Supplementary Material

Figure 1. Visual comparison of $2 \times$ super-resolution results by QuickSRNet and existing solutions on Urban100 images.

Figure 2. Visual comparison of $3 \times$ super-resolution results by QuickSRNet and existing solutions on Urban100 images.

Figure 3. Visual comparison of $4 \times$ super-resolution results by QuickSRNet and existing solutions on DIV2K images.
Figure 4. More examples of visual artifacts by ABPN vs QuickSRNet-Medium (4×) on Urban 100 images.

Figure 5. SISR (2×) for Gaming: (a) Low-resolution, (b) Bicubic interpolation, (c) FSR1.0, and (d) QuickSRNet-Small (ours).
<table>
<thead>
<tr>
<th>Scaling Factor</th>
<th>QuickSRNet Specification</th>
<th>Set5 PSNR / SSIM</th>
<th>Set14 PSNR / SSIM</th>
<th>BSD100 PSNR / SSIM</th>
<th>Urban100 PSNR / SSIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>2×</td>
<td>f32 - m1</td>
<td>36.83 / 0.9563</td>
<td>32.35 / 0.9085</td>
<td>31.43 / 0.8900</td>
<td>29.66 / 0.8999</td>
</tr>
<tr>
<td></td>
<td>f32 - m2 (small)</td>
<td>37.12 / 0.9575</td>
<td>32.57 / 0.9107</td>
<td>31.72 / 0.8942</td>
<td>30.15 / 0.9067</td>
</tr>
<tr>
<td></td>
<td>f32 - m3</td>
<td>37.30 / 0.9583</td>
<td>32.72 / 0.9117</td>
<td>31.82 / 0.8955</td>
<td>30.43 / 0.9104</td>
</tr>
<tr>
<td></td>
<td>f32 - m5 (medium)</td>
<td>37.39 / 0.9586</td>
<td>32.82 / 0.9130</td>
<td>31.98 / 0.8964</td>
<td>30.75 / 0.9142</td>
</tr>
<tr>
<td></td>
<td>f32 - m7</td>
<td>37.51 / 0.9593</td>
<td>32.95 / 0.9136</td>
<td>31.95 / 0.8973</td>
<td>30.93 / 0.9164</td>
</tr>
<tr>
<td></td>
<td>f32 - m11 (large)</td>
<td>37.87 / 0.9603</td>
<td>33.29 / 0.9166</td>
<td>32.12 / 0.8992</td>
<td>31.14 / 0.9186</td>
</tr>
<tr>
<td>3×</td>
<td>f32 - m1</td>
<td>32.75 / 0.9112</td>
<td>29.08 / 0.8234</td>
<td>28.41 / 0.7880</td>
<td>24.47 / 0.8026</td>
</tr>
<tr>
<td></td>
<td>f32 - m2 (small)</td>
<td>33.10 / 0.9157</td>
<td>29.29 / 0.8258</td>
<td>27.07 / 0.7699</td>
<td>24.74 / 0.8128</td>
</tr>
<tr>
<td></td>
<td>f32 - m3</td>
<td>33.33 / 0.9180</td>
<td>29.39 / 0.8298</td>
<td>27.16 / 0.7719</td>
<td>24.90 / 0.8182</td>
</tr>
<tr>
<td></td>
<td>f32 - m5 (medium)</td>
<td>33.58 / 0.9206</td>
<td>29.49 / 0.8327</td>
<td>27.24 / 0.7791</td>
<td>25.08 / 0.8266</td>
</tr>
<tr>
<td></td>
<td>f32 - m7</td>
<td>33.69 / 0.9216</td>
<td>29.60 / 0.8335</td>
<td>27.30 / 0.7896</td>
<td>25.22 / 0.8303</td>
</tr>
<tr>
<td></td>
<td>f32 - m11 (large)</td>
<td>34.14 / 0.9258</td>
<td>29.88 / 0.8397</td>
<td>29.02 / 0.8038</td>
<td>27.81 / 0.8459</td>
</tr>
<tr>
<td>4×</td>
<td>f32 - m1</td>
<td>30.48 / 0.8659</td>
<td>27.31 / 0.7559</td>
<td>26.94 / 0.7147</td>
<td>24.47 / 0.7262</td>
</tr>
<tr>
<td></td>
<td>f32 - m2 (small)</td>
<td>30.84 / 0.8741</td>
<td>27.55 / 0.7635</td>
<td>27.07 / 0.7199</td>
<td>24.74 / 0.7382</td>
</tr>
<tr>
<td></td>
<td>f32 - m3</td>
<td>31.04 / 0.8773</td>
<td>27.65 / 0.7656</td>
<td>27.16 / 0.7226</td>
<td>24.90 / 0.7447</td>
</tr>
<tr>
<td></td>
<td>f32 - m5 (medium)</td>
<td>31.27 / 0.8821</td>
<td>27.79 / 0.7699</td>
<td>27.24 / 0.7253</td>
<td>25.08 / 0.7517</td>
</tr>
<tr>
<td></td>
<td>f32 - m7</td>
<td>31.39 / 0.8838</td>
<td>27.93 / 0.7709</td>
<td>27.30 / 0.7275</td>
<td>25.22 / 0.7573</td>
</tr>
<tr>
<td></td>
<td>f32 - m11 (large)</td>
<td>31.50 / 0.8856</td>
<td>28.15 / 0.7729</td>
<td>27.35 / 0.7289</td>
<td>25.32 / 0.7619</td>
</tr>
<tr>
<td></td>
<td>f32 - m11 (large)</td>
<td>31.77 / 0.8908</td>
<td>28.15 / 0.7797</td>
<td>27.50 / 0.7344</td>
<td>25.74 / 0.7761</td>
</tr>
</tbody>
</table>

Table 1. QuickSRNet PSNRs (dB) evaluated for different scaling factors (2×, 3×, and 4×) on benchmark SISR datasets before and after quantization.

<table>
<thead>
<tr>
<th>Scaling Factor</th>
<th>QuickSRNet Specification</th>
<th>Set5 PSNR / SSIM</th>
<th>Set14 PSNR / SSIM</th>
<th>BSD100 PSNR / SSIM</th>
<th>Urban100 PSNR / SSIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>2×</td>
<td>f32 - m1</td>
<td>36.83 / 0.9563</td>
<td>32.35 / 0.9085</td>
<td>31.43 / 0.8900</td>
<td>29.66 / 0.8999</td>
</tr>
<tr>
<td></td>
<td>f32 - m2 (small)</td>
<td>37.12 / 0.9575</td>
<td>32.57 / 0.9107</td>
<td>31.72 / 0.8942</td>
<td>30.15 / 0.9067</td>
</tr>
<tr>
<td></td>
<td>f32 - m3</td>
<td>37.30 / 0.9583</td>
<td>32.72 / 0.9117</td>
<td>31.82 / 0.8955</td>
<td>30.43 / 0.9104</td>
</tr>
<tr>
<td></td>
<td>f32 - m5 (medium)</td>
<td>37.39 / 0.9586</td>
<td>32.82 / 0.9130</td>
<td>31.98 / 0.8964</td>
<td>30.75 / 0.9142</td>
</tr>
<tr>
<td></td>
<td>f32 - m7</td>
<td>37.51 / 0.9593</td>
<td>32.95 / 0.9136</td>
<td>31.95 / 0.8973</td>
<td>30.93 / 0.9164</td>
</tr>
<tr>
<td></td>
<td>f32 - m11 (large)</td>
<td>37.87 / 0.9603</td>
<td>33.29 / 0.9166</td>
<td>32.12 / 0.8992</td>
<td>31.14 / 0.9186</td>
</tr>
<tr>
<td>3×</td>
<td>f32 - m1</td>
<td>30.48 / 0.8659</td>
<td>27.31 / 0.7559</td>
<td>26.94 / 0.7147</td>
<td>24.47 / 0.7262</td>
</tr>
<tr>
<td></td>
<td>f32 - m2 (small)</td>
<td>30.84 / 0.8741</td>
<td>27.55 / 0.7635</td>
<td>27.07 / 0.7199</td>
<td>24.74 / 0.7382</td>
</tr>
<tr>
<td></td>
<td>f32 - m3</td>
<td>31.04 / 0.8773</td>
<td>27.65 / 0.7656</td>
<td>27.16 / 0.7226</td>
<td>24.90 / 0.7447</td>
</tr>
<tr>
<td></td>
<td>f32 - m5 (medium)</td>
<td>31.27 / 0.8821</td>
<td>27.79 / 0.7699</td>
<td>27.24 / 0.7253</td>
<td>25.08 / 0.7517</td>
</tr>
<tr>
<td></td>
<td>f32 - m7</td>
<td>31.39 / 0.8838</td>
<td>27.93 / 0.7709</td>
<td>27.30 / 0.7275</td>
<td>25.22 / 0.7573</td>
</tr>
<tr>
<td></td>
<td>f32 - m11 (large)</td>
<td>31.50 / 0.8856</td>
<td>28.15 / 0.7729</td>
<td>27.35 / 0.7289</td>
<td>25.32 / 0.7619</td>
</tr>
<tr>
<td></td>
<td>f32 - m11 (large)</td>
<td>31.77 / 0.8908</td>
<td>28.15 / 0.7797</td>
<td>27.50 / 0.7344</td>
<td>25.74 / 0.7761</td>
</tr>
</tbody>
</table>

Table 2. QuickSRNet PSNRs (dB) and SSIM numbers evaluated for different scaling factors (2×, 3×, and 4×) on benchmark SISR datasets before quantization.
Exporting QuickSRNet to ONNX for on-device profiling

Before running the model on device, we shuffle the weights of some of the convolutional layers, before depth-to-space and after space-to-depth (for 1.5x model) operations. This is necessary because the data layout of PyTorch’s depth-to-space operation (CRD) is not optimized on our target device (Hexagon Processor of a mobile device with Snapdragon 8 Gen 1). For better on-device performance, the data layout needs to be changed to DCR. The appropriate method of creating a QuickSRNet model instance with the shuffled weights (in DCR format) can be done with the following steps. Below are a bunch of prerequisites to accomplish this task:

- The PyTorch implementation of QuickSRNet can be found [here](#).
- Pre-trained weights (including AIMET-quantized weights and encodings) are available [here](#).
- A Jupyter Notebook that shows how to load and use QuickSRNet is also available [here](#).

**Step 1** Load the quantized QuickSRNet model from the checkpointed weights and encodings. With the PyTorch implementation of QuickSRNet, the model can be instantiated with the appropriately shuffled weights as follows:

```python
import torch

# Use one of QuickSRNetSmall, QuickSRNetMedium or QuickSRNetLarge with the desired scaling factor.
scaling_factor = 2
model = QuickSRNetSmall(scaling_factor=scaling_factor)

state_dict = torch.load(model_checkpoint_path, map_location='cpu')['state_dict']
model.load_state_dict(state_dict)
model.to(device)  # device is one of 'cuda' or 'cpu'

# Re-arrange the weights of the appropriate conv layer(s)
model.to_dcr()
```

**Step 2 (optional)** To use QuickSRNet quantized using AIMET, use the following steps:

```python
dummy_input_shape = (1, 3, 256, 256)  # Expected input shape for the model (1 x C x H x W)
dummy_input = torch.randn(dummy_input_shape)

sim = QuantizationSimModel(model=model,
dummy_input=dummy_input,
quant_scheme=QuantScheme.post_training_tf_enhanced,
default_output_bw=8,
default_param_bw=8)

sim.set_and_freeze_param_encodings(encoding_path=encoding_path)
sim.compute_encodings(forward_pass_callback=pass_calibration_data,
forward_pass_callback_args=(calibration_data,
scale_factor,
use_cuda))
```

**Step 3** Export the model to ONNX:

```python
import os
import torch
from aimet_torch.onnx_utils import OnnxExportApiArgs

filename = "<onnx_filename>"
output_dir = "<output_dir>"
model_save_path = "<output_dir>/<filename>.onnx"

# PixelUnshuffle does not map to space-to-depth without the code below
import torch.onnx.symbolic_helper as SymHelp
import torch.onnx.symbolic_opset11 as Opset11
```
from torch.onnx.symbolic_helper import parse_args, _unimplemented

@parse_args('v', 'i')
def pixel_unshuffle(g, self, downscale_factor):
    rank = sym_help._get_tensor_rank(self)
    if rank is not None and rank != 4:
        return _unimplemented("pixel_unshuffle", "only support 4d input")
    return g.op("SpaceToDepth", self, blocksize_i=downscale_factor)
opset11.pixel_unshuffle = pixel_unshuffle

# Set 'use_quantized' to 'True' if exporting the quantized model, else 'False'
if use_quantized:
    sim.export(output_dir, filename, dummy_input, onnx_export_args=OnnxExportApiArgs(opset_version=11))
else:
    torch.onnx.export(model, dummy_input, model_save_path, export_params=True, opset_version=11)

Step 4  Convert the ONNX space-to-depth and/or depth-to-space operations to DCR:

import onnx
from onnx.helper import make_attribute
def overwrite_onnx_d2s_mode_to_dcr(onnx_path):
    """Manual override of the depth-to-space mode to DCR."""
    onnx_model = onnx.load(onnx_path)
    graph = onnx_model.graph
    for node in graph.node:
        if node.op_type == 'DepthToSpace':
            depth_to_space_attribute = node.attribute
            found = False
            for idx, attr in enumerate(node.attribute):
                if attr.name == 'mode':
                    found = True
                    break
            if found:
                node.attribute.pop(idx)
            new_attr = make_attribute('s', 'DCR')
            new_attr.name = 'mode'
            depth_to_space_attribute.extend([new_attr])
    onnx.save(onnx_model, onnx_path)

onnx_path = "<output_dir>/<filename>.onnx"  # Path to the exported ONNX file
overwrite_onnx_d2s_mode_to_dcr(onnx_path)

Step 5  Re-order per-channel encodings for the quantized model to DCR:

import json
def reorder_per_channel_encodings_to_dcr(encodings_path, layer_names):
    """Used to re-arrange the per-channel encodings of the conv layer(s) preceding the final depth-to-space operation."

    This is necessary because the data layout of PyTorch's depth-to-space operation (CRD) is not optimized on device. For better on-device performance, the data layout needs to be changed to DCR.

    Note: in the case of per-layer quantization, this function does not do anything."""

```python
with open(encodings_path) as f:
    encodings = json.load(f)

new_encodings = encodings.copy()
to_shuffle = [key for layer_name in layer_names for key in encodings['param_encodings'] if layer_name in key]
for key in to_shuffle:
    per_channel_enc = encodings['param_encodings'][key]
    if len(per_channel_enc) > 1:
        scaling_factor = int((len(per_channel_enc) / 3) ** 0.5)
        new_encodings['param_encodings'][key] = [per_channel_enc[i + k * (scaling_factor ** 2)]
                                                for i in range(scaling_factor ** 2) for k in range(3)]
    else:
        # per-layer quantization: do nothing
        pass

with open(encodings_path, 'w') as f:
    json.dump(new_encodings, f, sort_keys=True, indent=4)

reorder_per_channel_encodings_to_dcr(encodings_path, ['anchor', 'conv_last'])
```