A Algorithm 1 explanation

We follow the standard Knapsack problem dynamic programming solution to break down the original problem into sub-problems. Specifically, for each neuron j (or neuron group) in layer l, we can choose to either include or not include it under the latency constraint c. When it is kept, the total importance score increases $\mathcal{I}_l^j$ while the latency constraint for the other neurons becomes $c - c_l^j$; If the neuron is removed, the latency constraint for the other neurons remains c. We choose to keep or remove the current neuron to maximize total importance. At the same time, we check whether the more important neurons in the same layer are included to ensure the correctness of the latency. The neuron selection from the remaining neurons is a sub-problem to solve.

Precisely, we use a vector $\text{maxV} \in \mathbb{R}^{(C+1)}$ to store the maximum importance that we can achieve under the latency constraint c, $0 \leq c \leq C$ and $\textit{keep} \in \mathbb{R}^{L \times (C+1)}$, a 2D vector where $\text{keep}[l, c]$ denotes the number of neuron groups we need to maintain in layer l to obtain the maximum importance $\text{maxV}[c]$. We process the neurons according to their importance score in decreasing order. In this way, all preceding neurons to the current one (i.e., neurons with a higher importance score in the same layer) will be always considered first. To decide if we keep or remove the current neuron, we check the total importance score and the inclusion status of its preceding neurons, so we can maximize the total importance and ensure the latency cost correctness.

B Experimental settings

For image classification, in the main paper we focus on pruning networks on the large-scale ImageNet ILSVRC2012 dataset [53] (1.3M images, 1000 classes). Each pruning process consumes a single node with eight NVIDIA Tesla V100 GPUs. We use PyTorch [49] V1.4.0 model zoo for pretrained weights for our pruning for a fair comparison with literature.

In our experiments we perform iterative pruning. Specifically, we prune every 320 minibatches after loading the pretrained model with $k = 30$ pruning steps in total to satisfy the constraint. Unless otherwise specified, we finetune the network for 90 epochs in total with an individual batch size at 128 for each GPU. For finetuning, we follow NVIDIA's recipe [48] with mixed precision and Distributed Data Parallel training. The learning rate is warmed up linearly in the first 8 epochs and reaches the highest learning rate, then follows a cosine decay over the remaining epochs [38]. For the result in Fig.1 that is trained with knowledge distillation, we use RegNetY-16GF (top1 82.9%) as the teacher model when finetuning the pruned model. We use hard distillation on the logits and the final training loss is calculated as $L = (1 - \alpha)L_{base} + \alpha L_{distil}$ where $\alpha = 0.5$ to balance between the original loss L_{base} and the distillation loss L_{distil}.

For latency lookup table construction, we target a NVIDIA TITAN V GPU with batch size 256 for latency measurement to allow for highest throughput for inference, and target a Jetson TX2 with inference batch size 32. We pre-generate a layer latency look-up table on the platform by iteratively reducing of the number of neurons in a layer to characterize the latency with NVIDIA cuDNN [7] V7.6.5. We profile each latency measurement 100 times and take the average to avoid randomness.

We also provide pruning results on the small-scale CIFAR10 [29] dataset in appendix Sec. C. For CIFAR10 experiments with ResNet-50/-56, we train the model on a single GPU for 200 epochs in total where we perform pruning step very one epoch in the first 30 epochs and finetune the pruned model during the remaining 170 epochs. The initial learning rate is set to 0.1 with batch size 128. For DenseNet, we extend the finetuning epochs to 300 epochs

C More pruning results on CIFAR10 and ImageNet

We provide the pruning results of our method on CIFAR10 dataset in this section. We chose 3 network architectures for the experiment: ResNet50, ResNet56 and DenseNet40-12 [26]. As most of the prior methods perform pruning under the FLOPs constraint, in our CIFAR10 experiment we also use FLOPs constraint instead of the latency constraint. We also add some additional ImageNet-ResNet50 results comparison in the table. For a fair comparison, in the ImageNet50 experiment, we also provide the model accuracy under the same finetuning recipe and epochs as the methods to be compared to alleviate the potential impact of the different finetuning settings. Specifically, when compared to

Table 5: Additional pruning results and comparison on CIFAR10 and ImageNet dataset. FLOPS (%) are relative to those of the unpruned network

Dataset	Model	Method	FLOPs (%) ↓	Top1 (%) ↑
CIFAR10	ResNet50	ChipNet [60]	17.7	92.8
		HALP(Ours)	**13.7**	**93.2**
	ResNet56	CHIP [57]	27.7	92.05
		HALP (Ours)	**26.8**	**93.22**
		GDP [19]	34.36	93.55
		HALP (Ours)	**33.72**	**93.68**
	DenseNet40-12†	QCQP [28]	29.2	93.80†
		HALP (Ours)	29.2	93.15†
ImageNet	ResNet50	GBN-60 [71]	59.46	76.19
		QCQP [28]	59.0	76.00
		HALP (Ours)	**58.12**	**76.93** (**76.49*** / **76.53****)
		GBN-50 [71]	44.94	75.18
		HALP (Ours)	**42.05**	**76.09** (**75.27****)

Dataset	Model	Method	Inf speedup (%) ↑	Top1 drop (%) ↓
ImageNet	ResNet50	QCQP [28]	1.52×	0.32
		HALP (Ours)	**1.60**×	**−0.22** (**0.16****)

† The baseline model used in QCQP has 95.01% top1 accuracy, while our pretrained model has 94.40% top1 accuracy. The accuracy drop is 1.21% vs. 1.25%, which is comparable.
* use the same finetune recipe and epochs as GBN [71]
** use the same finetune recipe and epochs as QCQP [28]

Table 6: Pruning MobileNet-V1 and MobileNet-V2 on the ImageNet dataset with different targets.

Method	FLOPs (M)	Top1 (%)	Top5 (%)	FPS (im/s)	Speedup
MobileNet-V1					
No pruning	569	72.64	90.88	3415	1×
HALP-40%	154	67.20	87.32	8293	2.43×
HALP-42%	171	68.30	88.08	7940	2.32×
HALP-50%	237	69.79	89.08	6887	2.02×
HALP-60%	297	71.31	90.05	5754	1.68×
HALP-70%	360	71.78	90.39	4870	1.43×
HALP-80%	416	72.52	90.78	4167	1.22×
HALP-90%	507	72.95	91.02	3765	1.10×

Method	FLOPs (M)	Top1 (%)	Top5 (%)	FPS (im/s)	Speedup
MobileNet-V2					
No pruning	301	72.10	90.60	3080	1×
HALP-60%	183	70.42	89.75	5668	1.84×
HALP-65%	218	71.41	90.08	5003	1.62×
HALP-70%	227	71.88	90.39	4478	1.45×
HALP-75%	249	72.16	90.44	4109	1.33×
HALP-90%	273	72.45	90.68	3443	1.12×
HALP-95%	281	72.55	90.79	3265	1.06×

GBN [71], we finetune the pruned network for 60 epochs, where initial learning rate is set to 0.01 with batch size 256. The learning rate is divided by 10 at epoch 36, 48 and 54. When compared to QCQP [28], the pruned network is finetuned for 80 epochs with batch size 384 and the initial learning rate of 0.015. Then, the learning rate is decayed at epoch 30 and 60 by deviding 10. As shown in Tab. 5, HALP method consistently outperforms with lower FLOPs and higher Top1 accuracy. When it comes to the actual inference speed comparison with QCQP [28], our method yields 0.16% less accuracy drop while getting 0.08× faster speed.

Table. 6 and Fig. 5 provide additional pruning results for lightweight networks such as MobileNet-V1 and MobileNet-V2. For the unpruned models, we find that even MobileNet-V2 has significantly lower FLOPs, the inference time is larger compared to MobileNet-V1.In both cases, HALP yields inference speeds-ups of 1.22× and 1.33× for MobileNet-V1 and MobileNet-V2 respectively, while maintaining the original top1 accuracy.

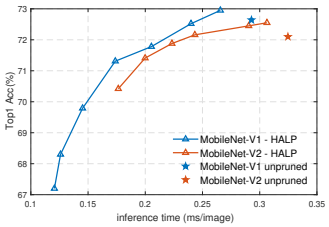


Figure 5: Pruning MobileNets on the ImageNet dataset.

D Efficacy of neuron grouping on MobileNet

In this section, we show the benefits of latency-aware neuron grouping and the performance under different group size settings on MobileNetV1.

Since MobileNet has group convolutional layers to speedup the inference, we take the group convolutional layer with its preceding connected convolutional layer together as coupled cross-layers [17] to

make sure the input channel number and output channel number of the group convolution remain the same. All the 27 convolutional layers can be divided into 14 coupled layers. In our method, with the neuron grouping, we set the individual group size of 1 coupled layer to 16, of 3 coupled layers to 32 and 10 coupled layers to 64. Also, for MobileNetV1 pruning, we add the additional constraint that each layer has at least one group of neurons remaining to make sure that the pruned network is trainable.

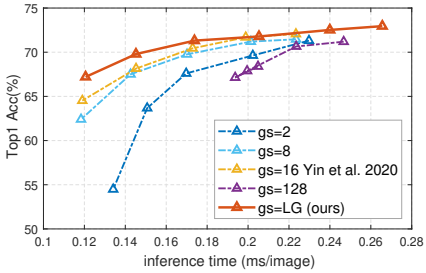


Figure 6: Performance comparison of our latency-aware grouping to different fixed sizes for a MobielNetV1 pruned with different latency constraints on ImageNet. We compare to heuristic-based group selection studied by [69]. LG denotes the proposed latency-aware grouping in HALP that yields consistent latency benefits per final accuracy.

We compare our latency-aware neuron grouping with an heuristic option by setting a fixed group size for all layers. Fig. 6 shows the comparison results between our neuron grouping method and various fixed group sizes for a MobielNet pruned with different latency constraints on ImageNet. As shown, similar to ResNet50, using small group sizes such as 8, 16 leads to worse performance; a large group size like 128 also harms the performance significantly. Our observations on ResNet50 pruning also hold in MobileNetV1 setting, further emphasizing the efficacy of our latency-aware neuron grouping.

E Ablation study of pruning step k

In this work, similar to many other prior methods [2, 45, 72], we do iterative pruning with k pruning steps in total. In this experiment, we analyze the the accuracy of the final result as a function of k. We set the value of k to 10, 20, 30 and 40 for iterative pruning, and also use $k = 1$ to perform a single-shot pruning. The result of this experiment is shown in Fig. 7. As shown, we get similar results independent of k. Imporantly, all these results outperform EagleEye [32]. As expected, there is a drop in accuracy for single-shot pruning ($k = 1$), especially for large pruning ratios. The main reason is the neuron importance would change as we remove some other neurons and, in this setting, the value is not updated. Iterative pruning does not have this limitation as the importance score and the latency cost of the remaining neurons is updated after each pruning step to reflect any changes. In our experiments, we use $k = 30$ as it provides a good trade off between latency and accuracy.

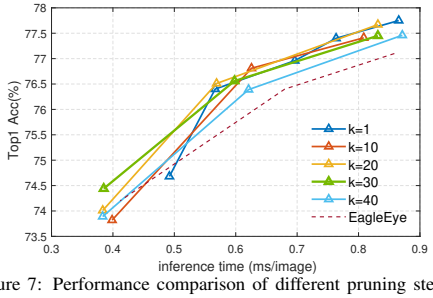


Figure 7: Performance comparison of different pruning steps k for ResNet50 pruning on ImageNet.

F Comparison with EagleEye on ImageNet

We now use the same unpruned baseline model provided by EagleEye [32] to compare our proposed HALP method with EagleEye [32] varying the latency constraint. As shown in Fig. 8, our approach dominates EagleEye by consistently delivering a higher top-1 accuracy with a significantly faster inference time.

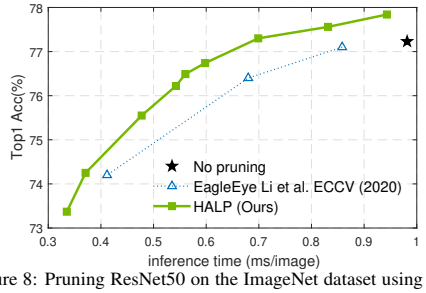


Figure 8: Pruning ResNet50 on the ImageNet dataset using the same baseline model as in EagleEye with a top-1 accuracy of 77.23%. The proposed HALP surpasses EagleEye ECCV20 [32] in accuracy and latency. Top-left is better.

We then analyze the structure difference between our pruned model and the EagleEye model. As mentioned in the main text that the proposed HALP method tries to make the number of remaining neurons in each layer fall to the right side of a step if the latency on the targeting platform presenting a staircase pattern. Fig. 9 shows two examples of pruned layers after pruning from HALP-45% and EagleEye-2G model. In the

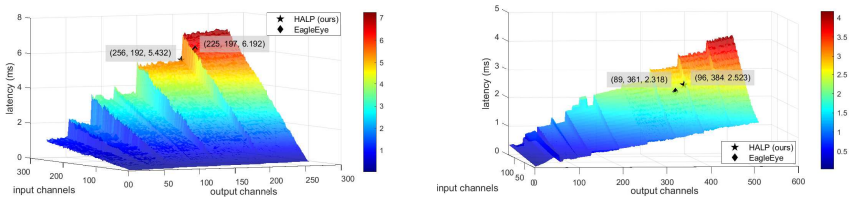


Figure 9: Two examples of pruned layers from HALP model and EagleEye [32] model. The scattered black points are the locations of the layers fall to after pruning.

Table 7: Pruning ResNet50 on the ImageNet dataset (TITAN V) targeting on inference with batch size 1. HALP-$X\%$ indicates that $X\%$ latency to remain after pruning. The speedup is calculated as the ratio of FPS between the pruned network and the unpruned model.

Method	FLOPs (G)	Top1 Acc (%)	Top5 Acc (%)	FPS (imgs/s)	Speedup
No pruning	4.1	76.2	92.87	181	1×
0.75× ResNet50 [21]	2.3	74.8	-	192	1.06×
AutoSlim [72]	2.0	75.6	-	181	1.00×
MetaPruning [36]	2.0	75.4	-	190	1.05×
EagleEye-2G [32]	2.1	76.4	92.89	190	1.05×
GReg-2 [63]	1.8	75.4	-	196	1.09×
HALP-90% (Ours)	2.9	**76.4**	**93.10**	**220**	**1.22×**
0.50× ResNet50 [21]	1.1	72.0	-	193	1.07×
AutoSlim [72]	1.0	74.0	-	191	1.06×
MetaPruning [36]	1.0	73.4	-	196	1.09×
EagleEye-1G [32]	1.0	74.2	91.77	192	1.06×
GReg-2 [63]	1.3	73.9	-	206	1.14×
HALP-80% (Ours)	2.3	**75.3**	**92.35**	**247**	**1.37×**

left figure, we show that the layer in our pruned model has only 5 more neurons pruned than that in EagleEye model, the latency is reduced to a much lower level which is a 0.76ms drop while we have 31 more input channels. In the right figure, we also show that sometimes we can remain a lot more neurons (30 neurons) in layer with only little latency (0.21ms) increase. These two examples both show the ability of method to fully exploit the latency traits and benefit the inference speed.

Our method benefits a lot from the non-linear latency characteristic since we are trying to keep as many neurons as possible under the latency constraint. If the latency of the layer on the targeting platform shows linear pattern, the advantage of our method becomes smaller. Fig. 9 shows the latency behavior of the example layers on the targeting platform when reducing the number of input and output channels. As we can see, the staircase pattern becomes less obvious as the number of input channel reduces and the GPU has sufficient capacity for the reduced computation. This happens during pruning, especially for large prune ratios. In such a case, the FLOP count reflects the latency more accurately, and the performance gap between reducing FLOPs and reducing latency can possibly become small. Nevertheless, our method can help avoid some latency peaks as shown in Fig. 9, which could otherwise happen using other pruning methods.

G Pruning results for small batch size

In the main paper, we use a large batch 256 in the experiment to allow for highest throughput for inference, which also makes the latency of the convolution layers show apparent staircase pattern so that we can take full advantage of the latency characteristic. In this section, we show that with small batch size 1 that no obvious staircase pattern showing up in layer latency, our HALP algorithm still delivers better results compared to other methods.

When we use batch size 1 for inference, the layer latency of ResNet50 does not show obvious staircase pattern in most of the layers due to the insufficient usage of GPU. Therefore in this experiment, we use the latency lookup table granularity as a neuron grouping size, which in our case is 2, to fully exploit the hardware latency traits during pruning. We show our pruned results and the comparison

Table 8: HALP for object detection on the PASCAL VOC dataset.

Model	mAP	FLOPs (G)	params (M)	FPS (BS=1)	FPS (BS=32)
SSD512-RN50, base model	77.98	65.56	21.97	68.24	103.48
SSD512-RN50-slim	75.83	46.09	16.33	76.49	114.80
SSD300-RN50	75.69	16.23	15.43	128.85	309.32
SSD300-VGG16 [35]	76.72	31.44	26.29	122.28	262.93
FasterRCNN-VGG16 [52]	70.10	91.23	137.08	29.21	-
RetinaNet-RN50 [34]	77.27	106.50	36.50	36.92	-
SSD512-RN50-HALP (Ours)	**77.42**	**15.38**	**10.40**	**132.57**	**323.36**

with other methods in Tab. 7. As shown in the table, while other methods reduce the total FLOPs of the network after pruning, they do not reduce the actual latency much, which is up to $1.09\times$ faster than the original one at the cost of 2.8% top1 accuracy drop. Compared to these methods, although we get less FLOPs reduction using our proposed method, the pruned models are faster and get higher accuracy, which is $1.22\times$ faster than the unpruned model while getting slightly higher accuracy and $1.37\times$ faster with only 0.9% accuracy drop.

H Pruning results on object detection

In this section we show the detailed pruning results on objection detection task for Sec. 4.5. To prune the detector, we first train a SSD512 with ResNet50 as backbone. We also train some other popular models for performance comparison. The detailed numbers of Fig. 4 are shown in Tab. 8.

I Implementation details

Convert latency in float to int. Solving the neuron selection problem using the proposed augmented knapsack solver (Algo. 1 in the main paper), requires the neuron latency contribution and the latency constraint to be integers as shown in line 4 of the algorithm. To convert the measured latency from a full precision floating-point number to integer type, we multiply the latency by 1000 and perform rounding. Accordingly, we also scale and round the latency constraint value.

Deal with negative latency contribution. The neuron latency contribution in our augmented knapsack solver must be a non-negative value since we have dp_array $\in \mathbb{R}^C$ and we need to visit dp_array$[c - c_n]$ as in line 5 of Algo. 1 in the main paper. However, by analyzing the layer latency from the look-up table we find that for some layers the measured latency might even increase when reducing some number of neurons. This means that the latency contribution could possibly be negative. The simplest way to deal with the negative values is to directly set the negative latency contributions to be 0. This leads to the problem that the summed latency contribution would be larger than the actual latency value, causing less neurons being selected. Thus, during our implementation, we keep those negative latency values as they are, but update the vector size of dp_array to $\mathbb{R}^{C-\min(\min(\mathbf{c}),0)}$ where $\min(\mathbf{c})$ is the minimum latency contribution. With such, the vector size of dp_array would be extended when there is negative latency contribution. This makes it possible to add one neuron with negative latency contribution to a subset of neurons whose summed latency is larger than the latency constraint. After the addition, the total latency will still remain under the constraint.

Pruning of the first layer. In our ImageNet experiments, we leave the first convolutional layer of ResNets unpruned to help maintain the top-1 classification accuracy. For MobileNet, the first convolutional layer is coupled with its following group convolutional layer. In our MobileNet experiments, we prune the first coupled layers at most to the half of neurons.

SSD for object detection. Our SSD model is based on [35]. When we train SSD-VGG16, we use the exactly same structure as described in the paper. When we train a SSD-ResNet50, the main difference between our model and the model described in the original paper is in the backbone, where the VGG is replaced by the ResNet50. Following [27], we apply the following enhancements in our backbone:

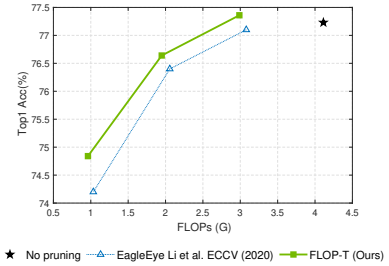
- The last stage of convolution layers, last avgpool and fc layers are removed from the original ResNet50 classification model.
- All strides in the 3rd stage of ResNet50 layers are set to 1×1.

Table 9: The additional convolution layers in SSD.

layer	SSD512	SSD512-slim	SSD300
layer1-conv1	(512, 3, 1, 1)	(256, 1, 1, 0)	(512, 3, 1, 1)
layer1-conv2	(512, 3, 2, 1)	(512, 3, 2, 1)	(512, 3, 2, 1)
layer2-conv1	(256, 1, 1, 0)	(256, 1, 1, 0)	(256, 1, 1, 0)
layer2-conv2	(512, 3, 2, 1)	(512, 3, 2, 1)	(512, 3, 2, 1)
layer3-conv1	(128, 1, 1, 0)	(128, 1, 1, 0)	(128, 1, 1, 0)
layer3-conv2	(256, 3, 2, 1)	(256, 3, 2, 1)	(256, 3, 2, 1)
layer4-conv1	(128, 1, 1, 0)	(128, 1, 1, 0)	(128, 1, 1, 0)
layer4-conv2	(256, 3, 2, 1)	(256, 3, 2, 1)	(256, 3, 1, 0)
layer5-conv1	(128, 1, 1, 0)	(128, 1, 1, 0)	(128, 1, 1, 0)
layer5-conv2	(256, 3, 2, 1)	(256, 3, 2, 1)	(256, 3, 1, 0)
layer6-conv1	(128, 1, 1, 0)	(128, 1, 1, 0)	-
layer6-conv2	(256, 4, 1, 1)	(256, 4, 1, 1)	-

Table 10: Pruning ResNet50 on the ImageNet dataset with FLOPs constraint and comparison with state-of-the-art method EagleEye (ECCV'20) [32]. We remeasure the FLOPs, top1 and top5 accuracy of EagleEye to get results with two digits.

Method	FLOPs (G)	Top1 Acc (%)	Top5 Acc (%)
No pruning	4.1	77.23	93.70
EagleEye-3G	3.08	77.10	93.36
FLOP-T (Ours)	**2.99**	**77.36**	**93.62**
EagleEye-2G	2.06	76.38	92.90
FLOP-T (Ours)	**1.95**	**76.64**	**93.21**
EagleEye-1G	1.03	74.18	91.78
FLOP-T (Ours)	**0.96**	**74.84**	**92.26**



The backbone is followed by 6 additional coupled convolution layers for input size 512×512, or 5 for input size 300×300. A BatchNorm layer is added after each convolution layer. The settings of these additional convolution layers are listed in Tab. 9, each layer is represented as (output channel, kernel size, stride, padding).

The detector heads are similar to the ones in the original paper. The first detection head is attached to the last layer of the backbone.The rest detection heads are attached to the corresponding additional layers. No additional BatchNorm layer in the detector heads.

J FLOPs-constrained pruning

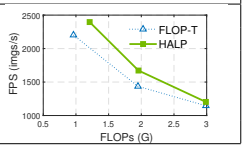
Our implementation of latency-constrained pruning can be easily converted to be a FLOPs-constrained. When constraining on FLOPs, $\Phi(\cdot)$ in the objective function (Eq.1 in the main paper) becomes the FLOPs measurement function and C becomes the FLOPs constraint. Since the FLOPs of a layer linearly decreases as the number of neurons decreases in the layer, we do not need to group neurons in a layer any more. The problem can also be solved by original knapsack solver since each neuron's FLOPs contribution in a layer is exactly the same and no preceding constraint is required. We conduct some experiments by constraining the FLOPs and compare the results with EagleEye [32]. We name the experiments using the same algorithm as HALP but targeting on optimizing the FLOPs as FLOP-T. As shown in Tab. 10, with our pruning framework applying the knapsack solver, our results show higher top-1 accuracy compared to the pruned networks of EagleEye with similar FLOPs remaining. We also observe a larger gap between the methods when it comes to a more compact network.

K FLOPs vs. latency

FLOPs can be regarded as a proxy of inference latency; however, they are not equivalent [4, 33, 35, 40, 42]. We do global filter-wise pruning and have the same problem as NAS. The latency on a GPU usually imposes staircase-shaped patterns for convolutional operators with varying channels and requires pruning in groups. In contrast, FLOPs will change linearly. Depth-wise convolution, compared to dense counterparts, has significantly fewer FLOPs but almost the same GPU latency

Table 11: ResNet50 pruning with FLOPs/latency constrain.

method	Top1(%)	FLOPs (G)	FPS (imgs/s)	FPS vs FLOPs
FLOP-T	74.84	**0.962**	2202	
HALP	**74.92**	1.210	**2396**	
FLOP-T	**76.64**	**1.949**	1436	
HALP	76.55	1.957	**1672**	
FLOP-T	77.36	**2.988**	1146	
HALP	**77.45**	**2.988**	**1203**	



due to execution being memory-bounded[2]. The discrepancy also holds for ResNets where the same amount of FLOPs impose more latency in earlier layers than later ones as the number of channels increases and feature map dimension shrinks – both increase compute parallelism. For example, the first 7×7 conv layer and the first bottleneck 3×3 conv in ResNet50 have nearly identical FLOPs but the former is 60% slower on-chip.

We compare our results of FLOPs-targeted (FLOP-T) showed in Sec. J and results using latency-targeted pruning (HALP) in Tab. 11. As shown in the table, using different optimization targets leads to quite different FPS vs FLOPs curves. In overall, with similar FLOPs remaining, using our HALP algorithm targeting on reducing the actual latency can get more efficient networks with more image being processed per second.

We also show a more straightforward relationship between the actual latency of a layer and its FLOPs in Fig. 10. We use the 2nd convolution layer in the 1st residual block of ResNet50 as an example. We vary the number of neurons of the layer from 0 to 128 and measure the actual latency on GPU (TITAN V) as well as the FLOPs of the layer. We can see from the figure that the actual latency does not strictly linearly decrease as the FLOPs decreasing.

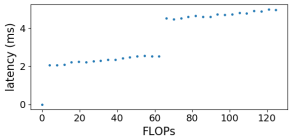


Figure 10: The measured latency vs. FLOPs of the 2nd convolution layer in the 1st residual block of ResNet50.

L Different choice of importance calculation

We use the first Taylor expansion [45] to estimate the loss change induced by pruning as the importance score of the neurons. It is a gradient-based importance calculation and is shown to be given promising results. In this section, we use the L2 norm of the neuron weights as the importance measurement and apply HALP framework to ResNet50 ImageNet classification task. As shown in Tab. 12, Our algorithm is generic applying to different importance measurements. As shown, using L2 norm of weights as importance measurements leads to slightly lower accuracy.

Table 12: The results of HALP algorithm on ResNet50 ImageNet classification task with different choices of neuron importance measurements.

Method	First-Taylor Expansion (gradient-based)				L2 norm (magnitude-based)			
	FLOPs(G)	Top1(%)	Top5(%)	FPS(im/s)	FLOPs(G)	Top1(%)	Top5(%)	FPS(im/s)
HALP-80%	3.0	77.5	93.60	1203	3.0	77.3	93.60	1196
HALP-55%	2.1	76.7	93.16	1672	2.0	75.7	92.66	1595

M Latency look-up table creation and calibration

In this section, we provide additional details to build the latency look-up table used in HALP, computational cost and the correlation between the estimated and the real ones. As mentioned in Appendix B, we pre-generate the layer latency look-up table on the platform with NVIDIA cuDNN [7] V7.6.5. For each layer, we iteratively reduce the number of neurons in the layer (each time reduce 8 neurons) and characterize the corresponding latency. For each latency measurement, we use one profile for GPU warm up and another 3 profiles and take the average to avoid randomness. The average standard deviation of profiles for an operation is $8.67e^{-3}$.

[2] https://tlkh.dev/depsep-convs-perf-investigations/

On a single TITAN V GPU, it takes around 5 hours to build the look-up table for ResNets family and 1 hour for MobileNets family. Note that the LUT can be shared by the network architectures within the same family since they usually have similar layer structures. We only need to create the latency look-up table once for all the possible latency targets.

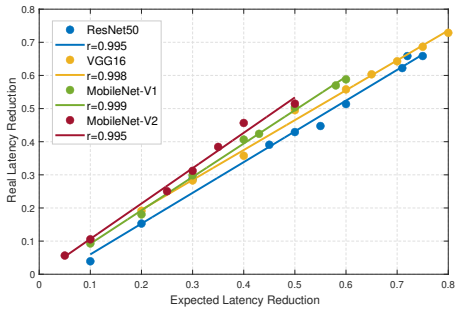


Figure 11: The correlation between the predicted latency reduction and the real latency reduction of the pruned models.

There are some gaps between the predicted latency and the real latency of the model, because the latency look-up table is created layer-wise on convolution layers. There are additional costs in real inference such as pooling, non-linear activation etc. We plot the correlation between the expected latency reduction from look-up table and the real latency reduction ratio of our pruned models in Fig. 11. We also calculate the Pearson Correlation Coefficient r for all the networks in the figure. We can clearly see a linear correlation between the predicted and real latency reduction from the figure, showing that the latency lookup table provides a good approximation and it is possible to calibrate the latency estimation using the linear coefficient to have a better estimation.

Limitation of layer-wise latency lookup table. We apply layer-wise latency lookup table in our method, which does not consider the caching and parallelization among layers. For networks like VGG without bypass paths, the layer operations will be executed sequentially, in which case, the latency lookup table models the actual latency well. For models with parallel paths, it depends on the hardware implementation whether there will be parallel execution in practice. When there is parallelization, e.g., in accelerators, models like ResNets still work well because the skip connection only takes small portion of computation; for models like InceptionNet, taking the parallelization and into consideration when generating the lookup table would help better estimate the latency. For wider applications in the future, the more domain knowledge we have about the GPU execution and improve accordingly, the more accurate estimation we will obtain using the latency lookup table.

Comparison with quadratic model. Recent work QCQP [28] models latency using a quadratic equation and solve a latency constrained optimization problem. The main difference between our estimation from lookup table and the QCQP's estimation from quadratic modeling is that we use different ways to model each layer's latency. QCQP[1] uses $\alpha_l + \beta_l||r^{(l-1)}||_1 + \delta_l||r^{(l-1)}||_1||r^{(l)}||_1$ to model the layer latency where $||r||_l$ is the number of remaining channels in the corresponding layer, α, β and δ are the coefficients that need to be optimized for the targeting platform. Note that QCQP also needs to profile latency for different samples of each layer, like what we do to create the lookup table but will less samples, in order to optimize α, β and δ. It is also important to note that QCQP uses a linear model between the layer latency and FLOPs (the quadratic part) and memory. Therefore, QCQP fails to capture the latency staircase pattern (see Fig. 3a in the QCQP paper) which is the key to maximize GPU utilization (see latency surface in Fig. 1). As shown in Fig.3a in the QCQP paper, the quadratic modeling gives different latency estimations to layers with different number of input and output channels. However, these layers have the same real inference time. As a result, the larger the number of input and output channels, the larger the error in the estimation of QCQP.

N Pruning for INT8 quantization

We now focus on results when the target is INT8 inference which is a common requirement for real-world applications. In particular, we use NVIDIA Xavier as the target platform as, in this platform, INT8 speedup is supported. We create a INT8 latency look-up table for INT8 inference. For comparison, we also create a FP32 latency look-up table on the same platform and use both look-up tables for pruning a ResNet50 model on ImageNet classification. After convergence, results are quantized into INT8 and the latency

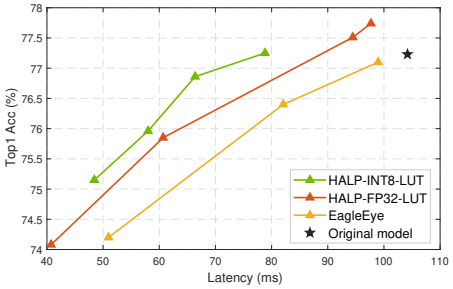


Figure 12: The ResNet50 ImageNet pruning targeting INT8 inference on NVIDIA Xavier.

and accuracy is measured directly on the Xavier platform. We measure latency with a batch size 128, using TensorRT (V8.2.0.1) to get INT8 speedup. Results for this experiment are shown in Fig. 12.

As shown, even use a FP32 latency look-up table as the latency guidance, our method HALP outperforms the state-of-the-art method EagleEye [32]. These results are consistent with Sec. 4.2 where we show the HALP acceleration on GPUs with TensorRT. Pruning results of using a INT8 look-up table show that our method yields higher accuracy with lower latency on this platform. We obtain a $1.32\times$ speedup while maintaining the original Top1 accuracy. Compared to EagleEye, HALP achieves up to $1.26\times$ relative speedup and 0.15% higher accuracy.

O Breakdown of the algorithm execution time

We provide additional details of the algorithm process of different methods in this section. We estimate the time cost needed to get the pruned network structure for each method. The time of following finetuning is not taken into consideration. For a fair comparison, we set the number of pruning steps k for all iterative pruning methods to 30. All the values are approximated as all the methods are running on the same device (a NVIDIA V100 GPU) to get a pruned ResNet50. For AutoSlim [72], MetaPruning [36] and AMC [22], more GPU time is needed for additional training of the network.

Table 13: The breakdown details of the execution process of different methods.

Method	Evaluate proposals?	Auxiliary net training?	Sub-network selection	Additional time cost	Estimate time (RN50)
NetAdapt	Y	N	N candidates evaluation + finetune after each prune. Repeat k times	Latency look-up table creation	$\sim$ 195h GPU
ThiNet	Y	N	1 or 2 train epochs after each pruning. Repeat k times	Additional forward pass to get neuron importance	$\sim$ 210h GPU
EagleEye	Y	N	1000 candidates evaluation	Monte Carlo sampling, prune to get 1000 candidates	30h GPU
AutoSlim	Y	Y	Train slimmable model k candidates evaluation		
MetaPruning	Y	Y	Train an auxiliary network k candidates evaluation		
AMC	N	Y	Train an RL agent		
HALP(**Ours**)	**N**	**N**	40 train iterations after each pruning. Repeat 30 times. ($<$ 1 train epoch in total)	Augmented Knapsack solver ($\sim$ 30min in total) Latency look-up table creation	6.5h GPU 0.5h CPU

P Difference with prior work

KnapsackPruning (KP) [1] is one work that is mostly close to our method in the paper. While both works look at similar problems from the same combinatorial perspective, there are several key differences. First, KP focuses on constraining FLOPs and shows an instantiation on latency; in contrast, we directly optimize the latency, which is more practical. Second, we show the latency characterization on device and augment the original knapsack problem formulation accordingly, Eq. 7, to accommodate to the latency traits - the neuron latency is dependent on the order of neuron pruning in a layer, while KP uses a standard knapsack where the neuron FLOP cost is independent of each other. Please note here that the formulation in KP assumes the independence thus can not directly apply to the latency-targeted pruning, Third, we are the first ones to use the latency-aware grouping which assigns different grouping sizes to each layer according to the latency traits rather than predefined fixed values.

For a fair comparison of pruning to the KP method, we use the PyTorch baseline as unpruned model and both without knowledge distillation during training. The results for ResNet50 pruning on ImageNet are show as Tab. 16. As shown, our method performs significantly better leading to pruned model with higher accuracy but less FLOPs. On the other side, as they are not considering the actual latency, the resultant network structure is not GPU friendly and would fail to maximize the GPU utilization.

Q Detailed configuration of pruned models

We provide the detailed configuration of our pruned models of Tab. 1. For each model, we list the number of neurons remaining in each convolution layer, starting from the input to the output. For

Table 14: ResNet50 pruning results on ImageNet comparison with Knapsack Pruning [1].

Method	FLOPs (G)	Top1 (%)	Acc Drop (%)
KP [1]	2.38	76.17	0.03
KP [1]	2.03	75.94	0.21
HALP (Ours)	**2.01**	**76.46**	**−0.26**

ResNets we use $[\cdot]$ to denote a residual block and $(\cdot)$ to denote the neuron number of the residual bypass layer.

Note that for ResNets, the configuration is the "raw" configuration after the pruning. For each residual block $[x_1, x_2, x_3]$, because of the existing of bypass layer, we allow the entire layer pruning and it still leads to trainable networks. Whenever there is entire layer pruning ($x_1 == 0$ or $x_2 == 0$ or $x_3 == 0$), all the other layers in the block can be removed and this branch can be replaced by a constant value. In such a case, the corresponding block in our final pruned ResNet50 models is cleaned to be $[0, 0, 0]$. We also provide the detailed structure plot of two pruned ResNet50 models in Fig. 13 for a better visualization.

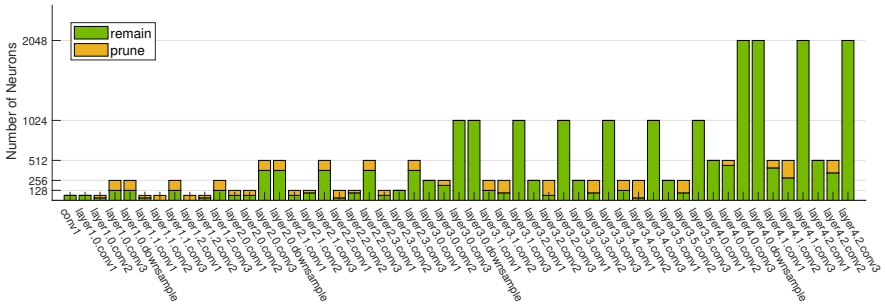
Table 15: The detailed configuration of the HALP pruned models.

	ResNet50 - EagleEye [32] baseline
HALP-80%	64, [64, 32, 256](256), [32, 32, 256], [0, 32, 256], [128, 128, 512](512), [64, 96, 512], [64, 128, 512], [64, 128, 512], [256, 256, 1024](1024), [256, 160, 1024], [256, 160, 1024], [256, 160, 1024], [256, 128, 1024], [256, 160, 1024], [512, 512, 2048](2048), [448, 416, 2048], [512, 512, 2048]
HALP-45%	64, [64, 32, 128](128), [32, 0, 128], [0, 32, 128], [64, 64, 384](384), [64, 96, 384], [32, 96, 384], [64, 128, 384], [256, 192, 1024](1024), [128, 96, 1024], [256, 64, 1024], [256, 96, 1024], [128, 32, 1024], [256, 96, 1024], [512, 448, 2048](2048), [416, 288, 2048], [512, 352, 2048]
HALP-30%	64, [0, 0, 64](64), [0, 0, 64], [0, 32, 64], [64, 64, 256](256), [64, 64, 256], [64, 128, 256], [64, 96, 256], [256, 64, 896](896), [128, 64, 896], [0, 0, 896], [128, 64, 896], [0, 0, 896], [0, 0, 896], [512, 320, 2048](2048), [320, 160, 2048], [480, 160, 2048]
	ResNet101
HALP-60%	64, [64, 32, 192](192), [32, 32, 192], [64, 32, 192], [64, 128, 384](384), [64, 96, 384], [96, 96, 384], [128, 128, 384], [256, 256, 896](896), [256, 96, 896], [256, 128, 896], [256, 96, 896], [256, 96, 896], [256, 128, 896], [256, 96, 896], [256, 96, 896], [256, 96, 896], [256, 96, 896], [256, 64, 896], [256, 64, 896], [256, 96, 896], [256, 64, 896], [256, 64, 896], [256, 96, 896], [256, 64, 896], [256, 96, 896], [256, 96, 896], [256, 96, 896], [256, 64, 896], [256, 128, 896], [256, 64, 896], [512, 512, 2048](2048), [512, 448, 2048], [512, 480, 2048]
HALP-50%	64, [64, 32, 128](128), [0, 32, 128], [64, 32, 128], [64, 128, 384](384), [32, 32, 384], [128, 96, 384], [128, 96, 384], [256, 256, 896](896), [256, 96, 896], [256, 96, 896], [256, 96, 896], [256, 64, 896], [256, 96, 896], [256, 95, 896], [256, 96, 896], [256, 64, 896], [256, 64, 896], [0, 0, 896], [0, 0, 896], [256, 64, 896], [0, 0, 896], [256, 64, 896], [256, 64, 896], [256, 64, 896], [256, 64, 896], [256, 64, 896], [256, 64, 896], [256, 64, 896], [256, 64, 896], [512, 512, 2048](2048), [480, 416, 2048], [512, 416, 2048]
HALP-40%	64, [64, 32, 128](128), [0, 32, 128], [0, 0, 128], [64, 128, 384](384), [32, 64, 384], [64, 96, 384], [64, 128, 384], [256, 256, 896](896), [0, 64, 896], [128, 96, 896], [0, 64, 896], [0, 64, 896], [128, 64, 896], [128, 64, 896], [0, 0, 896], [0, 0, 896], [128, 96, 896], [128, 64, 896], [256, 96, 896], [256, 64, 896], [256, 64, 896], [128, 32, 896], [128, 64, 896], [0, 0, 896], [0, 0, 896], [256, 64, 896], [256, 64, 896], [256, 96, 896], [256, 64, 896], [256, 96, 896], [512, 512, 2048](2048), [512, 352, 2048], [512, 352, 2048]
HALP-30%	64, [64, 32, 128](128), [0, 0, 128], [0, 0, 128], [64, 128, 256](256), [0, 96, 256], [64, 96, 256], [64, 128, 256], [256, 192, 768](768), [0, 64, 768], [128, 64, 768], [0, 64, 768], [0, 64, 768], [128, 64, 768], [128, 64, 768], [0, 0, 768], [0, 0, 768], [128, 64, 768], [128, 96, 768], [256, 64, 768], [0, 0, 768], [0, 0, 768], [128, 32, 768], [128, 64, 768], [128, 64, 768], [128, 64, 768], [0, 0, 768], [0, 0, 768], [256, 64, 768], [256, 0, 768], [256, 64, 768], [512, 512, 2048](2048), [480, 288, 2048], [480, 256, 2048]
	MobileNet-V1
HALP-60%	896, 14, 14, 32, 32, 64, 64, 64, 64, 192, 192, 192, 192, 384, 384, 320, 320, 384, 384, 384, 384, 384, 384, 448, 448, 960, 960
HALP-42%	832, 16, 16, 32, 32, 32, 32, 32, 32, 64, 64, 128, 128, 320, 320, 256, 256, 256, 256, 256, 256, 320, 320, 320, 320, 896, 896
	MobileNet-V2
HALP-75%	16, 16, 16, 64, 64, 24, 64, 64, 24, 112, 112, 32, 128, 128, 32, 128, 128, 32, 192, 192, 64, 384, 384, 64, 352, 352, 64, 352, 352, 64, 384, 384, 96, 512, 512, 96, 512, 512, 96, 576, 576, 160, 960, 960, 160, 960, 960, 160, 960, 960, 320, 1280
HALP-60%	16, 16, 8, 32, 32, 16, 16, 16, 16, 64, 64, 32, 32, 32, 32, 64, 64, 32, 176, 176, 64, 288, 288, 64, 320, 320, 64, 320, 320, 64, 384, 384, 96, 448, 448, 96, 448, 448, 96, 576, 576, 160, 960, 960, 160, 960, 960, 160, 960, 960, 192, 1152

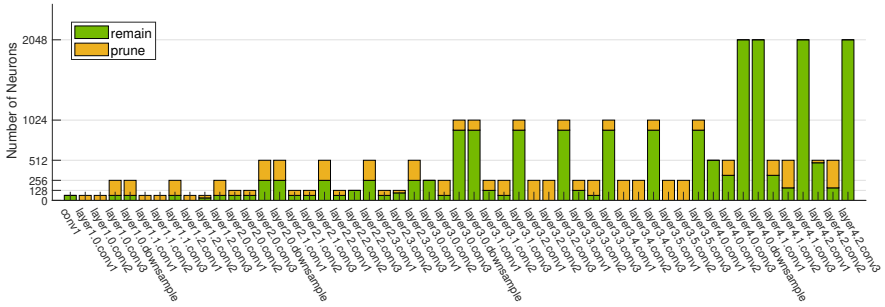
R Discussion on the augmented knapsack solver

Our augmented knapsack solver in Algo. 1 is modified based on the standard dynamic programming solution for the 0-1 knapsack problem [3, 43, 44]. With the original 0-1 knapsack problem formulation, each neuron can be selected to be removed or kept independently. However, the fact is that 1). the layer always has the same latency with the same number of channels remaining no matter which neurons are selected; 2). the neurons with higher importance are favored to maximize the accuracy. So in this latency-aware pruning problem, we add an additional constraint, namely, each neuron can be selected only when all the more importance neurons in the same layer are already kept, leading to Eq. 7. The original solution is modified in lines 6-9 in Algo. 1 accordingly, and is converted into a greedy approximation selection to retain the same time complexity as the original problem. We'll discuss the non-greedy solution later.

In the augmented solver lines 6-9, every time when we decide whether to keep a neuron, we not only compare the total importance score can be reached under the constraint, but also check the inclusion of the "preceding" neurons. This is a greedy selection as it does not consider the potential importance value that the following neurons in the same layer would bring if the current neuron is kept. The



(a) ResNet50 HALP-45%



(b) ResNet50 HALP-30%

Figure 13: Visualization of the pruned ResNet50 structure.

overall time complexity of the solution is $O(N \times C)$ where $N = \sum_{l=1}^{L} N_l$ is the total number of neuron groups in the network and C is the latency constraint. We also provide the non-greedy solution in Algo. 2. In this solution, for each neuron we add a process calculating and comparing the potential importance score that the layer would further bring if the current neuron is selected to be kept. This brings additional $O(N_l)$ complexity for each neuron in layer l. As a result, the total time complexity of the solution increases to $O(\sum_{l=1}^{L} N_l^2 \times C)$. We test both of the solutions and observe similar performance in the ImageNet experiments as shown in Tab. 16. We hypothesize that the efficacy of the greedy approach suffices from the already decreasingly ranked neurons feeding into the solver and the iterative nature during pruning. Thus, the greedy approximation solution Algo. 1 is applied in our method to have a better pruning efficiency.

Table 16: ResNet50 pruning results on ImageNet with the greedy method Algo. 1 and non-greedy method Algo 2

Method	1G model		2G model		3G model	
	Algo. 1	Algo. 2	Algo. 1	Algo. 2	Algo. 1	Algo. 2
Top1 (%)	77.45	77.51	76.56	76.51	74.45	74.51
FPS (img/s)	1203	1185	1672	1688	2597	2524

Algorithm 2 Non-greedy solution for Eq. 7

Input: Importance score $\{\mathcal{I}_l \in \mathbb{R}^{N_l}\}_{l=1}^{L}$ where $\mathcal{I}_l$ is sorted descendingly; Neuron latency contribution $\{c_l \in \mathbb{R}^{N_l}\}_{l=1}^{L}$; Latency constraint C.

maxV $\in \mathbb{R}^{(C+1)}$, keep $\in \mathbb{R}^{L\times(C+1)}$ ▷ maxV[c]: max importance under constraint c; keep[l, c]: # neurons to keep in layer l to achieve maxI[c]
for $l = 1, \ldots, L$ **do**
  for $j = 1, \ldots, N_l$ **do**
    for $c = 1, \ldots, C$ **do**
      $v_{keep} = \mathcal{I}_l^j + \text{maxV}[c - c_l^j],\ v_{prune} = \text{maxV}[c]$ ▷ total importance can achieve under constraint c with object n being kept or not
      flag = False
      for $p_l = j + 1, \ldots, N_l$ **do**
        $v_{potential} = \sum_{j'=j}^{p_l} \mathcal{I}_l^{j'} + \text{maxV}[c - \sum_{j'=j}^{p_l} c_l^{j'}]$ ▷ calculate the potential score this layer would bring if keep this neuron.
        if $v_{potential} > v_{prune}$ **and** keep[$l, c - c_l^j$] $== j - 1$ **then** ▷ check if leads to higher score and more important neurons in layer are kept
          flag = True
          break
        end if
      end for
      if flag == True **then**
        keep[l, c] $=$ j, update_maxV[c] $= v_{keep}$
      else
        keep[l, c] $=$ keep[$l, c-1$], update_maxV[c] $= v_{prune}$
      end if
    end for
    maxV $\leftarrow$ update_maxV
  end for
end for

keep_n $=$ to save the kept neurons in model
for $l = L, \ldots, 1$ **do** ▷ retrieve the set of kept neurons
  p_l $=$ keep[l, C]
  keep_n $\leftarrow$ keep_n $\cup$ $\{p_l$ top ranked neurons in layer $l\}$
  $C \leftarrow C - \sum_{j=1}^{p_l} c_l^j$
end for

Output: Kept important neurons (keep_n).