MatMLA: Matryoshka Multi-Head Latent Attention

Anonymous Authors[1]

Abstract

Language models are increasingly deployed under diverse inference constraints, from low-latency interactive serving to higher-quality generation with larger compute budgets. Matryoshka-style Transformers address this setting by exposing multiple compute-quality operating points within a single set of weights, avoiding the need to train, store, and serve separate models for each budget. In parallel, multi-head latent attention (MLA) improves autoregressive decoding by replacing full per-head Key-Value (KV) caches with a compact latent cache. These ideas target complementary aspects of inference efficiency: Matryoshka controls the compute budget, while MLA controls the cache footprint. We introduce Matryoshka Multi-Head Latent Attention (MatMLA), a Matryoshka-style extension of MLA that exposes multiple attention-head budgets within a single latent-attention module. MatMLA inherits the compact KV cache of MLA while adding the nested sub-model structure of Matryoshka-style attention. This also resolves a key limitation of nested Multi-Head Attention (MHA): varying the active head count during inference requires irregular KV-cache layouts or full caches with inactive heads, whereas MatMLA's fixed-size latent cache gives all sub-models a shared memory format. We show that a 210M-parameter MatMLA model trained on 4.2B FineWeb-Edu tokens with nested head budgets of 12, 8, and 4 heads achieves perplexities close to the corresponding fixed-head MLA baselines, while preserving MLA's compact KV cache and enabling a shared nested parameterization across compute budgets.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

1. Introduction

Foundation models are deployed under widely varying inference constraints. Some applications require low-latency interactive serving under tight memory budgets (Kwon et al., 2023), including mobile and edge deployments (Zheng et al., 2025). Others prioritize quality and can spend substantially more computation at inference time, for example in reasoning (Snell et al., 2024; Muennighoff et al., 2025) and code generation workloads (Li et al., 2025). Training and maintaining many independently sized foundation models is costly, so model providers typically release only a small set of discrete sizes (Team et al., 2025). This forces practitioners to choose the closest available model to their latency, memory, and quality constraints, rather than selecting a model that directly meets the target budget.

MatFormers (Devvrit et al., 2024) address this limitation by training a single model with multiple nested sub-models, exposing several compute-quality operating points within one set of weights; this design has also been adopted in deployed open models such as Gemma 3n (Team et al., 2025). However, for autoregressive decoding, switching between MatFormer attention budgets by changing the active head count changes the per-token Key-Value (KV) representation, which can require separate caches, KV recomputation, or full caches with inactive heads. In a parallel line of work, Multi-Head Latent Attention (MLA) replaces full per-head KV caches with a compact latent cache, reducing the memory cost of autoregressive decoding while matching or improving the performance of standard Multi-Head Attention (MHA) in DeepSeek-V2 (DeepSeek-AI et al., 2024).

Together, these ideas suggest a best-of-both-worlds design: an attention module whose compute can be scaled by selecting different head budgets, while its KV cache remains compact and layout-compatible across all budgets. We introduce Matryoshka Multi-Head Latent Attention (MatMLA), a Matryoshka-style extension of MLA that realizes this design by nesting the MLA latent-to-head projections. The core idea is to decouple the nested attention width from the cached representation. For example, a single MatMLA layer can be evaluated with 4, 8, or 12 active attention heads, with smaller budgets using fewer heads and larger budgets using more heads. Unlike nested MHA, changing the active head count does not change the object stored in the KV

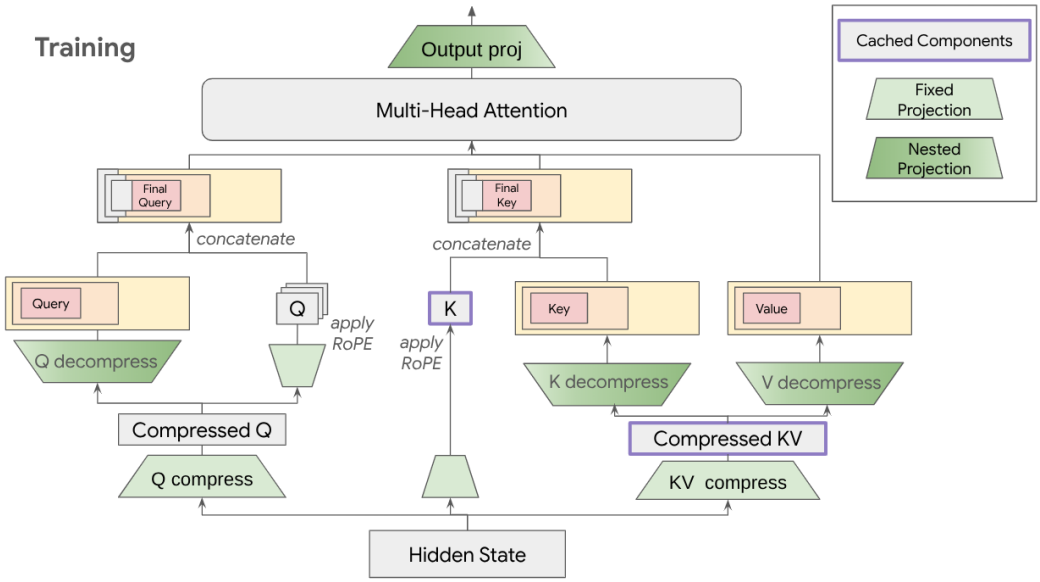


Figure 1. Diagram of a Matryoshka-style multi-head latent attention layer during training. The nested MLA varies the number of attention heads using nested decompression projections while the compressed KV cache stays a constant size.

cache: every token is written to the same dense latent KV representation, consisting of the dense latent and the fixed RoPE component. Thus, the head budget determines the attention computation performed by the current sub-model, but not the cache format inherited by future tokens. This allows all nested attention sub-models to share both weights and memory layout, avoiding the cache-layout mismatch that arises when Matryoshka-style head nesting is applied directly to MHA-based models.

We evaluate MatMLA in preliminary language modeling experiments. We train 210M-parameter models on 4.2B FineWeb-Edu tokens with nested head budgets of 12, 8, and 4 heads. Across these budgets, MatMLA remains competitive with the corresponding fixed-head MLA baselines while providing Matryoshka-style head nesting with shared compact KV cache.

2. Preliminaries

Notation. We denote the batch size by b, the sequence length by s, the model width by d, the number of attention heads by h, and the per-head dimension by d_h. Matrices and sequence-valued tensors are written in bold uppercase, e.g., $\boldsymbol{X}, \boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$, while vectors are written in bold lowercase, e.g., $\boldsymbol{x}_t, \boldsymbol{z}$. An input sequence is represented as $\boldsymbol{X} \in \mathbb{R}^{s\times d}$, with token vector $\boldsymbol{x}_t \in \mathbb{R}^d$. We use "block" and "layer" interchangeably. For MLA, we denote the KV latent dimension by c and the optional query latent dimension by c_q.

Sequence models map an input $\boldsymbol{X} \in \mathbb{R}^{s\times d}$ to an output $\boldsymbol{Y} \in \mathbb{R}^{s\times d}$.

Attention. The attention mechanism takes queries, keys, and values $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{s\times d_h}$, where d_h is the head dimension, and computes

$$\mathrm{Attn}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \mathrm{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_h}}\right)\boldsymbol{V}.$$

Multi-Head Attention. In Multi-Head Attention (MHA), attention is performed across h heads. The input is first projected into queries, keys, and values:

$$\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}^Q, \qquad \boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}^K, \qquad \boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}^V,$$

where $\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V \in \mathbb{R}^{d\times hd_h}$ and $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{s\times hd_h}$. The resulting h attention outputs are concatenated and projected back to width d. During autoregressive decoding, keys and values are cached to avoid recomputation; however, the KV cache scales with both the sequence length and the number of heads, making it costly at long contexts.

Multi-Head Latent Attention. Multi-Head Latent Attention (MLA) reduces the KV-cache memory footprint by storing key-value information in a compact latent representation rather than caching full per-head keys and values. While standard MHA materializes and caches $\boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{s\times hd_h}$, MLA first projects the input into a compressed latent $\boldsymbol{C}^{KV} \in \mathbb{R}^{s\times c}$, where $c \ll hd_h$:

$$\boldsymbol{C}^{KV} = \boldsymbol{X}\boldsymbol{W}^{DKV}, \tag{1}$$

with $\boldsymbol{W}^{DKV} \in \mathbb{R}^{d\times c}$. Keys and values are then obtained through latent-to-head projections:

$$\boldsymbol{K} = \boldsymbol{C}^{KV}\boldsymbol{W}^{UK}, \qquad \boldsymbol{V} = \boldsymbol{C}^{KV}\boldsymbol{W}^{UV}. \tag{2}$$

where $\boldsymbol{W}^{UK}, \boldsymbol{W}^{UV} \in \mathbb{R}^{c \times h d_h}$. Thus, instead of caching full keys and values, MLA caches $\boldsymbol{C}^{KV}$, which substantially reduces memory usage.

MLA also compresses the query into a latent $\boldsymbol{C}^{Q} \in \mathbb{R}^{s \times c_q}$, but to reduce activation memory during training instead of cache reduction:

$$\boldsymbol{C}^{Q} = \boldsymbol{X}\boldsymbol{W}^{DQ}, \qquad \boldsymbol{Q} = \boldsymbol{C}^{Q}\boldsymbol{W}^{UQ}. \tag{3}$$

with similar up- and down-projections $\boldsymbol{W}^{DQ} \in \mathbb{R}^{d \times c_q}, \boldsymbol{W}^{UQ} \in \mathbb{R}^{c_q \times h d_h}$.

When MLA is used with rotary position embeddings (Su et al., 2021), the embeddings are decoupled from the compressed latents, as the encodings would prevent $\boldsymbol{W}^{UK}$ from being absorbed into $\boldsymbol{W}^{Q}$ during inference. To enable relative encoding, MLA uses additional default queries and a non-compressed key. Thus, the cached components contain both the compressed KV latent and a small fixed RoPE key component which are appended prior to attention. We refer readers to DeepSeek-AI et al. (2024) for a more comprehensive analysis.[1]

The resulting queries, keys, and values are then used in the usual multi-head attention computation. Overall, MLA changes the parameterization and storage of key-value representations, while preserving standard attention computation.

Matryoshka-style models. Matryoshka-style models introduce an elastic architecture by training one universal model with a nested structure, allowing smaller submodels to be extracted without additional training. This principle originates from Matryoshka representation learning (Kusupati et al., 2024), which trains a single high-dimensional embedding to contain a family of smaller, lower-dimensional embeddings within it. This makes the representation elastic: at inference time, one can use a short prefix for lower storage and compute cost, or a longer prefix for higher accuracy, without training separate models for each embedding size.

The Matryoshka nested structure can also be applied to other layer and block types that include both sequence operations and channel operations (Cai et al., 2025; Devvrit et al., 2024; Shukla et al., 2024). Instead of training separate models that might vary the MLP width or Transformer head count, these works aim to train a model consisting of blocks that contain a nested family of subblocks

$$T_1 \subset T_2 \subset \cdots \subset T_g,$$

where the parameters of a smaller block T_i are contained within the parameters of every larger block $T_{i+1}, \ldots, T_g$. For instance, for a feed-forward network (FFN) block with hidden width d_{ff}, MatFormer chooses granularities

$$1 \le m_1 < m_2 < \cdots < m_g = d_{\text{ff}},$$

where each subblock T_i contains the first m_i hidden neurons. As such, the "earlier" neurons are more significant as they are shared across all subblocks.

After training, these methods extract a large family of submodels from the full nested model by selecting the active subblock in each layer. This allows a single trained model to support deployment across a range of latency constraints: different combinations of subblock granularities can be chosen to obtain models that meet a target compute or latency budget, without training separate models from scratch.

[1] In our work, we will ignore the RoPE components in certain cases to benefit clarity.

3. Methods

3.1. Matryoshka Multi-Head Latent Attention

We introduce *Matryoshka Multi-head Latent Attention* (MatMLA), a nested variant of MLA. MatMLA applies the nested structure to the number of attention heads within a layer. With the full MatMLA block containing h heads each with head dimension d_h, we choose g head granularities

$$h_1 < h_2 < \cdots < h_g = h,$$

where the i-th granularity uses the first h_i heads and requires a dimension of

$$d_i = h_i d_h.$$

The largest granularity recovers the full MLA block, while smaller granularities result in smaller subblocks.

Nested up-projections. As in vanilla MLA, the input is first mapped into compressed query and key-value latent representations as in Equation (1). To vary the number of heads, MatMLA nests the up-projections through prefix slices of the weights. For granularity i, the queries, keys, and values are calculated as follows:

$$\begin{gathered}\boldsymbol{Q}_i = \boldsymbol{C}^{Q}\boldsymbol{W}^{UQ}_{[:,\,1:d_i]}, \boldsymbol{K}_i = \boldsymbol{C}^{KV}\boldsymbol{W}^{UK}_{[:,\,1:d_i]}, \\ \boldsymbol{V}_i = \boldsymbol{C}^{KV}\boldsymbol{W}^{UV}_{[:,\,1:d_i]}.\end{gathered} \tag{4}$$

where $\boldsymbol{Q}_i, \boldsymbol{K}_i, \boldsymbol{V}_i \in \mathbb{R}^{s \times d_i}$. These decompressed representations are reshaped into h_i heads of dimension d_h and standard MHA is applied. The output is then projected back to the fixed model dimension d with a nested output projection

$$\boldsymbol{Y}_i = \mathrm{Attn}(\boldsymbol{Q}_i, \boldsymbol{K}_i, \boldsymbol{V}_i)\boldsymbol{W}^{O}_{[1:d_i,\,:]}. \tag{5}$$

In implementation, this is equivalent to zero-padding the inactive outputs and applying the full non-nested output projection.

Fixed RoPE components. In MatMLA, we apply the nested structure only to the heads produced by the up-projections for the compressed queries, keys, and values. For MatMLA, the RoPE components are kept fixed across all granularities where the RoPE-based query up-projections and key weights are shared across all $i \in \{1, \ldots, g\}$.

Our design decouples the position encoding part of the block from the nested structure. Although the weights and representations associated with the position encodings could also be made Matryoshka-style through a similar head count-based technique, we keep RoPE fixed to avoid introducing additional complications for further adaptive-compute extensions (Section 5). We leave nested RoPE parameterizations to future work.

3.2. MatMLA Training

We train MatMLA using a layer-wise nested sampling technique. Rather than selecting a single granularity for the entire model, we independently sample the granularity at each layer. Given a full model with l layers, let $\boldsymbol{g} = (g_1, \ldots, g_l)$ denote the sampled granularity across all layers where each granularity is sampled independently:

$$\boldsymbol{g}_\ell \sim \mathcal{D}(\{1, \ldots, g\}), \ell \in [l].$$

Given one such granularity vector, the corresponding MatMLA model can be represented as

$$M_{\boldsymbol{g}} = [T_{\boldsymbol{g}_1}, T_{\boldsymbol{g}_2}, \ldots, T_{\boldsymbol{g}_l}],$$

where $T_{\boldsymbol{g}_\ell}$ denotes the ℓ-th layer MatMLA subblock instantiated with granularity $\boldsymbol{g}_\ell$. A minibatch shares the same $\boldsymbol{g}$ across its samples. The overall model is optimized according to standard pretraining where the loss is defined as

$$\mathcal{L}_{\text{MatMLA}}(x, y) = \mathcal{L}(M_{\boldsymbol{g}}(x), y).$$

Thus, throughout training, the model learns to operate with mixed-granularities across layers.

During evaluation, the fixed submodel is extracted by passing in a granularity vector $\boldsymbol{g}$ into the full nested model. Before inference, the selected submodel would utilize the associated prefix weights of each respective layer, and the query/key up-projections and value/output up-projections would need to be fused. The exact vector's configuration depends on the use case of the practitioner and can follow that of Devvrit et al. (2024).

4. Empirical Results

4.1. Training Procedure

We trained the models described in this writeup on a FineWeb-Edu blend using a Llama-3 tokenizer and a GPT-style decoder implemented within Nvidia's Megatron training stack. Across runs, the shared architecture used 12 layers with $d_{\text{model}} = 768$, FFN hidden size 3072, sequence length 2048, and a global batch size of 256 sequences. All runs used bfloat16 training and a total budget of 4.2B training tokens. The learning rate schedule was cosine decay with a peak learning rate of 3×10^{-3} and a warmup phase covering 10% of the total training tokens.

Table 1. Validation perplexity of MatMLA sub-models and fixed-head MLA baselines. MatMLA is trained once with the nested head family $\{12, 8, 4\}$ and evaluated at each fixed head budget.

Active heads	**MatMLA** $\{12, 8, 4\}$	**MLA**
12	22.75	22.35
8	22.86	22.64
4	24.11	22.75

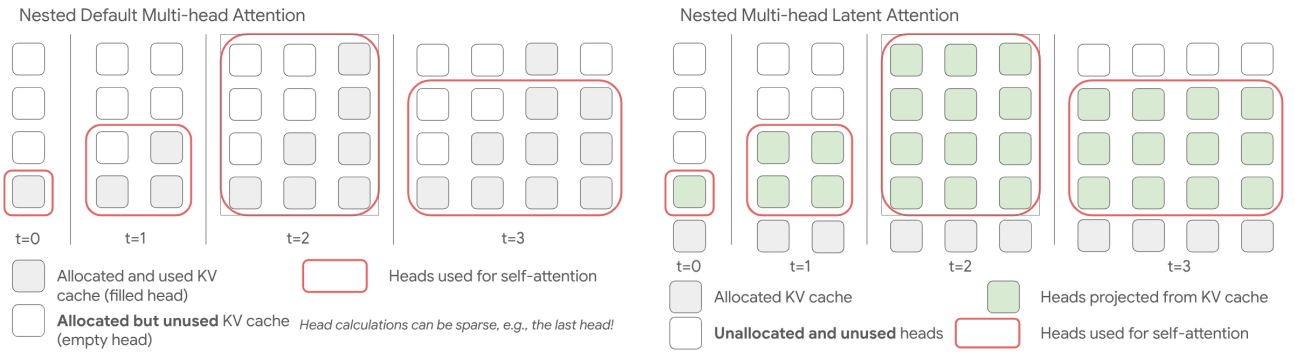
The four runs used in our setup were as follows. `mla-12-8-4` denotes the MatMLA training run with the explicit head family $\{12, 8, 4\}$ sampled proportionally to the head counts. At evaluation time, this single trained model is evaluated at fixed head budgets of 12, 8, and 4 heads. We report validation perplexity for each MatMLA sub-model and its corresponding fixed-head MLA baseline in Table 1. Overall, MatMLA underperforms the corresponding fixed-head MLA baselines in validation perplexity across the evaluated head budgets, but does so while enabling all sub-models to share a nested parameterization an a common KV-cache. We note that at the 210M-parameter scale, Matryoshka models can be sensitive to training configuration. Further investigation is needed to characterize MatMLA's performance across model scales and training settings.

5. Future Adaptive Inference with MatMLA

Our MatMLA architecture lays the foundation for enabling adaptive nested selection on the sequence level, where fewer heads are allocated to easier tokens and more heads are allocated to harder ones. In this section, we first discuss why directly applying adaptive head count selection to nested MHA leads to potential issues, analyze the FLOP and memory of MatMLA in an adaptive setting, and outline possible routing mechanisms.

5.1. Overcoming Nested MHA Sparsity

While Figure 2 shows that nested MHA can lead to KV cache sparsity due to empty heads, this can be overcome by duplicating the KV cache to fill all heads completely. This would enable the full utilization of all heads by timesteps that use the full model. However, this mechanism poses potential issues where many heads may have similar outputs. This can occur if their KV values are identical due to duplication, thus only relying on the differences in query values to result in varying attention matrices. Our nested



(a) Nested MHA. *(b)* Nested MLA.

Figure 2. Comparison of nested MHA and nested MLA under variable head budgets. In nested MHA, changing the active head count changes the per-token KV representation, which can require inactive cache entries, irregular cache layouts, or recomputation. In MatMLA, every token is stored in the same fixed-format KV cache, independent of the active head budget. The head budget changes the current attention computation, while the persistent cache representation remains fixed.

MLA aims to overcome this by having the KV cache be projected via learned weights to the required number of heads, resulting in unique tensors per head. This can be done without recalculation as seen in Figure 3.

5.2. FLOP and Cache-Memory Analysis

We compare MatMLA and nested MHA in a decoding setting with one attention layer and a sequence of length l. An α fraction of tokens uses the 50% submodel with $h/2$ active heads, and the remaining $1-\alpha$ fraction uses the full model with h active heads. We count projection FLOPs for forming $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$, and cached elements for autoregressive decoding. We omit the compute associated with applying RoPE, running attention, and output-projection costs, since these terms are comparable under the same number of heads and head dimension.

Nested MHA. For standard MHA, $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ are each produced by a projection from width d to width hd_h. Thus, at the full head count, each projection requires $2bdhd_h$ FLOPs per token, and the total QKV projection cost is

$$2bdhd_h + 2bdhd_h + 2bdhd_h = 6bdhd_h.$$

For the 50% submodel with $h/2$ active heads, each projection has output width $(h/2)d_h$, so the corresponding QKV projection cost is

$$2bd\frac{h}{2}d_h + 2bd\frac{h}{2}d_h + 2bd\frac{h}{2}d_h = 6bd\frac{h}{2}d_h.$$

Therefore, over a sequence of length l, where an α fraction of tokens uses the 50% submodel and the remaining $1-\alpha$ fraction uses the full model, the total nested-MHA projection FLOPs are

$$\text{FLOPs}_{\text{MHA}} = 6\alpha bld\frac{h}{2}d_h + 6(1-\alpha)bldhd_h = (6-3\alpha)bldhd_h.$$

If the KV cache is allocated for the maximum head count at every timestep, nested MHA requires

$$\text{Cache}_{\text{MHA}} = 2blhd_h,$$

where the factor of 2 accounts for keys and values. However, only $(2-\alpha)blhd_h$ cache entries are actually used.

MatMLA. For MatMLA, the overarching query, key, value projections are decomposed into down- then up-projections for the latent. Recall from Section 2 that c denotes the compressed latent dimension, and let r denote the fixed RoPE dimension. We count the FLOPs required to construct the query, key, and value representations used by attention.

At all h heads, the projection costs are as follows. The compressed query projection costs $2bdc$. The non-RoPE query up-projection maps from width c to $h(d_h-r)$, costing $2bch(d_h-r)$. The RoPE query component costs $2bchr$. The compressed KV projection costs $2bdc$. The non-RoPE key up-projection maps from width c to $h(d_h-r)$, costing $2bch(d_h-r)$. The value up-projection maps from width c to hd_h, costing $2bchd_h$. Finally, the RoPE key component costs $2bdr$. Thus, the full-head MatMLA projection cost per token is

$$\begin{aligned}&2bdc + 2bch(d_h-r) + 2bchr + 2bdc\\ &\quad + 2bch(d_h-r) + 2bchd_h + 2bdr\\ &= 6bchd_h - 2bchr + 4bdc + 2bdr.\end{aligned}$$

For the 50% submodel with $h/2$ active heads, only the non-RoPE up-projection terms scale. The compressed down-projection costs remain fixed. Therefore, the per-token projection cost is

$$6bc\frac{h}{2}d_h - 2bc\frac{h}{2}r + 4bdc + 2bdr.$$

Thus, under our fractional submodel setting, the total MatMLA projection FLOPs are

$$\begin{aligned}\text{FLOPs}_{\text{MatMLA}} &= \alpha l \left(6bc\frac{h}{2}d_h - 2bc\frac{h}{2}r + 4bdc + 2bdr\right) \\ &\quad +(1-\alpha)l\left(6bchd_h - 2bchr + 4bdc + 2bdr\right) \\ &= (6-3\alpha)blchd_h - (2-\alpha)blchr + 4bldc + 2bldr.\end{aligned}$$

Given that MatMLA caches the compressed latent which does not change given the submodel, the cache requirement is

$$\text{Cache}_{\text{MatMLA}} = bl(r+c).$$

Compression-ratio estimate. To express the MatMLA cost in terms of the standard MHA dimensions, let $c = \delta_1 d$ and $r = \delta_2 d_h$, where $\delta_1, \delta_2 < 1$. Substituting these into the MatMLA FLOP expression gives

$$\begin{aligned}\text{FLOPs}_{\text{MatMLA}} &= (6-3\alpha)\delta_1 bldhd_h - (2-\alpha)\delta_1\delta_2 bldhd_h \\ &\quad + 4\delta_1 bld^2 + 2\delta_2 bldd_h.\end{aligned}$$

Using $d \le hd_h$, we upper bound $4\delta_1 bld^2 \le 4\delta_1 bldhd_h$, yielding

$$\begin{aligned}\text{FLOPs}_{\text{MatMLA}} &\le (10-3\alpha)\delta_1 bldhd_h - (2-\alpha)\delta_1\delta_2 bldhd_h \\ &\quad + 2\delta_2 bldd_h.\end{aligned}$$

Using DeepSeek-V2-style ratios $\delta_1 \approx 0.3$ and $\delta_2 \approx 0.3$, and taking a loose upper bound on the negative term, this becomes

$$\text{FLOPs}_{\text{MatMLA}} < (3.5-\alpha)bldhd_h + 0.75bldd_h.$$

This estimate is conservative because the compressed KV dimension is typically smaller than the query latent dimension; using the same δ_1 for both slightly overestimates MatMLA FLOPs.

Under the same ratios, the latent-cache requirement is

$$\text{Cache}_{\text{MatMLA}} = bl(r+c) = 0.3bl(d_h+d).$$

Since $d \le hd_h$, we have

$$0.3bl(d_h+d) \le 0.3bl(d_h+hd_h) < blhd_h,$$

which is smaller than the $2blhd_h$ for nested MHA.

5.3. Token Routing Mechanisms

These architectural changes could enable token routing, but the mechanism for such remains an open question. We provide some possible preliminary suggestions here.

We believe that using token-level routers, similar to MoEs, would provide a strong baseline. Token-level routing has been shown to be better than router-level mechanisms, as it sidesteps causality issues and has been utilized for routing models like Mixture-of-Recursion. However, it is worth exploring expert-level routing. In order to encourage routing to the less expressive but smaller sub-models, auxiliary losses that are inversely correlated with the size of the sub-model can be added to both the final pre-training loss or reward during fine-tuning or reinforcement learning.

In order to train this token routing mechanism, we would project to the maximum number of heads for Q, K, and V. Because the KV values are generated in full, this enables all timesteps to flexibly choose the number of heads they want to utilize. Once the number of active heads are determined, we then mask out attention output of the heads deemed inactive. This ensures that for each timestep's self-attention output, certain heads are zeroed. When passed through the final output projection, the zeroed heads do not contribute to the final hidden state, resulting in the desired output. This query masking at each timestep is equivalent to utilizing the nested compressed KV cache up-projection.

6. Conclusion

We introduced MatMLA, a Matryoshka-style extension of multi-head latent attention that exposes multiple attention-head budgets within a single latent-attention module. MatMLA combines the elastic compute structure of Matryoshka-style Transformers with the compact, fixed-format KV cache of MLA. By storing every token in the same dense latent KV representation and fixed RoPE component, MatMLA avoids the cache-layout incompatibility that arises when nested head budgets are applied directly to MHA. Our preliminary language modeling experiments show that nested MatMLA sub-models with 12, 8, and 4 heads achieve perplexities close to their corresponding fixed-head MLA baselines. These results suggest that latent attention can support Matryoshka-style head nesting while preserving its cache-efficiency benefits. More broadly, MatMLA provides a simple foundation for adaptive inference, where future systems may select different attention budgets across tokens or chunks without changing the persistent cache format.

Impact Statement

This paper presents work whose goal is to improve the efficiency and deployment flexibility of machine learning systems. Potential societal impacts are primarily tied to the broader deployment of language models and adaptive inference systems; no additional application-specific risks are introduced by the mechanism itself.

References

Cai, R., Muralidharan, S., Yin, H., Wang, Z., Kautz, J., and Molchanov, P. LLamaflex: Many-in-one LLMs via generalized pruning and weight sharing. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=AyC4uxx2HW.

DeepSeek-AI, Liu, A., Feng, B., Wang, B., Wang, B., Liu, B., Zhao, C., Dengr, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Xu, H., Yang, H., Zhang, H., Ding, H., Xin, H., Gao, H., Li, H., Qu, H., Cai, J. L., Liang, J., Guo, J., Ni, J., Li, J., Chen, J., Yuan, J., Qiu, J., Song, J., Dong, K., Gao, K., Guan, K., Wang, L., Zhang, L., Xu, L., Xia, L., Zhao, L., Zhang, L., Li, M., Wang, M., Zhang, M., Zhang, M., Tang, M., Li, M., Tian, N., Huang, P., Wang, P., Zhang, P., Zhu, Q., Chen, Q., Du, Q., Chen, R. J., Jin, R. L., Ge, R., Pan, R., Xu, R., Chen, R., Li, S. S., Lu, S., Zhou, S., Chen, S., Wu, S., Ye, S., Ma, S., Wang, S., Zhou, S., Yu, S., Zhou, S., Zheng, S., Wang, T., Pei, T., Yuan, T., Sun, T., Xiao, W. L., Zeng, W., An, W., Liu, W., Liang, W., Gao, W., Zhang, W., Li, X. Q., Jin, X., Wang, X., Bi, X., Liu, X., Wang, X., Shen, X., Chen, X., Chen, X., Nie, X., Sun, X., Wang, X., Liu, X., Xie, X., Yu, X., Song, X., Zhou, X., Yang, X., Lu, X., Su, X., Wu, Y., Li, Y. K., Wei, Y. X., Zhu, Y. X., Xu, Y., Huang, Y., Li, Y., Zhao, Y., Sun, Y., Li, Y., Wang, Y., Zheng, Y., Zhang, Y., Xiong, Y., Zhao, Y., He, Y., Tang, Y., Piao, Y., Dong, Y., Tan, Y., Liu, Y., Wang, Y., Guo, Y., Zhu, Y., Wang, Y., Zou, Y., Zha, Y., Ma, Y., Yan, Y., You, Y., Liu, Y., Ren, Z. Z., Ren, Z., Sha, Z., Fu, Z., Huang, Z., Zhang, Z., Xie, Z., Hao, Z., Shao, Z., Wen, Z., Xu, Z., Zhang, Z., Li, Z., Wang, Z., Gu, Z., Li, Z., and Xie, Z. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024. URL https://arxiv.org/abs/2405.04434.

Devvrit, Kudugunta, S., Kusupati, A., Dettmers, T., Chen, K., Dhillon, I., Tsvetkov, Y., Hajishirzi, H., Kakade, S., Farhadi, A., and Jain, P. Matformer: Nested transformer for elastic inference, 2024. URL https://arxiv.org/abs/2310.07707.

Kusupati, A., Bhatt, G., Rege, A., Wallingford, M., Sinha, A., Ramanujan, V., Howard-Snyder, W., Chen, K., Kakade, S., Jain, P., and Farhadi, A. Matryoshka representation learning, 2024. URL https://arxiv.org/abs/2205.13147.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

Li, D., Cao, S., Cao, C., Li, X., Tan, S., Keutzer, K., Xing, J., Gonzalez, J. E., and Stoica, I. S*: Test time scaling for code generation, 2025. URL https://arxiv.org/abs/2502.14382.

Muennighoff, N., Yang, Z., Shi, W., Li, X. L., Fei-Fei, L., Hajishirzi, H., Zettlemoyer, L., Liang, P., Candès, E., and Hashimoto, T. s1: Simple test-time scaling, 2025. URL https://arxiv.org/abs/2501.19393.

Shukla, A., Vemprala, S., Kusupati, A., and Kapoor, A. Matmamba: A matryoshka state space model, 2024. URL https://arxiv.org/abs/2410.06718.

Snell, C., Lee, J., Xu, K., and Kumar, A. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., and Liu, Y. Roformer: Enhanced transformer with rotary position embedding, 2021. URL https://arxiv.org/abs/2104.09864.

Team, G., Kamath, A., Ferret, J., Pathak, S., Vieillard, N., Merhej, R., Perrin, S., Matejovicova, T., Ramé, A., Rivière, M., Rouillard, L., Mesnard, T., Cideron, G., bastien Grill, J., Ramos, S., Yvinec, E., Casbon, M., Pot, E., Penchev, I., Liu, G., Visin, F., Kenealy, K., Beyer, L., Zhai, X., Tsitsulin, A., Busa-Fekete, R., Feng, A., Sachdeva, N., Coleman, B., Gao, Y., Mustafa, B., Barr, I., Parisotto, E., Tian, D., Eyal, M., Cherry, C., Peter, J.-T., Sinopalnikov, D., Bhupatiraju, S., Agarwal, R., Kazemi, M., Malkin, D., Kumar, R., Vilar, D., Brusilovsky, I., Luo, J., Steiner, A., Friesen, A., Sharma, A., Sharma, A., Gilady, A. M., Goedeckemeyer, A., Saade, A., Feng, A., Kolesnikov, A., Bendebury, A., Abdagic, A., Vadi, A., György, A., Pinto, A. S., Das, A., Bapna, A., Miech, A., Yang, A., Paterson, A., Shenoy, A., Chakrabarti, A., Piot, B., Wu, B., Shahriari, B., Petrini, B., Chen, C., Lan, C. L., Choquette-Choo, C. A., Carey, C., Brick, C., Deutsch, D., Eisenbud, D., Cattle, D., Cheng, D., Paparas, D., Sreepathihalli, D. S., Reid, D., Tran, D., Zelle, D., Noland, E., Huizenga, E., Kharitonov, E., Liu, F., Amirkhanyan, G., Cameron, G., Hashemi, H., Klimczak-Plucińska, H., Singh, H., Mehta, H., Lehri, H. T., Hazimeh, H., Ballantyne, I., Szpektor, I., Nardini, I., Pouget-Abadie, J., Chan, J., Stanton, J., Wieting, J., Lai, J., Orbay, J., Fernandez, J., Newlan, J., yeong Ji, J., Singh, J., Black, K., Yu, K., Hui, K., Vodrahalli, K., Greff, K., Qiu, L., Valentine, M., Coelho, M., Ritter, M., Hoffman, M., Watson, M., Chaturvedi, M., Moynihan, M., Ma, M., Babar, N., Noy, N., Byrd, N., Roy, N., Momchev, N., Chauhan, N., Sachdeva, N., Bunyan, O., Botarda, P., Caron, P., Rubenstein, P. K., Culliton, P., Schmid, P., Sessa, P. G., Xu, P., Stanczyk, P., Tafti, P.,

Shivanna, R., Wu, R., Pan, R., Rokni, R., Willoughby, R., Vallu, R., Mullins, R., Jerome, S., Smoot, S., Girgin, S., Iqbal, S., Reddy, S., Sheth, S., Põder, S., Bhatnagar, S., Panyam, S. R., Eiger, S., Zhang, S., Liu, T., Yacovone, T., Liechty, T., Kalra, U., Evci, U., Misra, V., Roseberry, V., Feinberg, V., Kolesnikov, V., Han, W., Kwon, W., Chen, X., Chow, Y., Zhu, Y., Wei, Z., Egyed, Z., Cotruta, V., Giang, M., Kirk, P., Rao, A., Black, K., Babar, N., Lo, J., Moreira, E., Martins, L. G., Sanseviero, O., Gonzalez, L., Gleicher, Z., Warkentin, T., Mirrokni, V., Senter, E., Collins, E., Barral, J., Ghahramani, Z., Hadsell, R., Matias, Y., Sculley, D., Petrov, S., Fiedel, N., Shazeer, N., Vinyals, O., Dean, J., Hassabis, D., Kavukcuoglu, K., Farabet, C., Buchatskaya, E., Alayrac, J.-B., Anil, R., Dmitry, Lepikhin, Borgeaud, S., Bachem, O., Joulin, A., Andreev, A., Hardin, C., Dadashi, R., and Hussenot, L. Gemma 3 technical report, 2025. URL https://arxiv.org/abs/2503.19786.

Zheng, Y., Chen, Y., Qian, B., Shi, X., Shu, Y., and Chen, J. A review on edge large language models: Design, execution, and applications, 2025. URL https://arxiv.org/abs/2410.11845.

A. Schematic of Inference for MatMLA

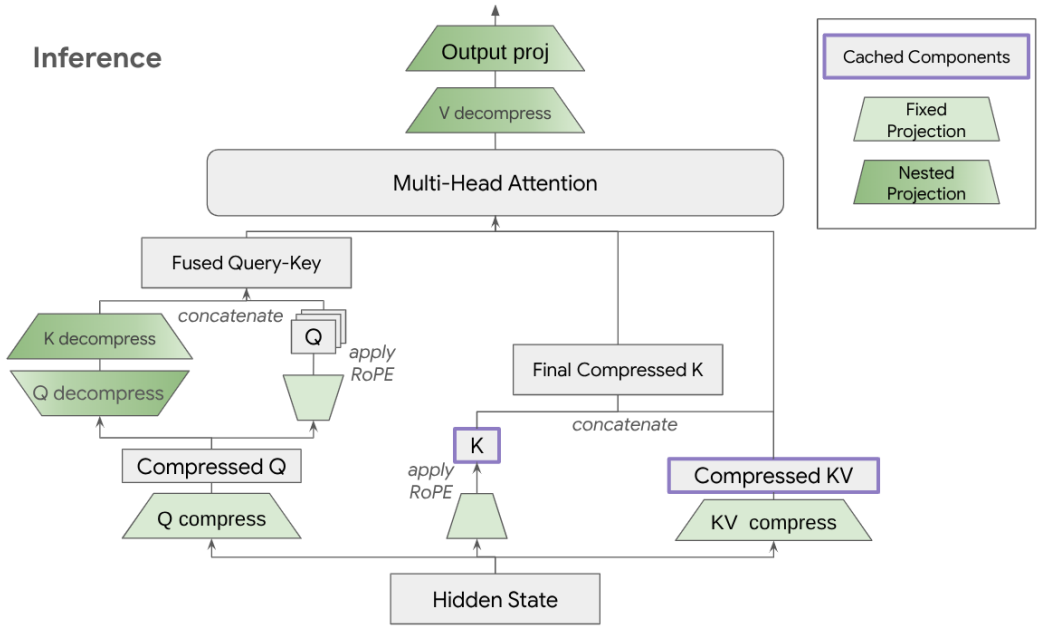


Figure 3. Diagram of Matryoshka-MLA during inference. The nested K decompression projection is placed after the Q decompression and the nested V decompression is placed before the output projection to allow the compressed KV cache to be directly passed into the self-attention kernel without recalculation.

B. Variable Sequence Length Calculation

Table 2. FLOP and cache-memory comparison for variable sequence length l, where an α fraction of tokens uses the 50% sub-model and the rest uses the full model.

	MatMLA (our proposal)	**Nested MHA (baseline)**
FLOP requirement	$(6-3\alpha)blchd_h - (2-\alpha)blchr + 4bldc + 2bldr$	$(6-3\alpha)bldhd_h$
Cache requirement	$bl(r+c) < bl(hd_h + d_h)$	$2blhd_h$
FLOP requirement with ratio	$(3.5-\alpha)bldhd_h + 0.75bldd_h$	$(6-3\alpha)bldhd_h$
Cache requirement with ratio	$0.3bl(d_h + hd_h) < blhd_h$	$2blhd_h$