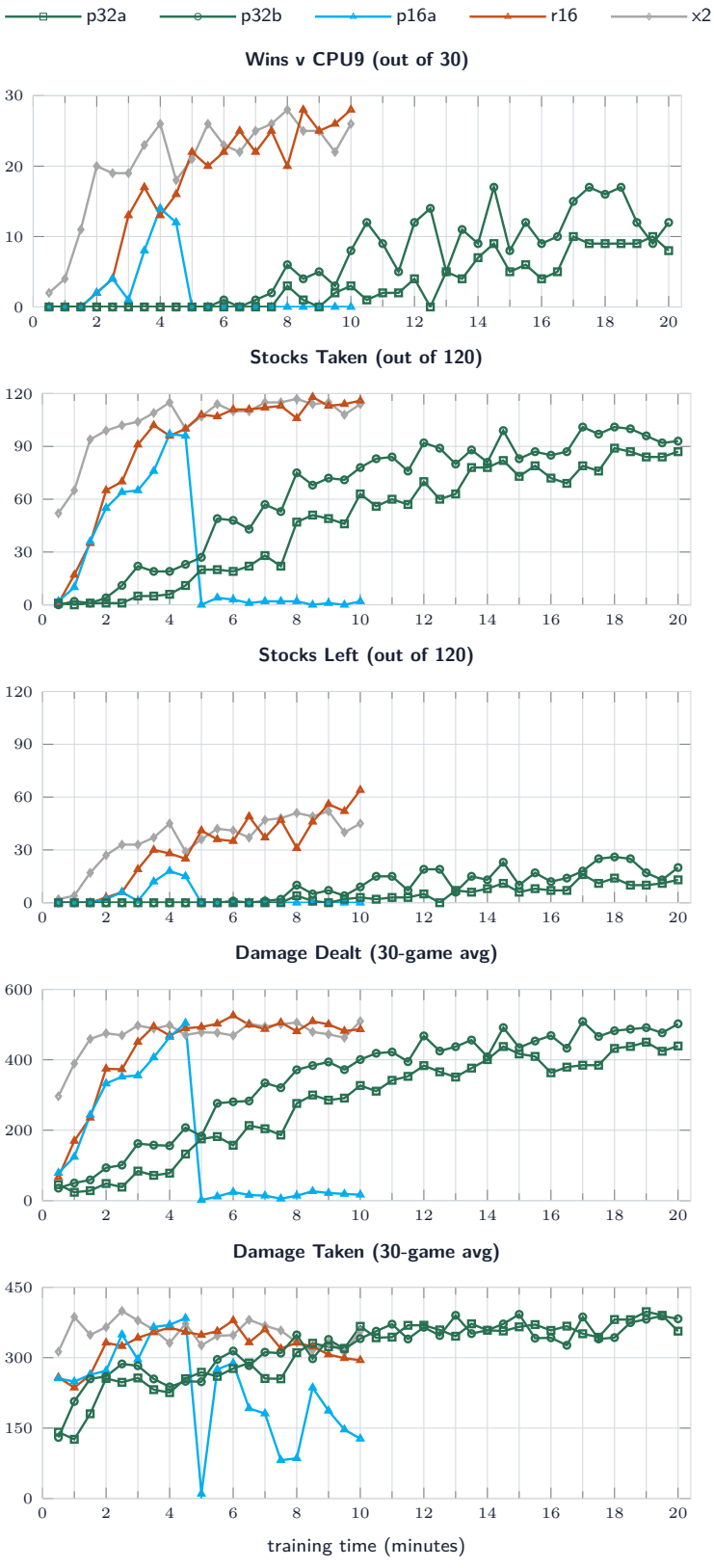
Decoder-Only Transformer Optimization 101



Frisson Labs

TongKe Xue[1], Rohit Swamy[2], Eric Gu[3], Ali Janati, Nikita Kuzmin, Charles Niu, Tod Sophon

At Frisson Labs, we're building AI that can play games with you. This report asks how efficiently we can train them: how fast can a single RTX 6000 Pro train a decoder-only transformer to play Smash Melee—Fox versus a level-9 CPU Fox on Final Destination?

We start with Eric Gu's HAL-20M blogpost (upstream commit `d7a2247`). We train it with two data samplers. **Uniform** draws replay windows without targeting particular events. **Event** targets combat and recovery using our chosen mixture: 30% uniform windows, 35% high-damage windows, and 35% recovery/edgeguard windows. On HAL-20M, `torch.compile` achieves:

— **p32a: 264,454 tokens/s**
FP32; uniform sampler

— **p32b: 265,813 tokens/s**
FP32; event sampler

— **p16a: 918,193 tokens/s**
BF16; event sampler. Some width-512 BF16 runs developed non-finite gradients or diverged; the root cause was not isolated. See the BF16 failures appendix for our measurements and diagnostics.

While using the identical architecture:
Naive cuTile-rs achieves 163,645 tokens/s.
Optimized cuTile-rs achieves 660,540 tokens/s.
CUDA-Oxide reaches:

— **r16: 1,125,109 tokens/s**
CUDA-Oxide v2 BF16; event sampler; 116.50 ms/batch.

By sampling every other frame and reducing model width, we are able to achieve:

— **x2: 3,217,492 tokens/s**
CUDA-Oxide BF16; 6L / D256 / H4, every 2nd frame, 5.3M-param model
20.37 ms/batch; 5.72× r16 windows/s.

In the next few sections, we take a quantitative tour of the optimizations involved in increasing tokens/s.

[1]Technical contact: tkxue@tkxue.org.
[2]Business contact: rohit@frisson-labs.com.
[3]Independent Researcher: https://ericyuegu.com/

Why speedrun

1. **Train faster, iterate faster.** Shorter training runs give us faster feedback on architectures, data and training choices. We can test an idea, inspect the result and try again sooner. More iterations give us more opportunities to find what works.

2. **Train bigger models for the same budget.** Efficiency gains let a fixed compute budget support larger models, more training data or longer training. The compute we save becomes room for a more ambitious experiment.

3. **Carry the techniques into bigger runs.** Kernel fusion, buffer reuse and better scheduling can transfer from small speedruns to larger Transformer training. At that scale, even a modest percentage matters: a 10% reduction in training cost saves $10M on a $100M run.

Why Eric Gu's model?

The Melee AI community includes projects such as Slippi-AI, MIMIC and Eric Gu's HAL. We chose HAL for its clear decoder-only Transformer architecture.

At roughly 20 million parameters, it gives us the core LLM pretraining workload in a small model: causal attention, matrix multiplies, MLPs, backpropagation and optimizer updates. Melee supplies the inputs and prediction targets, while the central computation closely resembles language-model training. This makes HAL a useful place to study optimization principles that also apply to much larger Transformers.

Inspiration

Keller Jordan's NanoGPT speedrun inspired the pursuit of reaching a target model quality as quickly as possible. Andrej Karpathy's autoresearch inspired the rapid cycle of automated experiments: change the training code, run a short experiment, measure the result and keep the improvements.

Two-Stage Speedrun

Our main goal is to train a model that beats CPU9, not merely one that reaches a given error on next-token prediction. A typical training speedrun optimizes time to a target next-token error; for us, that error is only a proxy for the gameplay performance we actually care about.

We therefore split the speedrun into two stages. In the first, `m1`, `m2`, `m3` and `r16` keep the model architecture fixed. Up to floating-point differences, they perform the same computation; the gains come from optimizing how that computation runs on the GPU.

In the second stage, `x1` changes the architecture while holding the required gameplay performance fixed. This broader design space shows that, once the goal is time to a CPU9-capable model rather than time to reproduce one particular predictor, architectural changes can produce even more dramatic speedups.

Overview

Our overall structure is:

- Benchmark `m0`, a naive implementation.
- `m1`: do faster matrix multiplies.
- `m2`: reduce memory usage via fusion.
- `m3`: remove redundant memory work.
- `r16`: switch to CUDA-Oxide for fine-grained control over threads/warps/blocks
- `x2`: switch to a smaller model architecture and sample every other frame

The token throughputs of the various configurations are:

Configuration	Description	tokens/s
`PT eager`	PyTorch eager BF16	401,766
`p16a`	`torch.compile` BF16	918,193
`m0`	Naive: batch 512, basic tiled GEMMs, unfused transformer.	163,645
`m1`	Faster GEMM: tiling & split-K.	207,146
`m2`	Fusion: pointwise, reductions, and FlashAttention.	445,372
`m3`	Remove redundant zero-fills, reuse storage, replay graphs and overlap input preparation.	660,540
`r16`	(CUDA-Oxide) specialized kernels, fused epilogues, concurrent lanes, and relative-position attention fusion.	1,125,109
`x2`	Every other frame and half the width, T128 / D256 / 6L	3,217,492

benchmarking m0

We start by benchmarking a batch of 512 sequences, each containing 256 frames: 131,072 tokens per training step. On the LHS:

- game state flows in, controller output flows out.
- we measure the timing of each block
- we note most of the time is spent in the decoder layers, and expand one layer on the right

Forward overview

	ms
Batch size 512 256-frame causal context categorical game state + continuous features	—
Embedding lookup and input assembly	0.86
Input projection to width 512	1.13
Input dropout	0.88
Decoder layer 1	58.02
Decoder layer 2	57.83
Decoder layer 3	57.45
Decoder layer 4	57.75
Decoder layer 5	58.02
Decoder layer 6	58.17
Final LayerNorm	1.01
Autoregressive controller heads c-stick → main stick → buttons	10.38
Controller logits action sampling excluded from training	—

One forward layer

Layer code v010	Kernel ms profiled excludes fills
01 X_l	—
02 $N_1 = \texttt{LN}(X_l)$	0.76
03 $(Q, K, V) = \texttt{split\_heads}(N_1 W_{qkv}^{\mathsf{T}} + b_{qkv})$	4.77
04 $S = \texttt{mask}((QK^{\mathsf{T}} + \texttt{skew}(QE_r^{\mathsf{T}}))/\sqrt{d_h})$	4.61
05 $P_{i:} = \texttt{softmax}(S_{i:})$	12.26
06 $P_{\text{drop}} = \texttt{dropout}(P)$	1.30
07 $O = P_{\text{drop}} V,\ C = \texttt{join\_heads}(O)$	2.33
08 $A = CW_o^{\mathsf{T}} + b_o$	1.36
09 $A_{\text{drop}} = \texttt{dropout}(A)$	0.33
10 $R_1 = X_l + A_{\text{drop}}$	0.26
11 $N_2 = \texttt{LN}(R_1)$	0.77
12 $U = N_2 W_1^{\mathsf{T}} + b_1$	5.61
13 $H_{\text{mlp}} = \texttt{GELU}(U)$	0.88
14 $M = H_{\text{mlp}} W_2^{\mathsf{T}} + b_2$	4.64
15 $M_{\text{drop}} = \texttt{dropout}(M)$	0.34
16 $X_{l+1} = R_1 + M_{\text{drop}}$	0.27
Kernel sum / layer (profiled; excludes fills)	40.48

Whole block vs. kernel sum

Left-hand timings: unprofiled CUDA-event intervals for each decoder layer, including initialization/fills, device copies and gaps.

Right-hand timings: profiled kernel durations, averaged over all six layers. Initialization/fills, device copies and gaps have no equation row and are excluded.

The RHS's 40.48 ms comes from an earlier capture. The paired block campaign gives:

Per-layer accounting	ms
Layer kernels (profiled)	40.69
Initialization/fill kernels (`full_entry`)	15.74
Copies / gaps / event-boundary remainder	1.47
Whole block (profiled)	57.91
Whole block (unprofiled, separate run)	57.87

Similarly, we benchmark the backward pass. This is the foundation of our analysis. In future sections, we will describe optimizations and compare timings before and after each optimization.

Backward overview

	ms
Controller cross-entropy losses losses + gradients with respect to logits	0.50
Controller-head backward[1] all three heads; one loss per head	74.00
Sum the three feature gradients	1.04
Final LayerNorm backward	1.31
Decoder layer 6 backward	62.41
Decoder layer 5 backward	62.76
Decoder layer 4 backward	62.83
Decoder layer 3 backward	62.47
Decoder layer 2 backward	62.57
Decoder layer 1 backward	62.62
Input dropout backward	0.51
Input projection backward	1.93
Input assembly and embedding backward stage, character, action gradients	3.96

One backward layer

Layer code v010		Kernel ms profiled excludes fills
01	dX_{l+1}	—
02	$dM = \texttt{dropout\_bwd}(dX_{l+1})$	0.26
03	$(dH_{\text{mlp}}, dW_2, db_2) = (dMW_2, dM^{\mathsf{T}} H_{\text{mlp}}, \sum_r dM_r)$	5.72
04	$dU = dH_{\text{mlp}} \odot \texttt{GELU}'(U)$	1.18
05	$(dN_2, dW_1, db_1) = (dUW_1, dU^{\mathsf{T}} N_2, \sum_r dU_r)$	4.83
06	$dR_{1,b} = \texttt{LN\_bwd}(dN_2; R_1)$	1.04
07	$dR_1 = dX_{l+1} + dR_{1,b}$	0.25
08	$dA = \texttt{dropout\_bwd}(dR_1)$	0.26
09	$(dC, dW_o, db_o) = (dAW_o, dA^{\mathsf{T}} C, \sum_r dA_r)$	1.99
10	$(dP_{\text{drop}}, dV) = (dOV^{\mathsf{T}}, P_{\text{drop}}^{\mathsf{T}} dO)$	6.06
11	$dP = \texttt{dropout\_bwd}(dP_{\text{drop}})$	
12	$dS = P \odot (dP - \texttt{rowsum}(dP \odot P))$	10.76
13	$dZ = dS/\sqrt{d_h}, \quad (dQ_{\text{dot}}, dK) = (dZK, dZ^{\mathsf{T}} Q)$	4.10
14	$dR = \texttt{unskew}(dZ), \quad (dQ_{\text{rel}}, dE_r) = (dRE_r, dR^{\mathsf{T}} Q)$	6.80
15	$G_{qkv} = \texttt{concat\_heads}(dQ_{\text{dot}} + dQ_{\text{rel}}, dK, dV)$	1.76
16	$(dN_1, dW_{qkv}, db_{qkv}) = (G_{qkv} W_{qkv}, G_{qkv}^{\mathsf{T}} N_1, \sum_r G_{qkv,r})$	4.07
17	$dX_b = \texttt{LN\_bwd}(dN_1; X_l)$	1.04
18	$dX_l = dR_1 + dX_b$	0.25
Kernel sum / layer (profiled; excludes fills)		50.35

Whole block vs. kernel sum

Left-hand timings: unprofiled CUDA-event intervals for each decoder layer, including initialization/fills, device copies and gaps.

Right-hand timings: profiled kernel durations, averaged over all six layers. Initialization/fills, device copies and gaps have no equation row and are excluded.

The RHS's 50.35 ms comes from an earlier capture. The paired block campaign gives:

Per-layer accounting	ms
Layer kernels (profiled)	50.58
Initialization/fill kernels (`full_entry`)	11.78
Copies / gaps / event-boundary remainder	0.05
Whole block (profiled)	62.41
Whole block (unprofiled, separate run)	62.61

[1] The 74 ms covers all three controller heads; in the paired profiling run, bias-gradient reductions (34 ms) and weight gradients (26 ms) dominate.

m1: faster matrix multiplies

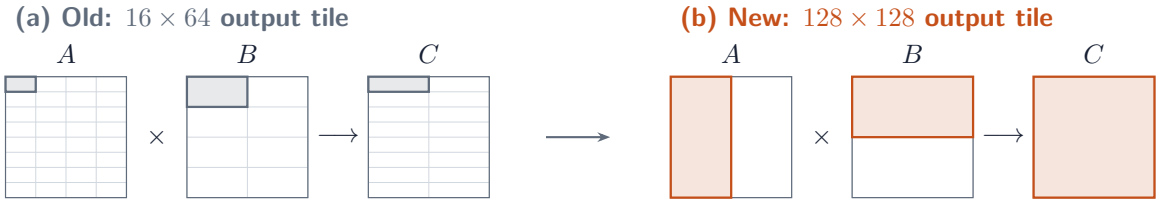
In this section, we will describe two optimizations for faster matrix multiplication that take us from 163,645 tokens/s to 207,146 tokens/s.

The two ideas are:

- tiling: reuse operands across more output elements
- split-K: parallelize long gradient reductions

1. Tiling: reuse operands across more output elements

We want to compute $C = AB$. Both versions use Tensor Cores, NVIDIA's specialized matrix multiply-accumulate units, with BF16 inputs and FP32 accumulation. We enlarge each thread block's output tile from 16×64 to 128×128, so loaded elements of A and B are reused across more output elements.



Each square is a 128×128 window. Shading shows one block's operand tiles and output accumulator: (a) covers part of the output window; (b) fills it.

2. Split-K: parallelize long gradient reductions

We want to compute

$$C = \sum_{i=0}^{n-1} A_i B_i.$$

Split the reduction into k segments of length $m = n/k$ (assuming k divides n). Compute their partial sums in parallel:

$$C_j = \sum_{i=jm}^{(j+1)m-1} A_i B_i, \qquad j = 0, \ldots, k-1.$$

Then add the partial results in a second kernel:

$$C = \sum_{j=0}^{k-1} C_j.$$

For each output tile, this gives the GPU k shorter reductions to run in parallel, which helps when a long gradient reduction has too few output tiles to keep the GPU busy.

Forward timings: tiling and split-K applied

Layer code m0 → m1 (v010 → v040)		Kernel ms (profiled) excludes fills Old	New
01	X_l	—	—
02	$N_1 = \texttt{LN}(X_l)$	0.76	0.77
03	$(Q, K, V) = \texttt{split\_heads}(N_1 W_{qkv}^{\mathsf{T}} + b_{qkv})$	4.77	2.57
04	$S = \texttt{mask}((QK^{\mathsf{T}} + \texttt{skew}(QE_r^{\mathsf{T}}))/\sqrt{d_h})$	4.61	4.60
05	$P_{i:} = \texttt{softmax}(S_{i:})$	12.26	12.30
06	$P_{\text{drop}} = \texttt{dropout}(P)$	1.30	1.29
07	$O = P_{\text{drop}} V,\ C = \texttt{join\_heads}(O)$	2.33	2.35
08	$A = C W_o^{\mathsf{T}} + b_o$	1.36	0.59
09	$A_{\text{drop}} = \texttt{dropout}(A)$	0.33	0.34
10	$R_1 = X_l + A_{\text{drop}}$	0.26	0.26
11	$N_2 = \texttt{LN}(R_1)$	0.77	0.78
12	$U = N_2 W_1^{\mathsf{T}} + b_1$	5.61	2.67
13	$H_{\text{mlp}} = \texttt{GELU}(U)$	0.88	0.87
14	$M = H_{\text{mlp}} W_2^{\mathsf{T}} + b_2$	4.64	1.66
15	$M_{\text{drop}} = \texttt{dropout}(M)$	0.34	0.33
16	$X_{l+1} = R_1 + M_{\text{drop}}$	0.27	0.26
Kernel sum / layer (profiled; excludes fills)		40.48	31.66

Old − new: 8.815 ms/layer. Black box: affected group saving $> 10\%$ of this pass's total saving.

Backward timings: tiling and split-K applied

Layer code m0 → m1 (v010 → v040)		Kernel ms (profiled) excludes fills Old	New
01	dX_{l+1}	—	—
02	$dM = \texttt{dropout\_bwd}(dX_{l+1})$	0.26	0.26
03	$(dH_{\text{mlp}}, dW_2, db_2) = (dMW_2, dM^{\mathsf{T}} H_{\text{mlp}}, \sum_r dM_r)$	5.72	3.48
04	$dU = dH_{\text{mlp}} \odot \texttt{GELU}'(U)$	1.18	1.18
05	$(dN_2, dW_1, db_1) = (dUW_1, dU^{\mathsf{T}} N_2, \sum_r dU_r)$	4.83	3.31
06	$dR_{1,b} = \texttt{LN\_bwd}(dN_2; R_1)$	1.04	1.03
07	$dR_1 = dX_{l+1} + dR_{1,b}$	0.25	0.25
08	$dA = \texttt{dropout\_bwd}(dR_1)$	0.26	0.26
09	$(dC, dW_o, db_o) = (dAW_o, dA^{\mathsf{T}} C, \sum_r dA_r)$	1.99	0.89
10	$(dP_{\text{drop}}, dV) = (dOV^{\mathsf{T}}, P_{\text{drop}}^{\mathsf{T}} dO)$	6.06	6.06
11	$dP = \texttt{dropout\_bwd}(dP_{\text{drop}})$		
12	$dS = P \odot (dP - \texttt{rowsum}(dP \odot P))$	10.76	10.81
13	$dZ = dS/\sqrt{d_h}, \quad (dQ_{\text{dot}}, dK) = (dZK, dZ^{\mathsf{T}} Q)$	4.10	4.10
14	$dR = \texttt{unskew}(dZ), \quad (dQ_{\text{rel}}, dE_r) = (dRE_r, dR^{\mathsf{T}} Q)$	6.80	1.91
15	$G_{qkv} = \texttt{concat\_heads}(dQ_{\text{dot}} + dQ_{\text{rel}}, dK, dV)$	1.76	1.77
16	$(dN_1, dW_{qkv}, db_{qkv}) = (G_{qkv} W_{qkv}, G_{qkv}^{\mathsf{T}} N_1, \sum_r G_{qkv,r})$	4.07	2.64
17	$dX_b = \texttt{LN\_bwd}(dN_1; X_l)$	1.04	1.03
18	$dX_l = dR_1 + dX_b$	0.25	0.25
Kernel sum / layer (profiled; excludes fills)		50.35	39.23

Old − new: 11.122 ms/layer. Black box: affected group saving $> 10\%$ of this pass's total saving.

m2: fusion and FlashAttention

In this section, we go from 207,146 tokens/s to 445,372 tokens/s via fusion, a technique for reducing memory bandwidth demand, and our two-sweep FlashAttention-style implementation.

1. Fusion: keep intermediate values inside the kernel

We want to compute $Y[i] = g(f(X[i]))$, $0 \le i < N$, for elementwise f and g. X, U, Y are global-memory arrays; x, u, y are registers. Both versions round u to the stored type of U.

Old way

```
kernel_1:
    parallel for i = 0 .. N-1:
        x = load_global(X[i])
        u = round_U(f(x))
        store_global(U[i], u)
kernel_2:
    parallel for i = 0 .. N-1:
        u = load_global(U[i])
        y = g(u)
        store_global(Y[i], y)
```

New way

```
fused_kernel:
    parallel for i = 0 .. N-1:
        x = load_global(X[i])
        u = round_U(f(x))
        y = g(u)
        store_global(Y[i], y)
        if backward_needs(U):
            store_global(U[i], u)
```

Fusion removes the forward read of U by keeping u in a register. It also removes the write when backward does not need U.

Stable softmax

Softmax is generally computed as

$$\text{softmax}(s)_i = \frac{e^{s_i - \max(s)}}{\sum_j e^{s_j - \max(s)}}.$$

For intuition, consider $s = [1000000, 1000001]$ in FP32:

Direct

$$\begin{aligned} s &= [1000000, 1000001] \\ \text{softmax}(s) &= \left[\frac{e^{1000000}}{e^{1000000} + e^{1000001}}, \frac{e^{1000001}}{e^{1000000} + e^{1000001}}\right] \\ &\longrightarrow \left[\frac{\infty}{\infty + \infty}, \frac{\infty}{\infty + \infty}\right] \\ &= [\text{NaN}, \text{NaN}]. \end{aligned}$$

Subtract the maximum

$$\begin{aligned} s' &= s - 1000001 = [-1, 0] \\ \text{softmax}(s) &= \left[\frac{e^{-1}}{e^{-1} + e^{0}}, \frac{e^{0}}{e^{-1} + e^{0}}\right] \\ &\approx \left[\frac{0.36787945}{0.36787945 + 1}, \frac{1}{0.36787945 + 1}\right] \\ &\approx [0.2689414, 0.7310586]. \end{aligned}$$

We omit dropout to focus on softmax. For readability, the pseudocode also leaves out max subtraction and online rescaling; the kernels perform both.

2. FlashAttention-style attention: fuse scores, normalization and context

We want to compute $O = \mathrm{softmax}(S)V$. For each query row i:

$$O_{i,:} = \sum_{j=0}^{T-1} \frac{e^{S_{ij}}}{\sum_{k=0}^{T-1} e^{S_{ik}}} V_{j,:}$$

$$= \frac{e^{S_{i0}} V_{0,:} + e^{S_{i1}} V_{1,:} + \cdots + e^{S_{i,T-1}} V_{T-1,:}}{e^{S_{i0}} + e^{S_{i1}} + \cdots + e^{S_{i,T-1}}}.$$

One unpadded head: $Q, K, V \in \mathbb{R}^{T \times d_h}$; $R = QE_r^{\mathsf{T}}$ is precomputed. $\mathcal{I}, \mathcal{J}$: query/key tiles; R16: BF16 rounding; accumulators and reductions: FP32.

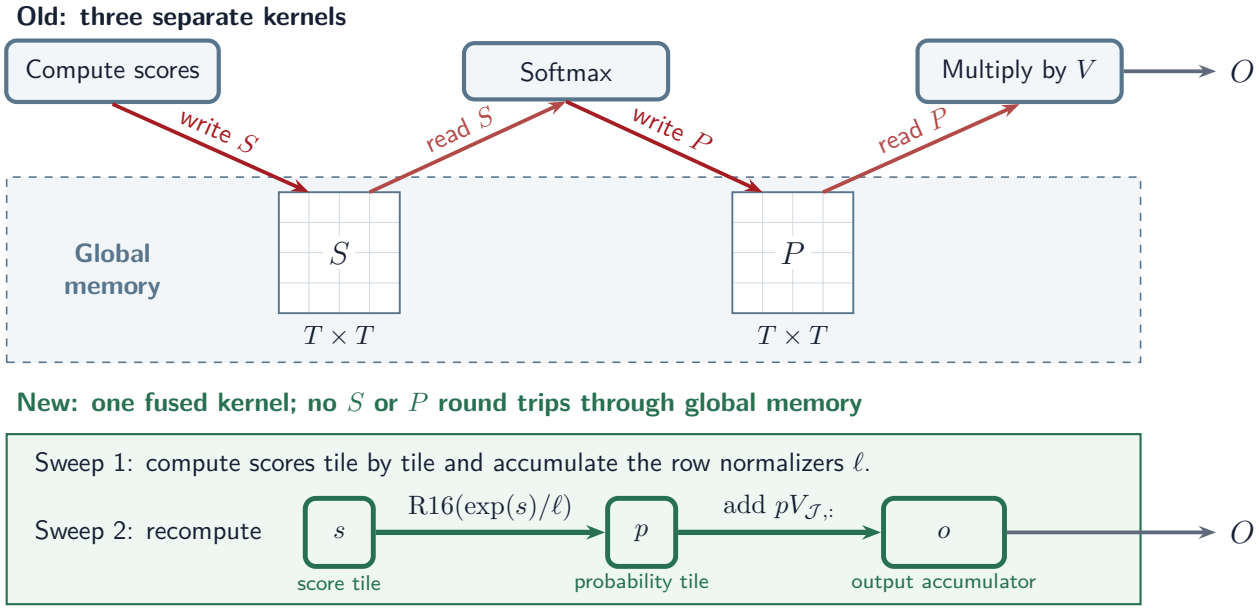
`Global-memory read` **`Global-memory write`** Register operation

Old way

```
kernel_scores: parallel for query tile I, key tile J:
    q, k = load_global(Q[I,:]), load_global(K[J,:])
    r = load_global(R[I, T-1-I+J], mask=J<=I, other=0)
    s = mask((q @k.T + fp32(r)) / sqrt(d_h), J <= I)
    store_global(S[I,J], s)              // full T x T score plane
kernel_softmax: parallel for query row i:
    ell = 0
    for J in key_tiles: ell += sum(exp(load_global(S[i,J])))
    for J in key_tiles:
        s = load_global(S[i,J])
        p = R16(exp(s) / ell)
        store_global(P[i,J], p)          // full T x T probability plane
    store_global(LSE[i], log(ell))
kernel_context: parallel for query tile I:
    o = 0
    for J in key_tiles:
        p, v = load_global(P[I,J]), load_global(V[J,:])
        o += p @v
    store_global(O[I,:], R16(o))
```

Our two-sweep FlashAttention-style implementation

Our fused kernel makes two sweeps over the key tiles: first compute each row's softmax statistics, then recompute scores and immediately consume the normalized probabilities in the context multiply. The second sweep preserves the BF16 rounding of probabilities before the context multiply.



`Global-memory read` **`Global-memory write`** Register operation

```
fused_kernel: parallel for query tile I:
    ell = 0                                // one normalizer per row
    for J in causal_key_tiles(I):          // sweep 1: softmax statistics
        q, k = load_global(Q[I,:]), load_global(K[J,:])
        r = load_global(R[I, T-1-I+J], mask=J<=I, other=0)
        s = mask((q @k.T + fp32(r)) / sqrt(d_h), J <= I)
        ell += rowsum(exp(s))
    o = 0
    for J in causal_key_tiles(I):          // sweep 2: recompute and consume
        q, k = load_global(Q[I,:]), load_global(K[J,:])
        r = load_global(R[I, T-1-I+J], mask=J<=I, other=0)
        s = mask((q @k.T + fp32(r)) / sqrt(d_h), J <= I)
        p = R16(exp(s) / ell)
        v = load_global(V[J,:])
        o += p @v                          // no global S or P
    store_global(O[I,:], R16(o))
    store_global(norm_stats[I], state(ell))  // stabilized representation + LSE
```

The kernel avoids global writes and reads of S and P. Backward reconstructs probability tiles from the saved row maximum and shifted denominator. The FP32 reduction order changes, so results need not be bitwise identical to the old kernels.

The relative-score plane R and the separate head join remain.

Forward timings: fusion and FlashAttention applied

Layer code m1 → m2 (v040 → v070)		Kernel ms (profiled) excludes fills Old	New
01	X_l	—	—
	02 $N_1 = \texttt{LN}(X_l)$	0.77	0.23
	03 $(Q, K, V) = \texttt{split\_heads}(N_1 W_{qkv}^{\mathsf{T}} + b_{qkv})$	2.57	1.97
	03r $R = QE_r^{\mathsf{T}}$	2.07	0.59
A1	04 $S = \texttt{mask}((QK^{\mathsf{T}} + \texttt{skew}(R))/\sqrt{d_h})$ 05 $P_{i:} = \texttt{softmax}(S_{i:})$ 06 $P_{\text{drop}} = \texttt{dropout}(P)$ 07 $O = P_{\text{drop}} V$	18.27	2.32
	07j $C = \texttt{join\_heads}(O)$	0.20	0.20
	08 $A = CW_o^{\mathsf{T}} + b_o$	0.59	0.43
	09 $A_{\text{drop}} = \texttt{dropout}(A)$	0.60	0.41
10	$R_1 = X_l + A_{\text{drop}}$		
	11 $N_2 = \texttt{LN}(R_1)$	0.78	0.23
	12 $U = N_2 W_1^{\mathsf{T}} + b_1$ 13 $H_{\text{mlp}} = \texttt{GELU}(U)$	3.54	2.40
	14 $M = H_{\text{mlp}} W_2^{\mathsf{T}} + b_2$	1.66	1.49
	15 $M_{\text{drop}} = \texttt{dropout}(M)$	0.59	0.40
16	$X_{l+1} = R_1 + M_{\text{drop}}$		
Kernel sum / layer (profiled; excludes fills)		31.66	10.68

Old − new: 20.988 ms/layer. Black box: affected group saving $> 10\%$ of this pass's total saving.

Backward timings: fusion and FlashAttention applied

Layer code m1 → m2 (v040 → v070)		Kernel ms (profiled) excludes fills Old	New
01	dX_{l+1}	—	—
02	$dM = \texttt{dropout\_bwd}(dX_{l+1})$	0.26	0.26
03	$(dH_{\text{mlp}}, dW_2, db_2) = (dMW_2, dM^\top H_{\text{mlp}}, \sum_r dM_r)$	4.65	3.92
04	$dU = dH_{\text{mlp}} \odot \texttt{GELU}'(U)$		
05	$(dN_2, dW_1, db_1) = (dUW_1, dU^\top N_2, \sum_r dU_r)$	3.31	3.35
06	$dR_{1,b} = \texttt{LN\_bwd}(dN_2; R_1)$	1.03	0.54
07	$dR_1 = dX_{l+1} + dR_{1,b}$	0.25	0.26
08	$dA = \texttt{dropout\_bwd}(dR_1)$	0.26	0.26
09	$(dC, dW_o, db_o) = (dAW_o, dA^\top C, \sum_r dA_r)$	0.89	0.91
10	$\delta_i = dO_i \cdot O_i$		
Q 11	$P^{(t)} = \exp(S^{(t)} - m)/\ell, \quad dS^{(t)} = P^{(t)} \odot (dP^{(t)} - \delta)$	20.97	4.85
Q 12	$dZ^{(t)} = dS^{(t)}/\sqrt{d_h}, \quad dQ_{\text{dot}} \mathrel{+}= dZ^{(t)} K, \quad dR = \texttt{unskew}(dZ^{(t)})$		
KV 13	$(dK, dV) \mathrel{+}= ((dZ^{(t)})^\top Q, (P^{(t)}_{\text{drop}})^\top dO)$		
14	$(dQ_{\text{rel}}, dE_r) = (dRE_r, dR^\top Q)$	1.91	1.35
L 15	$G_{qkv} = \texttt{concat\_heads}(dQ_{\text{dot}} + dQ_{\text{rel}}, dK, dV)$	1.77	0.64
16	$(dN_1, dW_{qkv}, db_{qkv}) = (G_{qkv} W_{qkv}, G_{qkv}^\top N_1, \sum_r G_{qkv,r})$	2.64	2.70
17	$dX_b = \texttt{LN\_bwd}(dN_1; X_l)$	1.03	0.54
18	$dX_l = dR_1 + dX_b$	0.25	0.25
Kernel sum / layer (profiled; excludes fills)		39.23	19.84

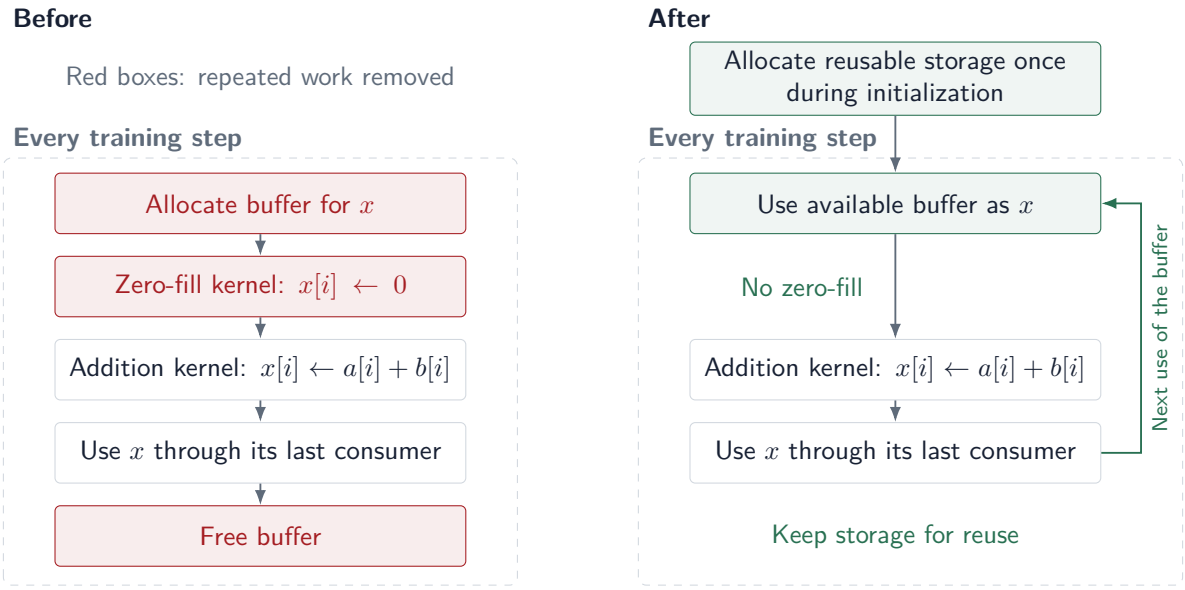
Old − new: 19.385 ms/layer. Black box: affected group saving $> 10\%$ of this pass's total saving.

The resulting m2 implementation achieves 445,372 tokens/s. This is the starting point for m3.

m3: remove redundant memory work

Starting from m2 at 445,372 tokens/s, m3 reaches 660,540 tokens/s: a 48.31% throughput increase.

Consider $x = a + b$. The addition kernel writes every element of x. Previously, we allocated storage for x, launched a kernel to fill it with zeros, then overwrote those zeros with the result. With hundreds of temporary tensors per step, the costs of buffer management and zeroing add up.



The addition and its consumers stay the same. We remove repeated allocation, zeroing and freeing by keeping the storage and writing the result directly into it. A buffer can be reused only after its previous value's last consumer finishes. Accumulators that add into an existing value still need initialization.

Where the time went

Across the full training step, initialization falls from 557 launches to 7. Non-initialization kernel time barely changes:

Per training step, profiled	Before (m2)	After (m3)
Non-initialization kernel time (ms)	207.03	206.57
Initialization kernel time (ms)	95.69	0.01
Gaps without traced GPU work (ms)	57.23	0.82
Initialization kernel launches	557	7
CUDA allocation requests (`cuMemAllocAsync`)	563	0
CUDA free requests (`cuMemFreeAsync`)	563	0

Buffer reuse and zero-fill removal together account for 47.84% more tokens/s; their individual contributions were not measured separately. Capturing the training step in a CUDA Graph replaces individual host submissions with one replay; the measured throughput difference was +0.36%. We also prepare and upload the next input batch during the current step; the measured difference in the fixed-batch GPU benchmark was -0.04%.

r16: switching from cuTile-rs to CUDA-Oxide

This optimization takes us from 660,540 tokens/s to 1,125,109 tokens/s

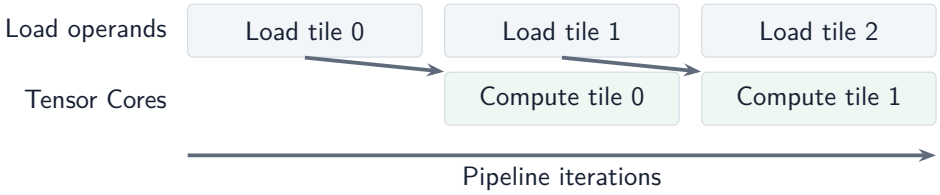
Both implementations use Rust; the GPU programming interface changes. This section gives a high-level view of the ideas and benchmarks.

Starting from m3, r16 combines better pipeline control, attention fusion, and two concurrent streams. The model remains six layers, $D = 512$, $d_h = 64$, eight heads, $B = 512$, $T = 256$.

1. Better pipeline control

Control blocks, warps and threads to load the next tile while Tensor Cores compute the current one. Fuse bias, dropout and the residual into the GEMM epilogue, so the projection result stays in registers until the final store:

$$R_1 = X_\ell + \text{dropout}(CW_o^\mathsf{T} + b_o).$$



2. Attention: one sweep and relative-position fusion

Compute each score tile once, including its relative-position term, and immediately accumulate the weighted values and normalizer. Divide after all causal key tiles (dropout omitted for simplicity):

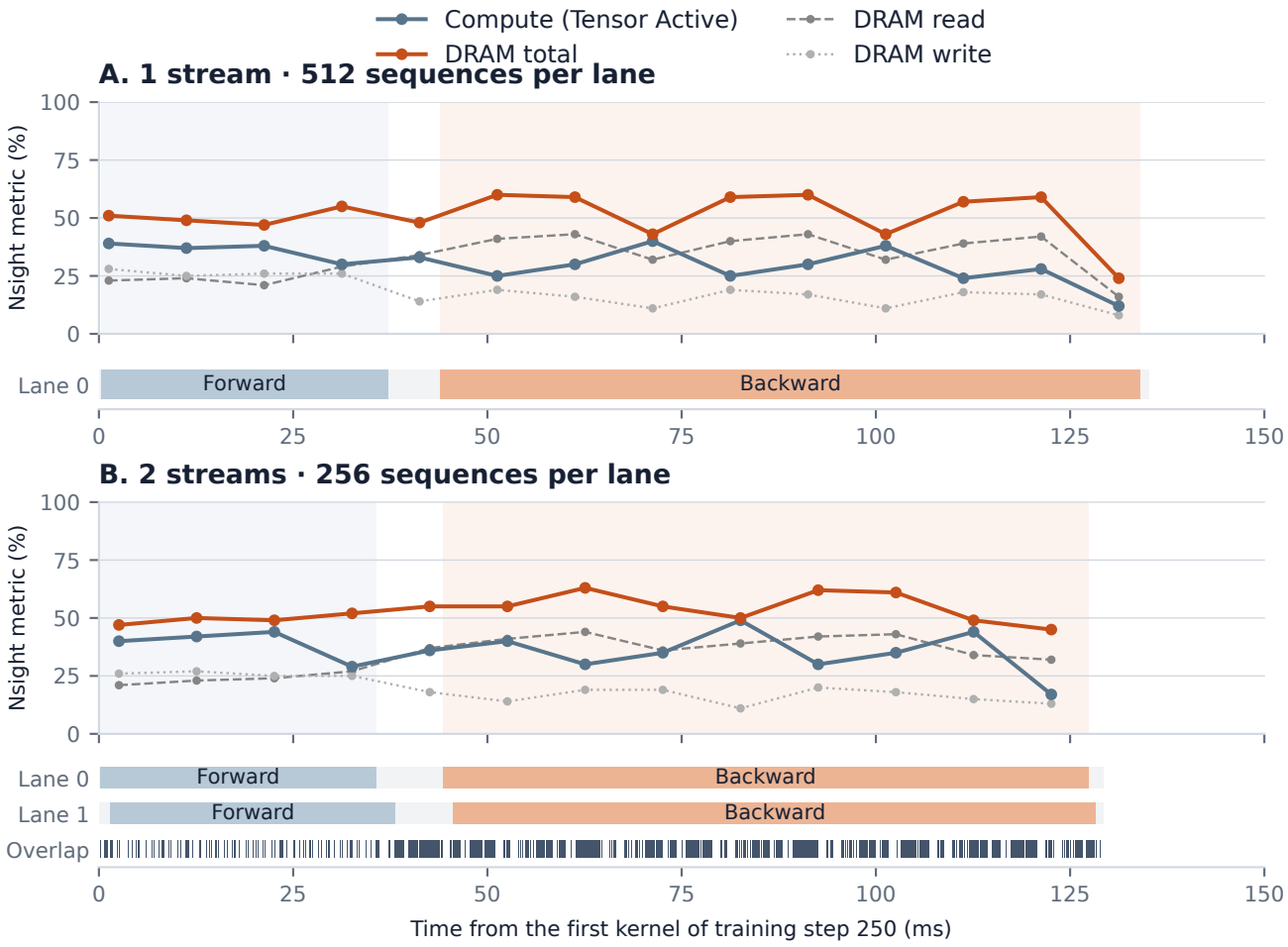
$$u_i = \sum_{j \le i} e^{s_{ij}} V_j, \qquad \ell_i = \sum_{j \le i} e^{s_{ij}}, \qquad O_i = \frac{u_i}{\ell_i}.$$

For numerical stability, the kernel maintains a running maximum and rescales both the numerator and denominator as that maximum changes. The two-sweep version rounds normalized probabilities to BF16 before multiplying by V. The one-sweep version rounds intermediate weights before final normalization. Moving this rounding changes floating-point results even though the underlying mathematical operation is the same.

Forward avoids global R, S and P matrices. Backward also consumes probability and derivative tiles locally; it retains dR for the relative-embedding gradient.

3. Run two streams concurrently

The resource mix changes through a training step. This motivates trying two independent streams: Tensor Core work in one may overlap with memory work in the other. The captures below show the measured activity and execution overlap.



Device-wide samples at 100 Hz. Tensor Active measures Tensor Core pipeline activity, not achieved FLOPs or model FLOPs utilization (MFU).

One lane: pass means across the full capture	Forward	Backward
Tensor Active / SM Issue (%)	39.0 / 22.0	31.2 / 13.8
DRAM read / write (%)	23.1 / 26.7	37.8 / 15.1

The two lanes' forward/backward windows remain close in this capture. Kernel overlap occurs within those windows; it covers 36.08% of the full trace. The measured schedule does not enforce a whole-pass stagger.

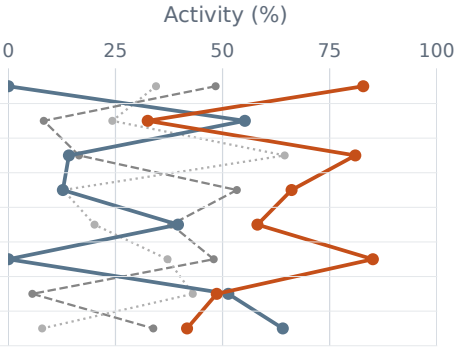
Layer pseudocode: measured compute and DRAM activity

Each code row below is one physical CUDA-Oxide v1 kernel. Follow the colored traces down the layer to see where Tensor Core activity and DRAM traffic rise. Gray traces split DRAM total into reads and writes.

Each point is a per-kernel average. Equal row spacing shows execution order, not elapsed time. Tensor Active measures pipeline activity, not achieved FLOPs or model FLOPs utilization (MFU).

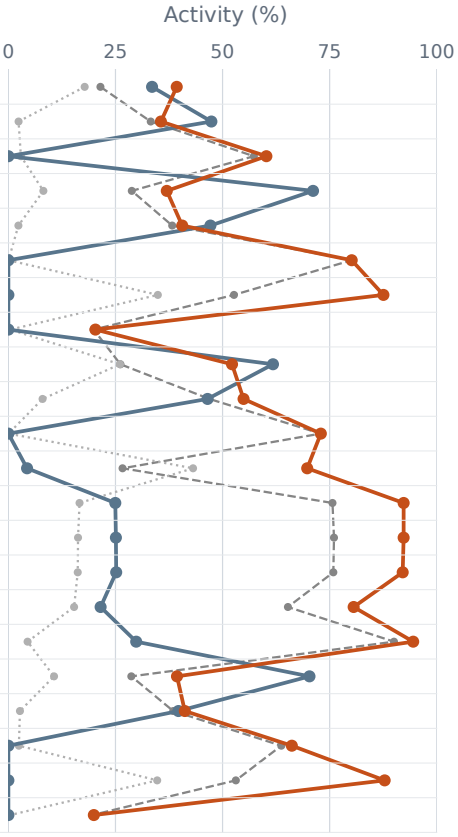
Forward

02	`N1 = LN(X)`
03	`Q,K,V = qkv_heads(N1, Wqkv, bqkv)`
03r	`R = Q @ E_r.T`
04–07j	`C,tape = attention_join(Q,K,V,R)`
08–10	`R1 = X + dropout(C @ Wo.T + bo)`
11	`N2 = LN(R1)`
12–13	`U,H_mlp = linear_gelu(N2, W1, b1)`
14–16	`Y = R1 + dropout(H_mlp @ W2.T + b2)`



Backward

03–04	`dU = (dM @ W2) * GELU′(U)`
03	`p2 = partial_Wb(dM, H_mlp)`
03	`dW2,db2 += finish(p2)`
05	`dN2 = dU @ W1`
05	`p1 = partial_Wb(dU, N2)`
05	`dW1,db1 += finish(p1)`
06–08	`dR1,dA,pLN2 = LN_res_drop_bwd(...)`
06	`dgamma2,dbeta2 += finish(pLN2)`
09	`dC = dA @ Wo`
09	`po = partial_Wb(dA, C)`
09	`dWo,dbo += finish(po)`
10–12	`dZ,P_drop,dR = attention_bwd(...)`
13	`Gqkv.V = P_drop.T @ dO`
13	`Gqkv.K = dZ.T @ Q`
12,15	`Gqkv.Q = dZ @ K`
14a,15	`Gqkv.Q += dR @ E_r`
14b	`pEr = partial(dR.T @ Q)`
16	`dN1 = Gqkv @ Wqkv`
16	`pqkv = partial_Wb(Gqkv, N1)`
14b,16	`dWqkv,dbqkv,dE_r += finish(pqkv,pEr)`
17–02	`dX,dM_prev,pLN1 = LN_res_drop_bwd(...)`
17	`dgamma1,dbeta1 += finish(pLN1)`



Two lanes: split the batch and merge gradients

Split the 512 sequences into two independent groups of 256. Each stream has its own workspace and runs forward and backward; after both finish, merge their gradients and update the shared weights once.

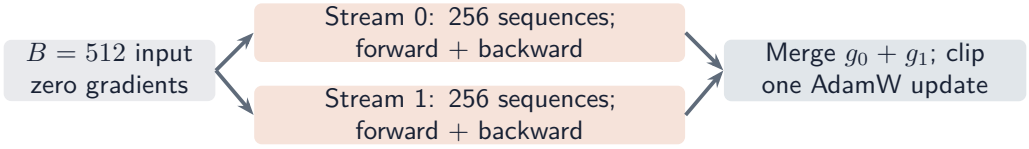
For token loss $\ell_{b,t}$, both schedules produce one batch gradient and one update, $\theta' \leftarrow \mathrm{AdamW}(\theta, \mathrm{clip}(g))$. In both formulas, $B = 512$ is the global batch size; each lane contributes 256 sequences.

Old math: one batch of 512

$$g \leftarrow \frac{1}{BT} \sum_{b=0}^{511} \sum_{t} \nabla_\theta \ell_{b,t}$$

New math: two batches of 256

$$g \leftarrow g_0 + g_1, \qquad g_r = \frac{1}{BT} \sum_{b \in \mathcal{B}_r, t} \nabla_\theta \ell_{b,t}$$



Forward timings: m3 versus r16

Layer code m3 → r16	Kernel ms (profiled) per layer m3	r16
01 X_l	—	—
02 $N_1 = \texttt{LN}(X_l)$	0.23	0.192
03 $(Q, K, V) = \texttt{split\_heads}(N_1 W_{qkv}^{\mathsf{T}} + b_{qkv})$	1.90	1.162
03r $R = QE_r^{\mathsf{T}}$		
04 $S = \texttt{mask}((QK^{\mathsf{T}} + \texttt{skew}(R))/\sqrt{d_h})$		
05 $P = \texttt{softmax}(S)$	3.13	1.016
06 $P_{\text{drop}} = \texttt{dropout}(P)$		
07 $O = P_{\text{drop}} V$		
07j $C = \texttt{join\_heads}(O)$		
08 $A = CW_o^{\mathsf{T}} + b_o$		
09 $A_{\text{drop}} = \texttt{dropout}(A)$	0.83	0.578
10 $R_1 = X_l + A_{\text{drop}}$		
11 $N_2 = \texttt{LN}(R_1)$	0.23	0.242
12 $U = N_2 W_1^{\mathsf{T}} + b_1$	2.37	1.797
13 $H_{\text{mlp}} = \texttt{GELU}(U)$		
14 $M = H_{\text{mlp}} W_2^{\mathsf{T}} + b_2$		
15 $M_{\text{drop}} = \texttt{dropout}(M)$	1.89	1.413
16 $X_{l+1} = R_1 + M_{\text{drop}}$		
Kernel sum / layer	10.57	6.401

Old − new: 4.169 ms/layer. Black box: affected group saving $> 10\%$ of this pass's total saving.

Profiled kernel sums	m3	r16
Average ms/layer	10.57	6.401
All six layers, ms/step	63.42	38.41

Durations sum kernels across both r16 streams. Overlapping execution is counted separately, so kernel sums can exceed elapsed step time. The same r16 capture is used in the subsequent x2 comparison.

Backward timings: m3 versus r16

Layer code m3 → r16		Kernel ms (profiled) per layer m3	r16
01	dX_{l+1}	—	—
02	$dM = \texttt{dropout\_bwd}(dX_{l+1})$	incl. 17–18	incl. 17–18
03	$dH_{\text{mlp}} = dMW_2, \quad (dW_2, db_2) = (dM^\top H_{\text{mlp}}, \sum_r dM_r)$	3.99	4.666
04	$dU = dH_{\text{mlp}} \odot \texttt{GELU}'(U)$		
05	$(dN_2, dW_1, db_1) = (dUW_1, dU^\top N_2, \sum_r dU_r)$	3.38	3.872
06	$dR_{1,b} = \texttt{LN\_bwd}(dN_2; R_1)$		
07	$dR_1 = dX_{l+1} + dR_{1,b}$	1.08	0.724
08	$dA = \texttt{dropout\_bwd}(dR_1)$		
09	$(dC, dW_o, db_o) = (dAW_o, dA^\top C, \sum_r dA_r)$	0.90	1.594
10	$\delta_i = dO_i \cdot O_i$		
11	$P = \exp(S - \text{LSE}), \quad dS = P \odot (dP - \delta)$		
12	$dZ = dS/\sqrt{d_h}, \quad dQ_{\text{dot}} = dZK, \quad dR = \texttt{unskew}(dZ)$		
13	$(dK, dV) = (dZ^\top Q, P_{\text{drop}}^\top dO)$	9.51	7.516
14a	$dQ_{\text{rel}} = dRE_r$		
15	$G_{qkv} = \texttt{concat\_heads}(dQ_{\text{dot}} + dQ_{\text{rel}}, dK, dV)$		
14b	$dE_r = dR^\top Q$		
16	$(dN_1, dW_{qkv}, db_{qkv}) = (G_{qkv}W_{qkv}, G_{qkv}^\top N_1, \sum_r G_{qkv,r})$		
17	$dX_b = \texttt{LN\_bwd}(dN_1; X_l)$	1.05	0.641
18	$dX_l = dR_1 + dX_b$		
Kernel sum / layer		19.91	19.012

Old − new: 0.896 ms/layer. Black box: affected group saving $> 10\%$ of this pass's total saving.

Profiled kernel sums	m3	r16
Average ms/layer	19.91	19.012
All six layers, ms/step	119.45	114.07

Durations sum kernels across both r16 streams. Overlapping execution is counted separately, so kernel sums can exceed elapsed step time. The same r16 capture is used in the subsequent x2 comparison.

x2: smaller model

We will now go from 1,125,109 tokens/s to 3,217,492 tokens/s via two model changes: processing every other frame and halving the width.

The CPU selects frames $0, 2, \ldots, 254$ from each 256-frame window; only 128 frames reach the GPU. Model width falls from 512 to 256, heads from eight to four, and MLP width from 2048 to 1024. Both models keep six layers, 64-wide heads, v2 attention fusion, batch 512 and two lanes.

Here, tokens/s counts processed frames: 256 per r16 sequence and 128 per x2 sequence. x2 also recomputes the GELU input during backward instead of storing it; that work is included in the timings.

In the CPU9 runs shown on page 1, x2 won 26/30 games after ten minutes of training, versus 28/30 for r16.

Forward timings

Layer code r16 → x2	Kernel ms (profiled) per layer: Old	New
01 X_l	—	—
02 $N_1 = \texttt{LN}(X_l)$	0.192	0.051
03 $(Q, K, V) = \texttt{split\_heads}(N_1 W_{qkv}^{\mathsf{T}} + b_{qkv})$	1.162	0.186
03r $R = QE_r^{\mathsf{T}}$		
04 $S = \texttt{mask}((QK^{\mathsf{T}} + \texttt{skew}(R))/\sqrt{d_h})$		
05 $P = \texttt{softmax}(S)$	1.016	0.146
06 $P_{\text{drop}} = \texttt{dropout}(P)$		
07 $O = P_{\text{drop}} V$		
07j $C = \texttt{join\_heads}(O)$		
08 $A = CW_o^{\mathsf{T}} + b_o$		
09 $A_{\text{drop}} = \texttt{dropout}(A)$	0.578	0.111
10 $R_1 = X_l + A_{\text{drop}}$		
11 $N_2 = \texttt{LN}(R_1)$	0.242	0.058
12 $U = N_2 W_1^{\mathsf{T}} + b_1$	1.797	0.280
13 $H_{\text{mlp}} = \texttt{GELU}(U)$		
14 $M = H_{\text{mlp}} W_2^{\mathsf{T}} + b_2$		
15 $M_{\text{drop}} = \texttt{dropout}(M)$	1.413	0.293
16 $X_{l+1} = R_1 + M_{\text{drop}}$		
Kernel sum / layer (both lanes)	6.401	1.124

Old − new: 5.277 ms/layer. Black box: affected group saving $> 10\%$ of this pass's total saving.

Backward timings

Layer code r16 → x2		Kernel ms (profiled) per layer Old	New
01	dX_{l+1}	—	—
02	$dM = \texttt{dropout\_bwd}(dX_{l+1})$	fused LN	fused LN
03	$dH_{\text{mlp}} = dMW_2, \quad (dW_2, db_2) = (dM^\top H_{\text{mlp}}, \sum_r dM_r)$	4.666	0.772
04	$dU = dH_{\text{mlp}} \odot \texttt{GELU}'(U)$		
05	$(dN_2, dW_1, db_1) = (dUW_1, dU^\top N_2, \sum_r dU_r)$	3.872	0.562
06	$dR_{1,b} = \texttt{LN\_bwd}(dN_2; R_1)$		
07	$dR_1 = dX_{l+1} + dR_{1,b}$	0.724	0.145
08	$dA = \texttt{dropout\_bwd}(dR_1)$		
09	$(dC, dW_o, db_o) = (dAW_o, dA^\top C, \sum_r dA_r)$	1.594	0.253
10	$\delta_i = dO_i \cdot O_i$		
11	$P = \exp(S - \text{LSE}), \quad dS = P \odot (dP - \delta)$		
12	$dZ = dS/\sqrt{d_h}, \quad dQ_{\text{dot}} = dZK, \quad dR = \texttt{unskew}(dZ)$	3.479	0.630
13	$(dK, dV) = (dZ^\top Q, P_{\text{drop}}^\top dO)$		
14a	$dQ_{\text{rel}} = dRE_r$		
15	$G_{qkv} = \texttt{concat\_heads}(dQ_{\text{dot}} + dQ_{\text{rel}}, dK, dV)$		
14b	$dE_r = dR^\top Q$	4.037	0.609
16	$(dN_1, dW_{qkv}, db_{qkv}) = (G_{qkv} W_{qkv}, G_{qkv}^\top N_1, \sum_r G_{qkv,r})$		
17	$dX_b = \texttt{LN\_bwd}(dN_1; X_l)$	0.641	0.158
18	$dX_l = dR_1 + dX_b$		
Kernel sum / layer (both lanes)		19.012	3.129

Old − new: 15.883 ms/layer. Black box: affected group saving $> 10\%$ of this pass's total saving.

PyTorch BF16 failures

Training instability

Separate width-512 PyTorch BF16 (`p16a`) runs developed non-finite gradients or diverged; the paired FP32 runs completed. Those records do not isolate a root cause. They limit what the short BF16 throughput windows establish.

The five-minute point on page 1

Checkpoint: 300 seconds of training, step 1,927; 30 Fox-versus-CPU9-Fox games on Final Destination. The model repeatedly moves off the left edge and fails to recover. The plotted damage-taken value is $301/30 = 10.03$, rounded to 10.0.

Replay-verified quantity	Value
Completed games / available Slippi replays	30 / 30
Replay SHA-512 matches against `complete.json`	30 / 30
Wins / losses	0 / 30
Model stocks lost / CPU stocks lost	120 / 0
Total damage taken / dealt by the model	301 / 47
Average damage taken / dealt per game	10.03 / 1.57
Games with zero damage taken	5 / 30
Stock losses at 0% damage	69 / 120
Deaths through the bottom blast zone (`DEAD_DOWN`)	119 / 120
Deaths with horizontal position $x < -85$	118 / 120
Replay length, frames	1,066–1,752
Replay length at 60 Hz, seconds	17.77–29.20

Damage taken accumulates positive per-frame increments across stocks:

$$\overline{D}_{\text{taken}} = \frac{1}{30} \sum_{g=1}^{30} \sum_{t} \max(0,\ p_{g,t} - p_{g,t-1}) \,.$$

Here $p_{g,t}$ is the model's damage percentage exposed by the replay reader. All 30 decoded frame counts, final stocks and cumulative damage exactly matched the recorded evaluation summaries.

Example: evaluation seed 0

Replay controls and positions show leftward running followed by offstage falls.

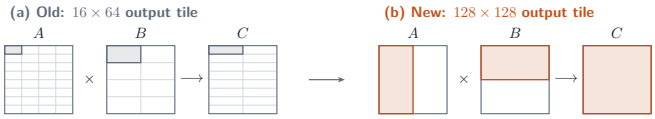
Stock transition	Slippi frame	Damage before death	x at death	y at death
$4 \to 3$	83	0%	−159.0	−140.7
$3 \to 2$	362	0%	−160.9	−142.1
$2 \to 1$	653	0%	−152.3	−141.1
$1 \to 0$	942	0%	−154.8	−142.0

All four deaths enter `DEAD_DOWN`; replay frames −123 through 942 give 1,066 recorded frames.

Conclusion

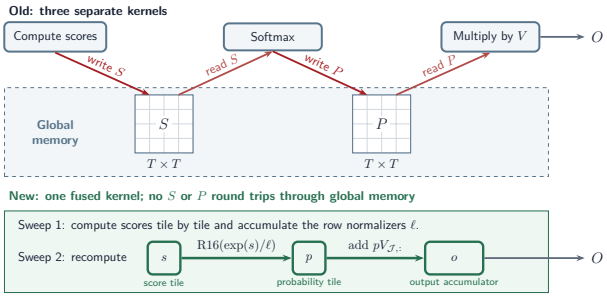
1. Use Tensor Cores efficiently

Matrix multiplication is expensive. Therefore, use Tensor Cores and tile the work to reuse operands across more output elements.



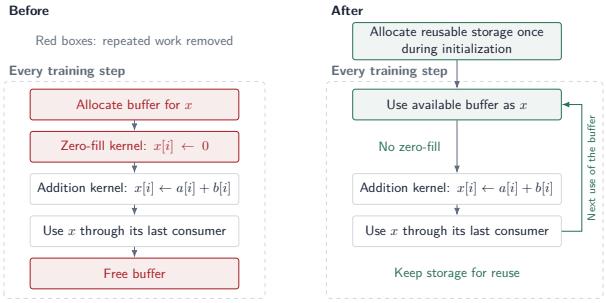
2. Reduce memory traffic through fusion

Memory transfers add latency. Therefore, use fusion and FlashAttention to keep intermediate results inside the kernel and avoid global-memory round trips.



3. Avoid unnecessary alloc/zero/free

Unnecessary allocation, zeroing, and freeing do useless work. Therefore, reuse buffers and write results directly, skipping zeroing when every element will be overwritten.

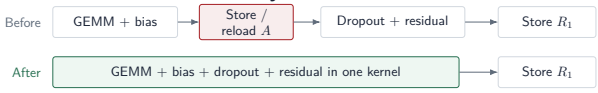


4. Design kernels around the workload

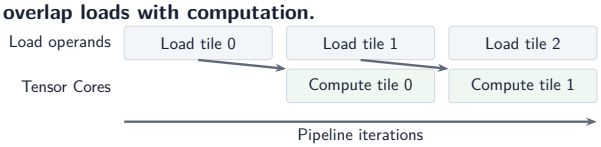
CUDA-Oxide addresses several costs with specialized attention and GEMM kernels, fused outputs and concurrent streams.

Extra attention sweeps and score storage add work. Therefore, compute scores once and keep them local.

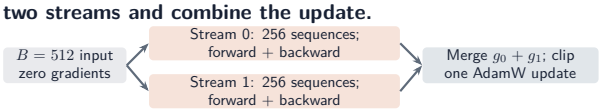
Separate output kernels add memory traffic. Therefore, fuse outputs and write the consumer's layout.



Waiting for operand loads leaves Tensor Cores idle. Therefore, overlap loads with computation.



Serial execution limits overlap. Therefore, run independent work on two streams and combine the update.



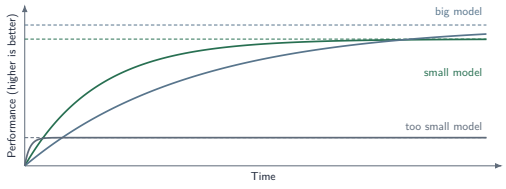
5. The black magic: trading a lower ceiling for faster learning

Even with perfect optimization, a standard SGEMM (FP32 matrix multiply), $C_{m\times n} = A_{m\times k}B_{k\times n}$, still costs:

$$\begin{aligned} \text{Compute:} &\quad \approx 2mnk \text{ FLOPs}, \\ \text{Memory traffic:} &\quad \geq 4(mk + kn + mn) \text{ bytes}. \end{aligned}$$

The memory bound assumes each input is read once and the output is written once. Fixed compute throughput and memory bandwidth put a floor on runtime. Halving all three dimensions cuts arithmetic by 8× and minimum memory traffic by 4×.

Therefore, reduce the actual work by shrinking matrix dimensions: use a smaller model. The tradeoff is a lower performance ceiling for faster learning; a model that is too small can plateau before it learns enough.



Illustrative curves, not measured results.

Obsessed with GPU optimizations?
Want to train recursively self-improving models that play games?
Email rohit@frisson-labs.com.