



CONFERENCE

—NORTH AMERICA 2026—

A Developer's Guide to Attention in vLLM

Lucas Wilkinson & Matthew Bonanni
RedHat, vLLM Committers

vLLM: The De Facto Open Inference Platform



Muse



Qwen



DeepSeek



Gemma



Mistral



Molmo



Phi



Nemotron



Granite



GPU



Instinct



TPU



Neuron



Gaudi



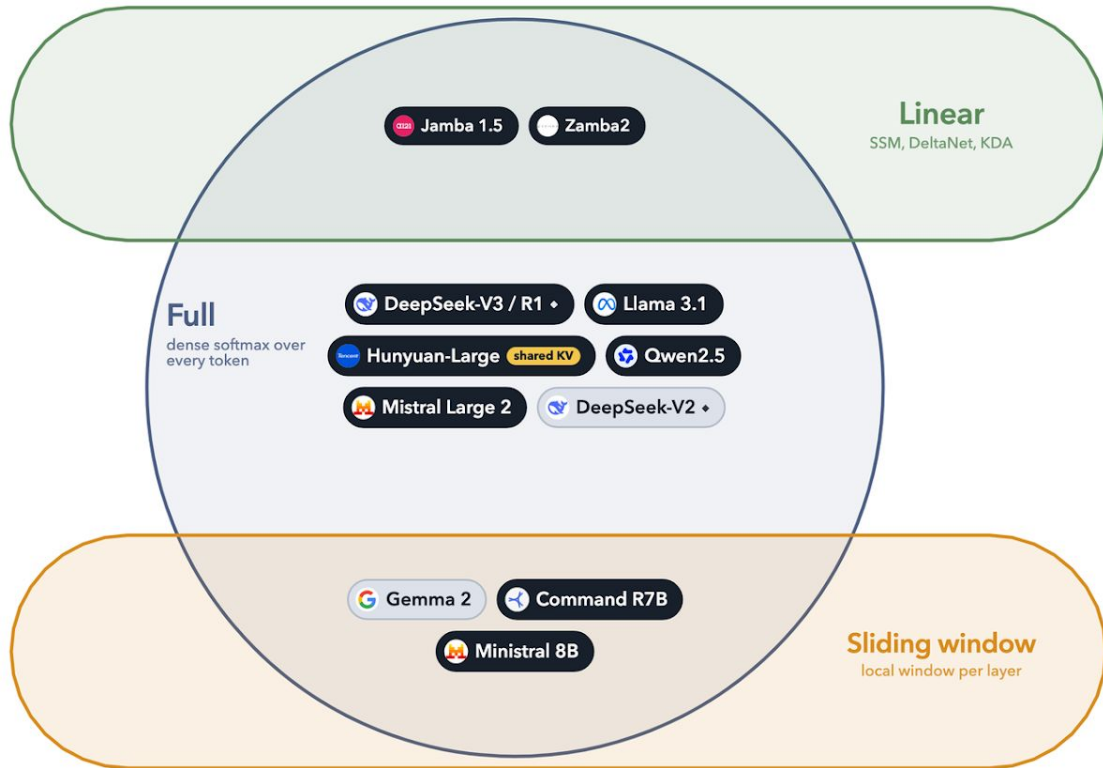
Spyre

2024

11 new designs

H1 · 2

H2 · 9



◆ MLA latent KV; others GQA

Paper paper, no weights yet

shared KV KV reused across layers

PyTorch

CONFERENCE

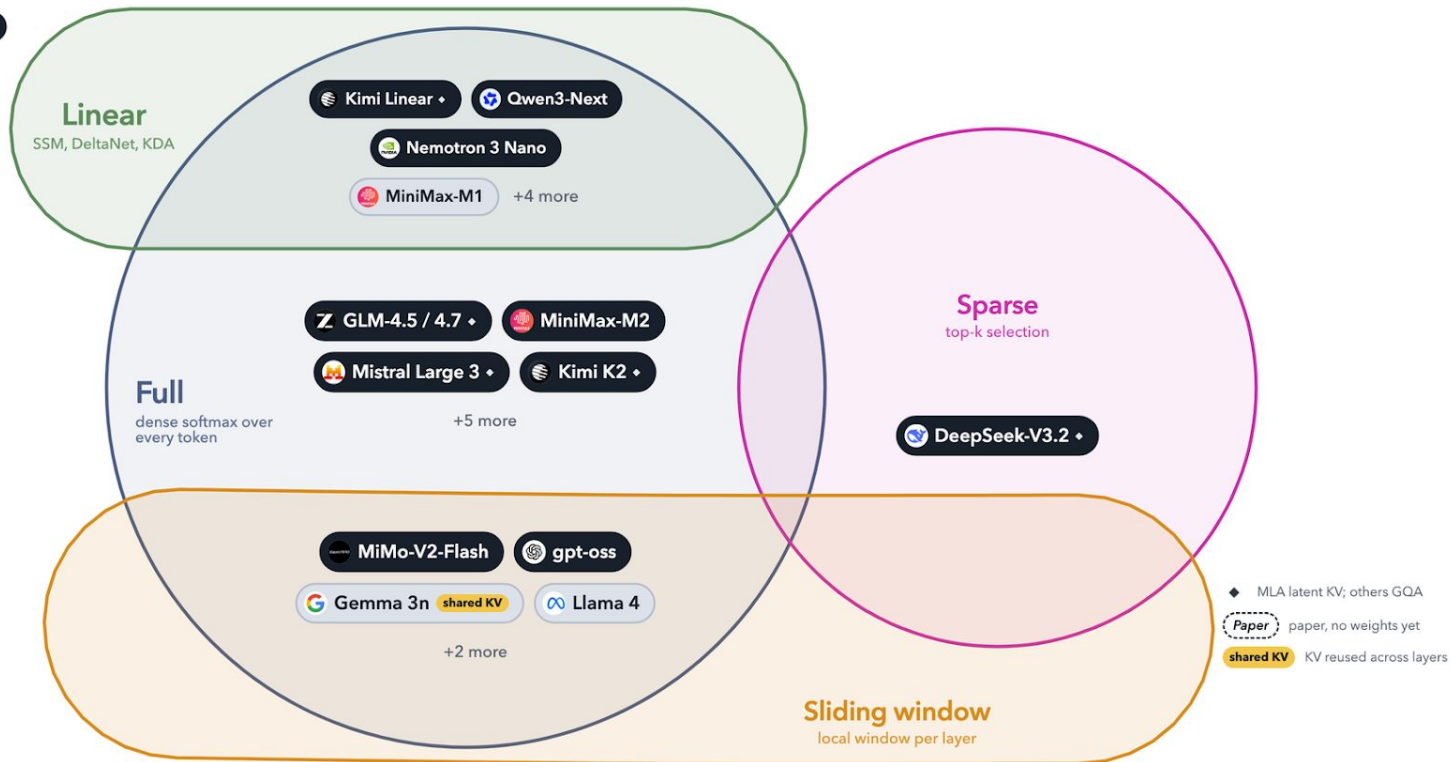
—NORTH AMERICA 2026—

2025

28 new designs

H1 · 3

H2 · 25

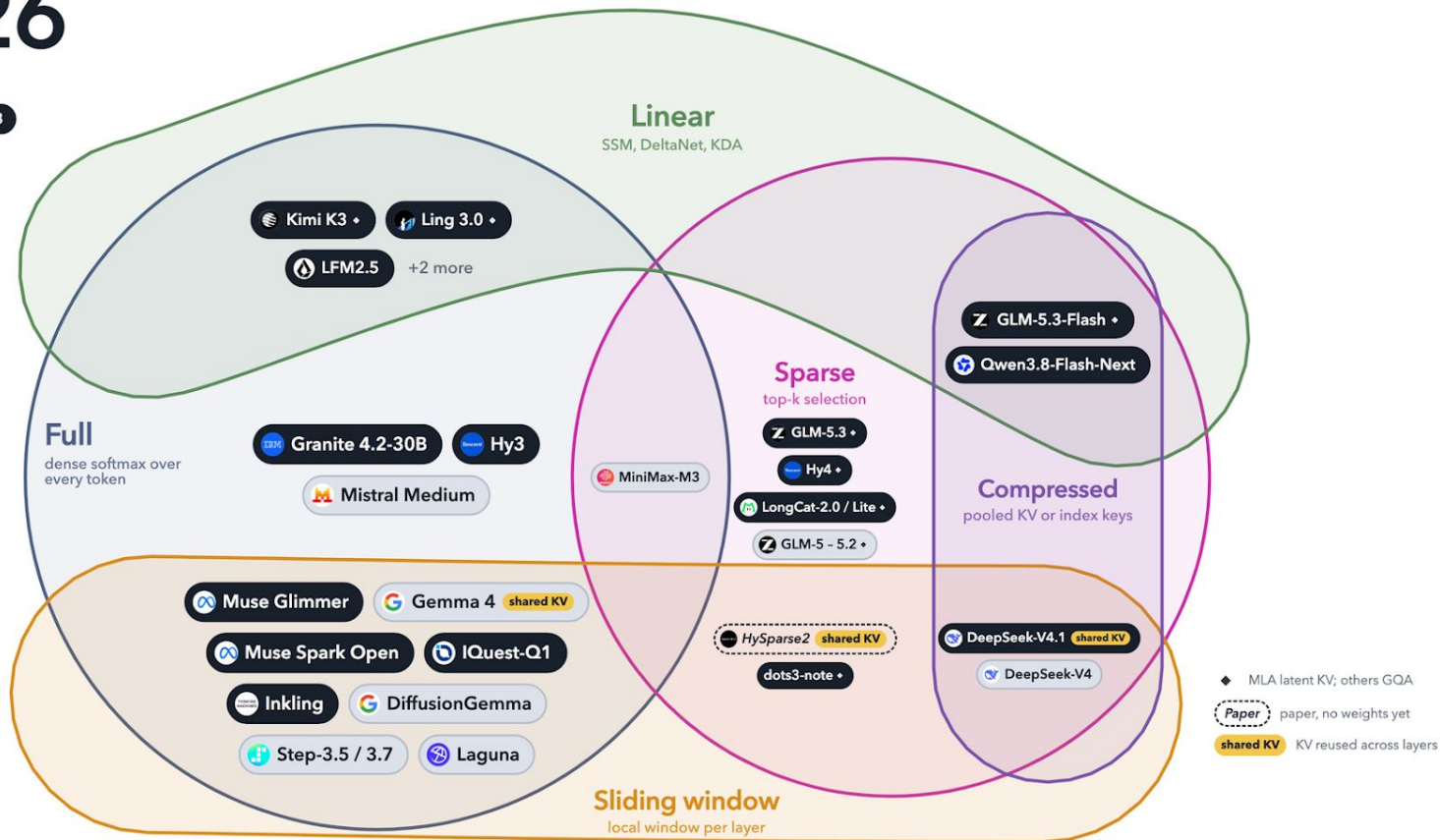


2026

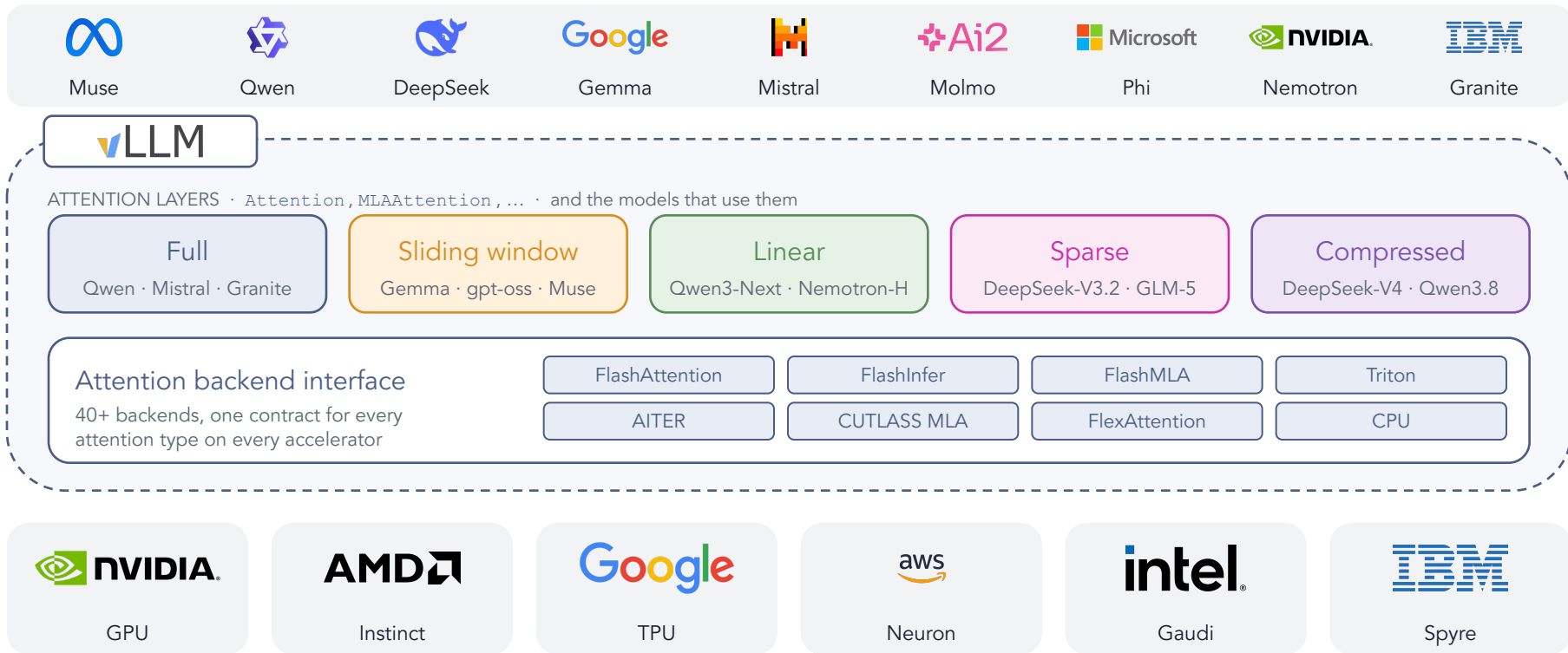
34 new designs

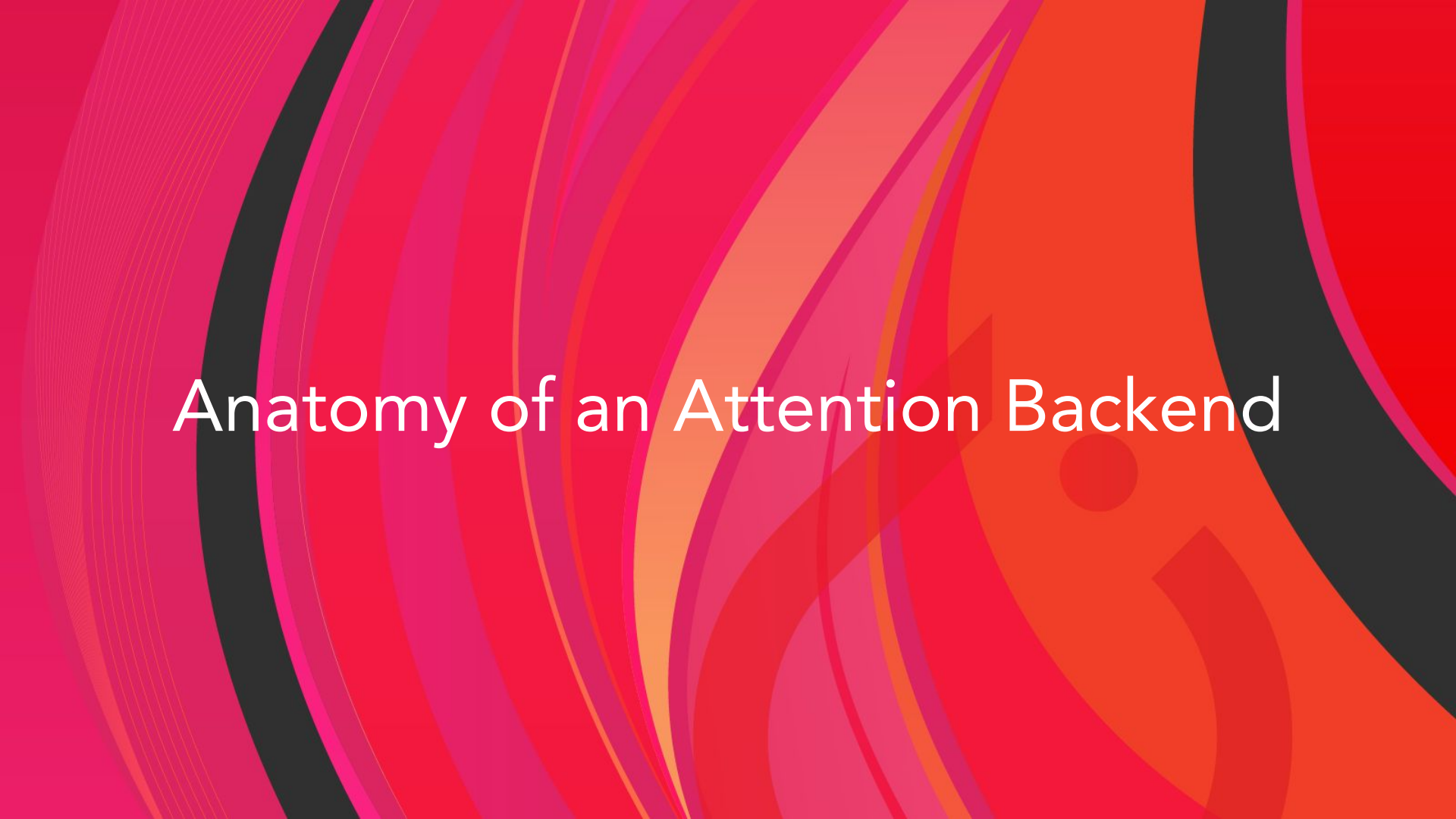
H1 · 16

H2 · 18



vLLM: The De Facto Open Inference Platform





Anatomy of an Attention Backend

Constructing batch metadata

Per-step metadata

describes this step's batch · changes every step · shared by all layers

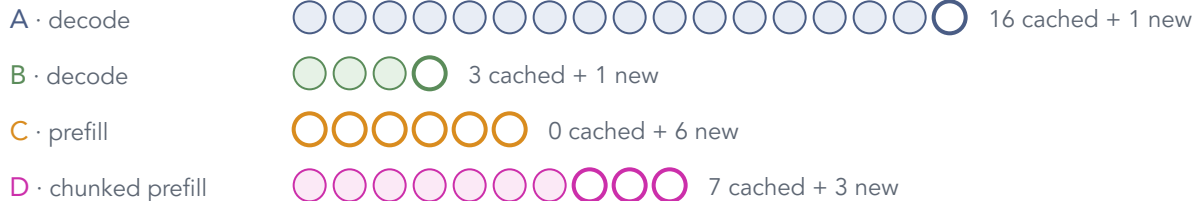
Static config

fixed · one per layer

`num_heads, head_size, scale` → `AttentionImpl`

Requests in this step

☐ cached context ☒ new this step



The kernel sees one flat batch of new tokens



`CommonAttentionMetadata` is built by `GPUModelRunner` once per step; your builder adds extras

`query_start_loc = [0, 1, 2, 8, 11]`
where each request's new tokens start

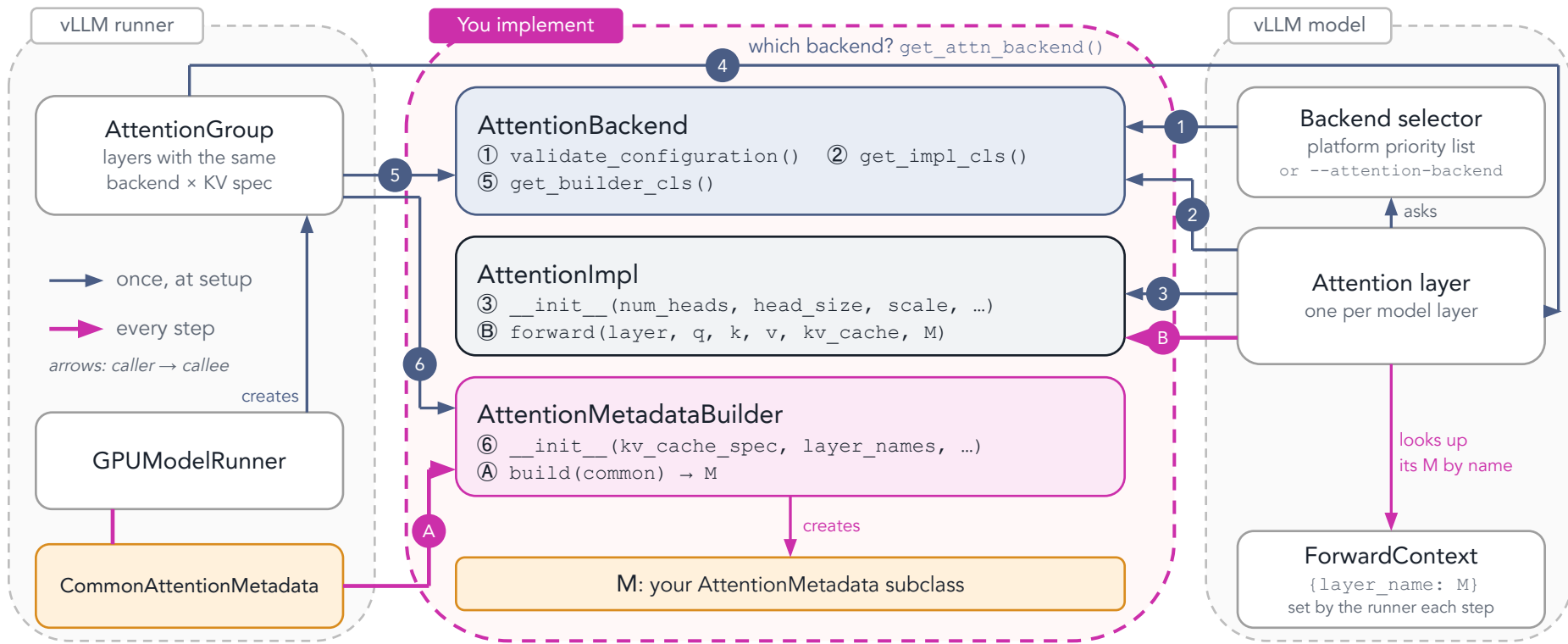
`seq_lens = [17, 4, 6, 10]`
how far back each request attends

`block_table · slot_mapping`
where each request's KV lives in paged memory

+ backend extras

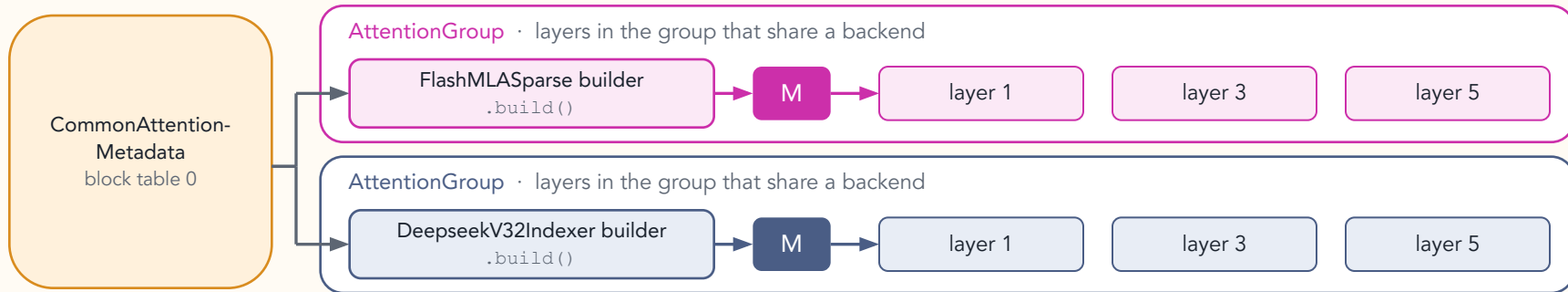
e.g. `num_decodes = 2, num_prefills = 2`

Backend interface

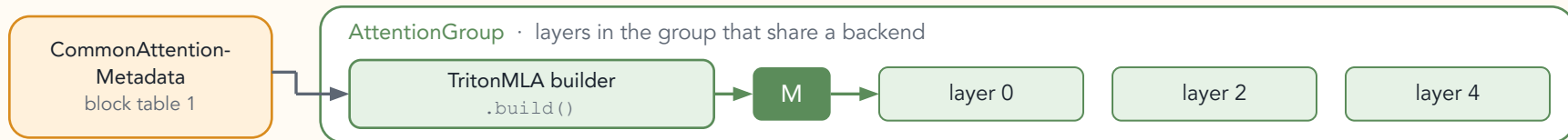


KV-cache groups and attention groups

KV-cache group 0 · layers that share one block table, e.g., full attention

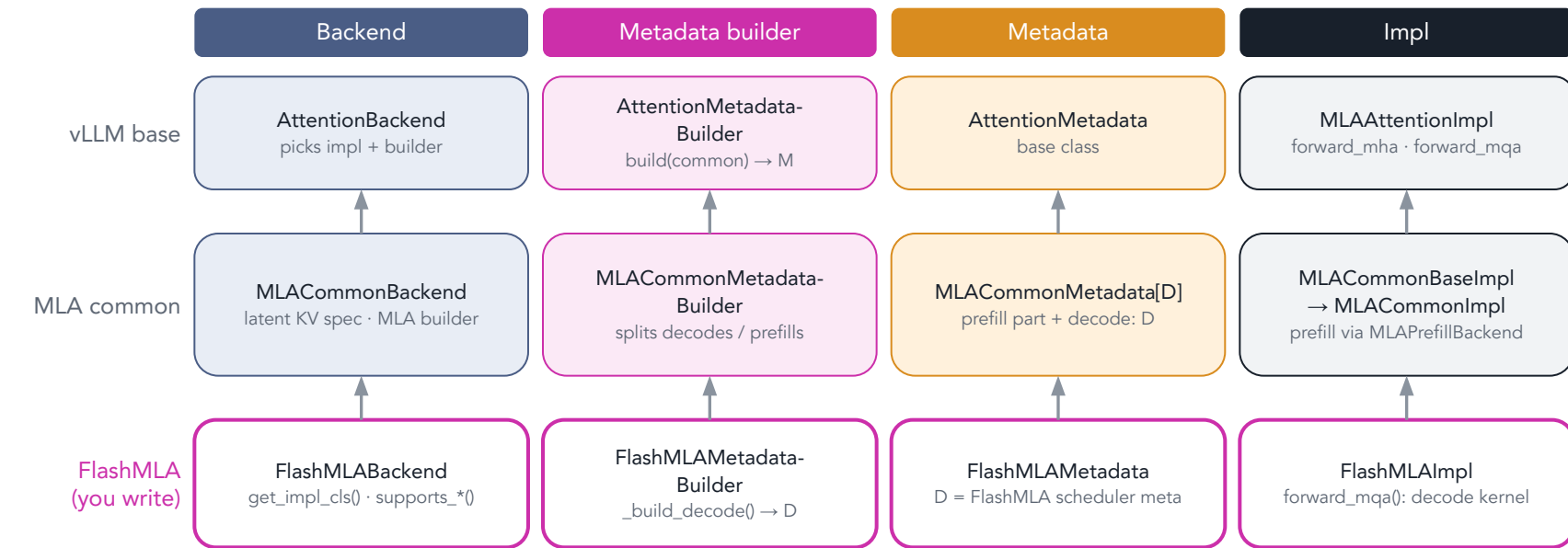


KV-cache group 1 · layers that share one block table, e.g., sliding window attention



Hybrid Memory Allocator: https://www.youtube.com/live/0cUFUtNW_S8?si=MIB1isg_PfPUYQkW

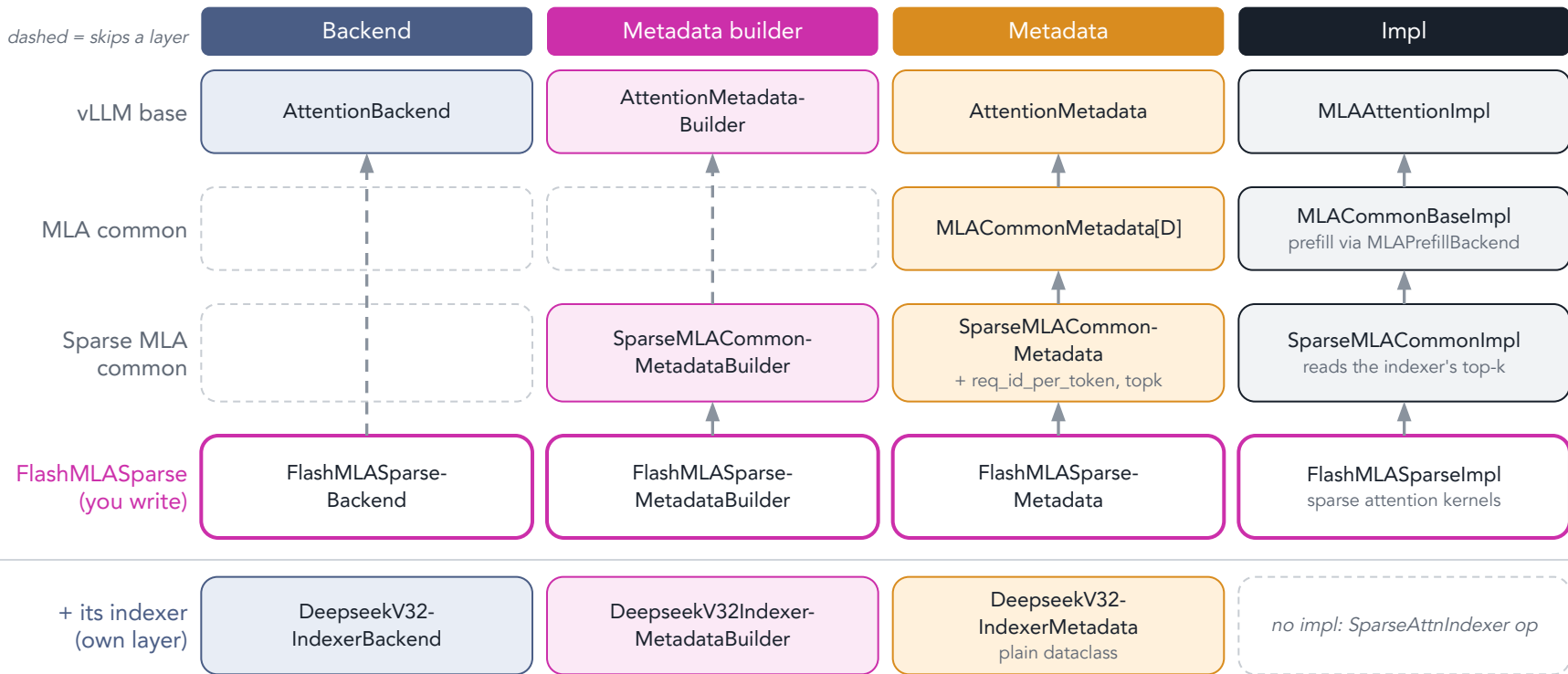
MLA backend class hierarchy



lines = inherits from

Shared prefill: MLAPrefillBackend (FlashAttention, FlashInfer, TRT-LLM ragged, ...) runs every MLA backend's prefill rows. A new MLA backend mostly writes its decode kernel and its decode metadata.

Sparse MLA backend class hierarchy



[illegible]

sort
→
decodes first

decode kernel · small query, long context

C ○○○○○○○○○○○○○○●

E ●●●●

prefill kernel · long query, little context

B ○○○○○○○○○○
D ○○○○○○○○
A ○○○○○

split decodes and prefills() → num_decodes, num_prefills in M

○ cached context

○ new this step

reorder batch threshold decides what counts as a decode

`threshold = 1` (default)

B 

B has 3 new tokens → prefill kernel

$$\text{threshold} = 1 + k \quad (k = 2)$$

B 

B is a decode $\rightarrow$ decode kernel (spec-as-decode)

CUDA graph support

ONE FORWARD PASS

Full graph
one replay per step



Piecewise
attention runs eagerly



YOUR BUILDER'S `_cudagraph_support`

ALWAYS

UNIFORM_BATCH

UNIFORM_SINGLE_TOKEN_DECODE

NEVER



pure decodes



spec decode (1+k)



mixed

full graph

full graph

full graph

full graph

full graph

piecewise

full graph

piecewise

piecewise

piecewise

piecewise

piecewise

Weakest backend wins: the minimum across all attention groups

Fixed addresses: keep M's tensors in persistent buffers for replay

KV Cache Layout Standardization



Who needs to be KV-layout aware?

Attention kernels

write it, then read it every step

FlashAttention, FlashInfer, Triton,
FlashMLA, CPU: each its own order

defines the layout

KV cache

one tensor per layer group

`kv_cache.shape`



which dim is which?

only the attention backend
knew the answer

P/D connectors

ship a request's blocks to another node

different TP on P and D splits heads;
cross-layer sends one chunk per block

must slice



CPU offload

copy blocks to host memory and back

block-outer: ~32 KB → ~2 MB DMA chunks,
up to 4× lower TTFT

must slice



Model runner

binds one view per layer

cross-layer: one buffer for all layers,
handed out as one view per layer

must slice



Scheduler: only sees block IDs and page bytes, so it is layout-blind

connectors: NIXL · Mooncake · MoRIIO · LMCACHE · HF3FS

 K/V

 blocks

 heads

 layers

 only for the old K/V-first layouts

Backend layouts leaked into common code

Model type Is it MLA? Mamba?

nixl/worker.py

```
if not self.use_mla and not self._has_mamba:
```

K/V layout Is K/V split?

kv_connector/utils.py

```
return cache if self.split_k_and_v else [cache]
```

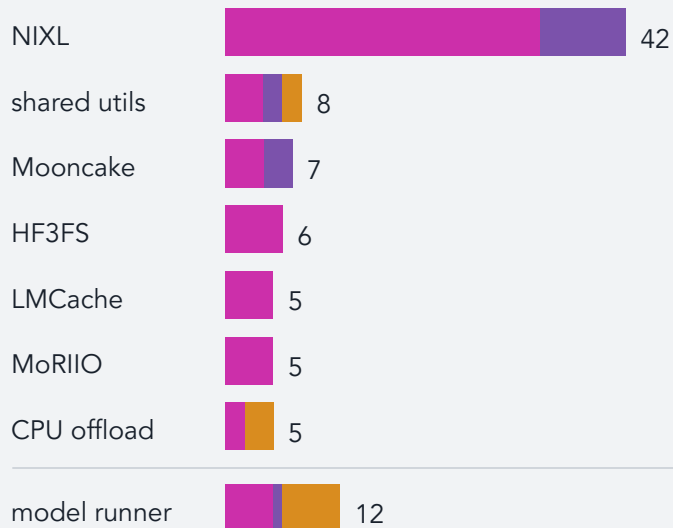
Layout probe Where is the block dim? What stride order?

offloading/worker.py

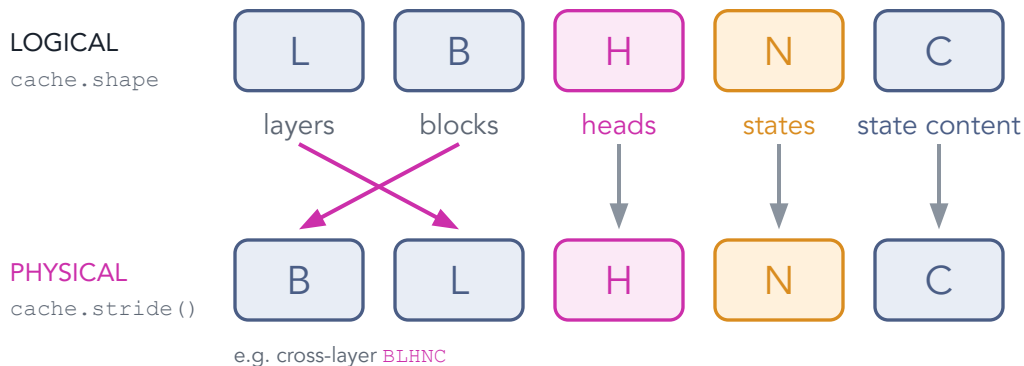
```
...get_kv_cache_shape(num_blocks=1234, ...)  
block_dim = test_shape.index(1234)  
physical_to_logical = ...get_kv_cache_stride_order(...)
```

if-branches and layout probes

■ is MLA / Mamba? ■ layout flag? ■ layout probe



One standard logical shape and dim ordering



TYPE	H	N	C
GQA	<code>kv_heads</code>	<code>block_size</code>	<code>[2, head_dim]</code>
MLA	<code>1</code>	<code>block_size</code>	<code>[latent]</code>
DeepSeek-V4 c4a	<code>1</code>	<code>block_size/4</code>	<code>[latent]</code>
Mamba2	<code>1</code>	<code>1</code>	<code>[conv, ssm]</code>

Each KV-cache user builds the view it needs

Layer view `cache[layer]`

Slice blocks `cache[block_ids]`

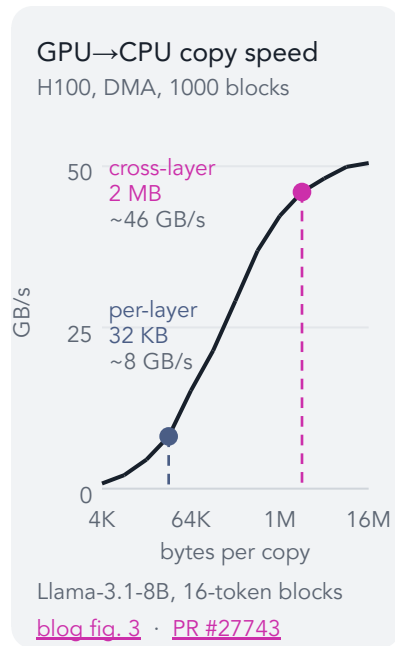
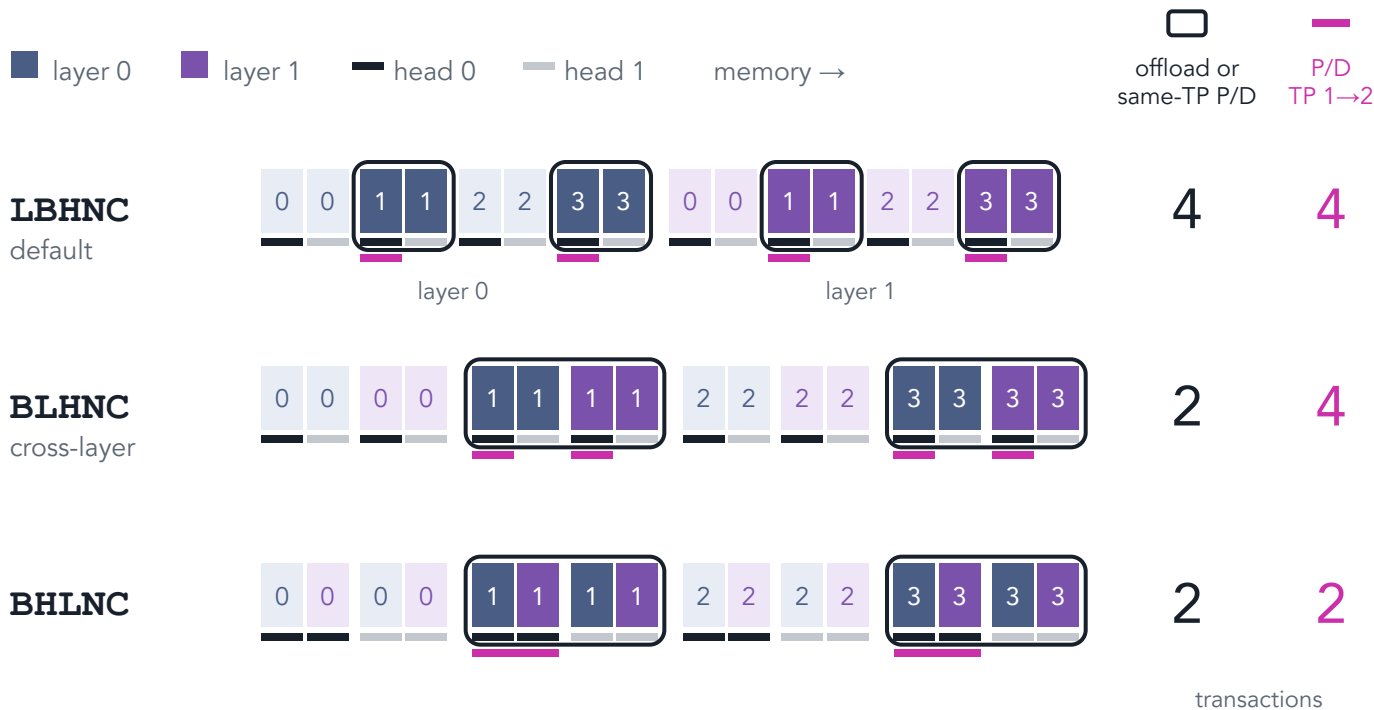
Slice heads `cache[:, h0:h1]`

Slice states `cache[:, :, n0:n1]`

Kernel dim order `cache.permute(*kernel_order)`

Split K and V `cache.split(D, -1)`

The stride order sets transfer cost



Pick it per deployment: `VLLM_KV_CACHE_LAYOUT`

RFC #42082

In-Progress: refactor connectors

Today

nixl/base_worker.py

```
if not self.use_mla and not self._has_mamba:
    assert not (tp_ratio < 0 and is_kv_replicated(...))
if self._has_mamba and remote_phys != local_phys:
    raise RuntimeError("... not supported for Mamba")
if not self.use_mla and remote.kv_cache_layout != layout:
    ... # permute HND <-> NHD, or raise
if abs(tp_ratio) != 1 and not self.use_mla and ...:
    raise RuntimeError("... requires contiguous heads")
if self._has_mamba and self.use_mla:
    ... # KimiLinear KDA+MLA block lengths
elif not self._has_mamba:
    replicated = self.use_mla or ...
    ... # per-head byte math for GQA
```

106 model-type checks

across 15 connector files

Proposed

RFC #42082 · step 3/N

```
# each spec slices itself (GQA by heads shown)
class AttentionSpec(KVCacheSpec):
    def slice_for_tp_transfer(self, t, tp, rank,
                             other, other_tp, other_rank):
        lo, hi = head_overlap(self, tp, rank, other, ...)
        return [t[:, :, lo:hi]] # [L, B, H, N, C]

# the connector: one loop for every model
src = src_spec.slice_for_tp_transfer(src_cache, ...)
dst = dst_spec.slice_for_tp_transfer(dst_meta, ...)
for sb, db in block_mapping:
    for s, d in zip(src, dst):
        for a, b in jointly_contiguous_chunks(
            s[:, sb], d[:, db]):
            rdma(a, b)
```

New models out of the box

0 model-type checks · a new cache type only implements its spec

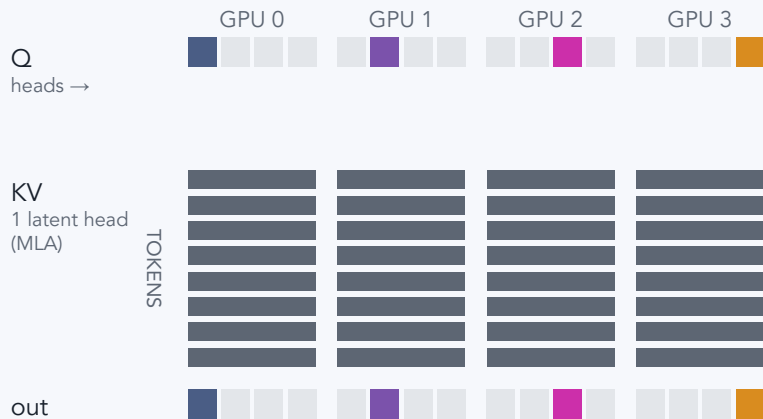


Context Parallelism

Decode Context Parallelism (DCP): KV-Sharding

TP4 · split the heads

each GPU owns 1/4 of the query heads

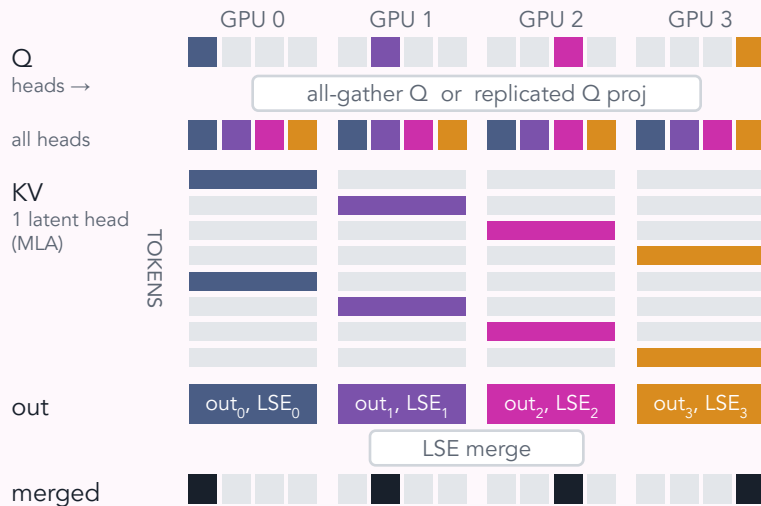


Every GPU stores every token: 4 copies

MLA has one KV head, so TP has nothing to shard

TP4 + DCP4 · split the tokens

each GPU stores 1/4 of the tokens · all heads · [DCP blog](#)



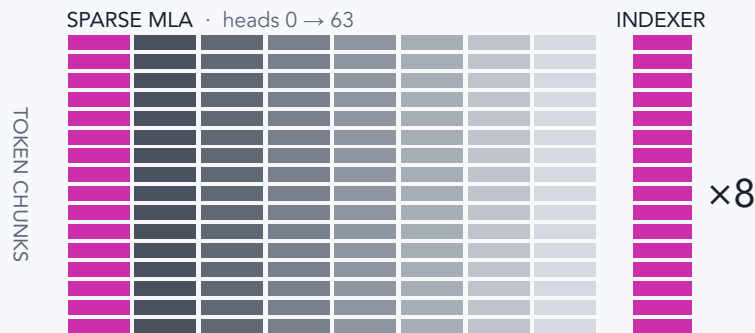
KV stored once: 4× the context

moves ~64 KiB per token per layer; a 128K request's KV is ~84 MB

Prefill Context Parallelism (PCP): Q-Sharding

TP8 · split the heads

each GPU owns 8 heads of every token



Every GPU computes the full indexer

SPARSE MLA FLOPs PER BYTE MOVED · GLM-5.3 DS-FP8

26

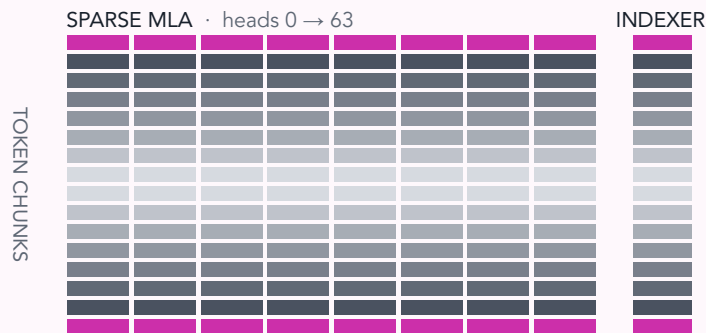
memory-bound · 8 heads per KV row

206

PCP8 · split the tokens

- GPU 0 1 2 3 4 5 6 7

each GPU: chunks r and $15 - r$, all heads · [load-balanced sharding](#)



Each GPU indexes only its own tokens

SPARSE MLA FLOPs PER BYTE MOVED · GLM-5.3 DS-FP8

191

near the ridge · 64 heads per KV row

206

Mostly backend-agnostic

Generic: lives in vLLM, not in the backend

DCP · KV sharding

PCP · Q sharding

Model runner (MRv2)

Local seq lens

`prepare_dcp_local_seq_lens`

PCPManager

`DualChunkSwap` split per rank

Slot mapping

`token i → rank (i // I) % N`

Slot mapping

slots for all-gathered KV

MLA attention layer

All-gather Q

then LSE merge, or all-gather KV (prefill)

All-gather KV

WHAT A BACKEND OWES Return LSE · accept H·N heads

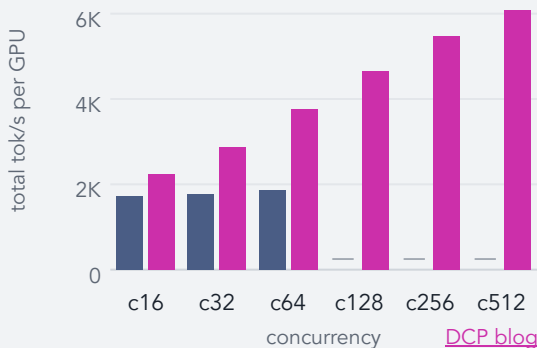
Backend support today

Backend	DCP	PCP
FlashMLA, FlashMLA sparse	✓	✓
FlashInfer / Triton MLA	✓	✓
FlashAttention, FlashInfer (GQA)	✓	✗
AITER MLA sparse (ROCm)	✗	✓
b12x, QSA (GQA)	✗	✗

Results: DCP and PCP

DCP · Kimi K2.6 NVFP4, 8× B200

TP8 + EP8 KV on all 8 GPUs, full at c64
TP8 + EP8 + DCP8 KV split 8 ways by token

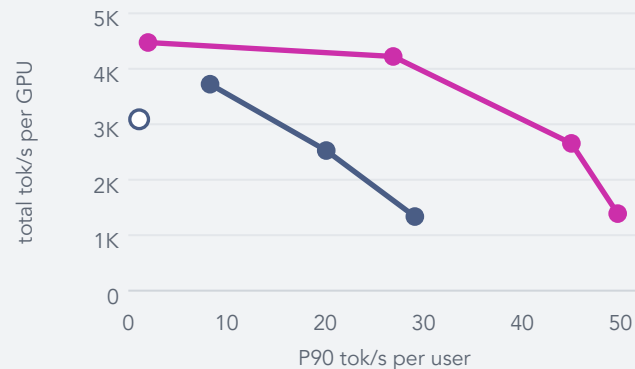
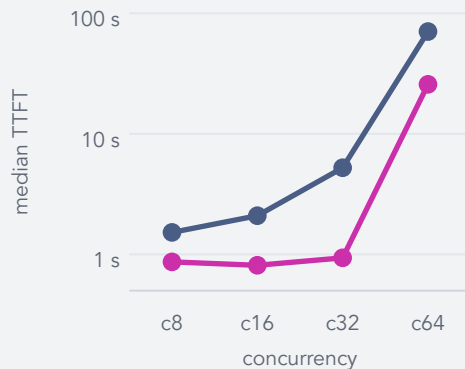


PCP × DCP · GLM-5.3 on 16× H200, AgentX (142K)

prefill
8× H200 PCP8 × DCP8 × EP8
DP8 × EP8

NIXL

decode
8× H200 DP8 × EP8
CPU KV offload, 64 GiB per rank



TP split heads · DP independent replicas · EP split MoE experts · DCP split the KV cache by token · PCP split the prompt by token

3.3×

DCP peak tok/s/GPU vs TP8
8× the concurrency on the same GPUs

1.7–3.2×

PCP per-user speed vs DP8 prefill
median TTFT 0.8–0.9 s vs 1.5–5.2 s

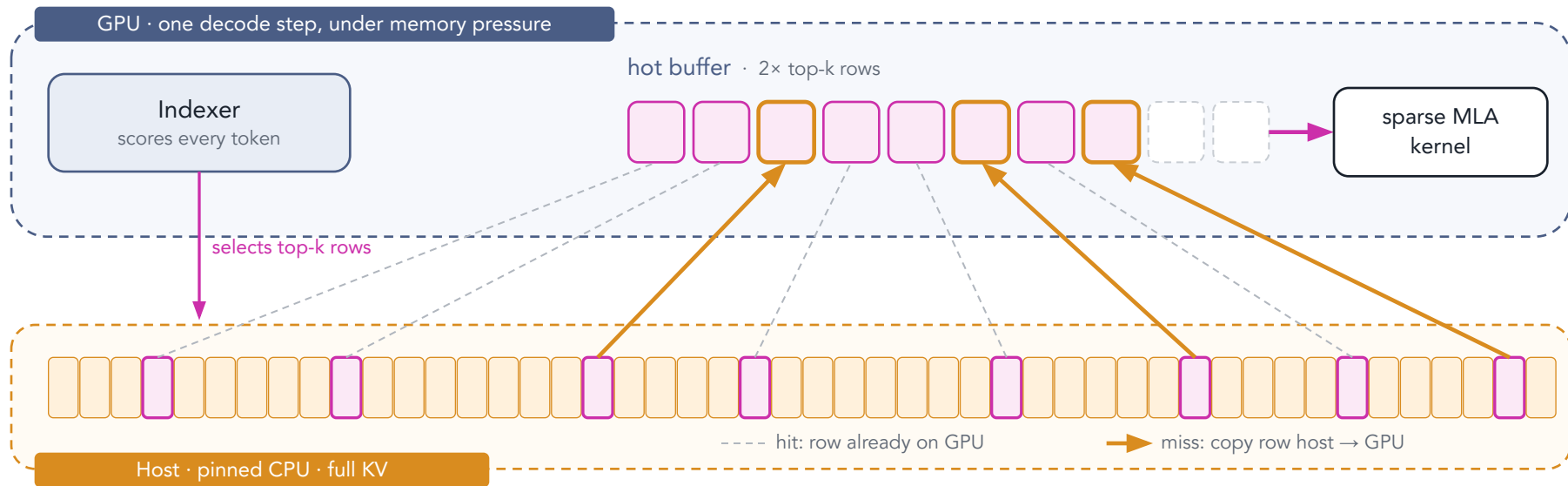
8×

smaller KV per GPU with DCP8
a 1M request: 53 → 6.7 GiB per GPU



Hybrid HiSparse

Hybrid HiSparse



Hybrid
KV stays fully on GPU
until it runs short; then offload

GPU KV per request
 $O(\text{context}) \rightarrow O(\text{top-k})$
~4K rows ($2 \times \text{top-k}$) at any length

Kernel unchanged
hot buffer = ordinary KV blocks
same tensor, block table

Built on KV-cache groups
host, resident, hot tiers
are groups in the HMA

Hybrid HiSparse benefits

Much longer context

context is bounded by host memory, not GPU KV

1M-token context on 16× H100

GLM-5.3: not even one 1M request fits without it

CONCURRENT REQUESTS, WITHOUT → WITH

128K

1M

16× H100

2 → 28

0 → 4

8× H200

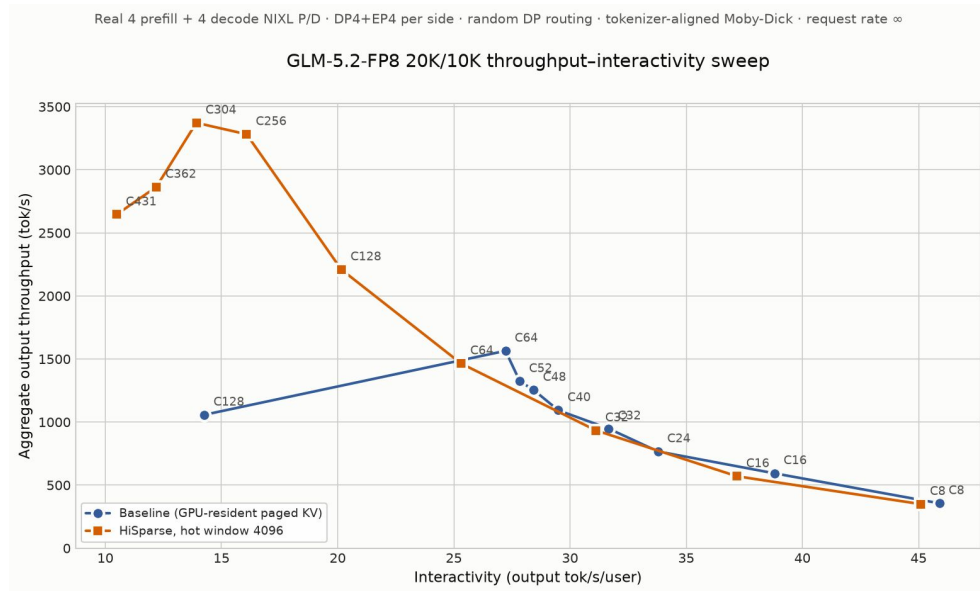
3 → 30

0 → 5

GLM-5.3 FP8, FP8 KV, TP8 (×DP2 on H100), EP. Estimates from the blog's concurrency calculator; 8× H100 can't hold the weights.

Much higher throughput

2.2× peak · scales to 304 requests (baseline peaks at 64)



GLM-5.2-FP8 · 20K in / 10K out · B300 · 4P + 4D NIXL P/D · preliminary, vllm#53781



Future Plans

Next: one class per attention backend

Today · 4 classes per backend

AttentionBackend
capability queries, factories

AttentionMetadataBuilder
builds metadata each step

AttentionMetadata
per-step batch state

AttentionImpl
per-layer config + kernel call



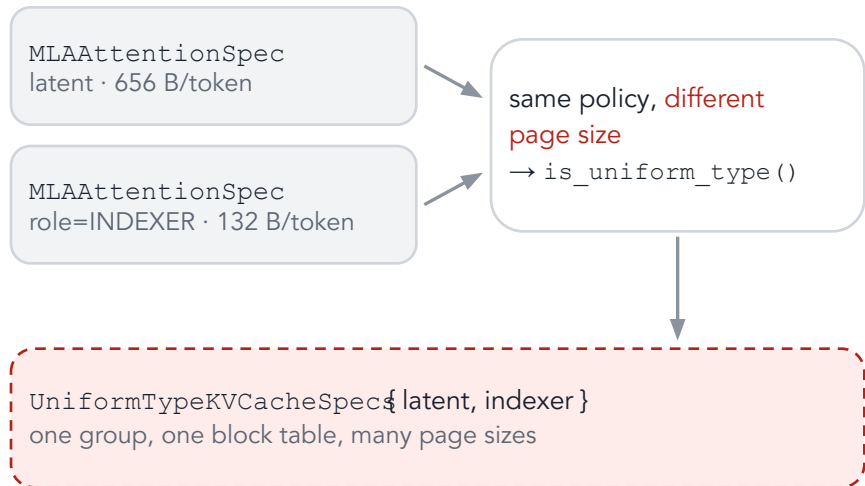
Proposed · one class

```
class AttentionBackend:
    def get_cache_config()
        # -> manager, config, tensor spec
    def bind_kv_caches(kv_caches, groups)
    def prep_forward(common_md, tables)
        # replaces Builder + Metadata
    def forward(layer_config, q, k, v, out)
```

RFC #42449

Remove UniformTypeKVCacheSpecs

Today

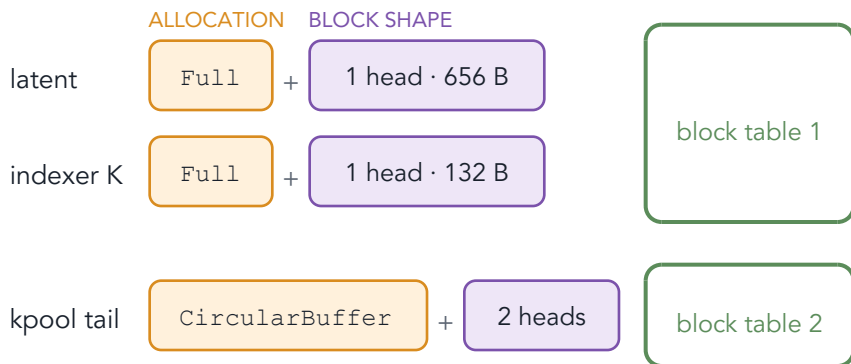


36

`isinstance` checks across
24 files to unwrap it

Proposed

split the **block shape** from the **allocation strategy**



Same manager → same block table.
No wrapper, nothing to unwrap.

[draft PR #61452](#) · [RFC #42449](#)



Thanks! Questions?

What standardizing layouts gives us

Less code

-2,046

lines, 9 merged PRs

model runners  -780

KV connectors  -607

tests  -421

backends  -199

`get_kv_cache_shape` 21 → 1

Cross-layer is just a layout

~~`prefer_cross_layer_blocks`~~
~~`register_cross_layers_kv_cache()`~~
~~`include_num_layers_dimension`~~



BLHNC

one layout value; works for hybrids

Fewer KV groups

LBHNC · one page size per block



46

BLHNC · mixed page sizes per block



5

 GDN
  full
 drafter SWA
  drafter full

Qwen3.6-35B-A3B + DFlash drafter

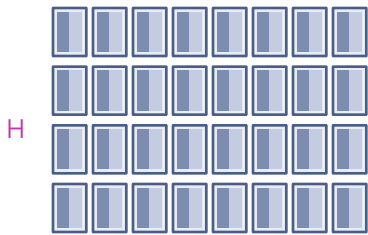
More stable one layout path, not one per backend

One logical shape for every KV cache

[L , B , H , N , C]
layers blocks heads states state content

GQA

K and V per head



H = kv_heads
N = block_size
C = [2, head_dim]

MLA

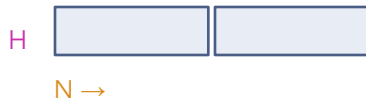
one shared latent



H = 1
N = block_size
C = [latent]

DeepSeek-V4 c4a

4 tokens per state



H = 1
N = block_size / 4
C = [latent]

Mamba2

conv + SSM state per block



H = 1
N = 1
C = [conv, ssm]

Hybrid memory allocator: [talk \(Chen Zhang\)](#) · [Jenga paper](#)

RFC #42082