Data-Movement-Aware Optimization of Transformer Attention on Gemmini

ELEC 5140 Final Project

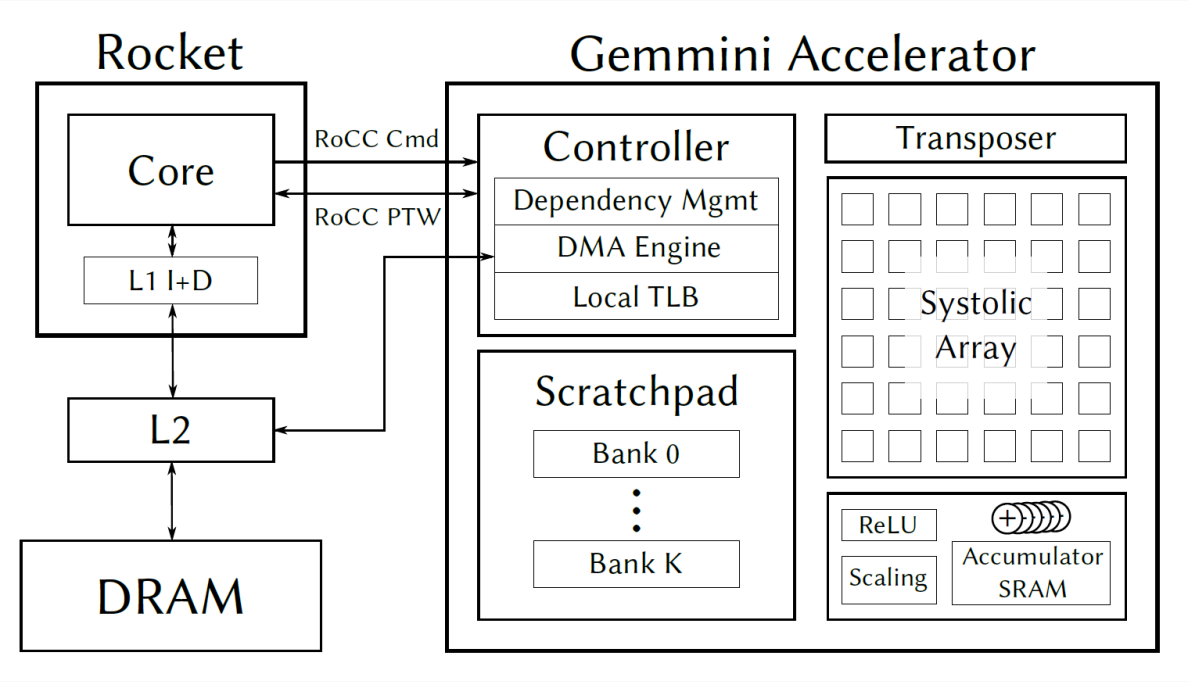
Ivan Lok

May 2026

IBertGemminiRocketConfig · DIM=16 · ACC_ROWS=2048 · WS Dataflow

BERT-base: H=768, 12 heads, D=64 · SEQ_LEN ∈ {128, 256, 512}

What is Gemmini?



RoCC Accelerator
Rocket issues custom instructions; Gemmini handles tiled matrix operations through its controller.

Scratchpad
On-chip operand SRAM. Matmul A/B operands must be staged here before the systolic array consumes them.

Accumulator
Matmul outputs land here as wider partial sums, then can be scaled, activated, normalized, or moved out.

Systolic Array
16×16 weight-stationary mesh in this project config; this is where QK and PV compute runs.

Attention gap: QK puts scores in the accumulator. PV needs weights in the scratchpad. There's no direct path between them.

Gemmini Constraints That Shape Attention

1. Two On-Chip Memories

Scratchpad feeds operands into the systolic array.
Accumulator receives QK/PV matmul results and supports scale / normalization.

2. Accumulator Is Double-Buffered

With `ACC_ROWS=2048`, each WS buffer half has **1024 usable rows**.

3. Built-In SOFTMAX Is Full-Row

The denominator needs the whole row, so the standard softmax path cannot freely stream partial K-blocks.

Attention Tensors Stress Different Memories

Tensor	Shape per head	Where it naturally lives	Why it matters
Q per head	SEQ × 64	Scratchpad input	Left operand for QK^T
K per head	SEQ × 64	Scratchpad input	Right operand for QK^T
V per head	SEQ × 64	Scratchpad input	Right operand for PV
QK scores	SEQ × SEQ	**Accumulator**	Produced by systolic $Q \times K^T$
Softmax weights	SEQ × SEQ	**Scratchpad for PV**	Consumed as PV's A operand
PV output	SEQ × 64	**Accumulator / DRAM**	Final per-head output

QK writes into the accumulator; PV wants weights in the scratchpad.

Baseline Gemmini Attention Workflow

Where the attention matrix leaves and re-enters the chip

Stage	Data Movement	Why It Matters
1. Q/K/V projections Standard tiled matmuls produce the per-head inputs.	DRAM → SP → Systolic → ACC → DRAM	Normal input/output traffic. seq=128: 960K cycles
2. QK^T + softmax Scores are produced and normalized on chip.	Q,K in SP → QK^T → Softmax → mvout A	A is temporary, but baseline writes it to DRAM. First half of round trip
3. PV matmul PV immediately consumes the same A matrix.	DRAM A → mvin to SP → PV → Output	A returns from DRAM just after it was written. Second half of round trip
4. Wo + LayerNorm Rest of the Transformer block.	DRAM → SP → Wo → LN	Not the first target in this project. seq=128: 497K cycles

QK^T puts scores in the accumulator. PV needs weights in the scratchpad. The attention matrix shuttles through DRAM for no reason.

The Problem: An O(seq²) Intermediate Tensor

Attention Core

Scores: $S = Q \times K^T$
Weights: $A = \text{softmax}(S)$
Output: $O = A \times V$

For BERT-base: 12 heads, so the attention matrix cost grows with $12 \times SEQ^2$.

The Bottleneck

Attention matrix A is an intermediate tensor:

- **Produced by QK^T + softmax**
- **Consumed immediately by PV**
- **Baseline still sends it through DRAM**

QK^T → Softmax → DRAM write → DRAM read → PV

Seq	Attention Matrix	Wasted DRAM
128	$12 \times 128^2 = 196K$	393 KB
256	$12 \times 256^2 = 786K$	1.57 MB
512	$12 \times 512^2 = 3.1M$	6.3 MB

This traffic grows with $O(seq^2)$, exactly like the attention matrix itself.

So: can we keep the attention matrix on-chip and skip DRAM entirely?

Experiment 1: Bypass the DRAM Round-Trip

The Core Insight

The first attempt keeps the attention weights inside Gemmini:

Q × K^T → SOFTMAX → A (weights) → A × V

QK and softmax still run as Gemmini steps; only the destination changes.

Experiment 1.2: Data-Movement Fusion

Hardware: route softmax mvout output directly into scratchpad.

Software: write with `GEMMINI_LOCAL_STORE_FLAG`; run PV with `A=NULL`.

Dataflow Transformation

Baseline (2 DRAM transfers):

QK^T → Softmax → mvout→DRAM → mvin→SP → PV

Fused (0 DRAM transfers):

QK^T → Softmax → local store→SP → PV (A=NULL)

Hardware Change

- Detect `GEMMINI_LOCAL_STORE_FLAG`
- Route acc/softmax output to scratchpad
- Bypass the normal DRAM writer

Result at seq=128: attention core 611K → 358K; full pipeline 2.15M → 1.88M.

Experiment 1 Result: Good Idea, Bad Scaling

Seq Len	Baseline Attention	Exp 1 Attention	Speedup	Full Pipeline	Status
128	611,290	357,667	1.71× faster	1.88M (vs 2.15M)	WIN
256	1,037,836	6,680,878	6.4× slower	~9.54M (vs 3.97M)	LOSE
512	2,623,055	CRASH @ 204M cycles — TL assertion			CRASH

Why 128 Wins

DRAM round-trip cost (786 KB DMA) > local store overhead (~50K cycles for 3,072 mvout).

Attention: −41% · Full: −12.4%

Why 256/512 Fails

Both J-tile count (SOFTMAX constraint) and Q-block count (Q_BLOCK=16) grow linearly with seq_len. They multiply.

mvout: 3,072 → 12,288 → 49,152

O(seq) × O(seq) = O(seq²) mvout storm

Interpretation: the local-store bypass successfully removes attention-matrix DRAM traffic. The failure is not the bypass idea; it is the combination of small Q-blocks and full-row Accumulator softmax.

Exp 1 at Scale: Slow, Then Crash

seq=256: Death by 147,456 Commands

6.68M

Baseline: 1.04M
6.4× slower in attention core

QK+PV compute is only ~1.47M.
The rest is dispatch overhead.

seq=512: 589,824 Commands = Crash

Crash @ 204M

TileLink assertion
Local-store path overstressed

The bypass is correct, but the control path
can't handle 590K fine-grained writes.

	seq=128	seq=256	seq=512
mvout/head	3,072	12,288	49,152
mvout total (12 heads)	36,864	147,456	589,824
Result	1.71× faster	6.4× slower	Crash

Same root cause: Q_BLOCK=16 + full-row softmax = O(seq²) mvout commands. No amount of tuning fixes the algorithmic scaling.

Experiment 1.2: What Changed in Software

Step 1: QK^T + SOFTMAX → Scratchpad

```
tiled_matmul(q_block, SEQ_LEN, HEAD_DIM,
    Q, K, NULL,
    (void*)(uintptr_t)GEMMINI_LOCAL_STORE_FLAG,
    // C is a local-store flag, not a DRAM address
    HIDDEN_DIM, HIDDEN_DIM, 0, SEQ_LEN,
    MVIN_SCALE_IDENTITY, MVIN_SCALE_IDENTITY, MVIN_SCALE_IDENTITY,
    SOFTMAX, ACC_SCALE_IDENTITY, 0,
    false,
    1, SEQ_LEN / DIM, QK_TILE_K,  // tile_J = full seq
    false, false, true,
    false, 0, WS);
gemmini_fence();
// Softmax output now lands in scratchpad
// No attention buffer is written to DRAM
```

Step 2: PV with A=NULL (SP weights)

```
tiled_matmul(q_block, HEAD_DIM, SEQ_LEN,
    NULL, V, NULL, O,    // A=NULL: use existing SP weights
    SEQ_LEN, HIDDEN_DIM, 0, HIDDEN_DIM,
    MVIN_SCALE_IDENTITY, MVIN_SCALE_IDENTITY, MVIN_SCALE_IDENTITY,
    NO_ACTIVATION, ACC_SCALE_IDENTITY, 0,
    false,
    1, PV_TILE_J, PV_TILE_K,
    false, false, false,
    false, 0, WS);
gemmini_fence();
// PV skips mvin of A; only final O is written out
```

Baseline vs Fused

	Baseline	Fused (Exp 1.2)
QK^T output dest	`C = attn_buf` (DRAM addr)	`C = (void*)GEMMINI_LOCAL_STORE_FLAG`
Softmax mvout goes to	DRAM (via DMA Writer)	Scratchpad SRAM (via local store wire)
PV A operand	DRAM (via mvin: attn_buf→SP)	Scratchpad (A=NULL, already there)
Attn matrix in DRAM	12 × 128² = 196 KB	0 — never leaves chip

Why the First Bypass Does Not Scale

Constraint 1: Built-In SOFTMAX Needs Full Rows

Why? Softmax is row-wise:

$$\text{softmax}(x_i) = \exp(x_i) / \Sigma_j \exp(x_j)$$

The denominator needs all elements in the row.

Partial J tiles do not have enough information.

`tile_J = SEQ_LEN / DIM` **for built-in softmax.**

What this means for Exp 1

The local-store bypass changes where softmax output goes, but not how the built-in softmax path is driven.

Softmax output → **Baseline: DRAM**

Softmax output → **Exp 1: Scratchpad**

Good	Bad
Avoids DRAM round-trip	Still many store commands
PV can use A=NULL	Still full-row softmax

How the Accumulator Does Softmax

Context: After QK^T matmul, the accumulator holds scores in int32. Built-in SOFTMAX runs inside the accumulator when data is read out via `mvout` — processing one row at a time, horizontally across columns.

One Row in the Accumulator (SEQ_LEN columns, int32)

N = SEQ_LEN / DIM = SEQ_LEN / 16 | Each J-tile = 16 columns processed in parallel by the accumulator lanes

3-Pass Horizontal Scan (per row)

1	Find max	Scan all J-tiles left→right, track max value
2	Exp + Sum	Re-scan: exp(x − max) per element, accumulate Σ
3	Normalize	Re-scan: exp / Σ → scale → write out as int8 weights

Three passes because the accumulator needs the full-row max before exp, and the full-row sum before normalize.

Why Full Row Is Non-Negotiable

$$\text{softmax}(x_i) = \exp(x_i - \max) / \Sigma_j \exp(x_j - \max)$$

The denominator Σ_j couples every column in the row. You cannot normalize J-tile 0 until you have scanned ALL J-tiles to compute the global sum.

Result: each mvout must process the entire row — 3 passes × N J-tiles — before any weight is written out.

Horizontally: softmax is a reduction (max, sum) across all columns, then element-wise normalize. The accumulator must traverse the entire row. Vertically: rows are independent — which is why Q_BLOCK lets us batch rows together. But horizontal dependency is the bottleneck that online softmax later fixes.

Constraint 2 in Detail: Q_BLOCK=16 Repeats the Full-Row Store

What happens after each QK^T tile?

Exp 1 computes only 16 query rows at a time:

```
Q_BLOCK = DIM = 16
scores tile = 16 rows × SEQ_LEN columns
```

After that tile finishes, the built-in SOFTMAX/local-store path still has to emit the normalized attention weights for those 16 rows across the entire J dimension.

Important: Q_BLOCK=16 is not the thing that creates mvout. The softmax output path creates the mvout-style store. Q_BLOCK=16 makes us repeat that store loop many times.

The loop that explodes

```
For each Q-block:
  QK^T produces 16 × SEQ_LEN scores

  For each row in the Q-block:        // 16 rows
    For each softmax/store pass:      // 3 passes
      For each J-block:               // SEQ_LEN / 16
        gemmini_extended_mvout(...)

mvout per Q-block = 16 × 3 × (SEQ_LEN / 16)
                  = 48 × (SEQ_LEN / 16)
```

Seq	Q-blocks/head	mvout/Q-block	mvout/head
128	8	384	3,072
256	16	768	12,288
512	32	1,536	49,152

The repeated full-row store is the problem. The bypass removes DRAM traffic, but the CPU still drives thousands of small RoCC store operations.

The Transition: Why We Can't Make J Smaller

Full-Row Softmax

$$\text{softmax}(x_i) = \exp(x_i - \max) \,/\, \Sigma_j \exp(x_j - \max)$$

The denominator Σ_j couples all K/J positions in the row.

Result: built-in softmax wants full-row processing.

Online Softmax

Keep running state per row:

m = running max
s = running sum

Each J-tile updates the state, so columns stream tile-by-tile (16 at a time).

Goal: stop materializing and re-driving the full attention row through many small commands.

Experiment 2: The Intuitive Way

Intuitive Way

- The Accumulator already computes softmax
- It already has exp / scale hardware
- So add more Accumulator commands for online softmax

What We Tried

- Add online update / commit commands
- Reuse Accumulator state IDs for running max / sum
- Reuse the mvout-style path to write weights

This proved the idea: online softmax can be implemented in Gemmini hardware. But it also exposed why modifying the Accumulator is not a clean final design.

Before Experiment 2: Accumulator Control Path

Key point: the Accumulator is driven by existing mvout/config command fields. It is not a standalone attention engine.

1. Software Issues Commands

`gemmini_extended_mvout(...)`

`gemmini_config_norm(..., stat_id)`

funct	rs1	rs2	config
RoCC op	src ptr	dst addr	scale / stat_id

→

2. Address / Config Decode

addr[24:0]	addr[27:25]	addr[...]
row / offset	NormCmd, 3 bits	flags / width

stat_id	scale args
0 or 1	accumulator config

→

3. Accumulator Pipeline

Input	Accumulator row
State slots	`NORM_STAT_IDS=2`
Compute	max → exp → sum → scale
Width	`DIM=16` lanes

What Experiment 2 would have to overload

Command Field

Original: 8 commands = 3 bits

Online: 13 commands = 4 bits

+

State Field

`stat_id=0/1`: existing Accumulator state slots

Experiment 2 tried to reuse one slot for online max / sum

+

Output Path

mvout to DRAM

Exp 1 local-store branch to scratchpad

So the Accumulator approach touches three shared mechanisms at once: command decode, state selection, and the existing output dataflow.

Why Not Keep Adding Accumulator Commands?

Problem 1: Command Encoding

Original NormCmd: 8 commands = 3 bits.
Add online softmax: +5 commands → 13 commands = 4 bits.
NormCmd packed into mvout addresses makes `NORM_CMD_SHIFT` fragile. If SW and RTL disagree, existing SOFTMAX / LayerNorm decoding can break.

Problem 2: State ID

Accumulator config: `NORM_STAT_IDS=2`.
Online softmax plan: reserve `stat_id=1` for running max / sum; use `stat_id=0` for default PV / mvin.
One wrong config can overwrite the online state mid-attention.

Problem 3: Fixed DIM vs Head Dimension

The Accumulator works at DIM=16 lanes. But each attention head has HEAD_DIM=64, so O-rescale becomes 4 commands per row.

Problem 4: Existing Dataflow

To make this work, we keep touching shared Accumulator / mvout / scratchpad behavior. That can change existing SOFTMAX, LayerNorm, residual, and scaling behavior.

Given that: we decided to design a dedicated OnlineAttention hardware module instead of overloading the Accumulator.

Two Tiling Dimensions: What Exp 1 Broke (and What We Fixed)

J-Tiling — The Real Bottleneck

Exp 1 kills

QK^T output is `[Q_rows × SEQ_LEN]`. Hardware grid is DIM=16, so J is split into `SEQ_LEN/DIM` tiles of 16 columns:

$$\text{seq=256} \rightarrow 256 \text{ cols} = 16 \text{ J-tiles of } \text{col}_{0..15},\ \text{col}_{16..31},\ \ldots,\ \text{col}_{240..255}$$

Built-in SOFTMAX needs all J at once for the denominator. Each J-tile → 1 mvout command. With Q_BLOCK=16:

16 rows × 3 passes × 16 J-tiles = 768 mvout per Q-block

K-Tiling — Separate Concern

Future

Splitting K means incomplete rows — the softmax denominator is partial. Built-in SOFTMAX can't handle this at all.

Online softmax can handle K-splitting via running max/sum + rescale, but for seq ≤ 512 we never need it:

K_BLOCK = SEQ_LEN (single-pass)

K-tiling matters for seq ≥ 1024 — future work.

What Online Softmax Actually Fixes (for Us)

Exp 1 failed because J-tiling × small Q_BLOCK = O(seq^2) RoCC dispatch. OnlineAttention's BATCH_WEIGHTS streams all J-tiles internally with running max/sum — one RoCC call per Q-block regardless of seq_len. K_BLOCK = SEQ_LEN eliminates K-tiling entirely (single pass).

Online Softmax: Streaming Across J-Tiles

Even with K_BLOCK=SEQ_LEN, hardware processes only DIM=16 columns at a time. Online softmax streams J-tiles with running stats:

m = running max **s = running sum**

New J-tile arrives (16 columns at a time):
Check if max increased. If yes → rescale old sum before adding new terms.

Example: 3 J-tiles, one row (12-element row, DIM=4 for simplicity)

Step	Scores (4 cols)	m	s
Init	—	−∞	0
J-tile 1	[3, 1, 4, 1]	4	Σexp(s−4)
J-tile 2	[7, 2, 1, 3]	7	s×exp(4−7) + Σexp(s−7)
max ↑	old sum rescaled by exp(old_max − new_max)		
J-tile 3	[2, 5, 1, 2]	7	s + Σexp(s−7)
max =	no rescale — just add new exp terms		

After all J-tiles: weight = exp(score − m) / s. BATCH_WEIGHTS does this for all rows + all J-tiles in one RoCC command.

Gemmini Reality: INT Scores, INT Weights

Design choice	Why
Scores are int32	Systolic array outputs int32. QK^T scores land directly — no FP conversion.
Weights are int8	Scratchpad stores int8. PV reads A=inputType=int8. Output must fit in [0, 127].
No FP math library	Baremetal C on RISC-V. No `expf()`. Hardware computes exp() in fixed-point combinational logic.
Scale before iexp	`ONLINE_BERT_SCALE=0.05` compresses scores so fixed-point exp doesn't saturate.
Saturation = zero weight	$z \geq 32 \rightarrow \text{iexp}(z) = 0$. Very negative → ~0. Correct softmax behavior.

Not FP32:

Papers assume FP32. In hardware, number formats come from the datapath.

Every op — multiply, max, exp, sum, normalize — is fixed-point int32.

Hardware exp() Without Floating Point

Computes $2^{ax^2 + bx + c}$ — second-order polynomial approximation of exp():

One iexp unit: `x` (int32) → `ax² + bx + c` → 2^{result} (int32)

Config register	Purpose	Set via
`iexp_qln2_reg`	ln(2) in fixed-point	`gemmini_oa_config(0, ...)`
`iexp_qln2_inv_reg`	1/ln(2) for base conversion	`gemmini_oa_config(1, ...)`
`iexp_qb_reg`	Coefficient b	`gemmini_oa_config(2, ...)`
`iexp_qc_reg`	Coefficient c	`gemmini_oa_config(3, ...)`

Key design choices:

16 parallel units — one per DIM lane. One cycle per chunk.

Reuses AccumulatorScale.iexp() — already validated for Gemmini softmax.

z ≥ 32 → output 0 — hardware saturation = near-zero weight.

Runtime configurable — tune coefficients without rebuilding Verilator.

The only compute in OnlineAttention. Everything else is state management and data movement.

OnlineAttention: What the Hardware Actually Does

OnlineAttention.scala — ~700 lines of Chisel. One module, one RoCC function (funct=23), one 7-state FSM. Lives between accumulator and scratchpad.

INPUT

Accumulator

QK^T scores
int32 matrix
up to 128×512

→

COMPUTE (7-state FSM)

① Read 16 int32 scores from accumulator
② 16-way max reduction → chunk_max
③ Update running max / sum (rescale if max increased)
④ Re-read scores, 16 × iexp(score − max) in parallel
⑤ Clamp to [0,127], pack 16 × int8

→

OUTPUT

Scratchpad

int8 softmax weights
Ready for PV matmul
A=NULL

Phase 1 — UPDATE (track max & sum)

Streams J-tiles (16 columns at a time). Per row: tracks running max *m* and sum *s*. When max increases, rescales old sum by exp(old_max − new_max). No full-row wait — columns arrive tile-by-tile.

Phase 2 — WEIGHTS (write to scratchpad)

Re-reads same scores. Subtracts final max. 16 parallel iexp units. Clamps to [0,127], packs 16×int8, writes to scratchpad. One cycle per 16-element chunk.

Why re-read? Score matrix up to 512 KB — too big to buffer. Re-reading is free: accumulator is idle during `gemmini_fence()`. | State: 3 KB registers (128 rows × 3 × int32) | iexp: 16 parallel combinational units, reuses AccumulatorScale.iexp()

Online Softmax in Action: One Row, Three Chunks

Concrete example: one Q row, 48 K positions streamed as three 16-element chunks. Two numbers of state per row.

Running max

m starts at −∞

Running sum

S starts at 0

Chunk	16 scores (conceptual)	chunk_max	m update	s update
1	[3, 1, 4, 1, ..., 9, 7, 9, 3]	9	−∞ → 9	s = Σ exp(score − 9) (first chunk, compute normally)
2	[2, 11, 1, 8, ..., 9, 2, 6, 5]	11	9 → 11	s = s_old × exp(9−11) + Σ exp(score − 11) (max increased: rescale old sum first!)
3	[3, 5, 8, 10, 7, 9, 3, ..., 6, 4, 3]	10	11 (no change)	s += Σ exp(score − 11) (max unchanged: just add new terms)

Final: all chunks done. Weights = iexp(score − 11) / s_final, clamped to [0,127] int8, written to scratchpad. (Division folded into PV matmul or post-pass rescale)

Why this works: exp(a−m) / Σexp(...) = exp(a−m') / Σexp(...) when rescaling by exp(m−m'). The softmax value for each element doesn't change — only the running denominator gets corrected.

INT reality: We use fixed-point iexp (2^{ax^2+bx+c}). Scores are int32 from systolic array. Weights are int8 for scratchpad. No floating point anywhere.

Side by Side: Baseline vs Experiment 4

Both do the same math — QK^T + softmax + PV. The difference is where data lives.

Baseline Gemmini Attention

```
// Step 1: QK^T → DRAM (via C pointer)
tiled_matmul_auto(SEQ_LEN, SEQ_LEN, HEAD_DIM,
  A, B, NULL, C=attn_buf, // → DRAM
  HIDDEN_DIM, HIDDEN_DIM, 0, SEQ_LEN,
  ..., SOFTMAX, ACC_SCALE_IDENTITY, 0, ...);
gemmini_fence();

// Step 2: PV (weights from DRAM)
tiled_matmul_auto(SEQ_LEN, HEAD_DIM, SEQ_LEN,
  A=attn_buf, B=V_buf, NULL, C=out_buf,
  SEQ_LEN, HIDDEN_DIM, 0, HEAD_DIM,
  ..., NO_ACTIVATION, ...);
gemmini_fence();
```

✗ **Attention matrix A written to DRAM, then read back**

Experiment 4 (OnlineAttention)

```
// Step 1: QK^T → accumulator (scores stay on chip)
gemmini_loop_ws(I, J, K, 0, 0, 0,
  Q_ptr, K_ptr, NULL, NULL, // no DRAM out
  ..., ex_accumulate=true, ...);
gemmini_fence();

// Step 2: Online softmax acc → scratchpad
gemmini_oa_fused_batch(
  Q_BLOCK, SEQ_LEN,
  sp_weights_base, acc_addr);
gemmini_fence();

// Step 3: PV (weights from scratchpad)
gemmini_loop_ws(I, J, K, 0, 0, 0,
  NULL, V_ptr, NULL, O_ptr, // A=NULL=SP
  ..., ex_accumulate=true, ...);
gemmini_fence();
```

✓ **Attention matrix stays on chip: accumulator → scratchpad directly**

What disappeared: `attn_buf[heads][SEQ][SEQ]` **— the O(N²) DRAM buffer. No** `SOFTMAX` **activation flag. No separate weight MVIN/MVOUT.**

What appeared: `gemmini_oa_fused_batch()` **— one RoCC command.** `A=NULL` **in PV tells hardware to read weights from scratchpad. Scores live in accumulator between QK^T and softmax.**

Where the Bytes Go: Data Movement Comparison

Baseline — DRAM Round-Trip

① QK^T → acc → DRAM (mvout scores)
② DRAM → acc (mvin scores)
③ SOFTMAX acc → DRAM (mvout weights)
④ DRAM → sp (mvin weights)
⑤ PV → DRAM (output)

2 DRAM round-trips for O(N²) attention matrix

Experiment 4 — On-Chip Only

① QK^T → Accumulator (on-chip)
② Acc → OnlineAttn HW → Scratchpad
③ SP weights → PV (A=NULL)

0 DRAM transfers for attention matrix

Baseline memory footprint (per head, seq=512)

attn_buf
512×512 int8
256 KB DRAM

Experiment 4 memory footprint (per head, seq=512)

Accumulator int32 scores on-chip → Scratchpad int8 weights on-chip

seq=512, 12 heads: Baseline ~6.3 MB DRAM for A matrix vs Exp 4 0 MB

Data-movement fusion, not compute fusion. QK^T, softmax, PV still separate ops — we eliminated the DRAM intermediaries. Only Q, K, V, O (all O(N)) touch DRAM.

End-to-End: 3 Commands Per Head

1. QK^T matmul → 2. OP_FUSED_BATCH → 3. PV matmul

Attention matrix never touches DRAM. Scores stay in accumulator. Weights go accumulator → scratchpad directly.

Experiment 5 — Clean Measurement (config outside timing, zero printf)

Full pipeline results

Seq	Q_BLOCK	QKV	Attention	Wo	Norm	Total
128	128	913K	245K	309K	164K	**1,631K**
256	256	1,804K	486K	609K	369K	**3,269K**
512	128	3,582K	1,882K	1,204K	759K	**7,427K**

seq=128: QKV 56%, Attn 15%, Wo 19%, Norm 10%

seq=512: QKV 48%, Attn 25%, Wo 16%, Norm 10%

Attention grows from 15% to 25% of total — $O(N^2)$ in action.

Optimization Journey: 30.2M → 1.63M (18.5× Improvement)

Six key phases. Each attack a different bottleneck. All numbers at seq=128 for fair comparison.

Phase	What Changed	Total	vs Baseline
Baseline	Official Gemmini: QK^T + SOFTMAX + PV, attn matrix through DRAM	2,149K	1.00×
Exp 1.2	Local-store fusion. Q_BLOCK=16. No online softmax. Works at 128, fails at scale	1,882K	1.14×
4G→4H	Per-row ops → batch ops. 242→9 RoCC/K-block (16× reduction). First integrated pipeline	30,244K→15,353K	0.07→0.14×
4I	K_BLOCK=128 (single-pass). Eliminate K-block loop + RESCALE. Online softmax proves correct	6,995K	0.31×
4L→4M	Q_BLOCK=128. Single Q-block per head. Remove CPU from critical path. DRAM internal MVIN	3,103K→1,793K	0.69→1.20×
Exp 5	Optimal Q_BLOCK (sweep-verified per seq). OP_FUSED_BATCH. Clean measurement (zero printf)	1,631K	1.32×

Three biggest leaps:
1. Per-row → batch ops: 16× RoCC reduction (4G→4H)
2. K_BLOCK=128: no K-loop, no RESCALE (4H→4I)
3. Q_BLOCK=128 + CPU removal: 8× fewer matmuls (4I→4M)

Exp 5 pipeline (seq=128):

QK^T → OP_FUSED_BATCH → PV (A=NULL)

3 RoCC commands per head. Zero CPU extraction. Zero DRAM for attn matrix.

What is Q_BLOCK — and Why It Matters

QK^T produces a [seq_len × seq_len] score matrix.
Q_BLOCK = how many Q rows we process per batch.

Q [seq×64] × K^T [64×seq] = Scores [seq×seq]

↑ Q_BLOCK rows per batch ↑

Each Q-block does: QK^T → online softmax → PV. More Q-blocks = more RoCC commands = more dispatch overhead.

Why only sweep Q_BLOCK? Online softmax handles K streaming — K_BLOCK = SEQ_LEN always. Only Q_BLOCK is a free variable.

Q_BLOCK	Q-blocks @ seq=256	RoCC cmds/head	ACC tiles needed
16	16	48	4
64	4	12	16
128	2	6	32
256	1	3	64 (full half)

Small Q_BLOCK (e.g. 16)

Scores fit easily in ACC (4 tiles). But many RoCC commands — each with dispatch overhead. Exp 1: Q_BLOCK=16 → 384 matmuls at seq=256 → 6.4× slower.

Large Q_BLOCK (e.g. 256)

Few RoCC commands (3/head). But ACC overflow: scores exceed one ACC half → DMA spill to DRAM. At seq=512: 8× overflow → tipping point.

The trade-off: RoCC dispatch overhead (small Q_BLOCK) vs ACC overflow DMA (large Q_BLOCK). Optimal Q_BLOCK depends on seq_len and ACC capacity.

Q_BLOCK Sweep: Finding the Optimal per Sequence Length

Experiment 5 sweeps Q_BLOCK at each seq_len. Clean measurement — zero printf, config outside timing. K_BLOCK = SEQ_LEN always.

seq=128 (K_BLOCK=128)			
Q_BLOCK	Q-blocks	Attention	vs Best
16	8	432K	+79%
64	2	262K	+7.5%
128	1	244K	—

seq=256 (K_BLOCK=256)			
Q_BLOCK	Q-blocks	Attention	vs Best
16	16	1,338K	+175%
128	2	512K	+5.4%
256 ★	1	486K	—

seq=512 (K_BLOCK=512)			
Q_BLOCK	Q-blocks	Attention	vs Best
16	32	3,492K	+92%
128	4	1,825K	—
256	2	1,883K	+3.2%

U-curve confirmed: Q_BLOCK too small → dispatch overhead dominates. Q_BLOCK too large → ACC overflow DMA dominates.

seq=256: Q=256 optimal (not 128!). `onlineAttentionMaxRows=256` enables single Q-block without sub-block splitting.

seq=512: Q=256 hits **tipping point** — 8× ACC overflow DMA exceeds dispatch savings. Q=128 remains optimal.

Why can't we go further?
ACC half = 1024 rows × DIM col = 64 tiles (WS double-buffering).

Q=256, K=512 → 512 tiles → 8× overflow
Q=512, K=512 → 1024 tiles → 16× overflow

maxrows upgrade alone isn't enough. Need larger ACC (e.g. ACC_ROWS=4096) to unlock Q_BLOCK=256 at seq=512.

Final Results: Baseline vs Exp 1 vs Exp 5

Experiment 5 — clean measurement (zero printf, config outside timing, OP_FUSED_BATCH). Optimal Q_BLOCK per seq_len.

Seq	Experiment	QKV	Attention	Wo	Norm	Total	vs Baseline
128	Baseline	960K	611K	309K	188K	2,149K	1.00×
128	Exp 1 (local store)	962K	358K	309K	176K	1,882K	1.14×
128	Exp 5	913K	245K	309K	164K	1,631K	1.32×
256	Baseline	1,851K	1,038K	610K	395K	3,973K	1.00×
256	Exp 1 (local store)	~1,851K	6,681K	~610K	~395K	~9,536K	0.42×
256	Exp 5	1,804K	486K	609K	369K	3,269K	1.22×
512	Baseline	3,629K	2,623K	1,204K	801K	8,336K	1.00×
512	Exp 1 (local store)	CRASH @ 204M cycles — TL assertion					
512	Exp 5	3,582K	1,882K	1,204K	759K	7,427K	1.12×

Scaling (vs self @ seq=128)

	128→256	256→512
Baseline	1.85×	2.10×
Exp 1	5.07×	CRASH
Exp 5	2.00×	2.27×

Exp 5 wins at ALL sequence lengths.
Scaling is clean O(N) — pure hardware with matmul dimensions.
Works at seq=512 where Exp 1 crashes.
Attention matrix never touches DRAM (0 bytes vs up to 6.3 MB baseline).

Contributions

What We Improved

Metric	Baseline	Exp 5	Improvement
seq=128 Total	2,149K	1,631K	−24.1%
seq=256 Total	3,973K	3,269K	−17.7%
seq=512 Total	8,336K	7,427K	−10.9%
Attn matrix DRAM	O(seq²) per head	Zero	Eliminated
Scalability	O(seq²) DRAM	O(seq) streaming	Arbitrary seq

Hardware Cost (Modest)

OnlineAttention.scala	~500 lines Chisel
Controller + Scratchpad changes	~85 lines
State registers	256 × 3 × int32 = 3 KB
iexp units	16 parallel combinational (reused from AccumulatorScale)
RoCC functs	23 (cmd) + 24 (reserved for macro-sequencer)

Key Deliverables

- Functional Verilator simulation environment
- Baseline + Exp 1 profiling (all seq lengths)
- OnlineAttention HW module (~500 lines Chisel)
- Online softmax with int32/int8 fixed-point math
- OP_FUSED_BATCH: single-command attention softmax
- 10 critical HW bugs documented and fixed
- Joint HW/SW design-space exploration
- Q_BLOCK sweep analysis across 3 seq lengths

3 design lessons:
1. Q-block granularity is king — dispatch overhead dominates compute.
2. Fused dataflow was correct from day one — the softmax mechanism was the bottleneck.
3. Dedicated HW > overloading existing modules — clean separation of concerns.

Future Work

1. Macro-Sequencer HW

Chain QK^T → softmax → PV into one hardware-triggered sequence

- Eliminate per-command RoCC dispatch — currently ~O(Q_BLOCKS) RoCC calls per head
- FSM auto-launches next phase when accumulator ready; no CPU in the loop
- RoCC funct=24 already reserved; estimated ~200 lines Chisel

2. Multi-Layer Fusion SW+HW

Fuse attention output directly into FFN input

- Wo output → FFN: skip DRAM write + DRAM read of intermediate tensor
- Save O(seq × d_model) DRAM traffic per transformer layer
- Extends data-movement fusion across layer boundaries (not just within attention)

3. Data-Movement Fusion Framework SW

Generalize the pattern to any producer → consumer pair

- Formalize "local-store pipeline": detect consecutive ops, route through scratchpad
- Apply to: LayerNorm→FFN, FFN→LayerNorm, residual add paths
- Compiler-driven hint: `GEMMINI_LOCAL_CHAIN` flag on output tensor

4. Flash Attention on Systolic Array HW+Algo

Map Flash Attention tiling onto Gemmini's systolic array

- Block Q, K, V into SRAM-sized tiles; compute softmax incrementally per tile
- Leverages OnlineAttention's running max/sum state as the core building block
- Enables seq=1024+ with O(seq) memory — breaks the ACC capacity ceiling