Supplementary Material

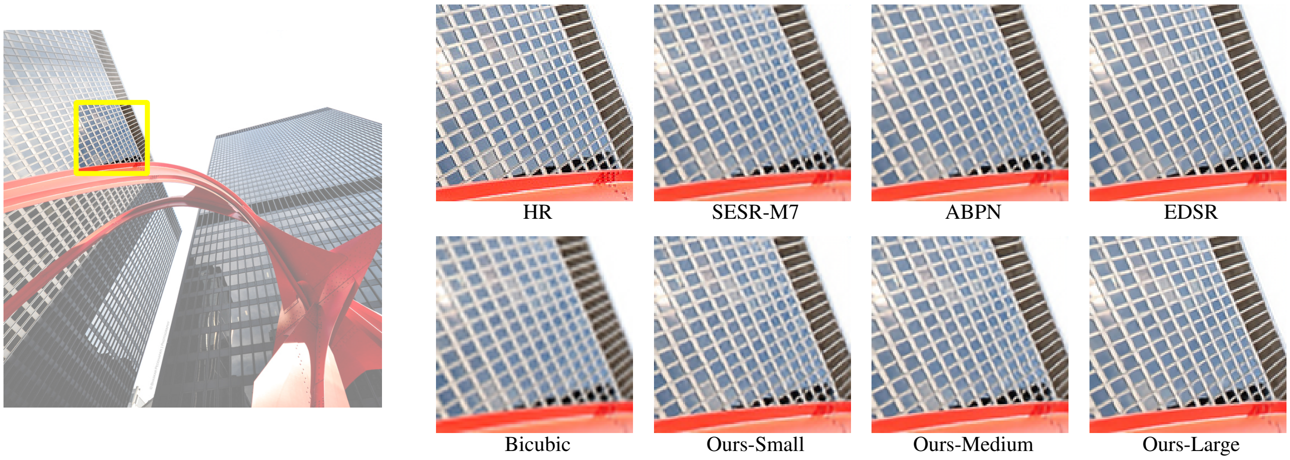


Figure 1. Visual comparison of $2\times$ super-resolution results by QuickSRNet and existing solutions on Urban100 images.

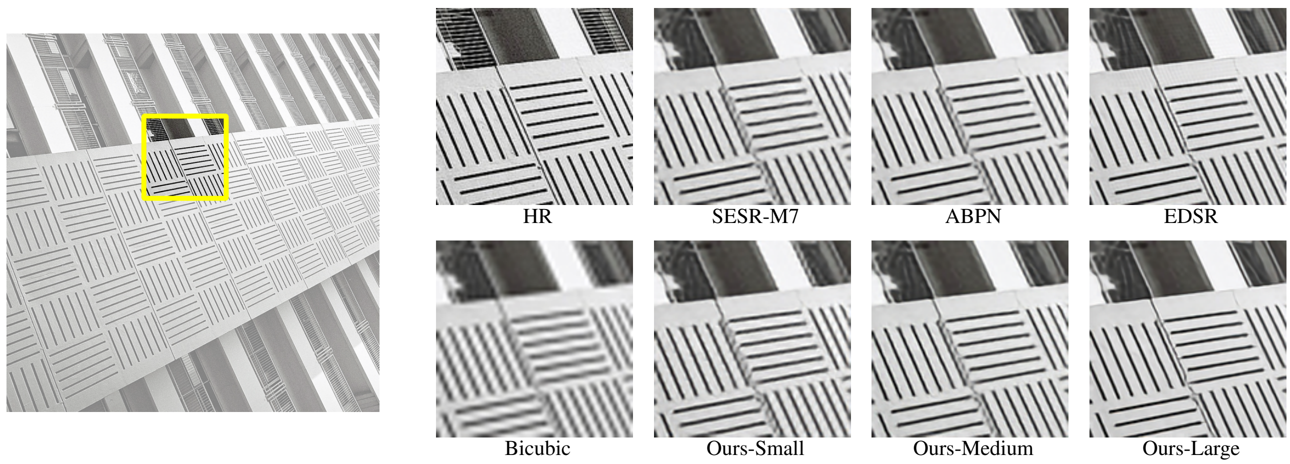


Figure 2. Visual comparison of $3\times$ super-resolution results by QuickSRNet and existing solutions on Urban100 images.

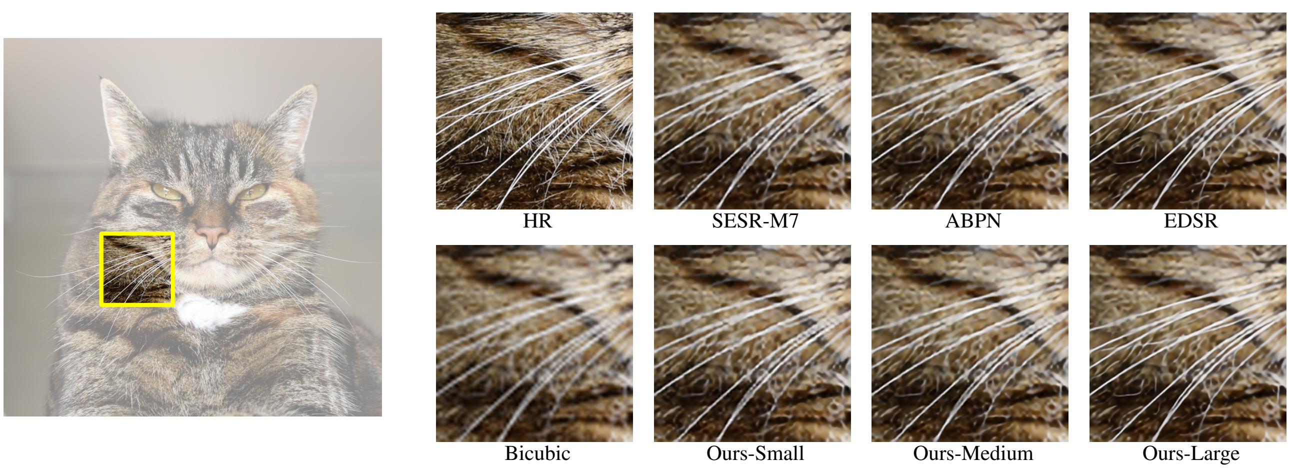
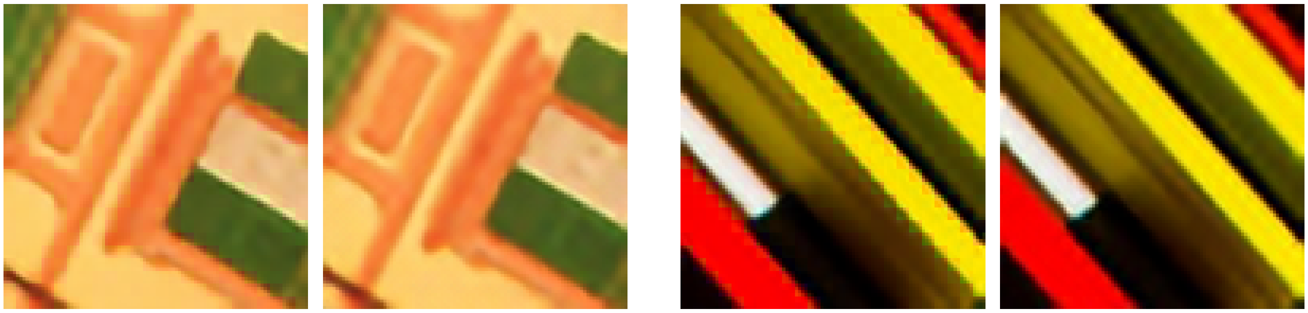
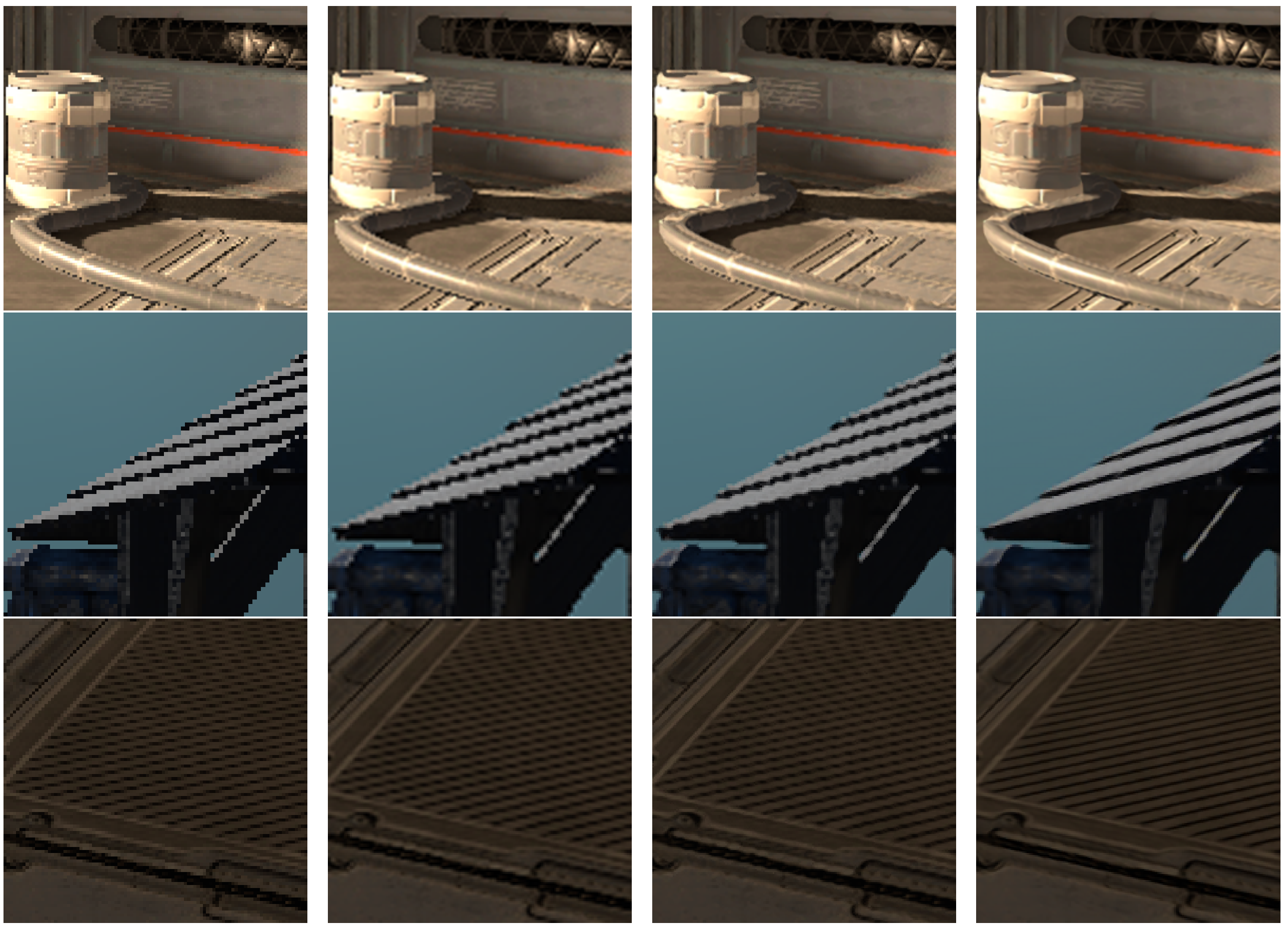


Figure 3. Visual comparison of $4\times$ super-resolution results by QuickSRNet and existing solutions on DIV2K images.



ABPN QuickSRNet-Medium ABPN QuickSRNet-Medium

Figure 4. More examples of visual artifacts by ABPN vs QuickSRNet-Medium ($4\times$) on Urban 100 images.



(a) LR (b) Bicubic (c) FSR1.0 (d) QuickSRNet-Small

Figure 5. SISR ($2\times$) for Gaming: (a) Low-resolution, (b) Bicubic interpolation, (c) FSR1.0, and (d) QuickSRNet-Small (ours).

Scaling Factor	QuickSRNet Specification	Set5		Set14		BSD100		Urban100	
		FP16	INT8	FP16	INT8	FP16	INT8	FP16	INT8
2×	*f32 - m1*	36.83	36.67	32.35	32.28	31.43	31.38	29.66	29.61
	f32 - m2 (small)	37.12	36.97	32.57	32.53	31.61	31.58	30.15	30.10
	f32 - m3	37.30	37.06	32.72	32.57	31.72	31.63	30.43	30.30
	f32 - m5 (medium)	37.39	37.22	32.82	32.75	31.82	31.77	30.75	30.66
	f32 - m7	37.51	37.27	32.95	32.84	31.88	31.81	30.93	30.84
	f32 - m11	37.59	37.19	33.00	32.86	31.95	31.80	31.14	30.91
	f64 - m11 (large)	37.87	37.61	33.29	33.18	32.12	32.04	31.74	31.64
3×	*f32 - m1*	32.75	32.69	29.08	29.05	28.41	28.38	26.19	26.16
	f32 - m2 (small)	33.10	33.03	29.29	29.25	28.57	28.55	26.53	26.51
	f32 - m3	33.33	33.25	29.39	29.35	28.67	28.63	26.77	26.72
	f32 - m5 (medium)	33.58	33.49	29.49	29.46	28.75	28.72	27.02	26.99
	f32 - m7	33.69	33.53	29.60	29.52	28.81	28.76	27.16	27.10
	f32 - m11	33.81	33.63	29.68	29.60	28.86	28.80	27.35	27.27
	f64 - m11 (large)	34.14	34.01	29.88	29.82	29.02	28.98	27.81	27.76
4×	*f32 - m1*	30.48	30.42	27.31	27.27	26.94	26.91	24.47	24.45
	f32 - m2 (small)	30.84	30.82	27.55	27.54	27.07	27.06	24.74	24.74
	f32 - m3	31.04	30.95	27.65	27.58	27.16	27.12	24.90	24.86
	f32 - m5 (medium)	31.27	31.21	27.79	27.76	27.24	27.21	25.08	25.06
	f32 - m7	31.39	31.29	27.83	27.80	27.30	27.27	25.22	25.18
	f32 - m11	31.50	31.36	27.93	27.85	27.35	27.29	25.32	25.27
	f64 - m11 (large)	31.77	31.73	28.15	28.12	27.50	27.48	25.74	25.72

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

Scaling Factor	QuickSRNet Specification	Set5 PSNR / SSIM	Set14 PSNR / SSIM	BSD100 PSNR / SSIM	Urban100 PSNR / SSIM
2×	*f32 - m1*	36.83 / 0.9563	32.35 / 0.9085	31.43 / 0.8900	29.66 / 0.8999
	f32 - m2 (small)	37.12 / 0.9575	32.57 / 0.9107	31.61 / 0.8925	30.15 / 0.9067
	f32 - m3	37.30 / 0.9583	32.72 / 0.9117	31.72 / 0.8942	30.43 / 0.9104
	f32 - m5 (medium)	37.39 / 0.9586	32.82 / 0.9130	31.82 / 0.8955	30.75 / 0.9142
	f32 - m7	37.51 / 0.9593	32.95 / 0.9136	31.88 / 0.8964	30.93 / 0.9164
	f32 - m11	37.59 / 0.9594	33.00 / 0.9142	31.95 / 0.8973	31.14 / 0.9186
	f64 - m11 (large)	37.87 / 0.9603	33.29 / 0.9166	32.12 / 0.8992	31.74 / 0.9248
3×	*f32 - m1*	32.75 / 0.9112	29.08 / 0.8234	28.41 / 0.7880	26.19 / 0.8026
	f32 - m2 (small)	33.10 / 0.9157	29.29 / 0.8282	28.57 / 0.7919	26.53 / 0.8128
	f32 - m3	33.33 / 0.9180	29.39 / 0.8298	28.67 / 0.7947	26.77 / 0.8195
	f32 - m5 (medium)	33.58 / 0.9206	29.49 / 0.8327	28.75 / 0.7971	27.02 / 0.8266
	f32 - m7	33.69 / 0.9216	29.60 / 0.8335	28.81 / 0.7986	27.16 / 0.8303
	f32 - m11	33.81 / 0.9226	29.68 / 0.8346	28.86 / 0.8000	27.35 / 0.8347
	f64 - m11 (large)	34.14 / 0.9258	29.88 / 0.8397	29.02 / 0.8038	27.81 / 0.8459
4×	*f32 - m1*	30.48 / 0.8659	27.31 / 0.7559	26.94 / 0.7147	24.47 / 0.7262
	f32 - m2 (small)	30.84 / 0.8741	27.55 / 0.7635	27.07 / 0.7196	24.74 / 0.7382
	f32 - m3	31.04 / 0.8773	27.65 / 0.7656	27.16 / 0.7226	24.90 / 0.7447
	f32 - m5 (medium)	31.27 / 0.8821	27.79 / 0.7699	27.24 / 0.7253	25.08 / 0.7517
	f32 - m7	31.39 / 0.8838	27.83 / 0.7709	27.30 / 0.7275	25.22 / 0.7573
	f32 - m11	31.50 / 0.8856	27.93 / 0.7729	27.35 / 0.7289	25.32 / 0.7619
	f64 - m11 (large)	31.77 / 0.8908	28.15 / 0.7797	27.50 / 0.7344	25.74 / 0.7761

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 $1.5\times$ 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:

```
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:

```
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,
                                                  scaling_factor,
                                                  use_cuda))
```

Step 3 Export the model to ONNX:

```
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 sym_help
import torch.onnx.symbolic_opset11 as opset11
```

```python
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:

```python
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:

```python
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'])
```