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

16: Guest Lecture

Lecturer: Dr. Zihao Ye *Scribes:*

Kailey Wong, Matthew Yang, Jason Weitz, Haoxiang Zhang, Meshal Nayim, Abhishek Asi, Yuting Zhou

1 Introduction

This lecture was given by Dr. Zihao Ye, a Senior Compiler Engineer at NVIDIA, where he works on machine learning systems with a focus on efficient LLM inference and ML compilers. He is the lead author of FlashInfer, which is an attention engine for LLM inference that is now adopted by major serving frameworks including vLLM, SGLang, and MLC-Engine. He is also a recipient of the 2024 NVIDIA Graduate Research Fellowship. The lecture centered around libraries, agents, and how to develop high-performance kernels for generative AI models.

2 FlashInfer: Efficient and Customizable Kernel Generation for LLM Inference Serving

2.1 The Motivation for FlashInfer

When FlashInfer was developed in 2023, there was a lack of good kernel libraries for attention, and more specifically, LLM inference. At the time, FlashAttention was only designed for training, and lacked optimizations like KV caching. There are many challenges with handling KV cache management in different complicated environments. For example, different dimensions or methods of calculating attention (e.g. adding epilogue, prologue). There are also more prefix caching opportunities now for long system prompts, which helps to increase the operational intensity of kernels.

2.2 Background: Attention and KV Cache Management

The main goal is to provide continuous batching for many users in a serving setting where KV cache may be corresponding to different lengths. The system should have flexibility in how it organizes the cache without large maintenance overhead. Paged KV cache is a natural method for organizing the cache, mimicking the paging mechanism in operating systems. One of the biggest challenges in writing an efficient kernel for PagedAttention is that the KV cache is no longer contiguous. Existing kernels aren't built with this consideration, so new kernels that are developed must handle non-contiguous memory access. Memory access is the main bottleneck, so kernels should avoid reading multiple times and directly read into the non-contiguous KV cache. For Radix Attention (sglang), prefixes are organized as a Radix Tree. This data structure helps with prefix- caching and reuse by grouping prefixes together. A page size of 1 can be used to reduce fragmentation and improve the cache hit rate. Larger page sizes reduce the opportunity to reuse the KV cache. Another popular form of attention is sparse attention (e.g. Quest). The first step in sparse

attention is to compute importance for keys and queries with a coarse-grained mask calculation. The top k KV pairs get passed as a sparse mask that is used to perform attention. Considering all of these challenges, FlashInfer uses a KV cache layout called Block Compressed Row (BSR) format. This format has a good balance of sparsity and is able to work well with hardware acceleration units. There is a minimal block size, usually a small square (e.g. 4x4, 16x16). When considering the KV cache, PagedAttention essentially performs block sparse attention. The logical to physical mapping corresponds to the non-zero column index of queries and their requests. Larger block sizes are able to achieve better performance because they are able to utilize tensor cores more efficiently. Radix Tree and Token Tree are sparse matrices by nature. However, tree masks can still be highly sparse. This may lead to high fragmentation in block-sparse representation. Research has found that column vector sparse representations are sufficiently effective to reduce fragmentation. Multiplying sparse rows by sparse columns is still compatible with Tensor Cores as long as the last dimension is contiguous. Even if a mask is col sparse, after gathering global non-contiguous rows in the query matrix and non-contiguous columns in the key matrix to faster on-chip shared memory, the computation becomes dense matrix multiplication at the on-chip level. In FlashInfer, this required designing a specialized memory copy for non-contiguous global memory to contiguous shared memory. When implemented, calculations are then similar to ordinary FlashAttention using dense tensor cores. There is great opportunity to use shared prefixes in reasoning algorithms, for example when system prompts or agents are used. The shared prefix is essentially a larger block in the block sparse matrix. The attention mask can be decomposed in the following way: some computations use a larger block size for shared prefix, while a smaller block size is used for the unique suffix portion. Attention can be performed for these two kinds of block sparse matrices separately, then their results can be mathematically combined by a joined kernel. This allows for 2 separate kernels: one that uses a larger block size and can pre-load a shared KV cache to faster shared memory and use tensor cores, the other uses standard attention for the unique KV cache.

3 Attentions in LLM Serving – KVCache Management

The lecture first discusses KV-cache compression methods such as Quest. For long-context inference, the current query often does not need to attend to every page of the KV cache. The example on the slide uses a two-stage approach. In Stage 1, the system computes an importance mask by estimating which KV-cache pages are likely to be critical for the current query. Each page can store reduced information, such as element-wise minimum and maximum keys, so the system can cheaply score pages. It then keeps only the top-K important pages. In Stage 2, the attention kernel computes sparse attention using that mask, skipping pages that are predicted to be unimportant.

A key observation is that paged attention can be viewed as a form of block-sparse attention. In paged attention, each request has a logical sequence of KV-cache pages, and the runtime maps those logical pages to physical KV-cache blocks. If we treat query tiles or requests as rows and physical KV-cache pages as columns, then the page table specifies which blocks are nonzero. Under this view, KV-cache management and sparse matrix computation become closely related: the logical-to-physical page mapping is also the sparse index structure for the attention computation.

4 KV-Cache Storage

FlashInfer stores sparse KV-cache layouts using a block sparse format. Instead of storing individual scalar nonzeros, a block sparse matrix stores small dense tiles as the unit of sparsity. The lecture describes a BSR, or block compressed row, format with block shape (B_r, B_c). For example, with block size $(B_r, B_c) = (4, 2)$, each nonzero block represents a dense 4×2 tile. Metadata such as `indptr` and `indices` record which block columns appear in each block row, while the values store the dense content of the selected blocks.

This representation is more GPU-friendly than scalar sparsity. Tensor cores are designed for matrix-tile operations, not isolated scalar operations. A block-sparse format therefore gives the hardware enough dense work inside each selected block. In paged attention, the page size corresponds to the column block size B_c, and the query tile size corresponds to the row block size B_r.

The lecture also emphasizes a tradeoff in block size. Larger blocks usually improve tensor-core utilization because they create more dense work, but if the attention mask is very sparse, large blocks may contain many unused entries. Smaller blocks reduce wasted computation but may not map well to tensor-core tile shapes. FlashInfer handles this by using a gather-then-compute pattern. It gathers sparse rows or columns from global memory into shared memory, where the selected data becomes dense, and then uses tensor cores on the dense shared-memory tiles.

5 Sparse Global-to-Shared Gathering

The sparse gather stage is the bridge between irregular memory layout and regular tensor-core computation. In global memory, selected K/V pages may be non-contiguous because of paging, sparse masks, radix-tree cache layouts, or other cache-management structures. FlashInfer first loads these selected pieces into contiguous shared-memory buffers. After that, the computation looks much closer to ordinary dense FlashAttention.

The workflow is: identify needed KV-cache blocks using page tables or sparse indices; copy selected global-memory regions into dense shared-memory tiles; run tensor-core operations on the shared-memory query and key tiles; then finish the rest of attention, such as logits transformation, masking, softmax, and value aggregation. This separates the irregular part of the problem from the regular compute part.

Sparse gathering is difficult because GPUs prefer coalesced memory access, where adjacent threads read adjacent memory locations. Contiguous K/V storage can use larger tile copy instructions. Sparse K/V storage may need multiple smaller asynchronous copies, such as `LDGSTS.128B`, because the data comes from different physical locations. The purpose is still the same: transform sparse global memory into dense shared memory. The lecturer also noted that newer hardware may make gather/scatter-style transfers easier, which shows how common LLM serving workloads can influence hardware support.

6 Composable Formats

Many LLM serving workloads contain shared prefixes. For example, many users may ask questions about the same long textbook or document, so their prompts share a long prefix but differ in the suffix. Reasoning and agent workloads also often reuse the same system prompt or tool descriptions. In the sparse-matrix view, the shared prefix appears as a large common block accessed by many requests, while the unique suffixes are smaller and more irregular.

FlashInfer can decompose the attention mask into multiple formats. It can use a small block size for the unique KV-cache part, where accesses are irregular, and a larger block size for the shared-prefix part, where many requests access the same data. The shared-prefix region can then be loaded into faster memory such as shared memory or registers and reused across requests. This improves throughput because repeated reads of the common prefix are amortized. The partial attention results over the shared and unique regions can be combined later, using the same kind of online reduction idea that appears in FlashAttention.

7 Attentions in LLM Serving – Variants

Modern serving systems need to support many attention variants, including grouped-query attention (GQA), multi-latent attention (MLA), Grok/Gemma-style logits soft cap, ALiBi bias, and other model-specific changes. Many of these changes are mathematically small. For example, ALiBi adds a position-dependent bias to the attention logits, and a soft cap applies a nonlinear transformation to logits before softmax. However, in a high-performance kernel library, even a small formula change can require a new kernel if everything is hard-coded.

This creates a maintainability problem. It is not sustainable to manually implement and ship a separate optimized kernel for every possible attention variant. The library needs a flexible interface for users while still generating specialized GPU code that is fast.

8 Compute Abstraction – Compiler & Runtime

FlashInfer solves this through a compute abstraction with two parts: a JIT compiler and a runtime scheduler. The JIT compiler handles information that is known at compile time, such as data type, head dimension, sparse versus continuous KV-cache storage, tile choices, and the specific attention variant. The runtime scheduler handles information that changes from batch to batch, such as sequence lengths, page mappings, and work distribution across CTAs.

The JIT compiler lets users define small functions such as `QueryTransform`, `KeyTransform`, `LogitsTransform`, and `LogitsMask`. These functions customize the attention behavior but are inserted into a kernel template so the final code is still compiled and specialized. For example, the slide shows a sigmoid attention variant,

$$\text{SigmoidAttn}(X) = \sigma\left(\frac{QK^T}{\sqrt{d_{qk}}}\right)V, \qquad \sigma(u) = \frac{1}{1+e^{-(u+b)}}.$$

Instead of writing an entirely new library by hand, a user can express the logits transformation and let the JIT system generate the corresponding kernel.

This template-based design reduces the need to ship all possible kernels in advance. It also helps with binary size and compilation time, because the system only compiles the variants that are actually needed by the current application.

9 Attention Template – Group Query Attention

Grouped-query attention creates a special challenge during decoding. In incremental decoding, each request may contribute only one query row at a time, which is too small to efficiently use tensor cores. If each group is processed separately, the kernel may not have enough rows for good tensor-core utilization, and different groups may not share data effectively.

FlashInfer uses head-group packing. The idea is to fuse query rows and head groups so the effective number of rows increases. If the GQA group size is g and the original query/output length is l_{qo}, then the fused query length is

$$l'_{qo} = g \cdot l_{qo}.$$

The fused row index i' maps back to the original row index by

$$i = i'/g,$$

and the query/output head index can be computed as

$$\texttt{qo\_head\_idx} = \texttt{kv\_head\_idx} \cdot g + (i' \bmod g).$$

This packing increases operational intensity and helps decoding or incremental prefill kernels better match tensor-core tile sizes.

10 Runtime Scheduler and Integration

The runtime scheduler handles variable-length requests. In real serving batches, some requests have long KV caches while others are short. A naive mapping can create stragglers, where some CTAs receive much more work than others. FlashInfer uses a cost-model-based deterministic scheduler. It inspects sequence lengths ahead of time, splits work into tiles, dispatches those tiles to CTAs, and records a reduction map so partial outputs can be combined into final outputs.

The goals are load balancing and zero wave-quantization. Load balancing means CTAs should receive similar amounts of work. Zero wave-quantization means avoiding wasted execution waves due to uneven work counts.

The final compute abstraction follows an inspector-executor pattern: during `init`, kernels are JIT-compiled for each attention configuration; during `plan`, the scheduler builds a deterministic execution plan; and during `run`, kernels execute according to that plan. This design is compatible with LLM serving engines, `torch.compile`, and CUDA Graphs, which are important for reducing overhead in production inference.

11 Takeaways

The main takeaway is that efficient attention serving is a joint storage, compiler, and runtime problem. Paged KV caches can be interpreted as block-sparse attention patterns. Sparse global memory can be gathered into dense shared-memory tiles so tensor cores remain useful. Shared prefixes require composable formats because different KV-cache regions benefit from different block sizes. Finally, JIT templates and runtime scheduling allow FlashInfer to support diverse attention variants and variable-length serving workloads without giving up performance.

12 Sparse gathering for non-contiguous KV cache

Modern LLM serving systems such as vLLM and SGLang often store the KV cache using paged or radix-tree layouts to support continuous batching, prefix reuse, and variable sequence lengths. These approaches improve memory management and allow shared prefixes to be reused across requests, but they also create highly non-contiguous memory access patterns. GPUs generally prefer contiguous, coalesced memory accesses, so directly applying standard dense attention kernels to these layouts becomes inefficient. FlashInfer addresses this by treating KV-cache access as a sparse matrix problem. Instead of requiring all KV-cache entries to be contiguous, the kernel gathers only the relevant rows and columns from global memory into

shared memory. Although the accesses are sparse globally, once the data is loaded into shared memory, the attention computation itself becomes dense again. This allows tensor cores to still be used efficiently for the matrix multiplications involved in attention.

To support this efficiently, FlashInfer uses block-sparse and vector-sparse representations. GPUs operate most efficiently on matrix-shaped blocks, so organizing sparse data into blocks helps preserve tensor-core utilization. However, highly sparse attention patterns, such as speculative decoding trees or radix attention, would waste computation if large dense blocks were enforced everywhere. Instead, FlashInfer uses sparse gathering operations that load only the necessary data into shared memory while still performing dense computation locally. A major focus is reducing the cost of these irregular memory accesses. GPUs prefer coalesced accesses where neighboring threads load neighboring memory locations, since this aligns with GPU cache-line behavior. Earlier implementations relied on many smaller gather instructions, but newer Blackwell GPUs now provide more native hardware support for sparse gather/scatter operations through updated TMA instructions. As a result, the performance gap between paged attention and fully contiguous attention layouts continues to shrink.

13 Runtime scheduling for variable-length requests

A major challenge in LLM inference is that different user requests can have very different sequence lengths, which creates highly imbalanced GPU workloads. Some thread blocks may finish quickly while others continue processing much longer sequences, leading to idle GPU resources and reduced utilization. This effect, referred to as wave quantization, becomes especially important during decoding where latency is critical. To address this, FlashInfer introduces a runtime scheduling system that splits requests into smaller chunks and dynamically distributes work across GPU units to keep workloads balanced. The scheduler creates work queues that attempt to evenly distribute computation across streaming multiprocessors (SMs), reducing situations where some GPU units remain idle waiting for others to finish.

The talk also discussed the tradeoff between static and dynamic scheduling approaches. Static scheduling performs most of the planning on the CPU ahead of execution, which minimizes runtime overhead but can lead to imbalance if workload estimates are inaccurate. Dynamic scheduling instead makes decisions during execution on the GPU itself, improving adaptability but introducing additional overhead and register usage. Since GPU registers are a limited resource, this creates a balance between scheduling flexibility and kernel efficiency. CUDA Graphs were also used to reduce kernel launch overhead by capturing repeated execution patterns ahead of time. Combined with runtime scheduling, this helps improve efficiency for highly dynamic inference workloads with varying sequence lengths and request patterns.

14 Fused Kernel for Mixed Prefill/Decode Attention

Sarathi-Serve introduced the idea of piggybacking decode requests alongside prefill requests to reduce waiting time, this is also called chunked prefill. The kernel-level challenge is that prefill and decode have fundamentally different characteristics. Decode has a query length of 1, while prefill can have thousands of rows. If you launch two separate kernels, each suffers from wave quantization and low occupancy. But if you use a single kernel with a fixed tile size, it's suboptimal for one or the other. PodAttention solves this by writing a single kernel that contains both a large tile path for prefill and a small tile path for decode. At runtime, the kernel branches depending on the workload type, prefill takes the large tile branch, decode takes the small tile branch. This addresses the tile size mismatch for mixed prefill/decode attention. PodAttention was integrated into FlashInfer.

15 Global Barrier

In the naive approach, mixed prefill/decode attention requires launching three kernels sequentially, prefill, decode, and reduce (where reduce merges partial KV-split results). Each kernel must wait for the previous one to fully finish before the next can start. This is the global barrier constraint imposed by the GPU execution model. The consequence is that wave quantization bubbles in prefill force you to wait before starting decode, and similarly decode must fully finish before reduce can begin. This sequential kernel-by-kernel launch means idle SM time accumulates across all three stages.

16 Persistent Kernel + Block-wise Barrier

The solution is to write a single persistent kernel that contains all three operations prefill, decode, and reduce instead of launching them one by one. Within this kernel, synchronization is handled not by global barriers but by task-level (block-wise) barriers. The key insight is that reduce only needs to wait for its specific upstream decode tasks to finish, not for every CTA in the entire kernel. This finer-grained dependency tracking eliminates the unnecessary waiting imposed by global barriers, reducing the idle bubbles between stages.

17 Performance: Chunked Prefill

When comparing the separated kernel approach versus the persistent kernel approach, there is a significant performance gap especially when the prefill-to-decode ratio is high around 0.9 to 0.95. In those regimes, the overhead of global barriers and wave quantization bubbles between kernels becomes substantial, and the persistent kernel with task-level barriers measurably closes that gap.

18 Generalization: Mega Kernels

This idea generalizes beyond just three kernels. The term MegaKernel, originally from graphics roughly a decade ago, was picked up by the ThunderKittens project and refers to fusing not just a few operations but potentially an entire transformer layer, or even the full transformer, into a single kernel. The core problem with kernel-by-kernel execution is the global barrier between every kernel launch. CUDA Graphs reduce the kernel launch gap but wave quantization between kernels still persists. MegaKernel restructures execution at a finer granularity so that each task only waits on its actual dependencies rather than on a global kernel-level barrier.

This introduces a scheduling problem. How do you order task execution to minimize bubbles and balance load? There are two approaches. Static scheduling pre-computes task order on the CPU leads to no on-chip overhead, but the CPU-side execution time estimates are imprecise, which can lead to severe load imbalance. Dynamic scheduling computes the next task on-the-fly on the GPU which is more accurate, but requires on-chip registers for the scheduler, which are a precious resource (each thread has 256 registers), creating register spill pressure. Neither is definitively better today, but the speaker expects future hardware to include dedicated units for real-time scheduling which could be a hardware separation of data plane and control plane.

19 Event Tensor Abstraction for Task Programming

The speaker's team at Nvidia, collaborating with CMU, developed an abstraction called the Event Tensor for mega-kernel programming. The idea is to model all data dependencies between tasks as a tensor, where rows and columns represent different task types. Each cell in the tensor holds a counter essentially a semaphore initialized to the number of dependencies that task must wait on. Each time a dependency signals completion, the counter is atomically decremented by one. When it hits zero, the corresponding task is unblocked and proceeds.

It's called a "lazy" abstraction because the dependency structure is expressed as a tensor, which naturally fits into existing tensor compiler abstractions. Compared to traditional GPU signal/barrier primitives, the advantage is that the dependency graph can be dynamic. A task can have a variable number of dependencies resolved at runtime. This is particularly useful for cases like top-K dependencies in MoE layers, where the dependency count isn't statically known. The tradeoff is that maintaining the event tensor consumes on-chip registers.

20 Major Operators in Large Language Models

Dr. Ye highlighted four major operator categories that dominate modern large language model workloads: attention, GEMM (general matrix multiplication), collective communication, and logits post-processing. A key challenge is that production-scale kernel development does not scale well with increasing model complexity. Developing highly optimized kernels remains time-consuming, often requiring months of effort even from experienced kernel engineers.

To address this challenge, the FlashInfer team developed `FlashInfer-Bench`, a benchmark and dataset designed for kernel generation and evaluation. The motivation was twofold: existing benchmarks contained many outdated workloads, such as convolution-heavy models, and they often failed to capture the tensor shapes and execution patterns seen in real LLM inference. FlashInfer-Bench addresses this by collecting real workloads and representing them as problem schemas paired with input tensors. These traces can then be used both for benchmarking kernels and as training environments for coding agents.

When FlashInfer was first developed, collecting and curating these workload traces required months of effort from a team of researchers. Today, advances in coding agents have significantly accelerated this process, reducing workload collection from months to only a few hours.

21 Kernel Evolution: Matching Human Performance

FlashInfer-Bench also serves as the foundation for an iterative kernel-evolution workflow. The process begins by defining a set of target kernels and then using coding agents to generate equivalent implementations in pure CUDA. Because FlashInfer is maintained as a centralized software stack, optimizations discovered for one kernel can often be transferred and reused across others, allowing improvements to propagate throughout the library.

The workflow is further supported by agents that monitor workloads, benchmark results, and emerging optimization techniques. Community competitions on challenging problems such as DeepSeek Sparse Attention and Gated DeltaNet have demonstrated the effectiveness of this approach. According to Dr. Ye, modern coding agents are increasingly capable of generating highly optimized kernels and, in some cases, can match or exceed human performance in both development speed and kernel quality.

22 Agents in Kernel Development

22.1 Why kernel development no longer scales

FlashInfer now lives at NVIDIA and serves as the kernel-collective layer for LLM inference engines (vLLM, SGLang), shipping four broad kernel categories: attention, GEMMs (especially grouped GEMMs for MoE), communication, and sampling. At production scale this development model has stopped scaling, for two reasons:

- Exploding demand for new data types. Each new precision format (MXFP8, NVFP4, etc.) requires rewriting every operator against the new acceleration units.
- Architectural complexity per kernel. A well-tuned kernel for Blackwell takes months even for experts.

The bottleneck is therefore the supply of human kernel experts, and the proposed response is to embrace coding agents across the entire development cycle.

22.2 FlashInfer-Bench

FlashInfer-Bench turns FlashInfer into a benchmark for LLM coding agents writing inference kernels.

Two gaps in the prior KernelBench effort motivate it:

1. KernelBench's problem set is partly outdated (heavy on convolutions) and ignores low-precision data types; FlashInfer itself contains no FP32 kernels.
2. KernelBench problems do not reflect the actual tensor shapes encountered in LLM inference workloads.

To address the second point, a workload-collection pipeline hooks into vLLM and SGLang and traces real inference workloads on current models (DeepSeek, Qwen, MiniMax, and others), recording both the problem schema (shapes, dtypes, and constraints such as "must use tensor cores at the required precision") and real input tensors. The resulting FlashInfer trace dataset serves both as an evaluation set and as a training environment for kernel-writing agents. Workload collection that originally required undergraduate effort spread over one to two months is now expected to take hours with current agents.

22.3 The Kernel Evolution Loop

The benchmark in turn serves as the substrate for an evolutionary loop that continuously improves FlashInfer's kernels:

- A set of reference targets — state-of-the-art kernels, possibly written in different DSLs or wrappers — defines the performance bar.
- Coding agents generate matching implementations in pure CUDA plus inline PTX.
- Multiple data sources — internal and public documentation, plus tool access — guide the search.

Two properties of this loop matter. First, because FlashInfer is a single centralized library, optimization tactics discovered for one kernel can be memorized as reusable skills and applied to others, producing broad performance improvements across the kernel set. Second, the loop is self-feeding: agents that monitor open-source activity can automatically open pull requests in FlashInfer-Bench when new workloads or optimizations appear, after which kernel-writing agents take over the implementation, with end-to-end evaluation agents as the natural next step.

An early-2026 Kernel Generation Contest organized by the FlashInfer team around deliberately hard problems — DeepSeek V3.2 sparse attention and gated DeltaNet — drew strong community submissions. The speaker's headline claim is that, as of May 2026, coding agents have surpassed humans across essentially every aspect of kernel development in both speed and capability.

23 Toward a Lean GPU Software Stack in 2026

The current GPU software stack is heavily duplicated at every layer: multiple serving engines (vLLM, SGLang, and others), multiple kernel libraries (FlashInfer, cuDNN, vendor-specific stacks), and multiple DSLs (Triton, CUTLASS, and emerging alternatives). For a small team — a startup with a handful of strong engineers — choosing among these is increasingly confusing.

The proposed reframing is to treat each of these systems as a starting point rather than as a finished product. In one experiment, Claude Code, given the vLLM (or SGLang) codebase and a specific workload-plus-hardware target, produced a minimal viable serving stack for that target in "less than a few hours." Consistent with this view, FlashInfer itself is now described less as a piece of software than as a dataset that drives the evolution of agent-written kernels.

24 Selected Q&A Highlights

Q: Multi-agent serving frameworks are increasingly common (with recent work on cache-eviction logic specifically for that case). Are new kernels needed to support them?

Probably not, pending empirical investigation of how vLLM and SGLang currently handle the multi-agent cache-eviction pattern. The expectation is that existing PagedAttention plus appropriate mathematical transformations should suffice.

Q: Is the agent-driven approach to kernel writing applicable to training kernels, not just inference?

Yes; this is the direction of ongoing work at NVIDIA. Training–inference consistency makes it natural to treat the two together rather than in isolation; the inference framing in the talk reflects FlashInfer's historical origins rather than a principled separation.

Q: Does FlashInfer provide omni-specific optimizations for multimodal workloads, and what are the main kernel-level differences from text-only LLM serving?

FlashInfer has not specifically prioritized omni, in part because multimodal effort at NVIDIA is dispatched across many engineers and is not centrally coordinated — a recurring motivation for the broader agent-driven program. The main differences from text-only inference are (i) text inference is dominated by autoregressive, causal attention, which does not necessarily fit multimodal, and (ii) multimodal workloads tend toward very long contexts, where sparse attention may eventually become important.

Q: Page size = 1 improves KV-cache reuse but introduces fine-grained memory access. How is this tradeoff balanced in latency-critical production systems?

The page-size-1 argument for cache reuse originated with SGLang. NVIDIA's perspective is largely the opposite: larger page sizes are easier to optimize and are friendlier to DMA engines, which matters particularly when KV cache is offloaded. From a hardware standpoint, larger pages win.

Q: If agents surpass humans in kernel development, what role remains for humans?

Two near-term roles. First, hardware–software co-design: today's hardware is taped out against a limited workload set, producing mismatches such as Blackwell B200's insufficient exponential-unit bandwidth (which FlashAttention 4 worked around with software-simulated exponentials, a trick that does not generalize to B300). If agents can rapidly produce a working software stack on day one of a new architecture, the feedback loop from software to hardware design tightens significantly. Second, and more immediate: verifying agent-written kernels to ensure they are not exploiting evaluation loopholes.

Q: Blackwell is heavily biased toward TMA and gather-scatter patterns optimized for FlashAttention and PagedAttention. Does this lock the ecosystem into Transformer-style attention? And can Event-Tensor abstractions bridge the compiler-level gap for Gated DeltaNet, or are such architectures fundamentally hardware-incompatible?

On lock-in: no. Matrix multiplication is the most hardware-friendly primitive, and even purpose-built sparse chips have historically failed to match dense GPUs on operational intensity. The operative principle is to maximize matrix-multiplication content in the workload rather than to assume Transformer attention specifically.

On compiler abstractions: no current design truly bridges the gap; each paper contributes inspiration rather than a solution, and the better path is for the compiler design itself to be driven and refined by agents rather than by human-imposed structure.

On fundamental incompatibility: also no. The inverse operation in Gated DeltaNet, for example, may admit a hardware-efficient approximation in the same spirit as the approximate exponential used in FlashAttention. A fruitful direction is auto-research over both kernel implementations and model architectures aimed at efficiency on existing hardware.

Q: What software stack would you choose if you were a startup founder?

Dr. Ye discussed the tradeoffs involved in choosing a software stack for AI systems. Large, highly integrated software stacks can become difficult to maintain because they are often optimized for specific hardware configurations and deployment settings. As these systems grow, bug fixes may require releasing an entirely new software version, forcing users to either upgrade or remain on a potentially buggy release.

Instead, he advocated for a more modular approach based on small, specialized kernels with minimal dependencies. Such components are easier to maintain, update, and optimize independently. Looking forward, he suggested that AI software stacks may evolve toward collections of lightweight kernel libraries rather than large monolithic frameworks.