

lecture_10.py



```
1 from sympy import symbols, oo
2 from execute_util import text, link, image
3 from lecture_util import article_link
4 from references import Reference, llama3, gqa, mla, longformer, sparse_transformer, mistral_7b
5
6 # Define symbols corresponding to the shape of the Transformer model
7 B, S, T, D, F, N, K, H, L, V = symbols("B S T D F N K H L V", positive=True)
8 c = symbols("c", positive=True) # Just a constant that helps with taking limits
9 memory_bandwidth = symbols("memory_bandwidth", positive=True)
10
11 scaling_book_transformers = Reference(title="[Scaling book chapter on Transformers]", url="https://jax-ml.github.io/scaling-book/transformers/")
12 scaling_book_inference = Reference(title="[Scaling book chapter on Transformers]", url="https://jax-ml.github.io/scaling-book/inference/")
13
14 def main():
15     Inference: given a fixed model, generate responses given prompts
16
17
```

Understanding the inference workload

```
18 landscape()
19 review_transformer()
20 review_of_arithmetic_intensity()
21 arithmetic_intensity_of_inference()
22 throughput_and_latency()
23
24
```

Taking shortcuts (lossy)

```
25 reduce_kv_cache_size()
26 alternatives_to_the_transformer()
27 quantization()
28 model_pruning()
29
30 Summary: reduce inference complexity without hurting accuracy
31
32 From scratch recipe:
33 1. Define faster model architecture
34 2. Train faster model
35
36 Distillation recipe:
37 1. Define faster model architecture
38 2. Initialize weights using original model (which has a different architecture)
39 3. Repair faster model (distillation)
40
41
```

Use shortcuts but double check (lossless)

```
42 speculative_sampling()
43
44
```

Handling dynamic workloads

```
45 Batching over sequences in live traffic is tricky because:
46 1. Requests arrive at different times (waiting for batch is bad for early requests)
```

```
47    2. Sequences have shared prefixes (e.g., system prompts, generating multiple samples)
48    3. Sequences have different lengths (padding is inefficient)
49
50    continuous_batching()
51    paged_attention()
52
53
```

Summary

```
54    • Inference is important (actual use, evaluation, reinforcement learning)
55    • Different characteristics compared to training (memory-limited, dynamic)
56    • Techniques: new architectures, quantization, pruning/distillation, speculative decoding
57    • Ideas from systems (speculative execution, paging)
58    • New architectures have huge potential for improvement
59
60
```

```
61 def landscape():
62     Inference shows up in many places:
63     • Actual use (chatbots, code completion, batch data processing)
64     • Model evaluation (e.g., on instruction following)
65     • Test-time compute (thinking requires more inference)
66     • Training via reinforcement learning (sample generation, then score)
67
68 Why efficiency matters: training is one-time cost, inference is repeated many times
```



Sam Altman ✅

@sama

openai now generates about 100 billion words per day.

all people on earth generate about 100 trillion words per day.

2:55 PM · Feb 9, 2024 · 2.8M Views



[tweet]



Aman Sanger ✅

@amanrsanger

Cursor writes almost 1 billion lines of accepted code a day.

To put it in perspective, the entire world produces just a few billion lines a day.

2:29 PM · Apr 28, 2025 · 94.1K Views



[tweet]

```
71
72 Metrics:
73    • Time-to-first-token (TTFT): how long user waits before any generation happens (matters for interactive
74      applications)
75    • Latency (seconds/token): how fast tokens appear for a user (matters for interactive applications)
76    • Throughput (tokens/second): useful for batch processing applications
```

```
77 Key considerations in efficiency:
78    • Training (supervised): you see all tokens, can parallelize over sequence (matmul in Transformer)
79    • Inference: you have to generate sequentially, can't parallelize, so harder to fully utilize compute
```

```
81 Companies doing inference (a big deal for anyone who has a product or platform):
82    • Providers serving closed models (OpenAI, Anthropic, Google, etc.)
83    • Providers serving open-weight models (Together, Fireworks, DeepInfra, etc.)
```

```
85 Open-source packages:
86    • vLLM (Berkeley) [talk]
87    • Tensor-RT (NVIDIA) [article]
```

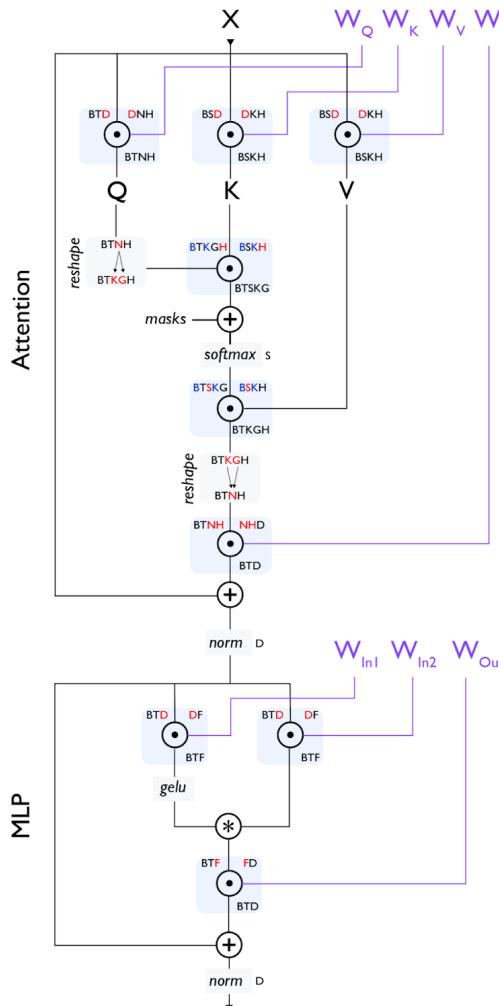
- 88 • TGI (Hugging Face) [article]

89

90

```
91 def review_transformer():
92     [Scaling book chapter on Transformers]
```

93



symbol	dimension
B	batch
L	number of layers
T	sequence length (query)
S	sequence length (key value)
V	vocab
D	d_{model} , embedding dimension
F	MLP hidden dimension
H	attention head dimension
N	number of query heads
K	number of key/value heads
G	q heads per kv head = $N // K$

94 Simplifications (following conventions): $F = 4*D$, $D = N*H$, $N = K*G$, $S = T$

95 FLOPs for a feedforward pass: $6 * (B*T) * (\text{num_params} + O(T))$

96

97

```
98 def review_of_arithmetic_intensity():
99     Setup: multiply X (B x D) and W (D x F) matrix
```

100 Intuition: B is batch size, D is hidden dimension, F is up-projection dimension in MLP

101

102 Let's do FLOPs and memory read/write accounting for the matrix multiplication ($X * W$).

103 flops = 0

104 bytes_transferred = 0

105

106 Steps:

107 1. Read X (B x D) from HBM

108 bytes_transferred += 2*B*D

109 2. Read W (D x F) from HBM

110 bytes_transferred += 2*D*F

111 3. Compute Y = X (B x D) @ W (D x F)

112 flops += 2*B*D*F

113 4. Write Y (B x F) to HBM

114 bytes_transferred += 2*B*F

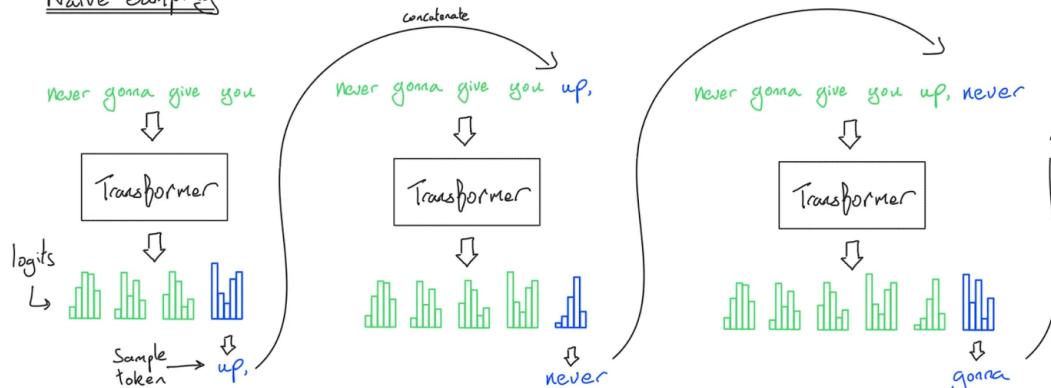
115

```

116 Let's take stock of the accounting results.
117 assert flops == 2*B*D*F
118 assert bytes_transferred == 2*B*D + 2*D*F + 2*B*F
119 Recall that arithmetic intensity is how much compute we do per byte transferred (want to be high).
120 intensity = (flops / bytes_transferred).simplify() # @inspect intensity
121
122 Assuming B is much less than D and F, then we can simplify:
123 intensity = intensity.subs(D, c*B).subs(F, c*B).limit(c, oo).simplify() # @inspect intensity
124 assert intensity == B
125
126 Accelerator intensity of H100:
127 flops_per_second = 989e12
128 memory_bandwidth = 3.35e12
129 accelerator_intensity = flops_per_second / memory_bandwidth # @inspect accelerator_intensity
130 assert round(accelerator_intensity) == 295
131
132 If computation intensity > accelerator intensity, compute-limited (good)
133 If computation intensity < accelerator intensity, memory-limited (bad)
134 Conclusion: compute-limited iff B > 295
135
136 Extreme case (B = 1, corresponding to matrix-vector product):
137 • Arithmetic intensity: 1
138 • Memory-limited (read D x F matrix, perform only 2DF FLOPs)
139 • This is basically what happens with generation...
140
141
142 def arithmetic_intensity_of_inference():
143     [Scaling book chapter on Transformers]
144
145

```

Naive Sampling

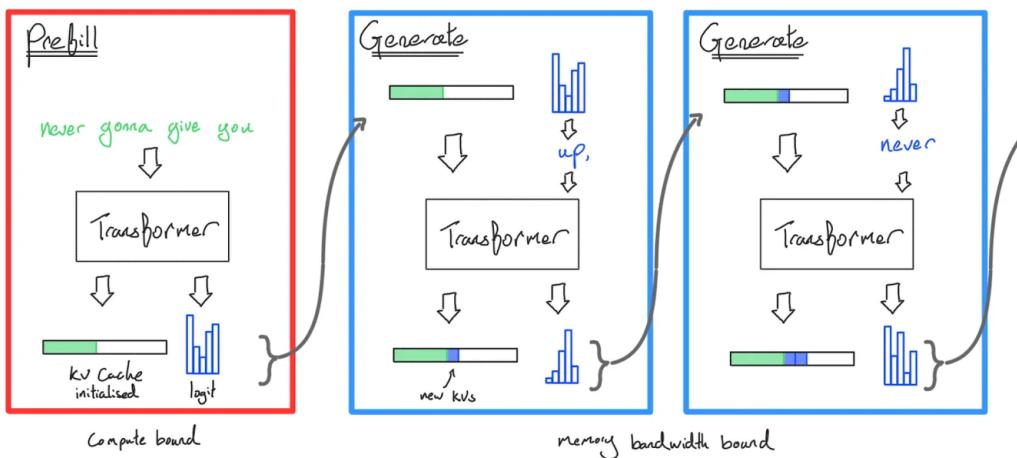


```

146 Naive inference: to generate each token, feed history into Transformer
147 Complexity: generating T tokens requires O(T^3) FLOPs (one feedforward pass is O(T^2))
148
149 Observation: a lot of the work can be shared across prefixes
150 Solution: store KV cache in HBM

```

151

Sampling with KV Cache

152

KV cache: for every sequence (B), token (S), layer (L), head (K), store an H-dimensional vector

153

Two stages of inference:

1. **Prefill:** given a prompt, encode into vectors (parallelizable like in training)

2. **Generation:** generate new response tokens (sequential)

157

Let's compute the FLOPs and memory IO for both the MLP and attention layers.

S is the number of tokens we're conditioning on, T is the number of tokens we're generating.

Later, we'll specialize to prefill ($T = S$) and generation ($T = 1$).

161

162

MLP layers (only looking at the matrix multiplications)

163 flops = 0

164 bytes_transferred = 0

165 Steps:

166 1. Read X ($B \times T \times D$) from HBM

167 bytes_transferred += 2*B*T*D

168 2. Read Wup ($D \times F$), Wgate ($D \times F$), Wdown ($F \times D$) from HBM

169 bytes_transferred += 3 * 2*D*F

170 3. Compute U = X ($B \times T \times D$) @ Wup ($D \times F$)

171 flops += 2*B*T*D*F

172 4. Write U ($B \times T \times F$) to HBM

173 bytes_transferred += 2*B*T*F

174 5. Compute G = X ($B \times T \times F$) @ Wgate ($D \times F$)

175 flops += 2*B*T*D*F

176 6. Write G ($B \times T \times F$) to HBM

177 bytes_transferred += 2*B*T*F

178 7. Compute Y = GeLU(G)*U ($B \times T \times F$) @ Wdown ($F \times D$)

179 flops += 2*B*T*D*F

180 8. Write Y ($B \times T \times D$) to HBM

181 bytes_transferred += 2*B*T*D

182

Let's take stock of the accounting results.

184 assert flops == 6*B*T*D*F

185 assert bytes_transferred == 4*B*T*D + 4*B*T*F + 6*D*F

186 intensity = (flops / bytes_transferred).simplify() # @inspect intensity

187 Assume that $B*T$ is much smaller than D and F .

188 intensity = intensity.subs(D, c*B*T).subs(F, c*B*T).limit(c, oo).simplify() # @inspect intensity

189 assert intensity == B*T

190

191 For the two stages:

- ```

192 1. Prefill: easy to make compute-limited (good) by making B T large enough
193 2. Generation:
194 • Generating one token at a time (T = 1)
195 • B is number of concurrent requests, hard to make large enough!
196
197

```

## Attention layers (focusing on the matrix multiplications with FlashAttention)

```

198 flops = 0
199 bytes_transferred = 0
200 Steps:
201 1. Read Q (B x T x D), K (B x S x D), V (B x S x D) from HBM
202 bytes_transferred += 2*B*T*D + 2*B*S*D + 2*B*S*D
203 2. Compute A = Q (B x T x D) @ K (B x S x D)
204 flops += 2*B*S*T*D
205 3. Compute Y = softmax(A) (B x S x T x K x G) @ V (B x S x K x H)
206 flops += 2*B*S*T*D
207 4. Write Y (B x T x D) to HBM
208 bytes_transferred += 2*B*T*D
209
210 assert flops == 4*B*S*T*D
211 assert bytes_transferred == 4*B*S*D + 4*B*T*D
212 intensity = (flops / bytes_transferred).simplify() # @inspect intensity
213 assert intensity == S*T / (S + T)
214
215 For the two stages:
216 1. Prefill: T = S
217 prefill_intensity = intensity.subs(T, S).simplify() # @inspect prefill_intensity
218 assert prefill_intensity == S/2 # Good!
219 2. Generation: T = 1
220 generate_intensity = intensity.subs(T, 1).simplify() # @inspect generate_intensity
221 assert generate_intensity < 1 # Bad!
222
223 Unlike MLPs, no dependence on B, so batching doesn't help!
224 Why?
225 • In MLP layers, every sequence hits the same MLP weights (Wup, Wgate, Wdown don't depend on B)
226 • In attention layers, every sequence has its own vectors KV cache (Q, K, V all depend on B)
227
228 Summary
229 • Prefill is compute-limited, generation is memory-limited
230 • MLP intensity is B (requires concurrent requests), attention intensity is 1 (impossible to improve)
231
232
233 def compute_transformer_stats(config): # @inspect config
234 """Return symbols corresponding to various statistics of a Transformer."""
235 The memory, throughput, and latency depends on the shape of the Transformer.
236
237 Compute the number of parameters in the Transformer:
238 num_params = 2*V*D + D*F*3*L + (2*D*N*H + 2*D*K*H)*L
239 To store parameters, just use bf16 (training requires fp32)
240 parameter_size = num_params * 2 # 2 for bf16
241
242 We also don't need gradients and optimizer states since we're not training.
243 But we do have to store the KV cache (which are some of the activations) for each sequence (of length S):
244 How much we have to store per sequence:
245 kv_cache_size = S * (K*H) * L * 2 * 2 # 2 for key + value, 2 for bf16
246
247 Total memory usage:
248 memory = B * kv_cache_size + parameter_size

```

```
249 Latency is determined by memory IO (read all parameters and KV cache for each step)
250 latency = memory / memory_bandwidth
251 Throughput is the inverse of latency, but we're generating B tokens in parallel
252 throughput = B / latency
253
254 # Substitute
255 num_params = num_params.subs(config).simplify() # @inspect num_params
256 memory = memory.subs(config).simplify() # @inspect memory
257 latency = latency.subs(config).simplify() # @inspect latency
258 throughput = throughput.subs(config).simplify() # @inspect throughput
259
260 return num_params, memory, latency, throughput
261
262 def llama2_13b_config(args={}):
263 return {S: 1024, D: 5120, F: 13824, N: 40, K: 40, H: 128, L: 40, V: 32000, memory_bandwidth: 3.35e12, **args}
264
265 def throughput_and_latency():
266 So we have shown that inference is memory-limited.
267 Let us now compute the theoretical maximum latency and throughput of a single request.
268 Assumption: can overlap compute and communication perfectly and ignore various types of overhead.
269
270 Instantiate latency and throughput for Llama 2 13B on an H100:
271 config = llama2_13b_config()
272 num_params, memory, latency, throughput = compute_transformer_stats(config)
273
274 If we use a batch size of 1:
275 bs1_memory = memory.subs(B, 1).simplify() # @inspect bs1_memory
276 bs1_latency = latency.subs(B, 1).simplify() # @inspect bs1_latency
277 bs1_throughput = throughput.subs(B, 1).simplify() # @inspect bs1_throughput
278
279 If we use a batch size of 64 (worse latency, better throughput):
280 bs64_memory = memory.subs(B, 64).simplify() # @inspect bs64_memory
281 bs64_latency = latency.subs(B, 64).simplify() # @inspect bs64_latency
282 bs64_throughput = throughput.subs(B, 64).simplify() # @inspect bs64_throughput
283
284 If we use a batch size of 256:
285 bs256_memory = memory.subs(B, 256).simplify() # @inspect bs256_memory
286 bs256_latency = latency.subs(B, 256).simplify() # @inspect bs256_latency
287 bs256_throughput = throughput.subs(B, 256).simplify() # @inspect bs256_throughput
288 Doesn't fit into memory, but throughput gains are diminishing too...
289
290 Tradeoff between latency and throughput:
291 1. Smaller batch sizes yields better latency but worse throughput
292 2. Larger batch sizes yields better throughput but worse latency
293
294 Easy parallelism: if you launch M copies of the model, latency is the same, throughput increases by M!
295 Harder parallelism: shard the model and the KV cache [Scaling book chapter on Transformers]
296
297 Note: time-to-first-token (TTFT) is essentially a function of prefill
298 Use smaller batch sizes during prefill for faster TTFT
299 Use larger batch sizes during generation to improve throughput
300
301
302 def reduce_kv_cache_size():
303 Recall that memory is the bottleneck for inference.
304 So let's try to reduce the size of the KV cache
305 ...but make sure we don't lose too much accuracy.
```

307

## Grouped-query attention (GQA)

[Ainslie+ 2023]

308

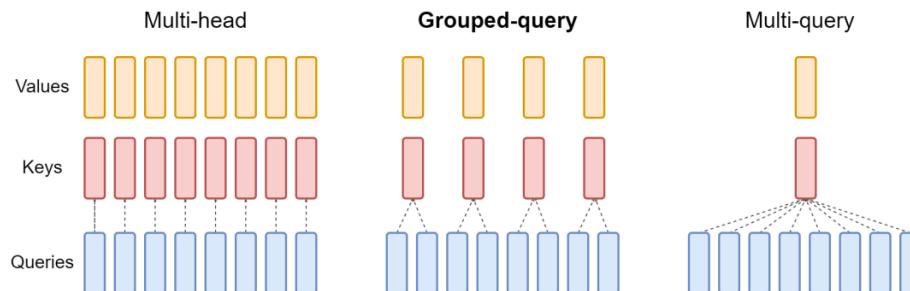


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between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

309

Idea: N query heads, but only K key and value heads, each interacting with  $N/K$  query heads

310

Multi-headed attention (MHA):  $K=N$

311

Multi-query attention (MQA):  $K=1$

312

Group-query attention (GQA):  $K$  is somewhere in between

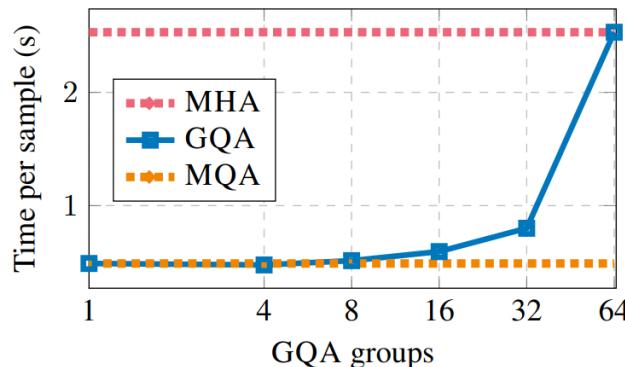
313

314

Latency/throughput improvements:

315

[Ainslie+ 2023]



316

Reduce the KV cache by a factor of  $N/K$

317

```
config = llama2_13b_config({K: 40, B: 64}) # Original Llama 2 13B
```

318

```
k40_num_params, k40_memory, k40_latency, k40_throughput = compute_transformer_stats(config) # @inspect k40_memory,
```

```
@inspect k40_latency, @inspect k40_throughput
```

319

320

```
config = llama2_13b_config({K: 8, B: 64}) # Use GQA with 1:5 ratio
```

321

```
k8_num_params, k8_memory, k8_latency, k8_throughput = compute_transformer_stats(config) # @inspect k8_memory,
```

```
@inspect k8_latency, @inspect k8_throughput
```

322

323

This also means we can use a larger batch size:

324

```
config = llama2_13b_config({K: 8, B: 256}) # Increase batch size
```

325

```
k8_bs_num_params, k8_bs_memory, k8_bs_latency, k8_bs_throughput = compute_transformer_stats(config) # @inspect
```

```
k8_bs_memory, @inspect k8_bs_latency, @inspect k8_bs_throughput
```

326

Worse latency, but better throughput (and it fits in memory now!).

327

328

Check that accuracy doesn't drop: [Ainslie+ 2023]

329

| Model     | T <sub>infer</sub> | Average | CNN            | arXiv          | PubMed         | MediaSum       | MultiNews      | WMT  | TriviaQA |
|-----------|--------------------|---------|----------------|----------------|----------------|----------------|----------------|------|----------|
|           | s                  |         | R <sub>1</sub> | BLEU | F1       |
| MHA-Large | 0.37               | 46.0    | 42.9           | 44.6           | 46.2           | 35.5           | 46.6           | 27.7 | 78.2     |
| MHA-XXL   | 1.51               | 47.2    | 43.8           | 45.6           | 47.5           | 36.4           | 46.9           | 28.4 | 81.9     |
| MQA-XXL   | 0.24               | 46.6    | 43.0           | 45.0           | 46.9           | 36.1           | 46.5           | 28.5 | 81.3     |
| GQA-8-XXL | 0.28               | 47.1    | 43.5           | 45.4           | 47.7           | 36.3           | 47.2           | 28.4 | 81.6     |

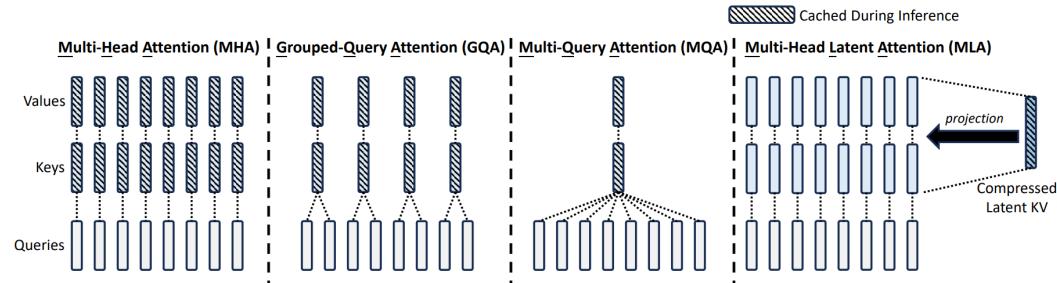
330

331

### Multi-head latent attention (MLA)

[DeepSeek-AI+ 2024]

332



333

Key idea: project down each key and value vector from N\*H dimensions to C dimensions

334

DeepSeek v2: reduce N\*H = 16384 to C = 512

335

Wrinkle: MLA is not compatible with RoPE, so need to add additional 64 dimensions for RoPE, so 512 + 64 = 576 total dimensions

336

Latency/throughput improvements follow similarly from the KV cache reduction as argued earlier

337

338

Let's now check the accuracy.

339

First, MHA is better than GQA (though more expensive) [Table 8] [DeepSeek-AI+ 2024]

340

| Benchmark (Metric) | # Shots | Dense 7B<br>w/ MHA | Dense 7B<br>w/ GQA (8 Groups) | Dense 7B<br>w/ MLA |
|--------------------|---------|--------------------|-------------------------------|--------------------|
|                    |         |                    |                               |                    |
| # Params           | -       | 7.1B               | 6.9B                          | 6.9B               |
| BBH (EM)           | 3-shot  | 33.2               | 35.6                          | <b>37.0</b>        |
| MMLU (Acc.)        | 5-shot  | 37.9               | 41.2                          | <b>45.2</b>        |
| C-Eval (Acc.)      | 5-shot  | 30.0               | 37.7                          | <b>42.9</b>        |
| CMMLU (Acc.)       | 5-shot  | 34.6               | 38.4                          | <b>43.5</b>        |

Table 8 | Comparison among 7B dense models with MHA, GQA, and MQA, respectively. MHA demonstrates significant advantages over GQA and MQA on hard benchmarks.

341

342

Second, MLA is a bit better than MHA (and much cheaper) [Table 9] [DeepSeek-AI+ 2024]

| Benchmark (Metric)             | # Shots | Small MoE<br>w/ MHA | Small MoE<br>w/ MLA | Large MoE<br>w/ MHA | Large MoE<br>w/ MLA |
|--------------------------------|---------|---------------------|---------------------|---------------------|---------------------|
|                                |         |                     |                     |                     |                     |
| # Activated Params             | -       | 2.5B                | 2.4B                | 25.0B               | 21.5B               |
| # Total Params                 | -       | 15.8B               | 15.7B               | 250.8B              | 247.4B              |
| KV Cache per Token (# Element) | -       | 110.6K              | 15.6K               | 860.2K              | 34.6K               |
| BBH (EM)                       | 3-shot  | 37.9                | <b>39.0</b>         | 46.6                | <b>50.7</b>         |
| MMLU (Acc.)                    | 5-shot  | 48.7                | <b>50.0</b>         | 57.5                | <b>59.0</b>         |
| C-Eval (Acc.)                  | 5-shot  | <b>51.6</b>         | 50.9                | 57.9                | <b>59.2</b>         |
| CMMLU (Acc.)                   | 5-shot  | 52.3                | <b>53.4</b>         | 60.7                | <b>62.5</b>         |

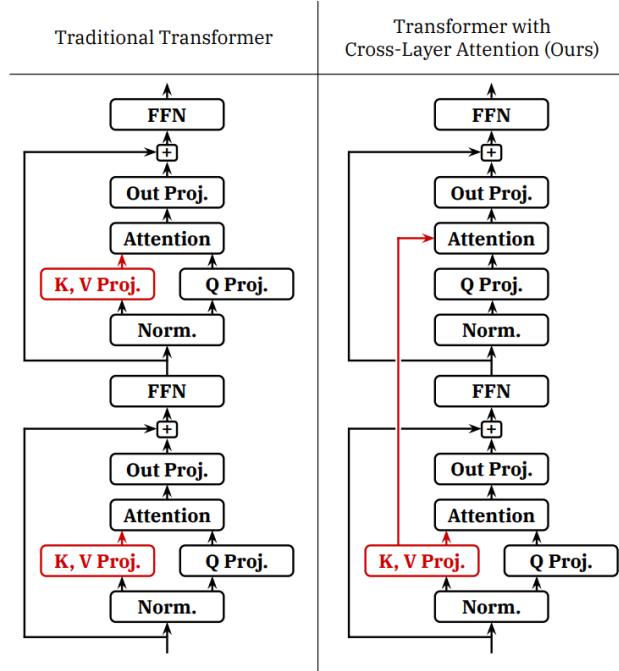
343

344

### Cross-layer attention (CLA)

[Brandon+ 2024]

345



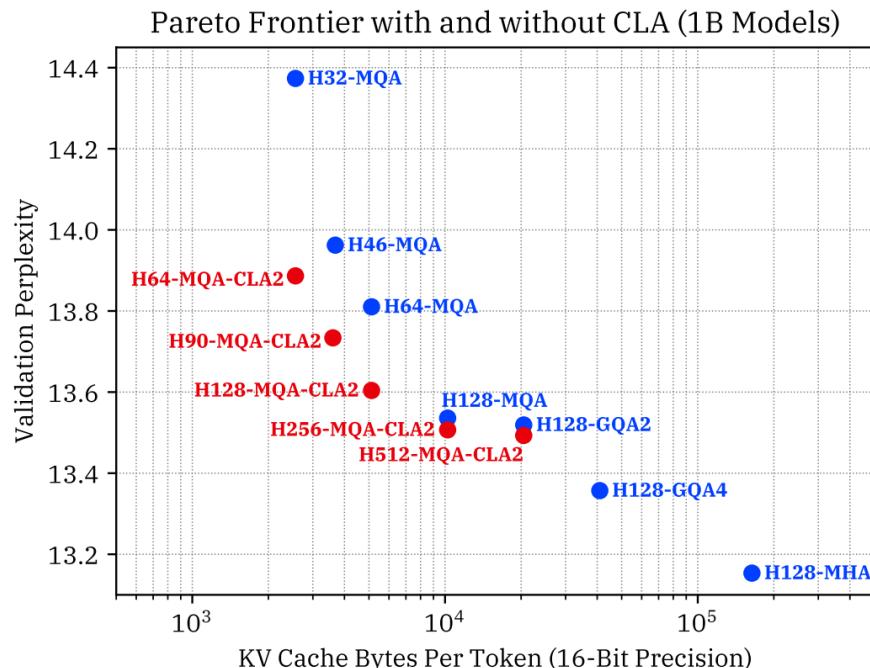
346

Idea: share KVs across **layers** (just as GQA shares KVs across heads)

347

Empirically improves the pareto frontier of accuracy and KV cache size (latency and throughput)

348



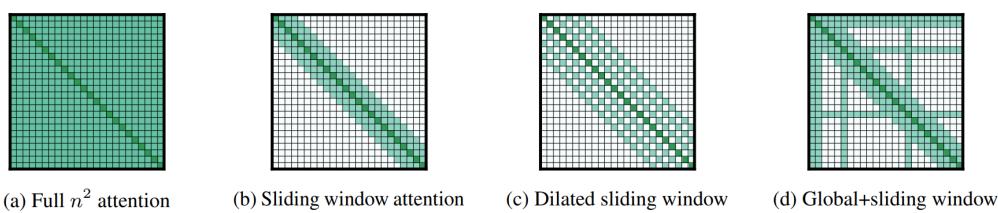
349

350

### Local attention

[Beltagy+ 2020][Child+ 2019][Jiang+ 2023]

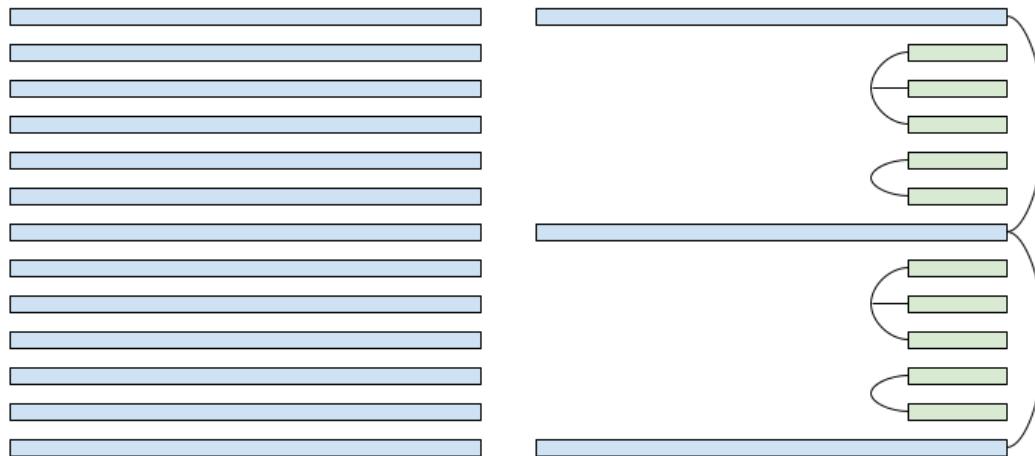
351



352

Idea: just look at the local context, which is most relevant for modeling

353 Effective context scales linearly with the number of layers  
 354 KV cache is independent of sequence length!  
 355  
 356 Problem: this can still hurt accuracy  
 357 Solution: interleave local attention with global attention (hybrid layers)  
 358 Example: character.ai uses 1 global layer every 6 layers (in addition to CLA) [article]

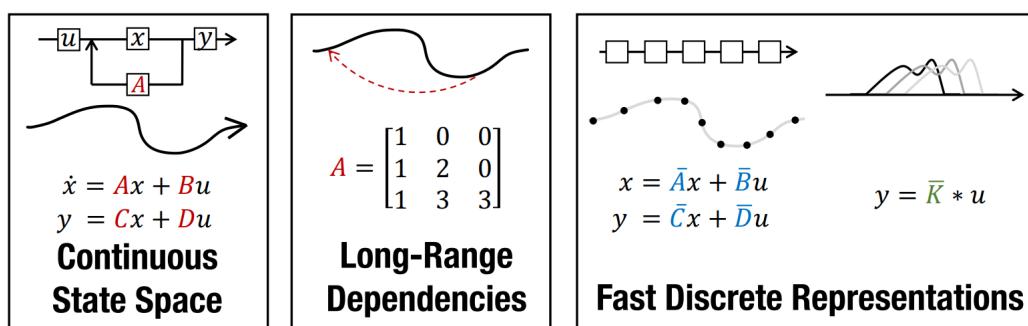


360  
 361 Summary:  
 362 • Goal: reduce the KV cache size (since inference is memory-limited) without hurting accuracy  
 363 • Lower-dimensional KV cache (GQA, MLA, shared KV cache)  
 364 • Local attention on some of the layers  
 365  
 366

367 `def alternatives_to_the_transformer():`  
 368 We have shown that tweaking the architecture of the Transformer, we can improve latency and throughput.  
 369 Attention + autoregression is fundamentally memory-limited (Transformers were not designed with inference in mind).  
 370 Can we substantially improve things if we go beyond the Transformer?  
 371 We will discuss two directions: state-space models and diffusion models.

### 373 State-space models

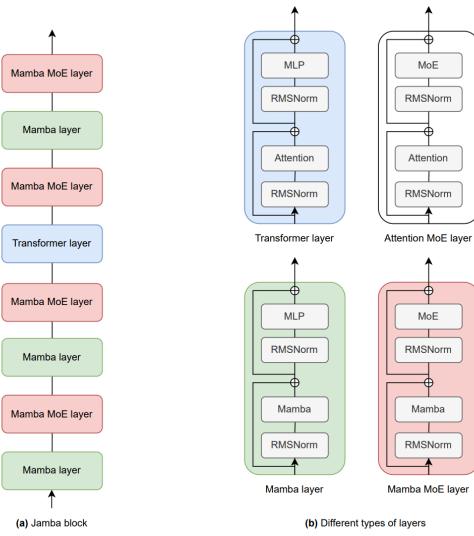
374 [presentation from CS229S]  
 375 • Idea: from signal processing to model long-context sequences in a sub-quadratic time  
 376 • S4: based on classic state space models, good at synthetic long-context tasks [Gu+ 2021]



378 • Weaknesses: bad at solving associative recall tasks important for language (where Transformers do well)  
 379  
 A 4 B 3 C 6 F 1 E 2 → A ? C ? F ? E ? B ?  
 Key-Value                                           Query

380 • Mamba: allow SSM parameters to be input-dependent, match Transformers at 1B scale [Gu+ 2023]  
 381 • Jamba: interleave Transformer-Mamba layers (1:7 ratio) with a 52B MoE [Lieber+ 2024]

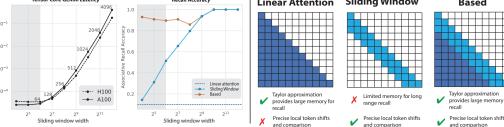
382



383

- BASED: use linear attention + local attention [Arora+ 2024]

384



385

- MiniMax-01: use linear attention + full attention (456B parameter MoE) [MiniMax+ 2025]

386

387

Takeaways:

388

- Linear + local attention (still need some full attention) yield serious SOTA models
- Replace O(T) KV cache with O(1) state => much more efficient for inference

389

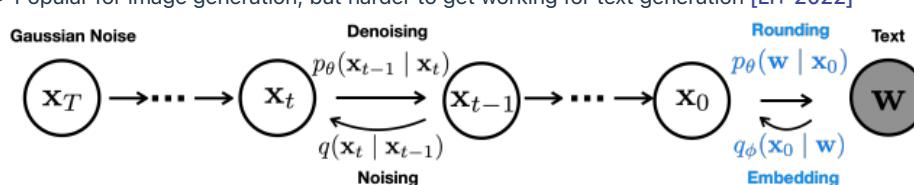
390

391

## Diffusion models

392

- Popular for image generation, but harder to get working for text generation [Li+ 2022]



394

- Idea: generate each token in parallel (not autoregressively), refine multiple time steps
- Start with random noise (over entire sequence), iteratively refine it
- Results from Inception Labs [article]

395

[demo video]

396

Much faster on coding benchmarks:

397



```

400
401 Overall, significant gains in inference to be made with more radical architecture changes!
402
403
404 def quantization():
405 Key idea: reduce the precision of numbers
406 Less memory means higher latency/throughput (since inference is memory-limited).
407 Of course we have to worry about accuracy...
408

```

## 409 Comparing number formats [article]



- ```

410 • fp32 (4 bytes): needed for parameters and optimizer states during training
411 • bf16 (2 bytes): default for inference
412 • fp8 (1 byte) [-240, 240] for e4m3 on H100s: can train if you dare [Peng+ 2023]
413 • int8 (1 byte) [-128, 127]: less accurate but cheaper than fp8, but for inference only [Baalen+ 2023]
414 • int4 (0.5 bytes) [-8, 7]: cheaper, even less accurate [Baalen+ 2023]
415

```

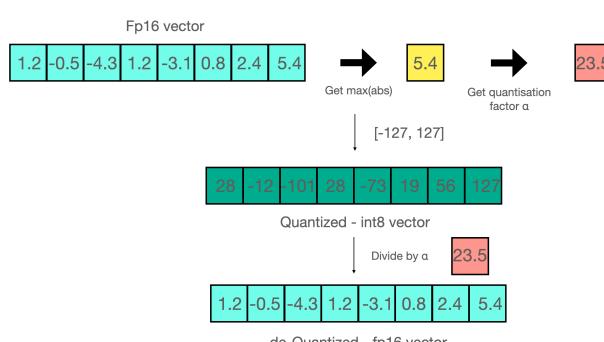
416 Quantization-aware training (QAT): train with quantization, but doesn't scale up
 417 Post-training quantization (PTQ): run on sample data to determine scale and zero point for each layer or tensor

418 [Overview of approaches]

419 LLM.int8()

420 [Dettmers+ 2022][article]

421 Standard quantization (scale by max of absolute values):



422 Problem: outliers (which appear in larger networks) screw everything up

423 Solution: extract outliers and process them in fp16

427 It works well (but is 15-23% slower than fp16):

428

benchmarks	-	-	-	-	-	difference - value
name	metric	value - int8	value - bf16	std err - bf16	-	-
hellawag	acc_norm	0.7274	0.7303	0.0044	0.0029	
hellawag	acc	0.5563	0.5584	0.005	0.0021	
piqa	acc	0.7835	0.7884	0.0095	0.0049	
piqa	acc_norm	0.7922	0.7911	0.0095	0.0011	
lambada	ppl	3.9191	3.931	0.0846	0.0119	
lambada	acc	0.6808	0.6718	0.0065	0.009	
winogrande	acc	0.7048	0.7048	0.0128	0	

429

430

Activation-aware quantization

431

[Lin+ 2023]

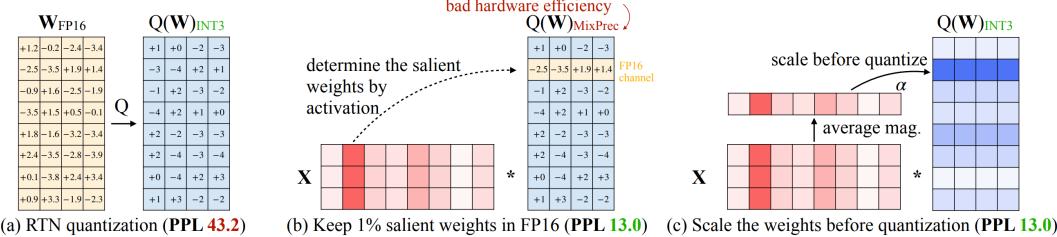
432

Idea: select which weights (0.1-1%) to keep in high precision based on activations

433

fp16 -> int3 produces 4x lower memory, 3.2x speedup

434



435

436

437 def model_pruning():

438

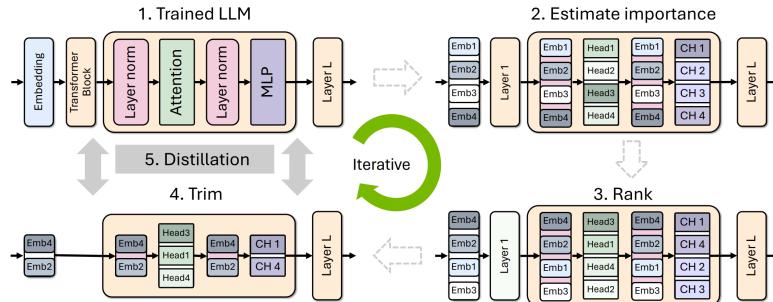
Key idea: just rip out parts of an expensive model to make it cheaper
...and then fix it up.

439

440

441 Paper from NVIDIA [Muralidharan+ 2024]

442



443

Algorithm:

444

1. Identify important {layer, head, hidden dimension} on a small calibration dataset (1024 samples)

445

2. Remove unimportant layers to get a smaller model

446

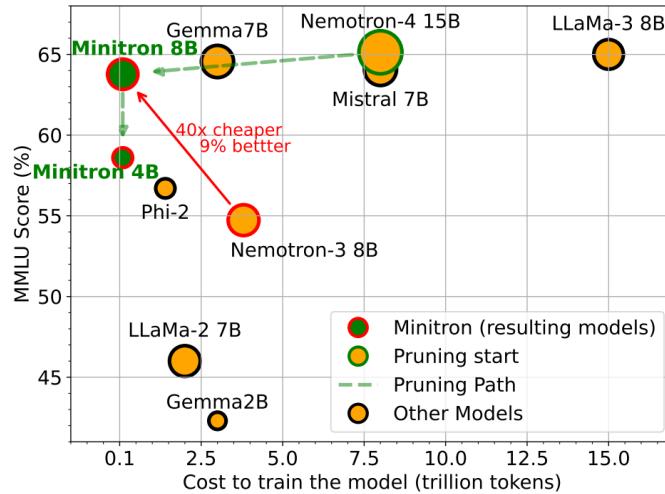
3. Distill the original model into pruned model

447

448

Results:

449



450

451

452 def speculative_sampling():

Recall the two stages of inference:

- Prefill: given a sequence, encode tokens in parallel (compute-limited) [note: also gives you probabilities]

455

- Generation: generate one token at a time (memory-limited)

456 In other words, checking is faster than generation.

457

Speculative sampling [Leviathan+ 2022][Chen+ 2023]

- Use a cheaper **draft model** p to guess a few tokens (e.g., 4)
- Evaluate with target model q (process tokens in parallel), and accept if it looks good

461 [Speculative sampling video]

462 [article]

463

464 **Algorithm 2** Speculative Sampling (SpS) with Auto-Regressive Target and Draft ModelsGiven lookahead K and minimum target sequence length T .Given auto-regressive target model $q(\cdot | \cdot)$, and auto-regressive draft model $p(\cdot | \cdot)$, initial prompt sequence x_0, \dots, x_t .Initialise $n \leftarrow t$.while $n < T$ do for $t = 1 : K$ do Sample draft auto-regressively $\tilde{x}_t \sim p(x|, x_1, \dots, x_n, \tilde{x}_1, \dots, \tilde{x}_{t-1})$

end for

 In parallel, compute $K + 1$ sets of logits from drafts $\tilde{x}_1, \dots, \tilde{x}_K$:
$$q(x|, x_1, \dots, x_n), q(x|, x_1, \dots, x_n, \tilde{x}_1), \dots, q(x|, x_1, \dots, x_n, \tilde{x}_1, \dots, \tilde{x}_K)$$
 for $t = 1 : K$ do Sample $r \sim U[0, 1]$ from a uniform distribution. if $r < \min\left(1, \frac{q(x|x_1, \dots, x_{n+t-1})}{p(x|x_1, \dots, x_{n+t-1})}\right)$, then Set $x_{n+t} \leftarrow \tilde{x}_t$ and $n \leftarrow n + 1$.

else

 sample $x_{n+t} \sim (q(x|x_1, \dots, x_{n+t-1}) - p(x|x_1, \dots, x_{n+t-1}))_+$ and exit for loop.

end if

end for

 If all tokens x_{n+1}, \dots, x_{n+K} are accepted, sample extra token $x_{n+K+1} \sim q(x|, x_1, \dots, x_n, x_{n+K})$ and set $n \leftarrow n + 1$.

end while

465

This is modified rejection sampling with proposal p and target q

466

Modification: always generate at least one candidate (rejection sampling will keep looping)

467

Key property: guaranteed to be an **exact sample** from the target model!

468

469

Proof by example: assume two vocabulary elements {A, B}

470

- Target model probabilities: $[q(A), q(B)]$

471

- Draft model probabilities: $[p(A), p(B)]$

472

- Assume $p(A) > q(A)$ [draft model oversamples A].

473

- Therefore $p(B) < q(B)$ [draft model undersamples B].

474

- Residual probabilities $\max(q-p, 0)$: $[0, 1]$

504

T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
S_1	S_1	S_1	S_1				
S_2	S_2	S_2					
S_3	S_3	S_3	S_3				
S_4	S_4	S_4	S_4	S_4			

T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
S_1	S_1	S_1	S_1	S_1	S_1	END	
S_2	END						
S_3	S_3	S_3	S_3	S_3	END		
S_4	END						

505

506 Solution: iteration-level scheduling

- 507 • Decode step by step
508 • Add new requests to the batch as they arrive (so don't have to wait until generation completes)

509

510 Problem:

- 511 • Batching only works when all sequences have the same dimensionality (right?)
512 • But each request might have a different length

513

514 Solution: selective batching

- 515 • Training: when all sequences of the same length, operate on a $B \times S \times H$ tensor
516 • But we might have different lengths: [3, H], [9, H], [5, H], etc.
517 • Attention computation: process each sequence separately
518 • Non-attention computation: concatenate all the sequences together to [3 + 9 + 5, H]

519

520

521 `def paged_attention():`

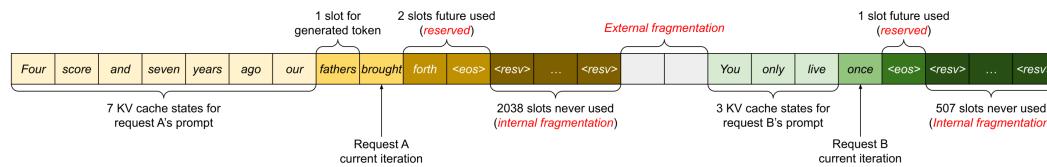
522 Paper that introduced vLLM in addition to PagedAttention [Kwon+ 2023]

523

524 Previous status quo:

- 525 • Request comes in
526 • Allocate section of KV cache for prompt and response (up to a max length)

527



528

Problem: fragmentation (what happens to your hard drive)

- But this is wasteful since we might generate much fewer tokens (internal fragmentation)!
- Might be extra unused space between sections (external fragmentation)!

529

530 Solution: PagedAttention (remember operating systems)

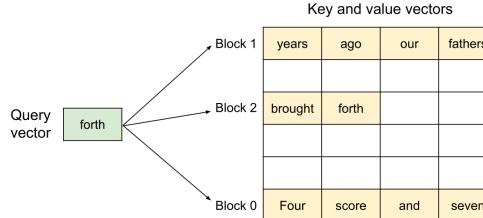
- Divide the KV cache of a sequence into non-contiguous **blocks**

531

532

533

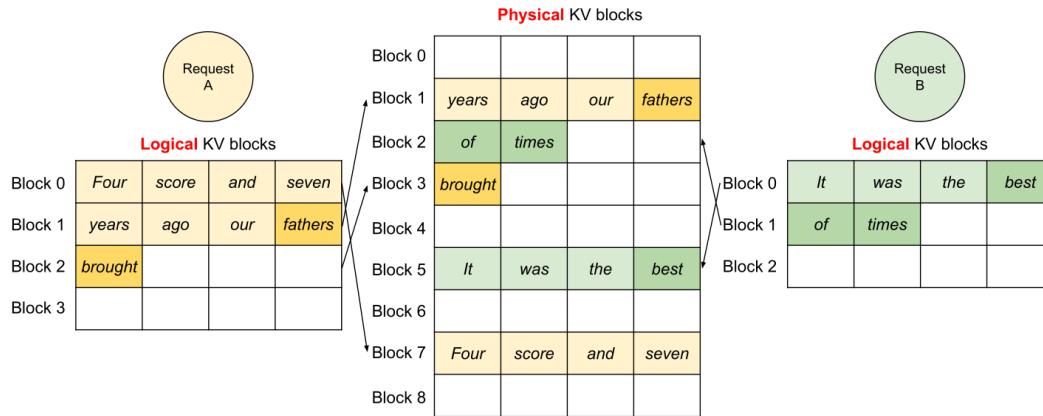
534



535

536 Two requests share the KV caches:

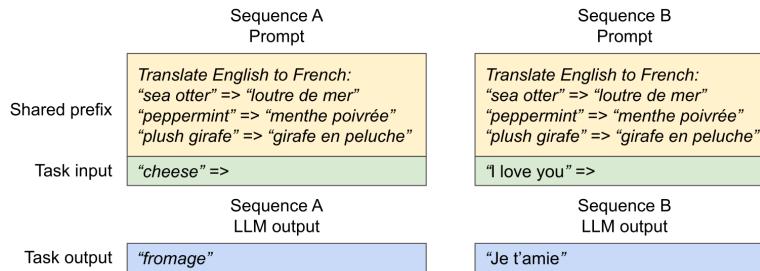
537



538

539 In general, multiple types of sharing KV caches across sequences:

540



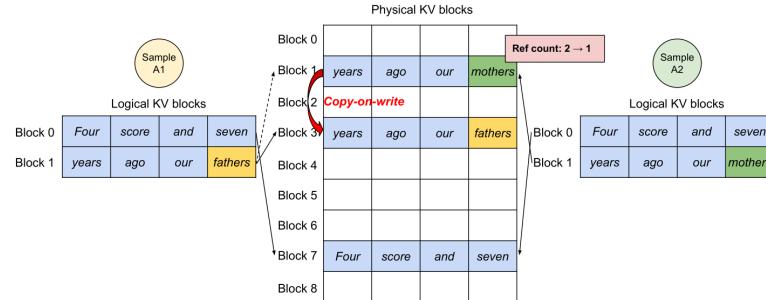
541

- Sharing the system prompt
- Sampling multiple responses per prompt (e.g., for program synthesis)

543

544 Solution: share prefixes, copy-on-write at the block level

545



546

547 Other vLLM optimizations:

- Kernel to fuse block read and attention (reduce kernel launch overhead)
- Use latest kernels (FlashAttention, FlashDecoding)
- Use CUDA graphs to avoid kernel launch overhead

551

552 Summary: use ideas from operating systems (paging) to make use of memory for dynamic workloads

553

554

```
555 if __name__ == "__main__":
556     main()
```