Moby: Empowering 2D Models for Efficient Point Cloud Analytics on the Edge

Jingzong Li, Yik Hong Cai, Libin Liu, Yu Mao, Chun Jason Xue, and Hong Xu
City University of Hong Kong
The Chinese University of Hong Kong
zhongguancun Laboratory

ABSTRACT

3D object detection plays a pivotal role in many applications, most notably autonomous driving and robotics. These applications are commonly deployed on edge devices to promptly interact with the environment, and often require near real-time response. With limited computation power, it is challenging to execute 3D detection on the edge using highly complex neural networks. Common approaches such as offloading to the cloud induce significant latency overheads due to the large amount of point cloud data during transmission. To resolve the tension between wimpy edge devices and compute-intensive inference workloads, we explore the possibility of empowering fast 2D detection to extrapolate 3D bounding boxes. To this end, we present Moby, a novel system that demonstrates feasibility and potential of our approach. We design a transformation pipeline for Moby that generates 3D bounding boxes efficiently and accurately based on 2D detection results without running 3D detectors. Further, we devise a frame offloading scheduler that decides when to launch the 3D detector judiciously in the cloud to avoid the errors from accumulating. Extensive evaluations on NVIDIA Jetson TX2 with real-world autonomous driving datasets demonstrate that Moby offers up to 91.9% latency improvement with modest accuracy loss over state of the art.

CCS CONCEPTS
• Information systems → Data analytics; Data streaming.

KEYWORDS
Point cloud analytics, edge computing, 3D object detection

1 INTRODUCTION

The rapid development of Deep Neural Networks (DNNs) has empowered a number of use cases, including object detection, face recognition and image super-resolution [23, 24, 61]. One of the most promising use cases is 3D object detection, which plays a pivotal role in a wide range of applications such as robotics and autonomous driving that require accurate perception of their surrounding environment to operate well [12, 46]. 3D object detection is a fundamental basis in such perception systems, and significant research effort has been devoted to improving its accuracy [36, 48, 49, 51, 59, 60]. It takes 3D sensory data represented by point cloud as input, which is generally captured by LiDAR, to generate 3D bounding boxes. An example is illustrated in Fig. 1. Compared to its 2D counterpart, 3D object detection introduces a third dimension to characterize the location and size of an object in the real world. Due to the task complexity and large amount of data to process, 3D detection models usually have more complicated structures with inflated model sizes, posing a higher demand for computation resources. We experimentally find that with the same hardware, the inference latency of 3D detection model can be up to 41× of the 2D model (§2.2). Meanwhile, edge computing is provisioned with limited computation power. For instance, the widely used TX2 [9] has significantly fewer CUDA cores (17x less) than desktop-class GPU RTX 2080Ti [4], and even fewer (31x less) than high-end GPU Tesla A100 [5]. Thus, it is notoriously difficult to run 3D detection models on edge for near real-time processing.

To address the tension between constrained computing resources and growing demand, the de-facto standard, cloud-only approach offloads compute-intensive 3D object detection to the cloud for inference. Although computation offloading shifts the heavy burden to the cloud and significantly improves inference latency, the end-to-end latency is still unsatisfactory for practical use due to transmission delay, which accounts for the majority of the latency and bottlenecks the entire pipeline (§2.2). Despite compression techniques that can reduce point cloud size, the non-negligible compression overhead still hinders real-time streaming.

To enable 3D object detection on constrained edge, we present Moby, a new framework that addresses the limitations of edge- and cloud-only inference, by using 2D models to accelerate 3D object
Moby proposes a lightweight transformation method that can run on the edge device, with only a few anchor frames offloaded to the server for 3D object detection. Moby’s approach is based on a 2D-to-3D transformation method that constructs 3D bounding boxes using outputs from the 2D model. This transformation allows Moby to generate 3D detection results on board, without running heavy 3D models that are ill-suited for edge computing.

However, creating such a full-fledged system involves two key challenges: (1) At the frame level, how can Moby accurately and efficiently transform 2D bounding boxes into 3D ones, thus maximizing the latency benefit from 2D detection? (2) Across frames, as the error of transformation accumulates over time, how can Moby monitor the accuracy drop and decide the offloading timing?

Moby addresses these two challenges by introducing a novel system design. First, Moby starts by running a fast on-board instance segmentation model to obtain both 2D detections and segmentation masks. To utilize previous 3D detection results for better transformation, a tracking-based association component is designed to establish the association of 2D bounding boxes in adjacent frames. Next, we propose the 2D-to-3D Transformation component, which is a light-weight geometric method that takes in both 2D results and point cloud to generate 3D bounding boxes efficiently. Specifically, we transfer semantic information contained in the segmentation masks to point cloud to obtain the point cluster of each potential object. Point filtration is designed to eliminate tainted points in each cluster. Then we design a sequence of geometric methods to accurately construct 3D bounding boxes by referring to previous detection results. Finally, as the error of transformation accumulates over time, a frame offloading scheduler is proposed to judiciously decide when to offload a new frame to launch 3D detectors for the subsequent transformation to utilize.

We implement Moby on an Jetson TX2 and a server equipped with RTX 2080Ti GPU. Evaluation on KITTI [27] dataset shows that, compared to existing approaches, Moby achieves up to 91.9% end-to-end latency reduction with only modest accuracy loss. It can even achieve 10 FPS on TX2, matching the scanning frequency and cloud-only inference incur severe latency overheads, making them ill-suited for latency-sensitive tasks.

We investigate the system challenges of deploying 3D detection models on edge or cloud and reveal that both edge-only and cloud-only inference incur severe latency overheads, making them ill-suited for latency-sensitive tasks.

We build Moby, the first system that enables 2D-to-3D transformation for robotics and autonomous driving applications. Moby innovatively utilizes previous detection results to enhance the transformation process. Additionally, we introduce a novel approach for coordinating edge and cloud computation using a frame offloading scheduler.

Our experiments demonstrate that Moby offers a significant latency improvement with only modest accuracy loss against several state-of-the-art 3D detection techniques.

2 BACKGROUND AND MOTIVATION
2.1 The Need for 3D Object Detection
State-of-the-art 2D detection techniques lack the depth information to localize objects and estimate their sizes, which is essential for tasks such as path planning and collision avoidance in robotics and autonomous driving [12, 46]. To overcome these limitations, 3D object detection methods are proposed. Generally, 3D object detection falls into one of three approaches based on the input modality [12]. The first approach uses the point cloud data from LiDAR or Radar sensors to generate 3D bounding boxes [36, 48, 49, 51, 60, 68]. The second one is image-based [17, 41, 47, 58, 62] that uses neural networks to recover the depth information from image planes for 3D bounding boxes, which can result in accuracy gaps compared to the point cloud approach. The third approach is fusion-based [18, 19, 44, 45, 56, 59] which uses one neural network to take both image and point cloud as inputs to improve performance. We emphasize that although Moby and fusion-based methods both utilize point cloud and images, the key difference is that fusion-based methods use one DNN to extract features from two modalities, while Moby replaces the 3D detection model with the lightweight 2D detection model for most LiDAR frames by leveraging an efficient and accurate 2D-to-3D transformation.

2.2 Challenges of 3D Detection on the Edge
Despite advancements in hardwares and deep neural networks [12, 46], performing 3D detection is still a daunting task for resource-constrained edge devices [15]. Meanwhile, frequent detection is required to track moving objects, necessitating short detection/inference time on the device. To clearly illustrate the challenges, we first measure the inference latency of 3D object detection models on a typical edge device, NVIDIA Jetson TX2 [9], with one 256-core Pascal GPU. Next, to experiment with offloading the 3D object detection models to the cloud, we build a testbed to run the models using a GPU server and replay the real-world network traces to emulate the wide-area network conditions. Table 1 summarizes the four representative point cloud-based models we measure. The point cloud data comes from the KITTI dataset [27], the most popular one in the field of autonomous driving.
Table 2: Statistics of four cellular bandwidth traces.

<table>
<thead>
<tr>
<th>Trace (Mbps)</th>
<th>Mean (± Std)</th>
<th>Range</th>
<th>P25%</th>
<th>Median</th>
<th>P75%</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCC-1</td>
<td>11.89 (± 2.83)</td>
<td>[7.76, 17.76]</td>
<td>9.09</td>
<td>12.08</td>
<td>13.42</td>
</tr>
<tr>
<td>Belgium-1</td>
<td>23.89 (± 4.93)</td>
<td>[16.02, 33.33]</td>
<td>19.84</td>
<td>23.46</td>
<td>27.73</td>
</tr>
<tr>
<td>Belgium-2</td>
<td>29.60 (± 4.92)</td>
<td>[20.17, 37.345]</td>
<td>25.18</td>
<td>30.761</td>
<td>32.76</td>
</tr>
</tbody>
</table>

Table 3: The latency and compression ratio of four common compression algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>gzip</th>
<th>zlib</th>
<th>bzip2</th>
<th>lats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compression Time (ms)</td>
<td>134</td>
<td>238</td>
<td>1007</td>
<td>1179</td>
</tr>
<tr>
<td>Compression Ratio</td>
<td>1.57</td>
<td>1.57</td>
<td>1.75</td>
<td>1.83</td>
</tr>
</tbody>
</table>

Edge-only inference. The results, shown in Fig. 2(a), indicate that: (i) The average inference latency on the device reaches 912ms for all models, which is impractical for real-world applications that require prompt environmental response; (ii) Even the one-stage models (PointPillar and SECOND) that classify bounding boxes in a single step without pre-generated region proposals have an average inference latency of 293ms and 677ms, respectively; it is not surprising that two-stage approaches take longer due to their more complex pipelines.

To gain a better understanding of the high computational cost of 3D models, we compare them against 2D detection models. We use four YOLOv5 [28] variants. \(^1\), and the input images also come from KITTI [27], captured by its car camera with a resolution of 1242×375. Results in Fig. 2(b) shows that the on-device inference latency of 2D models is much shorter. Notably, YOLOv5n only takes 33ms on average; even for the largest YOLOv5l, the inference latency is only 62.6% of the fastest 3D model PointPillar.

Cloud-only inference. Given the high latency of edge-only inference, naturally one considers the possibility of performing the inference tasks in the cloud with much more powerful hardware. However, with an average size of 6.96Mb per file, streaming point cloud data to the cloud is bandwidth-intensive and offsets the inference latency savings. To estimate the end-to-end latency of cloud-only inference, we simulate network conditions using bandwidth traces from empirical 4G/LTE datasets from the FCC [25] and Belgium [55], which are summarized in Table 2. The point cloud data is uploaded to the server using one TCP connection, and Linux TC [6] is used to throttle the link capacity according to the bandwidth traces. The server performs 3D detection on an RTX 2080Ti GPU. Note that we deliberately select the traces that cover different parts of the bandwidth spectrum, and they are all within a normal 4G/LTE cellular performance (~10–30Mbps) [1]. It is likely that the real-world performance of cloud-only inference could be worse than what we report here in certain environments [61].

As shown in Fig. 3, we observe that: (1) Even for trace with the highest average bandwidth, Belgium 2, the average end-to-end latency across four models reaches 391ms. This is much better than edge-only latency with 912ms mean latency, but it is still not satisfactory for practical use. (2) The transmission delay over the wide Internet accounts for the majority of latency. When the network condition deteriorates, the end-to-end latency grows noticeably. For instance, the inference time with FCC-1 trace is almost twice that of Belgium-2. In addition, we also consider compressing the point cloud data before transmission. We test the performance of four representative compression algorithms: gzip [5], zlib [11], bzip2 [2] and lzma [7] on TX2, whose CPU runs at 2GHz. We use the implementation in Python standard library to run these algorithms on 50 point cloud files and report the average. The results are presented in Table 3. We observe that the compression time is all above 100ms, and a larger compression ratio results in a longer time. With the fastest algorithm gzip, it can reduce the end-to-end latency of the slowest FCC-1 by 78ms. However, for higher bandwidth traces, compression does not contribute to latency reduction much and can even jeopardize it. Based on these results, it is evident that offloading all 3D frames to the cloud for inference is also impractical.

2.3 Using 2D Models for 3D Detection

To sum up, the bottleneck of 3D object detection lies in the sheer amount of data that either needs to be processed by complex models on the wimpy edge, or to be transmitted over cellular networks to the cloud. Motivated by the drastically lower inference time of 2D object detection (recall Fig. 2) and the close correspondence between the 2D and 3D bounding boxes (recall Fig. 1), we cannot help but wonder: what if we use 2D detection models to extrapolate the 3D bounding boxes? Evidently, this approach would require DNN-based 3D detection on an anchor frame to provide information about the third dimension, and 2D detection can be sufficient to effectively infer the 3D results in subsequent frames. Additionally, previous detection results of anchor frame can be incorporated to perform better transformation. To our best knowledge, these have not been explored before.

3 SYSTEM DESIGN

3.1 System Overview

Fig. 4 illustrates the overall design of Moby. We summarize its workflow into three stages as highlighted in the figure: (1) Preparation. At time t, the edge device offloads the LiDAR frame P_t (point cloud), i.e. an anchor frame, to the cloud for 3D object detection, and obtains the 3D bounding boxes to initiate the process. The 3D
The goal of tracking-based association is to utilize tracking in the 2D domain to build the mappings between bounding boxes of the same object in two adjacent frames, which serves as a key basis for 2D-to-3D transformation.

**On-device 2D Inference.** To obtain 2D bounding boxes, Moby either directly projects 3D results if the current LiDAR frame is the anchor frame, or runs a 2D detection model. Instance segmentation models are chosen as they output bounding boxes and segmentation masks simultaneously, with bounding boxes used in tracking and segmentation masks needed for the 2D-to-3D transformation.

**Kalman Filter-based Tracking.** Considering the limited computation resources, the tracking module must ensure real-time processing on edge devices and accurate inter-frame tracking. We extensively explore existing object tracking techniques in pixel domain and adopt Kalman filter-based tracking [14], which satisfies both requirements. With bounding boxes of each frame, the tracking pipeline (shown in Fig. 5) uses the Kalman filter to predict trajectories from frame $M_t$ to $M_{t+1}$ and estimate the position of boxes in the next frame. Predicted boxes are associated with 2D detection results on $M_{t+1}$ using the Hungarian algorithm [35] and Intersection-over-Union (IoU) criterion, with association rejected if the IoU is below an threshold. The Kalman filter then updates trajectory predictions using matched detections.

### 3.2 Tracking-based Association

The goal of tracking-based association is to utilize tracking in the 2D domain to build the mappings between bounding boxes of the same object in two adjacent frames, which serves as a key basis for 2D-to-3D transformation.

**On-device 2D Inference.** To obtain 2D bounding boxes, Moby either directly projects 3D results if the current LiDAR frame is the anchor frame, or runs a 2D detection model. Instance segmentation masks to 3D, we naturally choose to project the LiDAR frame to the 2D segmentation masks of the image frame in the same time slot. The point cluster of each object can be identified in 3D; points of the background that are erroneously labeled as objects are filtered out. Moby then uses a light-weight geometric method, leveraging previous detection results as reference, to estimate 3D bounding boxes based on point clusters without heavy 3D models.

**Point Projection.** As discussed in §2.3, the 3D and 2D frames captured at the same time have closely related semantic information. To transfer the semantic information contained in the segmentation masks to 3D, we naturally choose to project the LiDAR frame to the 2D domain. This projection is time-invariant since the camera and LiDAR are both fixed once they are installed, and the multimodal-sensor systems today usually provide the transformation matrices in the sensor calibration file to project points from LiDAR to camera coordinate. With the point projection, we can then mark each 3D point to indicate which object it belongs to based on the segmentation masks, by squeezing the stacked masks along the channel dimension. Then the point cluster of each potential object can be identified. Fig. 7 illustrates the process and Fig. 7(d) shows the result with point clusters highlighted in different colors.

**Point Filtration.** Directly transferring segmentation masks to the point cloud may cause certain points to be erroneously marked as objects of interest and degrade accuracy. This can also be seen in Fig. 7(d), where some background points are recognized as part of a vehicle since they are projected to the same region of the vehicle in 2D [46].

---

**Figure 4:** Moby system overview.

**Figure 5:** Overview of tracking-based association.

**Figure 6:** Overview of 2D-to-3D transformation.
To filter out these tainted points, we design an algorithm to “purify” each object’s point cluster for better 3D bounding box estimation. The details are summarized in Algorithm 1. Chiefly, for each point cluster, it first calculates the distance from all points to the origin of LiDAR coordinate (line 4). Next, it searches for the nearest point to the origin, which most likely represents the object’s center based on size and heading. It then estimates the heading angle, and calculates the object center based on size and heading.

Now the heading angle can be obtained from the normal vector of one of the object’s surfaces, if we know which side this surface is. Fig. 9 shows an example of this intuition with different surfaces of an object. Based on this idea, Moby uses the well-known RANSAC algorithm [26] to find a surface first. It iteratively selects three random points to form a plane until a best-fitting plane is found with the most inliers in it, with its normal vector \( \mathbf{v} \). Fig. 8(c) shows the plane it finds on the point filtration output. This plane can be either the front or rear surface as in Fig. 9(a), or the side surface as in Fig. 9(b). To determine the correct heading, Moby calculates the angle between the normal vector \( \mathbf{v} \) of the surface and the heading direction \( \mathbf{h}^{t−1} \) of this object’s associated bounding box in the previous LiDAR frame. If the angle is less than a threshold \( \xi \), it means the normal vector \( \mathbf{v} \) is nearly parallel to \( \mathbf{h}^{t−1} \), and the surface is the rear or front as shown in Fig. 9(a). Further, since the object moves continuously and is physically impossible to change its heading dramatically in one frame (i.e. 0.1s in KITTI), Moby obtains the heading direction \( \mathbf{h}^t \) directly from \( \mathbf{v} \).

3D Bounding Box Estimation

Moby now strives to estimate each object’s 3D bounding box based on its point cluster. A 3D bounding box is represented by a seven-tuple \([x, y, l, w, h, \theta]\), including the object’s center \([x, y, z]\), size \([l, w, h]\), and heading angle \(\theta \) relative to the \(x\) axis on the \(x−y\) plane of LiDAR coordinates. It is challenging to estimate all parameters solely based on the values of points, especially the heading and center, for two main reasons: 1) The point cloud is sparse and irregular; 2) Only part of the object has a point cloud because of the self-occlusion problem [46].

We introduce a novel approach for accurately and efficiently estimating 3D bounding boxes. For each point cluster that has been associated with an object from the previous LiDAR frame by the association module, Moby first obtains the object’s size directly from the previous frame’s result (size of the same object does not change). It then estimates the heading angle, and calculates the object center based on size and heading.

To filter out these tainted points, we design an algorithm to “purify” each object’s point cluster for better 3D bounding box estimation. The details are summarized in Algorithm 1. Chiefly, for each point cluster, it first calculates the distance from all points to the origin of LiDAR coordinate (line 4). Next, it searches for the nearest point to the origin, which most likely represents the boundary of this object and is thus called the critical boundary point (line 5). Then it calculates the distance from all points to the critical boundary point (line 7). Those points close to the critical boundary point are considered to belong to the object itself (line 8). If too few points are filtered this way, it suggests that the critical boundary point may not be the actual boundary of this object, which can happen when for instance vehicles are close to each other. We then add a small step size \( S_T \) and use the nearest point whose distance to the origin is at least \( S_T \) further away as the new critical boundary point (line 9). We repeat the process until it finds a cluster that has enough points or the number of iteration exceeds three (lines 6 and 11). Fig. 8 depicts an example of the effect of point filtration. The rationale of the algorithm is based on our observation that the points of potential objects are commonly much nearer to the origin than background points. As we measured quantitatively, our point filtration algorithm removed 98% of tainted points. After point filtration, the point cluster of each potential object is clean enough for the following 3D bounding box estimation.

**Algorithm 1: Point Filtration**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Function point_filtration(( PC_{old} ), ( Fr ), ( Mt ), ( St ));</strong></td>
</tr>
<tr>
<td>2</td>
<td><strong>for</strong> ( PC_i ) in ( PC_{old} ) <strong>do</strong></td>
</tr>
<tr>
<td>3</td>
<td>( PC_{new}, idx, iter \leftarrow { }, { }, 0; )</td>
</tr>
<tr>
<td>4</td>
<td><strong>init_list</strong> \leftarrow calculate the Euclidean distance from all points to the origin of LiDAR coordinate;</td>
</tr>
<tr>
<td>5</td>
<td>( n_j \leftarrow ) get the nearest point to the origin according to ( init_list );</td>
</tr>
<tr>
<td>6</td>
<td>( dist_list \leftarrow ) calculate the distance from all points to ( n_j );</td>
</tr>
<tr>
<td>7</td>
<td>( idx \leftarrow dist_list &lt; Mt );</td>
</tr>
<tr>
<td>8</td>
<td>( n_j \leftarrow ) get the point whose distance to origin is at least ( St ) further away;</td>
</tr>
<tr>
<td>9</td>
<td>( iter \leftarrow iter + 1; )</td>
</tr>
<tr>
<td>10</td>
<td><strong>if</strong> ( iter \rightarrow 3 ) <strong>then</strong> break;</td>
</tr>
<tr>
<td>11</td>
<td>( PC_{\text{new}}' \leftarrow PC_{\text{new}}[idx]; )</td>
</tr>
<tr>
<td>12</td>
<td>( PC_{\text{new}} \leftarrow PC_{\text{new}} \cup PC_{\text{new}}' );</td>
</tr>
<tr>
<td>13</td>
<td><strong>return</strong> ( PC_{\text{new}} );</td>
</tr>
</tbody>
</table>

**Figure 8:** Key steps of point filtration and 3D bounding box estimation for a point cluster.

**Figure 9:** Two possible situations in number of points inside both situations to determine the correct box.
we must be able to determine when a new anchor frame is needed. These are highly likely new objects that first appear in the current frame. For the result from server to continue the on-board processing, to make use of the idle computation resources on edge device during the waiting time, we propose a mechanism called recomputation to recompute the 3D results of past intermediate frames using the results of test frames. Specifically, some intermediate results such as outputs of instance segmentation model are stacked. When an anchor frame is triggered, the intermediate results and the stale output of test frame are utilized to re-run the 2D-to-3D transformation to recompute 3D bounding boxes. We observe that the recomputation time is much less than the offloading time and is completely hidden without inducing extra latency.

4 IMPLEMENTATION

Our framework is implemented with ~4K lines of Python. We use a Jetson TX2 [9] as the edge device and a desktop with Intel i7-9700K CPU and RTX 2080Ti GPU as the server. As shown in Fig. 12, they are physically connected to a Wi-Fi router and communicate via TCP. We deploy OpenPCDet [53] on the server as the inference engine. The filtering threshold $F_T$, number of minimum points $M_T$ and step size $S_T$ are empirically set to 4.5, 24 and 12 in point filtration. $\xi$ in the box estimation is 30 degrees. As for the offloading scheduler, the accuracy threshold and the gap of test frame, i.e. $Q_T$ and $N_T$, are set to 0.7 and 4, respectively.

5 EVALUATION

We now evaluate Moby by answering the following questions:

- How well does Moby work in terms of latency and accuracy?
- How much does each key design choice contribute to the overall performance gain?
- How sensitive is Moby to the key parameters?
- How efficiently does Moby utilize the edge resources?

5.1 Experimental Setup

Datasets. All experiments are done on the KITTI dataset [27], a real-world autonomous driving benchmark, with synced LiDAR scans and camera images at 10 FPS. It provides 3D ground truth for cars, pedestrians, and cyclists. Currently we only focus on the most challenging class, car, as it moves the fastest. We use the same empirical bandwidth traces as shown in Table 2.

Performance Metrics. We use the end-to-end latency and accuracy as main performance metrics. The end-to-end latency is the time between the input of point cloud cloud and the output of 3D detection results. We consider the object to be successfully detected if the 3D IoU between detection and ground truth exceeds 40% (i.e., accurately located). F1 score is used to measure accuracy, i.e. the harmonic mean of precision and recall, as widely adopted in existing work [16, 24, 50, 63, 65].

$$Q = \begin{cases} 
[a, b, c] + [0.5 \ast l \ast \cos \theta, 0.5 \ast l \ast \sin \theta, 0], & \text{if angle between } h^{-1} \text{ and } a' \text{ is less than } \xi, \vspace{1mm} \\
[a, b, c] + [0.5 \ast w \ast \cos \theta, 0.5 \ast w \ast \sin \theta, 0], & \text{otherwise.}
\end{cases}$$

Finally, we need to consider objects that are not associated with anything in the previous frame after tracking-based association. These are highly likely new objects that first appear in the current frame. Since we no longer have the reliable size information, we use the average length, width and height of all objects to estimate the new object’s size. This can be updated after Moby schedules another anchor frame to go through the 3D detector. To estimate the heading angle, Moby uses the same procedure as described above to identify a best-fitting surface based on the point clusters from point filtration. As we no longer have prior knowledge of the heading direction (from the previous LiDAR frame), we adopt an alternative method by calculating the bounding boxes for both possibilities by Eq. (2), and the one that fits more points inside is the final output as shown in Fig. 10.

3.4 Frame Offloading Scheduler

Moby’s 2D-to-3D transformation is based on the 3D detection results of the precedent anchor frame. As time goes this transformation becomes less effective simply because the scene and the objects may have changed quite much from the anchor frame. Thus from time to time, Moby needs to offload a new anchor frame to the cloud for 3D detection to assist the subsequent transformation and maintain overall accuracy. To efficiently schedule frame offloading, we must be able to determine when a new anchor frame is needed. For this purpose, a test LiDAR frame is sent to the cloud every $N_T$ frames. Detection on the test frames happen in parallel with Moby’s on-device processing. We deem the 3D detection results as ground truths to obtain the error of the 2D-to-3D transformation on the same test frame. When the error is larger than a threshold, the next LiDAR frame is designated as the new anchor frame and also sent to the cloud for inference. The subsequent on-device processing is blocked until results of the new anchor frame is received by the edge device. Fig. 11 depicts the complete process. Our design strikes a fine balance between overheads and effectiveness.

Recomputaion. Since the process of waiting anchor frame’s result is synchronous, the operational flow of Moby is blocked to wait for the result from server to continue the on-board processing. To
Models. We adopt YOLOv5n from the YOLOv5 codebase[28] as Moby’s default instance segmentation model. Unless otherwise specified, Moby uses the same 3D object detection model as the baseline systems. Note that Moby is not model-dependent: any 3D object detection and instance segmentation model can be applied.

5.2 Overall Performance

We evaluate Moby comprehensively from two different perspectives: (1) deployment approach. Here we show how well Moby performs compared to two common scenarios of edge inference, i.e., completely running on the edge and offloading to cloud; (2) acceleration method. Here we compare Moby to other methods that accelerate 3D object detection.

5.2.1 Deployment Approaches. We first compare Moby with the following two common deployment approaches:
- **Edge Only (EO):** In this scheme, the 3D models are deployed on the edge device only to run inference.
- **Cloud Only (CO):** The point cloud data is fully offloaded over 4G/5G networks to the server for inference. The end-to-end latency involves both transmission and inference delay.

Fig. 13 shows the latency comparison results using four representative point cloud based 3D detection models, as described in Table 1. Observe that Moby outperforms EO and CO in latency with significant margins ranging from 56.0% to 91.9% across models. Notably, Moby achieves the lowest latency 99ms with PointPillar in Belgium-2 trace, matching the 10 FPS frame rate in KITTI for real-time processing. The latency gain is much more salient for two-stage models, such as PointRCNN and PV-RCNN, as their architectures pose much severer computation burden. Moby delivers speedups across bandwidth traces. Even for the lowest bandwidth trace, FCC-1, Moby reduces latency by 56.0% compared to running the fastest PointPillar onboard. As the bandwidth increases, the latency gain reaches 66.2%. The latency improvement mainly comes from the design choice of replacing the 3D object detector with a much cheaper 2D model that can run on the edge device, which dramatically reduces both the inference time and transmission time (to the cloud).

Moby has little impact on 3D object detection accuracy. Fig. 13(e) reports overall accuracy for all schemes. Note that as EO and CO both run 3D object detectors for all frames, they have the same accuracy with the same model. For PointRCNN, Moby achieves almost the same or slightly higher accuracy (0.760 vs 0.751) due to the effective 2D-to-3D transformation mechanism. For the other three models, accuracy drops slightly between 0.027 to 0.056 due to: (i) possible occlusion problem in 2D images and (ii) bounding box estimation errors caused by sparse point clusters.

5.2.2 Acceleration Methods. Next, we compare Moby to three methods that accelerate 3D object detection in different ways.
- **Complex-YOLO [51]:** It converts point cloud data to birds-eye-view RGB maps and uses a light-weight YOLO-based architecture to predict 3D boxes. It is reported to achieve 50 FPS on an NVIDIA TitanX GPU [51].
- **Frustum-ConvNet [59]:** A fusion-based approach that utilizes 2D region proposals to narrow down the 3D space for acceleration.
- **Monodle [40]:** State-of-the-art image-based 3D detection approach that only use monocular images as input. For a fair comparison, we also run Moby fully onboard for the anchor frames since these methods here do not offload to cloud, and use the same 3D detection method as the respective baseline.

Based on Fig. 14(a), we observe that Moby shows significant latency improvement against the three alternative approaches. Specifically, Moby cuts down latency by 64.0% compared to Complex-YOLO with only minor accuracy loss, and the same advantage still holds against the fusion-based Frustum-ConvNet. Notably, compared with the image-based approach Monodle, Moby reduces the latency by 77.6% while improving accuracy by 5.5%. It is worth mentioning that Moby’s accuracy is hindered by Monodle’s performance compared to other approaches, as its inaccurate output indirectly affects the transformation accuracy of Moby. The relatively low accuracy of Monodle is attributed to its inability to detect locations of distant objects, which is common among image-based acceleration methods for 3D object detection [46, 62]. We also tried another two image-based methods, Deep3DBox [41] and Pseudo-LiDAR++ [62]. However, they take 2834ms and 5889ms, respectively, which are too slow to execute on edge devices.
We observe that TRS can already achieve decent accuracy, proving the efficiency of Algorithm 1.

Table 4: Impact of each design component.

<table>
<thead>
<tr>
<th>Components</th>
<th>Accuracy (max)</th>
<th>Latency (ms)</th>
<th>On-board Latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRS</td>
<td>0.762</td>
<td>88.44</td>
<td>88.44</td>
</tr>
<tr>
<td>TRS+FOS</td>
<td>0.787</td>
<td>112.96</td>
<td>89.45</td>
</tr>
<tr>
<td>TRS+FOS+TBA</td>
<td>0.814</td>
<td>99.25</td>
<td>76.29</td>
</tr>
</tbody>
</table>

Figure 15: The avg. execution time of key steps over 300 runs.

(a) Iteration number (b) Association Criterio (c) Association Criterion vs. Accuracy (d) Association Criterion vs. Latency

Figure 16: The impact of key parameters. Note that here we only report the on-board latency for clearer comparison.

5.3 Microbenchmarks

Contribution of Each Component. We evaluate the contribution of each key design component in Moby. Starting with only 2D-to-3D transformation (TRS), we incrementally add each component, namely Frame Offloading Scheduler (FOS), and Tracking-based Association (TBA). Table 4 shows the results. We observe that TRS can already achieve decent accuracy, proving the effectiveness of 2D-to-3D transformation. With FOS, accuracy is improved as some frames are offloaded to server for inference. Offloading increases the end-to-end latency while on-board latency remains unchanged. TBA further improves accuracy by using previous detection results to estimate more accurately. The slightly drop of the end-to-end and on-board latency is because TBA reduces the computation overhead of TRS if boxes are successfully associated.

Overheads. We measure the execution times of key steps of Moby and report the average in Fig. 15. Instance segmentation takes the longest, accounting for 43.9% of the onboard latency, followed by 3D bounding box estimation and point projection with 30.1% and 16.6%, respectively, because these two steps involve multiple matrix multiplications. TBA and FOS have negligible latency, taking only 0.14ms and 0.60ms. Notably, Point filtration has only 2.01ms, proving the efficiency of Algorithm 1.

5.4 Sensitivity Analysis

We also benchmark Moby by changing its key parameters.

Number of Iterations in RANSAC. The number of iterations in RANSAC is crucial in bounding box estimation and affects Moby’s accuracy and on-board latency, as shown in Fig. 16(a) and Fig. 16(b). A small number reduces latency but may lead to premature or inaccurate surface recognition, while a larger number improves robustness at the expense of computation overhead. We find that 30 iterations strike an appropriate balance between accuracy and computation cost.

Sensitivity to Association Criterion. Fig. 16(c) and Fig. 16(d) test the impact of association criterion, i.e., the IoU threshold used to associate two bounding boxes in 2D space as described in §3.2. A larger criterion makes it harder to associate bounding boxes and cannot effectively utilize previous 3D results. We empirically choose 0.3 as the association criterion, as the accuracy gain diminishes after it is greater than 0.3.

5.5 System Efficiency

Finally, we evaluate the system efficiency of Moby.

Energy Consumption. We measured Moby’s energy consumption using Tegrastats Utility [10] for 2 minutes with a 10Hz sampling rate. Moby consumes the least power compared to baselines with 73% savings on average. Its power consumption is only 24.2% of PointPillar, and reduces GPU and CPU power by 74.3% and 77.3%, respectively.

Memory Footprint. We measure the memory footprint of Moby and the results are shown in Fig. 17(b). Moby’s overall memory reduction ranges from 17.3% to 48.1%, which is expected as it only needs to load a 2D detection model into the memory that is much smaller than 3D object detection models. Moby shows its advantage in deployment on the wimpy edge devices, such as Jetson Nano [8] that only has 4GB memory shared by both CPU and GPU.

6 RELATED WORK

On-device Inference Acceleration. Lots of efforts have been devoted to accelerating DNN inference on edge devices. Model compression techniques reduce the model size and accelerate computation by network pruning [29, 30, 39], quantization [13, 21], and knowledge distillation [31, 42]. Hardware-based methods design specialized accelerators for inference pipelines using FPGA [54, 64] or ASIC [20, 57]. The last line of work is software-based methods [16, 33, 37, 52, 61]. Moby falls under the software-based category and is orthogonal to the existing work.

Edge-Cloud Collaboration. Besides on-device inference, the emergence of edge devices has also sparked research on edge-cloud collaboration. For example, filter-based methods [22, 38, 67] investigate how to prune the large data volume before transmission. Partition-based methods [32, 34, 66] explore partitioning the DNN models across edge and cloud. Feedback-based methods [16, 24, 43] rely on the feedback from server to assist local processing on the edge. Unlike existing work that only targets the 2D domain, our work explores edge-cloud collaboration for more compute-intensive 3D detection task that poses severer challenges to deploy on edge.

7 CONCLUSION

We present Moby, a novel framework for fast and accurate 3D object detection on edge devices. It leverages the 2D-to-3D transformation to significantly reduce the computational resource demand of inference, and judiciously offload the anchor frames to the cloud to maintain accuracy. Extensive evaluation shows that Moby reduces end-to-end latency by up to 91.9% with little accuracy drop.


