such leaf we visit all 512 (8^3) voxels in parallel and probe whether each is active and, if so, what linear index is – both fundamental NanoVDB operations, accelerated via built-in `popcnt` instructions on the GPU. Active voxels update the corresponding

entries of `leafIndex` and `voxelOffset` in shared memory. The operation requires no delicate synchronization and incurs no global memory traffic, so for all but the most extreme leaf-level sparsity (> 95% inactive voxels per leaf node), its cost is substantially lower than that of accessing the sidecar values themselves.

Stencil-tap Optimization. For stencil applications we employ a further optimization that mitigates the cost of computing linear indices for neighboring voxels. For common, narrowly-supported stencils, the number of leaf nodes a stencil can touch is substantially lower than its number of taps. The fifth-order Hamilton-Jacobi WENO stencil used in our fluid simulation examples (section 5.2) has 19 taps, at grid offsets $(-3 \ldots 3, 0, 0)$, $(0, -3 \ldots 3, 0)$, and $(0, 0, -3 \ldots 3)$, yet can touch at most *four* leaf nodes (the one containing the stencil center, and at most three more). We identify the required neighbors and fetch them in bulk via NanoVDB's `probeLeaf`; the indices of all 19 stencil taps are then computed with fully warp-synchronous, non-divergent operations. Similar optimizations apply to other stencil types; for example a $3 \times 3 \times 3$ convolution stencil for machine learning workloads touches at most 7 neighboring leaf nodes (typically far fewer), to compute all 27 taps.

5 Applications

The dynamic topology operators (section 3) and the voxel-sequential access paradigm (VBM, section 4) are general-purpose primitives that compose into algorithms spanning a wide variety of real-world domains. We demonstrate this with applications from 3D perception and reality capture (section 5.1), fluid and gas simulation (section 5.2), and large-scale shape voxelization and rendering (section 5.3). Furthermore, the algorithms built from these compositions are often more memory-efficient and performant than their counterparts in the literature. The details of the hardware used for our experiments can be found in Appendix A.

5.1 Perception and Reality Capture

Together, the topology operators of section 3 and the voxel-sequential paradigm of section 4 undergird an implementation of established reality-capture techniques: truncated signed distance fields (TSDF) for surface reconstruction [Curless and Levoy 1996; Newcombe et al. 2011], log-odds occupancy for path planning [Hornung et al. 2013], the Euclidean signed distance field (ESDF) for collision checking [Han et al. 2019; Oleynikova et al. 2017], and dynamic-scene decay to filter moving objects. Each operation is an integrator expressed in a few hundred lines of C++/CUDA, with per-voxel state held in PyTorch tensors aligned to the linear order of each grid's active voxels. We present TSDF fusion as a representative composition and cover the rest in Appendix B.

TSDF integration as a representative composition. We integrate a depth or LiDAR observation Z at time t into a grid G_{t-1} with per-voxel weights w_{t-1} and signed distances d_{t-1} using `voxelsToGrid`, `mergeGrids`, and `inject` from section 3 together with the per-ray HDDA traversal [Museth 2014; Williams et al. 2024]. The depth case carries a camera projection Π; the LiDAR case carries a sensor origin o. A voxel handle v has world position $x_v = \textsc{worldCoord}(v)$. The truncated-SDF helper $\textsc{truncSDF}(d, \tau)$ returns d/τ when $|d| \le \tau$, $+1$ when $d > \tau$ (free space ahead of the surface), and the sentinel *skip* when $d < -\tau$ or the underlying sample is undefined (e.g. the voxel projects outside the depth frame), in which case the voxel keeps its injected old value. The per-observation weight w_{obs} defaults to 1 (the depth integrator optionally accepts a per-pixel weight image).

Algorithm 1 shows the resulting integration. Lines 1–3 are shared across modalities: they build a shell of voxels within τ of the new observation, union it into a fresh grid G_t, and inject the old sidecars onto G_t, zeroing the newly created voxels. The update phase differs by modality. *Depth* dispatches one thread per voxel of *shell* and writes $(d_t[v], w_t[v])$ in place with a weighted running average—no atomics because each voxel has a single writer. *LiDAR* dispatches one thread per input point, walks the active voxels along the sensor ray $\overline{op}$ via HDDA, and accumulates into the running-sum sidecars $(s_d, s_w) = (d_t \cdot w_t,\ w_t)$ via `atomicAdd`; a final $d_t \leftarrow s_d/s_w$ recovers the running average. The same scaffold powers occupancy mapping (with a Bayesian log-odds add as the per-voxel update) and the dilated-grid stage of ESDF; dynamic-scene decay is `pruneGrid` plus `inject` with no integrator-side code at all (Appendix B).

ALGORITHM 1: TSDF integration of a single observation. Lines 1–3 are the shared topology+inject scaffold (`voxelsToGrid`, `mergeGrids`, `inject`). $\textsc{worldPoints}(Z)$ is depth-back-projection through Π or the identity for LiDAR. The update phase picks an iteration pattern to match the data flow: depth iterates voxels (in-place avg), LiDAR iterates rays (atomic running-sum then normalize). Occupancy reuses lines 1–3 with a Bayesian log-odds per-voxel update; ESDF reuses the scaffold with `dilateGrid` for `voxelsToGrid` (Appendix B).

Input: Persistent $(G_{t-1}, d_{t-1}, w_{t-1})$; observation Z with projection Π (depth) *or* sensor origin o (LiDAR); truncation τ; per-observation weight w_{obs}.
Output: (G_t, d_t, w_t).

```
/* Topology grow + inject (shared across modalities) */
1  shell ← voxelsToGrid(worldPoints(Z) ⊕ ball(τ))
2  G_t ← mergeGrids(G_{t−1}, shell)
3  (d_t, w_t) ← inject(G_{t−1} → G_t, (d_{t−1}, w_{t−1}), init=0)
4  if Z is a depth image then // per-voxel dispatch; no atomics
5      foreach v ∈ shell in parallel do
6          sdf_v ← truncSDF(Z[Π(x_v)] − z_cam(x_v), τ)
7          if sdf_v ≠ skip then (d_t[v], w_t[v]) ←
             weightedAvg(d_t[v], w_t[v], sdf_v, w_obs)
8      end
9  else // per-ray HDDA; atomic running-sums
10     (s_d, s_w) ← (d_t · w_t, w_t)        // transient accumulators
11     foreach p ∈ Z in parallel do
12         foreach v on HDDA(op, G_t) do
13             sdf_v ← truncSDF(‖p − o‖ − ‖x_v − o‖, τ)
14             if sdf_v ≠ skip then
                 atomicAdd(s_d[v], w_obs sdf_v), atomicAdd(s_w[v], w_obs)
15         end
16     end
17     (d_t, w_t) ← (s_d/s_w, s_w)     // normalize sums to running
         average
18 end
19 return (G_t, d_t, w_t)
```

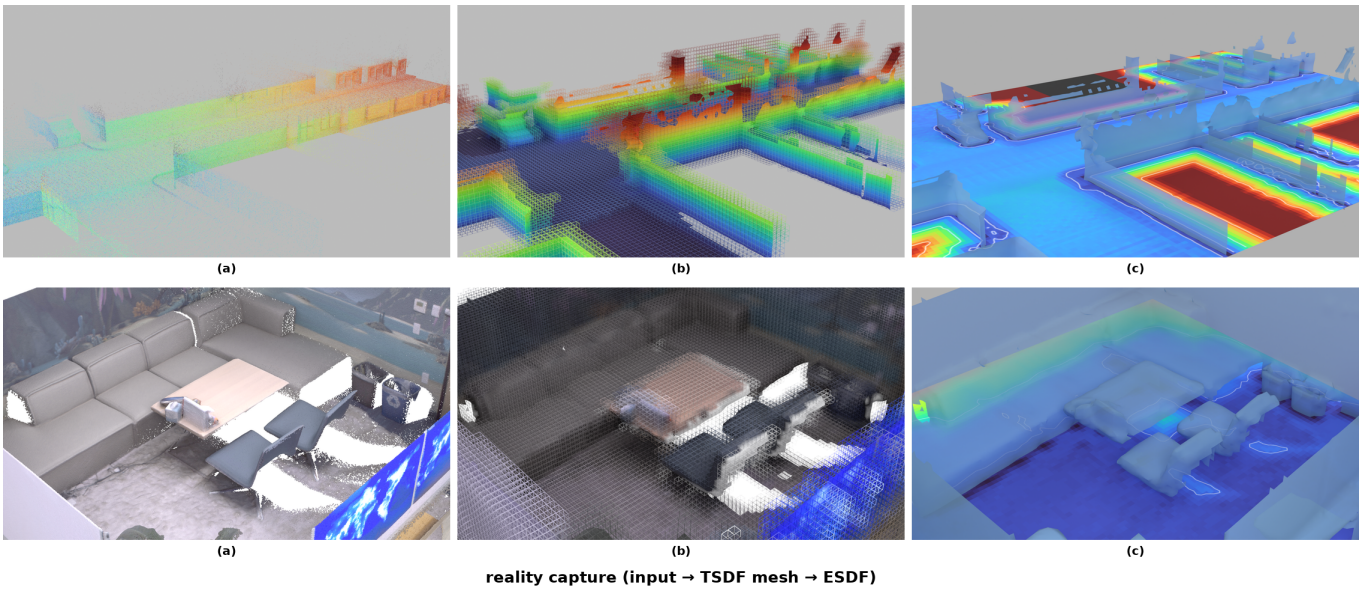


Fig. 8. Starting from an input LiDAR sweep or a sequence of depth images, we fuse a TSDF surface mesh and subsequently derive a Euclidean signed distance field (ESDF) to facilitate collision checking and path planning. Mai City synthetic LiDAR (top row, voxel size 0.33 m) and Replica OFFICE0 RGB-D (bottom row, voxel size 0.05 m), with (a) input sensor sweeps coloured by time (Mai City) and by source-pixel RGB (Replica), (b) fused TSDF mesh with the underlying voxel grid drawn as a faint wireframe scaffold, and (c) the mesh rendered translucently over a horizontal slice of the derived ESDF (turbo colormap, red = far from any obstacle).

Datasets and Results. We benchmark against nvblox [Millane et al. 2024], Open3D [Zhou et al. 2018], and VDBFusion [Vizzo et al. 2022] on four datasets: Mai City [Vizzo et al. 2021] synthetic Velodyne LiDAR (700-frame trajectory), Replica [Straub et al. 2019] depth renderings produced by NICE-SLAM [Zhu et al. 2022] (8-scene, 2000 frames each), KITTI Odometry [Geiger et al. 2013, 2012] real-sensor LiDAR (sequences 00, 02, 05; 2.2–5.1 km), and 7-Scenes [Shotton et al. 2013] RGB-D for long-trajectory persistence. VDBFusion comparisons run CPU-only (OpenVDB) and are provided for completeness and qualitative analysis. All measurements use truncation τ=3× voxel size and ESDF max-distance R=10× voxel size throughout; nvblox's depth and LiDAR integration distances are configured to match ours rather than the upstream defaults. Table 1 reports the finest voxel each system completes; per-voxel-size timings, the full nvblox-config disclosure, and qualitative mesh comparisons (fig. 13) are in Appendix B.

- *Scale.* At equal memory budget, our implementations reach voxel resolutions 1.5–5× finer than the block-hash baselines, with active voxel counts up to two orders of magnitude larger. The strongest single point is *1.67× 10^9 ESDF voxels in 38.6 GB on KITTI seq 00 at 2 cm*, where nvblox runs out of memory.
- *Speed.* On occupancy across the 20–1.5 cm sweep we are 3.5–11.5× faster than nvblox; on LiDAR TSDF we are 3.6–5.7× faster on Mai City and 4.8–5.3× faster on KITTI at 40 cm, narrowing to 2.6–3.5× at 20 cm before nvblox exhausts memory at 10 cm. Open3D leads depth-TSDF throughput on Replica by 11–46× at 20–5 mm before failing at 4 mm (see Appendix B).
- *Composability.* Every integrator decomposes into the same three kinds of step: dynamic topology operators, custom CUDA kernels (per-leaf-slot, per-ray HDDA, or VBM), and PyTorch operations on sidecar tensors. Dynamic-scene decay is the limit case – two operators, no integrator code – with a measured *78% reduction in persistent voxels* on KITTI seq 00 over 500 frames, as moving traffic stops accumulating in the persistent grid.

Table 1. **Voxel-size scale ceiling on a single 48 GB RTX 6000 Ada GPU.** Finest voxel each system completes; ours reaches the listed active voxel count and peak GPU memory at its finest voxel. Open3D has no LiDAR TSDF and no ESDF integrator. †Open3D succeeds on 6/8 Replica scenes at 5 mm and crashes (`illegal memory access`) at 3 mm on all eight; we list 5 mm as its finest universal ceiling. Per-voxel-size timing tables and the full nvblox-config are in Appendix B.

Workload	nvblox	Open3D	Ours	Voxels	GB
Mai City 700 f LiDAR TSDF	10 cm	n/a	**1.25 cm**	938.7 M	37.1
Mai City 700 f TSDF → ESDF	10 cm	n/a	**3 cm**	1.28 B	30.6
Replica 8×2000 f TSDF	10 mm	5 mm†	**3 mm**	109 M	10.3
Replica 8×2000 f ESDF	10 mm	n/a	**3 mm**	438 M	10.3
KITTI seq 02 4661 f TSDF	20 cm	n/a	**10 cm**	427 M	19.8
KITTI seq 00 100 f ESDF	5 cm	n/a	**2 cm**	1.67 B	38.6
KITTI seq 05 100 f Occupancy	1.5 cm	n/a	**5 mm**	2.35 B	44.1

5.2 Fluid and Gas Simulation

We include micro-benchmarks from physics-based simulations of fire (Figure 10), basing the combustion simulation on the techniques

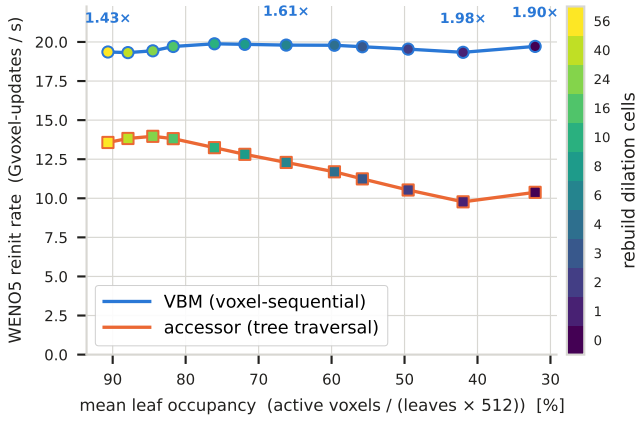


Fig. 9. WENO5 narrow-band reinitialization throughput for two dispatch policies—VBM (voxel-sequential, one thread per active voxel) and the accessor baseline (tree traversal, one thread per leaf slot)—against mean leaf occupancy ≡ `activeVoxels/(nLeaves · 512)` within active 8^3 leaf tiles, decreasing left to right. Timing is per-call time of one inner WENO5 iteration. VBM holds near 20 Gvoxel-updates/s, its launch sized by the active set alone, while the accessor falls from 14.0 to 9.8 as more threads land on inactive slots; VBM speedup reaches 1.98×, rising as occupancy drops. Marker color denotes rebuild dilation width. Hardware details in Appendix A.

detailed in [Hong et al. 2007; Nguyen et al. 2002, 2001], which benefit from our novel Voxel Block Manager and efficient topological operators. The ghost fluid method employs a narrow-band level set representation—a truncated signed distance field (SDF)—to track the combustion flame front, or "blue core". This approach treats the combustion region as a discrete interface across which chemical properties exhibit jump conditions. In our implementation, the interface between fuel and air is rasterized into the NanoVDB sparse volumetric field, and dilated outwards from the surface crossing voxels to create a narrow band of active voxels. The Eikonal equation is iteratively solved over just this sparse narrow band using a stencil operation that visits ±3 voxels in each of the 3 cardinal directions, for a total of 18 adjacent voxel visits in addition to the one central voxel visit. The discretization of the Euler equation is achieved with the optimally fifth-order accurate WENO scheme [Liu et al. 1994], which is both computationally and arithmetically complex.

Level set tracking. The flame front is represented implicitly as the zero contour of a signed-distance field ϕ, stored on a narrow band of active voxels around the interface. Each timestep advects ϕ by the flame-front velocity (derived from first-order detonation shock dynamics (DSD) [Hong et al. 2007]), then reinitializes it back to a signed-distance function so the contour itself stays sharp while ϕ remains usable elsewhere as a smooth distance. Because the field is sparse, the active set has to move with the interface: each step (i) dilates the band so the next advect has source voxels to read from, (ii) semi-Lagrangian advects ϕ along w, (iii) reinitializes ϕ inside the band with a few iterations of a WENO-discretized Hamilton–Jacobi update [Liu et al. 1994], and (iv) prunes voxels whose $|\phi|$ has grown past a band-width threshold. The microbenchmarks below isolate three operators from this pipeline—reinit, dilate, and prune.

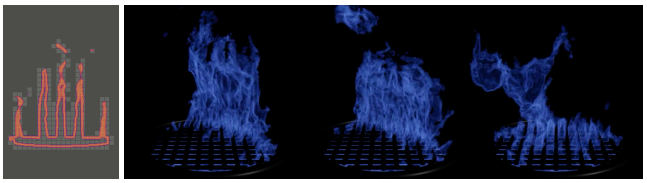


Fig. 10. Left: Flame interface narrow-band. Right: Dynamic fire sim

Reinitialization micro-benchmark. The reinit kernel implements WENO5 on top of the new Voxel Block Manager (VBM) interface, and is compared against an accessor-only baseline that represents best-practice NanoVDB usage today. Each WENO5 update consumes a 19-point cross stencil per active voxel (center, plus R=3 taps along each of $\pm x$, $\pm y$, $\pm z$); the two implementations differ in how they fetch those off-center taps. The sparse grid stores active voxels in 8^3 tiles, so an off-center tap usually lands in the same tile as the center but occasionally crosses into a neighboring one. The accessor variant launches one CUDA block per tile whose threads walk the tree for every off-center tap, employing a small tree-walk helper that caches the last-touched tile, so consecutive within-tile taps are O(1) but every tile-boundary tap re-walks the tree. The VBM variant amortizes that bookkeeping across a 128-thread cluster of consecutive active voxels: the cluster cooperatively resolves its 128 tile addresses into shared memory once, and because R=3 is strictly less than the 8-voxel tile width, a single neighboring-tile pointer per axis covers every off-center tap any thread in the cluster can reach. The 19 stencil reads then compile to direct array loads against four resident tile pointers, with no per-tap tree walk. Figure 9 plots the throughput of the VBM-based WENO5 iteration against the accessor baseline as a function of mean leaf occupancy. VBM's throughput is essentially constant and up to 1.98× that of the accessor implementation, whose throughput decreases as occupancy decreases and a growing fraction of its threads land on inactive slots.

Topological operators micro-benchmark. The fluid simulation workload leverages the newly introduced `DilateGrid` and `PruneGrid` dynamic topology operators. In the level set pipeline dilate grows the active set by one voxel each step (we use the 26-neighbor face-edge-vertex pattern, so the advect's trilinear backtrace always lands on a source voxel), and prune is the inverse, dropping voxels whose $|\phi|$ has grown past the band-width threshold so the band can't grow unbounded across time steps. As Figure 11 shows, NanoVDB dilates a 567 M active-voxel topology in 2.0 ms using a pure topological mutation that touches no field storage. To match this throughput, a 32-bit field-carrying equivalent (OpenVDB) would have to read and rewrite $4(N_{in} + N_{out}) \approx 4.7$ GB of field data. At the GPU's 1792 GB/s memory peak (Appendix A) that traffic alone takes 2.63 ms—longer than the 2.0 ms the topological formulation actually takes, so no field-carrying implementation can match it at any efficiency. Prune completes the same grid in 8.4 ms; the bandwidth argument does not extend to it, as its benchmark includes the predicate-eval read of the underlying field. Importantly, we compare dilation against a `voxelsToGrid`-based rebuild that drops every stencil tap that falls on an already-active voxel—about 13× faster than rebuilding without tap elision, which itself runs out of memory above 69.5 M active

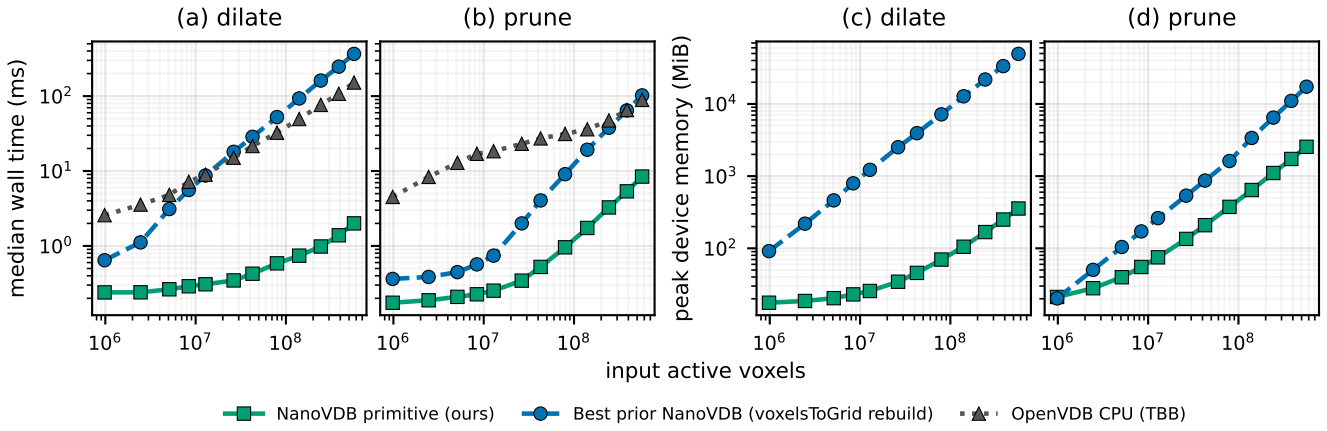


Fig. 11. Dilate (26-connected) and Prune ($|\phi| \leq 1$) against the best prior NanoVDB route—rebuild from active coordinates through `voxelsToGrid`—and OpenVDB's TBB-parallel host implementation. Both operators run on identical input topologies: a flame front level set at 12 resolutions, $128 \times 192 \times 128$ to $1792 \times 2688 \times 1792$, 0.98 M to 567 M active voxels. Axes are logarithmic, so the margins are wider than they look. At the largest input dilate is 183× faster than the rebuild and 76× faster than the host, in 139× less device memory (356 MiB against 48.2 GiB); prune is 12.1× and 10.6× faster, in 7× less (2.49 GiB against 16.9 GiB). (a), (b): median wall time per call. (c), (d): peak device memory (the omitted host series has no device allocation).

voxels. Even so, our dilation is 183× faster and peaks at 356 MiB against the baseline's 48.2 GiB of device memory, a 139× difference and half the card's memory.

5.3 Large-Scale Shape Voxelization

We demonstrate one further application of our topological pipeline: transforming a triangle mesh into a rasterized, sparse Unsigned Distance Field (UDF). Although distinct from the topological operators of section 3, it constructively interweaves with the pipeline of fig. 4. Since our objective is a narrow-band UDF, we use *intersection* between a triangle and a volume primitive to indicate intersection with a thickness equal to the narrow band half-width.

We first stream through the triangle list, identifying how many root tiles (4096^3 voxels each) each triangle interacts with. Deduplicating these root tiles seeds steps 1–2 of the topological pipeline. A hierarchical broad-phase intersection progressively refines the *(triangle,rootTile)* pairs into *(triangle,leafNode)* pairs, where each leafNode spans 8^3 voxels. This list marks the lower and leaf node bitmaps in Step 3; steps 4–5 then build the sparse tree that will hold the voxelized representation, with no voxels yet marked active. Step 6 visits the 8^3 voxels of each leaf appearing in a *(triangle,leafNode)* pair with a 512-thread CUDA block, marking active every voxel within the narrow-band half-width of the triangle. With the topology constructed, we allocate the UDF sidecar at exactly the storage the narrow band requires. Finally, we resolve the *(triangle,leafNode)* list once more and perform a *minimum reduction* in the sidecar, preserving at every voxel the smallest distance across all nearby triangles.

We evaluate the construction on the intricate "hairball" triangle mesh [McGuire 2017] shown in fig. 12, consisting of 2.8 million triangles. The mesh is rasterized into a NanoVDB index grid with UDF values stored in a separate sidecar array. At voxel size 0.003 and narrow-band half-width 2, OpenVDB's `meshToUDF` produced 1.025×10^9 active voxels in 23.87 seconds, while our `MeshToGrid` implementation produced 1.026×10^9 active voxels in 2.30 seconds—a roughly 10× speedup at comparable resolution. At voxel size 0.001 with the same narrow-band width, our implementation generated 9.88×10^9 active voxels in 3.24 seconds; OpenVDB did not complete, terminating on memory exhaustion. These results highlight the ability of NanoVDB's dynamic sparse-grid construction to capture fine geometric detail at multi-billion-voxel scale while maintaining interactive construction times. For visualization, we render the sparse grid directly with the hierarchical digital differential analyzer (HDDA) in the NanoVDB library[Museth 2014]. Rays traverse the sparse index-grid topology, and surface hits are identified by detecting UDF iso-crossings along the traversed active voxels, so the multi-billion-voxel result is visualized without conversion to a dense representation.

6 Limitations and Future Work

Topological Operations. Although our topological operation framework can easily accommodate larger dilation stencils (as long as their radius does not exceed the size of leaf nodes, i.e 8 voxels, to avoid modifications to other "generic" components of the pipeline), stencils that have substantially more irregular (non-rectangular, non-symmetric) shapes might offer fewer opportunities for the aggressive bit-parallel optimizations we employed. We note however that *box* stencils of radius up to 8 can be trivially accommodated with bit-parallel operations. Although we have included details on the implementation of morphological dilation, we have yet to implement its inverse operator, namely, morphological erosion. We plan to achieve this in terms of the existing prune operator. We should mention that our scratchpad-based bookkeeping commits to allocating approximately 16MB of storage for every 4096^3 root tile that potentially overlaps with the output of our operation. Although it is unlikely that this limitation will prevent us from scaling up (the data payload would be substantially larger), it could potentially limit

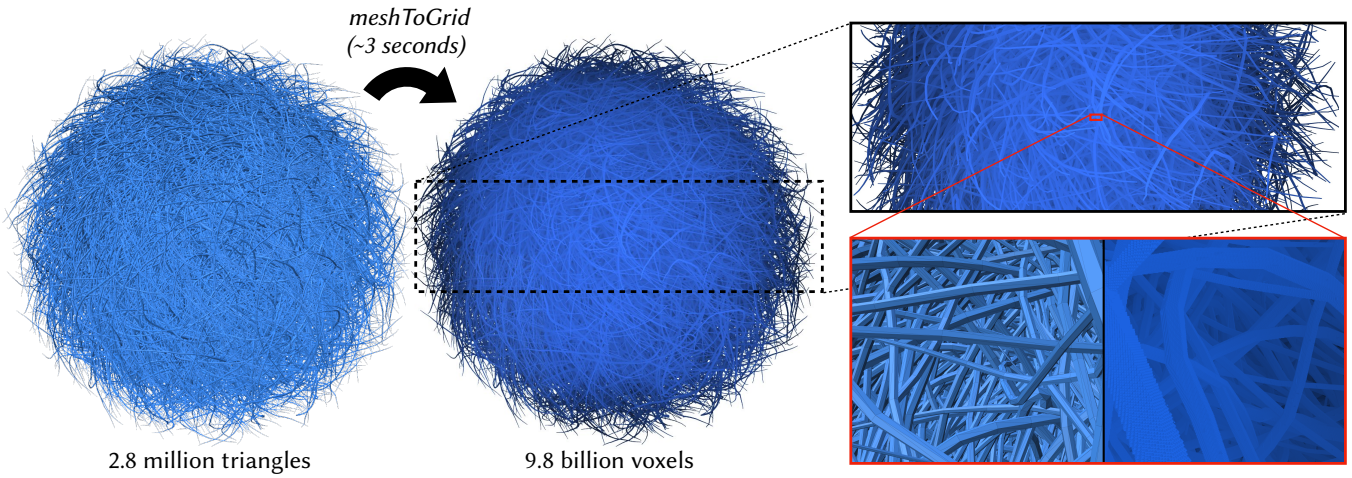


Fig. 12. Large-scale NanoVDB voxelization of a 2.8M-triangle hairball mesh [McGuire 2017]. At voxel size 0.001 and narrow-band half-width 2, our `MeshToGrid` implementation generates a sparse index grid with 9.88B active voxels and UDF values stored in a sidecar in 3.24s; the result is rendered directly with NanoVDB's hierarchical digital differential analyzer (HDDA). Hardware details in Appendix A.

our performance advantage against reconstruction-from-scratch approaches for domains with very small size.

Voxel-sequential processing. The key benefit of the Voxel-sequential access paradigm (via VBM) is maximization of compute bandwidth, by ensuring that data-parallel processors (e.g. GPUs) saturate execution units with regular work on *active* voxels. This benefit is diminished in domains where leaf nodes are mostly dense, and those are also examples of applications where "traditional" NanoVDB grids (leaf-level dense) might be more appropriate than sparse Index-Grids. The benefit of maximizing compute density and saturation is also more prominent on compute bound, stencil-based operations; the WENO stencil in our fluid simulation examples, or sparse convolutions with deep vector-valued features (reducing to GEMM operations) would be best examples of such compute-bound workloads that stand to gain the most out of VBM. Memory-bound workloads (e.g. very small stencils on scalar fields, like a gradient or divergence Finite Difference operator) would struggle to see substantial benefits. Finally, workloads that access sparse data in spatially coherent (but not streaming/sequential) patterns, would remain closer to random-access tools (e.g. accessors) as the paradigm of choice.

Cross-resolution operators. The coarsening and refinement operators share the same processing pipeline and building blocks as dilation, merging, and pruning; nevertheless, they are not exercised end-to-end by any of the applications of section 5. We also acknowledge that on an application-specific basis they may need to be paired with additional sidecar manipulation tools, beyond what we presented here. For example, one possible application would be the construction of a pyramid of sparse grids for multiresolution solvers, in the spirit of Setaluri et al. [2014]. Such a workflow would also require a cross-resolution analogue of `inject`: whereas our present sidecar injection relates voxels across two topologies by coordinate identity, a 2:1 transfer is a reduction in one direction and an interpolation in the other, calling for an upsample/downsample (prolongation/restriction) operator optimized for sparse GPU computation. We defer these application-specific extensions to future work.

Code Availability

Our improvements to NanoVDB and the VoxelBlockManager implementation are made publicly available and open source as part of the Academy Software Foundation's OpenVDB Repository which can be found at *https://github.com/AcademySoftwareFoundation/openvdb*. The code referenced in section 5.1 and Appendix B are released in the same manner but as part of the fVDB Repository which can be found at *https://github.com/openvdb/fvdb-core*.

References

Bernhard Braun, Jan Bender, and Nils Thuerey. 2025. Adaptive Phase-Field-FLIP for Very Large Scale Two-Phase Fluid Simulation. *ACM Trans. Graph.* 44, 4, Article 42 (July 2025), 23 pages. doi:10.1145/3730854

Christopher Choy, JunYoung Gwak, and Silvio Savarese. 2019. 4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 3075–3084.

Brian Curless and Marc Levoy. 1996. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 303–312.

Lorenzo De Rebotti, Emanuele Giacomini, Giorgio Grisetti, and Luca Di Giammarino. 2025. Resolution where it counts: hash-based GPU-accelerated 3D reconstruction via variance-adaptive voxel grids. *ACM Transactions on Graphics* 45, 2 (2025), 1–15.

Wei Dong, Yixing Lao, Michael Kaess, and Vladlen Koltun. 2023. ASH: A Modern Framework for Parallel Spatial Hashing in 3D Perception. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45, 5 (2023), 5417–5435. doi:10.1109/TPAMI.2022.3214347

Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. 2013. Vision meets robotics: The kitti dataset. *The international journal of robotics research* 32, 11 (2013), 1231–1237.

Andreas Geiger, Philip Lenz, and Raquel Urtasun. 2012. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE conference on computer vision and pattern recognition*. IEEE, 3354–3361.

Luxin Han, Fei Gao, Boyu Zhou, and Shaojie Shen. 2019. Fiesta: Fast incremental euclidean distance fields for online motion planning of aerial robots. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 4423–4430.

Jeong-Mo Hong, Tamar Shinar, and Ronald Fedkiw. 2007. Wrinkled flames and cellular patterns. In *ACM SIGGRAPH 2007 Papers* (San Diego, California) *(SIGGRAPH '07)*. Association for Computing Machinery, New York, NY, USA, 47–es. doi:10.1145/1275808.1276436

Armin Hornung, Kai M Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. 2013. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous robots* 34, 3 (2013), 189–206.

Ben Houston, Michael B. Nielsen, Christopher Batty, Ola Nilsson, and Ken Museth. 2006. Hierarchical RLE level set: A compact and versatile deformable surface representation. *ACM Trans. Graph.* 25, 1 (Jan. 2006), 151–175. doi:10.1145/1122501.1122508

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Transactions on Graphics (SIGGRAPH Asia)* 38, 6 (2019), 1–16. doi:10.1145/3355089.3356506

Jiahui Huang, Zan Gojcic, Matan Atzmon, Or Litany, Sanja Fidler, and Francis Williams. 2023. Neural Kernel Surface Reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 4369–4379.

Krishna Murthy Jatavallabhula, Edward Smith, Jean-Francois Lafleche, Clement Fuji Tsang, Artem Rozantsev, Wenzheng Chen, Tommy Xiang, Rev Lebeau, and Sanja Fidler. 2019. Kaolin: A PyTorch Library for Accelerating 3D Deep Learning Research. arXiv:1911.05063 [cs.LG] https://arxiv.org/abs/1911.05063

Doyub Kim, Minjae Lee, and Ken Museth. 2022. NeuralVDB: High-Resolution Sparse Volume Representation using Hierarchical Neural Networks. *ACM Transactions on Graphics (SIGGRAPH)* (2022). arXiv:2208.04448 [cs.GR] https://arxiv.org/abs/2208.04448

A.E. Lefohn, J.M. Kniss, C.D. Hansen, and R.T. Whitaker. 2003. Interactive deformation and visualization of level set surfaces using graphics hardware. In *IEEE Visualization, 2003. VIS 2003.* 75–82. doi:10.1109/VISUAL.2003.1250357

Xu-Dong Liu, Stanley Osher, and Tony Chan. 1994. Weighted Essentially Non-oscillatory Schemes. *J. Comput. Phys.* 115, 1 (1994), 200–212. doi:10.1006/jcph.1994.1187

Frank Losasso, Frédéric Gibou, and Ron Fedkiw. 2004. Simulating water and smoke with an octree data structure. *ACM Trans. Graph.* 23, 3 (Aug. 2004), 457–462. doi:10.1145/1015706.1015745

Morgan McGuire. 2017. *Computer Graphics Archive*. https://casual-effects.com/data

Alexander Millane, Helen Oleynikova, Emilie Wirbel, Remo Steiner, Vikram Ramasamy, David Tingdahl, and Roland Siegwart. 2024. nvblox: GPU-Accelerated Incremental Signed Distance Field Mapping. arXiv:2311.00626 [cs.RO] https://arxiv.org/abs/2311.00626

Chohong Min. 2004. Local level set method in high dimension and codimension. *J. Comput. Phys.* 200, 1 (2004), 368–382. doi:10.1016/j.jcp.2004.04.019

Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. 2022. Instant Neural Graphics Primitives with a Multiresolution Hash Encoding. *ACM Transactions on Graphics (SIGGRAPH)* 41, 4 (2022), 102:1–102:15. doi:10.1145/3528223.3530127

Ken Museth. 2013. VDB: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph.* 32, 3, Article 27 (July 2013), 22 pages. doi:10.1145/2487228.2487235

Ken Museth. 2014. Hierarchical digital differential analyzer for efficient ray-marching in OpenVDB. In *ACM SIGGRAPH 2014 Talks* (Vancouver, Canada) *(SIGGRAPH '14)*. Association for Computing Machinery, New York, NY, USA, Article 40, 1 pages. doi:10.1145/2614106.2614136

Ken Museth. 2021. NanoVDB: A GPU-Friendly Data Structure for Real-Time Volumetric Effects. *ACM SIGGRAPH 2021 Talks* (2021). doi:10.1145/3450623.3464653

Richard A Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. 2011. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE international symposium on mixed and augmented reality*. Ieee, 127–136.

Duc Quang Nguyen, Ronald Fedkiw, and Henrik Wann Jensen. 2002. Physically based modeling and animation of fire. *ACM Trans. Graph.* 21, 3 (July 2002), 721–728. doi:10.1145/566654.566643

Duc Q. Nguyen, Ronald P. Fedkiw, and Myungjoo Kang. 2001. A Boundary Condition Capturing Method for Incompressible Flame Discontinuities. *J. Comput. Phys.* 172, 1 (2001), 71–98. doi:10.1006/jcph.2001.6812

Michael Nielsen and Ken Museth. 2006. Dynamic Tubular Grid: An Efficient Data Structure and Algorithms for High Resolution Level Sets. *J. Sci. Comput.* 26 (03 2006), 261–299. doi:10.1007/s10915-005-9062-8

Michael B. Nielsen, Ola Nilsson, Andreas Söderström, and Ken Museth. 2007. Out-of-core and compressed level set methods. *ACM Trans. Graph.* 26, 4 (Oct. 2007), 16–es. doi:10.1145/1289603.1289607

Matthias Nießner, Michael Zollhöfer, Shahram Izadi, and Marc Stamminger. 2013. Real-time 3D reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (ToG)* 32, 6 (2013), 1–11.

Helen Oleynikova, Zachary Taylor, Marius Fehr, Roland Siegwart, and Juan Nieto. 2017. Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 1366–1373.

Xuanchi Ren, Jiahui Huang, Xiaohui Zeng, Ken Museth, Sanja Fidler, and Francis Williams. 2024. XCube: Large-Scale 3D Generative Modeling using Sparse Voxel Hierarchies. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Rajsekhar Setaluri, Mridul Aanjaneya, Sean Bauer, and Eftychios Sifakis. 2014. SPGrid: a sparse paged grid structure applied to adaptive smoke simulation. *ACM Trans. Graph.* 33, 6, Article 205 (Nov. 2014), 12 pages. doi:10.1145/2661229.2661269

Jamie Shotton, Ben Glocker, Christopher Zach, Shahram Izadi, Antonio Criminisi, and Andrew Fitzgibbon. 2013. Scene coordinate regression forests for camera relocalization in RGB-D images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2930–2937.

Spconv Contributors. 2022. spconv: Spatially Sparse Convolution Library. https://github.com/traveller59/spconv.

John Strain. 1999. Tree Methods for Moving Interfaces. *J. Comput. Phys.* 151, 2 (1999), 616–648. doi:10.1006/jcph.1999.6205

Julian Straub, Thomas Whelan, Lingni Ma, Yufan Chen, Erik Wijmans, Simon Green, Jakob J Engel, Raul Mur-Artal, Carl Ren, Shobhit Verma, et al. 2019. The replica dataset: A digital replica of indoor spaces. *arXiv preprint arXiv:1906.05797* (2019).

Haotian Tang, Zhijian Liu, Xiuyu Li, Yujun Lin, and Song Han. 2022. TorchSparse: Efficient Point Cloud Inference Engine. In *Proceedings of Machine Learning and Systems (MLSys)*.

Haotian Tang, Shang Yang, Zhijian Liu, Ke Hong, Zhongming Yu, Xiuyu Li, Guohao Dai, Yu Wang, and Song Han. 2023. TorchSparse++: Efficient Training and Inference Framework for Sparse Convolution on GPUs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 225–239.

Emanuele Vespa, Nils Funk, Paul H. J. Kelly, and Stefan Leutenegger. 2019. Adaptive-Resolution Octree-Based Volumetric SLAM. In *2019 International Conference on 3D Vision (3DV)*. 654–662. doi:10.1109/3DV.2019.00077

Ignacio Vizzo, Xieyuanli Chen, Nived Chebrolu, Jens Behley, and Cyrill Stachniss. 2021. Poisson surface reconstruction for LiDAR odometry and mapping. In *2021 IEEE international conference on robotics and automation (ICRA)*. IEEE, 5624–5630.

Ignacio Vizzo, Tiziano Guadagnino, Jens Behley, and Cyrill Stachniss. 2022. Vdbfusion: Flexible and efficient tsdf integration of range sensor data. *Sensors* 22, 3 (2022), 1296.

Peng-Shuai Wang, Yang Liu, Yu-Xiao Guo, Chun-Yu Sun, and Xin Tong. 2017. O-CNN: Octree-Based Convolutional Neural Networks for 3D Shape Analysis. *ACM Transactions on Graphics (SIGGRAPH)* 36, 4 (2017), 1–11. doi:10.1145/3072959.3073608

Francis Williams, Jiahui Huang, Jonathan Swartz, Gergely Klar, Vijay Thakkar, Matthew Cong, Xuanchi Ren, Ruilong Li, Clement Fuji-Tsang, Sanja Fidler, Eftychios Sifakis, and Ken Museth. 2024. fVDB: A Deep-Learning Framework for Sparse, Large Scale, and High Performance Spatial Intelligence. *ACM Transactions on Graphics* 43, 4 (July 2024), 1–15. doi:10.1145/3658226

Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. 2018. Open3D: A Modern Library for 3D Data Processing. arXiv:1801.09847 [cs.CV] https://arxiv.org/abs/1801.09847

Zihan Zhu, Songyou Peng, Viktor Larsson, Weiwei Xu, Hujun Bao, Zhaopeng Cui, Martin R Oswald, and Marc Pollefeys. 2022. Nice-slam: Neural implicit scalable encoding for slam. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 12786–12796.

A Hardware Configuration

All results, timings and memory usage numbers reported in our Applications, Section 5, were run on a machine running Ubuntu 26.04 using NVIDIA driver version 610.57 and CUDA 13.1 with an AMD Ryzen Threadripper PRO 3975WX 32-Cores CPU and an NVIDIA RTX PRO 6000 Blackwell Workstation Edition GPU with 96GB of GPU memory except for the results in the *Large-Scale Shape Voxelization* Section 5.3 which was done on a system with NVIDIA driver 595.84 using CUDA 13.2 and an AMD Ryzen Threadripper PRO 7965WX 24-Cores and an NVIDIA RTX PRO 6000 Blackwell Max-Q Workstation Edition GPU and the results in *Perception and Reality Capture* Section 5.1, including Appendix B, which was measured with an NVIDIA RTX 6000 Ada Generation GPU with 48GB of GPU memory.

B Reality Capture — Full Evaluation

This appendix expands the reality-capture summary of section 5.1 and describes the remaining four algorithms: one-shot ESDF, incremental ESDF, log-odds occupancy, and dynamic-scene decay; the per-voxel-size cross-library evaluation; and the regimes in which each baseline outperforms our corresponding implementation.

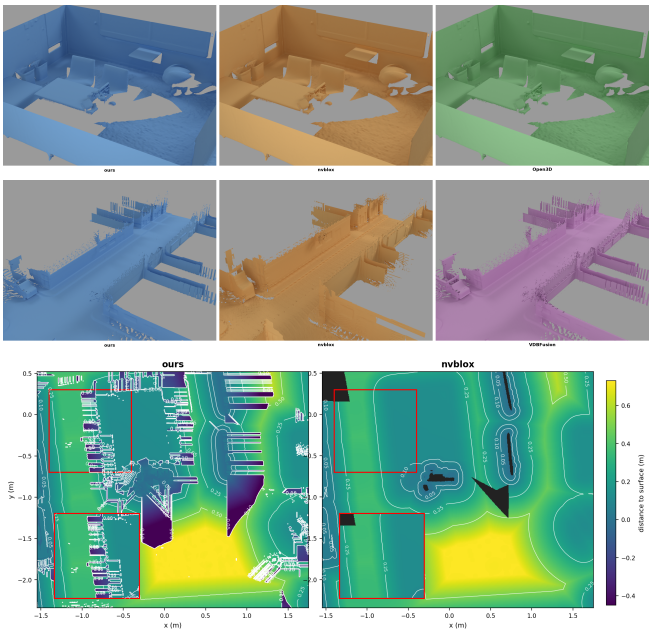


Fig. 13. Qualitative comparison on three of the reality-capture workloads. *Top:* Replica *office0* depth-TSDF mesh at 5 mm (500 frames, ceiling clipped); nvblox's 8^3-block padding shows as thicker walls. *Middle:* Mai City seq 00 LiDAR-TSDF mesh at 10 cm on a 60×40 m chunk; ours and VDBFusion show the cleanest sidewalk and lamp-post detail. *Bottom:* Replica *office0* ESDF horizontal slice at $z = -0.4$ m (5 mm voxels, R=1 m) with iso-contours at 0.05 / 0.10 / 0.25 / 0.50 m; the zoom inset (red box) highlights the resolution gap.

B.1 Algorithm overview

Composition pattern. Every integrator we implement follows the same three-phase recipe per frame: *(i)* build a topology shell of voxels that lie within the truncation band of a new observation and merge it into a persistent grid via `voxelsToGrid` [Williams et al. 2024] and `mergeGrids` (section 3); *(ii)* re-align all per-voxel sidecar tensors to the new grid via `inject` (section 3.4), with newly created voxels receiving a configurable initial value; *(iii)* run an integrator-specific update kernel over the active voxel sequence. Update rules differ per integrator (weighted running average for TSDF, atomic log-odds add for occupancy, monotone-min sweep for ESDF), but the surrounding grow-then-inject structure is shared.

Parallelism strategies. The five algorithms considered in section 5.1 and this appendix use three distinct GPU parallelism strategies depending on which is best matched to the workload's data flow. *(i)* Per-leaf-slot dispatch with an active-voxel mask, used by the depth-TSDF update kernel of algorithm 1: this is the simplest pattern and is competitive when leaf occupancy is high. *(ii)* Per-ray hierarchical-DDA, used by the LiDAR-TSDF update of algorithm 1 and by algorithm 4: one thread per input point walks the active voxels along its ray and writes via lock-free atomic accumulation. *(iii)* The voxel-sequential VBM dispatch of section 4, used by the ESDF sweep of algorithm 2: each active voxel is mapped to a single thread independently of leaf occupancy, which is the right pattern for low-occupancy stencil work. We quantify the VBM speedup over per-leaf-slot dispatch on the ESDF sweep in section B.3.

Euclidean signed distance field. Given a TSDF on grid G_{tsdf} (algorithm 1), we compute the ESDF as a composition of a topology dilation (allocating the support band around the zero-crossing), a 26-neighbour monotone-min Jacobi sweep, and an optional prune of unreached frontier voxels. algorithm 2 is a one-shot cold-start computation, and algorithm 3 is an incremental warm-start variant. The fixed-point loop is the standard parallel-chamfer envelope for 26-connectivity used by FIESTA [Han et al. 2019] and nvblox [Millane et al. 2024].

ALGORITHM 2: One-shot Euclidean signed distance field.

Input: TSDF grid G_{tsdf} with sidecars d, w; truncation τ; max ESDF distance R.
Output: ESDF grid G_{esdf} and signed-distance sidecar e.

$G_{\text{esdf}} \leftarrow \textsc{dilateGrid}(G_{\text{tsdf}}, \lceil R/\tau \rceil)$
$e \leftarrow \textsc{seedKernel}(G_{\text{esdf}}, d, w)$ `// e[v] = sign(d[v])·|d[v]| if w[v]>0 ∧ |d[v]|<τ, else +∞`
changed ← *True*
while *changed* **do**
  changed ← *False*
  foreach $v \in G_{\text{esdf}}$ iterated by VBM in parallel **do**
    $e_{\text{new}} \leftarrow \min_{n \in \mathcal{N}_{26}(v)} \left(e[n] + \text{sign}(e[n])\, \|x_v - x_n\|\right)$
    if $|e_{\text{new}}| < |e[v]|$ **then**
      $e[v] \leftarrow e_{\text{new}}$; *changed* ← *True*
    end
  end
end
$e \leftarrow \textsc{clamp}(e, -R, +R)$
return G_{esdf}, e

ALGORITHM 3: Incremental ESDF with dirty-mask short-circuit.

Input: Current TSDF G_{tsdf}, d, w; previous ESDF G_{prev}, e_{prev}; optional dirty mask m of length $|G_{\text{tsdf}}|$; truncation τ; max distance R.
Output: G_{esdf}, e.

if m is given and $m.\textsc{any}() = \mathit{False}$ **then**
  return $G_{\text{prev}}, e_{\text{prev}}$ `// Python short-circuit`
end
$G_{\text{esdf}} \leftarrow \textsc{dilateGrid}(G_{\text{tsdf}}, \lceil R/\tau \rceil)$
$e \leftarrow \textsc{inject}(G_{\text{prev}} \rightarrow G_{\text{esdf}}, e_{\text{prev}}, \mathit{init}{=}+\infty)$
foreach $v \in G_{\text{esdf}}$ where (m *is None or* $m[v]$) **do**
  $e[v] \leftarrow \textsc{seedValue}(v, d, w)$
end
Run the fixed-point sweep of algorithm 2 to convergence
return G_{esdf}, e

Log-odds occupancy. Occupancy mapping (algorithm 4) shares algorithm 1's grow-then-inject scaffold and replaces only the per-voxel update rule following OctoMap [Hornung et al. 2013]. Occupancy is the smallest of our integrators (~430 lines in one custom kernel).

ALGORITHM 4: Log-odds occupancy integration (single frame).

Input: Grid G; log-odds sidecar ℓ of length $|G|$; Π, Z, τ as in algorithm 1; $\ell_{\text{hit}}, \ell_{\text{miss}}$ Bayesian increments; clamp band $[\ell_{\min}, \ell_{\max}]$.
Output: G', ℓ'.

$\mathit{shell} \leftarrow \textsc{voxelsToGrid}(\textsc{LiDAREndpoints}(Z) \oplus \textsc{ball}(\tau))$
$G' \leftarrow \textsc{mergeGrids}(G, \mathit{shell})$
$\ell' \leftarrow \textsc{inject}(G \rightarrow G', \ell, \mathit{init}{=}0)$
foreach *ray* $r \in Z$ in parallel via HDDA ray-walk **do**
  foreach *voxel* v traversed by r **do**
    $\delta \leftarrow r.\textsc{isEndpoint}(v)\ ?\ \ell_{\text{hit}} : \ell_{\text{miss}}$
    $\textsc{atomicAdd}(\ell'[v], \delta)$
  end
end
$\ell' \leftarrow \textsc{clamp}(\ell', \ell_{\min}, \ell_{\max})$
return G', ℓ'

Dynamic-scene decay. Dynamic-scene support — fading old observations so that moving objects stop persisting — is a pure composition of two operators with no integrator-side code (algorithm 5, ~30 lines of Python). On the first 500 frames of KITTI seq 00 at 10 cm voxels with decay applied every 25 frames (γ=0.5, threshold 0.05), the persistent active voxel count drops from 48.6 M (no-decay baseline) to 10.6 M — a **78% reduction** — and is bounded at ~10 M voxels through the integration window, while the no-decay baseline grows monotonically (15 M / 23 M / 31 M / 41 M / 49 M at 100 / 200 / 300 / 400 / 500 frames). The bounded state is the architectural effect of expressing decay as a tensor op on the weight sidecar plus a topology `pruneGrid`, with no integrator-side code aware of temporal coherence.

ALGORITHM 5: Dynamic-scene decay (per-tick).

Input: Grid G; primary sidecar s; additional sidecars s_{extra}; decay factor $\gamma \in (0, 1)$; prune threshold ε.
Output: $G', s', s'_{\text{extra}}$.

$s \leftarrow s \cdot \gamma$ `// tensor multiply`
$m \leftarrow s \geq \varepsilon$ `// tensor mask`
$G' \leftarrow \textsc{pruneGrid}(G, m)$
$s' \leftarrow \textsc{inject}(G \rightarrow G', s)$
$s'_{\text{extra}} \leftarrow \textsc{inject}(G \rightarrow G', s_{\text{extra}})$
return $G', s', s'_{\text{extra}}$

B.2 Cross-library evaluation

Setup. We benchmark against nvblox [Millane et al. 2024] for LiDAR/depth TSDF, ESDF, and occupancy; Open3D [Zhou et al. 2018] for depth TSDF; and VDBFusion [Vizzo et al. 2022] for LiDAR TSDF. All measurements are on a single NVIDIA RTX 6000 Ada Generation GPU (48 GB) with PyTorch's `expandable_segments` allocator. All systems use truncation distance $\tau = 3 \times$ voxel size for TSDF and maximum ESDF distance $R = 10 \times$ voxel size, applied uniformly across baselines so the comparison is on integrator behaviour rather than parameter choices. nvblox in particular is configured with `projective_integrator_ truncation_distance_vox` = 3.0 (matching τ) and a value `lidar_projective_integrator_max_integration_distance_m` of 60 m on LiDAR plus an 8 m equivalent on Replica/7-Scenes depth, in place of nvblox's upstream defaults of 4.0× truncation and 10 m / 7 m integration ranges; the upstream defaults clip nvblox's input to a ±10 m strip on LiDAR and inflate its truncation band by 33%, neither of which is consistent with the workload fvdb processes.

Sensitivity to baseline defaults. As discussed, larger parameter values were chosen in order to produce larger scale scenes, consistent with the size of workloads our system was designed to process, rather than the upstream defaults assigned in nvblox. The voxel-size thresholds which produce out-of-memory errors for nvblox in table 4 and table 6 which support our scale claims are insensitive to using nvblox's default or the 6× larger value we chose.

We repeated the table 4 and table 6 sweeps with nvblox at its upstream defaults: 10 m LiDAR max integration distance and 4× voxel truncation (`ProjectiveIntegratorParams` defaults in nvblox version 0.0.10). In this experiment (table 2), nvblox's OOM voxel size moved by at most one voxel-size step in the sweep, and on Mai City it did not move at all.

nvblox's OOM is triggered by its GPU block-hash rebuild (hard-coded `max_load_factor = 0.5`, `growth_factor = 2`): when the accumulated map crosses half of the current power-of-two capacity, nvblox constructs a new table at twice the capacity while the old one is still resident and re-inserts every entry, with the backing voxel-block pool growing by the same ×2 policy. On Mai City at 5 cm, this rebuild transiently demands ~50 GB of device memory to admit 854 new blocks into the map. Using a 96 GB device, we measured that the run settles at 4.85 GB but peaks at 51.7 GB during the rebuild which will OOM with the 48 GB budget of the RTX 6000 Ada used in the experiments. Because the trigger is the accumulated block count crossing a capacity threshold, clipping the input from

Table 2. **Sensitivity of scaling results** to chosen values of (LiDAR max integration distance, voxel truncation)

Workload	nvblox @ (60 m, 3×)	nvblox @ defaults (10 m, 4×)	Ours @ defaults-matched
Mai City TSDF (700 f)	OOM at 5 cm	OOM at 5 cm*	1 cm (1.20 B voxels, 45.9 GB)
Mai City ESDF (700 f)	OOM at 5 cm	OOM at 5 cm*	2 cm (30.0 GB)
KITTI TSDF (seqs 00/02/05)	OOM at 10 cm (all)	seq 02: OOM at 10 cm*; seqs 00/05: OOM at 5 cm†	5 cm / 5 cm / 3 cm
KITTI ESDF (100 f, 3 seqs)	OOM at 3 cm	OOM at 2 cm†	1 cm (28–32 GB)

* No change from the paper configuration.
† Improved by one resolution step.

60 m to 10 m reduces nvblox's steady-state memory ~4× but does not keep the 700f Mai City trajectory below the fatal limit at 5 cm. Our index-grid topology allocates exactly the observed voxels with no doubling rebuild, so peak ≈ steady state, which is why it fills 45.9 GB of the 48 GB budget effectively at 1 cm on the same workload.

Speed (LiDAR TSDF). On Mai City and KITTI we are 3.6–5.7× faster than nvblox at every cell where both run (table 3), with the speedup narrowing monotonically as voxel size shrinks — 5.7× at 40 cm Mai City to 3.6× at 10 cm Mai City; 4.8–5.3× at 40 cm KITTI and 2.6–3.5× at 20 cm KITTI. nvblox's per-block bookkeeping cost is roughly fixed per frame regardless of voxel size, while our cost scales with the count of voxels actually stored in the truncation shell, so the relative gap closes at finer voxels. On KITTI 10 cm nvblox OOMs on every sequence (its hash allocator's resize step requests a contiguous slot buffer that the 48 GB device can no longer supply), while ours runs at 12–21 ms/frame. On the smallest KITTI sequence (seq 05, 2.2 km) we additionally reach 5 cm at 33.3 GB of GPU memory with 726 M voxels — twice as fine as the longer sequences, which OOM there. VDBFusion (CPU OpenVDB) [Vizzo et al. 2022] is 22–39× slower than ours across the KITTI 40–10 cm sweep and 38–84× slower on Mai City across 40 cm to 2 cm; the cross-CPU gap is large enough that we report it for completeness only.

Speed (Occupancy). Occupancy is the cleanest sweep — we win at every voxel size on KITTI (table 4), with the speedup widening monotonically from **3.5× at 20 cm to 11.5× at 1.5 cm**, and continuing past nvblox's OOM threshold (1 cm on every sequence) to **2.35 B voxels in 44.1 GB at 5 mm**. The widening tracks the occupancy state's growing sparsity: nvblox's 8^3 block padding caps its memory growth but penalises its time as the sparse occupancy sidecar stays sparse, while our per-voxel index stores exactly the active voxels and our update kernel touches only those.

Memory regime caveat. nvblox stores its state as fixed 8×8×8 voxel blocks, allocating every cell in a block whether the cell is active or not; ours allocates exactly the active voxels with a per-voxel index. The two strategies trade places depending on how densely each 8^3 block is occupied. On dense KITTI seq 02 / 20 cm LiDAR TSDF, urban ground returns fill blocks nearly to capacity, so nvblox's per-block overhead amortises and our per-voxel index has the larger footprint: our peak GPU memory (5.04 GB) exceeds nvblox's (3.08 GB) by 1.6× (table 3). The same regime appears at 40 cm KITTI where the gap is small (within ±20%). Conversely, on KITTI Occupancy at 1.5 cm where the state is spread thinly across ~600 M-voxel volumes, nvblox pays for many empty cells inside its blocks: our peak (3-seq mean 10.6 GB) is 3.5× smaller than nvblox's (36.9 GB) (table 4), and at 1 cm nvblox cannot fit at all while ours uses 19.8 GB. The advantage holds asymptotically for sparsely-occupied workloads (the truncation-shell of depth or LiDAR TSDF below 10 cm, the dilated band of ESDF, fine-voxel occupancy) and erodes for densely-occupied coarse-voxel ones. We report this rather than selecting datasets that suppress it.

Throughput scaling. On bounded-scene depth TSDF, Open3D's CUDA integrator [Zhou et al. 2018] is 11–46× faster than ours over the voxel range where both run on every scene (20–5 mm), with the gap *widening* monotonically because Open3D's per-frame block-hash bookkeeping is amortised over a tightly packed contiguous allocation while our cost grows with the count of voxels actually stored in the truncation shell. Open3D begins to fail on increasingly many Replica scenes as the voxel size shrinks below 5 mm and crashes on every scene at 3 mm (caption of table 5). Ours scales monotonically through 3 mm at an 8-scene mean 44.50 ms/f over ~109 M active TSDF voxels and an 8-scene-mean 1.61 s ESDF cold sweep over 438 M ESDF voxels. Our claims pertain to scale and breadth, not throughput parity on this single workload.

Long-trajectory persistence. To stress the grow-then-inject discipline of algorithm 1 on an indoor sequence longer than Replica or KITTI, we integrate 7-Scenes *chess* [Shotton et al. 2013] for its full 3000 RGB-D frames at 20/10/5/3 mm depth TSDF (lower block of table 5). Ours reaches 3 mm at 187 M active voxels in 10.6 GB with bounded mean per-frame cost across the trajectory; Open3D runs every cell at 9–49× our throughput, consistent with the Replica *throughput scaling* above.

ESDF scaling. On Mai City synthetic LiDAR over the full 700-frame trajectory, nvblox's cold ESDF is faster than ours at 20 cm by 1.4× but slower at 10 cm by 1.8×, and cannot build the TSDF substrate below 5 cm (table 3); we scale to **3 cm with 1.28 B ESDF voxels in 30.6 GB** on the same trajectory, and OOM at 2 cm (table 6). On KITTI we use a 100-frame TSDF prefix per sequence so that the dilated ESDF band fits in GPU memory across the voxel-size sweep on both systems; nvblox is 1.4–2.0× faster at 20 cm and competitive at 10 cm (KITTI seq 00: 0.88×; seq 02: 1.17× slower; seq 05: 1.60× slower), then we are 1.8–1.9× faster at 5 cm on seq 00 and seq 05, and nvblox OOMs from seq 02 / 5 cm onward. What ours wins on KITTI is *scale*: at 3 cm on seq 02 we run at 3.93 s on 1.20 B active voxels

Table 3. **LiDAR TSDF on Mai City synthetic Velodyne sweeps and KITTI Odometry real-sensor LiDAR.** Per-frame integration time and peak GPU memory (CPU peak host RSS for VDBFusion); faster GPU time per row in bold. nvblox is configured with `lidar_projective_integrator_max_integration_distance_m` = 60 m and `projective_integrator_truncation_distance_vox` = 3.0 to match our 60 m-range LiDAR fusion and our 3× truncation; nvblox's upstream defaults of 10 m and 4.0× would clip its input to a ±10 m strip and widen its band by 33%. nvblox first OOMs at 5 cm on Mai City and at 10 cm on every KITTI sequence we tested. Ours scales through 1.25 cm on Mai City at 37.1 GB / 938.7 M voxels and OOMs at 1.0 cm; KITTI seq 00 and seq 02 OOM at 5 cm while the smaller seq 05 reaches 5 cm at 33.3 GB on 726 M voxels, and all three KITTI sequences OOM at 3 cm. VDBFusion is 38–94× slower than ours on Mai City across 40–2 cm and 22–41× slower on KITTI across 40–10 cm; on KITTI 5 cm VDBFusion runs cleanly on its CPU-side RAM where ours OOMs on the GPU, and on the seq 05 / 5 cm cell where both run, VDBFusion is 27× slower. We did not run VDBFusion at KITTI 3 cm: extrapolating from the seq 02 / 5 cm cell's 96 GB peak host RSS, the 3 cm runs would exceed available system memory.

		Ours		nvblox		VDBFusion
Dataset / seq	Voxel	ms/f	GB	ms/f	GB	ms/f
Mai City 700 f	40 cm	**3.44**	0.17	19.52	0.93	130.04
	20 cm	**3.80**	0.29	19.06	1.14	240.02
	10 cm	**5.07**	0.77	18.03	1.98	474.15
	5 cm	**11.22**	4.47	OOM	—	965.28
	3 cm	**19.98**	9.30	—	—	1666.65
	2 cm	**32.26**	17.47	—	—	2723.86
	1.5 cm	**52.47**	27.48	—	—	—
	1.25 cm	**68.10**	37.05	—	—	—
	1.0 cm	OOM	—	—	—	—
KITTI seq 00 (4541 f)	40 cm	**4.10**	1.23	21.07	1.14	118.86
	20 cm	**7.01**	4.02	19.21	3.05	224.62
	10 cm	**17.81**	15.78	OOM	—	443.41
	5 cm	OOM	—	—	—	965.39
	3 cm	OOM	—	—	—	—
KITTI seq 02 (4661 f)	40 cm	**4.38**	1.48	20.94	1.41	122.83
	20 cm	**7.88**	5.04	20.59	3.08	229.05
	10 cm	**21.49**	19.77	OOM	—	462.97
	5 cm	OOM	—	—	—	1003.33
	3 cm	OOM	—	—	—	—
KITTI seq 05 (2761 f)	40 cm	**3.92**	0.80	20.96	1.00	127.72
	20 cm	**5.80**	2.30	20.49	1.98	236.73
	10 cm	**12.00**	8.43	OOM	—	469.41
	5 cm	**38.22**	33.31	—	—	1027.99
	3 cm	OOM	—	—	—	—

(27.7 GB) where nvblox is out of memory, and at 2 cm on seq 00 we reach **1.67 B active voxels in 38.6 GB**. The dirty-mask short-circuit of algorithm 3 reduces the no-change case to sub-400 μs at every voxel size we measured.

B.3 Performance Trade-offs and Limitations

Four regimes where the architectural argument degrades:

(1) **Open3D wins bounded depth-TSDF throughput by 11–46× at 20–5 mm Replica voxels** (see *throughput scaling* above), with the gap widening monotonically. nvblox is also faster than ours at 20 mm (3.3×) but loses to ours from 10 mm onward (table 5). Open3D's integrator has a decade of head start on hand-tuned block-hash allocators; we do not match it on this axis. We do scale past both: at 4 mm nvblox fails on

Table 4. **KITTI Odometry occupancy mapping.** Per-voxel-size means over the three sequences {00, 02, 05}. nvblox is configured to match our 60 m LiDAR range and 3× truncation (its defaults of 10 m and 4.0× clip the input to a thin strip and widen the band). nvblox first OOMs at 1 cm on every sequence (its hash allocator's resize step requests a single contiguous block of ~2 M slot entries that the 48 GB device can no longer supply); ours scales to 2.35 B active voxels in 44.1 GB at seq 05 / 5 mm. The split rows show finer voxels where ours continues running past nvblox's ceiling.

Voxel	Ours (ms/f)	nvblox (ms/f)	Speedup
20 cm	5.8	20.1	**3.5**×
10 cm	7.7	21.6	**2.8**×
5 cm	15.6	53.2	**3.4**×
3 cm	27.1	146.8	**5.4**×
2 cm	41.7	351.7	**8.4**×
1.5 cm	59.1	678.6	**11.5**×
1 cm	93.3	OOM	ours only
8 mm (seq 00 + 05)	114.2	OOM	ours only
8 mm (seq 02)	212.7	OOM	ours only
5 mm (seq 00 + 05)	270.2	OOM	ours only
5 mm (seq 02)	OOM	OOM	both OOM

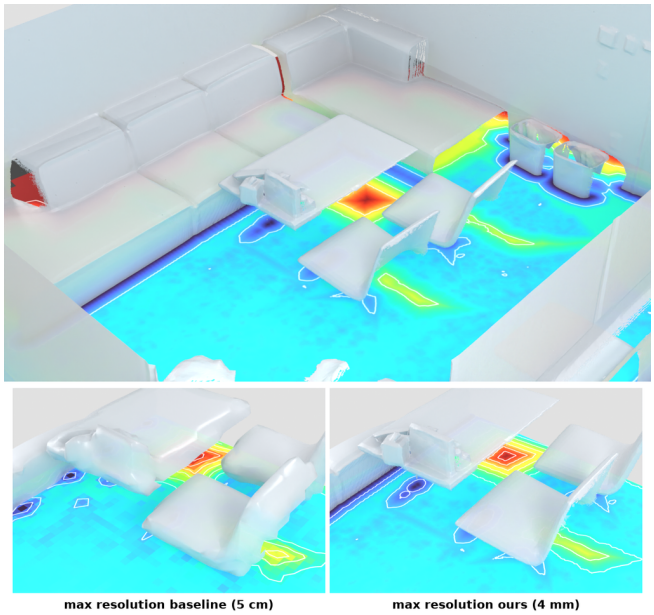


Fig. 14. The TSDF + ESDF pipeline pushed to the highest resolution each system can run on a single 48 GB GPU. 5 cm is roughly the finest setting at which the block-hash TSDF systems we compare against (nvblox, Open3D, VDBFusion) remain stable on Replica scenes; NanoVDB's index-grid lets us reach 4 mm (12.5× finer), at which point the mesh resolves individual chair backs and the ESDF iso-contours (drawn every 5 cm of distance, in white) become smooth and dense rather than blocky and pixelated.

Table 5. **Depth TSDF and ESDF on Replica (8 scenes × 2000 frames; mean) and 7-Scenes *chess* (3000 frames).** Per-frame TSDF time and cold post-TSDF ESDF sweep time; fastest TSDF / ESDF per row in bold. Open3D and nvblox have no ESDF integrator on 7-Scenes in our setup; we ran nvblox only on Replica. nvblox is configured with `projective_integrator_max_integration_distance_m` = 8 m to match the depth ranges in Replica's NICE-SLAM rendering, slightly above its upstream default of 7 m. nvblox fails with allocator OOM during its hash-resize step from Replica 5 mm onward; Open3D fails with `illegal memory` access from its CUDA integrator on a staircase — 8/8 Replica scenes succeed at 10 mm and coarser, 6/8 at 5 mm, 2/8 at 4 mm (office0 and office1 only), 0/8 at 3 mm. Ours scales monotonically through Replica 3 mm and 7-Scenes 3 mm with no per-frame creep across the 3000-frame chess trajectory (peak 10.6 GB at chess 3 mm).

		Ours		nvblox		Open3D CUDA
Dataset	Voxel	TSDF (ms/f)	ESDF (ms)	TSDF (ms/f)	ESDF (ms)	TSDF (ms/f)
Replica 8×2000 f	20 mm	2.64	**38.46**	0.81	75.60	**0.25**
	10 mm	5.23	**125.83**	6.43	529.54	**0.26**
	5 mm	**19.05**	**617.71**	OOM	OOM	**0.42**‡
	4 mm	**28.06**	**926.85**	OOM	OOM	**0.41**‡
	3 mm	**44.50**	**1614.89**	OOM	OOM	crash
7-Scenes *chess* 3000 f	20 mm	1.96	n/a	n/a	n/a	**0.21**
	10 mm	2.31	n/a	n/a	n/a	**0.23**
	5 mm	5.61	n/a	n/a	n/a	**0.26**
	3 mm	16.20	n/a	n/a	n/a	**0.33**

‡ Open3D mean over the Replica scenes that succeed.

every scene and Open3D on 6 / 8; at 3 mm both fail on every scene.

(2) **Cold ESDF cost is 1.0–2.0× higher than nvblox's at 20 cm KITTI voxels** (tied on seq 02, 1.5× slower on seq 05, 2.0× slower on seq 00), and 1.5× higher at 40 cm Mai City. We become uniformly faster from 20 cm on Mai City and from 5 cm on KITTI; nvblox OOMs from 5 cm KITTI seq 02 and 3 cm everywhere else.

(3) **Our memory advantage over nvblox's per-block storage is regime-dependent.** Dense urban LiDAR at coarse voxels (KITTI seq 02 / 20 cm TSDF) flips the per-voxel comparison to nvblox-favourable by 1.6× since its 8^3 blocks are nearly full; sparse-state fine-voxel occupancy (KITTI 1.5 cm) flips it back to a 3.5× advantage in our favour, and at 1 cm nvblox cannot fit at all. The advantage holds asymptotically for sparsely-occupied applications (*Memory regime caveat*, above).

(4) **The VBM speedup over per-leaf-slot dispatch on the ESDF sweep is 3–6%, not the order-of-magnitude one might expect** from a hierarchical-dispatch-replacement story. ESDF leaves end up densely occupied by the dilation step (algorithm 2, line 1), so per-leaf-slot iteration wastes few threads on inactive entries. VBM's advantage is realised on low-occupancy stencil work where active voxels are sparse within each leaf.

A fifth caveat was present in early designs of algorithm 3: incremental ESDF warm-reuse was 400× slower than nvblox's `update_esdf`. The dirty-mask Python short-circuit of algorithm 3 reduces the no-change case to sub-400 μs and matches nvblox's sub-100 μs block-cache hit on the typical sparse-update case.

Table 6. **LiDAR ESDF on Mai City over the full 700-frame trajectory and on KITTI Odometry over a 100-frame TSDF prefix per sequence.** Active voxel count, cold ESDF time, dirty-mask idempotent time, and peak GPU memory. nvblox is configured to match our 60 m LiDAR range and 3× truncation. nvblox first OOMs at Mai City 5 cm (TSDF substrate fails) and at KITTI seq 02 / 5 cm; on KITTI seq 00 and seq 05 nvblox runs at 5 cm but OOMs at 3 cm. Ours scales to 1.67 B active voxels at 38.6 GB on KITTI seq 00 / 2 cm and 1.28 B voxels at 30.6 GB on Mai City 700-frame / 3 cm.

			Ours			nvblox	
Dataset / seq	Voxel	ESDF voxels	Cold (ms)	Warm (μs)	Peak (GB)	Cold (ms)	Peak (GB)
Mai City 700 f	20 cm	38.7 M	129.0	25 μs	2.04	174.5	1.67
	10 cm	129.4 M	416.4	32 μs	4.17	740.4	2.50
	5 cm	459.2 M	1628.0	59 μs	11.82	OOM	—
	3 cm	**1.28 B**	5567.9	266 μs	**30.59**	OOM	—
	2 cm	OOM	—	—	—	OOM	—
KITTI seq 00 (100 f)	20 cm	21.6 M	81.3	25 μs	0.66	41.4	1.46
	10 cm	76.3 M	212.5	33 μs	1.94	186.2	1.90
	5 cm	277 M	800.4	49 μs	6.61	1477.1	3.37
	3 cm	779 M	2571	191 μs	18.12	OOM	—
	2 cm	**1.67 B**	5906	365 μs	**38.61**	OOM	—
KITTI seq 02 (100 f)	20 cm	27.2 M	87.0	25 μs	0.80	85.8	1.53
	10 cm	107 M	327.5	33 μs	2.66	383.9	2.41
	5 cm	416 M	1268	66 μs	9.83	OOM	—
	3 cm	1.20 B	4184	299 μs	27.71	OOM	—
	2 cm	OOM	—	—	—	OOM	—
KITTI seq 05 (100 f)	20 cm	19.5 M	62.6	24 μs	0.62	43.0	1.52
	10 cm	70.5 M	218.5	33 μs	1.81	349.4	1.88
	5 cm	258.9 M	788.2	50 μs	6.18	1516.6	3.34
	3 cm	732.3 M	2504.9	183 μs	17.07	OOM	—