lecture_10.py

```
from sympy import symbols, oo
from execute_util import text, link, image
from lecture_util import article_link
from references import Reference, llama3, gqa, mla, longformer, sparse_transformer, mistral_7b

# Define symbols corresponding to the shape of the Transformer model
B, S, T, D, F, N, K, H, L, V = symbols("B S T D F N K H L V", positive=True)
c = symbols("c", positive=True)  # Just a constant that helps with taking limits
memory_bandwidth = symbols("memory_bandwidth", positive=True)

scaling_book_transformers = Reference(title="[Scaling book chapter on Transformers]", url="https://jax-ml.github.io/scaling-book/transformers/")
scaling_book_inference = Reference(title="[Scaling book chapter on Transformers]", url="https://jax-ml.github.io/scaling-book/inference/")

def main():
```

Inference: given a **fixed model**, generate responses given prompts

Understanding the inference workload

```
    landscape()
    review_transformer()
    review_of_arithmetic_intensity()
    arithmetic_intensity_of_inference()
    throughput_and_latency()
```

Taking shortcuts (lossy)

```
    reduce_kv_cache_size()
    alternatives_to_the_transformer()
    quantization()
    model_pruning()
```

Summary: reduce inference complexity without hurting accuracy

From scratch recipe:
1. Define faster model architecture
2. Train faster model

Distillation recipe:
1. Define faster model architecture
2. Initialize weights using original model (which has a different architecture)
3. Repair faster model (distillation)

Use shortcuts but double check (lossless)

```
    speculative_sampling()
```

Handling dynamic workloads

Batching over sequences in live traffic is tricky because:
1. Requests arrive at different times (waiting for batch is bad for early requests)
2. Sequences have shared prefixes (e.g., system prompts, generating multiple samples)
3. Sequences have different lengths (padding is inefficient)

```
    continuous_batching()
    paged_attention()
```

Summary

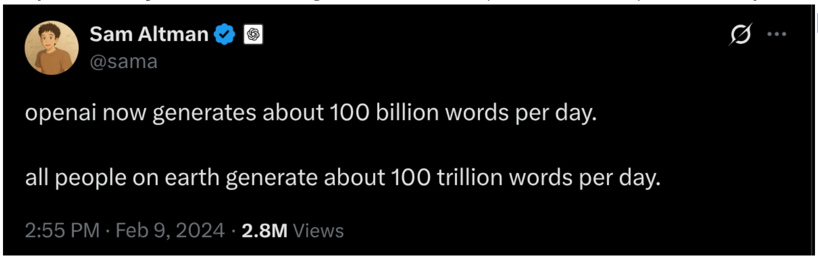
- Inference is important (actual use, evaluation, reinforcement learning)
- Different characteristics compared to training (memory-limited, dynamic)
- Techniques: new architectures, quantization, pruning/distillation, speculative decoding
- Ideas from systems (speculative execution, paging)
- New architectures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

```
def landscape():
```

Inference shows up in many places:

- Actual use (chatbots, code completion, batch data processing)
- Model evaluation (e.g., on instruction following)
- Test-time compute (thinking requires more inference)
- Training via reinforcement learning (sample generation, then score)

Why **efficiency** matters: training is one-time cost, inference is repeated many times



[tweet]

[tweet]

Metrics:

- Time-to-first-token (TTFT): how long user waits before any generation happens (matters for interactive applications)
- Latency (seconds/token): how fast tokens appear for a user (matters for interactive applications)
- Throughput (tokens/second): useful for batch processing applications

Key considerations in efficiency:

- Training (supervised): you see all tokens, can parallelize over sequence (matmul in Transformer)
- Inference: you have to generate sequentially, can't parallelize, so harder to fully utilize compute

Companies doing inference (a big deal for anyone who has a product or platform):

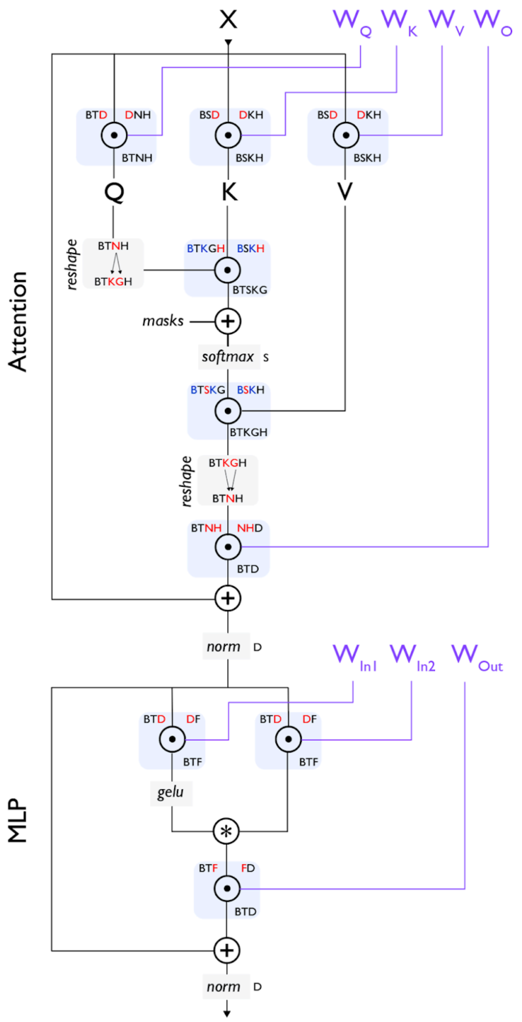
- Providers serving closed models (OpenAI, Anthropic, Google, etc.)
- Providers serving open-weight models (Together, Fireworks, DeepInfra, etc.)

Open-source packages:

- vLLM (Berkeley) [talk]
- Tensor-RT (NVIDIA) [article]
- TGI (Hugging Face) [article]

```
def review_transformer():
```

[Scaling book chapter on Transformers]



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

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

FLOPs for a feedforward pass: 6 * (B*T) * (num_params + O(T))

```
def review_of_arithmetic_intensity():
```

Setup: multiply X (B x D) and W (D x F) matrix

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

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

```
flops = 0
bytes_transferred = 0
```

Steps:

1. Read X (B x D) from HBM

```
bytes_transferred += 2*B*D
```

2. Read W (D x F) from HBM

```
bytes_transferred += 2*D*F
```

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

```
flops += 2*B*D*F
```

4. Write Y (B x F) to HBM

```
bytes_transferred += 2*B*F
```

Let's take stock of the accounting results.

```
assert flops == 2*B*D*F
assert bytes_transferred == 2*B*D + 2*D*F + 2*B*F
```

Recall that **arithmetic intensity** is how much compute we do per byte transferred (want to be high).

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

Assuming B is much less than D and F, then we can simplify:

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

Accelerator intensity of H100:

```
flops_per_second = 989e12
memory_bandwidth = 3.35e12
accelerator_intensity = flops_per_second / memory_bandwidth  # @inspect accelerator_intensity
assert round(accelerator_intensity) == 295
```

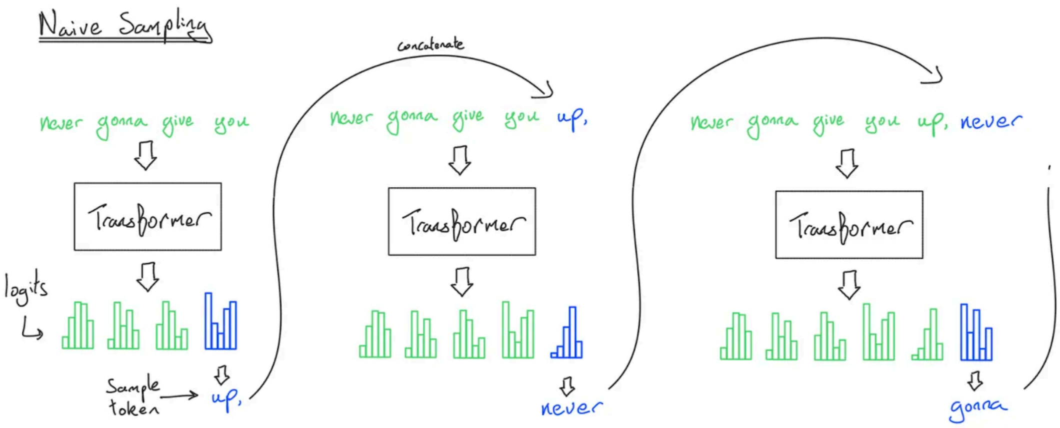
If computation intensity > accelerator intensity, **compute-limited** (good)
If computation intensity < accelerator intensity, **memory-limited** (bad)
Conclusion: compute-limited iff B > 295

Extreme case (B = 1, corresponding to matrix-vector product):

- Arithmetic intensity: 1
- Memory-limited (read D x F matrix, perform only 2*DF* FLOPs)
- This is basically what happens with generation...

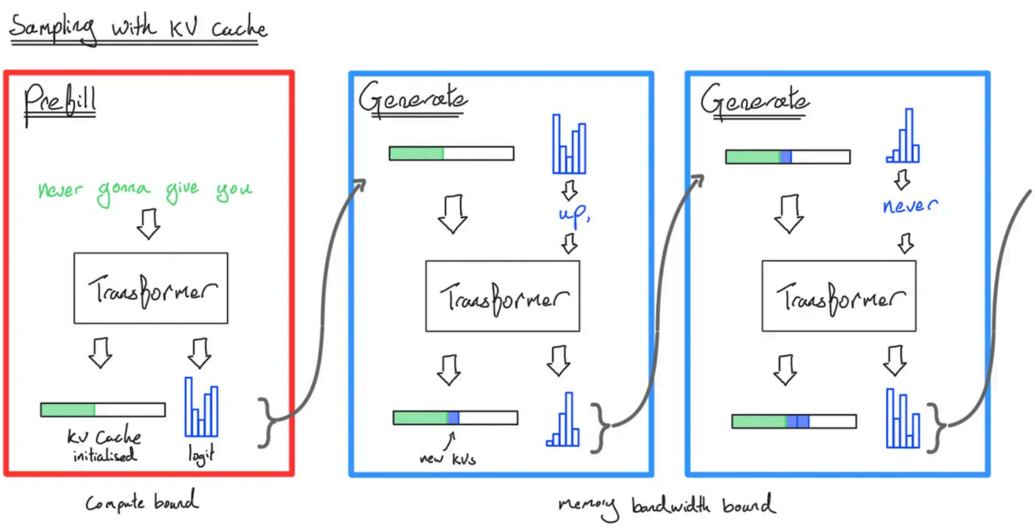
```
def arithmetic_intensity_of_inference():
```

[Scaling book chapter on Transformers]



Naive inference: to generate each token, feed history into Transformer
Complexity: generating T tokens requires O(T^3) FLOPs (one feedforward pass is O(T^2))

Observation: a lot of the work can be shared across prefixes
Solution: store **KV cache** in HBM



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

Two stages of inference:
1. **Prefill**: given a prompt, encode into vectors (parallelizable like in training)
2. **Generation**: generate new response tokens (sequential)

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).

MLP layers (only looking at the matrix multiplications)

```
flops = 0
bytes_transferred = 0
```

Steps:

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

```
bytes_transferred += 2*B*T*D
```

2. Read Wup (D x F), Wgate (D x F), Wdown (F x D) from HBM

```
bytes_transferred += 3 * 2*D*F
```

3. Compute U = X (B x T x D) @ Wup (D x F)

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

4. Write U (B x T x F) to HBM

```
bytes_transferred += 2*B*T*F
```

5. Compute G = X (B x T x F) @ Wgate (D x F)

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

6. Write G (B x T x F) to HBM

```
bytes_transferred += 2*B*T*F
```

7. Compute Y = GeLU(G)*U (B x T x F) @ Wdown (F x D)

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

8. Write Y (B x T x D) to HBM

```
bytes_transferred += 2*B*T*D
```

Let's take stock of the accounting results.

```
assert flops == 6*B*T*D*F
assert bytes_transferred == 4*B*T*D + 4*B*T*F + 6*D*F
intensity = (flops / bytes_transferred).simplify()  # @inspect intensity
```

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

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

For the two stages:

1. Prefill: easy to make compute-limited (good) by making B T large enough
2. Generation:

- Generating one token at a time (T = 1)
- B is number of concurrent requests, hard to make large enough!

Attention layers (focusing on the matrix multiplications with FlashAttention)

```
flops = 0
bytes_transferred = 0
```

Steps:

1. Read Q (B x T x D), K (B x S x D), V (B x S x D) from HBM

```
bytes_transferred += 2*B*T*D + 2*B*S*D + 2*B*S*D
```

2. Compute A = Q (B x T x D) @ K (B x S x D)

```
flops += 2*B*S*T*D
```

3. Compute Y = softmax(A) (B x S x T x K x G) @ V (B x S x K x H)

```
flops += 2*B*S*T*D
```

4. Write Y (B x T x D) to HBM

```
bytes_transferred += 2*B*T*D
```

```
assert flops == 4*B*S*T*D
assert bytes_transferred == 4*B*S*D + 4*B*T*D
intensity = (flops / bytes_transferred).simplify()  # @inspect intensity
assert intensity == S*T / (S + T)
```

For the two stages:

1. Prefill: T = S

```
prefill_intensity = intensity.subs(T, S).simplify()  # @inspect prefill_intensity
assert prefill_intensity == S/2  # Good!
```

2. Generation: T = 1

```
generate_intensity = intensity.subs(T, 1).simplify()  # @inspect generate_intensity
assert generate_intensity < 1  # Bad!
```

Unlike MLPs, no dependence on B, so batching doesn't help!

Why?

- In MLP layers, every sequence hits the same MLP weights (Wup, Wgate, Wdown don't depend on B)
- In attention layers, every sequence has its own vectors KV cache (Q, K, V all depend on B)

Summary

- Prefill is compute-limited, generation is memory-limited
- MLP intensity is B (requires concurrent requests), attention intensity is 1 (impossible to improve)

```
def compute_transformer_stats(config):  # @inspect config
    """Return symbols corresponding to various statistics of a Transformer."""
```

The memory, throughput, and latency depends on the shape of the Transformer.

Compute the number of parameters in the Transformer:

```
    num_params = 2*V*D + D*F*3*L + (2*D*N*H + 2*D*K*H)*L
```

To store parameters, just use bf16 (training requires fp32)

```
    parameter_size = num_params * 2  # 2 for bf16
```

We also don't need gradients and optimizer states since we're not training.

But we do have to store the KV cache (which are some of the activations) for each sequence (of length S):

How much we have to store per sequence:

```
    kv_cache_size = S * (K*H) * L * 2 * 2  # 2 for key + value, 2 for bf16
```

Total memory usage:

```
    memory = B * kv_cache_size + parameter_size
```

Latency is determined by memory IO (read all parameters and KV cache for each step)

```
    latency = memory / memory_bandwidth
```

Throughput is the inverse of latency, but we're generating B tokens in parallel

```
    throughput = B / latency

    # Substitute
    num_params = num_params.subs(config).simplify()  # @inspect num_params
    memory = memory.subs(config).simplify()  # @inspect memory
    latency = latency.subs(config).simplify()  # @inspect latency
    throughput = throughput.subs(config).simplify()  # @inspect throughput

    return num_params, memory, latency, throughput

def llama2_13b_config(args={}):
    return {S: 1024, D: 5120, F: 13824, N: 40, K: 40, H: 128, L: 40, V: 32000, memory_bandwidth: 3.35e12, **args}

def throughput_and_latency():
```

So we have shown that inference is memory-limited.

Let us now compute the theoretical maximum latency and throughput of a single request.

Assumption: can overlap compute and communication perfectly and ignore various types of overhead.

Instantiate latency and throughput for Llama 2 13B on an H100:

```
    config = llama2_13b_config()
    num_params, memory, latency, throughput = compute_transformer_stats(config)
```

If we use a batch size of 1:

```
    bs1_memory = memory.subs(B, 1).simplify()   # @inspect bs1_memory
    bs1_latency = latency.subs(B, 1).simplify()   # @inspect bs1_latency
    bs1_throughput = throughput.subs(B, 1).simplify()   # @inspect bs1_throughput
```

If we use a batch size of 64 (worse latency, better throughput):

```
    bs64_memory = memory.subs(B, 64).simplify()   # @inspect bs64_memory
    bs64_latency = latency.subs(B, 64).simplify()   # @inspect bs64_latency
    bs64_throughput = throughput.subs(B, 64).simplify()   # @inspect bs64_throughput
```

If we use a batch size of 256:

```
    bs256_memory = memory.subs(B, 256).simplify()   # @inspect bs256_memory
```

```
bs256_latency = latency.subs(B, 256).simplify()   # @inspect bs256_latency
bs256_throughput = throughput.subs(B, 256).simplify()   # @inspect bs256_throughput
```

Doesn't fit into memory, but throughput gains are diminishing too...

Tradeoff between latency and throughput:

1. Smaller batch sizes yields better latency but worse throughput
2. Larger batch sizes yields better throughput but worse latency

Easy parallelism: if you launch M copies of the model, latency is the same, throughput increases by M!
Harder parallelism: shard the model and the KV cache [Scaling book chapter on Transformers]

Note: time-to-first-token (TTFT) is essentially a function of prefill
Use smaller batch sizes during prefill for faster TTFT
Use larger batch sizes during generation to improve throughput

```
def reduce_kv_cache_size():
```

Recall that memory is the bottleneck for inference.
So let's try to reduce the size of the KV cache
...but make sure we don't lose too much accuracy.

Grouped-query attention (GQA)

[Ainslie+ 2023]

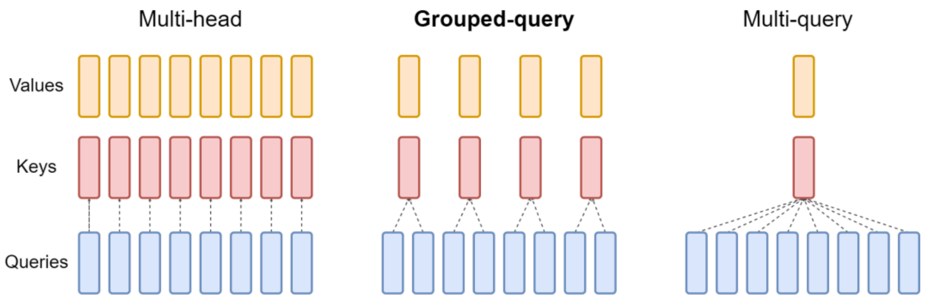
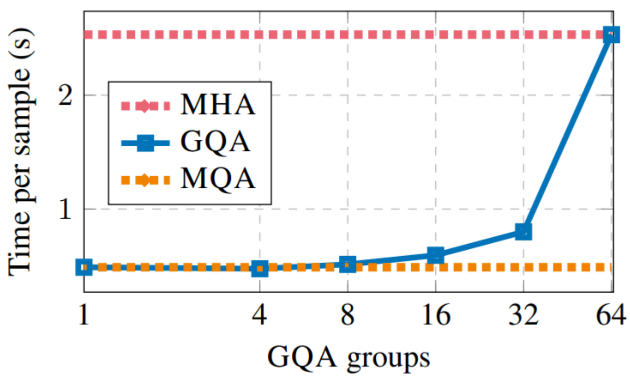


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between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

Idea: N query heads, but only K key and value heads, each interacting with N/K query heads
Multi-headed attention (MHA): K=N
Multi-query attention (MQA): K=1
Group-query attention (GQA): K is somewhere in between

Latency/throughput improvements:

[Ainslie+ 2023]



Reduce the KV cache by a factor of N/K

```
config = llama2_13b_config({K: 40, B: 64})  # Original Llama 2 13B
k40_num_params, k40_memory, k40_latency, k40_throughput = compute_transformer_stats(config)  # @inspect k40_memory, @inspect k40_latency, @inspect k40_throughput

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

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

This also means we can use a larger batch size:

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

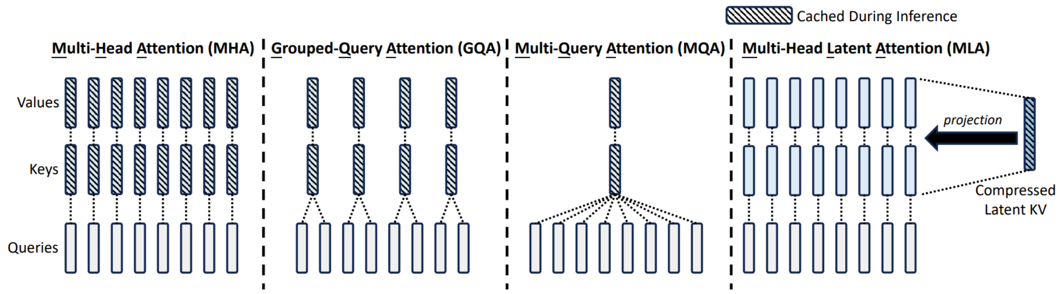
Worse latency, but better throughput (and it fits in memory now!).

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

Model	T_{infer}	Average	CNN	arXiv	PubMed	MediaSum	MultiNews	WMT	TriviaQA
	s		R_1	R_1	R_1	R_1	R_1	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

Multi-head latent attention (MLA)

[DeepSeek-AI+ 2024]



Key idea: project down each key and value vector from N*H dimensions to C dimensions
DeepSeek v2: reduce N*H = 16384 to C = 512
Wrinkle: MLA is not compatible with RoPE, so need to add additional 64 dimensions for RoPE, so 512 + 64 = 576 total dimensions
Latency/throughput improvements follow similarly from the KV cache reduction as argued earlier

Let's now check the accuracy.
First, MHA is better than GQA (though more expensive) [Table 8] [DeepSeek-AI+ 2024]

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

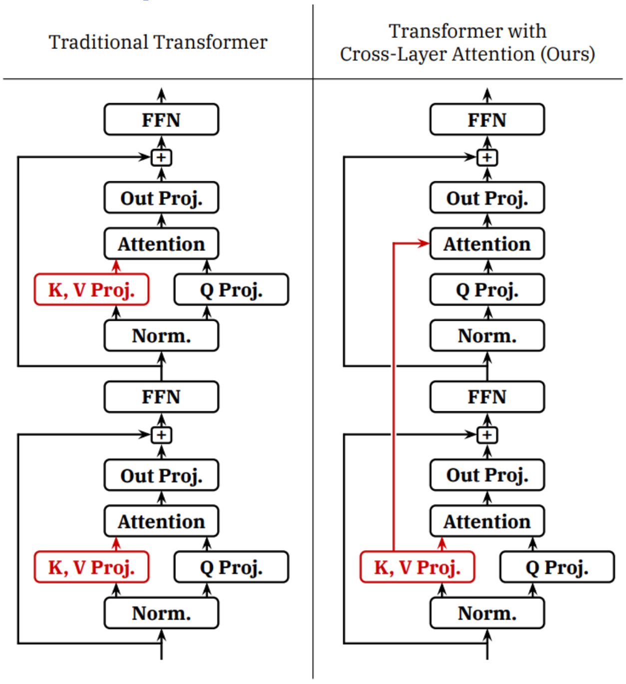
Table 8 | Comparison among 7B dense models with MHA, GQA, and MQA, respectively. MHA demonstrates significant advantages over GQA and MQA on hard benchmarks.

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

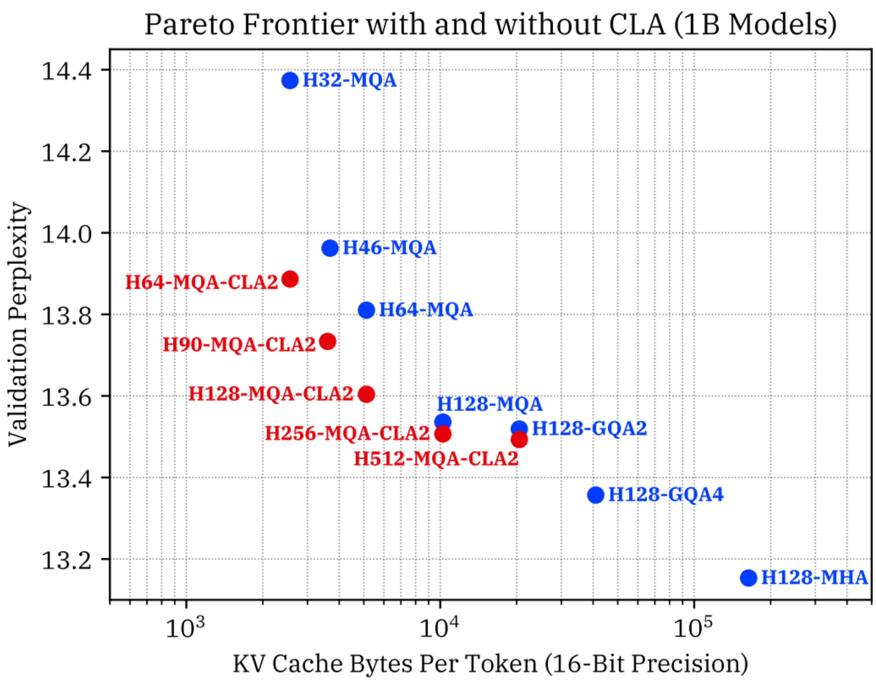
Benchmark (Metric)	# Shots	Small MoE w/ MHA	Small MoE w/ MLA	Large MoE w/ MHA	Large MoE w/ MLA
# Activated Params	-	2.5B	2.4B	25.0B	21.5B
# Total Params	-	15.8B	15.7B	250.8B	247.4B
KV Cache per Token (# Element)	-	110.6K	15.6K	860.2K	34.6K
BBH (EM)	3-shot	37.9	**39.0**	46.6	**50.7**
MMLU (Acc.)	5-shot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

Cross-layer attention (CLA)

[Brandon+ 2024]

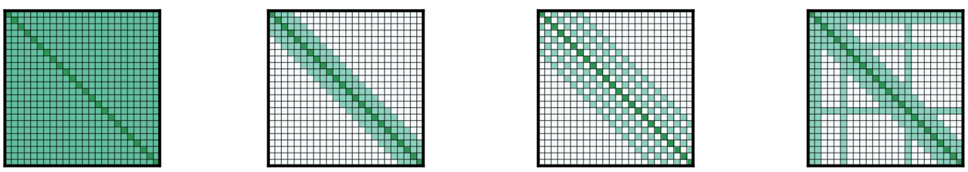


Idea: share KVs across **layers** (just as GQA shares KVs across heads)
Empirically improves the pareto frontier of accuracy and KV cache size (latency and throughput)



Local attention

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

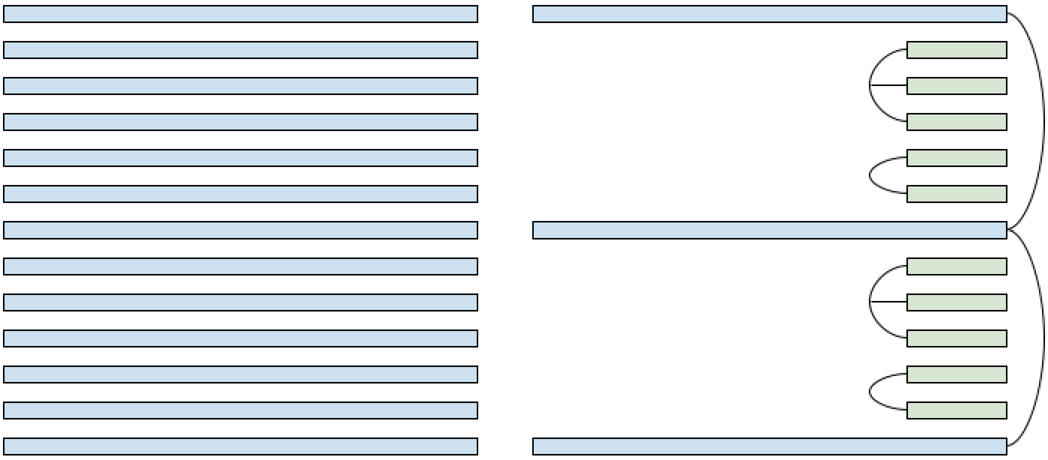


(a) Full n^2 attention (b) Sliding window attention (c) Dilated sliding window (d) Global+sliding window

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

Effective context scales linearly with the number of layers
KV cache is independent of sequence length!

Problem: this can still hurt accuracy
Solution: interleave local attention with global attention (hybrid layers)
Example: character.ai uses 1 global layer every 6 layers (in addition to CLA) [article]



Summary:

- Goal: reduce the KV cache size (since inference is memory-limited) without hurting accuracy
- Lower-dimensional KV cache (GQA, MLA, shared KV cache)
- Local attention on some of the layers

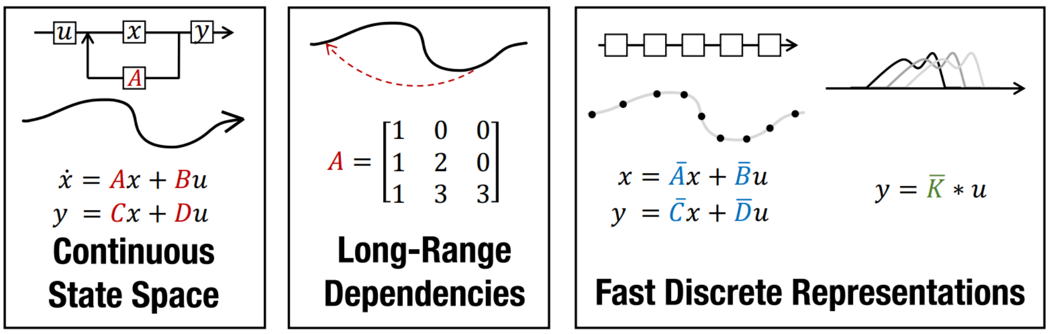
```
def alternatives_to_the_transformer():
```

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

State-space models

[presentation from CS229S]

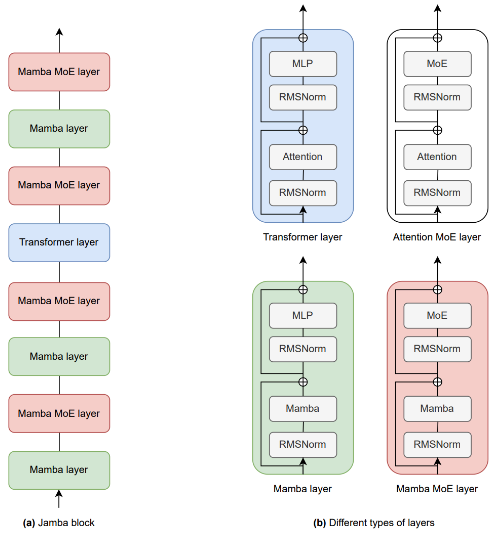
- Idea: from signal processing to model long-context sequences in a sub-quadratic time
- S4: based on classic state space models, good at synthetic long-context tasks [Gu+ 2021]



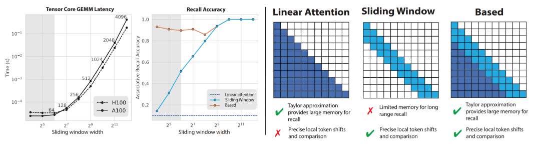
- Weaknesses: bad at solving associative recall tasks important for language (where Transformers do well)

$$\text{A 4 B 3 C 6 } \underbrace{\text{F 1}}_{\textbf{Key-Value}} \text{ E 2} \rightarrow \text{A ? C ? } \underbrace{\text{F ?}}_{\textbf{Query}} \text{ E ? B ?}$$

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



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



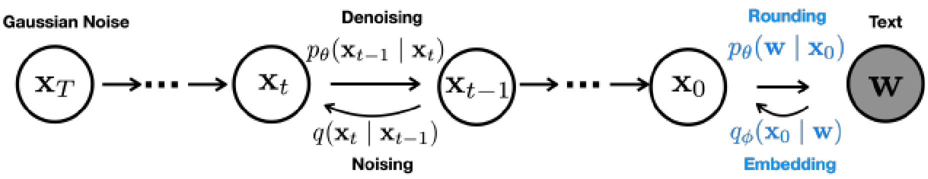
- MiniMax-01: use linear attention + full attention (456B parameter MoE) [MiniMax+ 2025]

Takeaways:

- 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

Diffusion models

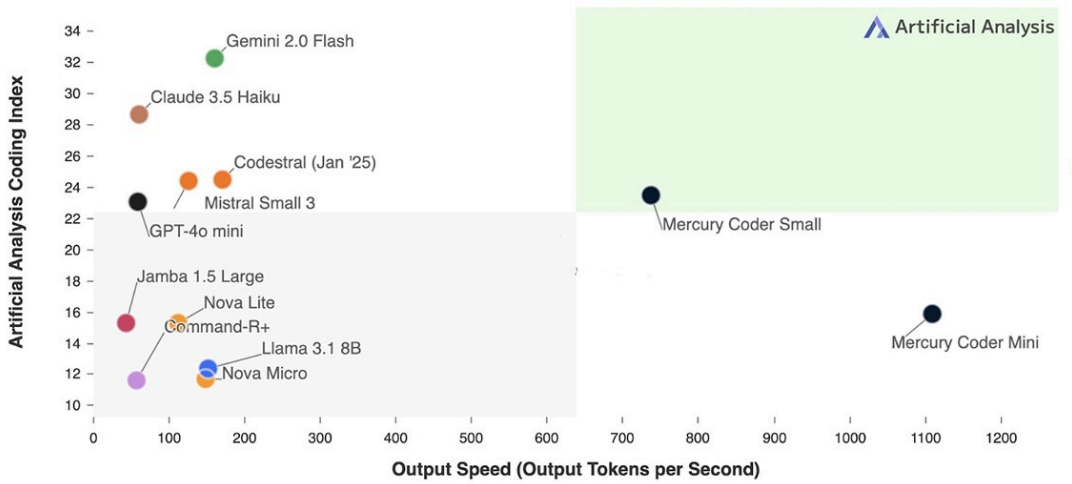
- Popular for image generation, but harder to get working for text generation [Li+ 2022]



- 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]

[demo video]

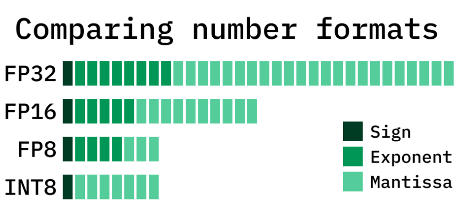
Much faster on coding benchmarks:



Overall, significant gains in inference to be made with more radical architecture changes!

```
def quantization():
```

Key idea: reduce the precision of numbers
Less memory means higher latency/throughput (since inference is memory-limited).
Of course we have to worry about accuracy...



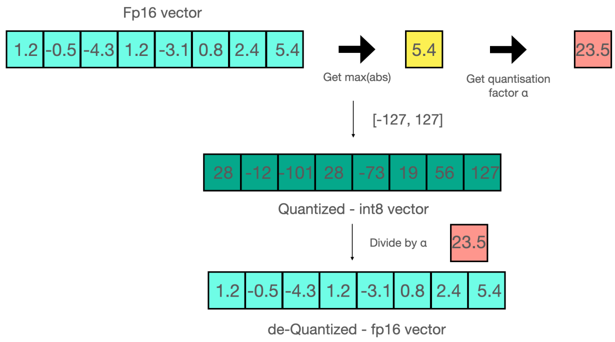
[article]

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

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

LLM.int8()

[Dettmers+ 2022][article]
Standard quantization (scale by max of absolute values):



Problem: outliers (which appear in larger networks) screw everything up
Solution: extract outliers and process them in fp16

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

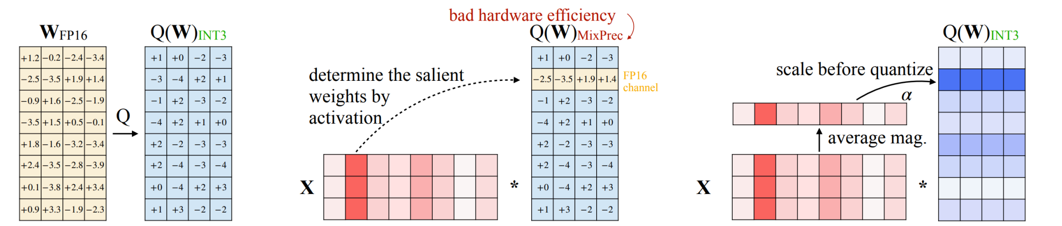
benchmarks	-	-	-	-	difference - value
name	metric	value - int8	value - bf16	std err - bf16	-
hellaswag	acc_norm	0.7274	0.7303	0.0044	0.0029
hellaswag	acc	0.5563	0.5584	0.005	0.0021
piqa	acc	0.7835	0.7884	0.0095	0.0049
piqa	acc_norm	0.7922	0.7911	0.0095	0.0011
lambada	ppl	3.9191	3.931	0.0846	0.0119
lambada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

Activation-aware quantization

[Lin+ 2023]

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

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



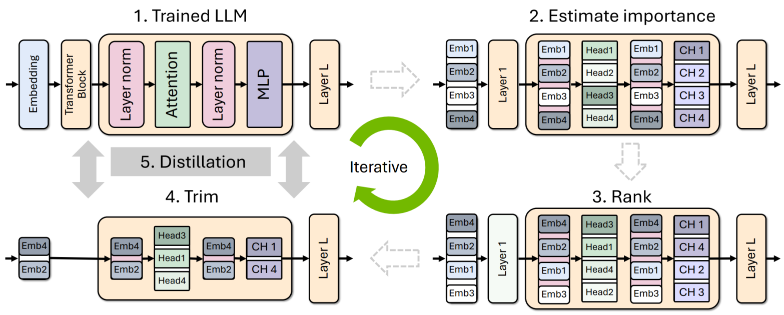
(a) RTN quantization (**PPL** 43.2) (b) Keep 1% salient weights in FP16 (**PPL** 13.0) (c) Scale the weights before quantization (**PPL** 13.0)

```
def model_pruning():
```

Key idea: just rip out parts of an expensive model to make it cheaper

...and then fix it up.

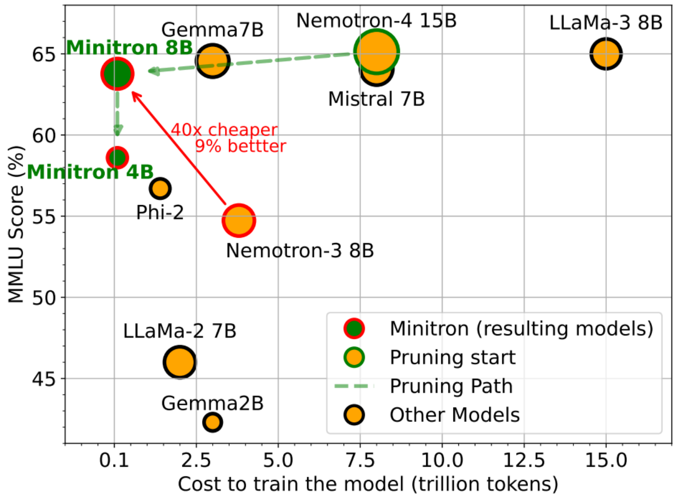
Paper from NVIDIA [Muralidharan+ 2024]



Algorithm:

1. Identify important {layer, head, hidden dimension} on a small calibration dataset (1024 samples)
2. Remove unimportant layers to get a smaller model
3. Distill the original model into pruned model

Results:




```
def speculative_sampling():
```

Recall the two stages of inference:

- Prefill: given a sequence, encode tokens in parallel (compute-limited) [note: also gives you probabilities]
- Generation: generate one token at a time (memory-limited)

In other words, checking is faster than generation.

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

[Speculative sampling video]
[article]

Algorithm 2 Speculative Sampling (SpS) with Auto-Regressive Target and Draft Models

Given lookahead K and minimum target sequence length T.
Given auto-regressive target model $q(.|.)$, and auto-regressive draft model $p(.|.)$, initial prompt sequence $x_0, \ldots, x_t$.
Initialise $n \leftarrow t$.
while $n < T$ **do**
  for $t = 1 : K$ **do**
    Sample draft auto-regressively $\tilde{x}_t \sim p(x|, x_1, \ldots, x_n, \tilde{x}_1, \ldots, \tilde{x}_{t-1})$
  end for
  In parallel, compute $K + 1$ sets of logits from drafts $\tilde{x}_1, \ldots, \tilde{x}_K$:

$$q(x|, x_1, \ldots, x_n),\ q(x|, x_1, \ldots, x_n, \tilde{x}_1),\ \ldots,\ q(x|, x_1, \ldots, x_n, \tilde{x}_1, \ldots, \tilde{x}_K)$$

  for $t = 1 : K$ **do**
    Sample $r \sim U[0, 1]$ from a uniform distribution.
    if $r < \min\left(1, \frac{q(x|x_1,\ldots,x_{n+t-1})}{p(x|x_1,\ldots,x_{n+t-1})}\right)$, **then**
      Set $x_{n+t} \leftarrow \tilde{x}_t$ and $n \leftarrow n + 1$.
    else
      sample $x_{n+t} \sim (q(x|x_1, \ldots, x_{n+t-1}) - p(x|x_1, \ldots, x_{n+t-1}))_+$ and exit for loop.
    end if
  end for
  If all tokens $x_{n+1}, \ldots, x_{n+K}$ are accepted, sample extra token $x_{n+K+1} \sim q(x|, x_1, \ldots, x_n, x_{n+K})$ and set $n \leftarrow n + 1$.
end while

This is modified rejection sampling with proposal p and target q
Modification: always generate at least one candidate (rejection sampling will keep looping)
Key property: guaranteed to be an **exact sample** from the target model!

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

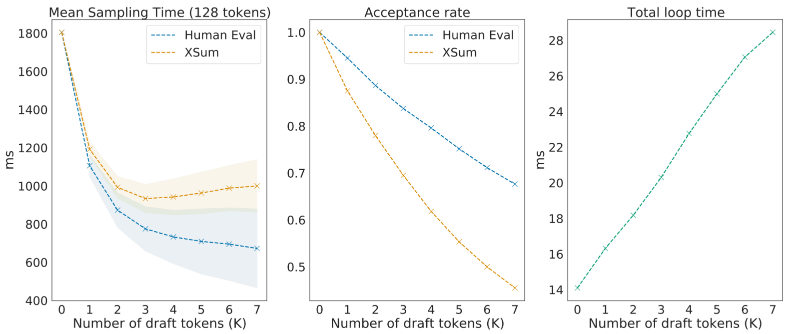
- Target model probabilities: [q(A), q(B)]
- Draft model probabilities: [p(A), p(B)]
- Assume p(A) > q(A) [draft model oversamples A].
- Therefore p(B) < q(B) [draft model undersamples B].
- Residual probabilities max(q-p, 0): [0, 1]

Compute the probabilities of speculatively sampling a token:

- P[sampling A] = p(A) * (q(A) / p(A)) + p(B) * 1 * 0 = q(A)
- P[sampling B] = p(B) * 1 + p(A) * (1 - q(A) / p(A)) * 1 = q(B)

Table 1 | **Chinchilla performance and speed on XSum and HumanEval with naive and speculative sampling at batch size 1 and $K = 4$.** XSum was executed with nucleus parameter $p = 0.8$, and HumanEval with $p = 0.95$ and temperature 0.8.

Sampling Method	Benchmark	Result	Mean Token Time	Speed Up
ArS (Nucleus)	XSum (ROUGE-2)	0.112	14.1ms/Token	1×
SpS (Nucleus)		0.114	7.52ms/Token	1.92×
ArS (Greedy)	XSum (ROUGE-2)	0.157	14.1ms/Token	1×
SpS (Greedy)		0.156	7.00ms/Token	2.01×
ArS (Nucleus)	HumanEval (100 Shot)	45.1%	14.1ms/Token	1×
SpS (Nucleus)		47.0%	5.73ms/Token	2.46×



In practice:

- Target model has 70B parameters, draft model has 8B parameters
- Target model has 8B parameters, draft model has 1B parameters
- Try to make draft model as close to target (distillation)

Extensions to improve the draft model:

- Medusa: draft model generates multiple tokens in parallel [Cai+ 2024]
- EAGLE: draft model takes high-level features from target model [Li+ 2024]

Summary:

- Exact sampling from target model (thanks to math)!
- Exploits asymmetry between checking and generation
- Lots of room for innovation on the draft model (involves training)

```
def continuous_batching():
```

Orca: A Distributed Serving System for Transformer-Based Generative Models[talk]

Problem:

- Training: get a dense block of tokens (batch size x sequence length)
- Inference: requests arrive and finish at different times, so you have a ragged array

Solution: iteration-level scheduling

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

Problem:

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

Solution: selective batching

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

```
def paged_attention():
```

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

Previous status quo:

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

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)!

Solution: PagedAttention (remember operating systems)

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

Two requests share the KV caches:

In general, multiples types of sharing KV caches across sequences:

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

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

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

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

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