Unfolding: Theory and Principles

Klay Moon
NukeTorch Inc.
klay@nuketorch.ai

Abstract

A tensor's values are fixed; how they are stored and read is a choice. We call changing that choice unfolding. We study it for four-bit model weights on one Apple M4 Max in our own runtime. Expanding a four-bit model's weights to 16 bits used 2.4× the memory for a 1.14× speedup; unfolding the same codes in the same bytes gave 1.27×. A control then showed that most of that gain came from the form of the kernel's loop, not the order: the order alone is worth 4–8%. We bound the memory words a kernel must fetch for a given storage order, and show that the weight bytes a computation reads set a floor that no order lowers; three large models already decode at 81–90% of it. A test kernel whose lanes read what the file already keeps together ran 4.9× faster than the engine's original loop, at the fastest read we measured, without moving a value. For a computation that reads only part of a tensor, the order is worth an order of magnitude. The same view applies along the other folds of a matrix product: splitting a long reduction raised a Neural Engine layer pair 3.12–3.46×, and removing waits from a GPU decode schedule ran it 1.98× faster. We prove the theorems and draw four principles.

1 Introduction

This paper asks what the structure of a model's numbers costs, and how much of that cost a different structure can remove. It matters because the structure is usually inherited from a file format and a general-purpose loop, and its cost is rarely measured.

This research began from one belief. We cannot perceive values without a structure, so we mistake the familiar structure for the values themselves, and we never count what it costs. A tensor's values are fixed; its structure is a choice. To unfold a structure is to open that choice: to separate the values from the order they arrived in, and lay them along the lines the computation uses. Here, unfolding the same four-bit codes in the same bytes ran faster than expanding them to 2.4× the memory.

That last sentence is true as we measured it, and it is where we went wrong. We believed the gain came from the order. This paper is also the record of the test that showed us otherwise, and of what we found when we looked for the cost of structure where it really was.

A matrix product has a three-dimensional iteration space: rows M, output width N and reduction length K, with one multiply-add at each point. The reduction length is the number of products summed to make one output value. The order in which the machine walks that space is a fourth axis, time. Memory is one line of addresses, and a processor is handed one line of instructions. So the hardware folds the three dimensions into one-dimensional memory, and the walk into one-dimensional instruction time (Figure 1). Each fold is a choice, and each has a cost. The journey of this paper is about the fold into memory: which values share a word. Section 11 follows the fold along the reduction, and the fold of time.

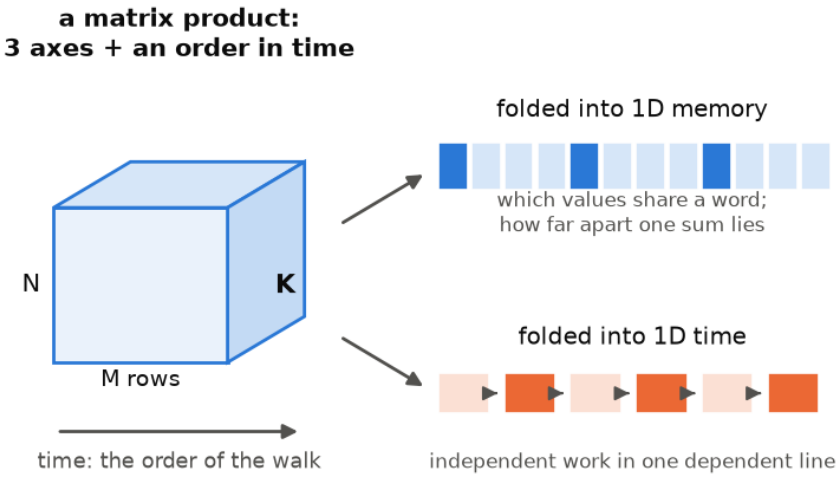


Figure 1. Folding and unfolding. A matrix product has three axes and an order in time; the hardware folds them into one line of memory and one line of time. Look at the two strips on the right: every structure this paper measures is a choice of how the product is laid onto them.

Our contributions:

1. **A model of structure** (Section 3): values, folds and draws; a count of the words a kernel must fetch; five theorems that bound it, proved in Appendix A; the floor; and the law that a kernel takes the longer of the time its bytes need and the time its work needs.
2. **A control that corrected us** (Section 6.2): the gain we credited to the order of the codes came mostly from the form of the kernel's loop.
3. **The second way to unfold** (Section 8): leave the file as it is and redraw the kernels to it. A test kernel ran at the fastest read we measured on this GPU, and no value moved.
4. **The floor, measured** (Section 9): three large models on two engines decode at 81–90% of it, so no order can make them much faster.
5. **Where order matters** (Section 10): an order of magnitude for a computation that reads part of a tensor.
6. **Space and time** (Section 11): the same act along the reduction on the Neural Engine and along time on the GPU.
7. **Four principles** (Section 12) for architecting the numbers of an intelligence.

Section 11 draws on companion work [1].

2 Related Work

The facts are not new mathematics. A vector is not its coordinates; linear algebra begins there. The tensor literature already calls a re-indexing that keeps every value an unfolding [2]; ours is the same act, without the rule that whole dimensions move together. Separating what data mean from how they are stored is an old principle of databases [3].

Layouts for kernels. Dense linear algebra packs matrices into the order its kernels consume [4]. Quantized weights are routinely repacked for the kernels that read them. Marlin reorders four-bit weights offline for the tensor cores of GPUs [5]; the four-bit kernels of FasterTransformer require their weights interleaved in memory before use [6]; AWQ's runtime packs weights to the width of the processor's SIMD units [7]; ExLlamaV2 rearranges quantized weights for its kernels [8]; and llama.cpp repacks four-bit weights into interleaved blocks when a model loads on CPUs [9]. The permutation of Section 5.2 belongs to this family. What we add is not the reordering but the control of Section 6.2, which shows that most of the gain we measured came from elsewhere.

Constants the compiler can see. Unrolling a loop whose trip count is known is an old optimization [10], and Metal's function constants let a program fix values such as a loop length or a group size when a kernel is compiled [11]. The block loop and the declared length of Sections 6.2 and 8 are instances of both. What we add is the measurement that, for this kernel, this effect is larger than the effect of the order of the values.

Counting transfers. Counting fetches of blocks is an established way to cost an algorithm, and the classical results include the cost of transposing a matrix held in blocks [12]; a matrix product walked in an order with a small fast memory is the subject of I/O complexity [12, 13]. Park, Hong and Prasanna bound the TLB misses of row and column accesses under any layout and show that block layouts come close to the bound [14]. Skewed storage in parallel memories lets the rows and the columns of a matrix both be read without bank conflicts [15]; there the cost is a conflict between banks, and a skew removes it. When a word is fetched whole, Theorem 2 shows that no layout serves two crossing kernels at once. Lower bounds on the communication of matrix multiplication follow from a geometric inequality on the projections of the computation [16]; Theorem 2 bounds a product of counts in the same spirit, for the layout of one tensor rather than for communication. Park, Hong and Prasanna's bound and these results are its closest relatives. We have not found Theorem 2 itself stated in this form: a bound on the product of the word counts of any two kernels, by how much their draws overlap. The floor is the memory side of the roofline model of performance [17], applied to the weights a token must read. Splitting the reduction, used in Section 11, is the split-K of GPU matrix multiplication [18, 19].

What we add is the statement of the idea for model weights; a control that separates the order of the values from the form of the kernel that reads them; the floor as the measure of what a structure costs; the reads of every tensor a kernel touches; an inequality for two computations; and the line between a computation that reads everything and one that reads a part.

3 Unfolding the Structure of Numbers

This section states the concept in plain words and then in symbols. The plain statements carry the argument of the paper; the theorems make them exact, and Appendix A proves each one. Readers who

want the results first can go to Section 5 and come back when a theorem is cited.

3.1 The concept in words

Values. A tensor is a set of values, and each value has a name. The name says what the value means: which output, which input. Write v(i) for the value named i. Nothing here has an order. A name is not a place.

Structure. To store a tensor we must give every name a place in memory. That assignment is the structure. Memory is read one word at a time, so the structure also decides which values share a word. We call each set of values that share a word a *fold*.

The computation has a structure too. A kernel does not take values one at a time. Each of its working units takes a handful together. We call each handful a *draw*.

Three facts follow, and everything in this paper is built on them.

1. The result does not depend on the structure. The computation is defined on the values. Wherever they sit, a kernel that finds each value by its name computes the same thing. The values have no form of their own. Whatever form they have, we gave them.

2. The cost does. Count the words a kernel must fetch: for every draw, the number of folds it touches. Suppose a word holds c values and every draw takes c values, and there are m draws. A draw touches at least one fold and at most c of them, so the count Φ obeys

$$m \;\le\; \Phi \;\le\; c\,m \qquad (1)$$

In words: a kernel fetches at least one word for each draw and at most one word for each value it takes.

The count is lowest when every fold is a draw. The structure of the store is then the structure of the computation. The count is highest when every fold holds one value from each of c different draws. The two structures then cross, and every fetch brings one value that is wanted and $c - 1$ that are not. The count is the cost we could see first. It is not the only way a structure costs, and the journey shows it is not the largest.

3. What must travel is fixed too. Every value a computation uses must reach the processor. A computation that uses all n values of a tensor, at e bits each (counting each value's share of its scales and offsets), moves at least $n \cdot e / 8$ bytes, wherever they sit. No order lowers that. We call it the *floor*. Only a computation that takes a part of the tensor can move less by keeping that part together.

Unfolding is bringing the store and the computation into one structure. The first way is to move the store to the lowest count. By the first fact, nothing else changes.

The cost belongs to the pair. Φ is a property of a fold and a kernel together. One can move the values to suit the kernel or build the kernel to suit the values. The mathematics prefers neither. Habit prefers the file. We began with the first way. The journey ends with the second.

The price of structure. The floor is what the values cost with no structure at all. Whatever a computation takes beyond the floor is the price of the structure imposed on it: by the store, by the kernel, or by the two together. That price is what we never counted.

3.2 Iteration space, store and schedule

A matrix product with K inputs, N outputs and M rows computes

$$y_{n,m} \;=\; \sum_{k=1}^{K} W_{n,k}\, x_{k,m}, \qquad (n,m,k) \in \mathcal{I} = [N] \times [M] \times [K] \qquad (2)$$

The set $\mathcal{I}$ is the iteration space: one multiply-add for each point. Two maps lay it onto the machine. A *store* ρ gives every name of a value an address; memory is read a word at a time, so ρ also cuts the names into folds F, the sets of names that share a word. A *schedule* $\sigma\colon \mathcal{I} \to \mathbb{N}$ gives every multiply-add a step of time, and the working units that run at one step take their values in draws D. In words: the store folds the values into one line of memory, the schedule folds the work into one line of time, and the result depends on neither (the first fact).

3.3 The fetch count and its bounds

The cost we can count first is the number of words a kernel fetches:

$$\Phi(F, D) \;=\; \sum_{d \in D} \Big| \{\, f \in F \colon f \cap d \neq \varnothing \,\} \Big| \qquad (3)$$

In words: for each draw, count the words it touches, and add up over the draws.

Theorem 1 (floor and ceiling of the count). Let a word hold c names and let each of m draws take c names, where the words, and also the draws, partition the names; so there are $|F| = m$ words. Then

$$m \;\le\; \Phi(F, D) \;\le\; m \cdot \min(c, m) \qquad (4)$$

The lower bound holds with equality exactly when the folds are the draws. When $m \ge c$ the upper bound is $cm = n$, the number of values, and it is reached exactly when no word meets a draw twice. In words: the best structure fetches each word once; the worst brings one wanted value per fetch, no better than storing one value in each word. (Proof in Appendix A.)

3.4 Two computations

Theorem 2 (two computations). Let two kernels read the same values in draws D_1 and D_2, each of which partitions the names, and let λ be the largest number of names that a draw of one shares with a draw of the other. For every fold into $|F|$ words of c names,

$$\Phi(F, D_1) \cdot \Phi(F, D_2) \;\ge\; \frac{c}{\lambda} |F|^2 \qquad (5)$$

Equality holds exactly when every word is a full rectangle of crossings of the two kernels' draws, with the same proportions in every word. In words: when two kernels cut a tensor across each other, no structure serves both; what one saves, the other pays, and the price is c/λ. *Proof sketch.* A word that touches r draws of one kernel and s of the other holds at most λrs names, so $rs \ge c/\lambda$. The two counts are the sums of r and of s over the words, and by the Cauchy–Schwarz inequality their product is at least $\left(\sum \sqrt{rs}\right)^2 \ge (c/\lambda)|F|^2$. (Proof in Appendix A, with its consequence for a workload that weights the two kernels.)

3.5 Lanes and every tensor

Theorem 3 (every tensor). Let a tensor be stored in name order in words of c names, and let L lanes each take runs of r neighbouring names, every L-th run, where r divides c or c divides r; the runs start at the first name, and n is a multiple of c. A lane keeps the word it last fetched and fetches again only when its next name lies in another word. Then the tensor costs

$$\min\Big(L,\ \max(1,\ c/r)\Big) \qquad (6)$$

times its floor in reads. In words: lanes that take every L-th value ($r = 1$) pay up to c times the floor on every tensor stored in name order, and lanes that take runs at least a word long pay the floor on every tensor at once. Reordering the weights for strided lanes repairs the weights and nothing else; redrawing the lanes to take neighbours repairs every tensor and moves no value. (Proof in Appendix A, which also shows that a structure unfolded for L lanes is at its floor only for the lane counts that divide L.)

3.6 The floor and the time of a kernel

Theorem 4 (the floor). A computation that uses each of the n values of a tensor, e bits each (counting each value's share of its scales and offsets), for every token, with no part of the weights kept in a cache from one token to the next, moves at least $S = ne/8$ bytes per token. On a machine that reads B bytes per second,

$$\text{tokens per second} \ \le\ \frac{B}{S}, \qquad S\ =\ \frac{n\,e}{8} \qquad (7)$$

and no structure changes S. In words: the bytes must arrive whatever their order, so only fewer bits per value or fewer values read per token can lower S. (Proof in Appendix A.)

The floor bounds the time from below. Measured kernels follow a stronger law, which we call **the time follows the bytes**: a kernel takes about the longer of the time its bytes need alone, $T_B = Q/B$ for Q bytes, and the time its work needs alone, T_W, measured on a few rows that stay in the cache:

$$T\ \approx\ \max\Big(T_B,\ T_W\Big) \quad \text{rather than} \quad T_B + T_W \qquad (8)$$

In words: a good kernel hides its work behind its reads, or its reads behind its work, and pays only for the longer of the two. The fit (Section 9, Figure 6): on one CPU thread 6 kernels in 6 runs took 0.98–1.01 times the longer of the two; on the GPU 8 of 9 kernels took 1.00–1.21 times it, and the engine's single loop took 1.39 times it, more than the sum (1.29). The GPU fit takes B as 513 GB per second, the fastest read we had measured before the last kernel of Table 6. For whole models the same law sets the decode rate: three large models read their weights at 81–90% of the fastest read we measured, 516 GB per second; Apple states 546 GB per second for this chip [20].

3.7 Partial readers

Theorem 5 (partial readers). Suppose a token needs only a of the n values, stored in $|F|$ words of c names. Its traffic, the words that must arrive, lies between

$$\lceil a/c \rceil \quad \text{and} \quad \min(a, |F|)\ \text{words} \qquad (9)$$

and some structure reaches each end. In words: for a computation that reads everything, the order changes requests but not traffic; for one that reads a part, keeping that part together can cut the traffic by up to a

factor of c. (Proof in Appendix A.)

3.8 Folding into time

The schedule σ has a cost of its own. A decode step is a chain of stages, and much of the work inside a stage does not depend on the rest of it. A *dependent line* orders every dispatch after the one before, so independent work waits as if it did. Let T_{bar} be the GPU time of a step with its barriers and T_{free} the time of the same kernels with the barriers removed, a run whose outputs race and which serves only as a measure. The share of the step spent waiting is

$$\omega \;=\; 1 - \frac{T_{\text{free}}}{T_{\text{bar}}} \qquad (10)$$

In words: the part of a step in which the GPU holds work back only because the schedule says it must wait. The common measure of time is the busy share

$$\beta \;=\; \frac{\left|\bigcup_i [s_i, e_i]\right|}{\text{span}} \qquad (11)$$

for command buffers that run from s_i to e_i. In words: the fraction of the timeline in which any command buffer is running. It keeps one bit of σ per instant, whether anything is scheduled, and drops what is scheduled, how much, and in what order; a wait inside a command buffer counts as busy. Section 11 measures both.

4 Setup

This section gives the method: the machine, the models, the tests and the checks. Every number in the paper comes from the saved records of these tests.

Sections 5 to 10 and Appendix B ran in NUKE (Neural Unfold Kernel Engine), our C++ and Metal runtime, on one Apple M4 Max with a 16-core CPU, a 40-core GPU and 128 GiB of memory; Apple states its memory bandwidth as 546 GB per second [20]. The first model is `hzang/Cosmos-Reason2-2B-4bit` at revision `7da78acf585bc20e002adfc7fe9e4ab4cc0c2548`, with its weights unchanged.

We used two kinds of test. A **screen** is six fixed requests: two text prompts that each generate 64 tokens, and four image questions. Each design warms all six and runs them twice, with control runs in checkpoint order before and after, and every gain is taken against the control of the same screen. A *panel* is the first 24 questions of CV-Bench [21], BLINK [22] or ERQA [23], run by four fresh processes in the order control, selected, selected, control.

Every output was checked. A check compares the tokens of one request with those of its reference run. A separate CPU decoder rebuilt all 1,720,451,072 weight values from the unfolded codes and found no difference. In a first run of the panels with two paths, checkpoint order and the selected path, all 336 panel requests returned the same tokens as the earlier path, and the program that compares the records refused 20 record sets we corrupted on purpose. Since the answers are identical, their accuracy is unchanged and is not a result of this paper.

The control and the redrawn kernels are builds of the same runtime that differ only in the bodies of its

storage kernels; the build with the original bodies is byte-identical to the one the first screens measured. Each build ran the six requests in a fresh process with controls first and last, and the three-way panels ran in six fresh processes in the order control, selected, file order, and back. The comparing program refused 30 record sets we corrupted on purpose. Every panel figure in this paper is from that three-way run. The earlier run of two paths, described above, agrees with it within 0.3% on every panel time.

The kernel test is a program of its own: one four-bit matrix–vector kernel in Metal, on three projections of `mlx-community/Qwen3-8B-4bit` and on a random matrix of 32,768 × 12,288 four-bit codes (201 MB; 252 MB with its scales and offsets). Each time is the median of 21 rounds (11 on the large matrix) whose order reverses every round. Kernels that add in the same order must return the same bits, and every kernel is checked against a product computed in double precision. The two CPU tests are single-threaded programs; in them the three structures of Table 8, and each kernel and its copy on a few rows, must return the same values. Theorems 1, 2, 3 and 5 of Section 3, with 8 related statements, were also checked by a program that enumerates small cases (Theorem 4 is tested on the machine, in Section 9): 31 checks over 13,372,496 cases, with no counterexample. Every prediction was written down before its test ran, including the ones that failed.

The records of Section 11 are those of the companion work [1], and this paper computes their figures with its program. The knee law is fitted on two runs of the dimension study on each chip. The decode records are S1-mini on the M5 Max GPU, each from a fresh build: a timeline before the unfolding, the timeline after it, and two sessions of 250 texts × 3 passes. Each rate is the median of its passes.

5 From Expansion to Unfolding

5.1 Expansion Versus Unfolding

This section shows where the journey began and why the first plan was wrong: it changed the size of the numbers and left their structure alone.

We started with the wrong plan. We wanted the machine to read a model's weights faster, so we made the weights larger. Every four-bit code became a 16-bit value when the model loaded. The weights went from 1.782 to 4.255 GB. The gain was 1.14×.

The plan was wrong because of what it assumed. It took the order of the weights as given and treated their size as the only thing we could change. So we changed the other thing. We kept every code at its size and changed where it sits, and we let the kernel use the one fact about the file it had been working out again at every step, the group size. In the same test, that returned 1.27× in the original 1.782 GB. Expansion enlarges the fold; unfolding opens it.

The gains in the next sections are measured on a fixed set of six requests, two of text and four with an image, against the original order in the same session. Section 4 gives the method.

5.2 Unfolding One Model

This section shows what unfolding does to one model's weights: the same codes in the same bytes, moved to where the kernel takes them. Every later comparison starts from this form.

Unfolding one model took three steps. **Folded:** the weights arrive in checkpoint order, a structure chosen for people and file formats. **Unfolded:** the same values are separated from that order and arranged for the kernel. **Never refolded:** the kernel reads the unfolded form and writes ordinary output, so nothing is ever put back.

5.2.1 Folded

The model is a public four-bit checkpoint. Each weight is a code q from 0 to 15, and each group of g neighbouring codes shares a scale s and an offset z. The weight is rebuilt as

$$w_{n,k} = R\left(q_{n,k}\, s_{n,\lfloor k/g \rfloor} + z_{n,\lfloor k/g \rfloor}\right) \qquad (12)$$

where n is the output, k the input, both counted from 0, and R the runtime's rounding. In words: a weight is its four-bit code, scaled and shifted by the numbers its group shares. The file stores eight neighbouring codes in each 32-bit word: q_0 to q_7, then q_8 to q_{15}. That is the order in which a person reads a row. Nothing in the equation asks for it.

The kernel that generates one token at a time reads differently. One SIMD group of 32 lanes computes one output, and each lane takes every 32nd code. The lanes stride across the row; the file packs along it.

5.2.2 Unfolded

Each block of 256 codes becomes 32 lane words. Lane ℓ receives $q[\ell + 32j]$ for $j = 0,\ldots,7$, so the eight codes a lane takes sit in one word. Every code still occupies four bits.

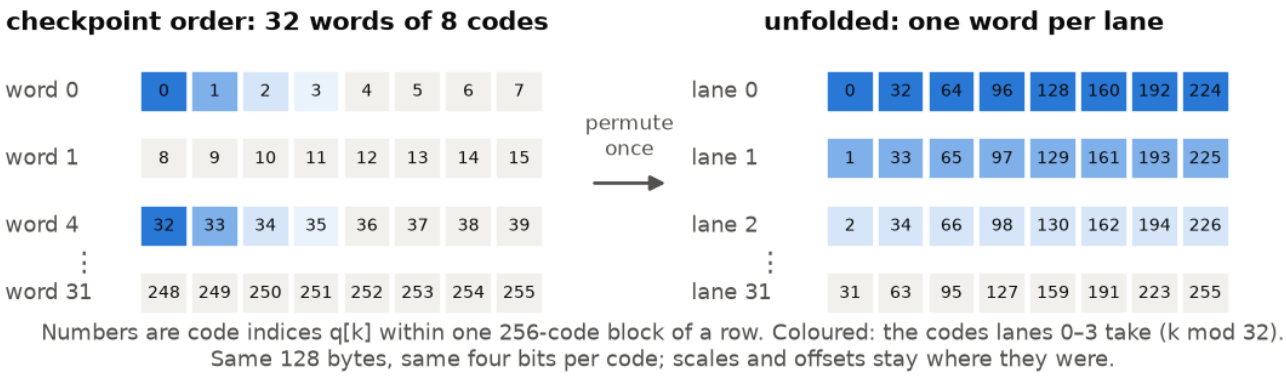


Figure 2. One block of 256 four-bit codes before and after unfolding. Look at the coloured codes: in the file, the eight codes lane 0 needs lie in eight different words; unfolded, they share one word. Their order changes; their values and their size do not.

For input k, the file finds a code at word k div 8 and nibble k mod 8. The unfolded form finds it at

$$\text{word}(k) = 32\lfloor k/256 \rfloor + (k \bmod 32), \qquad \text{nibble}(k) = \lfloor (k \bmod 256)/32 \rfloor \qquad (13)$$

In words: a code moves to the word of the lane that takes it, and its place inside that word is the step at which the lane takes it.

Every code has exactly one new place, so nothing is lost and nothing is copied. Scales and offsets stay where they were.

The same model gives a second example. The weights of its vision encoder were never quantized; they are 16-bit values. We arranged them in tiles that suit the kernel that reads them, again in the same bytes. The idea is the same. The gain is small, as the measurements show.

5.2.3 Never refolded

The permutation is applied once, when the model loads, inside the buffer that already holds the codes. No second copy of the weights is kept. The kernel reads the unfolded words and writes its output in the ordinary layout the next operator expects, so no step ever puts the codes back. We tested the alternative on an expanded layout: a step that refolds the output gave 1.07× where writing the output directly gave 1.11×.

Two other changes travel with the unfolded weights in the path we selected, and the measurements separate them. The kernel takes the file's group size, 64, as a constant. And the layers of a single-token pass are sent to the GPU as one group, where the earlier path sent each layer and waited, as Apple's guidance recommends [24]. The second is scheduling, not structure.

6 Why Order Matters

6.1 What the order saves

This section counts what the unfolded order saves and then measures what it gains. The count gives the direction; the measurement gives the size.

In the single-token projection kernel, one SIMD group of 32 lanes computes one output, and lane ℓ accumulates the terms with $k \equiv \ell \pmod{32}$. Within a 256-code block that lane needs eight codes, $q[\ell + 32j]$ for $j = 0,\ldots,7$. In checkpoint order these lie in eight different words, so the lane fetches eight words and uses one nibble of each; across the group, 256 fetches cover only 32 distinct words, each fetched by eight lanes. In execution order the eight codes are one word: the lane fetches once and shifts. Weight-word fetches fall from 256 to 32 per block, or from K to K/8 per output. Activation reads, quantization metadata reads and the arithmetic are unchanged. The count explains the direction of the gain, not its size. The multi-token kernel still fetches one word per code, which is consistent with the smaller gains on image prefill.

In the words of the concept: c is 8 and m is 32. The file's fold crosses the kernel's draws, so its count is 256, the highest there is. The unfolded fold is the kernel's draws, so its count is 32, the lowest. The order we received was not a poor order for this kernel. It was the worst one.

The count says which way the gain should go. The measurement says how far.

Table 1. Gain over checkpoint order in one screen with one control; "mixed" is total time over the six requests. Compare the second row with the last: expanding the codes to 16 bits used 2.4× the memory and gained less than unfolding them in place with the group size as a constant.

Path	Weights (GB)	Mixed	Short text	Long text	Largest image
Checkpoint order	1.782	1.00×	1.00×	1.00×	1.00×
Expanded to 16-bit values	4.255	1.14×	1.68×	1.33×	1.04×
Reordered codes, with their kernel	1.782	1.11×	1.63×	1.28×	1.01×
Group size as a constant only	1.782	1.12×	1.29×	1.21×	1.05×
Reordered codes + group size as a constant	1.782	1.27×	2.44×	1.61×	1.09×
Vision tiles only	1.782	1.03×	1.00×	1.00×	1.04×

The first row below the control is our wrong plan. The reordered codes with their kernel came within 2.6% of it in less than half the memory. Reordering together with the constant group size passed it, and passed every other expanded design we tried. The two parts do more together than either does alone. The vision tiles are a second case of the same idea, on weights that were never quantized; their gain is small.

That is what we believed when we selected this path: the order was the cause. One comparison was missing from the table.

6.2 Where the Gain Went

This section shows the control that corrected us. It matters because it moves the cause of the gain from the order of the values to the form of the kernel that reads them.

Table 1 compared kernels that differ in two ways. The codes sit in a different order. And the loop is written differently. The kernel for the file's order walks a row in one loop whose length it learns while it runs. The kernel for the unfolded order walks it in fixed blocks of 256 codes. We had never run the file's order with the loop written in blocks. So we built it.

Fresh builds of the runtime differ only in the body of one decode kernel. The engine, its modes and the weights in memory are untouched, and every request returned the same tokens as before (126 checks).

Table 2. The control. Decode rate in tokens per second on the two text requests; "mixed" is the gain in total time over the six requests. Its second and fourth rows are the designs of Table 1 named "group size as a constant only" and "reordered codes + group size as a constant", measured again in this screen. Compare the second, third and fourth rows: the block loop alone gives most of the gain; the order adds little on top.

Decode kernel	Decode, short text	Decode, long text	Mixed
Checkpoint order (control)	33.20	28.02	1.00×
File order, single loop	42.82	34.30	1.11×
File order, block loop	75.17	49.78	1.23×
Unfolded, block loop	81.10	52.45	1.27×
File order, block loop, scales shared	88.14	58.94	1.26×
Unfolded, block loop, scales shared	92.14	60.34	1.29×

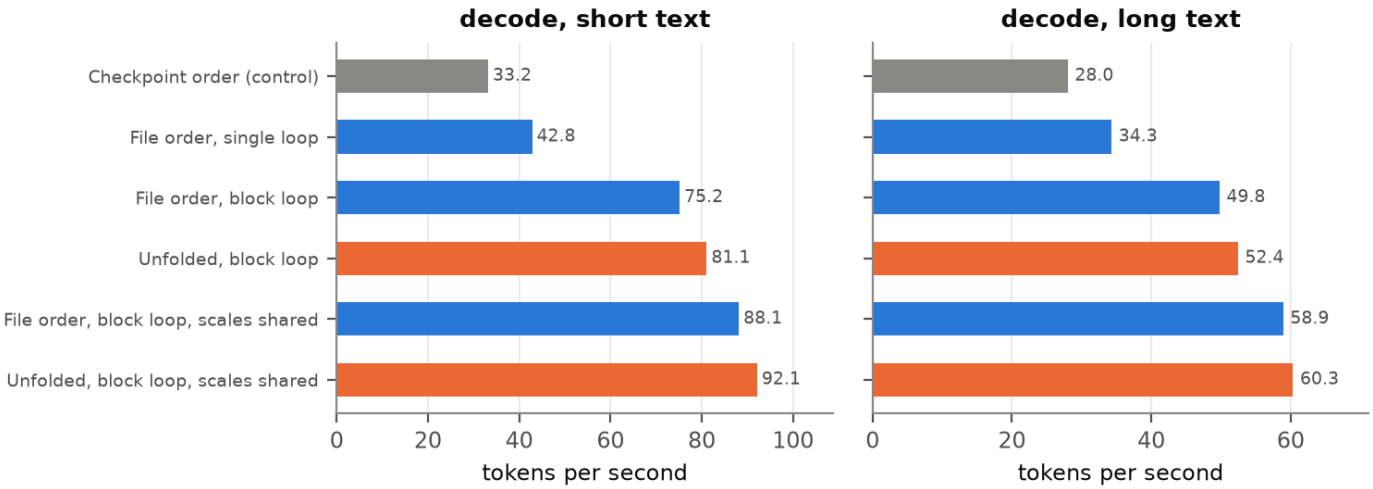


Figure 3. Table 2 as a chart: decode rate of six kernels on the two text requests. Grey is the control, blue the file's order, orange the unfolded order. Look at the jump from "single loop" to "block loop" in the file's own order: that is where the gain was, and no value moved.

Unfolding had raised the decode rate by 1.53–1.89× over the file-order kernel. The block loop alone, with no value moved, raises it by 1.45–1.76×. What the order adds on top of the block loop is 1.04–1.08× across the six requests.

A kernel test outside the engine says the same. It runs one four-bit matrix–vector kernel on three projections of a second public model, with every variant returning the same bits. With the loop written the same way, taking one word instead of eight for the same codes is worth 11–14%. Cutting the loop into blocks is worth 1.81–2.16×. The two loops execute the same number of intermediate-code instructions for each code, 38.2 and 39.2, and make the same reads. In the engine's own two kernels the counts are 41.3 and 40.3. What differs is that the length of the block is a constant in the source. Written with its whole length as a constant, the single loop runs 2.43–2.59× faster and returns the same bits. On one CPU thread the same cut runs the other way: there the loop in blocks is 1.16–1.50× slower, because the compiler turns the single loop into vector code and leaves the blocks alone.

So the gain was real, and we had put it in the wrong place. The kernel was not paying for the file's order. It was paying for a route it had to find out while it ran.

The last two rows of Table 2 show a second cost of the same kind. Every code needs a scale and an offset, and the kernel fetched both again for every code, although neighbours share them. Fetching them once for the codes that share them raises decoding by a further 1.08–1.18× in the file's order and 1.04–1.15× in the unfolded order.

None of this moved the time to the first token by much; reordering had moved it by 1.01–1.07×. That time belongs to another kernel, the one that reads the prompt.

7 One Structure, Two Computations

This section applies Theorem 2 to our model. It shows why no single order of a tensor serves every kernel that reads it, and what the price is.

Why could unfolding not help the kernel that reads the prompt as well? Because the two kernels take the same values in different ways.

A tensor is rarely read by one kernel. Theorem 2 (Equation 5) bounds two kernels that read the same values: the product of their counts is at least $(c/\lambda) \cdot |F|^2$, where a draw of one shares at most λ values with a draw of the other and $|F|$ is the number of words. When the draws share whole words, they nest, and one structure puts both at their lowest count. When they cross, no structure serves both. This is what to avoid in designing a model or an engine: two computations that cut the same tensor across each other. If they must cross, the price is c/λ, paid by one computation in full, by both in part with tiles, or in memory with a second copy.

In our model the two kernels cross completely. The kernel that generates takes every 32nd code of a row. The kernel that reads the prompt takes 32 neighbouring codes of a row at a time. A draw of one shares exactly one code with a draw of the other, so λ is 1. With $c = 8$ and $|F| = 32$ words in a block, the product of the two counts is at least 8,192. Table 3 counts four orders by enumeration.

Table 3. Fetches per 256-code block in four orders of the same codes, counted by enumeration. The prompt column is the fewest fetches possible while that kernel takes 32 neighbouring codes at a time. Look at the last column: every order meets the bound of Theorem 2 exactly, so moving between them only trades one kernel's fetches for the other's.

A word holds	Generating kernel	Prompt kernel	Product
8 lanes × 1 step (the file)	256	32	8,192
4 lanes × 2 steps	128	64	8,192
2 lanes × 4 steps	64	128	8,192
1 lane × 8 steps (unfolded)	32	256	8,192

All four meet the bound exactly. They are the frontier: moving along it trades one kernel's fetches for the other's, and no order lies below it. The order we selected puts the generating kernel at its lowest count and the prompt kernel at its highest. The loader we run today already makes 256 fetches in either order, so unfolding cost the prompt kernel nothing. But it also closed a door. In the file's order that kernel could come down to 32; in the unfolded order it cannot, as long as it takes 32 neighbouring codes at a time. Reading the prompt is where an image question spends its time: 6,238 of 6,333 ms on our largest one.

So there is no best structure for a tensor, only a best structure for a workload. Aligning fully to one computation pays only when it carries enough more of the work than the other. Weight each kernel's fetches by how often it runs. The bound of Theorem 2 favours full alignment once one weight is $c/\lambda = 8$ times the other (Appendix A). The counts are whole numbers, and for every fold of this block the unfolded order is the cheapest exactly when the generating kernel weighs at least 4 times as much as the prompt kernel; the file's order is the cheapest when the prompt kernel weighs at least 4 times as much; in between, a tile is cheaper (Appendix A). The way out of the inequality is to change the draws and not the store: make the computations take their values so that the draws nest.

The inequality counts fetches, and the control showed that a fetch is a small cost for this kernel. But its way out is the step the journey took next.

8 Unfolding the Kernel

This section takes the second way to unfold: leave the file as it is and redraw the kernels to it. It matters because it moves no value and still gained the most.

The cost belongs to the pair, and we had only ever moved one of the pair. So we left the file as it is and redrew the kernels to it.

There are three changes, and none of them touches a value. The decode kernel walks the file's order in fixed blocks. It fetches each scale and offset once for the codes that share them. And the kernel that reads the prompt loads its tile one whole file word at a time: one fetch brings eight codes that share one scale and one offset, where it used to fetch for every code. With that loader the prompt kernel is at its lowest count, 32 fetches, in the file's order, the door the unfolded order had closed.

Table 4. The engine with the kernels redrawn, grouped decode on. Decode rate in tokens per second; first-token times in milliseconds. Compare the last row with the second: the file's own order with redrawn kernels beats the unfolded path we had selected on the whole request.

Path	Decode, short text	Decode, long text	First token, long text	First token, largest image	Mixed
Checkpoint order (control, no grouping)	33.38	28.19	1,015	6,235	1.00×
Selected path: unfolded in place	121.85	73.40	848	5,745	1.34×
Selected path, scales shared	150.80	81.66	824	5,541	1.39×
File order, block loop, scales shared	140.67	78.48	864	5,827	1.32×
The same, loader takes whole words	139.84	78.43	573	5,026	1.55×

Every request returned the same tokens (108 checks). On every one of the six requests the file's order with both kernels redrawn is faster than the path we had selected, by 1.14–1.24×. The first token of the long text comes in 573 ms where it took 848. On text alone the selected path with shared scales still decodes a little faster, 150.8 tokens per second against 139.8; the file's order wins the whole request.

We then ran the three panels of Section 4 three ways.

Table 5. Three 24-question panels: time for the panel in seconds, and median time to the first token in milliseconds. Compare the last two time columns: on every panel the file's order with redrawn kernels finishes first and answers first.

Panel	Checkpoint order (s)	Selected path (s)	File order, redrawn (s)	First token, selected	First token, file order
CV-Bench	31.95	27.31	24.29	450	389
BLINK	91.80	74.98	68.09	964	833
ERQA	50.08	41.93	37.24	1,626	1,420

The file's order with redrawn kernels finishes the panels 1.10–1.13× faster than the selected path, and its first token comes 1.15–1.16× sooner. Against checkpoint order it is 1.32–1.35×, where the selected path is 1.17–1.22×. All 504 checks returned the same tokens. Nothing is reordered when the model loads.

How far can this go? We took it further in the kernel test, where a different order of the sum is allowed.

A kernel reads four things for every code: the code, its scale, its offset, and the activation it multiplies. All four are stored in the order of their names. A lane that takes every 32nd value crosses all four of them. Unfolding the weights repairs one. A lane that takes neighbours repairs all four at once, and moves nothing; this is Theorem 3. So we wrote the kernel whose lanes take one whole scale group: 64 neighbouring codes, with their one scale and one offset, in the file's order.

Table 6. One matrix of 201 MB of four-bit codes, seven kernels. "Bytes alone" is the time the kernel's 252 MB, codes with their scales and offsets, need at 516 GB per second: the rate of the last kernel itself, and the fastest read we measured on this GPU. Read the third column from top to bottom: each change of the kernel brings it closer to the last one.

Kernel	ms	Time ÷ bytes alone	Result
Engine's loop, file order	2.378	4.87×	same bits
Same loop, fixed blocks	1.087	2.23×	same bits
Unfolded order, fixed blocks	0.974	2.00×	same bits
Engine's loop, length declared	0.873	1.79×	same bits
Unfolded, scales shared by pairs	0.682	1.40×	same bits
File order, lanes take whole words	0.624	1.28×	to rounding
File order, lanes take whole scale groups	0.488	1.00×	to rounding

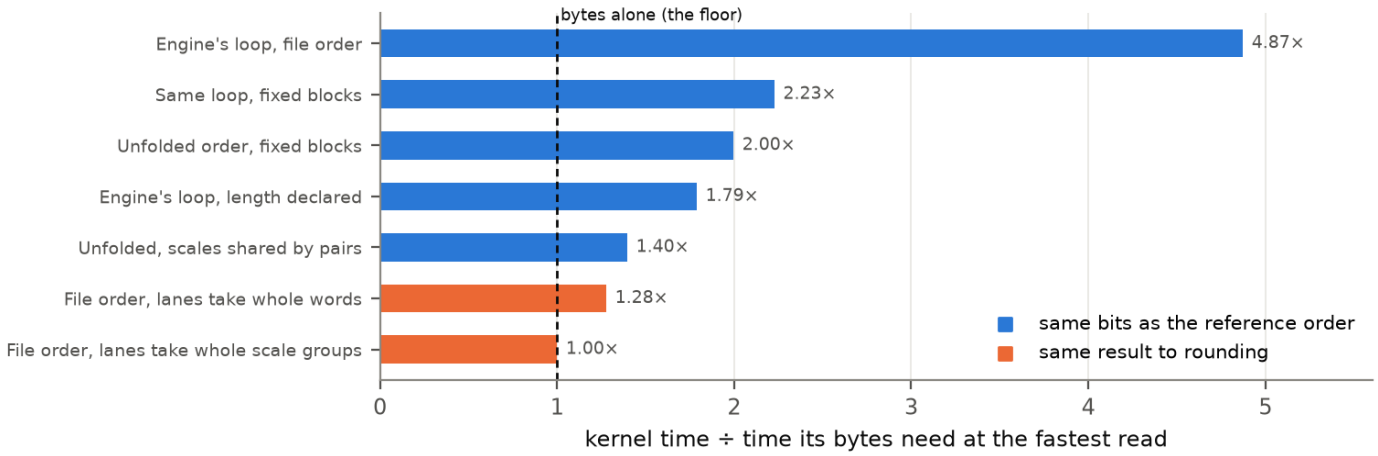


Figure 4. Table 6 as a chart: each kernel's time divided by the time its bytes need at 516 GB per second. The dashed line is that rate, set by the last kernel; no read we measured was faster. Blue kernels return the same bits as the reference order; orange ones add in another order and agree to rounding.

The last kernel computes the full product at 516 GB per second, 94% of the 546 GB per second that Apple states for this chip [20]. No other read we measured on this GPU was faster, so the floor of Table 6 is this kernel's own rate: it sits at 1.00× by definition, and the table shows how far the other kernels lie above it. It is 4.9× faster than the engine's loop on the same file, and 1.24–1.29× faster than lanes that take whole words on every matrix we tried. It takes what the file keeps together, in the file's order, and imposes nothing on it.

The file was not folded wrongly. Our kernel drew across it.

That kernel is not in the engine. Its sum runs in another order, so its result agrees with the reference to rounding and not bit for bit. When we put its first step into the engine, lanes that take whole words, 5 of 6 requests returned the same tokens and one did not: on the long text the first 49 tokens are the same and the next one differs. The engine's path that keeps every token is the one of Table 5.

9 The Floor

This section measures the floor for whole models: the time their weight bytes need to arrive. It matters because it says how much any change of structure can still gain.

The last column of Table 6 divides by a time: the time a kernel's bytes need to arrive. That is the floor of the third fact and of Theorem 4, and it holds for a whole model as well as for one kernel.

A model that reads every weight for every token moves all its weight bytes for every token, and no order of those bytes changes how many there are. In our tests this GPU reads at most 516 GB per second, the rate of the last kernel of Table 6. So a token cannot take less than its bytes divided by that rate. For each engine and model in our saved records we multiplied the decode rate by the bytes of weights one token must read.

Table 7. Weight bytes read while decoding. The last column is the fastest read we measured divided by the rate the model already reads at: the most any change of order or kernel could still gain if that read is the ceiling. Look at the share column: the large models are within a fifth of the floor, the small one is far from it. The MLX rows are from our saved records of earlier sessions on the same machine (Cosmos-Reason2-2B: MLX 0.31.2 [25], mlx-vlm 0.4.4 [26]); we did not run MLX again.

Model	Engine	GB per token	Tokens per second	GB per second	Share of the fastest read	Most that is left
Qwen3.5-9B, 16-bit	NUKE	15.859	29.41	466	90%	1.11×
Qwen3.5-27B, four-bit (NVFP4)	NUKE	14.418	29.74	429	83%	1.20×
Qwen3.5-27B, four-bit (NVFP4)	MLX	14.418	28.86	416	81%	1.24×
Qwen3-8B, four-bit	MLX	4.258	97.88	417	81%	1.24×
Cosmos-Reason2-2B, four-bit	MLX	0.968	186.22	180	35%	2.86×
Cosmos-Reason2-2B, four-bit	NUKE, file order redrawn	0.968	88.59	86	17%	6.01×
Cosmos-Reason2-2B, four-bit	NUKE, selected path	0.968	80.42	78	15%	6.62×
Cosmos-Reason2-2B, four-bit	NUKE, checkpoint order	0.968	29.62	29	6%	17.98×

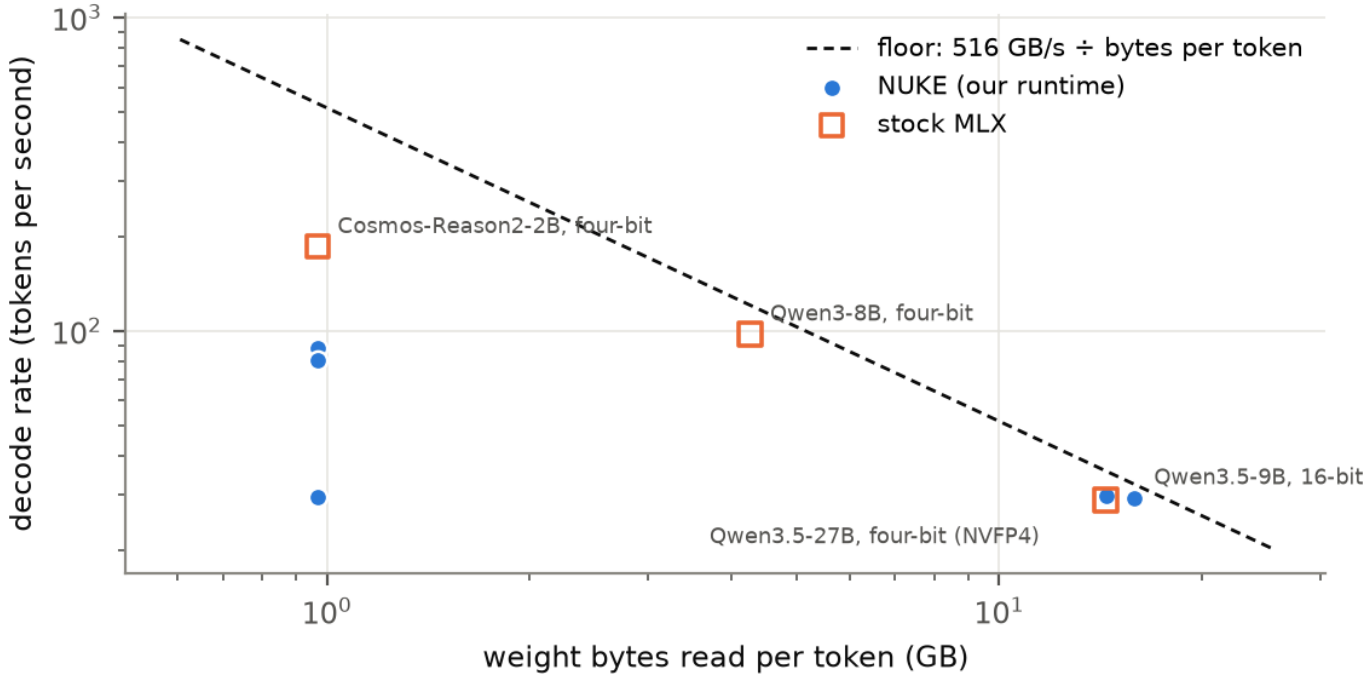


Figure 5. Table 7 as a chart, on logarithmic axes: decode rate against the weight bytes one token reads. The dashed line is the floor, 516 GB per second divided by the bytes; no point we measured lies above it. Blue points are our runtime, orange points stock MLX [25]. The large models sit just under the line; the small model lies far below it, where its time is in the structure and not in its bytes.

The three large models, on two engines and in three weight formats, decode at 416–466 GB per second: 81–90% of the fastest read. They are close to the floor. Measured against 516 GB per second, the fastest read we measured, at most 1.11–1.24× is left; against the 546 GB per second that Apple states, at most 1.31×. That is a measured margin, not a proven ceiling, and it is all that any order of their values could still gain.

That explains our second model. It is a public four-bit checkpoint in a different format, `arthurcollet/Qwen3.5-27B-mlx-nvfp4` at revision `094bb2302099616695699df62e1329eb54e3278d`, and it runs on kernels we had tuned for it before this study began. We had tried a reordering of it, with the same codes in the same bytes. In a first screen, decoding moved by +0.7 to +0.9%. In a confirmation that balanced the order of the runs, with 36 requests and identical outputs, it moved by −1.2 to −0.5%. Nothing was gained, and little could be: it already reads 429 GB per second.

The small model is far from the floor. Its time was never its bytes. That is why rewriting a kernel could move it, and why a faster engine on the same file can exist.

The last column of the table is a rule for choosing. With the bytes as they are, and the fastest read we measured as the ceiling, no change of order or of kernel can gain more than that number: 1.11–1.24× for the large models, 2.9–18.0× for the small one. Near 1, only fewer bytes help: fewer bits for each value, or fewer values read for each token. Far from 1, the time is in the structure we impose.

This also settles the wrong plan. Expanding every code to 16 bits removes the scale, the offset and the rounding, and moves four times the bytes of the codes. In the kernel test the expanded kernel is 2.7–3.3× slower than the best four-bit kernel that returns the same bits, on the two large projections. It has almost

nothing left to do, and it waits for its bytes.

A kernel is not faster, within 2%, than the longer of two times: the time its bytes need, and the time its own work needs (Equation 8). We measured both, the work alone by running each kernel on a few rows that stay in the cache. On one CPU thread 6 kernels, in 6 runs, took the longer of the two, within 2%, and not their sum. On the GPU no kernel was faster than the longer of its two times, but the two did not overlap as well: 8 of 9 kernels took 1.00–1.21× the longer, and the engine's single loop took 1.39×, more than their sum. The GPU times take the bytes at 513 GB per second, the fastest read we had measured before the last kernel of Table 6.

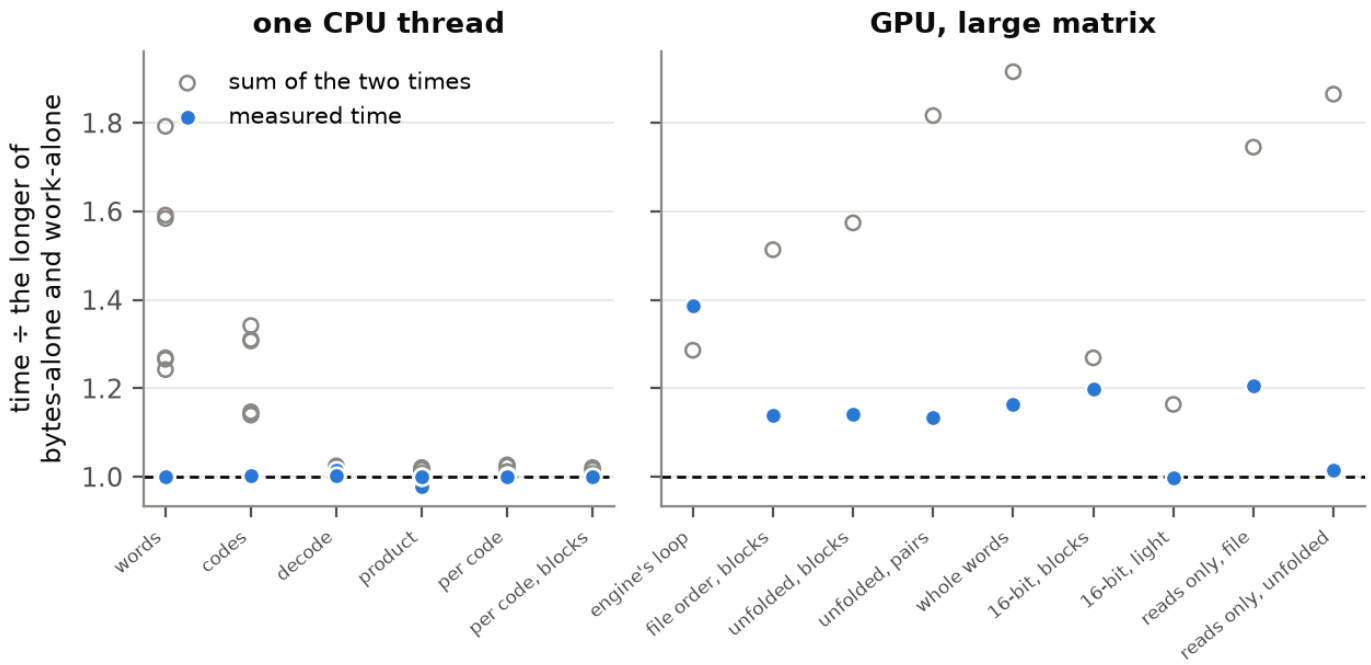


Figure 6. The time follows the bytes. Each kernel's measured time divided by the longer of its two times, the time its bytes need alone and the time its work needs alone (filled), and the sum of the two on the same scale (open). The dashed line is the law of Equation 8. On the CPU every kernel sits on the line; on the GPU most sit a little above it and one, the engine's single loop, above the sum.

10 Partial Readers

This section shows the one kind of computation for which the order of the values is worth an order of magnitude: one that reads only a part of a tensor. It is Theorem 5, measured, with the 128-byte cache line as the unit of traffic: a line holds 256 codes, so the theorem allows a factor of up to 256.

For a computation that reads everything, the order of the values was nearly free: 4–8% in the engine. The third fact says when it is not. A computation that takes only a part of a tensor moves less if that part is kept together, and more if it is spread.

We tested it on the CPU, on one thread: a table of 34 MB of four-bit codes and 8,192 lookups, each of which needs one row. The same values are stored three ways. **Kept**: a word holds eight neighbouring codes of one row. *Interleaved*: a word holds one code of each of eight rows, and a row's words follow one another, so a lookup moves eight times the bytes. *Scattered*: the same words as interleaved, with a row's words far

apart, so every word is its own cache line.

Table 8. Milliseconds for all the lookups in the first of 3 runs, and the ratio to the kept table over the runs; lines are the 128-byte cache lines of this CPU. First shape: 65,536 rows of 1,024 codes. Second shape: 16,384 rows of 4,096 codes. Compare the interleaved and scattered rows: the same words, read in order or scattered; only the scattered one pays.

Structure	Cache lines per lookup	First shape, ms	First shape, × kept	Second shape, ms	Second shape, × kept
Kept	4 / 16	1.338	1.00×	4.106	1.00×
Interleaved	32 / 128	1.664	1.21–1.26×	3.840	0.94–0.95×
Scattered	1,024 / 4,096	15.700	11.71–11.99×	78.158	18.38–19.04×

Scattered, a lookup takes 11.7–12.0× and 18.4–19.0× the time it takes from the table that keeps its row together. Eight times the bytes, read in order, cost 0.94–1.26×: at most 26% more. A computation that takes a part does not pay much for bytes that come in order. It pays for what is scattered.

A model that reads all its weights for every token is not such a computation, and that is why unfolding its weights gained so little. A table of embeddings is one. So is a cache of keys, and so is any model that will read only the part of itself that a token needs. For them, keeping together what is taken together is a law.

11 Other Folds: Space and Time

This section follows the same act along the other folds of a matrix product: along the reduction, in space, and along the walk, in time. It matters because it tests the premise of the introduction beyond the fold into memory that the journey measured.

The journey stayed with one fold: which values share a word of memory. A matrix product has three dimensions and is walked along a fourth, and the hardware folds them into one line of memory and one line of instruction time. The records of this section measure the same act along the reduction and along time. They come from later work, on other processors than the journey's: the Neural Engine of two chips, M4 Max and M5 Max, and the GPU of the M5 Max.

11.1 In space

The Neural Engine runs a linear layer as a 1×1 convolution with constant weights. The companion work [1] measured its rate and found that the reduction length K, the layer's input channels, sets it; neither the program's weight bytes nor the output width does. Its dimension study fits one law on 56 programs per chip. Past a knee $K^{*} = A/r_{\text{peak}}$, the rate falls as one over K:

$$R(K) \;=\; \min\!\left(r_{\text{peak}},\, \frac{A}{K}\right), \qquad A \;\approx\; \frac{C}{a\,w} \qquad (14)$$

with R in trillions of operations per second, a and w the bytes of an activation and of a weight, and C a constant of the chip. In words: up to the knee every layer runs at the chip's peak; past it, doubling the length of each sum halves the rate, and smaller numbers move the knee further out. The knee lies further out on M5 Max than on M4 Max. Halving the weight bytes multiplies A by 1.84–1.88, halving both by 3.22–3.29, where an exact 1/(a w) would give 2 and 4.

It is the floor's law one level down. Section 3 puts the floor as **the time follows the bytes**: a token takes at least the time its bytes need to arrive. Past the knee the time follows bytes as well, but bytes that the fold creates. Read through a model with one small fast memory [13], a reduction too long for the fast memory reloads its activations and weights for every block of outputs, so the traffic grows with K and with the bytes of each value, which is the form of Equation 14. The fold has put the terms of one sum too far apart to keep at hand. This is a reading of the measurements through a model, not a test of the hardware.

Splitting a long reduction into shorter ones, and adding their results inside the program, unfolds the reduction in space. No weight changes. On a layer pair of a language model it raised the rate 3.12–3.46× on the two chips. Like the test kernel of Table 6, it runs the sum in another order, so its outputs are not bit-identical; the companion work checks it by task accuracy instead.

11.2 In time

A decode step of a language model is a chain of kernels, each waiting for the one before. These records are a small model, S1-mini by Superwhisper (`superwhisper/s1-mini` at revision `88f6b15896c73bbb13a3b596e0afe8ea0d5150b4`, a Qwen3-0.6B model), decoding batches of texts on the M5 Max GPU. With the barriers between its kernels removed, a run whose outputs race and which serves only as a measure, the same kernels take 2,010 μs of GPU time per step instead of 2,709 with 16-bit weights, and 1,080 instead of 1,611 with 8-bit integer weights. By Equation 10, the waits are 26% and 33% of a step. Busy time counts a command buffer as busy from its start to its end (Equation 11), so these waits sit inside busy time, where it cannot see them.

We then unfolded this decode in time: work that did not depend on other work no longer waited for it. The GPU's busy share of the timeline rose from 88.6% to 99.7%. In one session, 250 texts × 3 passes, throughput with 16-bit weights rose from 30.39 to 60.04 texts per second, 1.98×, and all 250 texts returned the same tokens. A later build reached 64.48 texts per second in a later session: 2.12× the first session's old path, a ratio across sessions.

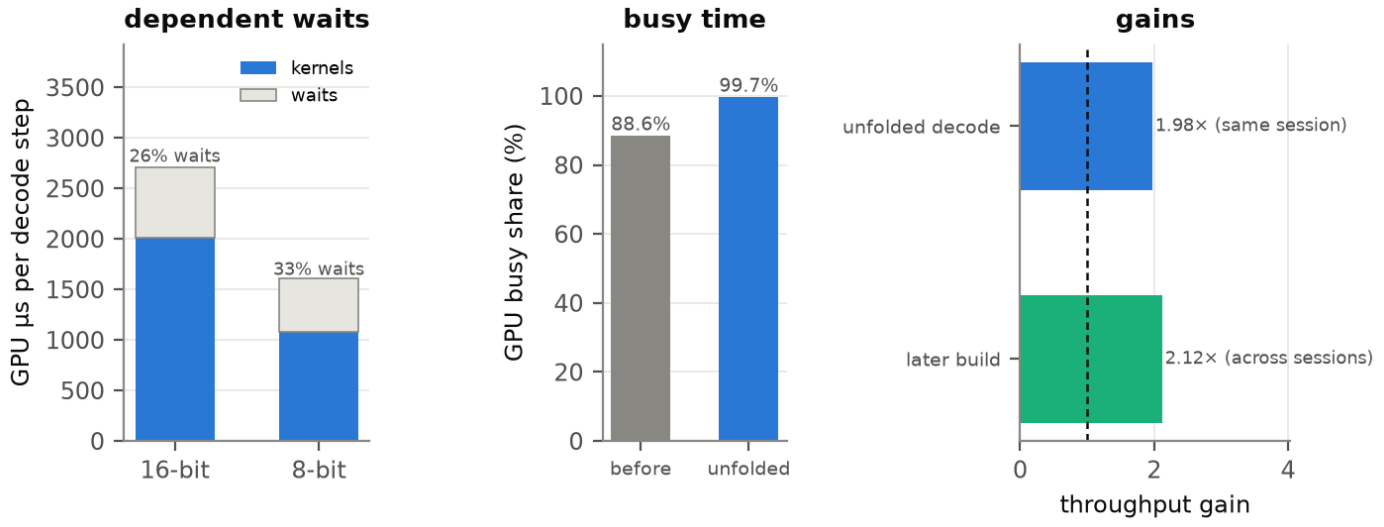


Figure 7. The time axis of S1-mini decoding on the M5 Max GPU. Left: GPU time per decode step with its kernels' barriers (full bar) and without them (blue); the grey part is waiting, 26% and 33% of a step. Middle: busy share of the timeline before and after unfolding. Right: throughput gains; green marks the one ratio taken across sessions. Busy time rose by 11 points while throughput nearly doubled.

Busy time registered only the gaps that unfolding closed; the waits inside each step were invisible to it. A measure in one dimension of an activity in four drops the axes on which the time is lost.

12 Four Principles

This section turns the journey into rules for design. Each rule names the step of the journey it comes from, so its evidence can be checked.

A model is numbers. How they are arranged is an architecture of its own, and it is usually inherited rather than designed. These are four principles for architecting the numbers of an intelligence. Each comes from one step of the journey.

1. Choose the structure. Do not inherit it. The values are fixed. Their order came from a file, and the way our kernel walked them came from a general-purpose loop. Both were structures, both were inherited, and we had been paying for the second without seeing it. Before changing how large the numbers are, look at what is imposed on them. Making the numbers larger cost 2.4× the memory and returned less than rewriting the loop alone: 1.14× against 1.23× (Tables 1 and 2, separate screens). The choice has one limit: structure is a choice as long as the sums keep their order.

2. Know the floor. A value costs at least the bytes that carry it. Measure how far a computation stands above the time its bytes need. Near the floor, only fewer bytes help. Far from it, the time is in the structure, and it can be removed without touching a value.

3. Declare the route. Whatever is fixed about the numbers should be written where the machine can see it before it runs: the length of a loop, the group that shares a scale, the neighbours that sit in one word. A kernel that has to find these out for every value pays for them at every value. Take the numbers as they are kept, and let the computations that share them take them in ways that nest.

4. Keep together what is taken together. For a computation that reads everything, this is nearly free either way. For a computation that takes a part, it is a law, worth an order of magnitude in our test. Fold the numbers along the draws of the computations that take a part of them.

None of the four names a best structure. The loop that ran faster in blocks on the GPU ran 1.16–1.50× slower in blocks on the CPU (Section 6.2). A structure unfolded for 32 lanes is at its floor only for the lane counts that divide 32. The file, the kernel, its lanes, the processor and the schedule of the work have to be chosen together. Everything must work together.

The second and the fourth rest on the theorems of Section 3. The first and the third rest on measurements of one model on one machine, and the third is known to depend on the processor. How far they reach is the next thing to test.

13 Limitations

Every gain in this paper is measured against our own earlier path on the same model; the study compares structures within one runtime, not engines, and engine-level results are reported separately.

The journey is one machine. The engine’s gains are measured on one model, against our own earlier path,

with two runs of each design in the screens and two sessions of each path in the panels. That is enough to see the direction and too little to put a width on it.

We were wrong once already. An earlier version of this work credited the gain to the order of the codes, and its own tables did not show the mistake. A control did. There may be others we have not thought to run.

The kernel that reaches the floor is a test kernel. Its sum runs in another order than the engine's. In the engine, its first step changed a token in one request of six, so the engine's exact path stops at the block loop, the shared scales and the loader. The loop with a declared length returns the same bits and is not in the engine yet.

The floor is measured, not derived. It is the rate of our own fastest kernel, 94% of the bandwidth Apple states, and it rose from 480 to 516 GB per second as we wrote faster kernels. On the CPU a kernel took the longer of its two times; on the GPU most took up to 1.21× of it, and one took more than their sum, which we cannot yet explain.

The test of a computation that reads a part ran on one CPU thread and a table of random codes. That scattering costs time is not new. What we report is the size of the difference between the two kinds of computation on one machine.

The form that wins belongs to the processor. Cut into fixed blocks, the same loop was faster on the GPU and slower on the CPU. The third principle is a statement about a pair, and we have measured one pair.

The inequality is proved for the draws as they are, and it counts fetches, which were a small cost for the kernel we measured. The widest case we expect, a matrix multiplied from both sides as in training, is untested.

The records of Section 11 are later work, on other processors and another model, and this paper only reads them. The knee is fitted on the Neural Engines of two chips, whose compiler is undocumented, and its reading as reloaded bytes goes through a model of memory; it is not a test of the hardware. Splitting the reduction changes the order of the sum, so its outputs are checked by task accuracy and not bit for bit. The time records are one small model on one GPU; the waits were measured in a run whose outputs race; and the 2.12× compares builds across sessions, where the in-session figure is 1.98×.

We have not compared our kernels with the specialized kernels of Section 2, which target other processors. Every number here is computed from saved records by the program that builds this paper. A package for others to reproduce it is not yet ready.

14 Conclusion

We began by trying to make the weights larger so the machine could read them faster. That helped less than unfolding them, and unfolding them helped less than leaving them where they were and changing how we took them. No value changed at any step. The form the weights arrived in was never part of them, and neither was the way our kernel walked them. Both were choices, and we had been paying for them unseen. What the values themselves cost is the time their bytes need to arrive, and a kernel that imposes nothing ran at the fastest read we measured. The work left us four principles: choose the structure, know the floor, declare the route, and keep together what is taken together. One machine and a handful of

models make this a small result, and the wider tests remain to be done.

The same held beyond the journey. A matrix product has three dimensions and is walked along a fourth, and the hardware folds them into one line of memory and one line of time. Where the fold put the terms of one sum too far apart, splitting the sum gave them back. Where it put independent work in one dependent line, removing the waits the work did not need ran a decode batch 1.98× faster. The lesson we take from it is larger. Numbers are formless. Choosing the structure in which a computation takes them, rather than inheriting it, is where the speed came from.

Acknowledgments

We thank NVIDIA for the open Cosmos-Reason2 models. We thank the CUDA community, where the author's early training in machine learning began. We thank the Qwen team for Qwen3 and Qwen3.5. We thank hzang, arthurcollet and the mlx-community for the four-bit conversions we measured. We thank Superwhisper for S1-mini. We thank the authors of CV-Bench, BLINK and ERQA for their open benchmarks. We thank our AI assistant Anthropic Claude Opus 5.5 for help with software and drafting. The author is responsible for the content.

References

[1] K. Moon. Understanding the Shapes of Neural Simulation. Preprint, 2026. https://nuketorch.ai/papers/understanding-the-shapes-of-neural-simulation.pdf

[2] T. G. Kolda, B. W. Bader. Tensor Decompositions and Applications. SIAM Review 51(3), 2009. doi:10.1137/07070111X.

[3] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. Communications of the ACM 13(6), 1970. doi:10.1145/362384.362685.

[4] K. Goto, R. A. van de Geijn. Anatomy of High-Performance Matrix Multiplication. ACM Transactions on Mathematical Software 34(3), 2008. doi:10.1145/1356052.1356053.

[5] E. Frantar, R. L. Castro, J. Chen, T. Hoefler, D. Alistarh. MARLIN: Mixed-Precision Auto-Regressive Parallel Inference on Large Language Models. PPoPP 2025, 239–251. doi:10.1145/3710848.3710871.

[6] NVIDIA. FasterTransformer. https://github.com/NVIDIA/FasterTransformer

[7] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, S. Han. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. MLSys 2024. arXiv:2306.00978.

[8] turboderp. ExLlamaV2: an inference library for running LLMs on consumer GPUs. https://github.com/turboderp-org/exllamav2

[9] ggml-org. llama.cpp: online repacking of Q4_0 weights for the CPU backend. Pull requests 9921 and 10446, 2024. https://github.com/ggml-org/llama.cpp/pull/9921, https://github.com/ggml-org/llama.cpp/pull/10446

[10] J. J. Dongarra, A. R. Hinds. Unrolling Loops in FORTRAN. Software: Practice and Experience 9(3):219–226, 1979. doi:10.1002/spe.4380090307.

[11] Apple. Metal Shading Language Specification, section “Program Scope Function Constants”. https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf

[12] A. Aggarwal, J. S. Vitter. The Input/Output Complexity of Sorting and Related Problems. Communications of the ACM 31(9), 1988. doi:10.1145/48529.48535.

[13] J.-W. Hong, H. T. Kung. I/O Complexity: The Red-Blue Pebble Game. Proceedings of the 13th ACM Symposium on Theory of Computing (STOC), 326–333, 1981. doi:10.1145/800076.802486.

[14] N. Park, B. Hong, V. K. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. IEEE Transactions on Parallel and Distributed Systems 14(7):640–654, 2003. doi:10.1109/TPDS.2003.1214317.

[15] P. Budnik, D. J. Kuck. The Organization and Use of Parallel Memories. IEEE Transactions on Computers C-20(12):1566–1569, 1971.

[16] D. Irony, S. Toledo, A. Tiskin. Communication Lower Bounds for Distributed-Memory Matrix Multiplication. Journal of Parallel and Distributed Computing 64(9):1017–1026, 2004. doi:10.1016/j.jpdc.2004.03.021.

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

[18] 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, 429–431. doi:10.1145/3572848.3577479. Full version: arXiv:2301.03598.

[19] NVIDIA. CUTLASS: CUDA Templates for Linear Algebra Subroutines. https://github.com/NVIDIA/cutlass

[20] Apple. MacBook Pro (16-inch, 2024) – Technical Specifications. https://support.apple.com/en-us/121554

[21] S. Tong, E. Brown, P. Wu, S. Woo, M. Middepogu, S. C. Akula, J. Yang, S. Yang, A. Iyer, X. Pan, Z. Wang, R. Fergus, Y. LeCun, S. Xie. Cambrian-1: A Fully Open, Vision-Centric Exploration of Multimodal LLMs. NeurIPS 2024. arXiv:2406.16860. CV-Bench: https://github.com/cambrian-mllm/cambrian

[22] X. Fu, Y. Hu, B. Li, Y. Feng, H. Wang, X. Lin, D. Roth, N. A. Smith, W.-C. Ma, R. Krishna. BLINK: Multimodal Large Language Models Can See but Not Perceive. ECCV 2024. arXiv:2404.12390. Project repository: https://github.com/zeyofu/BLINK

[23] Gemini Robotics Team. Gemini Robotics: Bringing AI into the Physical World. arXiv:2503.20020, 2025. ERQA: https://github.com/embodiedreasoning/ERQA

[24] Apple. Metal Best Practices Guide: Command Buffers. https://developer.apple.com/library/archive/documentation/3DDrawing/Conceptual/MTLBestPracticesGuide/CommandBuffers.html

[25] A. Hannun, J. Digani, A. Katharopoulos, R. Collobert. MLX: Efficient and Flexible Machine Learning on Apple Silicon. 2023. https://github.com/ml-explore/mlx

[26] P. Canuma. MLX-VLM. https://github.com/Blaizzy/mlx-vlm

Appendix A: Proofs

Throughout, a tensor is a set of n names. The words partition the names into $|F|$ folds of c names each, and the draws of a kernel partition the names. $\Phi(F, D)$ counts the pairs of a draw and a word that meet (Equation 3); counted by words instead of draws, $\Phi(F, D) = \sum_{f \in F} r_f$, where r_f is the number of draws of D that meet the word f.

Theorem 1. Each of the m draws holds c names, so $n = cm$ and $|F| = m$. A draw meets at least one word, so $\Phi \geq m$. It meets exactly one word only if it lies inside that word; both hold c names, so the draw is the word. Hence $\Phi = m$ exactly when the folds are the draws. A draw of c names meets at most c words, one for each name, and at most all m words, so $\Phi \leq m \cdot \min(c, m)$. When $m \geq c$ this is $cm = n$, and a draw meets c words exactly when no two of its names share a word. ∎

Theorem 2. Let the word f meet r_f draws of D_1 and s_f draws of D_2. Each of its c names lies in one draw of each kernel, so its names lie in at most $r_f s_f$ crossings $d_1 \cap d_2$, each of at most λ names. Hence $c \leq \lambda r_f s_f$, that is, $\sqrt{r_f s_f} \geq \sqrt{c/\lambda}$. By the Cauchy–Schwarz inequality,

$$\Phi(F, D_1)\,\Phi(F, D_2) \;=\; \left(\sum_f r_f\right)\left(\sum_f s_f\right) \;\geq\; \left(\sum_f \sqrt{r_f s_f}\right)^2 \;\geq\; \frac{c}{\lambda}\,|F|^2.$$

The last step is an equality exactly when $\lambda r_f s_f = c$ in every word, so that every word holds all λ names of each of its $r_f s_f$ crossings: a full rectangle. The Cauchy–Schwarz step is an equality exactly when r_f / s_f is the same in every word. ∎

A weighted workload. Give the two kernels weights $t_1, t_2 > 0$, for how often each runs, and let $\kappa = c/\lambda$. Every word meets at least one draw of each kernel, so $\Phi_i \geq |F|$, and Theorem 2 gives $\Phi_1 \Phi_2 \geq \kappa |F|^2$. Minimising $t_1 u + t_2 v$ over $u, v \geq |F|$ with $uv \geq \kappa |F|^2$: on the curve $uv = \kappa |F|^2$ the minimum is at $u = |F|\sqrt{\kappa\, t_2 / t_1}$, which is allowed exactly when $1/\kappa \leq t_1/t_2 \leq \kappa$, and there $t_1 \Phi_1 + t_2 \Phi_2 \geq 2|F|\sqrt{\kappa\, t_1 t_2}$. Otherwise the minimum is at an end of the curve, $|F|\,(t_{\max} + \kappa\, t_{\min})$, which a fold aligned fully to the heavier kernel reaches when that kernel's draws are words and every crossing holds λ names, as in Table 3. So the bound favours full alignment exactly when one weight is at least κ times the other.

The bound treats the counts as real numbers; they are whole numbers, and this moves the threshold. In the block of Table 3, λ = 1 and c = 8, so a word meets r_f draws of the generating kernel and s_f of the prompt kernel with whole numbers $1 \leq r_f, s_f \leq 8$ and $r_f s_f \geq 8$, and the cost of a fold is $\sum_f (t_1 r_f + t_2 s_f)$. The lower convex hull of these points (r_f, s_f) has the corners (1, 8), (2, 4), (4, 2) and (8, 1); the point (3, 3) lies on the edge between (2, 4) and (4, 2), and every other point is larger in both coordinates than one of them. So every word costs at least the cheapest of the four corners, and the order of Table 3 whose words all have that shape reaches it. This holds for every fold of the block, not only for the four orders of Table 3. The unfolded order, (1, 8), beats (2, 4) exactly when $32t_1 + 256t_2 \leq 64t_1 + 128t_2$, that is, when $t_1 \geq 4\, t_2$, below κ = 8; by symmetry the file's order, (8, 1), is the cheapest when $t_2 \geq 4\, t_1$, and a tile in between.

Theorem 3. Every word holds names that some lane takes, so it is fetched at least once: the floor is n/c reads. Number the runs $0, 1, 2, \ldots$; run j covers the names $jr, \ldots, jr + r - 1$ and belongs to lane $j \bmod L$. If c divides r, a run starts at a multiple of r, hence of c, and covers r/c whole words that no other run touches; its lane fetches each once, and the tensor costs its floor. If r divides c, the word w holds the c/r consecutive runs $wc/r, \ldots, (w+1)c/r - 1$. They belong to $\min(L, c/r)$ different lanes. A lane with several runs in the word takes them one after another in its own sequence, since its runs $j, j + L, j + 2L, \ldots$ inside the word are consecutive among its runs, so it fetches the word once. Every word is fetched $\min(L, c/r)$ times. In both cases the tensor costs $\min(L, \max(1, c/r))$ times its floor. ∎

Lane counts. Store a tensor unfolded for L lanes: a word holds the c names $\ell + Lt$ for c consecutive values of t. Read it with L' lanes that each take every L'-th name. The name $\ell + Lt$ goes to lane $(\ell + Lt) \bmod L'$, and as t runs over consecutive values, $Lt \bmod L'$ repeats with period $L'/\gcd(L, L')$. So $\min(c, L'/\gcd(L, L'))$ lanes share every word, and for $c \geq 2$ no two lanes share a word exactly when L' divides L. A shared word is fetched by every lane that shares it, so the structure is at its floor only if L' divides L. (That is not sufficient: with one kept word per lane, $L' = 1$ or $L' = 16$ on a structure unfolded for $L = 32$ lanes still fetches a new word for every name, so for $L' < L$ a lane would have to keep L/L' words.) ∎

Theorem 4. The computation uses each of the n values for every token, and by assumption no part of the weights stays in a cache from one token to the next, so every value arrives from memory at least once per token. (Without that assumption, at most the capacity of the caches can stay, and the bound becomes S minus that capacity; on this machine the caches are small next to the 0.97 to 15.9 GB of weights of Table 7.) Each takes e bits, counting its share of its scales and offsets, so at least $S = ne/8$ bytes arrive per token. A structure changes only where the bytes sit, not how many there are. At B bytes per second a token takes at least S/B seconds. ∎

Theorem 5. A word holds c names, so the a needed names fill at least $\lceil a/c \rceil$ words; a structure that packs them into whole words reaches that. Each needed name lies in one word and there are $|F|$ words, so at most $\min(a, |F|)$ words must arrive; a structure that puts the needed names in different words, as far as there are words, reaches that. ∎

Appendix B: Further Measurements

This section keeps apart what is not structure: the scheduling of the decode, the cost of loading, and the distance to another engine.

B.1 Scheduling, kept apart

Table 9. A second screen with its own control. “Unfolded” is the reordered codes, the constant group size and the vision tiles together. Compare the last two rows: grouping the decode adds most once the kernel is faster.

Path	Mixed	Short text	Long text
Checkpoint order + grouped decode	1.04×	1.21×	1.11×
Unfolded	1.30×	2.38×	1.61×
Unfolded + grouped decode	1.37×	3.42×	1.92×

Grouping the decode helps more once the kernel is faster. The largest number here, 3.42× on a short text

request, belongs to structure and scheduling together. It does not belong to the permutation alone.

B.2 Loading, memory and energy

The weights are 1,781,915,648 bytes in all three paths of Table 5. Energy per panel falls by 13.4–16.1% with the selected path and by 22.6–24.6% with the file’s order and redrawn kernels, because the runs are shorter. Loading takes 546 ms in checkpoint order, 690 ms with the selected path, which reorders the codes, and 552 ms with the file’s order, which does not. Peak memory while loading is 2.76–2.77 GB in checkpoint order and 3.91–3.92 GB with the selected path. With the file’s order it was 2.72 GB in one session and 3.71 GB in the other. We have not found the cause of that rise.

Appendix C: Data Availability

Every number, table and figure in this paper is computed from saved run records by a program that reads them; no number is typed by hand. The raw run records, the test programs and the program that builds this paper are available from the author on request.