AttnCache: Accelerating Self-Attention Inference on Big Memory Systems Using Attention Cache

Anonymous ACL submission

Abstract

Self attention, the kernel of transformer models, is computationally intensive, and hence a major focus for accelerating large language model (LLM) inference. Existing methods on transformer inference acceleration often require modifying transformer architectures or using specialized hardware accelerators, which limits their broad applicability. In this paper, we introduce an innovative method, called AttnCache, to accelerate self attention inference in LLM prefill phase without the above limitations. AttnCache draws inspiration from the intriguing observation of recurring and rich similarities in attention computations across different inference sequences. Based on a memorization database that leverages emerging big memory systems, we propose embedding and efficient caching techniques to identify inputs that produce similar attention maps, thereby reducing computation overhead. Experimental results show that AttnCache achieves 1.2X speedup on average on Semantic Textual Similarity (STS) benchmarks, with only 2% performance loss.

1 Introduction

Transformer-based Large Language Models (LLMs) provide superior inference accuracy and throughput. Central to this success is the highly parallelized self-attention mechanism, which enables Transformer to capture dependencies and relationships across different positions within a sequence. Attention maps, generated in the self-attention mechanism, represent the relevance and importance of each position to other positions. However, computational intensity of this mechanism poses a significant bottleneck, especially as model sizes and input sequences grow.

Various techniques have been proposed to accelerate self-attention inference through computation reduction. Token pruning (Ham et al., 2020; Wang et al., 2021) reduces computation by excluding less important tokens from the input, while layer-wise reuse (Ying et al., 2021; Xiao et al., 2019; Bhojanapalli et al., 2021) reduces computation by sharing attention maps calculated in prior layers in multiple subsequent layers. Despite their effectiveness in accelerating self-attention inference, these techniques often cause significant loss in model accuracy, especially in complex tasks that require full-contextual information. Other acceleration techniques, such as introducing sparsity into self-attention (Lu et al., 2022; Kitaev et al., 2020) or removing the attention or transformer layer (He et al., 2024; Men et al., 2024; Song et al., 2024; Zhang et al., 2024b), require specialized hardware accelerators (Yang et al., 2020) or model architecture changes, restricting their general applicability.

In this paper, we find that semantically different input sentences can have high similarity in their attention maps at different layers or different heads during the inference computation. By pre-storing (or caching) these similar attention maps into a database utilizing the emerging big memory system (called attention maps database), we can save self-attention computation using the attention cache to retrieve similar attention maps. For example, as shown in Figure 1, there are two different sentences. The sentence 1 is *"This sentence: 'you should never do it.' means in one word:"*. The sentence 2 is *"This sentence: 'how do you do that?' means in one word:"*. Although the two sentences have different semantics, their attention maps at different layers and different heads are very similar, which indicate they have similar relevance at each token position. Thus, we can reuse the attention maps of all layers computed by the sentence 1 for the sentence 2.

Our study is driven by the recent development of memory technologies (e.g., Compute Express Link (Sharma et al., 2023) or persistent memory (Izraelevitz et al., 2019)) that enable big memory systems at large scales (e.g., terabyte or

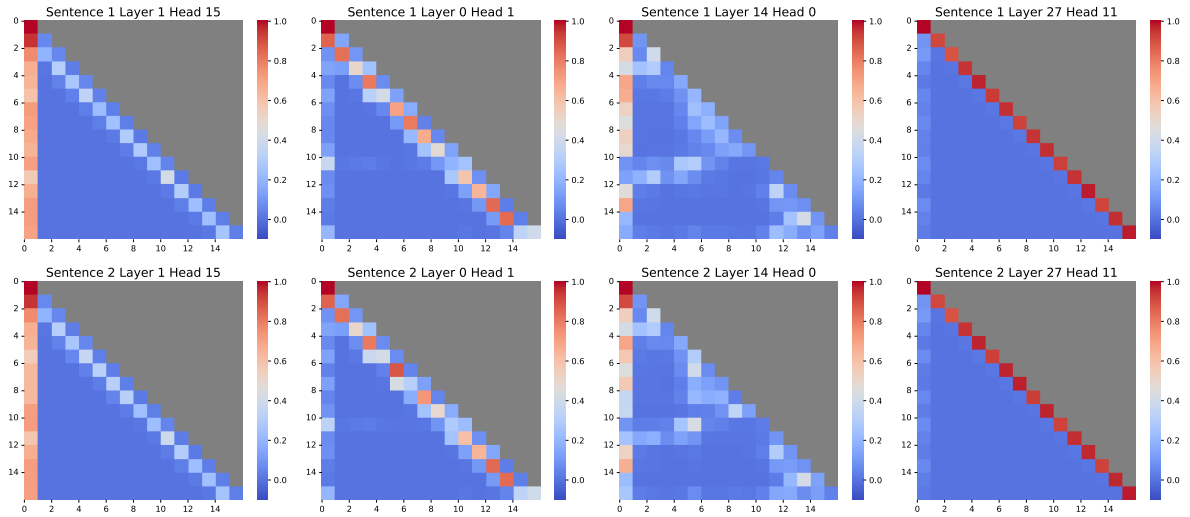

Figure 1: Visualization of the attention maps in Llama-3.2-3B over two sentences, each with a length of 32. The plots reveals that although Sentence 1 and Sentence 2 have different meanings, their attention maps at different layers and different heads are similar.

petabyte scale) (Mellor; MemVerge). The big memory system not only supports memory-intensive applications but also opens up possibilities for new programming and computational paradigms (Xie et al., 2021; Mansi and Swift, 2020). Specifically, we leverage increased memory capacity to accelerate self-attention inference through caching (or memoization). Memoization brings speedup unattainable by traditional system-level optimizations (e.g., vectorization and tiling (Dao et al., 2022; Dao, 2023)). We introduce a framework to accelerate self attention on big memory systems using memoization, called *AttnCache*.

The implementation of AttnCache poses several challenges. The first challenge lies in finding a proper data representation. Both the representation of input sentence and attention maps in transformer are high-dimensional tensors. It is challenging to find similar attention maps by directly comparing the representation of the input sentences. To identify similar attention maps, we design a proper data representation by embedding network. To reduce time in self attention, the embedding network must be lightweight such that its overhead plus the search in the attention maps database (a key-value store) is smaller than the cost of self-attention computation.

The second challenge is expensive memory accesses for storing and fetching pre-populated attention maps. To improve the search hit rate and use memory capacity offered by the big memory system, the attention maps database must be big and pre-populated with attention maps. However, the large key-value search in the database leads to highly sparse memory accesses. Besides, modern deep learning frameworks like PyTorch require the tensors to be placed in consecutive memory addresses for SIMD operations. Therefore, once the tensors are fetched from the pre-populated database, the tensors must be copied to a consecutive memory space and then loaded to the processor registers to be used by the attention function. As a result, one tensor fetch generates two memory reads and one write, which deter the time benefit of using caching. To reduce memory access overhead, we store the attention maps of a layer as a file object, and the file objects of the neighboring layers are stored continuously in the database, so as to utilize spatial and temporal locality of memory accesses. Moreover, we use a contiguous virtual memory space to store the references or pointers to the file objects. As a result, retrieving attention maps from the cache is a matter of pointer manipulation rather than of causing a real memory copy.

We use CPU for evaluation, because some scenarios make LLM embeddings of large text corpus, and GPU cannot bring enough memory capacity. For example, the recommendation/RAG systems get representations of billions of text pieces. Such scenarios do not need to get LLM embeddings on a single instance fast, but must compute on a lot of instances fast. In these scenarios, using CPU machines to compute is both time and energy efficient than GPU machines. We use CPU machines to demonstrate our idea in AttnCache. Extensive

experiments show that AttnCache with transformer-based LLM, such as Llama-3-8B, Llama-2-7B, and Mistral-7B, on seven STS tasks enables 1.2× speedup on average with 2% loss in the Spearman correlation score.

2 Related Work

Reuse mechanism in Neural Networks. The reuse mechanism is based on the wide existence of redundancy in neural networks. Some efforts (Ning et al., 2019; Ning and Shen, 2019; Wu et al., 2022; Köpüklü et al., 2019) reuse the similar data results and computation processes to improve performance. Silfa et al. (2019) accelerates RNN training by reusing the output of neurons. Previous works (Bhojanapalli et al., 2021; Xiao et al., 2019) have shown that attention maps of transformer (Vaswani et al., 2017) exhibit similar distributions across adjacent layers. Many previous efforts (Hunter et al., 2023; Xiao et al., 2019; Bhojanapalli et al., 2021; Ying et al., 2021; Liao and Vargas, 2024) focus on sharing the computed attention weights across multiple layers for the same input sequence, which may introduce dissimilar attention maps, thereby degrading performance. In this work, we concentrate on efficiently reusing similar attention maps across different sequences.

Sentence Embedding. Sentence embeddings encodes the semantic information of sentences into high-dimensional vector representations that are broadly applicable to language processing tasks. Prior works (Li and Zhou, 2024; Muennighoff et al., 2024; Ni et al., 2021) have demonstrated the potential of LLMs to generate high-quality sentence embedding. For example, Sentence-BERT (Reimers, 2019) employs contrastive learning to create embeddings by leveraging natural language inference datasets to construct positive and negative pairs. Recent studies (Zhuang et al., 2024; Qin et al., 2023; Zhang et al., 2024a) have focused on converting an LLM directly into a sentence encoder without training. To enhance the quality of embeddings, prompt-based techniques have become increasingly popular. MetaEOL (Lei et al., 2024) uses multitask prompts to generate general-purpose embeddings. The research by (Jiang et al., 2023) illustrates how to extract a sentence embedding by prompting the LLMs with the instruction *"This sentence: '[text]' means in one word:"*. In this paper, we use the LLMs to generate sentence embeddings without the need for LLMs fine-tuning.

3 Methodology

Figure 2 draws an overview of AttnCache. Given an input sentence, AttnCache embeds it into a feature vector using a neural network (feature projector). The feature vector is used to retrieve the index of the attention maps that have the highest similarity to the input sentence. Then, the search engine uses the index to fetch the corresponding attention maps from the attention maps database. The fetched attention maps are used in the self-attention computation during online inference, while the prefill stage (the initial processing of the input sequence) in LLMs inference is utilized to generate the sentence embedding.

3.1 Search Engine

As illustrated in Figure 1, two sentences with very different semantics and meaning could have similar attention maps. The input sentences are represented by high-dimensional hidden states, and it is difficult to determine whether their corresponding attention maps are similar by comparing the input hidden states. Rather than directly using the input embedding as a key to find an attention map, AttnCache uses its feature vector, which is embedded by the feature projector and has a lower dimension. We collect the input embeddings of input sequences and their corresponding attention maps at each layer, which are used for training of the feature projector.

Feature Projector. To quantify the similarity of input embeddings, we use a feature projector, which is an embedding network. Two input embeddings are matched during a search if their feature vectors are similar (in terms of the similarity score defined later). The feature vector is essentially an internal representation of the input embedding to capture similarity, and the feature projector learns this representation through training such that the input embeddings with similar attention maps have similar feature vectors. By searching for similar feature vectors, we are able to find input embeddings producing similar attention maps. Besides, the feature projector allows us to map input embeddings into a lower-dimensional representation, thus reducing the search space and computation complexity of measuring the similarity.

We use two layers of Multi-Layer Perceptron (MLP) as the feature projector, which maps the input embedding to a feature vector with lower dimension size. Compared with other embedding

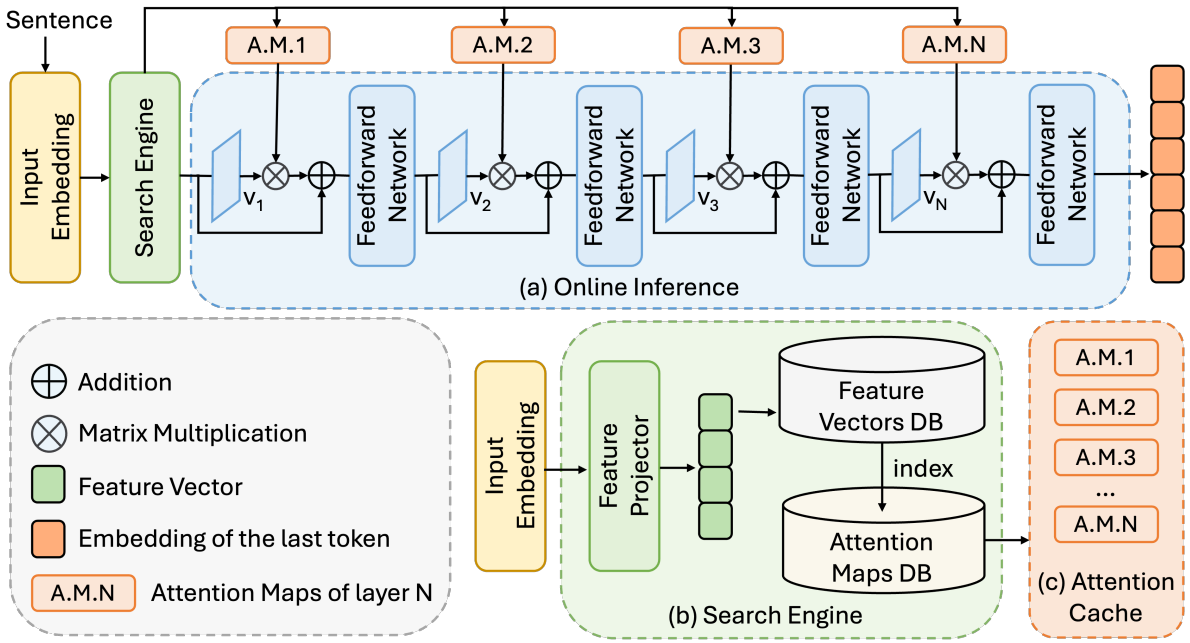

Figure 2: AttnCache overview. The search engine will identify the index of the sentence that produces the most similar attention maps based on the feature vector of the current input sentence and fetch attention maps for each layer from the attention maps database using the index. These fetched attention maps are stored in the attention cache and reused for the matrix multiplication calculation with value projection of the current input sentence.

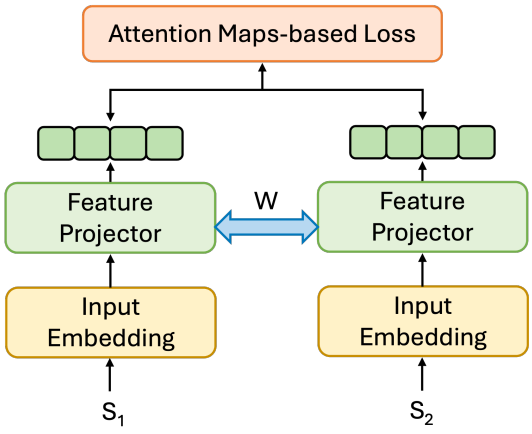

Figure 3: The training of the feature projector. The feature projector maps input embedding of a sentence **S** into a feature vector. Then we train the feature projector using the attention maps-based loss.

models, such as convolutional neural network or transformer, MLP is lightweight with less computational complexity and shorter inference time.

Training the feature projector is challenging due to a lack of labeled data. Deciding the similarity between input embeddings and labeling them as similar or not is prohibitively expensive. We use the Siamese network (Koch et al., 2015), which contains two identical embedding networks and shares the same weights, as shown in Figure 3. Once the Siamese network finishes training, it is used as a feature projector. The Siamese network is trained to minimize the distance between feature vectors whose attention maps have high similarity.

During each training iteration, two input embeddings are used as input to the two identical feature projectors in the Siamese network. After getting the feature vectors, the Euclidean distance (i.e. L2-norm) is calculated as follows.

$$\hat{y} = ||f_{\mathbf{W}}(\mathbf{X}_1) - f_{\mathbf{W}}(\mathbf{X}_2))||_2 \quad (1)$$

where $\mathbf{X}$ is the input embedding, $f_{\mathbf{W}}$ is the feature projector, and $||.||_2$ is the L2 norm. Besides, we measure the similarity score using the attention maps and the sequence length of tokens, which associate with the two input embeddings. We use a metric as the labels for training the feature projector based on the average distance of heads, which is defined as follows.

$$y = \frac{1}{n} \times \alpha \sum_{p=1}^{n} \frac{1}{2} ||\mathbf{A}_1[p,:] - \mathbf{A}_2[p,:]||_2 + ||s_1 - s_2||_1 \quad (2)$$

where $\mathbf{A}$ denotes the attention map, n indicates the number of head, $\mathbf{A}[p,:]$ is the p^{th} row of the attention map, $||.||_1$ is the L1 norm, s denotes the length of input token sequence, and α is the hyperparameter to control the relative importance of the similarity of the attention maps and the token length.

In addition to the inherent similarity of the attention maps, the sentence token sequence also plays an important role in determining whether two attention maps are similar. When the token sequences of two attention maps are very different in length, even if the attention maps are similar, they cannot be used directly in AttnCache for subsequent reuse, because that causes a large inference error. The final loss function of the feature projector is defined as follows.

$$L = \begin{cases} 0.5(\hat{y} - y)^2 & \text{if } |\hat{y} - y| < 1 \\ |\hat{y} - y| - 0.5 & \text{if } |\hat{y} - y| \geq 1 \end{cases} \quad (3)$$

We use Smooth L1 Loss as the loss function, which is able to balance the effects of outliers. The training process iteratively updates the parameters of the feature projector to minimize the loss function. Using the training process described above eliminate the need for labeling input embeddings and attention maps manually.

Database. To minimize the costly search for attention maps, we construct an indexed database, where feature vectors are stored and indexed for fast search. In essence, the feature vector database is a key-value store where the key and value are the feature vector and its index, respectively. The attention maps associated with feature vectors are stored in the attention maps database. The feature vector and associated attention maps have the same index in the two databases. The attention maps database is also a key-value store where the key is the index retrieved from the feature vector database, and the value is the attention map.

Given an input (i.e., the hidden state of the input query to LLM), the feature vector database uses an approximate nearest-neighbor search algorithm to find "similar" feature vectors. The similarity metric is defined in Equation 1. Only when the similarity between the input feature vector and a stored feature vector is larger than a pre-specified threshold θ, the index of the stored feature vector is returned.

Algorithm 1 depicts the search engine. The input sentence is embedded by input embedding (Line 2). The input embedding includes tokenization of the sentence, position encoding, and layer normalization. Then the result is mapped into a feature vector with lower dimension (Line 3). The feature vector is used for querying the feature vector database. After the query, the indices that have the closest similarity to the feature vectors are returned (Line 4). When the similarity is not less than the threshold θ, the corresponding index idx is used to fetch attention maps from the attention maps database.

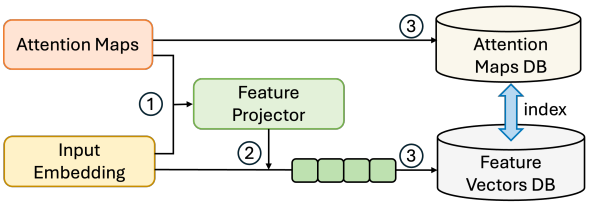

Figure 4: Database building includes three steps. 1. Train the feature projector with input embedding and attention maps; 2. Embed the input embeddings to feature vectors; 3. Store the feature vectors and attention maps to their respective databases. Both databases share the same index.

Algorithm 1: Search Engine

Input: Sentence S, Threshold θ;
Output: Attention Cache $attn\_cache$, Input embedding h;

1 **Function** `search_engine`(S, θ)
2 $h \leftarrow$ input_embedding(S);
3 $f \leftarrow$feature_projector(h);
4 $idx, sims \leftarrow$VecDB.search(f);
5 $attn\_cache \leftarrow []$;
6 **if** $sims \geq \theta$ **then**
7 $n \leftarrow num\_layers$;
8 $ams \leftarrow$ AttnMapsDB.get(idx, n));
9 $attn\_cache.append(ams)$;
10 **end**
11 **return** $attn\_cache$, h

The index idx corresponds to a sentence S that produces similar attention maps to the input sentence. The fetched attention maps are used from the first to the last layer of the LLM, and are stored into contiguous memory space called attention cache. Specifically, these attention maps are used in the matrix multiplication calculation with value projection in `online_inference`. All layers of the attention maps are fetched for the computation of self-attention before the LLM inference starts.

3.2 Online Inference

Algorithm 2 illustrates online inference with AttnCache. In the attention block of each layer, the value projection is computed. If the similar attention maps are found, the attention output can be obtained by multiplying attention maps with value v. Thus finding similar attention maps and reusing them in the self-attention calculation lead to perfor-

Algorithm 2: Online Inference

Input: Attention Cache $attn\_cache$, Input embedding h;
Output: Hidden state of last layer h;

Function `online_inference(`$attn\_cache$, h`)`
  for l *in range(*num_layers*)* **do**
    $residual \leftarrow h$;
    $v \leftarrow$ v_projection(h) ;
    if *attn_cache is not NULL* **then**
      $attn\_map \leftarrow attn\_cache[l]$;
      $h \leftarrow$ mat_mul($attn\_map, v$);
    end
    else
      $q \leftarrow$ q_projection(h) ;
      $k \leftarrow$ k_projection(h) ;
      $q, k \leftarrow$ rotary_pos_emb(q, k) ;
      $attn\_map \leftarrow$ softmax(q, k) ;
      $h \leftarrow$ mat_mul($attn\_map, v$) ;
    end
    $h \leftarrow residual + h$;
    $h \leftarrow h +$ feed_forward(h) ;
  end
return h;

mance benefits.

However, AttnCache cannot always find similar attention maps. For those hidden states with low similarities, the attention maps must be calculated at each layer during the inference, which means the query, key, rotary positional embedding, and softmax normalization must be computed. In this regard, AttnCache does not bring benefit in inference speed, and instead degrades performance due to its search overhead. However, given a batch of inferences, as long as the success rate during the search for all inferences is high, the overall inference is still accelerated.

4 Experiments

4.1 Setting

We evaluate AttnCache on a server equipped with two sockets, each with 24 cores Intel(R) Xeon(R) Silver 4410Y processors. The platform provides 512 GB DRAM and 14 TB Hard Disk Drive (HDD). We use the DRAM to store the attention maps database and feature vector database. To build the feature vector database, we use Faiss (Johnson et al., 2019), a vector database enabling efficient similarity search by the Hierarchical Navigable Small Worlds algorithm (Malkov and Yashunin, 2018). Faiss is highly efficient for similarity search. For example, our evaluation shows that searching 100K vectors with a dimension size of 128 takes less than 0.5 ms on the DRAM. As a result, the search process does not create a performance bottleneck for AttnCache. In addition, we save attention maps in a layer as a file object in the attention database, and assume that both the attention maps database and feature vector database are held in the DRAM. For the case that the two databases are larger than the real DRAM in our platform, we evaluate the model performance using both the DRAM and HDD, which does not impact the model quality. In this case, to evaluate the model inference time on a "virtual" big DRAM system with enough capacity, we use our limited DRAM assuming that the needed attention maps are in the DRAM. In such a case, the search overhead (not including the feature projector overhead) is ignored, because the search time on the DRAM is only a small portion (at most 2%) of total inference time due to highly optimized Faiss. We implement AttnCache with PyTorch 2.5.1. In the evaluation, all 48 CPU cores are fully utilized for maximum thread-level parallelism to minimize inference time. We use the original transformer model as the baseline, named *full model.*

4.2 Datasets and Models

We evaluate AttnCache with Llama-2-7B, Llama-3-8B and Mistral-7B on seven semantic textual similarity (STS) datasets, utilizing the SentEval toolkit (Conneau and Kiela, 2018). The STS datasets include STS 12-16, STS-B and SICK-R (Lei et al., 2024). The semantic similarity of each sentence pair in each dataset is annotated with a score of 0-5. For each sentence pair, we employ cosine similarity to measure the similarity between sentence embeddings (i.e., the LLM outputs). We use the Spearman correlation between the human annotated similarity score and the cosine similarity score, as the evaluation metric. The Spearman correlation is computed under the "all" setting.

4.3 Baselines

We use three baselines for evaluation.

LazyFormer (Ying et al., 2021) divides all layers of the transformer to multiple subblocks. In each subblock, the attention maps are only computed in the first layer and then used by the remaining layers in the same subblock. Like LazyFormer,

we set the number of layers in each sub-block to 2.

SAN (Xiao et al., 2019), like LazyFormer, shares attention maps across multiple adjacent layers. But different from Lazyformer, SAN does not use a uniform subblock size (i.e., the number of transformer layers in a subblock). The subblock size is dynamically determined based on the similarity of layers in terms of the JS divergence (Menéndez et al., 1997).

AttnCache-f is a variant of AttnCache. AttnCache-f applies memoization at the transformer layer level instead of the whole model level as AttnCache. In particular, at each layer, AttnCache-f searches the attention maps database for similar attention maps, hence applying a *f*ine-grained memoization. Moreover, AttnCache-f does not consider sequence length when training the feature projector, which means that the y in Equation 2 does not take the computation of $||s_1 - s_2||_1$ into account.

4.4 Implementation Details

We use test samples in STS datasets to evaluate AttnCache. We apply 8-bit (int8) quantization to LLM weights in order to save memory space. For each task, we collect attention maps and input embeddings from 1K sentences to train feature projector and build databases; we use 1000 samples to measure the inference time of self-attention; we use the remaining samples for testing. Across the seven datasets, we build the attention maps database using 7K sentences and use 7K test samples for speed measurement. To maintain high inference accuracy, we set the similarity threshold θ to 0.99, and set α, which is used to train the feature projectors (see Equation 2), to 0.2. We use the Speedup Degradation Ratio (He et al., 2024) γ to quantify trade-off between speed and performance degradation.

$$\gamma = \frac{\text{Avg}_{full} - \text{Avg}_{method}}{\text{Speedup}_{method} - \text{Speedup}_{full}}$$

where Avg_{full} and Avg_{method} are the average performance of LLM and each method across the seven tasks respectively, and Speedup_{full} and Speedup_{method} represent the corresponding speedup respectively. A smaller γ indicates that the method is more efficient.

4.5 Main Results

Table 1 summarizes the results. Across various models (Llama2-7B, Llama3-8B and Mistral-7B), SAN and LazyFormer both lead to notable performance declines, despite achieving higher speedups. For instance, LazyFormer results in an average 25.39% performance decline (from 71.88% to 46.49%) for Llama-3-8B, with a speedup of 1.42×, corresponding to a γ of 0.60. We also notice that the inter-sentence methods (i.e. AttnCache-f and AttnCache) exhibit higher performance but lower speedup compared to intra-sentence methods (i.e. SAN and LazyFormer) because they only reuse attention maps with high similarity. For example, for Llama-2-7B, AttnCache-f and AttnCache achieve 50.39% and 67.75% average performance with 1.14× and 1.19× speedup, while SAN and LazyFormer have 30.34% and 34.48% performance with 1.39× and 1.45× speedup separately. Moreover, AttnCache maintains near full model performance on various datasets and strikes a better balance between speed and performance, with γ values of 0.04, 0.11 and 0.09 for three LLMs, making it a superior method for the acceleration of self attention.

5 Analysis

5.1 Impacts of Model Quantinization and Pruning

The model quantization represents weights and activations with lower-precision data type, and can improve efficiency in memory usage and inference speed. We integrate AttnCache with quantization, and study whether AttnCache can maintain the performance. Specifically, we apply Quanto (Optimum, 2024) to all weights, and use 4-bit quantization. We also combine AttnCache with recent LLM pruning methods, AttnDrop and BlockDrop (He et al., 2024), which remove redundant attentions and layers by measuring the similarity between input and output of each layer. Table 2 shows the results. The integration of model quantization and pruning with AttnCache maintains performance: the difference between AttnCache and Quanto/BlockDrop is only 1%, and the difference between AttnCache and AttnDrop is only 2%, on average.

5.2 Impact of Similarity Thresholds

Assume that there are N input sentences for an LLM to generate sentence embeddings, we count how many times AttnCache is successfully applied (indicating similar attention maps are found), denoted as M. We use the ratio M/N as the hit rate.

We randomly select 100 sentences from STS15,

Table 1: **Spearman correlation score (in %) across 7 STS tasks**

Llama-2-7B										
Method	STS12	STS13	STS14	STS15	STS16	STS-B	SICK-R	Avg. (↑)	SpeedUp (↑)	γ (↓)
Full Model	60.88	73.93	58.30	70.27	75.46	73.89	67.44	68.60	1.00×	–
SAN	5.02	42.63	19.84	43.49	44.70	18.01	38.71	30.34	1.45×	0.85
LazyFormer	23.79	34.88	27.80	35.93	44.04	32.50	42.45	34.48	1.39×	0.87
AttnCache-f	22.06	67.75	31.52	61.15	53.89	53.97	62.40	50.39	1.14×	1.30
AttnCache	60.59	73.46	57.97	69.01	75.38	72.02	65.85	67.75	1.19×	0.04
Llama-3-8B										
Method	STS12	STS13	STS14	STS15	STS16	STS-B	SICK-R	Avg. (↑)	SpeedUp (↑)	γ (↓)
Full Model	61.57	76.41	63.23	75.27	80.41	75.84	70.45	71.88	1.00×	–
SAN	27.61	53.81	37.18	57.20	57.43	39.46	54.98	46.81	1.49×	0.51
LazyFormer	27.25	60.37	36.21	53.85	59.21	40.30	48.24	46.49	1.42×	0.60
AttnCache-f	24.89	51.15	36.19	67.81	61.39	48.05	63.77	50.46	1.16×	1.34
AttnCache	60.82	72.49	60.59	74.67	79.52	72.61	66.68	69.63	1.21×	0.11
Mistral-7B										
Method	STS12	STS13	STS14	STS15	STS16	STS-B	SICK-R	Avg. (↑)	SpeedUp (↑)	γ (↓)
Full Model	63.28	74.89	61.57	75.64	81.89	78.26	69.39	72.13	1.00×	–
SAN	25.04	54.66	35.30	53.11	61.55	39.59	55.45	46.39	1.44×	0.58
LazyFormer	38.90	54.41	38.71	37.18	57.61	42.23	50.66	45.67	1.38×	0.70
AttnCache-f	35.03	55.07	40.28	54.51	50.22	54.75	64.52	50.63	1.15×	1.43
AttnCache	62.66	72.23	61.85	73.32	81.59	74.66	65.89	70.31	1.20×	0.09

Table 2: **Integration with model Quantization and Pruning**. "w/Quant" denotes integration with 4-bit quantized model. "w/AttnDrop" and "w/BlockDrop" represents integration with attention pruning and layer pruning repectively.

Llama-3.2-3B					
Method	STS13	STS14	STS15	STS16	Avg.
Full Model	76.56	60.05	74.76	79.30	72.67
AttnCache	74.74	59.95	74.19	77.38	71.57
Quanto	75.27	57.55	74.41	76.96	71.05
w/Quanto	74.25	54.75	74.49	76.92	70.10
AttnDrop	75.33	59.04	69.92	78.37	70.67
w/AttnDrop	73.21	56.01	69.48	75.49	68.55
BlockDrop	67.98	50.44	72.42	75.52	66.59
w/BlockDrop	67.18	50.49	70.44	73.67	65.45

and change the similarity threshold θ from 0.995 to 0.85. We measure the hit rate and loss in the Spearman correlation score. Figure 5 shows the results. When we reduce θ, the hit rate increases, which means that more attention maps are found and AttnCache leads to higher acceleration. However, this might lead to replacement with less similarity, decreasing the Spearman correlation score. By setting θ to 0.99, our results show that AttnCache provides 30% hit rate with only 2% reduction in the Spearman correlation score.

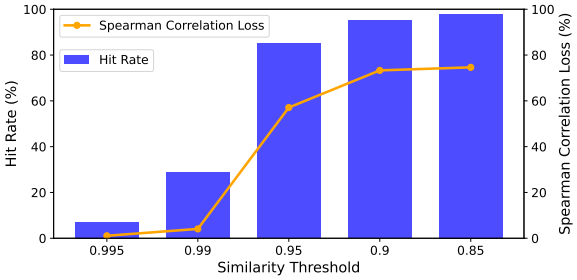

Figure 5: Impact of Threshold on Spearman Correlation

6 Conclusions

The emerging big memory system brings new optimization opportunities for the acceleration of LLMs. In this paper, we propose AttnCache to accelerate self attention on big memory systems. Our work is based on the observation that semantically different input sentences can have high similar attention maps at different layers or different heads during inference computation. By pre-storing similar attention maps into a database, we save self attention computation using a memory cache to retrieve similar attention maps, which are reused in the computation of self attention. AttnCache brings 1.2× speedup on average with negligible performance loss in the Spearman correlation score.

Limitations

AttnCache requires preloading attention maps datasets into the big memory systems. When new reusable attention maps need to be added, the feature projector needs to be retrained. Therefore, to improve the hit rate of attention maps reuse, incremental training is essential while expanding the attention maps database.

Emerging CXL memory expansion and memory pooling easily provide memory capacity at TB scale (Mellor; Petrucci et al.) (or even PB scale (MemVerge)), and meet the needs of building the databases. In addition, the performance of NVMe ultra-low latency (ULL) SSD (Jo, 2024) (e.g., Optane SSD 800p and Z-SSD) is close to that of existing big memory solutions, while providing TB-scale (or PB-scale (Samsung)) capacity.

Ethics Statement

In this paper, we rigorously follow ethical guidelines by solely relying on open-source datasets and leveraging models that are either open-source or widely accepted within the scientific community. Our approach underscores a commitment to maintaining ethical standards, emphasizing transparency, and fostering the responsible application of technology to benefit society.

References

Srinadh Bhojanapalli, Ayan Chakrabarti, Andreas Veit, Michal Lukasik, Himanshu Jain, Frederick Liu, Yin-Wen Chang, and Sanjiv Kumar. 2021. Leveraging redundancy in attention with reuse transformers. *arXiv preprint arXiv:2110.06821*.

Alexis Conneau and Douwe Kiela. 2018. Senteval: An evaluation toolkit for universal sentence representations. *arXiv preprint arXiv:1803.05449*.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness.

Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W Lee, et al. 2020. Aˆ 3: Accelerating attention mechanisms in neural networks with approximation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 328–341. IEEE.

Shwai He, Guoheng Sun, Zheyu Shen, and Ang Li. 2024. What matters in transformers? not all attention is needed. *arXiv preprint arXiv:2406.15786*.

Rosco Hunter, Łukasz Dudziak, Mohamed S Abdelfattah, Abhinav Mehrotra, Sourav Bhattacharya, and Hongkai Wen. 2023. Fast inference through the reuse of attention maps in diffusion models. *arXiv preprint arXiv:2401.01008*.

Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714.

Ting Jiang, Shaohan Huang, Zhongzhi Luan, Deqing Wang, and Fuzhen Zhuang. 2023. Scaling sentence embeddings with large language models. *arXiv preprint arXiv:2307.16645*.

Insoon Jo. 2024. Toward Ultra-Low Latency SSDs: Analyzing the Impact on Data-Intensive Workloads. *Electronics*, 13(1).

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*.

Gregory Koch, Richard Zemel, Ruslan Salakhutdinov, et al. 2015. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, volume 2, pages 1–30. Lille.

Okan Köpüklü, Maryam Babaee, Stefan Hörmann, and Gerhard Rigoll. 2019. Convolutional neural networks with layer reuse. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 345–349. IEEE.

Yibin Lei, Di Wu, Tianyi Zhou, Tao Shen, Yu Cao, Chongyang Tao, and Andrew Yates. 2024. Meta-task prompting elicits embedding from large language models. *arXiv preprint arXiv:2402.18458*.

Ziyue Li and Tianyi Zhou. 2024. Your mixture-of-experts llm is secretly an embedding model for free. *arXiv preprint arXiv:2410.10814*.

Bingli Liao and Danilo Vasconcellos Vargas. 2024. Beyond kv caching: Shared attention for efficient llms. *arXiv preprint arXiv:2407.12866*.

Pan Lu, Liang Qiu, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, Tanmay Rajpurohit, Peter Clark, and Ashwin Kalyan. 2022. Dynamic prompt learning via policy gradient for semi-structured mathematical reasoning. *arXiv preprint arXiv:2209.14610*.

Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836.

Mark Mansi and Michael M Swift. 2020. sim: Preparing system software for a world with terabyte-scale memories. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 267–282.

Chris Mellor. CXL Memory Pools: Just How Big Can They Be? https://blocksandfiles.com/2022/07/07/cxl-memory-pools-size/.

MemVerge. MemVerge Announces Memory Machine Cloud Edition and Memory Viewer to Usher in the Era of CXL. https://memverge.com/memverge-announces-memory-machine-cloud-edition-and-memory-viewer-to-usher-in-the-era-of-cxl/.

Xin Men, Mingyu Xu, Qingyu Zhang, Bingning Wang, Hongyu Lin, Yaojie Lu, Xianpei Han, and Weipeng Chen. 2024. Shortgpt: Layers in large language models are more redundant than you expect. *arXiv preprint arXiv:2403.03853*.

María Luisa Menéndez, JA Pardo, L Pardo, and MC Pardo. 1997. The jensen-shannon divergence. *Journal of the Franklin Institute*, 334(2):307–318.

Niklas Muennighoff, Hongjin Su, Liang Wang, Nan Yang, Furu Wei, Tao Yu, Amanpreet Singh, and Douwe Kiela. 2024. Generative representational instruction tuning. *arXiv preprint arXiv:2402.09906*.

Jianmo Ni, Gustavo Hernandez Abrego, Noah Constant, Ji Ma, Keith B Hall, Daniel Cer, and Yinfei Yang. 2021. Sentence-t5: Scalable sentence encoders from pre-trained text-to-text models. *arXiv preprint arXiv:2108.08877*.

Lin Ning, Hui Guan, and Xipeng Shen. 2019. Adaptive deep reuse: Accelerating cnn training on the fly. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1538–1549. IEEE.

Lin Ning and Xipeng Shen. 2019. Deep reuse: Streamline cnn inference on the fly via coarse-grained computation reuse. In *Proceedings of the ACM International Conference on Supercomputing*, pages 438–448.

Optimum. 2024. Optimum-quanto.

Vinicius Petrucci, Eishan Mirakhur, Nikesh Agarwal, Su Wei Lim, Vishal Tanna, Rita Gupta, and Mahesh Wagh. CXL Memory Expansion: A Closer Look on Actual Platform. https://www.micron.com/content/dam/micron/global/public/products/white-paper/cxl-memory-expansion-a-close-look-on-actual-platform.pdf.

Zhen Qin, Rolf Jagerman, Kai Hui, Honglei Zhuang, Junru Wu, Le Yan, Jiaming Shen, Tianqi Liu, Jialu Liu, Donald Metzler, et al. 2023. Large language models are effective text rankers with pairwise ranking prompting. *arXiv preprint arXiv:2306.17563*.

N Reimers. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*.

Samsung. Samsung Announces 256TB SSDs and Unveils Peta-Byte Scale PBSSDs. https://www.tomshardware.com/news/samsung-announces-256tb-ssds-and-unveils-peta-byte-scale-pbssds.

Debendra Das Sharma, Robert Blankenship, and Daniel S. Berger. 2023. An Introduction to the Compute Express Link (CXL) Interconnect.

Franyell Silfa, Gem Dot, Jose-Maria Arnau, and Antonio Gonzàlez. 2019. Neuron-level fuzzy memoization in rnns. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 782–793.

Jiwon Song, Kyungseok Oh, Taesu Kim, Hyungjun Kim, Yulhwa Kim, and Jae-Joon Kim. 2024. Sleb: Streamlining llms through redundancy verification and elimination of transformer blocks. *arXiv preprint arXiv:2402.09025*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NeurIPS*.

Hanrui Wang, Zhekai Zhang, and Song Han. 2021. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 97–110. IEEE.

Ruofan Wu, Feng Zhang, Jiawei Guan, Zhen Zheng, Xiaoyong Du, and Xipeng Shen. 2022. Drew: Efficient winograd cnn inference with deep reuse. In *Proceedings of the ACM Web Conference 2022*, pages 1807–1816.

Tong Xiao, Yinqiao Li, Jingbo Zhu, Zhengtao Yu, and Tongran Liu. 2019. Sharing attention weights for fast transformer. *arXiv preprint arXiv:1906.11024*.

Zhen Xie, Wenqian Dong, Jie Liu, Ivy Peng, Yanbao Ma, and Dong Li. 2021. Md-hm: memoization-based molecular dynamics simulations on big memory system. In *Proceedings of the ACM International Conference on Supercomputing*, pages 215–226.

Xiaoxuan Yang, Bonan Yan, Hai Li, and Yiran Chen. 2020. Retransformer: Reram-based processing-in-memory architecture for transformer acceleration. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9.

Chengxuan Ying, Guolin Ke, Di He, and Tie-Yan Liu. 2021. Lazyformer: Self attention with lazy update. *arXiv preprint arXiv:2102.12702*.

Bowen Zhang, Kehua Chang, and Chunping Li. 2024a. Simple techniques for enhancing sentence embeddings in generative language models. In *International Conference on Intelligent Computing*, pages 52–64. Springer.

Yang Zhang, Yawei Li, Xinpeng Wang, Qianli Shen, Barbara Plank, Bernd Bischl, Mina Rezaei, and Kenji Kawaguchi. 2024b. Finercut: Finer-grained interpretable layer pruning for large language models. *arXiv preprint arXiv:2405.18218*.

Shengyao Zhuang, Xueguang Ma, Bevan Koopman, Jimmy Lin, and Guido Zuccon. 2024. Promptreps: Prompting large language models to generate dense and sparse representations for zero-shot document retrieval. *arXiv preprint arXiv:2404.18424*.