Toward Mobile 3D Vision

Huanle Zhang
University of California, Davis
Davis, California
dtczhang@ucdavis.edu

Bo Han
AT&T Labs — Research
Bedminster, New Jersey
bohan@research.att.com

Prasant Mohapatra
University of California, Davis
Davis, California
pmohapatra@ucdavis.edu

***Abstract*—In the past few years, the computer vision community has developed numerous novel technologies of 3D vision (*e.g.,* 3D object detection and classification and 3D scene segmentation). In this work, we explore the opportunities brought by these innovations for enabling real-time 3D vision on mobile devices. Mobile 3D vision finds various use cases for emerging applications such as autonomous driving, drone navigation, and augmented reality (AR). The key differences between 3D vision and 2D vision mainly stem from the input data format (*i.e.,* point clouds or 3D meshes vs. 2D images). Hence, the key challenge of 3D vision is that it is could be more computation intensive and memory hungry than 2D vision, due to the additional dimension of input data. For example, our preliminary measurement study of several state-of-the-art machine learning models for 3D vision shows that none of them can execute faster than one frame per second on smartphones. Motivated by these challenges, we present in this position paper a research agenda on offering systems support for real-time mobile 3D vision, focusing on improving its computation efficiency and memory utilization.**

***Index Terms*—3D vision, point cloud, 3D feature extraction, mobile systems, augmented reality, deep learning**

I. Introduction

There are numerous innovations in the mobile computing community on leveraging *2D vision* that takes an image as input for creative applications [1], [2], [3]. For instance, mobile continuous vision is a key building block for augmented reality (AR) [4], [5], [6] and cognitive assistance [7]. Due to the hardware constraints, continuous vision for mobile devices should be lightweight, memory efficient, and energy friendly [8]. Meanwhile, with the recent advance of 3D capturing devices, such as Microsoft Azure Kinect [9], Intel RealSense [10], and various LiDAR scanners, the computer vision community has developed novel machine learning models for *3D vision* that takes a point cloud or 3D mesh as input. Among them, Deep Neural Network (DNN) models achieve the state-of-the-art accuracy for tasks such as 3D object detection and classification and 3D scene segmentation [11], [12], [13], [14].

In this position paper, we argue that mobile devices are becoming increasingly powerful and thus it may be feasible for enabling continuous 3D vision on mobile devices such as smartphones, tablets, and headsets. Actually, both Microsoft HoloLens [15] and Magic Leap One [16] can generate 3D scans of surrounding environments. Newly released smartphones (*e.g.,* Huawei Mate 30 Pro [17]) are equipped with 3D Time-of-Flight (ToF) cameras that can resolve distance between the camera and nearby objects. AR software development kits such as ARCore from Google [18] and ARKit from Apple [19] already utilized point cloud, a popular representation of 3D objects, for plane detection on mobile devices.

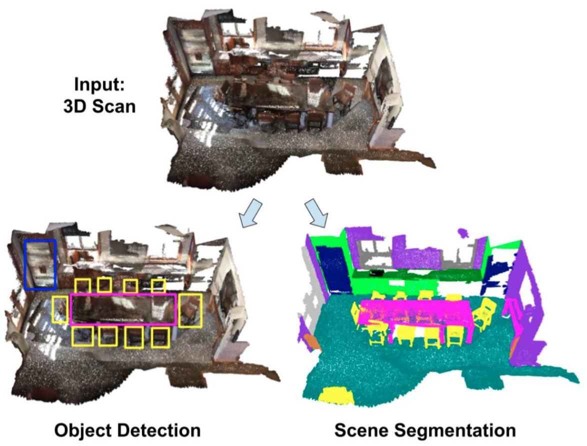


Fig. 1: Illustration of 3D object detection and scene segmentation. The input is a 3D scan of a scene that includes multiple objects. Object detection localizes the objects of interest, while scene segmentation classifies each point.

3D vision exploits depth information, which is crucial for many applications that may not be efficiently supported by 2D vision, such as AR, autonomous driving, and field inspection. AR may heavily involve interactions between human beings and virtual creations in 3D space [20], [21]. Autonomous cars use depth information for avoiding collisions [22], [23], [24]. Field inspection of network operators benefits from depth information for checking the alignment of antennas on cellular towers and the bend radius of a fiber cable. Take autonomous driving as an example. Although it can detect, via 2D images, the objects (*e.g.,* pedestrians and cars), in front of a vehicle, it is difficult to know the distance between these objects and the vehicle through 2D vision, which is essential for safe driving.

We explore whether we can smoothly execute 3D vision in real time on mobile devices. We use object detection and scene segmentation illustrated in Figure 1 as examples, because they are key components for 3D vision systems and applications. In object detection [12], each 3D object of interest is localized, whereas in scene segmentation [25], each point of the input point cloud is classified with a label (*e.g.,* a chair or a table). From the application perspective, scene segmentation can be used for object detection, which in turn can be used for object classification, but not vice versa. This paper focuses on 3D object detection and scene segmentation, rather than the classification of 3D objects, as a camera usually captures a scene rather than a single object.

To gain a deeper understanding of 3D vision, we examine several exemplary DNN models for object detection and scene segmentation, *i.e.,* ComplexYolo [26], VoxelNet [12], PointPillars [27], and submanifold SparseConvNet [25] (Section III). These models all use point clouds as their input. They apply different methods to extract features from the input 3D data, ranging from conversion to 2D images, voxelization and pillarization, to direct consumption of points. To understand the performance of these models, we measure their computation and memory overhead on commodity servers and smartphones (Section IV). Our preliminary results show that it is challenging to support real-time 3D vision on mobile devices. For example, none of these models can execute faster than one frame per second on smartphones.

Hence, we present a research agenda for improving the efficiency of mobile 3D vision (Section V). We explore several acceleration techniques, including down-sampling input, offloading to cloud, adaptive model selection, locality in continuous vision, and hardware parallelism, and identify several research challenges and opportunities of mobile 3D vision. Note that although some of the technologies have been leveraged to accelerate 2D vision on mobile devices, we cannot directly apply them to 3D vision. For example, since a mobile system may not be able to effectively determine the complexity of a point cloud due to the non-uniform distribution of sampled points in 3D space, it is difficult to partition the point cloud based on the estimated computation overhead and process each part on multiple hardware engines in parallel.

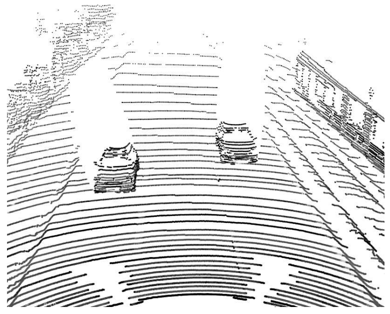
This paper is structured as follows. We present preliminaries of 3D vision in Section II. We categorize existing DNN models for 3D vision based on their feature extraction in Section III and measure their performance in Section IV. We discuss the challenges and possible solutions for mobile 3D vision in Section V. Finally, we conclude this paper in Section VI.

II. 3D Vision: A Primer

In this section, we offer a background introduction of 3D vision, focusing on the data representation.

A. 3D Vision is Essential

The most significant difference between 2D vision and 3D vision is that 3D vision can benefit from depth information, which is crucial for many applications. (1) An autonomous car not only needs to detect objects (*e.g.,* other cars and pedestrians), but also determines the distance between the objects and itself [22], [23], [24]. (2) Moving robots and flying drones need to understand their surrounding 3D environment to plan routes intelligently and avoid colliding with nearby objects [28], [29]. (3) Co-present avatars for those who are at different locations [30] can benefit from 3D vision for better understanding of the surrounding environment. (4) 3D data are also widely used in the medical sector to detect abnormal organs such as liver and kidney [31].


(a) (X, Y, Z, I) Point Cloud

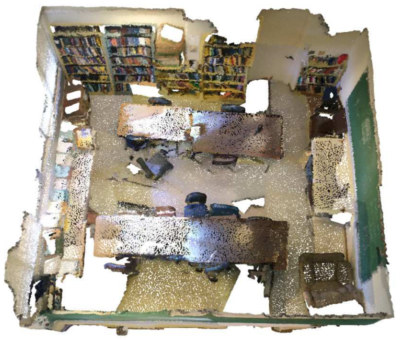

(b) (X, Y, Z, R, G, B) Point Cloud

Fig. 2: Illustration of point clouds. (a) A (X, Y, Z, I) point cloud from the KITTI autonomous driving dataset; (b) A (X, Y, Z, R, G, B) point cloud from the ScanNet indoor scene segmentation dataset.

B. 3D Data Representation

3D mesh and point cloud are two common representations for modeling 3D objects.

1) 3D Mesh: 3D polygonal mesh approximates the surface of 3D objects via a set of 2D polygons in 3D space [32]. The mesh provides an efficient and non-uniform representation of the shape of an object. It has several advantages including that (1) only a small number of polygons are required to capture a large surface; and (2) the high resolution of 3D mesh allows a faithful reconstruction and portrayal of shapes that are geometrically intricate. However, DNN models for processing 3D meshes are currently constrained on manifold meshes such as organic objects and it is not obvious how to extend existing models to a scene that includes multiple non-isometric objects such as furniture [11].

2) Point Cloud: A point cloud is an unordered set of points. Each point has its coordinate (X, Y, Z) and a property P. Point cloud is a simple and unified structure that avoids the combinatorial irregularities and complexities of 3D mesh, and thus it is easier for DNN models to learn from [11]. For different applications, P has different formats, such as $\varnothing$ (empty) in the ShapeNet dataset[1], I (reflectance value) in the

[1]ShapeNet part segmentation datset. https://www.shapenet.org/

KITTI dataset[2], and (R, G, B) in the ScanNet dataset[3]. Figure 2 illustrates a (X, Y, Z, I) point cloud from the KITTI dataset and a (X, Y, Z, R, G, B) point cloud from the ScanNet dataset.

Point clouds have different characteristics from 2D images and 3D meshes. First, point clouds are usually sparse, while images are dense. For example, more than 90% grid cells are empty for the ScanNet dataset and almost 100% cells are empty for the KITTI dataset when a point cloud is voxelized/partitioned into gird cells of $10cm \times 10cm \times 10cm$. Therefore, the voxelization representation of point clouds leads to severe computation and memory waste by spending resources on empty cells. On the other hand, although increasing cell size can reduce the number of empty cells, it degrades data granularity as more points are dropped during voxelization. Second, points in a point cloud are structureless. This property of point cloud is in contrast with 3D mesh in which vertices form edges and edges form faces.

This paper focuses on point cloud due to its high flexibility for modification. Besides capturing and generating point clouds from a RGB-D camera (D for depth) such as Intel RealSense [33], a point cloud can be created using SLAM (simultaneous localization and mapping) on mobile devices. Direct methods such as LSD-SLAM [34] can reconstruct the environment as a point cloud using only a monocular camera.

III. Feature Extraction from Point Clouds

Existing DNN models for computer vision problems generate various feature vectors from the input 2D images or point clouds. Different methods of feature extraction result in different degrees of data dimensionality, which in turn determines the model complexity [35]. For example, in the 2D space about 30% inference time is reduced for the same type of model structure if the image resolution (hence, the number of feature vectors) is reduced by a factor of two [36]. In this section, we categorize the mainstream methods of 3D feature extraction for point clouds, and investigate an example DNN model for each category.

A. Converting to 2D Feature Vectors

A point cloud can be represented by images generated from different viewpoints and viewing angles, and then a 2D DNN model is applied to these images for the vision task. ComplexYolo [26] is a simple case in that it generates only one image for a point cloud. In ComplexYolo, a point cloud is mapped to a RGB image of 1024×512 pixels, and a feature vector of length 3 is generated for each pixel. Therefore, the inference of ComplexYolo is based on 1.6M ($1024 \times 512 \times 3$) features. Finally, the results of 2D object detection are projected back to the point cloud.

This method has the following merits: (1) It can leverage widely available models for 2D vision; and (2) Its computation and memory overheads are small because these DNN models are generally much more lightweight than ones for 3D. However, models of this category have the drawback that their accuracy is low because feature vectors for 2D data cannot completely represent 3D information, and thus detailed relationship among 3D points may be lost during feature vector generation. In addition, it is difficult to apply this type of model structure to 3D semantic segmentation in which each point needs to be classified.

[2]KITTI autonomous driving dataset. http://www.cvlibs.net/datasets/kitti/

[3]ScanNet indoor scene segmentation dataset. http://www.scan-net.org/

B. A Feature Vector for Each Grid Cell

A point cloud could be voxelized/partitioned into grid cells and then a feature vector is generated for each cell. At most one point is remained in each grid cell, either by randomly selecting a point or selecting the nearest point to the center of that cell. A representative work is VoxelNet [12]. In VoxelNet, a feature vector is generated for each cell *no matter it is empty or not*. With this feature representation, a 3D convolutional network is built for detecting objects. Specifically, VoxelNet adopts a grid cell size of $0.4m \times 0.2m \times 0.2m$ for the point cloud of $4m \times 80m \times 70.4m$, and a feature vector length of 128 is used for each grid cell. Therefore, the inference of VoxelNet is based on 180.2M features.

This method provides the flexibility that one can adjust the grid cell size for different accuracy targets. The smaller the grid cell size, the less information loss the voxelized point cloud has. However, reducing the grid cell size increases the computation and memory overhead dramatically because it significantly increases the number of cells. Furthermore, models with voxelized point clouds are challenging to train as the total size of feature vectors is enormous [11], [35].

C. A Feature Vector for Each Pillar

Different methods have been proposed to reduce the number of grid cells. An example work is PointPillars [27], which partitions a point cloud into pillars and then generates feature vectors for only *non-empty* pillars that have points. In other words, only a grid cell is used for all points from the bottom to the top within a given region. Specifically, PointPillars generates an average of 5719 non-empty pillars for a point cloud from the KITTI dataset and uses a feature length of 64 for each pillar. Therefore, the inference of PointPillars is based on $\sim$0.4M features.

This method has the advantage of reducing the computation and memory overhead compared to traditional voxelization. However, the data granularity is worsened as only a single feature vector is generated for a pillar that may include many points. In addition, a pillar may include points from multiple objects for complicated scenes, which makes 3D object detection and scene segmentation challenging. Furthermore, it does not generalize well to object layouts (*e.g.,* paintings on wall may prefer horizontal pillarization but cars on street may prefer vertical pillarization).

D. A Feature Vector For Each Point

There are DNN models that have been proposed to consuming points directly and thus avoid voxelization/partitioning, which has the tug-of-war between resource overhead and accuracy. An example is submanifold SparseConvNet [25].

Model	ComplexYolo	VoxelNet	PointPillars	SparseConvNet
Application	Detection	Detection	Detection	Segmentation
Dataset	KITTI	KITTI	KITTI	ScanNet
#Points	17.7K	18.4K	18.9K	158.8K
Accuracy	64.9% AP	66.8% AP	74.1% AP	72.5% IoU
#Features	1.6M	180.2M	0.4M	1.0M

TABLE I: Comparison of the selected DNN models. During inference, the models make predictions based on different numbers of input features.

Model	ComplexYolo	VoxelNet	PointPillars	SparseConvNet
Memory	0.4GB	76.0GB	0.6GB	1.9GB
Time	0.3s	27.1s	1.3s	1.8s

TABLE II: Memory usage and execution time of selected DNN models on a commodity server.

SparseConvNet takes the raw point cloud data as feature vectors. In other words, a point has a feature vector of (X, Y, Z, R, G, B), which has a length of 6. For a point cloud from the ScanNet dataset that has an average of 158.8K points in a point cloud, the inference of submanifold SparseConvNet is based on $\sim$1.0M features.

This method is efficient with regards to computation and memory because a point cloud is highly sparse. However, for very dense point clouds, the computation overhead of this type of DNN models may be higher than others, as there is no preprocessing of the input data.

E. Model Comparisons

Table I tabulates details of the selected DNN models. ComplexYolo, VoxelNet, and PointPillars are 3D object detection models. They preprocess the point clouds and thus the numbers of input points are slightly different for the same KITTI dataset. Submanifold SparseConvNet is a 3D scene segmentation model. The table also shows the accuracy reported by these papers. As expected, ComplexYolo has the lowest Average Precision (AP) in detecting cars as it relies on 2D features. SparseConvNet achieves a high Intersection over Union (IoU) and is one of the most accurate models for the ScanNet segmentation competition. During inference, these models make predictions based on different sizes of features, from 0.4M in PointPillars to 180.2M in VoxelNet.

IV. Preliminary Results

To understand the performance of ComplexYolo, VoxelNet, PointPillars, and submanifold SparseConvNet, we implement them[4] and measure their computation and memory overhead on a server (Dell PowerEdge T640 with 40 2.2GHz CPU cores) and two smartphone models (Huawei Mate 20 and Google Pixel 2). We use Tensorflow for ComplexYolo, VoxelNet and PointPillars, and Pytorch for submanifold SparseConvNet for the implementation on the commodity server. We use Tensorflow Lite on phones for the first three models as it provides optimized acceleration for executing DNN models on mobile devices, *e.g.,* removing operators that are not used in inference. We leave it as future work to implement SparseConvNet on smartphones. Unless otherwise stated, we conduct experiments on CPUs.

Table II tabulates the execution time per point cloud and the memory consumption for these models on the server. We can see that different categories of models have dramatic performance difference: 90$\times$ in speed and 190$\times$ in memory overhead when comparing the heaviest model VoxelNet with the lightest model ComplexYolo. In addition, we have the following observations for each model: (1) ComplexYolo is lightweight because it converts a point cloud to 2D feature vectors and is based on a model for 2D images. However, its accuracy is low (Table I); (2) VoxelNet is extremely slow and requires tremendous memory because the model generates a large number of grid cells and features; (3) PointPillars dramatically reduces the computation and memory overhead compared to VoxelNet thanks to the reduced number of grid cells. Although PointPillars has a fewer number of input features than ComplexYolo, it adopts a more complicated model structure because the problem of 3D object detection is more challenging than the 2D one; (4) Submanifold SparseConvNet is efficient. Even though it takes a significantly larger number of points as input and is designed for scene segmentation which is more challenging than object detection, it requires only 1.9GB memory and on average processes a point cloud every 1.8 seconds.

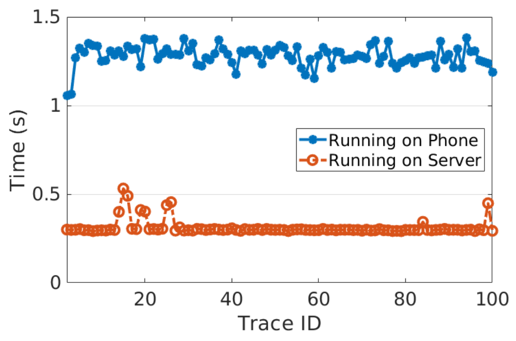
We compare the execution time of the models running on the Huawei Mate 20 phone versus on the server. Figure 3a shows the execution time of ComplexYolo for 100 runs. On average, Huawei Mate 20 takes 1.3 seconds per point cloud, which is 3.9 times slower than the server. Figure 3b shows the execution time of PointPillars for 100 runs. The phone runs 375.5 times slower than the server. This is because the Tensorflow Lite does not support the variable-length 1D convolutional layer[5], which is used by PointPillars to generate the feature vectors of all the pillars. Instead, we trigger a function call to generate a feature vector for each non-empty pillar on the mobile device, which incurs a significant delay. We expect similar performance comparison of PointPillars on the phone versus the server as ComplexYolo, if Tensorflow Lite supports the variable-length 1D convolutional layer.

An intuitive way to speed up the execution of a DNN model is to run it on GPU. However, a practical challenge is that if some model operators are not supported by the GPU, Tensorflow Lite has to execute only a part of the model on GPU and the remaining on CPU. In this case, due to the high cost of synchronizing CPU and GPU, the model execution with GPU may even be slower than when the model is run solely on the CPU[6]. Take ComplexYolo as an example: the execution

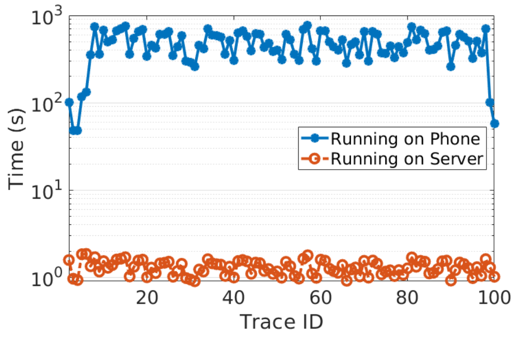
[4]We re-implement the model structures of ComplexYolo, VoxelNet, and PointPillars in Tensorflow.

[5]Tensorflow Lite compatible operations. https://www.tensorflow.org/lite/guide/ops_compatibility.

[6]Tensorflow Lite non-supported models and ops of GPU results in performance slower than running on CPU alone. https://www. tensorflow.org/lite/performance/gpu.



(a) ComplexYolo



(b) PointPillars

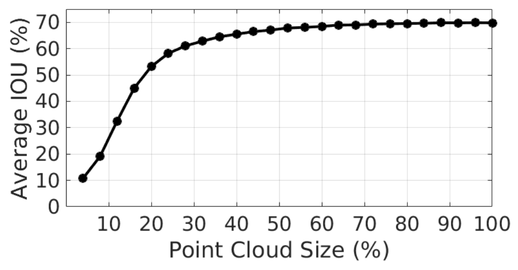
Fig. 3: Comparison of execution time of models running on a commodity server and a Huawei Mate 20 smartphone. The phone is (a) 3.9× slower if Tensorlfow Lite supports the model, and (b) 375.5× slower otherwise.

time increases from 1.3 second for CPU alone to 2.3 seconds with GPU/CPU on Huawei Mate 20, and 2.6 seconds for CPU alone to 3.4 seconds with GPU/CPU on Google Pixel 2.

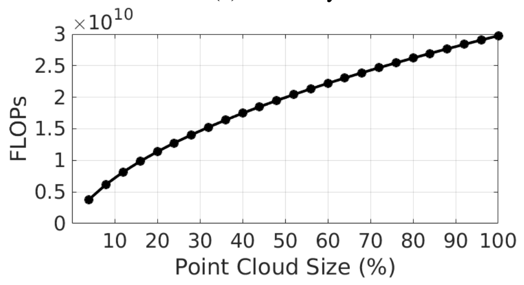
Summary. From the measurement results, we can see that it is challenging to support 3D vision in real time on mobile devices. None of these models can execute faster than one point cloud per second. As a continuous vision system, it is typically preferred to be faster than a dozen Hz, and thus there is still a gap between the current and the desired speeds. Besides, these models require larger than 0.4GB memory, which is demanding for smartphones since memory is shared by many applications. In summary, there are tremendous opportunities for significantly reducing the computation and memory overhead of DNN models for enabling real-time 3D vision on mobile devices.

V. Research Agenda

Our preliminary measurement results above demonstrate the challenges of mobile 3D vision. We identify five areas for potentially improving the efficiency of executing 3D vision in real time on mobile devices, by presenting the challenges and possible solutions for point cloud based object detection and scene segmentation.



(a) Accuracy



(b) Computation Overhead

Fig. 4: The pre-trained model performance versus the reduced point cloud size with random simplification. The performance at 100% point cloud size represents the model performance without data simplification.

A. Down-sampling Input

The input size significantly affects the DNN model complexity, because a more powerful DNN model is needed for fitting a large input [35]. For example, about 30% inference time is reduced for the same type of model structure if the image resolution is reduced by a factor of two [36]. Researchers find that down-sampling input can not only reduce the system overhead, but may also improve the model accuracy. The accuracy can be improved because down-sampling can avoid unnecessary details of input and scale objects that are too big [37]. However, deciding the down-sampling factor for each given input is not trivial. In the literature, AdaScale [37] trains several 2D object detection models for different image resolutions, and designs a neural network to predict the optimal down-sampling factor for each input image. Based on the prediction, the input image is down-sampled and then the corresponding model for that image resolution is used.

We found that instead of training several models for each point cloud size, we can use a single pre-trained model for point clouds of any size. Figure 4 shows the performance of a pre-trained model with regards to the accuracy and computation overhead (in Floating Point Operations, FLOPs) using different sizes of randomly simplified point clouds. The performance at 100% point cloud size represents the model performance without data simplification (*i.e.*, the pre-trained model using full-size point clouds). We have the following observations.

1) Model Accuracy. The IoU remains almost the same even when only circa 60% points are used. It slightly

decreases to 60% IoU when point clouds are simplified to 27% of the original size. The results indicate that real-world point clouds are highly redundant for a pre-trained 3D semantic segmentation model, the SparseConvNet model [25]. The accuracy plummets when the sizes of simplified point clouds are small, *e.g.,* smaller than 20% of the original size.

2) Computation Overhead. As we can see, FLOPs are approximately linearly correlated with the point cloud size. That is, the smaller the point cloud size, the less computation overhead the model has. A point cloud of 50% points takes about 2/3 FLOPs of the full-size point cloud. Therefore, it is preferred to sparsify point clouds as long as the accuracy does not significantly drop.

However, unlike AdaScale, it is still unknown how to predict the optimal down-sampling factor for each point cloud, which is part of our ongoing work.

B. Offloading

A cloud server, either at a central data center or at the network edge, is usually equipped with much more powerful hardware than mobile devices. Therefore, offloading computation-intensive tasks to the cloud can alleviate hardware constraints of mobile devices [38]. However, offloading raw data is not always practical because of the large size and privacy concerns. For example, the size of a ScanNet point cloud is 2.4 MB (158.8K points, 32 bits for a coordinate, and 8 bits for a color). The standard of point cloud compression is still under active development and there is no widely adopted hardware accelerator for point cloud compression.

To reduce the upload data size, we can either offload intermediate results of data processing or offload partial raw data. (1) *Intermediate Result Offloading*. For example, VisualPrint [39] extracts image features on a smartphone and uploads the fingerprints of these features to a cloud for further processing. (2) *Partial Raw Data Offloading*. The authors of [6] propose to offload Regions of Interest (RoI) of images for object detection. These RoI sub-images are likely to contain the target objects. However, it remains an open problem on efficiently identifying RoIs for point clouds. Moreover, the pre-processing of raw data on mobile devices may potential increase the end-to-end latency. Hence, an interesting research direction is to investigate how to balance the tradeoffs between accuracy, bandwidth, and end-to-end latency for continuous mobile 3D vision.

C. Model Selection

There are generic techniques to reduce the complexity of DNN models, including parameter quantization [40] and network pruning [41]. Parameter quantization reduces the number of bits to represent each weight and pruning can be applied to remove redundant connections or neurons. However, these standalone techniques do not consider the run-time resources of mobile devices. For example, a smartphone that has many background processes may prefer a fast model in order to reduce the end-to-end latency; a smartphone that has limited remaining battery may prefer an energy-efficient (*e.g.,* less memory access) model.

A possible solution is to automatically select a model from a pool that balances various resource-accuracy tradeoffs [2], [42]. As cameras output images of the same resolution, the models' computation and memory overhead can be determined in advance to facilitate the selection. By comparison, a 3D scanner generates point clouds with different number of points, *e.g.,* higher point density for furniture than walls. For example, the point clouds of ScanNet have an average number of 158.8K points, but with a minimum and a maximum number of 32.8K points and 438.6K points respectively, and a high standard deviation of 84.3K points. Therefore, different point clouds result in different computation overhead for data processing (*e.g.,* feature generation) and DNN model execution, which makes the prediction of the run-time resource overhead of point cloud based 3D vision systems and thus the model selection difficult.

D. Locality in Continuous Vision

One approach to reducing the computational workload of continuous vision is to exploit the temporal locality of consecutive images/frames. Instead of running object detection on each input separately, we can treat the video as a sequence. Object detection is only performed for two frames that are dramatically different and caching is used for frames in between. The systems of this kind typically consist of two components: an object detection and a tracker. The tracker component is used to match objects in consecutive frames and to estimate motion information.

In 2D object detection, an image is divided into several blocks and then blocks in two consecutive frames are compared for object tracking [3]. A recent advance is to apply a neural network to predict the object locations in next frames, which can greatly improve the tracker accuracy [43]. However, 2D tracking technologies may not be directly applied to point clouds because of the much larger search space of 3D domain. Recently, FlowNet3D [44] builds a DNN model for tracking scene flow in point clouds, and shows promising results. However, the computation overhead of FlowNet3D is too high to be applicable for mobile systems. Hence, designing a lightweight tracker for point clouds remains an unsolved open problem.

E. Hardware Parallelism

Smartphones are equipped with many computation resources such as CPU, GPU, and DSP. It can greatly speed up model execution if all these resources could be used in parallel. However, different hardware have varied computation capability and memory size. The heterogeneity of these hardware makes the parallelism difficult, and thus it is critical to consider the capabilities of different hardware resources when parallelizing model execution.

There are two straightforward methods of parallelizing a DNN based system. (1) *Parallilizing DNN Model.* We can partition a DNN model into three parts, and run each part on

CPU, GPU, and DSP. However, it is unclear how to optimize the model partition. The extra inter-hardware communication overhead should be taken into consideration and minimized. (2) *Parallelizing Input Data*. MobiSR [45] partitions an image into small patches, and runs these patches on multiple hardware in parallel. Each hardware runs a model tailored to its capability, and a scheduler is designed to determine which patch runs on which hardware based on the complexity of the patch. However, partitioning a point cloud is more challenging than partitioning an image. We intend to examine how to partition point clouds and how to evaluate the complexity of point clouds for scheduling.

VI. Conclusion

In this position paper, we argue that the mobile computing community should pay close attention to the recent advance of 3D compute vision and its applications on mobile devices, by exploring the emerging necessity and the associated technical challenges of real-time mobile 3D vision. Our preliminary measurement study reveals that it is not only computation-intensive, but also memory-inefficient for mobile devices to execute existing DNN models for 3D vision directly. To shed the light in this new research direction, we present a research agenda for accelerating these DNN models and point out several possible solutions to better support continuous 3D vision on mobile devices, by considering the unique characteristics of point clouds, a popular representation of 3D objects.

The discussed directions in the research agenda are by no means exhaustive. For example, it would be interesting to extend the work for simultaneously supporting multiple vision applications on mobile devices [1] to the 3D space. This paper focuses mature DNN models for 3D vision that are designed for point clouds. When the DNN models for 3D meshes are mature, we expect the emergence of different research challenges and opportunities.

References

[1] Robert LiKamWa and Lin Zhong. Starfish: Efficient Concurrency Support for Computer Vision Applications. In *Proceedings of ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2015.

[2] Biyi Fang, Xiao Zeng, and Mi Zhang. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *Proceedings of ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2018.

[3] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. DeepCache: Principled Cache for Mobile Deep Vision. In *Proceedings of ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2018.

[4] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices. In *Proceedings of ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2015.

[5] Puneet Jain, Justin Manweiler, and Romit Roy Choudhury. Overlay: Practical Mobile Augmented Reality. In *Proceedings of ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2015.

[6] Luyang Liu, Hongyu Li, and Marco Gruteser. Edge Assisted Real-time Object Detection for Mobile Augmented Reality. In *Proceedings of ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2019.

[7] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards Wearable Cognitive Assistance. In *Proceedings of ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2014.

[8] Robert LikamWa, Bodhi Priyantha, Matthai Philipose, Lin Zhong, and Paramvir Bahl. Energy Characterization and Optimization of Image Sensing Toward Continuous Mobile Vision. In *Proceedings of ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2013.

[9] Azure Kinect DK. https://azure.microsoft.com/en-us/services/kinect-dk/. [accessed on 10-May-2020].

[10] Intel RealSense Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/realsense-overview.html. [accessed on 10-May-2020].

[11] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. In *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[12] Yin Zhou and Oncel Tuzel. VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection. In *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[13] Angela Dai, Angel X. Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Niebner. ScanNet: Richly-annotated 3D Reconstructions of Indoor Scenes. In *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[14] Kaichun Mo, Shilin Zhu, Angel X. Chang, Li Yi, Subarna Tripathi, Leonidas J. Guibas, and Hao Su. PartNet: A Large-scale Benchmark for Fine-grained and Hierarchical Part-level 3D Object Understanding. In *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[15] Microsoft HoloLens 2. https://www.microsoft.com/en-us/hololens. [accessed on 10-May-2020].

[16] Magic Leap One. https://www.magicleap.com/en-us/magic-leap-1. [accessed on 10-May-2020].

[17] HUAWEI Mate 30 Pro. https://consumer.huawei.com/en/phones/mate30-pro/. [accessed on 10-May-2020].

[18] Google ARCore. https://developers.google.com/ar. [accessed on 10-May-2020].

[19] Apple ARKit. https://developer.apple.com/augmented-reality/. [accessed on 10-May-2020].

[20] Yuki Ishikawa, Ryo Hachiuma, Naoto Ienaga, Wakaba Kuno, Yuta Sugiura, and Hideo Saito. Semantic Segmentation of 3D Point Cloud to Virtually Manipulate Real Living Space. In *Proceedings of the Asia Pacific Workshop on Mixed and Augmented Reality (APMAR)*, 2019.

[21] Lei Han, Tian Zheng, Yinheng Zhu, Lan Xu, and Lu Fang. Live Semantic 3D Perception for Immersive Augmented Reality. *IEEE Transactions on Visualization and Computer Graphics*, 26(5):2012–2022, 2020.

[22] Luyang Liu, Hongyu Li, Jian Liu, Cagdas Karatas, Yan Wang, Marco Gruteser, Yingying Chen, and Richard P. Martin. BigRoad: Scaling Road Data Acquisition for Dependable Self-Driving. In *Proceedings of ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2017.

[23] Hang Qiu, Fawad Ahmad, Fan Bai, Marco Gruteser, and Ramesh Govindan. Augmented Vehicular Reality. In *Proceedings of ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2018.

[24] Fawad Ahmad, Hang Qiu, Ray Eells, Fan Bai, and Ramesh Govindan. CarMap: Fast 3D Feature Map Updates for Automobiles. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[25] Benjamin Graham, Martin Engelcke, and Laurens van der Maaten. 3D Semantic Segmentation with Submanifold Sparse Convolutional Networks. In *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[26] Martin Simon, Stefan Milz, Karl Amende, and Horst-Michael Gross. Complex-YOLO: An Euler-Region-Proposal for Real-time 3D Object Detection on Point Clouds. In *Proceedings of European Conference on Computer Vision Workshop (ECCV Workshop)*, 2018.

[27] Alex H. Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. PointPillars: Fast Encoders for Object Detection From Point Clouds. In *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[28] Andre Ückermann, Robert Haschke, and Helge Ritter. Realtime 3D Segmentation for Human-Robot Interaction. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2013.

[29] Lucas Vago Santana, Alexandre Santos Brandão, Mário Sarcinelli-Filho, and Ricardo Carelli. A trajectory tracking and 3D positioning controller for the AR.Drone quadrotor. In *Proceedings of International Conference on Unmanned Aircraft Systems (ICUAS)*, 2014.

[30] Bumsoo Kang, Inseok Hwang, Jinho Lee, Seungchul Lee, Taegyeong Lee, Youngjae Chang, and Min Kyung Lee. My Being to Your Place, Your Being to My Place: Co-present Robotic Avatars Create Illusion of Living Together. In *Proceedings of ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2018.

[31] Ozgun Cicek, Ahmed Abdulkadir, Soeren S. Lienkamp, Thomas Brox, and Olaf Ronneberger. 3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation. In *Proceedings of International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2016.

[32] Rana Hanocka, Amir Hertz, Noa Fish, Rajia Giryes, Shachar Fleishman, and Daniel Cohen-Or. MeshCNN: A Network with an Edge. *ACM Transactions on Graphics (TOG)*, 38(4):90:1–90:12, 2019.

[33] Intel. Build an Android application for Intel RealSense SDK. https://dev.intelrealsense.com/docs/build-an-android-application-for-intel-realsense-sdk. [accessed on 10-May-2020].

[34] Jakob Engel, Thomas Schöps, and Daniel Cremers. LSD-SLAM: Large-Scale Direct Monocular SLAM. In *Proceedings of Proceedings of European Conference on Computer Vision (ECCV)*, 2014.

[35] Aurelien Geron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, 2019.

[36] Jonathan Huang, Vivek Rathod, Chen Sun, Menlong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy Trade-offs for Modern Convolutional Object Detectors. In *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[37] Ting-Wu Chin, Ruizhou Ding, and Diana Marculescu. AdaScale: Towards Real-time Video Object Detection using Adaptive Scaling. In *Proceedings of Conference on Systems and Machine Learning (SysML)*, 2019.

[38] Yong Li and Wei Gao. Minimizing Context Migration in Mobile Code Offload. *IEEE Transactions on Mobile Computing (TMC)*, 16(4):1005–1018, 2017.

[39] Puneet Jain, Justin Manweiler, and Romit Roy Choudhury. Low Bandwidth Offload for Mobile AR. In *Proceedings of International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2016.

[40] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized Convolutional Neural Networks for Mobile Devices. In *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[41] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Networks with Prunning, Trained Quantization and Huffman Coding. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2016.

[42] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. On-Demand Deep Model Compression for Mobile Devices: A Usage-Driven Model Selection Framework. In *Proceedings of ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2018.

[43] Huizi Mao, Taeyoung Kong, and William J. Dally. CaTDet: Cascaded Tracked Detector for Efficient Object Detection from Video. In *Proceedings of Conference on Systems and Machine Learning (SysML)*, 2019.

[44] Xingyu Liu, Charles R. Qi, and Leonidas J. Guibas. FlowNet3D: Learning Scene Flow in 3D Point Clouds. In *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[45] Royson Lee, Stylianos I. Venieris, Lukasz Dudziak, Sourav Bhattacharya, and Nicholas D. Lane. MobiSR: Efficient On-Device Super-Resolution through Heterogeneous Mobile Processors. In *Proceedings of ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2019.