This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2020.3025300, IEEE Transactions on Mobile Computing

Supremo: Cloud-Assisted Low-Latency Super-Resolution in Mobile Devices

Juheon Yi, Seongwon Kim, Joongheon Kim, *Senior Member, IEEE,* and Sunghyun Choi, *Fellow, IEEE*

Abstract—We present Supremo, a cloud-assisted system for low-latency image Super-Resolution (SR) in mobile devices. As SR is extremely compute-intensive, we first further optimize state-of-the-art DNN to reduce the inference latency. Furthermore, we design a mobile-cloud cooperative execution pipeline composed of specialized data compression algorithms to minimize end-to-end latency with minimal image quality degradation. Finally, we extend Supremo to video applications by formulating a dynamic optimal control algorithm to design Supremo-Opt, which aims to maximize the impact of SR while satisfying latency and resource constraints under practical network conditions. Supremo upscales 360p image to 1080p in 122 ms, which is 43.68× faster than on-device GPU execution. Compared to cloud offloading-based solutions, Supremo reduces wireless network bandwidth consumption and end-to-end latency by 15.23× and 4.85× compared to baseline approach of sending and receiving whole images, and achieves 2.39 dB higher PSNR compared to using conventional JPEG to achieve similar data size compression. Furthermore, Supremo-Opt guarantees robust performance in practical scenarios.

Index Terms—Mobile Deep Learning, Cloud Offloading, Image Super-Resolution

1 Introduction

In recent years, the widespread of High-Resolution (HR) displays on mobile devices has led to unprecedented demands for HR images. However, acquiring HR images is costly; capturing HR images requires high-end cameras, and images (either captured from camera or downloaded from the Internet) are often intentionally degraded (via downsampling or compression) to save storage. When a user zooms into such images (e.g., to take a closer look on a distant object or identify a tiny text), degraded resolution significantly harms the user experience.

Image Super-Resolution (SR), a technique to reconstruct HR image from Low-Resolution (LR) image, provides a promising opportunity to obtain HR images without additional hardware costs. While the task has been a classical computer vision problem, Deep Neural Network (DNN)-based methods have recently achieved remarkable performance. However, running them on mobile devices is extremely challenging due to high computational complexity.

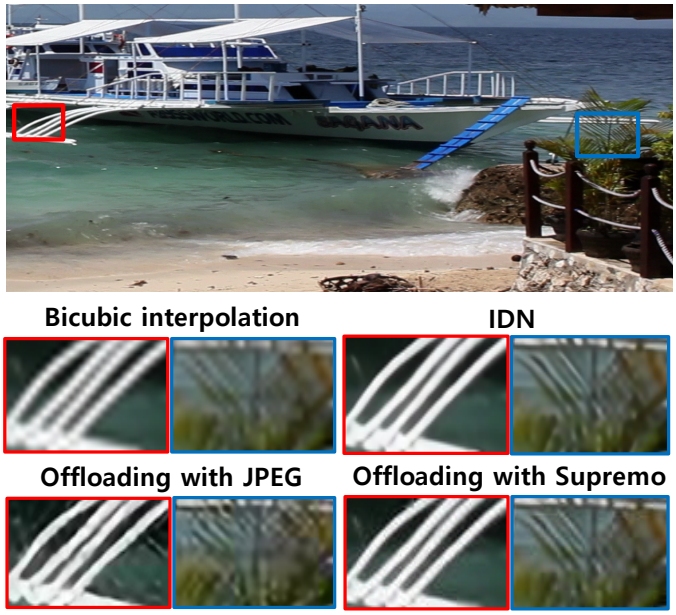


Fig. 1: Performance example of Supremo with IDN [1] as the SR model. Using JPEG to compress the image size during the offloading process incurs annoying artifacts. With the same data size compression, Supremo retains most of the performance gain of IDN.

Especially, SR requires much more computation than other tasks, making low-latency on-device execution extremely difficult despite several recent attempts to run DNNs (mostly for classification and detection) on mobile devices at the scale of 1 fps [2], [3], [4]. Even with cutting-edge framework TensorFlow-Lite, running the most optimized SR model [1] on high-end mobile GPU takes more than 5

- *J. Yi is with the Institute of Computer Technology, Seoul National University, Seoul, Korea (e-mail: johnyi0606@snu.ac.kr).*
- *S. Kim is with T-Brain, SK Telecom, Seoul, Korea (e-mail: s1kim@sktbrain.com).*
- *J. Kim is with the School of Electrical Engineering and also with Artificial Intelligence Engineering Research Center, Korea University, Seoul, Korea (e-mail: joongheon@korea.ac.kr).*
- *S. Choi is with the Advanced Communications Research Center, Samsung Research, Samsung Electronics, Korea (e-mail: sunghyunc.research@gmail.com).*
- *S. Kim and J. Kim are corresponding authors of this paper.*

- *This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2020-2017-0-01637) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation) and also partially by the Institute for Information & Communications Technology Promotion Grant funded by the Korea government (MSIT) 2018-0-00170, Virtual Presence in Moving Objects through 5G.*

seconds to upscale a 360p image to 1080p (Fig. 2).

In this paper, we present Supremo (**Sup**er-**re**solution in **mo**bile devices), a cloud-assisted system for low-latency SR in mobile devices. The goal of Supremo is to enhance resolution of a zoomed-in LR image (e.g., 360p) to commodity mobile device display resolutions (e.g., 1080p for Google Pixel 2) in soft real-time (e.g., 100 ms).[1] Supremo is composed of comprehensive optimization across DNN model architecture as well as the entire offloading pipeline to minimize end-to-end latency with minimal image quality degradation. Fig. 1 shows an example performance of Supremo: It efficiently compresses the image to minimize the network transmission latency in the offloading process while at the same time preserve the performance gain of SR, whereas utilizing conventional JPEG to achieve similar compression results in annoying artifacts.

We face several technical challenges is designing Supremo. Although several recent studies have utilized cloud servers to offload the heavy DNN computation [5], [6], [7], [8], [9], [10], they have been mostly focused on image classification and object detection tasks. Simply extending existing work for SR is non-trivial due to several reasons. First, prior studies have paid little attention to the DNN inference latency due to the availability of highly optimized models (e.g., Faster R-CNN [11] object detector runs in 60 fps on NVIDIA Titan XP GPU [10]). However, SR models incur high latency even at the servers (e.g., IDN [1] takes 77.0 ms to upscale a 640×360 image to 1,920×1,080 on GTX 1080 Ti GPU), requiring an optimization at the model architecture itself to minimize end-to-end latency. Second, there exists a large gap in the data size that needs to be sent through the wireless network (i.e., labels or bounding boxes vs. HR images), resulting in significant network transmission latency. While prior studies have mostly utilized conventional image compression (e.g., JPEG) to compress the image size sent to the cloud, such approach severely degrades SR performance (see Section 2.2 for details).

To tackle the aforementioned challenges, we first further optimize state-of-the-art SR model IDN [1] to design IDN-Lite. We observe that IDN is designed by stacking basic blocks; inherent redundancy arises as performance increment per each block quickly becomes marginal in latter stages. Alternatively, we gradually stack compressed unit blocks to reduce model size with minimum performance loss. Our approach achieves 3.01× computational complexity reduction with minimal 0.1 dB Peak Signal-to-Noise Ratio (PSNR) loss in reconstruction performance.

Second, we design a mobile-cloud cooperative execution pipeline to run the offloading process at low latency. We first develop two data compression algorithms specialized for SR, namely *Priority Ordering* and *Residual Encoding*. While Priority Ordering operates on the mobile-side to offload only necessary parts of an image, Residual Encoding operates on the cloud-side to send back compressed residual signal. Furthermore, we employ *Offload-Inference Pipelining* to optimize latency by parallelizing components in the execution pipeline that consumes different resources (i.e., CPU, GPU, and wireless network). Overall, Supremo upscales 360p image to 1080p in 122 ms, which is 43.68× faster than on-device GPU execution. Compared to cloud offloading, Supremo reduces network bandwidth consumption and end-to-end latency by 15.23× and 4.85×, respectively with 0.81 dB PSNR loss compared with sending and receiving whole images. In contrast, using JPEG to achieve similar compression suffers from 3.20 dB loss.

Finally, we extend Supremo to video applications (e.g., real-time zoom-in, video streaming) to design Supremo-Opt. As offloading stream of images incurs large network bandwidth consumption, we need to pay close attention to network latency and resource constraints on mobile devices (e.g., LTE data plan) as well as latency requirements of the application. Supremo-Opt incorporates a new dynamic optimal control algorithm inspired by *Lyapunov optimization framework* [12] to maximize the impact of SR while jointly satisfying the target latency and resource constraints under non-ideal network conditions.

Our major contributions can be summarized as follows:

- We carefully optimize the state-of-the-art SR model IDN to design IDN-Lite, which optimizes the inference time by 3.01× with only 0.1 dB PSNR loss.
- We develop Supremo, a cloud-assisted system to for low-latency mobile SR. Supremo achieves 43.68× latency gain compared to on-device mobile GPU execution, and reduces wireless network bandwidth consumption and end-to-end latency by 15.23× and 4.85×.
- We extend Supremo on videos applications to develop Supremo-Opt, which incorporates a dynamic optimal control algorithm to maximize the performance of SR under resource and latency constraints.
- We implement Supremo on Android smartphone and desktop server, and validate its performance via real-world experiments.

The rest of the paper is organized as follows. We first analyze the challenges in Section 2, followed by an overview of Supremo in Section 3. Section 4 and Section 5 provide detailed explanation of the components of Supremo. We then introduce Supremo-Opt in Section 6. We describe the implementation of Supremo in Section 7, and evaluate the performance in Section 8. We summarize related work in Section 10 and conclude the paper in Section 11.

2 Preliminary Studies

In this section, we conduct preliminary studies to analyze challenges in delivering low-latency SR in mobile devices.

2.1 Why On-Device Execution is Difficult

Due to high computational complexity, running SR on mobile devices is extremely challenging. Table 1 compares the complexity of representative DNNs for classification, detection, and SR in terms of MultAdds. We use the same setting as in [13], [14], [15] where classification assumes ImageNet [16], detection assumes COCO [17], and SR assumes upscaling a 360p image to 720p. For SR, PSNR on benchmark Urban100 [18] is also compared. SR models are

1. While supporting higher resolutions (e.g., 4K UHD) may be desirable, display resolutions of mobile devices are limited at the moment (e.g., even cutting-edge Galaxy S20+ only supports 3,200×1,800 resolution). In this work, we target 1080p resolution support which is similar to common mobile multimedia applications (e.g., YouTube and Netflix only support up to 1080p resolution at the moment).

TABLE 1: Complexity comparison of DNNs for various tasks in terms of the number of MultAdds required for inference. DNNs for SR are also compared in terms of PSNR.

Task	Model	MultAdds	PSNR
Image classification	VGG16 [21]	15.3G	-
	ResNet-18 [22]	1,818M	-
	MobileNetV1 [19]	575M	-
	MobileNetV2 [14]	300M	-
	CondenseNet [13]	274M	-
Object detection	SSD512 [23]	99.5G	-
	SSD300 [23]	35.2G	-
	YOLOv2 [24]	17.5G	-
	MobileNetV2+SSDLite [14]	0.8G	-
Super-resolution	DRCN [25]	9,788.7G	30.75 dB
	DRRN [26]	6,796.9G	31.23 dB
	MemNet [27]	623.9G	31.31 dB
	VDSR [28]	612.6G	30.76 dB
	CARN [15]	222.8G	31.51 dB
	IDN [1]	168.7G	31.27 dB
	SRCNN [20]	52.7G	29.50 dB

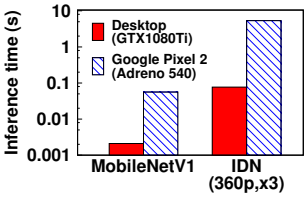


Fig. 2: Inference time comparison of DNNs on mobile and desktop GPUs.

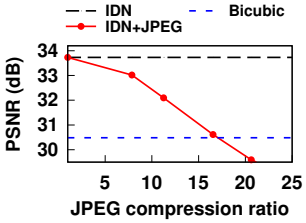


Fig. 3: Impact of JPEG compression on SR performance.

significantly heavier than others (as high as 1,000×), as they output HR images whereas others output labels or bounding boxes.

As a result, Figure 2 shows that even with TensorFlow-Lite, it takes more than 5 seconds to run the most optimized model IDN [1] on Google Pixel 2 with Qualcomm Adreno 540 GPU (for comparison, we also measure the inference time of MobileNetV1 [19] provided in the official TensorFlow-Lite demo app). Even at the cloud, the inference latency is unignorable; DNN optimization needs to take place especially when aiming to minimize end-to-end offloading latency. Simply employing small-sized models such as SRCNN [20] cannot be a solution, as they incur significant drop in PSNR as shown in Table 1.

2.2 Why Cloud Offloading is Challenging

As offloading for SR involves sending images and receiving upscaled HR counterparts, large data traffic flying on the network incurs significant network bandwidth consumption and latency. For example, our evaluation in Section 8.2 shows that with 90.2 Mbps Wi-Fi connection, naively offloading a 640×360 image for ×3 upscaled 1080p image results in 594 ms latency, out of which the DNN inference takes only 77 ms. Such latency would further increase in more challenging network conditions (e.g., average Wi-Fi speed on mobile devices is 24.4 Mbps [29]).

Utilizing conventional image encoding techniques (e.g., JPEG) to compress data size can be a tempting option to alleviate the latency issue. However, they come at the cost of noticeable image quality degradation, quickly diminishing the gain from SR. Specifically, the core mechanism of image encoding is to transform the image into frequency domain and quantize high frequency components more coarsely,

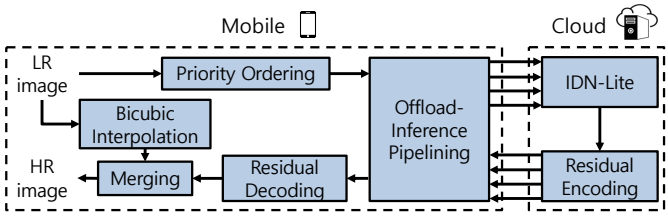


Fig. 4: Overall system architecture of Supremo.

mostly concentrated on the edges in the image. However, this directly contradicts with SR, which enhances image quality by sharpening the edges; Fig. 3 shows that PSNR drops sharply when IDN is applied on a JPEG compressed image. Even worse, compressing beyond certain level yields lower PSNR than applying bicubic interpolation (the simplest upsampling method) on mobile device without offloading. Fig. 1 also shows that offloading with 16.5× JPEG compression results in annoying distortions. Thus, specialized compression algorithms must be designed to reduce network bandwidth consumption while minimizing the SR performance degradation.

3 Supremo: Overview

Fig. 4 depicts the operational flow of Supremo. The mobile-side operation begins with *Priority Ordering*, which divides the LR image into blocks and determines which of them should be offloaded to the cloud with highest priority based on the expected impact of SR. Upon receiving the image from the mobile device, the cloud upscales it with *IDN-Lite*, a lightweight variant of the state-of-the-art model IDN, to obtain the residual signal (i.e., difference) between the HR image and bicubic interpolation. *Residual Encoding* leverages the sparsity of this residual signal to compress the data size sent back to the mobile device.

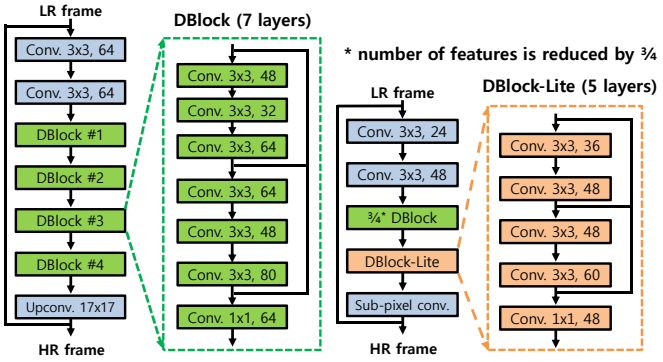
While the mobile device sends the LR image and waits for the residual signal from the cloud, it also runs bicubic interpolation on the LR image in parallel on a separate thread. Upon receiving the encoded residual signal, the mobile device decodes and merges it on top of the bicubic interpolated image and renders it on the screen. Finally, the overall offloading process is executed with *Offload-Inference-Pipelining* to parallelizable components (i.e., network transmission, DNN inference, encoding and decoding) and optimize end-to-end latency.

4 Design of IDN-Lite

We first design an optimized DNN to minimize the inference latency. From the two state-of-the-art optimized models IDN [1] and CARN [15], we choose IDN [1] as starting point as it fits better for our purpose. Specifically, although the two models show similar performance on benchmark datasets, CARN-M takes RGB image as input whereas IDN takes only Y channel in YCbCr color space and applies bicubic interpolation on Cb and Cr channel. Consequently, CARN-M incurs 3× more wireless network bandwidth consumption and network latency than IDN.

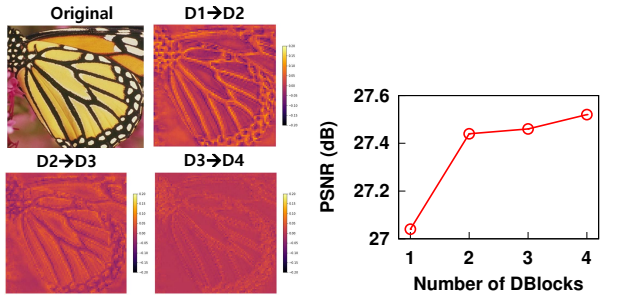
Fig. 5 depicts the architecture of IDN and our lightweight variant *IDN-Lite*. Overall, the number of features is scaled to 0.75×. The initial two convolutional layers for feature

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2020.3025300, IEEE Transactions on Mobile Computing



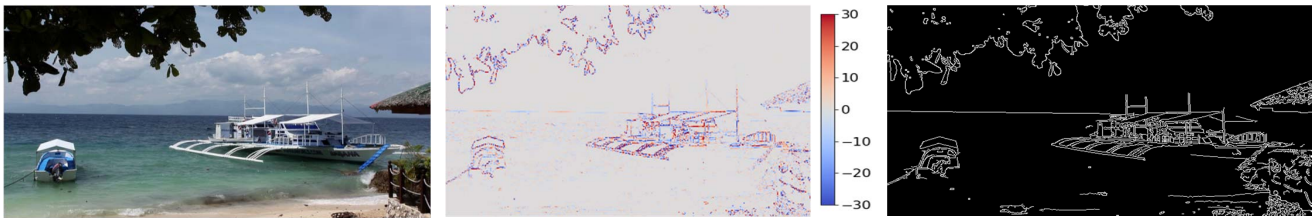
(a) Original IDN (31 layers). (b) IDN-Lite (15 layers).

Fig. 5: Model architectures of IDN and IDN-Lite.



(a) Feature map increments in baseline IDN with 4 DBlocks. (b) PSNR on Urban100 for IDN with various number of DBlocks.

Fig. 6: Performance analysis of IDN.



(a) Raw image. (b) Residual signal (IDN−bicubic). (c) Detected edges.

Fig. 7: Motivation of *Priority Ordering*. As the performance of DNN-based SR and bicubic interpolation differ only on the edges in the image, it only suffices to send the edge blocks.

extraction are modified to gradually increase the number of features. We also replace the last upconvolutional layer with 17×17 kernel to a sub-pixel convolutional layer [30] with 3×3 kernel to enhance efficiency.

Our key idea lies in the modified cascade architecture. IDN stacks four 7-layered *Distillation Blocks* (or *DBlocks*) as shown in Fig. 5(a). Such architecture embeds structural redundancy; performance gain per each additional DBlock becomes marginal in latter stages. Fig. 6(a) shows that the difference between the output feature maps of each DBlock averaged over the feature dimension becomes more sparse in latter stages, meaning that the additional information (i.e., performance gain) becomes marginal. Generalizing this on benchmark Urban100 [18], Fig. 6(b) shows that PSNR gain per additional DBlock becomes marginal when the number of DBlocks is larger than two.

To this end, we arrive at two ideas: (1) stacking just two DBlocks would yield almost the same performance as the original model, and (2) since the additional information per each DBlock becomes marginal in latter stages, gradually reducing the stacked DBlock size would be a more efficient design than simply repeating the equal-sized ones. Accordingly, we only stack one DBlock and one compressed 5-layered *DBlock-Lite* as depicted in Fig. 5(b). Our evaluation verifies that such design yields 3.01× complexity reduction (168.7G vs. 56.1G MultAdds) with only 0.1 dB PSNR loss.

5 SUPREMO: EXECUTION PIPELINE

In this section, we detail techniques to run the offloading process at low latency. We first develop two specialized data compression algorithm to reduce network bandwidth consumption; *Priority Ordering* operates on the mobile-side to offload only necessary parts of an image, while *Residual Encoding* compresses the data sent back to the mobile device. We also develop *Offload-Inference Pipelining* to run components in the execution pipeline that consumes different resources in parallel to optimize end-to-end latency.

5.1 Priority Ordering

The question we want to answer here is the following: *"If we could apply SR only on part of an image, which part should have the top priority?"* It is shown in Fig. 7 that when we run both IDN and bicubic interpolation on an image in Fig. 7(a) and observe the residual signal (i.e., difference) between the two as shown in Fig. 7(b), they differ mostly on the edges in the image. From this observation we obtain some insights: (1) Offloading only the parts of an image that contain edges to the cloud and running bicubic interpolation on the rest on the mobile device will effectively yield the same performance as offloading the whole image, and (2) the part of the image where the expected impact of SR would be the greatest is where the edges are most concentrated.

Based on this idea, Priority Ordering operates as follows. First, we detect edges in an image using Canny edge detector, which can robustly detect edges in the image as in Fig. 7(c). As the complexity of Canny edge detector for an $H \times W$ image is $O(HW \cdot \log(HW))$, its latency is negligible on commodity devices (e.g., 2 ms for 640×360 image in Google Pixel 2). Second, we divide the image into $n \times n$ blocks, and sort them according to edge intensity. Finally, we create a priority queue of blocks based on the sorted order that lets us determine which block in an image has the highest priority for offloading. This prioritized list of blocks is utilized to efficiently adapt the amount of data offloaded

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2020.3025300, IEEE Transactions on Mobile Computing

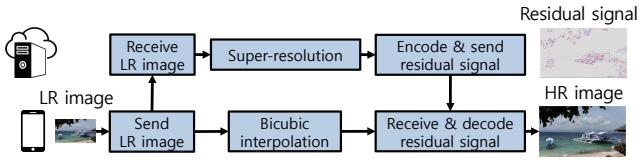


Fig. 8: Operational flow of Residual Encoding.

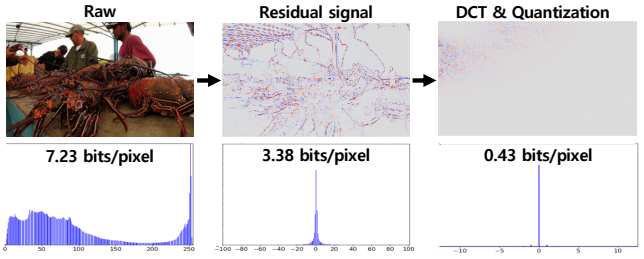


Fig. 9: Residual Encoding process and entropy comparison.

under resource constraints and latency requirements, as will be introduced in Section 6.

5.2 Residual Encoding

Compared to previous offloading frameworks for classification or detection [5], [6], [8], [9], [10] where the size of downlink data (i.e., label or bounding box coordinates) is negligible compared to that of uplink (i.e., image stream), offloading for SR is much more challenging as the server needs to send back a stream of HR frames to the client. We employ *Residual Encoding* to compress this huge downlink data by exploiting the sparsity of residual signals. As illustrated in Figure 8, Residual Encoding operates in a cooperative manner, where the client sends the LR frame to the server and runs bicubic interpolation in background, while the server sends back the residual signal (i.e., difference) between the HR frame reconstructed by DNN and bicubic interpolation. Not only does this approach fit well with the operation of Priority Ordering (i.e., the mobile-side runs bicubic interpolation on the image), it also enables further data compression, as residual signal between HR frame and bicubic interpolation is more sparse than that of the HR frame. Furthermore, when the DNN at the cloud-side employs global residual learning as in IDN (i.e., the output of the network is the difference from bicubic interpolation), residual signal can be obtained without any overhead.

For encoding of the residual signal, we take the similar approach taken in JPEG; we transform raw pixel values into frequency domain using 2-dimensional Discrete Cosine Transform (DCT) and quantize their coefficients. Figure 9 shows that such process makes the residual signal extremely sparse, thus enabling a significant size compression. Note that applying the quantization process directly to the output HR image would normally incur significant degradation on image quality, as analyzed in Section 2.2. However, utilizing the residual signal helps alleviate such degradation since the majority of coefficients are already zero (i.e., quantization does not incur additional degradation).

To achieve near-entropy compression, we use Huffman encoding after applying run-length encoding and zigzag scan. Huffman code table is generated offline from pre-collected image dataset to remove the overhead of creating the optimal code for each input frame. As quantized residual signals are very sparse (zero coefficients taking up more than 90%), a single universal code yields almost the same performance, as will be verified in Section 8.

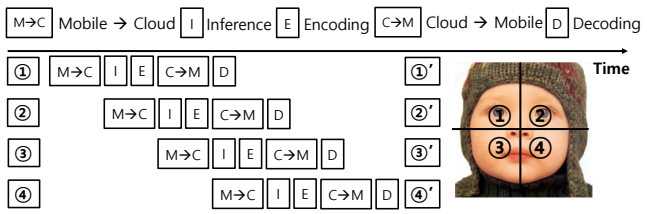


Fig. 10: Operation of Offload-Inference Pipelining.

5.3 Offload-Inference Pipelining

While the aforementioned compression algorithms dramatically reduce network bandwidth consumption (and thus network latency), they incur additional processing overhead. Simply running them sequentially may cause high latency, diminishing the gain in network latency. We employ *Offload-Inference Pipelining* technique to retain the gain. As depicted in Fig. 10, rather than processing the image as a whole, we divide it into macroblocks[2] and process each of them as a separate image. As each component in the offloading process can run in parallel (e.g., the server can run DNN on GPU, encode residual signal on CPU, and send encoded data to the client in the wireless link simultaneously), such approach can greatly optimize end-to-end latency.

An important parameter to be determined for pipelining is the number of macroblocks in which the image is divided into. Dividing the image too finely suffers from performance degradation due to the following issues: i) blocking artifacts due to more macroblock boundaries, and ii) CPU-GPU memory copy overhead in DNN inference computation. Through an empirical evaluation of the tradeoff in Section 8, we divide each input image into four macroblocks.

Note that while pipelining requires that each macroblock is offloaded sequentially, Priority Ordering sorts the blocks according to the number of edge pixels, regardless of which macroblocks they are included in. This issue can be simply resolved by rearranging the transmission order of blocks in the priority queue so that the server can receive the blocks belonging to each macroblock consecutively.

5.4 Frame Merging and Rendering

At the final stage, we merge the decoded residual signal on top of the bicubic interpolated image, and render it on the screen. The frame merging step introduces negligible latency overhead compared to prior processing steps, as it involves simply adding the two arrays and clipping the values outside the pixel range. While applying different upscaling on different sub-images and stitching them may yield blocking artifacts, we observe such issue not so significant in

2. To avoid confusion with the basic unit in the compression algorithms (8×8 block), we use the term macroblock here to indicate the pipeline unit.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2020.3025300, IEEE Transactions on Mobile Computing

Algorithm 1 Runtime execution flow.

1: $PriorityQueue \leftarrow \{\}$
2: $Edge \leftarrow CannyEdgeDetector(LR)$
3: **for** $Block$ in LR **do**
4: **if** edge pixel present in $Block$ **then**
5: $PriorityQueue \leftarrow PriorityQueue \cup Block$
6: $Bicubic \leftarrow RunBicubicInBackground(LR)$
$Residual \leftarrow Offload(PriorityQueue)$
7: $HR \leftarrow Add(Bicubic, Residual)$
8: Render HR on screen

Supremo. It is because we apply bicubic interpolation only to regions with very few or no edges (e.g., background). For such regions, applying bicubic interpolation does not make any difference compared to applying SR, and thereby incurs minimal artifacts (see Fig. 1 and Fig. 13 for visual examples).

To further deal with the deblocking artifact issue, we can utilize the deblocking filter commonly used in video codecs (e.g., H.265/HEVC) to alleviate the blocking artifacts. As deblocking filter does not impose much overhead (only 13% of the total latency in the entire real-time HEVC video decoding process on ARM [31]), employing it will not significantly affect the end-to-end latency (besides, it can also be parallelized using pipelining).

5.5 Putting Things Together

Algorithm 1 summarizes the overall flow. Given an LR image, edges are detected using Canny edge detector (line 2). Blocks that contain edge pixels are inserted to the priority queue to be offloaded (lines 3–5). Afterwards, the mobile runs bicubic interpolation in background simultaneously while offloading the priority queue and receiving and decoding the residual signal (line 6). Finally, the decoded residual signal (received from the cloud) is merged on top of bicubic interpolation to generate the HR image (line 7), and rendered on the screen (line 8).

6 On-Demand Optimization on Videos

In this section, we introduce Supremo-Opt, Supremo extended for video applications (e.g., real-time zoom-in, video streaming). Offloading continuous frames incurs large network bandwidth consumption, incurring significant network latency especially under non-ideal network conditions (e.g., bandwidth fluctuation in Wi-Fi [32]). Therefore, we need to pay close attention on latency and resource constraints (e.g., LTE data plan), as well as user demands on latency-performance tradeoff (e.g., one might want to sacrifice frame rate for better image quality, while another may put more emphasis on seamless latency). To tackle the challenge, we formulate a dynamic optimal control algorithm inspired by Lyapunov optimization framework [12] (which has been commonly utilized for video streaming [32], [33], [34]) to maximize the impact of SR while satisfying user-specified latency requirement and resource constraints.

Fig. 11 illustrates the operation of Supremo-Opt: for each frame, Priority Ordering (denoted as PO) first sorts the blocks by edge intensity, as in baseline Supremo. Furthermore, we also extend Caching Mechanism (denoted as CM) commonly used in prior continuous mobile vision

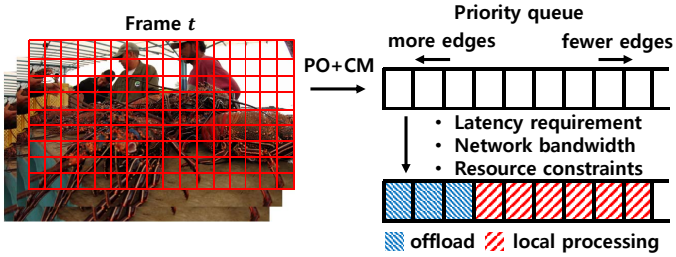
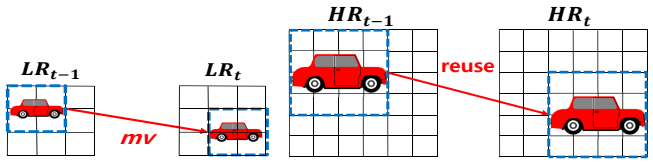


Fig. 11: Overview of Supremo-Opt.



(a) Find motion vector between input frames.

(b) Based on the motion vector, reuse previous upscaled frame.

Fig. 12: Operation of Caching Mechanism applied for SR.

systems [3], [4], [5], [10] to exploit the temporal redundancy of continuous frames and reuse the results from the previous upscaled HR frame instead of offloading (note that the idea of Caching Mechanism has been mostly utilized for image classification or object detection tasks in prior works; we verify that it can be effective for SR as well in Section 8.6). Once the priority queue of edge blocks sorted in edge intensity is created, we determine how many blocks to offload based on latency requirement, estimated network bandwidth, and resource constraints. As the impact of SR is expected to be stronger on blocks with higher edge intensity, this approach guarantees that resources are used in a most efficient manner.

Specific operation of the Caching Mechanism is depicted in Figure 12. We first run the Hexagon Search-based block matching algorithm [35] between the consecutive LR frames to find matching blocks. We use the Mean Absolute Error (MAE) between the pixel values as distance metric. For the blocks with matching distance below a pre-defined threshold, we reuse the corresponding blocks in upscaled previous HR frame rather than offloading it to the cloud for processing. With higher distance threshold, the number of blocks reused increases (thus reducing the network transmission latency), while at the same time degrade SR performance due to possible distortions that occur in stitching upscaled blocks from adjacent frames. The performance tradeoff is analyzed in Section 8.6). We also note that incorporating highly-optimized video encoding (e.g., H.264) can further improve the performance of Caching Mechanism, which we would like to investigate in our future work. Specifically, it can benefit our system in two aspects: (i) optimize the latency of calculating the motion vectors between the input frames, and (ii) reduce the data size by sending the residual signals between the continuous frames.

Objective. Our goal is to efficiently allocate offloading resources on input frame sequence to maximize the overall impact of SR. We define the utility (i.e., reward) U_t for frame t as a logarithmic function of the ratio of the number of edge blocks covered by SR, either by offloading (Y_t) or caching

(Z_t), to the total number of edge blocks in the frame (X_t):

$$U_t = \log\left(1 + \frac{Y_t + Z_t}{X_t}\right), \quad (1)$$

where constant term one is added to make the utility value positive. The rationale behind this modeling is to reflect the fact that the increments in the impact SR applied according to the sorted priority queue will gradually decrease, as will be verified in Section 8.

Constraints. We assume that the application can offload d_{target} (Mbps) on average due to network usage budget (e.g., LTE data plan). Furthermore, to satisfy user-specified latency requirement while preventing undesirable resource wastage caused by stale data arriving after the frame has already been rendered, we adjust the number of blocks to offload based on estimated network bandwidth, so that offloading process terminates within target latency L_{max}. Specifically, during each offloading event, the server and the client exchange timestamps to estimate the uplink and downlink bandwidth, assuming that the clocks are synchronized (e.g., by IEEE 1588 Precision Time Protocol). We use exponential moving weight average to estimate network bandwidth for next offloading event as in [32].

To summarize, our problem is formulated as follows:

$$\begin{aligned} \text{Maximize} \quad & \sum_{t=1}^{N} U_t = \sum_{t=1}^{N} \log\left(1 + \frac{Y_t + Z_t}{X_t}\right), \\ \text{subject to} \quad & \max\left(\frac{b^2 \cdot Y_t}{BW_{up,t}} + \frac{\gamma_t \cdot s^2 \cdot b^2 \cdot Y_t}{BW_{down,t}}, T_{SR}, Y_t \cdot T_{dec}\right) \\ & \leq L_{max}, \\ & \frac{1}{N}\sum_{t=1}^{N}(1 + \gamma_t \cdot s^2) \cdot b^2 \cdot Y_t \leq d_{target}, \end{aligned} \quad (2)$$

where b and s represent the block size and the scale factor, γ_t represents the compression ratio of Residual Encoding (estimated in a similar manner as the network bandwidth), $BW_{up,t}, BW_{down,t}$ represent the estimated uplink and downlink bandwidth for the t-th offloading event among N total offloading events. T_{SR} represents the DNN inference latency for SR, and T_{dec} represents the residual decoding latency per each block, which is profiled at the target mobile device (e.g., it takes 5 μs to decode a 8×8 block in Google Pixel 2). Determining Y_t to satisfy the first constraint ensures that the offloading latency does not exceed the latency constraint; note that as we employ Offload-Inference Pipelining, the end-to-end offloading latency is bounded by the largest latency component in the execution pipeline (i.e., network transmission, processing steps at the mobile and the cloud). The second constraint guarantees that the application satisfies the network usage budget.

Resource Constraints into Virtual Queues. Time average resource constraints can be modeled and incorporated in the Lyapunov optimization framework as virtual queue to be stabilized. We model network usage constraint as $Q_d[t]$ whose backlog evolves as follows:

$$Q_d[t+1] = \max\left(Q_d[t] + (1 + \gamma_t \cdot s^2) \cdot b^2 \cdot Y_t - d_{target}, 0\right). \quad (3)$$

Stabilizing $Q_d[t]$ guarantees that network usage constraint is satisfied. Similarly, other resource constraints (e.g., energy consumption) can be incorporated easily as virtual queues.

Drift-Plus-Penalty Minimization. Lyapunov optimization framework minimizes drift-plus-penalty. Before defining drift-plus-penalty, Lyapunov drift $\Delta[t]$ is defined as the difference between Lyapunov function $L[t]$ per time slot (or offloading event), which is the square of the virtual queue backlog $Q_d[t]$.

$$\Delta[t] = L[t+1] - L[t] = \frac{1}{2}\Big((Q_d[t+1])^2 - (Q_d[t])^2\Big). \quad (4)$$

We also define penalty as $-U_t$ to follow the conventional penalty minimization in Lyapunov optimization. Finally, drift-plus-penalty ($DPP[t]$) is defined as:

$$DPP[t] = \Delta[t] + V \cdot penalty[t] = \Delta[t] - V \cdot \log\left(1 + \frac{Y_t + Z_t}{X_t}\right), \quad (5)$$

where V is a positive constant that controls the tradeoff between virtual queue backlog and deviation from theoretical bound of maximum utility.

Greedy Scheduling Algorithm. By greedily minimizing $DPP[t]$ at each time slot, Lyapunov optimization framework achieves time-average utility maximization performance deviating by at most $O(1/V)$ from optimality, while satisfying time-average resource consumption deviating from constraints by at most $O(V)$ [12]. In Supremo-Opt, the greedy scheduling algorithm involves iteratively computing $DPP[t]$ for each possible Y_t (ranging from 0 to $X_t - Z_t$) and choosing the value that maximizes the $DPP[t]$. As calculating $DPP[t]$ is a simple computation, greedy scheduling algorithm incurs minimal overhead at runtime. For example, with the 640×360 input image divided into 8×8 blocks, X_t becomes 3,600 (=80×45), and calculating $DPP[t]$ for all possible Y_t takes less than 1 ms in Google Pixel 2.

7 Implementation

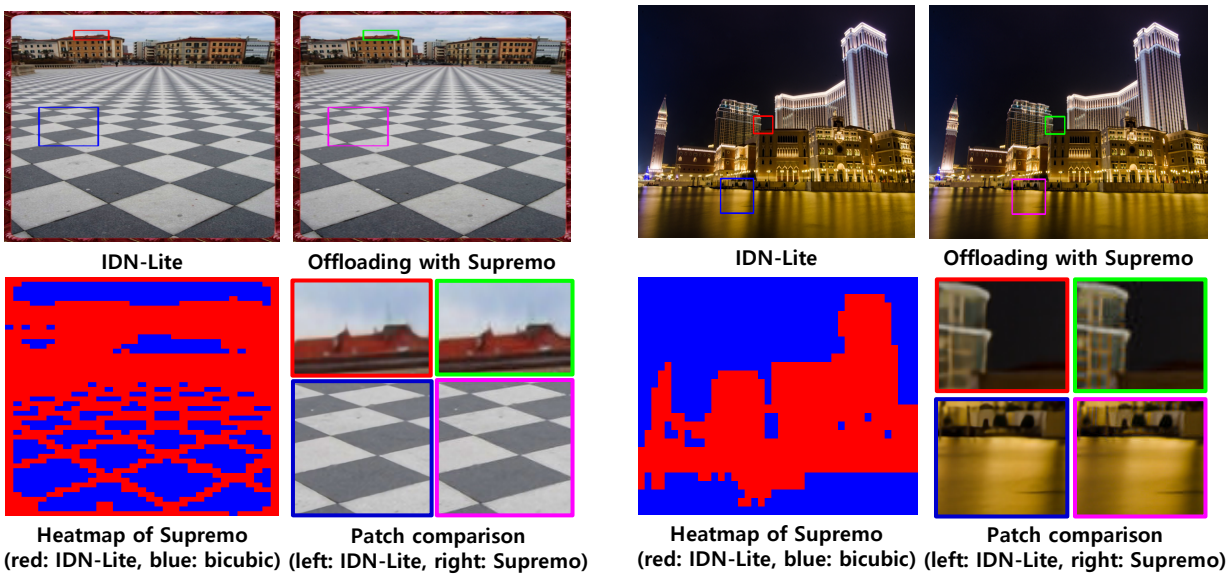
We implement both the client and the server of Supremo fully based on commodity hardware.

7.1 Client

We implement the client side of Supremo on Google Pixel 2 equipped with Qualcomm Kyro octa-core CPU (4×2.35 GHz, 4×1.9 GHz) running on Android 8.1.0. We use Camera2 API [36] to create a camera capture session. For each input frame, we extract edges using the Canny edge detector in the OpenCV for Android SDK version 3.4.1 [37]. Block matching algorithm in Caching Mechanism is implemented using Android NDK r17c [38]. For bicubic interpolation and inverse discrete cosine transform for Residual Decoding, we use the functions provided in the OpenCV as well.

The client is connected to the server via a TCP connection, as TCP guarantees the in-order-delivery of the edge blocks in Priority Ordering. For each block offloaded, a 2-dimensional index (i, j) is included as a header to indicate its location in the original frame. Assuming that the index of the block can be represented with 1 byte per dimension (i.e., the LR frame is divided into no more than 256×256 blocks), header transmission overhead is typically negligible (e.g., when block size is 8×8, the overhead becomes 3.125%).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2020.3025300, IEEE Transactions on Mobile Computing



(a) Urban100- img021. (b) Urban100-img085.

Fig. 13: Visual examples of the performance of Supremo. Top rows show the upscaled images by baseline full offloading (left) and Supremo (right), respectively. Bottom rows show the heatmaps of how Supremo applies different upscaling techniques on image blocks (left), and close-view patch comparison of the upscaled images (right). Notice that Supremo yields similar image quality compared to baseline full offloading, with minimal visual artifacts.

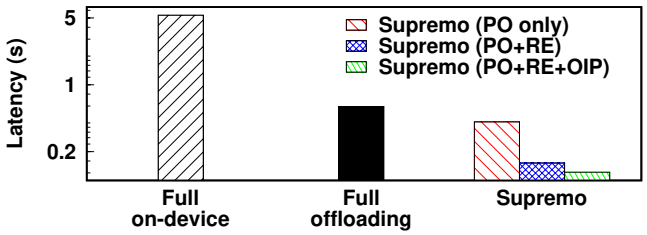


Fig. 14: End-to-end latency of Supremo compared to full on-device execution and baseline full offloading. PO, RE, and OIP denotes Priority Ordering, Residual Encoding, and Offload-Inference Pipelining, respectively.

7.2 Server

The server side of Supremo is implemented on a desktop PC running on Ubuntu 16.04 OS, equipped with Intel Core i7-8700 3.2 GHz CPU and a NVIDIA GTX 1080 Ti GPU. We implement most of the server side functions in Python 3.5.2 and utilize Numba [39], a Just-In-Time (JIT) compiler for Python, to accelerate the code execution speed comparable to C/C++. Upon receiving the LR frame from the client, the server upscales it with the DNN implemented using PyTorch 0.4.0 [40]. For discrete cosine transform in Residual Encoding, we use Python OpenCV 3.4.5 [41].

8 Evaluation

8.1 Experimental Setup

DNN Training. We train IDN-Lite using DIV2K dataset [42] composed of 900 2K resolution images. For training, we use randomly cropped 17×17 patches from the LR images (the size of the HR patches are determined accordingly by the upscale factor), with random horizontal flips and 90° rotations for data augmentation. We use the ADAM optimizer [43] for training, with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. Initial learning rate is set as 10^{-4} and halved after every 2×10^5 minibatch updates (with batch size 64) until it reaches 1.25×10^{-5}. The training terminates after 8×10^5 minibatch updates. We use four widely used benchmark datasets for validation: Set5 [44], Set14 [45], BSD100 [46], and Urban100 [18]. For fair comparison with the original IDN, we evaluate IDN-Lite trained in YCbCr color space with Y value range (16, 235). However, Y value range in YUV420 format employed in Android Camera2 API is (0, 255]). Accordingly, we train our model in corresponding range for subsequent evaluations. When trained in such range, the absolute PSNR value of the trained model (and also bicubic interpolation) decreases due to increased quantization error, but the gain remains almost the same as when trained in (16, 235) range.

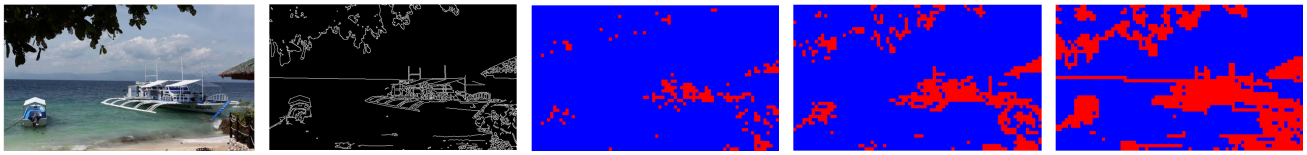
TABLE 2: PSNR/SSIM evaluation on Urban100 (trained on YUV420 color space with Y range [0, 255] used in the Android *Camera2* library.

Scheme	PSNR (dB)	SSIM
Bicubic interpolation	23.11	0.7370
Full offloading	26.00	0.8358
JPEG	24.96	0.7862
Supremo	25.45	0.8121

Evaluation Dataset. For repeatable evaluation Supremo, we obtain uncompressed 30 fps 1080p (1,920×1,080) videos from CDVL database [47] containing diverse real-world scenes to generate datasets of 50 images and 20 videos (7 seconds per video). We denote them as *CDVL-I* and

TABLE 3: PSNR (dB)/inference time (ms) of IDN-Lite compared with baselines.

Dataset (max. size)	Scale	IDN (31 layers)	IDN-Lite (15 layers)	IDN-D1 (10 layers)	PSNR recovery
Set5 (512×512)	×2	37.83/7.87	37.74/3.16	37.64/2.96	53%
	×3	34.11/4.36	33.95/2.22	33.70/2.01	61%
	×4	31.82/3.15	31.68/2.07	31.51/1.78	55%
Set14 (768×512)	×2	33.30/14.63	33.20/5.28	33.13/4.98	41%
	×3	29.99/7.13	29.90/2.84	29.70/2.60	69%
	×4	28.25/4.43	28.18/2.02	28.10/1.82	53%
BSD100 (480×321)	×2	32.08/9.63	31.97/3.45	31.90/3.28	39%
	×3	28.95/4.73	28.88/1.80	28.75/1.67	65%
	×4	27.41/2.92	27.34/1.21	27.29/1.10	42%
Urban100 (1,023×1,023)	×2	31.27/48.29	31.21/17.07	31.02/16.31	76%
	×3	27.42/22.82	27.36/7.95	27.04/7.60	84%
	×4	25.41/13.06	25.38/4.63	25.25/4.49	81%



(a) Raw frame. (b) Detected edges. (c) 5% offloaded. (d) 15% offloaded. (e) 30% offloaded.

Fig. 15: Example operation of Priority Ordering. Red blocks in the heatmaps in (c)-(e) indicates the offloaded image blocks.

CDVL-V, respectively.[3] Unless specified otherwise, evaluation assumes scale factor ×3 (640×360 to 1,920×1,080).

Topology. We evaluate the performance of the end-to-end system by connecting the mobile device to the desktop server through a Wi-Fi connection in 2.4 GHz band (IEEE 802.11n) with 90.2 Mbps network bandwidth.

8.2 Performance Overview

We first evaluate the overall performance of Supremo compared with baselines in terms of end-to-end offloading latency and image quality.

Latency. Fig. 14 shows the end-to-end latency of Supremo for upscaling 360p image to 1080p, compared with full on-device mobile GPU execution using TensorFlow-Lite and full offloading approach of sending the whole image to the cloud and receiving the upscaled HR counterpart. Indoor office scenes captured from the camera are fed as input, in which around 50% of the image blocks are background blocks with no edges (similar to the edge block ratios of the images in the CDVL-I dataset). Supremo only offloads the blocks that contain at least one edge pixel. Overall, adding on each of our techniques gradually reduces the latency. In total, Supremo upscales 360p image to 1080p in 122 ms, which is 43.68× and 4.85× faster compared to full on-device execution and full offloading, respectively. The performance gain of Supremo mainly comes from the fact that Supremo enables a tight cooperation between the mobile and the cloud to effectively balance the computational load and the network bandwidth consumption required for SR. In this sense, we expect that the performance of Supremo would be scalable across different network conditions (e.g., 4G LTE).

3. Datasets we use for evaluation can be found at https://www.dropbox.com/sh/03erqyagxounmj9/AADyeOIbfxKogxBM84S3fc9na?dl=0.

Image Quality. Qualitatively, Fig. 13 shows that Supremo preserves most of the sharp edges reconstructed by SR. Note that Supremo incurs minimal blocking artifacts as bicubic interpolation is applied only to regions with no edges as shown in the heatmap. Quantitatively, Table 2 shows that on Urban100, Supremo minimizes the loss of PSNR/SSIM compared to full offloading, whereas utilizing JPEG to achieve similar compression results in significant performance degradation.

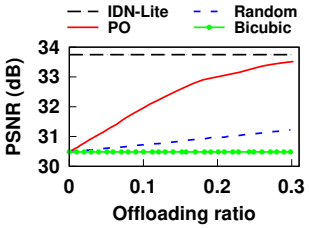
8.3 Performance of IDN-Lite

Next, we present detailed component-wise evaluations of Supremo. We first evaluate the performance of our proposed IDN-Lite. Table 3 compares the PSNR and inference time of IDN-Lite compared with with baseline IDN and IDN-D1, which is obtained by naively removing three DBlocks (i.e., leaving only one DBlock) from the original IDN. Scale $\times n$ indicates downsampling the original image by a factor of n and recovering the original resolution with the DNN. IDN-Lite is 2.48× faster than IDN (and comparable to IDN-D1), while its PSNR loss is around 0.1 dB. In terms of PSNR recovery (calculated as the ratio of PSNR loss of IDN-Lite over that of IDN-D1), IDN-Lite shows better performance on larger images, as it has deeper architecture than IDN-D1.

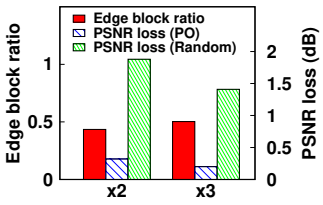
8.4 Data Compression Performance

Priority Ordering. Figure 15 shows an example operation of Priority Ordering. After Canny edge detector extracts edges from the frame (Figure 15(a)) as shown in Figure 15(b), blocks are offloaded in the order of edge intensity, indicated as red in Figure 15(c)–(e). Figure 16(a) shows that the PSNR of output frame obtained with such offloading mechanism quickly reaches the case when the whole frame is offloaded, with only 0.23 dB PSNR loss when all the edges in the frame are offloaded as in Figure 15(e). Compared with the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2020.3025300, IEEE Transactions on Mobile Computing

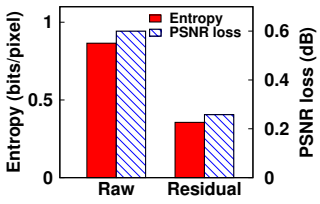


(a) Performance on Fig. 15(a).

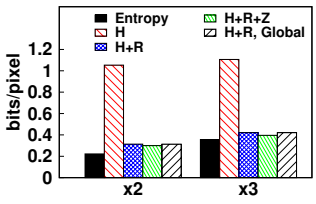


(b) Performance on CDVL-I.

Fig. 16: Performance of Priority Ordering.



(a) Impact of DCT and quantization.

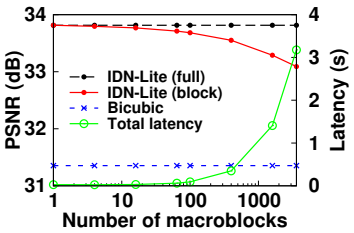


(b) Encoding performance of Residual Encoding.

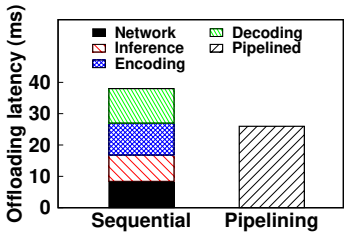
Fig. 17: Peformance of Residual Encoding.

TABLE 4: Compression performance comparison of Supremo and baselines on CDVL-I.

Approach	Total network bandwidth consumption (Mbits)				Average PSNR (dB)
	Up	Down	Total	Compression	
Bicubic	-	-	-	-	31.35
Baseline	87.89	791.02	878.91	1×	33.81
PO	30.95	278.57	309.52	2.84×	33.19
PO+RE (Supremo)	30.95	26.77	57.72	15.23×	33.00
JPEG	4.55	52.39	56.94	15.44×	30.61
JPEG w/ retrained IDN	4.55	52.39	56.94	15.44×	31.49



(a) Impact of macroblock size on performance.



(b) Offloading latency comparison.

Fig. 18: Performance of Offload-Inference Pipelining.

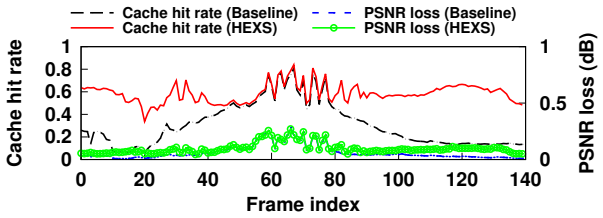


Fig. 19: Caching performance on CDVL-V video #0.

case of randomly selecting the blocks to offload (denoted as Random), we can see the effectiveness of Priority Ordering in determining which blocks should be offloaded with highest priority. Generalizing this to the CDVL-I dataset, Figure 16(b) shows that offloading only the edge blocks achieves 2.3× and 1.99× data compression with only 0.3 and 0.2 dB PSNR loss for ×2 and ×3 scale, respectively.

Residual Encoding. Figure 17 shows the performance of Residual Encoding on CDVL-I dataset. As shown in Figure 17(a), utilizing the residual signal not only yields 2.43× smaller entropy compared to the raw frame (thus enabling further compression), but also reduces PSNR loss due to quantization by 0.34 dB, verifying the effectiveness of Residual Encoding. Figure 17(b) compares the encoding performance of various combinations of encoding algorithms for ×2 and ×3 scale. While Huffman code (denoted as H) is the optimal prefix code, its performance is bounded by 1 bit per pixel, as it is a one-to-one mapping between symbol and code. Combining run-length encoding (denoted as R) breaks this bound and achieves near-entropy compression. Since quantized residual signals are very sparse, additional gain of zigzag scan (denoted as Z) is marginal, and therefore we do not employ it in our implementation for latency efficiency. Furthermore, the sparsity of residual signals allows a single global Huffman code generated from CDVL-I dataset to yield almost the same performance, eliminating the overhead of having to generate optimal code for each input frame. In summary, compared to sending raw HR frame which requires 8 bits per pixel, Residual Encoding achieves 25.53× and 18.99× compression for ×2 and ×3 scale, respectively.

Combined Performance. Table 4 shows the combined performance on CDVL-I for ×3 scale. For Priority Ordering, we only offload blocks with edge intensity over 0.15. As Priority Ordering and Residual Encoding are independent and complimentary to each other, combining them results in a synergetic gain in compression performance. Overall, combining the two achieves 15.23× compression with 0.81 dB PSNR loss (still 1.65 dB higher than bicubic interpolation), whereas achieving comparable compression with JPEG suffers from 3.20 dB PSNR loss. Notice that simply re-training IDN with HR and JPEG-compressed LR image pairs does not solve the problem (1.51 dB inferior to Supremo), as it is hard for a single IDN-Lite to learn both SR and JPEG artifact removal simultaneously.

8.5 Performance of Offload-Inference Pipelining

Impact of Macroblock Size on SR Latency. We first analyze the impact of the macroblock size on SR performance by dividing the images in CDVL-I dataset into different number of macroblocks and processing them separately. Figure 18(a) shows that dividing the image too finely (e.g., over 64 macroblocks) suffers from performance degradation

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2020.3025300, IEEE Transactions on Mobile Computing

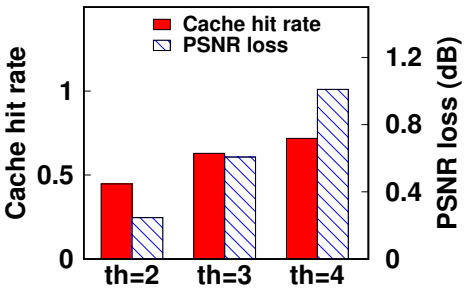
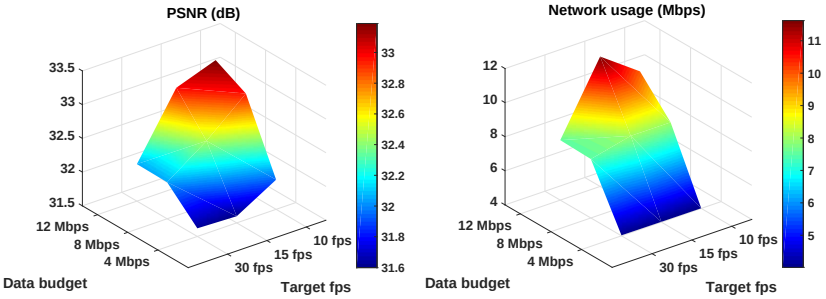


Fig. 20: Caching performance on CDVL-V.



(a) Average PSNR. (b) Network usage.

Fig. 21: Performance of Supremo-Opt under various network usage budget and frame rate constraints.

in terms of both PSNR and latency due to issues stated in Section 5.3, diminishing the potential benefit from pipelining. Closely analyzing the tradeoff, we set the number of macroblocks as 4 for subsequent evaluations. Note that the optimal value would depend on various factors including the image content and sizes, which we would like to further investigate in future to make our system more applicable in diverse settings.

End-to-End Latency. Figure 18(b) shows the gain of pipelining in offloading latency, defined as the total delay between the time the client starts sending frame to the server and finishes receiving and decoding residual signal (the frame merging latency is omitted as it is negligible compared to other steps). For fair comparison, we evaluate in a static office environment with 2.4 GHz Wi-Fi link (802.11n) and offload 150 8×8 blocks in a 320×240 image for ×4 scale. As pipelining enables parallel execution of functional components at different resources, offloading latency is reduced by 31.6%.

8.6 Performance of Supremo-Opt

Effectiveness of Caching Mechanism for SR. We first verify the effectiveness of Caching Mechanism applied for super-resolution. Figure 19 shows an example operation of the Caching Mechanism applied on CDVL-V video #0. While baseline approach of comparing the blocks in the same coordinate (as in [3], [5]) suffers from poor cache hit rate when motion is present between frames (frames 0 to 40 and 80 to 140), utilizing Hexagon Search (HEXS) block matching algorithm (as in [4], [10]) yields higher cache hit rate with minimal PSNR loss. Generalized on the CDVL-V dataset, Figure 20 shows the performance for different distance threshold for caching. While higher threshold yields higher cache hit rate (i.e., offload less), PSNR loss increases significantly as well. Based on experiments on diverse videos for various scale factors, we empirically set the threshold to 3 in the subsequent evaluation of Supremo-Opt.

Supremo-Opt Under Various Resource Constraints. We evaluate Supremo-Opt for various target frame rate and network usage budget. For comparative evaluation, we implement Supremo-Opt in Python and emulate its operation using an LTE TCP throughput trace measured with *Iperf* in a residential environment with Google Pixel 2 and the desktop server as client and server, respectively. The measured bandwidth is 17.93 Mbps on average, and we assume that the uplink and downlink equally shares the bandwidth. Figure 21 shows that in general, Supremo-Opt offloads more when latency requirement is lenient (i.e., target fps is low) and data budget is abundant, resulting in higher PSNR. As target frame rate increases, average PSNR decreases as data budget is split across more number of frames. In terms of network usage, Lyapunov optimization guarantees that Supremo-Opt does not consume more than the available budget, while under-utilization occurs when either i) the number of edge blocks to offload is smaller than data budget (e.g., target frame 10 fps, data budget 12 Mbps), or ii) network bandwidth is insufficient (e.g., target frame 30 fps, data budget 12 Mbps).

9 Discussion and Future Work

In this section, we discuss possible issues that can be considered to further improve Supremo, which we would like to deal with in our future work.

Integration with On-Device Deep Learning. Supremo can be incorporated with on-device inference systems to seek for further balance between the mobile and the cloud. Specifically, dynamically adjusting the number of edge blocks offloaded to the cloud and processed on the mobile depending on the network conditions would make Supremo more useful in diverse practical environments.

Energy Consumption. While we only consider network usage budget for the resource constraint in designing Supremo-Opt, energy consumption can also be considered similar to [32], [48]. Specifically, it can be done by monitoring the battery status and incorporating energy constraint in the Lyapunov optimization framework as a virtual queue to be stabilized, defined similarly as in Eq. (3). More fundamentally, Supremo-Opt can be extended to (i) profile the energy consumption of different wireless networks (e.g., Wi-Fi, LTE, 5G) as well as various upscaling algorithms (e.g., bicubic, linear, or nearest neighbor interpolation), and (ii) jointly determine the local upscaling algorithm, wireless network, and the amount of data offloaded to maximize the PSNR.

Potential Applications. We plan to improve Supremo to support higher resolutions (e.g., 2K and above), which requires further optimization in various components in the

system. We also envision that combining Supremo with other technologies (e.g., object detection, face recognition) will further diversify continuous mobile vision services. For example, Supremo can be used to enhance safety in self-driving vehicles by improving the resolution of distant objects that appear very small in the frames for accurate detection.

10 Related Work

Image Super-Resolution. Ever since SRCNN [20] took the first step of employing DNN for image super-resolution, various model architectures have been actively studied in recent years [1], [15], [25], [26], [28], [30], [49]. Supremo enables low-latency super-resolution on mobile devices through cloud offloading.

Continuous Mobile Vision. LiKamWa *et al.* [50] optimize energy consumption of image sensors for continuous mobile vision, while Starfish [51] efficiently supports concurrent vision applications. Supremo is in line with prior work on continuous mobile vision, but has a clear distinction in that it considers low-latency super-resolution which has not been investigated yet.

On-Device Deep Learning Inference. Several studies have tackled the challenge of running DNNs on mobile devices by compressing the model size [52], [53], [54], accelerating the inference speed [2], [3], [4], [55], [56], [57], [58], [59], and resource-aware model size adaptation [60], [61], [62]. However, on-device deep learning compromises with either low frame rate (e.g., 1 fps [3]) or small input image size (e.g., 128x128 [54]) due to lack of resources. Recently, MobiSR [63] utilized heterogeneous mobile processors to accelerate on-device super-resolution, but it takes more than 750 ms to upscale the 320×180 input image to 1280×720. MobiSR can be combined with Supremo to enable further collaboration between the mobile and the cloud (e.g., by allowing the mobile-side to apply super-resolution on some of the edge blocks detected in Priority Ordering using mobile GPU) to minimize end-to-end latency.

Offloading for Mobile Vision. MAUI [64] proposes an energy-aware offloading decision framework, while Gabriel [65] uses cloudlets for offloading in wearable devices. OverLay [7] utilizes offloading for mobile AR. VisualPrint [66] offloads extracted features rather than raw image to reduce network usage. Glimpse [5] enables real-time object tracking by offloading only trigger frames for detection and tracking them in local device. Marvel [8] and Liu *et al.* [10] design real-time detection and tracking system for mobile AR via offloading. Marvel [8] utilizes inertial sensors for tracking and occasionally offloads the images to calibrate tracking errors, while Liu *et al.* [10] employ a suite of techniques built on video encoding. EagleEye [67] utilizes the cloud server for executing multi-DNN face identification pipeline at low latency. MCDNN [6] determines whether to process a frame in local or offload based on energy and network usage budget, but does not take into account network conditions. Perhaps the most similar approach to Supremo is DeepDecision [9], which elaborates MCDNN by considering network conditions as well in the optimization framework. However, it cannot be employed for Supremo, as it operates in 1 fps scale, and more importantly, optimizes network bandwidth consumption by adapting the frame resolution using video encoding, whose operation directly contradicts with super-resolution. Recently, GRACE [68] customized the JPEG quantization matrix coefficients to minimize the performance loss of DNNs for image segmentation and classification in the offloading process. GRACE can be incorporated in Supremo to further improve compression performance in Residual Encoding.

11 Conclusion

We presented Supremo, a cloud-assisted system for low-latency SR in mobile devices. Supremo incorporates various techniques ranging from DNN optimization, specialized data compression algorithms, and implementation technique to optimize latency. Supremo achieves 43.68× faster latency than on-device mobile GPU execution, and reduces network bandwidth consumption by 15.23× and end-to-end latency by 4.85× with 0.81 dB PSNR loss compared to baseline offloading, whereas JPEG with similar compression suffers from 3.20 dB loss.

References

[1] Z. Zheng Hui, X. Wang, and X. Gao, "Fast and accurate single image super-resolution via information distillation network," in *Proc. IEEE CVPR*, 2018.

[2] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *Proc. ACM/IEEE IPSN*, 2016.

[3] L. N. Huynh, Y. Lee, and R. K. Balan, "DeepMon: Mobile GPU-based deep learning framework for continuous vision applications," in *Proc. ACM MobiSys*, 2017.

[4] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu, "DeepCache: Principled cache for mobile deep vision," in *Proc. ACM MobiCom*, 2018.

[5] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, "Glimpse: Continuous, real-time object recognition onmobile devices," in *Proc. ACM SenSys*, 2015.

[6] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "MCDNN:an approximation-based execution framework for deep stream processing under resource constraints," in *Proc. ACM MobiSys*, 2016.

[7] P. Jain, J. Manweiler, and R. R. Choudhury, "OverLay: Practical mobile augmented reality," in *Proc. ACM MobiSys*, 2015.

[8] K. Chen, T. Li, H.-S. Kim, D. E. Culler, and R. H. Katz, "MARVEL: Enabling mobile augmented reality with low energy and low latency," in *Proc. ACM SenSys*, 2018.

[9] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen, "DeepDecision: A mobile deep learning framework for edge video analytics," in *Proc. IEEE INFOCOM*, 2018.

[10] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *Proc. ACM MobiCom*, 2019.

[11] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Proc. NIPS*, 2015.

[12] M. Neely, *Stochastic network optimization with application to communication and queueing systems*. Morgan & Claypool Publishers, 2010.

[13] G. Huang, S. Liu, L. v. d. Maaten, and K. Q. Weinberger, "CondenseNet: An efficient densenet using learned group convolutions," in *arXiv preprint arXiv:1711.09224*, 2017.

[14] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE CVPR*, 2018.

[15] N. Ahn, B. Kang, and K.-A. Sohn, "Fast, accurate, and lightweight super-resolution with cascading residual network," in *Proc. ECCV*, 2018.

[16] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, and *et al.*, "ImageNet large scale visual recognition challenge," in *International Journal of Computer Vision*, 2014.

[17] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollar, and C. L. Zitnick, "Microsoft COCO: common objects in context," in *Proc. ECCV*, 2014.

[18] J.-B. Huang, A. Singh, and N. Ahuja, "Single image super-resolution from transformed self-exemplars," in *Proc. IEEE CVPR*, 2015.

[19] A. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," in *arXiv preprint arXiv:1704.04861*, 2017.

[20] C. Dong, C. C. Loy, K. He, and X. Tang, "Learning a deep convolutional network for image super-resolution," in *Proc. ECCV*, 2014.

[21] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *arXiv preprint arXiv:1409.1556*, 2014.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE CVPR*, 2016.

[23] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg., "SSD: Single shot multibox detector," in *Proc. ECCV*, 2016.

[24] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," in *Proc. IEEE CVPR*, 2017.

[25] J. Kim, K. Lee, and K. Lee, "Deeply-recursive convolutional network forimage super-resolution," in *Proc. IEEE CVPR*, 2016.

[26] Y. Tai, J. Yang, and X. Liu, "Image super-resolution via deep recursive residual network," in *Proc. IEEE CVPR*, 2017.

[27] Y. Tai, J. Yang, X. Liu, and C. Xu, "MemNet: A persistent memory network for image restoration," in *Proc. ICCV*, 2017.

[28] J. Kim, J. Lee, and K. Lee, "Accurate image super resolution using very deep convolutional networks," in *Proc. IEEE CVPR*, 2016.

[29] "Cisco Visual Networking Index: Forecast and Trends, 2017–2022," https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html, Nov. 2018.

[30] W. Shi, J. Caballero, F. Huszar, J. Totz, A. P. Aitken, R. Bishop, D. Ruecker, and Z. Wang, "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network," in *Proc. IEEE CVPR*, 2016.

[31] F. Bossen, B. Bross, K. Suhring, and D. Flynn, "Hevc complexity and implementation analysis," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1685–1696, 2012.

[32] J. Koo, J. Yi, J. Kim, M. Hoque, and S. Choi, "REQUEST: Seamless dynamic adaptive streaming over http for multi-homed smartphone under resource constraints," in *Proc. ACM Multimedia*, 2017.

[33] K. Spiteri, R. Urgaonkar, and R. K. Sitaraman, "BOLA: near-optimal bitrate adaptation for online videos," in *Proc. IEEE INFOCOM*, 2016.

[34] J. Kim, G. Caire, and A. F. Molisch, "Quality-aware streaming and scheduling for device-to-device video delivery," in *IEEE/ACM Transactions on Networking*, 2016.

[35] C. Zhu, X. Lin, and L.-P. Chau, "Hexagon-based search pattern for fast block motion estimation," in *IEEE Transactions on Circuits and Systems for Video Technology*, 2002.

[36] "Android Camera2 API," https://developer.android.com/reference/android/hardware/camera2/package-summary/. Accessed: 11 Jul. 2020.

[37] "OpenCV for Android SDK," https://opencv.org/platforms/android/. Accessed: 11 Jul. 2020.

[38] "Android NDK," https://developer.android.com/ndk/guides/?hl=en. Accessed: 11 Jul. 2020.

[39] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: a llvm-based python jit compiler," in *Proc. ACM LLVM*, 2015.

[40] "PyTorch," https://pytorch.org/. Accessed: 11 Jul. 2020.

[41] "OpenCV for Python." https://pypi.org/project/opencv-python/. Accessed: 11 Jul. 2020.

[42] R. Timofte, E. Agustsson, L. V. Gool, M.-H. Yang, and L. *et al.*. Zhang, "NTIRE 2017 challenge on single image super-resolution: methods and results," in *Proc. IEEE CVPR Workshops*, 2017.

[43] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[44] M. Bevilacqua, A. Roumy, C. Guillemot, and M. L. Alberi-Morel, "Low-complexity single-image super-resolution basedon nonnegative neighbor embedding," in *BMVC*, 2012.

[45] R. Zeyde, M. Elad, and M. Protter, "On single image scale-up using sparse-representations," in *Curves and Surfaces*, 2010.

[46] D. Martin, C. Fowlkes, D. Tal, and J. Malik, "A database of human segmented natural images and its application to e-valuating segmentation algorithms and measuring ecological statistics," in *Proc. IEEE CVPR*, 2011.

[47] "Consumer Digital Video Library," https://www.cdvl.org/. Accessed: 11 Jul. 2020.

[48] J. Koo, J. Yi, J. Kim, M. Hoque, and S. Choi, "Seamless dynamic adaptive streaming in lte/wi-fi integrated network under smartphone resource constraints," in *IEEE Transactions on Mobile Computing*, 2019.

[49] B. Lim, S. Son, H. Kim, S. Nah, and K. M. Lee, "Enhanced deep residual networks for single image super-resolution," in *Proc. IEEE CVPR Workshops*, 2017.

[50] R. LiKamWa, B. Priyantha, M. Philipose, L. Zhong, and P. Bahl, "Energy characterization and optimization of image sensing toward continuous mobile vision," in *Proc. ACM MobiSys*, 2013.

[51] R. LiKamWa and L. Zhong, "Starfish: Efficient concurrency support for computer vision applications," in *Proc. ACM MobiSys*, 2015.

[52] N. D. Lane, P. Georgiev, and L. Qendro, "DeepEar: Robust smartphone audio sensing in unconstrained acoustic environments using deep learning," in *Proc. ACM UbiComp*, 2015.

[53] X. Zeng, K. Cao, and M. Zhang, "MobileDeepPill: A small-footprint mobile deep learning system for recognizing unconstrained pill images," in *Proc. ACM MobiSys*, 2017.

[54] B. Zhu, Y. Chen, J. Wang, S. Liu, B. Zhang, and M. Tang, "Fast deep matting for portrait animation on mobile phone," in *Proc. ACM Multimedia Conference*, 2017.

[55] S. S. L. Oskouei, H. Golestani, M. Hashemi, and S. Ghiasi, "CNNDroid: GPU-accelerated execution of trained deep convolutional neural networks on android," in *Proc. ACM Multimedia Conference*, 2016.

[56] S. Bhattacharya and N. D. Lane, "Sparsifying deep learning layers for constrained resource inference on wearables," in *Proc. ACM SenSys*, 2016.

[57] M. Alzantot, Y. Wang, Z. Ren, and M. B. Srivastava, "RSTensorFlow: GPU enabled tensorflow for deeplearning on commodity android devices," in *Proc. ACM 1st International Workshop on Embedded and Mobile Deep Learning (EMDL)*, 2017.

[58] Q. Cao, N. Balasubramanian, and A. Balasubramanian, "MobiRNN: Efficient recurrent neural network executionon mobile GPU," in *Proc. ACM 1st International Workshop on Embedded and Mobile Deep Learning (EMDL)*, 2017.

[59] A. Mathur, N. D. Lane, S. Bhattacharya, A. Boran, C. Forlivesi, and F. Kawsar, "DeepEye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware," in *Proc. ACM MobiSys*, 2017.

[60] S. Yao, Y. Zhao, S. Huajie, S. Liu, D. Liu, L. Su, and T. Abdelzaher, "FastDeepIoT: Towards understanding and optimizing neural network execution time on mobile and embedded devices," in *Proc. ACM SenSys*, 2018.

[61] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," in *Proc. ACM MobiSys*, 2018.

[62] B. Fang, X. Zeng, and M. Zhang, "NestDNN: Resource-aware multi-tenant on-device deep learning for continuous mobile vision," in *Proc. ACM MobiCom*, 2018.

[63] R. Lee, S. I. Venieris, L. Dudziak, S. Bhattacharya, and N. D. Lane, "MobiSR: Efficient on-device super-resolution through heterogeneous mobile processors," in *Proc. ACM MobiCom*, 2019, pp. 1–16.

[64] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *Proc. ACM MobiSys*, 2010.

[65] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Proc. ACM MobiSys*, 2014.

[66] P. Jain, J. Manweiler, and R. R. Choudhury, "Low bandwidth offload for mobile AR," in *Proc. ACM CoNEXT*, 2016.

[67] J. Yi, S. Choi, and Y. Lee, "EagleEye: Wearable camera-based person identification in crowded urban spaces," in *Proc. ACM MobiCom*, 2020.

[68] X. Xie and K.-H. Kim, "Source compression with bounded DNN perception loss for IoT edge computer vision," in *Proc. ACM MobiCom*, 2019.

Juheon Yi received his B.S and M.S. degree in Electrical Engineering from Seoul National University (SNU), Seoul, Korea in 2016 and 2018, respectively. He is currently with the Institute of Computer Technology, Seoul National University. His research interests include mobile/embedded deep learning systems, Augmented Reality (AR), and wireless networking.

Seongwon Kim is a Research Engineer at T-Brain, SK Telecom, Seoul, Korea. Before joining SK Telecom, he was a Post-Doctoral Researcher with Seoul National University (SNU), Seoul, Korea, from Oct. 2017 to Feb. 2019. He was also a Visiting Research Scholar at the Department of Electrical and Computer Engineering, Rice University, USA, from July 2017 to Oct. 2017. He received the B.S. degree in Electrical Engineering from the Pohang University of Science and Technology (POSTECH) in 2011, and the M.S. and Ph.D. degrees in Electrical and Computer Engineering from SNU in 2013 and 2017, respectively. His current research interests include multimodal AI, IoT connectivity, and next-generation wireless networks.

Joongheon Kim (M'06–SM'18) is currently an assistant professor of electrical engineering with Korea University, Seoul, Korea. He received his B.S. (2004) and M.S. (2006) in computer science and engineering from Korea University, Seoul, Korea; and his Ph.D. (2014) in computer science from the University of Southern California (USC), Los Angeles, CA, USA.

Before joining Korea University as an assistant professor in 2019, he was with LG Electronics Seocho R&D Campus as a research engineer (Seoul, Korea, 2006–2009), InterDigital as an intern (San Diego, CA, USA, 2012), Intel Corporation as a systems engineer (Santa Clara in Silicon Valley Area, CA, USA, 2013–2016), and Chung-Ang University as an assistant professor of computer science and engineering (Seoul, Korea, 2016–2019).

He is a senior member of the IEEE. He was a recipient of the Annenberg Graduate Fellowship with his Ph.D. admission from USC (2009), Intel Corporation Next Generation and Standards (NGS) Division Recognition Award (2015), KICS Haedong Young Scholar Award (2018), IEEE Vehicular Technology Society (VTS) Seoul Chapter Award (2019), KICS Outstanding Contribution Award (2019), Gold Prize from IEEE Seoul Section Student Paper Contest (2019), and IEEE Systems Journal Best Paper Award (2020).

Sunghyun Choi (S'96–M'00–SM'05–F'14) is a Senior Vice President and Head of the Advanced Communications Research Center at Samsung Research, Samsung Electronics, Seoul, Korea. He was a professor at the Department of Electrical and Computer Engineering, Seoul National University (SNU), Seoul, Korea from Sept. 2002 to Aug. 2019, and served as a Vice Dean for Academic Affairs, College of Engineering during the last two years at SNU. Before joining SNU, he was with Philips Research USA, Briarcliff Manor, New York, USA, as a Senior Member Research Staff for three years. He was also a visiting associate professor at the Electrical Engineering department, Stanford University, USA, from June 2009 to June 2010. He received his B.S. (summa cum laude) and M.S. degrees in electrical engineering from Korea Advanced Institute of Science and Technology (KAIST) in 1992 and 1994, respectively, and received Ph.D. at the Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor in September 1999.

He is currently heading researches and standardization for 6G, B5G, and IoT connectivity at Samsung Research. He co-authored over 250 technical papers and a book "Broadband Wireless Access and Local Networks: Mobile WiMAX and WiFi," Artech House, 2008 (with B. G. Lee). He holds over 160 patents, and numerous patents pending. He has served as a General Co-Chair of COMSWARE 2008, a Program Committee Co-Chair of IEEE WCNC 2020, IEEE DySPAN 2018, ACM Multimedia 2007, and IEEE WoWMoM 2007. He has also served on program and organization committees of numerous leading wireless and networking conferences including ACM MobiCom, IEEE INFOCOM, IEEE SECON, and IEEE WoWMoM. He has served as a division editor of Journal of Communications and Networks, and as an editor of IEEE Transactions on Wireless Communications, IEEE Transactions on Mobile Computing, IEEE Wireless Communications, ACM SIGMOBILE Mobile Computing and Communications Review, Computer Networks, and Computer Communications. He has served as a guest editor for IEEE Journal on Selected Areas in Communications, IEEE Wireless Communications, IEEE Transactions on Cognitive Communications and Networking, and ACM Wireless Networks. From 2000 to 2007, he was an active contributor to IEEE 802.11 WLAN Working Group.

He has received a number of awards including the Presidential Young Scientist Award (2008); IEEK/IEEE Joint Award for Young IT Engineer (2007); KICS Dr. Irwin Jacobs Award (2013); Shinyang Scholarship Award (2011); the Outstanding Research Award (2008) and the Best Teaching Award (2006) both from the College of Engineering, SNU; and the Best Paper Award from IEEE WoWMoM 2008. He is an IEEE Fellow.