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

Lecture 15: KV Cache Management and Flash Attention

Lecturer: Hao Zhang *Scribe: Manas Jain, Tamoghno Kandar, Satyam Srivastava, Jason Zhu, Jay Chaudhary, Divyansh Srivastava, Jason Kong*

1 Overview

This lecture covers two major topics in efficient LLM inference and training:

1. **PagedAttention and vLLM**: An OS-inspired approach to managing KV cache memory efficiently during LLM serving, enabling high-throughput inference by eliminating memory fragmentation.
2. **Flash Attention**: An exact attention algorithm that avoids materializing the large $N \times N$ attention score matrix, dramatically reducing memory usage and improving speed for both training and inference.

The lecture also briefly discusses upcoming topics: linear/sub-quadratic attention variants (to be introduced by a guest lecturer in the next class), and several advanced LLM serving techniques such as disaggregated prefill/decode and chunked prefill.

2 Motivation: KV Cache and Memory Bottlenecks

2.1 Why KV Cache Matters for Throughput

In autoregressive LLM inference (the decode phase), each new token must attend to all previously generated tokens. To avoid recomputing key and value projections for all prior tokens at every step, frameworks cache these tensors — referred to as the *KV cache*. The KV cache is the primary consumer of GPU memory beyond model weights. For a 13B parameter LLaMA-style model [7] on an NVIDIA A100-40GB GPU, roughly:

- **26 GB (65%)** is consumed by model weights.
- **13 GB (33%)** is available for KV cache. Note that a single token state (one k vector and one v vector across all layers) occupies approximately **820 KB** of memory in LLaMA-13B.

The central insight is: *increasing the number of requests processed in parallel (batch size) directly translates into higher GPU throughput*, because the GPU's many streaming multiprocessors (SMs) can be kept busy. However, as shown in Figure 1, the number of concurrent requests is tightly bounded by how efficiently we can fit their KV caches into limited GPU memory. If memory is wasted, we cannot admit more requests and the GPU remains underutilized.

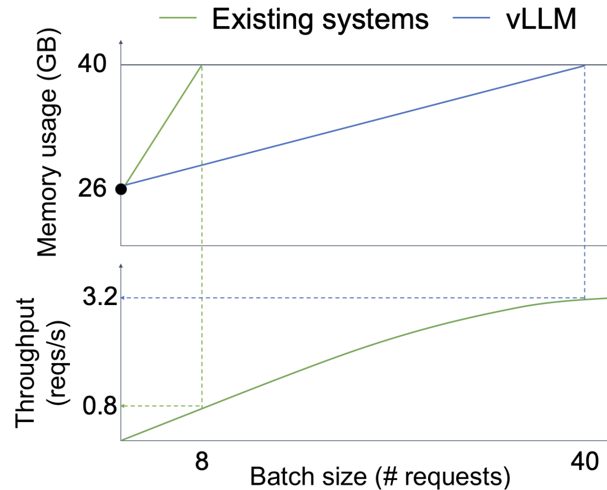


Figure 1: Memory usage and throughput comparison between vLLM and existing serving systems as batch size increases. vLLM scales to a larger batch size of around 40 requests within the same 40 GB memory budget for A100-40GB GPU, achieving 3.2 requests/s throughput, while existing systems reach the memory limit at batch size 8 with only 0.8 requests/s throughput.

2.2 Three Forms of Memory Waste in Naive KV Cache Implementation

Naive systems allocate a *fixed, contiguous block* of memory for each request's KV cache at the moment the request arrives - typically sized for the maximum possible output length (e.g., 4096 tokens) (see Figure 2). Empirical profiling reveals that only **20–40%** of allocated KV cache actually stores token states. The rest is wasted in three ways:

1. **Reservation**: Slots that are not used at the current decoding step but are reserved for tokens the model may generate in the future. Because output length is unknown a priori, the system must pessimistically reserve the full allocation upfront, even if the request ultimately terminates after very few tokens. These reserved-but-unused slots represent an *opportunity cost*: they could otherwise accommodate another request.

2. **Internal Fragmentation**: Slots that are allocated for a request but *never* used, because the actual output was shorter than the allocation. For example, if 4096 slots are allocated but the model generates only 3072 tokens, 1024 slots are permanently wasted for the lifetime of that request. This is the direct LLM analog of internal fragmentation in OS memory management.

3. **External Fragmentation**: Gaps in physical memory that arise because different requests have different sequence lengths, causing the allocator to leave interstitial space that is too small for any new request. This is again a well-known concept from OS virtual memory.

The relative magnitude of each form of waste varies across allocation strategies but is consistently large in prior systems. Figure 3 summarizes a measurement from the vLLM paper [4]: across three Orca [8] configurations, only 20–38% of the KV cache budget is actually used to store token states.

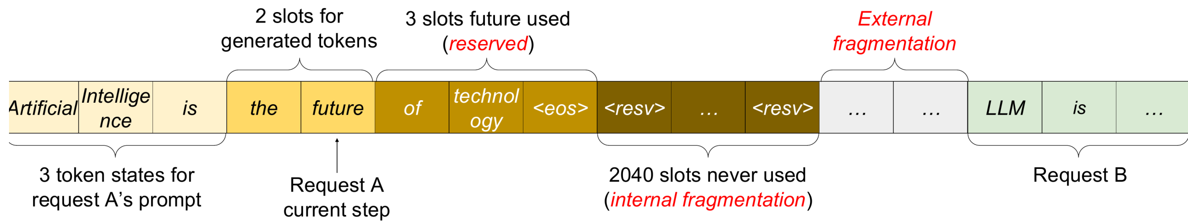


Figure 2: The three forms of memory waste in a naive KV cache layout. Only a few slots actually hold the prompt and generated token states (yellow/orange); the rest is split between *reservation* (slots reserved for future tokens), *internal fragmentation* (slots in the over-allocated block that the request will never use), and *external fragmentation* (gaps between requests).

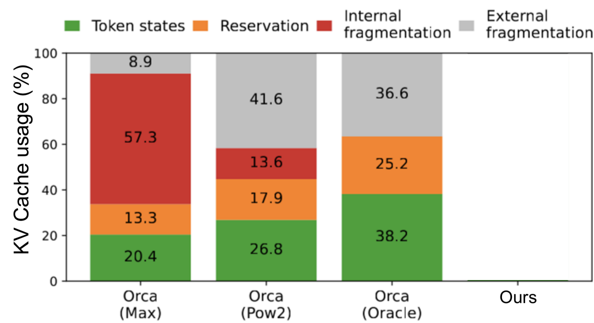


Figure 3: Breakdown of KV cache memory usage for three Orca configurations. Only 20–40% of the cache stores actual token states; the remainder is wasted in reservation, internal fragmentation, and external fragmentation. Figure reproduced from Yu et al., "Orca: A Distributed Serving System for Transformer-Based Generative Models," OSDI 2022.

3 PagedAttention and vLLM

3.1 Key Insight: Virtual Memory and Paging for KV Cache

The solution, implemented in the **vLLM** serving framework [4], is a direct application of the classical OS idea of *virtual memory with paging* (Figure 4). Just as an OS maps process virtual address spaces onto non-contiguous physical pages, vLLM maps each request's *logical* KV cache onto non-contiguous *physical token blocks* (fixed-size, contiguous chunks of GPU memory capable of storing the key and value vectors for a fixed number of tokens) in GPU memory. The analogy is precise, as summarized in Table 1.

3.2 PagedAttention: The Attention Algorithm

PagedAttention is the modified attention kernel at the heart of vLLM. It enables the attention computation to read key and value vectors from *non-contiguous* physical memory blocks, guided by a *block table*. Standard

OS Concept	vLLM Concept
Process	LLM Request
Virtual Page	Logical Token Block
Physical Page Frame	Physical Token Block
Page Table	Block Table
Physical RAM	GPU HBM (KV Cache)

Table 1: Concept-by-concept correspondence between OS virtual memory and vLLM's KV cache management.

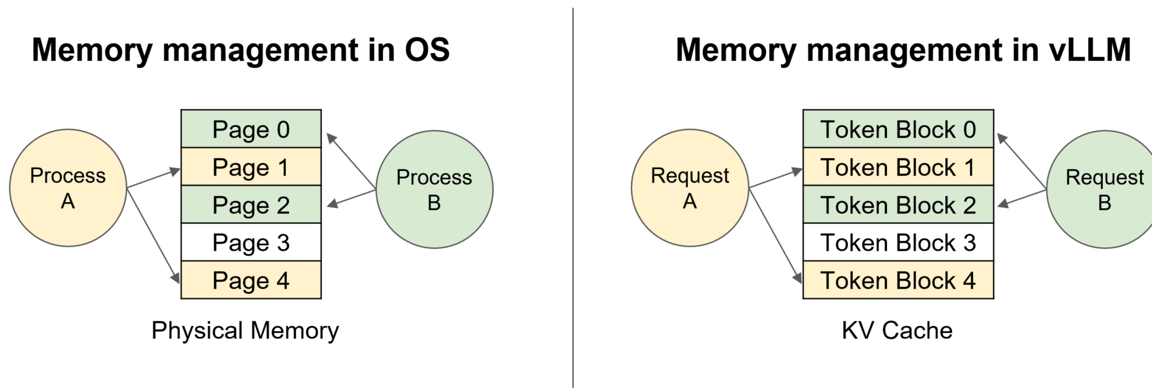


Figure 4: The OS–vLLM analogy. On the left, an operating system maps a process's virtual address space onto non-contiguous physical pages. On the right, vLLM maps each request's logical KV cache onto non-contiguous physical token blocks. Multiple requests can share the same physical pool without external fragmentation.

attention implementations (e.g., in PyTorch) assume all KV states for a sequence are stored in a contiguous memory region; PagedAttention removes this constraint.

3.3 Logical and Physical Token Blocks

For each request, vLLM maintains two views of the KV cache:

- **Logical view**: The token blocks are numbered consecutively from 0 and appear as a contiguous sequence to the attention computation.
- **Physical view**: The token blocks may be placed at arbitrary locations in GPU HBM.

A **block table** maintained per request maps each logical block number to its corresponding physical block number (and tracks how many slots in that block are filled). This is the exact analog of a page table in an OS. Figure 5 shows the state of these structures after the request "Alan Turing is a computer scientist" has been extended with the generated tokens "and mathematician renowned": the third logical block is allocated on demand to a previously free physical block.

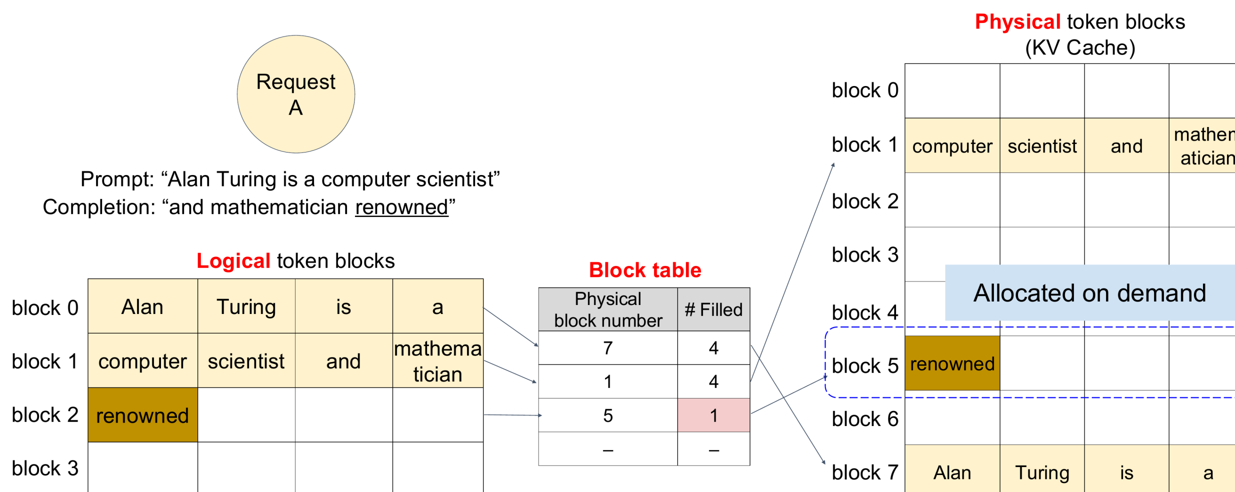


Figure 5: Logical-to-physical block mapping after the request has generated “and mathematician renowned”. The logical view (left) is contiguous and request-local; the physical view (right) is non-contiguous and shared across all requests. The block table maps each logical block to a physical block number and tracks the fill count. Logical block 2 holding “renowned” was *allocated on demand* only after logical block 1 filled up.

3.4 Serving Multiple Requests

Multiple requests are handled concurrently by maintaining independent block tables, as illustrated in Figure 6. Since all token blocks have the same fixed size, there is no external fragmentation: the physical memory pool is divided into identical units, and any unit can serve any request. vLLM pre-allocates all physical token blocks at startup and maintains a free-list; a scheduler decides which physical blocks to assign to which requests.

3.5 Memory Efficiency Analysis

PagedAttention achieves:

1. **Minimal internal fragmentation**: The only wasted memory is in the *last block* of each sequence. Since block sizes are small (16–32 tokens) relative to typical sequence lengths (hundreds to thousands of tokens), the wasted fraction is bounded and small.
2. **Zero external fragmentation**: All physical blocks have the same size, so there are no interstitial gaps.
3. **Zero reservation overhead**: Memory is allocated strictly on demand. A request never holds more physical blocks than it has actually consumed tokens to fill.

Empirically, vLLM achieves **96.3% KV cache utilization** (fraction of allocated memory storing actual token states), compared to 20–38% for systems based on Orca (the prior state-of-the-art); see Figure 7. This translates to approximately **3× higher throughput** in production LLM serving.

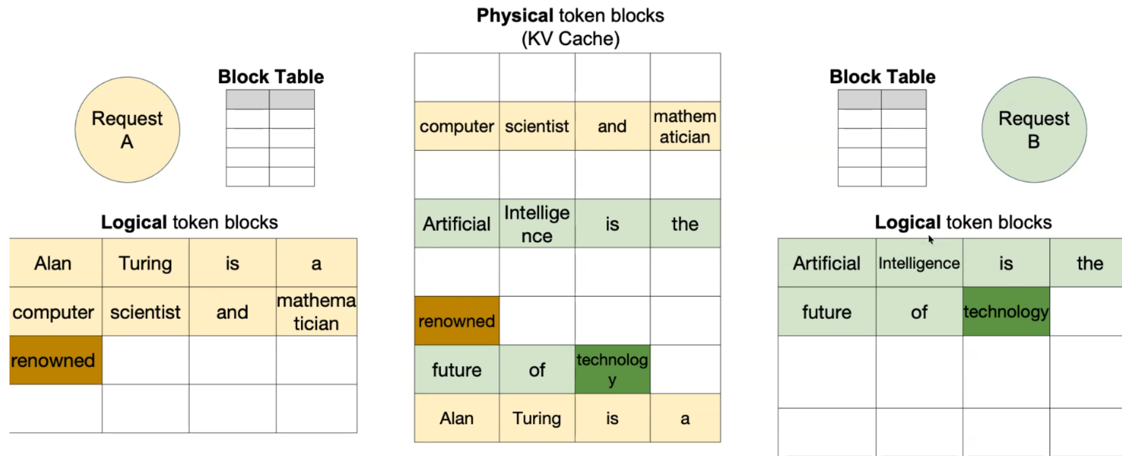


Figure 6: Example of PagedAttention with multiple requests. Request A ("Alan Turing is a computer scientist and mathematician renowned") and Request B ("Artificial Intelligence is the future of technology") are divided into logical token blocks. Their tokens are stored in different non-contiguous physical KV-cache blocks, while each request maintains a block table that maps logical blocks to physical locations. This indirection allows both requests to share the KV cache efficiently, dynamically allocate new blocks as tokens are generated, and avoid memory fragmentation.

4 Attention Complexity and the Need for Flash Attention

4.1 Attention Recap

Standard self-attention computes:

$$O = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V$$

where $Q, K, V \in \mathbb{R}^{N\times d}$, N is the sequence length, and d is the head dimension. The intermediate score matrix $S = QK^\top$ has shape $N \times N$ (or, in batched multi-head form, $B \times n \times S \times S$ where B is batch size and n is the number of heads).

4.2 Quadratic Bottleneck

The compute and memory complexity of standard attention scales quadratically with sequence length:

- **Compute**: $O(BnS^2d)$, or approximately $4BS^2H$ FLOPs (where H is the model hidden dimension).
- **Memory** (to materialize S and the softmax output): $O(BnS^2)$ — scales as S^2.

Concrete example (from the slides): With B=1024, n=32 heads, H=4096, S=4096:

- Compute $\approx$ 256 TFLOPs $\Rightarrow$ feasible on an NVIDIA H100 (FP16 peak: $\sim$1979 TFLOPs/s).
- Memory for S: $1024 \times 4096 \times 4096 \times 32 \times 2\,$bytes $\approx$ **512** GB — far exceeds H100's 80 GB HBM.

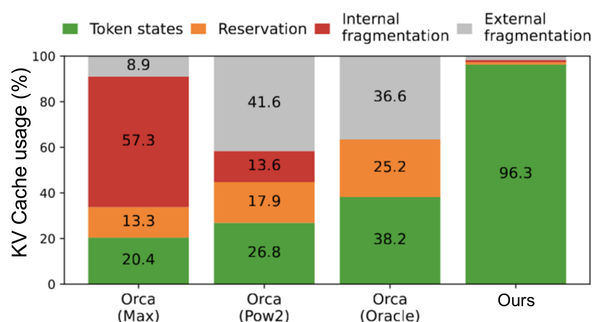


Figure 7: Effectiveness of PagedAttention. The rightmost bar ("Ours") shows that vLLM stores token states in 96.3% of allocated KV cache memory, eliminating nearly all of the reservation, internal, and external fragmentation that consume 60–80% of the budget in the Orca baselines.

Even at $S = 4096$ (a modest context length by modern standards), materializing the attention score matrix is *impossible* on a single GPU. At $S = 1\text{M}$ (common for long-context agents) or in video generation (where $S \propto H \times W \times T$, easily 100K–200K), attention completely dominates both compute and memory.

4.3 Additional Problem: Repeated HBM Traffic

Beyond sheer size, the standard algorithm (Algorithm 0 below) incurs repeated costly reads/writes of the large S matrix to HBM:

Algorithm 1 Standard Attention (Algorithm 0)

Require: Matrices $Q, K, V \in \mathbb{R}^{N \times d}$ in HBM
1: Load Q, K by blocks from HBM; compute $S = QK^\top$; write S to HBM.
2: Read S from HBM; compute $P = \text{softmax}(S)$; write P to HBM.
3: Load P and V by blocks from HBM; compute $O = PV$; write O to HBM.
4: **return** O

S and P are both $N \times N$, so steps 1–3 involve moving hundreds of GB of data between HBM (1.5 TB/s, 40–80 GB) and SRAM (19 TB/s, only 20 MB). HBM bandwidth is the true bottleneck, not compute.

The memory hierarchy specs that motivate this analysis (Figure 8):

- **SRAM (on-chip shared memory)**: ~19 TB/s bandwidth, ~20 MB capacity.
- **HBM (GPU global memory)**: ~1.5–3.35 TB/s bandwidth, 40–160 GB capacity.
- **CPU DRAM**: ~12.8 GB/s, >1 TB capacity.

5 Flash Attention

Flash Attention (FA) [3] solves both problems: it computes exact attention *without ever materializing* S (or the softmax weights P) in HBM. The key algorithmic insight is that softmax can be computed *online* via a

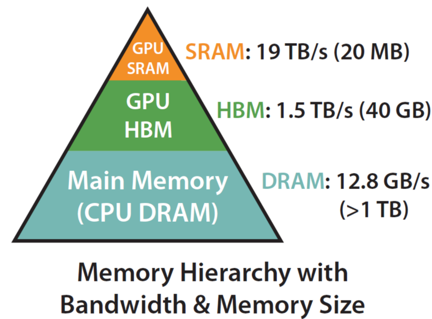


Figure 8: GPU memory hierarchy. SRAM is roughly 13× faster than HBM but 2000× smaller. The standard attention algorithm pays the HBM cost three times per layer because it reads and writes the full $N \times N$ score matrix; Flash Attention's goal is to keep the $N \times N$ tile inside SRAM and never spill it.

recurrence, enabling a fully tiled implementation analogous to tiled matrix multiplication.

5.1 The Core Challenge: Tiling Softmax

In tiled matrix multiplication (as studied in the Matmul lecture), we decompose the computation into small tiles that fit in SRAM and accumulate partial sums into register tiles, never fully materializing the output matrix in SRAM. The same idea applies to attention — *if* we can tile the softmax.

The difficulty is that softmax is a global reduction: to compute $\text{softmax}(x)_i = e^{x_i} / \sum_j e^{x_j}$, the denominator depends on *all* elements of x. How do we compute this incrementally, one tile at a time?

5.2 Step 1: Naive Softmax (Algorithm 1)

The naive two-pass algorithm:

Algorithm 2 Naive Softmax (Algorithm 1)

1: $d_0 \leftarrow 0$
2: **for** $j \leftarrow 1, V$ **do** ▷ Pass 1: compute denominator
3: $\quad d_j \leftarrow d_{j-1} + e^{x_j}$
4: **end for**
5: **for** $i \leftarrow 1, V$ **do** ▷ Pass 2: normalize
6: $\quad y_i \leftarrow e^{x_i} / d_V$
7: **end for**

Problem: e^{x_j} overflows FP16/FP32 for large x_j (e.g., large logits), and the algorithm requires two passes over the data.

5.3 Step 2: Safe Softmax (Algorithm 2)

Subtract the maximum value $m = \max_k x_k$ before exponentiating:

$$y_i = \frac{e^{x_i - m}}{\sum_j e^{x_j - m}}$$

The largest exponent is now $e^0 = 1$, eliminating overflow. However, this requires *three passes*: find the max, compute the sum of shifted exponentials, then normalize.

Algorithm 3 Safe Softmax (Algorithm 2)

1: $m_0 \leftarrow -\infty$
2: **for** $k \leftarrow 1, V$ **do** ▷ Pass 1: find max
3: $\quad m_k \leftarrow \max(m_{k-1}, x_k)$
4: **end for**
5: $d_0 \leftarrow 0$
6: **for** $j \leftarrow 1, V$ **do** ▷ Pass 2: compute denominator
7: $\quad d_j \leftarrow d_{j-1} + e^{x_j - m_V}$
8: **end for**
9: **for** $i \leftarrow 1, V$ **do** ▷ Pass 3: normalize
10: $\quad y_i \leftarrow e^{x_i - m_V} / d_V$
11: **end for**

5.4 Step 3: Online Safe Softmax (Algorithm 3)

The key observation, due to Milakov and Gimelshein [5], is that passes 1 and 2 can be fused via a *running denominator with a running maximum.* Define the alternative sequence:

$$d'_i := \sum_{j=1}^{i} e^{x_j - m_i}$$

where $m_i = \max_{k \le i} x_k$ is the running maximum. Note that $d'_V = d_V$ (the true denominator). This sequence satisfies the recurrence:

$$d'_i = d'_{i-1} \cdot e^{m_{i-1} - m_i} + e^{x_i - m_i}$$

This recurrence is derivable by decomposing the sum:

$$d'_i = \sum_{j=1}^{i-1} e^{x_j - m_i} + e^{x_i - m_i} = \left(\sum_{j=1}^{i-1} e^{x_j - m_{i-1}} \right) e^{m_{i-1} - m_i} + e^{x_i - m_i} = d'_{i-1} \cdot e^{m_{i-1} - m_i} + e^{x_i - m_i}$$

Each element of this recurrence depends only on quantities already computed, enabling a single sequential left-to-right scan. This fuses passes 1 and 2 into one, reducing the algorithm to two passes:

5.5 Step 4: Applying the Online Recurrence to Self-Attention

In self-attention, we do not actually need the softmax weights a_i explicitly. We only need the *output vector*:

$$\mathbf{o}_i := \sum_{j=1}^{i} \frac{e^{x_j - m_N}}{d'_N} V[j, :]$$

Algorithm 4 Online Safe Softmax (Algorithm 3)

1: $m_0 \leftarrow -\infty$; $d_0 \leftarrow 0$
2: **for** $j \leftarrow 1, V$ **do** ▷ Pass 1: fused max + running sum
3: $m_j \leftarrow \max(m_{j-1}, x_j)$
4: $d_j \leftarrow d_{j-1} \cdot e^{m_{j-1}-m_j} + e^{x_j - m_j}$
5: **end for**
6: **for** $i \leftarrow 1, V$ **do** ▷ Pass 2: normalize
7: $y_i \leftarrow e^{x_i - m_V}/d_V$
8: **end for**

Define the alternative running output:

$$\mathbf{o}'_i := \sum_{j=1}^{i} \frac{e^{x_j - m_i}}{d'_i} V[j,:]$$

At the end of the sequence ($i = N$), $\mathbf{o}'_N = \mathbf{o}_N$ (the true attention output). The recurrence for $\mathbf{o}'_i$ is:

$$\mathbf{o}'_i = \mathbf{o}'_{i-1} \cdot \frac{d'_{i-1} \cdot e^{m_{i-1}-m_i}}{d'_i} + \frac{e^{x_i - m_i}}{d'_i} V[i,:]$$

This is derived analogously to the denominator recurrence: $\mathbf{o}'_i$ depends only on $\mathbf{o}'_{i-1}$, d'_{i-1}, d'_i, m_{i-1}, m_i, and $V[i,:]$ — all quantities available during a single left-to-right pass.

5.6 Flash Attention Algorithm

Combining both recurrences yields the Flash Attention single-loop formulation for the k-th query row:

Algorithm 5 Flash Attention (single query row)

1: Initialize: $m_0 \leftarrow -\infty$, $d'_0 \leftarrow 0$, $\mathbf{o}'_0 \leftarrow \mathbf{0}$
2: **for** $i \leftarrow 1, N$ **do**
3: $x_i \leftarrow Q[k,:] \cdot K^\top[:,i]$ ▷ scalar attention score
4: $m_i \leftarrow \max(m_{i-1}, x_i)$
5: $d'_i \leftarrow d'_{i-1} \cdot e^{m_{i-1}-m_i} + e^{x_i - m_i}$
6: $\mathbf{o}'_i \leftarrow \mathbf{o}'_{i-1} \cdot \dfrac{d'_{i-1} \cdot e^{m_{i-1}-m_i}}{d'_i} + \dfrac{e^{x_i - m_i}}{d'_i} \cdot V[i,:]$
7: **end for**
8: $O[k,:] \leftarrow \mathbf{o}'_N$

This achieves the three key guarantees (highlighted in the slides):

1. We only read Q and K **once** (no repeated HBM reads of S).
2. We **never materialize the full** S (the $N \times N$ score matrix).
3. We **never materialize the full** softmax weights $P = \text{softmax}(S)$.

5.7 Tiling: GPU Implementation

The loop above is sequential. In practice, Flash Attention tiles the computation across the key/value dimension so that small blocks of Q and $K^\top$ are loaded into SRAM at a time, dot products are computed on-chip, and the recurrences are applied to accumulate the running output $\mathbf{o}'$ in registers. The large $N \times N$ matrix only ever exists as a tiny transient tile in SRAM — it is never written to HBM.

The tiling procedure is illustrated in Figure 9:

1. Load a block of queries Q_{block} and keys K_{block} from HBM into SRAM.
2. On-chip, compute the local dot products $Q_{\text{block}} K_{\text{block}}^\top$ (a small tile of S).
3. Immediately apply the online softmax recurrence, updating running $(m, d', \mathbf{o}')$.
4. Accumulate the corresponding contribution from V_{block} into $\mathbf{o}'$.
5. Write the final output O to HBM once, at the end.

5.8 Backward Pass: Recomputation

During training, we need the softmax weights to compute gradients with respect to Q, K, and V. Since we never stored the full S or P, Flash Attention handles the backward pass via **recomputation**: it saves only the small normalization factors (m_N, d'_N) — two scalars per row of the attention matrix, total $O(N)$ storage. During the backward pass, a forward pass of the attention tile is re-run to recompute the needed values on-the-fly in SRAM. This mirrors the activation checkpointing idea [1] discussed earlier in the course, but applied specifically to attention.

Although recomputation increases FLOPs (Flash Attention executes approximately 75 GFLOPs vs. 66 GFLOPs for standard attention on the same benchmark), the reduction in global memory traffic (40.3 GB $\to$ 4.4 GB, a $\sim$9× reduction) yields a net **5.7× end-to-end speedup** (41.7 ms $\to$ 7.3 ms runtime).

5.9 Empirical Results

Figure 10 summarizes the empirical gains of FlashAttention-style kernels. The speed results in fig. 10a show that FlashAttention-2 achieves substantially higher forward-plus-backward attention throughput than standard PyTorch attention, reaching about 182 TFLOPs/s at sequence length 8192 compared to roughly 34 TFLOPs/s for PyTorch. The memory results in fig. 10b show the complementary benefit: FlashAttention reduces memory usage increasingly strongly as sequence length grows, reaching roughly 20× lower memory usage at sequence length 4096. These results highlight the main practical advantage of FlashAttention: it improves throughput not by changing the mathematical definition of attention, but by avoiding unnecessary materialization of the full $N \times N$ attention matrix and reducing expensive global memory traffic. Representative benchmark numbers are reported in table 2. Overall, the memory savings grow *quadratically* with sequence length because the $O(N^2)$ attention matrix is never stored, while the additional compute from recomputation remains only a small constant-factor overhead.

5.10 Cascaded Benefits for Training

The memory savings from Flash Attention unlock a cascade of further improvements in LLM training:

Metric	Standard Attention	FlashAttention-2
Speed at $S = 8192$	∼34 TFLOPs/s (PyTorch)	∼182 TFLOPs/s
Memory reduction at $S = 4096$	1× (baseline)	∼20×
GFLOPs (fwd+bwd)	66.6	75.2
Global memory access	40.3 GB	4.4 GB
Runtime	41.7 ms	7.3 ms

Table 2: FlashAttention vs. standard attention on A100-80GB.

1. **Larger batch sizes**: Without Flash Attention, the $N \times N$ attention matrix forces models to train with batch size 1 for long sequences. Flash Attention frees memory, enabling much larger batches, which increases arithmetic intensity and GPU SM utilization (moving toward compute-bound operation).
2. **Disabled gradient checkpointing**: FlashAttention internally recomputes attention statistics during backward to avoid storing the full attention matrix. Its reduced activation memory footprint often removes the need for coarse-grained activation checkpointing, thereby reducing overall recomputation overhead.
3. **Net effect**: Significantly faster convergence per wall-clock hour, despite slightly higher FLOPs per step.

The instructor described this as the **Flash Attention cascade**: one algorithmic improvement enables a chain of system-level gains that multiply together.

5.11 Flash Attention 2, 3, and 4

Flash Attention 1 [3] established the foundational algorithm. Subsequent versions introduce hardware-specific optimizations:

- **Flash-Attention-2** [2]: Improved parallelism (better partition of work across warps and thread blocks), additional kernel fusions, and refined memory access patterns. This is the version where the community recognized the transformative impact and began widespread production adoption.
- **Flash-Attention-3** [6]: Optimized for NVIDIA Hopper (H100) architecture, exploiting Hopper-specific features (e.g., Tensor Memory Accelerator, asynchronous pipelines).
- **Flash-Attention-4** [9]: Optimized for NVIDIA Blackwell (B200/B100) architecture.

Flash Attention has become a community-driven project, with NVIDIA engineers actively collaborating on architecture-specific versions with each hardware generation.

6 Summary

Table 3 positions the two techniques covered in this lecture alongside two further serving optimizations briefly mentioned.

The two main techniques studied today solve complementary problems:

Technique	**Problem Solved**	**Analogy**
vLLM / PagedAttention	KV cache memory waste	OS virtual memory + paging
Flash Attention	$O(N^2)$ memory & HBM traffic	Tiled matmul + online softmax
Continuous Batching	GPU underutilization (scheduling)	OS process scheduling
Chunked Prefill	SM underutilization in decode	Fine-grained co-scheduling

Table 3: Recap of LLM serving and training optimizations, the bottleneck each addresses, and the OS/systems analogy that motivates it.

PagedAttention addresses *serving-time* memory management: how to fit more concurrent requests into fixed GPU memory to maximize throughput. It borrows classical OS paging to eliminate the three forms of KV cache fragmentation, achieving >96% memory utilization and roughly 3× throughput gains.

Flash Attention addresses the *algorithmic* bottleneck of attention's $O(N^2)$ memory complexity: it rewrites the attention computation as an online recurrence that never materializes the $N \times N$ score matrix, achieving 10–20× memory reduction and 2–4× speedup at long context lengths. Its memory savings also cascade into larger batch sizes and eliminated gradient checkpointing, further accelerating training.

Together, these innovations illustrate a recurring theme in the course: *efficient deep learning systems require co-designing algorithms, memory management, and hardware-aware kernels.*

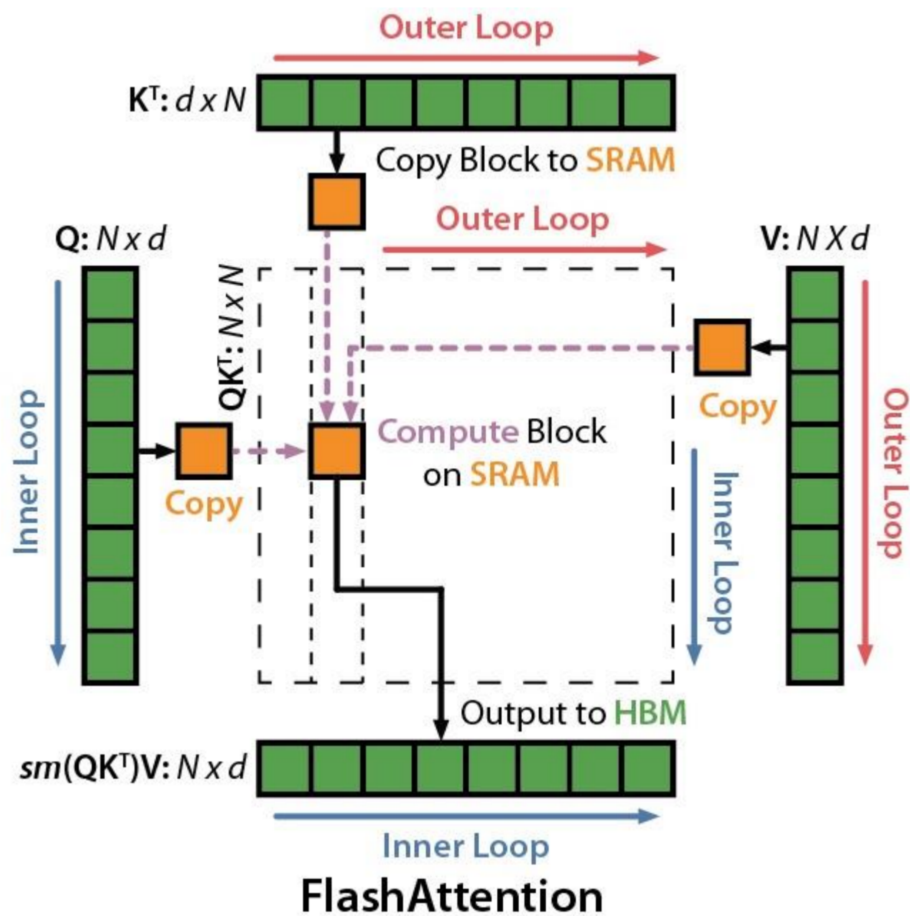
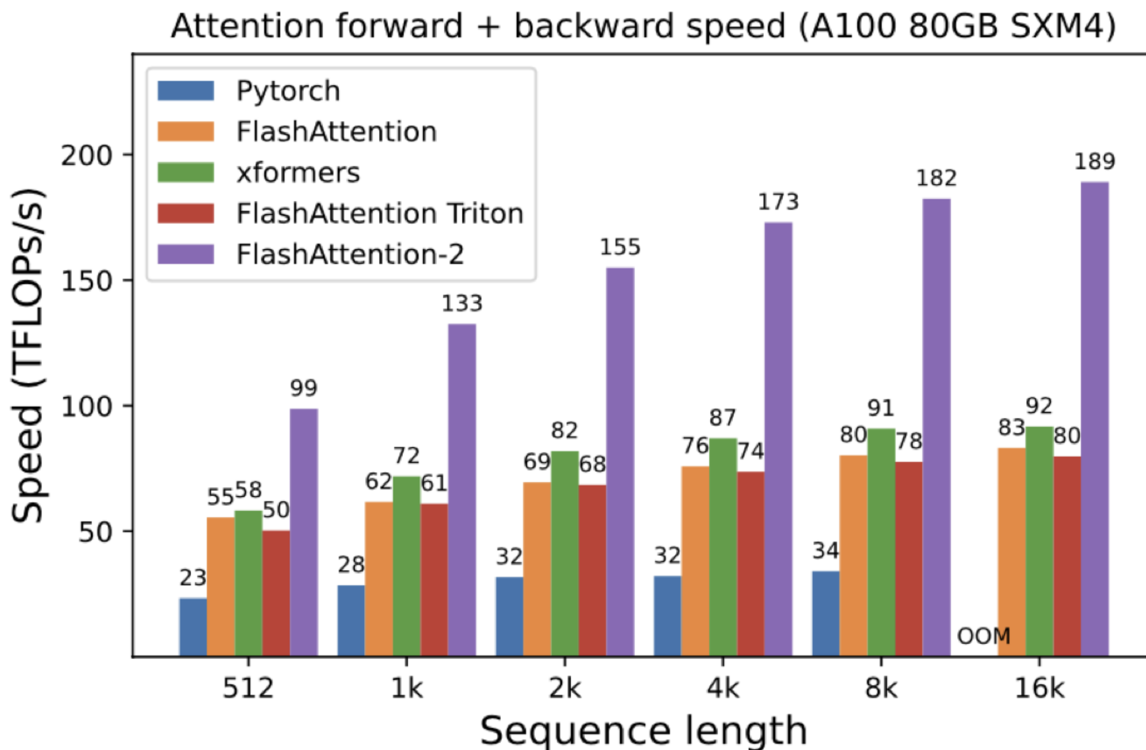
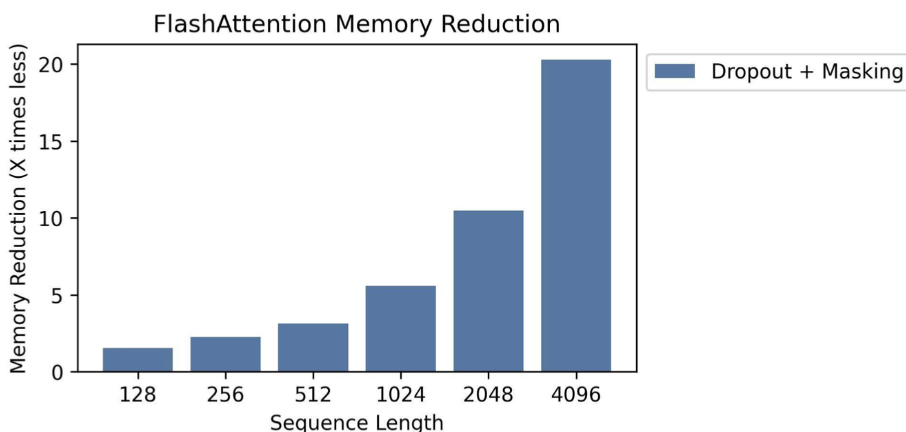


Figure 9: Flash Attention tiling. Blocks of $K^\top$ and V are streamed from HBM into SRAM along the outer loop, while a block of Q is held in SRAM along the inner loop. The attention tile $QK^\top$ is computed, online-softmax updated, and combined with V entirely on-chip; only the final output O is written back to HBM.



(a) Forward-plus-backward attention speed on an A100-80GB SXM4.



(b) Memory reduction from FlashAttention as sequence length increases.

Figure 10: Empirical speed and memory benefits of FlashAttention. FlashAttention-2 provides much higher attention throughput than standard PyTorch attention, while FlashAttention reduces memory usage increasingly strongly at longer sequence lengths.

References

[1] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[2] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations*, volume 2024, pages 35549–35562, 2024.

[3] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[4] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.

[5] Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.

[6] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

[7] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[8] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[9] Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. Flashattention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling, 2026.