H-CNN: Spatial Hashing Based CNN for 3D Shape Analysis

Tianjia Shao, *Member, IEEE,* Yin Yang, *Member, IEEE,* Yanlin Weng, Qiming Hou,
and Kun Zhou, *Fellow, IEEE*

Abstract—We present a novel spatial hashing based data structure to facilitate 3D shape analysis using convolutional neural networks (CNNs). Our method builds hierarchical hash tables for an input model under different resolutions that leverage the sparse occupancy of 3D shape boundary. Based on this data structure, we design two efficient GPU algorithms namely **hash2col** and **col2hash** so that the CNN operations like convolution and pooling can be efficiently parallelized. The perfect spatial hashing is employed as our spatial hashing scheme, which is nearly minimal so that our data structure is almost of the same size as the raw input. Compared with existing 3D CNN methods, our data structure significantly reduces the memory footprint during the CNN training. As the input geometry features are more compactly packed, CNN operations also run faster with our data structure. The experiment shows that, under the same network structure, our method yields comparable or better benchmark results compared with the state-of-the-art while it has only one-third memory consumption when under high resolutions (i.e. 256^3).

Index Terms—perfect hashing, convolutional neural network, shape classification, shape retrieval, shape segmentation.

---✦---

1 INTRODUCTION

3D shape analysis such as classification, segmentation, and retrieval has long stood as one of the most fundamental tasks for computer graphics. While many algorithms have been proposed (e.g. see [1]), they are often crafted for a sub-category of shapes by manually extracting case-specific features. A general-purpose shape analysis that handles a wide variety of 3D geometries is still considered challenging. On the other hand, convolutional neural networks (CNNs) are capable of learning essential features out of the raw training data. They have demonstrated great success in various computer vision problems of 2D images/videos [2]–[4]. The impressive results from these works drive many follow-up investigations of leveraging CNNs to tackle more challenging tasks in 3D shape analysis.

Projecting a 3D model into multiple 2D views is a straightforward idea which maximizes the re-usability of existing 2D CNN frameworks [5]–[8]. If the input 3D model has complex geometry however, degenerating it to multiple 2D projections could miss original shape features and lower quality of the final result. It is known that most useful geometry information only resides at the surface of a 3D model. While embedded in $\mathbb{R}^3$, the boundary of a 3D model is essentially two-dimensional. Inspired by this fact, some prior works try to directly extract features out of the model's surface [9], [10] using, for instance the Laplace-Beltrami operator [11]. These methods assume that the model's surface be second order differentiable, which may not be the case in practice. In fact, many scanned or man-made 3D models are of multiple components, which are not even manifold with the presence of a large number of holes, dangling vertices and

---

- *T. Shao, Y. Weng, Q. Hou, and K. Zhou are with the State Key Lab of CAD&CG, Zhejiang University, Hangzhou 310058, China.*
  E-mail: tianjiashao@gmail.com, kunzhou@acm.org
- *Y. Yang is with Department of Electrical and Computer Engineering, University of New Mexico, NM, 87131.*
  E-mail: yangy@unm.edu

intersecting/interpenetrating polygons. Using dense voxel-based discretization is another alternative [12], [13]. Unfortunately, treating a 3D model as a voxelized volume does not scale up as both memory usage and computational costs increase cubically with the escalated voxel resolution. The input data would easily exceed the GPU memory limit under moderate resolutions.

Octree-based model discretization significantly relieves the memory burden for 3D shape analysis [14], [15]. For instance, Wang et al. [15] proposed a framework named O-CNN (abbreviated as OCNN in this paper), which utilizes the octree to discretize the surface of a 3D shape. In octree-based methods, whether or not an octant is generated depends on whether or not its parent octant intersects with the input model. As a result, although octree effectively reduces the memory footprint compared to the "brute-force" voxelization scheme, its memory overhead is still considerable since many redundant empty leaf octants are also generated, especially for high-resolution models.

In this paper, we provide a better answer to the question of how to wisely exploit the sparse occupancy of 3D models and structure them in a way that conveniently interfaces with various CNN architectures, as shown in Figure 1. In our framework, 3D shapes are packed using the perfect spatial hashing (PSH) [16] and we name our framework as Hash-CNN or HCNN. PSH is *nearly minimal* meaning the size of the hash table is almost the same as the size of the input 3D model. As later discussed in § 6.3, our memory overhead is tightly bounded by $\mathbf{O}(N^{\frac{4}{3}})$ in the worst case while OCNN has a memory overhead of $\mathbf{O}(N^2)$, not to mention other $\mathbf{O}(N^3)$ voxel-based 3D CNNs (here, N denotes the voxel resolution at the finest level). Our primary contribution is investigating how to efficiently parallelize CNN operations with hash-based models. To this end, two GPU algorithms namely **hash2col** and **col2hash** are designed to facilitate CNN operations like convolution and pooling. Our experiments show that HCNN achieves comparable benchmark scores in various shape analysis tasks compared with existing 3D CNN methods. However, HCNN consumes much less memory and it also runs

faster due to its compact data packing.

2 RELATED WORK

3D shape analysis [1], [17], [18] is one of the most fundamental tasks in computer graphics. Most existing works utilize manually crafted features for dedicated tasks such as shape retrieval and segmentation. Encouraged by great successes in 2D images analysis using CNN-based machine learning methods [19]–[21], many research efforts have been devoted to leverage CNN techniques for 3D shape analysis.

A straightforward idea is to feed multiple projections of a 3D model as the CNN input [5]–[8] so that the existing CNN architectures for 2D images can be re-used. SnapNet [22] further performs semantic labeling on 3D point clouds by back-projecting the 2D label predictions. However, self-occlusion is almost inevitable for complicated shapes during the projection, and the problem of how to faithfully restore complete 3D information out of 2D projections remains an unknown one to us.

Another direction is to perform CNN operations over the geometric features defined on 3D model surfaces [23]. For instance, Boscaini et al. [10] used windowed Fourier transform and Masci et al. [9] used local geodesic polar coordinates to extract local shape descriptors for the CNN training. These methods, however require that input models should be smooth and manifold, and therefore cannot be directly used for 3D models composed of point clouds or polygon soups. Alternatively, Sinha et al. [24] parameterized a 3D shape over a spherical domain and re-represented the input model using a geometry image [25], based on which the CNN training was carried out. Guo et al. [26] computed a collection of shape features and re-shaped them into a matrix as the network input, while Maron et al. [27] applied CNN to sphere-type shapes using a global parametrization to a planar flat-torus.

Non-CNN-based deep neural networks can also be applied to 3D point clouds. Qi et al. [28] proposed PointNet to analyze 3D point clouds. This method uses shared multi-layer perceptron and max pooling for the feature extraction. A follow-up research called PointNet++ [29] improves the original PointNet by incorporating neighborhood information in the point cloud. Kd-Net [30] partitions the raw point cloud into a Kd-tree structure and constructs a network based on the Kd-tree hierarchy. These methods are all memory efficient since they take the raw point cloud as the network input, and they can obtain good benchmark scores. However, because input point clouds are not in a regular format, it is still unclear how to apply convolution operations in these methods.

Similar to considering images as an array of 2D pixels, discretizing 3D models into voxels is a natural way to organize the shape information for CNN-based shape analysis. Wu et al. [12] proposed 3D ShapeNets for 3D object detection. They represented a 3D shape as a probability distribution of binary variables on voxels. Maturana and Scherer [13] used similar strategy to encode large point cloud datasets. They used a binary occupancy grid to distinguish free and occupied spaces, a density grid to estimate the probability that the voxel would block a sensor beam, and a hit grid to record the hit numbers. Such volumetric discretization consumes memory cubically w.r.t. the voxel resolution, thus is not feasible for high-resolution shape analysis. Observing the fact that the spatial occupancy of 3D data is often sparse, Wang et al. [31] designed a feature-centric voting algorithm named Vote3D for fast recognition of cars, pedestrians and bicyclists from the KITTI database [32] using the sliding window method. More importantly, they demonstrated mathematical equivalence between the sparse convolution and voting. Based on this, Engelcke et al. [33] proposed a method called Vote3Deep converting the convolution into voting procedures, which can be simply applied to the non-empty voxels. However, with more convolution layers added to the CNN, this method quickly becomes prohibitive.

Octree-based data structures have been proven an effective way to reduce the memory consumption of 3D shapes. For example, Riegler et al. [14] proposed a hybrid grid-octree data structure to support high-resolution 3D CNNs. Our work is most relevant to OCNN [15], which used an octree to store the surface features of a 3D model and reduced the memory consumption for 3D CNNs to $\mathbf{O}(N^2)$. For the octree data structure, an octant is subdivided into eight children octants if it intersects with the model's surface regardless if all of those eight children octants are on the model. Therefore, an OCNN's subdivision also yields $\mathbf{O}(N^2)$ futile octants that do not contain useful features of the model. On the other hand, we use multi-level PSH [16] to organize voxelized 3D models. PSH is nearly minimal while retaining an as cache-friendly as possible random access. As a result, the memory footprint of HCNN is close to the theoretic lower bound. Unlike in the original PSH work [16], the main hash table only stores the data index, and the real feature data is compactly assembled in a separate data array. We investigate how to seamlessly synergize hierarchical PSH-based models with CNN operations so that they can be efficiently executed on the GPU.

3 ALGORITHM OVERVIEW

Coupling CNN architectures with spatial hashing is challenging. Many CNN operations like convolution and pooling alter the spatial arrangement of the 3D model's geometry features, which require our hash-based data structure being able to freely exchange information between different resolutions. Unlike the child-parent linkage in tree-based data structures however, a spatial hashing does not possess such built-in hierarchical connections. Furthermore, we also need to bear in mind while designing our data structure that CNN training is typically performed over a group of 3D models, i.e. a mini-batch, and we need to efficiently fetch the correspondences of features under different resolutions for all the models in the group. Finally, we would like to make CNN operations GPU-friendly so they can be parallelized and accelerated with modern GPU architectures.

To achieve these objectives, on the top of the regular PSH data structure (§ 4.1), we design an auxiliary data structure named model index table, which instantly returns the model's index within a mini-batch for a given hash query. The corresponding geometry data can then be efficiently accessed with accumulated index tables (§ 4.2). As the we only use simple modulo functions for the spatial hashing [16], we can efficiently locate the corresponding data at a different resolution using constant indirections. All the known CNN operations such as convolution can then be parallelized with our data structure by assigning a CUDA thread to collect necessary feature values within the receptive field for each output hash slot per channel as illustrated in Figure 2 (§ 5).

4 SPATIAL HASHING FOR 3D CNN

For a given input 3D shape, either a triangle soup/mesh or a point cloud, we first uniformly scale it to fit a unit sphere pivoted at the
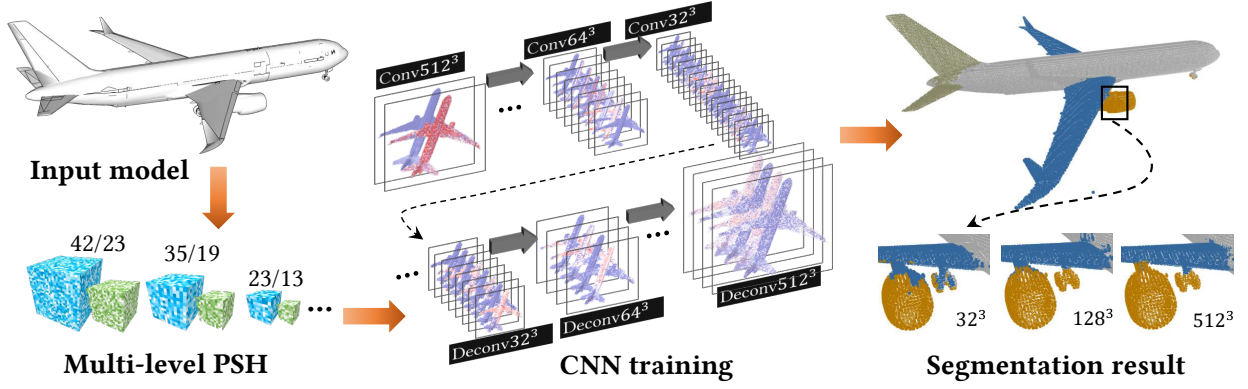
Fig. 1. An overview of HCNN framework for shape analysis. We construct a set of hierarchical PSHs to pack surface geometric features of an input airplane model at different resolution levels. Compared with existing 3D CNN frameworks, our method fully utilizes the spatial sparsity of 3D models, and the PSH data structure is almost of the same size as the raw input. Therefore, we can perform high-resolution shape analysis with 3D CNN efficiently. The final segmentation results demonstrate a clear advantage of high-resolution models. Each part of the airplane model is much better segmented at the resolution of 512^3, which is currently only possible with HCNN.
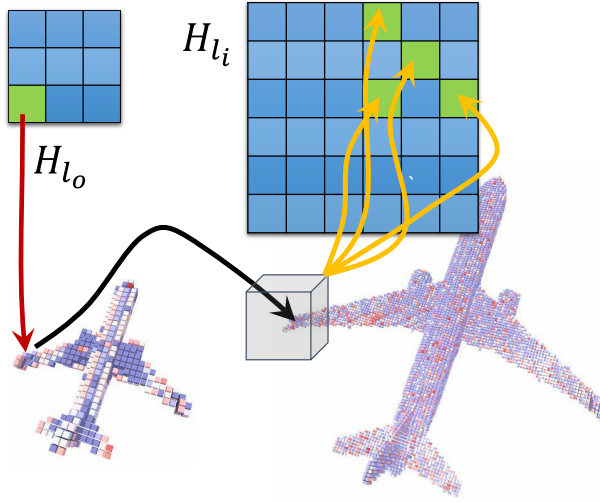

Fig. 2. Accessing the receptive field between different resolutions based on PSH. Here, l_i and l_o denote the input and output resolution levels. For each output hash slot in H_{l_o}, we obtain its voxel position in the volume from the position tag (the red arrow). The input receptive field at resolution level l_i for the voxel can be located based on the kernel size, stride and padding (the black arrow). For each voxel within the receptive field, we access its hash slot using the PSH hashing function (the yellow arrows) so that its corresponding feature values in the data array can be retrieved.

4.1 Multi-level PSH

The core data structure of PSH includes a hash table and an offset table. The size of hash table is almost the same as the number of non-empty voxels, and the offset table is typically mush smaller than the hash table. By applying two simple modulo functions separately on the two tables, we can obtain a perfect hashing mapping for all non-empty voxels. Such simple implementation helps greatly reduce the memory usage. Based on the core PSH data structure, we build a set of hierarchical PSHs. At each level of the hierarchy, we construct a data array D, a hash table H, an offset table Φ and a position tag T. The data array at the finest level stores the input feature (i.e. the normal direction of the voxel).
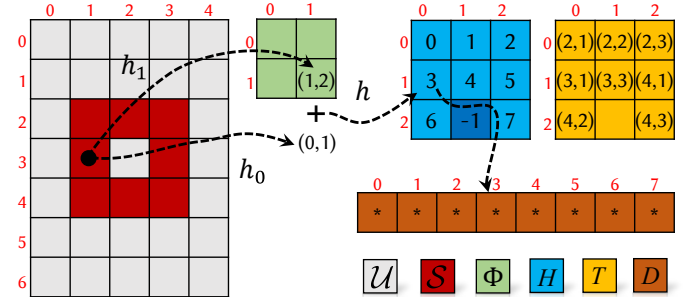

Fig. 3. An illustrative 2D example of the constitution of our PSH. The domain $\mathcal{U}$ consists of 7×5 2D voxels or pixels. The red-shaded pixels stand for the input model. The green, blue, yellow and brown tables are the offset table (Φ), hash table (H), position tag (T) and data array (D) respectively.

model's geometry center. Then, an axis-aligned bounding cube is built, whose dimension equals to the sphere's diameter. Doing so ensures that the model remains inside the bounding box under arbitrary rotations, so that we can further apply the training data augmentation during the training (see e.g. §6.4). This bounding cube is subdivided into grid cells or *voxels* along x, y, and z axes. A voxel is a small equilateral cuboid. It is considered non-empty when it encapsulates a small patch of the model's boundary surface. As suggested in [15], we put extra sample points on this embedded surface patch, and the averaged normal of all the sample points is fed to the CNN as the input signal. For an empty voxel, its input is simply a zero vector.

Let $\mathcal{U}$ be a d-dimensional discrete spatial domain with $u = \bar{u}^d$ voxels, out of which the sparse geometry data $\mathcal{S}$ occupies n grid cells (i.e. $n = |\mathcal{S}|$). In other words, $\mathcal{U}$ represents all the voxels within the bounding cube at the given resolution, and $\mathcal{S}$ represents the set of voxels intersecting with the input model. We seek for a hash table H, which is a d-dimensional array of size $m = \bar{m}^d \geq n$ and a d-dimensional offset table Φ of size $r = \bar{r}^d$. By building maps $h_0 : p \to p \bmod \bar{m}$ from $\mathcal{U}$ to the hash table H and $h_1 : p \to p \bmod \bar{r}$ from $\mathcal{U}$ on the offset table Φ, one can obtain the perfect hash function mapping each non-empty voxel on the 3D shape

$p \in \mathcal{S}$ to a unique slot $s = h(p)$ in the hash table as:

$$h(p) = h_0(p) + \Phi[h_1(p)] \bmod \bar{m}. \tag{1}$$

The hash table H possesses slightly excessive slots (i.e. $m = \bar{m}^d \geq n$) to make sure that the hashing representation of $\mathcal{S}$ is collision free. A `NULL` value is stored at those redundant slots in H. Clearly, these `NULL` values should not participate in the CNN operations like batch normalization and scale. To this end, we assemble all the data for $\mathcal{S}$ into a compact d-dimensional array D of size n. H only houses the data index in D. If a slot in H is redundant, it is indexed as -1 so that the associated data query is skipped.

Empty voxels (i.e. when $p \in \mathcal{U} \setminus \mathcal{S}$) may also be visited during CNN operations like convolution and pooling. Plugging these voxels' indices into Eq. (1) is likely to return incorrect values that actually correspond to other non-empty grid cells. To avoid this mismatch, we adopt the strategy used in [16] adding an additional position tag table T, which has the same size of H. $T[i]$ stores the voxel index for the corresponding slot at $H[i]$. Therefore when a grid cell p is queried, we first check its data index in H or $H[h(p)]$. If it returns a valid index other than -1, we further check the position tag $T[h(p)]$ to make sure $T[h(p)] = p$. Otherwise, $p \in \mathcal{U} \setminus \mathcal{S}$ is an off-model voxel and the associated CNN operation should be skipped. In our implementation, we use a 16-bit position tag for each x, y and z index, which supports the voxelization resolution up to $65,536^3$.

Figure 3 gives an illustrative 2D toy example. The domain $\mathcal{U}$ is a 7×5 2D pixel grid. The red-shaded pixels stand for the input model, thus $n = |\mathcal{S}| = 8$. We have a 3×3 hash table H (i.e. $\bar{m} = 3$ and it is the blue table in the figure) and a 2×2 offset table (i.e. $\bar{r} = 2$ and it is the green table in the figure). Assume that the pixel $p(3,1)$ is queried and h_0 yields $h_0(3,1) = (3 \bmod \bar{m}, 1 \bmod \bar{m}) = (0,1)$. $h_1(3,1) = (3 \bmod \bar{r}, 1 \bmod \bar{r}) = (1,1)$ gives the 2D index in the offset table. $\Phi(1,1) = (1,2)$, which is added to $h_0(p)$ to compute the final index in H: $\Phi(1,1) + h_0(p) = (1 + 0 \bmod \bar{m}, 2 + 1 \bmod \bar{m}) = (1,0)$. Before we access the corresponding data cell in D (the fourth cell in this example because $H(1,0) = 3$), the position tag table (the yellow table) is queried. Since $T(1,0) = (3,1)$, which equals to the original pixel index of p, we know that $p \in \mathcal{S}$ is indeed on the input model. Note that in this example, $H(2,1)$ is a redundant slot (colored in dark blue in Figure 3). Therefore, the corresponding index is -1.

4.2 Mini-batch with PSH

During the CNN training, it is typical that the network parameters are optimized over a subset of the training data, referred to as a mini-batch. Let b be the batch size and l be the resolution level. In order to facilitate per-batch CNN training, we build a "super-PSH" by attaching H, Φ, T for all the models in a batch: $H_l^* = \{H_l^1, H_l^2, ..., H_l^b\}$, $\Phi_l^* = \{\Phi_l^1, \Phi_l^2, ..., \Phi_l^b\}$, and $T_l^* = \{T_l^1, T_l^2, ..., T_l^b\}$ as illustrated in Figure 4. That is we expand each of these d-dimensional tables into a 1D array and concatenate them together. The data array D_l^* of the batch is shaped as a row-major c_l by $\sum_{i=1}^b |\mathcal{S}_l^i|$ matrix, where c_l is the number of channels at level l, and $\sum_{i=1}^b |\mathcal{S}_l^i|$ is the total number of non-empty voxels of all the models in the batch. A column of D_l^* is a c_l-vector, and it stores the features of the corresponding voxel. The dimensionality of H_l^i, Φ_l^i, and D_l^i is also packed as $\bar{m}_l^* = \{\bar{m}_l^1, \bar{m}_l^2, ..., \bar{m}_l^b\}$, $\bar{r}_l^* = \{\bar{r}_l^1, \bar{r}_l^2, ..., \bar{r}_l^b\}$, and $n_l^* = \{n_l^1, n_l^2, ..., n_l^b\}$.

In addition, we also record accumulated indices for H, Φ and D as: $M_l^* = \{0, M_l^1, M_l^2, ..., M_l^b\}$, $R_l^* = \{0, R_l^1, R_l^2, ..., R_l^b\}$ and $N_l^* = \{0, N_l^1, N_l^2, ..., N_l^b\}$ where

$$M_l^i = \sum_{k=1}^i (\bar{m}_l^k)^d = \sum_{k=1}^i m_l^k, \quad R_l^i = \sum_{k=1}^i (\bar{r}_l^k)^d = \sum_{k=1}^i r_l^k, \quad N_l^i = \sum_{k=1}^i n_l^k.$$

Indeed, M_l^*, R_l^* and N_l^* store the super table (i.e. H_l^*, Φ_l^*, T_l^*, and D_l^*) offsets of the k-th model in the batch. For instance, the segment of H_l^* starting from $H_l^*[M_l^*[k-1]]$ to $H_l^*[M_l^*[k] - 1]$ corresponds to the hash table H_l^k; the segment from $\Phi_l^*[R_l^*[k-1]]$ to $\Phi_l^*[R_l^*[k] - 1]$ corresponds to the offset table Φ_l^k; the segment from $T_l^*[M_l^*[k-1]]$ to $T_l^*[M_l^*[k] - 1]$ corresponds to the position tag T_l^k; and the segment from $D_l^*[N_l^*[k-1]]$ to $D_l^*[N_l^*[k] - 1]$ is the data array D_l^k. Lastly, we build a model index table $V_l^* = \{V_l^1, V_l^2, ..., V_l^b\}$ for the inverse query. Here, V_l^i has the same size as H_l^i does, and each of its slots stores the model's index in a batch: $V_l^i(\cdot) = i$.

5 CNN OPERATIONS WITH MULTI-LEVEL PSH

In this section we show how to apply CNN operations like convolution/transposed convolution, pooling/unpooling, batch normalization and scale to the PSH-based data structure so that they can be efficiently executed on the GPU.

Convolution The convolution operator Ψ_c in the unrolled form is:

$$\Psi_c(p) = \sum_n \sum_i \sum_j \sum_k W_{ijk}^{(n)} \cdot F^{(n)}(p_{ijk}), \tag{2}$$

where p_{ijk} is a neighboring voxel of voxel $p \in \mathcal{S}$. $F^{(n)}$ and $W_{ijk}^{(n)}$ are the feature vector and the kernel weight of the n-th channel. This nested summation can be reshaped as a matrix product [34] and computed efficiently on the GPU:

$$\mathbf{D}_o = \mathbf{W} \cdot \widetilde{\mathbf{D}}_i. \tag{3}$$

Let l_i and l_o denote the input and output hierarchy levels of the convolution. $\mathbf{D}_o$ is essentially the matrix representation of the output data array $D_{l_o}^*$. Each column of $\mathbf{D}_o$ is the feature signal of an output voxel. A row vector in $\mathbf{W}$ concatenates vectorized kernel weights for all the input channels, and the number of rows in $\mathbf{W}$ equals to the number of convolution kernels employed. We design a subroutine `hash2col` to assist the assembly of matrix $\widetilde{\mathbf{D}}_i$, which fetches feature values out of the input data array $D_{l_i}^*$ so that a column of $\widetilde{\mathbf{D}}_i$ stacks feature signals within the receptive fields covered by kernels.

The algorithmic procedure for `hash2col` is detailed in Algorithm 1. In practice, we launch $c_{l_i} \cdot M_{l_o}^*[b]$ `CUDA` threads in total, where c_{l_i} is the number of input channels. Recall that $M_{l_o}^*[b]$ is the last entry of the accumulated index array $M_{l_o}^*$ such that $M_{l_o}^*[b] = M_{l_o}^b$, and it gives the total number of hash slots on $H_{l_o}^*$. Hence, our parallelization scheme can be understood as assigning a thread to collect necessary feature values within the receptive field for each output hash slot per channel. The basic idea is to find the receptive field $\mathcal{R}_{l_i} \subset \mathcal{U}_{l_i}$ that corresponds to an output voxel p_{l_o} and retrieve features for $\widetilde{\mathbf{D}}_i$. A practical challenge lies in the fact that output and input data arrays may reside on the voxel grids of different hierarchy levels. Therefore, we need to resort to the PSH mapping (Eq. (1)) and the position tag table to build the necessary output-input correspondence.

Given a thread index i_{thrd} ($0 \leq i_{thrd} \leq c_{l_i} \cdot M_{l_o}^*[b] - 1$), we compute its associated channel index i_c as $i_c = \lfloor i_{thrd} / M_{l_o}^*[b] \rfloor$.
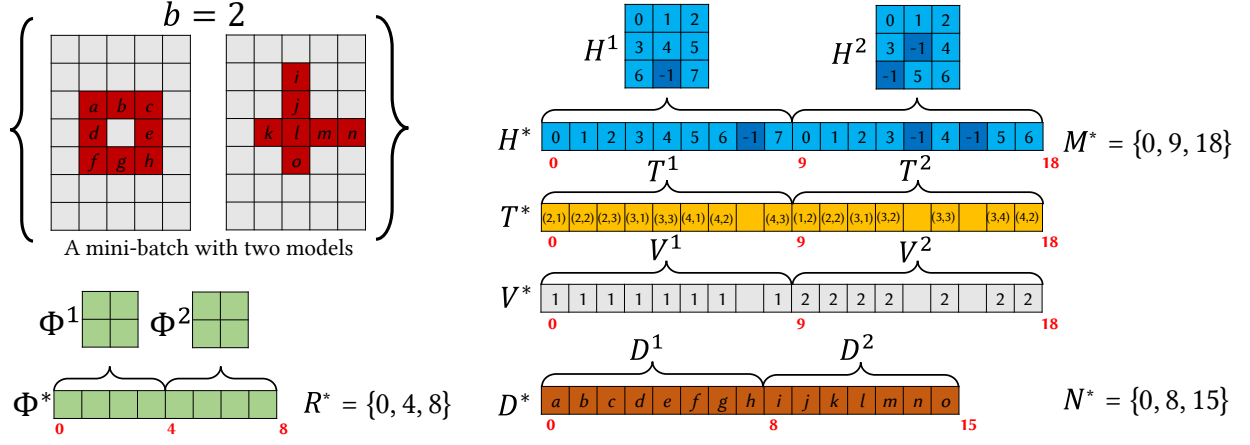
Fig. 4. PSH data structures for a mini-batch of two models. All the feature data for the red-shaded pixels are stored in the super data array D^*, which consists of the data arrays of each individual models. Super hash table H^*, position tag T^* and model index table V^* are of the same size. For a give hash slot indexed at i_{H^*}, one can instantly know that this voxel, if not empty, is on the $V^*[i_{H^*}]$-th model in the batch by checking the model index table V^*. This information bridges the data sets from different hierarchy levels. With the auxiliary accumulated index tables R^*, M^*, and N^*, we can directly pinpoint the data using local index offset by $R^*[V^*[i_{H^*}]-1]$, $M^*[V^*[i_{H^*}]-1]$ and $N^*[V^*[i_{H^*}]-1]$ respectively. For instance in this simple example, when the local hash index is computed using Eq. (1) for a non-empty voxel on the second model, its hash index in H^* can then be obtained by offsetting the local hash index by $M^*[2-1]=9$.

Its super hash index $i_{H^*_{l_o}}$ (i.e. the index in $H^*_{l_o}$) is simply $i_{H^*_{l_o}} = i_{thrd} - i_c \cdot M^*_{l_o}[b]$, so that we know that this thread is for the $V^*_{l_o}[i_{H^*_{l_o}}]$-th model in the batch (recall that $V^*_{l_o}$ is the model index table). If $H^*_{l_o}[i_{H^*_{l_o}}] \neq -1$ meaning this thread corresponds to a valid non-empty voxel, the index of the column in $\mathbf{D}_o$ that houses the corresponding output feature is $N^*_{l_o}[V^*_{l_o}[i_{H^*_{l_o}}]-1] + H^*_{l_o}[i_{H^*_{l_o}}]$.

With the help of the position tag table $T^*_{l_o}$, the index of the output voxel in $\mathcal{U}_{l_o}$ associated with the thread i_{thrd} can be retrieved by $p_{l_o} = T^*_{l_o}[i_{H^*_{l_o}}]$, based on which we can obtain the input voxel positions within the receptive field and construct the corresponding column in $\widetilde{\mathbf{D}}_i$. Specifically, if the stride size is one, indicating the voxel resolution is unchanged after the convolution or $l_i = l_o$, the input model has the same hash structure as the output. In this case, the receptive field associated with p_{l_o} spans from $p_{l_o} - (F-1)/2$ to $p_{l_o} + (F-1)/2$ along each dimension on $\mathcal{U}_{l_i}$ denoted as $\mathcal{U}_{l_i}[p_{l_o} - (F-1)/2, p_{l_o} + (F-1)/2]^d$. Here, F is the kernel size. On the other hand, if the stride size is larger than one, the convolution will down-sample the input feature, and the receptive field on $\mathcal{U}_{l_i}$ is $\mathcal{U}_{l_i}[p_{l_o} \cdot S_s - S_p, p_{l_o} \cdot S_s - S_p + F - 1]^d$ with the stride size S_s and the padding size S_p. For irregular kernels [35], [36], we can similarly obtain the corresponding receptive field on $\mathcal{U}_{l_i}$ based on p_{l_o}.

As mentioned, for a non-empty voxel $p_{l_i} \in \mathcal{U}_{l_i}$ within the receptive field of a given output voxel $p_{l_o} \in \mathcal{U}_{l_o}$, we know that it belongs to the v-th model of the batch, where $v = V^*_{l_o}[i_{H^*_{l_o}}]$. Therefore, its offset index in $\Phi^*_{l_i}$ can be computed as:

$$i_{\Phi^*_{l_i}} = R^*_{l_i}[v-1] + h_1(p_{l_i}), \tag{4}$$

where $R^*_{l_i}$ is the accumulated offset index array at level l_i, and $R^*_{l_i}[v-1]$ returns the starting index of the offset table $\Phi^v_{l_i}$ in the super table $\Phi^*_{l_i}$. $h_1(p_{l_i})$ computes the (local) offset index. Thus, the offset value of p_{l_i} can be queried by $\Phi^*_{l_i}[i_{\Phi^*_{l_i}}]$. The index of p_{l_i} in the super hash table $H^*_{l_i}$ can be computed similarly as:

$$\begin{aligned} i_{H^*_{l_i}} &= M^*_{l_i}[v-1] + \big(h_0(p_{l_i}) + \Phi^*_{l_i}[i_{\Phi^*_{l_i}}] \big) \bmod \bar{m}_{l_i} \\ &= M^*_{l_i}[v-1] + \big(h_0(p_{l_i}) + \Phi^*_{l_i}[R^*_{l_i}[v-1] + h_1(p_{l_i})] \big) \bmod \bar{m}_{l_i}. \end{aligned} \tag{5}$$

Here, $h_0(p_{l_i})$ and $h_1(p_{l_i})$ are maps defined on hierarchy level l_i. If $H^*_{l_i}[i_{H^*_{l_i}}] \neq -1$ and the position tag is also consistent (i.e. $T^*_{l_i}[i_{H^*_{l_i}}] = p_{l_i}$), we fetch the feature from the data array by $D^*_{l_i}[i_{D^*_{l_i}}]$, where

$$i_{D^*_{l_i}} = N^*_{l_i}[v-1] + H^*_{l_i}[i_{H^*_{l_i}}]. \tag{6}$$

Otherwise, a zero value is returned.

Back propagation & weight update During the CNN training and optimization, the numerical gradient of kernels' weights is computed as:

$$\delta\mathbf{W} = \delta\mathbf{D}_o \cdot \widetilde{\mathbf{D}}_i^\top, \tag{7}$$

where $\delta\mathbf{D}_o$ is the variation of the output data array $\mathbf{D}_o$. In order to apply Eq. (7) in previous CNN layers, we also calculate how the variational error is propagated back:

$$\delta\widetilde{\mathbf{D}}_i = \mathbf{W}^\top \cdot \delta\mathbf{D}_o. \tag{8}$$

Clearly, we need to re-pack the errors in $\delta\widetilde{\mathbf{D}}_i$ in accordance with the format of the data array $D^*_{l_i}$ so that the resulting matrix $\delta\mathbf{D}_i$ can be sent to the previous CNN layer. This process is handled by the **col2hash** subroutine, outlined in Algorithm 2. As the name implies, **col2hash** is quite similar to **hash2col** except at line 26, where variational errors from the receptive field is lumped into a single accumulated error.

Pooling, unpooling & transposed convolution The pooling layer condenses the spatial size of the input features by using a single representative activation for a receptive field. This operation can be regarded as a special type of convolution with a stride size $S_s > 1$. Therefore, **hash2col** subroutine can also assist the pooling operation. The average-pooling is dealt with as applying a convolution kernel with all the weights equal to $1/F^3$. For the max-pooling, instead of performing a stretched inner product across the receptive field, we output the maximum signal after the traversal of the receptive field (the `for` loop at line 20 in Algorithm 1). Unlike OCNN [15], our framework supports any stride sizes for the pooling since the PSH can be generated on the grid of an arbitrary resolution.

Input: $b, c_{l_i}, D_{l_i}^*, H_{l_i}^*, M_{l_i}^*, R_{l_i}^*, N_{l_i}^* V_{l_o}^*, T_{l_o}^*, H_{l_o}^*, M_{l_o}^*, N_{l_o}^*, F, S_s, S_p$
Output: $\widetilde{\mathbf{D}}_i$

1: launch $c_{l_i} \cdot M_{l_o}^*[b]$ threads;
`/* parallel for loop and `i_{thrd}` is the thread index   */`
2: **for** $i_{thrd} = 0 : c_{l_i} \cdot M_{l_o}^*[b] - 1$ **do**
3:     $i_c \leftarrow \lfloor i_{thrd}/M_{l_o}^*[b] \rfloor$;   `// `i_c` is the channel index`
4:     $i_{H_{l_o}^*} \leftarrow i_{thrd} - i_c \cdot M_{l_o}^*[b];$
5:     $v \leftarrow V_{l_o}^*[i_{H_{l_o}^*}];$ `// `v` is the model index in the`
        `mini-batch`
6:     $col \leftarrow N_{l_o}^*[v-1] + H_{l_o}^*[i_{H_{l_o}^*}];$ `// `col` is the column index`
7:     **if** $H_{l_o}^*[i_{H_{l_o}^*}] = -1$ **then**
8:         return ;   `// `$i_{H_{l_o}^*}$` points to an empty hash slot`
9:     **end**
10:    **else**
11:        $p_{l_o} \leftarrow T_{l_o}^*[i_{H_{l_o}^*}];$ `// `p_{l_o}` is the voxel position`
12:        $\mathcal{R}_{l_i} \leftarrow \emptyset;$ `// `$\mathcal{R}_{l_i}$` is the receptive field on `$\mathcal{U}_{l_i}$
           `/* `S_s` and `S_p` are stride and padding sizes     */`
13:        **if** $S_s = 1$ **then**
14:            $\mathcal{R}_{l_i} \leftarrow \mathcal{U}_{l_i}[p_{l_o} - (F-1)/2, p_{l_o} + (F-1)/2]^3;$
15:        **end**
16:        **else**
17:            $\mathcal{R}_{l_i} \leftarrow \mathcal{U}_{l_i}[p_{l_o} \cdot S_s - S_p, p_{l_o} \cdot S_s - S_p + F - 1]^3;$
18:        **end**
19:        $row \leftarrow 0;$ `// `row` is current row index in `$\widetilde{\mathbf{D}}_i$
           `/* iterate all the voxels within the kernel */`
20:        **for** $p_{l_i} \in \mathcal{R}_{l_i}$ **do**
21:            $i_{\Phi_{l_i}^*} \leftarrow R_{l_i}^*[v-1] + h_1(p_{l_i});$
22:            $i_{H_{l_i}^*} \leftarrow M_{l_i}^*[v-1] + (h_0(p_{l_i}) + \Phi_{l_i}^*[i_{\Phi_{l_i}^*}] \bmod \bar{m}_{l_i});$
23:            **if** $H_{l_i}^*[i_{H_{l_i}^*}] \neq -1$ and $p_{l_i} = T_{l_i}^*[i_{H_{l_i}^*}]$ **then**
24:                $i_{D_{l_i}^*} \leftarrow N_{l_i}^*[v-1] + H_{l_i}^*[i_{H_{l_i}^*}];$
25:                $\widetilde{\mathbf{D}}_i[i_c \cdot F + row, col] \leftarrow D_{l_i}^*[i_c, i_{D_{l_i}^*}];$
26:            **end**
27:            **else**
28:                $\widetilde{\mathbf{D}}_i[i_c \cdot F + row, col] \leftarrow 0;$
29:            **end**
               `/* assume `p_{l_i}` is iterated according to its`
                  `spatial arrangement in `$\mathcal{R}_{l_i}$`         */`
30:            $row \leftarrow row + 1;$
31:        **end**
32:    **end**
33: **end**

Algorithm 1: `hash2col` subroutine

Input: $b, c_{l_i}, \delta\widetilde{\mathbf{D}}_i, H_{l_i}^*, M_{l_i}^*, R_{l_i}^*, N_{l_i}^* V_{l_o}^*, T_{l_o}^*, H_{l_o}^*, M_{l_o}^*, N_{l_o}^*, F, S_s, S_p$
Output: $\delta D_{l_i}^*$

1: $\delta D_{l_i}^*[:] \leftarrow 0;$ `// `$\delta D_{l_i}^*$` is initialized as 0`
2: launch $c_{l_i} \cdot M_{l_o}^*[b]$ threads;
`/* parallel for loop and `i_{thrd}` is the thread index   */`
3: **for** $i_{thrd} = 0 : c_{l_i} \cdot M_{l_o}^*[b] - 1$ **do**
4:     $i_c \leftarrow \lfloor i_{thrd}/M_{l_o}^*[b] \rfloor$;   `// `i_c` is the channel index`
5:     $i_{H_{l_o}^*} \leftarrow i_{thrd} - i_c \cdot M_{l_o}^*[b];$
6:     $v \leftarrow V_{l_o}^*[i_{H_{l_o}^*}];$ `// `v` is the model index in the`
        `mini-batch`
7:     $col \leftarrow N_{l_o}^*[v-1] + H_{l_o}^*[i_{H_{l_o}^*}];$ `// `col` is the column index`
8:     **if** $H_{l_o}^*[i_{H_{l_o}^*}] = -1$ **then**
9:         return ;   `// `$i_{H_{l_o}^*}$` points to an empty hash slot`
10:    **end**
11:    **else**
12:        $p_{l_o} \leftarrow T_{l_o}^*[i_{H_{l_o}^*}];$ `// `p_{l_o}` is the voxel position`
13:        $\mathcal{R}_{l_i} \leftarrow \emptyset;$ `// `$\mathcal{R}_{l_i}$` is the receptive field on `$\mathcal{U}_{l_i}$
           `/* `S_s` and `S_p` are stride and padding sizes     */`
14:        **if** $S_s = 1$ **then**
15:            $\mathcal{R}_{l_i} \leftarrow \mathcal{U}_{l_i}[p_{l_o} - (F-1)/2, p_{l_o} + (F-1)/2]^3;$
16:        **end**
17:        **else**
18:            $\mathcal{R}_{l_i} \leftarrow \mathcal{U}_{l_i}[p_{l_o} \cdot S_s - S_p, p_{l_o} \cdot S_s - S_p + F - 1]^3;$
19:        **end**
20:        $row \leftarrow 0;$ `// `row` is current row index in `$\delta\widetilde{\mathbf{D}}_i$
           `/* iterate all the voxels within the kernel */`
21:        **for** $p_{l_i} \in \mathcal{R}_{l_i}$ **do**
22:            $i_{\Phi_{l_i}^*} \leftarrow R_{l_i}^*[v-1] + h_1(p_{l_i});$
23:            $i_{H_{l_i}^*} \leftarrow M_{l_i}^*[v-1] + (h_0(p_{l_i}) + \Phi_{l_i}^*[i_{\Phi_{l_i}^*}] \bmod \bar{m}_{l_i});$
24:            **if** $H_{l_i}^*[i_{H_{l_i}^*}] \neq -1$ and $p_{l_i} = T_{l_i}^*[i_{H_{l_i}^*}]$ **then**
25:                $i_{D_{l_i}^*} \leftarrow N_{l_i}^*[v-1] + H_{l_i}^*[i_{H_{l_i}^*}];$
26:                $\delta D_{l_i}^*[i_c, i_{D_{l_i}^*}] \leftarrow \delta D_{l_i}^*[i_c, i_{D_{l_i}^*}] + \delta\widetilde{\mathbf{D}}_i[i_c \cdot F + row, col];$
27:            **end**
28:            $row \leftarrow row + 1;$
29:        **end**
30:    **end**
31: **end**

Algorithm 2: `col2hash` subroutine

Other CNN operations Because all the feature values in HCNN are compactly stored in the data array D^*, operations that are directly applied to the feature values like batch normalization [43] and scale can be trivially parallelized on GPU.

6 EXPERIMENTAL RESULTS

Our framework was implemented on a desktop computer equipped with an `Intel I7-6950X` CPU (3.0 GHz) and an `nVidia GeForce 1080 Pascal` GPU with 8 GB DDR5 memory. We used `Caffe` framework [44] for the CNN implementation. The 3D models used are from `ModeNet40` [12] and `ShapeNet Core55` [45]. Both are publicly available. The source code of HCNN can be found in the accompanying supplementary file. The executable and some of the training data in PSH format (4.4 GB in total) can also be downloaded via the anonymous `Google Drive` link, which can also be found in the supplementary file. We encourage readers to test HCNN by themselves.

Model rectification It has been noticed that normal information on 3D models from the `ModeNet` database are often incorrect or missing. We fix the normal information by casting rays from 14 virtual cameras (at six faces and eight corners of the bounding cube). Some 3D models use a degenerated 2D plane to represent a thin shape. For instance, the back of a chair model may only consist of two flat triangles. To restore the volumetric information of such thin geometries, we displace the sample points on the

The unpooling operation aims to partially revert the input activation after the pooling, which could be useful for understanding the CNN features [37], [38] or restoring the spatial structure of the input activations for segmentation [39], flow estimation [40], and generative modeling [39]. During the max-pooling, we record the index of the maximum activation for each receptive field (known as the *switch*). When performing the max-unpooling, the entire receptive field corresponding to an input voxel is initialized to be zero, and the feature signal is restored only at the recorded voxel index. The average-unpooling is similarly handled, where we evenly distribute the input activation over its receptive field.

Transposed convolution is also referred to as deconvolution or fractionally strided convolution [41], which has been proven useful for enhancing the activation map [37], [42]. Mathematically, the transposed convolution is equivalent to the regular convolution and can be dealt with using `hash2col` subroutine. However, doing so involves excessive zero padding and thus degenerates network's performance. In fact, the deconvolution flips the input and output of the forward convolution using a transposed kernel as: $\widetilde{\mathbf{D}}_o = \mathbf{W}^\top \cdot \mathbf{D}_i$, which is exactly how we handle the error back propagation (i.e. Eq. (8)). Therefore, the `col2hash` subroutine can be directly used for deconvolution operations.

model towards its normal direction by $1/(2 \cdot \bar{u}_{max})$, where $\bar{u}_{max}$ denotes the voxel resolution at the finest hierarchy level. In other words, the model's surface is slightly dilated by a half-voxel size.

6.1 Network Architecture

A carefully fine-tuned network architecture could significantly improve the CNN result and relieve the training efforts. Nevertheless, this is neither the primary motivation nor the contribution of this work. In order to report an apple-to-apple comparison with peers and benchmark our method objectively, we employ a network similar to the well-known LeNet [46]. Following the same concern, we do not compare our method with other non-CNN-based methods like PointNet++ [29] or Kd-Net [30]

In our framework, the convolution and pooling operations are repeated from the finest level, and ReLU is used as the activation function. A batch normalization (BN) is also applied to reduce the internal covariance shift [43]. Our PSH hierarchy allows very dense voxelization at the resolution of 512^3 (i.e. see Figure 5), where the hierarchy level $l = 9$. Each coarser level reduces the resolution by half, and the coarsest level has the resolution of 4^3, where $l = 2$. Such multi-level PSH configuration exactly matches the OCNN hierarchy, which allows us to better evaluate the performance between these two data structures. At each level, we have the same operation sequence as: $Convolution \rightarrow BN \rightarrow ReLU \rightarrow Pooling$. The receptive field of kernels is $3 \times 3 \times 3$, and the number of channels at the l-th level is set as $\max\{2, 2^{9-l}\}$.

Three classic shape analysis tasks namely shape classification, retrieval, and segmentation are benchmarked. For the classification, two fully connected (FC) layers, a softmax layer and two dropout layers [47], [48] ordered as: $Dropout \rightarrow FC(128) \rightarrow Dropout \rightarrow FC(N_c) \rightarrow Softmax \rightarrow Output$ are appended. Here, $FC(K)$ indicates K neurons are set at the FC layer. For the shape retrieval, we use the output from the object classification as the key to search for the most similar shapes to the query. For the segmentation, we follow the DeconvNet [39] structure, which adds a deconvolution network after a convolution network for dense predictions. The deconvolution network simply reverses the convolution procedure where the convolution and pooling operators are replaced by the deconvolution and unpooling operators. Specifically, at each level we apply $Unpooling \rightarrow Deconvolution \rightarrow BN \rightarrow ReLU$ and then move to the next finer level.

The reader may notice that our experiment setting transplants the one used in [15] except that all the features are organized using PSH rather than octrees. This is because we consider OCNN [15] as our primary competitor and would like the report an objective side-by-side comparison with it. Lastly, we would like to remind the reader again that HCNN is not restricted to power-of-two resolution changes. To the best of our knowledge, our HCNN is compatible with all the existing CNN architectures and operations.

Training specifications The network is optimized using the stochastic gradient descent method. We set momentum as 0.9 and weight decay as 0.0005. A mini-batch consists of 32 models. The dropout ratio is 0.5. The initial learning rate is 0.1, which is attenuated by a factor of 10 after 10 epochs.

6.2 PSH Construction

As the data pre-processing, we construct a multi-level PSH for each 3D model. The size of the hash table is set as the smallest
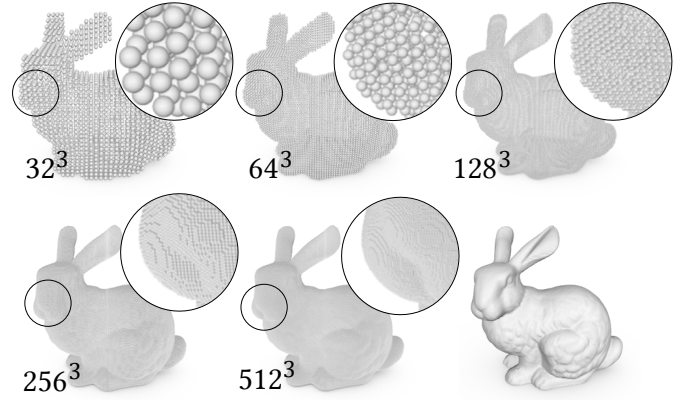

Fig. 5. The benefit of dense voxelization is obvious. The discretized model better captures the geometry of the original shape at higher resolutions.

value satisfying $\bar{m}^3 > |\mathcal{S}|$. Each hash table slot is an `int` type, which stores the data array index of D. Therefore, the hash table supports the high-resolution models up to $|\mathcal{S}| = 2^{31}$, which is sufficient in our experiments. Next, we seek to make the offset table as compact as possible. The table size $\bar{r}$ is initialized as the smallest integer such that $\bar{r}^3 \geq \sigma|\mathcal{S}|$ with the factor σ empirically set as $\sigma = 1/2d$, as in [16]. An offset table cell is of 24 bits ($d \times 8$), and each offset value is a 8-bit `unsigned char`, which allows an offset up to 255 at each dimension. We used the greedy algorithm to assign the offset in a descending order of how many collisions h_1 has. If the hash construction fails, we increase $\bar{r}$ by $\sqrt[3]{2}$ (i.e. double the offset table capacity) until construction succeeds. We refer readers to [16] for implementation details.

6.3 Memory Analysis

An important advantage of using PSH is its excelling memory performance over state-of-the-art methods. Our closest competitor is OCNN [15], where the total number of the octants at the finest level does not depend on whether leaf octants intersect with the input model. Instead, it is determined by the occupancy of its parent: when the parent octant overlaps with the model's boundary, all of its eight children octants will be generated. While OCNN's memory consumption is quadratically proportional to the voxel resolution in the asymptotic sense, it also wastes $\mathbf{O}(N^2)$ memory for leaf octants that are not on the model. On the other hand, the memory overhead of our PSH-based data structure primarily comes from the difference between the actual model size i.e. the number of voxels on the model at the finest level and the hash table size (the offset tables are typically much smaller than the main hash table). Assume that the input model size is $|\mathcal{S}| = N^2$. The hash table size is $\bar{m} = \lceil N^{\frac{2}{3}} \rceil$, which is the smallest integer satisfying $\bar{m}^3 > N^2$. By splitting $\lceil N^{\frac{2}{3}} \rceil$ as: $\lceil N^{\frac{2}{3}} \rceil = N^{\frac{2}{3}} + \Delta M$, $0 \leq \Delta M \leq 1$, the memory overhead of PSH can then be estimated via:

$$\lceil N^{\frac{2}{3}} \rceil^3 - N^2 = \left(N^{\frac{2}{3}} + \Delta M\right)^3 - N^2 \propto \Delta M N^{\frac{4}{3}}, \qquad (9)$$

which is $\mathbf{O}(N^{\frac{4}{3}})$. In other words, the memory overhead of our HCNN is *polynomially smaller* than OCNN.

Figure 6 compares the sizes of the primary data structure for the bunny model (Figure 5) using OCNN and HCNN – the total number of leaf octants and the size of the hash table (H) at the finest level. The size of the offset table Φ is typically an order
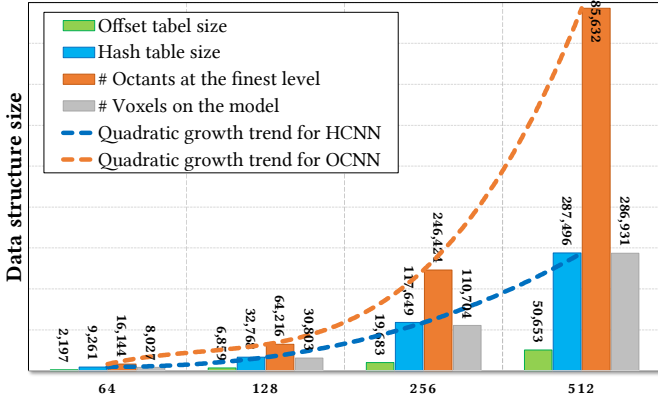
Fig. 6. The sizes of PSH and octree data structures used to encode the bunny model under resolutions of 64^3, 128^3, 256^3 and 512^3. Under each resolution, the total number of octants and the sizes of the hash table (H) and offset table (Φ) are reported. Their quadratic growth trends are also plotted.

smaller than H. Besides, the number of voxels on the model is also reported. It can be clearly seen from the figure that the size of the hash table is very close to the actual model size (i.e. the lower bound of the data structure). The latter is highlighted as grey bars in the figure. The asymptotic spatial complexity of both HCNN and OCNN are $\mathbf{O}(N^2)$, however the plotted growth trends show that HCNN is much more memory efficient than OCNN.
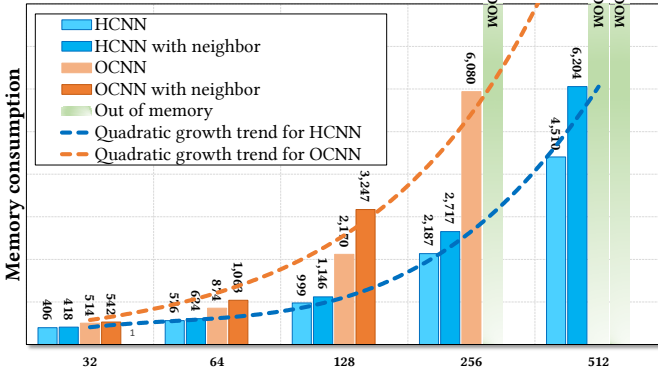

Fig. 7. The actual memory consumption using OCNN and HCNN over a mini-batch of 32 models. The physical memory cap of the 1080 GTX card is 8 GB. HCNN allows very dense voxelization up to 512^3 even with pre-stored neighbor information, while OCNN can only handle resolution of 128^3 with recorded neighborhood.

In reality, the memory footprint follows the similar pattern. Figure 7 compares the memory usage for OCNN and HCNN during the mini-batch training. A mini-batch consists of 32 random models, and memory usage is quite different for different batches. Therefore, we report the batch which uses the largest amount of memory during 1,000 forward and backward iterations. It can be seen from the figure that when the resolution is 256^3, OCNN consumes $6,080$ MB memory, and our method just needs $2,187$ MB memory. This is over 170% less memory consumption. When the resolution is further increased to 512^3, OCNN is unable to fit the entire batch into 8 GB memory of the 1080 GTX video card, while our method is not even close to the cap, which only uses $4,510$ MB memory. If one chooses to use the entire voxel grid, a mini-batch would need over 2 GB memory (with

nVidia cuDNN) under resolution of 64^3, which is roughly four times of HCNN. During CNN training, one could accelerate the convolution-like operations by saving the neighborhood information for each non-empty voxel (or each leaf-octant with OCNN). With this option enabled, OCNN is even not able to handle the batch under 128^3, while our method is sill able to deal with the batch under 512^3. The plotted growth trends also suggest that the gap of the memory consumption between OCNN and HCNN should be quickly widened with the increased voxel resolution.

Network architecture	Without voting	With voting
HCNN(32)	89.3%	89.6%
OCNN(32)	89.3%	89.8%
FullVox(32)	89.3%	89.8%
HCNN(64)	89.3%	89.9%
OCNN(64)	89.3%	89.8%
FullVox(64)	89.0%	89.6%
HCNN(128)	89.4%	90.1%
OCNN(128)	89.2%	90.0%
HCNN(256)	89.2%	90.2%
OCNN(256)	89.2%	90.2%
HCNN(512)	89.1%	89.6%
OCNN(512)	OOM	OOM
VoxNet(32)	82.0%	83.0%
Geometry image	83.9%	–
SubVolSup(32)	87.2%	89.2%
FPNN(64)	87.5%	–
PointNet	89.2%	–
VRN(32)	89.0%	91.3%

TABLE 1

Benchmark of shape classification on ModelNet40 dataset. In the first portion of the table, we report the classification results using HCNN and OCNN. The classification accuracy using fully voxelized models (FullVox) is also reported. The number followed by a network name indicates the resolution of the discretization. In the second half of the table, the benchmarks of other popular nets are listed for the comparison. The best benchmark among a given group is highlighted in blue color.

6.4 Shape Classification

The first shape analysis task is the shape classification, which returns a label out of a pre-defined list that best describes the input model. The dataset used is ModeNet40 [12] consisting of 9,843 training models and 2,468 test models. The upright direction for each model is known, and we rotate each model along the upright direction uniformly generating 12 poses for the training. At the test stage, the scores of these 12 poses can be pooled together to increase the accuracy of the prediction. This strategy is known as *orientation voting* [13]. The classification benchmarks of HCNN under resolutions from 32^3 to 512^3 with and without voting are reported in Table 1.

In the first half of the table, we also list the prediction accuracy using OCNN [15] and FullVox under the same resolution. The notion of HCNN(32) in the table means the highest resolution of the HCNN architecture is 32^3. The so-called FullVox refers to treating a 3D model as a fully voxelized bounding box, where a voxel either houses the corresponding normal vector, as HCNN or OCNN does, if it intersects with the model's surface, or a zero vector. In theory, FullVox explicitly presents the original geometry of the model without missing any information – even for empty voxels. All the CNN operations like convolution and pooling are applied to both empty and non-empty voxels. This naïve discretization is not scalable and becomes prohibitive when the voxel resolution goes above 64^3. Even under moderate resolutions

like 32^3 and 64^3, FullVox does not yield noticeably better scores than OCNN or HCNN, implying that incorporating empty voxels during CNN operations barely improves the shape analysis results. As the primary input difference between HCNN and OCNN lies in the empty voxels OCNN possesses that are not included in HCNN, it is not difficult to understand why HCNN has comparable benchmark scores as OCNN while using much less memory. The reported performance of OCNN is based on the published executable at https://github.com/Microsoft/O-CNN. As mentioned above, we shape our HCNN architecture to exactly match the one used in OCNN to avoid any influences brought by different networks. We can see from the benchmarks that under moderate resolutions like 32^3 and 64^3, HCNN, OCNN and FullVox perform equally well, and employing the voting strategy is able to improve the accuracy by another five percentages on average. When the voxel resolution is further increased, overfitting may occur as pointed out in [15], since there are no sufficient training data to allow us to fine-tune the network's parameters. As a result, the prediction accuracy slightly drops even with voting enabled.

The second half of Table 1 lists the classification accuracy of some other well-known techniques including VoxNet [13], Geometry image [24], SubVolSup [6], FPNN [49], PointNet [28] and VRN [50]. VRN was originally designed for object classification. Its best classification accuracy comes from a very deep architecture (i.e. 45 layers) based on ResNet, and it requires 6 days of training. We also noticed that the performance of OCNN in our experiment is slightly different from the one reported in the original OCNN paper. We suspect that this is because different parameters used during the model rectification stage (i.e. the magnitude of the dilation).

Network architecture	32^3	64^3	128^3	256^3	512^3
HCNN	25.2	73.1	217.3	794.3	2594.2
OCNN	27.5	78.8	255.0	845.3	OOM
OCNN with neighbor	24.0	72.0	244.4	OOM	OOM
HCNN with neighbor	22.9	67.9	205.4	772.7	2555.5
FullVox	39.7	269.0	OOM	OOM	OOM

TABLE 2
Average forward-backward iteration speed using HCNN, OCNN and FullVox (in *ms*). For a fair comparison, we exclude the hard drive I/O time.

Compact hashing also improves the time performance of the networks. Table 2 reports the average forward-backward time in *ms* over 1,000 iterations. We can see that HCNN is consistently faster than OCNN regardless if the neighbor information is pre-recorded, not to mention the FullVox scheme. The timing information reported does not include the hard drive I/O latency for a fair comparison. In our experiment, HCNN is typically 10% faster than OCNN under the same resolution.

6.5 Shape Retrieval

The next shape analysis task is the shape retrieval. In this experiment, we use the `ShapeNet Core55` dataset, which consists of 51,190 models with 55 categories. Subcategory information associated with models is ignored in this test. 70% of the data is used for training; 10% is used for validation, and the rest 20% is for testing. Data augmentation is performed in this test by rotating 12 poses along the upright direction for each model. The orientational pool is also used [6], [7]. The neural network produces a vector of the category probability scores for an input query model, and
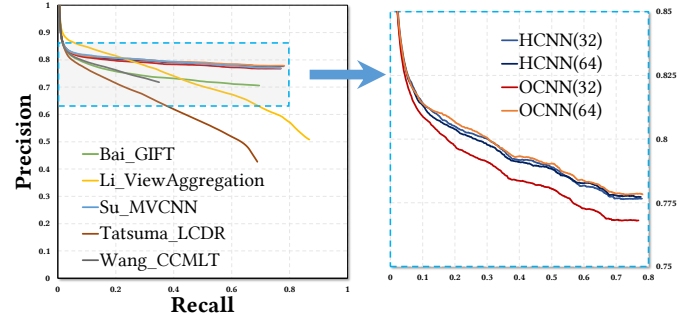

Fig. 8. The precision recall curves for HCNN, OCNN as well as other five famous multi-view CNN methods from SHREC16. The difference between HCNN and OCNN (under resolutions of 32^3 and 64^3) is quite subtle even after zooming in.

the model is considered belonging to the category of the highest score. The retrieval set corresponding to this input query shape is a collection of models that have the same category label sorted according to the L-2 distance between their feature vectors and the query shape's. Precision and recall are two widely-used metrics, where precision refers to the percentage of retrieved shapes that correctly match the category label of the query shape, and recall is defined as the percentage of the shapes of the query category that have been retrieved. For a given query shape, with more instances being retrieved, the precision drops when a miss-labeled instance is retrieved. On the other hand, recall quickly goes up since more models out of the query category have been retrieved.

The comparative precision and recall curves are shown in Figure 8. Together with our HCNN under resolutions of 32^3 and 64^3, we also plot the curves for OCNN(32) and OCNN(64) as well as several widely-known methods including GIFT [5], Multi-view CNN [7], Appearance-based feature extraction using pre-trained CNN and Channel-wise CNN [51]. The performance benchmarks of these latter methods are obtained using the published evaluator at https://shapenet.cs.stanford.edu/shrec16/. From the figure, we can see that 3D CNN methods like HCNN and OCNN outperform multi-view based methods, since the geometry information of the original models is much better encoded. The performances of HCNN and OCNN are very close to each other. After enlarging curve segments associated with HCNN(32), HCNN(64), OCNN(32) and OCNN(64) within the precision interval of $[0.75, 0.85]$, one can see that OCNN(32) is slightly below (worse) the other three.

Another interesting finding is that HCNN seems to be quite inert towards the voxel resolution. As shown on the right, HCNN(32) already has a very good result while further increasing the resolution to 512^3 does not significantly improve the performance. Curves for HCNN(32) to HCNN(512)

are hardly discernible. We feel like this actually is reasonable since identifying a high-level semantic label of an input 3D model does not require detailed local geometry information in general – even a rough shape contour may suffice. Similar conclusion can be drawn when evaluating the retrieval performance using other metrics as reported in Table 3. Here in addition to precision and recall, we
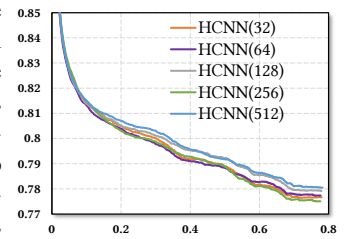
also compare the retrieval performance in terms of mAP, F-score and NDCG, where mAP is the mean average precision, and F-score is the harmonics mean of the precision and recall. NDCG reflects the ranking quality and the subcategory similarity. It can be seen from the table that, HCNN has a comparable performance as OCNN does. Both outperform multi-view based methods.

Method	P@N	R@N	mAP	F1@N	NDCG
Tatsuma_LCDR	0.427	0.689	0.728	0.472	0.875
Wang_CCMLT	0.718	0.350	0.823	0.391	0.886
Li_ViewAgg.	0.508	0.868	0.829	0.582	0.904
Bai_GIFT	0.706	0.695	0.825	0.689	0.896
Su_MVCNN	0.770	0.770	0.873	0.764	0.899
OCNN(32)	0.768	0.769	0.871	0.763	0.904
OCNN(64)	0.778	0.782	0.875	0.775	0.905
HCNN(32)	0.777	0.780	0.877	0.773	0.905
HCNN(64)	0.777	0.777	0.878	0.772	0.905
HCNN(128)	0.778	0.779	0.878	0.774	0.906
HCNN(256)	0.775	0.776	0.878	0.773	0.906
HCNN(512)	0.780	0.783	0.874	0.777	0.906

TABLE 3
Shape retrieval benchmarks.

Method	Plane	Car	Chair	Guitar	Lamp	Table
Yi et al. 2016	81.0%	75.7%	87.6%	92.0%	82.5%	75.3%
PointNet	83.4%	74.9%	89.6%	91.5%	80.8%	80.6%
SpecCNN	81.6%	75.2%	90.2%	93.0%	84.7%	82.1%
OCNN(32)	84.2%	74.1%	90.8%	91.3%	82.5%	84.2%
OCNN(64)	85.5%	77.0%	91.1%	91.9%	83.3%	84.4%
HCNN(32)	85.4%	75.8%	91.3%	91.8%	83.3%	85.8%
HCNN(64)	85.5%	77.0%	91.3%	92.0%	83.7%	85.7%
HCNN(128)	85.6%	78.7%	91.3%	92.0%	83.6%	85.9%
HCNN(256)	85.8%	79.3%	91.4%	92.0%	84.0%	86.0%
HCNN(512)	86.8%	80.2%	91.3%	91.9%	84.0%	85.9%
HCNNP(32)	81.1%	77.2%	90.7%	90.8%	83.2%	85.3%
HCNNP(64)	85.0%	78.9%	91.5%	91.7%	83.8%	85.9%
HCNNP(128)	86.2%	79.9%	91.8%	91.9%	83.9%	86.2%
HCNNP(256)	86.3%	79.8%	91.8%	92.0%	84.1%	86.1%
HCNNP(512)	86.9%	80.1%	91.8%	91.9%	84.3%	86.2%

TABLE 4
Benchmarks for shape segmentation. HCNNP(·) refers to the benchmarks based on IoU counts over the original input point clouds under the corresponding voxel resolution.

6.6 Shape segmentation

Finally, we discuss the experimental results of the shape segmentation, which assigns each point or triangle on the input model a part category label. Our experiment is based on the dataset in [52], which adds extra semantic part annotations over a subset of models from ShapeNet. The original dataset includes 16 categories of shapes, and each category has two to six parts. Clearly, segmentation is more challenging than classification or retrieval since part segmentation often relies on local geometry features, and we would like to fully test the advantage of the high-resolution voxelization that is only possible with HCNN. On the other hand, more hierarchy levels also induce more network parameters to be tuned during the CNN training. Therefore, we only test the segmentation performance when there are sufficient training data. Again, we rotate 12 poses along the upright direction for each model to augment the dataset. The training/test split is set the same as in [15]. We consider the segmentation as a per-point classification problem, and use intersection over union (IoU) to quantitatively evaluate the segmentation quality as did in [28]. It is noteworthy that the statistics reported in [15] were actually based on IoU counts on the leaf octants. It is easy to understand that under moderate voxel resolutions, the mean IoU (mIoU) defined on the voxel grid trends to have a better benchmark than on the original point cloud because a coarse discretization could alias the true boundary between two segments on the original model. To avoid such confusion, we report benchmarks of HCNN under different resolutions on both voxel grids and the input point clouds (i.e. the so-called HCNNP in the table) in Table 4. We can see that discretizing models at higher resolutions effectively improves the segmentation quality. While the mIoU improvement may read incremental from the table, those improvements lead to a better classification of points near the segmentation boundary. As shown in Figure 10, the segmentation result improvement is visually noticeable with higher voxel resolutions.

Discussion In summary, as clearly demonstrated in our experiments, HCNN acts like a "superset" of OCNN, which we consider as the most state-of-the-art 3D CNN method and our closest competitor. The benchmarks in different shape analysis tasks of using our HCNN are at least very comparable to the ones obtained using OCNN, if not better. However, we would like to remind the reader that the memory consumption of HCNN is significantly less than OCNN and the time performance is also slightly better (i.e. ∼ 10% as reported in Table 2). As a result, HCNN allows 3D CNNs to take high-resolution models during the training. For shape classification and retrieval, the primary task for the neural network is to *reduce* a complex input 3D model to few semantic labels i.e. from a very high-dimension vector to a low-dimension one. It is not surprising to us that a dense voxelization has limited contributions towards the final benchmark result. On the other hand, a high-quality segmentation requires detailed local geometry features which is somewhat commensurate to the voxel resolution. Therefore, increasing the resolution improves segmentation result in general. Undoubtedly, being able to input high-resolution models to the CNN will broaden the 3D CNN applications and potentially allow us to leverage CNNs to deal with more challenging tasks for 3D graphics contents such as shape synthesis [53], [54]. Besides what has been discussed in the experiment, our HCNN is more versatile and is compatible with all CNN configurations like arbitrarily-strided convolution and pooling. While not yet particularly popular, research efforts have already been devoted to investigate the advantages of such irregular CNNs [55]. Our HCNN would facilitate such possible future research endeavors more friendly.

For real-time applications, our data structure can also be conveniently modified to work with the real-time GPU hashing algorithm [56]. Similar to HCNN with mini-batch (§ 4.2), we can concatenate GPU hash tables as well as their *buckets* of all the models in a mini-batch to create a super hybrid hash during the CNN training, with necessary accumulated indices tables. The PSH function code in `hash2col` (lines 21–22) and `col2hash` (lines 22–23) should be replaced by the GPU hashing functions accordingly. This hashing scheme consumes more memory than PSH. According to comparison reported in [56], GPU hashing uses $1.42|\mathbb{S}|$ memory compared with $1.16|\mathbb{S}|$ memory usage of PSH, where $|\mathbb{S}|$ is the raw data size. However, as GPU hashing does not generate $\mathbf{O}(N^2)$ empty voxels as OCNN does, it is still more memory-efficient than OCNN under high resolutions.
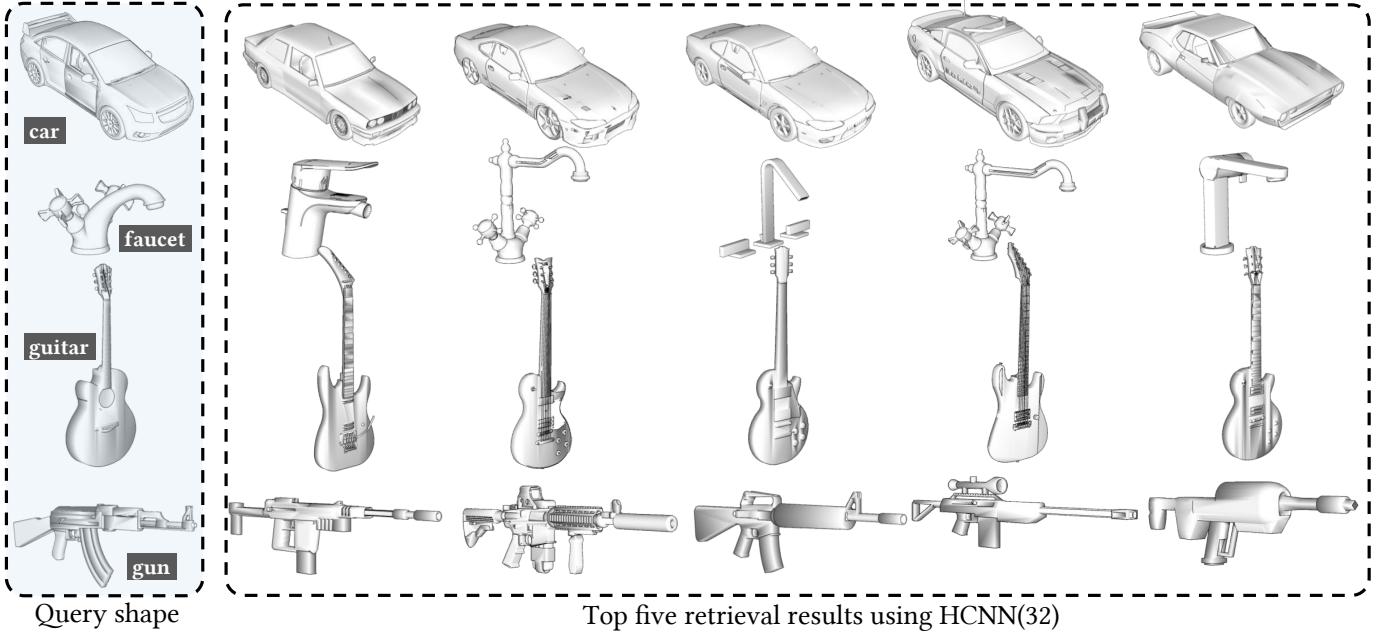
Fig. 9. The top five retrieval result of input queries of four categories, namely car, faucet, guitar and gun. The leftmost column is the query input.
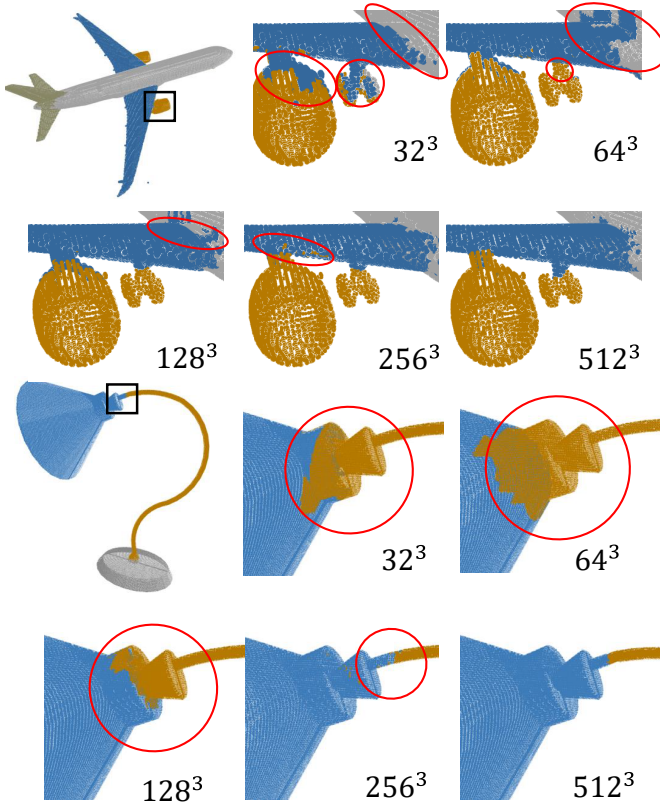

Fig. 10. In shape segmentation, high voxel resolution better captures local geometry features between adjacent object parts and yields better results.

7 CONCLUSION

In this paper, we present a novel 3D CNN framework, named HCNN, for high-resolution shape analysis. Our data structure constructs a set of hierarchical perfect spatial hashing of an input 3D model at differen resolutions. HCNN is memory-efficient, and its memory overhead is polynomially smaller than existing octree-based methods like OCNN [15]. We test the proposed HCNN for three classic shape analysis tasks: classification, retrieval and segmentation. The experimental results show that HCNN yields similar or better benchmarks compared with state-of-the-art, while reducing the memory consumption up to three times.

Thanks to its superior memory performance, HCNN allows high-resolution shape analysis that is not possible with existing methods, which paves the path of using CNN to deal with more challenging shape analysis tasks such as shape synthesis, denoising and morphing. We tested HCNN's performance with the standard 3D shape databases for an objective comparison with other peer techniques. Applying HCNN to 3D data sets other than manmade objects such as indoor scene database [57] is also an interesting future work for us.

REFERENCES

[1] R. Hu, O. van Kaick, Y. Zheng, and M. Savva, "Siggraph asia 2016: course notes directions in shape analysis towards functionality," in *SIGGRAPH ASIA 2016 Courses*. ACM, 2016, p. 8.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[4] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 3431–3440.

[5] S. Bai, X. Bai, Z. Zhou, Z. Zhang, and L. Jan Latecki, "Gift: A real-time and scalable 3d shape search engine," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 5023–5032.

[6] C. R. Qi, H. Su, M. Nießner, A. Dai, M. Yan, and L. J. Guibas, "Volumetric and multi-view cnns for object classification on 3d data," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 5648–5656.

[7] H. Su, S. Maji, E. Kalogerakis, and E. Learned-Miller, "Multi-view convolutional neural networks for 3d shape recognition," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 945–953.

[8] B. Shi, S. Bai, Z. Zhou, and X. Bai, "Deeppano: Deep panoramic representation for 3-d shape recognition," *IEEE Signal Processing Letters*, vol. 22, no. 12, pp. 2339–2343, 2015.

[9] J. Masci, D. Boscaini, M. Bronstein, and P. Vandergheynst, "Geodesic convolutional neural networks on riemannian manifolds," in *Proceedings of the IEEE international conference on computer vision workshops*, 2015, pp. 37–45.

[10] D. Boscaini, J. Masci, S. Melzi, M. M. Bronstein, U. Castellani, and P. Vandergheynst, "Learning class-specific descriptors for deformable shapes using localized spectral convolutional networks," in *Computer Graphics Forum*, vol. 34, no. 5. Wiley Online Library, 2015, pp. 13–23.

[11] B. Vallet and B. Lévy, "Spectral geometry processing with manifold harmonics," in *Computer Graphics Forum*, vol. 27, no. 2. Wiley Online Library, 2008, pp. 251–260.

[12] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao, "3d shapenets: A deep representation for volumetric shapes," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1912–1920.

[13] D. Maturana and S. Scherer, "Voxnet: A 3d convolutional neural network for real-time object recognition," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2015, pp. 922–928.

[14] G. Riegler, A. O. Ulusoy, and A. Geiger, "Octnet: Learning deep 3d representations at high resolutions," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017, pp. 6620–6629.

[15] P.-S. Wang, Y. Liu, Y.-X. Guo, C.-Y. Sun, and X. Tong, "O-cnn: Octree-based convolutional neural networks for 3d shape analysis," *ACM Trans. Graph.*, vol. 36, no. 4, pp. 72:1–72:11, Jul. 2017.

[16] S. Lefebvre and H. Hoppe, "Perfect spatial hashing," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 579–588, Jul. 2006.

[17] N. J. Mitra, M. Wand, H. Zhang, D. Cohen-Or, V. Kim, and Q.-X. Huang, "Structure-aware shape processing," in *ACM SIGGRAPH 2014 Courses*, ser. SIGGRAPH '14. New York, NY, USA: ACM, 2014, pp. 13:1–13:21. [Online]. Available: http://doi.acm.org/10.1145/2614028.2615401

[18] K. Xu, V. G. Kim, Q. Huang, N. Mitra, and E. Kalogerakis, "Data-driven shape analysis and processing," in *SIGGRAPH ASIA 2016 Courses*, ser. SA '16. New York, NY, USA: ACM, 2016, pp. 4:1–4:38. [Online]. Available: http://doi.acm.org/10.1145/2988458.2988473

[19] A. Sharif Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "Cnn features off-the-shelf: an astounding baseline for recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2014, pp. 806–813.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[21] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Region-based convolutional networks for accurate object detection and segmentation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 1, pp. 142–158, 2016.

[22] A. Boulch, J. Guerry, B. Le Saux, and N. Audebert, "Snapnet: 3d point cloud semantic labeling with 2d deep segmentation networks," *Computers & Graphics*, 2017.

[23] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.

[24] A. Sinha, J. Bai, and K. Ramani, "Deep learning 3d shape surfaces using geometry images," in *European Conference on Computer Vision*. Springer, 2016, pp. 223–240.

[25] X. Gu, S. J. Gortler, and H. Hoppe, "Geometry images," *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3, pp. 355–361, 2002.

[26] K. Guo, D. Zou, and X. Chen, "3d mesh labeling via deep convolutional neural networks," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 1, p. 3, 2015.

[27] H. Maron, M. Galun, N. Aigerman, M. Trope, N. Dym, E. Yumer, V. G. KIM, and Y. Lipman, "Convolutional neural networks on surfaces via seamless toric covers." SIGGRAPH, 2017.

[28] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," *arXiv preprint arXiv:1612.00593*, 2016.

[29] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," in *Advances in Neural Information Processing Systems*, 2017, pp. 5105–5114.

[30] R. Klokov and V. Lempitsky, "Escape from cells: Deep kd-networks for the recognition of 3d point cloud models," in *2017 IEEE International Conference on Computer Vision (ICCV)*. IEEE, 2017, pp. 863–872.

[31] D. Z. Wang and I. Posner, "Voting for voting in online point cloud object detection." in *Robotics: Science and Systems*, 2015.

[32] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *International Journal of Robotics Research (IJRR)*, 2013.

[33] M. Engelcke, D. Rao, D. Z. Wang, C. H. Tong, and I. Posner, "Vote3deep: fast object detection in 3d point clouds using efficient convolutional neural networks," in *Robotics and Automation (ICRA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 1355–1361.

[34] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[35] J. Dai, H. Qi, Y. Xiong, Y. Li, G. Zhang, H. Hu, and Y. Wei, "Deformable convolutional networks," *arXiv preprint arXiv:1703.06211*, 2017.

[36] J. Ma, W. Wang, and L. Wang, "Irregular convolutional neural networks," *arXiv preprint arXiv:1706.07966*, 2017.

[37] M. D. Zeiler, G. W. Taylor, and R. Fergus, "Adaptive deconvolutional networks for mid and high level feature learning," in *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE, 2011, pp. 2018–2025.

[38] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*. Springer, 2014, pp. 818–833.

[39] H. Noh, S. Hong, and B. Han, "Learning deconvolution network for semantic segmentation," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 1520–1528.

[40] A. Dosovitskiy, P. Fischer, E. Ilg, P. Hausser, C. Hazirbas, V. Golkov, P. van der Smagt, D. Cremers, and T. Brox, "Flownet: Learning optical flow with convolutional networks," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2758–2766.

[41] L. Zhu, Y. Chen, and A. Yuille, "Learning a hierarchical deformable template for rapid deformable object parsing," *IEEE transactions on pattern analysis and machine intelligence*, vol. 32, no. 6, pp. 1029–1043, 2010.

[42] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang, "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 1874–1883.

[43] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, 2015, pp. 448–456.

[44] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[45] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su *et al.*, "Shapenet: An information-rich 3d model repository," *arXiv preprint arXiv:1512.03012*, 2015.

[46] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[47] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[48] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting." *Journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[49] Y. Li, S. Pirk, H. Su, C. R. Qi, and L. J. Guibas, "Fpnn: Field probing neural networks for 3d data," in *Advances in Neural Information Processing Systems*, 2016, pp. 307–315.

[50] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, "Generative and discriminative voxel modeling with convolutional neural networks," *arXiv preprint arXiv:1608.04236*, 2016.

[51] M. Savva, F. Yu, H. Su, M. Aono, B. Chen, D. Cohen-Or, W. Deng, H. Su, S. Bai, X. Bai *et al.*, "Shrec16 track: large-scale 3d shape retrieval from shapenet core55," in *Proceedings of the Eurographics Workshop on 3D Object Retrieval*, 2016.

[52] L. Yi, V. G. Kim, D. Ceylan, I. Shen, M. Yan, H. Su, A. Lu, Q. Huang, A. Sheffer, L. Guibas *et al.*, "A scalable active framework for region annotation in 3d shape collections," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 6, p. 210, 2016.

[53] M. Fisher, D. Ritchie, M. Savva, T. Funkhouser, and P. Hanrahan, "Example-based synthesis of 3d object arrangements," *ACM Transactions on Graphics (TOG)*, vol. 31, no. 6, p. 135, 2012.

[54] L. Ying, A. Hertzmann, H. Biermann, and D. Zorin, "Texture and shape synthesis on surfaces," in *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, 2001, pp. 301–312.

[55] S. Han, Z. Meng, J. O'Reilly, J. Cai, X. Wang, and Y. Tong, "Optimizing filter size in convolutional neural networks for facial action unit recognition," *arXiv preprint arXiv:1707.08630*, 2017.

[56] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time parallel hashing on the gpu," *ACM Transactions on Graphics (TOG)*, vol. 28, no. 5, p. 154, 2009.

[57] I. Armeni, A. Sax, A. R. Zamir, and S. Savarese, "Joint 2D-3D-Semantic Data for Indoor Scene Understanding," *ArXiv e-prints*, Feb. 2017.