/nuketorch.ai

Understanding the Shapes of Neural Simulation

Klay Moon
NukeTorch Inc.
klay@nuketorch.ai

Abstract

We call any neural network running on any device a neural simulation. Its matrix products have three axes and an order in time, yet devices are mostly judged by one-axis indicators such as busy time or saturation. We measured the axes one at a time on Apple's Neural Engine (NE) on M4 Max and M5 Max. For FP16 1×1 convolutions with constant weights the rate is set by K: well past a knee K* (1,682 and 2,059 input channels), rate × K is constant, and the constant scales roughly as 1/(activation bytes × weight bytes). Neither program size nor output width sets the rate. Input-dimension shaping, which splits K into P slices whose partial outputs are summed inside one compiled program, raises a long-K layer pair 3.40× and 3.12× at P = 4. We then use Living Calibration to choose P per chip by timing a one-block proxy, and keep the choice in a chip profile. A model fitted on one run and fixed before the next predicted unseen programs with 8.2% and 7.0% median error, and timing its top three candidates kept the best in 7 of 8 groups. On a real-world model, Whisper large-v3-turbo's encoder in FP16, the chosen programs cut transcription time for 1,000 LibriSpeech utterances to 0.785× and 0.781× (21.5% and 21.9% less time), while word error rate went from 3.10% to 3.08% and from 3.08% to 3.06%. A roofline view shows the knee as a fall below the compute roof, and decoding on the bandwidth roof.

1 Introduction

By neural simulation we mean any neural network running on any device. We use this word on purpose: a chip that runs a network simulates it, and how well it does depends on the shapes of the operations it is handed as much as on its peak rate. (In computational neuroscience the term usually means simulating biological neurons, as in NEURON [1] or NEST [2].)

A matrix product has a three-dimensional iteration space: rows M, output width N and reduction length K, the number of products summed into each output value (for a 1×1 convolution, the number of input channels). The order in which hardware walks this space is a fourth axis. Yet devices are usually judged by one-axis indicators such as busy time, which hide the other three. We hypothesized that measuring the true shape of this activity, and shaping the dimensions of the neural simulation to it, would make it run faster.

We first measured the axes one at a time on a device whose behaviour is largely undocumented: Apple's Neural Engine (NE). The NE is reached through Core ML. A program is handed to Apple's compiler in MIL, Core ML's intermediate language [3], and compiled for the NE when it is first loaded. Neither the compiler nor the NE's instruction set is documented. What a machine learning engineer controls is the program: its operations, its constants and their shapes. Which program runs fast has to be found by measurement.

Public guidance and community practice treat size as the limit. Apple's transformer guidance [4] fixes a channels-first layout with 1×1 convolutions for linear layers and splits attention per head for on-chip residency; its vision-transformer guidance [5] splits projections along their outputs. Community runtimes cut models into small programs and split large layers along their outputs [6–8], and a recent study ties a slowdown near N = 2,048 on square operands to the NE's on-chip working set [9]. We held the same view: the size of a program's constant weights decides its speed.

We wrote programs that vary one quantity at a time, and on both chips this view failed. With enough rows per call, the rate of an FP16 1×1 convolution is set by its reduction length K, not by the program's total weight bytes, one convolution's weight bytes or its output width. We rewrote the program to compute the same product as P convolutions over K/P slices of the input channels and add the partial outputs inside the program. The algebra is the split-K of GPU matrix multiplication [10, 11], and NPU simulators model input-channel splits [12], in a different setting. On a GPU, K is split to create parallel work when the output is small. Here the output is large, the NE runs one long-K convolution slowly, and the only lever is the program handed to a closed compiler.

Krizhevsky et al. [13] split AlexNet across two GPUs because one GPU's memory limited the network's size. We apply the same idea inside one processor: each operation is shaped to what the NE runs well, which our model reads as the capacity of its fast memory (Section 4.2).

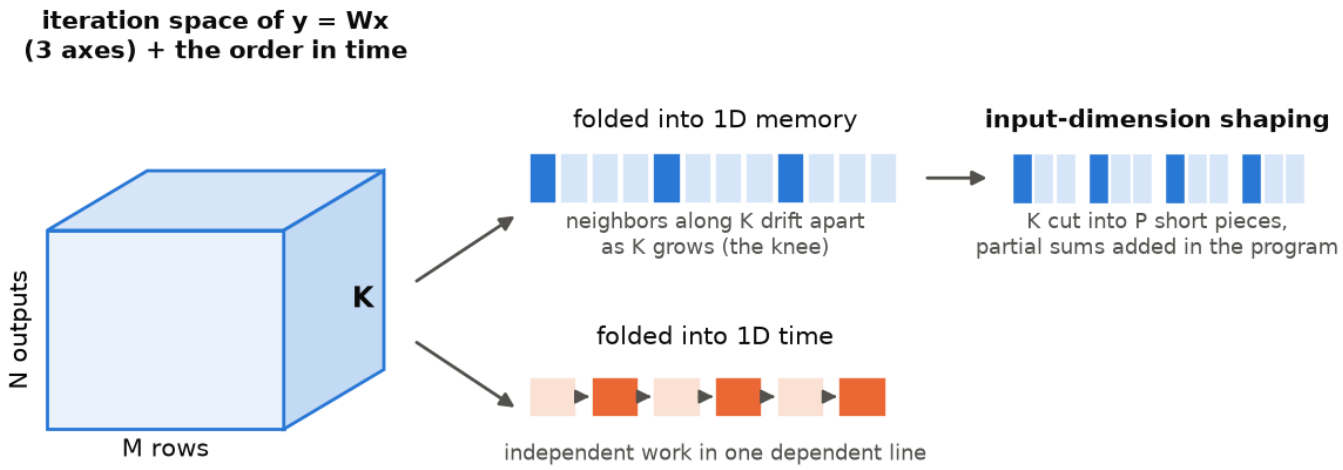


Figure 1. A matrix product has three axes and an order in time. A chip folds them into one-dimensional memory and one-dimensional time. Input-dimension shaping unfolds the reduction in space. Read the figure from left to right. The unfolded form on the right is the speed-up this paper measures.

Our contributions:

1. **The measurement.** On M4 Max and M5 Max the rate of an FP16 constant-weight 1×1 convolution falls with K. We checked this with three controls: equal program bytes at different K, output-channel splits, and the number of encoder blocks per program. A dimension study fits the fall as a knee law. Well past K*, rate × K is constant, and the constant scales roughly as 1/(activation bytes × weight bytes) (Section 4).
2. **Input-dimension shaping.** This is a graph rewrite inside one compiled program. It raises the rate of a long-K layer pair 3.12–3.46× at each chip's best P, and we measured its cost in accumulation error (Section 5).
3. **Living Calibration.** We choose graph constants per chip by timing a one-block proxy on the device, and we keep the choice in a chip profile. We then extend it to predict first and confirm second, with the predictions fixed before timing (Section 6).
4. **An end-to-end test.** We tested Whisper large-v3-turbo's encoder in FP16 and W8A8 on 192 and 1,000 utterances, on both chips. We gated the test by task accuracy rather than by bit identity. The proxy's block times predict the whole program's time per utterance within 0.25% (Section 7).
5. **The boundaries.** A small decoder MLP (K = 3,072) at one row per call gains at most 3.4%. A long-K MLP that decodes at the bandwidth roof gains nothing (Section 8).
6. **A roofline view** of the NE, with every ceiling measured. It shows that input-dimension shaping pays on the compute side, and that busy time near 100% hides the headroom the rewrite recovers (Sections 4.3 and 7.4).

Section 2 describes how we build and time NE programs, and Section 3 defines the terms we use throughout. Section 4 presents the measurement and the knee law. Section 5 presents the rewrite, and Section 6 shows how each chip chooses its own settings. Section 7 gives the end-to-end test on Whisper, and Section 8 shows where the method does not help. Section 9 discusses related work, and Section 10 states the limitations.

2 Setup

This section explains what an NE program is, how we build and time one, and which machines we used. These details matter because every comparison in the paper changes one thing in a program and holds everything else fixed.

Programs. A Core ML program holds MIL text and one file of constant weights [3]. The MIL text lists typed operations in single-assignment form. Activations use the layout [1, K, 1, M], with K channels and M positions. We call the positions rows. A linear layer $y = Wx + b$ becomes a 1×1 convolution whose weight is a constant tensor of shape [N, K, 1, 1]. Apple's compiler builds the NE executable at first load, and Core ML caches it. Whisper large-v3-turbo's encoder has 41,407 operations and 1.27 GB of weights. Its first load took 218.7 s on M4 Max and 173.0 s on M5 Max, and one call then takes 320.3 ms and 261.8 ms.

Our programs. NukeTorch, the engine we develop, builds NE programs directly. A model plugin describes the graph, and the engine's core supplies the operations, including the rewrite of Section 5. At the reference choices, its FP16 Whisper encoder program is identical, operation for operation, to the coremltools conversion. Both have 41,407 operations, their weights are identical, and only the names differ. On 40 inputs, its NE outputs are bit-identical to the converted program's on both chips, with 76,800,000 values compared on each chip. A reference program and a rewritten program therefore differ only where we rewrite them.

Formats. FP16 means FP16 weights and activations. W8A16 means int8 weights and FP16 activations. W8A8 means int8 weights and int8 activations, quantized inside the program. Every rate counts a multiply-add as two operations. We give rates in TFLOPS for FP16 and in TOPS for the 8-bit formats.

Proxies. A proxy is a small program that stands in for a larger one and is quick to compile and time. Here a proxy is either one Whisper encoder block with its real weights, or a stack of MLP layers with a model's shapes (Section 8).

Timing. We load each program through Core ML with the NE as its only compute unit. We call it once to warm it up, and then call it in a loop with bound input and output buffers and uniform$(-1, 1)$ FP16 inputs. We time each probe for 5.0 s and each Whisper proxy for 3 s. The rate is calls per second times operations per call. Probe weights are drawn from N(0, 1/K) and stored in FP16. We timed the same proxy program in two separate series, and the two times differed by a median of 0.04% and at most 0.81% over 32 pairs on both chips. A preflight check before and after each series confirmed a quiet machine: no busy process, mains power, no thermal limit and no other NE compile.

Machines. Both Macs ran on mains power in High Power mode, and we recorded their conditions.

model	chip	cores	memory	macOS	power
Mac16,5	Apple M4 Max	16 (12 Performance and 4 Efficiency)	128 GB	26.7.1 (25G241)	High Power, 140W USB-C Power Adapter
Mac17,6	Apple M5 Max	18 (6 Super and 12 Performance)	48 GB	26.7.1 (25G241)	High Power, 140W USB-C Power Adapter

3 Definitions

This section fixes the vocabulary we use in the rest of the paper.

3.1 Iteration space and schedule. A 1×1 convolution with K input channels, N output channels and M positions (rows) computes

$$y_{n,m} = \sum_{\kappa=1}^{K} W_{n,\kappa}\, x_{\kappa,m} \qquad (1)$$

In words: each output value is a sum of K products, which is why we call K the reduction length. This is a matrix product. Its iteration space, the set of all multiply-adds it must perform, is $\mathcal{I} = [M] \times [N] \times [K]$, with MNK multiply-adds. A schedule $\sigma{:}\mathcal{I} \to \mathbb{N}$ assigns each multiply-add a time step, and the hardware realizes the pair $(\mathcal{I}, \sigma)$. Put simply, a schedule is the order in which the hardware performs these multiply-adds. Busy time keeps only one bit of σ per instant, namely whether any work is scheduled, and drops the rest.

3.2 Work, traffic and intensity. The work is $F = 2MNK$ operations. We count the traffic as

$$Q = w\,KN + a\,MK + o\,MN \qquad (2)$$

bytes. Here w, a and o are the bytes per weight, input and output element (2 for FP16 and 1 for int8), and the sum runs over the convolutions of a program. The arithmetic intensity is $I = F/Q$, the number of operations performed for each byte moved. In words: low intensity means memory traffic limits the speed, and high intensity means the arithmetic units do. Equation 2 counts every operand once per call. An engine that keeps intermediates on chip moves less, so the true intensity is at least I.

3.3 Three roofs. The roofline model [14] draws the best rate a device can reach as a roof over intensity. The roof has a sloped part, where memory bandwidth sets the limit, and a flat part, where the arithmetic units do. For format f the attainable rate is bounded by

$$R \;\le\; \min\Big(BI,\; R_f^{\text{knee}}(k)\Big), \qquad R_f^{\text{knee}}(k) = \min\Big(C_f,\; \frac{A_f}{k}\Big) \qquad (3)$$

Here C_f is the compute ceiling of format f and B is the main-memory bandwidth. R_f^{knee} is the knee roof of a convolution whose reduction length as it runs is k: $k = K$ unsplit, and $k = K/P$ for each piece of the rewrite in 3.4. For a chain of convolutions, the knee roof is their operation-weighted harmonic mean. C_f and B are rates we measured (Section 4.3), and A_f, in TFLOPS $\times$ channels, is fitted (Section 4.2). $\min(C_f, BI)$ is the roofline model, while the term A_f/k depends on the layer's shape and not on its intensity. In words: a convolution can be no faster than the tightest of three limits, namely the arithmetic units, the memory, and, for long sums, the knee.

3.4 Input-dimension shaping. The rewrite cuts one long sum into P shorter ones. Each piece produces a partial sum, which is the output that its slice of the input channels alone would give. Adding the P partial sums gives the full output. Formally, for P dividing K, we write $W = [W_1 \cdots W_P]$ by column blocks of width K/P and $x = [x_1; \ldots; x_P]$ by the matching channel blocks:

$$y \;=\; Wx + b \;=\; \left(\sum_{p=1}^{P} W_p\, x_p\right) + b, \qquad W_p \in \mathbb{R}^{N \times K/P} \qquad (4)$$

In words: the same result is computed as P shorter sums whose results are added at the end. Each piece reduces over K/P channels, so its knee roof rises to $\min(C_f, PA_f/K)$. The price is $P-1$ additions of $N \times M$ values (Equation 6) and $P-1$ extra roundings of the partial outputs (Section 5.3).

4 The finding: K sets the rate

This section shows the paper's central measurement: on the NE, the length of each convolution's sum decides its speed, while program size and output width do not. This matters because it tells a model builder which dimension to keep short.

4.1 Controls

Table 1 lists the FP16 probes. Each probe is a chain of 1×1 convolutions in → out → in, and the pair is repeated the number of times shown after ×.

Table 1. FP16 probe rates in TFLOPS.

layers (in → out → in) × pairs	K	MB per layer	MB per program	rows	M4 Max	M5 Max
1,024 → 1,024 → 1,024 × 4	1,024	2.1	16.8	2,048	15.32	17.33
2,048 → 2,048 → 2,048 × 2	2,048	8.4	33.6	2,048	13.69	17.63
2,048 → 2,048 → 2,048 × 3	2,048	8.4	50.3	2,048	13.96	18.17
2,048 → 2,048 → 2,048 × 4	2,048	8.4	67.1	2,048	14.29	18.48
2,048 → 2,048 → 2,048 × 6	2,048	8.4	100.7	2,048	14.61	18.78
2,560 → 9,728 → 2,560 × 1	2,560 / 9,728	49.8	99.6	2,048	3.34	4.35
2,560 → 9,728 → 2,560 × 2	2,560 / 9,728	49.8	199.2	256	3.37	4.30
1,024 → 1,024 → 1,024 × 4	1,024	2.1	16.8	256	11.28	12.55
1,280 → 5,120 → 1,280 × 4	1,280 / 5,120	13.1	104.9	1,500	7.91	10.27
2,896 → 2,896 → 2,896 × 2	2,896	16.8	67.1	2,048	8.87	11.49
4,096 → 4,096 → 4,096 × 1	4,096	33.6	67.1	2,048	7.46	9.64

- **K.** On M4 Max, square layers run at 15.32 and 14.61 TFLOPS at K = 1,024 and 2,048, and then at 8.87 at K = 2,896 and 7.46 at 4,096. On M5 Max they run at 17.33, 18.78, 11.49 and 9.64. Whisper's MLP shape (1,280 ↔ 5,120) runs at 7.91 and 10.27, and the 2,560 ↔ 9,728 pair runs at 3.34 and 4.35.
- **Program bytes.** At K = 2,048, programs of 33.6 to 100.7 MB run at 13.69–14.61 (M4 Max) and 17.63–18.78 (M5 Max). The rate does not fall as the bytes grow. At equal bytes (99.6 against 100.7 MB), the long-K pair runs 4.37× (M4 Max) and 4.31× (M5 Max) slower.
- **Output width and per-convolution bytes.** We split the pair's layers into 2, 4 or 8 convolutions by output channels. The outputs stay bit-identical, and the rate stays within 1.4% (Table 4). These splits also cut each convolution's weight bytes 2–8×, so per-convolution bytes do not set the rate either. Splitting by input channels, which shortens K by the same factors, does set it (Section 5).
- **Program size in a real model.** We put 1 to 8 Whisper encoder blocks in one program (39.3 to 314.8 MB in FP16). This changes the time per block by -2.7% on M4 Max and -2.9% on M5 Max (Table 7).
- **Rows.** With 256 rows instead of 2,048, the K = 1,024 chain runs at 11.28 instead of 15.32 (M4 Max) and 12.55 instead of 17.33 (M5 Max). Fewer rows per call lower the rate, which is a separate effect.
- **W8A8.** The 8-bit form falls much less. The 2,560 → 9,728 pair runs at 0.82 (M4 Max) and 0.86 (M5 Max) of the K = 2,048 rate, against 0.23 and 0.24 in FP16 (Table A1).

4.2 The Knee Law: a constant past the knee

A finer sweep turns the observation into a simple law. The law has two numbers per chip and format: a peak rate, and a constant that sets how the rate falls past the knee.

The dimension study. Run 1 timed 56 programs per chip for the fit, of three kinds. The first were square

1×1-convolution chains (K = N) at 2,048 rows, for 13 values of K from 512 to 8,192 in each of the three formats (39 programs). The second were single FP16 layers at K = 2,048 and 4,096 with 512 to 8,192 output channels (8). The third was one layer with K = N = 8,192, cut into P = 1, 2, 4, 8, 16 pieces in FP16 and W8A8 (9). A further 12 programs vary the rows (64 to 4,096) at K = 2,048 and 4,096, and 8 more measure accumulation error (Section 5.3). Run 2 timed the 51 programs of Section 6.2.

The law. The study fits each convolution's rate as a smooth knee,

$$r_f(k) \;=\; \left(r_{\text{peak},\,f}^{-n} + (k/A_f)^n\right)^{-1/n} \qquad (5)$$

In words: well below the knee $K_f^* = A_f/r_{\text{peak},\,f}$, a convolution runs close to $r_{\text{peak},\,f}$. Well past it, the rate is close to A_f/k, so doubling the length of the sum halves the rate. The exponent n sets how sharp the corner is. At K_f^* itself the rate is $2^{-1/n}\, r_{\text{peak},\,f}$, which is 0.80 of r_{peak} on M4 Max and 1.00 on M5 Max. We use this rate in a program-time model

$$T \;=\; T_0 \;+\; \gamma_{\text{io}}\, Q_{\text{io}} \;+\; \sum_{\text{convolutions, pieces}} \frac{2kNM}{r_f(k)} \;+\; (P-1)\,\gamma_{\text{add}}\, NM \;+\; [f = \text{W8A8}]\,\gamma_{\text{code}}\, KM \qquad (6)$$

Here T_0 is the measured time of a minimal program, Q_{io} is the program's input and output in megabytes, and P is the number of pieces of a convolution. In words: a program's time is a fixed start-up cost, plus the time to move its inputs and outputs, plus each convolution's time at its knee-limited rate, plus the cost of the extra additions and of the 8-bit coding. We fitted the other parameters ($r_{\text{peak},\,f}$, A_f, n and the three γ) by least squares on log time over the 56 programs per chip. The fit gives $n = 3.08$ on M4 Max and 236 on M5 Max, which is a nearly sharp corner. The rms log residual is 0.100 and 0.074, and the largest residual is 26.2% and 18.5%. Constraining $A_{\text{W8A16}} = 2A_{\text{FP16}}$ and $A_{\text{W8A8}} = 4A_{\text{FP16}}$ costs little (rms 0.103 and 0.081).

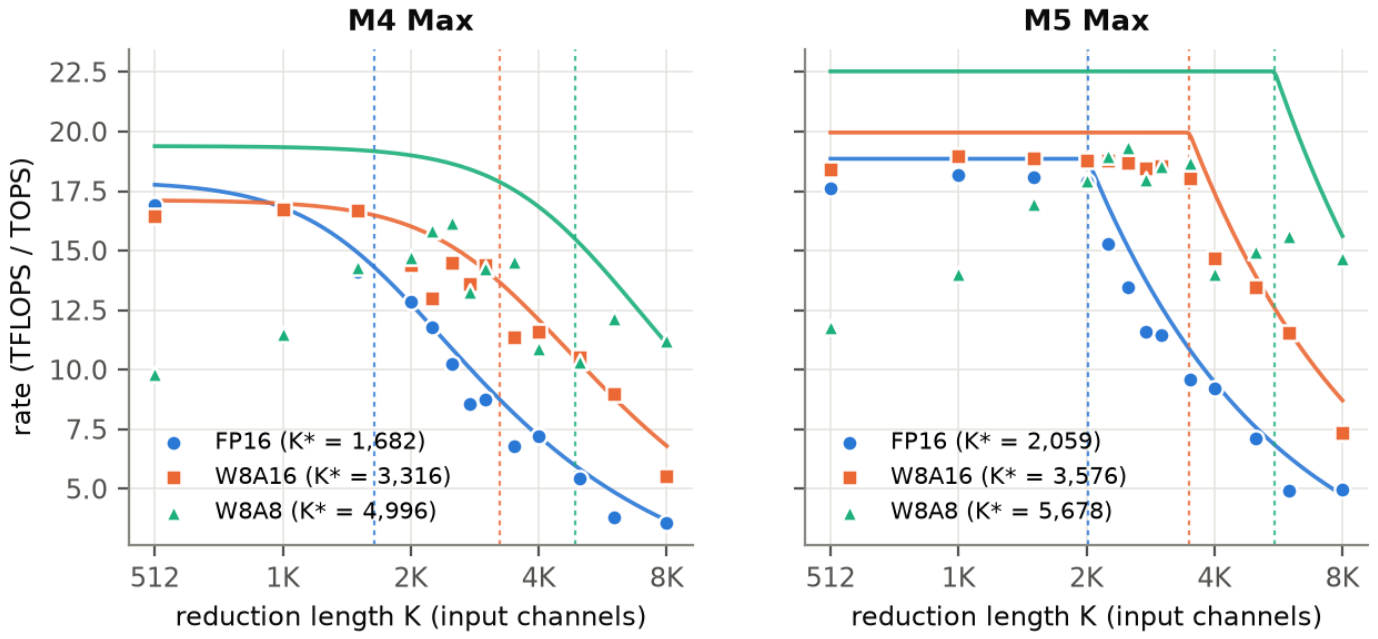


Figure 2. The rate of square 1×1-convolution chains (2,048 rows) against their reduction length K, for three formats on both chips (dimension study, run 1). The lines show the fitted per-convolution rate r(k) of Equation 5, and the dashed verticals show the fitted knees K*. W8A8 programs also pay the activation coding of Equation 6, so they sit below r(k). Each format stays flat up to its knee and then falls roughly as 1/K. Smaller elements move the knee to longer K.

Table 2. Knee-law constants from a free fit per format, over 56 programs per chip. The peak r_{peak} is in TFLOPS (FP16) or TOPS, A is in TFLOPS × channels, and K* is in channels.

chip	format	bytes (activation, weight)	peak	A	knee K* = A / peak	A / A(FP16)
M4 Max	FP16	2, 2	17.9	30,134	1,682	1.00
M4 Max	W8A16	2, 1	17.1	56,785	3,316	1.88
M4 Max	W8A8	1, 1	19.4	96,885	4,996	3.22
M5 Max	FP16	2, 2	18.9	38,834	2,059	1.00
M5 Max	W8A16	2, 1	20.0	71,371	3,576	1.84
M5 Max	W8A8	1, 1	22.5	127,908	5,678	3.29

A single constant for the three formats, scaled by 1/(activation bytes × weight bytes), fits about as well as a free constant per format. Halving the weight bytes (W8A16) multiplies A by 1.84–1.88, and halving both (W8A8) multiplies it by 3.22–3.29, against 2 and 4 for the exact law. Output width shows no knee. At K = 2,048 and 4,096, the rate at 8,192 output channels is at least the rate at 2,048 on both chips. Rows change the K effect in one place only. From 256 rows up, K = 4,096 runs at 0.40–0.54 of K = 2,048's rate on both chips. At 64 rows, the M5 Max runs it at 0.98 of that rate (7.9 against 8.0 TFLOPS), and the M4 Max runs it at 0.35 (2.1 against 5.9).

Two knees on M4 Max. On M4 Max the fitted corner is soft, so r_{peak} is an asymptote rather than a reached rate. The FP16 peak of 17.9 TFLOPS lies 17% above the best rate we measured on any FP16 probe (15.32 TFLOPS, Table 3). In Equation 3, the knee term therefore meets the measured ceiling C_f at A_f/C_f = 1,966 channels on M4 Max, not at K* = 1,682. On M5 Max the corner is sharp, and the two nearly agree (2,068

against 2,059, with r_{peak} 0.4% above C_f). K* in Table 2 and in the abstract is the fit's own corner, $A_f/r_{\text{peak},f}$.

A memory reading. Suppose a core computes an $M_t \times N_t$ block of outputs with the whole reduction resident. It then holds KM_t activations and KN_t weights in a fast memory of S bytes. Every block reloads its panels at bandwidth B_s, the bandwidth that refills the fast memory. This is not the main-memory bandwidth B of Equation 3. The best tiles then move $4awK^2MN/S$ bytes for $2KMN$ operations, so

$$R = \frac{B_s S}{2awK}, \qquad K^* = \frac{B_s S}{2aw r_{\text{peak}}} \qquad (7)$$

This has both signatures of Equation 5: a constant $A = B_s S/(2aw)$ past the knee, and scaling as $1/(aw)$. In words: if a core must keep the whole sum at hand in its fast memory, a longer sum leaves room for fewer outputs at a time. More data is then fetched again, and the rate falls as $1/K$. Smaller elements leave more room. Tiling the reduction removes the K dependence, at the price of partial sums (the Hong–Kung bound [15]). Read through Equation 7, the FP16 constants give $B_s S = 2.41 \times 10^{17}$ B²/s on M4 Max and 3.11×10^{17} on M5 Max, which are 1.29× apart. So the M5 Max has more fast memory, more bandwidth to it, or both. The model cannot separate the two. The 64-row result fits the same reading. A 512 KB activation panel (K = 4,096, 64 rows, FP16) behaves as if it fits on the M5 Max and not on the M4 Max. These are readings of the measurements through the model, not tests of the hardware.

4.3 A roofline view: the knee is not intensity

The roofline [14] is standard in high-performance computing. On-device ML reporting usually reads an engine's busy percentage instead, for example a GPU timeline, Activity Monitor's GPU history or powermetrics residency. A one-axis indicator such as busy time is a projection. It drops the axes on which the loss lives: how much of the compute is used, how many bytes move, how much runs at the same time and how much work is repeated. The roofline adds two axes, arithmetic intensity and the two measured ceilings. Adding the layer's shape (K, rows, output width) then exposes the input-dimension knee, which neither busy time nor intensity shows. The payoff is that these views describe the hardware itself. Past the knee, rate × K is constant on each chip. Read through the memory model, that constant is the product of the fast memory's capacity and its bandwidth (Section 4.2).

Every ceiling in Table 3 is the best rate we measured, so each roof is a lower bound of the hardware's. Every point in Figure 3 is one measured operation (Table A3). The bandwidth roof comes from the Fish MLP proxy of Section 8, which moves 597.7 MB of weights per call. The NE's ridge, where the two roofs meet, lies at 125 operations per byte on M4 Max and 116 on M5 Max.

Table 3. NE ceilings and their sources.

chip	ceiling	value	source
M4 Max	compute, FP16	15.32 TFLOPS	highest FP16 probe rate (1,024 → 1,024 → 1,024 × 4, 2,048 rows)
M4 Max	compute, W8A8	15.13 TOPS	highest W8A8 probe rate (2,048 → 2,048 → 2,048 × 4, 2,048 rows)
M4 Max	bandwidth	122.9 GB/s	Fish MLP proxy at 64 rows, P = 8: bytes moved per call over time; its 597.7 MB of weights alone read at up to 119.2 GB/s
M5 Max	compute, FP16	18.78 TFLOPS	highest FP16 probe rate (2,048 → 2,048 → 2,048 × 6, 2,048 rows)
M5 Max	compute, W8A8	18.27 TOPS	highest W8A8 probe rate (2,048 → 2,048 → 2,048 × 4, 2,048 rows)
M5 Max	bandwidth	161.3 GB/s	Fish MLP proxy at 64 rows, P = 2: bytes moved per call over time; its 597.7 MB of weights alone read at up to 156.5 GB/s

- **The knee is not explained by intensity.** Every FP16 probe at 2,048 rows has I = 410–1,024 operations per byte. This is right of the ridge, where the roofline predicts the compute roof. Across the square layers, I grows with K (410 at K = 1,024 to 1,024 at K = 4,096). Yet the rate falls from 100% to 49% of the roof on M4 Max and from 92% to 51% on M5 Max. The long-K pair, at I = 1,019, reaches 22% and 23%.
- **The knee law is a third roof.** Past K*, the rate stays near A/K (Equation 3), a ceiling set by the layer's shape. Our probes sit at most 1.7% above the knee roof fitted on the dimension study's chains. Each of the P pieces of a split reduces over K/P channels, so its knee roof is P times higher, up to the compute peak (Section 5.2).

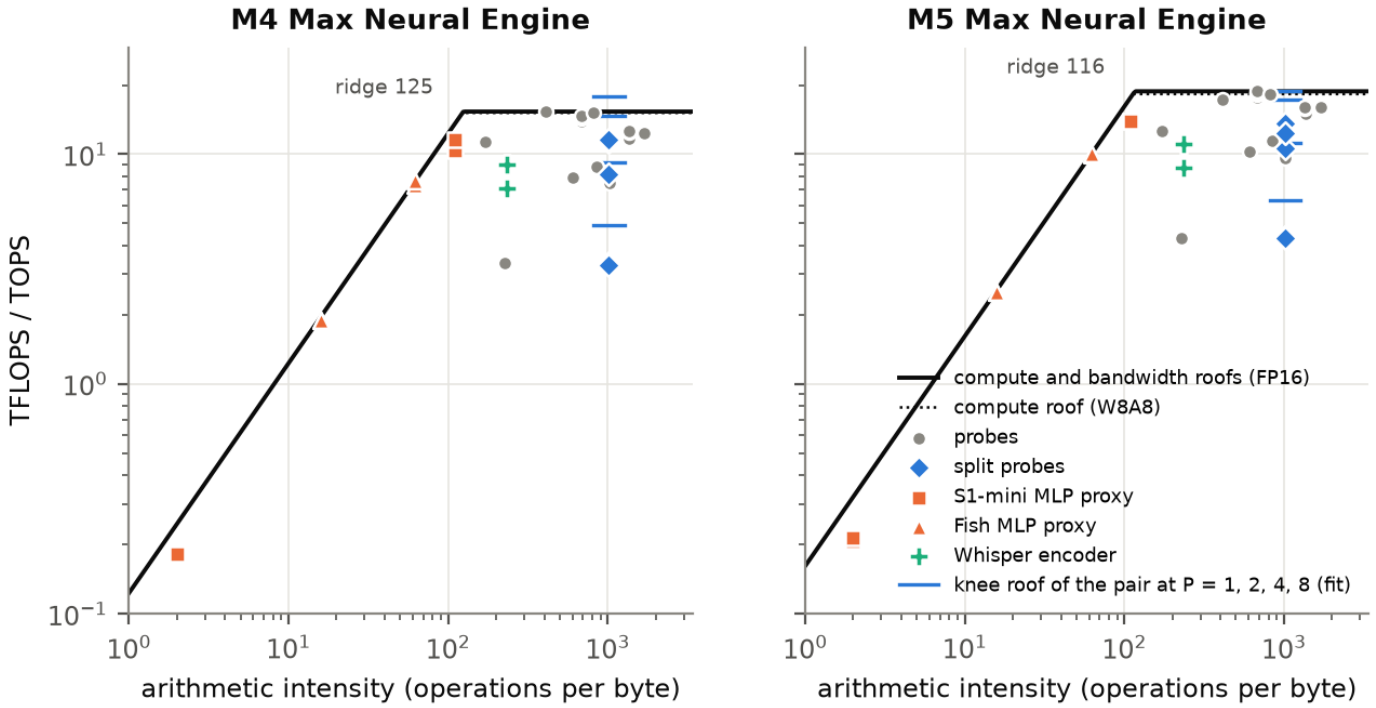


Figure 3. Roofline of the NE on both chips, with every measured operation (Table A3). The black lines are the measured compute and bandwidth roofs. The blue segments are the fitted knee roof of the long-K layer pair at P = 1, 2, 4, 8 pieces, drawn at the pair's intensity. Points far below the flat roof on the right are held back by the knee, not by memory. The blue segments show the knee roof rising as the pair is cut into more pieces. The decoding proxies on the left (Section 8) sit on the sloped memory roof.

5 Input-dimension shaping

This section turns the finding into a fix. If long sums are slow, we cut each long sum into shorter ones inside the same program and add the partial sums at the end. We call this input-dimension shaping.

5.1 The rewrite. Input-dimension shaping rewrites a layer so that no convolution reduces over more input channels than the device runs well. Here it takes the form of Equation 4. In the program it becomes one `split` of x along channels into P equal parts, P `conv` operations with constant weights W_p of shape [N, K/P, 1, 1], and $P-1$ `add` operations that sum in order $p = 1, \ldots, P$. The bias is added once, after the sum (Figure 4). Weight bytes do not change, but the operation count grows: the probe pair goes from 14 to 36 / 68 / 132 operations at P = 2 / 4 / 8. The engine applies the rewrite to fc2, the Whisper encoder's only layer with K above 1,280 (K = 5,120).

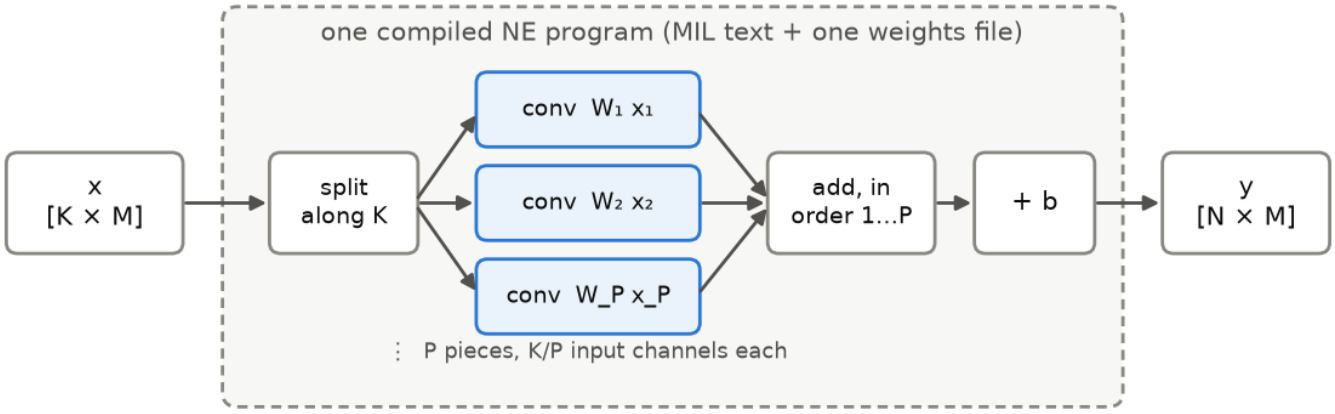


Figure 4. Input-dimension shaping inside one compiled program. The activation is split along K, and each piece meets the matching columns of W as a constant-weight 1×1 convolution. The P partial outputs are then added in order, before the bias. All pieces and additions run inside the same NE program.

5.2 Rate. Table 4 and Figure 5 compare the two ways of cutting the 2,560 → 9,728 → 2,560 pair.

Table 4. The pair at 2,048 rows, split by output channels or by input channels. The rate is in TFLOPS, followed by its ratio to the whole program. The last column gives the difference from the whole program: the largest $|\Delta|$, in FP16 units in the last place at the output's largest magnitude (3.17), and the share of output values that differ. The differences are identical on both chips.

layers split	ops	M4 Max	M5 Max	difference from whole
whole	14	3.34 (1.00×)	4.35 (1.00×)	identical
output ÷ 2	34	3.33 (1.00×)	4.34 (1.00×)	identical
output ÷ 4	62	3.33 (1.00×)	4.34 (1.00×)	identical
output ÷ 8	118	3.30 (0.99×)	4.29 (0.99×)	identical
input ÷ 2	36	8.16 (2.44×)	10.60 (2.44×)	0.0029 (1.5 ulp); 71.3%
input ÷ 4	68	11.36 (3.40×)	13.55 (3.12×)	0.0039 (2.0 ulp); 79.1%
input ÷ 8	132	11.55 (3.46×)	12.40 (2.85×)	0.0059 (3.0 ulp); 85.0%

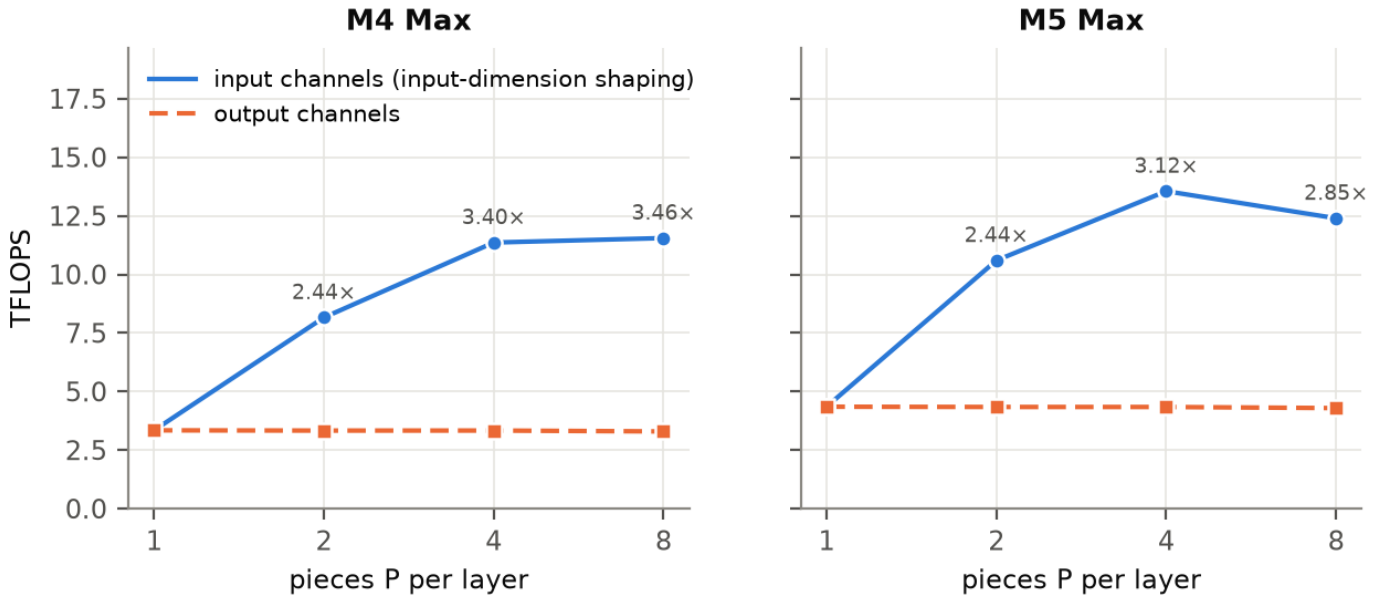


Figure 5. The 2,560 → 9,728 → 2,560 FP16 layer pair (2,048 rows), cut into P pieces per layer. We cut either by input channels, with the partial outputs summed in the program, or by output channels, with the outputs concatenated. The labels give the rate relative to the whole program. Cutting by input channels raises the rate several-fold, while cutting by output channels leaves it flat. So the length of the sum, not the size of each piece, sets the speed.

On the roofline (Figure 3), the rewrite moves no extra weights, so the split pair stays at the whole pair's intensity. It rises from 22% to 74% of the roof at P = 4 on M4 Max, and from 23% to 72% on M5 Max. Output-channel splits stay at 22–23%.

5.3 Accumulation error. Measured against an FP64 product of the same FP16 operands, the NE's error above the FP16 rounding of its output grows as $\sqrt{K}$. The constant per $\sqrt{K}$ holds across K = 1,024 to 12,288. A sequential accumulator with a t-bit significand behaves this way, and a CPU model of the same statistics matches the NE at $t \approx$ 15.2 bits. Under Equation 4, each partial output is an FP16 tensor in the program (t = 11). The rewrite therefore changes the order of accumulation and adds P − 1 FP16 roundings per output element [16]. Its outputs are not bit-identical to the unsplit program's (Table 4). We tested the hypothesis that summing P partials approaches pairwise summation and so lowers the error. It failed. The NE accumulates more finely than FP16, so moving part of the sum into FP16 partials raises the error with P. At K = 12,288, the rms error relative to the rms output is 8.95e-04 unsplit, and 9.30e-04, 9.81e-04, 1.08e-03 and 1.26e-03 at P = 2, 4, 8 and 16 (identical on both chips). We therefore gate the rewrite by task accuracy against the unsplit reference program. We compare transcripts token by token and measure word error rate on the same utterances (Section 7.5). The chosen split became the default only after the 1,000-utterance check on both chips.

6 Living Calibration

This section explains how a program chooses its own settings on each chip instead of using fixed values written by hand. It also shows how a model makes that choice cheaper.

Choosing settings by timing them on the device is an established practice, as in ATLAS [17], cuDNN's algorithm search [18], and TVM and Ansor [19, 20]. We apply it to the graph constants of a compiled NE

program, and we rank the candidates with a fitted model before timing them.

6.1 Choosing by timing

Living Calibration times a few candidate programs on the device and keeps the best. Each candidate is timed on a proxy (Section 2). Here the proxy is encoder block 0 alone, with the checkpoint's weights. Take candidates $c_1, \ldots, c_J$ in order of preference, with the reference or the cheapest first, and let t_j be their measured proxy times. The choice is

$$j^{*} \;=\; \min\left\{\, j : t_j \le (1+\tau)\min_i t_i \,\right\}, \qquad \tau = 0.015 \qquad (8)$$

In words: among the candidates that are nearly as fast as the fastest, we take the most preferred one, so that timing noise never buys a costlier program. The tolerance lies above the timing noise of Section 2 (at most 0.81%).

We choose two graph constants of the Whisper encoder on the chip it runs on. The first is q, the number of chunks each head's 1,500 queries are cut into in attention (reference 8). The second is P for fc2 (reference 1). For each candidate, the engine compiles the proxy, times it on the NE for 3 s and records the median time per call. It then applies Equation 8 with τ = 1.5%. The choice and every candidate's time are stored in a chip profile (Table 5), one per chip and OS build. Later loads read the stored choice and time nothing. We choose q first with P = 1, and then P at the chosen q. From the proxy records, one format's selection costs about 65 s on M4 Max and 61 s on M5 Max, with 7 + 4 candidates compiled and timed. This cost is paid once per chip and OS build (Figure 6).

Table 5. Chip profiles: proxy time per call (ms) for each candidate, in order of preference, and the choice. These are the calibration's own timings. Tables 7 and A2 time the same programs in a separate series.

chip	format	constant	candidates (ms per call)	choice
M4 Max	FP16	q	8: 10.24, 1: 14.49, 2: 14.54, 4: 10.35, 6: 10.20, 12: 10.21, 16: 11.06	**8**
M4 Max	FP16	P	1: 10.25, 2: 8.57, 4: 8.10, 8: 8.23	**4**
M4 Max	W8A8	q	8: 8.47, 1: 12.75, 2: 12.79, 4: 8.58, 6: 8.42, 12: 8.36, 16: 9.30	**8**
M4 Max	W8A8	P	1: 8.47, 2: 8.00, 4: 8.11, 8: 8.24	**2**
M5 Max	FP16	q	8: 8.39, 1: 11.18, 2: 11.46, 4: 8.46, 6: 8.21, 12: 8.31, 16: 9.21	**6**
M5 Max	FP16	P	1: 8.21, 2: 6.94, 4: 6.58, 8: 6.90	**4**
M5 Max	W8A8	q	8: 7.09, 1: 9.96, 2: 10.20, 4: 7.18, 6: 6.94, 12: 7.02, 16: 7.92	**6**
M5 Max	W8A8	P	1: 6.93, 2: 6.63, 4: 6.74, 8: 7.03	**2**

M4 Max keeps q = 8. M5 Max picks q = 6, which takes 2.1% less time per call than q = 8 on its proxy.

Both chips pick P = 4 in FP16 and P = 2 in W8A8.

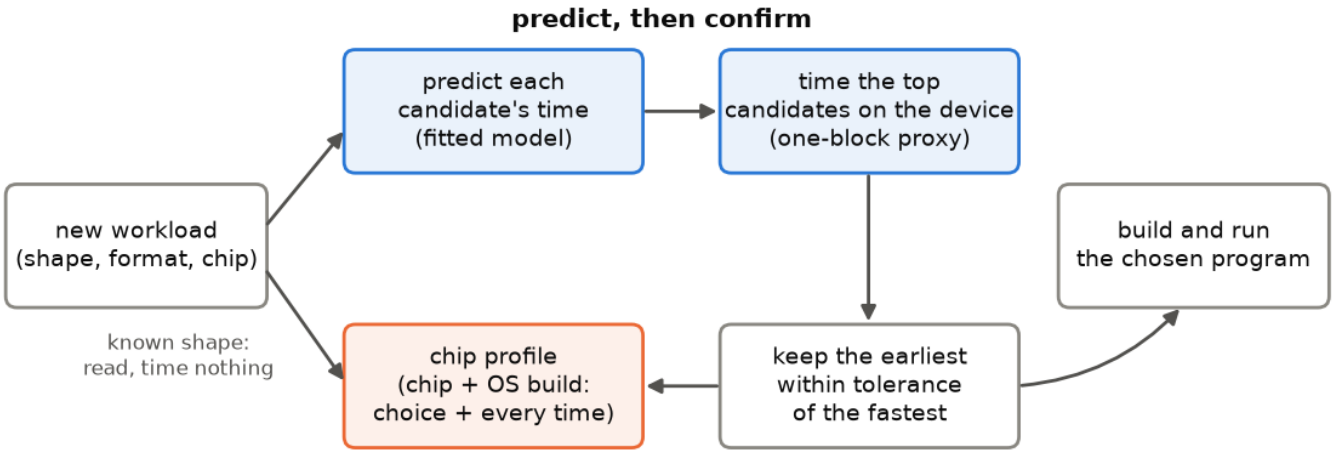


Figure 6. Living Calibration. For each chip and workload, a model predicts every candidate, and the device times the best few on a proxy. The choice and every time are kept in the chip profile. A later load of a known shape reads the profile and times nothing. Only the first visit to a shape pays for timing.

6.2 Predict, then confirm

Timing every candidate costs one compile and one timing each. Instead, a model fitted once per chip ranks the candidates, and only the best few are timed. We rank all candidates by the predicted time of Equation 6, time only the top three, and apply Equation 8 to them. We fitted the model on run 1 of the dimension study. Its predictions for unseen programs with the shapes of the Qwen team's open Qwen3.5-9B [21] were fixed before run 2 was timed on each chip. Over 51 programs per chip, the median absolute error was 8.2% on M4 Max (largest 21.4%) and 7.0% on M5 Max (largest 18.7%). The model's single best was the measured best in 4 of 8 groups. Timing only its top three kept the measured best in 7 of 8, and came within 4.3% in the other (Table 6, Figure 7). So a chip is calibrated once, the model ranks a new shape's candidates, and three timings replace a full sweep. Exact times remain a job for timing. Qwen3.5-9B has a hidden width of 4,096 and a feed-forward width of 12,288. Its layout alternates three Gated DeltaNet layers with one gated-attention layer [21]. On these shapes, the best split runs the FP16 MLP (4,096 → 12,288 → 4,096) 4.70× (M4 Max) and 4.57× (M5 Max) faster than unsplit (Figure 8).

Table 6. Predictions fixed before timing. The columns give the measured gain of the best split, the regret of the model's single best and of the best of its top three, and the median absolute error of the predicted times. Regret is the measured time over the measured best.

chip	program	best split vs unsplit	model best: regret	model top 3: regret	median error
M4 Max	FP16 4096 × 4096, 2 layers	1.88×	+13.1%	+0.0%	8.7%
M4 Max	W8A8 4096 × 4096, 2 layers	1.36×	+0.0%	+0.0%	6.3%
M4 Max	FP16 MLP 4096 → 12288 → 4096	4.70×	+12.7%	+4.3%	8.2%
M4 Max	W8A8 MLP 4096 → 12288 → 4096	1.73×	+0.0%	+0.0%	9.7%
M5 Max	FP16 4096 × 4096, 2 layers	1.77×	+0.0%	+0.0%	6.4%
M5 Max	W8A8 4096 × 4096, 2 layers	1.26×	+26.2%	+0.0%	8.8%
M5 Max	FP16 MLP 4096 → 12288 → 4096	4.57×	+0.0%	+0.0%	7.0%
M5 Max	W8A8 MLP 4096 → 12288 → 4096	1.62×	+15.1%	+0.0%	13.0%

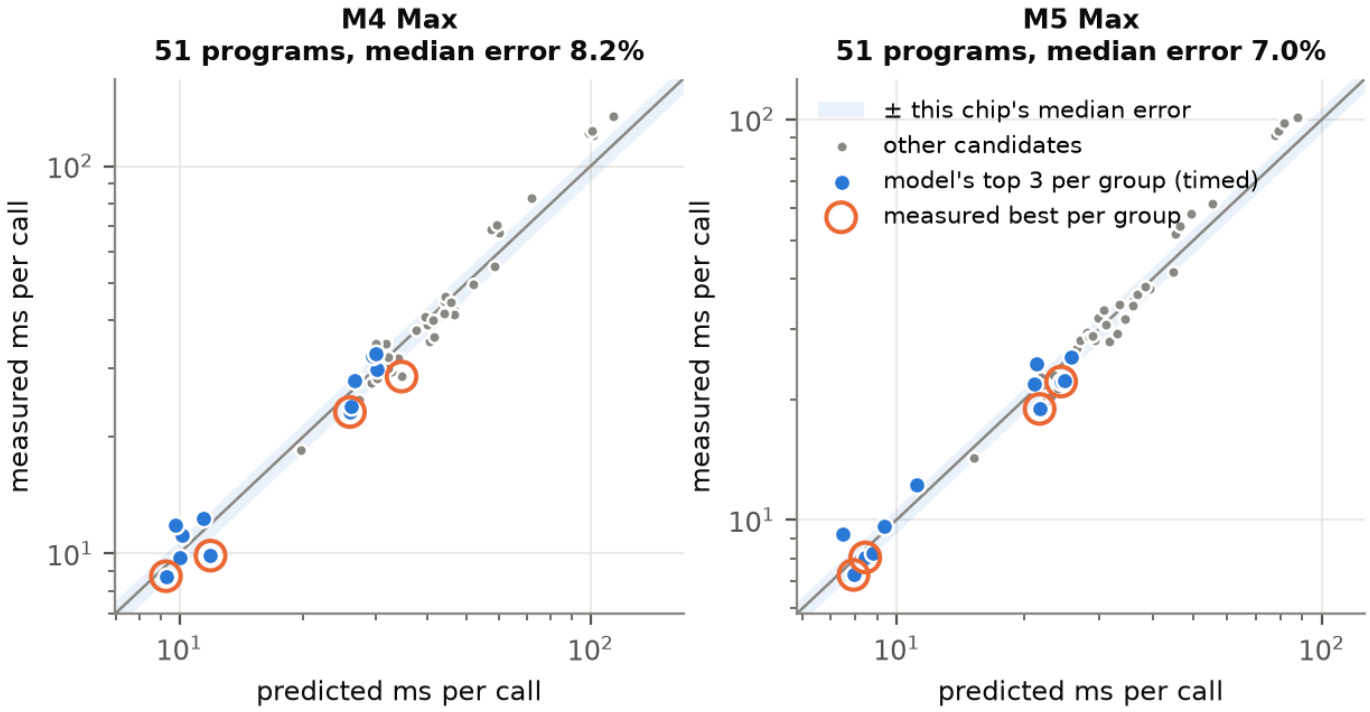


Figure 7. Pre-registered predictions. The model fitted on run 1 predicted every Qwen3.5-9B-shaped program [21] of run 2 before it was timed. Blue points mark the three candidates per shape group that the model would time, and orange rings mark the measured best. Points near the diagonal were predicted well. In 7 of 8 groups the measured best is among the blue points, so timing three candidates finds it.

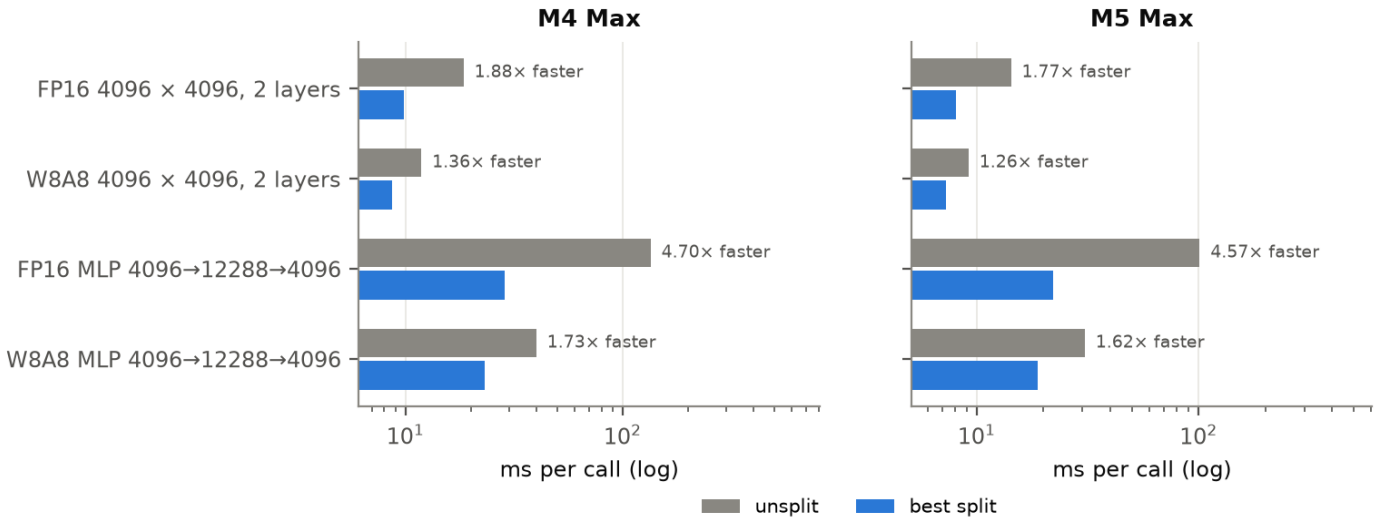


Figure 8. Qwen3.5-9B shapes [21] on the NE (dimension study, run 2), showing the unsplit program and the fastest input-dimension split. On every shape the split is faster. The gain is largest for the FP16 MLP, whose down projection has the longest sum.

7 Whisper large-v3-turbo end to end

This section tests the method end to end on a real model, the audio encoder of Whisper large-v3-turbo, on both chips. We test first on small timing proxies, and then on full transcription runs.

Workload. We use Whisper large-v3-turbo [22], with 32 encoder blocks, width 1,280, 20 heads and 1,500 encoder positions, from its FP16 checkpoint. Every utterance's encoder is forced onto the NE. The decoder runs on the GPU with identical settings in every configuration, and we use the full 30-second window. The utterances are the first 1,000 LibriSpeech test-clean [23] utterances of at most 30 s, in id order (2.05 h of audio, longest 27.2 s). The short runs use the first 192. For word error rate (WER), we take lowercase word tokens (letters, digits and apostrophes) and compute the word edit distance against the LibriSpeech transcripts, summed over the set.

7.1 Proxies. The proxy is encoder block 0 alone with the real weights, timed for 3 s per program. Table 7 gives the fc2 split and the number of blocks per program. The attention-chunk sweep is in Table A2.

Table 7. Top: fc2 split into P parts at q = 8, as median ms per call, with the change from P = 1. Bottom: encoder blocks per program at q = 8, P = 1, in ms per block.

format	P	ops	M4 Max ms	M5 Max ms
FP16	1	1293	10.24 (+0.0%)	8.39 (+0.0%)
FP16	2	1305	8.58 (-16.1%)	7.11 (-15.3%)
FP16	4	1321	8.08 (-21.0%)	6.74 (-19.6%)
FP16	8	1353	8.24 (-19.5%)	7.05 (-15.9%)
W8A8	1	1409	8.47 (+0.0%)	7.09 (+0.0%)
W8A8	2	1415	7.99 (-5.6%)	6.81 (-4.0%)
W8A8	4	1421	8.09 (-4.5%)	6.90 (-2.7%)
W8A8	8	1433	8.24 (-2.7%)	7.19 (+1.3%)

format	blocks	MB	ops	M4 Max ms per block	M5 Max ms per block
FP16	1	39.3	1,293	10.24	8.39
FP16	2	78.7	2,586	10.08	8.25
FP16	4	157.4	5,172	10.01	8.19
FP16	8	314.8	10,344	9.97	8.15
W8A8	1	19.7	1,409	8.47	7.09
W8A8	2	39.4	2,818	8.31	6.98
W8A8	4	78.8	5,636	8.25	6.93
W8A8	8	157.7	11,272	8.21	6.91

7.2 Short runs, 192 utterances × 3 passes. We run one process per configuration, through the engine's standard model path. Each configuration first loads untimed. For the chosen configuration, this load includes Living Calibration and the compile. It then runs 3 timed passes. We report the median and list every pass (Table 8). In FP16, the reference here is the coremltools-converted program, which is bit-identical on the NE to ours at the reference choices (Section 2). In W8A8, it is the converted W8A8 program, which we did not compare with ours bit for bit. The 1,000-utterance runs (7.3) use our own program as the reference in both formats.

Table 8. Short runs, wall seconds per pass.

chip	format	chosen	reference passes s	chosen passes s	time ratio	token-identical	WER % ref / chosen
M4 Max	FP16	q 8, P 4	61.928, 61.485, 61.483	48.714, 48.281, 48.269	0.7852 (1.27× faster)	189/192	2.38 / 2.42
M4 Max	W8A8	q 8, P 2	51.052, 50.601, 50.605	48.285, 47.811, 47.801	0.9448 (1.06× faster)	167/192	2.10 / 1.78
M5 Max	FP16	q 6, P 4	50.885, 50.488, 50.485	39.751, 39.426, 39.422	0.7809 (1.28× faster)	189/192	2.42 / 2.38
M5 Max	W8A8	q 6, P 2	43.444, 43.032, 43.022	40.521, 40.173, 40.167	0.9336 (1.07× faster)	168/192	1.89 / 1.85

7.3 1,000 utterances × 1 pass. We use the same path, and the reference is our program at the reference choices (Table 9, Figure 9). With the split, FP16 on the NE is faster than the W8A8 reference program on both chips: 251.8 against 263.8 s on M4 Max, and 205.5 against 224.4 s on M5 Max. It is also within 2.0% of W8A8 with its own choices.

Table 9. Wall seconds for 1,000 utterances.

chip	format	program	wall s	time vs reference	token-identical	WER %
M4 Max	FP16	reference (q 8, P 1)	320.509	–	–	3.10
M4 Max	FP16	chosen (q 8, P 4)	251.760	**0.785** (1.27× faster)	986/1,000	3.08
M4 Max	W8A8	reference (q 8, P 1)	263.809	–	–	2.76
M4 Max	W8A8	chosen (q 8, P 2)	249.252	**0.945** (1.06× faster)	911/1,000	2.58
M5 Max	FP16	reference (q 8, P 1)	263.243	–	–	3.08
M5 Max	FP16	chosen (q 6, P 4)	205.493	**0.781** (1.28× faster)	992/1,000	3.06
M5 Max	W8A8	reference (q 8, P 1)	224.355	–	–	2.70
M5 Max	W8A8	chosen (q 6, P 2)	209.749	**0.935** (1.07× faster)	921/1,000	2.71

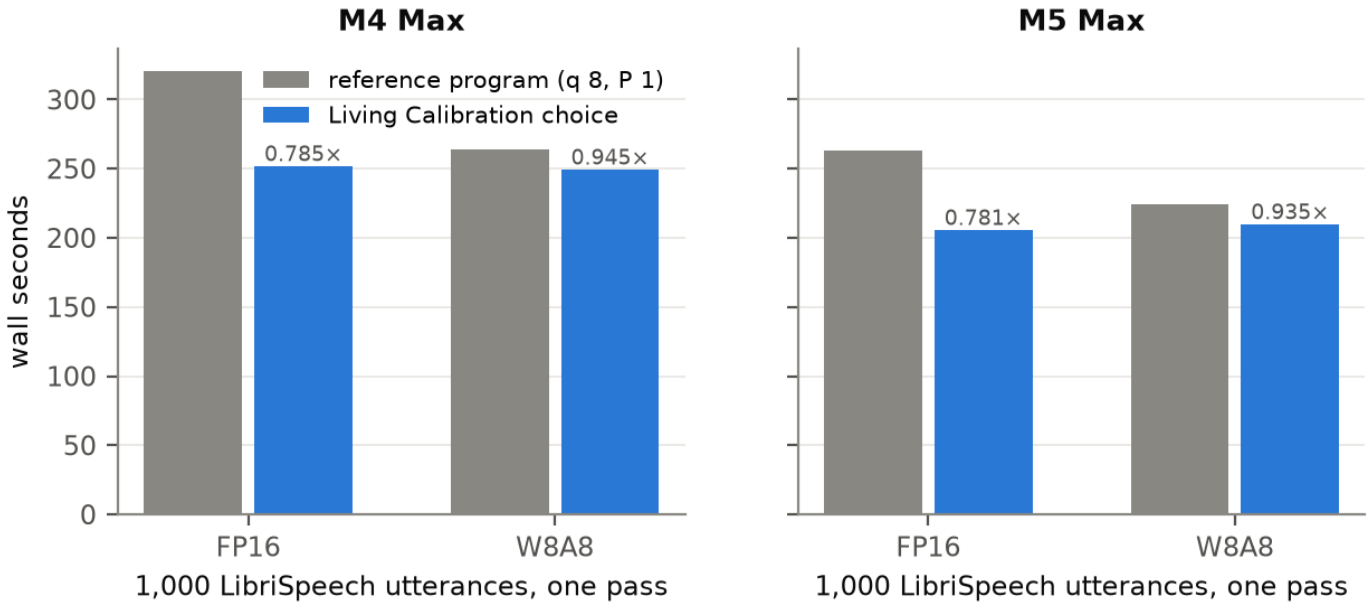


Figure 9. Whisper large-v3-turbo transcription with the encoder on the NE. The bars show the wall time for the reference program and for the program that Living Calibration chose on each chip, and the labels give the time ratio. Shorter blue bars mean faster transcription. FP16 gains more than W8A8, as the weaker K effect of the 8-bit form leads one to expect (Section 4).

7.4 The proxy predicts the encoder. On this route, the encoder sets the wall time. The reference's wall time per utterance (320.5 ms on M4 Max, 263.2 ms on M5 Max) is within 0.6% of one encoder call measured alone (320.3 and 261.8 ms). The encoder is 32 blocks of one shape. So the chosen program should save 32 × (proxy at the reference choices − proxy at the chosen ones) per utterance. Table 10 uses the chip profiles' own proxy times.

Table 10. Predicted and measured wall time per utterance (ms), 1,000 utterances.

chip	format	reference	proxy ms, reference → chosen	predicted	measured	measured vs predicted	saving predicted / measured
M4 Max	FP16	320.5	10.243 → 8.096	251.8	251.8	-0.03%	68.7 / 68.7
M4 Max	W8A8	263.8	8.474 → 8.000	248.7	249.3	+0.24%	15.2 / 14.6
M5 Max	FP16	263.2	8.392 → 6.578	205.2	205.5	+0.14%	58.0 / 57.7
M5 Max	W8A8	224.4	7.092 → 6.626	209.4	209.7	+0.15%	14.9 / 14.6

Busy time near 100% hides the remaining headroom. The NE is busy for 99.9% (M4 Max) and 99.4% (M5 Max) of each utterance's wall time. Here we compare one encoder call measured alone against the wall time per utterance. Yet the encoder, which does 2.27 TFLOP per call in its matrix products at $I = 233$ on the compute side, runs at 46% of the FP16 roof on both chips. With the split, it reaches about 59% of the roof on both chips. This rate comes from the chosen program's wall time per utterance, so it is a lower bound.

7.5 Accuracy. The split changes FP16 rounding (Section 5.3), so some transcripts change. Table 11 counts them on the 1,000 utterances (20,404 reference words). For scale, the last row of each chip shows the change that W8A8 itself makes against the FP16 reference.

Table 11. WER and token-identical transcripts. Among the changed transcripts, the last column counts how many have fewer / equal / more word errors than the reference's.

chip	comparison	WER %	token-identical	changed: fewer / equal / more errors
M4 Max	FP16 split vs FP16 reference	3.10 → 3.08	986/1,000	3 / 8 / 3
M4 Max	W8A8 split vs W8A8 reference	2.76 → 2.58	911/1,000	25 / 51 / 13
M4 Max	W8A8 reference vs FP16 reference	3.10 → 2.76	783/1,000	53 / 134 / 30
M5 Max	FP16 split vs FP16 reference	3.08 → 3.06	992/1,000	2 / 4 / 2
M5 Max	W8A8 split vs W8A8 reference	2.70 → 2.71	921/1,000	12 / 53 / 14
M5 Max	W8A8 reference vs FP16 reference	3.08 → 2.70	785/1,000	55 / 132 / 28

The largest increase from the split is +1 word error (+0.005 points, M5 Max W8A8). The split changes 14 (M4 Max) and 8 (M5 Max) FP16 transcripts. W8A8 itself, against the FP16 reference, changes 217 and 215. We do not test the significance of these differences, and we do not study here why W8A8 transcribes slightly better than FP16.

8 Boundaries

This section tests two decoder MLPs where the method should not help. At few rows per call, the convolutions' arithmetic is not the limit.

S1-mini (Qwen3-0.6B [24]) MLP. We built a proxy of S1-mini's MLP: 7 layers of 1,024 → 3,072 → 1,024 1×1 convolutions in W8A16. We split the down projection ($K = 3{,}072$) with $P = 1, 2, 3, 4, 6$. We timed it

at 1 row per call (decode) and at 64 rows (a prefill chunk), for 3 s per program. At one row per call, the program does 0.088 GFLOP in 0.485 ms on M4 Max (0.18 TOPS). This is far below the rates of Table 1, so the call is not limited by the convolutions' arithmetic. The fastest P is 1 on M4 Max (+0.0%) and 2 on M5 Max (-3.4%). At 64 rows it is 6 (-10.5%) and 1 (+0.0%) (Table 12).

Table 12. S1-mini MLP proxy: ms per call by P, and the fastest P (change from P = 1).

chip	rows	GFLOP per call	TOPS at P = 1	P = 1	P = 2	P = 3	P = 4	P = 6	fastest P
M4 Max	1	0.088	0.18	0.485	0.486	0.487	0.492	0.489	1 (+0.0%)
M4 Max	64	5.637	10.35	0.545	0.542	0.521	0.489	0.487	6 (-10.5%)
M5 Max	1	0.088	0.21	0.424	0.410	0.412	0.412	0.413	2 (-3.4%)
M5 Max	64	5.637	13.87	0.406	0.408	0.411	0.413	0.413	1 (+0.0%)

Fish S2 MLP. This proxy is FP16, with 6 layers of 2,560 → 9,728 → 2,560. We split the down projection (K = 9,728) with P = 1, 2, 4, 8, at 16 and 64 rows per call. Four times the rows take 1.030× (M4 Max) and 0.999× (M5 Max) the time, and the split changes the time by at most 3.3%. The call reads its weights at the bandwidth roof. With no gain on the proxy, we did not build the Fish rewrite, and Fish S2 has no end-to-end result here (Table 13).

Table 13. Fish MLP proxy: ms per call by P, and the fastest P (change from P = 1).

chip	rows	P = 1	P = 2	P = 4	P = 8	fastest P
M4 Max	16	5.037	5.070	5.201	5.216	1 (+0.0%)
M4 Max	64	5.188	5.201	5.203	5.015	8 (-3.3%)
M5 Max	16	3.834	3.825	3.826	3.820	8 (-0.4%)
M5 Max	64	3.831	3.822	3.823	3.843	2 (-0.2%)

On the roofline (Figure 3), both sit on the bandwidth side. S1-mini's MLP at one row has I = 2.0, left of the ridge. It runs at 74% of the bandwidth roof on M4 Max and 64–67% on M5 Max. We place W8A16 points against the bandwidth roof, because we measured no W8A16 compute ceiling. The Fish MLP proxy keeps the same time from 16 to 64 rows, and it sits on the bandwidth roof at both (97–100%). Its points set B. No split lifts a point above the bandwidth roof.

9 Related work

This section places our work among earlier work on splitting sums, on the Apple Neural Engine, and on choosing program settings by timing them.

Split-K in GPU libraries and compilers. Splitting a matrix product's reduction dimension and summing the partial results is standard on GPUs. CUTLASS offers serial and parallel split-K [10], and Stream-K [11] divides the multiply-accumulate iterations evenly across persistent thread blocks. GPU compiler backends also carry split-reduction as a pass, for example IREE [25] and ONNX Runtime's WebGPU backend for MatMul and Conv [26]. There, the split creates parallel work when the output tiles are too few to fill the machine. Tiling the reduction to bound data movement is textbook, as in the Hong–Kung I/O bound [15]. The knee law matches the I/O cost of a product that keeps its whole reduction in fast memory. Input-dimension shaping writes that tiling into a graph, for a compiler we cannot change. On GPUs, FP8 training splits the reduction for accuracy, by promoting partial sums to higher precision at fixed intervals along K

[27]. Our split adds FP16 roundings instead, and we gate it for that reason.

Splits on neural processors. TESSERA [12] is a design-space simulator for NPUs. It splits operators across tiles, reduces input-channel splits as partial sums over the on-chip network, and accepts a split when its cost model says it finishes sooner. It is the closest concept we found. We differ in four ways: we use a real device, we rewrite into separate convolutions inside one compiled program, we measure timing rather than model it, and we gate by accuracy. An Apple patent on its neural processor [28] describes a compiler that splits input data spatially into slices, tiles and work units. Its description also includes a mode that feeds a subset of input channels to each neural engine. We did not find there a graph-level rewrite into separate convolutions whose outputs are summed.

Apple's Neural Engine. Apple's transformer guidance [4] sets 1×1 convolutions for linear layers and the (B, C, 1, S) layout. It chunks large intermediate tensors, such as attention as per-head functions, for L2 residency and multicore use. Its vision-transformer guidance [5] splits the QKV projection to raise the chance of L2 residency. Both split along outputs and heads. Community runtimes cut models for size, with shards of a few layers, a size limit per model package, and language-model heads split along the vocabulary [6–8]. Another runtime splits a dense model's prefill FFN between the GPU and the NE by intermediate channels [29]. ANEForge programs the NE directly from Python, without Core ML, and compiles a tensor graph into one NE program [30]. A recent study [9] reports that a fused matmul stalls on weight streaming once a square operand passes the NE's on-chip working set near N = 2,048. It also reports that error grows with the contraction dimension. Square shapes move K and output width together. Our square layers agree, with the largest step between K = 2,048 and 2,896. Our controls separate the two. Output-channel splits shrink each operation's weights as much as input-dimension shaping does, and they leave the rate unchanged (Table 4). So a per-operation weight working set does not by itself explain the fall. K does. Whisper's encoder already runs on the NE through Core ML in public systems [31, 32]. Community notes [33] collect what is known of the NE from outside.

Choosing by timing on the device. cuDNN's algorithm search, used by frameworks' benchmark and autotune modes, times candidate algorithms on the device and caches the choice per shape [18]. ATLAS [17] and FFTW's planner [34] time variants on the target machine. TVM, AutoTVM and Ansor [19, 20, 35] keep tuning logs per target and workload and reapply them at compile. Living Calibration is narrower. It chooses graph constants of a compiled accelerator program. It times them on a one-block proxy and tries them in preference order with a tolerance. It keeps the choice in a chip profile that later processes reuse, and it ranks the candidates first by a model fitted to the chip.

What is new here. Split-K and on-device autotuning are not new. Among the sources above, we found none of the following: (1) a measurement on NE hardware that K alone sets the FP16 rate of a constant-weight 1×1 convolution, with output width, per-operation weight bytes and program bytes ruled out by controls; (2) input-dimension shaping as a rewrite inside one compiled NE program, every part and the sum on the NE; (3) Living Calibration: the split, and the attention chunking, chosen per chip by timing a proxy block on the device and kept in a chip profile; (4) a task-accuracy gate in place of bit identity for such a rewrite.

10 Limitations

This section lists what our measurements do not show.

- **Mechanism inferred, not observed.** The 1/K law and its 1/(activation bytes × weight bytes) scaling match a one-level memory model (Section 4.2). Output-channel splits argue against per-operation weight bytes alone. But the NE's internals are not visible, and the model does not separate capacity from bandwidth. It also does not explain the few-row regime (1 to 64 rows) or the residual structure around the smooth curve.
- **One OS, two chips.** We measured on macOS 26.7.1, build 25G241 with one M4 Max and one M5 Max. The compiler may change with any OS release. For that reason, the chip profile is keyed by chip and OS build, and Living Calibration reruns when either changes.
- **Coverage.** The fine K sweep, rows and width come from the dimension study's square chains and single layers. We measured W8A8 past K = 8,192 only inside one MLP shape, and we tested no kernel larger than 1×1.
- **One large-K layer type in Whisper.** Only fc2 has K above 1,280, so fc2's share of the encoder bounds the end-to-end gain. Fish S2's MLP (2,560 ↔ 9,728) has a longer K. But its decoding reads weights at the bandwidth roof, and the proxy shows no gain. Long-K models with enough rows per call to be compute-bound remain to be measured end to end.
- **The boundary.** At one row per call (decoding), the split gains at most 3.4% on S1-mini's MLP. It needs enough rows per call for the convolutions' arithmetic to be the limit.
- **W8A8 gains little.** The int8 form shows a much weaker K effect (Table A1), and its split gains only 5.6% (M4 Max) and 4.0% (M5 Max) on the proxy.
- **Prediction not integrated.** The model that ranks candidates before timing is a study result. The engine's Living Calibration still times every candidate.
- **Roofline.** We count bytes as if every weight, input and output crossed memory once per call. The ceilings are the best rates we measured, so the roofs are lower bounds. The knee roof is a fit.
- **Runs.** The 1,000-utterance check is one pass per configuration, and the short runs are three. The NE's arithmetic differs between the two chips, so each chip's reference is its own unsplit program.

11 Conclusion

Our hypothesis was that measuring the true shape of a neural simulation and shaping its dimensions to the device would make the simulation run faster. On Apple's Neural Engine it did. The rate of a convolution was set by one axis, the reduction length K, and not by the program size that public guidance and our own first view pointed to. Shaping that one axis raised a long-K layer pair 3.40× and 3.12×. On Whisper's encoder it cut transcription time by 21.5% and 21.9%, while word error rate went from 3.10% to 3.08% and from 3.08% to 3.06%. Living Calibration let each chip choose its own shape, and a model fitted on one run predicted unseen programs on the next with 8.2% and 7.0% median error.

The wider lesson is that a device judged on one axis hides the others. Busy time near 100% looked like a full chip, yet it hid the headroom that shaping recovered. Understanding the shape of a neural simulation, and shaping the simulation to the device, turned an undocumented processor into one we could predict and tune. We expect the same approach to apply to other devices and other axes, and leave that to future work.

The method has limits. Where an operation waits on memory rather than on arithmetic, as in decoding, shaping does not help, and the roofline shows why. And the idea of shaping a network's dimensions to its hardware is not new - it goes back at least to AlexNet [13]. We simply revisited what the founders of modern neural simulation did with our own theory and methods.

Acknowledgments

We thank the Qwen team. Their open models and architecture made this research and these experiments straightforward to run. They are released with open weights under Apache 2.0. These include the Qwen3 models [24] and Qwen3.5 with its hybrid of Gated DeltaNet and gated attention [21]. We thank Apple for Core ML Tools and for its public guidance on the Neural Engine. They made this research possible. We thank our AI assistant Anthropic Claude Opus 5.5 for help with software and drafting. The author is responsible for the content.

Data availability

The raw measurement records (per-call timings, per-utterance transcripts and machine conditions) and the program that computes every number, table and figure in this paper from them are available from the author on request.

References

[1] M. L. Hines, N. T. Carnevale. The NEURON Simulation Environment. Neural Computation 9(6):1179–1209, 1997. doi:10.1162/neco.1997.9.6.1179.

[2] M.-O. Gewaltig, M. Diesmann. NEST (NEural Simulation Tool). Scholarpedia 2(4):1430, 2007. doi:10.4249/scholarpedia.1430.

[3] Apple. Guide to Core ML Tools: Model Intermediate Language; ML programs. https://apple.github.io/coremltools/docs-guides/source/model-intermediate-language.html, https://apple.github.io/coremltools/docs-guides/source/convert-to-ml-program.html

[4] Apple. Deploying Transformers on the Apple Neural Engine. Apple Machine Learning Research, June 2022. https://machinelearning.apple.com/research/neural-engine-transformers

[5] Apple. Deploying Attention-Based Vision Transformers to Apple Neural Engine. Apple Machine Learning Research, January 2024. https://machinelearning.apple.com/research/vision-transformers

[6] A. Videla. The Apple Neural Engine Inference Book. Online book, 2026. https://alvaro-videla.com/ane-book/

[7] ANEMLL: Artificial Neural Engine Machine Learning Library. https://github.com/Anemll/Anemll

[8] S. Panaro. more-ane-transformers: Run transformers (incl. LLMs) on the Apple Neural Engine. https://github.com/smpanaro/more-ane-transformers

[9] S. H. Bryngelson. Apple Neural Engine: Architecture, Programming, and Performance. arXiv:2606.22283, 2026.

[10] NVIDIA. CUTLASS: CUDA Templates for Linear Algebra Subroutines and Solvers (split-K, serial and parallel reduction modes). https://github.com/NVIDIA/cutlass

[11] M. Osama, D. Merrill, C. Cecka, M. Garland, J. D. Owens. Stream-K: Work-centric Parallel

Decomposition for Dense Matrix-Matrix Multiplication on the GPU. PPoPP 2023 (poster). arXiv:2301.03598.

[12] A. Das, H. Kim, S. Lee, A. Raha, D. A. Mathaikutty, V. Raghunathan. TESSERA: A Workload-Driven Simulation and Design-Space Exploration Framework for Heterogeneous NPUs. arXiv:2606.05362, 2026.

[13] A. Krizhevsky, I. Sutskever, G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. Advances in Neural Information Processing Systems 25 (NIPS 2012), 1097–1105. https://papers.nips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

[14] S. Williams, A. Waterman, D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. Communications of the ACM 52(4):65–76, 2009. doi:10.1145/1498765.1498785.

[15] J.-W. Hong, H. T. Kung. I/O complexity: The red-blue pebble game. STOC 1981, 326–333. doi:10.1145/800076.802486.

[16] N. J. Higham. The Accuracy of Floating Point Summation. SIAM Journal on Scientific Computing 14(4):783–799, 1993. doi:10.1137/0914050.

[17] R. C. Whaley, J. J. Dongarra. Automatically Tuned Linear Algebra Software. SC 1998. doi:10.1109/SC.1998.10004.

[18] NVIDIA. cuDNN API: cudnnFindConvolutionForwardAlgorithm. https://docs.nvidia.com/deeplearning/cudnn/backend/latest/api/cudnn-cnn-library.html ; Apache MXNet: MXNET_CUDNN_AUTOTUNE_DEFAULT. https://mxnet.apache.org/versions/1.9.1/api/faq/env_var

[19] T. Chen et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. OSDI 2018. arXiv:1802.04799.

[20] L. Zheng et al. Ansor: Generating High-Performance Tensor Programs for Deep Learning. OSDI 2020. arXiv:2006.06762.

[21] Qwen Team. Qwen3.5: Towards Native Multimodal Agents. Blog, 2026. https://qwen.ai/blog?id=qwen3.5 ; model card: https://huggingface.co/Qwen/Qwen3.5-9B

[22] A. Radford et al. Robust Speech Recognition via Large-Scale Weak Supervision. ICML 2023, PMLR 202:28492–28518. arXiv:2212.04356. Model: https://huggingface.co/openai/whisper-large-v3-turbo

[23] V. Panayotov, G. Chen, D. Povey, S. Khudanpur. LibriSpeech: An ASR Corpus Based on Public Domain Audio Books. ICASSP 2015, 5206–5210. doi:10.1109/ICASSP.2015.7178964.

[24] A. Yang et al. Qwen3 Technical Report. arXiv:2505.09388, 2025. Models: https://huggingface.co/Qwen/Qwen3-0.6B

[25] IREE: pass iree-dispatch-creation-split-reduction-ops. https://iree.dev/reference/mlir-passes/DispatchCreation/

[26] ONNX Runtime pull request #26461: [WebGPU] Implement Split-K on Conv|MatMul. https://github.com/microsoft/onnxruntime/pull/26461

[27] DeepSeek-AI. DeepSeek-V3 Technical Report. arXiv:2412.19437, 2024.

[28] C. L. Mills (Apple Inc.). Splitting of input data for processing in neural network processor. US Patent 11,783,174 B2, filed 2018, issued 2023. https://patents.google.com/patent/US11783174B2/en

[29] incoai/splash, pull request #260: Split the dense prefill FFN with the Neural Engine. https://github.com/incoai/splash/pull/260

[30] S. H. Bryngelson. ANEForge: Python for Direct Computation on the Apple Neural Engine. Software: Practice and Experience, 2026. doi:10.1002/spe.70124. arXiv:2606.17090.

[31] G. Gerganov et al. whisper.cpp (Core ML encoder support). https://github.com/ggml-org/whisper.cpp

[32] A. Orhon et al. WhisperKit: On-device Real-time ASR with Billion-Scale Transformers. arXiv:2507.10860, 2025. Code: https://github.com/argmaxinc/argmax-oss-swift

[33] M. Hollemans. The Neural Engine — what do we know about it? https://github.com/hollance/neural-engine

[34] M. Frigo, S. G. Johnson. The Design and Implementation of FFTW3. Proceedings of the IEEE 93(2), 2005. doi:10.1109/JPROC.2004.840301.

[35] T. Chen et al. Learning to Optimize Tensor Programs. NeurIPS 2018. arXiv:1805.08166.

Appendix A: Additional tables

Table A1. W8A8 probe rates in TOPS, with operations counted as for FP16.

layers (in → out → in) × pairs	K	MB per layer (int8)	MB per program	rows	M4 Max	M5 Max
4,096 → 4,096 → 4,096 × 1	4,096	16.8	33.6	2,048	11.63	14.99
5,792 → 5,792 → 5,792 × 1	5,792	33.5	67.1	2,048	12.37	15.97
2,048 → 2,048 → 2,048 × 4	2,048	4.2	33.6	2,048	15.13	18.27
2,560 → 9,728 → 2,560 × 1	2,560 / 9,728	24.9	49.8	2,048	12.44	15.76
2,560 → 9,728 → 2,560 × 2	2,560 / 9,728	24.9	99.6	2,048	12.68	16.03

Table A2. Attention query chunks q at P = 1, as median ms per call, with the compile time on M4 Max.

format	q	ops	M4 Max ms	M5 Max ms	M4 Max load s
FP16	1	213	14.48	11.18	0.5
FP16	2	453	14.54	11.47	0.9
FP16	4	733	10.35	8.47	1.7
FP16	6	1013	10.19	8.21	2.4
FP16	8	1293	10.24	8.39	3.2
FP16	12	1853	10.13	8.31	4.6
FP16	16	2413	11.05	9.21	5.5
W8A8	1	329	12.75	9.96	0.6
W8A8	2	569	12.80	10.21	1.0
W8A8	4	849	8.56	7.18	1.9
W8A8	6	1129	8.41	6.94	2.6
W8A8	8	1409	8.47	7.09	3.3
W8A8	12	1969	8.34	7.02	4.8
W8A8	16	2529	9.28	7.93	5.7

Table A3. The NE points of Figure 3, with intensity I in operations per byte, rate in TFLOPS or TOPS, and the share of the roof at that I.

operation	precision	I	M4 Max rate (% of roof)	M5 Max rate (% of roof)	side of the ridge
1,024 → 1,024 → 1,024 × 4, 2,048 rows	FP16	409.6	15.32 (100%)	17.33 (92%)	compute
2,048 → 2,048 → 2,048 × 2, 2,048 rows	FP16	682.7	13.69 (89%)	17.63 (94%)	compute
2,048 → 2,048 → 2,048 × 3, 2,048 rows	FP16	682.7	13.96 (91%)	18.17 (97%)	compute
2,048 → 2,048 → 2,048 × 4, 2,048 rows	FP16	682.7	14.29 (93%)	18.48 (98%)	compute
2,048 → 2,048 → 2,048 × 6, 2,048 rows	FP16	682.7	14.61 (95%)	18.78 (100%)	compute
2,560 → 9,728 → 2,560 × 1, 2,048 rows	FP16	1,018.6	3.34 (22%)	4.35 (23%)	compute
2,560 → 9,728 → 2,560 × 2, 256 rows	FP16	227.3	3.37 (22%)	4.30 (23%)	compute
1,024 → 1,024 → 1,024 × 4, 256 rows	FP16	170.7	11.28 (74%)	12.55 (67%)	compute
1,280 → 5,120 → 1,280 × 4, 1,500 rows	FP16	608.6	7.91 (52%)	10.27 (55%)	compute
2,896 → 2,896 → 2,896 × 2, 2,048 rows	FP16	848.3	8.87 (58%)	11.49 (61%)	compute
4,096 → 4,096 → 4,096 × 1, 2,048 rows	FP16	1,024.0	7.46 (49%)	9.64 (51%)	compute
4,096 → 4,096 → 4,096 × 1, 2,048 rows	W8A8	1,365.3	11.63 (77%)	14.99 (82%)	compute
5,792 → 5,792 → 5,792 × 1, 2,048 rows	W8A8	1,696.5	12.37 (82%)	15.97 (87%)	compute
2,048 → 2,048 → 2,048 × 4, 2,048 rows	W8A8	819.2	15.13 (100%)	18.27 (100%)	compute
2,560 → 9,728 → 2,560 × 1, 2,048 rows	W8A8	1,355.8	12.44 (82%)	15.76 (86%)	compute

2,560 → 9,728 → 2,560 × 2, 2,048 rows	W8A8	1,355.8	12.68 (84%)	16.03 (88%)	compute
layer pair, whole	FP16	1,018.6	3.34 (22%)	4.35 (23%)	compute
layer pair, output ÷ 2	FP16	1,018.6	3.33 (22%)	4.34 (23%)	compute
layer pair, output ÷ 4	FP16	1,018.6	3.33 (22%)	4.34 (23%)	compute
layer pair, output ÷ 8	FP16	1,018.6	3.30 (22%)	4.29 (23%)	compute
layer pair, input ÷ 2	FP16	1,018.6	8.16 (53%)	10.60 (56%)	compute
layer pair, input ÷ 4	FP16	1,018.6	11.36 (74%)	13.55 (72%)	compute
layer pair, input ÷ 8	FP16	1,018.6	11.55 (75%)	12.40 (66%)	compute
S1-mini MLP proxy, 1 row, P = 1	W8A16	2.0	0.18 (74%)	0.21 (64%)	bandwidth
S1-mini MLP proxy, 1 row, fastest P	W8A16	2.0	0.18 at P = 1 (74%)	0.21 at P = 2 (67%)	bandwidth
S1-mini MLP proxy, 64 rows, P = 1	W8A16	109.7	10.35 (77%)	13.87 (78%)	bandwidth
S1-mini MLP proxy, 64 rows, fastest P	W8A16	109.7	11.57 at P = 6 (86%)	13.87 at P = 1 (78%)	bandwidth
Fish MLP proxy, 16 rows, P = 1	FP16	15.9	1.90 (97%)	2.49 (97%)	bandwidth
Fish MLP proxy, 16 rows, fastest P	FP16	15.9	1.90 at P = 1 (97%)	2.50 at P = 8 (98%)	bandwidth
Fish MLP proxy, 64 rows, P = 1	FP16	62.0	7.37 (97%)	9.99 (100%)	bandwidth
Fish MLP proxy, 64 rows, fastest P	FP16	62.0	7.63 at P = 8 (100%)	10.01 at P = 2 (100%)	bandwidth
encoder, reference (one call alone)	FP16	233.1	7.10 (46%)	8.69 (46%)	compute
encoder, chosen (wall per utterance)	FP16	233.1	9.03 (59%)	11.06 (59%)	compute