edge–SR: Super–Resolution For The Masses

– Appendix –

Pablo Navarrete Michelini, Yunhua Lu, Xingqun Jiang
BOE Technology Group Co., Ltd.

Abstract

We provide the following additional information:

- *Details of evaluation metrics :*
  - *image quality,*
  - *speed,*
  - *power and cpu usage.*
- *Extended analysis :*
  - *speed vs parameters,*
  - *additional edge device (GTX 1080 MaxQ),*
  - *qualitative evaluations,*
  - *effect of datasets and metrics,*
  - *magnified scatter plots.*
- *Reproducibility :*
  - *test results,*
  - *pytorch models and code.*

A. Evaluation Metrics

Image quality. Quantitative evaluations in our experiments include objective metrics PSNR and SSIM. These are reference–based metrics that measure the difference between an impaired image and ground truth. Higher values are better in both cases. The PSNR (range 0 to ∞) is a log–scale version of mean–square–error and SSIM (range 0 to 1) uses image statistics to better correlate with human perception. Full expressions are as follows:

$$PSNR(X,Y) = 10 \cdot \log_{10}\left(\frac{255^2}{MSE}\right) , \tag{1}$$

$$SSIM(X,Y) = \frac{(2\mu_X\mu_Y + c_1)(2\sigma_{XY} + c_2)}{(\mu_X^2 + \mu_Y^2 + c_1)(\sigma_X^2 + \sigma_Y^2 + c_2)} , \tag{2}$$

where $MSE = \mathbb{E}\left[(X-Y)^2\right]$ is the mean square error of the difference between X and Y; μ_X and μ_Y are the averages of X and Y, respectively; σ_X^2 and σ_Y^2 are the variances of X and Y, respectively; σ_{XY} is the covariance of X and Y; $c_1 = 6.5025$ and $c_2 = 58.5225$.

Benchmark in the literature can show big differences in PSNR and SSIM metrics due to different ways to evaluate color. For real–time applications it is common to process color images in YUV space, and the super–resolution task applies only to the luminance channel Y. Color in U and V channels can use a faster bicubic upscaler with small impact in perceptual quality. We follow the implementation in [4] using a conversion of RGB to YUV color–spaces following the BT.709 standard, including offsets that are often avoided in other implementations.

In Appendix B and C we also provide measurements of the perceptual quality metric LPIPS defined in [3] and implemented using the PIQA library [2].

Speed. We run each model to output a set of 14 Full–HD images, downscaling appropriately from randomly selected images of the DIV2K dataset [1]. We use 16–bit floating point precision during inference. For each image we run the model 10 times to avoid warm–up effects, measuring: the minimum CPU and GPU processing time from profiler's data. We computed the speed of a model using the total number of pixels processed (considering only one run per image) divided by the processing time (using the minimum time over each one of the 10 runs). To make the measurement of speed easier to read we use units of `[FHD/s]`, this is, number of Full–HD pixels (1920×1080) per second. Figure 1 shows the code used to parse Pytorch's profiler output to obtain the CPU and GPU processing time.

Power and CPU usage. We run each model to output a set of 14 Full–HD images, downscaling appropriately from randomly selected images of the DIV2K dataset [1]. We use 16–bit floating point precision during inference. During this process we monitor the maximum power consumption using `nvidia-smi` for GTX 1080 Max–Q GPU, and `tegrastats` for Jetson AGX Xavier. We register the maximum power measured in this process. The power data provided by Max–Q driver is in units of watts, whereas the AGX device uses units of milliwatts. The AGX device allows different profiles for power consumption and in our experiments we used 30 Watts.

The Raspberry Pi 400 device does not include power sensors and in this case we replace the power measurement by CPU usage, monitor by parsing the `top` command with delay–time of $0.05s$ and registering the average CPU reading during the inference process.

```python
import numpy as np
import torch.autograd.profiler as profiler

def str_to_time(s):
  if s.endswith('ms'):
    return float(s[:-2])*1e-3
  if s.endswith('us'):
    return float(s[:-2])*1e-6
  return float(s[:-1])

def speed(model, input):
  dt = np.inf
  for _ in range(10):
      with profiler.profile(
        record_shapes=True, use_cuda=True
      ) as prof:
          with profiler.record_function(
            'model_inference'
          ):
              output = model(input)
      dt1 = str_to_time(
          prof.key_averages().table(
            sort_by='cpu_time_total', row_limit=10
          ).split(
            'CPU time total: '
          )[1].split('\n')[0]
      )
      dt2 = str_to_time(
          prof.key_averages().table(
            sort_by='cpu_time_total', row_limit=10
          ).split(
            'CUDA time total: '
          )[1][:-1]
      )
      dt = min(dt1+dt2, dt)

  pix = np.asarray(output.shape).prod()

  return pix / (dt * 1920*1080)
```

Figure 1. Python function used to measure the speed of models in Pytorch v1.8. It runs a model 10 times to avoid warm–up effects. Then, it parses the output of Pytorch profiler to get both CPU and GPU runtime. The speed is the number of output pixels divided by the minimum runtime in the 10 runs.

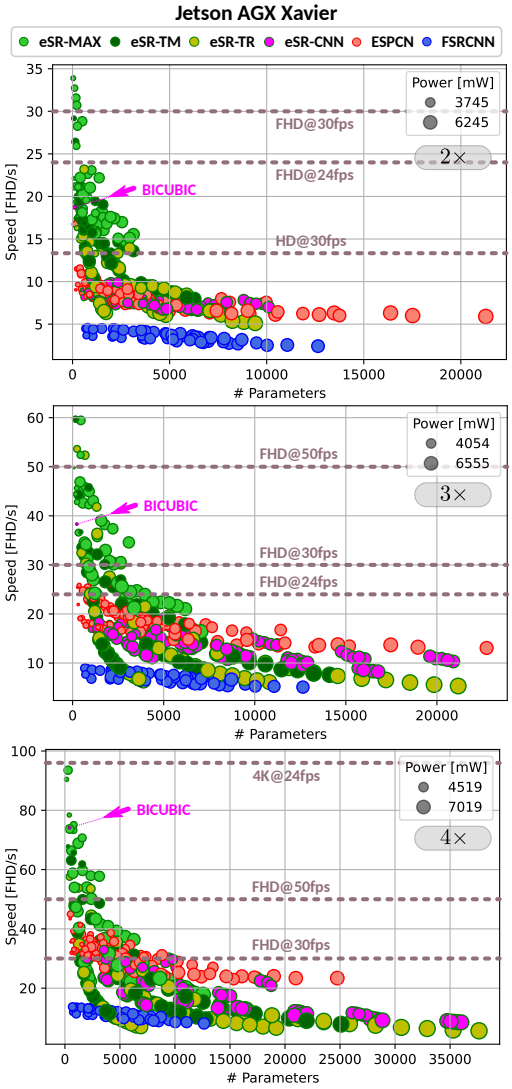

Figure 2. Example of the change in speed of the models with respect to their number of parameters for experiments on the Jetson AGX Xavier device. Models become faster at an exponential rate as the number of parameter reduces.

B. Extended Analysis

Speed vs parameters. In Figure 2 we show the relationship between the size of the model, given by the number of parameters, and the execution speed when running the models on GPU devices. We observe that smaller models run faster, and the speed increases exponentially. The non–linear relation between speed and number of parameters becomes critical under: $5,000$ parameters for $2\times$ upscaling factor, $10,000$ parameters for $3\times$ upscaling factor, and $15,000$ parameters for $4\times$ upscaling factor. This shows the importance of focusing on speed compared to number of parameters in our study. Research on lightweight SR architectures often focuses on number of parameters and typically use several hundred thousand parameters where the non–linearity is still not critical.

Additional device. In Figure 3 we show scatter plots including Nvidia GeForce GTX–1080 Max–Q GPU. This is a mobile high–end GPU from the Pascal series typically used for laptop computers. The power consumption of the Max–Q design ranges between 90 and 110 Watt, compared to 30 Watt used in the Jetson AGX Xavier. Although much more powerful than a Raspberry Pi and Jetson AGX devices, the GTX 1080 Max–Q device can fit in high–end display TV panels and thus serve a different range of applications for edge devices. Consequently, Figure 3 shows a performance that can deliver 8K videos in real–time (even with 16–bit floating point precision). We also observe that ESPCN performance improves for $3\times$ and $4\times$ upscaling factors, indicating the important effect of the increased number of cores in GPU architectures as well as a significant increase in

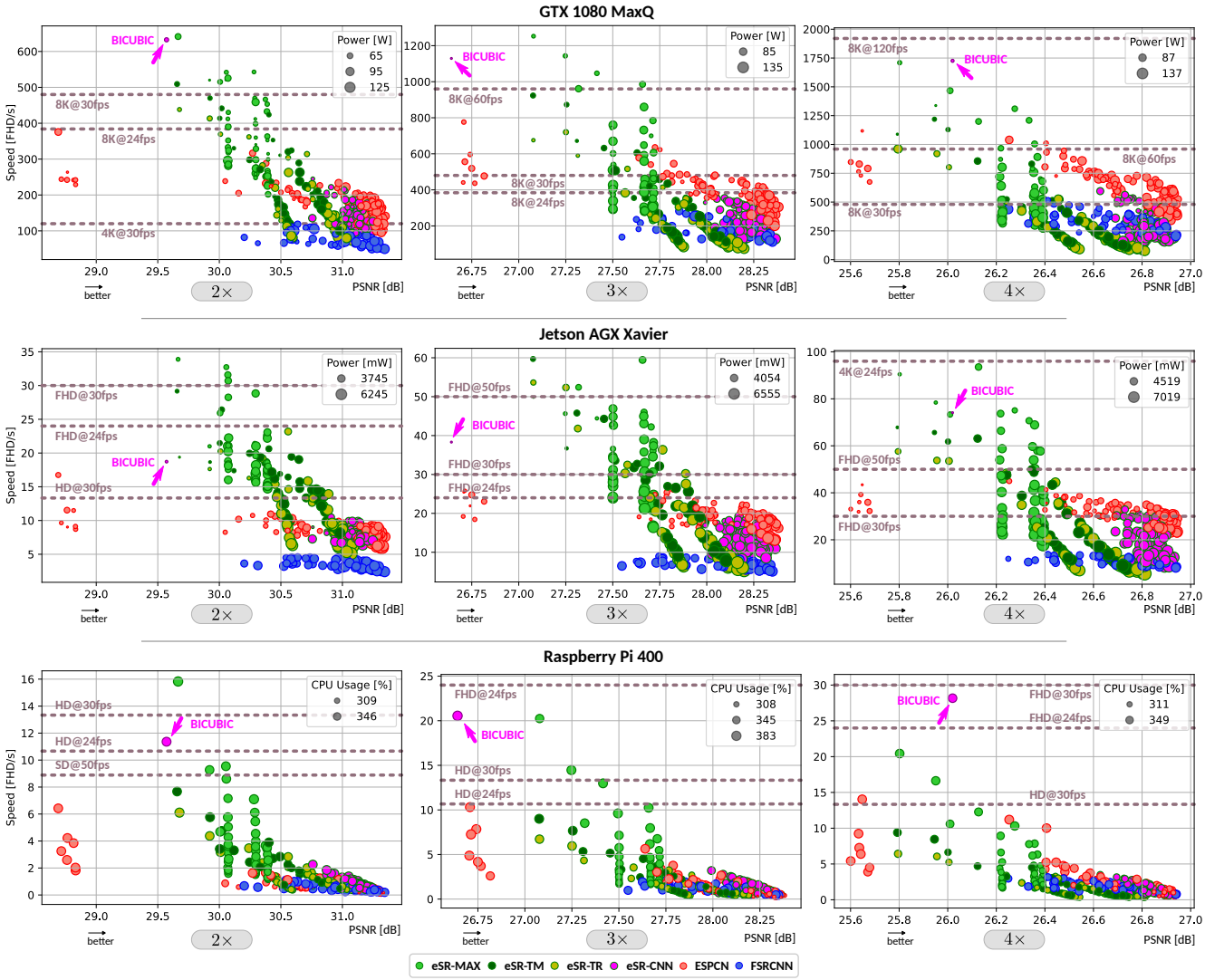

Figure 3. Scatter plot to compare speed, in number of Full–HD pixels per second, with respect to quality, measured as PSNR for the BSDS–100 dataset. A total of $1,185$ models were identically trained considering different upscaling factors ($2\times$, $3\times$ and $4\times$) and architectures (eSR, ESPCN and FSRCNN). We run all models on edge devices: GTX–1080 Max–Q (GPU with 16–bit floating point precision), Jetson AGX Xavier (GPU with 16–bit floating point precision) and Raspberry Pi 400 (CPU with 32–bit floating point precision). Magnified plots with model annotations are provided in the Figures 20, 21, 22, 14, 15 and 16.

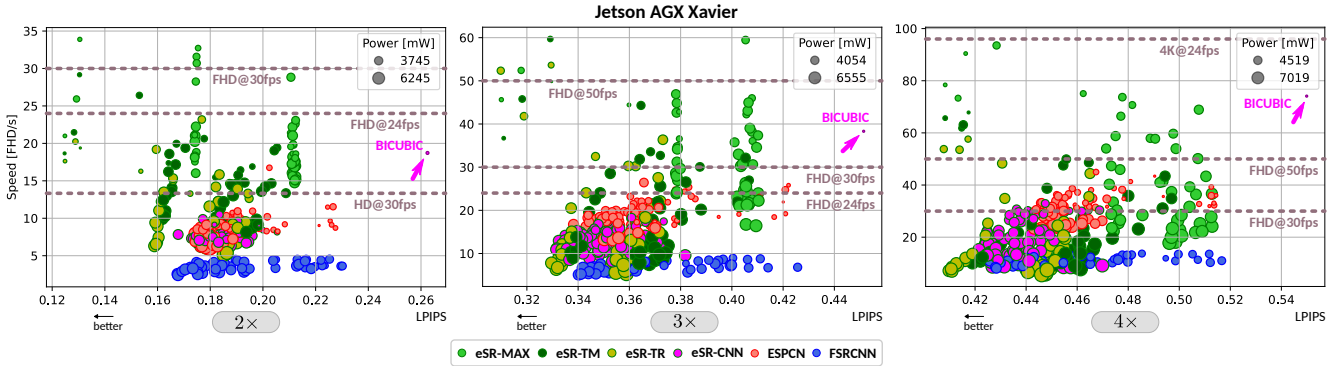

Figure 4. Scatter plot to compare speed, in number of Full–HD pixels per second, with respect to quality, measured as LPIPS for the BSDS–100 dataset. Lower values of LPIPS mean better quality as opposed to PSNR where larger values are better. Compared with the same case but using PSNR quality measure in Figure 3, eSR–TM performs much better and Bicubic shows the worst quality.

```
>>> import pickle
>>> test = pickle.load(open('tests.pkl', 'rb'))
>>> test['Bicubic_s2']
  {'psnr_Set5': 33.72849620514912,
   'ssim_Set5': 0.9283912810369976,
   'lpips_Set5': 0.14221979230642318,
   'psnr_Set14': 30.286027790636204,
   'ssim_Set14': 0.8694934108301432,
   'lpips_Set14': 0.19383049915943826,
   'psnr_BSDS100': 29.571233006609656,
   'ssim_BSDS100': 0.8418117904964167,
   'lpips_BSDS100': 0.26246454380452633,
   'psnr_Urban100': 26.89378248655882,
   'ssim_Urban100': 0.8407461069831571,
   'lpips_Urban100': 0.21186692919582129,
   'psnr_Manga109': 30.850672809780587,
   'ssim_Manga109': 0.9340133711400112,
   'lpips_Manga109': 0.102985977955641,
   'parameters': 104,
   'speed_AGX': 18.72132628065749,
   'power_AGX': 1550,
   'speed_MaxQ': 632.5429857814075,
   'power_MaxQ': 50,
   'temperature_MaxQ': 76,
   'memory_MaxQ': 2961,
   'speed_RPI': 11.361346064182795,
   'usage_RPI': 372.8714285714285}
```

Figure 5. Example of how to read our test data from the Python dictionary file `tests.pkl`.

power consumption.

Qualitative evaluation. In Figures 11, 12 and 13 we show example output images for different models. These models were selected by moving in an approximately optimal trajectory in the Speed vs Quality (SQ) plane for the Jetson AGX Xavier device. As mentioned before, the best advantage of edge–SR models is observed for $2\times$ upscaling factor. This is both the most difficult and important factor for applications due to the high input throughput in high resolution displays (e.g. HD to FHD). The classic bicubic upscalers offers over–smooth outputs that are somehow effective to reduce *jaggy* artifacts. edge–MAX models can significantly improve sharpness with limited control over jaggies. Next, edge–TM and edge–TR models show the best trade–off with very similar performance. These models are more effective at reducing jaggies before multi–layer networks like edge–CNN and ESPCN become better with both sharp and smooth edges.

In Figure 4 we show a few scatter plots of the trade–off between image quality and runtime performance using the perceptual quality metric LPIPS [3] for the BSDS–100 dataset. Here, lower values of LPIPS mean better quality as opposed to PSNR where larger values are better. Compared with the same case but using PSNR quality measure in Figure 3, eSR–TM performs much better and Bicubic shows the worst quality.

At $3\times$ and $4\times$ upscaling factor the pattern is similar but it becomes more difficult for single–layer models to effectively reduce jaggy artifacts. Results get worsen and we observe an increased gap between bicubic and other architectures. ESPCN becomes better at high quality ranges and overcomes edge–SR models for the most part. Finally we note that trade–off evaluations at $3\times$ and $4\times$ upscaling factors could be misleading, as flickering video artifacts are

```python
import torch
from torch import nn

class edgeSR_MAX(nn.Module):
    def __init__(self, C, k, s):
        super().__init__()

        self.pixel_shuffle = nn.PixelShuffle(s)
        self.filter = nn.Conv2d(
            in_channels=1,
            out_channels=s*s*C,
            kernel_size=k,
            stride=1,
            padding=(k-1)//2,
            bias=False,
        )

    def forward(self, input):
        return self.pixel_shuffle(
            self.filter(input)
        ).max(dim=1, keepdim=True)[0]
```

Figure 6. Pytorch v1.8 implementation of edge–SR Maximum (eSR–MAX) single–layer architecture.

```python
import torch
from torch import nn

class edgeSR_TM(nn.Module):
    def __init__(self, C, k, s):
        super().__init__()

        self.pixel_shuffle = nn.PixelShuffle(s)
        self.softmax = nn.Softmax(dim=1)
        self.filter = nn.Conv2d(
            in_channels=1,
            out_channels=2*s*s*C,
            kernel_size=k,
            stride=1,
            padding=(k-1)//2,
            bias=False,
        )

    def forward(self, input):
        filtered = self.pixel_shuffle(
            self.filter(input)
        )
        B, C, H, W = filtered.shape

        filtered = filtered.view(B, 2, C, H, W)
        upscaling = filtered[:, 0]
        matching = filtered[:, 1]
        return torch.sum(
            upscaling * self.softmax(matching),
            dim=1, keepdim=True
        )
```

Figure 7. Pytorch v1.8 implementation of edge–SR Template Matching (eSR–TM) single–layer architecture.

```python
import torch
from torch import nn

class edgeSR_TR(nn.Module):
    def __init__(self, C, k, s):
        super().__init__()

        self.pixel_shuffle = nn.PixelShuffle(s)
        self.softmax = nn.Softmax(dim=1)
        self.filter = nn.Conv2d(
            in_channels=1,
            out_channels=3*s*s*C,
            kernel_size=k,
            stride=1,
            padding=(k-1)//2,
            bias=False,
        )

    def forward(self, input):
        filtered = self.pixel_shuffle(
            self.filter(input)
        )
        B, C, H, W = filtered.shape

        filtered = filtered.view(B, 3, C, H, W)
        value = filtered[:, 0]
        query = filtered[:, 1]
        key = filtered[:, 2]
        return torch.sum(
            value * self.softmax(query*key),
            dim=1, keepdim=True
        )
```

Figure 8. Pytorch v1.8 implementation of edge–SR TRansformer (eSR–TR) single–layer architecture.

likely to become visible at this point and video SR solutions might be needed. For this reason, the results at $2\times$ are arguably the most important for practical applications.

Effect of datasets and metrics. Scatter plots in the main text use only PSNR metric measured in the BSDS–100 dataset. In Figure 10 we show the effect of changing both the metric, from PSNR to SSIM, and dataset, among Set5, Set14, BSDS–100, Urban–100 and Manga–109. The range of values and relative position of scatter points changes. For example, SSIM shows a larger margin in image quality between bicubic and other models. PSNR values show significant changes depending on the dataset. Nevertheless, the trade–off trajectory and the advantage shown by different architectures remains the same.

Magnified scatter plots. It is useful to identify the hyper–parameters of each specific model in scatter plots. Figures 20, 21, 22, 14, 15 and 16 show magnified plots using all the page width and include annotations with model hyper–parameters. The annotations are useful at middle and high speed ranges where data is more sparse. In the high image quality range the performance of different models become clustered and the annotations are not readable. Here, we recommend to look at Figures 11, 12 and 13 to identify the best models. Finally, we make all test data available in a Python dictionary file (see Appendix C).

```python
import torch
from torch import nn

class edgeSR_CNN(nn.Module):
    def __init__(self, C, D, S, s):
        super().__init__()

        self.softmax = nn.Softmax(dim=1)
        if D == 0:
            self.filter = nn.Sequential(
                nn.Conv2d(D, S, 3, 1, 1),
                nn.Tanh(),
                nn.Conv2d(
                    in_channels=S,
                    out_channels=2*s*s*C,
                    kernel_size=3,
                    stride=1,
                    padding=1,
                    bias=False,
                ),
                nn.PixelShuffle(s),
            )
        else:
            self.filter = nn.Sequential(
                nn.Conv2d(1, D, 5, 1, 2),
                nn.Tanh(),
                nn.Conv2d(D, S, 3, 1, 1),
                nn.Tanh(),
                nn.Conv2d(
                    in_channels=S,
                    out_channels=2*s*s*C,
                    kernel_size=3,
                    stride=1,
                    padding=1,
                    bias=False,
                ),
                nn.PixelShuffle(s),
            )

    def forward(self, input):
        filtered = self.filter(input)
        B, C, H, W = filtered.shape

        filtered = filtered.view(B, 2, C, H, W)
        upscaling = filtered[:, 0]
        matching = filtered[:, 1]
        return torch.sum(
            upscaling * self.softmax(matching),
            dim=1, keepdim=True
        )
```

Figure 9. Pytorch v1.8 implementation of edge–SR CNN (eSR–CNN) multi–layer architecture.

C. Reproducibility

Test results. Test results are provided in the Python dictionary file `tests.pkl` using the native *pickle* module. A sample code to read the file is provided in Figure 5. The keys of the dictionary identify the name of each model and its hyper–parameters using the following format:

- `'Bicubic_s#'`,
- `'eSR-MAX_s#_K#_C#'`,
- `'eSR-TM_s#_K#_C#'`,
- `'eSR-TR_s#_K#_C#'`,
- `'eSR-CNN_s#_C#_D#_S#'`,

- `'ESPCN_s#_D#_S#'`, or
- `'FSRCNN_s#_D#_S#_M#'`,

where # represents an integer number with the value of the correspondent hyper–parameter. For each model the data of the dictionary contains a second dictionary with the information displayed in Figure 5. This includes: number of model parameters; image quality metrics PSNR, SSIM and LPIPS measured in 5 different datasets; as well as power, speed, CPU usage, temperature and memory usage for devices `AGX` (Jetson AGX Xavier), `MaxQ` (GTX 1080 MaxQ) and `RPI` (Raspberry Pi 400).

Pytorch implementations. Figures 6, 7, 8 and 9 show the Python code to implement all proposed edge–SR models using the Pytorch tensor processing framework version 1.8. Model files and sample code are also available in https://github.com/pnavarre/eSR.

References

[1] Eirikur Agustsson and Radu Timofte. NTIRE 2017 challenge on single image super-resolution: Dataset and study. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017. 1

[2] François Rozet. Pytorch image quality assessment (PIQA). `https://github.com/francois-rozet/piqa`, 2021. 1

[3] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 586–595, 2018. 1, 4

[4] Yulun Zhang, Yapeng Tian, Yu Kong, Bineng Zhong, and Yun Fu. Evaluation code for residual dense networks. `https://github.com/yulunzhang/RDN/blob/master/RDN_TestCode/Evaluate_PSNR_SSIM.m`, 2018. [Online; accessed 20-May-2020]. 1

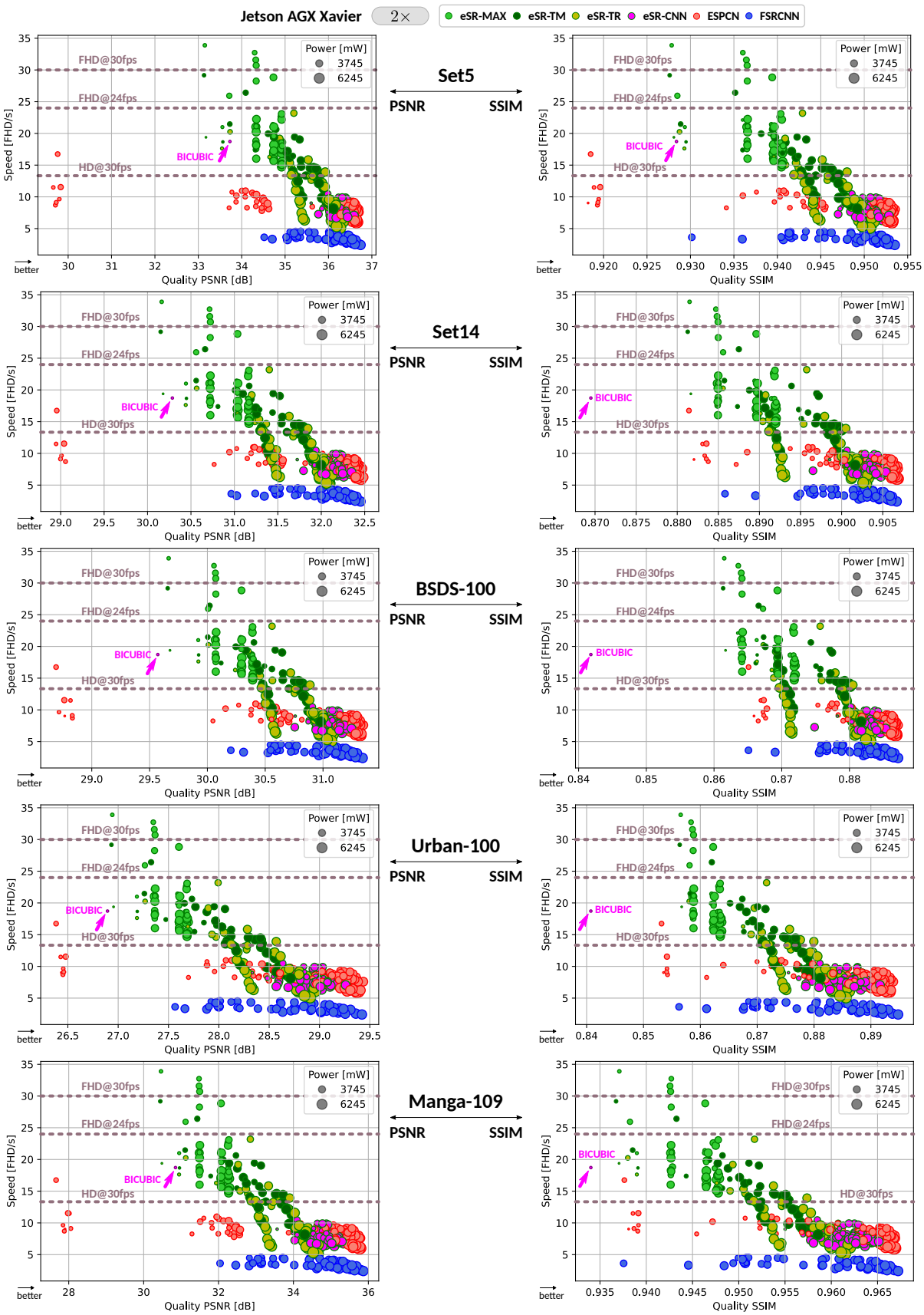

Figure 10. Example of the effect of different metric (PSNR/ SSIM) and datasets in the trade–off between image quality and runtime performance. We observe that even though the range of values and relative positions change, edge–SR models remain with better performance at high speed, and multi–layer networks remain better at low speed.

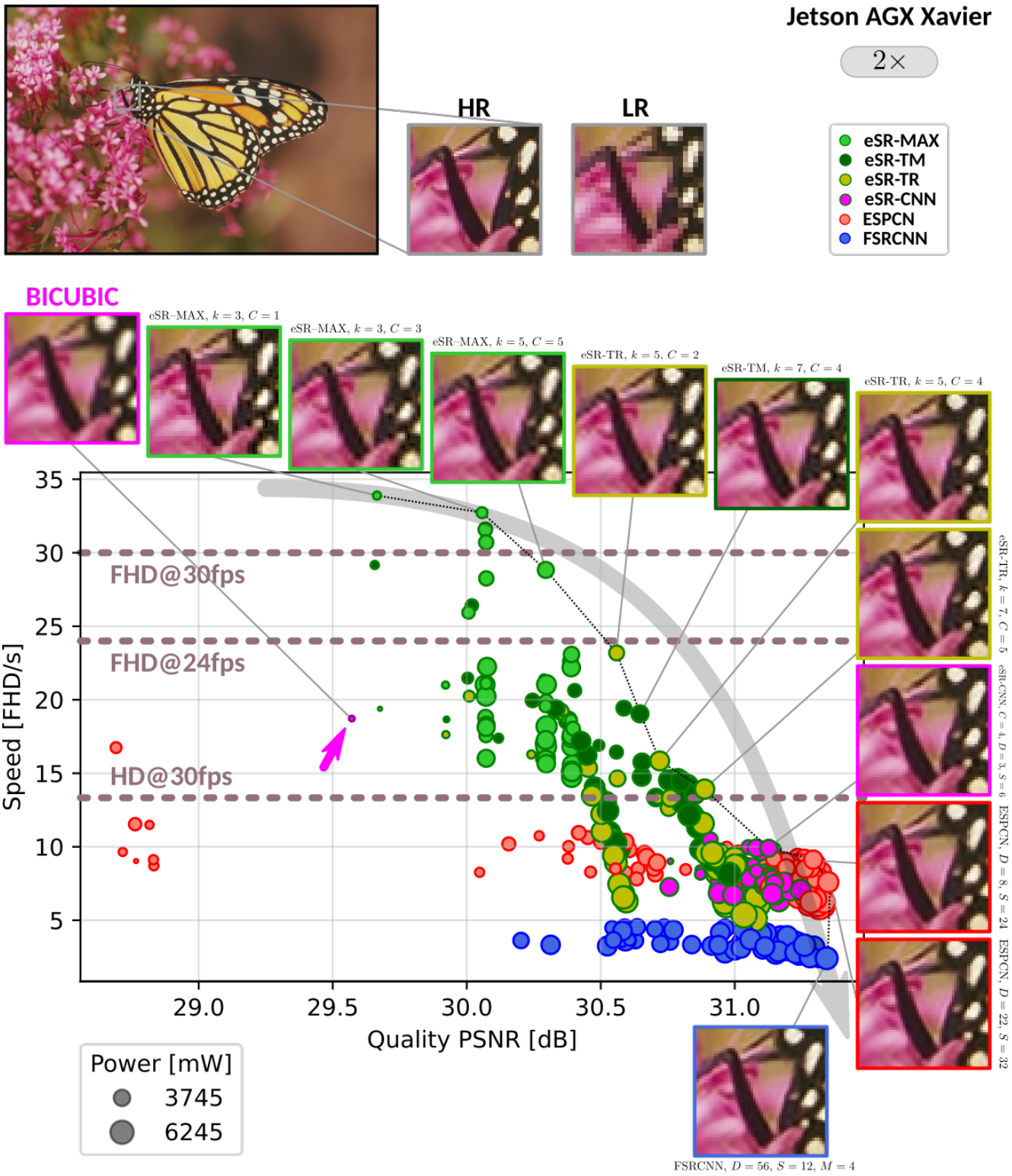

Figure 11. Example output images in the trajectory through the trade–off between image quality and runtime performance for 2× upscaling on a Jetson AGX Xavier edge device.

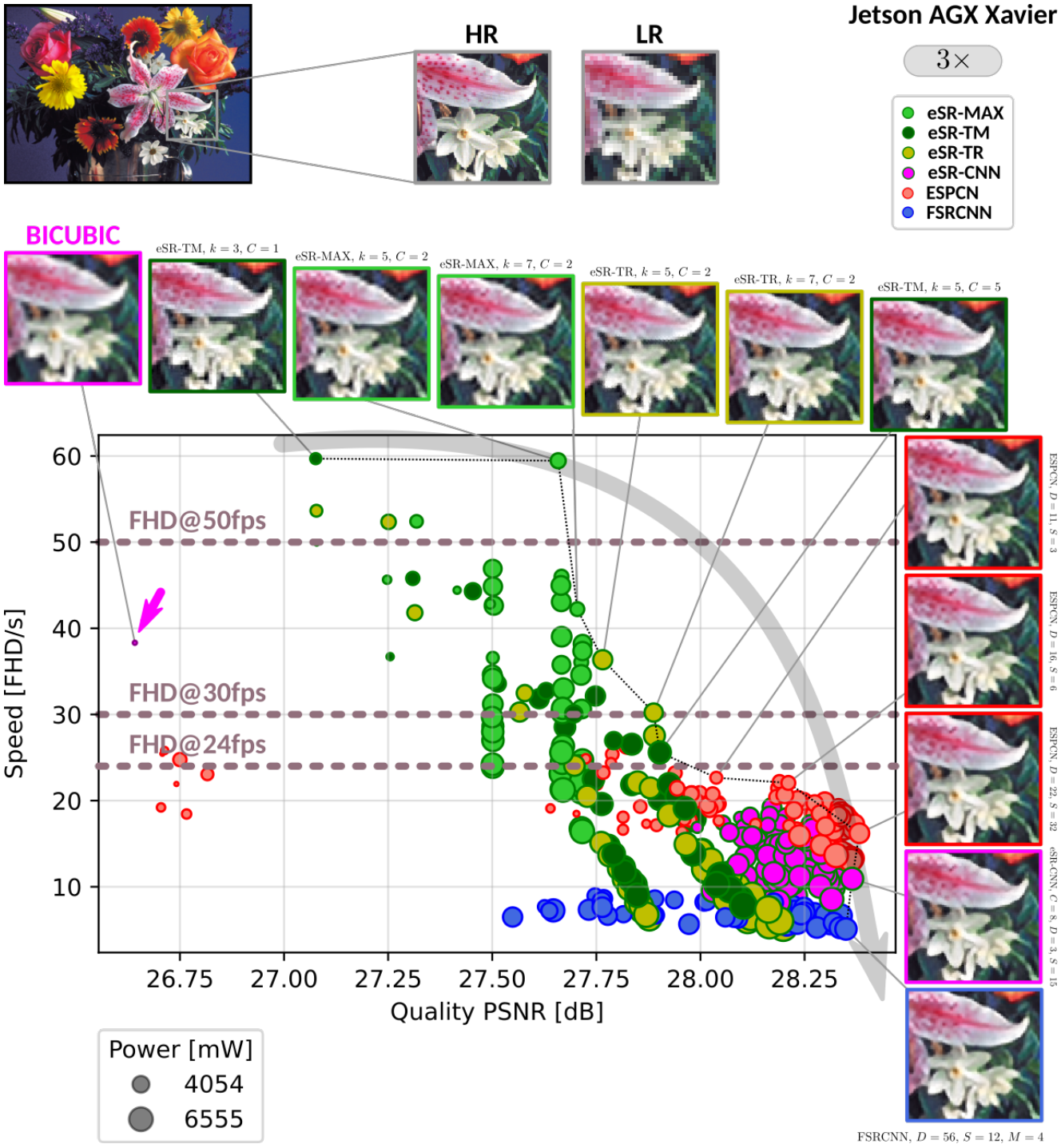

Figure 12. Example output images in the trajectory through the trade–off between image quality and runtime performance for 3× upscaling on a Jetson AGX Xavier edge device.

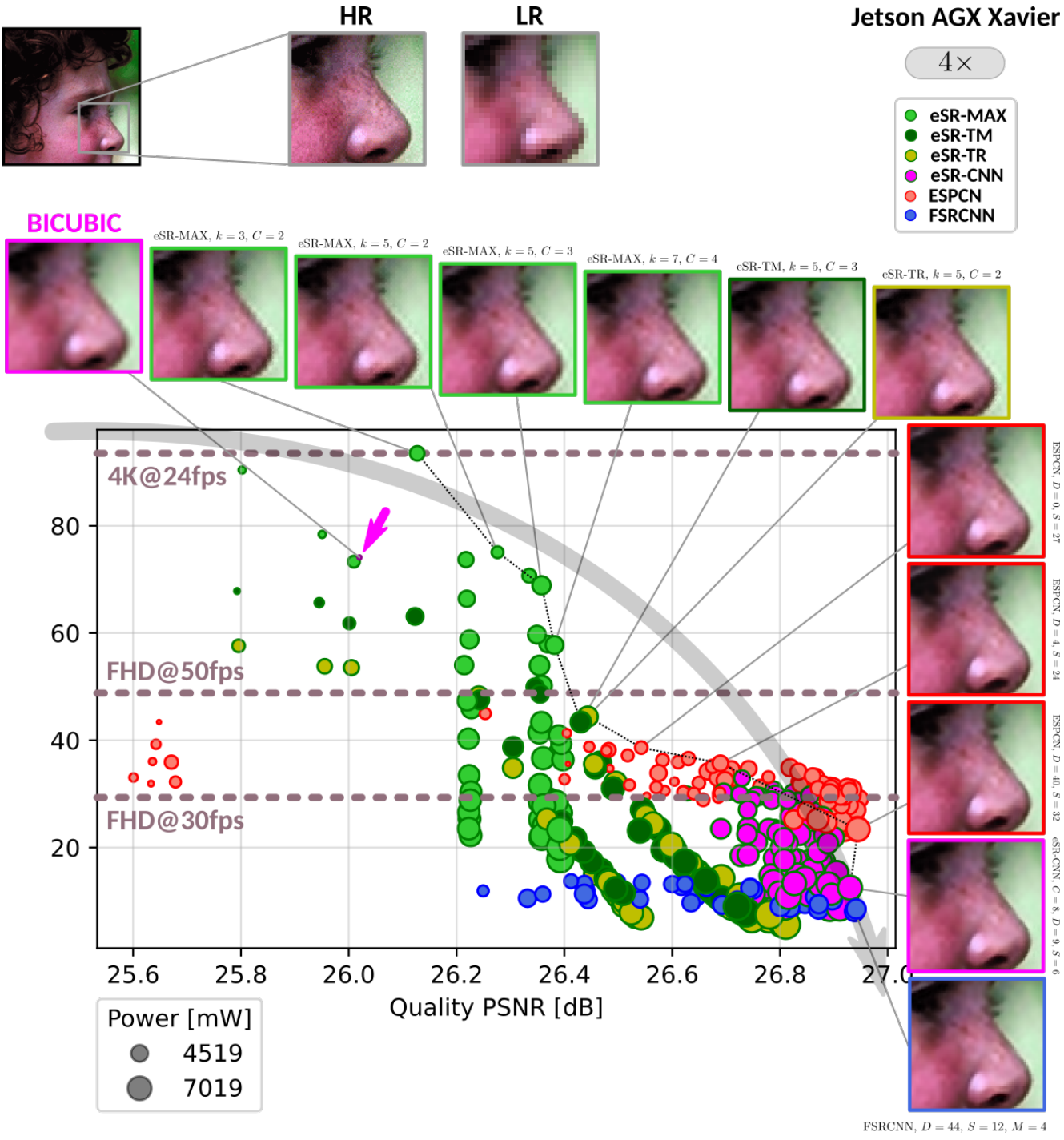

Figure 13. Example output images in the trajectory through the trade–off between image quality and runtime performance for $4\times$ upscaling on a Jetson AGX Xavier edge device.

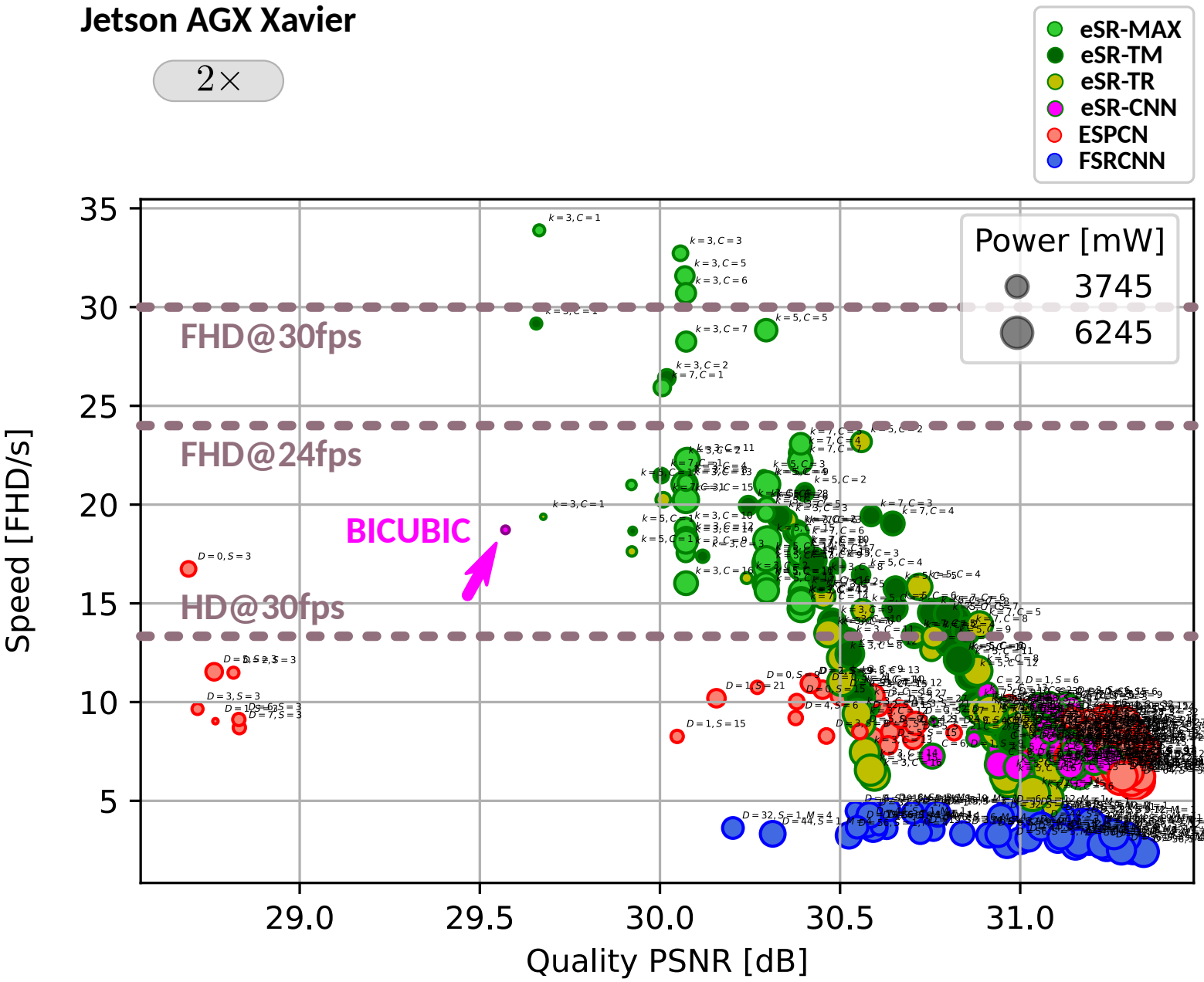

Figure 14. Speed of 2× image SR models, measured in Full–HD pixels per second on a Jetson AGX Xavier GPU using 16–bit floating point precision, with respect to image quality, measure as PSNR in the BSDS–100 dataset.

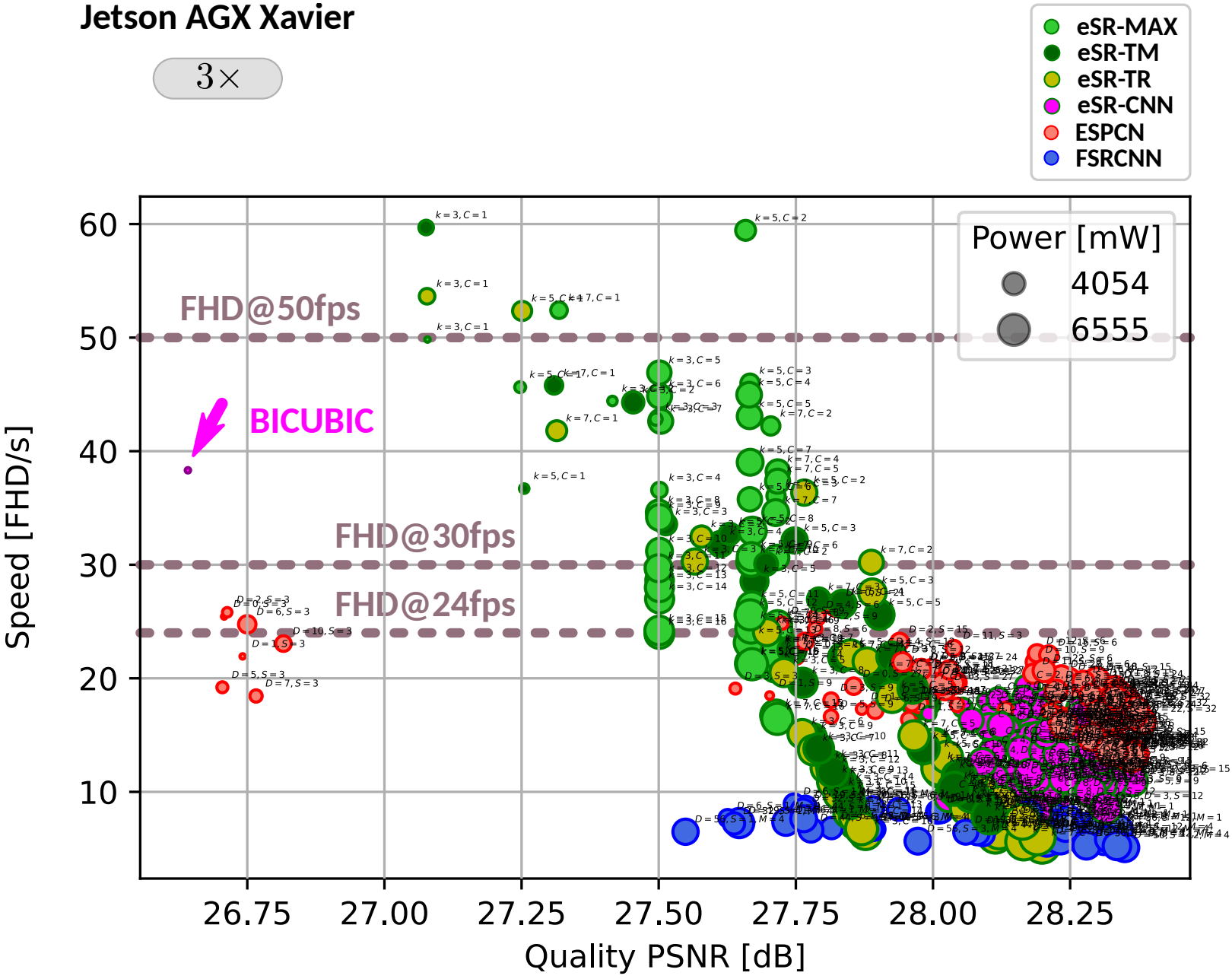

Figure 15. Speed of 3× image SR models, measured in Full–HD pixels per second on a Jetson AGX Xavier GPU using 16–bit floating point precision, with respect to image quality, measure as PSNR in the BSDS–100 dataset.

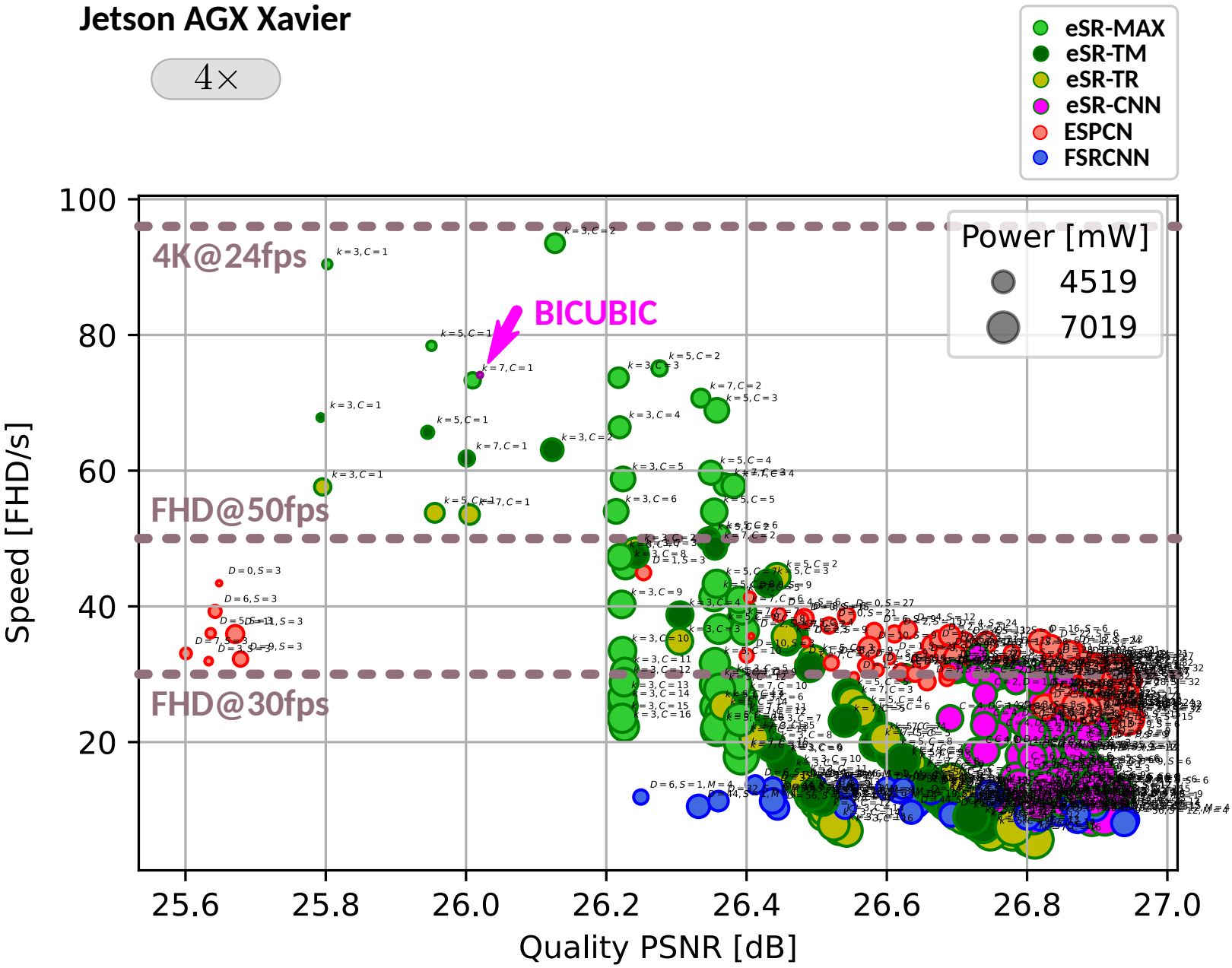

Figure 16. Speed of $4\times$ image SR models, measured in Full–HD pixels per second on a Jetson AGX Xavier GPU using 16–bit floating point precision, with respect to image quality, measure as PSNR in the BSDS–100 dataset.

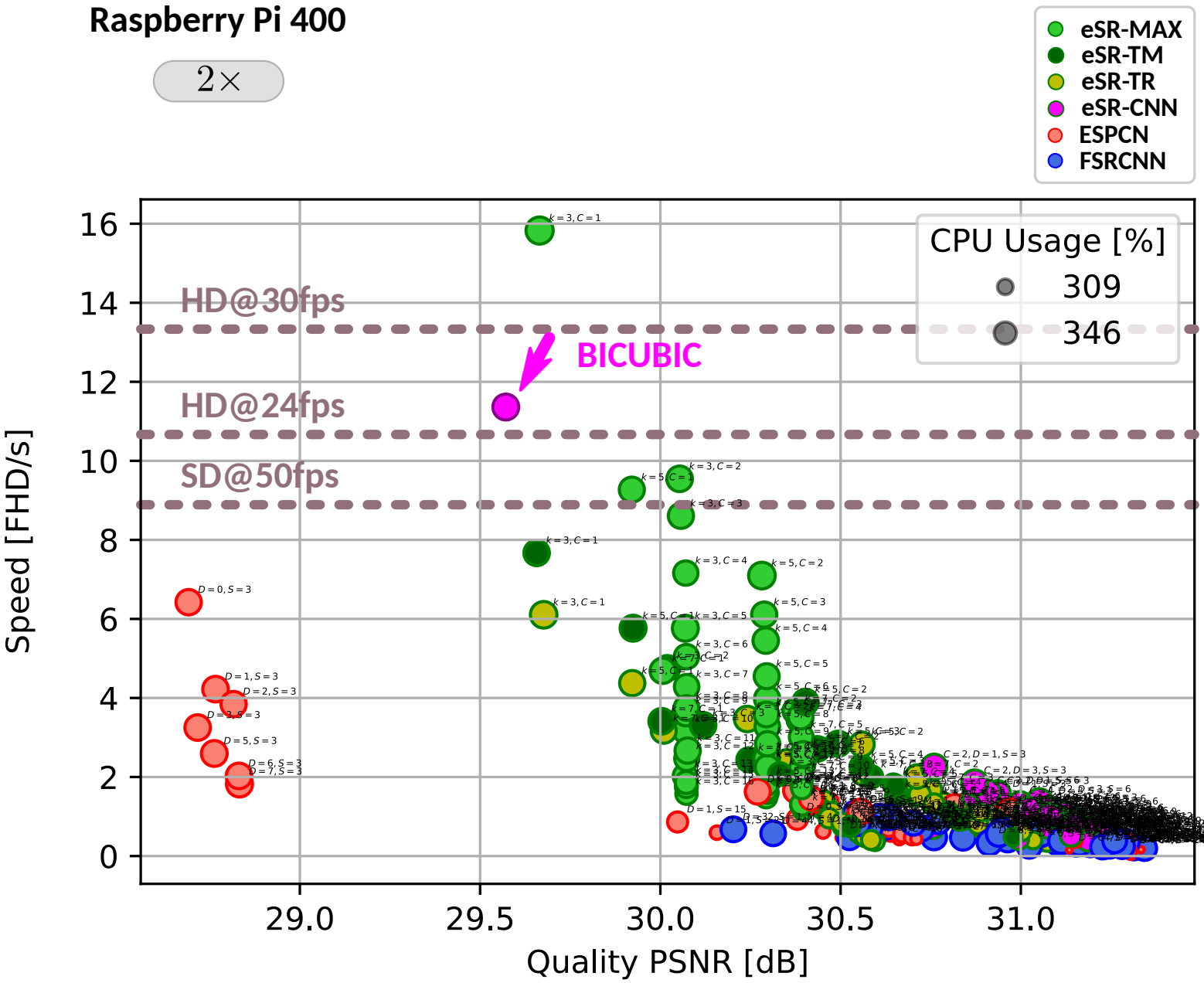

Figure 17. Speed of 2× image SR models, measured in Full–HD pixels per second on a Raspberry Pi 400 CPU using 32–bit floating point precision, with respect to image quality, measure as PSNR in the BSDS–100 dataset.

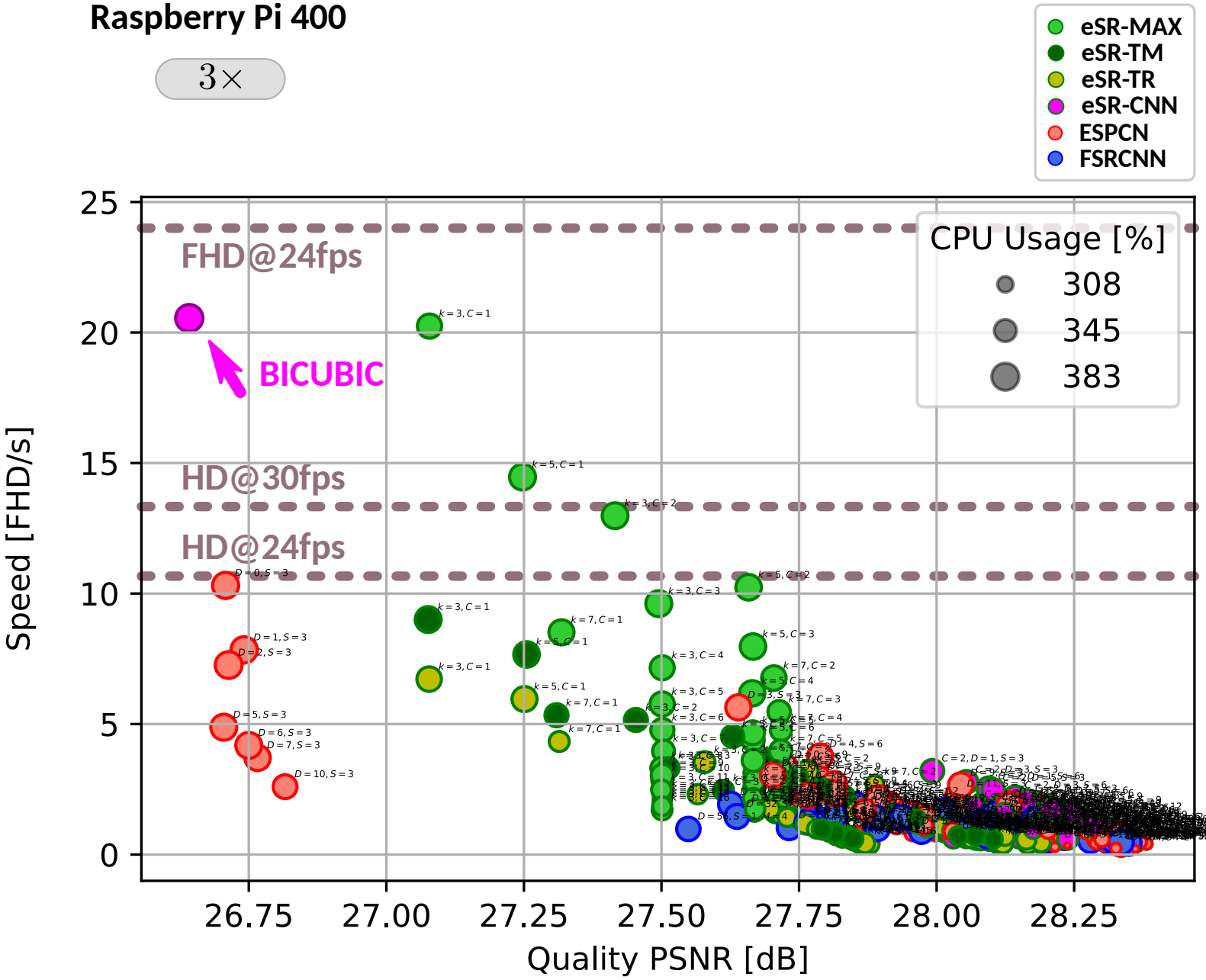

Figure 18. Speed of 3× image SR models, measured in Full–HD pixels per second on a Raspberry Pi 400 CPU using 32–bit floating point precision, with respect to image quality, measure as PSNR in the BSDS–100 dataset.

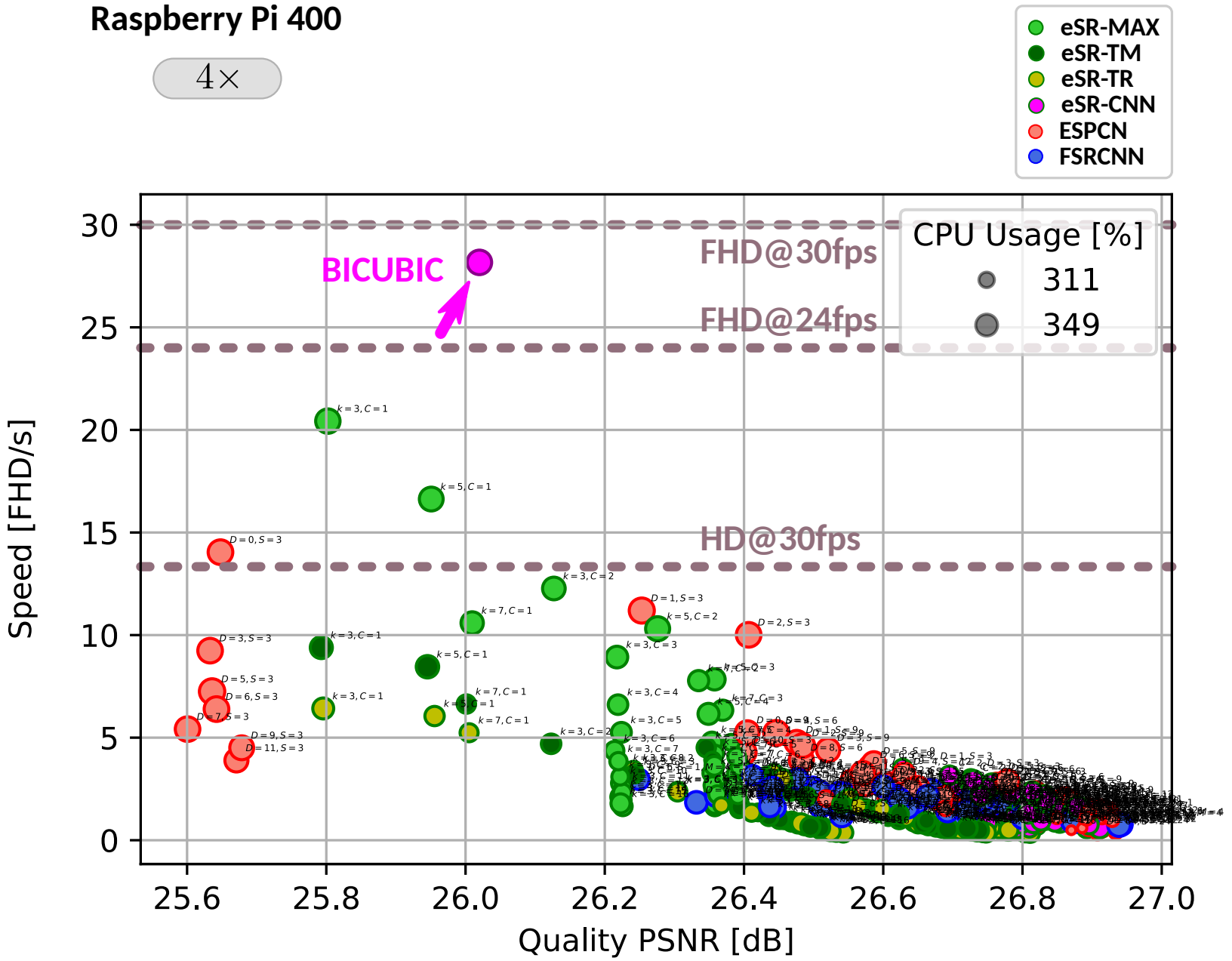

Figure 19. Speed of $4\times$ image SR models, measured in Full–HD pixels per second on a Raspberry Pi 400 CPU using 32–bit floating point precision, with respect to image quality, measure as PSNR in the BSDS–100 dataset.

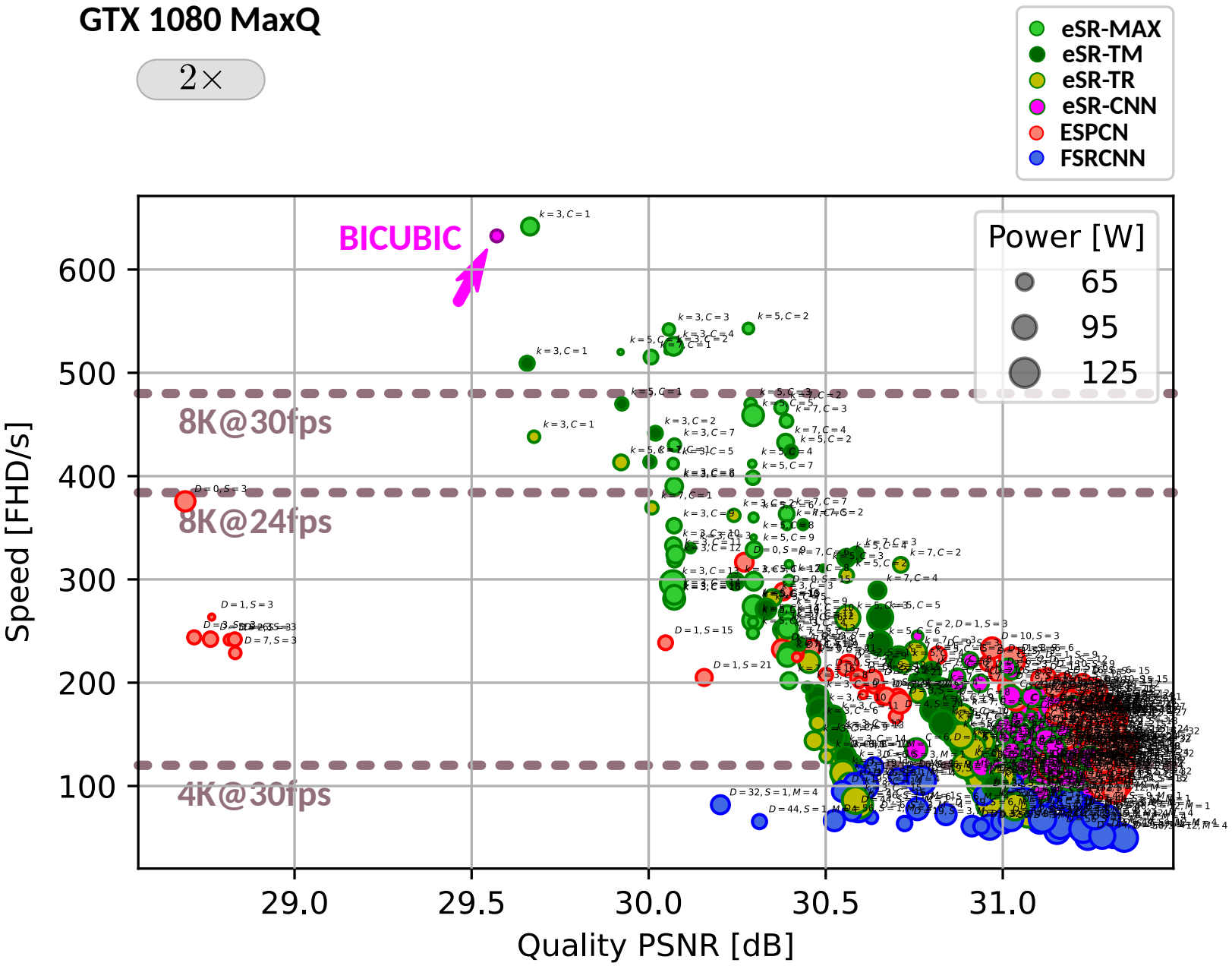

Figure 20. Speed of 2× image SR models, measured in Full–HD pixels per second on a GTX 1080 Max–Q GPU using 16–bit floating point precision, with respect to image quality, measure as PSNR in the BSDS–100 dataset.

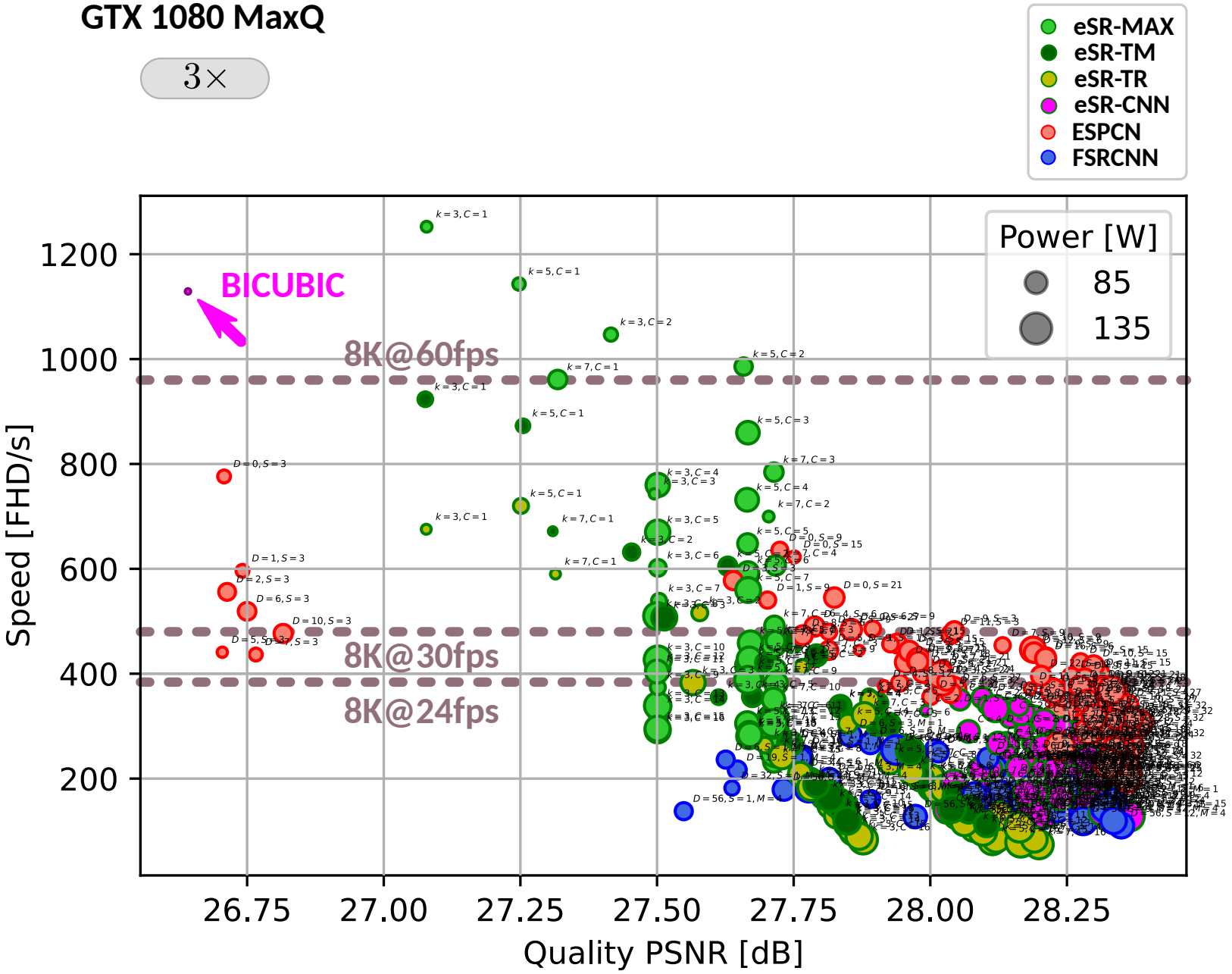

Figure 21. Speed of 3× image SR models, measured in Full–HD pixels per second on a GTX 1080 Max–Q GPU using 16–bit floating point precision, with respect to image quality, measure as PSNR in the BSDS–100 dataset.

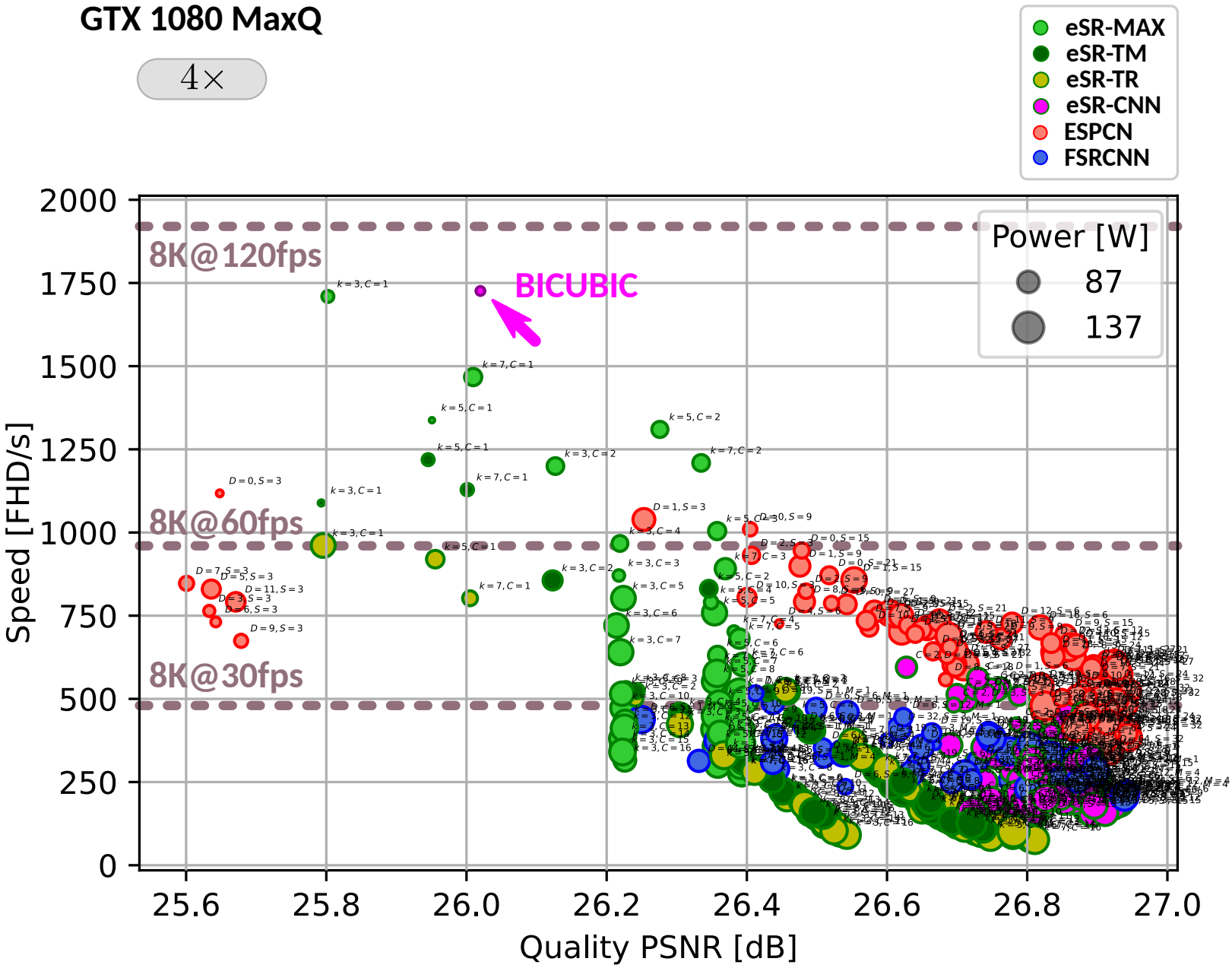

Figure 22. Speed of $4\times$ image SR models, measured in Full–HD pixels per second on a GTX 1080 Max–Q GPU using 16–bit floating point precision, with respect to image quality, measure as PSNR in the BSDS–100 dataset.