Resilient Sensor Fusion under Adverse Sensor Failures via Multi-Modal Expert Fusion

Supplementary Material

We present detailed pseudo-code to describe the MoME framework and AQR mechanism in MoME. In the following sections, we detail the implementation specifications of MoME. The performance of MoME is thoroughly evaluated across various sensor failure scenarios. Our comprehensive qualitative analysis further validates MoME's robustness and performance advantages.

A. Algorithms

We provide detailed pseudo-code with PyTorch including AQR and RAM in Algorithm 1 and Algorithm 2.

B. Implementation Details

B.1. Training Details

The training strategy of MoME consists of two stages to handle sensor failures. In the first stage, all object queries in Q are processed in parallel by each expert decoder and matched with ground truth through bipartite matching, without any sensor drop augmentation. The second stage focuses on handling sensor failures by applying sensor drop augmentation, where we randomly mask either the camera or LiDAR inputs, with each sensor having a $1/3$ probability of being dropped, while retaining both sensors for the remaining $1/3$ of cases.

B.2. Adverse Sensor Scenarios

Our experimental validation encompasses both sensor failure scenarios and adverse weather conditions. Our sensor failure experiments incorporate BEVFusion's [16] Beam Reduction settings and nuScenes-R's [34] protocols for LiDAR Drop, Limited FOV, Object Failure, View Drop, and Occlusion. For adverse weather conditions, we utilize scene descriptions in the nuScenes [2] validation set to identify Rainy and Night scenarios, while adopting Fog, Snow, and Sunlight conditions from nuScenes-C [6].

C. Extensive Performance Comparisons

We present additional experimental results by extending our analysis across different parameter settings for each sensor failure scenario. While Table 1 shows the results with fixed configurations, Tables 6-9 provide comprehensive evaluations with various parameter ranges for each failure case.

- **Limited FOV** (Table 6): We observe that MoME demonstrates better performance gains over CMT [29] as the field of view becomes more restricted. While both methods achieve comparable mAP scores of 71.2% and 70.3% respectively in the full FOV range of *[-180, 180]*, the performance gap widens significantly under severe FOV limitations, where MoME achieves 44.0% mAP compared to CMT's 35.0% mAP at *[-30, 30]*, demonstrating MoME's superior robustness to limited FOV.
- **Object Failure** (Table 7): We evaluate MoME and CMT with different object failure ratios. Specifically, our method achieves 72.8% NDS, outperforming CMT which achieves 71.6% NDS at *ratio=0.1*.
- **Beam Reduction** (Table 8): We analyze performance from *1* to *32 beams*, showing significant improvements especially with reduced beams, as our method achieves 30.5% mAP while CMT reaches 25.9% mAP using *1 beam*.
- **View Drop** (Table 9): MoME and CMT show gradual performance degradation as the number of dropped views increases. MoME maintains consistently higher performance, achieving 68.6% mAP compared to CMT's 68.1% mAP with *1 drop*, and the performance gap widens with *6 drops*, where MoME achieves 63.6% mAP while CMT reaches 61.7% mAP.

D. Local Feature Extraction Methods.

The AQR module selects one of the expert decoders based on features extracted from a local region identified by a query. In Table 10, we compare different feature extraction methods within our router architecture, including cross-attention, deformable attention, MLP, and our proposed approach. These methods exhibit notable performance differences in Limited FOV scenarios, where partial sensor failure occurs. This is because selecting an appropriate decoder depends on the query's position. Notably, cross-attention demonstrates limited effectiveness, as it struggles to focus specifically on locally degraded regions. Deformable attention performs better due to its dynamic spatial sampling capability, but it lacks explicit control over attention regions. While the MLP approach, which utilizes max-pooled multi-modal features, shows reasonable robustness to failure cases, it underperforms in the Clean scenario. In contrast, our proposed AQR module consistently achieves superior performance across all cases, thanks to its use of a local attention mask.

Method	Limited FOV *[-180, 180]*		Limited FOV *[-150, 150]*		Limited FOV *[-120, 120]*		Limited FOV *[-90, 90]*		Limited FOV *[-60, 60]*		Limited FOV *[-30, 30]*	
	mAP	NDS	mAP	NDS	mAP	NDS	mAP	NDS	mAP	NDS	mAP	NDS
CMT [29]	70.3	72.9	65.4	69.8	53.8	62.2	49.0	58.4	43.9	54.0	35.0	46.1
MoME (ours)	**71.2**	**73.6**	**67.8**	**71.2**	**56.2**	**63.4**	**54.2**	**61.2**	**50.6**	**58.3**	**44.0**	**53.0**

Table 6. Performance comparison between CMT [29] and MoME (ours) on *Limited FOV*

Method	Object Failure *ratio=0.0*		Object Failure *ratio=0.1*		Object Failure *ratio=0.3*		Object Failure *ratio=0.5*		Object Failure *ratio=0.7*		Object Failure *ratio=0.9*	
	mAP	NDS	mAP	NDS	mAP	NDS	mAP	NDS	mAP	NDS	mAP	NDS
CMT [29]	70.3	72.9	68.8	71.6	67.6	70.8	66.7	70.4	64.5	68.2	62.7	67.2
MoME (ours)	**71.2**	**73.6**	**70.0**	**72.8**	**68.5**	**71.8**	**67.0**	**71.0**	**64.8**	**68.8**	**63.0**	**67.8**

Table 7. Performance comparison between CMT [29] and MoME (ours) on *Object Failure*

Method	Beam Reduction *1 beams*		Beam Reduction *4 beams*		Beam Reduction *8 beams*		Beam Reduction *16 beams*		Beam Reduction *32 beams*	
	mAP	NDS	mAP	NDS	mAP	NDS	mAP	NDS	mAP	NDS
CMT [29]	25.9	42.6	54.9	62.2	59.5	65.4	62.3	67.5	70.3	72.9
MoME (ours)	**30.5**	**43.4**	**55.0**	**63.0**	**60**	**66.5**	**62.7**	**68.3**	**71.2**	**73.6**

Table 8. Performance comparison between CMT [29] and MoME (ours) on *Beam Reduction*

Method	View Drop *1 drop*		View Drop *2 drops*		View Drop *3 drops*		View Drop *4 drops*		View Drop *5 drops*		View Drop *6 drops*	
	mAP	NDS	mAP	NDS	mAP	NDS	mAP	NDS	mAP	NDS	mAP	NDS
CMT [29]	68.1	71.7	67.1	71.1	65.6	70.4	64.0	69.5	62.6	68.6	61.7	68.1
MoME (ours)	**68.6**	**72.4**	**67.7**	**71.9**	**66.6**	**71.2**	**64.6**	**70.4**	**63.8**	**69.9**	**63.6**	**69.5**

Table 9. Performance comparison between CMT [29] and MoME (ours) on *View Drop*

Method	Clean		LiDAR failure								Camera failure			
			Beam Reduction *4 beams*		LiDAR Drop *all*		Limited FOV *[-60, 60]*		Object Failure *rate = 0.5*		View Drop *6 drops*		Occlusion *w obstacle*	
	mAP	NDS	mAP	NDS	mAP	NDS	mAP	NDS	mAP	NDS	mAP	NDS	mAP	NDS
cross attention	71.0	73.6	54.6	62.8	42.3	48.1	25.7	48.0	66.0	69.8	63.0	69.4	64.1	69.3
deformable attention	71.0	73.6	54.8	62.9	42.3	48.1	32.1	50.1	66.0	70.1	63.1	69.4	64.0	69.6
MLP	70.7	73.5	54.8	62.9	42.3	48.1	44.2	54.4	66.4	70.3	63.1	69.5	64.6	69.8
AQR (ours)	**71.2**	**73.6**	**55.0**	**63.0**	**42.5**	**48.2**	**50.6**	**58.3**	**67.0**	**71.0**	**63.6**	**69.5**	**65.6**	**70.5**

Table 10. **Ablation Studies on feature extraction methods for AQR query routing.** AQR with LAM shows the superior robustness, particularly under local-aware sensor failure scenarios, such as Limited FOV, Object Failure, and Occlusion.

E. Additional Qualitative Results

Fig. 5 compares the detection results of MoME and CMT [29] under six sensor failure scenarios in real-world scenes. The visualizations demonstrate our model's consistent performance across different failure conditions.

- **Beam Reduction**: Even with reduced LiDAR input, MoME accurately detects objects in the first three rows and successfully captures vehicles behind pedestrians in the last three rows.
- **LiDAR Drop**: MoME effectively leverages camera information to detect small and partially occluded objects that CMT fails to identify.
- **Limited FOV**: MoME successfully detects small objects in LiDAR-absent regions compared to CMT.
- **Object Failure**: MoME achieves lower false positive rates than CMT and maintains accurate detection of nearby objects under object failure conditions.
- **View Drop**: MoME successfully detects occluded small objects that CMT misses and reduces false positive detec-

tions in complex scenes with dropped camera views.

- **Occlusion**:When parts of the scene are masked to simulate occlusions, MoME successfully detects objects in the affected regions while CMT shows degraded performance.

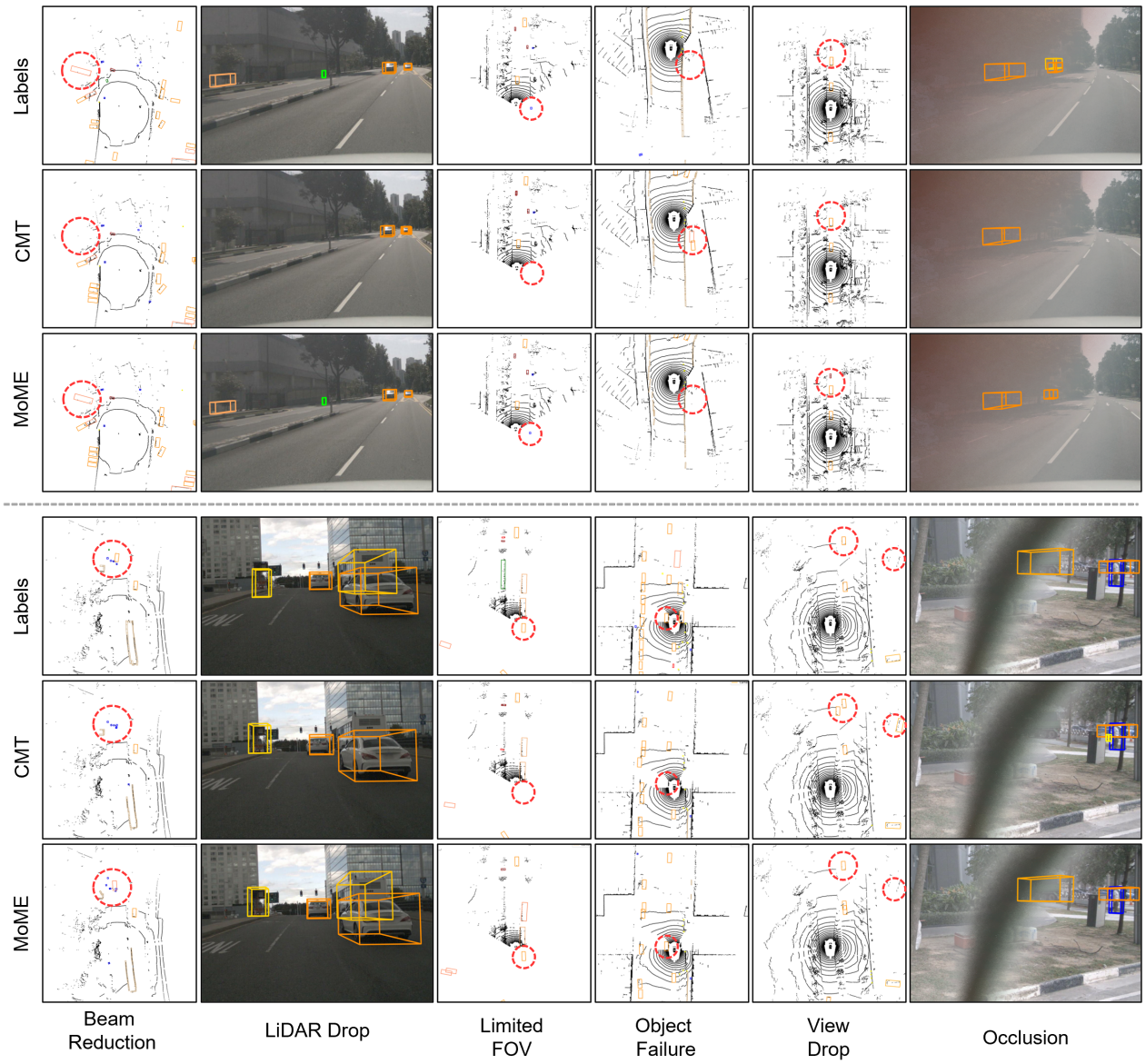

Figure 5. **Qualitative results under various sensor failure scenarios.** Comparison of detection results between MoME and CMT [29] under six sensor failure scenarios: Beam Reduction, LiDAR Drop, Limited FOV, Object Failure, View Drop, and Occlusion. The results demonstrate MoME's detection capabilities across these challenging conditions.

Algorithm 1 Adaptive Query Router (AQR)

```
import torch
import torch.nn as nn
from mmcv.cnn.bricks.transformer import build_transformer_layer_sequence

class AQR(nn.Module):
    def __init__(self, encoder, hidden_dim, **kwargs):
        super().__init__()
        self.encoder = build_transformer_layer_sequence(encoder)
        self.linear = nn.Linear(hidden_dim, 3)

    def forward(self, c_dict, ref_pts, pc_range, img_feats, metas):
        # 3D -> 2D projection.
        rp = torch.stack([ref_pts[..., i:i+1] * (pc_range[i+3] - pc_range[i]) + pc_range[i]
                          for i in range(3)], -1)
        b, n = rp.shape[:2]
        p2d = torch.einsum('bni,bvij->bvnj',
                     torch.cat([rp, torch.ones((b, n, 1)).cuda()], -1),
                     torch.tensor(np.stack([np.stack(i['lidar2img']) for i in metas]))
                     .float().cuda().transpose(2,3))
        p2d[..., :2] /= torch.clip(p2d[..., 2:3], min=1e-5)

        # Get valid points & prepare inputs.
        h, w = metas[0]['img_shape'][0][:2]
        v = ((p2d[..., 0] < w) & (p2d[..., 0] >= 0) & (p2d[..., 1] < h) & (p2d[..., 1] >= 0))
        v = (torch.cat([torch.zeros_like(v[:,:1,:], dtype=torch.bool), v], 1)
           .float().argmax(1) - 1) * (v != 0)

        m = v != -1
        p_cam = torch.zeros((b, n, 3), device=p2d.device)
        p_cam[m] = torch.cat([v[m].unsqueeze(-1),
                p2d[torch.where(m)[0], v[m], torch.where(m)[1]][...,[1,0]] *
                 (img_feats.shape[2] / h)], -1)
        p_lidar = torch.floor((rp[..., :2] + 54.0) * (180 / 108))[..., [1,0]]
        attention_mask = [RAM(p_lidar, p_cam, b, n).unsqueeze(1)
                          .repeat(1, self.e_num_heads, 1, 1).flatten(0, 1)]
        # Forward.
        out = self.encoder(
            torch.zeros_like(c_dict['query_embed_l'][0]),
            c_dict['memory_l'][0],
            c_dict['memory_v_l'][0],
            c_dict['query_embed_l'][0],
            c_dict['pos_embed_l'][0],
            attention_mask)
        out = self.linear((out[-1] if out.shape[0] != 0 else out.squeeze(0))
                              .transpose(1, 0))
        celoss = nn.CrossEntropyLoss()
        loss = celoss(out.reshape(-1, 3), torch.tensor([i['modalmask'] for i in metas]).cuda()
        .unsqueeze(1).repeat(1, n, 1).reshape(-1, 3).float()) if self.training else None
        return out, loss
```

Algorithm 2 Local Attention Mask

```
import torch

def LAM(c_dict, p_lidar, p_cam, b, n):
    # LiDAR mask.
    rs, wl = 180, 5  # row stride, window size for lidar.
    l_idx = p_lidar[..., 0] * rs + p_lidar[..., 1]
    off = torch.arange(-(wl // 2), wl // 2 + 1, device=p_lidar.device)
    y, x = torch.meshgrid(off, off)
    l_win = (y * rs + x).reshape(-1)
    l_indices = l_idx.unsqueeze(-1) + l_win

    l_valid = (l_indices >= 0) & (l_indices < rs * rs) & \
              ((l_indices % rs - l_idx.unsqueeze(-1) % rs).abs() <= wl // 2)
    l_mask = torch.ones(b, n, rs * rs, dtype=torch.bool, device=p_lidar.device)

    # Camera mask.
    h, w, wc = 40, 100, 15  # height, width, window size for camera.
    c_idx = p_cam[..., 0] * h * w + p_cam[..., 1] * w + p_cam[..., 2]
    off = torch.arange(-(wc // 2), wc // 2 + 1, device=p_cam.device)
    c_win = (off.unsqueeze(1) * w + off.unsqueeze(0)).reshape(-1)
    c_indices = c_idx.unsqueeze(-1) + c_win

    qp = c_idx % (h * w)
    c_valid = ((c_indices % (h * w) // w - qp.unsqueeze(-1) // w).abs() <= wc // 2) & \
              ((c_indices % w - qp.unsqueeze(-1) % w).abs() <= wc // 2)
    c_mask = torch.ones(b, n, 6 * h * w, dtype=torch.bool, device=p_cam.device)
    c_indices = torch.clamp(c_indices, 0, 6 * h * w - 1)

    # Update masks with valid indices.
    bid = torch.arange(b, device=p_lidar.device).view(-1,1,1)
    qid = torch.arange(n, device=p_lidar.device).view(1,-1,1)
    l_mask[bid.expand_as(l_indices)[l_valid], qid.expand_as(l_indices)[l_valid], l_indices[l_valid]] = False
    c_mask[bid.expand_as(c_indices)[c_valid], qid.expand_as(c_indices)[c_valid], c_indices[c_valid]] = False

    return torch.cat([l_mask, c_mask], -1)
```