fVDB: A Deep-Learning Framework for Sparse, Large-Scale, and High-Performance Spatial Intelligence

FRANCIS WILLIAMS, JIAHUI HUANG, and JONATHAN SWARTZ, NVIDIA Research
GERGELY KLÁR, VIJAY THAKKAR, and MATTHEW CONG, NVIDIA Research
XUANCHI REN, RUILONG LI, and CLEMENT FUJI-TSANG, NVIDIA Research
SANJA FIDLER, EFTYCHIOS SIFAKIS*, and KEN MUSETH, NVIDIA Research

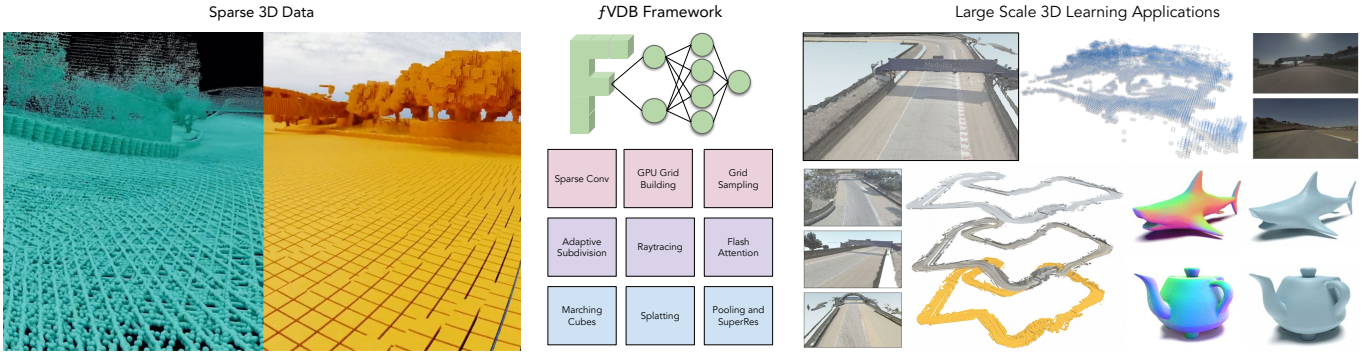

Fig. 1. fVDB is an integrated Deep Learning framework for large-scale, and high-performance spatial intelligence. It can process 3D data from a broad range of sources, including voxels, point clouds, and surface meshes. fVDB also offers a rich set of state-of-the art differentiable operators, which can be used to build Deep Learning architectures for tasks in 3D Deep Learning, thus facilitating DL applications on large scale and high-resolution 3D data.

We present fVDB, a novel GPU-optimized framework for deep learning on large-scale 3D data. fVDB provides a complete set of differentiable primitives to build deep learning architectures for common tasks in 3D learning such as convolution, pooling, attention, ray-tracing, meshing, etc. fVDB simultaneously provides a *much larger* feature set (primitives and operators) than established frameworks with no loss in efficiency: our operators match or exceed the performance of other frameworks with narrower scope. Furthermore, fVDB can process datasets with much larger footprint and spatial resolution than prior works, while providing a competitive memory footprint on small inputs. To achieve this combination of versatility and performance, fVDB relies on a single novel VDB index grid acceleration structure paired with several key innovations including GPU accelerated sparse grid construction, convolution using tensorcores, fast ray tracing kernels using a Hierarchical Digital Differential Analyzer algorithm (HDDA), and jagged tensors. Our framework is fully integrated with PyTorch enabling interoperability with existing pipelines, and we demonstrate its effectiveness on a number of representative tasks such as large-scale point-cloud segmentation, high resolution 3D generative modeling, unbounded scale Neural Radiance Fields, and large-scale point cloud reconstruction.

CCS Concepts: • **Computing methodologies** → **Neural networks**; **Spatial and physical reasoning**.

Additional Key Words and Phrases: Deep learning frameworks, spatial intelligence, GPU, sparse convolution, point cloud processing, neural rendering

ACM Reference Format:
Francis Williams, Jiahui Huang, Jonathan Swartz, Gergely Klár, 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 Trans. Graph.* 43, 4, Article 133 (July 2024), 15 pages. https://doi.org/10.1145/3658226

1 INTRODUCTION

Deep Learning methods have been foundational to solving a wide variety of previously intractable problems in computer science. These include building agents capable of passing the Turing test, generating high quality images from text prompts, speech and audio synthesis, and perception for robotics to name a few. Underlying these innovations lies a rich software ecosystem of deep learning primitives (such as convolution, pooling, and attention) which can be composed to build neural networks such as transformers or convolutional networks. These primitives are exposed to the programmer through deep learning frameworks such as PyTorch [Paszke et al. 2019], JAX [Bradbury et al. 2018], or TensorFlow [Abadi et al. 2015]. In common frameworks, these primitives operate on dense tensors of data, which often encode 1D or 2D signals (*e.g.* text or images). In the case of tasks in 3D, dense tensors are fundamentally limited

*Also with University of Wisconsin-Madison; Madison, WI, USA.

Authors' addresses: Francis Williams, fwilliams@nvidia.com; Jiahui Huang, jiahuih@nvidia.com; Jonathan Swartz, jswartz@nvidia.com, NVIDIA Research; Gergely Klár, gklar@nvidia.com; Vijay Thakkar, vithakkar@nvidia.com; Matthew Cong, mcong@nvidia.com, NVIDIA Research; Xuanchi Ren, xuanchir@nvidia.com; Ruilong Li, ruilongl@nvidia.com; Clement Fuji-Tsang, cfujitsang@nvidia.com, NVIDIA Research; Sanja Fidler, sfidler@nvidia.com; Eftychios Sifakis, sifakis@cs.wisc.edu, esifakis@nvidia.com; Ken Museth, kmuseth@nvidia.com, NVIDIA Research.

ACM 0730-0301/2024/7-ART133
https://doi.org/10.1145/3658226

in size due to cubic scaling and memory constraints. Fortunately, 3D data is often sparse in nature, only requiring information to be encoded in a subset of the volume such as in the interior or near the surface of a shape. Thus, there has been an emergence of frameworks [Choy et al. 2019; Tang et al. 2022, 2023] which operate on sparse 3D tensors of data. Correspondingly, many recent works propose network architectures which can operate on sparse 3D data [Choy et al. 2019; Huang et al. 2023; Qi et al. 2017; Ren et al. 2023; Wang et al. 2017].

Past sparse 3D learning frameworks leverage hash tables as the primary data structure for mapping 3D integer coordinates to tensor data. Such a data structure works well for operators such as convolution and pooling, but the lack of spatial coherence of accesses makes it inefficient for operators such as sampling, splatting, and ray tracing without the use of auxiliary acceleration structures. Thus, past frameworks typically include a small number of operators such as convolution and pooling. However, we note that modern 3D learning tasks often involve a number of complex operators that must be combined together. For example, [Liu et al. 2023a] performs image-to-3d generation by unprojecting image features to a dense volume, leveraging a dense and sparse convolutional network to produce a sparse volume of learned features, then differentiably meshing and rendering this volume to produce a textured shape. Such a pipeline requires a number of complex differentiable operators (ray tracing, splatting, convolution, pooling, attention, meshing, and rendering) which can operate on sparse grids of learnable features. Currently, such pipelines are built using bespoke operators which glue together different acceleration structures (e.g. hash tables, occupancy bit fields and meshes) from different libraries.

In this paper, we present *f*VDB, a novel deep-learning framework for operating on sparse 3D tensors. Our framework provides a wide host of differentiable GPU accelerated 3D operators which can be easily composed to build complex 3D learning pipelines. Each of these operators delivers performance that is on par with or exceeding the performance of state-of-the-art operators from other frameworks which are much narrower in scope. Furthermore, *f*VDB is memory efficient and is capable of processing *much larger* inputs than existing alternatives. Table 1 summarizes the features of *f*VDB in contrast to existing 3D learning frameworks.

The key innovation that enables us to develop a flexible and composable framework while still achieving state-of-the-art performance is a new data structure derived from NanoVDB [Museth 2021], which we call *IndexGrid*. This is paired with a novel ecosystem of tools for grid construction and traversal (see Section 3.3), accelerated ray marching (see Section 3.4), and a novel data processing paradigm that unlocks aggressive optimizations in the application of stencil-based operators (*e.g.* convolution in Section 3.5). While incorporating algorithms originally used in hash grid methods, which can be trivially adapted to our VDB structure, we also introduce new design paradigms that fit naturally within our representation. Specifically, we design optional convolutional alternatives that leverage efficient construction of locally densified, windowed views into the sparse data on which data regularity and aggressive utilization of tensorcores enable exceptional compute efficiency.

Our core contributions include:

- The design and deployment of a comprehensive API for spatial intelligence, with necessary primitives to accommodate a wide spectrum of high-value 3D Machine Learning tasks.
- A new sparse data structure, *IndexGrid*, derived from NanoVDB [Museth 2021] but with a drastically re-imagined programming and execution model aimed to aggressively accelerate stencil-centric operations.
- A collection of GPU-optimized fast operators (convolution, attention, raytracing, etc) built around the IndexGrid structure, engineered to specifically target high efficiency on spatially sparse data.
- A new benchmark for sparse convolution that highlights different workloads in terms of sparsity pattern and feature depth.
- Memory efficient algorithms which enable scaling to much larger inputs than prior works.
- A demonstration of the applicability of our framework to a variety of end-to-end training and inference applications from a broad spectrum of 3D Deep Learning tasks.

2 RELATED WORK

Sparse Voxel Data Structures for Deep Learning. Sparse 3D voxel grids are a common representation for deep learning on 3D data. Many past works such as [Choy et al. 2019; Contributors 2022; Tang et al. 2022, 2023] use a hash table to encode a mapping between 3D integer `ijk` coordinates and offsets into a tensor of features. Such a mapping enables on average $O(1)$ lookup of arbitrary features, however accesses are not spatially coeherent. Furthermore, hash tables are not effective acceleration structures for operations such as ray marching since they are not a BVH.

Another line of works [Jatavallabhula et al. 2019; Wang et al. 2017] use octrees instead of a hash table. These preserve spatial coherence and can be ray-marched efficiently, but at the cost of $O(\log N)$ access, and can grow quite deep for high resolutions. In contrast, *f*VDB uses a fixed depth, shallow VDB [Museth 2013] tree, which enables $O(1)$ amortized reads and writes, and serves as an effective acceleration structure for a wide range of operations (See Table 1). VDB is a widely used data structure in computer graphics and simulation with several implementations including OpenVDB [Museth 2013] and NanoVDB [Museth 2021] which implements a subset of OpenVDB on the GPU. More recently, NeuralVDB [Kim et al. 2022] added neural compression on top of NanoVDB.

Lastly, there are works that allow for the definition of custom, sparse volumetric data structures, such as the Taichi domain-specific language [Hu et al. 2020, 2019], which provides a means to emit optimized, differentiable code, with emphasis on simulation tasks. In contrast, *f*VDB is a general purpose framework targeting spatial sparsity, providing a collection of primitives that are useful to build end-to-end deep learning applications.

Deep Learning Frameworks. Deep learning architectures are constructed by composing together a series of differentiable operators with trainable parameters and optimizing those parameters via minimizing a loss functional over a dataset. In order to enable research and development of deep learning architectures, a number of software framework with composable primitives have

arisen in the past decade. The most commonly used frameworks include PyTorch [Paszke et al. 2019], TensorFlow [Abadi et al. 2015], JAX [Bradbury et al. 2018], and Keras [Chollet et al. 2015]. These libraries expose primitives for operating on dense tensors of data (such as images and audio signals).

3D Deep Learning Software. 3D deep learning tasks often involve more complex primitives which operate on sparse tensors. Common libraries such as the Minkowski Engine [Choy et al. 2019], TorchSparse [Tang et al. 2022, 2023], and SpConv [Contributors 2022] add support for constructing sparse tensors with basic operations such as convolution and pooling. Other libraries such as NerfAcc [Li et al. 2023], PyTorch3D [Ravi et al. 2020] and Kaolin [Jatavallabhula et al. 2019] provide other graphics operators such as ray tracing using dense bitfields and octrees as well as operators for meshes and graphs. Our framework, *f*VDB unifies many of these operations under a single library, providing a broader set of features than past works using only a single, highly versatile novel VDB acceleration structure.

Applications of Sparse 3D Learning Frameworks. Frameworks for deep learning on sparse tensors have been used in a number of important applications in deep learning including Point Cloud Processing [Choy et al. 2019; Zhao et al. 2021], 3D reconstruction of geometry from point clouds and/or images [Huang et al. 2022, 2023; Tancik et al. 2023], perception [Choy et al. 2019; Liu et al. 2023b; Shi et al. 2020], and, more recently, 3D generative modelling [Ren et al. 2023]. *f*VDB exposes the operators to perform all these tasks under a single library using only our IndexGrid VDB as an acceleration structure. Section 4.3 shows some demonstrative applications of our framework to different tasks in 3D Deep Learning.

3 METHOD

As the name suggests, *f*VDB is built on the VDB data structure [Museth 2013], which offers both compact storage and fast access to sparse 3D data. However, unlike previous adoptions of VDB, e.g. in OpenVDB[ASWF] and NanoVDB [Museth 2021], we have developed novel techniques specifically for machine learning on the GPU. This includes indexed storage, fast grid construction on the GPU, hierarchical Digital Differential Analyzers (DDAs) [Museth 2014] for accelerated GPU raymarching, and blocked computation, each of which will be discussed below. Many of these improvements build on NanoVDB, yet they are essential to the *f*VDBframework and play a critical role in enhancing the performance of our ML system.

3.1 Background: VDB

As a preamble, let's briefly summarize some of the main characteristics of the VDB data structure (see [Museth 2013] for more details). At the core, VDB is a shallow 3D tree structure, with a hash table at the root level and a fixed hierarchy of dense child nodes with progressively decreasing block sizes. The default configuration in OpenVDB, and only configuration in NanoVDB, is three levels deep with the fan-out-factors 32, 16, and 8, i.e. node sizes from root to leaf cover 4096^3, 128^3, and 8^3 voxels respectively. This configuration is denoted [Map, 5, 4, 3] in [Museth 2013], where the integers are $log2$ of the nodes fan-out-factors. The fact that VDB is shallow means that it supports fast random (*i.e.* coordinate-based) access to values. Furthermore, VDB allows for inverse tree-traversal, by means of node-caching, which in practice makes random-access $O(1)$. However, despite these attractive properties of VDB we found that it had several shortcomings when naively attempting to use it for ML applications on the GPU. Specifically, ML applications require more flexibility in terms of supporting complex high-dimensional data types, and the ML computations, e.g. sparse convolution, on the GPU are typically bandwidth-limited, which means random-access operations should be limited and data should be reused (cached) as much as possible.

3.2 VDB IndexGrids for ML Features

By design, standard VDB encodes data, e.g. `float` or `Vec3f`, directly into the tree structure, i.e. values and topology (*i.e.* sparsity pattern) are mixed. That is, the data types (typically templated) and their numerical values are intertwined (both in terms of code and actual memory layouts) with their spatial occupancy (topology) information, compactly represented with bit-masks. This is problematic when dealing with data of arbitrary type and dimension (*i.e.* ML features). It severely complicates code if each feature needs its own template specialization, and it is memory inefficient in cases when the sparsity (*i.e.* topology) is shared between multiple feature/data types. Ironically, VDB was originally designed to handle situations where both topology and values are dynamic, but in ML we often found that topology is fixed, whereas data (payload) change in terms of type, value and dimension.

To overcome these inefficiencies we developed a completely new grid type in NanoVDB, dubbed IndexGrid, which effectively *separates topology and values* encoded in VDB trees. Whereas the core idea behind IndexGrid is arguably simple, its efficient implementation is not. The idea is for the tree to return keys in the form of indices into external linear arrays of values as opposed to the data values, as is the case of standard VDB. In other words, the IndexGrid exclusively encodes topology information that is used to access any number of types of data values that resides in "sidecars", i.e. separate memory blocks. This seemingly trivial technique greatly simplifies code and allows for a single IndexGrid to be reused with multiple data (features), which amortizes the cost of encoding shared topology.

There is another less obvious benefit to this IndexGrid, which is related to the fact that all nodes in VDB are fundamentally dense blocks, e.g. a leaf node traditionally encodes $8^3 = 512$ values, regardless of the occupancy of the sparse data. A naive implementation of an IndexGrid indices all 512 leaf values, but there is a much more memory efficient version of the IndexGrid that only indices the sparse (denoted active) leaf values. This significantly reduces the memory footprints of the sparse data (features stored externally as sidecars) since it eliminates the need to explicitly store values in leaf nodes that represent background values (as opposed to inserted active values). We achieve this sparse (vs dense) indexing of active values with the following highly efficient code.

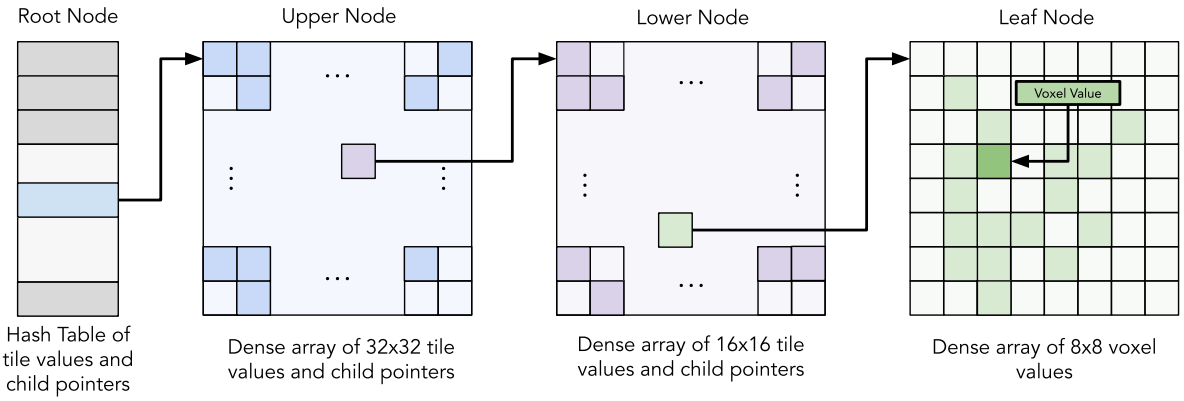

Fig. 2. Illustration of a 2D VDB tree with the default configuration used in both OpenVDB and NanoVDB, which are 3D. The depth is 4, and the top-down fanout-factors are 32, 16, and 8 respectively. Values reside at all levels of the tree, and are denoted voxel values in the leaf node and tile values everywhere else.

```
1  class LeafNode {uint64_t mOffset,mPrefixSum,mBitMask[8];
2  ...
3  int off(int i,int j,int k){return (i&7)<<6|(j&7)<<3|k&7;}
4  uint64_t getValue(int i, int j, int k) {
5      int m = this->off(i, j, k), n = m >> 6;
6      uint64_t w = mBitMask[n], mask = 1 << (m & 63);
7      if (w & mask == 0) return 0;// index to background
8      uint64_t sum = n-- ? mPrefixSum >> (9*n) & 511 : 0;
9      return sum + mOffset + countOn(w & (mask-1));
10 }};
```

Listing 1. C++ code that computes sparse indices from coordinates.

```
1  int lower::off(int i, int j, int k) {
2      auto a = [](int n){return (n & 127) >> 3;};
3      return a(i) << 8 | a(j) << 4 | a(k);// 0,1,..,16^3-1
4  };
5  int upper::off(int i, int j, int k) {
6      auto a = [](int n){return (n & 4095) >> 7;};
7      return a(i) << 10 | a(j) << 5 | a(k);// 0,1,..,32^3-1
8  };
```

Listing 2. C++ code that computes offsets in nodes from coordinates.

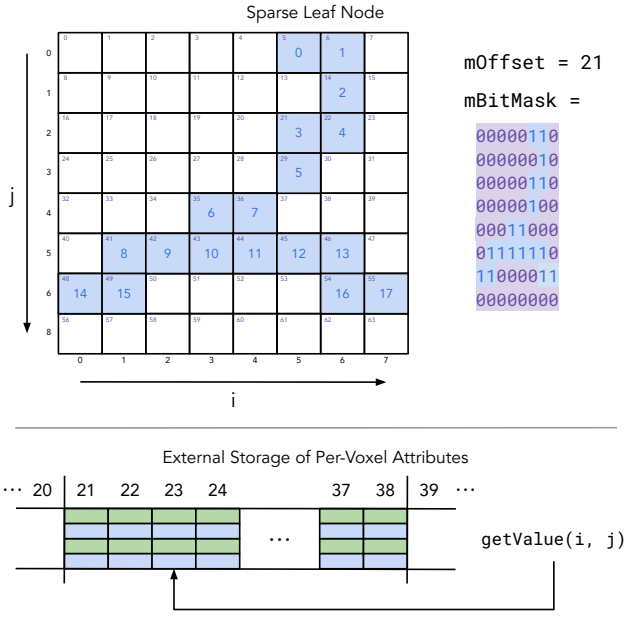

Fig. 3. Illustration of dense local indexing (0-63) vs sparse global indexing (21-38) in a 2D leaf node of size $8^2 = 64$. The sparse global indexes correspond to offsets into a dense tensor of per-voxel attributes illustrated at the bottom as one column per attribute, allocated as sidecars to the IndexGrid.

In words, this compact code computes the linear offset from the signed coordinates `i,j,k` to values stored in an external array, starting at `mOffset`. Specifically, $m \in \{0, 511\}$ is the linear index inside the leaf node, $n \in \{0, 7\}$ is the offset into the 64-bit array `mBitMask` that indicates which of the dense 512 values are active,*i.e.* on. w is the 64-bit word in `mBitMask` that contains `i,j,k`, and `mask` masks out all higher bits in w, so as to only consider active states of values proceeding `i,j,k`. Line 6 return a zero offset if `i,j,k` maps to an inactive value, which corresponds to a unique background index. If w is not the first word in `mBitMask`, then line 7 extracts the preceding active value count encoded in the $7 * 9$ bits of `mPrefixSum` as prefix sums of the first 7 64-bit words in `mBitMask` (excluding last word). Finally line 8 computes the number of on bits in w, excluding any bits that comes after `i,j,k`.

Despite the apparent complexity of this code, it is very fast since it includes few (2) conditionals, and fast operations like bit and intrinsic function calls (e.g. `countOn`). Also, note that each leaf node in an IndexGrid only requires 80 bytes to encode all indices as opposed to over 4KB in `nanovdb::LeafNode<uint64_t>`, i.e. a memory reduction of over 50× relative to a naive indexing approach. As mentioned above, IndexGrid also introduces memory saving by reusing topology for multiple data and avoiding explicitly storing inactive, i.e. background, values, which is especially important for sparse data.

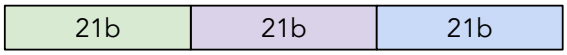

Fig. 4. Breakdown of the 64 bit key constructed from voxel coordinates i, j, k in step 2 of our build algorithm. The lower 21 bits (blue) encode the signed k coordinate right-shifted $5+4+3 = 12$ bits, the next 21 bits (purple) encode the signed j coordinate right-shifted 12 bits, and the upper 21 bits (green) encode the signed i coordinate right-shifted by 12 bits.

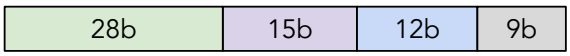

Fig. 5. Breakdown of the unique 64 bit key constructed from voxel coordinates in step 5 of our build algorithm. The lower 9 bits (gray) encode the offset local into leaf nodes ($2^9 = 512 = 8^3$), the next 12 bits (blue) encode the local offsets into lower nodes ($2^{12} = 4096 = 16^3$), the next 15 bits (purple) encode the local offsets into the upper nodes ($2^{15} = 32768 = 32^3$), and finally the remaining upper 28 bits (green) encode tile ID (0 to total tile count - 1) into the hash table of the root node. Note, this imposes a limit of $2^{28} = 268435456$ entries in the root hash table, which is extremely unlikely to be exceeded since each entry corresponds to a child node of the root that spans an index domain of size 4096^3 voxels.

3.3 GPU Accelerated IndexGrid Construction

While shared topology information is efficiently handled with our new IndexGrid, there is still a need to dynamically change the sparsity patterns, *e.g.* during morphological dilation, which is essential when building Level-of-Detail (LOD) hierarchies for sparse CNNs. In OpenVDB, dynamic topology is handled with allocation on insertion on the CPU, whereas in standard NanoVDB the topology is assumed to be fixed on both the GPU and CPU. Thus, there is a need to develop new techniques for building IndexGrids on the GPU, in order to rapidly build grids with different topology. A high-level description of our novel algorithm that builds IndexGrids from coordinates is as follows:

1: Input: N signed voxel coordinates `i,j,k`.
2: Define N 64-bit keys in Fig. 4:
```
a = k >> 12 | (j >> 12) << 21 | (i >> 12) << 42
```
3: Full radix sort of N keys a.
4: Run-Length-Encode N keys a.
5: For i,j,k in each run, $M = 0, 1, \ldots$, define keys in Fig. 5:
```
b  = M << 36 | upper::off(i,j,k) << 21 |
     lower::off(i,j,k) << 9 | leaf::off(i,j,k)
```
6: Partial radix sort of keys, b, associated with run M.
7: Upper node count is number of runs, M, in a.
8: Lower node count is number of unique keys `b>>21`.
9: Leaf node count is number if unique keys `b>>9`.
10: Use node counts to allocate device memory in Fig 6.
11: Build NanoGrid using the following top-down steps:
11:1: Use a to register upper nodes into the root table.
11:2: Use `b>>21` to register lower nodes into its parent nodes.
11:3: Use `b>>9` to register leaf nodes into its parent nodes.
11:4: Use `b&511` to register active voxels into `leaf::mBitMask`.
12: Optionally add ML features as blind data in Fig 6.

Note that despite the complexity of the build algorithm outlined above, it is fast since virtually all steps can be performed in parallel on the GPU, and high-performance implementation of both radix sort and run-length-encoding are available in CUDA's CUB library[Merrill 2015]. In fact, this build algorithm allow us to construct an IndexGrid from millions of voxel coordinates in a few milliseconds.

3.4 Hierarchical DDA for fast Ray-Marching of VDB

Efficient ray marching of our underlying data structure is essential for multiple tasks typical in 3D deep-learning workflows, including differentiable rendering, unprojecting image features into a 3D volume, depth computation, debug visualization, and final rendering. To this end we are using an acceleration technique, dubbed HDDA, that employs a hierarchy of Digital Differential Analyzers (DDAs), which accelerate ray marching on each of the tree levels of a VDB. While this technique was previously announced in a technical talk [Museth 2014], we reiterate the process with more detail and technical elaboration in this paper.

The core idea of the HDDA is to associate four different DDAs with a given VDB tree structure – one for each of the node levels corresponding to the coordinate domains $\{4096^3, 128^3, 8^3, 1^3\}$. In other words, the first DDA rasterizes a ray at the granularity of the root's child nodes of size 4096^3 voxels, and the last (fourth) DDA rasterizes a ray at the fine voxel level. So, instead of slowly advancing the ray-marching at the voxel level, which would require numerous redundant random accesses into the VDB, we can use the coarser DDA in the hierarchy to effectively leapfrog through empty space. Given the fact that the VDB tree configuration is known at compile-time, we can use Template Meta-Programming to inline the logic of the four DDAs, resulting in a single high-performance HDDA. This significantly accelerates ray-marching and allows for real-time ray-tracing of VDB volumes on the GPU (typically marching millions of rays per second). We have illustrated this idea using two spatial dimensions in Fig. 7. Our benchmark demonstrates a runtime that is 1.5x to 3x faster than DDA in the dense bitfield and over 100x less memory footprint, as reported in § 4.1.2.

3.5 Accelerated Sparse Convolutional Operators

fVDB has been designed to be compatible with highly efficient algorithms for convolutional operations on sparse data, such as the Sorted Implicit Gemm (SpConv v2) paradigm used in TorchSparse++. We emphasize that leveraging such highly-tuned libraries in the context of our hierarchical, tree-based indexing structure is a straightforward exercise: fVDB is effectively a locality-optimizing mapping between a sparse collection of lattice indices and a one-dimensional, linear index space. Contrary to random hash-based maps, fVDB inherently provides the property that active indices that are geometrically proximate in the containing 3D lattice, will have high probability of also being proximate in linear index space. Conversely, active voxels corresponding to a contiguous sub-sequence of linear indices are highly likely to be geometrically clustered together in the containing 3D lattice. Other than this (favorable) inherent property of the fVDB indexing scheme, our data structure is drop-in compatible with implementations that originate from hash-based structures

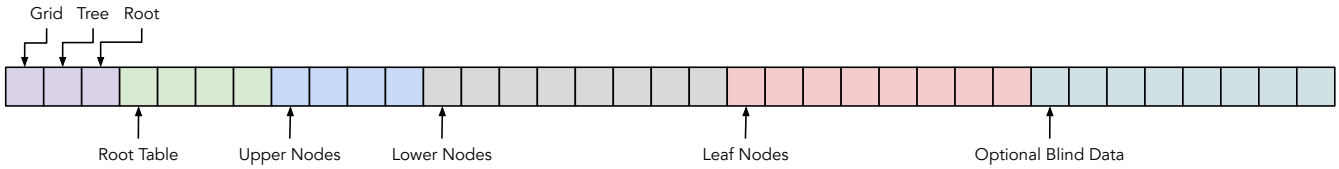

Fig. 6. **NanoVDB Memory Layout:** By design NanoVDB, unlike OpenVDB, has the following serialized memory layout. Note that in our application the trailing optional blind data could be ML features of any type or dimension. Alternatively those data could reside in the separate memory block.

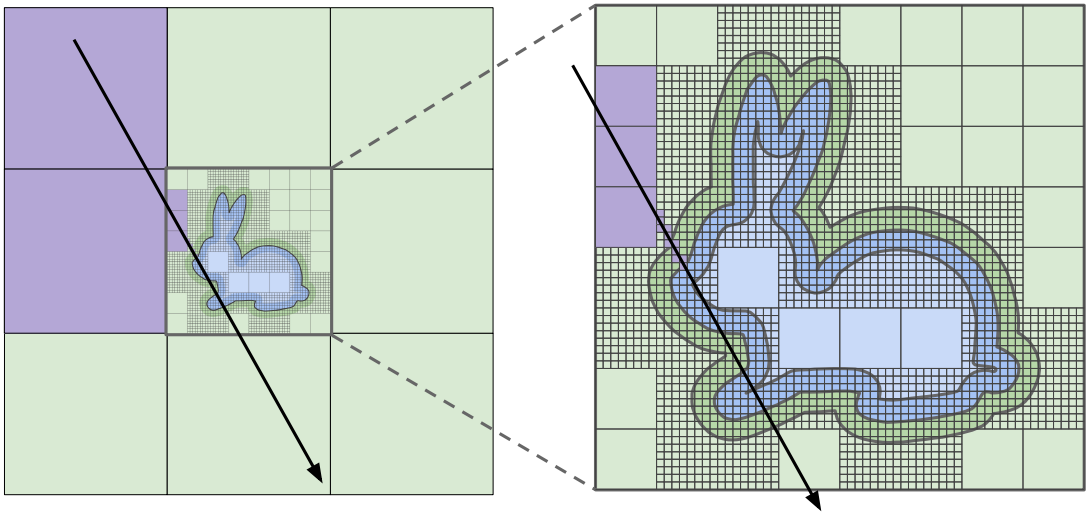

Fig. 7. HDDA: Hierarchical Digital Differential Analyser allows for fast ray-marching of our VDB tree structure. It works by employing four DDAs, one per tree level, that facilitates efficient leapfrogging through empty space. **Left**: 2D illustration demonstrating the use of three DDAs to quickly skip large constant (or empty) regions of space, represented by upper tiles. **Right**: 2D illustration of how two DDAs can accelerate ray-marching (violet squares represent tiles intersected by the ray).

(e.g. SpConv v2) by simply treating the linear index of each active voxel as a "hash key" (but with built-in locality properties). We have incorporated SpConv v2 into our operator toolkit and, as our micro-benchmarks reveal, we at minimum match the efficiency of TorchSparse++ at the operator level within our framework.

Even though SpConv v2 is trivially compatible with *f*VDB, we have identified a number of scenarios where a new design perspective on convolutional kernel design can provide even higher performance. Although we present the circumstances leading to this acceleration opportunity, and detail our proposed algorithmic design choices, we highlight that *f*VDB retains the ability to select the best applicable algorithm to match each case, including either the all-around performer SpConv v2, or our new kernels for those scenarios that warrant their use. Although we defer discussion of esoteric details of SpConv v2 to the related publications [Tang et al. 2023], we highlight that its design is motivated by the following objectives:

a) Minimization of wasted computation, in the form of MACs (multiply-accumulate operations); relative to dense convolution, wasted computation could be either due to sparse occupancy of the background lattice, or sparse presence of the (max 27) stencil "spokes" across different lattice locations where a convolution stencil is applied.
b) Maximization of regularity of operations; this typically manifests as an aspiration to perform the largest structured GEMM operation afforded by data layout and sparsity pattern.
c) Minimization (or elimination) of scatter operations, and spatial localization of gather operations.

These design objectives become much more difficult to reconcile in the presence of significant sparsity and geometric irregularity.

*Scenario 1: Low-depth convolutions (**Leaf**).* The first scenario where approaches striving for economy of computation might face diminishing returns is when the kernel is severely memory-bound. This possibility can easily materialize in the case of a convolution where both the input and output feature dimension is relatively low (e.g. not exceeding 8-16). As a tangible example: consider a convolution at TF32/FP32 precision with activation dimension of 8, and output dimension of 16. A *dense* convolution at those depths, applied to an N^3 grid requires streaming at minimum $96N^3$ bytes (assuming perfect caching), and the performance of $6912N^3$ operations. On an RTX 6000 Ada Generation GPU (peak memory bandwidth of $960GB/s$) this would require about 70TFLOP/s, which is an achievable compute density, to have this kernel be memory- rather than

compute-bound. The calculus is not so straightforward when we contemplate sparsity, but we have practically witnessed this operation being pronouncedly memory-bound even at (local) sparsity of as little as 15-20%. This is due to the inefficiency of necessary gather operations, the cost of indirection for accessing low-depth feature vectors, and the overhead of indexing data structures themselves. Additionally, even compute efficiency may be challenging due to the complexity of harvesting large-enough GEMM operations when the contraction dimension (8, in this example) is so shallow.

In light of this, we consider an alternative where we prioritize regularity over sparsity of computation, essentially tolerating a higher compute burden for the sake of more local structure. Specifically, we have implemented a kernel that performs *local densification* in GPU shared memory, at the level of an $8 \times 8 \times 8$ fVDB leaf node, and performs a fully regular and (locally) dense convolution within this window. In detail, we allocate space in shared memory for a locally densified copy of the input activations in a window of size $10 \times 10 \times 10$ straddling the leaf node, plus a one-voxel halo in its immediate neighborhood (a footprint of 31.25KB for 4-byte FP32/TF32 data, at feature width of 8). Likewise, the output of this operation is an $8 \times 8 \times 8$ buffer of 16-wide output feature vectors (footprint of 32KB) also stored in shared memory. We subdivide the 8^3 local domain into 32 $8 \times 2 \times 1$ subtiles, assign each of them to a warp (1024 total threads) and use $16 \times 16 \times 8$ WMMA tensorcore GEMMs (at TF32 precision with FP32 accumulate) within each warp to apply each of the 27 spokes of the stencil. Even though this paradigm clearly performs more computation than strictly necessary (foregoing sparity due to either voxel or stencil occupancy), the regularity of the computation in combination with the memory-bound nature of this scenario allows for superior performance (relative to our SpConv v2 default backend) in leaf nodes that have an occupancy of 20% or higher (all the way to an approximately 2.5x-3x advantage for a dense domain). It should also be noted that no auxiliary indexing structures are necessary for this kernel approach, all gather offsets are computed directly and efficiency from the (very lightweight) metadata of the core fVDB tree structure, taking advantage of amortization. Finally, due to the compact and local storage of all (output) feature vectors within a leaf node, the writeback of the convolution result into global memory occurs on a fully sequential memory range (all active indices within a leaf node are sequentially indexed).

*Scenario 2: High local occupancy convolutions (**Brick**).* The second scenario we target for a tuned approach is when the sparsity pattern exhibits high density in the vicinity of active indices (e.g. when on average every active index has more than 70-80% of its stencil neighbors as active), even though the domain is macroscopically sparse. Typical cases where this scenario materializes is when the active indices are predominantly clustered in a narrow band of small but nontrivial thickness (e.g. 2-3 voxels wide), and also on dense or semi-dense domains that are still targeted with our fVDB representation. In addition, we look for instances where such topology is coincident with moderate-to-high depth of input/output features (width of 32 or higher), when the kernel no longer is memory-bound as in scenario 1 above. For this case, we have implemented a solution that replicates the local densification paradigm, as above, but instead of this being performed at the granularity of an $8 \times 8 \times 8$ window, we focus on a kernel that monolithically produces the convolution output on a narrower $4 \times 2 \times 2$ window. Input activations are fetched on-demand from the spatial extent encompassing the $6 \times 4 \times 4$ window (including a 1-voxel halo) around the $4 \times 2 \times 2$ block. We have developed a custom tensorcore implementation of the convolution operation using the CuTe library [Thakkar et al. 2023] that achieves exceptionally high compute density (exceeding 70% peak compute bandwidth for moderate feature depths of about 32-64, and reaching above 90% for feature depths of 128 or higher) for the task of computing the locally-dense convolution on the $4 \times 2 \times 2$ output window. Any residual suboptimality in this case is due to inactive voxels at the scale of the $4 \times 2 \times 2$ window, or stencil spokes that are not present for any of the active voxels. In practice, we have observed that for occupancy patterns that exceed 60-70% on average across such windows, this implementation outperforms SpConv v2, with the most notable margin observed in dense or semi-dense domains that have even higher average occupancy.

*Scenario 3: Highly sparse topology, high feature depth (**LGGS**).* The last scenario where we have provided a custom implementation addresses the instance where the occupancy pattern is so sparse that on average every active index is expected to have no more than 4-5 active neighbors (out of 26 max). In addition, this has to be combined with relatively high feature depth, typically of 128 or above. This scenario is characteristic of LiDAR data, as those presented in SemanticKITTI [Behley et al. 2021; Geiger et al. 2012]. Although our default SpConv v2 implementation performs an adequate job at minimizing wasted MAC operations, the number of those may still exceed the essential MACs mandated by the stencil occupancy of active indices.

In principle, if our sole objective was to minimize wasted MACs, the traditional gather-GEMM-scatter paradigm provides a pathway to achieving this goal. However, the reasons why the straightforward implementation of this paradigm will typically underperform SpConv v2 is due to the need for several independent streaming passes over the input activations (one for each of the 27 stencil offsets), and due to the suboptimality of scattering results to global memory. We circumvent these concerns by taking the following steps:

(a) We block the gather-GEMM-scatter operation so that it is performed on a contiguous subsequence of output indices from the fVDB data structure, typically 64 indices at a time. Due to the locality of the fVDB mapping, those indices are expected to correspond to highly clustered geometric coordinates from one or more IndexGrid leaf nodes.

(b) Instead of scattering results to global memory, we use a temporary buffer in GPU shared memory as the destination of scatter operations on these 64 indices, which collect the contribution of each of the 27 stencil offsets within this block. At the end of the local computation, this result is sequentially copied back to global memory without the need of a scatter operation.

(c) For each of the 27 stencil offsets, we collect all input/output index pairs that are linked by this offset (such that the output index is within the range of the block being processed), and pack them

contiguously again in shared memory buffers. For each stencil offset, the input of this packed buffer is gathered from global memory (benefiting from locality across offsets). A GEMM operation is performed to produce the output, still in packed format, to be scattered (purely in shared memory) to the accumulation buffer that stores all 64 output vectors. We pad these packed collections of input/output index pairs to the next multiple of 16, for purposes of easy mapping to tensorcore-accelerated GEMM. This is the only source of wasted MACs, which is now limited to at most 15 MACs per block of 64 output indices (practically, the expected length of this padding is closer to 8 entries per 64 output indices).

Our benchmarks demonstrate a runtime that is approximately 25% faster than SpConv v2 (at feature length 128 or higher) for the single-scan point clouds of SemanticKITTI.

3.6 *f*VDB Framework Overview

At its core, *f*VDB exposes a set of differentiable deep learning primitives which operate a *minibatch* of sparse voxel grids. *i.e.* a set of multiple sparse voxel grids where each voxel contains some multi-dimensional tensor of data. To encode such a minibatch of grids, *f*VDB employs two classes: a `GridBatch` which represents a set of NanoVDB index grids (one per item in the batch) and a `JaggedTensor` which encodes a tensor of per-voxel features at each voxel in the minibatch. Internally, a `GridBatch` is simply a contiguous block of NanoVDB IndexGrids stored one after the other with some metadata to quickly access any grid in the batch. Below, we give a description of the `GridBatch` and `JaggedTensor` classes as well as a summary of the primary operators exposed to the programmer by *f*VDB .

3.6.1 `JaggedTensor` . In general, we cannot expect each grid within a minibatch to have the same number of voxels. Thus, *f*VDB must expose operations on *jagged* arrays of data. *f*VDB exposes the `JaggedTensor` class for this purpose. Conceptually a `JaggedTensor` can be thought of a list of tensors $[t_1, t_2, \ldots t_B]$ where each tensor t_i has shape $[N_i, *]$ *i.e.* each tensor has different first dimension but matches in subsequent dimensions. For example, if a `JaggedTensor` represents per-voxel attributes in a batch of grids, then N_i will be the number of voxels in the i^{th} grid in the batch. Under the hood, *f*VDB efficiently encodes these tensors contiguously in memory to enable fast operators on them. Specifically, a `JaggedTensor` consits of three parts:

a) `jdata` which is a $[N_1 + \ldots + N_B, *]$-shaped tensor equivalent to concatenating $t_1, \ldots t_B$ along their first axis
b) `joffsets` which is a $[B, 2]$-shaped tensor such that `joffsets`$[i, :]$ is the start and end tensor t_i in `jdata`
c) `jidx` which is a $[N_1 + \ldots N_B]$-shaped tensor such that `jidx`[i] is the index (from 0 to $B-1$) of the i^{th} element in `jdata`

Figure 8 shows this layout pictorially. Note that `joffsets` and `jidx` are also available for `GridBatch` since these represent a jagged collection of voxels. In the subsequent paragraphs, a tensor shape of -1 refers to a jagged dimension. For example, a `JaggedTensor` containing the voxel coordinates of a `GridBatch` would have shape $[B, -1, 3]$.

3.6.2 List of Operators. *f*VDB supports a range of differentiable operators on minibatches of sparse voxel grids of tensor data. These operators are written in CUDA and C++ and interoperate with PyTorch. Here we give a high-level description of the major operators in *f*VDB. A concise summary of these are given in Table 1.

Grid Construction Operators. A `GridBatch` in *f*VDB can be created from a `JaggedTensor` of point clouds; voxel (ijk) coordinates; triangle meshes (the set of voxels which intersect a mesh); other `GridBatches` via padding, coarsening, or subdivision; and from dense grids with masks.

Sampling Operators. A common operator is to sample tensor values on a voxel grid at a set of query points $Q \in 2^{\mathbb{R}^3}$. *f*VDB provides differentiable sampling operators which accept a `GridBatch` G, a `JaggedTensor` of per-voxel features Z with shape $[B, -1, *]$, and a `JaggedTensor` of query points Q with shape $[B, -1, 3]$. These operators return a set of features Z_Q sampled at each point $q \in Q$ using Trilinear or Bézier interpolation.

Splatting Operators. *f*VDB supports splatting data stored at points onto a grid using Trilinear or Bézier interpolation. These operators accept a `GridBatch` G, a `JaggedTensor` P of points, and a `JaggedTensor` Z of per-point features. They produce a `JaggedTensor` of features (one per voxel in G) by splatting the feature at each point onto the neighboring voxels.

Convolution, Pooling, Upsampling, and Attention. *f*VDB supports sparse convolution via a novel accelerated implementation (Section 3.5). The convolution operator accepts a `GridBatch` G_{in}, a kernel K, and a `JaggedTensor` of features Z_{in} and produces a `GridBatch` G_{out}, and `JaggedTensor` Z_{out} by performing sparse convolution. We further support average and max pooling operators on a `GridBatch` and `JaggedTensor` pair as well as an upsampling operator which upsamples a `GridBatch` and `JaggedTensor` of features via subdivision and nearest neighbor sampling. *f*VDB supports attention by calling out to Flash Attention [Dao et al. 2022] on a `JaggedTensor` .

Ray Marching. *f*VDB comes with a number of operators for intersecting rays with grids. These include enumerating the set of voxels along a ray, parameterized by intervals of t along a ray which intersect a grid; finding the intersection between rays and the level set of an implicit function stored on a grid; and volume rendering. Ray marching operations are implemented using a hierarchical DDA algorithm outlined in Section 3.4.

4 EXPERIMENTS

In this section, we demonstrate the effectiveness of *f*VDB through a series of benchmarks and qualitative examples of use cases. Our experiments demonstrate that our framework successfully covers a broad variety of use cases and operations, while achieving state-of-the-art runtime performance and memory efficiency. First, we perform micro-benchmarks of the most important operators in *f*VDB, comparing them against corresponding state-of-the-art operators in other sparse deep learning frameworks in terms of both memory usage and speed. Next, we run a macro-benchmark showing that *f*VDB remains performant in the real-world use case of training a sparse convolutional neural network (CNN). Finally, we demonstrate

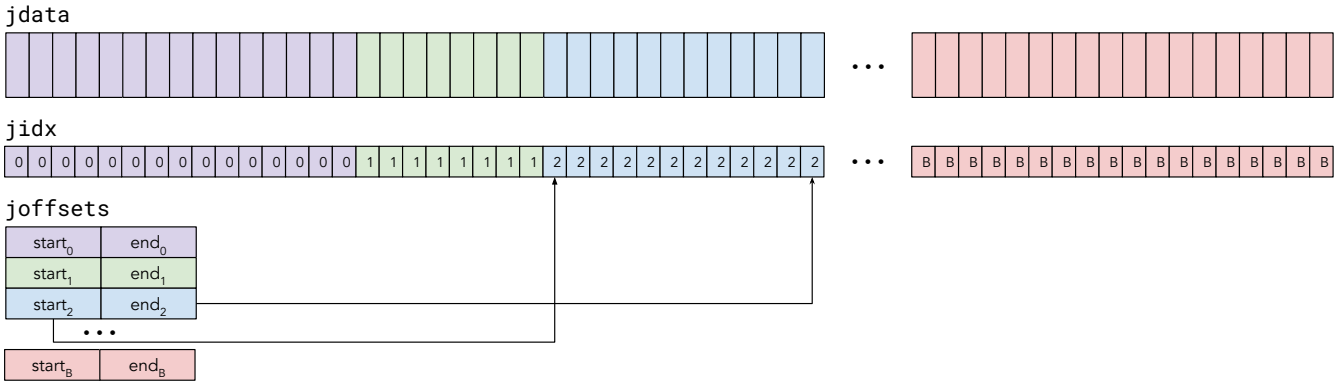

Fig. 8. Conceptually, a JaggedTensor is a list of tensors whose first dimension differs between each list item and all other dimensions match. Internally, JaggedTensor concatenates all the tensors in the list into one dense tensor (jdata) and stores two auxiliary pieces of metadata: jidx , which specifies the index in the list that each tensor is in and joffsets , which specifies the start and end indexes of each list item in jdata .

the utility of fVDB by showing its use in several key applications on high-resolution 3D data. These applications include 3D reconstruction from points, semantic completion, 3D shape generation, and neural radiance field rendering.

4.1 Micro-benchmarks

We evaluate the runtime performance and memory efficiency of the core primitive operations in fVDB, comparing against operators available in other frameworks. First, we compare the speed and memory footprint of our core algorithm for index grid construction, which converts a list of ijk integer or xyz point coordinates to a VDB IndexGrid on the GPU. All grid construction operations (*e.g.* from meshes) make use of this build algorithm, so this is a crucial benchmark. Second, we evaluate the performance of our HDDA ray marching algorithm, which is the backbone of all ray-tracing algorithms in the framework. Finally, we evaluate the performance of our convolution operator on a novel benchmark consisting of a variety of real-world examples spanning different sparsity patterns and channel depths.

Each data-point for the experiments on grid construction and convolution, sections 4.1.1 and 4.1.3 respectively, were averaged from the 4 best runs out of 5 runs to mitigate outliers. Between each run we made sure to clear the device's L2 cache to make sure that no framework was benefiting from the uneven advantages of a warm cache. The experiment in sections 4.1.3 was run on a machine with an AMD 7950X 16-Core CPU and GeForce RTX 4090 GPU, with 128GB of host memory and 24GB of device memory. The experiment in section 4.1.1 was run on a machine with an AMD 3975WX 32-Core CPU and RTX 6000 Ada Generation GPU, with 128GB of host memory and 48 GB of device memory.

The experiment on ray marching in section 4.1.2 was performed by averaging the results of 1,000 runs where each run consisted of casting 1,024 rays. This experiment was run on a machine with an AMD 3975WX 32-Core CPU and GeForce RTX 3090 Ti GPU, with 128GB of host memory and 24GB of device memory.

4.1.1 IndexGrid Construction. The IndexGrid construction algorithm, detailed in Section 3.3, converts a list of ijk integer or xyz point coordinates into a VDB IndexGrid on the GPU. It forms the backbone of all grid constructions in fVDB, while also acting as a means to initialize sparse grids. We evaluate the runtime performance and memory footprint of our grid construction algorithm against those in TorchSparse++ [Tang et al. 2023], Minkowski Engine [Choy et al. 2019], and spconv [Contributors 2022] by constructing a grid with random points sampled from a normal distribution. Figure 9 shows the maximum memory usage and runtime when constructing a grid from an increasing number of input points. Our method is comparable to baselines in terms of runtime performance while offering significant advantages in terms of memory efficiency. We remark that the three baseline approaches run out of memory long before ours. Thus, fVDB can process much larger input data than current state-of-the-art sparse DL frameworks.

4.1.2 Hierarchical DDA. We profile our HDDA ray marching on a 3-voxel-wide narrow-band level set of the Stanford bunny extracted at various (effective) resolutions ranging from 32^3 to 1024^3. The ray marching axis-aligned bounding box of the bunny is 1.2x of its tight axis-aligned bounding box and all rays are always marched through the entire volume constructing intervals along the ray. We compare our algorithm with the widely used NerfAcc [Li et al. 2023] library (e.g. by NeRFStudio [Tancik et al. 2023]) for ray marching and volume rendering. NerfAcc provides a highly optimized DDA over a dense binary grid implemented in CUDA. Table 2 shows that fVDB constantly achieves 1.5x to 3x faster runtimes than NerfAcc while maintaining a comparable or lower (up to 100x at high resolutions) memory footprint. The same conclusion applies to the real-world scene as well, where in the large-scale NeRF application (§4.3.3) we observe 1.3x faster ray marching with fVDB comparing to NerfAcc, and 30x less memory footprint at effective 1024^3 resolution on the Laguna Seca Raceway scene.

4.1.3 Sparse Convolution. We profile our core convolution operators across a range of different feature depths: A low-depth regime with input depth of 8 and output depth of 16, a medium depth case

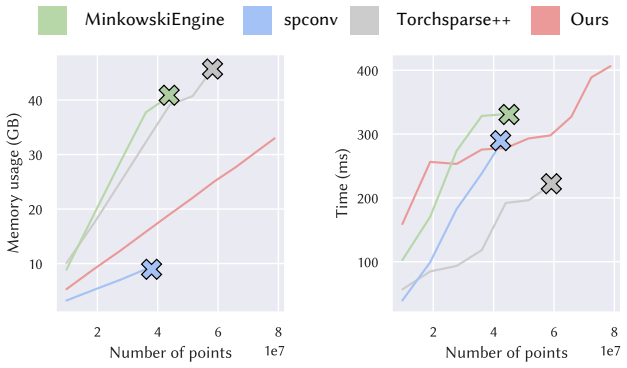

Fig. 9. **Left:** Maximum memory utilization (y-axis) when constructing a grid for a given number of coordinates (x-axis). While baselines run out of memory quickly, ours (*f*VDB) remains much more memory efficient. **Right:** Runtime (y-axis) required to construct a grid for a given number of coordinates (x-axis). While our method is not always the fastest, it remains competitive with baseline approaches while scaling to much larger inputs. The cross marks on MinkowskiEngine and Torchsparse++ indicate an out of memory error, while those on spconv, where we otherwise seem to be within memory limits, indicate an illegal memory access exception that occurs within the framework at these point counts.

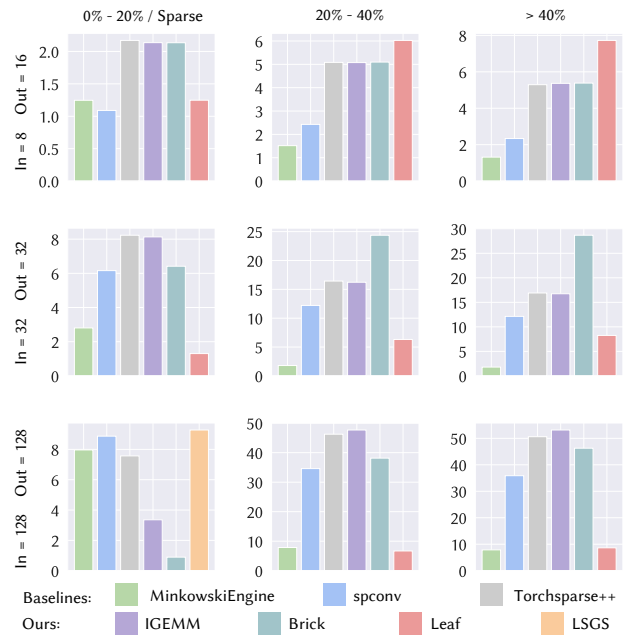

Fig. 10. Micro-benchmarks of our convolution operator. The three columns represent increasing leaf node-level occupancy reflecting grid sparsity, while the three rows denote different input (In) and output (Out) channel sizes. The bars represent speed measured in "effective" TFLOPs [Section 4.1.3]. To incorporate extreme sparse cases we use the KITTI dataset for the lower-left subplot.

Table 1. Feature comparison between our *f*VDB and four alternative sparse DL frameworks that represent state-of-the-art.

	*f*VDB	Torchsparse++ [Tang et al. 2022]	Minkowski Engine [Choy et al. 2019]	SpConv [Contributors 2022]	NerfAcc [Li et al. 2023]
Grid Construction Methodology					
Coordinate Lists	✓	✓	✓	✓	-
Dense Grids	✓	-	✓	✓	✓
Pointclouds	✓	✓[1]	✓	✓[1]	-
Meshes	✓	-	-	-	-
Dual Grid	✓	-	-	-	-
Grid Topology Feature Set					
Spatial Dimensions	3	3	Arbitrary	3	3
Subdivision + Coarsening	✓	✓	✓	✓	-
Adaptive Subdivision	✓	-	-	-	-
Device Accelerated Grid Building	✓	-	-	-	-
Mutable Grids	✓	-	-	-	-
Zero-Copy Grid Cropping	✓	-	-	-	-
Indexing and Sampling					
Point → Grid Sampling	✓	-	✓	-	-
Trilinear Interpolation	✓	-	✓	-	-
Bézier Interpolation	✓	-	-	-	-
Gradient Sampling Support	✓	-	✓	-	-
Grid → Point Splatting	✓	-	-	-	-
Trilinear Interpolation	✓	-	-	-	-
Bézier Interpolation	✓	-	-	-	-
Accelerated Spatial Neighbour Indexing	✓	-	-	-	-
Geometry Functionality					
Point/Voxel Intersections	✓	-	-	-	-
Cube/Voxel Intersections	✓	-	-	-	-
Marching Cubes Mesher	✓	-	-	-	-
Raytracing Feature Set					
Ray Sampling	✓	-	-	-	✓
Implicit Field Intersection	✓	-	-	-	✓
HDDA Device-Accelerated Raytracing	✓	-	-	-	-
ML Operators					
Sparse Convolution	✓	✓	✓	✓	-
Pooling	✓	-	✓	✓	-
Flash Attention	✓	-	-	-	-
Gaussian Splatting	✓	-	-	-	-
Interoperability					
PyTorch Extension	✓	✓	✓	✓	✓
Interoperable Volume Format	✓	-	-	-	-
Modeling/Manipulation Toolset	✓	-	-	-	-
Shared Datamodel w/DCC Applications	✓	-	-	-	-
Shared Datamodel w/Industry Renderers	✓	-	-	-	-

[1] Supported via quantization of points and index-based construction

Table 2. Comparison between HDDA ray marching in *f*VDB and DDA ray marching in NerfAcc [Li et al. 2023] on the 3-voxel-wide shell of the Stanford bunny. Our approach consistently outperforms that of NerfAcc by 1.5x to 3x on runtime while also maintaining up to 100x lower GPU memory footprint.

Grid Resolution	32^3	64^3	128^3	256^3	512^3	1024^3
Rays / Sec (M)						
NerfAcc	2.57	2.46	2.09	1.41	0.82	0.47
*f*VDB	3.77	3.40	2.81	2.24	1.83	1.43
GPU Mem. (MB)						
NerfAcc	0.24	0.41	2.15	16.3	129	1028
*f*VDB	0.38	0.37	0.40	0.79	2.46	8.85
Cell Intersections/Ray	0.71	0.72	0.73	0.78	0.70	0.66

with input and output depths of 32, and a high-depth scenario with input and output depths of 128. Orthogonal to feature depth, we examine three different degrees of sparsity:

a) a highly sparse regime leading to voxel occupancy (at the IndexGrid leaf node level) below 20%, harvested from typical single-scan LiDAR datasets of rasterized point clouds [Behley et al. 2021]
b) a case of moderate leaf node-level occupancy of 20-40%, originating from rasterized surfaces, and
c) a case of higher density stemming from rasterization of volumetric data with nontrivial codimensional thickness, with leaf node-level occupancy in excess of 40%

The performance plots in Figure 10 include four implementations available in our framework:

a) an adaptation of SpConv v2 (labeled *IGEMM*) that employs our tree-derived indexing scheme instead of a spatial hash
b) local densification at the leaf-node level (Scenario 1 in Section 3.5; labeled *Leaf* in the figure)
c) local densification at a $4 \times 2 \times 2$ "brick" (Scenario 2 in Section 3.5; labeled *Brick* in the figure)
d) the shared-memory Local Gather-GEMM-Scatter paradigm of scenario 3 in Section 3.5 (labeled *LGGS* in the figure); this last option is only leveraged for high-depth convolution operations

As can be surmised from Figure 10, these four approaches allow us to select an operator implementation that is the most competitive to alternatives (i.e. those not incorporated as possible backends in fVDB) in each case. We note that in our experiments, optimizations beyond the *IGEMM* baseline were deployed when appropriate as part of the inference pipeline only; for training we defaulted to the *IGEMM* option for simplicity and as to avoid further specialization of the gradient computation for the filter coefficients.

Our benchmark also indicates the TFLOPS achieved by the top performer in each instance. This is an "effective" TFLOPs figure that reflects the method's degree of success in leveraging both spatial sparsity, and stencil sparsity (e.g. avoiding, to the degree possible, unnecessary multiply-and-accumulate (MAC) operations for stencil weights that are absent at specific grid locations). We compute this "effective TFLOPS" figure by counting the bare minimum number of operations essential for the stencil application, excluding from this count operations that would be associated with null weights. These numbers should be contrasted with the architectural ceiling of 73TFLOPS (or 82.6TFLOPS with a boost clock) on the RTX 4090 platform used in these experiments.

4.2 Macro-benchmarks

4.2.1 Full Network Inference. We benchmark the end-to-end performance of fVDB -based network inference. To this end, we leverage the generative backbone of XCube from [Ren et al. 2023]. Such a backbone has a typical encoder-decoder structure and is representative for sparse U-Net designs by first applying a set of downsampling operations to reduce spatial resolution and then upsampling to the original scale. Our dataset is based on a voxelized version of the KartonCity [kar 2023] dataset containing 500 representative samples, where we uniformly pick spatial resolutions from 256, 512, and 1024. This dataset contains dense geometry of a synthetic city that is suitable for generative tasks. Detailed speed comparison on different configurations of the network are shown in Figure 11. We consistently perform better than the state-of-the-art baselines under different spatial resolutions and channel sizes. Our results were averaged from the 4 best runs out of 5 runs to mitigate outliers. Between each run we made sure to clear the device's L2 cache to make sure that no framework was benefiting from the uneven advantages of a warm cache. The experiment was run on a machine with an AMD 7950X 16-Core CPU and GeForce RTX 4090 GPU, with 128GB of host memory and 24GB of device memory.

4.2.2 Neural Radiance Fields. We run the full end-to-end neural radiance fields training and testing session based on a reference implementation of Instant-NGP (iNGP) [Müller et al. 2022]. In order to query the color of a sampled ray, one would first perform ray marching through the scene to obtain samples close to the scene surface. The features at the sample positions are then retrieved and volume rendered to aggregate the final color. In [Müller et al. 2022], a cascade of binary grids of varying voxel sizes is used to represent the rough sparsity of the scene. By replacing the cascaded grid structure with the fVDB grid representation, we can accelerate the process of ray marching using the HDDA algorithm as introduced, while benefiting from the modest memory consumption provided by the VDB data structure. We run the neural radiance fields on a GeForce RTX 4090 GPU on one scene in the Waymo Open Dataset [Sun et al. 2020]. The training speed of ours compared to iNGP is 26.1it/s vs. 26.4it/s, while the inference speed of ours compared to iNGP is 1.90FPS vs 1.62FPS. As fVDB is initialized from LiDAR point clouds and offers more precise locations of the samples, we reached a test PSNR of 27.07, in comparison to 25.89 for iNGP.

4.3 Example Applications

We demonstrate that fVDB is a practical tool for building real-world 3D deep learning applications. Here we present several applications of fVDB , some of which are reimplementations of published works. These include large scale surface reconstruction from point clouds using NKSR [Huang et al. 2023], high resolution hierarchical object and scene generation using XCube [Ren et al. 2023], large-scale Neural Radiance Fields, and Deep-Learning based simulation super-resolution.

4.3.1 Large-scale Surface Reconstruction. NKSR [Huang et al. 2023] uses a sparse voxel hierarchy to encode a neural field of features which are used to perform a learned kernel ridge regression to solve a variational surface reconstruction problem from oriented point clouds. NKSR achieves state-of-the-art reconstruction and generalization results. We fully re-implemented NKSR using fVDB replacing the convnet with our implementation, the meshing with our marching cubes implementation, and implementing a batched Kernel Ridge Regression solver as an fVDB C++ extension. We remark that this extension is a single file consisting of a few hundred lines of code which only depends on PyTorch and fVDB. Figure 12 shows a mesh reconstructed using our implementation from 350 million input points. This reconstruction took 2 minutes on 8 V100 GPUs.

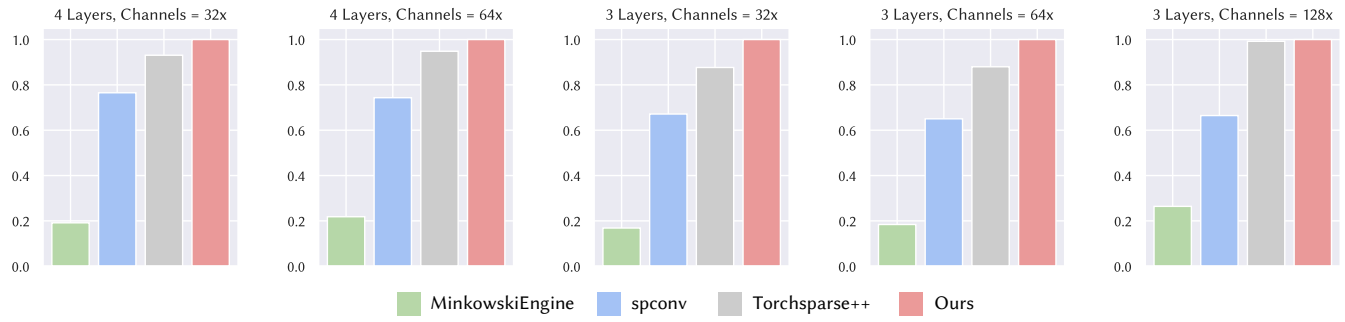

Fig. 11. End-to-end speed comparison of *f*VDB with state-of-the-art sparse frameworks under different configurations of the XCube [Ren et al. 2023] backbone. Runtime speed is normalized over the best model (always ours in this case), and the higher the better. Results are averaged over 10 runs.

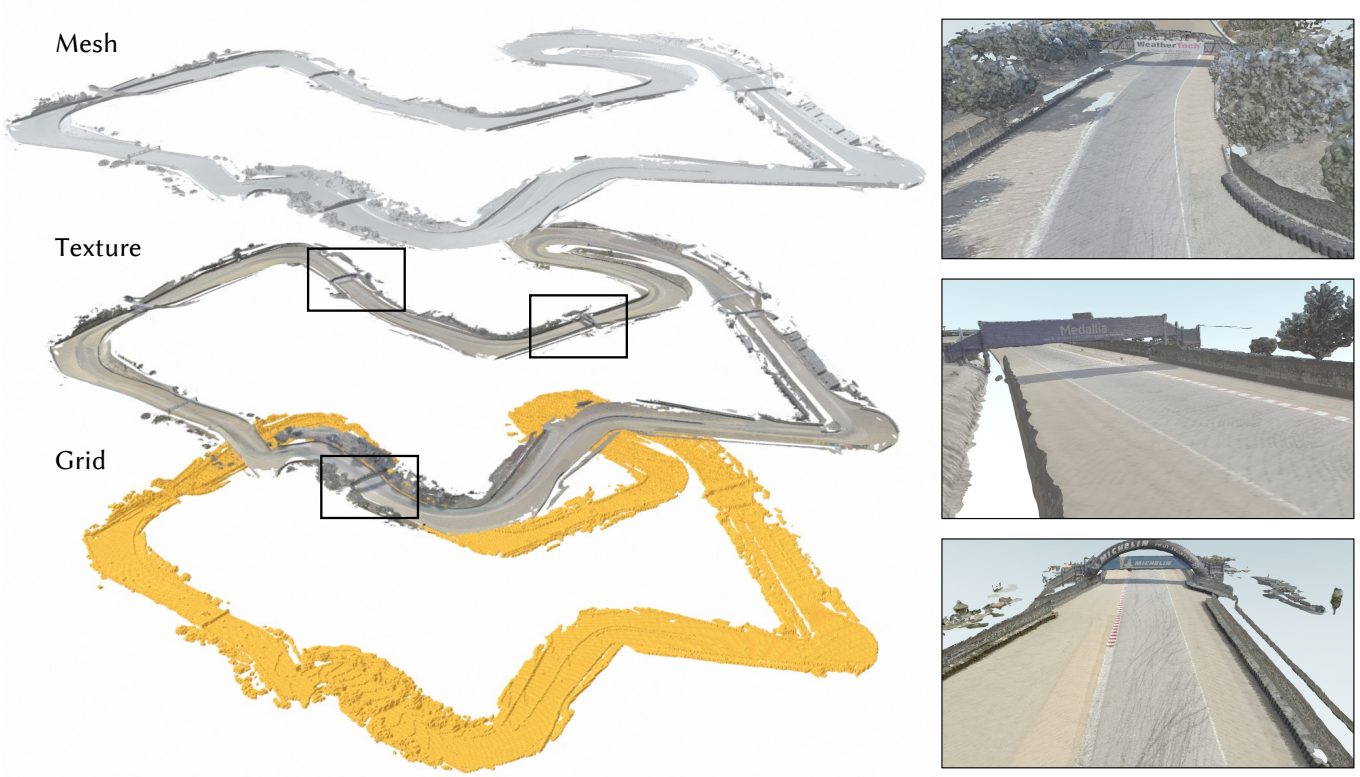

Fig. 12. *f*VDB helps state-of-the-art surface reconstruction models to scale to larger scenes spanning kilometers. Textures can be re-projected from images in a faster way with the help of our rendering operators.

4.3.2 3D Generative Models. We used *f*VDB to re-implement XCube [Ren et al. 2023], a 3D generative model for high-resolution voxel hierarchies of objects and scenes. XCube benefits directly when using *f*VDB to enable it to train on datasets with substantially larger footprints and higher spatial resolution while consuming less GPU memory. With the support of *f*VDB, XCube can be scaled up to spatial scale of 100m × 100m at 10cm resolution. Figure 13 demonstrates unconditional generation of high-resolution 3D objects trained using the Objaverse [Deitke et al. 2023] dataset and large-scale outdoor scenes trained on the Waymo [Sun et al. 2020] dataset.

4.3.3 Large-scale Neural Radiance Fields. *f*VDB can be used to support large-scale Neural Radiance Fields by providing a memory efficient acceleration structure for spatial skipped ray marching. Figure 14 provides two showcases of this application including a 1km squared area capture of the Laguna Seca Raceway and a

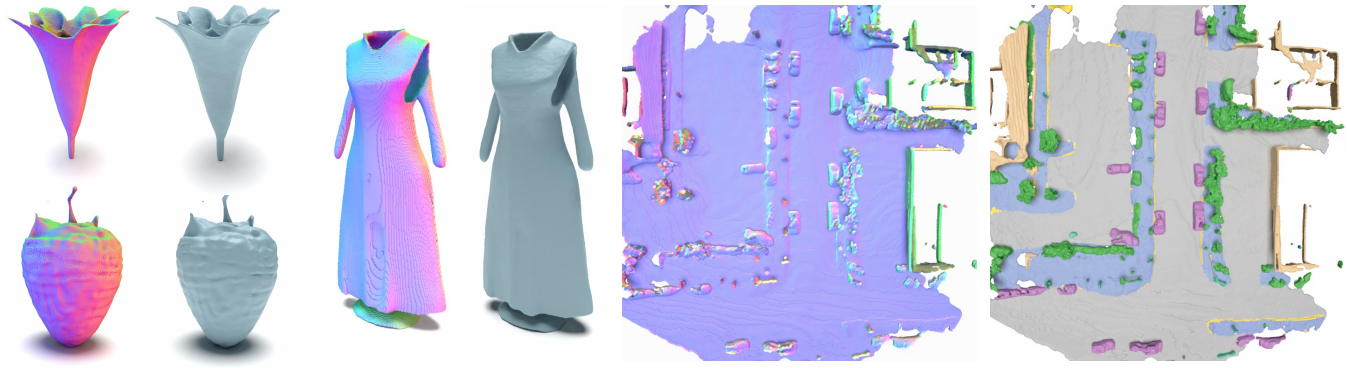

Fig. 13. fVDB helps push the limit of 3D generative models in terms of resolution and scale. With less memory usage and faster speed, we generate high-resolution 3D objects (512^3) and large-scale scenes (1024^3). We provide the generated sparse voxel grid colored by normal and the extracted mesh for each sample.

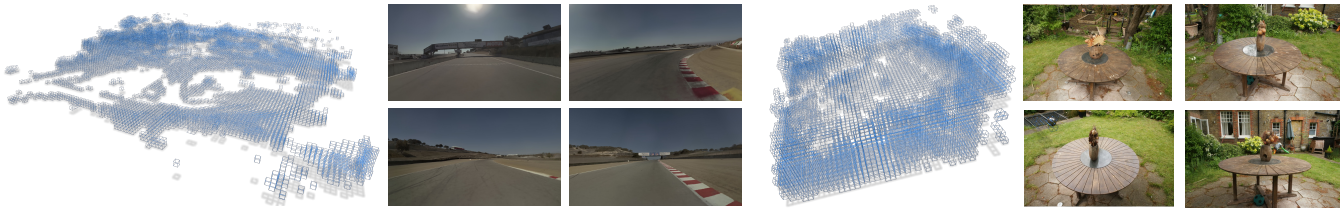

Fig. 14. fVDB can be used to support large-scale Neural Radiance Fields (NeRF) training and rendering by providing a memory efficient acceleration structure for spatial skipped ray marching. Here we showcase the fVDB grid and NeRF renderings on two scenes, where the left one is our capture of the Laguna Seca Raceway (1km squared area) and the right one is the Garden scene from the Mip-NeRF 360 dataset [Barron et al. 2022].

standard Garden scene in the NeRF literature from Mip-NeRF 360 dataset [Barron et al. 2022].

4.3.4 Simulation Super-Resolution. fVDB can enable novel applications of super-resolution techniques to inherently sparse, 3D data such as those produced by physical simulations which operate in unbounded domains. Previous approaches can be memory constrained and computationally prohibitive for large domains if approached with dense data structures and operators. Figure 15 shows preliminary results of work we are currently undertaking which trains fully convolutional super-resolution networks such as DCSRN [Chen et al. 2018] and 3D-FSRCNN [Mane et al. 2020] with operators implemented in fVDB. Currently in development are super-resolution models for several simulation domains including muscle and skin dynamics as well as fluid simulations.

5 CONCLUSION AND FUTURE WORK

We presented fVDB, a novel GPU-optimized framework for deep learning on large-scale 3D data. Our framework includes a broad set of novel differential primitives which can be used to build deep-learning pipelines for a wide variety of 3D tasks. These primitives include GPU accelerated grid building, ray marching, convolution, sampling, splatting, etc. Furthermore, fVDB has a significantly more comprehensive suite of features than existing frameworks, runtime performance that is at-par or superior to state-of-the-art and memory efficiency that exceeds state-of-the-art by a large margin. fVDB uses a single, novel VDB IndexGrid data structure to accelerate all operations, making it composable and easily extensible. We demonstrated the effectiveness of fVDB via extensive quantitative benchmarks and qualitative demonstrations on real-world 3D learning use cases, showing that fVDB enables high-performance deep learning on large scale 3D data.

In the future, we plan to extend fVDB with more differentiable operators such as hierarchical dual marching cubes, and particle/blob to grid conversion functions (for differentiable physics and particle rendering *e.g.* Gaussian Splatting [Kerbl et al. 2023]). We further plan to develop a high level utility library of neural network architectures for common tasks that can be used off-the-shelf for downstream applications. Beyond new features, an exciting avenue of future work which can lead to even greater sparse convolution performance is to dispatch the optimal kernel on a per-leaf basis depending on local sparsity pattern. Finally, we plan to release the code for fVDB as open-source software expeditiously following publication.

REFERENCES

2023. 3D Karton City model. https://www.turbosquid.com/3d-models/3d-karton-city-2-model-1196110. Accessed: 2023-08-01.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin

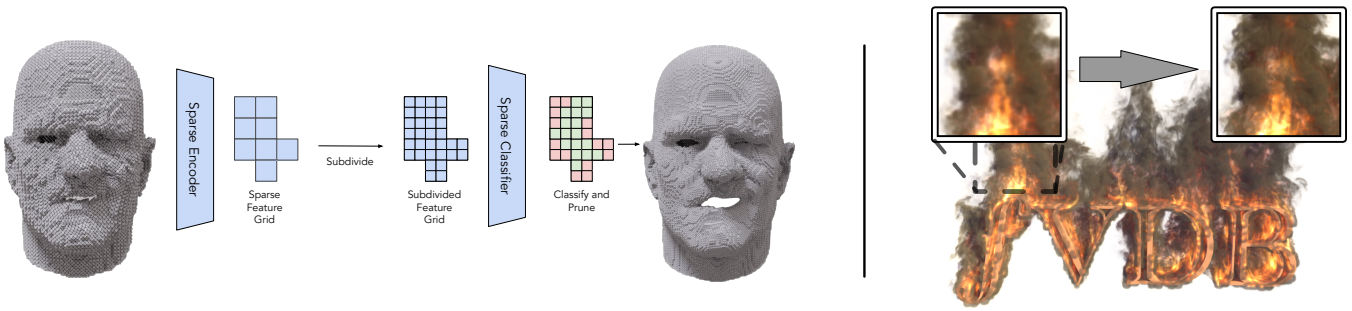

Fig. 15. *f*VDB can enable super-resolution applications on unbounded, sparse physical simulation data where previous methods using traditional dense data structures and operations were prohibitive on large-scale, sparse simulation data. Here we showcase some of the methods we are applying to simulations of fire and facial muscle + skin simulations to super-resolve details using fully convolutional network architectures utilizing *f*VDB operators.

Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

Academy Software Foundation (ASWF). 2012 – 2024. OpenVDB. https://www.openvdb.org

Jonathan T Barron, Ben Mildenhall, Dor Verbin, Pratul P Srinivasan, and Peter Hedman. 2022. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 5470–5479.

J. Behley, M. Garbade, A. Milioto, J. Quenzel, S. Behnke, J. Gall, and C. Stachniss. 2021. Towards 3D LiDAR-based semantic scene understanding of 3D point cloud sequences: The SemanticKITTI Dataset. *The International Journal on Robotics Research* 40, 8-9 (2021), 959–967. https://doi.org/10.1177/02783649211006735

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax

Yuhua Chen, Yibin Xie, Zhengwei Zhou, Feng Shi, Anthony G. Christodoulou, and Debiao Li. 2018. Brain MRI super resolution using 3D deep densely connected neural networks. In *2018 IEEE 15th International Symposium on Biomedical Imaging (ISBI 2018)*. IEEE, Washington, DC, 739–742. https://doi.org/10.1109/ISBI.2018.8363679

Francois Chollet et al. 2015. *Keras*. https://github.com/fchollet/keras

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

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

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135 [cs.LG]

Matt Deitke, Dustin Schwenk, Jordi Salvador, Luca Weihs, Oscar Michel, Eli VanderBilt, Ludwig Schmidt, Kiana Ehsani, Aniruddha Kembhavi, and Ali Farhadi. 2023. Objaverse: A Universe of Annotated 3D Objects. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 13142–13153.

A. Geiger, P. Lenz, and R. Urtasun. 2012. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*. 3354–3361.

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. 2020. DiffTaichi: Differentiable Programming for Physical Simulation. *ICLR* (2020).

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 (TOG)* 38, 6 (2019), 201.

Jiahui Huang, Hao-Xiang Chen, and Shi-Min Hu. 2022. A Neural Galerkin Solver for Accurate Surface Reconstruction. *ACM Trans. Graph.* 41, 6, Article 229 (nov 2022), 16 pages. https://doi.org/10.1145/3550454.3555457

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*. 4369–4379.

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

Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 2023. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. *ACM Transactions on Graphics* 42, 4 (July 2023). https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/

Doyub Kim, Minjae Lee, and Ken Museth. 2022. NeuralVDB: High-resolution Sparse Volume Representation using Hierarchical Neural Networks. arXiv:2208.04448 [cs.LG]

Ruilong Li, Hang Gao, Matthew Tancik, and Angjoo Kanazawa. 2023. Nerfacc: Efficient sampling accelerates nerfs. *arXiv preprint arXiv:2305.04966* (2023).

Minghua Liu, Ruoxi Shi, Linghao Chen, Zhuoyang Zhang, Chao Xu, Xinyue Wei, Hansheng Chen, Chong Zeng, Jiayuan Gu, and Hao Su. 2023a. One-2-3-45++: Fast Single Image to 3D Objects with Consistent Multi-View Generation and 3D Diffusion. arXiv:2311.07885 [cs.CV]

Zhijian Liu, Haotian Tang, Alexander Amini, Xinyu Yang, Huizi Mao, Daniela L. Rus, and Song Han. 2023b. BEVFusion: Multi-Task Multi-Sensor Fusion with Unified Bird's-Eye View Representation. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*. 2774–2781. https://doi.org/10.1109/ICRA48891.2023.10160968

Vanita Mane, Suchit Jadhav, and Praneya Lal. 2020. Image Super-Resolution for MRI Images using 3D Faster Super-Resolution Convolutional Neural Network architecture. *ITM Web of Conferences* 32 (2020), 03044. https://doi.org/10.1051/itmconf/20203203044

Duane Merrill. 2015. Cub. *NVIDIA Research* (2015).

Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. 2022. Instant neural graphics primitives with a multiresolution hash encoding. *ACM transactions on graphics (TOG)* 41, 4 (2022), 1–15.

Ken Museth. 2013. VDB: High-Resolution Sparse Volumes with Dynamic Topology. *ACM Trans. Graph.* 32, 3, Article 27 (jul 2013), 22 pages. https://doi.org/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. https://doi.org/10.1145/2614106.2614136

Ken Museth. 2021. NanoVDB: A GPU-Friendly and Portable VDB Data Structure For Real-Time Rendering And Simulation. In *ACM SIGGRAPH 2021 Talks* (Virtual Event, USA) *(SIGGRAPH '21)*. Association for Computing Machinery, New York, NY, USA, Article 1, 2 pages. https://doi.org/10.1145/3450623.3464653

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG]

Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. 2017. PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space. arXiv:1706.02413 [cs.CV]

Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. 2020. Accelerating 3D Deep Learning with PyTorch3D. *arXiv:2007.08501* (2020).

Xuanchi Ren, Jiahui Huang, Xiaohui Zeng, Ken Museth, Sanja Fidler, and Francis Williams. 2023. XCube: Large-Scale 3D Generative Modeling using Sparse Voxel Hierarchies. *arXiv preprint* (2023).

Shaoshuai Shi, Chaoxu Guo, Li Jiang, Zhe Wang, Jianping Shi, Xiaogang Wang, and Hongsheng Li. 2020. PV-RCNN: Point-Voxel Feature Set Abstraction for 3D Object Detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, Vijay Vasudevan, Wei Han, Jiquan Ngiam, Hang Zhao, Aleksei Timofeev, Scott Ettinger, Maxim Krivokon, Amy Gao, Aditya Joshi, Yu Zhang, Jonathon Shlens, Zhifeng Chen, and Dragomir Anguelov. 2020. Scalability in Perception for Autonomous Driving: Waymo Open

Dataset. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2443–2451.

Matthew Tancik, Ethan Weber, Evonne Ng, Ruilong Li, Brent Yi, Justin Kerr, Terrance Wang, Alexander Kristoffersen, Jake Austin, Kamyar Salahi, Abhik Ahuja, David McAllister, and Angjoo Kanazawa. 2023. Nerfstudio: A Modular Framework for Neural Radiance Field Development. In *ACM SIGGRAPH 2023 Conference Proceedings (SIGGRAPH '23)*.

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

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 *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. 2023. *CUTLASS*. https://github.com/NVIDIA/cutlass

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* 36, 4 (July 2017), 1–11. https://doi.org/10.1145/3072959.3073608

Hengshuang Zhao, Li Jiang, Jiaya Jia, Philip H.S. Torr, and Vladlen Koltun. 2021. Point Transformer. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. New York, NY, USA, 16259–16268.