

DySpin: A Plug-and-Play Library Advancing Dynamic Sparse Long-Context Inference

Chengyu Sun · Yaqi Xia · Ruirui Pan · Donglin Yang · Xiaobo Zhou · Dazhao Cheng
Wuhan University · Central China Normal University · NVIDIA · University of Macau

The context window is becoming the working set

As context grows, the "state" of inference grows with it.

- **Query** — one new token
- **Read KV** — history grows
- **Attention** — QK → softmax → PV
- **Next token** — autoregressive loop

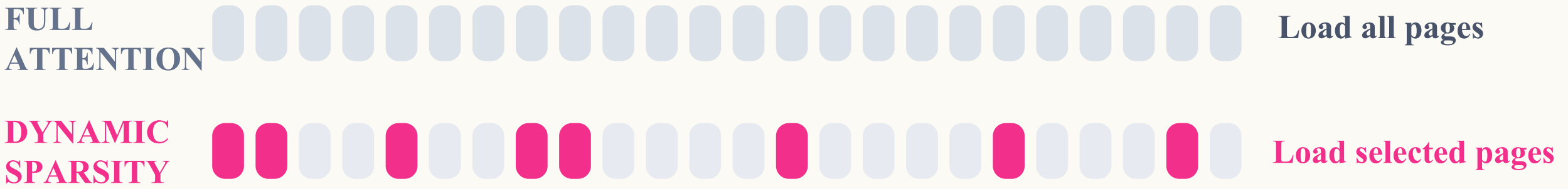
HBM → SM: K/V pages streamed every decode step

Long-context decode → low arithmetic intensity → **bandwidth-bound** → **latency scales with bytes moved**

HPC lens: the optimization target shifts from "do fewer FLOPs" to "move fewer bytes."

BACKGROUND: Dynamic Sparse Attention

The key idea: score cheaply, fetch only what matters.



1. Estimate importance

Compressed K → approximate page scores

→

2. Select pages

Top-K retrieval

→

3. Sparse attention

Load selected KV and compute attention

Dynamic sparsity solves bandwidth—not capacity

A paradox of recallable sparsity

Bandwidth win

Only a subset of K/V pages is loaded for each query.

Good: fewer HBM bytes per decode step.

Capacity loss

All pages remain resident because future queries may retrieve them.

Bad: memory footprint still grows with context.

CPU offload?

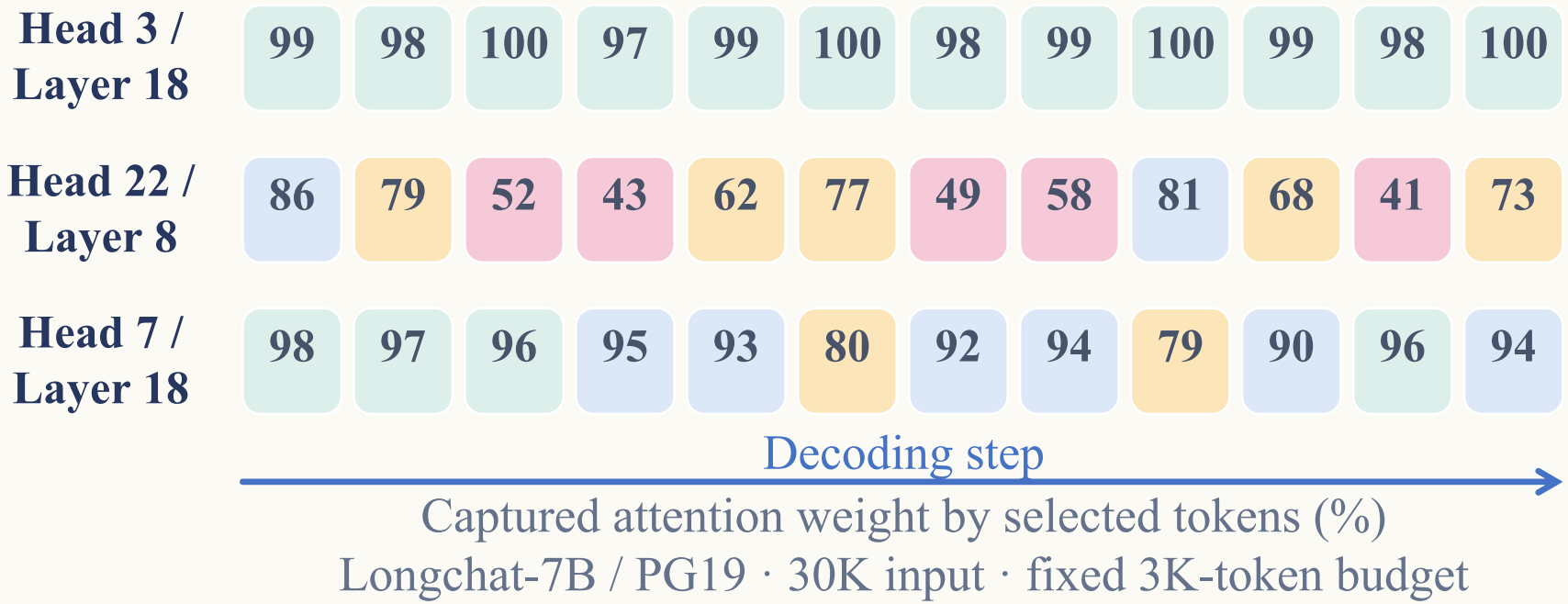
It avoids OOM, but shifts the bottleneck to PCIe.

Pure-GPU execution is still desirable.

DySpin asks: can we preserve dynamic recallability while compressing the resident state?

A fixed token budget ignores runtime heterogeneity

Different heads need different amounts of context—and that demand changes over time.



Over-allocate

Concentrated heads waste bandwidth.

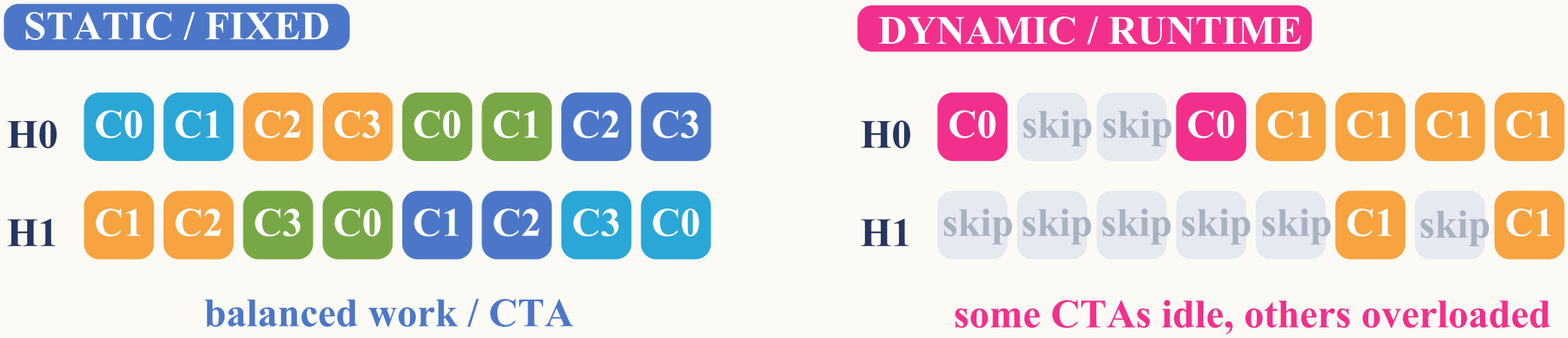
Under-allocate

Diffuse heads lose critical tokens.

One-size-fits-all Top-K is simultaneously wasteful and accuracy-limiting.

Adaptivity is good for accuracy—but hard for GPUs

Dynamic allocation turns regular sparse attention into an irregular scheduling problem.



The systems problem is no longer "which tokens?" only.
It is "how do we map a changing set of tokens onto a massively parallel GPU?"

Co-design the algorithm, the runtime, and the representation

1 Dynamic allocation

Per-head, per-step threshold allocation
→ preserve critical tokens

→

2 Fast execution

FastEstimate + load-balanced sparse attention
→ regularize irregular work

→

3 Compact state

6-bit KV cache + FusedRoPE
→ reduce capacity and preprocessing cost

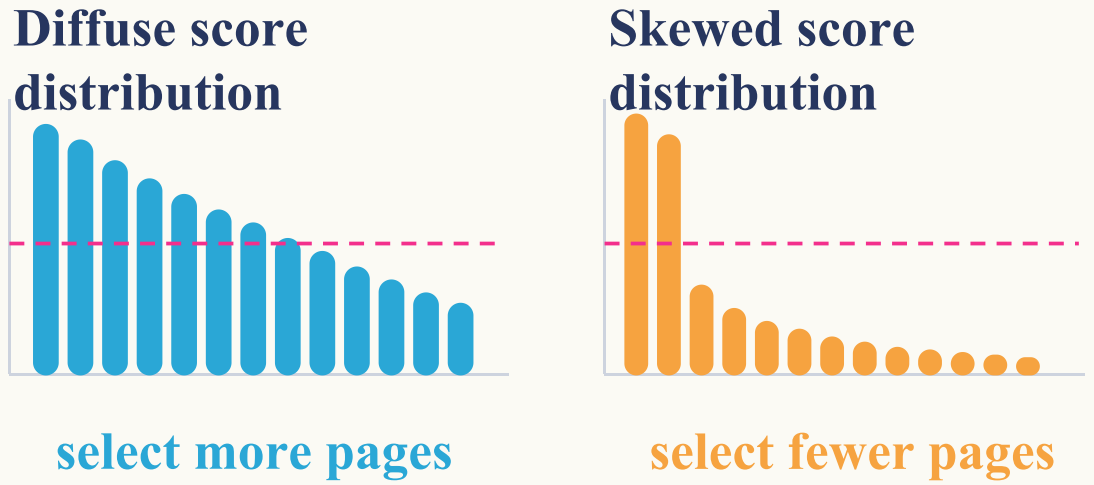
Runtime allocation: budget follows demand

Replace “select exactly K pages” with “select every page above a head-specific threshold.”

$$P_{selected} = \{\, p \in P \mid S(p) > \tau \,\}$$

$$\tau = S_{max} - \theta\,(S_{max} - S_{min})$$

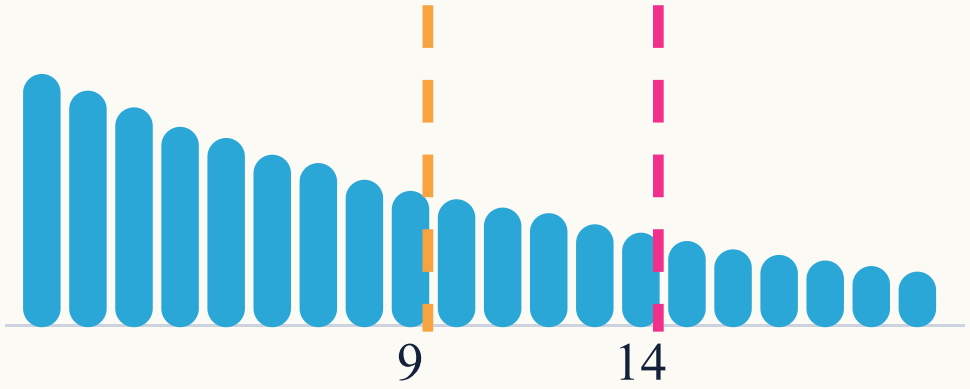
θ controls how aggressively each head expands its budget (Larger θ → lower τ → more selected pages).



What changes when the budget becomes dynamic?

Same total page reads: 9 + 9 = 14 + 4 = 18.
Reallocate page reads from concentrated heads to heads requiring broader context.

Head A — broad importance distribution

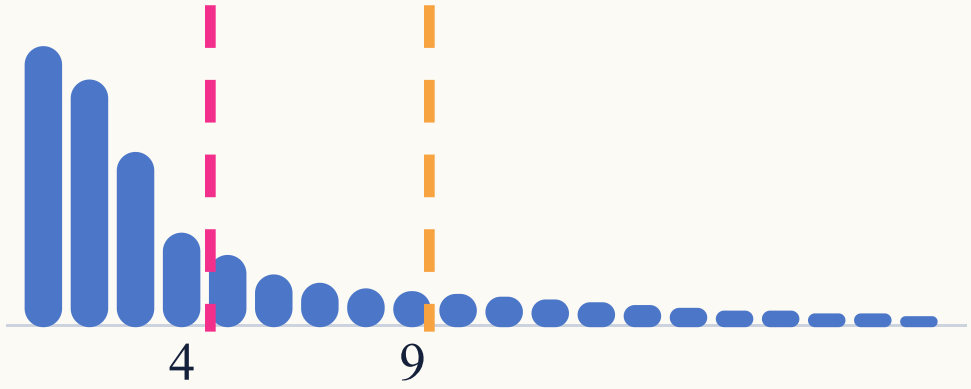


fixed K

dynamic

More pages when evidence is distributed.

Head B — a few dominant pages



Fewer pages when a small set dominates.

What matters is recall—not a fixed token count

DySpin optimizes “how much important context did we recover?” rather than “did every head get K pages?”



Critical target set: minimum top-ranked tokens whose cumulative attention weight reaches 95%

88.5% of heads achieve >95% recall

Recall target defined by the 95% cumulative attention-weight set.

97.7% of heads beat fixed allocation

Compared head-by-head under the paper’s Longchat/PG19 evaluation.

FastEstimate: one fused kernel, four concrete steps

1 Score pages

Q0 score the same 20-page range owned by each CTA.

2 Find threshold

Each CTA reduces local min/max; grid sync produces global Smin/Smax and τ.

3 Compact pages

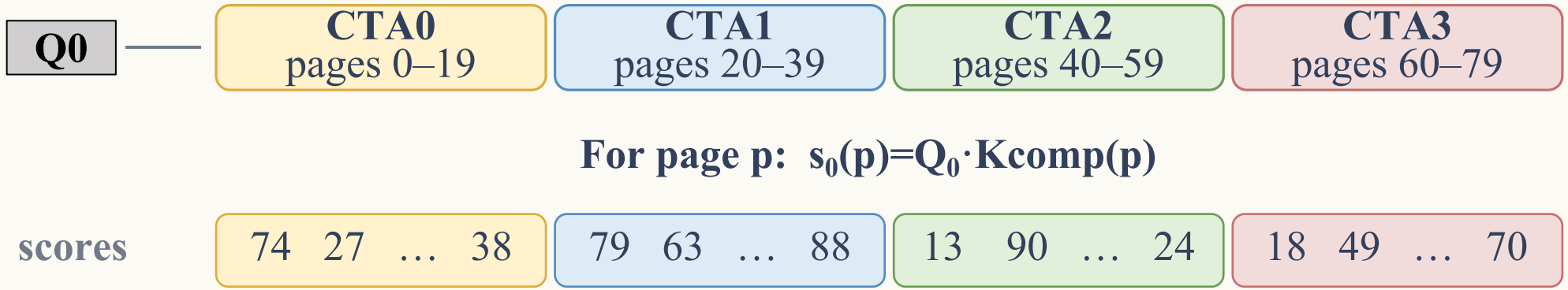
$S(p)>\tau$ becomes a binary mask; prefix sums compute compact write positions.

4 Emit schedule

Write selected page IDs plus $P_{head,i}$ and P_{all} for the attention kernel.

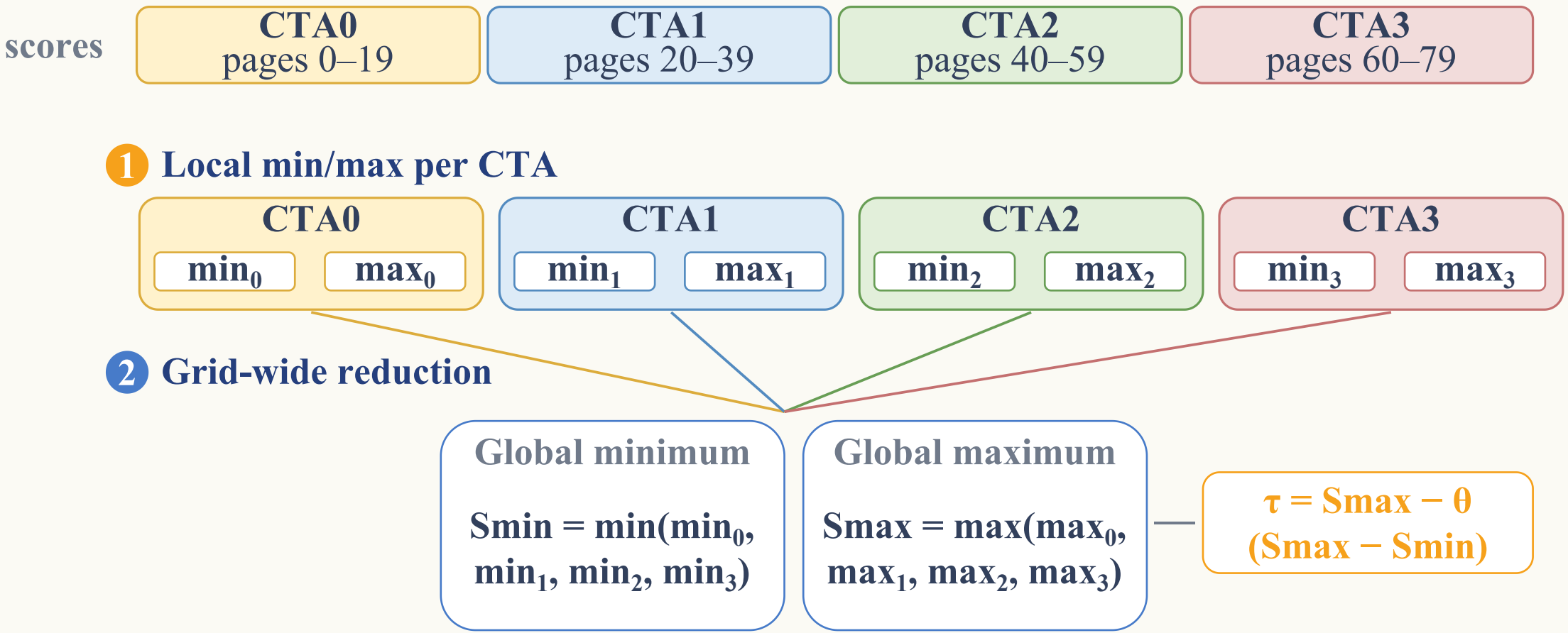
FastEstimate (1/4): page scoring

80 candidate pages = 4 equal CTA ranges × 20 pages



Step-1 output: S[0…79], still partitioned into four 20-page CTA ranges.

FastEstimate (2/4): local extrema → global threshold



FastEstimate (3/4): threshold → mask → prefix sum

Thresholding + prefix-sum replaces explicit Top-K/sorting with linear compaction.

CTA3 owns pages 60–79 (20 candidates). A page survives if $S(p) > \tau$.

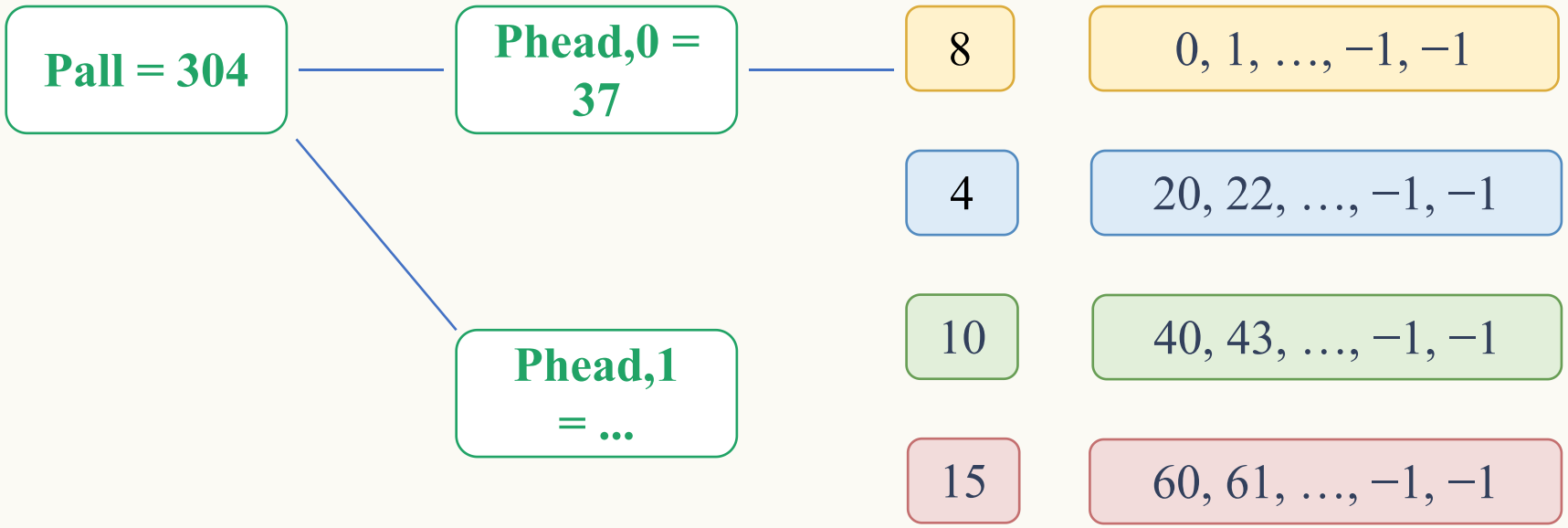
Page ID	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
Mask	1	1	0	1	1	1	1	0	1	1	1	0	1	1	0	1	1	1	0	1
Prefix sum	1	2	2	3	4	5	6	6	7	8	9	9	10	11	11	12	13	14	14	15

1. **Mask** 1 = selected, 0 = rejected.
2. **Prefix sum** Gives each selected page its compact write position.

Compact CTA3 list: 60, 61, 63, 64, 65, 66, 68, 69, 70, 72, 73, 75, 76, 77, 79

FastEstimate (4/4): flattened output

Selected page IDs are stored in CTA-colored chunks; counts guide the attention kernel.

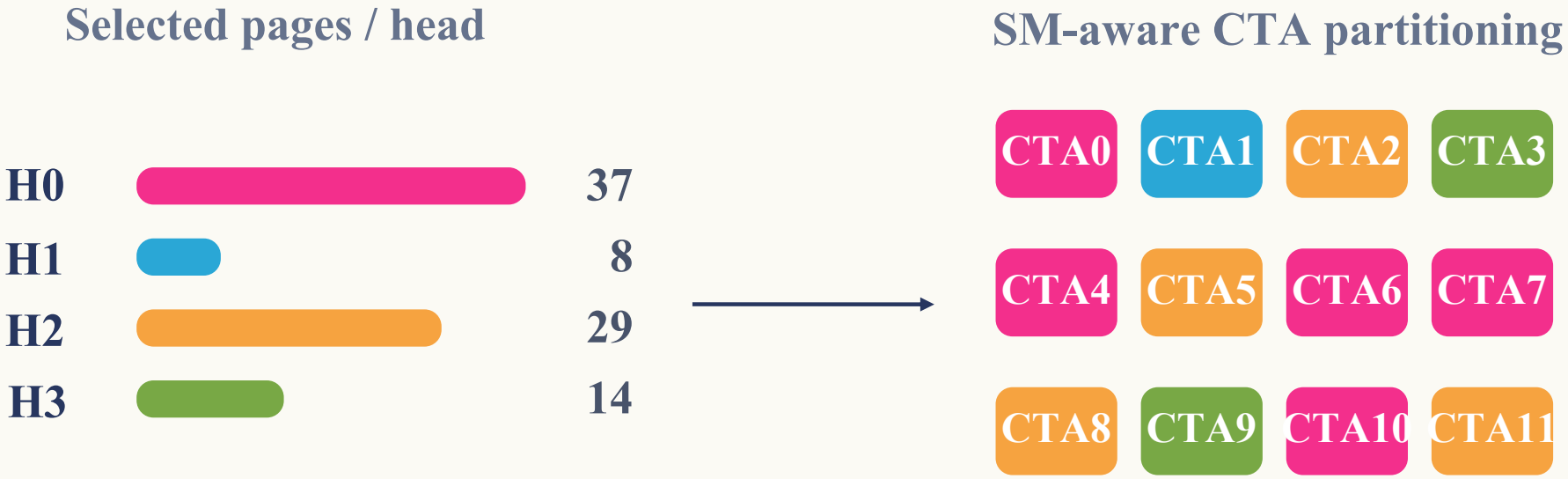


Why "−1"?

−1 is the sentinel that tells the attention kernel to jump to the next chunk.

From irregular demand to balanced CTAs

FastEstimate turns per-head variability into a compact scheduling list.



Each CTA receives a similar number of selected pages.

FastEstimate is designed as a lightweight producer

Three details keep selection overhead small on the decoding critical path

1. Fusion

Importance evaluation and selection are merged into one kernel, eliminating separate Top-K and planning passes.

2. Local communication

Intermediate block data stays in shared memory; only small global tallies require atomics.

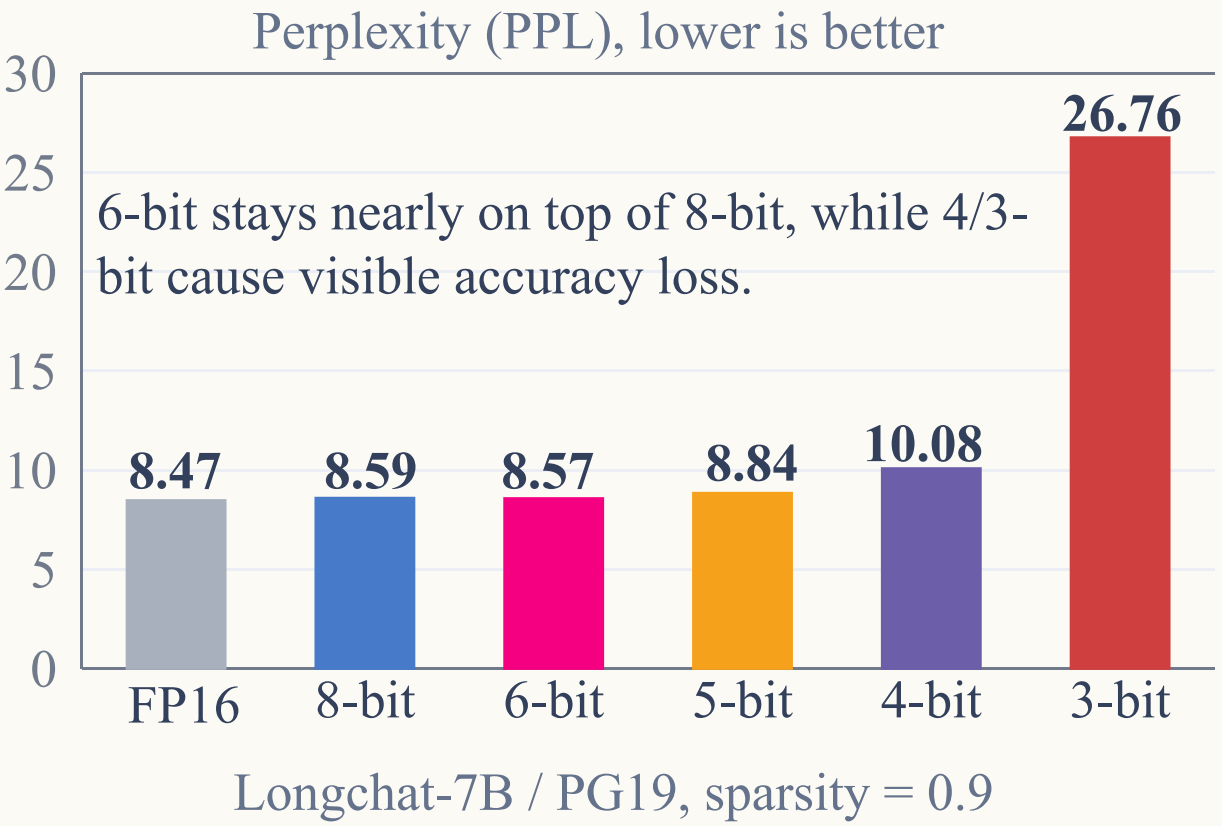
3. Reusable metadata

The compact output array has a statically known maximum size and is pre-allocated across decoding steps.

Dynamic sparsity reduces traffic. Quantization reduces the resident state.

Why 6-bit? An empirical accuracy–memory sweet spot

Lower precision is only useful if sparsity and quantization do not compound each other's error.



+0.10

PPL vs FP16

6-bit at 90% sparsity

−25%

vs 8-bit storage

Raw quantized code width

The non-power-of-two problem: packing vs dequantization

Compression only helps when the compressed representation is cheap to consume.

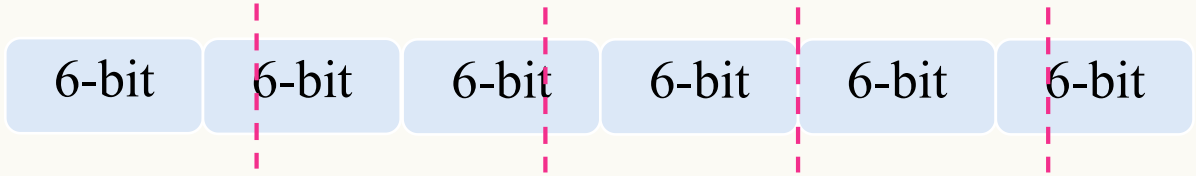
8-BIT ALIGNED



easy byte loads / casts

but 2 bits wasted per value

NAÏVE COMPACT



no space wasted

but values cross byte / word boundaries

DySpin's goal **dense 6-bit storage** + **parallel register-level unpack** + **fast FP16/BF16 conversion**

DySpin packing: 16 × 6-bit values → 3 × 32-bit words

Representation becomes part of the kernel design:
the physical bit layout is chosen for 4-way register-level extraction.

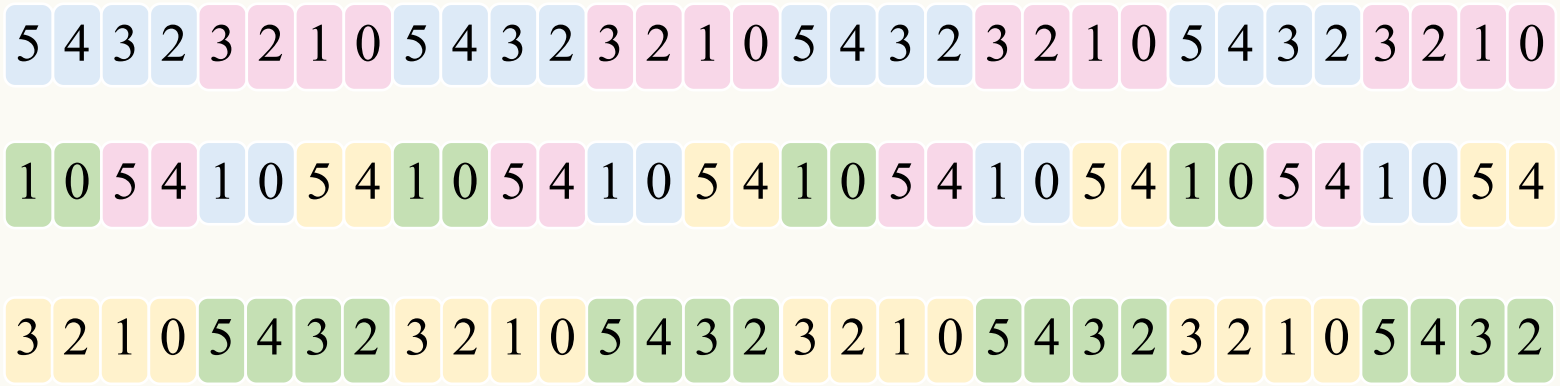
16 quantized values

0 1 2 3 4 5 6 7

8 9 10 11 12 13 14 15

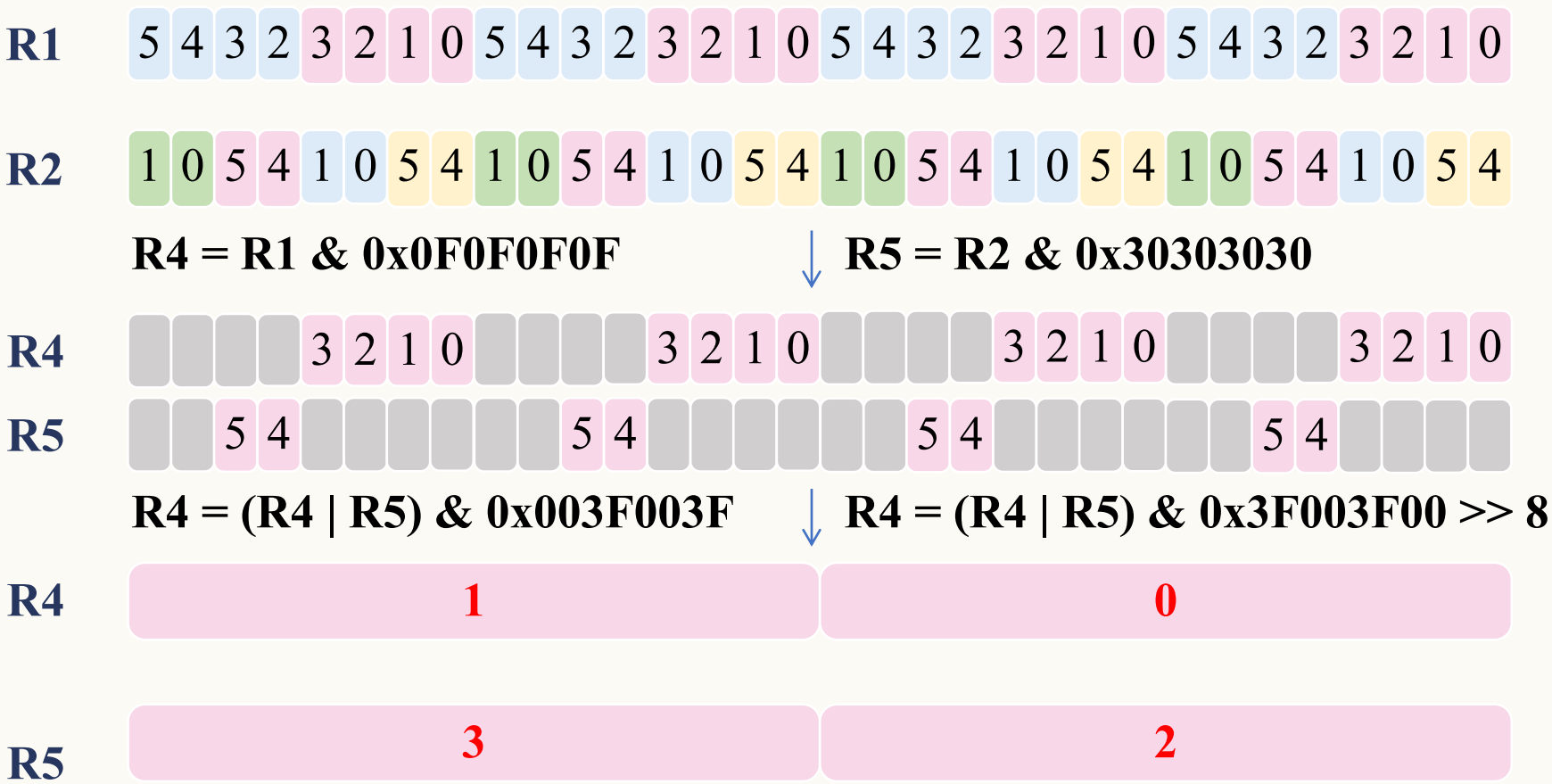
0–3 & 12–15: split 4+2 bits
4–11: split 2+4 bits

Three 32-bit registers



16 × 6 = 96 bits exactly: no padding. The layout is optimized for how the kernel unpacks, not for sequential bit order.

4-way register-level unpack + dequantization



A few masks and ORs reconstruct four INT6 codes in parallel—without scalar per-value extraction

4-way register-level unpack + dequantization

R4	1	0
R5	3	2

R4 = R4 | 0x64006400 ↓ **R5 = R5 | 0x64006400**

R4	[1] + 1024	[0] + 1024
R5	[3] + 1024	[2] + 1024

Set exponent bits directly to construct FP16 values—no integer-to-float conversion instruction.

4-way register-level unpack + dequantization

R4	[1] + 1024	[0] + 1024
R5	[3] + 1024	[2] + 1024

__hsub2()

↓

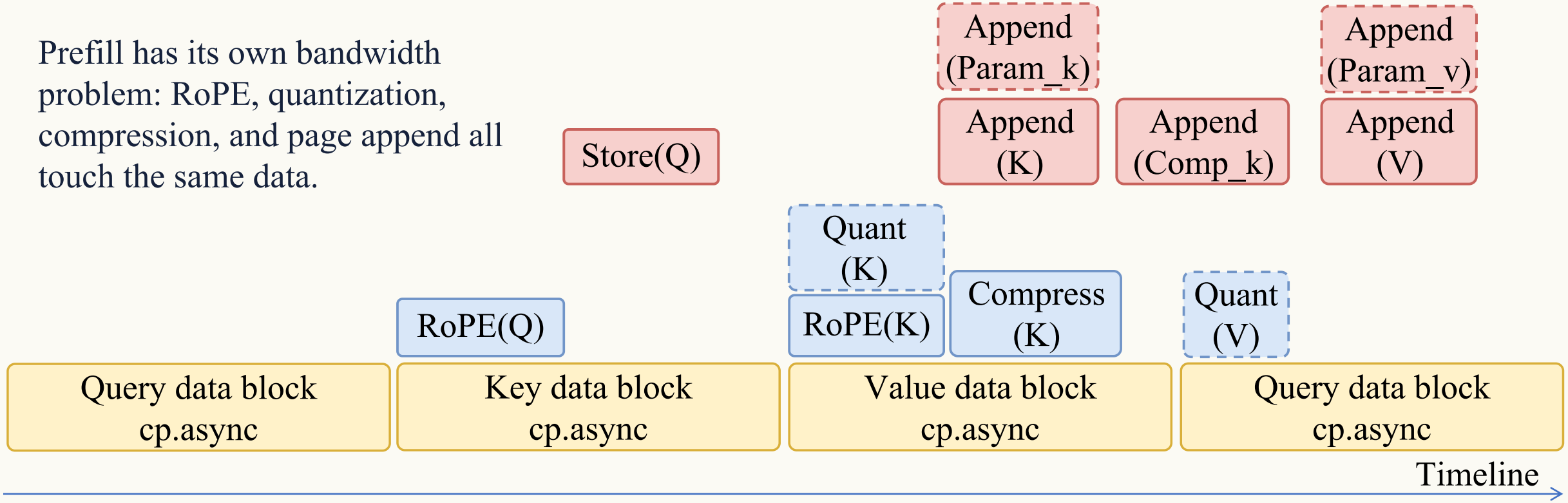
__hfma2()

R4	[1] * s + z	[0] * s + z
R5	[3] * s + z	[2] * s + z

Subtract the 1024 bias, then fuse scale and zero-point with half2 arithmetic.
BF16 uses an analogous +128 construction; 6-bit values naturally fit its valid conversion range.

FusedRoPE: one memory pass for preprocessing

Prefill has its own bandwidth problem: RoPE, quantization, compression, and page append all touch the same data.



Why it works

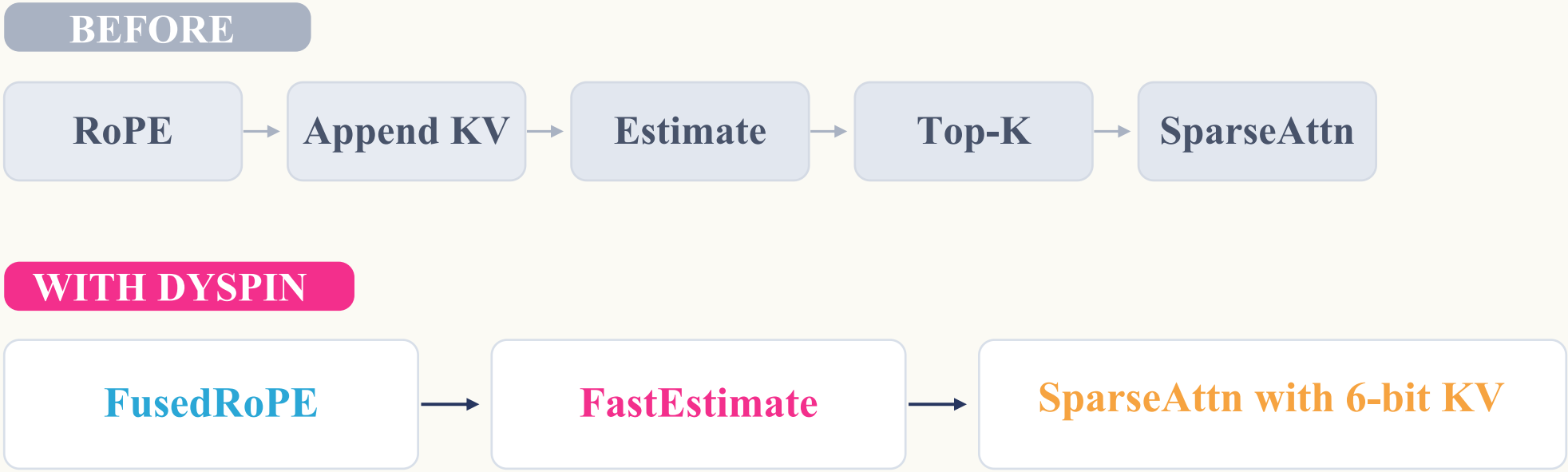
cp.async overlaps global-memory movement with register-level work.

Why it matters

One fused pass avoids repeatedly reading/writing the same prefill data.

Plug-and-play by design

DySpin upgrades existing sparse frameworks without replacing their high-level algorithm.



Evaluation: H100 first, then a 10 GB consumer GPU

The experiments test accuracy, kernel efficiency, end-to-end throughput, and memory capacity.

GPUs

H100 PCIe 80 GB
RTX 3080 10 GB

Models

Longchat-7B (MHA)
Llama-3.1-8B (GQA)
Qwen3-0.6B (GQA)

Baselines

Quest
RocketKV
FlashInfer

Workloads

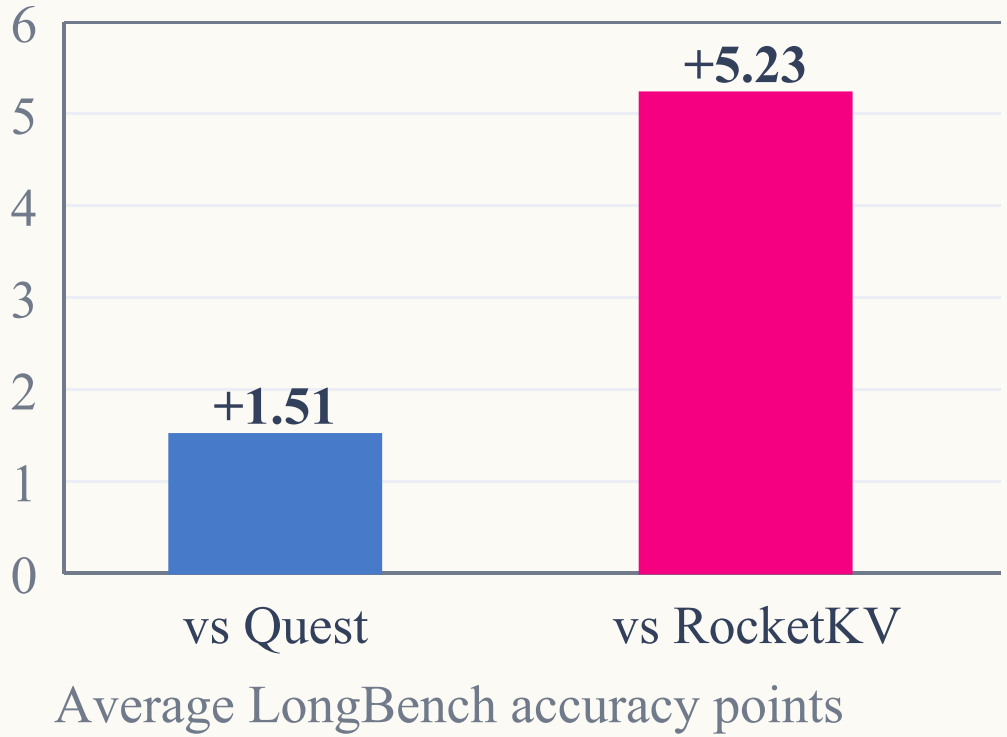
LongBench
PG19
32K → 320K contexts

Fairness rule: DySpin's average loaded-token budget is reused as the fixed budget for baselines.

[1] ICML '24 · MIT Han Lab / UW
[2] ICML '25 · NVIDIA / Georgia Tech
[3] MLSys '25 Best Paper · UW / NVIDIA / CMU

Accuracy: dynamic allocation improves quality at fixed budget

Across LongBench, DySpin's main algorithmic win is better use of the sparse-token budget.



Interpretation
At the same sparse budget, DySpin beats Quest by +1.51 and RocketKV by +5.23.

4-bit accuracy drop	**−0.69**
6-bit accuracy drop	**−0.08**

vs unquantized DySpin

At extreme sparsity
At 0.97 sparsity on Llama / MuSiQue, DySpin maintains full-attention accuracy while Quest falls to 87%.

Runtime allocation helps first; 6-bit quantization preserves most of that gain.

Perplexity: 6-bit stays close to unquantized DySpin

PG19 confirms the same trend: allocation improves PPL; 6-bit adds very little error.

−0.32	−4.07	−6.08	−2.78
Longchat vs Quest	**Longchat vs RocketKV**	**Llama vs Quest**	**Llama vs RocketKV**
PPL reduction from unquantized DySpin	PPL reduction from unquantized DySpin	PPL reduction from unquantized DySpin	PPL reduction from unquantized DySpin

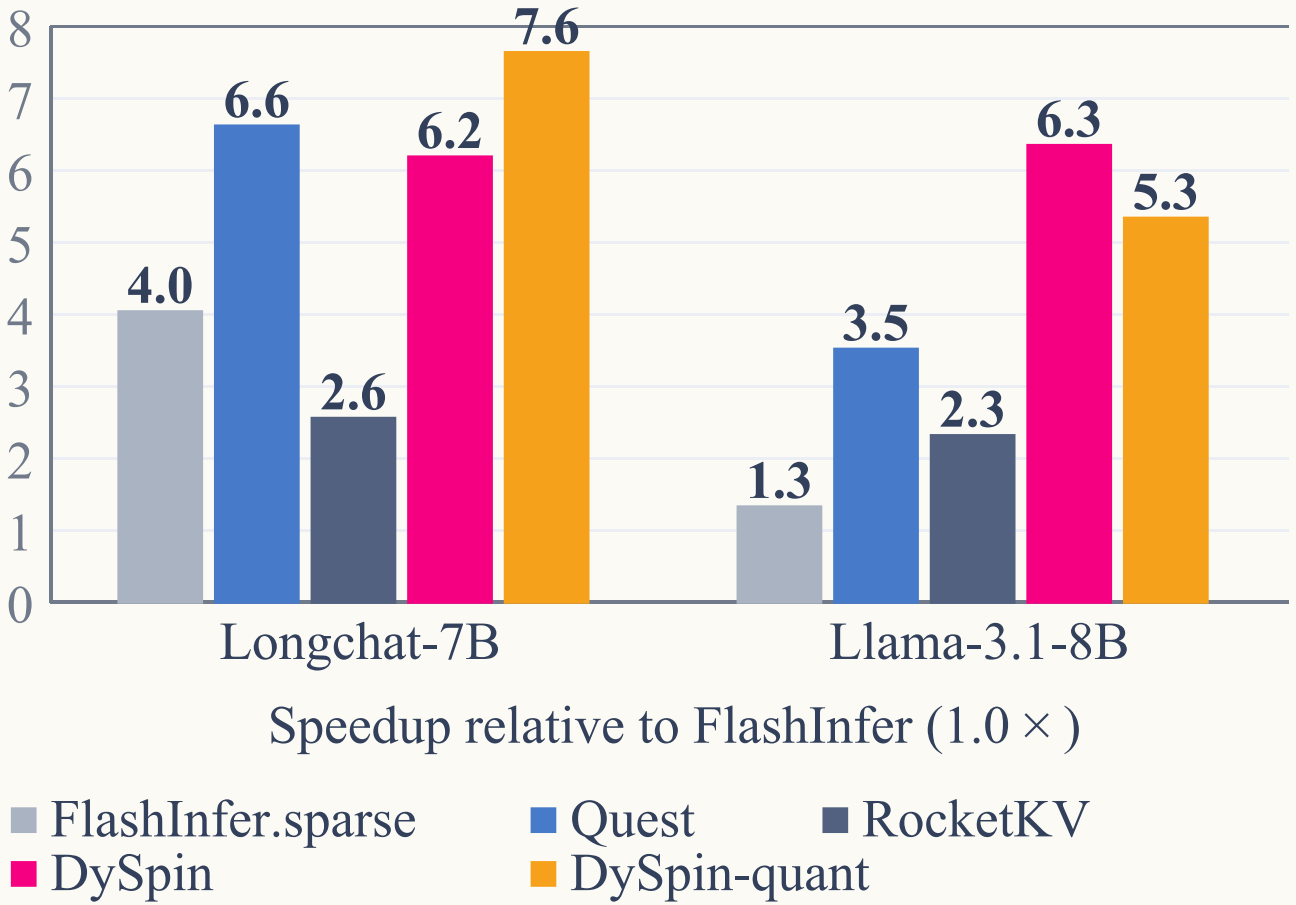
Extra PPL over unquantized DySpin



6-bit is the practical point where compression remains nearly lossless for the main models.

Sparse attention kernel: large speedups at 90% sparsity

MHA and GQA stress the sparse-attention kernel differently.



Longchat / MHA
DySpin-quant reaches 7.64× and is the fastest implementation in the MHA setting.

Llama / GQA
Unquantized DySpin peaks at 6.35×; the 6-bit variant gives up some speed because of extra register pressure.

Microbenchmarks: remove the “small” overheads that become dominant

FASTESTIMATE

1.96×

vs Quest Estimate+TopK

Longchat-7B

6.29×

vs Quest Estimate+TopK

Llama-3.1-8B

2.29×

vs FlashInfer-plan variant

Longchat-7B

6.58×

vs FlashInfer-plan variant

Llama-3.1-8B

FUSEDROPE

3.05×

Longchat, 16-bit

vs unfused RoPE + page management

3.14×

Longchat, INT6

vs unfused RoPE + page management

3.03×

Llama, 16-bit

vs unfused RoPE + page management

2.90×

Llama, INT6

vs unfused RoPE + page management

End-to-end: higher throughput and much longer contexts

The combination of sparsity + quantization changes both speed and feasibility.

144K

Longchat context

6-bit DySpin runs; baselines OOM at the extreme point.

320K

Llama context

6-bit DySpin runs; baselines OOM at the extreme point.

up to 1.62×

end-to-end speedup

DySpin.quant-enhanced RocketKV on Longchat.

Runtime allocation mainly improves quality; quantization mainly pushes the OOM boundary.

Beyond H100: 128K inference on a 10 GB GPU

Qwen3-0.6B · RTX 3080 (10 GB) · Throughput and accuracy under constrained memory

128K

feasible context

Quest / RocketKV: OOM
6-bit DySpin: runs

≈29.7 tokens/s

With 6-bit DySpin

Quest: 29.6 · RocketKV: 29.7
128K context · 0.9 sparsity

−3.8 PPL

vs sparse baselines

−3.80 Quest · −3.82 RocketKV
16K input · 0.9 sparsity

Positive result

Both unquantized and 6-bit DySpin outperform the sparse baselines and extend feasible context length.

Accuracy trade-off

For Qwen3-0.6B, 6-bit quantization increases PPL relative to unquantized DySpin, despite beating the baselines.

DySpin: Co-designing Dynamic Sparse Inference

What to load · How to execute · How to store

1. Adaptive Allocation

Per-head, per-step budgets preserve critical context.

Higher accuracy at the same average token budget.

2. GPU-Efficient Execution

FastEstimate and load-balanced attention handle irregular work.

FusedRoPE cuts preprocessing overhead.

3. Compact KV State

Execution-aware 6-bit packing reduces KV-cache footprint.

Up to 1.62× speedup and 320K context.

Efficient sparse inference requires co-designing what to load, how to execute, and how to store.



Thanks
Questions?

Author's Email: chengyusun@whu.edu.cn