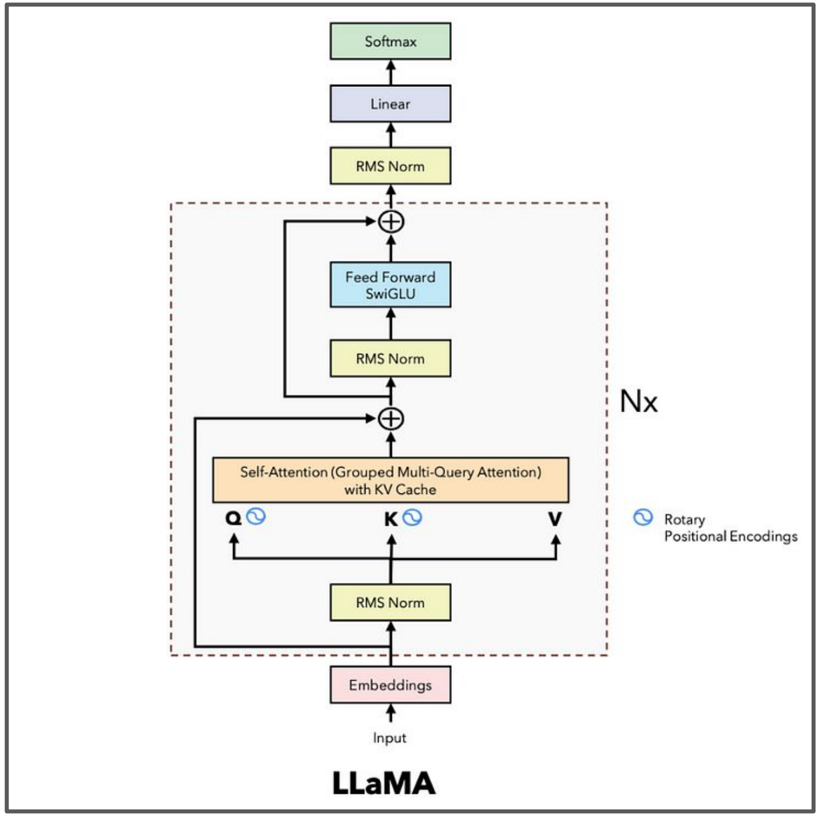
GenAI Talk: Episode 1

Assisted Generation: Speeding up LLM inference

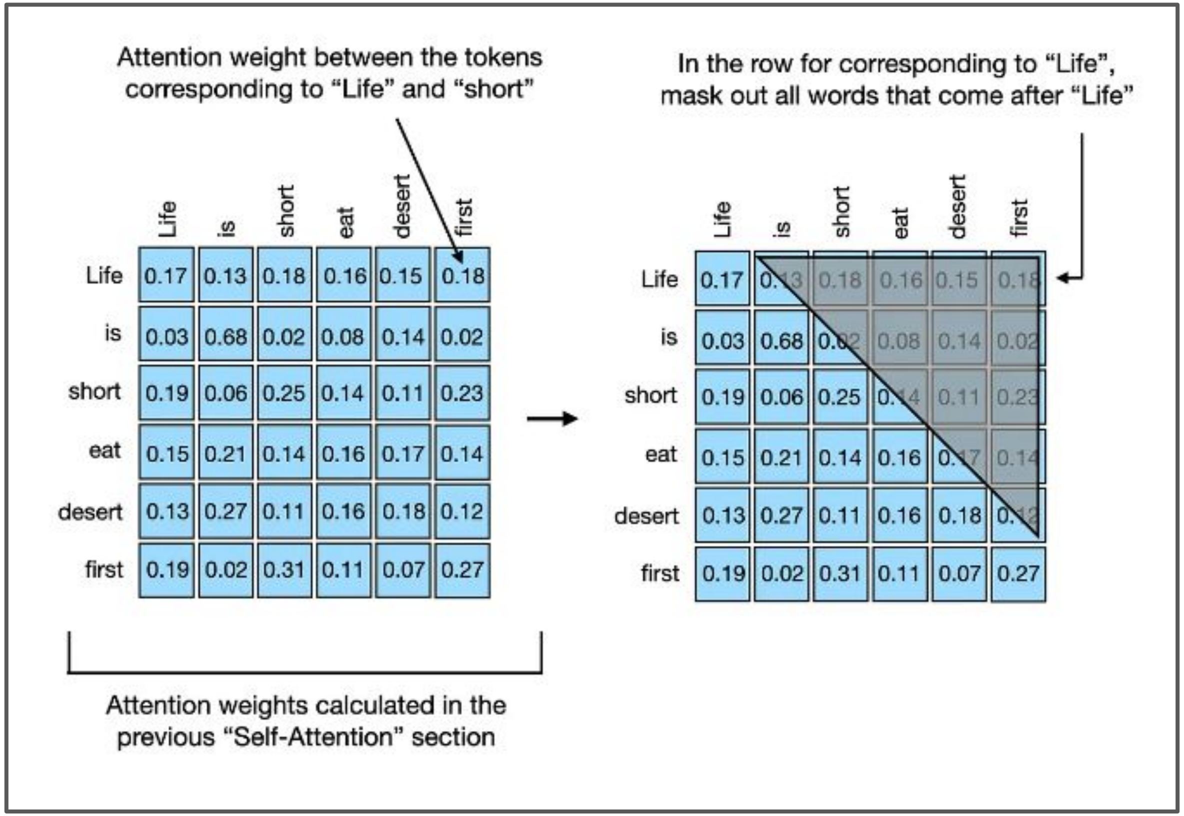
Sudhanshu Mishra
Shubhanshu Mishra

Large Language Models

We will mostly be talking about Decoder only Models like LLama, Mistral, GPT …


Softmax
Linear
RMS Norm
Feed Forward SwiGLU
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input
LLaMA

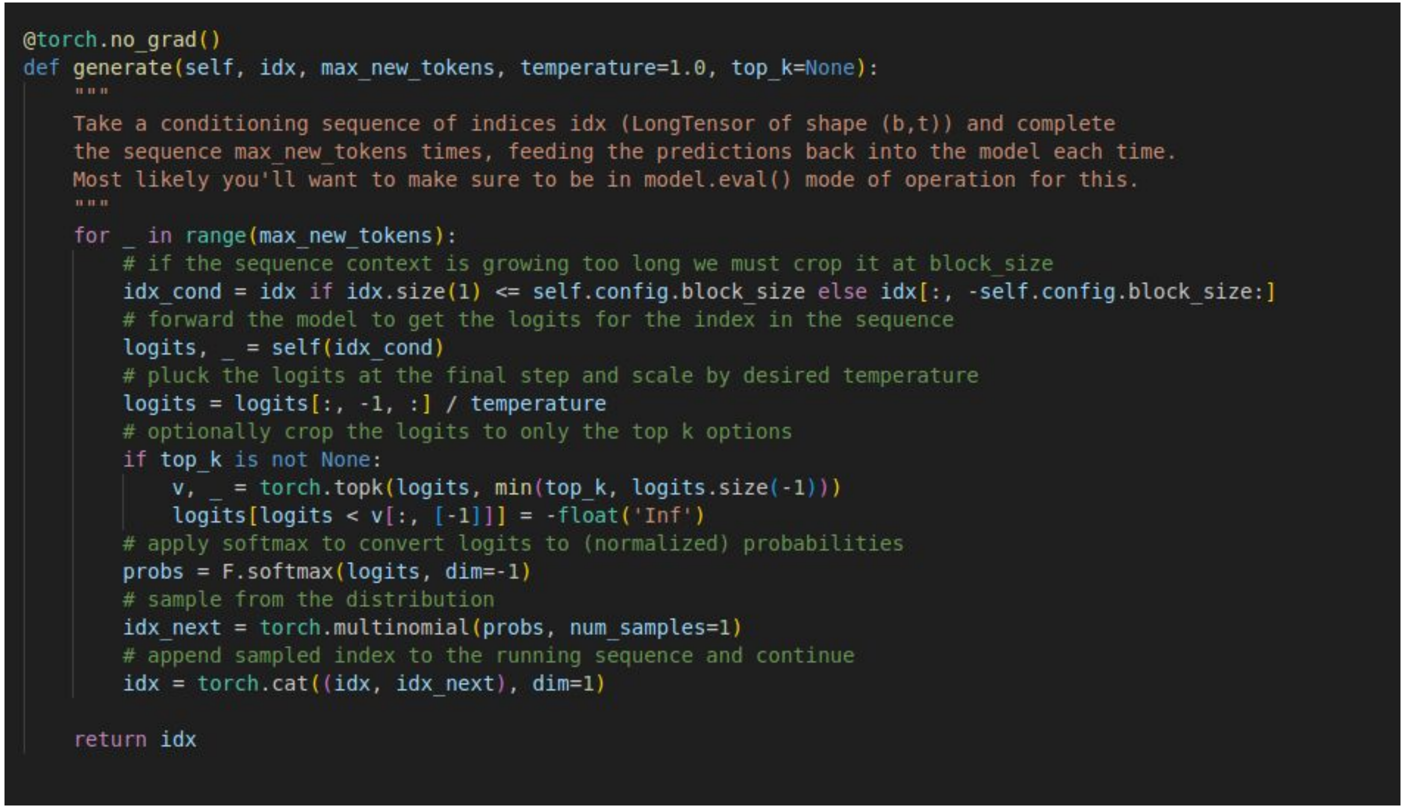
Causal Self Attention


Attention weight between the tokens corresponding to "Life" and "short"
In the row for corresponding to "Life", mask out all words that come after "Life"
Attention weights calculated in the previous "Self-Attention" section

	Life	is	short	eat	desert	first
Life	0.17	0.13	0.18	0.16	0.15	0.18
is	0.03	0.68	0.02	0.08	0.14	0.02
short	0.19	0.06	0.25	0.14	0.11	0.23
eat	0.15	0.21	0.14	0.16	0.17	0.14
desert	0.13	0.27	0.11	0.16	0.18	0.12
first	0.19	0.02	0.31	0.11	0.07	0.27

	Life	is	short	eat	desert	first
Life	0.17	0.13	0.18	0.16	0.15	0.18
is	0.03	0.68	0.02	0.08	0.14	0.02
short	0.19	0.06	0.25	0.14	0.11	0.23
eat	0.15	0.21	0.14	0.16	0.17	0.14
desert	0.13	0.27	0.11	0.16	0.18	0.12
first	0.19	0.02	0.31	0.11	0.07	0.27

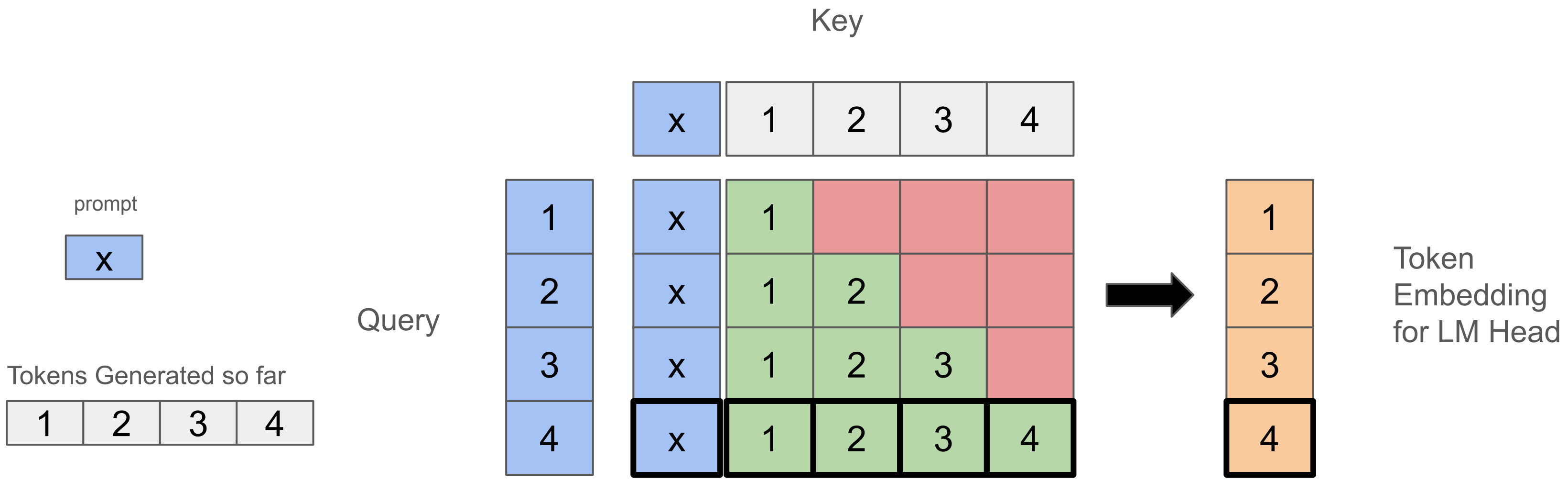
Nano GPT (Andrej Karpathy)



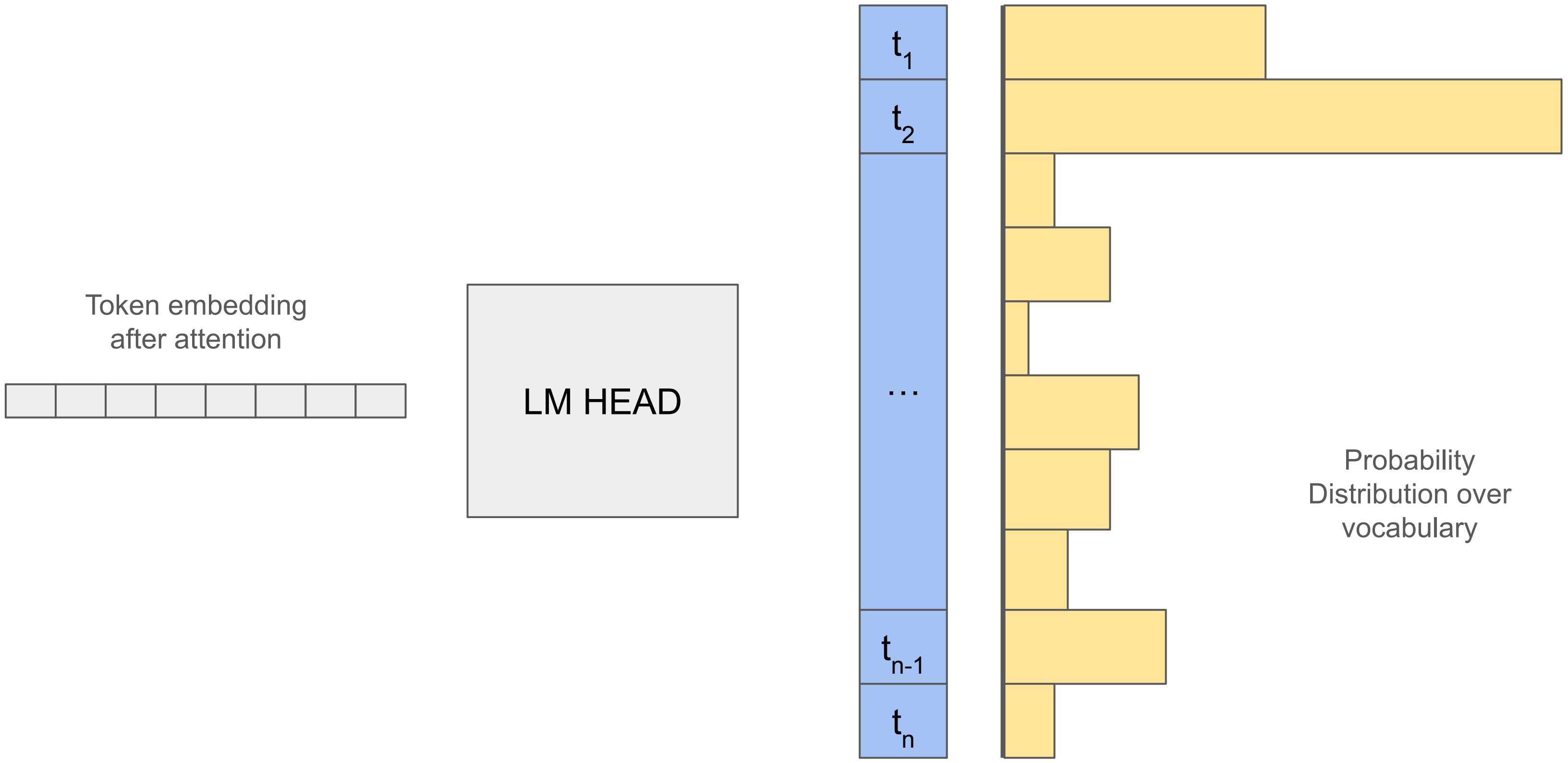
```python
@torch.no_grad()
def generate(self, idx, max_new_tokens, temperature=1.0, top_k=None):
    """
    Take a conditioning sequence of indices idx (LongTensor of shape (b,t)) and complete
    the sequence max_new_tokens times, feeding the predictions back into the model each time.
    Most likely you'll want to make sure to be in model.eval() mode of operation for this.
    """
    for _ in range(max_new_tokens):
        # if the sequence context is growing too long we must crop it at block_size
        idx_cond = idx if idx.size(1) <= self.config.block_size else idx[:, -self.config.block_size:]
        # forward the model to get the logits for the index in the sequence
        logits, _ = self(idx_cond)
        # pluck the logits at the final step and scale by desired temperature
        logits = logits[:, -1, :] / temperature
        # optionally crop the logits to only the top k options
        if top_k is not None:
            v, _ = torch.topk(logits, min(top_k, logits.size(-1)))
            logits[logits < v[:, [-1]]] = -float('Inf')
        # apply softmax to convert logits to (normalized) probabilities
        probs = F.softmax(logits, dim=-1)
        # sample from the distribution
        idx_next = torch.multinomial(probs, num_samples=1)
        # append sampled index to the running sequence and continue
        idx = torch.cat((idx, idx_next), dim=1)

    return idx
```

Causal Attention in Action

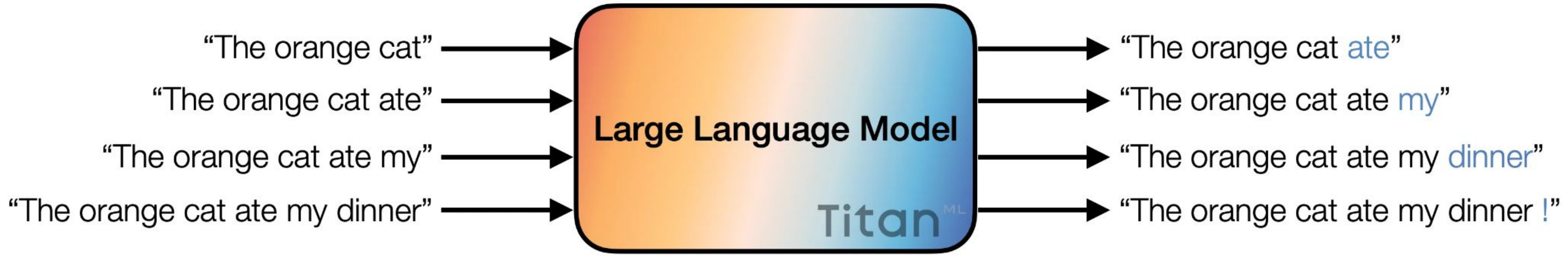


Final Step in the generation Process


Token embedding after attention
LM HEAD
t_1
t_2
…
t_{n-1}
t_n
Probability Distribution over vocabulary

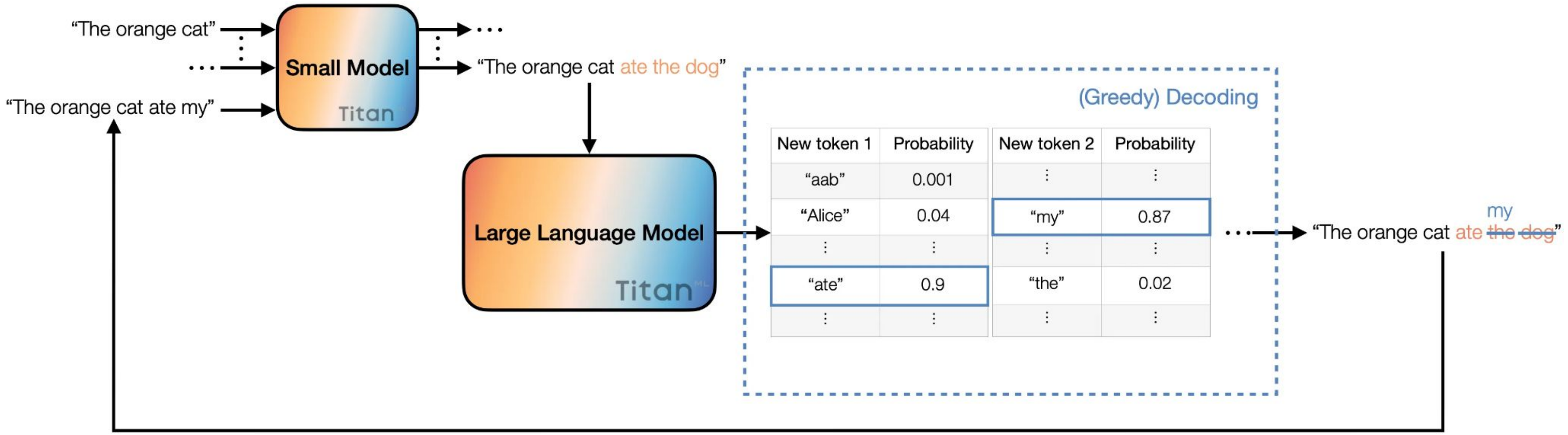
Major Challenge of LLM Inference

At inference time, the model generates one token at a time, this doesn't allow us to use the multi token computation speedups given by GPUs.



Source: [In the Fast Lane! Speculative Decoding — 10x Larger Model, No Extra Cost | by TitanML | Medium]

Speculative Decoding



New token 1	Probability	New token 2	Probability
“aab”	0.001	⋮	⋮
“Alice”	0.04	“my”	0.87
⋮	⋮	⋮	⋮
“ate”	0.9	“the”	0.02
⋮	⋮	⋮	⋮

Source: In the Fast Lane! Speculative Decoding — 10x Larger Model, No Extra Cost | by TitanML | Medium

Speculative Decoding

1. Main Concept behind Speculative Decoding is having candidate tokens for future positions and verifying them in a single forward pass parallely.

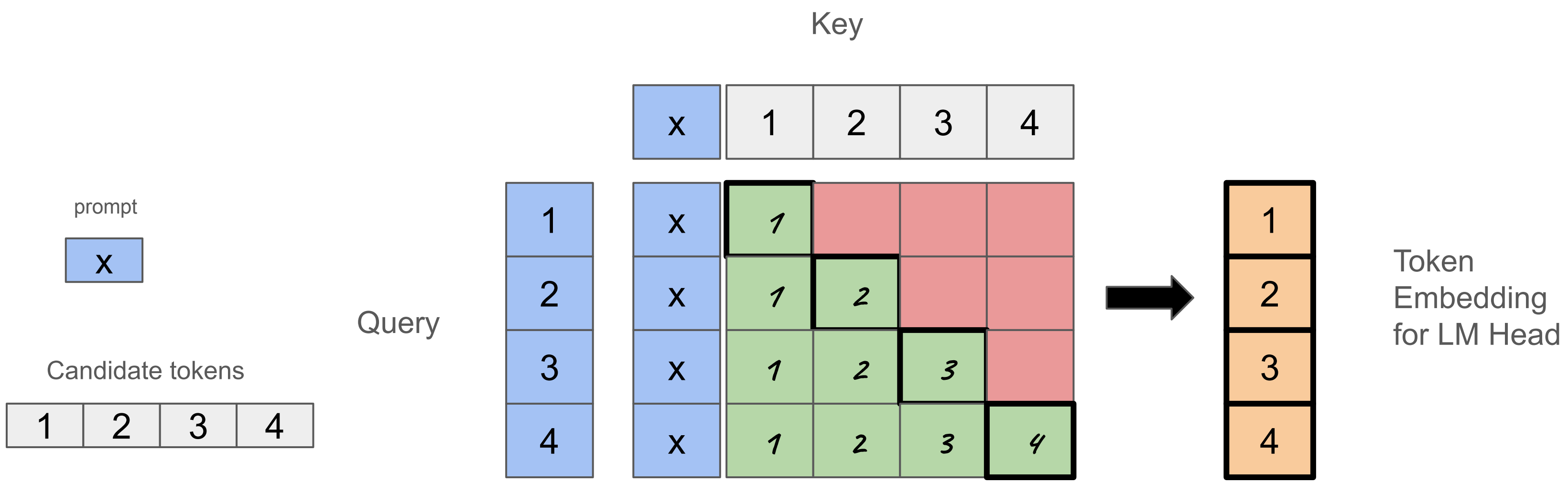
2. Trading off Compute for more tokens in a single step.

3. Usually requires a Draft Model that provides the candidate tokens quickly

4. Common Rule of thumb is to use a Draft Model that is at least 3x faster than the target model

5. For best acceptance of the candidate and speedup, the draft model should be fine tuned aligning with the target model

Verification in Action



Why is LLM generation inefficient? (From the Medusa Paper)

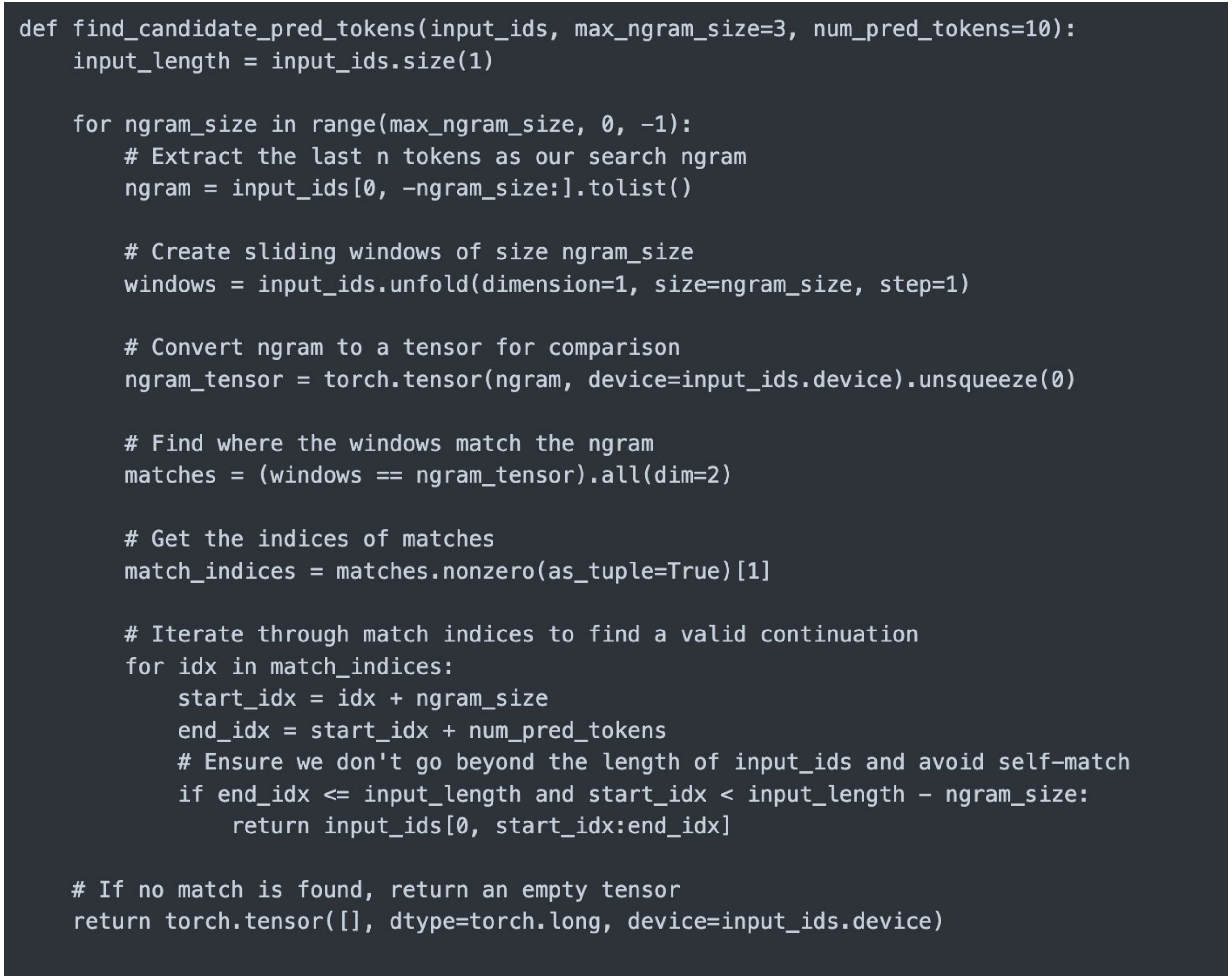
- Generation follows a **memory-bound** computational pattern
- Main latency bottleneck arises from **memory reads/writes** rather than arithmetic computations due to the inherently **sequential nature of the autoregressive** decoding process
- A common mitigation for this inefficiency was to simply increase the batch size, enabling the parallel production of more tokens
- Increasing the batch size in this context not only introduces **higher latency** but also substantially inflates the memory requirements for the Transformer **model's key-value cache**
- As of September 2023, generation costs approximately **2x higher for GPT-4** and roughly **3x for Claude 2**, compared to merely processing prompts

Verification (Explained in Depth) Mitigating Batch Size

1. Instead of creating a batch for every new token, you can just append the new candidate tokens in the prompt and verify those tokens in one forward pass of the model.
2. Assuming your draft model gives candidate tokens $\mathbf{y}_{\mathbf{1:n}}$ for the prompt $\mathbf{x}$
3. Now during the forward pass, instead of just getting the last row of the attention layer, you take the last n rows where for row i, the last unmasked attention weight corresponds to the x+(n-1)th token.
4. Now the attention embeddings for each token can be sent to the LM head.
5. Each candidate token can now be verified.

Prompt Lookup (Speedup is task dependent)

- In several LLM use cases where you're doing input grounded generation (summarization, document QA, multi-turn chat, code editing), there is high n-gram overlap between LLM input (prompt) and LLM output.
- This could be entity names, phrases, or code chunks that the LLM directly copies from the input while generating the output.
- In the next slide is the prompt lookup function used in hf generate.



```python
def find_candidate_pred_tokens(input_ids, max_ngram_size=3, num_pred_tokens=10):
    input_length = input_ids.size(1)

    for ngram_size in range(max_ngram_size, 0, -1):
        # Extract the last n tokens as our search ngram
        ngram = input_ids[0, -ngram_size:].tolist()

        # Create sliding windows of size ngram_size
        windows = input_ids.unfold(dimension=1, size=ngram_size, step=1)

        # Convert ngram to a tensor for comparison
        ngram_tensor = torch.tensor(ngram, device=input_ids.device).unsqueeze(0)

        # Find where the windows match the ngram
        matches = (windows == ngram_tensor).all(dim=2)

        # Get the indices of matches
        match_indices = matches.nonzero(as_tuple=True)[1]

        # Iterate through match indices to find a valid continuation
        for idx in match_indices:
            start_idx = idx + ngram_size
            end_idx = start_idx + num_pred_tokens
            # Ensure we don't go beyond the length of input_ids and avoid self-match
            if end_idx <= input_length and start_idx < input_length - ngram_size:
                return input_ids[0, start_idx:end_idx]

    # If no match is found, return an empty tensor
    return torch.tensor([], dtype=torch.long, device=input_ids.device)
```

Prompt Lookup in action

Jacobi Decoding (Batched Decoding Process) (Inefficient)

1. Let's say through a helper model you have n future tokens (y_1 y_n) for a given prefix **x.**
2. Create a batch of size n with ith input in the batch being [**x** $y_{1:i-1}$]
3. Now pass the batch through the model.
4. The forward pass will generate the y_i token assuming the previous generated tokens are correct.
5. If the first k tokens generated by the model match the candidate tokens, then the k+1 th token will also match the generated token.
6. So k tokens will be generated in 1 forward pass.

Lookahead Decoding (extended Jacobi Decoding)

- Guess and Verify Paradigm
- Initially randomly select the next k tokens as candidate tokens
- Create the causal mask and verify.
- Take tokens that matched in the generation
- For the tokens that did not match create ngrams and put them in the a cache pool to be reused later (This creates the candidate tokens for the future).
- As more iterations take place, the random guesses become more useful ngrams that can be accepted in the future.

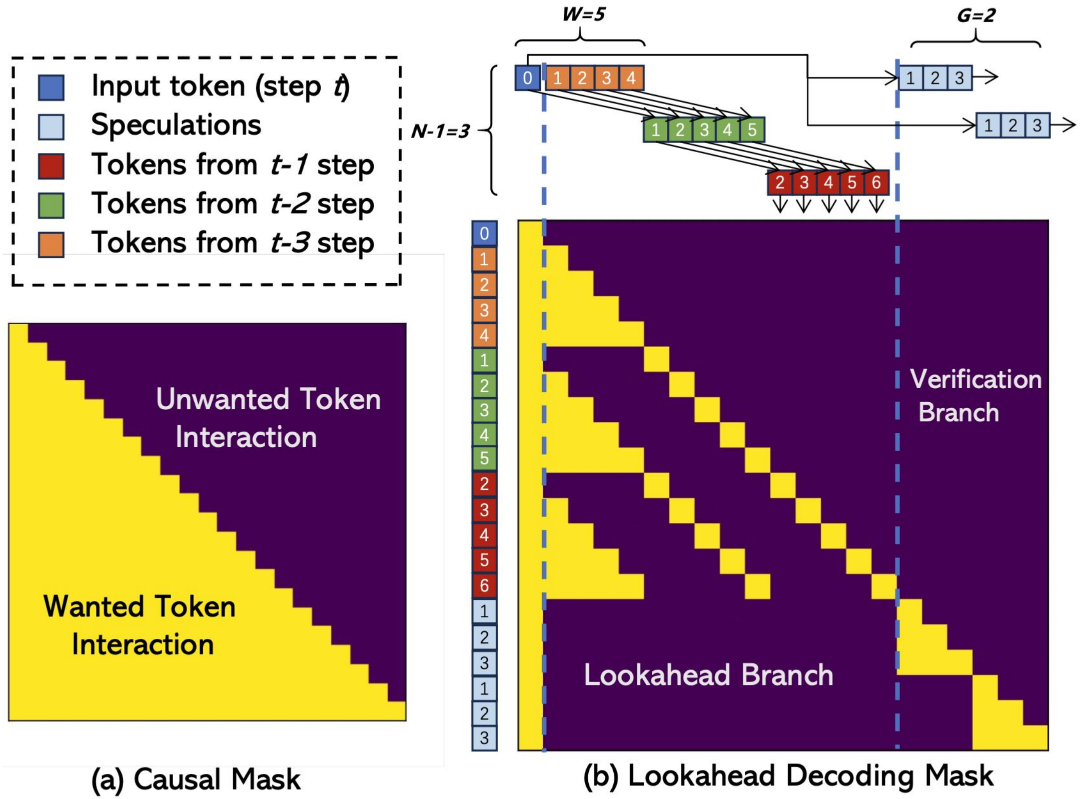
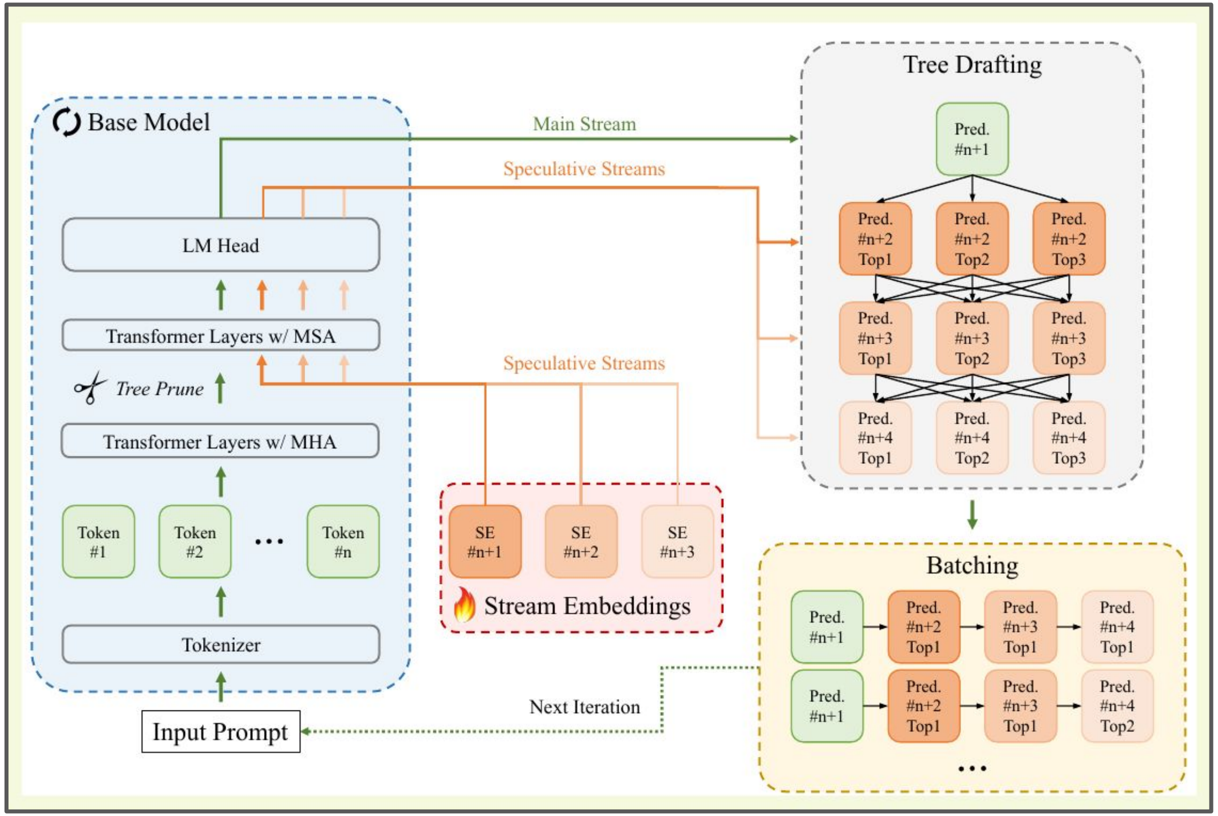


Figure 2: (a) Causal mask for decoder models. (b) Attention mask for LOOKAHEAD DECODING with $W = 5$, $N = 4$, and $G = 2$. Digits on tokens indicate relative positions.

Streaming LLMs



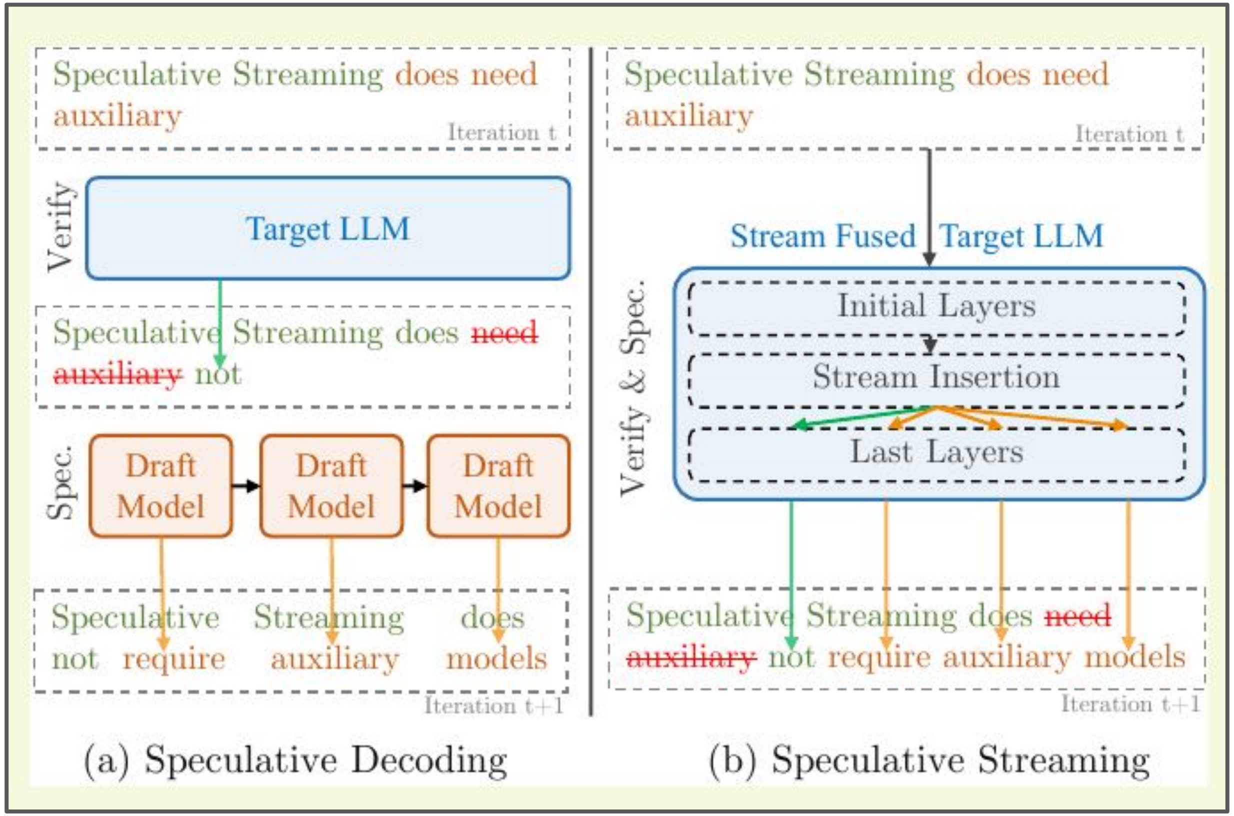
Motivation

1. Eliminating the need for a draft model
2. Streamlining Fine Tuning of just a single model
3. End-to-end trainable single-model framework capable of simultaneously predicting the next token and speculating future tokens.
4. Speed Up the decoding Process

Main Modifications in the Paper

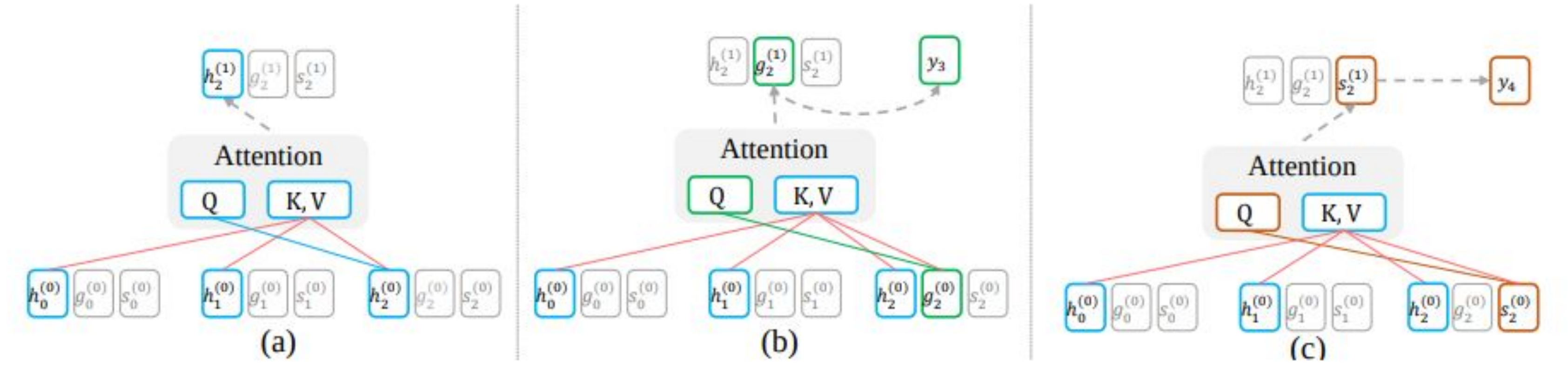
- Speculative Stream Design and Initialization
- Parallel Speculation
- Parallel Verification
- Parallel Tree Draft Pruning
- Modified Training Objective for future ngram prediction

Comparison with Speculative Decoding



Speculative Streaming does need auxiliary
Iteration t
Verify
Target LLM
Speculative Streaming does ~~need~~ ~~auxiliary~~ not
Spec.
Draft Model
Draft Model
Draft Model
Speculative Streaming does not require auxiliary models
Iteration t+1
(a) Speculative Decoding
Speculative Streaming does need auxiliary
Iteration t
Stream Fused Target LLM
Verify & Spec.
Initial Layers
Stream Insertion
Last Layers
Speculative Streaming does ~~need~~ ~~auxiliary~~ not require auxiliary models
Iteration t+1
(b) Speculative Streaming

Multi Stream Attention (From ProphetNet)



$h_2^{(1)}$ $g_2^{(1)}$ $s_2^{(1)}$
Attention
Q
K, V
$h_0^{(0)}$ $g_0^{(0)}$ $s_0^{(0)}$ $h_1^{(0)}$ $g_1^{(0)}$ $s_1^{(0)}$ $h_2^{(0)}$ $g_2^{(0)}$ $s_2^{(0)}$
(a)
y_3
(b)
y_4
(c)

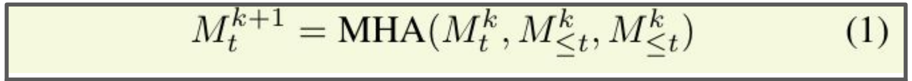
Figure 2: N-stream self-attention mechanism which contains a main stream self-attention and n predicting stream self-attention. For simplicity sake, we take 2-stream self-attention ($n = 2$) as an example here. Figure (a) presents the attention process of the main stream self-attention. Figure (b) and Figure (c) show the attention process of 1-st predicting stream and 2-nd predicting stream, respectively.

Speculative Stream Design and Initialisation

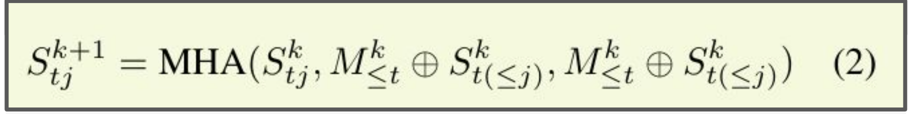
1. The main multi headed attention (MHA) of the model is the main stream. It’s output is used for predicting the next token
2. We have k additional attention layers in parallel with the main MHA, which are the speculative streams.
3. The output of these streams is used for predicting the future tokens, with stream i responsible for predicting y_i

Multi Stream Attention Initialisation

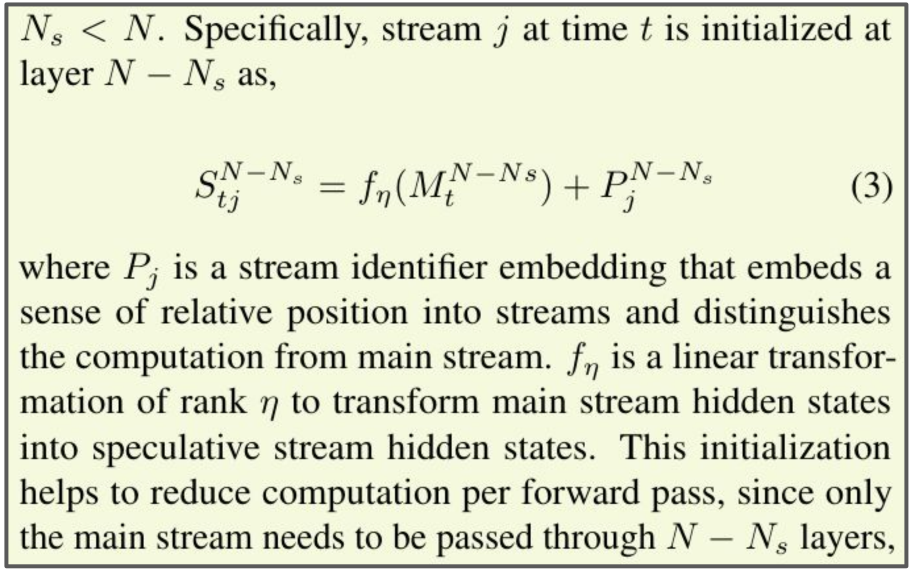
1. In the model, MSA is applied only to the last Ns layers of the Transformer.
2. Each stream j is initialised using the output of the main-stream from the previous Transformer block.
3. P_j is a positional embedding that incorporates a sense of position to the stream enabling it to predict the future token according to its position



$$M_t^{k+1} = \text{MHA}(M_t^k, M_{\le t}^k, M_{\le t}^k) \qquad (1)$$



$$S_{tj}^{k+1} = \text{MHA}(S_{tj}^k, M_{\le t}^k \oplus S_{t(\le j)}^k, M_{\le t}^k \oplus S_{t(\le j)}^k) \qquad (2)$$



$N_s < N$. Specifically, stream j at time t is initialized at layer $N - N_s$ as,

$$S_{tj}^{N-N_s} = f_\eta(M_t^{N-Ns}) + P_j^{N-N_s} \qquad (3)$$

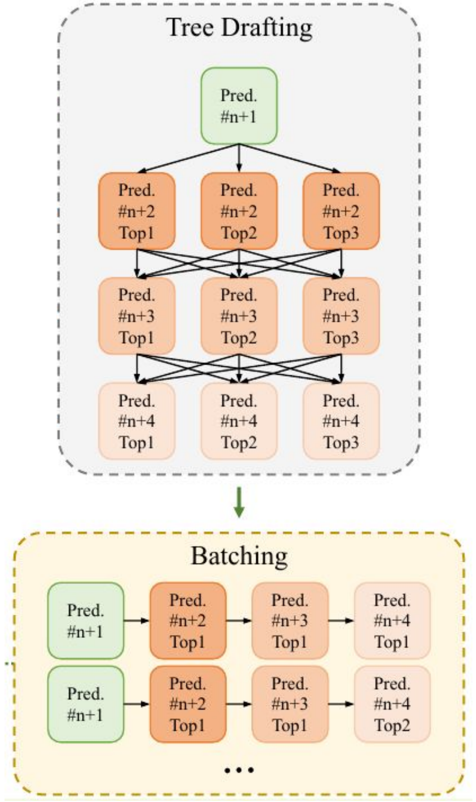
where P_j is a stream identifier embedding that embeds a sense of relative position into streams and distinguishes the computation from main stream. f_η is a linear transformation of rank η to transform main stream hidden states into speculative stream hidden states. This initialization helps to reduce computation per forward pass, since only the main stream needs to be passed through $N - N_s$ layers,

Parallel Speculation

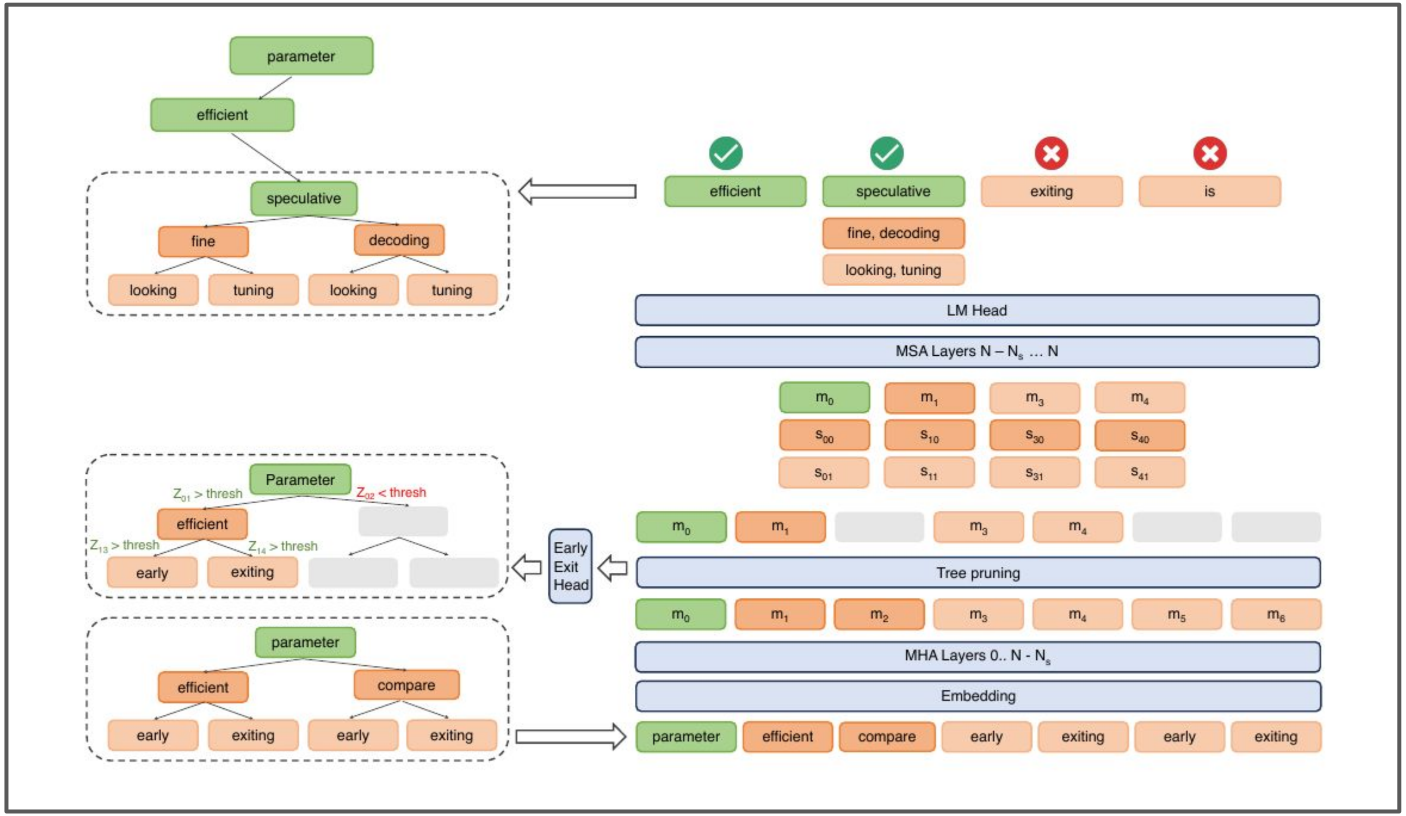
1. Now as we have the output from the attention layers from the main and speculative stream.
2. We can use them to generate $[y_1 .. y_{k+1}]$ where y_1 is generated using the main-stream and the rest are generated using the speculative streams.
3. In order to keep track of the generated speculations, instead of keeping the top 1 token from the speculated streams we keep the top k tokens from each stream in a Tree.

Tree Draft

1. Each Layer of the tree corresponds to the top k predictions for the corresponding stream.
2. Each path in the tree corresponds a viable candidate for verification.
3. The root of the tree is generated from the main-stream.
4. Edges of the tree correspond to the transition probability from parent to child token.



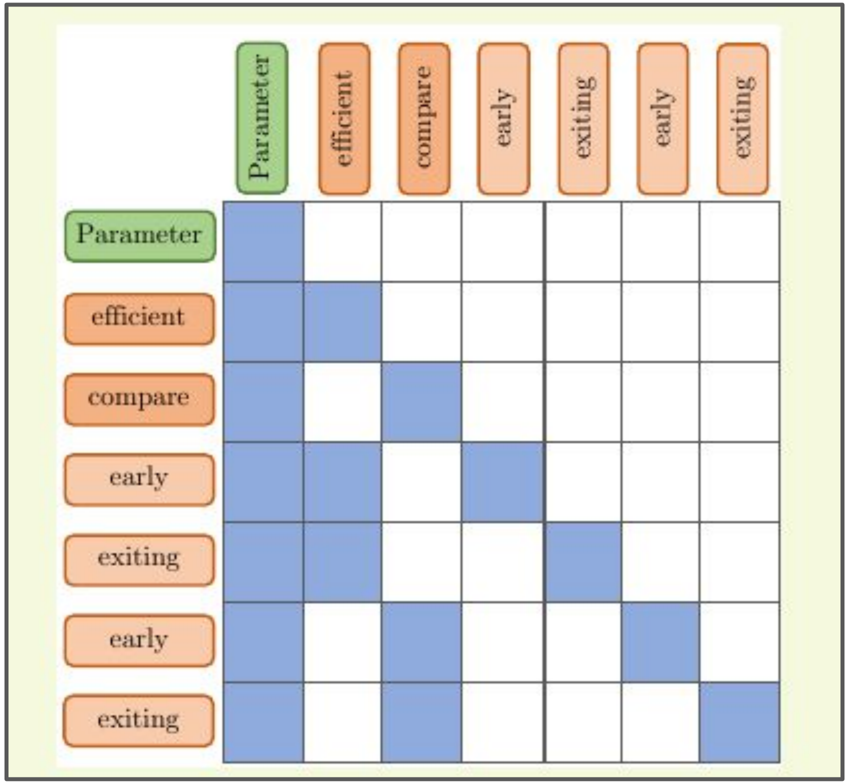
Parallel Verification



Attention Mask For Tree Draft

1. Tree Draft is flattened and the attention mask is set in such a way that the children attend to all its predecessors.

2. Now since we have the output for the attention for the future token we can feed it to the LM Head and get the next token, hence verifying multiple candidates simultaneously.



Tree Draft Pruning

1. One issue with tree draft is that every permutation of k tokens sampled from a stream is a viable candidate.
2. As the batch size containing candidates for verification increases the verification starts becoming compute bound.
3. We need to prune the Tree Draft to reduce the batch size.
4. Some tokens from the tree are removed based on the transition probability between the parent and the immediate child token.
5. To estimate the transition probability we get the hidden state output from an early Transformer layer (just before MSA) and pass it through a low rank linear transformation and use the LM Head to get the probability distribution which can be used to calculate the transition probability.

Training Objective

1. Along with the next token prediction training objective we also minimize the log probability for the future n-grams obtained using MSA.

$$L_{ss} = -\alpha_0\left(\sum_{t=1}^{T} \log p_\theta(y_t|y_{<t}, x)\right) - \sum_{j=1}^{\gamma} \alpha_j\left(\sum_{t=1}^{T-j} \log p_\theta(y_{t+j}|y_{<t}, x)\right) \tag{5}$$

Results

Comparisons are made using the following models:

1. OPT 1.3B, 6.7B with OPT 125m as draft for speculative decoding
2. Phi 1.3B
3. Open-Llama 7B

Additionally the following methods are compared against:

1. Baseline (Standard Autoregressive decoding)
2. Medusa
3. Speculative Decoding

Results

1. Metric used for generation quality is
   a. Exact Match for SqlContext
   b. Rouge1/RougeLSum for DialogSum and E2E-NLG

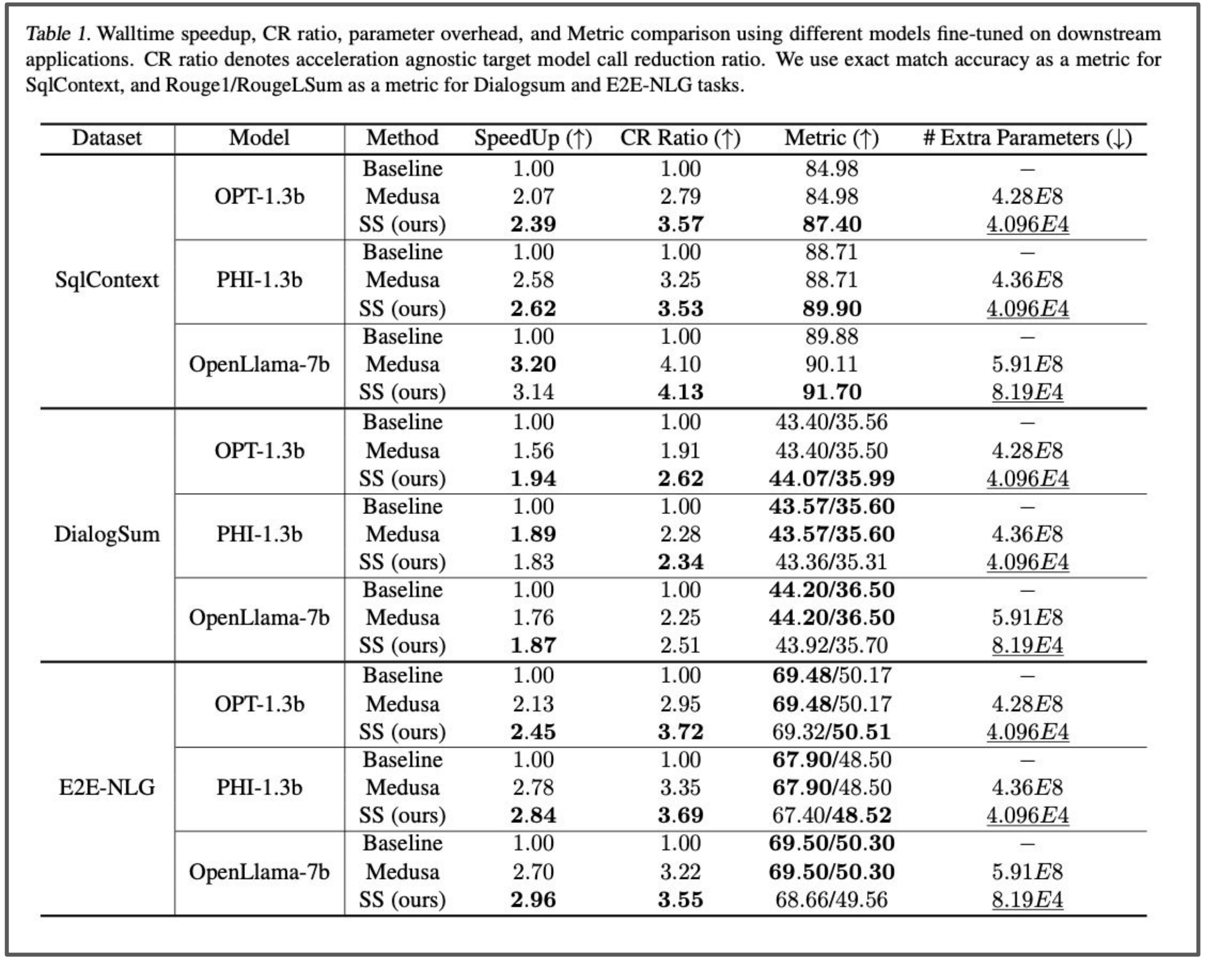


Table 1. Walltime speedup, CR ratio, parameter overhead, and Metric comparison using different models fine-tuned on downstream applications. CR ratio denotes acceleration agnostic target model call reduction ratio. We use exact match accuracy as a metric for SqlContext, and Rouge1/RougeLSum as a metric for Dialogsum and E2E-NLG tasks.

Dataset	Model	Method	SpeedUp (↑)	CR Ratio (↑)	Metric (↑)	# Extra Parameters (↓)
SqlContext	OPT-1.3b	Baseline	1.00	1.00	84.98	–
		Medusa	2.07	2.79	84.98	4.28E8
		SS (ours)	**2.39**	**3.57**	**87.40**	4.096E4
	PHI-1.3b	Baseline	1.00	1.00	88.71	–
		Medusa	2.58	3.25	88.71	4.36E8
		SS (ours)	**2.62**	**3.53**	**89.90**	4.096E4
	OpenLlama-7b	Baseline	1.00	1.00	89.88	–
		Medusa	**3.20**	4.10	90.11	5.91E8
		SS (ours)	3.14	**4.13**	**91.70**	8.19E4
DialogSum	OPT-1.3b	Baseline	1.00	1.00	43.40/35.56	–
		Medusa	1.56	1.91	43.40/35.50	4.28E8
		SS (ours)	**1.94**	**2.62**	**44.07/35.99**	4.096E4
	PHI-1.3b	Baseline	1.00	1.00	**43.57/35.60**	–
		Medusa	**1.89**	2.28	**43.57/35.60**	4.36E8
		SS (ours)	1.83	**2.34**	43.36/35.31	4.096E4
	OpenLlama-7b	Baseline	1.00	1.00	**44.20/36.50**	–
		Medusa	1.76	2.25	**44.20/36.50**	5.91E8
		SS (ours)	**1.87**	2.51	43.92/35.70	8.19E4
E2E-NLG	OPT-1.3b	Baseline	1.00	1.00	**69.48**/50.17	–
		Medusa	2.13	2.95	**69.48**/50.17	4.28E8
		SS (ours)	**2.45**	**3.72**	69.32/**50.51**	4.096E4
	PHI-1.3b	Baseline	1.00	1.00	**67.90**/48.50	–
		Medusa	2.78	3.35	**67.90**/48.50	4.36E8
		SS (ours)	**2.84**	**3.69**	67.40/**48.52**	4.096E4
	OpenLlama-7b	Baseline	1.00	1.00	**69.50/50.30**	–
		Medusa	2.70	3.22	**69.50/50.30**	5.91E8
		SS (ours)	**2.96**	**3.55**	68.66/49.56	8.19E4

Wall Time Latency Comparison

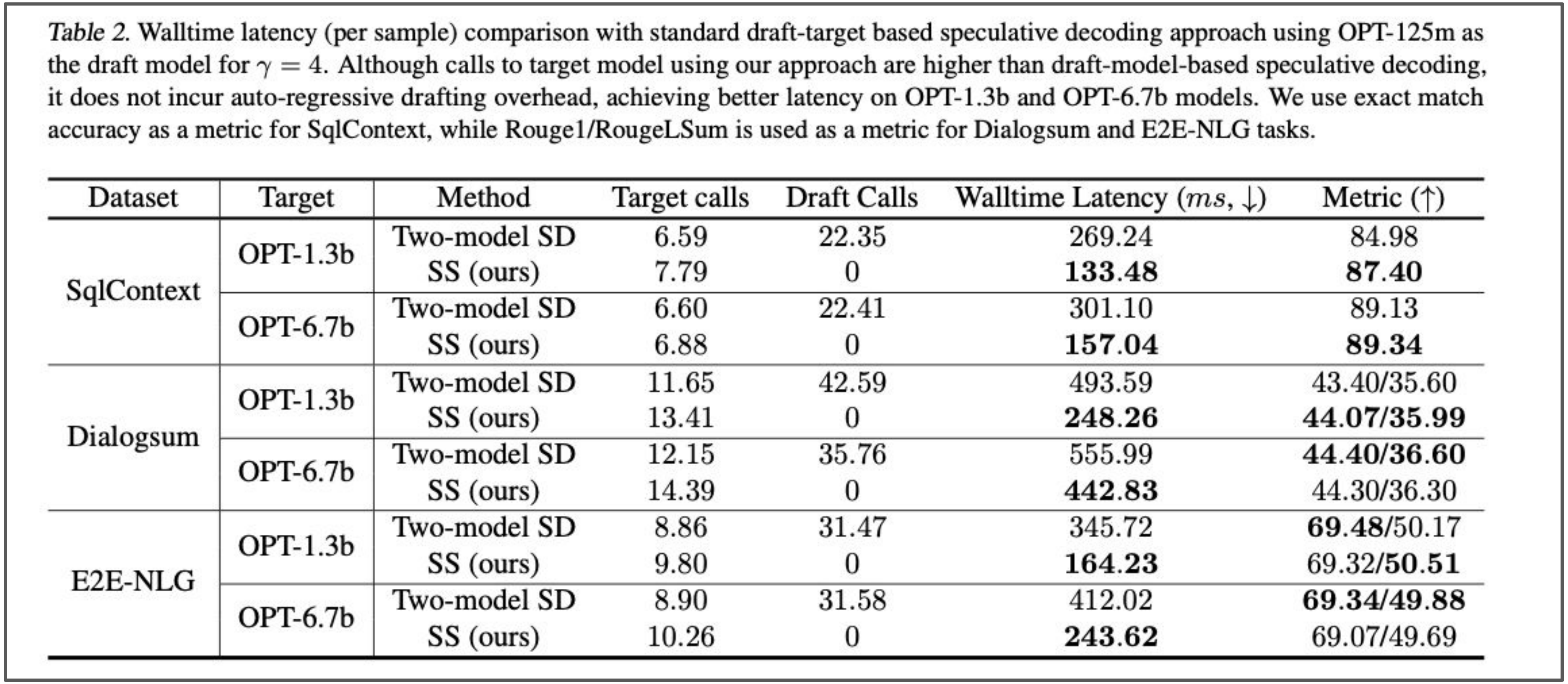


Table 2. Walltime latency (per sample) comparison with standard draft-target based speculative decoding approach using OPT-125m as the draft model for $\gamma = 4$. Although calls to target model using our approach are higher than draft-model-based speculative decoding, it does not incur auto-regressive drafting overhead, achieving better latency on OPT-1.3b and OPT-6.7b models. We use exact match accuracy as a metric for SqlContext, while Rouge1/RougeLSum is used as a metric for Dialogsum and E2E-NLG tasks.

Dataset	Target	Method	Target calls	Draft Calls	Walltime Latency (ms, $\downarrow$)	Metric ($\uparrow$)
SqlContext	OPT-1.3b	Two-model SD	6.59	22.35	269.24	84.98
		SS (ours)	7.79	0	**133.48**	**87.40**
	OPT-6.7b	Two-model SD	6.60	22.41	301.10	89.13
		SS (ours)	6.88	0	**157.04**	**89.34**
Dialogsum	OPT-1.3b	Two-model SD	11.65	42.59	493.59	43.40/35.60
		SS (ours)	13.41	0	**248.26**	**44.07/35.99**
	OPT-6.7b	Two-model SD	12.15	35.76	555.99	**44.40/36.60**
		SS (ours)	14.39	0	**442.83**	44.30/36.30
E2E-NLG	OPT-1.3b	Two-model SD	8.86	31.47	345.72	**69.48**/50.17
		SS (ours)	9.80	0	**164.23**	69.32/**50.51**
	OPT-6.7b	Two-model SD	8.90	31.58	412.02	**69.34/49.88**
		SS (ours)	10.26	0	**243.62**	69.07/49.69

Analysis (Kernel and Memory Utilisation)

High Kernel Utilisation is Good

Low Memory Utilisation is Good

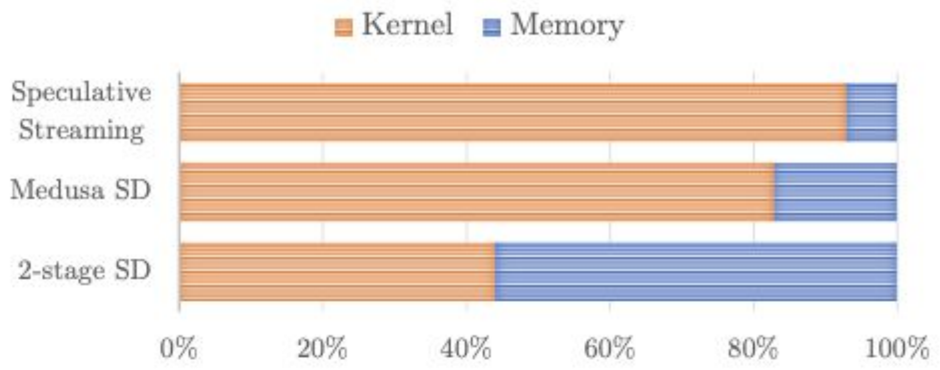


Figure 3. Speculative Streaming speeds up decoding by increasing arithmetic intensity of memory bound auto-regressive decoding step. Kernel and memory utilization of OPT-1.3b model with Medusa-style approach and draft model (OPT-125m) based speculative decoding approach is also shown for comparison.

Analysis (Speed up comparison with Draft-Target SD)

ζ = No. of decoding tokens advanced during verification

β = No. of tokens advanced in Speculative Streaming

$$(\gamma * C_{draft} + C_{target})/\zeta = C_{ss}/\beta \qquad (6)$$

$$(\gamma + C_{target}/C_{draft})/\zeta = (C_{ss}/C_{draft})/\beta$$

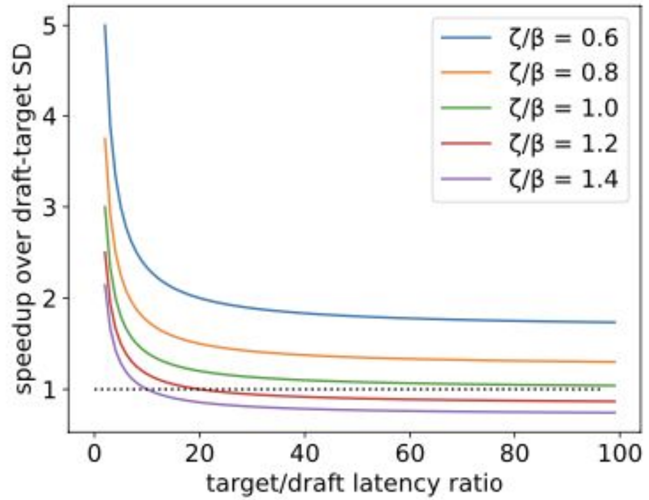


Figure 4. Speculative Streaming speedup over draft-based speculative decoding for different ζ/β and target/draft latency ratios, where ζ denotes the number of advancements per verification step for draft-based speculative decoding while β denotes the same for Speculative Streaming.

Analysis (Speed up and Tree Pruning as Tokens increase)

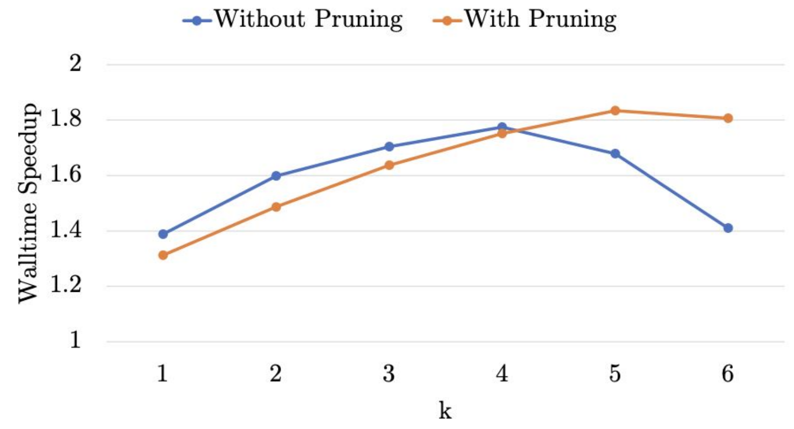


Figure 5. As more tokens (k) are sampled from each stream keeping γ fixed for the creation of a tree draft, walltime speedup increases due to the increased number of candidates. This trend reverses as k continues to increase and the model transits into the compute-bound phase. Pruning less probable paths from tree draft helps to reduce compute for higher values of k thereby reducing latency per forward pass and offering more speedup.

Metric Improvement with Number of MSA layers

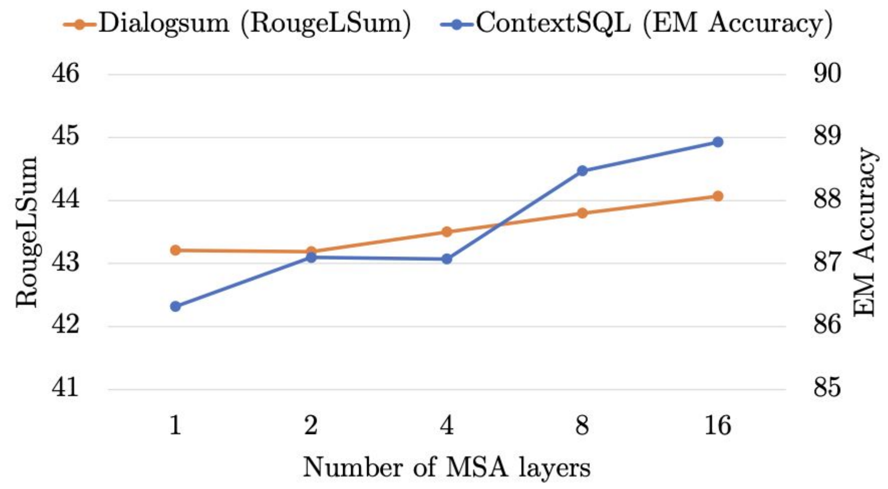
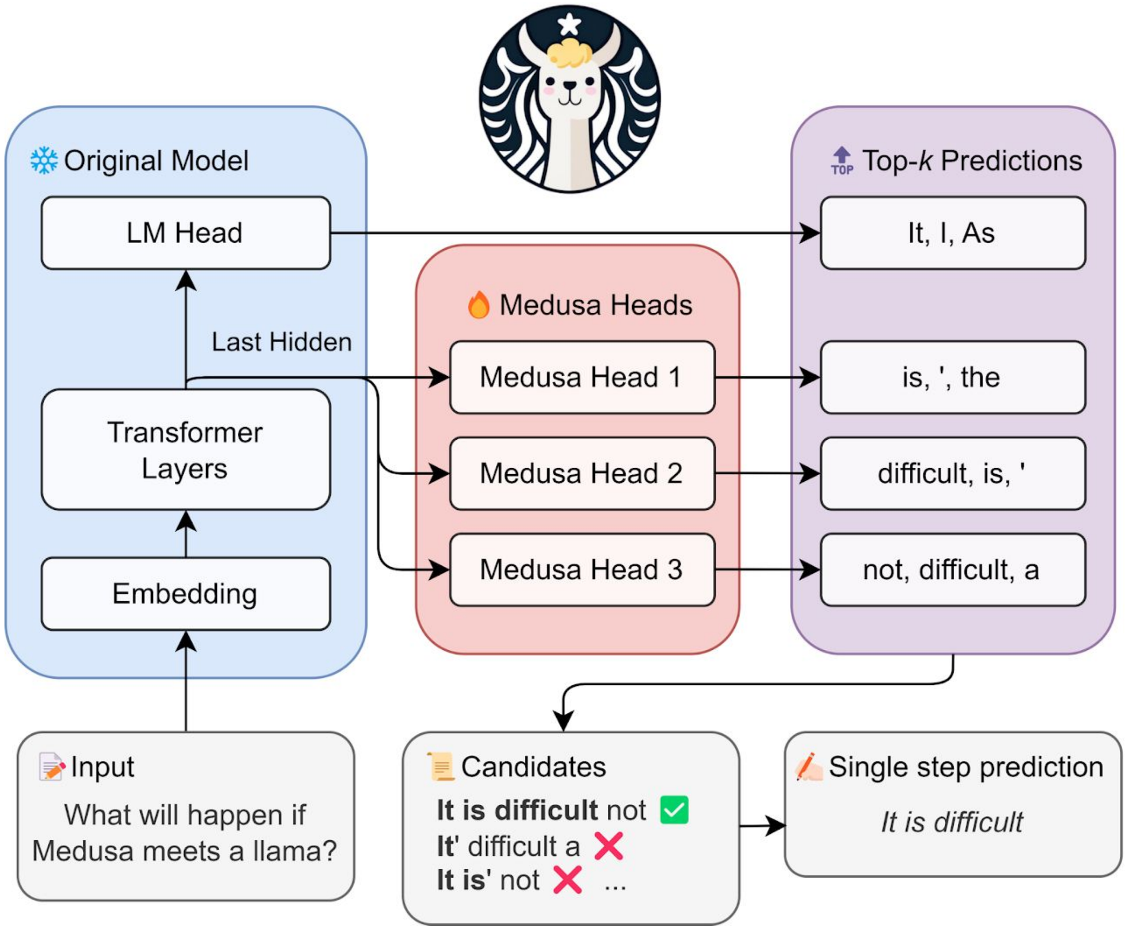


Figure 6. As the number of multi-stream attention layers increases, metrics on downstream tasks improve as well. We use RougeLSum as the metric for the Dialogsum task, and Exact Match (EM) accuracy as the metric for the ContextSQL task.

Medusa


Original Model
LM Head
Last Hidden
Transformer Layers
Embedding
Medusa Heads
Medusa Head 1
Medusa Head 2
Medusa Head 3
Top-k Predictions
It, I, As
is, ', the
difficult, is, '
not, difficult, a
Input
What will happen if Medusa meets a llama?
Candidates
It is difficult not ✓
It' difficult a ✗
It is' not ✗ ...
Single step prediction
It is difficult

Medusa (Main steps)

- Extend the original model to predict future tokens
- This is done by using extra LM Heads (Medusa Heads)
- The next tokens are verified in one pass using Tree Attention Mask
- The Medusa heads can be trained alone or in conjunction with the base model
- During inference, each head generates multiple top predictions for its designated position. These predictions are assembled into candidates and processed in parallel using a tree-based attention mechanism.
- The final step involves utilizing a typical acceptance scheme to select reasonable continuations, and the longest accepted candidate prefix will be used for the next decoding phase.

Layer Skip

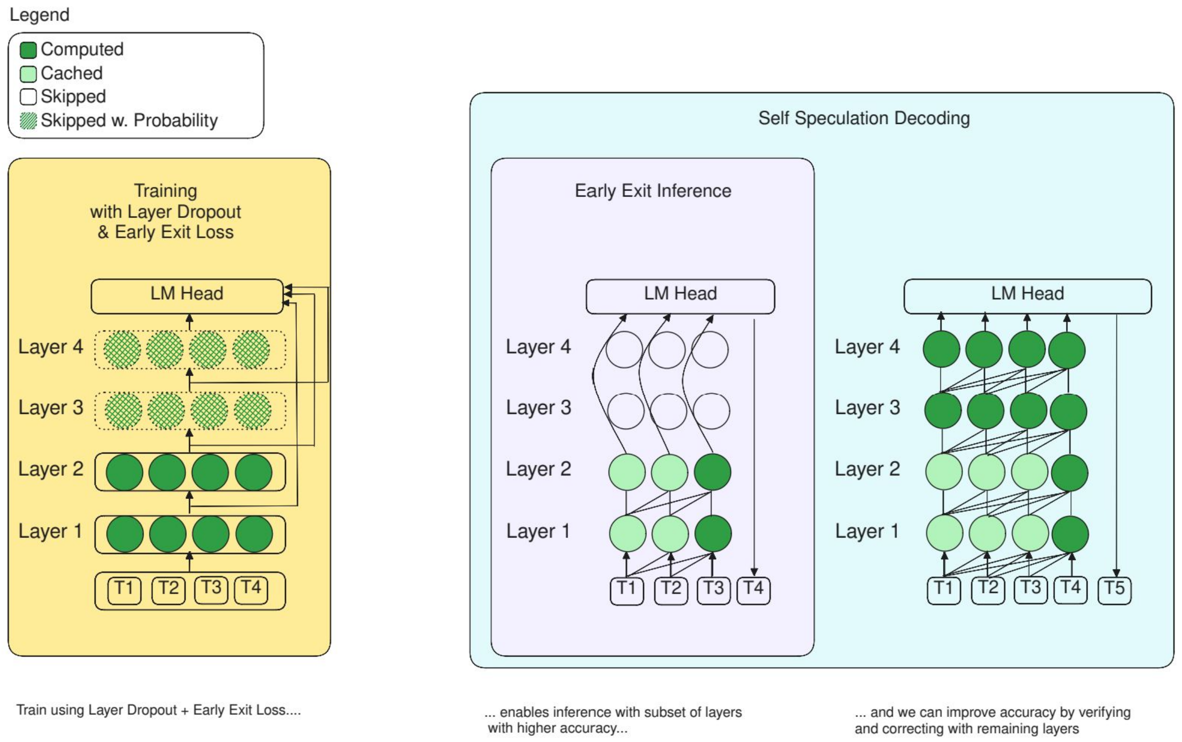
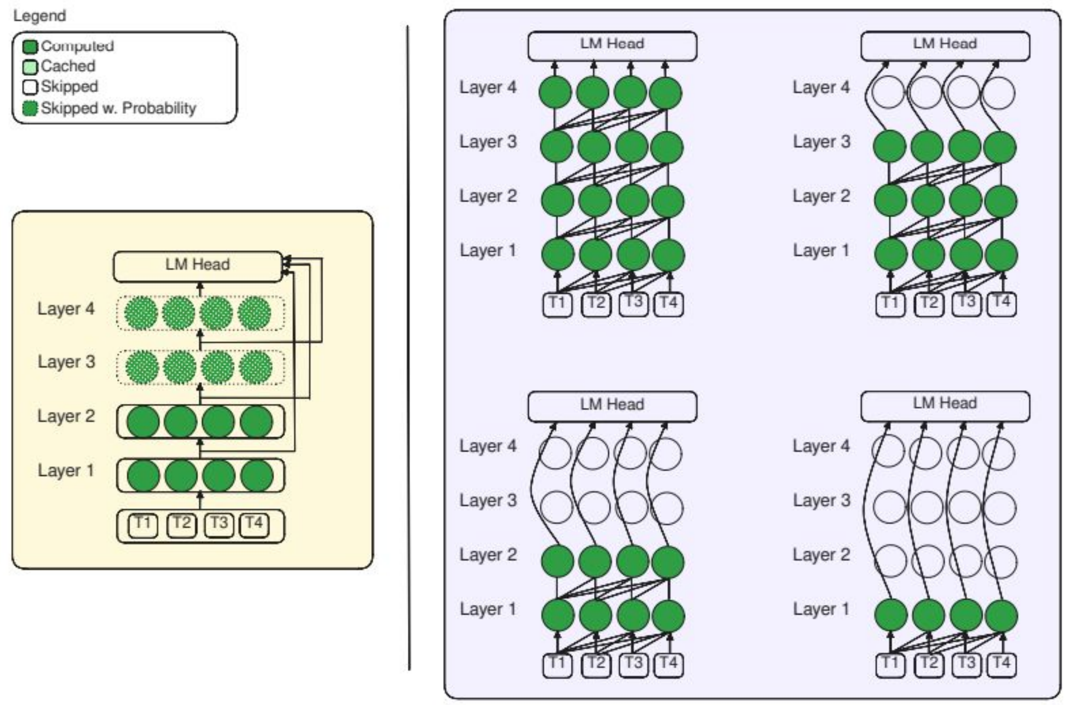

Legend
Computed
Cached
Skipped
Skipped w. Probability
Training with Layer Dropout & Early Exit Loss
LM Head
Layer 4
Layer 3
Layer 2
Layer 1
T1
T2
T3
T4
Self Speculation Decoding
Early Exit Inference
T5
Train using Layer Dropout + Early Exit Loss....
... enables inference with subset of layers with higher accuracy...
... and we can improve accuracy by verifying and correcting with remaining layers

Figure 1 Overview of our end-to-end solution, LayerSkip, showing its 3 components.

[2404.16710] Layer Skip: Enabling Early Exit Inference and Self-Speculative Decoding

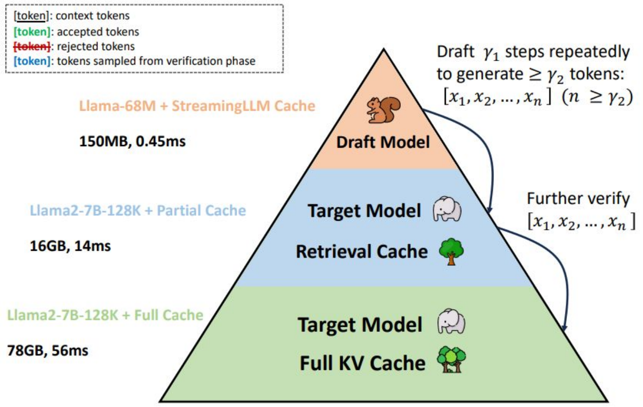
Layer Skip

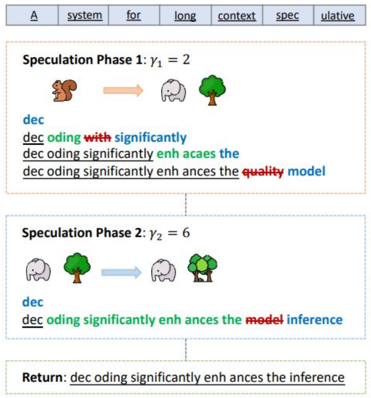


Train once using Layer Dropout + Early Exit.... ... to create different sized models with shared weights

[2404.16710] Layer Skip: Enabling Early Exit Inference and Self-Speculative Decoding

TriForce


[token]: context tokens
[token]: accepted tokens
~~[token]~~: rejected tokens
[token]: tokens sampled from verification phase
Llama-68M + StreamingLLM Cache
150MB, 0.45ms
Draft Model
Llama2-7B-128K + Partial Cache
16GB, 14ms
Target Model
Retrieval Cache
Llama2-7B-128K + Full Cache
78GB, 56ms
Target Model
Full KV Cache
Draft γ_1 steps repeatedly to generate $\geq \gamma_2$ tokens: $[x_1, x_2, \ldots, x_n]$ $(n \geq \gamma_2)$
Further verify $[x_1, x_2, \ldots, x_n]$


A | system | for | long | context | spec | ulative
Speculation Phase 1: $\gamma_1 = 2$
dec
dec oding ~~with~~ significantly
dec oding significantly enh acaes the
dec oding significantly enh ances the ~~quality~~ model
Speculation Phase 2: $\gamma_2 = 6$
dec
dec oding significantly enh ances the ~~model~~ inference
Return: dec oding significantly enh ances the inference

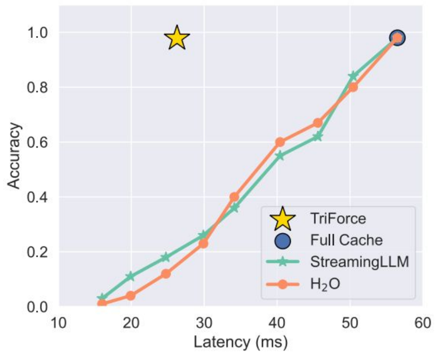
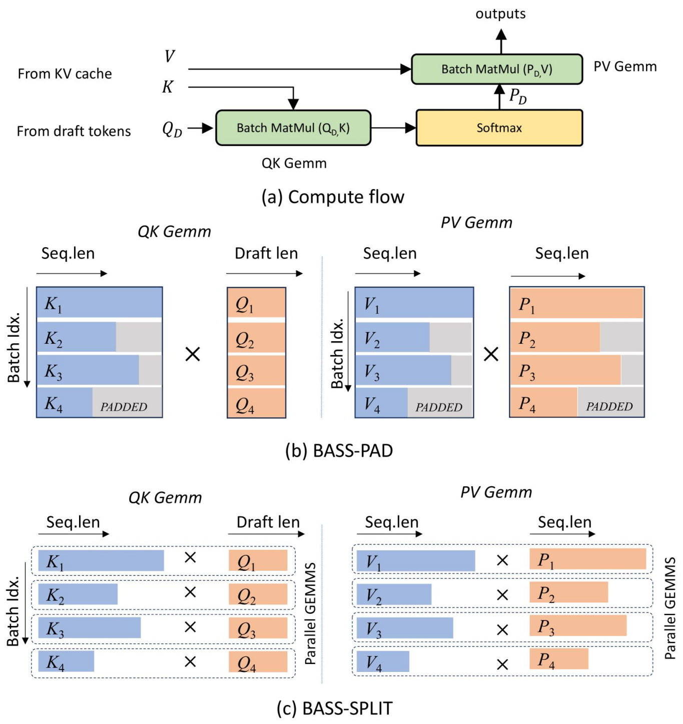


Figure 1: **Left**: TRIFORCE employs retrieval-based drafting and hierarchical speculation to effectively address the dual bottlenecks. It integrates two models and three caches, comprising a draft model, a target model, and a StreamingLLM cache for the draft model, alongside a retrieval cache and a full cache for the target model. The process initiates by repeatedly drafting for γ_1 steps, assisting the target model with retrieved partial KV cache in generating over γ_2 tokens, which will be further verified by the target model using full KV cache. **Right**: Evaluating the Llama2-7B-128K on a needle retrieval task indicates that KV cache eviction-based methods, such as StreamingLLM, require a trade-off between latency and accuracy. In contrast, our TRIFORCE successfully maintains low latency without sacrificing accuracy.

[2404.11912] TriForce: Lossless Acceleration of Long Sequence Generation with Hierarchical Speculative Decoding

BASS: Batched Attention-optimized Speculative Sampling



```
Algorithm 1 A heuristic to adjust draft length
$l_{\text{draft}} \leftarrow l_0$
$s \leftarrow 0$
for each speculative decoding step do
    $x_1, \cdots, x_b \leftarrow$ numbers of accepted tokens
    if $\max(x_1, \cdots, x_b) = l_{\text{draft}}$ then
        $l_{\text{draft}} \leftarrow \min(l_{\text{draft}} + l_{\text{incre}}, l_{\text{limit}})$
        $s \leftarrow 0$
    else
        $l_{\text{draft}} \leftarrow l_{\text{draft}} - \lceil l_{\text{draft}}/l_{\text{mod}} \rceil - s$
        $l_{\text{draft}} \leftarrow \max(1, x_1, \cdots, x_b, l_{\text{draft}})$
        $s \leftarrow 1$
    end if
end for
```

Figure 4: Attention calculation in BASS: (a) Attention compute flow, (b) BASS-PAD launches one kernel for QK GEMM and one kernel for PV GEMM by padding the K, V and P tensors to the maximum sequence length across the batch, and (c) BASS-SPLIT launches one kernel per sequence and thereby accommodates variable sequence lengths.

[2404.15778] BASS: Batched Attention-optimized Speculative Sampling

THANK YOU

Questions??

Additional Links

1. Medusa Paper: https://arxiv.org/abs/2401.10774
2. Medusa Repo: https://github.com/FasterDecoding/Medusa
3. Speculative Streaming Paper: https://arxiv.org/abs/2402.11131
4. Lookahead Decoding Paper: https://arxiv.org/abs/2402.02057
5. Lookahead Decoding Repo: https://github.com/hao-ai-lab/LookaheadDecoding
6. PromptLookup Repo: https://github.com/apoorvumang/prompt-lookup-decoding
7. Nanogpt Repo: https://github.com/karpathy/nanoGPT
8. ProphetNet Paper: https://arxiv.org/abs/2001.04063
9. HuggingFace Assisted Generation Blog : https://huggingface.co/blog/assisted-generation
10. Accelerating Generative AI with PyTorch II: GPT, Fast
11. At the Intersection of LLMs and Kernels - Research Roundup