Appendix

A. Compiler Optimization Details

We provide more details of our compiler optimizations in this section. Different from prior DNN inference acceleration frameworks [1, 2, 3, 8, 18, 17] that only only support dense models or pattern-based pruned models, our compiler optimizations are general, support both dense (unpruned) model and sparse (pruned) model with different pruning schemes for fast inference on various mobile platforms. The optimizations include 1) a layer fusion mechanism to fuse different layers together for the reduction of memory consumption of intermediate results and number of operators; 2) the supports for sparse models with different pruning schemes; 3) an auto-tuning process to determine the best-suited configurations of parameters for different mobile CPUs/GPUs; 4) Domain Specific Language (DSL) based code generation.

A.1. Layer Fusion Mechanism

To reduce the inference latency effectively for dense (unpruned) models, a layer fusion technique is incorporated in our compiler optimization to fuse the computation operators in the computation graph. With layer fusion, both the memory consumption of the intermediate results and the number of operators can be reduced. The fusion candidates in a model are identified based on two kinds of polynomial calculation properties, i.e., compression laws and data access patterns. The compression laws include associative property, communicative property, and distributive property.

However, looking for the fusion candidates in such a large space of all combinations of computation operations is too expensive. Therefore, we introduce two constraints to guide the looking up process: 1) only explore the opportunities that are specifically provided due to the above properties, and 2) only consider enlarging the overall computation for CPU/GPU utilization improvement and reducing the memory access for memory performance improvement as the cost metrics in the fusion. Compared with prior works on loop fusion [4, 5, 6], our method is more aggressive without high exploration cost.

A.2. Supports for Different Pruning Schemes

Different from other DNN inference frameworks, our framework also supports sparse model accelerations with different pruning schemes including unstructured pruning, coarse-grained structured pruning, pattern-based pruning, and block-based pruning. Note that fine-grained unstructured pruning and coarse-grained structured pruning can be viewed as two extreme cases of block-based pruning by adopting 1×1 and the whole weight matrix size as the block size, respectively. Thus, accelerating block-based pruned model also indicates the inference speedup for unstructured pruned and the traditional structured pruned models. For the sparse (pruned) model, the framework first compacts the model storage with a novel compression format called Blocked Compressed Storage (**BCS**) format, as shown in Figure A1, and then performs computation reordering to reduce the branches within each thread and eliminate the load imbalance among threads.

BCS stores non-zero weights as Compressed Sparse Row format (CSR) with an even better compression rate by further compressing the index with a hierarchical structure. Traditional CSR has to store each non-zero weight with an explicit column index. Take block-based pruning as an example, it preserves non-zero weights in identical columns in each block, inducing many repeated column indices if we use CSR. BCS eliminates this redundancy with a hierarchical compression on the column index only.

For block-based pruning, it partitions the weight matrix of a whole layer into blocks with different pruning configurations. Without any further optimization, it will encounter the well-known challenges for sparse matrix multiplications, i.e., heavy control-flows within each thread, load imbalance among multiple threads, and irregular memory access. To address this issue, a row reordering optimization is also included to further improve the regularity of the weight matrix. After this reordering, the continuous rows with identical or similar numbers of non-zero weights are processed by multi-threads simultaneously, thus eliminating thread divergence and achieving load balance.

Figure A1 shows a simplified example. `Weights array` stores all non-zero weights. `Compact column array` stores the compressed column index, e.g., [0, 3, 7] denotes the column id of the first three weights [2, 3, 4]. `Column stride array` denotes the starting and ending index of each row in compact column array, e.g., [0, 3] denotes that the column index for the first row starts from index 0 and ends at index 2 in compact column array. The same column indices may be used for multiple rows. `Occurrence array` is used to specify the starting and ending rows with the identical column index, e.g., [0, 2] means that row 0 and 1 share the same column index. BCS also contains a `row offset array` to specify the starting location of each row in weight array.

Usually, the weight distribution is not as regular as the above simplified example, thus, a row reordering optimization is also included to further improve the regularity of the weight matrix. After this reordering, the continuous rows with identical or similar numbers of non-zero weights are processed by multi-threads simultaneously, thus eliminating thread divergence and achieving load balance. Each thread processes more than one rows, thus eliminating branches and improving instruction-level parallelism. Moreover, a similar optimization flow (i.e., model compaction and computation reorder and other optimizations) is employed to

support all compiler optimizations for pattern-based pruning as PatDNN [18].

A.3. Auto-Tuning for Different Mobile CPUs/GPUs

During DNN execution, there are many tuning parameters, e.g., matrix tiling sizes, loop unrolling factors, and data placement on GPU memory, that influence the performance. It is hard to determine the best-suited configuration of these parameters manually. To alleviate this problem, our compiler incorporates an auto-tuning approach for both sparse (pruned) model and dense (unpruned) model. The Genetic Algorithm is leveraged to explore the best-suited configurations automatically. It starts parameter search after an initialization with an arbitrary number of chromosomes and explores the parallelism better. Acceleration codes for different DNN models and different mobile CPUs/GPUs can be generated efficiently and quickly through this auto-tuning process, providing the foundation for fast end-to-end inference. The auto-tuning optimizations together with the layer-fusion optimizations make our framework outperform other acceleration frameworks.

A.4. DSL based Code Generation

In deep learning, a computational graph of a DNN model can be represented by a directed acyclic graph (DAG). Each node in this graph corresponds to an operator. We propose a high-level Domain Specific Language (DSL) to specify such kind of operators. Each operator in a computational graph also with a layerwise Intermediate Representation (IR) which contains BCS pruning information. The input and output are different tensors in terms of different shapes. This DSL also provides a `Tensor` function for users to create matrices (or tensors).

In this way, DSL is equivalent to a computational graph (that is, DSL is another type of high-level functions used to simulate the data flow of the DNN model), and they can be easily converted to each other. DSL provides users with the flexibility to use existing DNNs or create new DNNs, improving the productivity of DNN programming. If the DNN already exists, we will convert it into an optimized calculation graph and convert this graph into a DSL. Otherwise, the user writes the model code in the DSL, converts it back to a calculation graph, performs advanced optimization, and regenerates the optimized DSL code.

Finally, our compiler translates the DSL into low-level C++ code for mobile CPU and OpenCL code for mobile GPU, and optimizes the low-level code through a set of optimizations enabled by BCS pruning. The generated code can be then deployed on the mobile device.

B. Reweighted ℓ_1 Algorithm

Prior weight pruning algorithms such as using the group Lasso regularization [23, 12, 16] or Alternating Direction

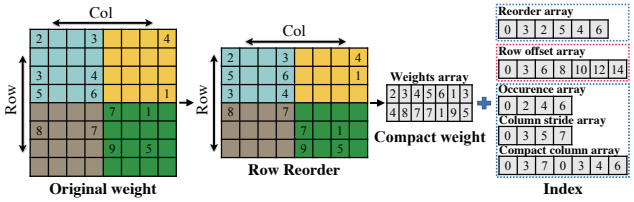

Figure A1. Matrix Reorder and Blocked Compressed Storage (BCS) for weights.

Methods of Multipliers (ADMM) [25, 19, 14] either suffer from potential accuracy loss or require manually compression rate tuning. To overcome the limitations, we leverage reweighted group Lasso [7] method to discover the sparsity with automatically determined pruning ratio. The basic idea is to systematically and dynamically reweight the penalties. More specifically, the reweighted method reduces the penalties on weights with larger magnitudes, which are likely to be more critical weights, and increases the penalties on weights with smaller magnitudes.

We formulate the general reweighted pruning problem as below

$$\underset{\boldsymbol{W}}{\text{minimize}} \quad f\left(\boldsymbol{W};\mathcal{D}\right)+\lambda\sum_{l=1}^{L}R(\boldsymbol{\alpha}^l,\boldsymbol{W}^l), \tag{A1}$$

where λ is the hyperparameter to adjust the relative importance between accuracy and sparsity. $\mathcal{D}$ stands for the dataset. $\boldsymbol{W}^l$ denotes the weight matrix for the l-th layer and $\boldsymbol{W} := \{\boldsymbol{W}^l\}_{l=1}^{L}$. Let $\boldsymbol{\alpha}^l$ represent the collection of penalty values that applied on the l-th layer weights. Note that each element in $\boldsymbol{\alpha}^l$ is positive.

The regularization term $R(\boldsymbol{\alpha}^l,\boldsymbol{W}^l)$ denotes the penalties on the weights for the l-th layer. The method can be applied to models with different pruning schemes for each layer. For block-based pruning, each $\boldsymbol{W}^l$ is divided into J blocks with the same size $p^l \times q^l$, namely, $\boldsymbol{W}^l = [\boldsymbol{W}_1^l, \boldsymbol{W}_2^l, ..., \boldsymbol{W}_J^l]$, and the regularization term for block-based column pruning is defined as

$$R(\boldsymbol{\alpha}^l,\boldsymbol{W}^l)=\sum_{j=1}^{J}\sum_{n=1}^{q^l}\left\|\alpha_{jn}^l\cdot[\boldsymbol{W}_j^l]_{:,n}\right\|_F^2, \tag{A2}$$

where $[\boldsymbol{W}_j^l]_{:,n}$ is the n-th column of $\boldsymbol{W}_j^l$ and α_{jn}^l is updated by $\frac{1}{\|[\boldsymbol{W}_j^l]_{:,n}\|_F^2+\epsilon}$ to help increase the degree of sparsity. ϵ is a small value to avoid zero denominator. From the equation we can see that small $\|[\boldsymbol{W}_j^l]_{:,n}\|_F^2$ leads to a large penalty α_{jn}^l, thus is more likely to be pruned.

Similarly, the regularization term for block-based row pruning is defined as

$$R(\boldsymbol{\alpha}^l,\boldsymbol{W}^l)=\sum_{j=1}^{J}\sum_{m=1}^{p^l}\left\|\alpha_{jm}^l\cdot[\boldsymbol{W}_j^l]_{m,:}\right\|_F^2, \tag{A3}$$

where $[\boldsymbol{W}_j^l]_{m,:}$ represents the m-th row of $\boldsymbol{W}_j^l$ and α_{jm}^l is updated by $\frac{1}{\|[\boldsymbol{W}_j^l]_{m,:}\|_F^2+\epsilon}$. For coarse-grained structured pruning, it can be viewed as the special case of block-based pruning by setting $J = 1$ and $p^l \times q^l$ as the original weight matrix size of layer l in equation (A2) and equation (A3).

As for pattern-based pruning, as it acts on the kernel levels and suits tensor-based computation better, we formulate it with tensor representations. We represent the weight tensor for the l-th layer as $\mathcal{W}^l \in \mathbb{R}^{N^l \times C^l \times K_h^l \times K_w^l}$, where N^l, C^l, K_h^l, K_w^l represent the number of filters, the number of channels, kernel height and kernel width for the l-th layer, respectively. We first apply a kernel pattern from a predefined kernel pattern library to each 3×3 kernel in the model, resulting in weight tensor $\mathcal{W}'^l$ for $l = 1, \cdots, L$. We further apply connectivity pruning and the regularization term is defined as

$$R(\boldsymbol{\alpha}^l, \mathcal{W}'^l) = \sum_{n=1}^{N^l} \sum_{m=1}^{C^l} \left\| \alpha_{nm}^l \cdot [\mathcal{W}'^l]_{n,m,:,:} \right\|_F^2, \tag{A4}$$

where $[\mathcal{W}'^l]_{n,m,:,:}$ stands for the kernel that connects the m-th input channel with the n-th output channel, and α_{nm}^l is updated by $\frac{1}{\|[\mathcal{W}'^l]_{n,m,:,:}\|_F^2+\epsilon}$.

In each iteration of the prune ratio determination, we solve problem (A1) with certain epochs of training. Then with the obtained $\boldsymbol{W}$, we can update $\boldsymbol{\alpha}$. Thus in the next iteration, we again solve problem (A1) with updated $\boldsymbol{\alpha}$. After iterations, we can obtain the sparse weights without human intervention. We see that the reweighted method only requires the hyperparameter λ and the soft constraints formulation allows the automatic determination of the prune ratio for each layer.

C. Comparison with State-of-the-Art for $\times 3$ Scaling Task

Besides the results for $\times 2$ and $\times 4$ upscaling task, we further compare our models on $\times 3$ upscaling with state-of-the-art efficient SR models. As shown in Table A1, for the $\times 3$ upscaling task, our model obtained with a target latency $t = 150$ms reaches higher PSNR/SSIM than SRCNN and FSRCNN with comparable or even fewer MACs. With $t = 290$ms, our model provides better PSNR/SSIM compared with CARN-M with $3.7\times$ MACs reduction. Compared with ESRN-V, EDSR, and WDSR, competitive image quality in terms of PSNR/SSIM can be obtained by our model with much fewer MACs. By setting $t = 50$ms, our model reaches real-time inference while keeping fairly good PSNR/SSIM.

D. Visual Comparison with Other SR methods

In this section, we include more visual comparisons with other SR models on $\times 4$ upscaling task, as shown in Figure A2. The low-resolution images are img_091 and img_013 from Urban100. As observed, high-resolution images generated by our models demonstrate undetectable visual difference compared with the baseline WDSR while greatly saving the parameters and MACs.

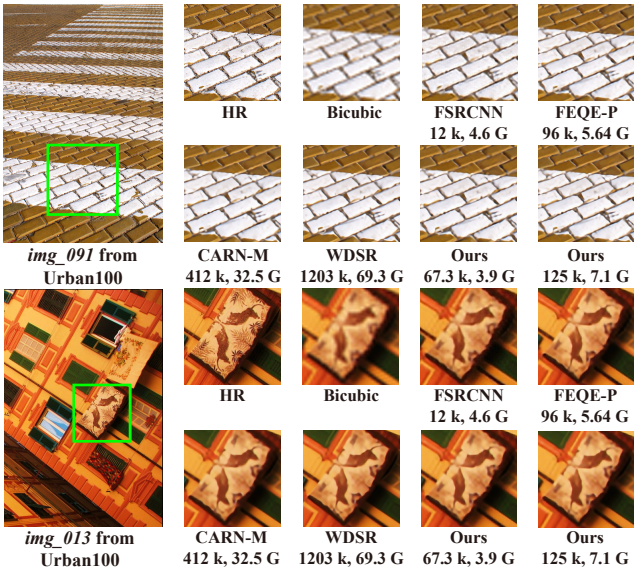

Figure A2. Visual comparison with other SR models on $\times 4$ scale. Model parameters and MACs are listed under model name.

E. Ablation Study

We compare our method with other SR models in terms of FPS and PSNR. For a fair comparison, we implement our derived models and other baselines with MNN (not support sparse models for further inference accelerations) on mobile CPU as the baselines. We want to promote reproducibility and evaluate speedup using the same framework to show the generalization of our searched results. Note that we modify the sparse models derived by our method by filling each pruned weight with a zero value as MNN does not support sparse models for further speedups. In this way, the models being dealt with MNN are dense models with a bunch of zero-value weights.

As shown in Figure A3, compared with CARN-M [13] and FSRCNN [10], our method with large t can result in higher FPS and PSNR. The derived models with smaller t have slight PSNR degradation with significant FPS improvements. Note that our model with compiler optimization can satisfy the real-time requirement (such as $t = 50$ms). And our derived model implemented with MNN still maintain fairly good results (such as 10.8FPS for the $t = 50$ms searched result on the $\times 3$ upscaling task) as it demonstrated in Figure A3.

Scale	Model	Params (K)	Multi-Adds (G)	Set5 (PSNR/SSIM)	Set14 (PSNR/SSIM)	B100 (PSNR/SSIM)	Urban100 (PSNR/SSIM)
	SRCNN [9]	57	52.7	32.75/0.9090	29.28/0.8209	28.41/0.7863	26.24/0.7989
	FSRCNN [10]	12	5.0	33.16/0.9140	29.43/0.8242	28.53/0.7910	26.43/0.8080
	CARN-M [13]	412	46.1	33.99/0.9236	30.08/0.8367	28.91/0.8000	27.55/0.8385
	ESRN-V [20]	324	36.2	34.23/0.9262	30.27/0.8400	29.03/0.8039	27.95/0.8481
× 3	EDSR [15]	1518	160.8	34.37/0.9270	30.28/0.8418	29.09/0.8052	28.15/0.8527
	WDSR [24]	1203	122.5	34.48/0.9279	30.39/0.8434	29.16/0.8067	28.38/0.8567
	Ours ($t = 290$ms)	122	12.5	34.13/0.9252	30.12/0.8372	28.98/0.8015	27.71/0.8420
	Ours ($t = 150$ms)	51	5.2	33.85/0.9225	29.95/0.8347	28.86/0.7984	27.35/0.8340
	Ours ($t = 50$ms,**real-time**)	16	1.5	33.29/0.9160	29.57/0.8261	28.61/0.7929	26.44/0.8106

Table A1. Comparison of searched results with state-of-the-art efficient SR models for the ×3 upscaling task.

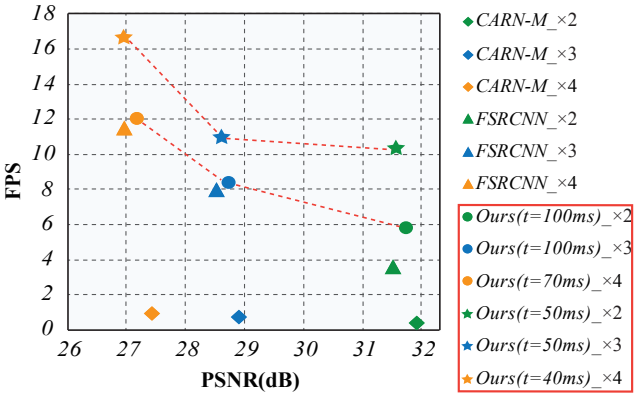

Figure A3. FPS v.s. PSNR for different SR methods implemented with MNN on mobile CPU evaluated on B100.

F. Fast Evaluation for Architecture and Pruning Search

Though Bayesian Optimization (BO) is leveraged to reduce the evaluation cost, it is still time-consuming to get the precise image quality of each candidate g in the selected B candidates as it requires the complex pruning and the full retraining process. Conducting the architecture and pruning search process with such an evaluation method will take a huge amount of time and computations. To reduce the overall search time, we utilize several strategies to accelerate the image quality evaluation process.

First, instead of using a complex pruning algorithm such as iterative pruning [11] and regularization-based methods [19, 25], we conduct a magnitude-based one-shot pruning by removing weights based on the L2-norm of the selected structural sparsity and the pruning ratio according to the latency model. Though it might lead to a more severe accuracy degradation compared with other pruning methods, one-shot pruning can still distinguish the different performance among different pruning schemes. Moreover, it is the relative accuracy performance, not the precise accuracy of different pruning schemes, that the search process cares for. Therefore, adopting a magnitude-based one-shot pruning for fast evaluation if suitable.

Metric	Method	Params	Multi-Adds	Set5	Set14	B100	Urban100
LPIPS	FSRCNN	12K	4.6G	0.2187	0.3032	0.3354	0.3414
	CARN-M	412K	32.5G	0.1810	0.2733	0.3128	0.2793
	WDSR	1203K	69.3G	0.1764	0.2640	0.3047	0.2535
	Ours	125K	7.1G	0.1793	0.2725	0.3117	0.2742
	Ours	12K	0.7G	0.1954	0.2882	0.3218	0.3135

Table A2. Comparison on ×4 upscaling tasks using LPIPS. Lower is better for LPIPS. Multi-Adds is reported for an input 320×180 image patch. Red/blue text: best/second-best LPIPS result.

Second, as the supernet is well-trained with each candidate dense net $a \in \mathcal{A}$ is optimized simultaneously, we leverage an early stopping mechanism in the retraining phase by only retraining for several epochs. The partially regained accuracy can predict the final model accuracy and be used to compare the performance among different schemes [26, 21]. With the fast pruning and retraining of each selected candidate g, we could greatly accelerate the image quality evaluation, thus significantly reducing the search cost.

G. LPIPS Performance

We further evaluate the perceptual quality of our method in terms of LPIPS. The results are shown in Table A2. According to the results, our method needs much less resource with second-best LPIPS.

H. Comparison with APQ

APQ [22] jointly searches network architecture, pruning, and quantization for efficient DNN deployment. The differences between our method and APQ are summarized as below: (i) We have different search objectives. While APQ focuses on classification accuracy, we try to achieve real-time (RT) Super Resolution (SR) on mobile with specific RT requirements, which is more challenging due to huge computations for high resolutions. (ii) We have different search strategies. We decouple pruning ratio search from architecture and pruning scheme search to reduce search complexity, thus accelerating the search process, while APQ unifies NAS, pruning and quantization as joint optimization. (iii) We have a larger pruning scheme search space. We can choose channel pruning, pattern pruning, or block pruning for each layer with higher flexibility for both high accuracy

and speed, while APQ only supports coarse-grained channel pruning with potential accuracy degradation when pruning ratio is high. (iv) We adopt additional methods (Bayesian optimization with neural predictors) to reduce search cost with higher efficiency. (v) The connection with hardware is different. We have a specific target hardware device (mobile phones) with detailed compiler optimization (CO), while APQ uses an ASIC design and optimizes model for it.

References

[1] https://www.tensorflow.org/mobile/tflite/. 1

[2] https://pytorch.org/mobile/home. 1

[3] https://github.com/alibaba/MNN. 1

[4] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P Sadayappan. On optimizing machine learning workloads via kernel fusion. *ACM SIGPLAN Notices*, 50(8):173–182, 2015. 1

[5] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017. 1

[6] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Alexandre V Evfimievski, and Prithviraj Sen. On optimizing operator fusion plans for large-scale machine learning in systemml. *arXiv preprint arXiv:1801.00829*, 2018. 1

[7] Emmanuel J Candes, Michael B Wakin, and Stephen P Boyd. Enhancing sparsity by reweighted 1 minimization. *Journal of Fourier analysis and applications*, 14(5-6):877–905, 2008. 2

[8] Tianqi Chen, Thierry Moreau, et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *USENIX*, pages 578–594, 2018. 1

[9] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Learning a deep convolutional network for image super-resolution. In *European conference on computer vision*, pages 184–199, 2014. 4

[10] Chao Dong, Chen Change Loy, and Xiaoou Tang. Accelerating the super-resolution convolutional neural network. In *European conference on computer vision*, pages 391–407. Springer, 2016. 3, 4

[11] Song Han, Jeff Pool, et al. Learning both weights and connections for efficient neural network. In *NeuIPS*, pages 1135–1143, 2015. 4

[12] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *ICCV*, pages 1389–1397, 2017. 2

[13] Zheng Hui, Xiumei Wang, and Xinbo Gao. Fast and accurate single image super-resolution via information distillation network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 723–731, 2018. 3, 4

[14] Tuanhui Li, Baoyuan Wu, Yujiu Yang, Yanbo Fan, Yong Zhang, and Wei Liu. Compressing convolutional neural networks via factorized convolutional filters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3977–3986, 2019. 2

[15] Bee Lim, Sanghyun Son, Heewon Kim, Seungjun Nah, and Kyoung Mu Lee. Enhanced deep residual networks for single image super-resolution. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 136–144, 2017. 4

[16] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 2736–2744, 2017. 2

[17] Xiaolong Ma et al. Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices. In *AAAI*, 2020. 1

[18] Wei Niu et al. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. *arXiv:2001.00138*, 2020. 1, 2

[19] Ao Ren, Tianyun Zhang, Shaokai Ye, Jiayu Li, Wenyao Xu, Xuehai Qian, Xue Lin, and Yanzhi Wang. Admm-nn: An algorithm-hardware co-design framework of dnns using alternating direction methods of multipliers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 925–938. ACM, 2019. 2, 4

[20] Dehua Song, Chang Xu, Xu Jia, Yiyi Chen, Chunjing Xu, and Yunhe Wang. Efficient residual dense block search for image super-resolution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 12007–12014, 2020. 4

[21] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2820–2828, 2019. 4

[22] Tianzhe Wang, Kuan Wang, Han Cai, Ji Lin, Zhijian Liu, Hanrui Wang, Yujun Lin, and Song Han. Apq: Joint search for network architecture, pruning and quantization policy. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2078–2087, 2020. 4

[23] Wei Wen, Chunpeng Wu, et al. Learning structured sparsity in deep neural networks. In *NeurIPS*, pages 2074–2082, 2016. 2

[24] Jiahui Yu, Yuchen Fan, Jianchao Yang, Ning Xu, Zhaowen Wang, Xinchao Wang, and Thomas Huang. Wide activation for efficient and accurate image super-resolution. *arXiv preprint arXiv:1808.08718*, 2018. 4

[25] Tianyun Zhang, Shaokai Ye, et al. Systematic weight pruning of dnns using alternating direction method of multipliers. *ECCV*, 2018. 2, 4

[26] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2423–2432, 2018. 4