A Search space details

The following is how we construct the search space for NAS.
Search Space Construction:

- *Fixed Initial Layers:* We started with a Conv2D layer (3x3 kernel, stride 2) in the first stage for quick resolution reduction, followed by NAS-optimized FusedIB blocks (stride 2) in the second stage to balance between efficiency and accuracy.
- *NAS-Driven Optimization:* The NAS process precisely determined the ideal number of UIB blocks and parameter instantiations across the remaining four stages, ensuring an optimal structure for performance.
- *Fixed Head Layers*: We use the same head layer configuration as MobileNet V3.

Observing that pointwise convolutions within UIB blocks tend to exhibit low operational intensity at higher resolutions, we prioritized operations with higher computational density in the initial layers to balance efficiency and accuracy.
Our optimization targets:

- MNv4-Conv-S: Dual goals—285M MACs and 0.2ms latency (Pixel 6 EdgeTPU, 224px inputs).
- MNv4-Conv-M: 0.6ms latency (Pixel 6 EdgeTPU, 256px inputs).
- MNv4-Conv-L: Dual latency goals of 2.3ms (Pixel 6 EdgeTPU) and 2.0ms (Pixel 7 EdgeTPU) with 384px inputs.

To be noted, by restricting our search space to components with well-correlated cost models across devices, we found that EdgeTPU latency optimization directly yields universally efficient models, as demonstrated in later sections.

B Benchmarking methodology

We applied a consistent benchmarking strategy across various mobile platforms, with an exception for the Apple Neural Engine. To enhance efficiency, models were converted to TensorFlow Lite format and quantized to INT8 for mobile CPUs, Hexagon, and EdgeTPUs, while FP16 was used for mobile GPUs. We run each model roughly 1000 times and take the mean latency of those runs. We then repeat that process 5 times for each model and report the median of means. To optimize performance, we set the CPU affinity to the fastest core and use the XNNPACK backend for CPU evaluations. In contrast, for benchmarks on the Apple Neural Engine (conducted on an iPhone 13 with iOS 16.6.1, CoreMLTools 7.1, and Xcode 15.0.1 for profiling), PyTorch models were converted to CoreML's MLProgram format in Float16 precision, with float16 MultiArray inputs to minimize input copying.

C Training setup for ImageNet-1k classification

To enhance model performance, our training recipe incorporates widely adopted data augmentation techniques and regularization methods. For data augmentation, we use Inception Crop [41], horizontal flip, RandAugment [10], Mixup [54], and CutMix [53]. For regularization, we apply L2 normalization and stochastic depth drop [24]. The intensity of augmentation and regularization is adjusted according to model size, as detailed in Tab. 10.

Table 10: Training hyper-parameters for ImageNet-1k classification.

	Conv-S	Conv-M	Hybrid-M	Conv-L	Hybrid-L
Batch size	4096	4096	16384	16384	16384
Peak learning rate	0.002	0.004	0.016	0.004	0.01
Cosine decay alpha	0.0	0.0	0.0	0.0	0.001
Cosine decay epochs	9600	500	500	500	500
Warm-up epochs	5	5	20	20	20
Training epochs	9600	500	500	500	500
AdamW weight decay	0.01	0.1	0.1	0.2	0.2
AdamW β_1	0.6	0.9	0.9	0.9	0.9
AdamW β_2	0.999	0.999	0.999	0.999	0.999
AdamW ϵ	10^{-6}	10^{-7}	10^{-7}	10^{-7}	10^{-7}
EMA decay	0.9999	-	-	-	-
L2-regularization	10^{-5}	-	-	-	-
Gradient clipping	-	-	-	-	-
Label smoothing	0.1	0.1	0.1	0.1	0.1
Dropout	0.3	0.2	0.2	0.2	0.2
Peak Stochastic Depth drop rate	0	0.075	0.075	0.35	0.35
RandAugment probability	0.5	0.7	0.7	1.0	1.0
RandAugment layers	2	2	2	2	2
RandAugment magnitude	9	15	15	15	15
RandAugment excluded ops	Cutout	Cutout	Cutout	Cutout	Cutout
Mixup/Cutmix probability	-	-	-	0.3	0.3
Mixup α	-	-	-	0.8	0.8
Cutmix α	-	-	-	1.0	1.0
Mixup/Cutmix switch probability	-	-	-	0.5	0.5

D Model details

The architecture details of our MNv4 models are described from Tab. 11 to Tab. 15.

Now, let's examine the details of the TuNAS-optimized MNv4-Conv models. TuNAS optimized macro architecture strategically combines four UIB instantiations: Extra DW, ConvNext, IB, and FFN. This combination demonstrates the flexibility of UIB and the importance of using different instantiation blocks in different stages of the network. Specifically, at the start of each searchable stage, where the spatial resolution significantly drops, ExtraDW emerges as the preferred choice. The design of duo depthwise layers in ExtraDW helps to enlarge the receptive field, enhances spatial mixing, and effectively mitigates resolution loss. Similarly, ExtraDW is frequently selected in the early stages of MNv4-Conv models for similar reasons. For the final layers, where preceding layers have conducted substantial spatial mixing, FFN and ConvNext are chosen because channel mixing provides a larger incremental gain.

Table 11: Architecture specification of MNv4-Conv-S.

Input	Block	DW K_1	DW K_2	Expanded Dim	Output Dim	Stride
$224^2 \times 3$	Conv2D	-	3×3	-	32	2
$112^2 \times 32$	FusedIB	-	3×3	32	32	2
$56^2 \times 32$	FusedIB	-	3×3	96	64	2
$28^2 \times 64$	ExtraDW	5×5	5×5	192	96	2
$14^2 \times 96$	IB	-	3×3	192	96	1
$14^2 \times 96$	IB	-	3×3	192	96	1
$14^2 \times 96$	IB	-	3×3	192	96	1
$14^2 \times 96$	IB	-	3×3	192	96	1
$14^2 \times 96$	ConvNext	3×3	-	384	96	1
$14^2 \times 96$	ExtraDW	3×3	3×3	576	128	2
$7^2 \times 128$	ExtraDW	5×5	5×5	512	128	1
$7^2 \times 128$	IB	-	5×5	512	128	1
$7^2 \times 128$	IB	-	5×5	384	128	1
$7^2 \times 128$	IB	-	3×3	512	128	1
$7^2 \times 128$	IB	-	3×3	512	128	1
$7^2 \times 128$	Conv2D	-	1×1	-	960	1
$7^2 \times 960$	AvgPool	-	7×7	-	960	1
$1^2 \times 960$	Conv2D	-	1×1	-	1280	1
$1^2 \times 1280$	Conv2D	-	1×1	-	1000	1

Table 12: Architecture specification of MNv4-Conv-M.

Input	Block	DW K_1	DW K_2	Expanded Dim	Output Dim	Stride
$256^2 \times 3$	Conv2D	-	3×3	-	32	2
$128^2 \times 32$	FusedIB	-	3×3	128	48	2
$64^2 \times 48$	ExtraDW	3×3	5×5	192	80	2
$32^2 \times 80$	ExtraDW	3×3	3×3	160	80	1
$32^2 \times 80$	ExtraDW	3×3	5×5	480	160	2
$16^2 \times 160$	ExtraDW	3×3	3×3	640	160	1
$16^2 \times 160$	ExtraDW	3×3	3×3	640	160	1
$16^2 \times 160$	ExtraDW	3×3	5×5	640	160	1
$16^2 \times 160$	ExtraDW	3×3	3×3	640	160	1
$16^2 \times 160$	ConvNext	3×3	-	640	160	1
$16^2 \times 160$	FFN	-	-	320	160	1
$16^2 \times 160$	ConvNext	3×3	-	640	160	1
$16^2 \times 160$	ExtraDW	5×5	5×5	960	256	2
$8^2 \times 256$	ExtraDW	5×5	5×5	1024	256	1
$8^2 \times 256$	ExtraDW	3×3	5×5	1024	256	1
$8^2 \times 256$	ExtraDW	3×3	5×5	1024	256	1
$8^2 \times 256$	FFN	-	-	1024	256	1
$8^2 \times 256$	ConvNext	3×3	-	1024	256	1
$8^2 \times 256$	ExtraDW	3×3	5×5	512	256	1
$8^2 \times 256$	ExtraDW	5×5	5×5	1024	256	1
$8^2 \times 256$	FFN	-	-	1024	256	1
$8^2 \times 256$	FFN	-	-	1024	256	1
$8^2 \times 256$	ConvNext	5×5	-	512	256	1
$8^2 \times 256$	Conv2D	-	1×1	-	960	1
$8^2 \times 960$	AvgPool	-	8×8	-	960	1
$1^2 \times 960$	Conv2D	-	1×1	-	1280	1
$1^2 \times 1280$	Conv2D	-	1×1	-	1000	1

Table 13: Architecture specification of MNv4-Hybrid-M.

Input	Block	DW K_1	DW K_2	Expanded Dim	Output Dim	Stride
$256^2 \times 3$	Conv2D	-	3×3	-	32	2
$128^2 \times 32$	FusedIB	-	3×3	128	48	2
$64^2 \times 48$	ExtraDW	3×3	5×5	192	80	2
$32^2 \times 80$	ExtraDW	3×3	3×3	160	80	1
$32^2 \times 80$	ExtraDW	3×3	5×5	480	160	2
$16^2 \times 160$	ExtraDW	3×3	3×3	640	160	1
$16^2 \times 160$	ExtraDW	3×3	3×3	640	160	1
$16^2 \times 160$	ExtraDW	3×3	5×5	640	160	1
$16^2 \times 160$	Mobile-MQA	-	-	-	160	1
$16^2 \times 160$	ExtraDW	3×3	3×3	640	160	1
$16^2 \times 160$	Mobile-MQA	-	-	-	160	1
$16^2 \times 160$	ConvNext	3×3	-	640	160	1
$16^2 \times 160$	Mobile-MQA	-	-	-	160	1
$16^2 \times 160$	FFN	-	-	640	160	1
$16^2 \times 160$	Mobile-MQA	-	-	-	160	1
$16^2 \times 160$	ConvNext	3×3	-	640	160	1
$16^2 \times 160$	ExtraDW	5×5	5×5	960	256	2
$8^2 \times 256$	ExtraDW	5×5	5×5	1024	256	1
$8^2 \times 256$	ExtraDW	3×3	5×5	1024	256	1
$8^2 \times 256$	ExtraDW	3×3	5×5	1024	256	1
$8^2 \times 256$	FFN	-	-	1024	256	1
$8^2 \times 256$	ConvNext	3×3	-	1024	256	1
$8^2 \times 256$	ExtraDW	3×3	5×5	512	256	1
$8^2 \times 256$	Mobile-MQA	-	-	-	256	1
$8^2 \times 256$	ExtraDW	5×5	5×5	1024	256	1
$8^2 \times 256$	Mobile-MQA	-	-	-	256	1
$8^2 \times 256$	FFN	-	-	1024	256	1
$8^2 \times 256$	Mobile-MQA	-	-	-	256	1
$8^2 \times 256$	FFN	-	-	1024	256	1
$8^2 \times 256$	Mobile-MQA	-	-	-	256	1
$8^2 \times 256$	ConvNext	5×5	-	1024	256	1
$8^2 \times 256$	Conv2D	-	1×1	-	960	1
$8^2 \times 960$	AvgPool	-	8×8	-	960	1
$1^2 \times 960$	Conv2D	-	1×1	-	1280	1
$1^2 \times 1280$	Conv2D	-	1×1	-	1000	1

Table 14: Architecture specification of MNv4-Conv-L.

Input	Block	DW K_1	DW K_2	Expanded Dim	Output Dim	Stride
$384^2 \times 3$	Conv2D	-	3×3	-	24	2
$192^2 \times 24$	FusedIB	-	3×3	96	48	2
$96^2 \times 48$	ExtraDW	3×3	5×5	192	96	2
$48^2 \times 96$	ExtraDW	3×3	3×3	384	96	1
$48^2 \times 96$	ExtraDW	3×3	5×5	384	192	2
$24^2 \times 192$	ExtraDW	3×3	3×3	768	192	1
$24^2 \times 192$	ExtraDW	3×3	3×3	768	192	1
$24^2 \times 192$	ExtraDW	3×3	3×3	768	192	1
$24^2 \times 192$	ExtraDW	3×3	5×5	768	192	1
$24^2 \times 192$	ExtraDW	5×5	3×3	768	192	1
$24^2 \times 192$	ExtraDW	5×5	3×3	768	192	1
$24^2 \times 192$	ExtraDW	5×5	3×3	768	192	1
$24^2 \times 192$	ExtraDW	5×5	3×3	768	192	1
$24^2 \times 192$	ConvNext	3×3	-	768	192	1
$24^2 \times 192$	ExtraDW	5×5	5×5	768	512	2
$12^2 \times 512$	ExtraDW	5×5	5×5	2048	512	1
$12^2 \times 512$	ExtraDW	5×5	5×5	2048	512	1
$12^2 \times 512$	ExtraDW	5×5	5×5	2048	512	1
$12^2 \times 512$	ConvNext	5×5	-	2048	512	1
$12^2 \times 512$	ExtraDW	5×5	3×3	2048	512	1
$12^2 \times 512$	ConvNext	5×5	-	2048	512	1
$12^2 \times 512$	ConvNext	5×5	-	2048	512	1
$12^2 \times 512$	ExtraDW	5×5	3×3	2048	512	1
$12^2 \times 512$	ExtraDW	5×5	5×5	2048	512	1
$12^2 \times 512$	ConvNext	5×5	-	2048	512	1
$12^2 \times 512$	ConvNext	5×5	-	2048	512	1
$12^2 \times 512$	ConvNext	5×5	-	2048	512	1
$12^2 \times 512$	Conv2D	-	1×1	-	960	1
$12^2 \times 960$	AvgPool	-	12×12	-	960	1
$1^2 \times 960$	Conv2D	-	1×1	-	1280	1
$1^2 \times 1280$	Conv2D	-	1×1	-	1000	1

Table 15: Architecture specification of MNv4-Hybrid-L.

Input	Block	DW K_1	DW K_2	Expanded Dim	Output Dim	Stride
$384^2 \times 3$	Conv2D	-	3×3	-	24	2
$192^2 \times 24$	FusedIB	-	3×3	96	48	2
$96^2 \times 48$	ExtraDW	3×3	5×5	192	96	2
$48^2 \times 96$	ExtraDW	3×3	3×3	384	96	1
$48^2 \times 96$	ExtraDW	3×3	5×5	384	192	2
$24^2 \times 192$	ExtraDW	3×3	3×3	768	192	1
$24^2 \times 192$	ExtraDW	3×3	3×3	768	192	1
$24^2 \times 192$	ExtraDW	3×3	3×3	768	192	1
$24^2 \times 192$	ExtraDW	3×3	5×5	768	192	1
$24^2 \times 192$	ExtraDW	5×5	3×3	768	192	1
$24^2 \times 192$	ExtraDW	5×5	3×3	768	192	1
$24^2 \times 192$	Mobile-MQA	-	-	-	192	1
$24^2 \times 192$	ExtraDW	5×5	3×3	768	192	1
$24^2 \times 192$	Mobile-MQA	-	-	-	192	1
$24^2 \times 192$	ExtraDW	5×5	3×3	768	192	1
$24^2 \times 192$	Mobile-MQA	-	-	-	192	1
$24^2 \times 192$	ExtraDW	5×5	3×3	768	192	1
$24^2 \times 192$	Mobile-MQA	-	-	-	192	1
$24^2 \times 192$	ConvNext	3×3	-	768	192	1
$24^2 \times 192$	ExtraDW	5×5	5×5	768	512	2
$12^2 \times 512$	ExtraDW	5×5	5×5	2048	512	1
$12^2 \times 512$	ExtraDW	5×5	5×5	2048	512	1
$12^2 \times 512$	ExtraDW	5×5	5×5	2048	512	1
$12^2 \times 512$	ConvNext	5×5	-	2048	512	1
$12^2 \times 512$	ExtraDW	5×5	3×3	2048	512	1
$12^2 \times 512$	ConvNext	5×5	-	2048	512	1
$12^2 \times 512$	ConvNext	5×5	-	2048	512	1
$12^2 \times 512$	ExtraDW	5×5	3×3	2048	512	1
$12^2 \times 512$	ExtraDW	5×5	5×5	2048	512	1
$12^2 \times 512$	Mobile-MQA	-	-	-	512	1
$12^2 \times 512$	ConvNext	5×5	-	2048	512	1
$12^2 \times 512$	Mobile-MQA	-	-	-	512	1
$12^2 \times 512$	ConvNext	5×5	-	2048	512	1
$12^2 \times 512$	Mobile-MQA	-	-	-	512	1
$12^2 \times 512$	ConvNext	5×5	-	2048	512	1
$12^2 \times 512$	Mobile-MQA	-	-	-	512	1
$12^2 \times 512$	ConvNext	5×5	-	2048	512	1
$12^2 \times 512$	Conv2D	-	1×1	-	960	1
$12^2 \times 960$	AvgPool	-	12×12	-	960	1
$1^2 \times 960$	Conv2D	-	1×1	-	1280	1
$1^2 \times 1280$	Conv2D	-	1×1	-	1000	1

E Larger Pareto curve

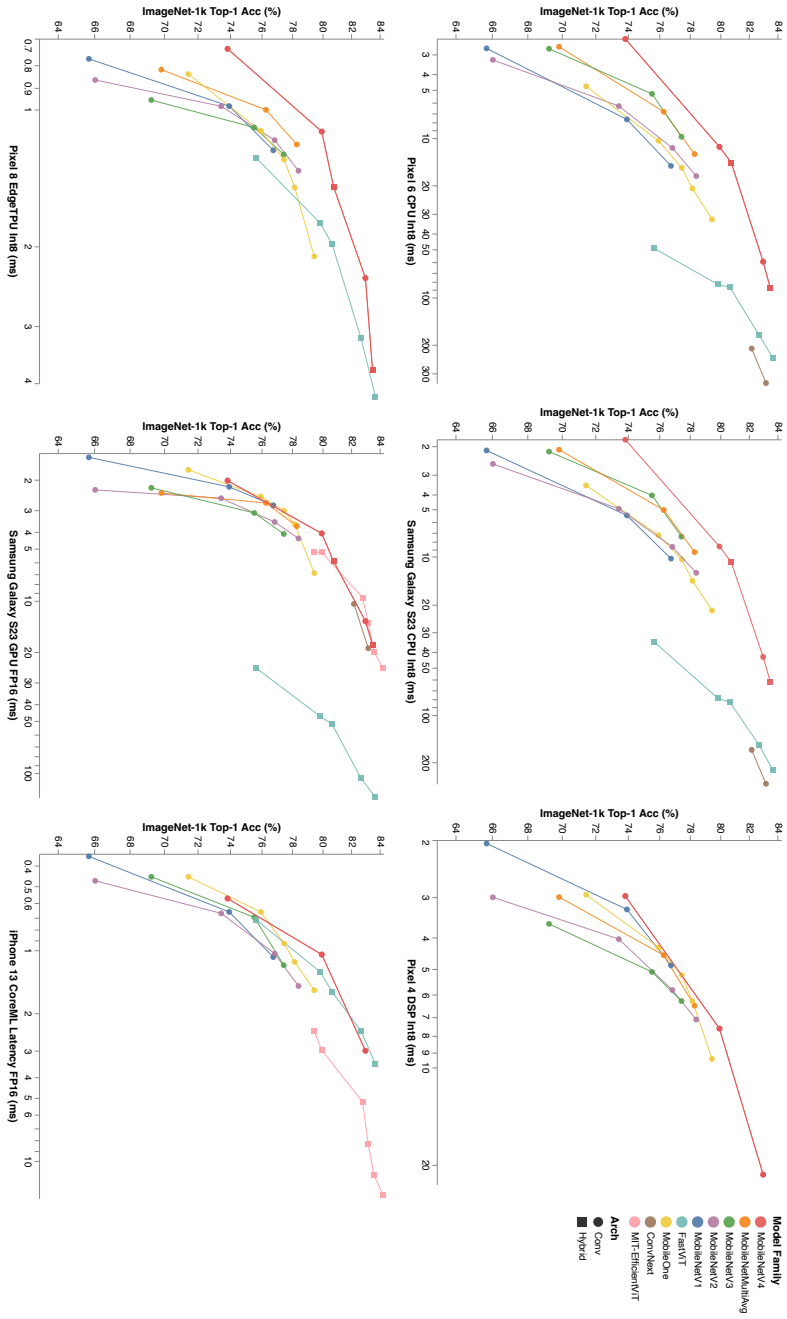

Fig. 5: MNv4 Models are Universally Mostly Pareto Optimal: This is the same chart as Fig. 1, but expanded to be easier to read.

F Additional Roofline Analysis

This extends the analysis from Fig. 3 to include MobileNetV4-Conv-Small (Fig. 7), MobileNetV4-Conv-Medium (Fig. 8), and MobileNetV4-Conv-Large (Fig. 9). These figures also break out each order of magnitude in the sweep from a 0.0 MACs/byte ridge point (MACs-only, infinite memory bandwidth) to a 500.0 MACs/byte ridge point (Accelerator-like, bottlenecked on memory bandwidth). Also included is a correlation analysis between measured latencies, empirically-fit roofline models, and counting MACs (Tab. 16 and Fig. 6).

Table 16: Correlation Between Roofline Models and Real Hardware: The Ridge Points (RPs) for these roofline models were empirically fit to the networks' measured performance. r_s-Roofline is Spearman's rank correlation coefficient between the target's measured latencies and roofline predictions. r_s-MAC is rank correlation between the target's measured latencies and the networks' MAC counts. Counting MACs has high rank correlation for low-RP targets, but much lower rank correlation for high-RP targets. r_s-Roofline is high for all targets under consideration. This shows the accuracy improvement from considering memory bandwidth in addition to MACs when estimating latency. All Ridge Point (RP) values fit in the 0-500 MACs/B range considered in our design analysis. These results are visualized in Fig. 6.

Execution Target	Ridge Point (MACs/B)	r_s-Roofline	r_s-MAC
Pixel 6 CPU (Int8)	31.2	0.973	0.962
Samsung Galaxy S23 CPU (Int8)	39.7	0.962	0.940
Pixel 4 DSP (Int8)	347.3	0.962	0.758
Pixel 8 EdgeTPU (Int8)	433.8	0.973	0.857

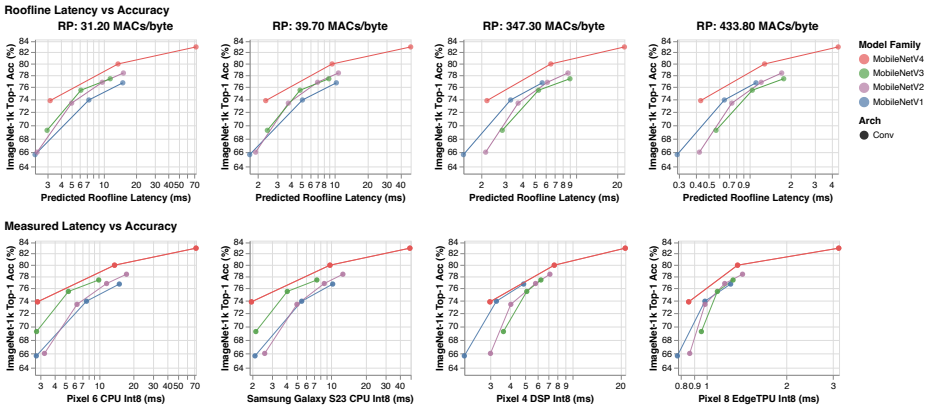

Fig. 6: Correlation Between Roofline Models and Real Hardware: These are the models considered to produce Tab. 16. The roofline models successfully capture the relative ordering of each model family on each hardware target with respect to the Pareto frontier, but each target contains additional nuance that is not captured by the roofline models.

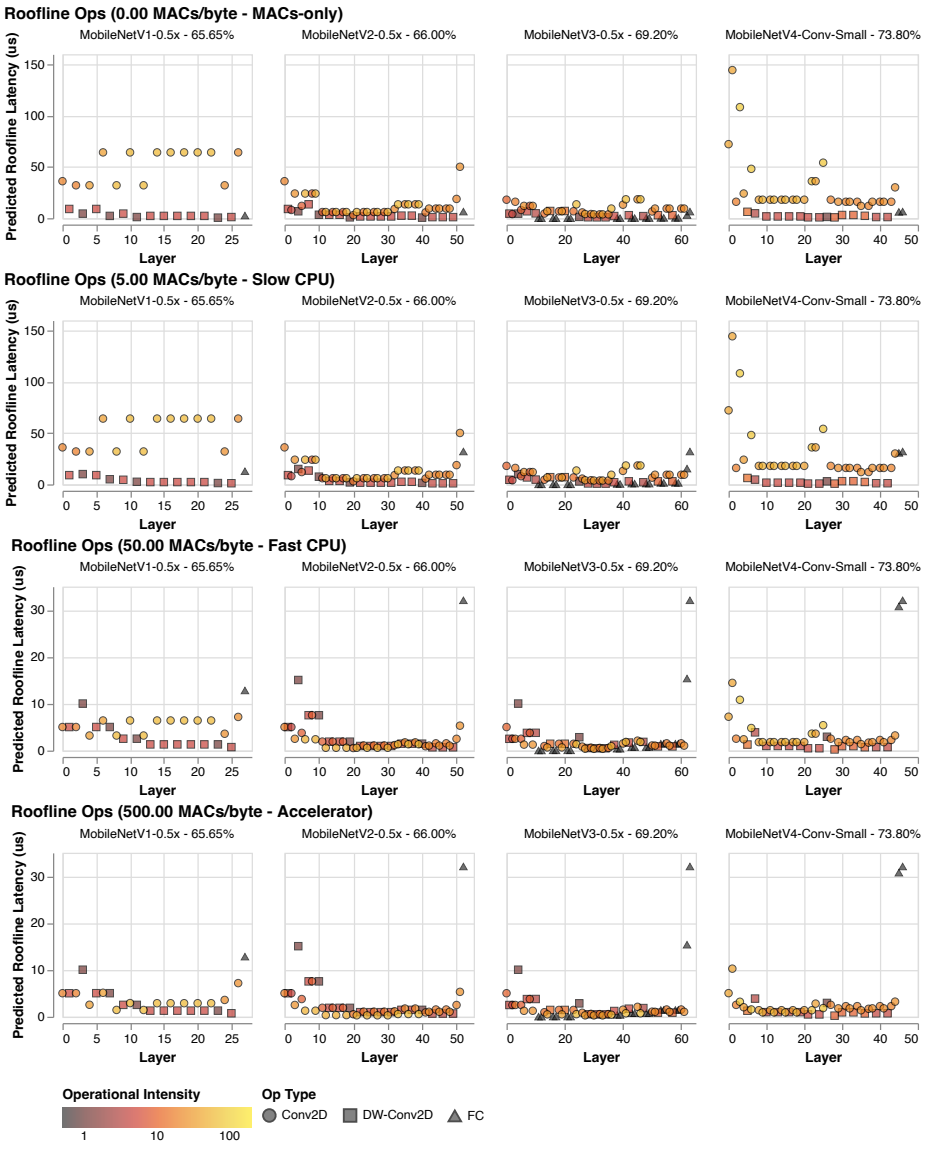

Fig. 7: Roofline RP Sweep Analysis - Small Models

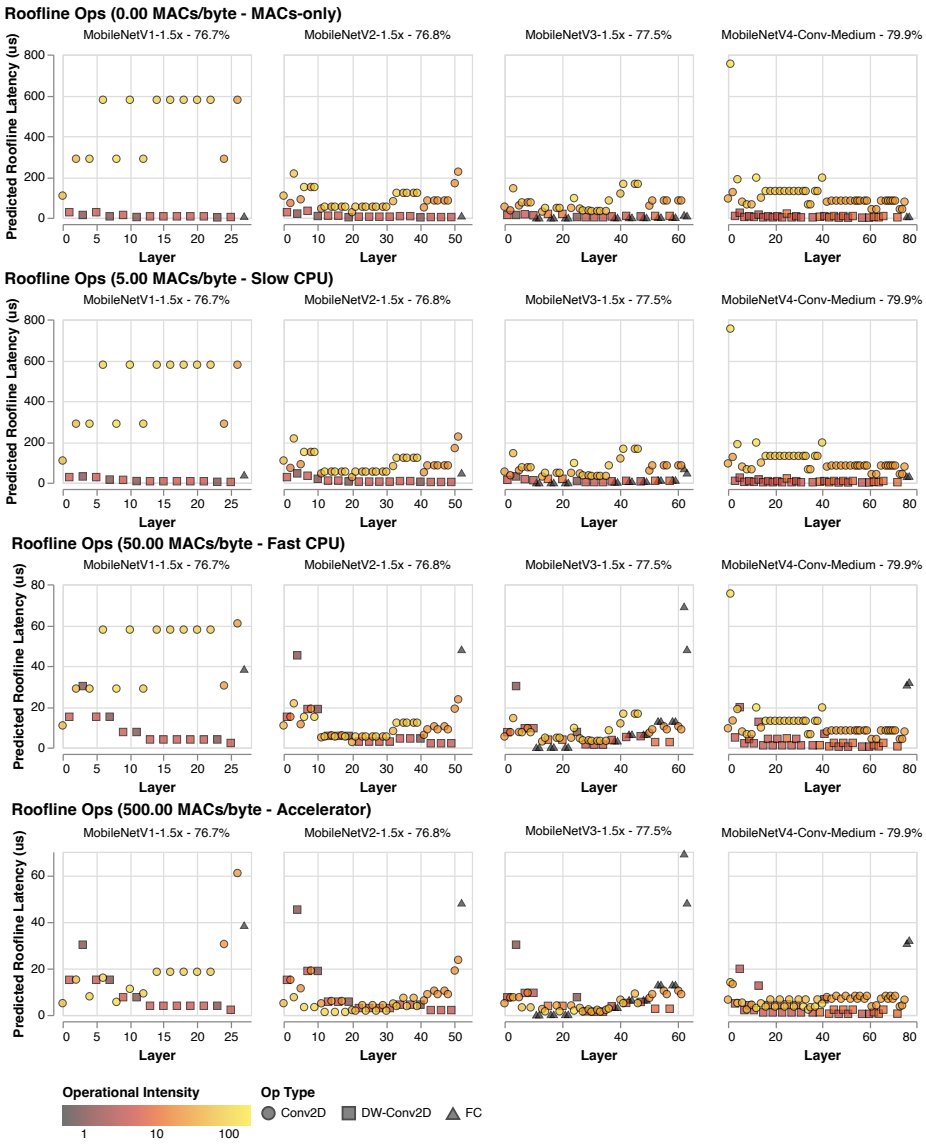

Fig. 8: Roofline RP Sweep Analysis - Medium Models

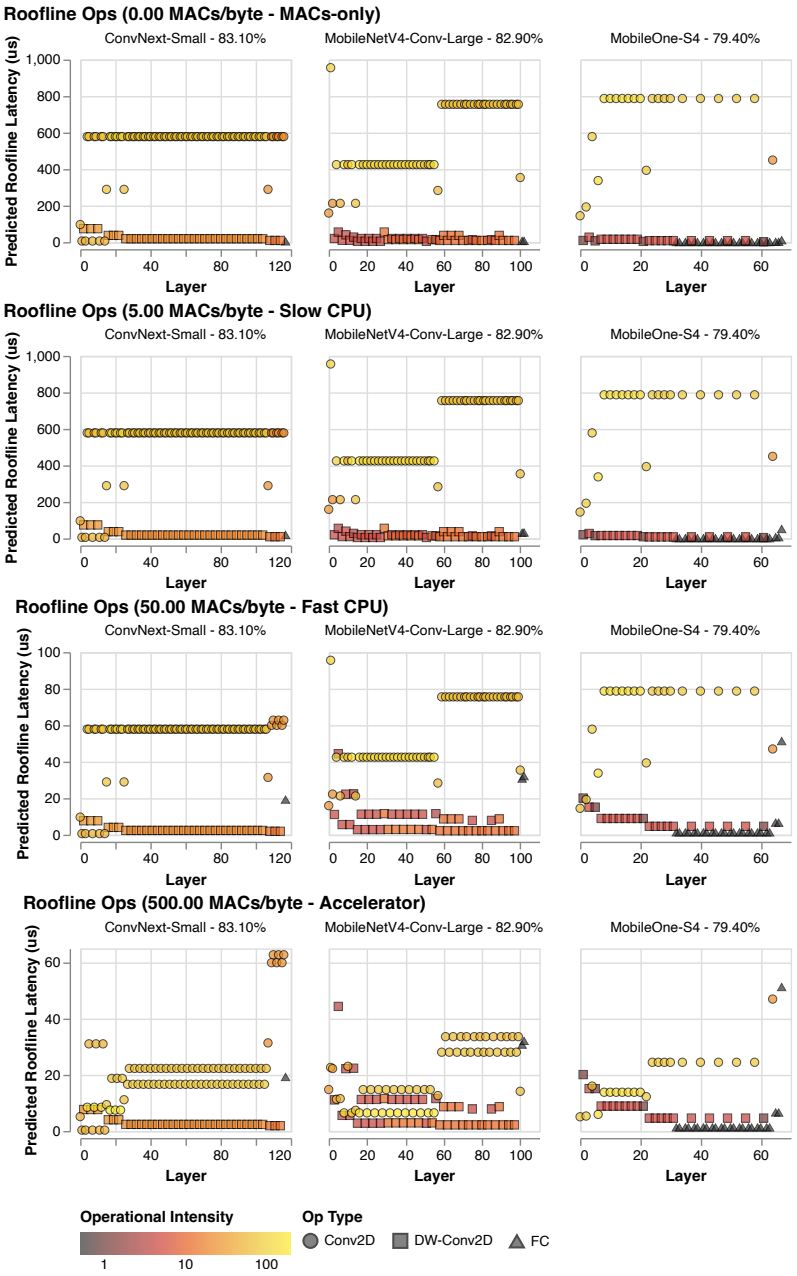

Fig. 9: Roofline RP Sweep Analysis - Large Models: No previous convolutional MobileNets are as big as MobileNetV4-Conv-Large, so this compares to ConvNext-Small (*left*) and MobileOne-S4 (*right*) for contrast. ConvNext-Small is included because it has a similar latency to MobileNetV4-Conv-Large on S23 GPU. MobileOne-S4 is included because it has a similar latency to MobileNetV4-Conv-Large on Pixel 8 EdgeTPU.

```
def MQA(X, M, mask, P_q, P_k, P_v, P_o):
  Q = tf.einsum("bnd,hdk->bhnk", X, P_q)
  K = tf.einsum("bmd,dk->bmk", M, P_k)
  V = tf.einsum("bmd,dv->bmv", M, P_v)
  logits = tf.einsum("bhnk,bmk->bhnm", Q, K)
  weights = tf.softmax(logits + mask)
  O = tf.einsum("bhnm,bmv->bhnv", weights, V)
  Y = tf.einsum("bhnv,hdv->bnd", O, P_o)
```

Fig. 10: Pseudo code of original MQA.

```
def Mobile_MQA(
    X, M, mask, P_q, P_k, P_v, P_o):
  Q = tf.einsum("bnd,dhk->bnhk", X, P_q)
  K = tf.einsum("bmd,dk->bmk", M, P_k)
  V = tf.einsum("bmd,dv->bmv", M, P_v)
  logits = tf.einsum("bnhk,bmk->bnhm", Q, K)
  weights = tf.softmax(logits + mask)
  O = tf.einsum("bnhm,bmv->bnhv", weights, V)
  Y = tf.einsum("bnhv,dhv->bnd", O, P_o)
```

Fig. 11: Pseudo code of optimized Mobile MQA.

G Einsum Optimization

MQA, while faster than MHSA, is still 12x slower than UIB for the same MACs. In the following, we investigate MQA's implementation to uncovers performance issues, and introduce Mobile MQA to improve efficiency.

Einstein summation (Einsum), extensively used in MQA and MHSA implementations across TensorFlow Keras, PyTorch, and JAX, can obscure underlying computational inefficiencies. When executed on-device, Einsum operations are decomposed into sequences of tensor transposes, reshapes, and batched matrix multiplications, aiming to minimize MACs. However, transposes, despite not involving MACs, are highly resource-intensive due to requiring complete tensor reads and writes in memory, significantly impacting performance. Key inefficiencies in Einsum execution include:

Contracted and non-contracted indices are not contiguous in the input: Mobile inference generally does not use a batch dimension. Accordingly an Einsum with two inputs can be translated to a matrix multiplication if both the contracted and non-contracted indices from each input are adjacent to each other. This is because adjacent indices can be very cheaply reshaped to a single index for the purpose of running the matrix multiplication and then very cheaply reshaped back. If the indices are not cleanly split into a set of contracting indices and non-contracting indices, then transposition operations are needed to bring them into this form for matrix multiplication. In the example below, for the slow implementation, two transposes must be introduced: one to transpose O, and one to transpose P_o.

Table 17: Impact of Einsum Optimization: Percentage improvement in Mobile MQA block latency compared to vanilla MQA (without Einsum) in an MNv4-Conv-L model with attention blocks added to the last stage.

model	Top-1 Acc(%)	MACs (G)	Params (M)	EdgeTPU Pixel 7	EdgeTPU Pixel 8
base model	84.88	6.0	30.9	4.31 ms	2.35 ms
+3 MQA (w/o Einsum optimization)	85.22	6.5	34.7	7.00 ms	2.64 ms
+3 Mobile MQA (w/ Einsum optimization)	85.24	6.5	34.7	5.16 ms **(-68.4%)**	2.60 ms **(-13.8%)**

```
# Slow implementation: 2 tranposes needed
Y = tf.einsum("bhnv,hdv->bnd", O, P_o)
# Fast implementation: no tranpose needed
Y = tf.einsum("bnhv,hvd->bnd", O, P_o)
```

Interleaving non-contracting indices or reordering indices in the output: In the slow implementation of the following example, the non-contracted indices h and k from P_q are interleaved with index n from X. As a result, two transposes will be introduced, one before the matrix multiplication to transpose P_q, and one afterwards to swap the indices in the output tensor.

```
# Slow implementation: 2 tranposes needed
Q = tf.einsum("bnd,hdk->bhnk", X, P_q)
# Fast implementation: no tranpose needed
Q = tf.einsum("bnd,dhk->bnhk", X, P_q)
```

In Figure 11, we provide the pseudo-code details of our optimized Mobile MQA and compare them with the original MQA in Figure 10. Mobile MQA achieves significant efficiency improvements by addressing the two main inefficiency issues identified earlier. It requires only a few line of code changes to reorder the indices in the EinSum computations and weight tensor definitions in MQA to eliminate unnecessary transposes. Table 17 demonstrates that we reduced the latency of MQA by threefold on Pixel 7 and by 13.8% on Pixel 8. Additionally, we observed an almost threefold training speedup on Google Tensor v4 using the tf.keras training framework.

H Implementation details of our distillation recipe

To create the offline distillation dataset, we first augment a candidate image and then compress it using JPEG encoding. We then decode the image and run inference using the teacher. The teacher's predicted probability (the softmax output) of the 1k classes is recorded and will be used later as soft labels to train the student. We use EfficientNet L2 [51] as the teacher model. This teacher model has 480 M parameters and 290 G MACs. It achieves 87.5% top-1 accuracy on ImageNet 1k.

During student training, we disable all data augmentation except left-right

flip. We also apply dropout in the final fully connected layer as the only regularization method. We use cross-entropy between the teacher's soft labels and the student's predicted class probability vectors as our training loss. We train the student with the AdamW optimizer. We use a cosine learning rate schedule with warm-up.

Besides the significant boost in model accuracy, there are other benefits of using distillation training. For example, we found that distillation training is 2.5x faster than training on ImageNet-1k. With a batch size of 16k and 400 training epochs, distillation training on MNv4-Conv-Large takes only 2.3 hours on 128 TPU v5e. Finally, distillation training provides significant relief from hyper-parameter tuning.