Topological Transformer: A Redesign for Permanent Domain-Memory and cheaper kernel operations

Lorenzo Moriondo
Independent Researcher - tuned.org.uk
ORCID: 0000-0002-8804-2963

5 January 2026

Abstract

Starting from previous research on vector search for topological spaces (spectral search with *taumode*), I introduce in this paper the definition, concept implementation and testing of a new class of Transformer architecture focused on improving the attention mechanism. The Topological Transformer, or *Tauformer* applies all the features from current attention-based transformers (taking as baseline *nanoGPT*) and redesigns the attention mechanism at its core to: *(i)* allow delivering domain-specific context at the level of the attention mechanism for more domain-relevant token generation, *(ii)* improve time per token by $\sim 20\%$ and provide savings in KV-cache memory size by $\sim 50\%$ and *(iii)* draw a path to improve the Transformer performance on larger context windows with longer prompt lengths and high-dimensional embeddings; this is made possible by substituting inner-product with *taumode*'s synthetic index-based distance, aiming to provide relevant linear gains in training and generation compared to current GPTs.

1 Introduction

Concepts and tools developed in my previous publication *ArrowSpace: introducing Spectral Indexing for vector search* [1] have been here reused to redesign the attention mechanism forking the *nanoGPT* [2] implementation of Transformer architecture [3]; in particular *taumode*, a synthetic index based on the Rayleigh quotient [4] to compute the distribution of energy in the network defined by the embeddings space. Examples about how to compute the *taumode* distribution for any space of embeddings is available in the *pyarrowspace* repository.

Tauformer makes possible downstream technical improvements in computing and memory usage for the Transformer (GPT) and they are detailed in the following sections. These improvements are a consequence of pursuing the concept of providing a memory layer to LLMs that brought forward the idea of leveraging **distilled knowledge graphs (dkb) to deliver domain information to the attention mechanism at token generation level**. For dkb is intended here for example: *(i)* Text embeddings generated by feeding to an embedding model (in the tested example: TSDAE [5]) a representative corpus of the domain to map, this provides a vector space with manageable number of dimensions (384 in the example) that is a prerequisite to build the context windows at the attention level; *(ii)* Graph embeddings of triples generated using techniques like the SEPAL workflow demonstrated in [6], again the result is a vector space with a manageable dimensionality.
A coherent dkb is generated starting from the vector space defined by its embeddings (and its

Graph Laplacian computed using *arrowspace* [1]), this is considered from the point of view of the LLM system as a persistent memory that is founded on real-world-defined relations. Assuming a well-designed embedding pipeline, text embeddings from a curated text corpus or a curated Knowledge Base turned into graph embeddings become the numerical ground truths for the topology of the context, that is eventually delivered to the attention mechanism. Theoretically the Tauformer can also work, as any other transformer, on images and audio vectors or a mix of them but one of the critical points focuses on the process of producing the embeddings, how the vector space is designed and how its dimensionality fits in the attention mechanism.
The choices for: embeddings, attention mechanism and seeding the latent space are taken as design constraints for the data pipeline from the corpus to the latent space of the model. What is meant with seeding the latent space? Once the attention mechanism is working with *taumode* synthetic scores the content of the dkb percolates in the context windows through the usual Q, K, V iterations. Each distance measurements computed for Q, K, V is considered in the optics of the corpus/dataset from which the original embeddings were generated, delivering domain-specific metadata directly in the attention mechanism and through that to the latent space.

2 Statement of Need

This research is motivated by encoding at generation time an explicit (controllable and interpretable) domain structure beyond what the model can implicitly absorb into weights during training. Tauformer keeps the familiar Q, K, V and causal softmax pipeline but replaces the dot-product kernel with a distance metrics based on domain-specific manifold: each token/head is mapped to a topological signal and built into a scalar (*taumode* score) derived from the Graph Laplacian (manifold); this way attention is driven by distances in that manifold-aware scalar space rather than raw geometrical vector similarity. The hypothesis is that this process makes the model more scoped to the context in the training phase (forward) and in the generating phase (decoding), because weights and attention scores can be biased toward tokens that are similar under the learned domain manifold (i.e. a knowledge graph, citation graph, text embedding or any dataset originally passed to the embedding model). By hypothesis, this has the potential of improving contextual faithfulness relative to purely dot-product-based retrieval inside the context window. Not having permanent reference about the context is especially consequential in scientific and knowledge-intensive settings, where the desired notion of "relevance" is frequently defined less by raw generic geometrical proximity and more by the topology of domain relations and how information (in `arrowspace`, energy) propagates over them.

Tauformer is also motivated by the practical difficulty of scaling traditional GPTs to very long contexts and large embeddings, in particular at inference time: full-sequence self-attention has $O(T^2)$ compute in both the dot-product logits and the value aggregation, and decode-time still grows $O(T)$ per token while KV-cache memory grows linearly with T. Tauformer's cache design reduces KV memory by storing values plus a compact key-side scalar (k_i, λ_i) instead of full K and V tensors, yielding roughly a halving of KV-cache memory per layer (up to a small overhead for the scalar stream). With a sparse Laplacian from a domain manifold, the extra cost of computing the *taumode* scalars can shift from $O(D^2)$ to a sparsity-dependent cost $O(\text{nnz}(L))$ (non-zero elements of the Laplacian matrix), making the incremental overhead largely independent of sequence length and therefore more attractive as context windows and embeddings dimensions grow.

Tauformer addresses this by redesigning attention around a synthetic, topology-aware index computed over the Graph Laplacian of the domain's embedding space. Beside accuracy and controllability, this substitution is also motivated by systems constraints: the approach reduces dependence on storing key/value histories (full vectors) for long contexts and enables more memory-efficient inference, opening space for higher-dimensional embeddings and richer internal representations under fixed hardware budgets. The Graph Laplacian (gl) is usually a DxD matrix where

D is the number of dimensions that for the test dataset are defined by current SOTA embeddings models to 384. So basically a constant 384x384 sparse matrix ($nnz \sim 10^4$ for a realistic sparsity of 1-8%) allows limiting the computing cost for the iteration of Q, K, V in the attention mechanism. The Graph Laplacian can be seen as a distillation of the domain knowledge and its generation takes a cheap pre-training step that for the test dataset is $\sim$ 300 seconds on a common laptop hardware for a dataset of $3 * 10^5$ vectors on 384 dimensions.

The code for the GPT implementation is available at [7]. The code used to test and collect benchmark for the KV-cache is available at [8].

3 Memory Model and Algorithm

3.1 Generics

By leveraging the Graph Laplacian matrix (gl) built on the vector space defined by the embeddings for a given domain, it is possible to define a synthetic score ($\lambda\tau$, lambda-tau or `taumode`) that can be used to spot approximate nearest neighbours of a given vector (x) as: $\lambda\tau = taumode(x, gl)$. In the Tauformer architecture this distance metrics has been applied to the process of computing query, keys, value vectors (Q, K, V) in the attention mechanism; allowing the substitution of $Q^T\dot{K}$ with $-|\lambda_q - \lambda_k|$. This also enable saving a relevant percentage of the memory space used by the KV-cache and opens to having larger vectors and attention heads because part of the computing used by the inner-product operation can be reused to compute on higher dimensional vectors. These improvements sound already a nice step forward but, from knowledge engineering perspective, they are side-effects of the main concept brought forward by the implementation overall: *to develop a delivery system to bring distilled domain knowledge from the vector space (representing the knowledge graph, graph of citations, or any relations-representative structure) directly into the attention mechanism.* This to test the hypothesis that token generation can be made more context-relevant if domain-specific metadata is provided at the attention level. If this is made possible a new class of Transformers that are more or less narrowly scoped for context-specific generation in a given domain can be developed.

3.2 Tau Attention

In the process of causal attention using the *softmax* function, attention is driven by a transcendental normalisation over dot-product logits, where each new query must form inner-products against *all cached keys* (length T) in head dimension D, i.e., the decode-time kernel cost scales as $O(B\,H\,T\,D)$ for the $QK^\top$ logits plus another $O(B\,H\,T\,D)$ to aggregate values, with an additional $O(B\,H\,T)$ for masking and the softmax itself. Where the letters are:

- **B**: Batch size (number of sequences processed in parallel).
- **H**: Number of attention heads.
- **T**: Sequence length / number of time steps (tokens) in the context window.
- **D**: Head dimension (the per-head vector size, typically $D = C/H$ where C is the model embedding width).

In practice this also forces the KV-cache to store both K and V tensors, i.e., $2 \cdot B\,H_{kv}\,T\,D$ floats per layer, because the dot products cannot be reconstructed without the full key vectors. As context length grows, this "match against all previous keys and renormalise" pattern dominates both compute and memory, since every step repeats the same D-dimensional comparisons against an ever-growing set of cached vectors. Obviously this can be become a problem for high D.

Tauformer's *taumode* mechanism replaces the dot-product kernel with a scalar spectral signature per head vector: for each query/key $x \in \mathbb{R}^D$, it computes a bounded Rayleigh quotient energy $x^\top Lx/(x^\top x)$ (optionally blended with an item/edge dispersion statistic), producing a single λ-like score per token per head. Attention logits are then built by comparing the query's λ_q to **previously stored** λ_k values for cached keys:

$$a_{ij} = -|\lambda_{q,i} - \lambda_{k,j}|/\text{temp}$$

after which the causal mask and V-weighted sum are reused unchanged. This changes what must be cached: tauformer stores V and the **scalar** λ_k history rather than full K, reducing cache size from $2B\,H_{kv}\,T\,D$ floats to $B\,H_{kv}\,T\,D + B\,H_{kv}\,T$ floats (about $\sim 50\%$ savings for typical D). Computing shifts accordingly, instead of spending $O(B\,H\,T\,D)$ on dot-products at each decode step, Tauformer pays $O(B\,H\,\text{nnz}(L))$ to compute the query's Rayleigh term with a sparse Laplacian and only $O(B\,H\,T)$ to compare scalars against cached λ_k, making the scoring step dimension-light while keeping value aggregation $O(B\,H\,T\,D)$ the same (equivalence of outcome compared to `nanoGPT` are available in the implementation's unit tests).

The key improvement is that the Graph Laplacian is designed to be a sparse matrix, so optimising the code for a sparse representation the $O(D)$ terms become $O(nnz)$ with nnz being the number of non-zero elements in the matrix.

Further computing advantages are discussed in 4.

3.3 Built-in Memory

Beside the additional potential computing advantages that are analysed below, the Tauformer and relative `tauGPT` concept allow the kind of persistent memory described in 3.1; **delivering a persistent (long-term, unchanging in this initial design) memory within the attention mechanism**. This follows current research for supplementing LLMs with memory sub-systems: for example as seen in [9] where a neural long-term memory module for 2M+ token contexts works in parallel and provides prefix sub-vectors to the context window; or in [10] that categorises memory into sensory, short-term and long-term within agentic workflows; or in [11] that uses a "Memory Bank" to store and retrieve long-form text for in-context learning. In addition, the papers that try to supplement LLMs with agentic workflows: for example [12] that evaluates "Agentic Memory" across multi-hop and temporal reasoning tasks. For the purpose of this paper I call the running attention mechanism the "momentary memory" and the Graph Laplacian against which the distance scores are computed "persistent memory". This aligns with the scope of my current research for building an "AI Memory Layer" based on topological vector search or, more briefly, "Memory Models" (MM) [13].

Tauformer/tauGPT is the only, at my current knowledge, Transformer architecture that delivers domain knowledge directly in the attention mechanism providing the self-attention itself with some kind of starting context via the knowledge base metadata compressed in *gl*; leveraging the compression made possible by the *arrowspace* library and the *taumode* synthetic index. While technical and conceptual advantages are brought forward by this paper, further research is necessary to ascertain the improved accuracy in the training and token generation on very large context windows and the improved capability in avoiding generation problems sooner than at reasoning time.

3.4 Code

The first version of Tauformer/tauGPT [7] has been implemented using the Rust programming language [14] and the Burn Deep-Learning framework [15] as they provide fast, structured and type-safe development cycles with mature production ecosystem. Being a concept architecture this code is obviously not meant for production use at the publication of this paper.

No inner-product `tauGPT`'s most peculiar change is that it swaps dot-product attention for lambda-distance attention, where each token/head vector is compressed into a single scalar λ derived from a Laplacian energy, so that attention logits become $-|\Delta\lambda|/T$. The core of compression is in `lambdas_from_heads(...)`, which flattens $[B, H, T, D]$ into $[N, D]$, computes xL via a matmul, then forms $E_{\text{raw}} = (x^\top Lx)/(x^\top x + \epsilon)$ and bounds it as $e_raw/(e_raw + tau)$. The scalar λ is then broadcast into a full attention matrix using `taumode_distance_logits(...)`, which literally builds logits as $-((lq - lk).abs()/temp)$ after reshaping into $[B, H, Tq, 1]$ and $[B, H, 1, Tk]$.

```rust
// taumode.rs
pub fn lambdas_from_heads<B: Backend>(
x: Tensor<B, 4>,
lap: Param<Tensor<B, 2>>,
cfg: &TauModeConfig,
) -> Tensor<B, 3> {
  let [b, h, t, d] = x.dims();

  // Flatten [B,H,T,D] -> [N,D]
  let n = b * h * t;
  let x_nd = x.reshape([n, d]);

  // y = x L  -> [N,D]
  let y_nd = x_nd.clone().matmul(lap.val());

  // numerator = sum_i x_i * (xL)_i
  let numerator = (x_nd.clone() * y_nd).sum_dim(1); // [N]

  // denominator = sum_i x_i^2 + eps
  let denom = x_nd.powf_scalar(2.0).sum_dim(1) + cfg.eps; // [N]

  let e_raw = numerator / denom; // [N]
  let e_bounded = e_raw.clone() / (e_raw + cfg.tau); // [N]

  e_bounded.reshape([b, h, t])
}

pub fn taumode_distance_logits<B: Backend>(
lambda_q: Tensor<B, 3>,
lambda_k: Tensor<B, 3>,
cfg: &TauModeConfig,
) -> Tensor<B, 4> {
  let lq = lambda_q.unsqueeze_dim::<4>(3); // [B,H,Tq,1]
  let lk = lambda_k.unsqueeze_dim::<4>(2); // [B,H,1,Tk]
  let temp = cfg.temperature.max(cfg.eps);
  -((lq - lk).abs() / temp)
}
```

Sparse Laplacian The other distinctive design choice is that `TauModeAttention` is built to operate with either a dense test Laplacian or, as intended by design, a sparse Laplacian loaded from a manifold (a `.parquet` file with the computed Laplacian). The code explicitly switches between those representations at runtime. In `TauModeAttention`, you can see the dual storage for the sparse matrix and the dense tensor. All tests are run using the sparse matrix from a pre-trained `arrowspace` (manifold file and embeddings for the test queries available in [8]).

```rust
// tauattention.rs (struct fields + sparse/dense selection)
pub struct TauModeAttention<B: Backend> {
  // ...
  laplacian_tensor: Option<Param<Tensor<B, 2>>>, // using dense
  laplacian_matrix: Ignored<Option<CsMat<f64>>>, // using sparse
  pub(crate) tau_mode: Ignored<Option<TauMode>>,
```

```
    // ...
  }

  fn lambdas_from_heads_any(&self, heads: Tensor<B, 4>) -> Tensor<B, 3> {
    let tau_cfg = self.get_tau_config();
    if let Some(lap) = self.laplacian_matrix.0.as_ref() {
      let mode =
          self.tau_mode.0.unwrap_or(crate::pretraining::parquet::TauMode::Median);
      crate::taumode::lambdas_from_heads_sparse::<B>(heads, lap, mode,
          tau_cfg.eps)
    } else {
      let lap = self.get_laplacian_tensor().clone();
      crate::taumode::lambdas_from_heads::<B>(heads, lap, &tau_cfg)
    }
  }
```

KV-caching layout KV-cache in `tauGPT` is also different compared to standard GPT: instead of caching K and V, each layer caches $(V, lambda_k)$ only, which matches the fact that scoring uses lambda scalars rather than key vectors. The cache type is defined as *pub type TauCacheLayer<B> = Option<(Tensor<B, 4>, Tensor<B, 3>)>;* and `tauGPT` wraps a vector of these per layer in *TauKVCache store: Vec<TauCacheLayer<B»*, *position: usize* .

```
  // tauattention.rs (KV-cache payload + append logic)
  pub type TauCacheLayer<B> = Option<(Tensor<B, 4>, Tensor<B, 3>)>;

  let lambda_k_new = self.lambdas_from_heads_any(k_new); // [B, Hkv, 1]

  // Cache management
  let (v_full, lambda_k_full) = match cache_layer.take() {
    Some((v_all, lk_all)) => (
    Tensor::cat(vec![v_all, v_new.clone()], 2),              // time axis
    Tensor::cat(vec![lk_all, lambda_k_new.clone()], 2),      // time axis
    ),
    None => (v_new.clone(), lambda_k_new.clone()),
  };

  *cache_layer = Some((v_full.clone(), lambda_k_full.clone()));
  let tk = v_full.dims()[2];
  let y = self.scaled_tau_attention_decode(q, lambda_k_full, v_full, 1, tk);
```

Inside `TauModeAttention::forward_decode(...)`, the cache logic appends along the time axis and then runs attention against the full cached history via `scaled_tau_attention_decode(q, lambda_k_full, v_full, 1, tk)`.

```
  // taugpt.rs (model-level decode uses cache.position for RoPE step
      slicing)
  pub fn forward_decode(
  &self,
  last_ids: Tensor<B, 2, Int>, // [B,1]
  cache: &mut TauKVCache<B>,
  use_softcap: bool,
  ) -> Tensor<B, 3> {
    let tpos = cache.position;
    let d2 = self.cos.dims()[3];

    // Slice RoPE for the current absolute position: [1,1,1,D/2]
    let cos_step = self.cos.clone().slice([0..1, tpos..tpos + 1, 0..1,
        0..d2]);
    let sin_step = self.sin.clone().slice([0..1, tpos..tpos + 1, 0..1,
        0..d2]);
```

```
14
15      for (i, block) in self.blocks.iter().enumerate() {
16        let layer_cache = &mut cache.store[i];
17        x = block.forward_decode(x, (&cos_step, &sin_step), layer_cache);
18      }
19
20      logits
21    }
```

3.5 Summary of similarities and differences

Both `nanoGPT` (`CausalSelfAttention`) and `tauGPT` (`TauModeAttention`) use identical V projection and fetching logic, confirmed by code review and tests.

V projection equivalence **Construction** - both create the same `c_v` linear layer with identical initialization:

```
1     cv: LinearConfig::new(n_embd, n_kv_head * head_dim)
2     .with_bias(false)
3     .with_initializer(KaimingUniform { gain: 0.5, fan_out_only: false })
```

Forward path (prefill) - both follow the exact same pipeline for V:

1. Project: `self.cv.forward(x).clamp(-5.0, 5.0)`
2. Reshape: `.reshape([b, t, self.nkvhead, self.headdim])`
3. Transpose: `v.swap_dims(1, 2)` → `[B, H_kv, T, D]`
4. MQA expand (`if n_head != n_kv_head`): both use `unsqueeze(dim=2).expand(...).reshape(...` to replicate KV heads
5. Weighted sum: `att.matmul(v)` after softmax

Decode path (single-step) - identical logic for computing new V and caching:

- Project step: `self.cv.forward(x_step).clamp(-5.0, 5.0).reshape([b, 1, n_kv_head, head_dim]).swap_dims(1, 2)`
- Append to cache: `Tensor::cat(vec![v_all, v_new.clone()], 2)` (time dimension)
- MQA expansion and weighted sum identical to forward

What differs The only difference is attention scoring:

- `nanoGPT`: $QK^T/\sqrt{d}$ then softmax
- `tauGPT`: lambda-distance logits `tau_distance_logits(lambda_q, lambda_k)` then softmax

After that, both apply the same softmax and the same `att.matmul(v)` to produce output.

Test coverage confirms equivalence Tests verify V is handled identically:

- `test_forward_shape_consistency`: both produce `[B,T,C]` outputs (V contributes correctly)
- `test_decode_single_step`: both cache V with shape `[B, H_kv, T, D]`
- `test_cache_accumulation`: V cache grows identically (time dimension increments)
- `test_mqa_expansion`: both correctly expand V when `n_head != n_kv_head`

The difference in final outputs (`test_tau_vs_causal_outputs_differ_expected`) is purely due to scoring

4 Results

Multiple tests on both attention mechanisms have been run and results have been collected using [8].

Here some data and diagrams from the tests results. All this data was collected running tests on CPU hardware because the objective of this concept architecture's test results are preliminary and to be taken as a reference and a baseline, to demonstrate that Tauformer/tauGPT can work as any other Transformer but with delivering of domain-specific (topological) metadata directly in the attention mechanism and with some performance gains for larger windows and higher dimensional embeddings.

Considering generated tokens, the results contains 500000 decoded-token latency rows spanning both engines ("nano", "tau"), both modes (`kv_cache`, `no_cache`), and `gen_tokens` (context window) $\in 128, 256, 512, 1024, 2048$. Per-token comparisons were computed by matching "nano" and "tau" tokens on `prompt_id`, `mode`, `gen_tokens`, `token_index` and then taking the ratio,

The full normalised token-latency results confirms that `tauGPT` is consistently faster in "kv-cache" mode (a proxy for inference decoding), and consistently slower in "no cache" (training/uncached proxy) for small context windows (up to 2048 tokens); with the gap widening in `tauGPT`'s favour as generation length grows in "kv-cache" while the gap in "no cache" mode gets smaller (see 1). The difference in "no cache" between the two gets smaller as the lengths grow so there is room to make the hypothesis that at some threshold where context window and embeddings lengths are large enough `tauGPT` will be at least as fast as `nanoGPT` in training while being increasingly faster at inference time. Also to consider that `nanoGPT` is a relatively much more mature implementation. This could confirm that Tauformer could be viable also for training on multimodal data where multiple modalities are concatenated in the same embedding space while keeps being faster and more efficient (-50% memory for KV-cache) at inference. The assumption that gains are provided as the context window and the embeddings dimensions grows is for now supported only by regressions run on 0.5M decoded-token rows across hundreds of runs of testing with different lengths. In particular an idea about at which threshold *tauGPT* could reach equivalent performance also in training is hinted in 1, 2, 3.

See table 1 for summary statistics (repository [8] may contain more up-to-date data in the `report/` directory).

TTFT (Time To First Token) is measured differently depending on the caching mode:

- "no_cache" mode: TTFT includes the full forward pass through the entire prompt context, since there's no cached state. The timer starts when processing begins and stops when the first token is generated; `end_to_end_ms = ttft_ms + decode_total_ms`.

- "kv_cache" mode: TTFT measures only the decode step for producing the first new token, after the cache has already been built from the prompt. The prefill/priming time is tracked separately in the `prefill_ms` and `prime_ms columns`; `end_to_end_ms = prefill_ms + ttft_ms + decode_total_ms`.

Table 1: Benchmark Summary Statistics for low dimensional vectors

Throughput (tokens/sec) – Higher is Better

Engine	Mode	Mean	Min	Max
nano	kv_cache	37.47	13.91	59.96
nano	no_cache	4.55	0.89	9.71
tau	kv_cache	49.39	15.04	71.21
tau	no_cache	3.43	0.81	6.91

Per-Token Latency (ms) – Lower is Better

Engine	Mode	Mean	p50	p95
nano	kv_cache	36.25	36.26	41.35
nano	no_cache	327.58	295.88	550.58
tau	kv_cache	30.44	30.42	33.13
tau	no_cache	390.04	359.90	636.52

Speedup: Tau/Nano
(> 1.0 = Tau faster)

Metric	Factor
Tokens/sec	1.068×
TTFT	0.997×
p50 latency	1.066×
p95 latency	1.104×
Decode time	1.067×
Total time	1.028×

4.1 Inference proxy (kv_cache) results

On a matched per-token basis, nano/tau speedup medians (p50) increase with `gen_tokens`: 1.09× (128), 1.12× (256), 1.17× (512), and 1.22× (1024), meaning tau's per-token latency advantage strengthens as sequences get longer. Tail behaviour also improves in tau's favour as sequences grow: the p95 of nano/tau per-token speedup rises from 1.35× (128) to 1.52× (1024), suggesting tau reduces worse-case token stalls relative to nano at longer generations in "kv-cache" mode. Looking specifically at the last 25% of tokens (75–100% bins), the median-latency ratio nano/tau reaches 1.33× at `gen_tokens=1024`, indicating the end of long generations is where tau helps most under `kv_cache`.

4.2 Training/uncached proxy (no_cache) results

For "no cache", matched per-token speedups are ≤ 1 across all lengths (tau slower): median nano/tau speedup is 0.75× (128), 0.75× (256), 0.76× (512), and 0.80× (1024). The late-token-bin summary also stays ≤ 1 for "no cache" (e.g., last-25% median ratio is ~ 0.86 at 1024), but the ratio moves upward with length, which suggests tauGPT's relative penalty shrinks at longer sequences even without caching. In the "no cache" distribution, even the p99 nano/tau speedup stays below 1 ($\simeq 0.95\check{}0.97$), meaning tauGPT is slower almost everywhere token-by-token in this mode rather than losing only in a small tail, the situation improves when the window and embeddings lengths grow. Considering the results of the regressions for larger lengths (context window, embeddings dimensions and prompt length $> 10^5$) there is still a lot of margin to harvest in training also considering that this implementation is at concept stage without any optimisation. Results at 2048 are already better promising even if the number of runs is limited at the moment.

4.3 Matched per-token speedup distribution (nano/tau)

This is the test matrix and the relative results:

mode	gen_tokens	matched token pairs	speedup p50	speedup p95	speedup p99
kv_cache	128	96,768	1.093	1.346	1.437
kv_cache	256	165,888	1.124	1.395	1.494
kv_cache	512	62,208	1.166	1.443	1.557
kv_cache	1024	359,424	1.223	1.523	1.658
no_cache	128	96,768	0.747	0.889	0.965
no_cache	256	165,888	0.749	0.898	0.968
no_cache	512	62,208	0.762	0.901	0.954
no_cache	1024	359,424	0.805	0.924	0.965

Table 2: Matched per-token speedup distribution (nano/tau) across cache modes and sequence lengths (higher numbers mean tau is better).

These diagrams compare the gains as the context window, the prompt length and embeddings dimensions grow up beyond $\sim 10^7$, this is only indicative as for `nanoGPT` is running on CPU and `tauGPT` is unoptimised and a concept model but I think it can provides hints to frame the orders of magnitude at which Tauformer can become performant also in `no_cache` mode or in other scenarios:

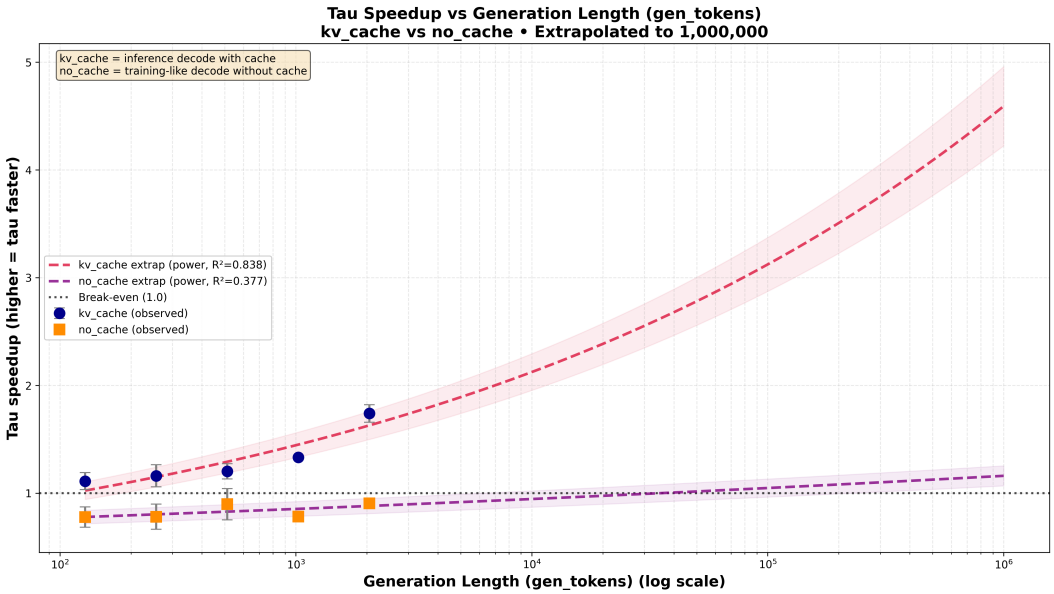

Figure 1: Regressions for tau/nano speedups as the context window grows

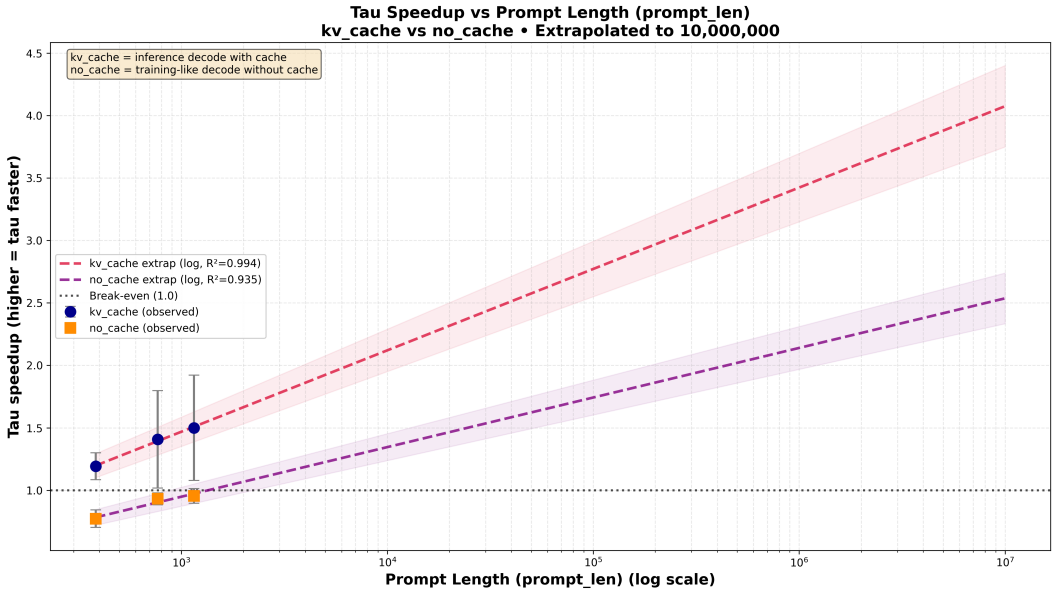

Figure 2: Regressions for tau/nano speedups as the prompt length grows

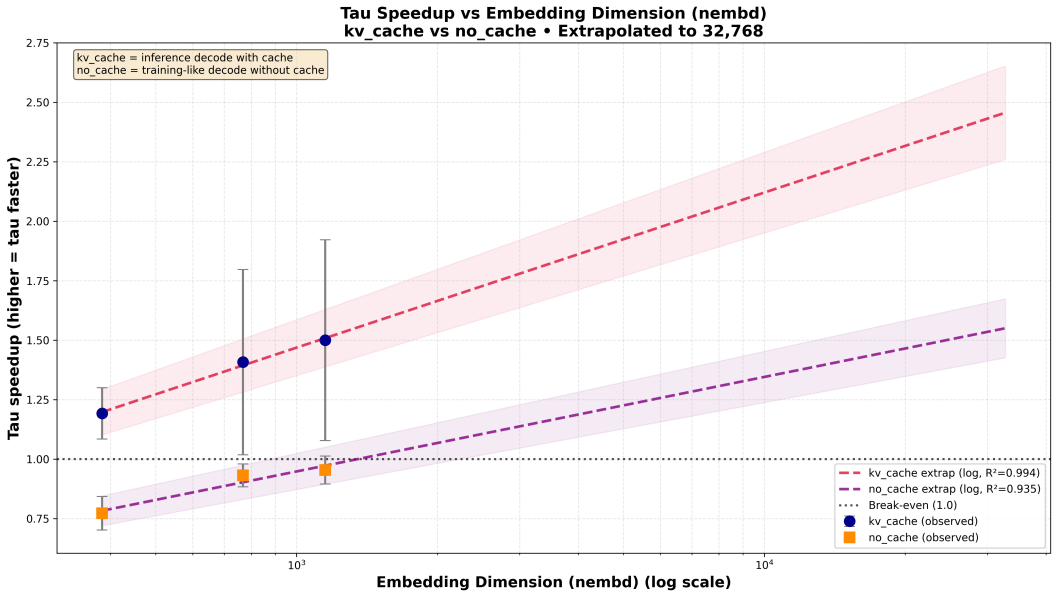

Figure 3: Regressions for tau/nano speedups as embeddings' dimensions grows

5 Conclusion: A Beginning

Potential for larger improvements are tested in larger windows generation tests (256, 512, 1024, ..., 10^6 tokens) and larger embeddings dimensions. The driving hypothesis is that when caching becomes too expensive memory-wise (like for very long windows with high-dimensional embeddings), `tauGPT` can provide a better solution both in terms of contextual accuracy and performance. Current research is exploring at the order of magnitude of 10^4 window length (average for LLMs is 6000 to 14000) with values for embeddings dimensions in the same order of magnitude. The idea is to test to have a prompt length and embeddings length that are of the same order of magnitude of number of non-zero values of the sparse representation in the pre-trained manifold, so that the model can adapt to the number of dimensions of the manifold processing an element in a single query (Q). The prompt length should be obviously modifiable at will.

The normalised token latencies indicate tau's current value proposition is "decode-heavy inference on long windows and high dimensions" with promising performance gains both in a "no cache" and "kv-cache" mode for large scales. If training work is well proxied by `no_cache`, `tauGPT` currently underperforms across essentially the entire per-token distribution on lower scales; so demonstrating training wins require architectural/implementation optimisations, or for the model to be tested on very long context windows and very high embeddings dimensions (as showed in the regression diagrams). The next experiments that would most strengthen Tauformer position should aim to achieve the same results beyond lengths of 10^7 for the H D and T.

Current results demonstrate the promising characteristics of a Topological Transformer architecture, Tauformer; designed to deliver

- *(i)* domain-specific metadata at the level of attention computation and
- *(ii)* make self-attention more convenient at very large "lengths" (sequences, heads, embeddings dimensions) leveraging topological search (using `arrowspace`'s `taumode` vector compression).

To synthesise in few words, Tauformer leverages `taumode` as attention score, allowing to avoid usage of inner-products in the attention computation, potentially making possible very large context windows and to run attention on higher dimensional (multimodal) embeddings with accessible costs and infrastructural requirements on multi nodes hardware.

In the current implementation, `TauModeAttention` still forms attention logits/probabilities over the whole key-time axis and then does the usual weighted sum `att.matmul(v)`. That means decode-time work and intermediate memory scale like $O(T)$ per generated token (and $O(H \cdot T)$ across heads), which becomes prohibitive at $T = 10^7$ unless of a redesign of the attention computation to be streaming/chunked/sparse that is one of the principles stated (to use sparse representation).

To benefit at 10^9 context, Tauformer needs a variant where it does not cache raw per-token V (even compressed) and then multiply by a length-10^9 attention vector. Instead, cache per-centroid aggregates (like S_c, U_c) and compute attention over K centroids (or over a small retrieved subset), so both memory and compute scale with K, not T. This idea brings forward the possibility of compressing the historical V using `taumode` and `gl` for the latent space as well: for example, use the same Laplacian idea already applied to get λ scalars (via Laplacian-based mapping helper) but apply it as a projection of V onto a low-dimensional basis (e.g., first r spectral components), store only coefficients, and reconstruct an approximate V when needed. This is analogous to how `arrowspace` does dimensionality reduction before computing the Laplacian but in the latent/value manifold rather than in token embedding space. This is another potential example of the hypothetical members of a Topological Transfomers' class of models built for very large lengths.

6 Acknowledgments

The author is an independent researcher who self-funded this work. All works available at his research page. Thanks to my supporting network of peers and to all the backers and sponsors to my research effort.

References

[1] Lorenzo Moriondo, *ArrowSpace: introducing Spectral Indexing for vector search*, 2025. https://github.com/tuned-org-uk/arrowspace-rs DOI: 10.21105/joss.09002

[2] Andrej Karpathy *nanoGPT: The simplest, fastest repository for training open-source GPT models*. https://github.com/karpathy/nanoGPT

[3] Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N and Kaiser, Lukasz and Polosukhin, Illia *Attention is All You Need*, Advances in Neural Information Processing Systems (NeurIPS), volume 30, 2017

[4] J. W. Strutt (Lord Rayleigh), *The Theory of Sound*, Vol. 1, London: Macmillan and Co., 1877.

[5] Kexin Wang, Nils Reimers, and Iryna Gurevych. 2021. TSDAE: Using Transformer-based Sequential Denoising Auto-Encoder for Unsupervised Sentence Embedding Learning. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 671–688, Punta Cana, Dominican Republic. Association for Computational Linguistics.

[6] Félix Lefebvre and Gaël Varoquaux. 2025. SEPAL: Scalable Embedding Propagation ALgorithm for large knowledge graphs. In *Proceedings of the 13th International Conference on Learning Representations (ICLR 2025)*, Singapore.

[7] Lorenzo Moriondo (@Mec-iS) Tauformer . https://github.com/tuned-org-uk/taugpt-kvcache-bench

[8] Lorenzo Moriondo (@Mec-iS) KV-cache bench for Tauformer. https://github.com/tuned-org-uk/taugpt-kvcache-bench

[9] Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to Memorize at Test Time. *arXiv preprint arXiv:2501.00663*, 2025.

[10] Lianlei Shan, Shixian Luo, Zezhou Zhu, Yu Yuan, Yong Wu Cognitive Memory in Large Language Models. *arXiv preprint arXiv:2504.02441*, 2025.

[11] Weizhi Wang, Li Dong, Hao Cheng, Xiaodong Liu, Xifeng Yan, Jianfeng Gao, Furu Wei Augmenting Language Models with Long-Term Memory. *arXiv preprint arXiv:2306.07174*, 2024 (Updated).

[12] Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, Yongfeng Zhang A-MEM: Agentic Memory for LLM Agents. *arXiv preprint arXiv:2502.12110*, 2025.

[13] Blog posts at tuned.org.uk. Tuned Blog. www.tuned.org.uk/blog, 2025. Accessed: December 29, 2025.

[14] The Rust Project Developers, *The Rust Programming Language*, 2024. [Online]. Available: www.rust-lang.org

[15] Tracel-AI, *Burn: A Deep Learning Framework Designed from Engineers' Perspectives*, 2025. [Online]. Available: https://github.com/tracel-ai/burn