

# You Only Segment Once: Towards Real-Time Panoptic Segmentation

## Supplementary Material

### A. Qualitative Results for Panoptic Segmentation

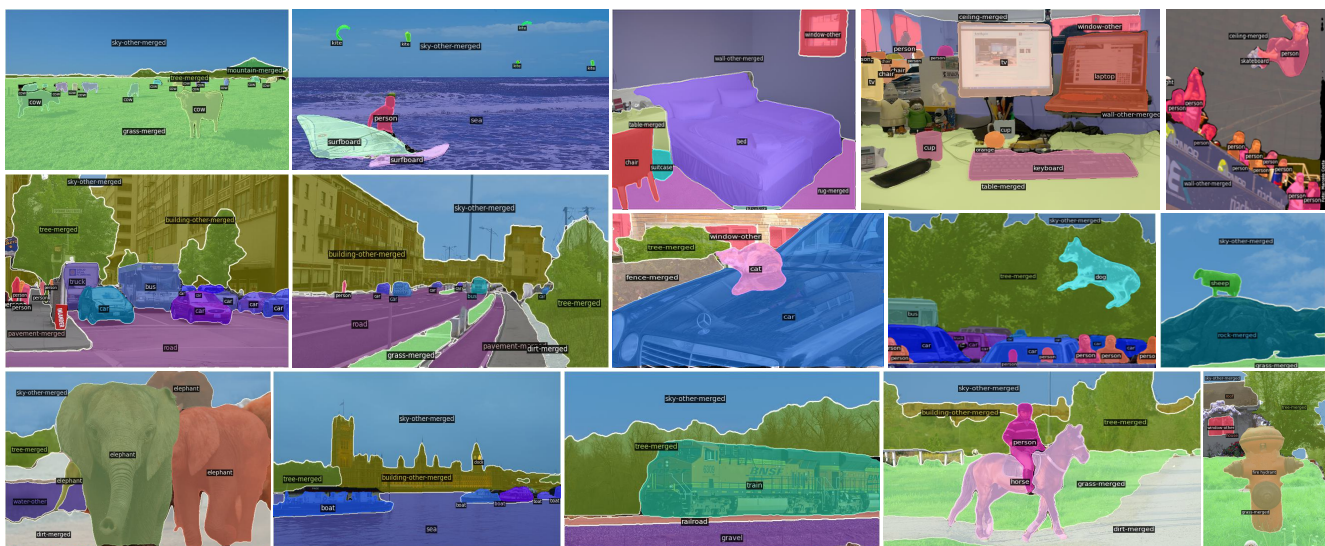


Figure 6. Panoptic segmentation on the COCO validation set.



Figure 7. Panoptic segmentation on the Cityscapes validation set.



Figure 8. Panoptic segmentation on the **ADE20K** validation set.



Figure 9. Panoptic segmentation on the **Mapillary Vistas** validation set.

## B. Quantitive Results on Real-Time Instance Segmentation

Solving the instance segmentation task efficiently is one of the keys to achieving real-time panoptic segmentation. Therefore, we study the performance of YOSO for real-time instance segmentation in Tab. 11. Note that the model is not specifically trained for instance segmentation. We show the results from the model trained on the COCO training set for panoptic segmentation. From the results on Tab. 11, we find that YOSO also achieves competitive performance for instance segmentation on the COCO validation set. When scaling the input images to 550, YOSO achieves 38.7 FPS and 34.7 mAP. The speed is only 5.9 FPS lower than the current state-of-the-art model, *i.e.*, SparseInst, while the mAP is 0.3 points higher. Specifically, the performance of YOSO on large objects, *i.e.*,  $AP^l$ , is better than all the state-of-the-art models. For example, when scaling the input images to 448, YOSO still can achieve 57.6  $AP^l$  for large objects, which is approximately 2.2 points higher than the performance of SOLOv2. The result suggests that YOSO is good at segmenting large objects.



Method	Backbone	Scale	AP	AP <sub>50</sub>	AP <sub>75</sub>	AP <sup>s</sup>	AP <sup>m</sup>	AP <sup>l</sup>	GPU	FPS
YOLACT [1]	DarkNet-53	550	28.9	46.9	30.3	9.8	30.9	47.3	2080Ti	45.9
YOLACT++ [2]	ResNet-50	550	33.7	52.7	35.5	11.9	36.6	54.6	2080Ti	40.8
BlendMask [3]	ResNet-50	550	34.5	54.7	36.5	14.4	37.7	52.1	2080Ti	35.6
SOLOv2 [6]	ResNet-50	448	33.7	53.3	35.6	11.3	36.9	55.4	2080Ti	39.6
OrienMask [5]	DarkNet-53	544	34.5	56.0	35.8	16.8	38.5	49.1	2080Ti	41.9
SparseInst [4]	ResNet-50	608	34.4	55.2	36.1	14.2	36.8	51.9	2080Ti	44.6
<b>YOSO, ours</b>	ResNet-50	448	33.0	52.8	34.6	11.3	35.4	57.6	2080Ti	46.1
<b>YOSO, ours</b>	ResNet-50	550	34.7	55.0	36.4	13.2	37.6	58.6	2080Ti	38.7
<b>YOSO, ours</b>	ResNet-50	608	35.6	56.3	37.5	14.4	39.0	59.2	2080Ti	33.5

Table 11. Real-time instance segmentation results on the **COCO validation** set.

## C. Codes

In this section, we provide the codes for the proposed components in YOSO as well as for the computation of FLOPs and GPU latency. Specifically, in Code. 1, we provide the implementation for IFA and CFA, which demonstrates: 1) the outputs of IFA and CFA are exactly the same; 2) the number of FLOPs for CFA is less than that for IFA; 3) CFA runs faster than CFA in terms of GPU latency. Similarly, in Code. 2, we provide the implementation for DCA, DSCA, DDCA, and DPCA. As the FLOPs counter does not compute the FLOPs for ‘nn.functional.conv1d’, we add the number of this operation in L171-172 and L178-181.

Listing 1. Pytorch code for computing FLOPs and GPU latency of different aggregation blocks.

```

1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4 from fvcore.nn import FlopCountAnalysis
5
6
7 class IFAModule(nn.Module):
8     def __init__(self, in_channels=128, out_channels=[1024, 512, 256, 128]):
9         super().__init__()
10        self.weights = nn.Parameter(torch.rand((in_channels, sum(out_channels), 1, 1)))
11
12    def forward(self, p5, p4, p3, p2):
13        x5 = F.interpolate(p5, scale_factor=8, align_corners=False, mode='bilinear')
14        x4 = F.interpolate(p4, scale_factor=4, align_corners=False, mode='bilinear')
15        x3 = F.interpolate(p3, scale_factor=2, align_corners=False, mode='bilinear')
16        x2 = p2
17        x_fuse = torch.concat([x5, x4, x3, x2], dim=1)
18        # x_fuse: [1, 1920, 256, 256]
19        output = F.conv2d(x_fuse, self.weights)
20        return output
21
22
23 class CFAModule(nn.Module):
24     def __init__(self, in_channels=128, out_channels=[1024, 512, 256, 128]):
25         super().__init__()
26        self.weights = nn.Parameter(torch.rand((in_channels, sum(out_channels), 1, 1)))
27        self.out_channels = out_channels
28
29    def forward(self, p5, p4, p3, p2):
30        x5 = F.interpolate(F.conv2d(p5, self.weights[:, :self.out_channels[0]]),
31                           scale_factor=8, align_corners=False, mode='bilinear')
32        x4 = F.interpolate(F.conv2d(p4, self.weights[:,
33                                     sum(self.out_channels[:1]):sum(self.out_channels[:2])]),
34                           scale_factor=4,
35                           align_corners=False, mode='bilinear')

```

```

32     x3 = F.interpolate(F.conv2d(p3, self.weights[:,
33         sum(self.out_channels[:2]):sum(self.out_channels[:3])]), scale_factor=2,
34         align_corners=False, mode='bilinear')
35     x2 = F.conv2d(p2, self.weights[:, sum(self.out_channels[:3]):])
36     output = x5 + x4 + x3 + x2
37     return output
38
39 class CFAModuleForFlops(nn.Module):
40     def __init__(self, in_channels=128, out_channels=[1024, 512, 256, 128]):
41         super().__init__()
42         self.weights = nn.Parameter(torch.rand((in_channels, sum(out_channels), 1, 1)))
43         self.conv = nn.Conv3d(4, 1, 1, bias=False)
44         nn.init.constant_(self.conv.weight, 1)
45         self.out_channels = out_channels
46
47     def forward(self, p5, p4, p3, p2):
48         x5 = F.interpolate(F.conv2d(p5, self.weights[:, :self.out_channels[0]]),
49             scale_factor=8, align_corners=False, mode='bilinear')
50         x4 = F.interpolate(F.conv2d(p4, self.weights[:,
51             sum(self.out_channels[:1]):sum(self.out_channels[:2])]), scale_factor=4,
52             align_corners=False, mode='bilinear')
53         x3 = F.interpolate(F.conv2d(p3, self.weights[:,
54             sum(self.out_channels[:2]):sum(self.out_channels[:3])]), scale_factor=2,
55             align_corners=False, mode='bilinear')
56         x2 = F.conv2d(p2, self.weights[:, sum(self.out_channels[:3]):])
57         concat = torch.concat([x5.unsqueeze(0), x4.unsqueeze(0), x3.unsqueeze(0),
58             x2.unsqueeze(0)], dim=0)
59         concat = concat.permute(1, 0, 2, 3, 4)
60
61         # To compute the number of FLOPs for accumulation operation
62         output = self.conv(concat)
63         # output = x5 + x4 + x3 + x2
64         # output = torch.add([x5, x4, x3, x2])
65         return output
66
67 # ----- FLOPs -----
68 inputs = (
69     torch.rand((1, 1024, 32, 32)),
70     torch.rand((1, 512, 64, 64)),
71     torch.rand((1, 256, 128, 128)),
72     torch.rand((1, 128, 256, 256)),
73 )
74
75 IFA = IFAModule()
76 CFA = CFAModuleForFlops()
77
78 print()
79 flops = FlopCountAnalysis(IFA, inputs)
80 print("IFA flops counter: ")
81 print(flops.total())
82 print(flops.by_operator())
83
84 print()
85 flops = FlopCountAnalysis(CFA, inputs)
86 print("CFA flops counter: ")
87 print(flops.total())

```



```

82 print(flops.by_operator())
83
84 # ----- GPU Latency -----
85 inputs = (
86     torch.rand((1, 1024, 32, 32)).cuda(),
87     torch.rand((1, 512, 64, 64)).cuda(),
88     torch.rand((1, 256, 128, 128)).cuda(),
89     torch.rand((1, 128, 256, 256)).cuda(),
90 )
91
92 IFA = IFAModule().cuda()
93 CFA = CFAModule().cuda()
94 IFA.weights.data = CFA.weights.data
95
96 # warm up
97 ifa_output = IFA(inputs[0], inputs[1], inputs[2], inputs[3])
98 cfa_output = CFA(inputs[0], inputs[1], inputs[2], inputs[3])
99 print("check the same output: ", (cfa_output.sum() - ifa_output.sum()))
100
101 with torch.autograd.profiler.profile(enabled=True, use_cuda=True, record_shapes=False)
    as prof:
102     IFA(inputs[0], inputs[1], inputs[2], inputs[3])
103 # NOTE: some columns were removed for brevity
104 print(prof.key_averages().table(sort_by="self_cpu_time_total"))
105
106 with torch.autograd.profiler.profile(enabled=True, use_cuda=True, record_shapes=False)
    as prof:
107     CFA(inputs[0], inputs[1], inputs[2], inputs[3])
108 # NOTE: some columns were removed for brevity
109 print(prof.key_averages().table(sort_by="self_cpu_time_total"))

```

---

Listing 2. Pytorch code for computing FLOPs and GPU latency of different attention types.

```

1 import torch
2 from torch import nn
3 import torch.nn.functional as F
4 from timm.models.layers import trunc_normal_
5
6 from fvcore.nn import FlopCountAnalysis
7 from fvcore.nn import ActivationCountAnalysis
8 from ptflops import get_model_complexity_info
9
10 HIDDEN_DIM = 256
11 NUM_PROPOSALS = 100
12 CONV_KERNEL_SIZE_1D = 3
13
14
15 class MultiHeadCrossAtten(nn.Module):
16     def __init__(self):
17         super(MultiHeadCrossAtten, self).__init__()
18         self.hidden_dim = HIDDEN_DIM
19         self.num_proposals = NUM_PROPOSALS
20         self.conv_kernel_size_1d = CONV_KERNEL_SIZE_1D
21
22         self.atten = nn.MultiheadAttention(embed_dim=self.hidden_dim * 1**2, num_heads=8,
            dropout=0.0)
23         self.f_norm = nn.LayerNorm(self.hidden_dim)

```

```

24
25 def forward(self, query, value):
26     query = query.permute(1, 0, 2)
27     value = value.permute(1, 0, 2)
28
29     out = self.attn(query, value, value)[0]
30     out = out.permute(1, 0, 2)
31     out = self.f_norm(out)
32     return out
33
34
35 class DyConvAtten(nn.Module):
36     def __init__(self):
37         super(DyConvAtten, self).__init__()
38         self.hidden_dim = HIDDEN_DIM
39         self.num_proposals = NUM_PROPOSALS
40         self.conv_kernel_size_1d = CONV_KERNEL_SIZE_1D
41
42         self.f_linear = nn.Linear(self.hidden_dim, self.num_proposals *
43             self.conv_kernel_size_1d)
44         self.f_norm = nn.LayerNorm(self.hidden_dim)
45
46     def forward(self, f, k):
47         # f: [B, N, C]
48         # k: [B, N, C * K * K]
49         B = f.shape[0]
50         weight = self.f_linear(f)
51         weight = weight.view(B, self.num_proposals, self.num_proposals,
52             self.conv_kernel_size_1d)
53         res = []
54         for i in range(B):
55             # input: [1, N, C * K * K]
56             # weight: [N, N, convK]
57             # output: [1, N, C * K * K]
58             out = F.conv1d(input=k.unsqueeze(1)[i], weight=weight[i], padding='same')
59             res.append(out)
60         # [B, N, C * K * K]
61         f_tmp = torch.cat(res, dim=0)
62         f_tmp = self.f_norm(f_tmp)
63         return f_tmp
64
65 class DySepConvAtten(nn.Module):
66     def __init__(self):
67         super(DySepConvAtten, self).__init__()
68         self.hidden_dim = HIDDEN_DIM
69         self.num_proposals = NUM_PROPOSALS
70         self.kernel_size = CONV_KERNEL_SIZE_1D
71
72         self.weight_linear = nn.Linear(self.hidden_dim, self.num_proposals +
73             self.kernel_size)
74         self.norm = nn.LayerNorm(self.hidden_dim)
75
76     def forward(self, query, value):
77         assert query.shape == value.shape
78         B, N, C = query.shape
79         dy_conv_weight = self.weight_linear(query)
80         res = []

```

```

79     value = value.unsqueeze(1)
80     for i in range(B):
81         # input: [1, N, C]
82         # weight: [N, 1, K]
83         # output: [1, N, C]
84         out = F.relu(F.conv1d(input=value[i], weight=dy_conv_weight[i, :,
            :self.kernel_size].view(self.num_proposals,1,self.kernel_size), groups=N,
            padding="same"))
85         # input: [1, N, C]
86         # weight: [N, N, 1]
87         # output: [1, N, C]
88         out = F.conv1d(input=out, weight=dy_conv_weight[i, :,
            self.kernel_size:].view(self.num_proposals,self.num_proposals,1),
            padding='same')
89
90         res.append(out)
91     point_out = torch.cat(res, dim=0)
92     point_out = self.norm(point_out)
93     return point_out
94
95
96 class DyDepthwiseConvAtten(nn.Module):
97     def __init__(self):
98         super(DyDepthwiseConvAtten, self).__init__()
99         self.hidden_dim = HIDDEN_DIM
100        self.num_proposals = NUM_PROPOSALS
101        self.kernel_size = CONV_KERNEL_SIZE_1D
102
103        self.weight_linear = nn.Linear(self.hidden_dim, self.kernel_size)
104        self.norm = nn.LayerNorm(self.hidden_dim)
105
106    def forward(self, query, value):
107        assert query.shape == value.shape
108        B, N, C = query.shape
109        dy_conv_weight =
            self.weight_linear(query).view(B,self.num_proposals,1,self.kernel_size)
110
111        res = []
112        value = value.unsqueeze(1)
113        for i in range(B):
114            # input: [1, N, C]
115            # weight: [N, 1, K]
116            # output: [1, N, C]
117            out = F.conv1d(input=value[i], weight=dy_conv_weight[i], groups=N,
                padding="same")
118            res.append(out)
119        point_out = torch.cat(res, dim=0)
120        point_out = self.norm(point_out)
121        return point_out
122
123
124 class DyPointwiseConvAtten(nn.Module):
125     def __init__(self):
126         super(DyPointwiseConvAtten, self).__init__()
127         self.hidden_dim = HIDDEN_DIM
128         self.num_proposals = NUM_PROPOSALS
129         self.kernel_size = CONV_KERNEL_SIZE_1D
130

```



```

131     self.weight_linear = nn.Linear(self.hidden_dim, self.num_proposals)
132     self.norm = nn.LayerNorm(self.hidden_dim)
133
134     def forward(self, query, value):
135         assert query.shape == value.shape
136         B, N, C = query.shape
137
138         dy_conv_weight =
139             self.weight_linear(query).view(B, self.num_proposals, self.num_proposals, 1)
140
141         res = []
142         value = value.unsqueeze(1)
143         for i in range(B):
144             # input: [1, N, C]
145             # weight: [N, N, 1]
146             # output: [1, N, C]
147             out = F.conv1d(input=value[i], weight=dy_conv_weight[i], padding='same')
148
149             res.append(out)
150         point_out = torch.cat(res, dim=0)
151         point_out = self.norm(point_out)
152         return point_out
153
154     # ----- FLOPs -----
155     q = torch.rand((1, NUM_PROPOSALS, HIDDEN_DIM))
156     v = torch.rand((1, NUM_PROPOSALS, HIDDEN_DIM))
157
158     MHCA = MultiHeadCrossAtten()
159     DCA = DyConvAtten()
160     DSCA = DySepConvAtten()
161     DDCA = DyDepthwiseConvAtten()
162     DPCA = DyPointwiseConvAtten()
163
164     flops = FlopCountAnalysis(MHCA, (q, v))
165     print("MHCA flops counter: ")
166     print(flops.total())
167     print(flops.by_operator())
168
169     flops = FlopCountAnalysis(DCA, (q, v))
170     print("DCA flops counter: ")
171     conv = nn.Conv1d(in_channels=NUM_PROPOSALS, out_channels=NUM_PROPOSALS,
172                     kernel_size=CONV_KERNEL_SIZE_1D, bias=False, padding='same')
173     macs, _ = get_model_complexity_info(conv, (NUM_PROPOSALS, HIDDEN_DIM),
174                                       as_strings=False, print_per_layer_stat=False, verbose=True)
175     print(flops.total() + macs)
176     print(flops.by_operator(), "conv: ", macs)
177
178     flops = FlopCountAnalysis(DSCA, (q, v))
179     print("DSCA flops counter: ")
180     depthwise_conv = nn.Conv1d(in_channels=NUM_PROPOSALS, out_channels=NUM_PROPOSALS,
181                                kernel_size=CONV_KERNEL_SIZE_1D, bias=False, groups=NUM_PROPOSALS, padding='same')
182     macs_depthwise, _ = get_model_complexity_info(depthwise_conv, (NUM_PROPOSALS,
183                                                                    HIDDEN_DIM), as_strings=False, print_per_layer_stat=False, verbose=True)
184     pointwise_conv = nn.Conv1d(in_channels=NUM_PROPOSALS, out_channels=NUM_PROPOSALS,
185                                 kernel_size=1, bias=False, padding='same')
186     macs_pointwise, _ = get_model_complexity_info(pointwise_conv, (NUM_PROPOSALS,
187                                                                    HIDDEN_DIM), as_strings=False, print_per_layer_stat=False, verbose=True)

```

```

182 print(flops.total() + macs_depthwise + macs_pointwise)
183 print(flops.by_operator(), "depthwise: ", macs_depthwise, "pointwise: ", macs_pointwise)
184
185 flops = FlopCountAnalysis(DDCA, (q, v))
186 print("DDCA flops counter: ")
187 print(flops.total() + macs_depthwise)
188 print(flops.by_operator(), "depthwise: ", macs_depthwise,)
189
190 flops = FlopCountAnalysis(DPCA, (q, v))
191 print("DPCA flops counter: ")
192 print(flops.total() + macs_pointwise)
193 print(flops.by_operator(), "pointwise: ", macs_pointwise)
194
195
196 # ----- GPU Latency -----
197 q = torch.rand((1, NUM_PROPOSALS, HIDDEN_DIM), requires_grad=False).cuda()
198 v = torch.rand((1, NUM_PROPOSALS, HIDDEN_DIM), requires_grad=False).cuda()
199
200 MHCA = MultiHeadCrossAtten().cuda()
201 DCA = DyConvAtten().cuda()
202 DSCA = DySepConvAtten().cuda()
203 DDCA = DyDepthwiseConvAtten().cuda()
204 DPCA = DyPointwiseConvAtten().cuda()
205
206 # warm up
207 o = MHCA(q, v)
208 o = DCA(q, v)
209 o = DSCA(q, v)
210 o = DDCA(q, v)
211 o = DPCA(q, v)
212
213 with torch.autograd.profiler.profile(enabled=True, use_cuda=True, record_shapes=False)
    as prof:
214     o = MHCA(q, v)
215 # NOTE: some columns were removed for brevity
216 print(prof.key_averages().table(sort_by="self_cpu_time_total"))
217
218 with torch.autograd.profiler.profile(enabled=True, use_cuda=True, record_shapes=False)
    as prof:
219     o = DCA(q, v)
220 # NOTE: some columns were removed for brevity
221 print(prof.key_averages().table(sort_by="self_cpu_time_total"))
222
223 with torch.autograd.profiler.profile(enabled=True, use_cuda=True, record_shapes=False)
    as prof:
224     o = DSCA(q, v)
225 # NOTE: some columns were removed for brevity
226 print(prof.key_averages().table(sort_by="self_cpu_time_total"))
227
228 with torch.autograd.profiler.profile(enabled=True, use_cuda=True, record_shapes=False)
    as prof:
229     o = DDCA(q, v)
230 # NOTE: some columns were removed for brevity
231 print(prof.key_averages().table(sort_by="self_cpu_time_total"))
232
233 with torch.autograd.profiler.profile(enabled=True, use_cuda=True, record_shapes=False)
    as prof:
234     o = DPCA(q, v)

```

```
235 # NOTE: some columns were removed for brevity
236 print (prof.key_averages().table(sort_by="self_cpu_time_total"))
```

---

## References

- [1] Daniel Bolya, Chong Zhou, Fanyi Xiao, and Yong Jae Lee. Yolact: Real-time instance segmentation. In *Proceedings of the IEEE International Conference on Computer Vision*, 2019. 3
- [2] Daniel Bolya, Chong Zhou, Fanyi Xiao, and Yong Jae Lee. Yolact++: Better real-time instance segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020. 3
- [3] Hao Chen, Kunyang Sun, Zhi Tian, Chunhua Shen, Yongming Huang, and Youliang Yan. Blendmask: Top-down meets bottom-up for instance segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2020. 3
- [4] Tianheng Cheng, Xinggang Wang, Shaoyu Chen, Wenqiang Zhang, Qian Zhang, Chang Huang, Zhaoxiang Zhang, and Wenyu Liu. Sparse instance activation for real-time instance segmentation. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2022. 3
- [5] Wentao Du, Zhiyu Xiang, Shuya Chen, Chengyu Qiao, Yiman Chen, and Tingming Bai. Real-time instance segmentation with discriminative orientation maps. In *Proceedings of the IEEE International Conference on Computer Vision*, 2021. 3
- [6] Xinlong Wang, Rufeng Zhang, Tao Kong, Lei Li, and Chunhua Shen. Solov2: Dynamic and fast instance segmentation. *Advances in Neural Information Processing Systems*, 2020. 3