CSE 291A/DSC 291: Data Systems for Machine Learning, Spring 2026

Guest Lecture: Engineering vLLM for Efficient LLM Inference

Lecturer: Woosuk Kwon

Scribes: Liam Chen, Priyanka Nidadavolu, Sanjana Garimella, Shao Duan, Mythili Velapakam, Sharvaani Thoguluva, Kate Muret

1 Introduction

This guest lecture was delivered by Woosuk Kwon (Inferact), co-creator of vLLM and a maintainer of the open-source project. The talk focused on how vLLM is engineered in practice: APIs and routing, rendering inputs, the scheduler and KV cache manager, CUDA workers, and the optimizations used to serve modern models at scale.

vLLM is an open-source LLM inference engine whose stated goal is to make inference "efficient and effortless." It originated at UC Berkeley and has grown into a highly collaborative project with major contributors across industry and academia, including NVIDIA, Red Hat, Meta, and Inferact. The project currently has over 79K GitHub stars, reflecting its widespread adoption.

Modern inference workloads combine tight per-step latency budgets, heterogeneous hardware topologies, rapidly evolving model architectures (dense, MoE, and hybrid attention), and cluster-scale deployments. The lecture organizes vLLM's optimization philosophy around four recurring pillars:

1. minimizing CPU overheads,
2. using efficient GPU kernels,
3. choosing appropriate model parallelisms,
4. managing GPU memory and KV cache efficiently across heterogeneous memory tiers.

2 User-facing APIs

vLLM exposes two primary interfaces that mirror common developer workflows.

Offline batched inference (`LLM`). The Python `LLM` class accepts a model identifier (for example from Hugging Face), a list of prompts (strings), and returns completions via `generate`. Under the hood vLLM performs initialization, kernel selection, memory planning, batching, and scheduling so

that researchers can obtain throughput-oriented batched inference without manually orchestrating kernels.

Online serving (OpenAI-compatible server). The engine ships with a FastAPI-based server launched via `vllm serve <model>`. Clients may call OpenAI-style HTTP endpoints with familiar JSON payloads or use OpenAI Python clients against a local base URL. This path targets interactive workloads where token streaming, concurrency, and request routing dominate operational concerns.

3 Architecture Overview

The speaker presented a layered architecture aligned with how requests traverse a real deployment.

Routing layer. Large deployments shard a model across many GPUs and nodes. A Rust-based router accepts inbound requests and assigns them to a concrete vLLM engine instance. This component becomes essential once horizontal scaling and/or disaggregated prefill/decode setups appear.

Frontend / "renderer" generalizing tokenization. Each engine runs an API server plus a rendering pipeline that generalizes classical tokenization. Beyond converting raw text to token IDs, this stage may apply chat templates, tool parsers, and multimodal preprocessing so that downstream GPU kernels consume a uniform tensor-centric representation (sometimes accompanied by auxiliary multimodal tensors).

Engine core. The engine core includes the scheduler and KV cache manager. Together they decide *which* requests participate in each forward step, how batches are formed under continuous batching semantics, and how GPU memory is carved into KV blocks (including prefix caching metadata).

CUDA workers. Each GPU typically hosts a worker process owning model weights and executing forwards. Workers sample next tokens (or verify speculative proposals), returning minimal control-plane state back to the engine core each decoding step.

Continuous batching context. The scheduler and engine core implement iteration-level batching for variable-length generations: requests join and leave the active batch as they complete or are cancelled. That dynamism is why cheap per-step CPU bookkeeping is insufficient; every iteration revisits membership, memory residency, and speculative verification metadata.

Tensor and Data Parallelism in vLLM For tensor parallelism, model weights are sharded across different worker threads, and managed by a single engine core. For data parallelism, for each DP rank vLLM creates a designated engine core. A separate data coordinator process is used for load-balancing between all the engine cores when DP is enabled.

4 Minimizing CPU Overheads

4.1 Why CPU overhead matters in inference

Although LLM inference executes its arithmetic almost entirely on GPUs, host-side overhead (Python, scheduler, allocator, dispatcher) frequently caps end-to-end utilization, because each de-

coding step is short. Strong setups deliver $r \approx 100$ to 200 output tokens/s, which corresponds to a per-step budget of

$$t_{\text{step}} \;=\; \frac{1}{r} \;\approx\; 5 \text{ to } 10 \text{ ms},$$

compared to 100 ms to several seconds for a training step. A back-of-the-envelope estimate makes the sensitivity explicit: an extra $\Delta t = 1$ ms of CPU bookkeeping on top of a 5 ms step degrades throughput by

$$\frac{\Delta t}{t_{\text{step}} + \Delta t} \;=\; \frac{1}{6} \;\approx\; 17\text{–}20\%,$$

matching the 20% degradation cited in the lecture. A single careless tensor copy, an extra dispatcher round-trip, or a stray callback in the streaming layer suffices to incur this in Python, and the problem worsens as GPU per-step latency continues to shrink (A100 $\rightarrow$ H100 $\rightarrow$ GB200) while single-thread CPU speeds plateau.

vLLM addresses host stalls across three layers: the *API server* (rewritten in Rust, with asynchronous de-tokenization), the *engine core* (asynchronous scheduling), and the *worker* (GPU-native input preparation, full and piecewise CUDA graphs).

4.2 API server and renderer

At sustained concurrency the frontend services thousands of streaming clients. Two costs dominate: *input rendering*, which converts chat messages, tool calls, and multimodal payloads into the tensor representation consumed downstream, and *output streaming*, which fans out one chunk per client per generated token, where even microseconds of per-token serialization can saturate a Python event loop.

The Python API server was iteratively optimized for years; vLLM is now rewriting hot frontend paths in Rust because a Python-first serving layer no longer scales at modern deployment fan-outs. Asynchronous de-tokenization moves incremental decoding off the critical path so that response delivery does not gate the next scheduling decision.

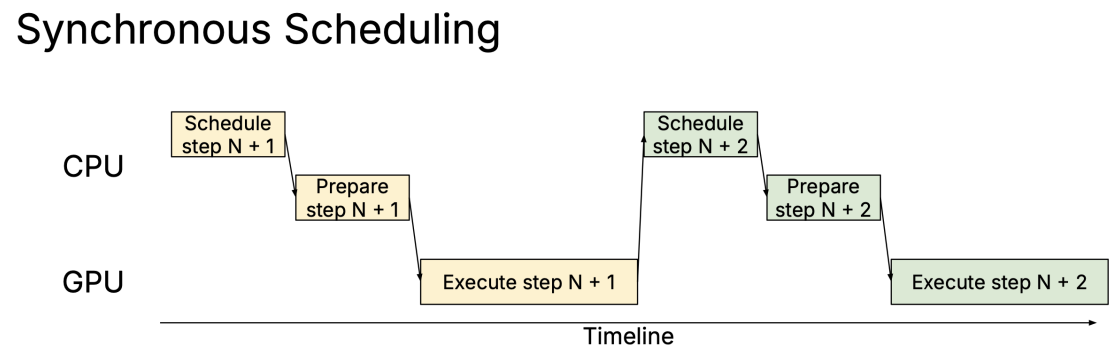
4.3 Engine core: synchronous vs. asynchronous scheduling

"Scheduling" here means two coupled jobs run by the engine core every iteration: (i) a batching decision (which requests join the next forward), and (ii) KV-cache manager work (allocating new page indices, looking up prefix-cache hits, freeing finished blocks).

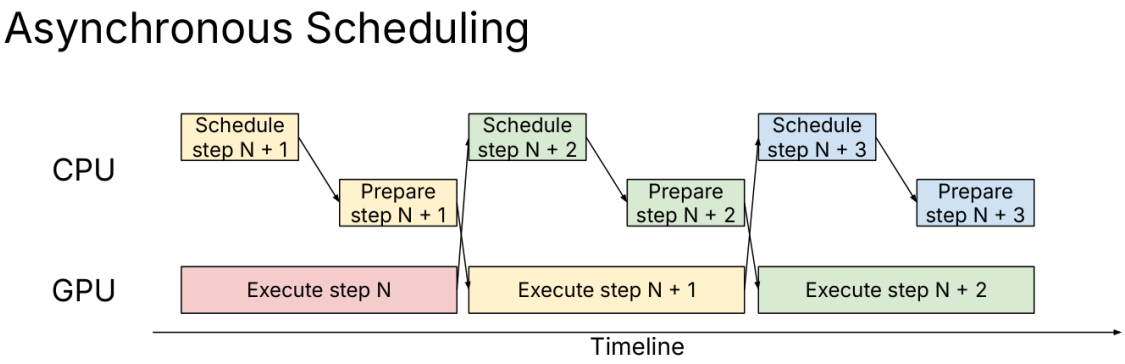
Synchronous scheduling. The original vLLM design ran scheduling and input preparation *after* each forward completed and *before* the next launch (Fig. 1, top). This was natural on A100-class hardware where step latencies amortized the bookkeeping cost, and it has a tidy invariant: scheduling for step $n+2$ uses the actual completion mask of step $n+1$ (stop tokens, EOS), so the engine never wastes a launch on a finished request. The cost is that scheduling sits on the GPU's critical path, idling the device while host bookkeeping runs.

Asynchronous scheduling. Following NanoFlow [1], vLLM now overlaps host work with GPU execution by scheduling *one step ahead*: while the GPU executes step $n+1$, the CPU optimistically

schedules and prepares step $n+2$ under the assumption that no request in step $n+1$ will terminate. Mispredictions (e.g., a request that does emit EOS at step $n+1$) are reconciled by discarding the speculative work for that row at no GPU cost. The stated engineering goal is *zero synchronization*: the CPU should never block on a GPU result inside the steady-state loop. The speaker noted that, although the idea is conceptually simple, achieving full feature coverage (prefix caching, speculative decoding as discussed in Sec. 4.4, early-termination signals) took several months and only landed in vLLM relatively recently.



(a) Synchronous scheduling



(b) Asynchronous scheduling

Figure 1: Synchronous scheduling leaves scheduling and preparation on the GPU-critical path, while asynchronous scheduling overlaps host preparation with GPU execution.

4.4 Worker: GPU-native input preparation

Each step the worker materializes per-request metadata: paged-attention block tables, sampling-parameter slices (temperature, top-p, top-k, presence/frequency penalties), positional offsets, and speculative-batch bookkeeping for verifying drafted tokens. Historically this was a chain of small PyTorch ops executed on the CPU and then copied to device, producing a fragmented timeline of tiny tensors and per-op dispatcher work. vLLM has progressively replaced these chains with *custom Triton kernels* that perform the bookkeeping directly on the GPU. Dispatcher and host-to-device overheads disappear from the steady-state loop, and the speaker noted the Triton implementations are often easier to read than the equivalent forest of `torch.where`/`torch.gather` fragments needed

to keep the logic vectorized on the CPU.

Example (heterogeneous sampling parameters). Concurrent requests in the same batch may specify different top-p thresholds (e.g., 0.5, 0.95, 0.99, or 1.0, i.e., disabled). The worker must track which rows actually require a top-p truncation kernel and skip it entirely when no row needs it. Expressed as CPU-side tensor ops the logic is launch-heavy; a single fused GPU kernel with a per-row mask is both faster and clearer.

Enabling asynchronous scheduling with speculative decoding. Under spec decoding the engine drafts k candidate tokens and a verifier accepts a prefix of them; the *number* of accepted tokens (a result of rejection sampling) is data-dependent and only known after the verifier kernel runs. With CPU-side input preparation, the next step's bookkeeping needed this count, forcing a CPU–GPU synchronization that defeated async scheduling. Because GPU-native input preparation runs on-device, the next step's preparation kernels consume the rejection-sampling tensor *directly* via in-stream dependencies, eliminating the sync point and making async scheduling and spec decoding composable.

4.5 Worker: CUDA graphs and the piecewise refinement

PyTorch eager launches every operator through the dispatcher, paying dtype/shape checks and allocator calls per kernel; the resulting host overhead can occupy up to roughly 50% of per-step wall-clock during single-token decode. CUDA graphs capture a sequence of kernel launches once and replay them via a thin dispatcher, bypassing the Python/PyTorch layers and removing the inter-kernel gaps.

Full CUDA graphs The most aggressive strategy captures the entire forward into a single replayable graph. This is the lowest-overhead option but imposes static shapes (requiring padding or bucketing), forbids CPU operations inside the captured region, and forces the kernel set to be frozen at capture time so any runtime kernel heuristic is unavailable.

Dynamism in LLM inference. Three sources of dynamism that are difficult to capture monolithically: (i) *mixed prefill/decode batches*, where a prefill (many tokens, $\mathcal{O}(L)$ work per row) coexists with several decodes (one token each), making tensor shapes vary across steps; (ii) *kernel heuristics*, e.g., *cascade attention* [3], which chooses whether to share KV-prefix reads across a batch based on the runtime overlap pattern; (iii) *CPU offloading*, where tiered KV-storage paths (Sec. 7) interleave host operations with model execution. Historically the options were unappealing: fall back to PyTorch eager and concede launch overhead, or write a graph-compatible specialization for every dynamic kernel and concede engineering burden.

Piecewise CUDA graphs. In a typical transformer layer, only the attention sub-block is dynamic/stateful; the surrounding operators (QKV projection, RMSNorm, RoPE, output projection, FFN/MoE MLPs) are *token-wise* and stateless, and thus cheaply captured. vLLM splits each layer at the attention boundary using `torch.compile`, inserting an intentional graph break immediately before and after the attention operator. Token-wise pieces run inside CUDA graphs; attention runs in eager PyTorch with full access to its runtime heuristics; the engine alternates between the two regimes per layer.

Table 1: Comparison of model-execution strategies for the decode loop.

Strategy	**CPU overhead**	**Flexibility**	**Engineering burden**
PyTorch eager	High	Full	Lowest
Full CUDA graph	Lowest	Limited (static shapes; no CPU ops; frozen kernel choice)	High: each kernel must be made graph-compatible
Piecewise CUDA graph	Low (slightly above full)	Attention retains full dynamism	Moderate: relies on `torch.compile` graph breaks

On the benchmark shown (input length 4K, output length 50, batch size 1, single H100), piecewise graphs were 6 to 39% faster than PyTorch eager and only 2 to 7% slower than full graphs in the worst case (the batch-1 regime where launch overhead is least amortized). At batch size ≥ 8 the gap to full graphs shrinks to 0 to 2%, while eager remains substantially worse. The pragmatic rule is to use full CUDA graphs when the kernels admit them, and fall back to piecewise whenever any of the dynamism sources above is active.

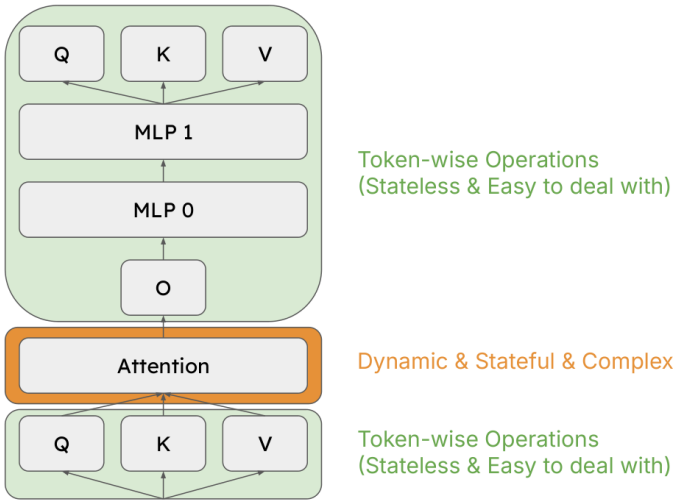


Figure 2: Piecewise CUDA graphs: capture the "easy," largely token-wise blocks while leaving attention in eager PyTorch for flexibility (diagram from the lecture deck).

5 Efficient GPU Kernels

5.1 Motivation

Beyond reducing CPU overhead, vLLM must also make the GPU-side computation itself efficient. Once asynchronous scheduling and CUDA graphs keep the GPU busy, the next bottleneck is whether kernels make good use of hardware bandwidth and compute.

5.2 Hybrid Kernel Strategy

The lecture described a hybrid strategy for kernel optimization. For compute-heavy, performance-critical operators, vLLM integrates specialized external libraries rather than re-implementing every primitive itself. Examples include attention variants handled by **FlashInfer** and matrix multiplications for MoE and MLA layers handled by **DeepGEMM**. To make these integrations modular, vLLM exposes abstractions such as `AttentionBackend` and fused MoE interfaces, allowing kernel implementations to be swapped in without rewriting the serving stack.

5.3 Memory-Bandwidth-Bound Fusion

For memory-bandwidth-bound operators, the optimization target is different. Operations such as RMSNorm, rotary position embeddings (RoPE), and quantization/dequantization have low arithmetic intensity: they read and write large tensors but perform little computation per byte. Running them as separate kernels increases global memory traffic and adds many small kernel launches. vLLM addresses this with `torch.compile`-generated fusion or manually implemented fused CUDA/Triton kernels.

5.4 DeepSeek V3.2 Fused Kernel Example

The DeepSeek V3.2 example illustrates this distinction concretely (Figure 3). The unfused graph contains many small operations for indexing, normalization, RoPE, FP8 quantization, and cache updates across parallel branches. In the fused version, these auxiliary operations are consolidated into five fused kernels, while the core MLA attention via **FlashInfer** and matrix multiplications via **DeepGEMM** remain delegated to specialized libraries. This reflects vLLM's broader engineering pattern: use expert libraries for compute-heavy primitives, and fuse the surrounding memory-bound operations into as few kernel launches as possible.

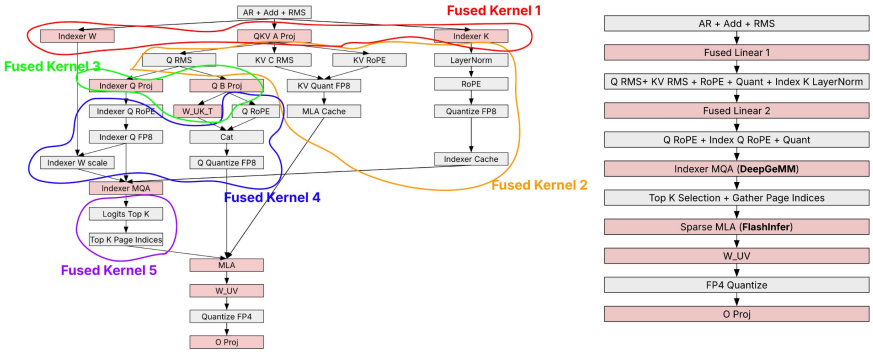


Figure 3: DeepSeek V3.2 fused kernel example. vLLM fuses auxiliary operations while delegating MLA attention and matrix multiplication to FlashInfer and DeepGEMM.

6 Model Parallelism and Serving Topology

6.1 Motivation

Models already exceed single-GPU memory: DeepSeek V4 Pro, for instance, has approximately 1.6 T parameters with a quantized memory footprint exceeding 900 GB, compared against approx-

imately 285 GB of HBM on a single high-end GPU. Beyond parameters, additional memory is consumed by activations (especially during long prefills) and KV caches.

Parallelism supports different goals. Adding hardware can raise aggregate throughput when work is spread across replicas or parallel partitions. It can also reduce per-step latency within a single replica when fast interconnects make intra-replica parallelism practical. These are not the same: scaling out full replicas mainly improves throughput, whereas schemes such as tensor parallelism inside a replica often target latency and time-per-token under strong networking assumptions.

6.2 Design axes

Choosing a parallel strategy reduces to balancing:

- **Communication volume and pattern.** Some partitions require expensive all-to-all or all-reduce collectives; others rely on cheaper point-to-point traffic between neighboring stages.
- **Overlap opportunities.** Whether communication can be hidden behind computation, in the same spirit as overlapping communication and compute in distributed training.
- **Topology heterogeneity.** NVLink within a node is typically much faster than links between nodes, so a partition that is cheap locally can be costly across nodes.
- **Load imbalance.** MoE routing skew, pipeline bubbles, or uneven stage times reduce effective throughput because progress is often limited by the slowest participant.

6.3 Modalities discussed

Data parallelism (DP) Duplicate full engines behind a router and assign different requests to different replicas. Unlike training data parallelism, inference replicas do not synchronize gradients across copies, so steady-state serving avoids training-style collectives across replicas. **Pros:** aggregate throughput often scales roughly linearly with replica count when routing and capacity keep replicas busy. **Cons:** replication does not reduce parameter memory *per* replica, and adding replicas alone does not shorten single-request latency the way intra-replica parallelism can.

Tensor parallelism (TP) Shard linear layers (attention heads motivate natural partitions on transformers). **Pros:** strong latency wins within NVLink-rich servers; KV cache can shard together with attention shards. **Cons:** expensive collectives across slow links; effective tensor-parallel width interacts with GQA/MLA head counts, and models with very few KV heads make TP less attractive because KV layout becomes awkward or effectively more replicated.

Expert parallelism (EP) Spread MoE experts across devices. **Pros:** expert count provides a natural dimension to scale wide MoE models. **Cons:** token routing skew can create severe load imbalance across experts and GPUs.

Context parallelism (CP) Shard very long sequences across devices and exchange portions of queries, keys, and values as needed, which matters as long-context inference grows.

Pipeline parallelism (PP) Assign disjoint layer ranges to different GPUs and stream work through stages. **Pros:** communication per step can be small point-to-point traffic between adjacent stages. **Cons:** bubbles and stage imbalance waste time; latency for an isolated single request can increase relative to a fused single-node path because work is serialized across stages.

Prefill/disaggregation (PD) Inspired by DistServe [2]. Separate prefill workers from decode workers so time-to-first-token (TTFT) and time-per-output-token (TPOT) can be tuned independently. vLLM supports multiple routing backends including its own router, NVIDIA Dynamo, and Ray Serve LLM, with KV blocks transferred between workers using NIXL.

Hybrid compositions Practical clusters often tensor-parallelize within a node where NVLink is fast and use pipeline parallelism or other partitions across nodes where interconnects are slower and higher latency.

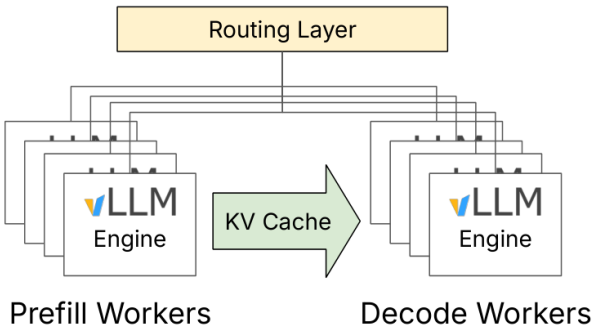


Figure 4: Prefill/disaggregation separates prefill and decode workers, with KV cache movement between them.

6.4 Pareto Trade-offs on Real Clusters

Empirical throughput versus latency curves for DeepSeek V4 Pro on GB200-class hardware illustrate how parallelism choices shift depending on the operating point. Two metrics capture different priorities:

- *Interactivity* (output tokens/s *per user*), which reflects perceived responsiveness.
- *Aggregate throughput* (total output tokens/s *per GPU* summed over concurrent users), which reflects hardware utilization and therefore cost per token when GPUs stay saturated.

Low-latency points favored decode tensor parallelism with tensor-parallel degree eight (TP8) because NVLink-heavy collectives shorten per-step decode time, even though KV heads replicate across TP ranks and hurt memory efficiency when the model exposes only one KV head (MLA-style designs). High-throughput points favored eight-way data parallelism among replicas (DP8): lighter cross-device traffic per replica and no KV replication within each replica. In practice, engineers trace a Pareto frontier between these extremes rather than choosing one universally "best" parallelism, depending on interconnect topology and SLA emphasis.

7 Efficient Memory Management and KV Cache Offloading

7.1 Hybrid attention and the limits of PagedAttention

KV cache management in vLLM started with PagedAttention, which divided GPU memory into fixed-size blocks analogous to OS virtual memory pages. That design assumed all layers in a model use the same attention type. Recent architectures break that assumption by mixing attention variants within a single model. GPT-OSS interleaves full GQA attention with sliding-window attention at roughly a 1:1 ratio; Qwen 3.5 uses gated delta-net, a form of linear attention, as the dominant type alongside full attention at about a 3:1 ratio. Managing these hybrids means running two fundamentally different allocators at once: one for KV cache blocks that grow with context length, and one for recurrent states that are fixed-size per sequence. The two differ in allocation granularity and eviction semantics.

7.2 Static partitioning and its failure mode

The straightforward response is *static partitioning*: reserve a fixed memory region for each attention type and let each manage its own pool independently. The speaker gave a concrete example: reserve one pool for gated delta-net states and a separate pool for full-attention KV cache. If traffic skews toward one type, that pool fills up while the other sits mostly empty, and the engine stalls despite having aggregate free memory available. The speaker called this a "severe memory fragmentation issue."

7.3 Dynamic hybrid allocator

vLLM replaces static partitioning with a shared memory pool that all attention types draw from, with the allocator adjusting block granularity at runtime for each type.

The two allocators expose different interfaces. The full-attention allocator works in token-count units, for example allocating 1024 tokens at a time, consistent with paged-attention semantics. The linear/gated-state allocator (used for gated delta-net layers) allocates per sequence, since the recurrent state does not grow with context length. Making both coexist over one backing store with compatible eviction logic took considerable engineering work; the speaker described it as requiring "a few clever tricks."

The result is 0–12% memory waste across all tested open-source hybrid models. The speaker was candid that the 12% figure is not ideal, but said it was the best the team could achieve given the range of architecture combinations in the wild. Full implementation details are in the SOSP paper from the vLLM team.

7.4 KV cache offloading

Motivation: Agentic workloads produce very long multi-turn traces. A single trajectory involving tool calls can span dozens to hundreds of turns, and the cumulative KV cache grows too large to

keep in GPU memory for the whole session. Recomputing from scratch on every turn is expensive, so vLLM extends KV cache storage to CPU RAM and, experimentally, to disk.

Mooncake integration: vLLM integrates with Mooncake, a distributed KV cache storage and transfer engine. When a request finishes, vLLM writes its KV blocks into Mooncake, which shards them across CPU-memory nodes in the cluster. When a later request arrives with a matching prefix, vLLM queries Mooncake; on a hit the blocks are transferred back to the prefill worker over DMA and the prefill phase skips those tokens.

The design uses a **MultiConnector** abstraction that chains connector backends together. A **MooncakeStore Connector** talks to the Mooncake distributed KV cache pool while a **PD Connector** handles transfers between prefill and decode instances; these are peers in the chain rather than a hierarchy. KV transfer between nodes uses NIXL in PD-disaggregated deployments, or multi-node NVLink/RDMA otherwise.

Selected Q&A Highlights

Q: Is there any practical "magic box" for estimating rollout length to improve scheduling?

Predicting output lengths can improve scheduling and packing, but remains hard to generalize across domains. Collecting traffic traces and training a ranker or regressor per workload is a reasonable heuristic, though vLLM does not yet have a built-in universal predictor for this.

Q: Would the piecewise CUDA graph approach still work for dynamic-depth models like looped transformers, or does the variable compute depth require a different execution strategy?

Variable iteration counts make it tricky to define clean capture boundaries. One reasonable approach would be to capture graphs at the per-layer granularity and handle the looping logic externally, which is in the same spirit as the piecewise strategy but adapted for dynamic depth.

Q: How does vLLM reduce the overhead between the CPU scheduler and the GPU workers?

A few techniques work together here: async scheduling overlaps the next scheduling step with the current GPU execution, bookkeeping logic has been moved onto the GPU via custom Triton kernels, CUDA graphs eliminate Python and PyTorch dispatch overhead between kernel launches, and the API serving layer has been rewritten in Rust to reduce per-request overhead at scale.

Q: Is there any attempt to address the training–inference mismatch in RL?

Some mitigations exist today. Batch-invariance is supported for certain quantized kernels, and routed expert indices can be exposed to the training framework so that training forward passes can mirror the exact routing decisions made during inference.

8 Conclusion

vLLM illustrates how LLM serving systems succeed through stacked marginal gains: shaving CPU control-path overhead, careful kernel fusion, parallelism mixes tuned to hardware hierarchies, and memory managers that evolve with hybrid architectures. The system continues to be actively developed as an open-source community effort, with ongoing work spanning new model architectures, hardware backends, and serving topologies.

References

[1] B. Zhu et al. NanoFlow: Towards Optimal Large Language Model Serving Throughput. *arXiv preprint*, 2024.

[2] Y. Zhong et al. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. *Proceedings of OSDI*, 2024.

[3] Y. Ye et al. Cascade Inference: Memory Bandwidth Efficient Shared Prefix Batch Decoding. *arXiv preprint*, referenced in talk slides.