LLM Intro Course: 3 – Inference: KV-cache & Long context

Prefill vs decode, attention variants (MQA, GQA, MLA), speculative decoding

Simon Vary

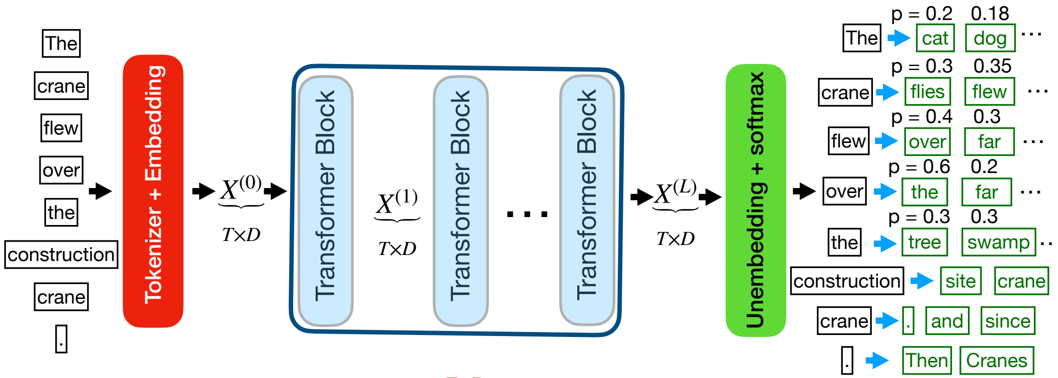
Mathematical Institute, University of Oxford

March 18, 2026

The plan for today

- **Prefill vs. decode:** why inference is different from training
- **KV cache:** what is stored, why it helps, and what it costs
- **Why generation is memory-bound:** an arithmetic-intensity view
- **Reducing KV-cache cost:** MQA, GQA, MLA, CLA, local attention
- **Faster decoding:** speculative decoding

GPT2-like Decoder Model

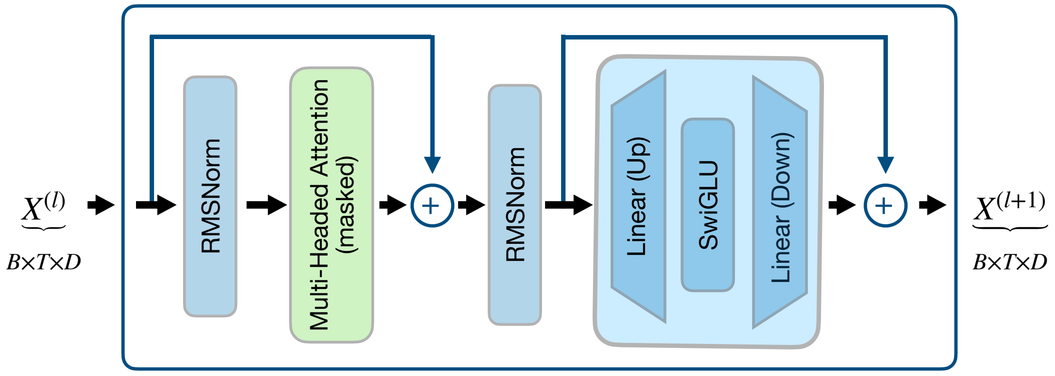


Tokenizer + Embedding : $\Sigma^* \to \mathbb{R}^{T\times D}$ string to sequence embedding

Transformer Block : $\mathbb{R}^{T\times D} \to \mathbb{R}^{T\times D}$ sequence embeddings

Unembedding + Softmax : $\mathbb{R}^{T\times D} \to \mathbb{R}^{T\times |V|}$ probability over token vocabulary

Transformer diagram


$\underbrace{X^{(l)}}_{B\times T\times D}$
RMSNorm
Multi-Headed Attention (masked)
+
RMSNorm
Linear (Up)
SwiGLU
Linear (Down)
+
$\underbrace{X^{(l+1)}}_{B\times T\times D}$

Inference: Prefill vs Decode

Metrics for Inference

Metrics

- *Time-to-first-token (TTFT):* How long user waits before ay generation appears
- *Latency (seconds/token):* How fast tokens appear for user
- *Throughput (tokens/second):* How many tokens the model outputs overall per second (among all users).

Metrics for Inference

Metrics

- *Time-to-first-token (TTFT):* How long user waits before ay generation appears
- *Latency (seconds/token):* How fast tokens appear for user
- *Throughput (tokens/second):* How many tokens the model outputs overall per second (among all users).

What is different now?

- Training (supervised): you feed whole sentences (all tokens) $\Rightarrow$ can parallelize over sequence.
- Inference: you have to generate sequentially $\Rightarrow$ cannot paralelize.

Metrics for Inference

Metrics

- *Time-to-first-token (TTFT):* How long user waits before ay generation appears
- *Latency (seconds/token):* How fast tokens appear for user
- *Throughput (tokens/second):* How many tokens the model outputs overall per second (among all users).

What is different now?

- Training (supervised): you feed whole sentences (all tokens) $\Rightarrow$ can parallelize over sequence.
- Inference: you have to generate sequentially $\Rightarrow$ cannot paralelize.

Why important?

- Providers of closed / open models must solve this problem
- Influences architecture

Why we broadcast over batches?

Arithmetic Intensity: for $X \cdot W$, where $X \in \mathbb{R}^{B \times D}$, $W \in \mathbb{R}^{D \times F}$

Why we broadcast over batches?

Arithmetic Intensity: for $X \cdot W$, where $X \in \mathbb{R}^{B \times D}$, $W \in \mathbb{R}^{D \times F}$

1. *Read X and W:* bytes transfered: $2 \cdot B \cdot D + 2 \cdot D \cdot F$

Why we broadcast over batches?

Arithmetic Intensity: for $X \cdot W$, where $X \in \mathbb{R}^{B \times D}$, $W \in \mathbb{R}^{D \times F}$

1. *Read X and W:* bytes transfered: $2 \cdot B \cdot D + 2 \cdot D \cdot F$
2. *Compute XW:* flops: $2 \cdot B \cdot D \cdot F$

Why we broadcast over batches?

Arithmetic Intensity: for $X \cdot W$, where $X \in \mathbb{R}^{B \times D}$, $W \in \mathbb{R}^{D \times F}$

1. *Read X and W:* bytes transfered: $2 \cdot B \cdot D + 2 \cdot D \cdot F$
2. *Compute XW:* flops: $2 \cdot B \cdot D \cdot F$
3. *Write XW:* bytes transfered: $2 \cdot B \cdot F$

Why we broadcast over batches?

Arithmetic Intensity: for $X \cdot W$, where $X \in \mathbb{R}^{B \times D}$, $W \in \mathbb{R}^{D \times F}$

1. *Read X and W:* bytes transfered: $2 \cdot B \cdot D + 2 \cdot D \cdot F$
2. *Compute XW:* flops: $2 \cdot B \cdot D \cdot F$
3. *Write XW:* bytes transfered: $2 \cdot B \cdot F$

$$\text{Intensity} = \frac{\text{flops}}{\text{bytes transfered}} = \frac{2BDF}{2BD + 2DF + 2BF} \approx B.$$

How many operations per read/write

Why we broadcast over batches?

Arithmetic Intensity: for $X \cdot W$, where $X \in \mathbb{R}^{B \times D}$, $W \in \mathbb{R}^{D \times F}$

1. *Read X and W:* bytes transfered: $2 \cdot B \cdot D + 2 \cdot D \cdot F$
2. *Compute XW:* flops: $2 \cdot B \cdot D \cdot F$
3. *Write XW:* bytes transfered: $2 \cdot B \cdot F$

$$\text{Intensity} = \frac{\text{flops}}{\text{bytes transfered}} = \frac{2BDF}{2BD + 2DF + 2BF} \approx B.$$

How many operations per read/write

Theoretical arithmetic-intensity limit of the hardware

- **L40S:** $\approx$ 200 FLOP/byte (TF32), $\approx$ 400 FLOP/byte (BF16/FP16)
- **H100 SXM:** $\approx$ 300 FLOP/byte (TF32), $\approx$ 600 FLOP/byte (BF16/FP16)

Why we broadcast over batches?

Arithmetic Intensity: for $X \cdot W$, where $X \in \mathbb{R}^{B \times D}$, $W \in \mathbb{R}^{D \times F}$

1. *Read X and W:* bytes transfered: $2 \cdot B \cdot D + 2 \cdot D \cdot F$
2. *Compute XW:* flops: $2 \cdot B \cdot D \cdot F$
3. *Write XW:* bytes transfered: $2 \cdot B \cdot F$

$$\text{Intensity} = \frac{\text{flops}}{\text{bytes transfered}} = \frac{2BDF}{2BD + 2DF + 2BF} \approx B.$$

How many operations per read/write

Theoretical arithmetic-intensity limit of the hardware

- **L40S:** $\approx$ 200 FLOP/byte (TF32), $\approx$ 400 FLOP/byte (BF16/FP16)
- **H100 SXM:** $\approx$ 300 FLOP/byte (TF32), $\approx$ 600 FLOP/byte (BF16/FP16)

- If computation intensity $>$ accelerator intensity, compute-limited (good)
- If computation intensity $<$ accelerator intensity, memory-limited (bad)

Why we broadcast over batches?

Arithmetic Intensity: for $X \cdot W$, where $X \in \mathbb{R}^{B \times D}$, $W \in \mathbb{R}^{D \times F}$

1. *Read X and W:* bytes transfered: $2 \cdot B \cdot D + 2 \cdot D \cdot F$
2. *Compute XW:* flops: $2 \cdot B \cdot D \cdot F$
3. *Write XW:* bytes transfered: $2 \cdot B \cdot F$

$$\text{Intensity} = \frac{\text{flops}}{\text{bytes transfered}} = \frac{2BDF}{2BD + 2DF + 2BF} \approx B.$$

How many operations per read/write

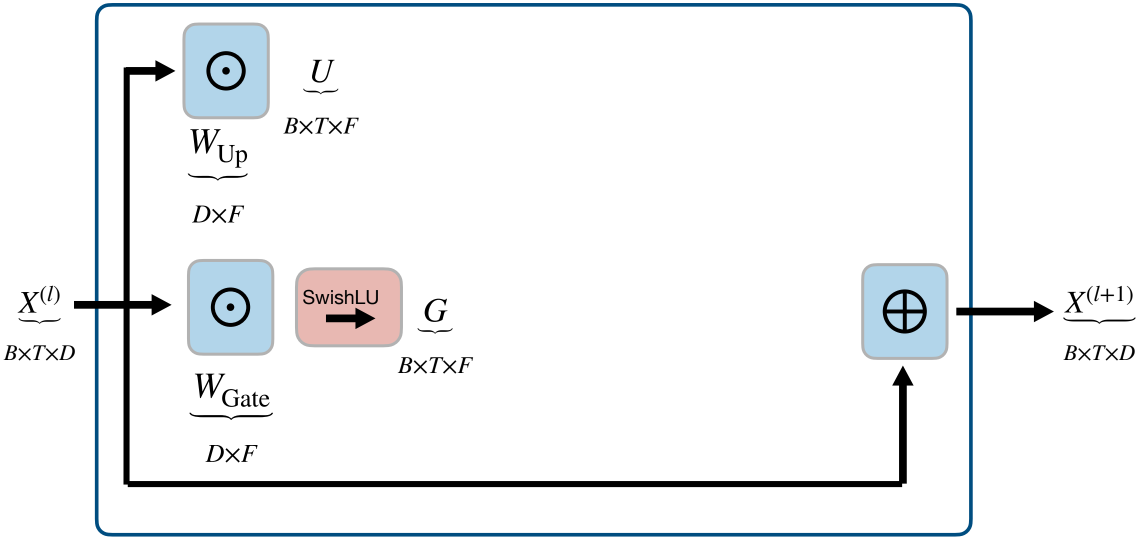
Theoretical arithmetic-intensity limit of the hardware

- **L40S:** $\approx$ 200 FLOP/byte (TF32), $\approx$ 400 FLOP/byte (BF16/FP16)
- **H100 SXM:** $\approx$ 300 FLOP/byte (TF32), $\approx$ 600 FLOP/byte (BF16/FP16)

- If computation intensity $>$ accelerator intensity, compute-limited (good)
- If computation intensity $<$ accelerator intensity, memory-limited (bad)

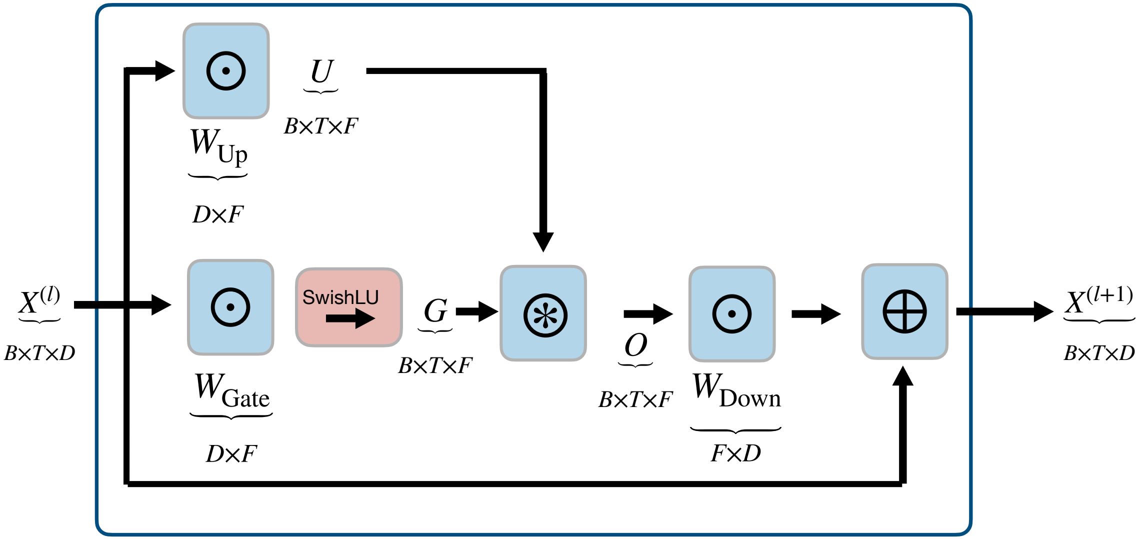
Conclusion: compute-limited in BF16 if $B > 400$.

MLP Arithmetic



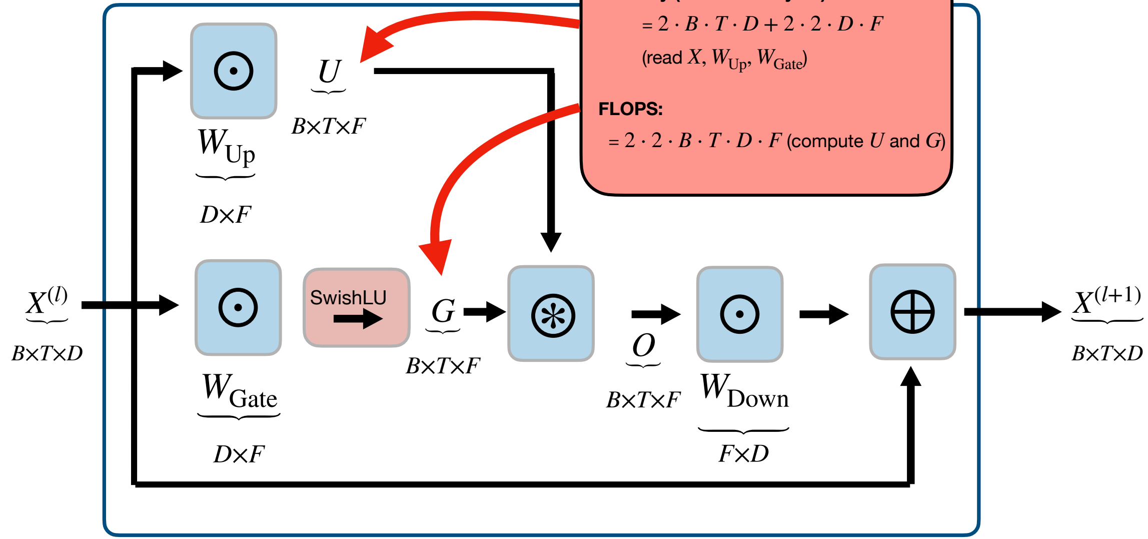
$\underbrace{U}_{B\times T\times F}$
$\underbrace{W_{\text{Up}}}_{D\times F}$
$\underbrace{X^{(l)}}_{B\times T\times D}$
$\underbrace{W_{\text{Gate}}}_{D\times F}$
SwishLU
$\underbrace{G}_{B\times T\times F}$
$\underbrace{X^{(l+1)}}_{B\times T\times D}$

MLP Arithmetic



$\underbrace{X^{(l)}}_{B \times T \times D}$
$\underbrace{W_{\text{Up}}}_{D \times F}$
$\underbrace{U}_{B \times T \times F}$
$\underbrace{W_{\text{Gate}}}_{D \times F}$
SwishLU
$\underbrace{G}_{B \times T \times F}$
$\underbrace{O}_{B \times T \times F}$
$\underbrace{W_{\text{Down}}}_{F \times D}$
$\underbrace{X^{(l+1)}}_{B \times T \times D}$

MLP Arithmetic



$\underbrace{X^{(l)}}_{B\times T\times D}$
$\underbrace{W_{\text{Up}}}_{D\times F}$
$\underbrace{U}_{B\times T\times F}$
$\underbrace{W_{\text{Gate}}}_{D\times F}$
SwishLU
$\underbrace{G}_{B\times T\times F}$
$\underbrace{O}_{B\times T\times F}$
$\underbrace{W_{\text{Down}}}_{F\times D}$
$\underbrace{X^{(l+1)}}_{B\times T\times D}$

Memory (number of bytes):

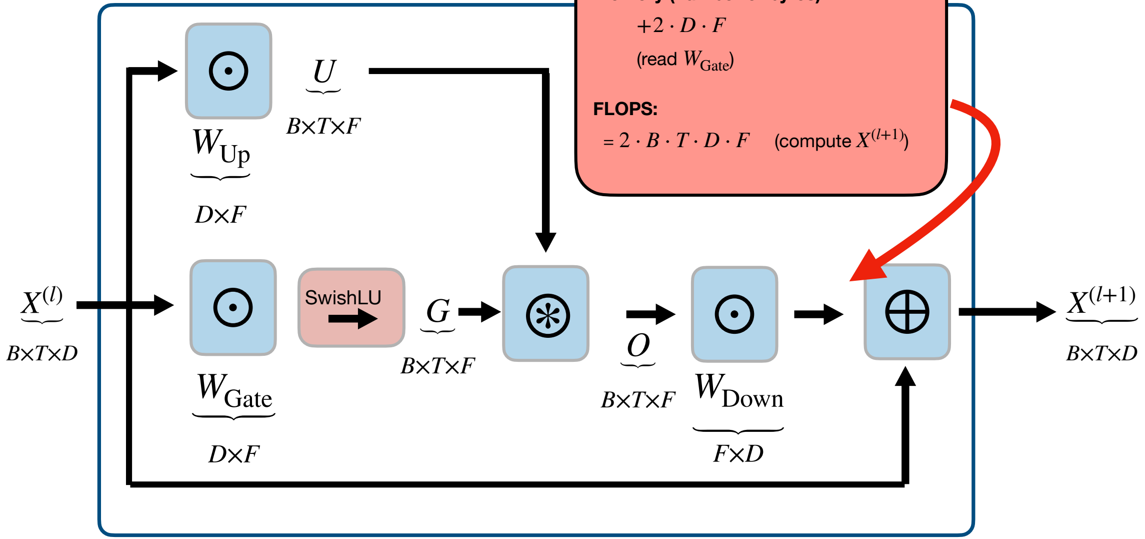
$= 2 \cdot B \cdot T \cdot D + 2 \cdot 2 \cdot D \cdot F$

(read X, W_{Up}, W_{Gate})

FLOPS:

$= 2 \cdot 2 \cdot B \cdot T \cdot D \cdot F$ (compute U and G)

MLP Arithmetic



$\underbrace{X^{(l)}}_{B \times T \times D}$
$\underbrace{W_{\text{Up}}}_{D \times F}$
$\underbrace{U}_{B \times T \times F}$
$\underbrace{W_{\text{Gate}}}_{D \times F}$
SwishLU
$\underbrace{G}_{B \times T \times F}$
$\underbrace{O}_{B \times T \times F}$
$\underbrace{W_{\text{Down}}}_{F \times D}$
$\underbrace{X^{(l+1)}}_{B \times T \times D}$

Memory (number of bytes):

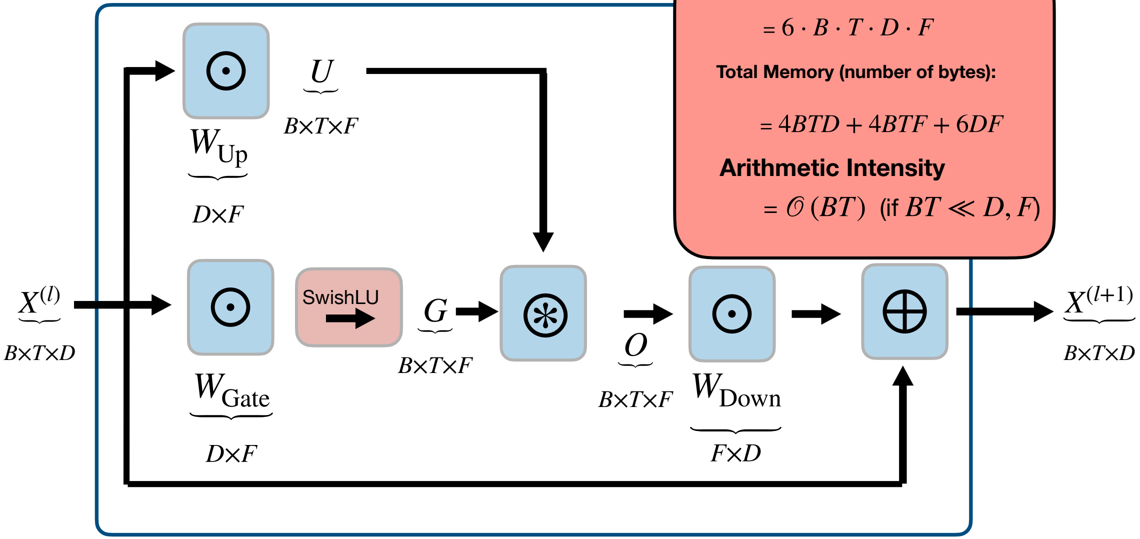
$+2 \cdot D \cdot F$

(read W_{Gate})

FLOPS:

$= 2 \cdot B \cdot T \cdot D \cdot F$ (compute $X^{(l+1)}$)

MLP Arithmetic


$\underbrace{X^{(l)}}_{B \times T \times D}$
$\underbrace{W_{\text{Up}}}_{D \times F}$
$\underbrace{U}_{B \times T \times F}$
$\underbrace{W_{\text{Gate}}}_{D \times F}$
SwishLU
$\underbrace{G}_{B \times T \times F}$
$\underbrace{O}_{B \times T \times F}$
$\underbrace{W_{\text{Down}}}_{F \times D}$
$\underbrace{X^{(l+1)}}_{B \times T \times D}$

Total FLOPS:

$$= 6 \cdot B \cdot T \cdot D \cdot F$$

Total Memory (number of bytes):

$$= 4BTD + 4BTF + 6DF$$

Arithmetic Intensity

$$= \mathcal{O}(BT) \text{ (if } BT \ll D, F)$$

MLP Arithmetic Summary

Prefill ($T = S$)

$$\text{FLOPs} \approx 6BTDF$$

$$\text{Memory} \approx 4BTD + 4BTF + 6DF$$

$$\text{Intensity} = \mathcal{O}(BT)$$

- Easy to make compute-limited
- Good regime for hardware utilization

MLP Arithmetic Summary

Prefill ($T = S$)

$$\text{FLOPs} \approx 6BTDF$$

$$\text{Memory} \approx 4BTD + 4BTF + 6DF$$

$$\text{Intensity} = \mathcal{O}(BT)$$

- Easy to make compute-limited
- Good regime for hardware utilization

Generation ($T = 1$)

$$\text{FLOPs} \approx 6BDF$$

$$\text{Memory} \approx 4BD + 4BF + 6DF$$

$$\text{Intensity} = \mathcal{O}(B).$$

- One token: $T = 1$, for B requests
- Hard to make B large enough

MLP Arithmetic Summary

Prefill ($T = S$)

$$\text{FLOPs} \approx 6BTDF$$

$$\text{Memory} \approx 4BTD + 4BTF + 6DF$$

$$\text{Intensity} = \mathcal{O}(BT)$$

- Easy to make compute-limited
- Good regime for hardware utilization

Generation ($T = 1$)

$$\text{FLOPs} \approx 6BDF$$

$$\text{Memory} \approx 4BD + 4BF + 6DF$$

$$\text{Intensity} = \mathcal{O}(B).$$

- One token: $T = 1$, for B requests
- Hard to make B large enough

Takeaway

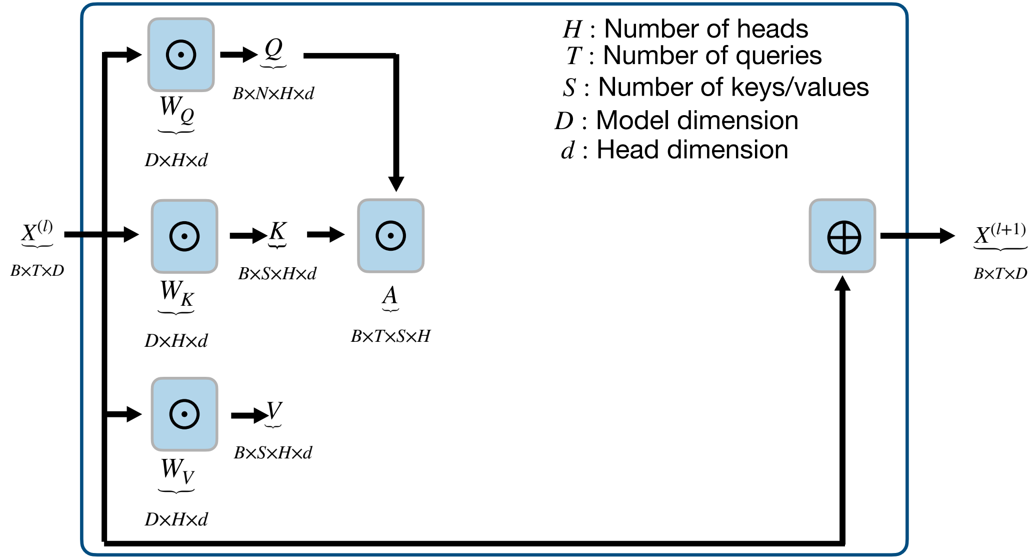
For $BT \ll D, F$,

$$\text{Intensity} = \mathcal{O}(BT).$$

So MLP layers benefit strongly from batching:

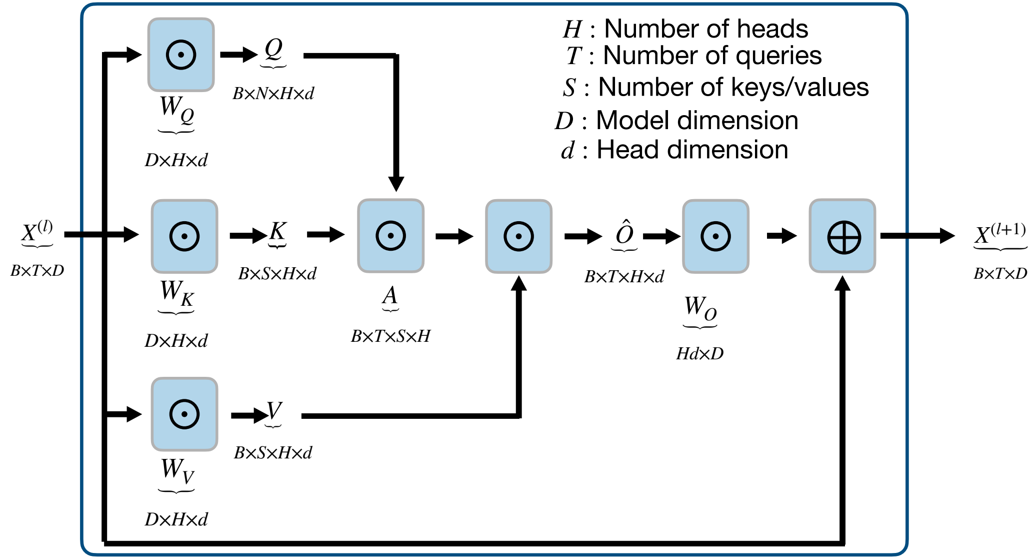
$$\text{Prefill: increase } BT, \qquad \text{Generation: increase } B.$$

(FlashAttention-style) MHA Arithmetic


$\underbrace{X^{(l)}}_{B \times T \times D}$
$\underbrace{W_Q}_{D \times H \times d}$
$\underbrace{Q}_{B \times N \times H \times d}$
$\underbrace{W_K}_{D \times H \times d}$
$\underbrace{K}_{B \times S \times H \times d}$
$\underbrace{A}_{B \times T \times S \times H}$
$\underbrace{W_V}_{D \times H \times d}$
$\underbrace{V}_{B \times S \times H \times d}$
$\underbrace{X^{(l+1)}}_{B \times T \times D}$

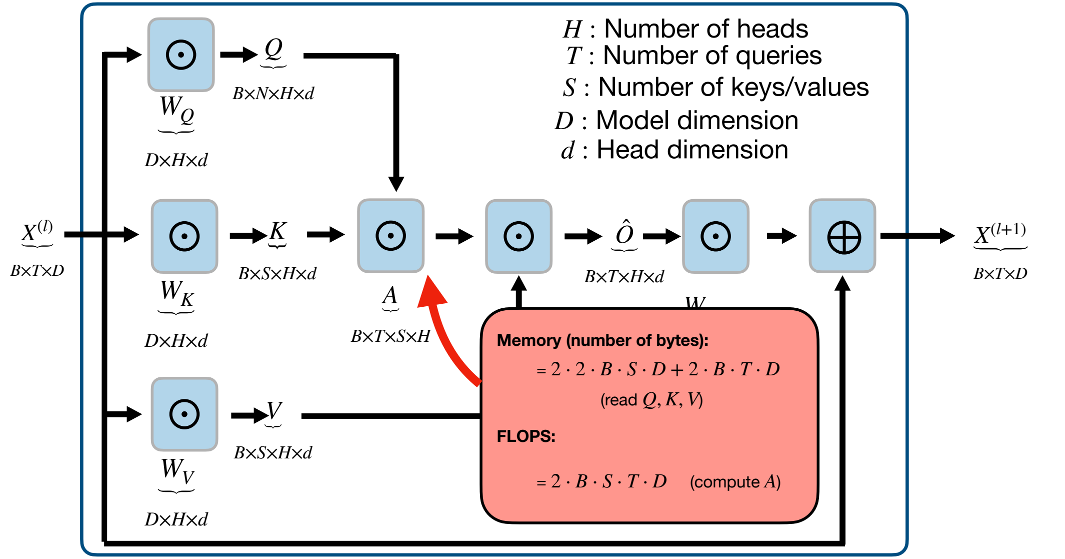
H : Number of heads
T : Number of queries
S : Number of keys/values
D : Model dimension
d : Head dimension

(FlashAttention-style) MHA Arithmetic



H : Number of heads
T : Number of queries
S : Number of keys/values
D : Model dimension
d : Head dimension
$\underbrace{X^{(l)}}_{B\times T\times D}$
$\underbrace{W_Q}_{D\times H\times d}$
$\underbrace{W_K}_{D\times H\times d}$
$\underbrace{W_V}_{D\times H\times d}$
$\underbrace{Q}_{B\times N\times H\times d}$
$\underbrace{K}_{B\times S\times H\times d}$
$\underbrace{V}_{B\times S\times H\times d}$
$\underbrace{A}_{B\times T\times S\times H}$
$\underbrace{\hat{O}}_{B\times T\times H\times d}$
$\underbrace{W_O}_{Hd\times D}$
$\underbrace{X^{(l+1)}}_{B\times T\times D}$
$\odot$
$\oplus$

(FlashAttention-style) MHA Arithmetic



$\underbrace{X^{(l)}}_{B\times T\times D}$
$\underbrace{W_Q}_{D\times H\times d}$
$\underbrace{Q}_{B\times N\times H\times d}$
$\underbrace{W_K}_{D\times H\times d}$
$\underbrace{K}_{B\times S\times H\times d}$
$\underbrace{W_V}_{D\times H\times d}$
$\underbrace{V}_{B\times S\times H\times d}$
$\underbrace{A}_{B\times T\times S\times H}$
$\underbrace{\hat{O}}_{B\times T\times H\times d}$
W
$\underbrace{X^{(l+1)}}_{B\times T\times D}$

H : Number of heads
T : Number of queries
S : Number of keys/values
D : Model dimension
d : Head dimension

Memory (number of bytes):

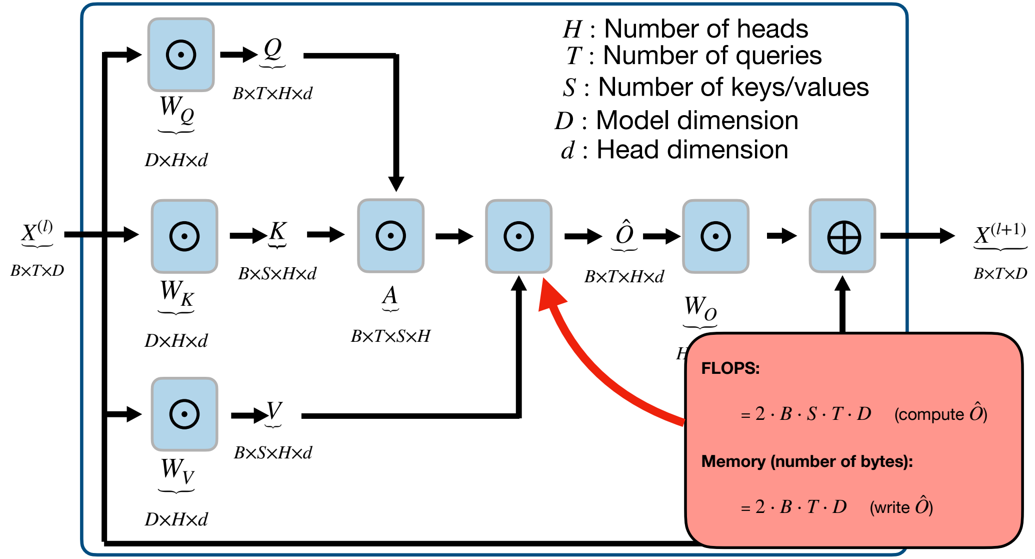
$$= 2\cdot 2\cdot B\cdot S\cdot D + 2\cdot B\cdot T\cdot D$$

(read Q, K, V)

FLOPS:

$$= 2\cdot B\cdot S\cdot T\cdot D \quad \text{(compute } A\text{)}$$

(FlashAttention-style) MHA Arithmetic



H : Number of heads
T : Number of queries
S : Number of keys/values
D : Model dimension
d : Head dimension
$\underbrace{X^{(l)}}_{B \times T \times D}$
$\underbrace{W_Q}_{D \times H \times d}$
$\underbrace{Q}_{B \times T \times H \times d}$
$\underbrace{W_K}_{D \times H \times d}$
$\underbrace{K}_{B \times S \times H \times d}$
$\underbrace{W_V}_{D \times H \times d}$
$\underbrace{V}_{B \times S \times H \times d}$
$\underbrace{A}_{B \times T \times S \times H}$
$\underbrace{\hat{O}}_{B \times T \times H \times d}$
W_O
$\underbrace{X^{(l+1)}}_{B \times T \times D}$

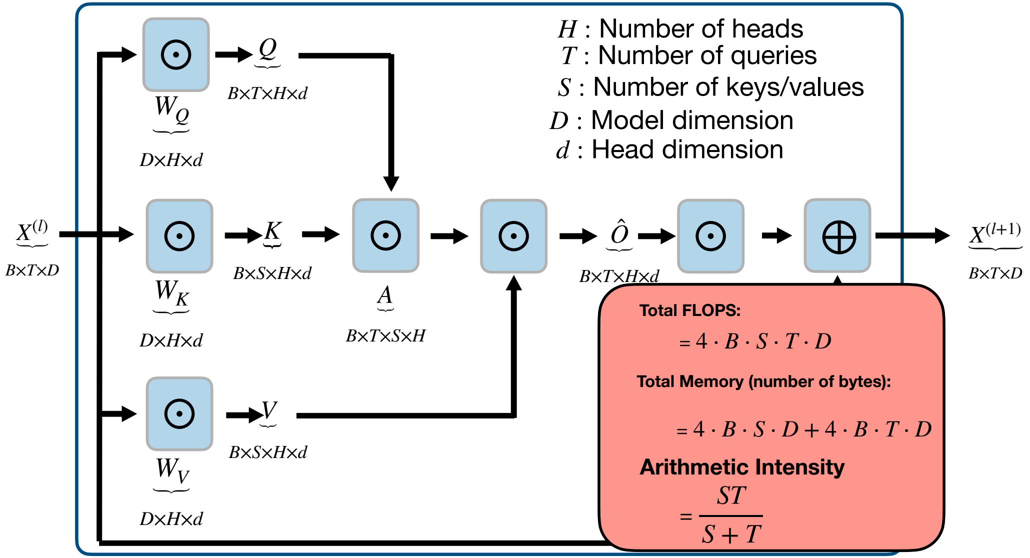
FLOPS:

$= 2 \cdot B \cdot S \cdot T \cdot D$ (compute $\hat{O}$)

Memory (number of bytes):

$= 2 \cdot B \cdot T \cdot D$ (write $\hat{O}$)

(FlashAttention-style) MHA Arithmetic



$\underbrace{X^{(l)}}_{B\times T\times D}$
$\underbrace{W_Q}_{D\times H\times d}$
$\underbrace{Q}_{B\times T\times H\times d}$
$\underbrace{W_K}_{D\times H\times d}$
$\underbrace{K}_{B\times S\times H\times d}$
$\underbrace{A}_{B\times T\times S\times H}$
$\underbrace{W_V}_{D\times H\times d}$
$\underbrace{V}_{B\times S\times H\times d}$
$\underbrace{\hat{O}}_{B\times T\times H\times d}$
$\underbrace{X^{(l+1)}}_{B\times T\times D}$

H : Number of heads
T : Number of queries
S : Number of keys/values
D : Model dimension
d : Head dimension

Total FLOPS:

$$= 4 \cdot B \cdot S \cdot T \cdot D$$

Total Memory (number of bytes):

$$= 4 \cdot B \cdot S \cdot D + 4 \cdot B \cdot T \cdot D$$

Arithmetic Intensity

$$= \frac{ST}{S+T}$$

FlashAttention-style MHA Arithmetic Summary

$$\text{Intensity} = \frac{ST}{S+T}. \qquad \Rightarrow \qquad \textsf{Prefill:}\ \text{Intensity} = \frac{S}{2}, \qquad \textsf{Generation:}\ \text{Intensity} < 1$$

FlashAttention-style MHA Arithmetic Summary

$$\text{Intensity} = \frac{ST}{S+T}. \qquad \Rightarrow \qquad \text{Prefill: } \text{Intensity} = \frac{S}{2}, \qquad \text{Generation: } \text{Intensity} < 1$$

Prefill ($T = S$)

$$\text{FLOPs} \approx 4BS^2D$$

$$\text{Memory} \approx 8BSD$$

$$\text{Intensity} = \frac{S}{2}$$

- Easy to make compute-limited
- Longer context directly helps

FlashAttention-style MHA Arithmetic Summary

$$\text{Intensity} = \frac{ST}{S+T}. \qquad \Rightarrow \qquad \text{Prefill: } \text{Intensity} = \frac{S}{2}, \qquad \text{Generation: } \text{Intensity} < 1$$

Prefill ($T = S$)

$$\text{FLOPs} \approx 4BS^2D$$

$$\text{Memory} \approx 8BSD$$

$$\text{Intensity} = \frac{S}{2}$$

- Easy to make compute-limited
- Longer context directly helps

Generation ($T = 1$)

$$\text{FLOPs} \approx 4BSD$$

$$\text{Memory} \approx 4BSD + 4BD$$

$$\text{Intensity} = \frac{S}{S+1} < 1$$

- One token attends to S cached tokens
- Strongly memory-limited

FlashAttention-style MHA Arithmetic Summary

$$\text{Intensity} = \frac{ST}{S+T}. \qquad \Rightarrow \qquad \text{Prefill: Intensity} = \frac{S}{2}, \qquad \text{Generation: Intensity} < 1$$

Prefill ($T = S$)

$$\text{FLOPs} \approx 4BS^2D$$

$$\text{Memory} \approx 8BSD$$

$$\text{Intensity} = \frac{S}{2}$$

- Easy to make compute-limited
- Longer context directly helps

Generation ($T = 1$)

$$\text{FLOPs} \approx 4BSD$$

$$\text{Memory} \approx 4BSD + 4BD$$

$$\text{Intensity} = \frac{S}{S+1} < 1$$

- One token attends to S cached tokens
- Strongly memory-limited

Takeaway

Batching B improves throughput, *but not* the intensity (each request reads its own KV cache.)

Transformer Block: Prefill vs Generation

Prefill

Input: many prompt tokens at once

$$T = S$$

- Attention can have high arithmetic intensity (for long sequences)
- Batching increases throughput, **but also**, time-to-first-token (TTFT).

Transformer Block: Prefill vs Generation

Prefill

Input: many prompt tokens at once

$$T = S$$

- Attention can have high arithmetic intensity (for long sequences)
- Batching increases throughput, **but also**, time-to-first-token (TTFT).

Generation

Decode one new token at a time

$$T = 1, \qquad S = \text{cache length}$$

- Attention becomes memory-limited
- Must repeatedly read the KV cache
- Can increase **throughput** by batching
- But, **latency** is limited by intensity.

Transformer Block: Prefill vs Generation

Prefill

Input: many prompt tokens at once

$$T = S$$

- Attention can have high arithmetic intensity (for long sequences)
- Batching increases throughput, **but also**, time-to-first-token (TTFT).

Generation

Decode one new token at a time

$$T = 1, \qquad S = \text{cache length}$$

- Attention becomes memory-limited
- Must repeatedly read the KV cache
- Can increase **throughput** by batching
- But, **latency** is limited by intensity.

Main takeaway

Prefill : TTFT vs throughput tradeoff   Generation: Intensity limits latency, throughput with B.

Strategy

Prefill

- Goal: reduce **TTFT**
- Use smaller batches
- Prioritize latency over throughput

Strategy

Prefill

- Goal: reduce **TTFT**
- Use smaller batches
- Prioritize latency over throughput

Generation

- Goal: improve throughput **tokens / second**
- Merge requests (larger batches) if possible
- Prioritize throughput (cannot improve latency)

Strategy

Prefill

- Goal: reduce **TTFT**
- Use smaller batches
- Prioritize latency over throughput

Generation

- Goal: improve throughput **tokens / second**
- Merge requests (larger batches) if possible
- Prioritize throughput (cannot improve latency)

Two regimes

$$\text{Prefill: small batch (long sequence)} \Rightarrow \text{fast first token}$$

$$\text{Generation: larger batch} \Rightarrow \text{better overall throughput}$$

Strategy

Prefill

- Goal: reduce **TTFT**
- Use smaller batches
- Prioritize latency over throughput

Generation

- Goal: improve throughput **tokens / second**
- Merge requests (larger batches) if possible
- Prioritize throughput (cannot improve latency)

Two regimes

Prefill: small batch (long sequence) $\Rightarrow$ fast first token

Generation: larger batch $\Rightarrow$ better overall throughput

... but what to do about latency in generation?

MHA Attention Variants

Grouped-query attention **(Ainslie et al., 2023)**

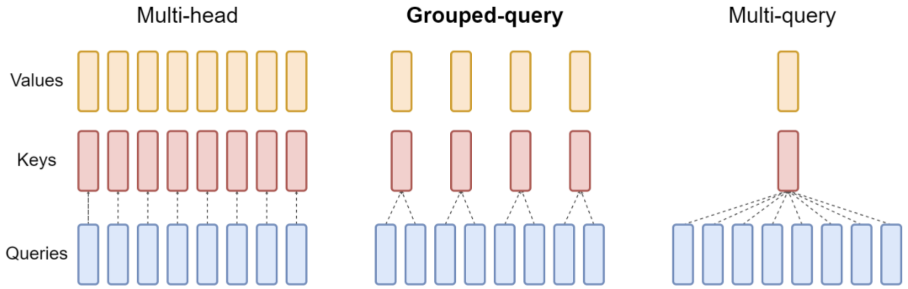


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.

Grouped-query attention **(Ainslie et al., 2023)**

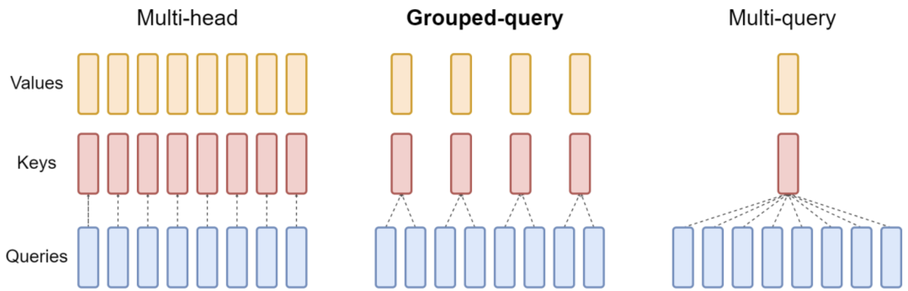


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.

Idea: Fewer key/value heads

- H_Q query heads, but only H_K key/value heads
- each interacting with $G = H_Q/H_K$ query heads

Grouped-query attention **(Ainslie et al., 2023)**

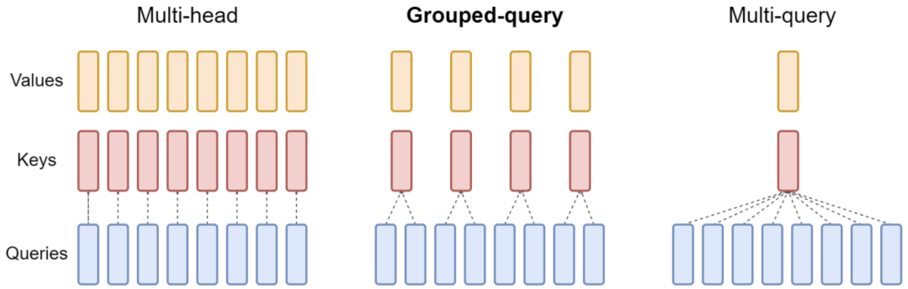


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.

Idea: Fewer key/value heads

- H_Q query heads, but only H_K key/value heads
- each interacting with $G = H_Q/H_K$ query heads

Why?

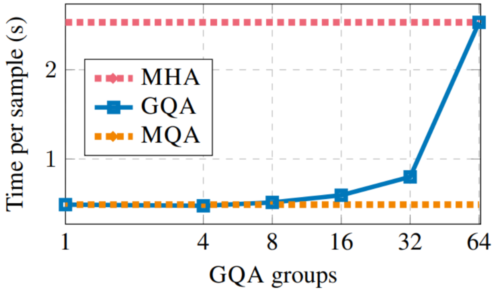
Smaller KV cache and less memory traffic during generation.

Why GQA Helps Inference

- **MHA:** $H_Q = H_K$
  - largest KV cache
  - best flexibility
- **MQA:** $H_K = 1$
  - smallest KV cache
  - strongest decoding speedup
- **GQA:** $1 < H_K < H_Q$
  - compromise between quality and efficiency

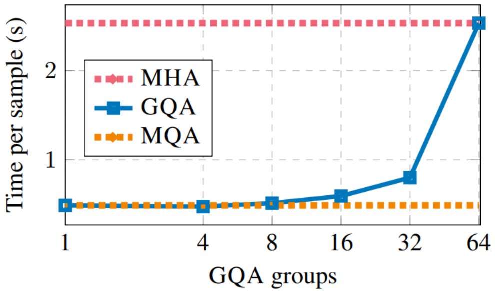
Why GQA Helps Inference

- **MHA:** $H_Q = H_K$
  - largest KV cache
  - best flexibility
- **MQA:** $H_K = 1$
  - smallest KV cache
  - strongest decoding speedup
- **GQA:** $1 < H_K < H_Q$
  - compromise between quality and efficiency



Why GQA Helps Inference

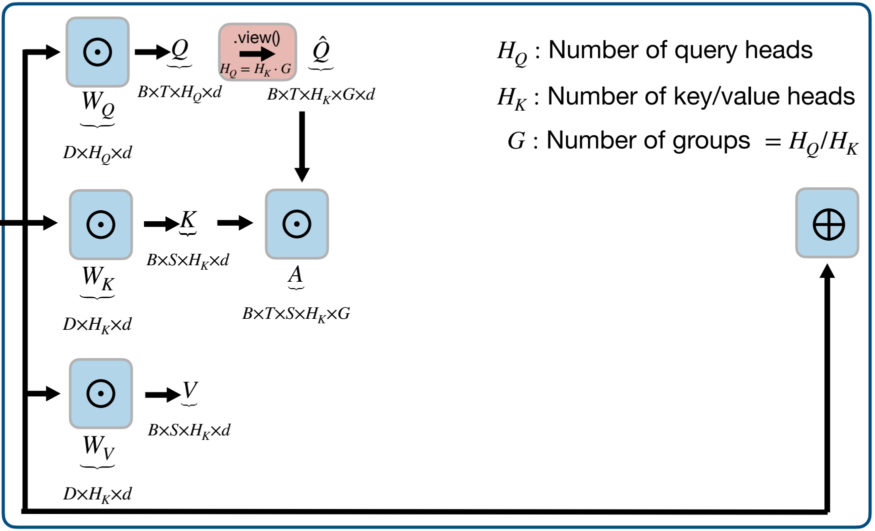
- **MHA:** $H_Q = H_K$
  - largest KV cache
  - best flexibility
- **MQA:** $H_K = 1$
  - smallest KV cache
  - strongest decoding speedup
- **GQA:** $1 < H_K < H_Q$
  - compromise between quality and efficiency



Takeaway

GQA keeps many query heads, but few keys/values heads $\Rightarrow$ reduce the memory cost + operations.

GQA Arithmetic



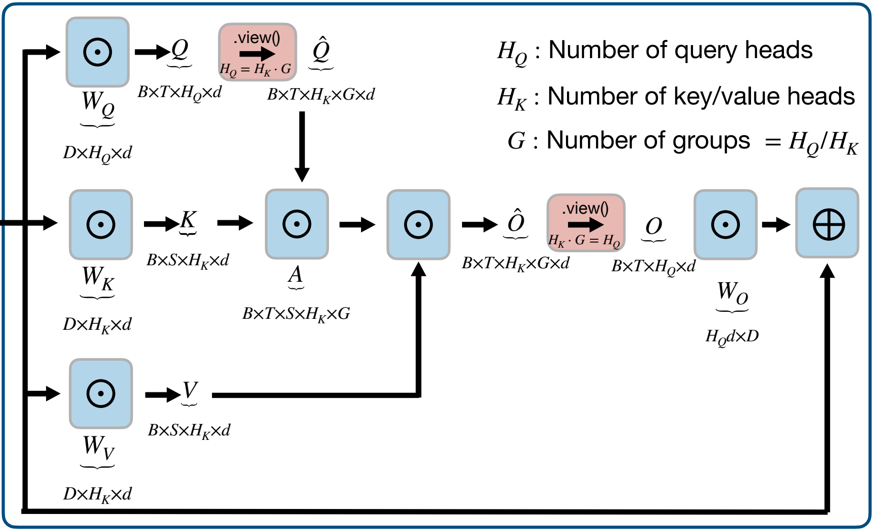
$\underbrace{X^{(l)}}_{B\times T\times D}$
$\odot$
$\underbrace{W_Q}_{D\times H_Q\times d}$
$\underbrace{Q}_{B\times T\times H_Q\times d}$
.view()
$H_Q = H_K \cdot G$
$\underbrace{\hat{Q}}_{B\times T\times H_K\times G\times d}$
$\odot$
$\underbrace{W_K}_{D\times H_K\times d}$
$\underbrace{K}_{B\times S\times H_K\times d}$
$\odot$
$\underbrace{A}_{B\times T\times S\times H_K\times G}$
$\odot$
$\underbrace{W_V}_{D\times H_K\times d}$
$\underbrace{V}_{B\times S\times H_K\times d}$
$\oplus$
$\underbrace{X^{(l+1)}}_{B\times T\times D}$

H_Q : Number of query heads

H_K : Number of key/value heads

G : Number of groups $= H_Q/H_K$

GQA Arithmetic



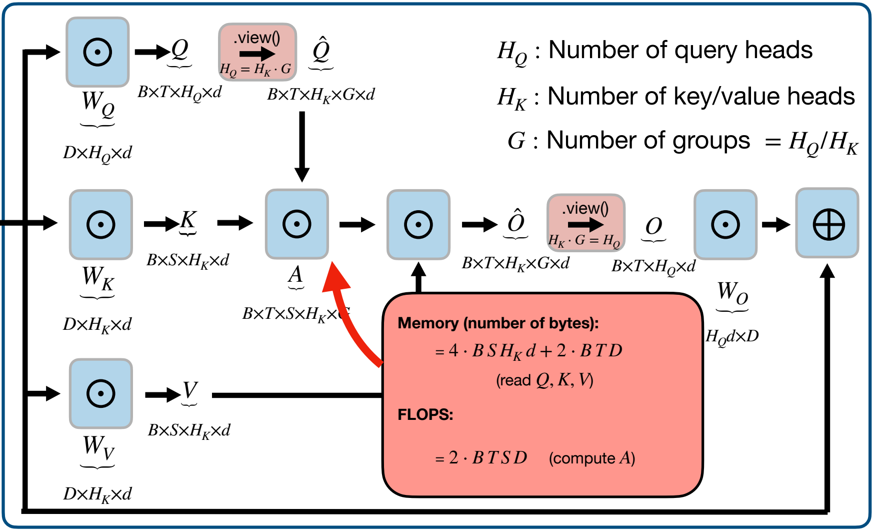
$\underbrace{X^{(l)}}_{B\times T\times D}$
$\underbrace{W_Q}_{D\times H_Q\times d}$
$\underbrace{Q}_{B\times T\times H_Q\times d}$
.view()
$H_Q = H_K \cdot G$
$\underbrace{\hat{Q}}_{B\times T\times H_K\times G\times d}$
$\underbrace{W_K}_{D\times H_K\times d}$
$\underbrace{K}_{B\times S\times H_K\times d}$
$\underbrace{A}_{B\times T\times S\times H_K\times G}$
$\underbrace{\hat{O}}_{B\times T\times H_K\times G\times d}$
.view()
$H_K \cdot G = H_Q$
$\underbrace{O}_{B\times T\times H_Q\times d}$
$\underbrace{W_O}_{H_Q d\times D}$
$\underbrace{X^{(l+1)}}_{B\times T\times D}$
$\underbrace{W_V}_{D\times H_K\times d}$
$\underbrace{V}_{B\times S\times H_K\times d}$

H_Q : Number of query heads

H_K : Number of key/value heads

G : Number of groups $= H_Q/H_K$

GQA Arithmetic


W_Q
$D \times H_Q \times d$
Q
$B \times T \times H_Q \times d$
.view()
$H_Q = H_K \cdot G$
$\hat{Q}$
$B \times T \times H_K \times G \times d$
H_Q : Number of query heads
H_K : Number of key/value heads
G : Number of groups $= H_Q / H_K$
$X^{(l)}$
$B \times T \times D$
W_K
$D \times H_K \times d$
K
$B \times S \times H_K \times d$
A
$B \times T \times S \times H_K \times G$
$\hat{O}$
$B \times T \times H_K \times G \times d$
.view()
$H_K \cdot G = H_Q$
O
$B \times T \times H_Q \times d$
W_O
$H_Q d \times D$
$X^{(l+1)}$
$B \times T \times D$
W_V
$D \times H_K \times d$
V
$B \times S \times H_K \times d$

Memory (number of bytes):

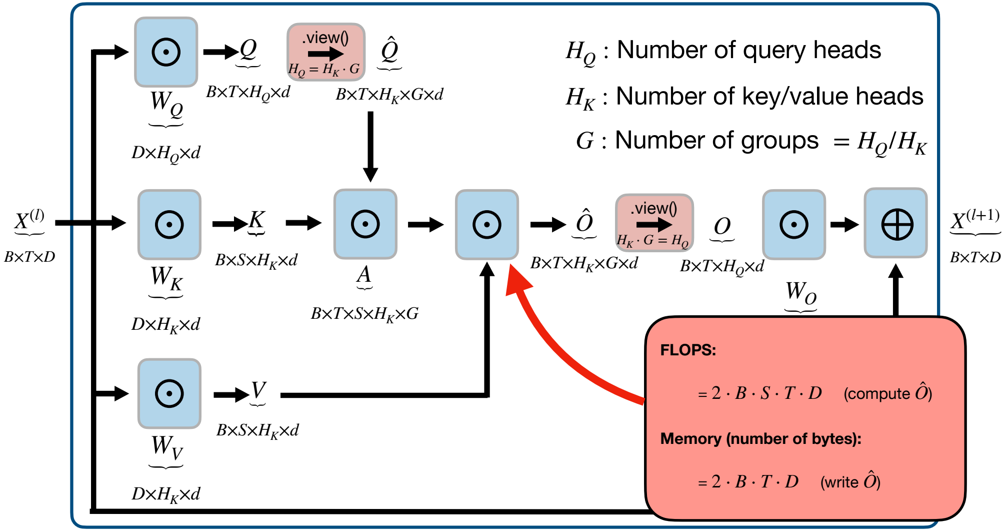
$$= 4 \cdot B\,S\,H_K\,d + 2 \cdot B\,T\,D$$

(read Q, K, V)

FLOPS:

$$= 2 \cdot B\,T\,S\,D \quad \text{(compute } A\text{)}$$

GQA Arithmetic


$\underbrace{X^{(l)}}_{B\times T\times D}$
$\underbrace{W_Q}_{D\times H_Q\times d}$
$\underbrace{Q}_{B\times T\times H_Q\times d}$
.view()
$H_Q = H_K \cdot G$
$\underbrace{\hat{Q}}_{B\times T\times H_K\times G\times d}$
$\underbrace{W_K}_{D\times H_K\times d}$
$\underbrace{K}_{B\times S\times H_K\times d}$
$\underbrace{A}_{B\times T\times S\times H_K\times G}$
$\underbrace{W_V}_{D\times H_K\times d}$
$\underbrace{V}_{B\times S\times H_K\times d}$
$\underbrace{\hat{O}}_{B\times T\times H_K\times G\times d}$
.view()
$H_K \cdot G = H_Q$
$\underbrace{O}_{B\times T\times H_Q\times d}$
$\underbrace{W_O}$
$\underbrace{X^{(l+1)}}_{B\times T\times D}$

H_Q : Number of query heads

H_K : Number of key/value heads

G : Number of groups $= H_Q/H_K$

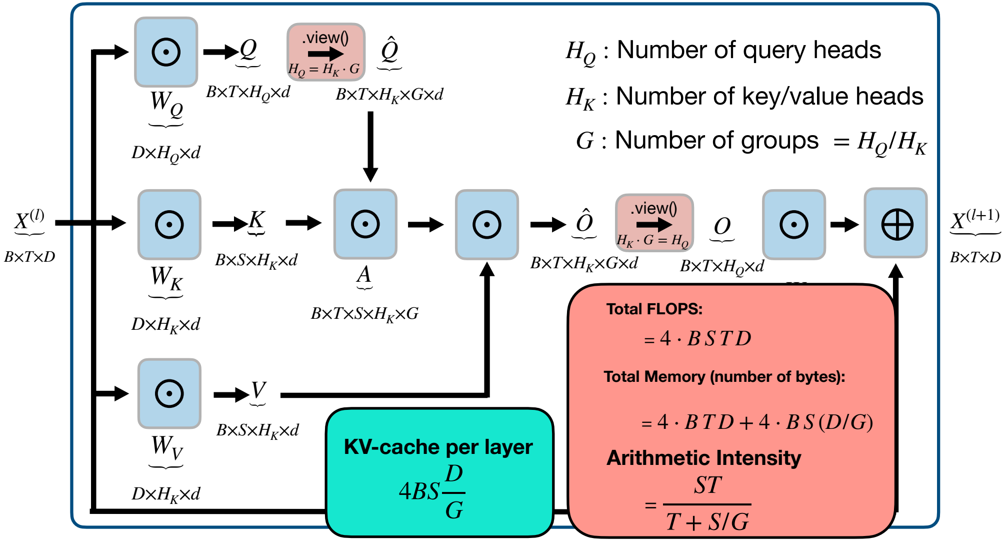
FLOPS:

$= 2 \cdot B \cdot S \cdot T \cdot D$ (compute $\hat{O}$)

Memory (number of bytes):

$= 2 \cdot B \cdot T \cdot D$ (write $\hat{O}$)

GQA Arithmetic



$\underbrace{X^{(l)}}_{B\times T\times D}$
$\underbrace{W_Q}_{D\times H_Q\times d}$
$\underbrace{Q}_{B\times T\times H_Q\times d}$
.view()
$H_Q = H_K \cdot G$
$\underbrace{\hat{Q}}_{B\times T\times H_K\times G\times d}$
$\underbrace{W_K}_{D\times H_K\times d}$
$\underbrace{K}_{B\times S\times H_K\times d}$
$\underbrace{A}_{B\times T\times S\times H_K\times G}$
$\underbrace{\hat{O}}_{B\times T\times H_K\times G\times d}$
.view()
$H_K \cdot G = H_Q$
$\underbrace{O}_{B\times T\times H_Q\times d}$
$\underbrace{X^{(l+1)}}_{B\times T\times D}$
$\underbrace{W_V}_{D\times H_K\times d}$
$\underbrace{V}_{B\times S\times H_K\times d}$

H_Q : Number of query heads

H_K : Number of key/value heads

G : Number of groups $= H_Q/H_K$

KV-cache per layer

$$4BS\frac{D}{G}$$

Total FLOPS:

$$= 4 \cdot BSTD$$

Total Memory (number of bytes):

$$= 4 \cdot BTD + 4 \cdot BS(D/G)$$

Arithmetic Intensity

$$= \frac{ST}{T + S/G}$$

GQA Arithmetic Summary

$$\begin{aligned}\text{Intensity}: \quad & \frac{ST}{T+S} \Rightarrow \frac{ST}{T+S/G} \\ \text{Memory}: \quad & 4BSD \Rightarrow 4BS\frac{D}{G}.\end{aligned}$$

GQA Arithmetic Summary

$$\begin{aligned} \text{Intensity}: &\quad \frac{ST}{T+S} \Rightarrow \frac{ST}{T+S/G} \\ \text{Memory}: &\quad 4BSD \Rightarrow 4BS\frac{D}{G}. \end{aligned}$$

Prefill $(T = S)$

$$\text{FLOPs} \approx 4BS^2D$$

$$\text{Memory} \approx 4BSD + 4BS(D/G)$$

$$\text{Intensity} = \frac{S}{1+1/G} = \frac{GS}{G+1}$$

- Slightly better than MHA

GQA Arithmetic Summary

$$\text{Intensity}: \quad \frac{ST}{T+S} \Rightarrow \frac{ST}{T+S/G}$$
$$\text{Memory}: \quad 4BSD \Rightarrow 4BS\frac{D}{G}.$$

Prefill ($T = S$)

$$\text{FLOPs} \approx 4BS^2D$$
$$\text{Memory} \approx 4BSD + 4BS(D/G)$$
$$\text{Intensity} = \frac{S}{1+1/G} = \frac{GS}{G+1}$$

- Slightly better than MHA

Generation ($T = 1$)

$$\text{FLOPs} \approx 4BSD$$
$$\text{Memory} \approx 4BD + 4BS(D/G)$$
$$\text{Intensity} = \frac{S}{1+S/G} \approx G \quad (S \gg G)$$

- Higher intensity than MHA
- Smaller KV-cache reads by factor G

GQA Arithmetic Summary

$$\text{Intensity}: \quad \frac{ST}{T+S} \Rightarrow \frac{ST}{T+S/G}$$
$$\text{Memory}: \quad 4BSD \Rightarrow 4BS\frac{D}{G}.$$

Prefill ($T = S$)

$$\text{FLOPs} \approx 4BS^2D$$
$$\text{Memory} \approx 4BSD + 4BS(D/G)$$
$$\text{Intensity} = \frac{S}{1+1/G} = \frac{GS}{G+1}$$

- Slightly better than MHA

Generation ($T = 1$)

$$\text{FLOPs} \approx 4BSD$$
$$\text{Memory} \approx 4BD + 4BS(D/G)$$
$$\text{Intensity} = \frac{S}{1+S/G} \approx G \quad (S \gg G)$$

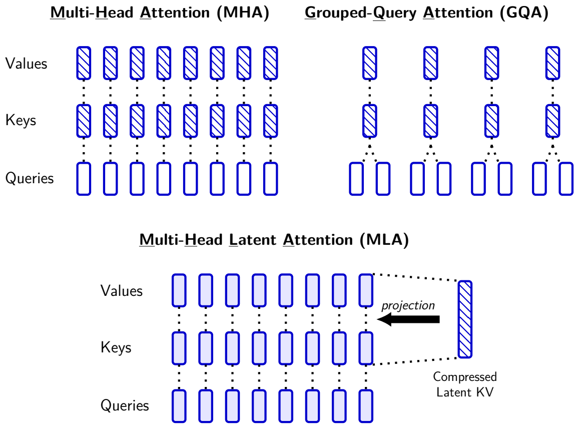
- Higher intensity than MHA
- Smaller KV-cache reads by factor G

Takeaway

Prefill: still good, Generation: latency and memory lower than MHA .. and same FLOPS!

Other inference techniques

Multi-Head Latent Attention (MLA), **(DeepSeek v2, 2024)**


Multi-Head Attention (MHA)
Grouped-Query Attention (GQA)
Values
Keys
Queries
Multi-Head Latent Attention (MLA)
Values
Keys
Queries
projection
Compressed Latent KV

Idea: K/V in a small latent space

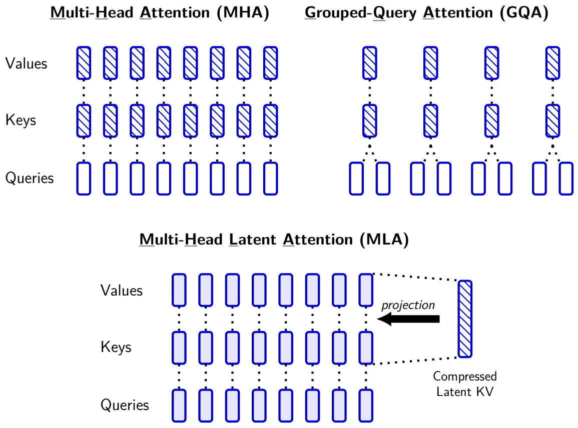
Instead of caching keys/values in Hd, store in **small latent state** of size C

$$c_t^{KV} = W^{DKV} h_t, \qquad c_t^{KV} \in \mathbb{R}^C.$$

The cached latent is mapped back by learned **up-projections**

$$k_t^C = W^{UK} c_t^{KV}, \qquad v_t^C = W^{UV} c_t^{KV}.$$

Multi-Head Latent Attention (MLA), **(DeepSeek v2, 2024)**


Multi-Head Attention (MHA)
Grouped-Query Attention (GQA)
Values
Keys
Queries
Multi-Head Latent Attention (MLA)
Values
Keys
Queries
projection
Compressed Latent KV

Idea: K/V in a small latent space

Instead of caching keys/values in Hd, store in **small latent state** of size C

$$c_t^{KV} = W^{DKV} h_t, \qquad c_t^{KV} \in \mathbb{R}^C.$$

The cached latent is mapped back by learned **up-projections**

$$k_t^C = W^{UK} c_t^{KV}, \qquad v_t^C = W^{UV} c_t^{KV}.$$

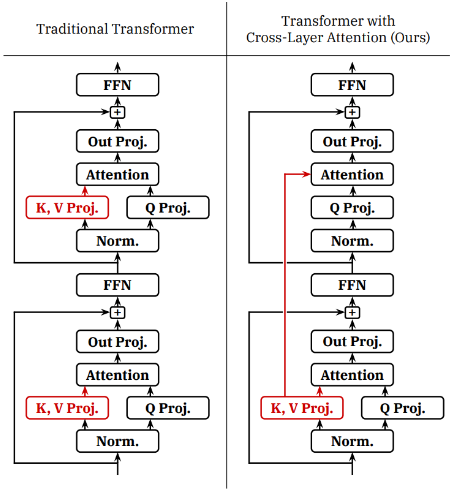
What about RoPE?

Keep RoPE in a small separate part:

$$C_{\text{total}} = C + d_{\text{RoPE}}.$$

$$Hd = 16384, \qquad C = 512, \qquad C_{\text{total}} = 576.$$

Cross-Layer Attention (CLA), **(Brandon et al., 2024)**



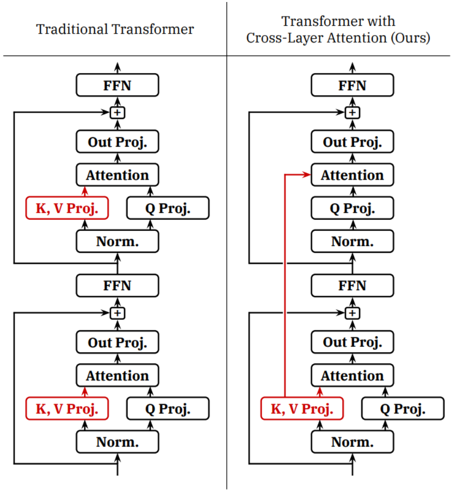
Idea: Share KV across *layers*

With sharing factor r, one KV projection/cache is reused by r adjacent layers:

CLA2: 2 layers, CLA3: 3 layers.

Only a subset of layers computes fresh K, V; the others reuse earlier-layer K, V.

Cross-Layer Attention (CLA), **(Brandon et al., 2024)**



Idea: Share KV across *layers*

With sharing factor r, one KV projection/cache is reused by r adjacent layers:

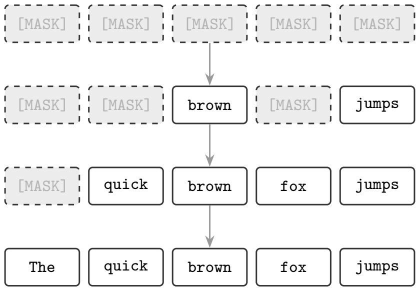
CLA2: 2 layers, CLA3: 3 layers.

Only a subset of layers computes fresh K, V; the others reuse earlier-layer K, V.

Why useful?

- KV-cache memory shrinks by about $r\times$
- Less KV-cache bandwidth during decoding
- Better latency / throughput at fixed memory budget
- Can be combined with GQA

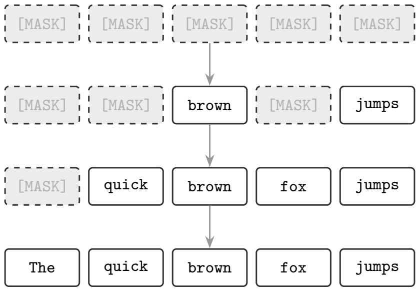
Masked diffusion models for text **(Li et al., 2022)**


[MASK] [MASK] [MASK] [MASK] [MASK]
[MASK] [MASK] brown [MASK] jumps
[MASK] quick brown fox jumps
The quick brown fox jumps

Idea: Parallel Text Generation

Abandon the causal by one generation (left-to-right). Generate parts of the sequence simultaneously through iterative unmasking.

Masked diffusion models for text **(Li et al., 2022)**



[MASK] [MASK] [MASK] [MASK] [MASK]
[MASK] [MASK] brown [MASK] jumps
[MASK] quick brown fox jumps
The quick brown fox jumps

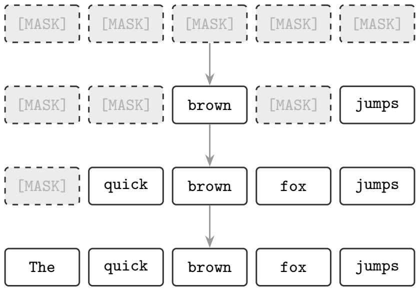
Idea: Parallel Text Generation

Abandon the causal by one generation (left-to-right). Generate parts of the sequence simultaneously through iterative unmasking.

How it works:

- **Initialize:** Start with an entire `[MASK]` sequence.
- **Predict:** Guess the missing words all at once.
- **Refine:** Lock in the most confident predictions, re-mask the rest, and repeat.

Masked diffusion models for text **(Li et al., 2022)**


[MASK] [MASK] [MASK] [MASK] [MASK]
[MASK] [MASK] brown [MASK] jumps
[MASK] quick brown fox jumps
The quick brown fox jumps

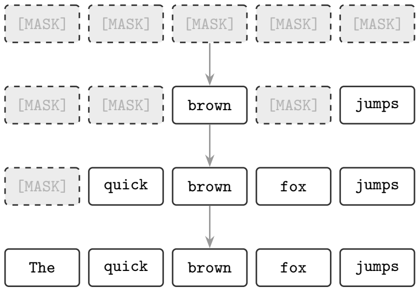
Idea: Parallel Text Generation

Abandon the causal by one generation (left-to-right). Generate parts of the sequence simultaneously through iterative unmasking.

How it works:

- **Initialize:** Start with an entire `[MASK]` sequence.
- **Predict:** Guess the missing words all at once.
- **Refine:** Lock in the most confident predictions, re-mask the rest, and repeat.

Masked diffusion models for text **(Li et al., 2022)**


[MASK] [MASK] [MASK] [MASK] [MASK]
[MASK] [MASK] brown [MASK] jumps
[MASK] quick brown fox jumps
The quick brown fox jumps

Idea: Parallel Text Generation

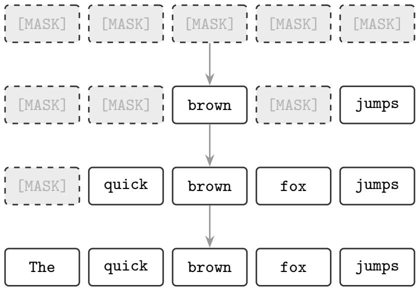
Abandon the causal by one generation (left-to-right). Generate parts of the sequence simultaneously through iterative unmasking.

How it works:

- **Initialize:** Start with an entire `[MASK]` sequence.
- **Predict:** Guess the missing words all at once.
- **Refine:** Lock in the most confident predictions, re-mask the rest, and repeat.

- **Throughput:** Updating many tokens in parallel bypasses the standard autoregressive bottleneck $\Rightarrow$ much faster decoding (generation).

Masked diffusion models for text **(Li et al., 2022)**


[MASK] [MASK] [MASK] [MASK] [MASK]
[MASK] [MASK] brown [MASK] jumps
[MASK] quick brown fox jumps
The quick brown fox jumps

Idea: Parallel Text Generation

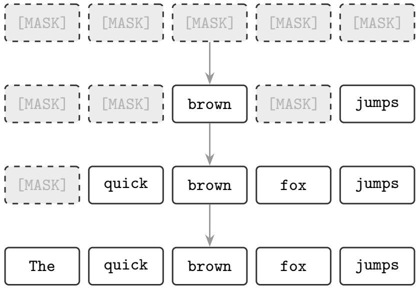
Abandon the causal by one generation (left-to-right). Generate parts of the sequence simultaneously through iterative unmasking.

How it works:

- **Initialize:** Start with an entire `[MASK]` sequence.
- **Predict:** Guess the missing words all at once.
- **Refine:** Lock in the most confident predictions, re-mask the rest, and repeat.

- **Throughput:** Updating many tokens in parallel bypasses the standard autoregressive bottleneck $\Rightarrow$ much faster decoding (generation).
- **Flexibility:** Allows for **infilling**, editing existing parts, and constrained generation.

Masked diffusion models for text **(Li et al., 2022)**


[MASK] [MASK] [MASK] [MASK] [MASK]
[MASK] [MASK] brown [MASK] jumps
[MASK] quick brown fox jumps
The quick brown fox jumps

Idea: Parallel Text Generation

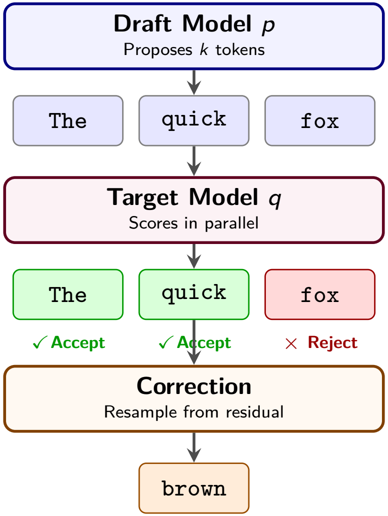
Abandon the causal by one generation (left-to-right). Generate parts of the sequence simultaneously through iterative unmasking.

How it works:

- **Initialize:** Start with an entire [MASK] sequence.
- **Predict:** Guess the missing words all at once.
- **Refine:** Lock in the most confident predictions, re-mask the rest, and repeat.

- **Throughput:** Updating many tokens in parallel bypasses the standard autoregressive bottleneck $\Rightarrow$ much faster decoding (generation).
- **Flexibility:** Allows for **infilling**, editing existing parts, and constrained generation.
- **Challenges:** Dependencies between tokens, discrete distribution (different than images).

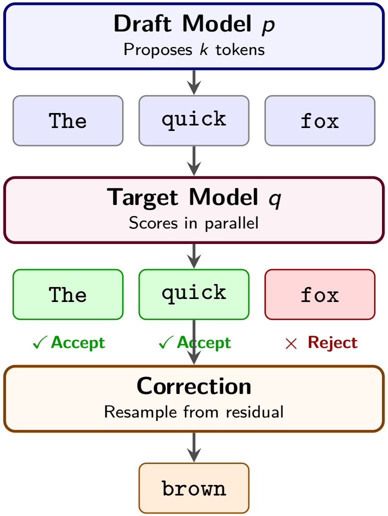
Speculative decoding **(Leviathan et al., 2022)**



Idea: Cheap generation + Full check

- A small **draft model** p to get a short token block
- A large **target model** q to test the block (parallel)
- Accept tokens based on $q \Rightarrow$ correct the residual.

Speculative decoding **(Leviathan et al., 2022)**



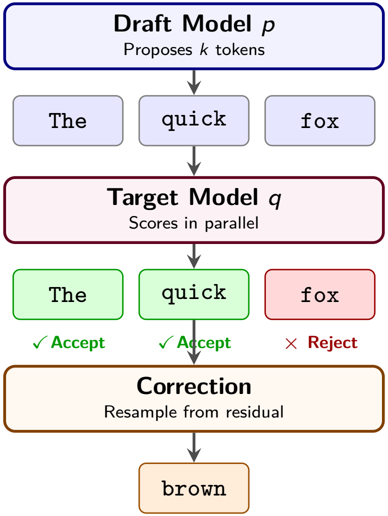
Idea: Cheap generation + Full check

- A small **draft model** p to get a short token block
- A large **target model** q to test the block (parallel)
- Accept tokens based on $q \Rightarrow$ correct the residual.

Modified rejection sampling

- Proposal distribution: p, target distribution: q
- Designed so we *always make progress*
- Output is an *exact sample from the target q*

Speculative decoding **(Leviathan et al., 2022)**



Idea: Cheap generation + Full check

- A small **draft model** p to get a short token block
- A large **target model** q to test the block (parallel)
- Accept tokens based on $q \Rightarrow$ correct the residual.

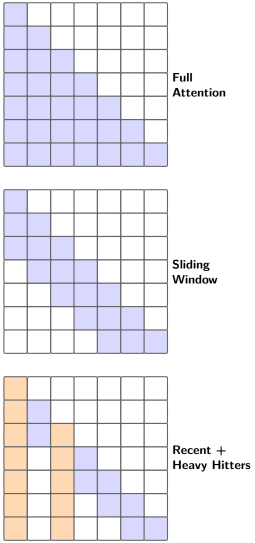
Modified rejection sampling

- Proposal distribution: p, target distribution: q
- Designed so we *always make progress*
- Output is an *exact sample from the target* q

Takeaway

- Draft model is accurate $\Rightarrow$ accept many guesses.
- One full target-model pass validates many tokens

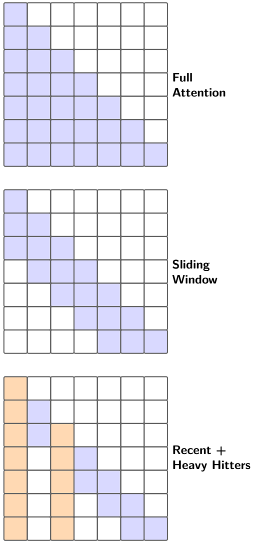
Sparsifying the context, **(Child et al., 2019)**



Idea: The attention matrices are sparse

- Tokens often attend only to small number of (nearby) tokens.
- Attention cost becomes *linear* in length.
- Multiple layers still allow for longer dependencies.
- *Generation:* KV size is bounded by the *window*.

Sparsifying the context, **(Child et al., 2019)**



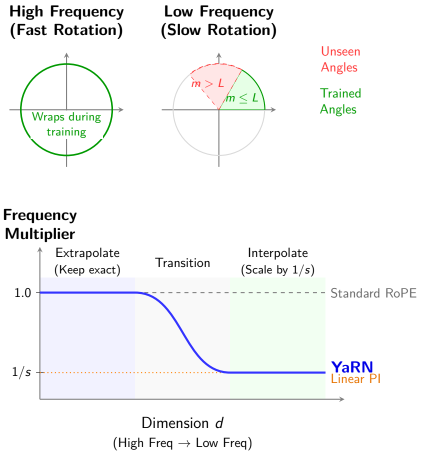
Idea: The attention matrices are sparse

- Tokens often attend only to small number of (nearby) tokens.
- Attention cost becomes *linear* in length.
- Multiple layers still allow for longer dependencies.
- *Generation:* KV size is bounded by the *window*.

Heavy hitters (H_2O) (Zhang et al., 2023)

- Some tokens are globally important.
- A useful compromise is to keep:
  - the **most recent** tokens, and
  - a small set of tokens that had large attention mass.
- *Preserve* a few globally important tokens, sparsify others.
- *Many* other works on sparsifying/reducing attention.

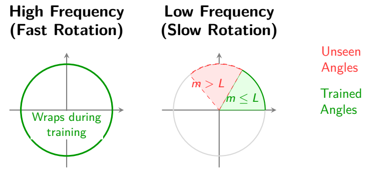
Overcoming limits of RoPE with YaRN **(Peng et al., 2023)**

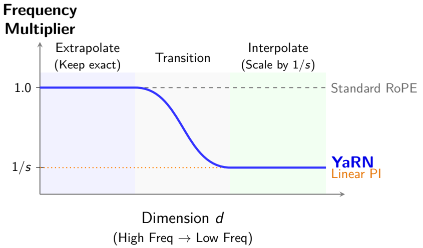


Problem: RoPE extrapolates poorly

- Trained only up to some maximum context length L (e.g. $4\mathrm{k}$).
- For $T > L$, RoPE induces relative phases the model was not trained to use.
- For long T, positions collide $\Rightarrow$ model cannot distinguishing between nearby and distant tokens.

Overcoming limits of RoPE with YaRN **(Peng et al., 2023)**





Problem: RoPE extrapolates poorly

- Trained only up to some maximum context length L (e.g. 4k).
- For $T > L$, RoPE induces relative phases the model was not trained to use.
- For long T, positions collide $\Rightarrow$ model cannot distinguishing between nearby and distant tokens.

Idea: Rescale low-frequencies

- Compress positions back into the training range.
- Apply *different scaling to different RoPE frequency bands*.
- Preserve *local/high-frequency* behavior and stabilize the **long-range/low-frequency** behavior.

Takeaways

Takeaways: Inference

- **Prefill** is parallel and often efficient; **decode** is sequential and much harder.

Takeaways: Inference

- **Prefill** is parallel and often efficient; **decode** is sequential and much harder.
- The **KV cache** makes autoregressive generation possible, but creates a memory-bandwidth bottleneck.

Takeaways: Inference

- **Prefill** is parallel and often efficient; **decode** is sequential and much harder.
- The **KV cache** makes autoregressive generation possible, but creates a memory-bandwidth bottleneck.
- During decode, **attention is usually memory-limited**, so batching mainly improves **throughput**, not **latency**.

Takeaways: Inference

- **Prefill** is parallel and often efficient; **decode** is sequential and much harder.
- The **KV cache** makes autoregressive generation possible, but creates a memory-bandwidth bottleneck.
- During decode, **attention is usually memory-limited**, so batching mainly improves **throughput**, not **latency**.
- Modern inference methods mostly try to:
  - reduce KV-cache size/bandwidth (GQA, MLA, CLA, local attention),
  - or recover parallelism (speculative decoding).

Takeaways: Inference

- **Prefill** is parallel and often efficient; **decode** is sequential and much harder.
- The **KV cache** makes autoregressive generation possible, but creates a memory-bandwidth bottleneck.
- During decode, **attention is usually memory-limited**, so batching mainly improves **throughput**, not **latency**.
- Modern inference methods mostly try to:
  - reduce KV-cache size/bandwidth (GQA, MLA, CLA, local attention),
  - or recover parallelism (speculative decoding).
- **Final takeaway:** LLM inference is not only a "systems" problem, but also an architecture modeling problem!