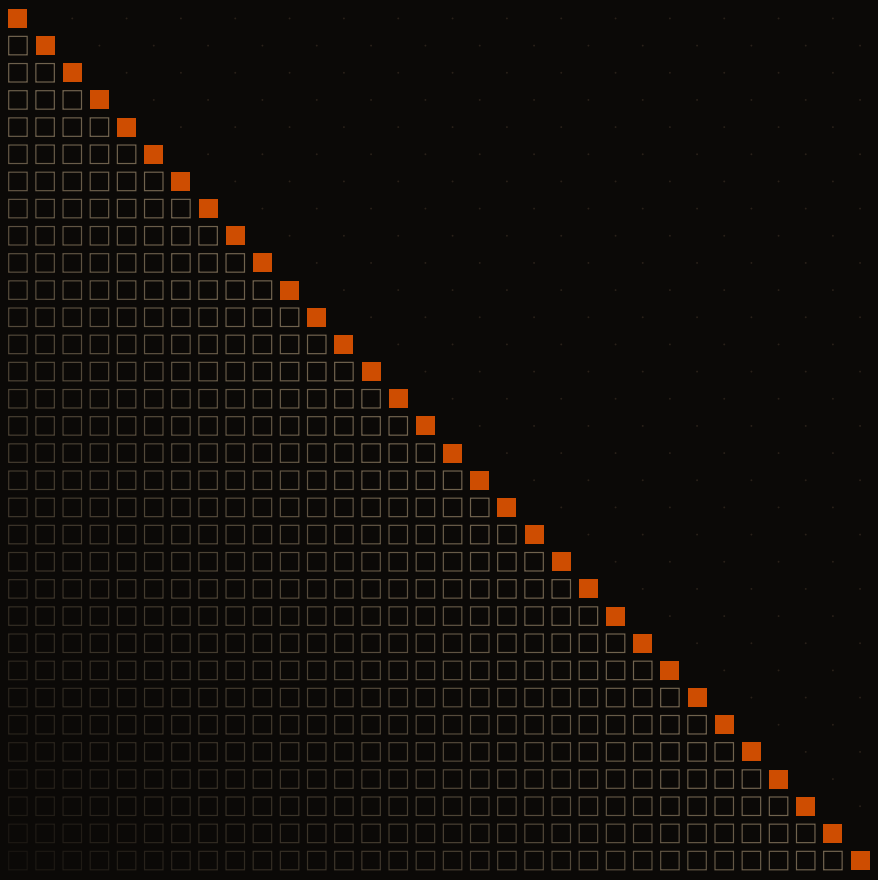


CAUSAL ATTENTION, N = 1024 · ONE SQUARE PER 32 × 32 TILE · NOTHING N × N WRITTEN — DERIVED, NOT WRITTEN

Don’t Write Kernels, Derive Them

Chasing the Speed of Light: Kernels from a Tensor IR of Views and Comprehensions

CAM PEDERSEN · MUNICIPAL ROBOTICS

Abstract

Every fast model today runs on kernels someone wrote by hand, one per operator, per chip and per fusion. Each one freezes choices about what to fuse, where data lives and how threads share work, and those choices are right for one shape on one device. We think kernels should be derived instead.

tang’s tensor IR is built from two constructs. Views are affine maps from an iteration space to tensor coordinates. Comprehensions are reductions under a monoid whose body is a scalar expression graph. Everything else is either a rewrite or a schedule. Fusion is one substitution rule. We do not write a flash attention kernel. The compiler derives its schedule by recognizing online softmax as a monoid. Gradients of comprehensions are comprehensions, so the same passes derive and optimize the backward pass. That includes a flash-style attention backward that writes no $n \times n$ intermediates on the three GPU backends we measured (on CUDA through a fused kernel that computes in bf16 and so misses an f32 tolerance, and on WGSL and Metal at a cost we report). The recent superoptimizers we discuss derive only the forward. Precision and thread mapping are schedule fields that cannot change the math (precision changes only rounding, which an emulator budgets with known failure modes), chosen per kernel by measurement on the device.

One graph lowers to WGSL, CUDA tensor cores, Metal simdgroup matrices, OpenCL, GLSL and C, from a datacenter GPU and a desktop GPU and a laptop down to a humanoid robot’s phone-class SoC. On an idle RTX 3090 a derived transformer block’s forward pass at sequence length 512 is 1.2× faster than PyTorch captured in a CUDA graph with TF32; at length 128 PyTorch is 1.2× faster, and forward plus backward ties. Qwen3-8B decodes as a single IR graph at 148.7 tok/s, tied with llama.cpp at short context and at 92% of it at 4k. We wrote no kernel for any one operator. A short list of emitters is keyed on the shape of a graph rather than on an operator: CUDA’s flash kernel (the online-softmax monoid), the fused Hopper attention backward, the narrow GEMV (matrix–vector product) stage, the integer tensor-core stage and the chunked scan plan. Derived attention is not uniformly fast. At $d = 64$ the forward trails FlashAttention-2 at every size we tested, and on the RTX 3090 no tensor-core forward is generated at $d = 128$, which makes it about 150× slower than FlashAttention-2 there. We then derive the kernels of a hand-tuned engine that serves an 80B mixture-of-experts model on one H200. With the work made equal, the derived MoE expert layer is 18% slower than the engine’s hand kernels at batch 1 and 7% slower at batch 4, ties at 8 and is 5% faster at 64. Inside the model, where the derived path runs only from 16 window rows up, a decode step takes 5–12% less time at 16 to 48 concurrent streams over five interleaved rounds on one H200, though the derived path misses the engine’s 48-stream agreement gate, 98.93% against a 99% bar.

The derived head-dim-256 attention and gated-delta scan cut the engine’s own time to first token at 44k tokens from 8.33 s to 5.54 s, but vLLM on the FP8 checkpoint reaches first token about 12× faster than the engine, so this is a gain over our own baseline, not over the state of the art. At $d = 256$ the derived attention runs at 0.43–0.54 of FlashAttention-3′s speed forward and 0.22–0.40 backward.

Finally, we ask how far a kernel is from the fastest any implementation could be. For a comprehension over affine views, the least memory traffic and the tile that attains it are the primal and dual of one linear program, and we prove weak duality in Lean for arrays indexed by subsets of the loop coordinates, along with the soundness of the core rewrites. A time floor built on that bound, with launch cost measured on both GPUs and DRAM latency measured on the H200 and assumed on the RTX 3090, never exceeds the measured time on the 24 GEMV-class and epilogue kernels we validated on two GPUs. On the RTX 3090 it reaches 0.76–0.93 of measured time on the large kernels. On the H200 it reaches only 0.05–0.66 (median 0.38), and we have no validated term for the gap. Inside the schedule space, a two-level version of the program picks the tile the M4 Max autotuner picks, but only for register capacities between 4096 and 8192 words, and the traffic-optimal tile does not win on time. In a timed race of every candidate tile on the M4 Max and the RTX 3090, no candidate ran faster than the floor. A per-tile estimate built on it picks a tile within 2.5% of the fastest on seven of eight kernels and loses the eighth by 1.27×.

1 Introduction

A modern model has a few dozen distinct operators. A modern framework has thousands of kernels. Every (operator, backend) pair needs one. So does every fusion that pays, such as attention with its softmax, a matmul with its bias and activation, or a norm with the projection after it. FlashAttention[1] is the best-known case. Its large speedup came from rewriting one operator by hand so its $n \times n$ score matrix never reaches memory. Its backward pass took a second hand-written kernel. Hopper took a third[2], and Blackwell's FP4 tensor cores are taking a fourth.

Each hand kernel is a separate copy of the same arithmetic, and each one freezes a schedule that was right for one shape on one device. We measured how often that schedule is wrong: the fastest schedule changes with the size, the chip and even between runs on the same machine (§7.1). The winning schedule is a property of the graph, the chip and the size, not of the operator.

Our thesis is that **a kernel is a theorem, not a program**. Every fast kernel we know of is a claim of the form "this comprehension, reassociated under this monoid and placed in this memory, equals the naive one over the reals." Such a claim can be derived, checked and searched. It does not need to be written. It can also be bounded: the least traffic any box-tiled schedule of a comprehension can move, and the tile that moves it, are dual solutions of one linear program. A theorem fixes what is computed. How fast it runs is still a schedule, and §11 lists the schedules tang does not yet have. If that is right, the kernel library should be replaced by an algebra small enough to reason about, together with a search over its schedules. Figure 1 shows one graph under the schedules that the field knows by name.

FIG 1 One causal attention graph ($n = 4096$, $d = 64$, f32) under four schedules, lowered to CUDA on an idle RTX 3090. Only where $S = qk^\top$ and $P = \mathrm{softmax}(S)$ live, and where kernels end, differ. Checkpointing and flash are points in this space, and the search lands on the flash placement: 18× faster than as written and 40× faster than checkpointing. Every point is checked against the f64 interpreter at this size, with tolerance 2×10^{-3}; flash and the searched schedule use TF32 tensor cores. "Traffic" is the compiler's model of DRAM bytes, one pass per buffer, and "rel. error" is the largest error over the largest reference value. Times are medians of 7 samples.

SCHEDULE	KERNELS	TRAFFIC	MS	RANGE	VS NAIVE	REL. ERROR
as written	11	815.9 MB	4.21	4.18–4.65	1×	2.2×10^{-6}
checkpointing	5	274.8 MB	9.33	9.33–9.41	0.5×	2.2×10^{-6}
flash	1	4.2 MB	0.24	0.24–0.25	17.4×	5.2×10^{-4}
searched	1	4.2 MB	0.23	0.23–0.24	18.1×	5.2×10^{-4}

The traffic column is not the time column. Checkpointing moves a third of the naive schedule's traffic and is 2.2× slower. The score contraction $S = qk^\top$ accounts for more than the whole difference: 7.12 ms against 1.51 ms, with the same arithmetic and the same output. The naive schedule writes $k^\top$ in its own small transpose kernel, so a warp's 32 lanes then read 32 adjacent floats. Checkpointing folds the transpose into the reader's view, so each lane reads its own row of k. Nsight shows the same 67.6 M L1 requests carrying 5.4 GB in one case and 35.5 GB in the other. The output is exact to 2×10^{-6}, so the lowering is correct but loses coalescing, and a traffic

model that counts DRAM bytes once per buffer cannot see it. Flash, which also has the least traffic, wins.

The backward tells a less flattering story. tang's fused attention backward kernel computes in bf16 tensor cores even when the graph is f32. On this graph its gradients miss the experiment's tolerance of 2×10^{-3}, measured as max error over max reference against the f64 interpreter (error 1.5×10^{-2} in dq), so the search rejects it. Without the fused kernel the searched placement passes at 1.2×10^{-3} in 1.91 ms, against 0.74 ms for the failing fused version. The search also rejected one placement whose output was simply wrong, at error 7.0; that bug is caught by the check but not yet explained.

We propose writing the model once, as math, and letting the compiler derive every kernel. That works only if the IR is small enough to reason about and expressive enough that nothing important falls outside it. tang is a `no_std` Rust library that started as a geometry and physics math crate. Its `tang-expr` crate traces Rust closures into scalar expression graphs that it can differentiate, simplify and compile, and its tensor IR uses those graphs as its body language. The IR has two constructs.

- **Views**: affine index maps that express broadcast, transpose, slicing, striding, padding and per-head column selection without touching memory.
- **Comprehensions**: $\text{out}[i] = \oplus_k \text{body}(x_0[v_0(i,k)], x_1[v_1(i,k)], \ldots)$, where `body` is a scalar expression graph and $\oplus$ is a monoid.

Contributions:

1. **Kernels from rewrites.** Fusion, flash attention, logsumexp and Welford variance are instances of a small set of rewrites (§2).
2. **Derived backward passes.** Reverse-mode differentiation maps comprehensions to comprehensions, including the online-softmax monoid. A proof procedure removes the stabilizing max's gradient, and with nested contractions this yields an attention backward with no $n \times n$ tensors (§3).
3. **Precision as a schedule.** An emulating interpreter checks a format choice against an error budget before it runs, with failure modes we report (§4).
4. **One graph, six targets.** Lowerings from shared emitters, measured against hand kernels, PyTorch and the bandwidth limit on four chips (§5, §7).
5. **A production engine, derived.** Top-k routing as one monoid, integer tensor-core and narrow GEMV schedules, a segmented-sum gather backward, chunked scans, and Hopper attention schedules. With them, derived kernels replace hand kernels in an 80B MoE engine on an H200, faster in some regimes and slower in others, which we report (§8, §11). A whole derived decode step matches the hand engine on Qwen3-8B and trails it by 13–35% on the 80B MoE.
6. **The speed of light.** The traffic lower bound and the optimal tile are dual solutions of one linear program read off a kernel's views, with weak duality in Lean. A time floor built on it never exceeds measured time on 24 GEMV-class and epilogue kernels on two GPUs; it is close on the large kernels of the RTX 3090 and loose on the H200 (§9).

2 The IR

2.1 Views

A view maps a point of an iteration space to a source tensor’s coordinates with one affine form per source dimension, offset $+\sum_j c_j \cdot \text{idx}_j$. Broadcast is a zero form, transpose a permutation, slice an offset, stride a coefficient. A form with two terms expresses “head h reads columns $64h + 2i$”. Views compose by substitution. They are the only way a comprehension reads its inputs, so every layout decision is a view and every view can be analyzed.

A view may be guarded, so that an out-of-range read returns the monoid’s identity (zero for sum, $-\infty$ for max) rather than raising a shape error. Padded, dilated and transposed (polyphase) convolutions and overlap-add are guarded reads, with no padding operator. The adjoint of a guarded read is itself a guarded read, not a scatter. For $y[o,t] = \sum_{c,k} w[o,c,k]x[c,t+3k-9]$, $\partial x[c,\tau]$ reads $\partial y[o,\tau-3k+9]$ through the same guard, so out-of-range (padding) terms never reach x.

Grouped-query attention’s head / heads_per_group is not affine. The IR expresses heads as two dimensions, [groups, per_group], and the views do the rest. Dimensions can be fixed or symbolic (`Dim::Sym("seq")`). Construction, rewriting and differentiation work with symbols, and `bind` makes them concrete for scheduling and lowering.

A view may also gather. One source coordinate is then data, read from an index tensor that is a sibling argument of the same comprehension, form $+ c \cdot \text{round}(\text{idx})$. MoE routing ($W[\text{sel}[t,k]]$), paged KV caches ($b_s \cdot \text{table}[\text{blk}] + \text{off}$) and embedding lookups become ordinary reads rather than special operators. The gradient of a gathered read is a scatter-add, written as a dense gated sum, which costs E times the needed work unless the keys are declared sorted (§8). An affine map read through a gather still fuses into it, and only nested gathers materialize.

2.2 Comprehensions and monoids

The only compute operation is

$$\text{out}[i_1, \ldots, i_p] = \bigoplus_{k_1,\ldots,k_r} \text{body}(x_0[v_0(i,k)], \ldots, x_m[v_m(i,k)])$$

A map is a comprehension with no reduce dimensions. Sum and max are monoids. Matmul is $\sum_k a[i,k]b[k,j]$. Attention scores, convolution and gather are the same operation with different views. The monoid’s `combine` is itself a scalar graph, and a comprehension may carry tuple-valued state. Online softmax is the monoid on (m, l), the running max and the rescaled sum, and attention carries the weighted values with it as (m, l, acc); arg-max is the monoid on $(\text{index}, \text{value})$.

2.3 Rewrites

Fusion is one rule. A map is a pure function of its inputs, so `substitute_maps` splices its body into every consumer and composes the views. Map→map, map→reduce and transpose→anything are the same case. The rule is cost-aware, so a map is recomputed inside a reduction only when it is cheap. Reductions are never substituted. They are inlined by a schedule decision, which separates algebra from scheduling. A row-major reshape that only merges dimensions is a view, not a copy.

Epilogues run on the reducing thread. A map that reads a reduction point for point (a bias, an activation, the acc $/l$ that finishes softmax) runs on the thread that holds the final accumulator.

Sibling reductions merge. Two reductions over the same space and monoid with the same readers become one reduction under the product monoid (`merge_siblings`). The gate and up projections of a SwiGLU MLP become one tuple-valued matmul, the activation becomes its epilogue, and the normalized input is read once.

Flash attention is a rewrite. `stabilize_softmax` finds $m = \max_j s_j$ and every $\sum_j g \cdot \exp(s_j - m)$ over the same canonical s, with masks and scales allowed, and replaces them with one reduction under the online-softmax monoid. The value column of $\sum_j p_j v_j$ becomes a point dimension (an output dimension, not reduced over) of the fused comprehension. Nothing $n \times n$ is materialized. Logsumexp, Welford variance and top-k are the same rewrite with different monoids. Figure 2 shows the attention case.

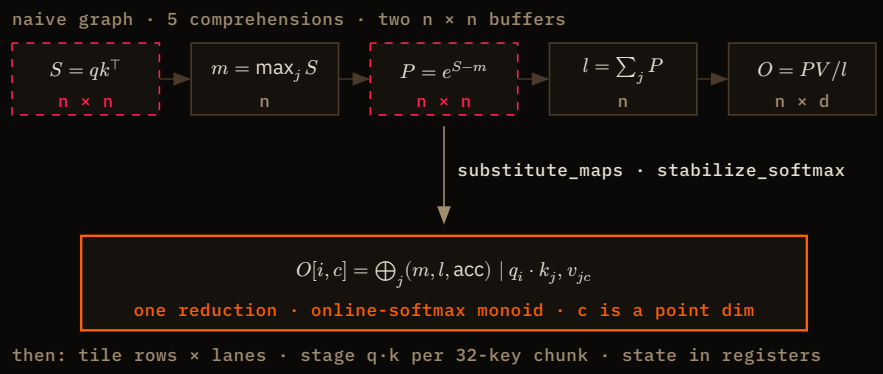


FIG 2 The rewrite that derives flash attention, schematically (the lab's typed graph also carries mask and scale maps, so the §2.3 counts are higher). The naive graph writes the scores and probabilities as $n \times n$ buffers (dashed). Fusion splices the maps into their readers; `stabilize_softmax` merges the max and both exponential sums into one reduction whose state is (m, l, acc). No operator-keyed code is involved; scheduling then finds the tiled kernel.

The rewrite is not specific to plain attention. Anything that changes the score or the mask leaves the monoid structure alone, so it derives the same way. Table 1 lists six variants the compiler derives.[1]

[1] The web edition derives these in the reader's browser and runs them through WebGPU: muni.works/research/tang.

TABLE 1 Six attention variants as typed into the lab, $n = 1024$, $d = 64$. Each cell is kernels · $n \times n$ buffers, as written → derived (WGSL). The forward always becomes one kernel with nothing $n \times n$; with nested contractions (§3) the backward also writes nothing $n \times n$, in 7 kernels. Generated by `tang-derive-wasm`'s `gallery` example.

VARIANT	SCORE	MASK	FWD KERNELS · N×N	FWD+BWD KERNELS · N×N
Causal	`s`	`j <= i`	11 · 5 → **1 · 0**	30 · 11 → **7 · 0**
Sliding window	`s`	`j <= i && i - j < 128`	11 · 5 → **1 · 0**	30 · 11 → **7 · 0**
ALiBi	`s - 0.0625*(i - j)`	`j <= i`	11 · 5 → **1 · 0**	30 · 11 → **7 · 0**
Soft cap	`30*tanh(s/30)`	`j <= i`	11 · 5 → **1 · 0**	30 · 11 → **7 · 0**
Prefix LM	`s`	`j <= i || j < 64`	11 · 5 → **1 · 0**	30 · 11 → **7 · 0**
Bidirectional	`s`	—	10 · 4 → **1 · 0**	28 · 9 → **7 · 0**

2.4 Rediscoveries

The rewrites above are general, so known algorithms fall out of them. Table 2 lists the ones tang derives today. Each row was a paper or a hand kernel. In tang each is a monoid, a view or a schedule field, and none adds an operator.

2.5 Schedules

A comprehension says what to compute. Schedule fields say how, and they cannot change the math (precision changes rounding, §4):

TABLE 2 Algorithms that tang derives rather than implements. Each derivation is checked against the f64 interpreter of the naive graph.

ALGORITHM	ORIGINALLY	IN TANG
online softmax	Milakov & Gimelshein 2018[3]	monoid on (m, l)
FlashAttention forward	Dao et al. 2022[1]	`stabilize_softmax` + staging rule (which operands a tile copies into shared memory or registers)
FlashAttention backward	hand kernel[1]	derived gradient, max elimination, nested contractions (§3)
Welford variance	Welford 1962[4]	monoid on (n, μ, M_2)
logsumexp, arg-max	folklore	monoids on (m, l) and (index, value)
top-k routing	hand kernel	`Monoid::topk`, merge of sorted lists
split-K GEMM	hand kernel	`split` on any associative monoid
SwiGLU gate|up fusion	hand kernel	`merge_siblings` under a product monoid
implicit-GEMM convolution	hand kernel	guarded views, flattened reduce dims
paged KV cache	vLLM	gather view
scatter-add backward, segmented	hand kernel	gated sum + `set_sorted` segment (§8)
chunked SSD / gated delta rule	Dao & Gu 2024[5]; Yang et al. 2025[6]	affine scan monoid + chunked plan (§8)

- `inline`: recompute inside the consumer instead of writing a buffer.
- `split`: one workgroup per point, with threads striding the reduce space and tree-combining. This is valid for any monoid because combine is associative, including online softmax.
- `tile`, `tile_2d`: blocks of threads, each owning a register tile of points. The emitter tracks every address as an affine form over loop variables. Anything that does not depend on the lane axis is staged in workgroup memory once per reduce chunk. That covers buffer reads (K and V tiles, matmul operand tiles) and inlined sub-computations (the $q \cdot k$ scores).
- `precision`: storage and accumulation formats (§4).

`schedule_auto` encodes rules of thumb distilled from measurement. `autotune_kernels` times each candidate mapping per materialized node on the device, using the graph's real inputs, and caches the winner by device and kernel source. With substitution and the online-softmax rewrite, the staging rule produces the structure of flash attention: rows × lanes per workgroup, cooperative score tiles, and online softmax in registers.

3 Gradients are comprehensions

For $y = \oplus_k \text{body}(x[v(i,k)])$, the gradient with respect to x through a pick view (one whose forms are each a single coordinate) is found by renaming coordinates. The coordinates that v picks become x's output dimensions, the rest become reduce dimensions, and every other view is pushed through the same renaming. The body is differentiated by `tang-expr`'s symbolic `diff`, which covers its nine arithmetic operations (add, multiply, negate, reciprocal, square root, sine, atan2, 2^x, $\log_2$) plus a select. A dimension read through any other form (an offset, stride, diagonal or broadcast) stays an output dimension, and `iota`, a node that turns coordinates into values, gates the body by $[c = \text{form}]$. Max gates its term to the arg-max. The backward graph is an ordinary graph, so the same passes optimize it. Rotary embedding and GQA (grouped-query attention) gradients come from the gated scatter with no special handling.

The online-softmax monoid has its own rule. With the forward's m and accumulators read back, $\partial \text{acc}_i / \partial s_{ij} = g_j \exp(s_{ij} - m_i) - \text{acc}_i \, [s_{ij} = m_i]$, where g_j is key j's value, so the flash-fused graph differentiates and its gradients equal the naive graph's.

A stabilizing max has no gradient. That the max can be stopped is standard practice in logsumexp implementations. What is new here is a proof the compiler runs per graph. Softmax is invariant to shifting its scores, so the $[s = m]$ terms sum to zero, but only in exact arithmetic. While those terms exist, the max must be materialized and masked tiles cannot be skipped. `backward` drops them when it can prove the loss does not depend on m. The proof tracks how each node downstream of m scales with $t = \exp(-m)$. In softmax, $p_j = \exp(s_j - m) = t \cdot \exp(s_j)$ has degree 1 in t, so does $l = \sum_j p_j$, and so does $\text{acc} = \sum_j p_j v_j$; their ratio $o = \text{acc} \ / l$ has degree 0, a function of the scores alone. In general a node is either $t^k h$ or $a + \kappa m$ with h, a free of m, combined

by exact identities.[2] If every path from m ends at a degree-0 node, $\partial L/\partial m \equiv 0$ and the gradient is not built. The f64 evaluation of a training step is unchanged (largest absolute difference 2.7×10^{-19}).

The flash backward recomputes inside tiles (Figure 3). `hoist_invariant_factors` factors row-invariant terms out of sums ($\sum_c \tilde{P} \cdot \mathrm{dO} \cdot V \to \tilde{P} \cdot \sum_c \mathrm{dO} \cdot V$). `schedule_nested_contractions` then inlines a short contraction (one reduce dimension of at most 128) into every tile that reads it. The attention backward reads $S = QK^\top$ and $\mathrm{dP} = \mathrm{dO}\ V^\top$ only through $\tilde{P}$ and dS, which the dQ, dK and dV tiles recompute per staged block on tensor cores. On CUDA this removed S, $\tilde{P}$, dP, the max over S and three correction kernels from a transformer training step, reducing 63 kernels to 49 and the summed best-per-kernel time from 1.80 to 1.22 ms at sequence length 512 (RTX 3090). WGSL and Metal now nest the same contractions with scalar dot products in workgroup memory, so no GPU backend we measured (CUDA, WGSL, Metal) writes anything $n \times n$. For now this costs speed. Forward plus backward runs at 0.68–0.93× the materialized schedule on Metal and 0.25–0.34× under Vulkan on a 3090, until the nested dots are register-blocked the way the `mma` path blocks them.

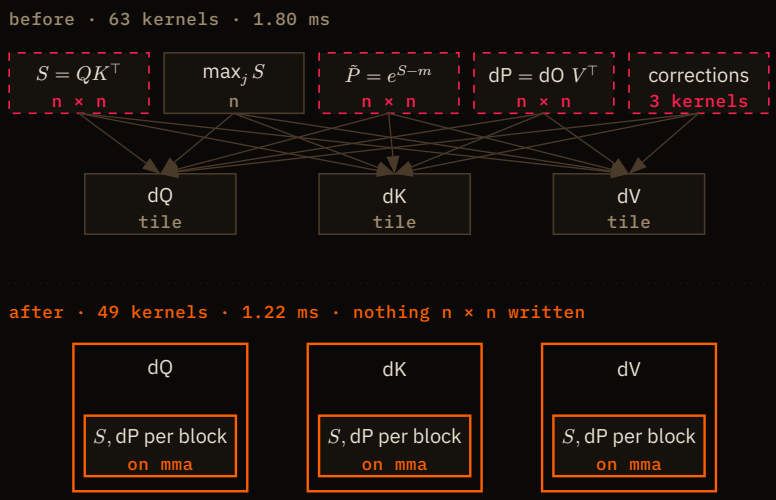


FIG 3 The attention backward of a training step, before and after. Before, the scores, probabilities and dP are written as $n \times n$ buffers (dashed) and read by three gradient tiles. After the stabilizing max loses its gradient and invariant factors leave the sums, the short contractions S and dP are recomputed per staged block inside each tile, on tensor cores. CUDA, RTX 3090, seq 512; summed best-per-kernel times.

4 Precision is a schedule

f16 is a different way to run the same program, not a different program. Each node has a storage format, applied where a value is written to memory, and an accumulation format, applied to a reduction's running state after each combine. The graph stays exact math.

The interpreter gives an error budget, with known failure modes. `eval` is pure f64. `eval_emulated` rounds every stored value and every accumulator step to its scheduled

[2] $t^a t^b = t^{a+b}$; $\sum_j t h_j = t \sum_j h_j$ when t's row does not depend on j; $2^{a+\kappa m} = 2^a t^{-\kappa \ln 2}$; $\log_2(t^k h) = \log_2 h - km/\ln 2$, again of the form $a + \kappa m$.

format (round to nearest even, subnormals, overflow), with body arithmetic in f64. An `ErrorBudget` accepts or rejects a schedule before it runs. Emulation matches the GPU's f16 kernels to 1 ulp when the reduction order is the same. Otherwise the order is part of the error: AdaIN's variance over $T = 2000$ with an f16 accumulator has error 2.8×10^{-4} under a 256-thread split and 1.3×10^{-2} with one thread per row, while the sequential emulation predicts 7.9×10^{-3}. Emulation models one sequential fold; a tree-combining split beats it, and the per-row GPU kernel, though also sequential, comes out somewhat worse, which we attribute to its different accumulation order and fused arithmetic rather than having verified it. The prediction can be far off when the reduction length or order differs: a conv1d sum of 3,584 terms measures 3.1×10^{-2} against a predicted 5.6×10^{-3}, a ConvNeXt LayerNorm statistic 6.9×10^{-2} against 1.5×10^{-4}, and a pointwise matmul 1.7×10^{-2} against 1.4×10^{-4}. Those predictions were made on smaller instances, whose shorter sums round less. At equal reduction length (matmul, AdaIN) prediction and measurement agree within about 1.7×. `tune_precision` lowers buffers to half precision one move at a time and keeps a move only if the program stays within budget and gets faster.

5 Lowering

Shared emitters produce the point, split and tiled schedules, register tiles, guarded views, epilogue fusion and tuple monoids once. A dialect switch then handles buffer access and surface syntax.

- **WGSL** (portable default, through wgpu on Metal, Vulkan and DX12). f16 storage is packed two per `u32`, because wgpu's shader compiler (naga) rejected `enable f16` on our targets.
- **CUDA.** Matmul-shaped tiles run on `mma.sync` (TF32, the tensor cores' 19-bit float mode, m16n8k8, or f16/bf16 m16n8k16 when the operands are half precision) with `cp.async` staging rings. Block shapes are tuning candidates. `cuda_flash_kernel` recognizes the online-softmax comprehension and keeps O, the row max and the partial sums in `mma` registers. The score accumulator is reused as the A operand of $O += PV$ after permuting keys within each 8-wide step, which is legal because the reduction is commutative and associative.
- **Metal (MSL).** Native `half` and `bfloat`. Contractions with f16 operands run on `simdgroup_half8x8` fragments. Reduce dimensions are flattened, so convolution becomes implicit GEMM, with dilated inputs staged once per chunk as a zero-padded, phase-de-interleaved window.
- **OpenCL and GLSL** for Adreno GPUs that have neither CUDA nor Metal. On one Adreno driver, WGSL is routed through GLSL because the direct SPIR-V path was miscompiled.
- **C**, for batch-1 control loops where a GPU round trip costs more than the arithmetic. A comprehension already is a loop nest, so `to_c` emits it directly with a vectorizable innermost point dimension, and `tang_run()` performs no allocation.

`Program::tuned` times each kernel's native lowering against the portable one and keeps the faster. Kernel sources are canonicalized, so the same kernel at another layer is the same string and compiles once.

6 Verification

Speed claims depend on every kernel being correct. `tang-verify` runs each GPU kernel against its CPU oracle on the same input, starting from the device's own state. Before each kernel runs, it fills the kernel's outputs with NaN, so unwritten memory cannot pass as a plausible 0. When a check fails, it names the kernel at fault by provenance: which earlier kernel's correct output the failing buffer contains. We built it after four structural bugs in a hand-written engine slipped past every unit test. One was a byte offset added where a float offset was meant, which wrote k's values over q. Derived kernels are less exposed to that class of bug, because addresses come from affine forms rather than pointer arithmetic. On the IR path there is a second check, because the f64 interpreter runs the same graph as the GPU.

The rewrites themselves are mechanized in Lean 4 with Mathlib, with no `sorry` (`proofs/`, listed in `docs/proofs.md`). Over the reals, view composition, fusion, split and tiled reductions, online softmax as a commutative semigroup (the proof has no identity element, and the compiler supplies $(-\infty, 0)$), its equality with two-pass softmax, shift invariance, gradients of sum comprehensions, the zero gradient of a stabilizing max, and the flash backward identity are theorems. One combine exported from the compiler is proved equal to the verified online-softmax combine. Floating point is not modelled, and guarded and gathered views are outside the fusion theorem.

7 Evaluation

Setup. Apple M4 Max (Metal), NVIDIA RTX 3090 (Vulkan and CUDA 12), a Qualcomm QCS8550 in a Booster K1 humanoid robot (Adreno 740, Cortex-X3), and, for §8, an NVIDIA H200. PyTorch 2.10 for the shared-GPU runs and 2.8 for the idle-GPU runs, both cu128, timed with CUDA events. Unless a caption says otherwise (Table 3 includes submit and readback), tang times are kernel-only. Several runs shared the GPU with other tenants. We mark those, and report minima rather than means, because contention only adds time. Derived results are checked against an f64 interpreter, except where a caption says otherwise. The tolerance differs per experiment and is stated with it: 10^{-6} to 10^{-4} relative for the kernels of §7.1–7.2, 2–5 $\times$ 10^{-3} for bf16 attention at $d = 256$, and arg-max agreement at 257 positions for the Qwen3-8B step.

7.1 Derived vs hand-written kernels

tang shipped hand-written WGSL kernels before the IR existed, which makes a controlled comparison possible: same runtime, same chip, same inputs. Here "as written" is the unfused graph, one kernel per comprehension; "derived" is any schedule the compiler produces, and the autotuner keeps the as-written graph's naive schedule as one candidate, which is why it can win. On the M4 Max the fastest

schedule for causal attention flipped between two runs of the same code: in one run $n = 512$ was fastest naive (n^2 threads) and $n = 1024$ tiled, and Table 3, a later run, has the reverse. The naive graph wins when its scores fit in cache and parallelism, not bandwidth, is the bottleneck, which depends on the size and the machine's power state. A 16×16 shared-memory matmul tile wins at $128 \times 576 \times 1536$ and loses at 1024^2, and rules of thumb tuned on the M4 picked the wrong matmul schedule on an RTX 3090. Table 3 is one consistent run on the M4 Max, including about 1.8 ms of submit and readback per measurement.

TABLE 3 Hand-written vs derived on M4 Max (Metal via WGSL). Every time includes about 1.8 ms of submit and readback, which dominates the small ops. The hand training step ran its attention backward on the CPU, since no GPU version existed, so that row shows coverage, not kernel quality.

OP	HAND	DERIVED	SCHEDULE
causal attention $n = 1024$	10.9 ms	**3.31**	naive (n^2 threads)
causal attention $n = 512$	4.91	**2.02**	auto (tiled)
matmul 1024^2	6.68	5.49	auto
rmsnorm 2048^2	2.52	2.70	split
transformer block, seq 512	23.9	**10.1**	per-kernel autotune
decode block, q 1 / kv 1024	8.80	**3.28**	per-kernel autotune
training step, seq 128	179	**7.96**	auto

Leaving aside the training step, the derived kernels win or lose by small margins on the small ops. With the 1.8 ms overhead subtracted, matmul is 1.32× faster than hand and rmsnorm 1.25× slower. The schedules that win differ across operators and sizes. On the 3090 under Vulkan, the derived transformer block first lost to the hand path. Three chip-agnostic fixes followed: cost-aware substitution, keeping reductions schedulable under their epilogues, and a bank-conflict-free tile layout. After them, the autotuned block won there too (2.5 vs 3.1 ms at seq 128, 6.2 vs 10.4 ms at seq 512, both including about 1.25 ms of submit and readback).

7.2 Against PyTorch

Table 4 tracks a 9-head GQA transformer block ($d = 576$, ff $= 1536$, rotary embeddings, SwiGLU) on the RTX 3090 as the backends matured. Through WGSL, which has no access to tensor cores, derived kernels stayed 2.6–10× behind PyTorch. With the CUDA lowering, flash attention on `mma.sync` and the backward passes of §3, the seq-512 forward block passes PyTorch (Figure 4).

TABLE 4 Transformer block on a shared RTX 3090 against eager PyTorch, an earlier run. The CUDA column and PyTorch were measured in interleaved rounds under `flock`, best of six; times in ms. The WGSL columns come from earlier runs and are shown to make the trajectory visible. Table 5 repeats the comparison on an idle GPU against PyTorch's faster modes, with a later build of tang, and is the one to cite.

BLOCK, SEQ	WGSL FIRST	WGSL FUSED	CUDA	EAGER TORCH FP32/TF32
fwd 128	1.97	1.06	0.178	0.306 / 0.286
fwd 512	4.41	3.34	0.328	0.602 / 0.449
fwd+bwd 128	5.0	3.15	0.552	1.196 / —
fwd+bwd 512	10.1	8.19	1.467	1.719 / —

On the CUDA path, the derived inference attention kernel alone is 0.032 ms at seq 512, down from 0.178 ms before the `mma.sync` rewrite. Table 4 compares against eager PyTorch, which launches about ten cuBLAS and flash kernels per block forward and is bound by launches at these sizes. Against a fair baseline the picture is narrower (Table 5). On an idle RTX 3090, PyTorch captured in a CUDA graph with TF32 beats tang's forward at seq 128 (0.162 against 0.196 ms) and ties forward plus backward at seq 128 and 512. tang still wins the seq-512 forward against every PyTorch mode, 0.331 ms against 0.396 ms for the CUDA graph and 0.407 ms for `torch.compile`. A run on a contended GPU had tang's tuned training step at 5.05 ms against PyTorch's 4.21 ms, so the training-step ranking is not established.

TABLE 5 The same block against PyTorch's faster modes on an idle rented RTX 3090. Best of six interleaved rounds; medians are within 0.02 ms of these except tang's fwd+bwd at 128 (0.503). Within 2% we call a tie.

BLOCK, MS	TANG, CUDA	TORCH.COMPILE, TF32	CUDA GRAPH, TF32
fwd 128	0.196	0.388	**0.162**
fwd 512	**0.331**	0.407	0.396
fwd+bwd 128	0.482	1.553	0.484
fwd+bwd 512	1.127	1.724	1.142

We do not claim a general win over cuBLAS: a lone 1024^2 matmul on the derived CUDA TF32 tile takes 0.103 ms at the default block shape and 0.088 ms at its best one, against cuBLAS TF32's 0.079 ms, 1.1× slower. The gain comes from fusion across operator boundaries that a library of separately tuned operators cannot do.

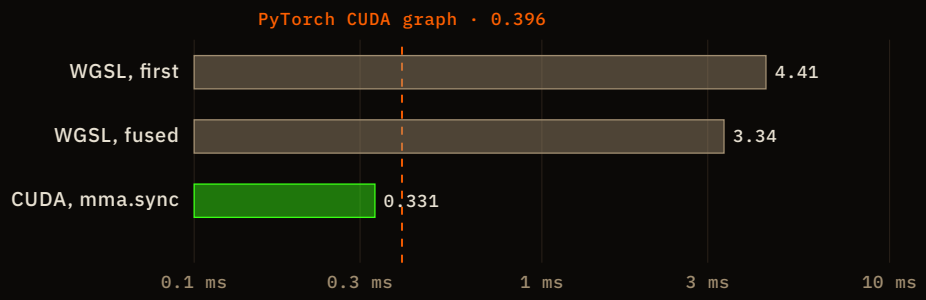


FIG 4 Transformer block forward, seq 512, RTX 3090, log scale. The same derived graph under three generations of lowering; only the last reaches tensor cores. Dashed: PyTorch captured in a CUDA graph with TF32. The CUDA bar and the dashed line are from the idle-GPU run of Table 5; the WGSL bars are from earlier shared-GPU runs.

7.3 Attention variants against FlexAttention and FA2

We ran the six attention variants from §2.3 on the RTX 3090 against FlexAttention[7] (`torch.compile`, fp32 with TF32 and bf16) and FlashAttention-2 in bf16 where it supports the mask. FA2 here means the kernels bundled with PyTorch, run through its scaled-dot-product-attention (SDPA) `FLASH_ATTENTION` backend, because the `flash_attn` package could not be built on the test machine. Flashlight and EqiForge publish no code. Every row was checked against an f64 reference; tang is pure fp32 or TF32.

TABLE 6 Attention variants on an RTX 3090 (shared; small rows carry up to ±2× noise). FA2 is PyTorch's bundled kernel and runs only causal and bidirectional masks. The fwd + bwd row uses the earlier 9-kernel backward with materialized $n \times n$ buffers, before the §3 and §8.3 rewrites. The full sweep, d = 64 and 128 with n from 1k to 8k, is in the repository.

D = 64 UNLESS NOTED, MS	TANG	FLEX FP32	FLEX BF16	FA2 BF16
soft cap, fwd, n = 2k	**0.075**	0.254	0.172	—
ALiBi, fwd, n = 2k	**0.060**	0.171	0.094	—
causal, fwd, n = 2k	0.060	0.171	0.090	**0.023**
causal, fwd, n = 8k	0.888	0.635	0.326	**0.177**
sliding window 128, fwd, n = 8k	0.911	0.037	**0.034**	—
causal, fwd + bwd, n = 4k	1.81	0.968	**0.420**	0.469
causal, fwd, n = 4k, d = 128	17.1	fails	0.318	**0.112**

Where a variant changes the score (soft cap, ALiBi), the derived forward is up to 3.4× faster than FlexAttention in fp32 up to $n = 4$k and beats it in bf16 up to $n = 2$k (the low end of that range is within the noise of a shared GPU), because the score modifier becomes part of the fused body. tang loses in three places, each a missing schedule rather than missing algebra. FA2 runs bf16 tensor cores where tang's tile is TF32. A sliding window is 25× slower because tang computes masked tiles that FlexAttention's block mask skips. And at $d = 128$ no tensor-core tile is chosen for the forward, a 150× gap at $n = 4$k in Table 6. Across the full sweep the $d = 64$ forward wins 0 of 8 against FA2 and the $d = 128$ forward 0 of 24 against FlexAttention bf16. From $n = 2$k the backward is 2.5–7× behind FlexAttention bf16, and 50× with a sliding window.

7.4 A whole model as one graph

Qwen3-8B with MLX 4-bit weights runs as one IR graph per decode step, lowered by `to_cuda_decode` into 584 kernels and captured as a CUDA graph. Nibble unpacking and the $q \cdot s + b$ dequantization are a map the GEMV reads. That map is inlined and evaluated in registers as weights stream in. Figure 5 shows the graph.

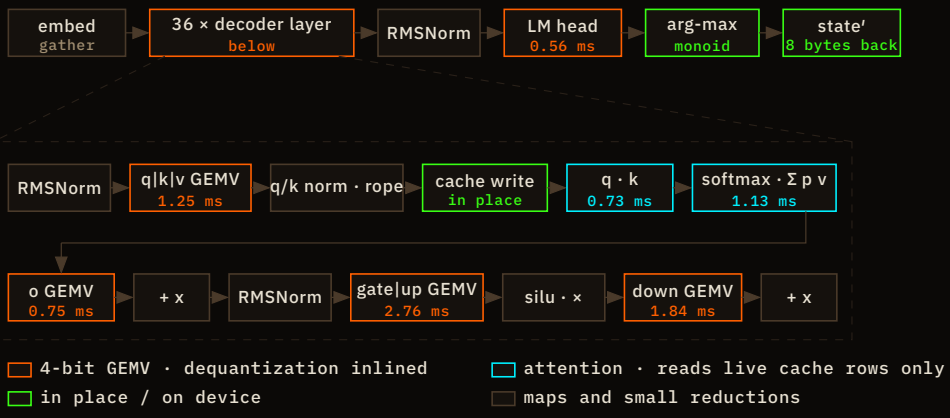


FIG 5 One Qwen3-8B decode step as a single IR graph. Top: the step, from the token in `state` to the arg-max written back. Bottom: one of the 36 identical layers. Times are each node's share of the 4k-context step on the 3090, summed over layers, from the first build (Figure 6). The current lowering emits 584 kernels for the whole step, captured as one CUDA graph.

The KV-cache update is a `select` over a runtime position, which the lowering turns into loop bounds, so a step touches only live cache rows. Table 7 compares it with llama.cpp on the same card (engineering notes in the appendix, Decode engineering notes), run back to back on the same weights. The model's greedy output matches an exact f32 reference token for token.

TABLE 7 Decode, one stream. llama.cpp runs `llama-bench -n 128 -fa on` on a Q4_0 GGUF requantized from the same MLX checkpoint (dequantized to f16, then `llama-quantize`). Teacher-forced on 256 greedy tokens after a 21-token prompt against an exactly dequantized f32 CPU reference with f64 accumulation, on an idle RTX 3090: arg-max equal at 257/257 positions. The largest logit error is 0.98 at a largest logit of 55.7, from the f16 caches, and grows with context.

QWEN3-8B, 3090	SHORT CTX	1K CTX	4K CTX
speed of light (936 GB/s)	220 tok/s	—	192 tok/s
llama.cpp, Q4_0 GGUF	149.4	145.3	134.9
tang IR, device step	**148.7**	**141.4**	**124.2**
tang IR, first build (shared GPU)	122	—	104

Every change between the first build and the current one is a schedule or a lowering rule, not a kernel; the appendix lists them.

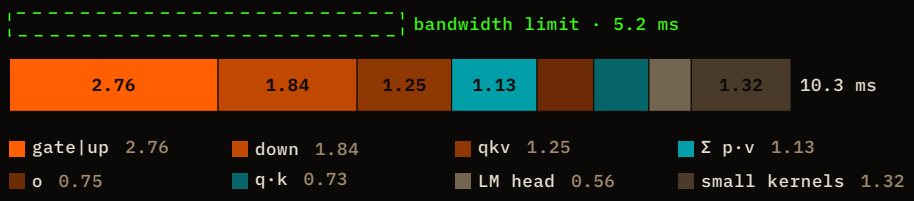


FIG 6 Where a Qwen3-8B decode step went at 4k context in the first build: the sum of isolated kernel launches (10.3 ms; the captured graph ran in 9.6). The current build takes 8.05 ms. Dashed: the time to stream the weights and KV cache once at 936 GB/s.

The same graph builder lowers to Metal. There, prefill and decode programs share one copy of the weights through buffer binding, prefill can start from a nonzero offset (300 + 200 tokens is bit-identical to 500), and the next token is chosen on the device by an arg-max monoid, so decoding reads back 8 bytes per token.

7.5 Data-dependent addressing

A top-k MoE FFN builds as one graph: router GEMM, top-k, softmax weights, SwiGLU experts reading $W[\mathrm{sel}[t,k]]$ through gather views, then the weighted combine. Top-k is exact, and outputs match the f64 interpreter to 4×10^{-6} on Metal and 4×10^{-5} on CUDA (the router runs as a TF32 tile). Table 8 has the timings.

TABLE 8 Top-k MoE at T = 8 tokens, D = 2048, E = 128 experts, k = 10, F = 512 (1 GB of weight reads). The data-dependent address costs nothing against the same expert math over weights gathered into contiguous buffers beforehand. The gathered graph is even faster; we have not isolated why, and do not claim the gap. 378 GB/s is 40% of the 3090's peak: per-point GEMV reads are uncoalesced. §8 adds the narrow GEMV schedule for gathered contractions.

EXPERT GEMMS + COMBINE	M4 MAX	RTX 3090 (CUDA)
gathered (one graph)	2.17 ms (464 GB/s)	2.66 ms (378 GB/s)
weights pre-gathered on the host, tiled	2.89	3.55
weights pre-gathered, one thread per point	2.91	3.77

7.6 Edge hardware

On the K1's Adreno 740, a derived matmul reaches 322 GFLOP/s at 1024^3, which is 21% of the measured fp32 peak, and a full transformer forward pass plus all seven weight gradients at seq 128 runs in 19.1 ms. Rules tuned for desktop GPUs underfill the Adreno's six compute units. The best schedule there (an 8×8 register tile with 8-deep chunks) is 1.45× better than `schedule_auto`'s choice, which again argues for search over remembered rules. Through `to_c` on one Cortex-X3 core, a batch-1 GRU policy step (6 contractions plus gate math, fused into one kernel) takes 22.5 µs median and a locomotion MLP takes 6.7 µs, both within 6×10^{-7} of f64.

8 Deriving a production engine

Every kernel in the sections above is derived. To show where derivation has to go next, we report the other half of tang: the hand-written serving engine in `tang-llm`, which serves Qwen3-Coder-Next (80B parameters, 3B active, 4.5-bit GGUF in Q4_K and Q5_K) on one NVIDIA H200 with kernels written by hand. Table 9's numbers are not results of the IR; they are the target it has to reach.

TABLE 9 The target: tang-llm on a dedicated H200 (tang-llm commit `fcfbb9d7`). Its MoE layers run a hand-written int8 tensor-core kernel that reads activations in B-fragment order, which cut the B = 64 step from 38.9 to 35.2 ms. Lossless speculation uses a trained MTP head (3.15 tokens per window).

H200, QWEN3-CODER-NEXT 80B-A3B	MEASURED
single-stream decode end to end, no drafting	234 tok/s
with speculative drafting (MTP / suffix)	319–439 tok/s
prefill, 1k → 8k tokens	6,424 → 7,534 tok/s
batched decode step, B = 64	1,814 tok/s
serving, 64 concurrent decode-heavy streams	1,279 tok/s
serving at 64k context, batch 48 (production shape)	1,155 tok/s

The hand engine also shows why the contract matters. Ported to GLM-5.3, the same MoE kernel is correct but up to 2.5× slower than the engine's generic path below 64 tokens per expert. A kernel tuned for one model's expert shapes is a point in the schedule space, and on the next model it is the wrong point. It is one data point, but from a production engine, and it is §1's claim.

Deriving this engine needed constructs the IR lacked: a gather view for routing tokens to experts, integer tensor-core fragments with fused dequantization, a top-k reduction, scans, and Hopper's asynchronous copies and warpgroup MMA as schedule fields. Each now exists, and each can replace hand kernels in the engine behind a switch, measured against them on the same bytes.

8.1 The MoE layer

One full MoE layer of the engine builds as a single graph: router, top-k, expert-sorted gather, the gate|up and down expert GEMMs on GGUF Q4_K/Q5_K weights, SwiGLU and the weighted combine. The int8 path uses schedule fields only: GGUF blocks as an input storage format, int8 activations quantized by their writer, and an `IntMma`

field that stages activations in fragment order for `m16n8k32` tensor cores. There is no new operator.

Routing is one reduction. `Monoid::topk(k)` keeps the k best (index, value) pairs, sorted, with ties to the smaller index. Its combine is a merge of two sorted lists ($k^2/2$ compare-selects), which is associative, so split and subgroup trees give the same answer. Top-10 over 512 experts is one 0.013 ms kernel in place of k arg-max rounds, and a whole layer at batch 1 drops from 32 kernels and 0.199 ms to 13 kernels and 0.103 ms.

Batch 1 needs a different schedule. `Stage::Gemv` computes one dot per (position, row) on `dp4a`, with 16-byte loads of half super-blocks and partial sums combined by shuffles. Its producers fuse: the $x \to$ Q8 and SwiGLU $\to$ Q8 nodes are marked inline, and the GEMV computes and quantizes its operand row into shared memory in its prologue, rounding exactly as the writer did.

TABLE 10 One Qwen3-Coder-Next MoE layer (hidden 2048, 512 experts, top-10, FFN 512), derived against the engine's own kernels on the same bytes, on an idle H200. The engine's combine also adds the residual, runs the next RMSNorm and quantizes the activations to int8; "equal work" adds those three steps, timed as derived ops (0.016–0.022 ms), to the derived half. Medians of three rounds of L2-flushed medians. Outputs agree to 3×10^{-7} and top-k is exact. Batch 1 and 4 use the fused GEMV stage, 8 and 64 the int8 fragment stage; the fragment stage at batch 1 takes 0.050 ms. Weights are random valid k-quant blocks.

ONE LAYER, H200, MS	DERIVED EXPERT HALF (STAGE)	DERIVED, EQUAL WORK	ENGINE, HAND-WRITTEN	DERIVED / HAND
batch 1	0.0356 (GEMV)	0.0518	**0.0439**	1.18
batch 4	0.0763 (GEMV)	0.0927	**0.0868**	1.07
batch 8	0.1062 (fragment)	0.1230	0.1248	0.99
batch 64	0.2957 (fragment)	**0.3179**	0.3357	0.95

With equal work the derived layer loses at batch 1 and 4, by 18% and 7%, ties at 8 and wins by 5% at 64. The derived expert half alone is faster than the hand layer at batch 1, but only because the hand layer's time includes the residual, norm and quantization that ours leaves out. Fragment-order staging alone is worth about 20% on gate|up at batch 64. The hand MoE kernels took a dedicated tuning effort; the derived ones are a schedule search, and on equal terms they are not yet faster at small batch.

The backward of a gathered contraction was dense: every expert's weight gradient summed over every pair under a $[\text{key} = e]$ gate, E times the needed work. Declaring the key index sorted (`set_sorted`) tags that reduction with a segment, and the point loop binary-searches each key's run and sums only over it. The gate stays in the body, so a lowering that ignores the tag is still correct. The expert-weight gradient (E = 512) drops from 8.27 to 3.89 ms on the H200, 16.3 to 2.67 ms on the 3090 and 39.7 to 2.65 ms on the M4 Max, with identical error.

8.2 Attention and scans in prefill

The engine's model alternates gated-delta (GDN) layers with full attention at head dimension 256, and prefill is where both cost. At $d = 256$ the old schedule left the online-softmax reduction untiled, and the score stayed an $n \times n$ buffer at 2.8 TFLOP/s. Allowing the tile up to 256 lanes lets the score inline, and the Hopper flash shape takes over with no change to the graph or the emitters (Table 11).

TABLE 11 TFLOP/s, bf16, 16k tokens per configuration, against PyTorch SDPA on the same idle H200; rel. error 2–3 $\times 10^{-3}$ against f64. FlashAttention-3 is faster than both (Table 12).

$d = 256$ FORWARD, H200	n = 1K	4K	16K
tang, non-causal	**318**	**398**	**422**
SDPA, non-causal	306	323	318
tang, causal	220	**310**	**391**
SDPA, causal	**251**	285	295

SDPA is not the strongest baseline at this head dimension. Against FlashAttention-3 built for head dimension 256, on an idle H200 in a later run, tang's forward runs at 0.43–0.54 of FA3's speed and its backward at 0.22–0.40 (Table 12). In that run tang's forward beats SDPA from $n = 4096$ up and loses at $n = 1024$ causal, and its backward loses to SDPA everywhere (SDPA's backward is not in the table).

TABLE 12 Attention at $d = 256$, bf16, ms, idle H200: tang against FlashAttention-3 (built from `flash-attention/hopper` with head dim 256 in bf16 only) and SDPA. Medians of three rounds, spread under 2%; every library lands at relative error 2–5 $\times 10^{-3}$ against f64 on two heads. tang's backward is its forward-plus-backward program minus its forward.

N	CAUSAL	TANG FWD	FA3 FWD	SDPA FWD	TANG BWD	FA3 BWD
1024	no	0.433	**0.199**	0.450	3.624	**0.914**
1024	yes	0.315	**0.137**	0.274	2.666	**0.574**
4096	no	1.379	**0.703**	1.712	8.748	**3.079**
4096	yes	0.885	**0.377**	0.963	5.075	**1.620**
16384	no	5.177	**2.783**	6.785	29.30	**11.79**

Gated delta rules, Mamba-2's SSD and linear attention are associative scans over an affine monoid on a matrix state. A `delta_scan` comprehension lowers by a chunked plan: a WY transform per chunk, a sequential state scan, and a parallel readout. Its backward is the derived reverse scan at chunk granularity. On the 3090 it matches fla's Triton kernels within 6% forward and beats mamba_ssm's SSD forward 1.3–1.8×. Against the engine's hand recurrence (a warp per state column, token by token), the chunked scan is 4.0–4.3× faster on the H200 and 2.1× on the 3090.

Both went into the engine behind switches, gated on the engine's own accuracy checks: KL divergence against the reference at 3k and 8k tokens with int8 and bf16 KV caches, equal greedy output, bitwise-identical batched decode, and a cached-prefix continuation equal to a cold one.

TABLE 13 Time to first token with the int8 KV cache, warm. The baseline is this engine’s prior code, not another serving stack: its attention ran about 5× below the derived kernel (roughly 70 TFLOP/s), and its GDN was a warp per state column, token by token. With the bf16 KV cache the derived attention takes 44k prefill from 6.67 to 5.79 s.

QWEN3-CODER-NEXT PREFILL, H200	TTFT, 44K TOKENS	8K TOKENS
engine, hand kernels	8.33 s	7,300 tok/s
+ derived GDN scan	7.70 s	8,200 tok/s
+ derived $d = 256$ attention	6.17 s	—
+ both	**5.54 s**	—

These gains are over the engine’s own prior code, and that baseline is far from the state of the art. vLLM 0.33 on the official FP8 checkpoint, on an idle H200 with DeepGEMM off, reaches first token in 0.17 s at 8k tokens and 1.08 s at 44k. The engine, on its 4.5-bit GGUF, took 2.1 s and 13.8 s in the same session, and 12.9 s with derived attention: vLLM is about 12× faster. The weight formats differ, so the comparison is not matched for precision, but a gap of an order of magnitude does not come from the format. The engine’s 13.8 s in that run is also slower than the 8.33 s in Table 13, which came from a warm server; we have not reconciled the two measurement paths, so Table 13 should be read as relative.

8.3 Attention backward on Hopper

Training needs the backward. The derived attention backward was nine kernels that round-tripped P and dS through memory, at 13–25 TFLOP/s. A lowering recognizes the pattern the gradient pass produces, a tuple reduction over queries whose operands are pointwise maps of two short contractions, and emits one kernel per key block. It is a shape-keyed emitter. dK and dV stay in registers, and dQ is added with Hopper’s bulk asynchronous reduce. Variants on `mma.sync`, on `wgmma` with two consumer warpgroups, and with a producer warpgroup are all derived from the same match, and the autotuner picks per shape. The fused kernel alone reaches 0.81–0.96× of SDPA’s backward kernel at $d = 128$, and the whole program is 4–8× faster than the nine kernels. The whole backward program is slower than SDPA’s: 0.43–0.74× at $d = 256$ in the run behind Table 12, which does not tabulate SDPA’s backward. The fused kernel also computes in bf16 tensor cores even for an f32 graph, which is fine for bf16 training but fails an f32 tolerance (§1).

8.4 The derived MoE in the model

A layer that wins in isolation still has to win inside the model. Behind a switch, the derived MoE replaces the engine’s expert kernels in every layer of Qwen3-Coder-Next. It passes the engine’s accuracy checks at 3k and 8k tokens and is bitwise-equal batched at 8 streams; the 48-stream agreement gate has a 99% bar that the derived path misses at 98.93%. The hand engine fails it too in the same run, at exactly 99.00% against a strict inequality. Getting there exposed one numerical choice: a block-split sum of squares in RMSNorm, split 256 ways, pushed the KL divergence at 3k tokens to 2.8×10^{-4}, over the 2×10^{-4} bar. Split 1024 ways it is 2.3×10^{-5}, lower than the serial sum’s 4.1×10^{-5}. Precision is a schedule here too.

At 1–8 streams the derived path was slower, so the engine takes it only for windows of at least 16 rows and keeps the hand kernels below that. Above that cutoff it

is faster. Over five interleaved rounds on one idle H200, with hand and derived alternating in fresh processes, a decode step takes 13.99 ms against 15.84 ms at 16 streams, 21.36 against 23.44 at 32 and 28.43 against 30.00 at 48 (medians). The derived step takes 5–12% less time, and the largest spread across rounds is 0.47 ms, so every round shows it. Earlier single runs disagreed because they compared different pods; hand at 48 streams has read 29.67 to 30.26 ms across four H200s. With equal work the layer alone is slower at batch 1 and 4, which matches where the cutoff sits.

The whole decode step can be derived as well. On an idle RTX PRO 6000, single stream, Qwen3-8B’s derived step takes 4.72 ms at context 128 and 5.12 ms at 4k, against 4.82 and 5.19 ms for the hand engine, and produces the same 64 greedy tokens. Qwen3-Coder-Next’s derived step, 1,924 kernels, also matches the hand engine’s 64 greedy tokens but is slower: 5.89 ms against 5.21 at context 128 (13%) and 6.93 against 5.15 at 4k (35%). The gap grows with context; we have not yet broken it down by kernel.

9 The speed of light

“Derived kernels beat hand kernels” is a claim about two programs. It says nothing about whether either is fast. The question that matters is how far a kernel is from the fastest any implementation could be. tang can answer it from the same comprehension it compiles.

9.1 The floor and the tile are dual

For a comprehension over affine views, the least memory traffic of any tiled schedule comes from a linear program. Each view projects the iteration space onto one operand, and the Brascamp–Lieb inequality bounds how many iterations a tile can run for the data it holds[8]. The program’s optimum is the traffic lower bound. Its dual is a tile shape that attains it. So the bound and the schedule are two solutions of one problem, and the duality gap is how far a schedule is from optimal. tang builds this program from a kernel’s views (`TensorGraph::kernel_lp`) and solves both sides.

It reproduces the classical bounds. Matmul gets exponent $3/2$ with a $\sqrt{M/3}$ cube tile. Attention gets Saha and Ye’s $\Theta\left(\min\left(N^2d^2/M, N^2d/\sqrt{M}\right)\right)$[9], and its dual tile is FlashAttention’s $M/(4d)$ blocks. We compared the dual with the tiles tang picks (Table 14). Two different things pick tiles: `schedule_auto` is a fixed rule that times nothing (in this comparison it gives 64 × 64 for matmul and 16 × 16 for attention; its tile depends on the kernel’s shape and the backend), and the autotuner is a timed search. The comparison does not show that the model predicts the autotuner.

TABLE 14 The tile the traffic program chooses against the tile tang picks. On the M4 Max the two-level program matches the autotuner only for register capacities from 4096 to 8192 words, which is the edge of the schedule space; with the real register file it chooses tiles the space cannot express, and it charges nothing for the occupancy a large register tile costs. On the RTX 3090 the timed search keeps the smallest attention tile, but dozens of attention candidates there time within 3% of each other, so that pick does not test the model.

KERNEL	MODEL TILE	WHAT TANG PICKS
matmul 4096^3**, M4 Max, two levels**	64 × 64 at 4096 register words; 256 × 128 at the register file	autotuner 64 × 64
attention $n = 4096$**, M4 Max, two levels**	64 rows × 32 keys at 4096 words; 512 × 32 at the register file	autotuner 64 × 32
attention $n = 1024$**/4096, RTX 3090**	one level 48 × 48; two levels 128 × 32	autotuner 16 × 16, the smallest tile in its space, moving 2.8–3.0× the one-level dual's traffic
matmul 1024^3**, RTX 3090**	—	autotuner 64 × 64, same as the rule

We also timed the model's tiles directly, in a race against the searched ones. On the M4 Max, matmul time stays flat as tile traffic changes, and the one-level dual tile for attention is 11–23% slower than the winner. On the RTX 3090 dozens of attention candidates, the dual tiles included, time within 3% of each other, so the race cannot rank tiles there. In neither case does the traffic-optimal tile win on time.

We then checked two time figures against the same race. The first is the time floor of the next section. It does not depend on how a candidate stages its data, so one value holds for every tile of a kernel. No candidate beat it in any of the 892 timings, which cover matmul at 1024^3 and 4096^3 and attention at $n = 1024$ and 4096 on both machines. On the RTX 3090 the fastest candidate runs at 1.38–2.34× the floor. On the M4 Max matmul runs at about 4× the floor and attention at 53–56×, so no term of the floor limits attention there.

The second figure charges each candidate the traffic of its own tile, at a cache bandwidth we probed separately. It is an estimate and not a floor. On the RTX 3090 most candidates run faster than it, 99 of 102 for matmul at 4096^3, because we compute the traffic term for the WGSL tile and the CUDA kernel stages data differently. It is also not a prediction. We added the cache bandwidth, and a rule that an unstaged kernel reuses data per point, after a first race in which the pick lost. The candidate with the lowest estimate is within 2.5% of the fastest on seven of the eight kernel and machine pairs. On the RTX 3090 that says little, because dozens of candidates tie within 3%. The eighth pair is attention at $n = 1024$ on the M4 Max, where the pick is 1.27× slower than the winner. Its 4 × 4 register tiles give each thread a long serial online-softmax chain, and its grid has only 128 workgroups. The twelve fastest candidates use 512 workgroups and smaller register tiles. The estimate has no term for chain length. At $n = 4096$ the same tile shape has 512 workgroups and wins. The validation in the next section uses GEMV-class and epilogue kernels, not these tiled ones.

Weak duality is mechanized in Lean, over the reals, for a model where each array is indexed by a subset of the loop coordinates: fitting tile exponents sum to at most covering exponents, and a box tiling of V points needs at least V/M^σ tiles. Strides and multi-term addresses are outside it, and so are general schedules, which need the full discrete Hölder–Brascamp–Lieb (HBL) theorem. Guarded views fit the program as it is. For a gather view the bound is $\max(\text{HBL}(\text{rest}), \text{compulsory} + U \cdot \text{slice})$, which is sound for the given index data and equals the analytic MoE floor under uniform top-k. Multi-term views, such as a convolution’s $x + k$, are dropped from the program, which keeps the bound sound but loose. Masks are not modelled.

9.2 From traffic to time

Traffic is one limit. A kernel’s time floor is

$$t \geq L + \max(\lambda + B/\beta, F/\pi, \lceil G/S \rceil \lambda)$$

for launch cost L, DRAM latency λ, B bytes at bandwidth β, F flops at peak π, and G workgroups with at most S resident at once. The first term is the fill: one DRAM latency, then streaming at full bandwidth. The last is a makespan bound: if every workgroup makes at least one DRAM round trip, each holds a slot for at least λ, so S slots need at least $\lceil G/S \rceil$ rounds. Adding L assumes launch overhead does not overlap memory traffic. A decode step’s floor is the longest path through its kernel graph, where independent kernels may overlap.

The constants are measured independently of the kernels they predict. On the H200, an empty kernel in a CUDA graph costs 0.48 µs and a one-thread pointer chase over 4 GB costs 340 ns, on an idle machine. On the 3090, L is an empty kernel in a CUDA graph (0.77 µs) and λ is an assumed 470 ns. Two terms we tried are not in the floor. A Little’s-law bytes-in-flight term never bound with the bytes per thread counted from the schedule. An instruction-issue term was a hypothesis: we chose which requests to count after the full count overshot, and measured instruction counts show plain issue cannot be the limit.

We measured DRAM bytes with Nsight on the RTX 3090 and time on both chips for every kernel of at least 0.5 MB in the MoE layer at 1, 8 and 48 streams and the Q8 GEMVs: 24 GEMV-class and epilogue kernels. No kernel runs faster than its floor on either chip. On the 3090 the large kernels sit at 0.76–0.93 of the measured time (median over all, 0.63). On the H200 the floor is loose everywhere, 0.05–0.66 with a median of 0.38, and we have no validated term for the gap. The H200 pods expose no byte counters, so B is the 3090′s count, and for a kernel that fits the 50 MB L2, B/β is not a proven floor on the H200. Attention and matmul tiles are compute-shaped and outside this validation: on the 3090 their traffic floors are 2–18% of their time, and against the TF32 peak the matmul tile reaches 0.47–0.62 and flash attention 0.25–0.28.

9.3 What the floor says about decode

On the H200 a whole decode step’s traffic floor is only 0.16–0.29 of its measured time, and the gaps between kernels inside the CUDA graph are only 0–8% of the step. The missing time is per kernel. A cold dependent GEMV costs 6.9 µs including launch, and any dependent kernel under about 2 MB costs about that much whatever it does.

At one stream the hand engine’s step, with the shared expert on a parallel branch, is 548 kernels in 4.04 ms, 7.4 µs each. That arithmetic suggests decode at small batch is bound by its chain of dependent kernels, but the obvious test refutes the simple form of it. Fusion did not speed up the step, because the work moved into tails on the same critical path (Table 15). What limits small-batch decode on the H200 is still an open term in the floor.

We used earlier versions of the floor, with superseded constants, to bound decode levers before building them (Table 15). Those bounds missed in both directions, so they are upper bounds at best and not predictions.

TABLE 15 Decode levers, bounded by earlier versions of the step floor and then measured. Gains are against the step time; the parallel shared expert is a switch in the engine.

LEVER	BOUND FROM THE FLOOR	MEASURED, H200
blind L2 prefetch of the next matrix	−36%, then +4 to +28% (slower) once the prefetcher was charged	net slower: next kernel −0.4 to −0.6 µs, prefetcher +0.6 to +4 µs
shared expert on a parallel graph branch	≤ 9% at one stream	within noise at 1–8 streams; −6 to −13% at 16–48, bitwise equal
fuse block-local chains	−25% hand, −38% derived at one stream; 4–8% at 48	no gain: with combine fused the derived step went from 4.05 to 4.44 ms, 96 of 548 launches removed

What the floor supports is narrower than its name. It is a lower bound that holds on every kernel we measured. On the RTX 3090 it is close for bandwidth-bound kernels. On the H200 it is loose, it says nothing yet about compute-shaped kernels, and as a guide to which optimizations pay it has been wrong as often as right.

10 Related work

Compilers and IRs. MLIR’s Linalg dialect[10] (a generic op with indexing maps and a body) and XLA[11] are closest in the IR’s shape. tinygrad[12] shares the goal of few constructs and few hand-written kernels across backends. Futhark[13], Lift[14] and Dex[15] are array languages with rewrites or differentiation. TensorIR[16] brings tensorized loops to Halide-style search. Suriana et al.[17] synthesize associative reductions in Halide, which is the closest precedent for recognizing a monoid. TASO[18] and PET[19] superoptimize tensor graphs.

Schedules. Halide[20] separated algorithm from schedule. TVM[21] and Ansor[22] searched the schedule space with learned cost models, and Tensor Comprehensions[23] used polyhedral compilation. tang inherits the separation and the search.

Deriving fused kernels. The closest work is recent. Mirage[24] superoptimizes multi-level μGraphs with probabilistic equivalence checks. EqiForge[25] uses equality saturation to derive FlashAttention-style forward kernels. Flashlight[26] generates flash-style kernels for arbitrary attention variants inside TorchInductor. tang differs on three axes. Its rewrite is deterministic, keyed on the monoid structure rather than found by search. It derives the backward pass, including the gradient of the online-

softmax monoid and the elimination of the stabilizing max. And it lowers to six targets, not one. AttentionEngine[27] and FlashInfer[28] generate attention variants from templates. Neither Flashlight nor EqiForge publishes code, so we measure against FlexAttention and FA2 instead (§7.3).

Tile languages and hand kernels. Triton[29], ThunderKittens[30] and TileLang[31] make kernels easier to write. FlashAttention[1,2,32] is the hand-derived frontier. Online softmax is due to Milakov and Gimelshein[3]. Baselines we measure against are fla[33], mamba_ssm[34], llama.cpp[35] and vLLM[36]. ScanWeaver[37] compiles affine recurrences by associative scan lowering, and Event Tensor[38] compiles dynamic megakernels. tang aims to make writing kernels unnecessary, and treats these kernels as the targets derived code must reach.

Lower bounds. Communication lower bounds for loop nests over arrays come from the Brascamp–Lieb inequality[8], following Hong and Kung’s red-blue pebble game[39]. Pebbling bounds are now derived automatically for affine programs[40–42], and COSMA[43] attains tight constants for matmul. Attention’s I/O complexity is $\Theta(N^2 d^2 / M)$ for large caches[9]. These analyze a fixed algorithm and model traffic only. tang reads the program off the kernel it compiles, uses its dual as a schedule, adds latency and launch terms, and checks the result against hardware counters. The time floor is a roofline[44] with latency, launch and occupancy terms. SOL-ExecBench[45] scores kernels against per-problem speed-of-light bounds, but as a benchmark rather than from a compiler’s IR.

Generated kernels. LLM agents now write CUDA, but KernelBench-Verified[46] shows frontier models reward-hacking benchmarks by hard-coding test values. A derived kernel is checked by construction against an f64 interpreter of the same graph.

11 What derivation covers and what remains

Deriving a production engine needed the IR constructs in Table 16. Each is built and measured above, with the limits stated here.

TABLE 16 The constructs a production engine needed, and what each one bought. None of these is an attention or MoE kernel. Each is a monoid, a schedule field or a lowering rule that any graph of the same shape gets. The table lists gains only; the next paragraph lists what each still loses.

GAP	CONSTRUCT	RESULT
MoE routing	`Monoid::topk`	10 arg-max rounds → 1 kernel; layer 0.199 → 0.103 ms
batch-1 gathers	`Stage::Gemv`, fused producers	expert half 0.036 ms at batch 1; 18% behind hand on equal work (Table 10)
gather backward	sorted-key segments	2.1–15× on the expert gradient
recurrences	scan monoid, chunked plan	4× the hand recurrence; ties fla on the 3090
$d = 256$ **attention**	tile width in `schedule_auto`	2.8 → 422 TFLOP/s; 44k TTFT 8.33 → 6.17 s
Hopper backward	recognized fused kernel, `wgmma`, warp specialization	9 kernels → 1; 0.8–1.0× SDPA
launch fusion	cooperative staged maps, megakernels	729 → 584 kernels; time-neutral on sm_86
decode	split GEMVs, load-bytes register model	122 → 148.7 tok/s; ties llama.cpp at short context

What remains is also measured. The engine still uses hand kernels below 16 window rows. The derived Hopper forward runs at 0.42–0.60× of FlashAttention-3 at $d = 128$ and 0.43–0.54× at $d = 256$, and the backward at 0.22–0.40× of FA3′s at $d = 256$: FA3 overlaps the softmax of one block with the GEMMs of the next, and loads through TMA from a producer warpgroup, and our consumers still issue their own copies. On the H200, fla’s scans are 1.2–2.4× ahead at long sequences, because our state scan runs sequentially over chunks; the affine monoid is associative at any granularity, so a two-level scan is a schedule change. The WGSL flash tile still runs every key tile under a sliding-window mask (Table 6), and the Metal backward’s nested dots are not yet blocked onto simdgroup matrices. `schedule_auto` is still rules, while every autotuning run writes (graph, chip, schedule, time) to a cache that can train a cost model. Sharded reductions joined by an all-reduce are already a rewrite, and only the transport is missing. The fused attention backward needs an f32 path, and one wrong placement that the check catches is not yet explained. Prefill is about 12× behind vLLM. Several earlier tables were taken on shared GPUs; the idle-GPU measurements in §1, §7 and §8 were not.

12 Conclusion

Two constructs, affine views and monoid comprehensions, express a transformer, its gradient and its quantized inference. Lowering them to fast code still takes shape-keyed emitters for the hottest schedules. Fusion, flash attention and the flash backward become rewrites and schedule choices, not kernels. Because the schedule is searched per chip, the same source wins or stays close to hand kernels on Metal and Vulkan, runs on an embedded Adreno, and on CUDA beats PyTorch’s best mode on a transformer block’s forward pass at sequence length 512 while losing at 128.

The hand-written kernel does not disappear in this design. It becomes an intrinsic, a fast implementation of one comprehension shape on one chip, which the compiler uses when it wins and replaces when it doesn’t. In the last hand-tuned engine we run, an 80B MoE model on an H200, the derived MoE layer is slower than the hand-written one at batch 1 and 4 on equal work and faster at 64, a derived decode step takes 5–12% less time from 16 streams up across five rounds, and derived prefill kernels cut the engine’s time to first token at 44k tokens by a third, though vLLM remains about 12× faster.

The comparison with hand kernels was never the point. The traffic floor and the tile that reaches it are dual solutions of one program, so a compiler that derives kernels can also say how far each one is from a lower bound on its time. So far that floor holds on every kernel we measured and is close on the large kernels of the RTX 3090. On the H200 it is loose and we cannot yet say why, and the model behind it does not predict the tiles the autotuner picks. Small-batch decode on the H200 is the open case: it is far from its floor, and removing launches did not close the distance.

> ASIDE · REPRODUCIBILITY
>
> Numbers come from `BENCHMARKS.md` in the tang repository, which records the command, date and machine state for each measurement; the idle-GPU measurements are in its section “Idle-GPU measurements” with raw outputs under `crates/tang-llm/docs/bench/review/`, and the floor table is in `docs/duality.md`. The `data/` directory next to this paper holds a snapshot of these files at tag `paper-2026-11-10b` (commit `15a66d95`). The whole-step decode results in §8 come from later runs on tang main; their outputs are in `data/whole-step/`. The IR is in `crates/tang-compile/src/tensor*.rs`; its design notes are in `docs/tensor-ir.md`.

Appendix: decode engineering notes

Every change between the first build and the current one is a schedule or a lowering rule, not a kernel. Norms are materialized once instead of recomputed in every GEMV lane. 4-bit GEMVs under 16k rows are split along k. The attention score kernel stops at the runtime length. A load-bytes model picks the register dimension, so P·V shares the probabilities across the 4 GQA heads instead of re-reading them. A cooperative lowering computes a small reduction (RMSNorm’s sum of squares, the per-head q/k-norm statistics) into shared memory once per block and evaluates the map that reads it from there, which takes the step from 729 to 584 kernels. The first build had 506.

On the 3090, as on the H200 (§9.3), removing launches saved nothing. Inside a CUDA graph on sm_86, a dependent launch costs what a grid barrier costs (0.77 vs 0.81 µs), so 145 fewer kernels is time-neutral, and a persistent megakernel that runs the whole step as one launch with 437 phases is correct but 19% slower, because one register budget for every stage spills the largest GEMV. The gap to llama.cpp at 4k is in the kernels: the 4-bit GEMVs run at 60–82% of bandwidth, and attention reads K and V about twice.

References

1. Dao, T., Fu, D. Y., Ermon, S., Rudra, A. & Ré, C. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. in Advances in Neural Information Processing Systems (2022).
2. Shah, J. et al. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. arXiv preprint arXiv:2407.08608 (2024).
3. Milakov, M. & Gimelshein, N. Online normalizer calculation for softmax. arXiv preprint arXiv:1805.02867 (2018).
4. Welford, B. P. Note on a method for calculating corrected sums of squares and products. Technometrics **4,** 419–420 (1962).
5. Dao, T. & Gu, A. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. in International Conference on Machine Learning (ICML) (2024).
6. Yang, S., Kautz, J. & Hatamizadeh, A. Gated Delta Networks: Improving Mamba2 with delta rule. in International Conference on Learning Representations (ICLR) (2025).
7. Dong, J., Feng, B., Guessous, D., Liang, Y. & He, H. Flex Attention: A Programming Model for Generating Optimized Attention Kernels. in MLSys (2025).
8. Christ, M., Demmel, J., Knight, N., Scanlon, T. & Yelick, K. Communication Lower Bounds and Optimal Algorithms for Programs That Reference Arrays – Part 1. arXiv preprint arXiv:1308.0068 (2013).
9. Saha, B. & Ye, C. The I/O Complexity of Attention, or How Optimal is Flash Attention? arXiv preprint arXiv:2402.07443 (2024).
10. Lattner, C., Amini, M., Bondhugula, U. & others. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. in CGO (2021).
11. Sabne, A. XLA: Compiling Machine Learning for Peak Performance. (2020).
12. tiny corp. tinygrad. https://github.com/tinygrad/tinygrad.
13. Henriksen, T., Serup, N. G. W., Elsman, M., Henglein, F. & Oancea, C. E. Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates. in PLDI (2017).
14. Steuwer, M., Fensch, C., Lindley, S. & Dubach, C. Generating Performance Portable Code Using Rewrite Rules. in ICFP (2015).
15. Paszke, A., Johnson, D. D., Duvenaud, D. & others. Getting to the Point: Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming. in ICFP (2021).
16. Feng, S., Hou, B., Jin, H. & others. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. in ASPLOS (2023).
17. Suriana, P., Adams, A. & Kamil, S. Parallel Associative Reductions in Halide. in CGO (2017).

18. Jia, Z. et al. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. in SOSP (2019).
19. Wang, H., Zhai, J., Gao, M. & others. PET: Optimizing Tensor Programs with Partial Equivalence Transformations and Automated Canonicalization. in OSDI (2021).
20. Ragan-Kelley, J. et al. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. in PLDI (2013).
21. Chen, T., Moreau, T., Jiang, Z. & others. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. in OSDI (2018).
22. Zheng, L., Jia, C., Sun, M. & others. Ansor: Generating High-Performance Tensor Programs for Deep Learning. in OSDI (2020).
23. Vasilache, N., Zinenko, O., Theodoridis, T. & others. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. arXiv preprint arXiv:1802.04730 (2018).
24. Wu, M., Cheng, X., Liu, S. & others. Mirage: A Multi-Level Superoptimizer for Tensor Programs. in OSDI (2025).
25. Zhan, Q., Hu, X., Xia, X. & Li, S. Unleashing the Power of Equality Saturation for Tensor Program Superoptimization. arXiv preprint arXiv:2609.12330 (2026).
26. You, B., Wang, I., Mustafaoglu, Z. S., Jangda, A. & others. Flashlight: PyTorch Compiler Extensions to Accelerate Attention Variants. arXiv preprint arXiv:2511.02043 (2025).
27. Chen, F., Cheng, Y., Wang, L. & others. AttentionEngine: A Versatile Framework for Efficient Attention Mechanisms on Diverse Hardware Platforms. arXiv preprint arXiv:2502.15349 (2025).
28. Ye, Z., Chen, L., Lai, R. & others. FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving. arXiv preprint arXiv:2501.01005 (2025).
29. Tillet, P., Kung, H. T. & Cox, D. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. in MAPL (2019).
30. Spector, B. F., Arora, S., Singhal, A., Fu, D. Y. & Ré, C. ThunderKittens: Simple, Fast, and Adorable AI Kernels. in ICLR (2025).
31. Wang, L. & others. TileLang: A Composable Tiled Programming Model for AI Systems. arXiv preprint arXiv:2504.17577 (2025).
32. Dao, T. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. arXiv preprint arXiv:2307.08691 (2023).
33. flash-linear-attention. https://github.com/fla-org/flash-linear-attention.
34. mamba_ssm. https://github.com/state-spaces/mamba.
35. llama.cpp. https://github.com/ggml-org/llama.cpp.
36. Kwon, W., Li, Z., Zhuang, S. & others. Efficient Memory Management for Large Language Model Serving with PagedAttention. in SOSP (2023).

37. Wu, Q. & Zolnikov, P. ScanWeaver: Compiler-Driven Parallelization of Affine Recurrences via Associative Scan Lowering. arXiv preprint arXiv:2606.00601 (2026).

38. Jin, H., Hou, B., Wang, G., Lai, R. & others. Event Tensor: A Unified Abstraction for Compiling Dynamic Megakernel. arXiv preprint arXiv:2604.13327 (2026).

39. Hong, J.-W. & Kung, H. T. I/O Complexity: The Red-Blue Pebble Game. in STOC (1981).

40. Olivry, A., Langou, J., Pouchet, L.-N., Sadayappan, P. & Rastello, F. Automated Derivation of Parametric Data Movement Lower Bounds for Affine Programs. in PLDI (2020).

41. Olivry, A. et al. IOOpt: Automatic Derivation of I/O Complexity Bounds for Affine Programs. in PLDI (2021).

42. Kwasniewski, G. et al. Pebbles, Graphs, and a Pinch of Combinatorics: Towards Tight I/O Lower Bounds for Statically Analyzable Programs. arXiv preprint arXiv:2105.07203 (2021).

43. Kwasniewski, G. et al. Red-Blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication. in SC (2019).

44. Williams, S., Waterman, A. & Patterson, D. Roofline: An Insightful Visual Performance Model for Multicore Architectures. Communications of the ACM **52,** (2009).

45. Lin, E. et al. SOL-ExecBench: Speed-of-Light Benchmarking for Real-World GPU Kernels Against Hardware Limits. arXiv preprint arXiv:2603.19173 (2026).

46. Zhang, Y., Yu, P., Wang, J. & others. KernelBench-Verified: Do LLM-Generated Kernels Actually Beat PyTorch? arXiv preprint arXiv:2607.16241 (2026).