Efficient Inference

Jonathan May

October 8, 2025

1 Why Inference is Inefficient

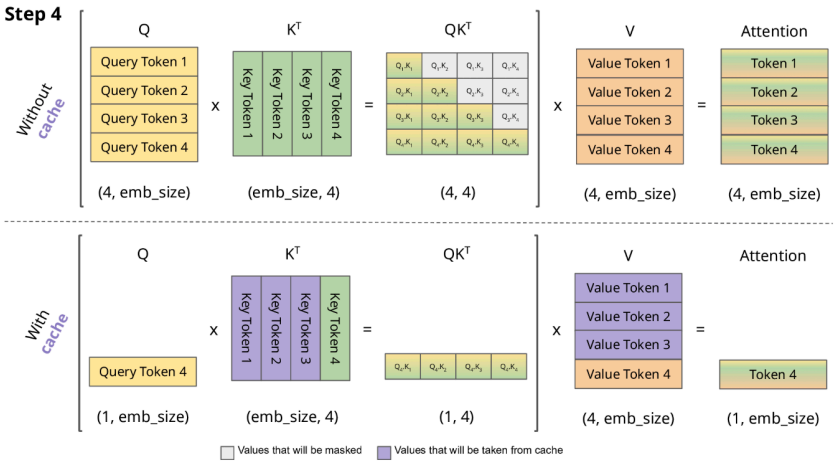
Repeated Steps As originally described, in order to calculate the next token you consider all the previous (or all if an encoder) tokens for suitability to attend to, by forming K representations of those tokens and Q representation of your current token. You then incorporate a V representation. But that means K_1 and V_1 (the K and V for token 1) is calculated for attending to position 2, position 3, etc. This is extra work!

GPUs are designed for throughput, not latency If you want to generate for a lot of input contexts at once using Transformers on GPUs you are in luck—the parallel operations make that rather efficient. Stack up all the contexts for your batch and the GPU will calculate the next token for all contexts all at once. If your batch size is 100 you get 100 outputs in (nearly) the time it takes to get one output. So the time per token is fast. But that still means you're waiting and see no output until you suddenly get all the outputs at once. This isn't helpful if you want low latency, i.e. you don't wait until you see output, you are stuck.

Quadratic Attention Remember that (assuming single head for a moment, though the math is the same) for every position, assuming n context length, attention is calculated for n previous positions. This is $O(n^2)$ calculations and as much memory stored as well, so as a sequence length grows the computational and memory needs explode.

2 Caching

K and V for the prefix tokens are repeatedly used as the text is being generated. It's slow to continuously do the matrix multiplication operation. Why not just save the pre-calculated K and V? The tradeoff is more memory usage; memory grows with the cache length.



However, OS-level considerations come into play, such as memory fragmentation, the need to distribute the cache across multiple servers, and paging policies and speculative loading. In practice a fixed cache is used, so then the question is when to evice memory from the cache. The considerations are the same as you would experience in a classic OS memory paging paradigm.

More interesting and specific to our architecture are lossy approaches designed to limit what is cached. One simple thing to do[1] is to recognize when a token is not being attended to frequently and then remove it from the cache and even refuse to attend to it any more. Other static policies similar to or combined with reduced attention may also be employed.

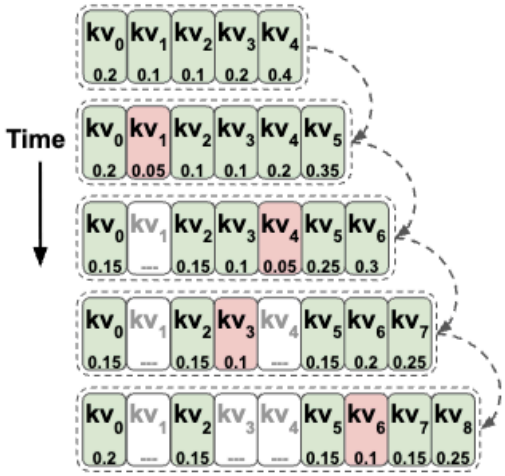
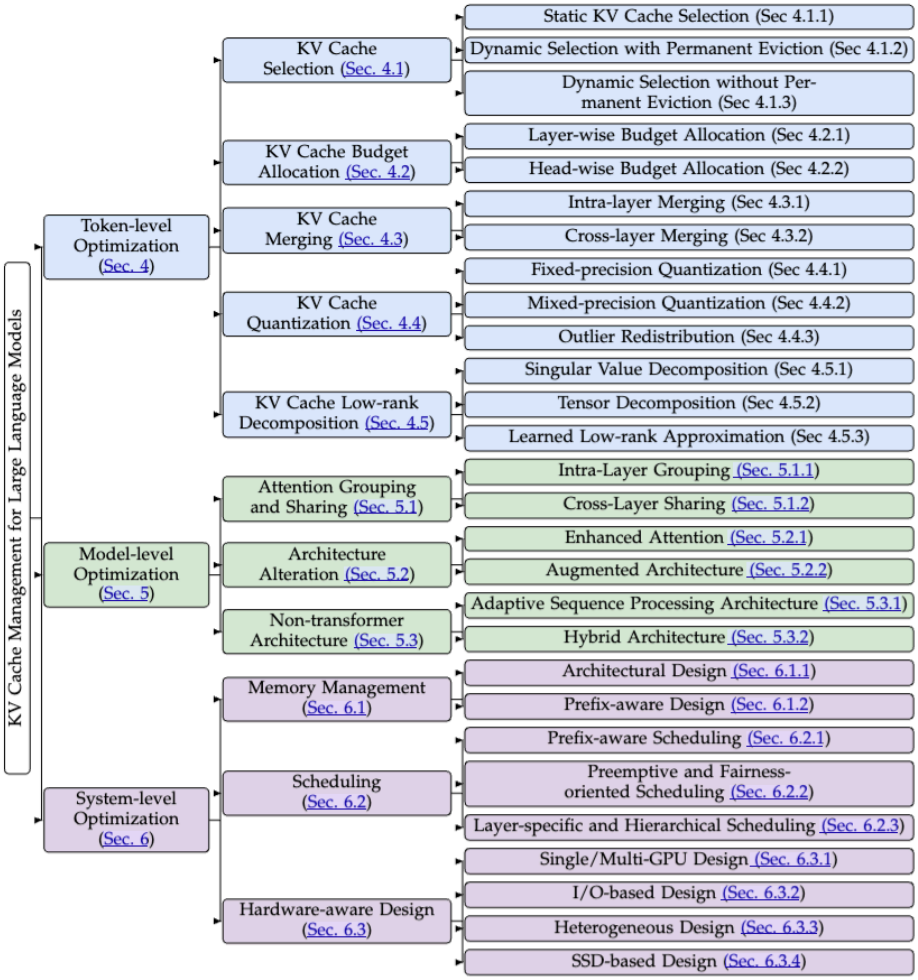


Figure 2: TOVA policy keeps a fixed-size multi-state (green cells). At each decoding step (different rows), the state with the lowest attention score is omitted (red cells, which become transparent in subsequent steps).

[1] https://arxiv.org/abs/2401.06104

A fairly comprehensive (looking?) survey in 2024[2] has a fairly exhaustive ontology of caching techniques, for the curious.



3 Attention Patterns

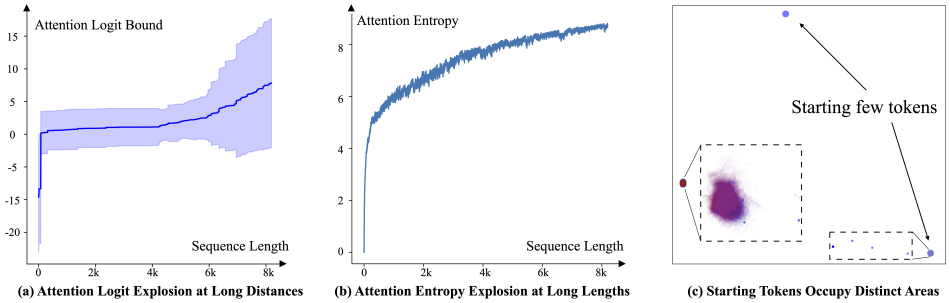
A big problem with transformer decoding is that attention to every previous token is conducted, which means decoding is quadratic. However, there is evidence that in practice most useful attention occurs within a fixed window of the current token, along with some attention to the beginning of a sequence. If a cap is placed on the number of positions attendable to, and there is not much net effect on performance, then we can obtain linear decoding instead of quadratic (because the number of attended tokens becomes a constant.

3.1 LM-Infinite

Apart from the difficulty of quadratic runtime, in this paper, authors observed that typical transformer models actually start to suffer when given inputs longer than their training

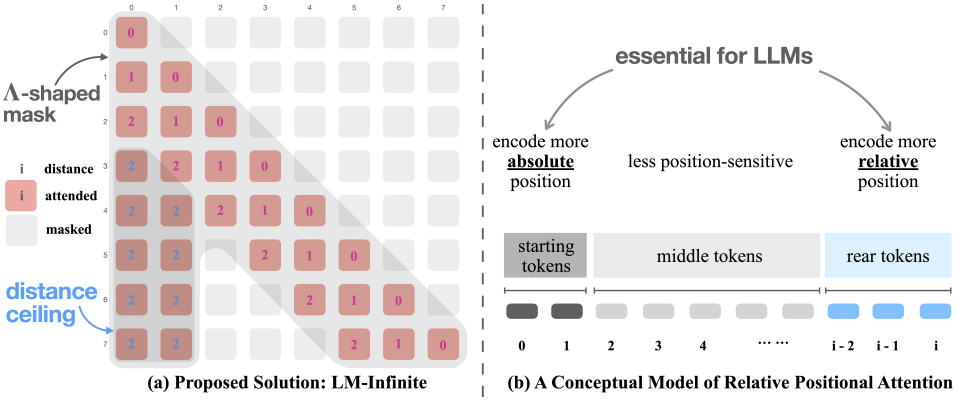
[2]https://arxiv.org/abs/2412.19442

inputs. Llama-2 is trained on segments up to 4k in length; beyond this the attention logits grow toward infinity, and entropy starts to grow. The authors also note that embeddings of the first couple dozen tokens in a sequence are somehow different from other tokens.



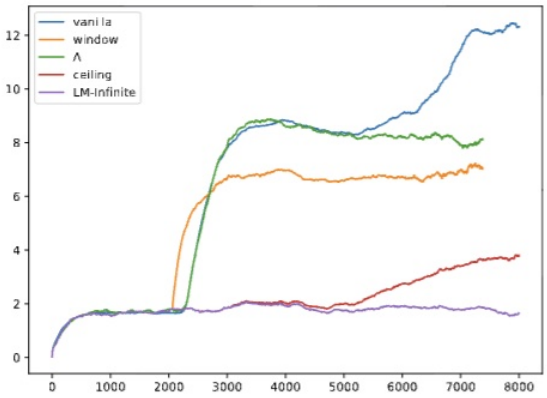
(a) Attention Logit Explosion at Long Distances (b) Attention Entropy Explosion at Long Lengths (c) Starting Tokens Occupy Distinct Areas

The solution is a statically defined attention mask. That is, just don't bother attending to some tokens. But which tokens? To begin with, just do sliding window attention. This means only the last L tokens are attended to. The idea is that since only L are ever seen in training, it's a bad idea to go beyond L (this was a practical limitation in the fixed position embedding days). But since there is something special about the start tokens, the authors decided to also attend to a fixed number of these. The result is a lambda-shaped pattern.



(a) Proposed Solution: LM-Infinite (b) A Conceptual Model of Relative Positional Attention

The final change (probably they did this at the beginning but the narrative is nice) is to alter the declared position for the initial tokens. Remember, those are still going to be more than L positions away and this is poorly modeled. So create a 'ceiling' and just treat them as being L away. These all turn out to make a pretty big difference.



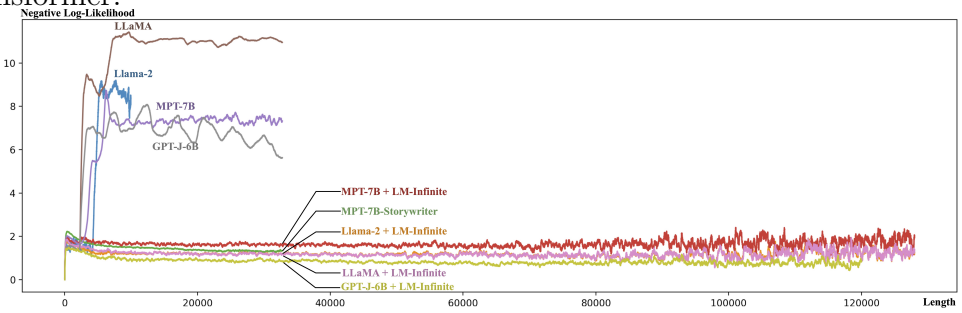
If you just use vanilla transformers, there's a spike at the training length limit (2k) and things get really bad after 6k. Reducing to a sliding window doesn't prevent the 2k spile

but seems to calm down the later spike. Interestingly, just resetting the first tokens to the ceiling and not sliding removes the 2k spike but naturally still leads to bad outcomes after 6k. Everything together is best, of course.

Overall results look pretty good. It's a little weird to just miss parts of the input but apparently these aren't used that much. Evaluated on QASPER, which is a QA dataset. "Each question is written by an NLP practitioner who read only the title and abstract of the corresponding paper, and the question seeks information present in the full text. The questions are then answered by a separate set of NLP practitioners who also provide supporting evidence to answers." Also evaluated on passkey retrieval, which is like needle in a haystack.

	Passkey Retrieval						Qasper
Model	6K	8K	10K	12K	16K	average	
Original	0.0	0.0	0.0	0.0	0.0	0.0	1.2
Truncated	66.0	55.3	38.8	32.8	27.3	44.0	30.1
LM-Infinite	**70.3**	**90.8**	**86.5**	**79.3**	**79.1**	**81.2**	**31.3**

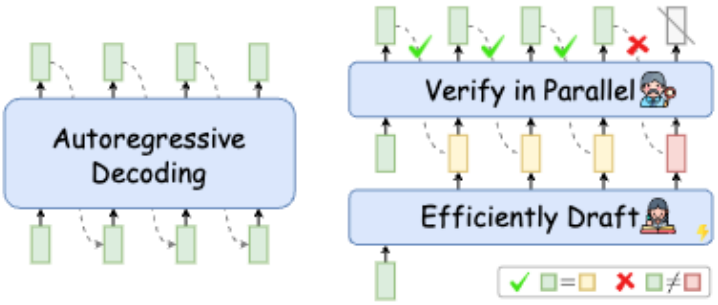
Results on NLL at long length don't look as good as MEGA (not directly compared) but MEGA is a whole separate architecture, while this can be added on top of any already-trained transformer.



4 Speculative Decoding

Let's assume we have a slow, good, large model S and a fast, bad, small model F. We want to decode with S but doing so is too slow because it requires a pass through the model for each token. KV caching speeds it up some but it's still too slow. We can't really take advantage of GPUs either because there's no parallelism (unless we have lots of people asking for inference at the same time). Imagine if we had a good idea of what we were going to generate ahead of time for the next n tokens. We're not sure the n tokens are correct but it's easier to verify them than to generate them because thanks to GPUs we can verify them all at once. That's the idea of speculative decoding:[3]

[3] `https://arxiv.org/pdf/2203.16487`



1. Using F, decode the next n tokens. This uses some compute but since F is so small, it doesn't use much.
2. Using S, check those n tokens, i.e. determine if they are indeed the most likely outputs. Note this is basically the same amount of compute as decoding the next 1 token.
3. At the first point i that S and F disagree, swap in the prediction from S and throw out the rest of the sequence from F.
4. Starting at $i+1$ or $n+1$ if the sequence was perfect, go back to step 1.

Worst case scenario we have the overhead from repeated calls to F on top of the cost to decode with S one token at a time (note that you are guaranteed one new S-generated token per call). Best case scenario you save $n-1$ calls to S. If you're willing to accept mismatches as long as S thinks they're fairly close to the top-1 then you can save even more time, though this is slightly lossy. The table below compares also to an earlier version, 'blockwise decoding', where S is used with extra heads to try to generate the next k tokens at a single position.

Models	EN→DE		DE→EN		EN→RO		RO→EN	
	Speed	BLEU	Speed	BLEU	Speed	BLEU	Speed	BLEU
Transformer-base ($b=5$)	1.0×	28.89	1.0×	32.53	1.0×	34.96	1.0×	34.86
Transformer-base ($b=1$)	1.1×	28.73	1.1×	32.18	1.1×	34.83	1.1×	34.65
Blockwise Decoding ($k=10$)	1.9×	28.73	2.0×	32.18	1.4×	34.83	1.4×	34.65
Blockwise Decoding ($k=25$)	1.6×	28.73	1.7×	32.18	1.2×	34.83	1.2×	34.65
SpecDec ($k=10$)	4.2×	28.90	4.6×	**32.61**	3.9×	35.29	4.1×	34.88
SpecDec ($k=25$)	**5.1×**	**28.93**	**5.5×**	32.55	**4.6×**	**35.45**	**4.8×**	**35.03**
12+2 Transformer-base ($b=5$)	1.0×	**29.13**	1.0×	32.45	1.0×	34.93	1.0×	34.80
12+2 Transformer-base ($b=1$)	1.1×	28.99	1.1×	32.08	1.1×	34.79	1.1×	34.55
Blockwise Decoding ($k=10$)	1.6×	28.99	1.7×	32.08	1.2×	34.79	1.2×	34.55
Blockwise Decoding ($k=25$)	1.4×	28.99	1.5×	32.08	1.1×	34.79	1.1×	34.55
SpecDec ($k=10$)	2.7×	29.08	3.0×	32.40	2.3×	**35.12**	2.4×	34.85
SpecDec ($k=25$)	**3.0×**	**29.13**	**3.3×**	**32.48**	**2.5×**	35.07	**2.6×**	**34.91**

In practice to get this to be efficient takes some careful engineering because you want to align your F and S model together so you're operating in lock step. Also, there are finicky considerations like that F and S need to have the same vocabulary. In the actual SpecDec paper the draft model is not just a small model; it is mildly non-autoregressive in that it predicts a sequence of subsequent tokens simultaneously given a context. There are, as always, lots of variations to explore.