CSE-234: Data Systems for Machine Learning, Winter 2025

20: LLM-4 (flash attention, deepseek-v3 review, ending words)

Lecturer: Hao Zhang
Scribe: Lucas Dionisopoulos, Hena Ahmed, Manikya Bardhan, Sara Chaudhari, Bhaavya Naharas, Xiang Xu, Tianyu Sun, Sirui Tao, Kade Shoemaker, Zihang He, Zichen Zhou

1 Recap: Solutions to Potential Bottlenecks in Serving

Serving refers to LLM inference on a large batch size $b > 1$. See the model architecture in Figure 1, which uses self-attention (specifically, FlashAttention) and MLP (FeedForward). Large batch sizes can create heavy online traffic and many incoming/outgoing requests, so the potential bottlenecks (and solutions) in LLM inference for serving include:

1. Batching prompts with varying lengths and unfamiliar tokens
   - Solution: **Continuous batching**
2. Memory and managing KV cache
   - Solution: **Paged attention**
   - Virtual memory paging is similar to virtual memory management in operating systems.
3. Service-level objectives (SLOs) and latency constraints
   - Solution: **Disaggregating prefill & decoding**

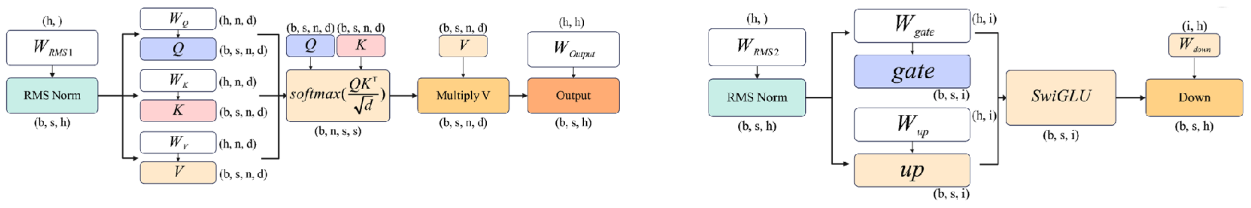


Figure 1: Llama2-7b Attention Block (Self-Attention & FeedForward)

2 Disaggregation of Prefill & Decode

2.1 What is disaggregation?

1. Disaggregation separates prefill and decoding onto different devices (e.g., GPUs).

- Model weights are replicated onto each device, which is suitable for large clusters (such as Deepseek-v3!) where memory costs for replication are less significant.

2. KV cache is computed first in the prefill worker and then migrates to the decode worker.
   - The migration between devices only occurs once, which is significantly less costly then in training, where the KV cache has to be communicated in every training iteration.

See Figure 2 for a diagram of disaggregated prefill and decoding.

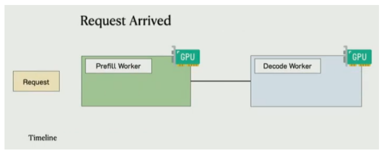
2.2 Disaggregation vs. Co-location

Co-location: uses only one device for prefill and decode.

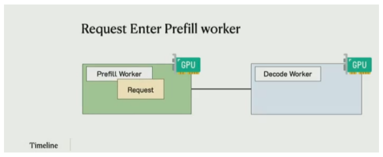
Drawbacks of co-location:

1. As more requests enter the system, the more competition there is for resources.
2. Average latency increases.
3. Fewer requests can be processed within the latency constraints.

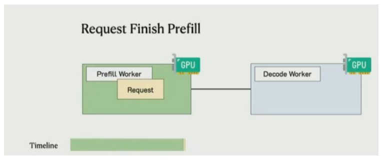
Disaggregation achieves better goodput (2× more goodput on each GPU!) because prefill and decode do not compete for resources.



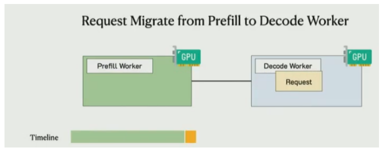
(a) Request arrives to the system.



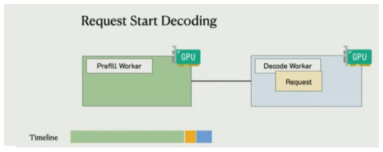
(b) Request enters prefill worker.



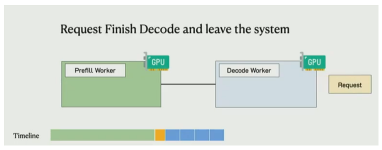
(c) Prefill ends.



(d) Request migrates from prefill to decode worker.



(e) Request starts decoding.



(f) Decoding ends, request leaves the system.

Figure 2: Diagram of a disaggregated prefill and decode system.

2.3 Large-Scale Disaggregation and Continuous Batching

The objective of *continuous batching* is to maximize throughput by increasing GPU utilization. The objective of disaggregation, on the other hand, is to optimize goodput by improving latency.

Similarities between disaggregation and continuous batching:

1. Attention and MLP are batched differently. In disaggregation, attention is not batched, and MLP is batched at the token level.

2. Their shared objective is to finish each request as quickly as possible to have it exit the system and let a new request enter.

2.4 Other Research Topics in LLM Inference

In addition to aforementioned continuous batching, disaggregated prefill & decoding, following works are proposed:

- **Chunked Prefill**: Continuous batching suffers from pipeline bubbles due to workload imbalance between prefill and decode requests. To solve this, chunked-prefill divides a prefill request into equal-sized chunks. During inference, single prefill chunk is batched with additional decode requests to maximize compute utilization and optimize the handling of decode requests.
- **Speculative Decoding**: Speculative Decoding speedups LLM inference by running two models in parallel: (1) target model, the main LLM for the task and (2) small draft model, a smaller, lightweight LLM that runs alongside to help speed up the main LLM's inference process. During inference, small draft model generates speculative tokens and then target model verify those draft output tokens generated by smaller draft model, generating several tokens in one forward pass of larger model, without changing the output distribution.
- **Sparse Attention & KV cache**: Performing attention without attending to all the tokens (or KV cache), i.e. trade accuracy for computation speed.
- **Kernel Optimization**: Optimizing computation at kernel level.

3 FlashAttention

3.1 Motivation

Motivation: The softmax portion of the attention calculation has quadratic compute and memory. Can we further improve this?

The softmax computation in attention, defined as:

$$O = \text{Softmax}(QK^\top)V$$

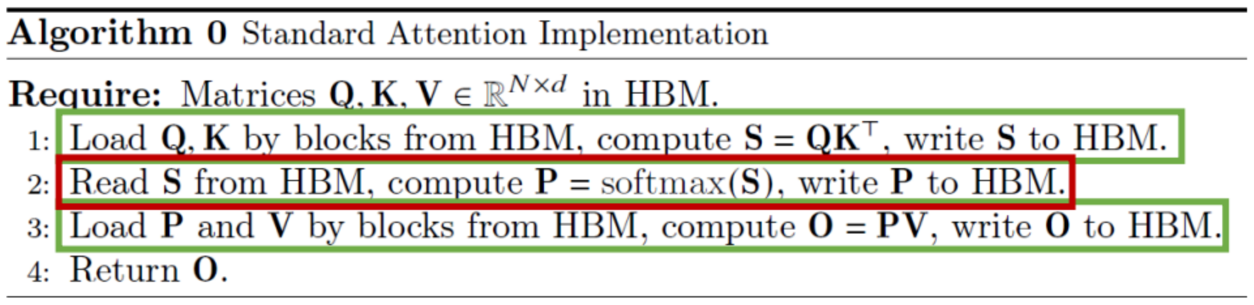
yields an intermediary matrix due to the calculation of $QK^\top$ that is quadratic in size w.r.t. the sequence length N. Now, let's view this from two lenses to understand the impact – and assume that we have $batch\_size = 1k$, $seq\_length = 4k$, $hidden\_dim = 4k$, and $num\_heads = 32$:

- **Compute**: The amount of computation required for softmax is a function of $4 \times batch\_size \times seq\_length^2 \times hidden\_dim$. Given our assumptions this requires 256 TFLOPS of compute.
- **Memory**: The amount of memory required for the same softmax function is $batch\_size \times seq\_length^2 \times num\_heads$. Given our assumptions this requires 512GB of memory.

Comparing this to the current specifications of SoTA chips (e.g., Nvidia H100 SXM) – GPU memory is far more scarce than compute. The H100 SXM chip has 80GB of memory and has 989 TFLOPS of FP32 Tensor Core compute – so if we can trade a bit more computation for much less memory, we'll have a better arithmetic intensity on this operation.

3.2 Standard Attention Implementation

The standard attention implementation requires the following:



```
Algorithm 0 Standard Attention Implementation
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.
1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.
```

Figure 3: Standard implementation of attention.

The key issue with this implementation is that you end up reading and writing both S and P to HBM – both of these are quadratic in nature w.r.t. the sequence length, which is very costly.

This defines our challenge that FlashAttention approaches from – can we compute the output without explicitly materializing S and P?

If you recall from the matrix multiplication tiling class, we could compute the final result without fully materializing the input and output via tiling. This is a perspective to approach understanding how FlashAttention works from.

3.3 Naive Softmax and its issues

As mentioned above, the softmax computation is done as follows:

$$O = \mathrm{softmax}(QK^\intercal)V$$

Performing that key-value query:

1. Takes quadratic compute $4bs^2h$
2. and quadratic memory bs^2n

At the moment, GPU memory is more scarce than compute.

So if $b = 1024$, $n = 32$, $h = 4096$, the compute required is 256 TFLOPS, which is very much doable, while the memory required is 512 GB. So even this fairly modest operation requires incredible memory usage. We want to optimize our softmax function to reduce the amount of memory usage so we can run the computations much more easily. The large (bnss) matrix makes things even worse as Q, K, V, S, P, O are in $\mathbb{R}^{Nd}$ and softmax is a fairly costly operation in both time and space. In addition, the naive softmax does memory reads and writes between different layers of memory hierarchy (HBM, SRAM) and of the large bnss matrix.

Just as a reminder, for GPUs, the memory hierarchy is GPU SRAM > GPU HBM > CPU DRAM (main memory), where > indicates that the one on the left is faster than the one on the right.

So now we want to tile the softmax to begin improving memory efficiency. Specifically, we want to avoid fully materializing S in memory as is costly in terms of memory I/O as discussed earlier.

So we need the softmax reduction O without access to the $N \times N$ matrix, and compute backwards without even saving the $N \times N$ softmax forward activations.

The solution to this is a new method known as *online softmax.* The derivation of this operation is discussed below.

3.4 Optimizing Softmax:Safe Softmax

Quickly – recall that there exists an equivalent implementation of softmax – *safe softmax* – that protects against potential numerical overflow. In the standard softmax implementation where V is the length of your input vector composed of elements e^{x_i}, we define the ith softmax value y_i as :

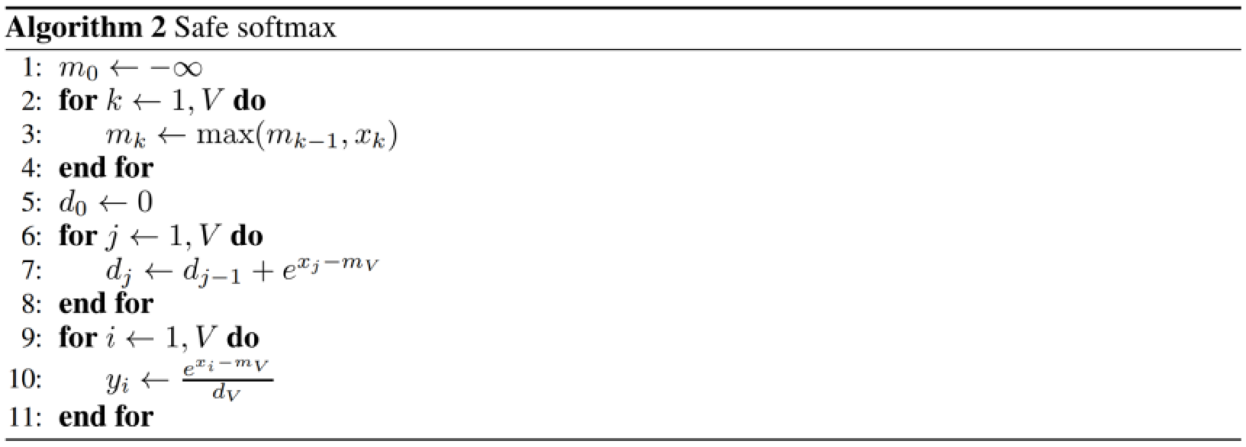
$$y_i = \frac{e^{x_i}}{\sum_k^V e^{x_i}}$$

However, the calculation of e^i can result in large numbers, which open up the risk of numerical overflow leading to training instability. To account for this, we leverage a numerically equivalent mathematically equivalent implementation called ”safe softmax” defined as:

$$m_V = \max_{k=1}^{V} x_k$$
$$y_i = \frac{e^{x_i - m_V}}{\sum_k^V e^{x_k - m_V}}$$

FYI: The way to prove mathematical equivalence is simply to convert $e^{x_i - m_V}$ to $e^{x_i}e^{-m_V}$. You can factor e^{-m_V} out of the numerator and denominator to cancel and yield standard softmax.

If we explicitly write out the looped computation of safe softmax, we identify the algorithm contains 3 loops, and can start exploring various fusing optimizations.



Algorithm 2 Safe softmax

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

Figure 4: Explicit computation of safe softmax computation.

3.5 Optimizing Softmax: Fusing the First Two For Loops (Online, Safe Softmax)

First, we will attempt fusing the first two loops on 2 and 6 in Figure 4 by finding a subsequence that allows us to define the function recursively. Notably, let's look at the accumulator d_j and try to compute this using an alternate partial sequence d'_i.

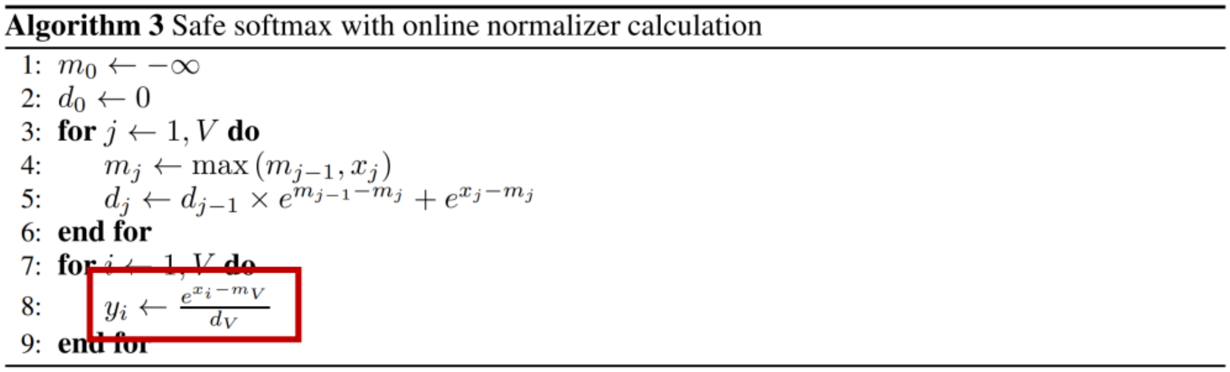
The intuition here is that we want to compute the accumulator over partial sequences. If we can do this, we may be able to combine those two for loops together so you don't need the first pass to find the max m_V and then use that in the second loop to compute your total accumulator. How we do this...? Math magic, naturally.

Let's define this alternative accumulator d'_i using the following – notice how it is not being computed over the full vector length V but now over a subsequence up to position i instead. Also note from above – m_i is computed as the max value of x_j encountered from position $j = 1 \to i$:

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

Essentially we can compute the accumulator that is mathematically equivalent, except now we can compute this from d'_1 all the way until $d'_V = d_V$ in one for loop.

This gives us a version of safe softmax, computed online.


Algorithm 3 Safe softmax with online normalizer calculation

```
1: $m_0 \leftarrow -\infty$
2: $d_0 \leftarrow 0$
3: for $j \leftarrow 1, V$ do
4:     $m_j \leftarrow \max(m_{j-1}, x_j)$
5:     $d_j \leftarrow d_{j-1} \times e^{m_{j-1} - m_j} + e^{x_j - m_j}$
6: end for
7: for $i \leftarrow 1, V$ do
8:     $y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
9: end for
```

Figure 5: Explicit computation of the online safe softmax computation.

3.6 Deriving Flash Attention via Further Fusing

Figure 5 displays our online safe softmax computation that we just walked through how to get to. The next question – can we further fuse the remaining two for loops? Unfortunately, this is not possible because the second for loop's denominator relies on the complete calculation of the accumulator d_V in the first for loop.

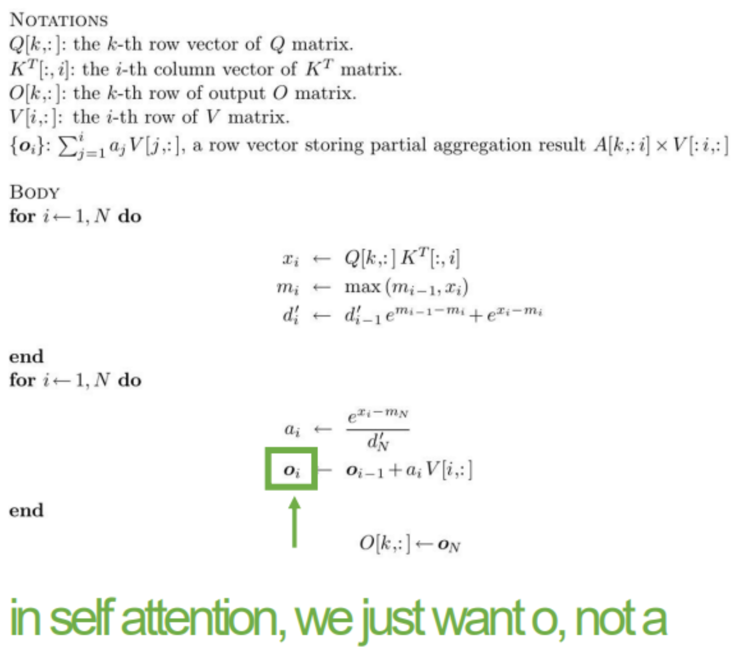
In the naive implementation of self-attention, we compute the matrix $P = \text{Softmax}(QK^\top)$ explicitly, which results in an $n \times n$ matrix. However, storing and operating on this entire matrix is computationally and memory inefficient, especially for large-scale models where n can be in the thousands.

In this setting, the naive implementation would require reading the input values **twice**:
First, to compute the values of $x_i = Q[k,:] \cdot K^T[:,i]$ and accumulate d'_N.
Then, to compute the normalized softmax values and apply them to V.

This redundant data access is inefficient. Instead, we aim to fuse operations such that each value is read only **once**, significantly reducing computational overhead. But since d_V is computed as an **accumulator** over all previous values in the sequence, it cannot be used for normalization until the first pass has completed. This dependency forces us to compute d_V fully before normalizing individual softmax terms.

If we only care about the self-attention case, we may not even need to solve for y_i in our computation directly. Rather – we may be able to *skip* materializing that step and jump straight to our output effectively fuse our operation. This is the trick that FlashAttention employs!

Let's walk through this. You'll want to take some time understanding the notation below and intuit what these computations are solving for.



Notations
$Q[k,:]$: the k-th row vector of Q matrix.
$K^T[:,i]$: the i-th column vector of K^T matrix.
$O[k,:]$: the k-th row of output O matrix.
$V[i,:]$: the i-th row of V matrix.
$\{\boldsymbol{o}_i\}$: $\sum_{j=1}^{i} a_j V[j,:]$, a row vector storing partial aggregation result $A[k,:i] \times V[:i,:]$

Body
```
for $i \leftarrow 1, N$ do
    $x_i \leftarrow Q[k,:]\, K^T[:,i]$
    $m_i \leftarrow \max(m_{i-1}, x_i)$
    $d'_i \leftarrow d'_{i-1} e^{m_{i-1}-m_i} + e^{x_i - m_i}$
end
for $i \leftarrow 1, N$ do
    $a_i \leftarrow \frac{e^{x_i - m_N}}{d'_N}$
    $\boxed{\boldsymbol{o}_i} \leftarrow \boldsymbol{o}_{i-1} + a_i V[i,:]$
end
$O[k,:] \leftarrow \boldsymbol{o}_N$
```

in self attention, we just want o, not a

Figure 6: View of self attention using online, safe softmax.

Each a_i is the true softmax value in the i^{th} column of the k^{th} row of the matrix computed when we do $P = \text{Softmax}(QK^\top)$. Recall that softmax results in the rows of P being normalized.

If this isn't clear, think about the standard softmax computation and consider just one row ($rowj$) of your matrix P. This row (shape $1 \times N$) can be multiplied by your entire matrix V (shape $N \times h$) to yield a $1 \times h$ matrix that is your answer for the j^{th} row of your output matrix, which we'll define as O. This is what Figure 6 is computing at the bottom.

Let's define this computation of o_i more formally and then similarly, derive a recurrence that will allow us to avoid materializing the softmax activation matrix. Starting with o_i:

$$o_i = \sum_{j=1}^{i} \left(\frac{e^{x_j - m_N}}{d'_N} V[j,:] \right)$$

Convince yourself that this is true value of o_i and that $i = N$ results in the mathematically correct vector k in our output matrix $O[k,:]$. Now, we can similarly derive an alternative sequence o'_i that is mathematically equal to o_i but is defined via recurrence:

$$\begin{aligned}
\boldsymbol{o}'_i &= \sum_{j=1}^{i} \frac{e^{x_j - m_i}}{d'_i} V[j,:] \\
&= \left(\sum_{j=1}^{i-1} \frac{e^{x_j - m_i}}{d'_i} V[j,:] \right) + \frac{e^{x_i - m_i}}{d'_i} V[i,:] \\
&= \left(\sum_{j=1}^{i-1} \frac{e^{x_j - m_{i-1}}}{d'_{i-1}} \frac{e^{x_j - m_i}}{e^{x_j - m_{i-1}}} \frac{d'_{i-1}}{d'_i} V[j,:] \right) + \frac{e^{x_i - m_i}}{d'_i} V[i,:] \\
&= \left(\sum_{j=1}^{i-1} \frac{e^{x_j - m_{i-1}}}{d'_{i-1}} V[j,:] \right) \frac{d'_{i-1}}{d'_i} e^{m_{i-1} - m_i} + \frac{e^{x_i - m_i}}{d'_i} V[i,:] \\
&= \boldsymbol{o}'_{i-1} \frac{d'_{i-1} e^{m_{i-1} - m_i}}{d'_i} + \frac{e^{x_i - m_i}}{d'_i} V[i,:]
\end{aligned}$$

Insight: o'_i only depends on:
$o'_{i-1}, d'_{i-1}, d'_i, m_i, m_{i-1}$

Figure 7: Alternative, recurrent definition of our output o_i.

This new recurrent definition of o'_i is mathematically equivalent to our previous output but is now defined only based on $o'_{i-1}, d'_{i-1}, d'_i, m_i,$ and m_{i-1}. Think about all of these values – these are all values that we can compute online!

Notably, we only read x_i once and we never materialize the full matrices for $QK^\top$ or Softmax($QK^\top$). And the computation is exactly the same. This yields FlashAttention:

```
for $i \leftarrow 1, N$ do
    $x_i \leftarrow Q[k,:]\, K^T[:, i]$
    $m_i \leftarrow \max(m_{i-1}, x_i)$
    $d'_i \leftarrow d'_{i-1}\, e^{m_{i-1}-m_i} + e^{x_i - m_i}$
    $\boldsymbol{o}'_i \leftarrow \boldsymbol{o}'_{i-1} \frac{d'_{i-1}\, e^{m_{i-1}-m_i}}{d'_i} + \frac{e^{x_i - m_i}}{d'_i} V[i,:]$
end
$O[k,:] \leftarrow \boldsymbol{o}'_N$
```

Figure 8: FlashAttention algorithm for computing one row of the output $O[k,:]$.

3.7 Further Optimizations: Tiling

Due to on-chip memory bandwidth constraints, we are able to use tiling for single-pass sequence processing. The term N in the $N \times N$ attention matrix does not have to be saved in GPU memory since the tileable sequence of length N can be broken down into B sized tiles. Each tile represents a group of tokens $[i, i+B-1]$. For that specific tile, only the pertinent portions of Q, K, and V are loaded into shared memory, which is more accessible, but has a limited capacity. We proceed to calculate $\exp(QK^\top)$ and perform a partial "softmax x V" over the result for the particular tile.

In order to ensure numerical stability, different exponent shifts can be employed by each tile. When merging partial outputs from the tiles, it is important to rescale. If two tiles with exponent shifts $m1$ and $m2$ are merged, the outputs from one tile must be multiplied by $\exp(m1 - m2)$. By traversing all tiles and adding up the results, along with the required scaling, the secondary attention output is obtained the same way the primary output is computed but with much lower memory usage. This technique requires less bandwidth, resulting in quicker attention computation (FlashAttention).

3.8 Recomputation in Backward Pass

Although FlashAttention helped us to avoid materializing the $s \times s$ matrix (and save memory), we still want to go through the backward pass. The solution to this is that during the backward pass, we recompute attention using stored softmax normalization factors from the forward pass. This may sound slightly counterintuitive since we are increasing FLOPs, but the main bottleneck was not on compute, but on moving content from HBM to SRAM.

The table below validates that the number of HBM accesses is the main determining factor of attention run-time. We see that even though more theoretical FLOPs are required in FlashAttention (75.2) compared to that of standard attention (66.6), the runtime is actually lower for FlashAttention. This is because FlashAttention has much fewer HBM access (and using more FLOPs for recomputation in backward does not matter much).

Attention	Standard	FlashAttention
GFLOPs	66.6	75.2
Global Memory Access (HBM R/W)	40.3 GB	4.4 GB
Runtime	41.7 ms	7.3 ms

3.9 FlashAttention Performance

We see about 10-20x memory reduction when using FlashAttention along with reduced memory loading between memory hierarchies. This leads to a roughly 2-4x speed-up. Figure 9 shows speedup and memory reduction with different sequence lengths.

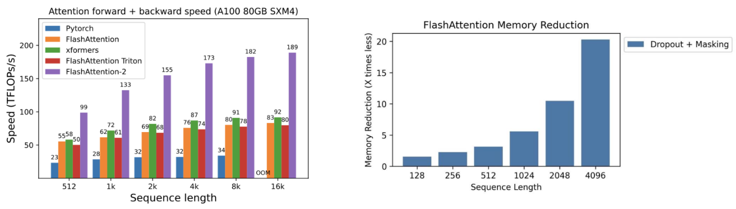


Figure 9: Speedup (left) and memory reduction (right) with different sequence lengths

3.10 High-level gists of FlashAttention

- It completely avoids materializing the large S of size bs^2n (where b is batch size, s is sequence length, and n is the number of heads).
- It will make the compute $4bs^2h$ even worse.
- It greatly reduces memory movement between memory hierarchy. This makes sense since we previously had 3 layer loop. We can even further do tiling which makes the memory usage even less.

FlashAttention is an even greater win than thought due to these cascading effects of not having to materialize the $s \times s$ matrix (saving memory of bs^2n), enabling two possibilities:

- Can train with a large b (in fact, pre-FA, most train with $b = 1$) which will lead to a higher AI.
- Can turn off gradient checkpointing since we don't need to accomodate the large $s \times s$ matrix and pay the extra forward cost (25% flops).

3.11 FlashAttention 2 and FlashAttention 3

More kernel-level optimizations over FA

- More aggressive fusion
- Uses advanced CUDA features to improve memory access patterns

Discussion: Why FA took off?

- Attention took off. Before attention, people used a lot of differnet model architectures but if self attention is mainly going to be used in the next 2-3 years then we should focus on optimizing attention.
- The second reason is because of GPU architectures. GPU has limited memory and saving materialization of $s \times s$ matrix in FA was good. We should always co-design between hardware and software (and pay attention to hardware stack while writing software).

4 How LLMs are Trained Today?

- Modern LLM training involves large-scale **GPU partitioning** into multiple **stages**, where each stage consists of a group of GPUs.
- **Pipeline parallelism** is employed to optimize computation, minimizing idle time ("bubble") during training.
- Different parallelism techniques are used:
  - **Data Parallelism** (e.g., Zero-2, Zero-3)
  - **Model Parallelism** (e.g., Megatron-LM)
  - **Tensor Parallelism**
  - **Expert Parallelism**
- **Cluster Coordination** ensures high **Model FLOP Utilization (MFU)** by efficiently orchestrating GPU usage.
- The training process operates in multiple nested loops:
  - **Outer loop: Inter-op parallelism**, using techniques like **1F1B scheduling**.
  - **Intermediate loop:** Within each stage, GPUs execute **device-mapped computations**.
  - **Innermost loop: Gradient checkpointing** and **computation optimization** are applied to reduce peak memory usage.
- **GPU Execution Details:**
  - Each GPU is assigned a **partial graph of the model**.
  - Operators are executed using **PyTorch or TensorFlow**.
  - Graph-level optimizations and efficient **kernel tiling** are applied (e.g., **flash attention**) to enhance performance.

5 DeepSeek-v3 Optimization

5.1 Overview

- **Deepseek-v3** is the **first open-source LLM on par with GPT-4o**.
- Claims to have been trained with **$5.576 million**, but this **does not include the cost of pre-training studies**, such as:
  - **Scaling law analysis** (determining data size, parameters, attention mechanisms).

  - Extensive experimentation before final training.

- The **actual cost** of planning and experimentation is **100 to 1000 times higher** than the final training job.

5.2 Sparse Mixture of Experts (MoE) Model

The DeepSeek-v3 model utilizes a Sparse Mixture of Experts **(MoE)** architecture, which significantly reduces computational costs while maintaining model scalability. Instead of using all parameters for each token, MoE dynamically selects a subset of experts per token, ensuring efficient inference.

MoE Structure:

- The model consists of **256 experts**, each specialized in handling different types of input representations.
- However, only **8 experts** are activated per token, meaning a majority of the parameters remain unused during inference.
- The **sparsity ratio** is calculated as:

$$\text{Sparsity} = 1 - \frac{8}{256} = 0.9688 \tag{1}$$

  which is a extremely high number compared to common designs.

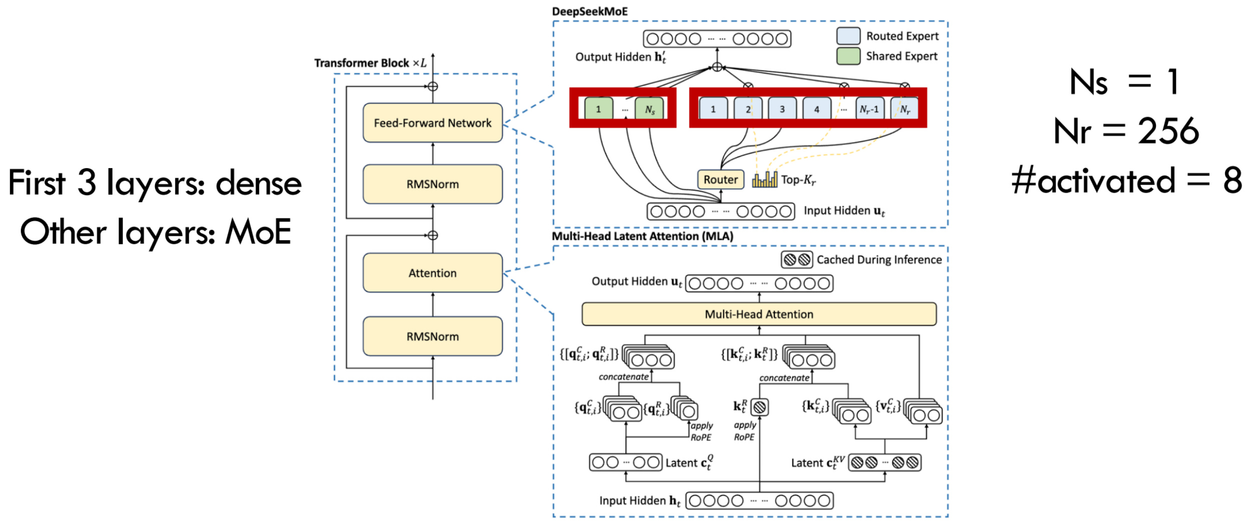


Figure 10: DeepSeek-v3 MoE

Advantages of MoE:

- **Optimized Scaling Law:** Unlike dense models, increasing parameters in MoE does not proportionally increase FLOPs. By selectively activating experts, the model achieves better parameter efficiency.
- **Efficient Computation:** Each token only activates a small fraction of the total network, reducing unnecessary computations while leveraging a large model capacity.

Challenges of MoE:

- **Communication Overhead:** Distributing token assignments across different experts requires significant GPU interconnect bandwidth.
- **Imbalance Issues:** If certain experts are favored more than others, it creates a load imbalance, leading to inefficient parallelization and straggler effects.

5.3 Multi-head Latent Attention (MLA)

Multi-head Latent Attention (MLA) is an optimization strategy designed to reduce memory overhead by minimizing KV cache storage requirements. Standard Transformer architectures store a full KV cache, which scales linearly with sequence length, creating a major bottleneck in long-sequence inference.

Key Idea: Instead of storing full key-value pairs for each token, MLA introduces a **latent space** C between the attention heads and the KV representation. By compressing information into C and later reconstructing K and V from it, the method reduces memory footprint while introducing a small computational tradeoff.

Mathematical Formulation:

- **Step 1: Compute the latent space from hidden states**

$$C_t^{KV} = W^{DKV} h_t \tag{2}$$

  This latent representation C^{KV} will be stored in KV Cache instead of full K and V.

- **Step 2: Reconstruct K and V during inference** Instead of storing K and V, we reconstruct them as:

$$K_t = W^{UK} C_t^{KV}, \quad V_t = W^{UV} C_t^{KV} \tag{3}$$

  where W^{UK} and W^{UV} are learned projection matrices.

Pros/Cons of MLA:

- **Reduced KV Cache Memory:** By storing only C instead of full KV tensors, the method drastically lowers memory consumption.
- **Additional Computation:** Although memory usage is reduced, there's a trade-off of additional matrix multiplications during inference.

5.4 Multi-token Prediction

DeepSeek-v3 uses a **next-two-token prediction** scheme by introducing an additional transformer block that predicts the next-next token and calculates its loss. This is likely inspired by EAGLE's speculative decoding method but applied **during the pre-training phase**. The DeepSeek team claims that by following this technique, one can expect the acceptance rate during inference to be up to 85%.

5.5 System Optimization

5.5.1 Challenges

DeepSeek faces a few unique challenges during their architecture design:

- **Large number of parameters in MoE** → Requires expert parallelism.
- **Expert parallelism challenge** → Requires expert balancing.
- **H800 instead of H100** → Lacks NVLink for high-bandwidth communication, so the model should minimize intra-op parallelism and instead prioritize inter-op parallelism.
  - **Solution**: Pipeline parallelism + minimized data parallelism + expert parallelism.

5.5.2 Fixing Expert Balancing

The performance of an MoE is very dependent on how well the model balances between different expert branches, as different experts see a different number of tokens (*Example: given 4 input sequences, a MoE with 3 expert branches, and a router that selects two experts, 1 expert will see 2 inputs sequences, while the others will see 3.*).

Typically, avoiding imbalance is handled using an expert balancing loss component. However since you are introducing a loss factor that is quite unrelated to next token predictions, which can hurt your ML model training.

DeepSeek's Solution: Adding an additional bias term b_i (controlled by a human monitor) to the router output. However, there is an inconsistency in the statement that the model aims to introduce a loss-free balancing technique while simultaneously introducing a sequence-level balancing loss.

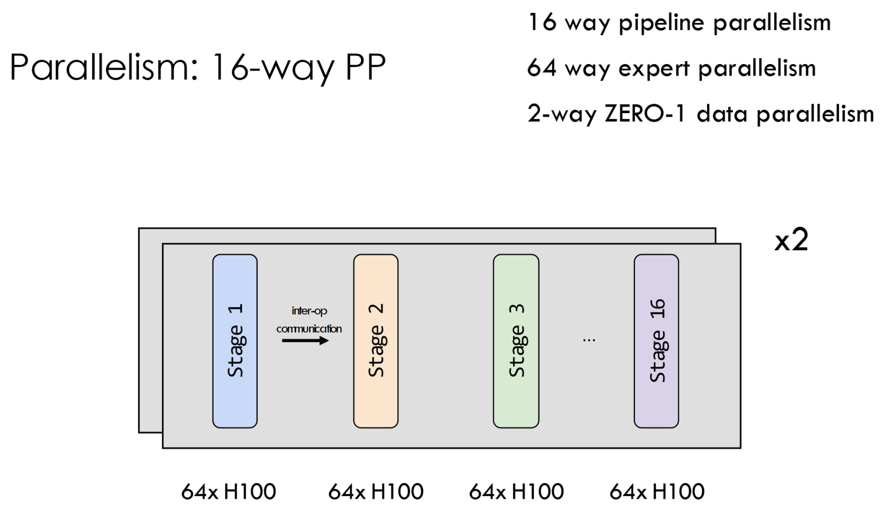
5.5.3 Parallelism

Deepseek uses the following setup for parallelism (see Figure 11):

- **Pipeline Parallelism**: 16-Way.
  - DualPipe
  - Communication to Computation Overlapping
  - All-to-all kernel
- **Expert Parallelism**: 64-Way. after finishing one
- **ZeRo-1 Data Parallelism**: 2-Way.

5.5.4 Parallelism Optimization

- DualPipe Mechanism: Overlap communication and computation to minimize pipeline stalls.
- All-to-All Kernel Optimization: Optimize tensor transfers to reduce data movement overhead between expert layers.


Parallelism: 16-way PP

16 way pipeline parallelism

64 way expert parallelism

2-way ZERO-1 data parallelism

Figure 11: DeepSeek 16-Way Parallelism

- Chimera: Store bi-directional stages and combine bidirectional pipeline to further reduce pipeline, reducing bubbles.
- Compute & Communication Patterns: LLM involves mix of all-to-all (for MoE layers) and P2P, **reverse pipelines** overlap compute and communication, reducing latency.
- All-to-All Communication Kernel: improve data transfer efficiency (across multiple GPUs)

5.5.5 System Tricks

- Going from "Traditional FP16-FP32 mixed precision training" to "FP8-FP16-FP32 mix precision training".
- Prefill-decode disaggregation

5.6 Topics Covered in the Course

- ML Systems
- CUDA Kernels
- ML Distributed systems
- Efficient ML algorithms
- The current technology market

Since Prof. Hao predict that

- The world now is changing 10x faster than before
- Innovations happen 10x faster

He wants us to have the

- Ability to identify the right problems
- Ability to understand “trends”
- Ability to “predict the future” (I hope so)

With that, he hopes that we will 1) get lots of citations, 2) get a good salary package, or 3) become a great investor and earn lots of money.

6 Contribution

- Lucas Dionisopoulos: Section 3.1 - 3.6
- Sirui Tao: Set up Google Sheet for picking note-taking time slots & added sections 5.5.5 - 5.6 and filled part of section 5.5.3
- Manikya Bardhan: Added sections 3.8 - 3.11, general formatting fixes
- Sara Chaudhari: Added and revised Section 3.6, Added Section 3.7, Revised Section 3.8
- Bhaavya Naharas: Section 4 - 5.1
- Xiang Xu: Section 5.2, Revised sections 5.3
- Tianyu Sun: Section 5.3 - 5.5
- Ming-Kai Liu: Revised Section 2.3, Added Section 2.4, Revised Section 3.1
- Kade Shoemaker: Revised sections 3.4-3.5, 5.5.2. Added section 5.5.3
- Hena Ahmed: Section 1 and 2.
- Zichen Zhou: Revised section 5.5, added section 5.5.4
- Akhil Pillai: added section 3.3, revised sections 3.1 - 3.2