FINAL REPORT: OPTIMIZING MULTI-HEAD LATENT ATTENTION FOR EFFICIENT INFERENCE

Yike Tan [*1] **Qingzheng Wang** [*1] **Xun Wang** [*1]
{yiket, qingzhew, xunwang}@andrew.cmu.edu
Information Networking Institute, Carnegie Mellon University[1]

ABSTRACT

Multi-head Latent Attention (MLA) has emerged as a promising alternative to standard Multi-Head Attention (MHA) by significantly reducing inference memory through latent space compression of Key-Value (KV) caches. In this project, we present an optimized implementation of MLA in PyTorch, targeting system-level performance improvements through Triton-based kernel fusion and efficient KV cache management. We reproduce MLA with absorbed upsampling weights and optimize four critical components — attention score computation, softmax, root mean square normalization, and rotary positional embedding — using custom Triton kernels. Furthermore, we implement a paged KV cache manager that dynamically allocates memory at runtime, minimizing memory waste compared to traditional static preallocation. By parallelizing cache update and retrieval operations with Triton kernels, we achieve further efficiency gains. Experimental results show that our optimized MLA implementation achieves up to 73% improvement in memory throughput and reduces memory usage by over 90% relative to the baseline. Our code is avalable at https://github.com/Qingzheng-Wang/ShallowMLA.

1 INTRODUCTION

Recently, DeepSeek released the official optimized implementation of Multi-head Latent Attention (MLA), FlashMLA. MLA (Shao et al., 2024) is a variant of Multi-Head Attention (MHA) (Vaswani et al., 2017) that significantly reduces inference memory by compressing the Key-Value (KV) cache into a latent space. Inspired by this innovation, our course project aims to re-implement MLA using PyTorch and improve its inference efficiency through system-level optimizations. We focus not only on reproducing the core attention mechanism but also on accelerating it with custom Triton (Tillet et al., 2019) kernels and optimized KV cache.

MHA forms the backbone of modern large language models (LLMs). MHA concatenates multiple independent single-head attention mechanisms, enabling the model to capture diverse contextual representations across different attention heads in parallel. In autoregressive models like LLMs, keys and values are computed independently at each step, making it possible to construct a KV cache during inference. This caching avoids redundant computation and significantly improves generation efficiency.

However, directly caching per-token keys and values leads to substantial memory consumption, especially in large models. To further improve efficiency, variants such as Multi-Query Attention (MQA) (Shazeer, 2019) and Grouped-Query Attention (GQA) (Ainslie et al., 2023) have been proposed. These approaches reduce cache memory overhead by sharing keys and values across multiple queries. However, this simplification leads to a reduction in representation capacity, and their performance often lags behind full MHA. To address this trade-off, MLA introduces a different approach by compressing keys and values into a latent space for caching, and then upsampling them back to the model dimension during computation. This design reduces the KV cache size while retaining performance comparable to standard MHA (Shao et al., 2024).

In this project, we focused on implementing Multi-head Latent Attention (MLA) in PyTorch and systematically optimizing its inference performance. Our contributions include: (1) reproducing MLA with absorbed KV up-projection weights for memory and computation savings, (2) performing system-level kernel fusion for critical MLA components, (3) developing a paged KV cache management system inspired by Kwon et al. (2023), and (4) optimizing cache updates and retrievals through batched operations using Triton (Tillet et al., 2019). Specifically, we implement an efficient version of MLA with absorbing the upsampling weights of latent vectors to queries following DeepSeek-V3 (DeepSeek-AI, 2024). In addition, we optimized four key operations in MLA — query-key multiplication, softmax computation, root mean square normalization (RMSNorm), and rotary positional embedding (RoPE) — by

implementing custom Triton kernels to improve computational efficiency. Furthermore, we introduced a paged KV cache manager that dynamically allocates memory based on sequence progression, significantly reducing memory waste compared to static preallocation. We also accelerated cache update and retrieval processes by parallelizing them with Triton kernels. Experimental results show that, compared to the baseline PyTorch implementation, our optimized MLA achieves substantial improvements in inference throughput. The paged KV cache design reduces memory usage by over 90%, while Triton-optimized cache operations yield approximately 30% additional throughput improvement over the basic paged cache implementation.

2 Related Work

2.1 MQA and GQA

Optimizing KV cache has long been a key challenge in improving inference efficiency. Prior works such as MQA (Shazeer, 2019) and GQA (Ainslie et al., 2023) address this from a parameter-sharing perspective. MQA's idea is that all attention heads share the same key (K) and value (V) representations. This significantly reduces KV cache size, making it one of the most memory-efficient solutions. While MQA significantly reduces KV cache, its aggressive compression can hinder learning efficiency and model performance. To address this, GQA (Ainslie et al., 2023) was introduced as a middle ground between MHA and MQA, which divides attention heads into groups, each sharing a distinct K, V pair. In this project, we focus on implementing MLA, which reduces KV cache and computation through latent KV representations, rather than sharing K and V across heads.

2.2 FlashMLA

FlashMLA is DeepSeek's high-performance CUDA implementation of Multi-head Latent Attention (MLA) (Shao et al., 2024), specifically optimized for NVIDIA's Hopper architecture. It leverages key optimization strategies from FlashAttention, including tiling techniques and memory-efficient algorithms to minimize reliance on GPU memory bandwidth. On H800 GPUs, FlashMLA can achieve memory bandwidth of up to 3000 GB/s through row-wise processing and split-KV handling. In the midterm stage of our project, we implement initial Triton-based optimizations, drawing inspiration from FlashMLA, with plans for further enhancements in later phases.

2.3 Triton

Triton (Tillet et al., 2019) is an open-source Python-based programming language and compiler designed for writing high-performance custom GPU kernels. Unlike traditional CUDA programming, which often requires low-level control and significant boilerplate, Triton offers a higher-level abstraction, allowing developers to express parallelism more naturally. It supports Single Program Multiple Data (SPMD) programming models and enables efficient blocked algorithms that are well-suited for modern deep learning workloads such as matrix multiplication and attention. In this project, Triton plays a key role in optimizing the system-level performance of MLA by enabling kernel fusion and maximizing GPU throughput. Its flexibility and ease of use allow for rapid development and tuning of custom kernels, bridging the gap between productivity and performance.

Symbol	Description
d	Embedding dimension of the input vector
n_{h}	Number of attention heads
d_{h}	Dimension per head for query and key
d_{h}^{v}	Dimension per head for value
d_{c}	Latent key-value rank
d_{c}^{q}	Query latent rank
d_{r}	Dimension per head for RoPE embedding

Table 1. Descriptions of model dimensions.

2.4 PagedAttention

PagedAttention (Kwon et al., 2023) improves KV cache efficiency by dividing keys and values into fixed-size memory pages, enabling dynamic allocation and reducing fragmentation in long-context inference. This system-level approach allows selective loading and eviction of KV blocks, enhancing scalability without modifying attention head structures. Adopted in frameworks like vLLM, PagedAttention complements KV compression methods like MLA by optimizing cache management and access patterns for large-scale inference. In this project, we follow the idea of PagedAttention and implement the paged KV cache management.

3 Methods

3.1 Efficient MLA Implementation

MLA enhances inference efficiency by optimizing KV cache. The core of MLA is the low-rank compression of KV. In our implementation, we did some optimizations based on the mechanism of MLA referenced to the official inference code of DeepSeek-V3, which implements MLA efficiently through absorbing weights. [1] Figure 1 provides an overview of the architecture adopted in our optimized MLA implementation.

Model dimension notations are depicted in Table 1. For

[1] https://github.com/deepseek-ai/DeepSeek-V3

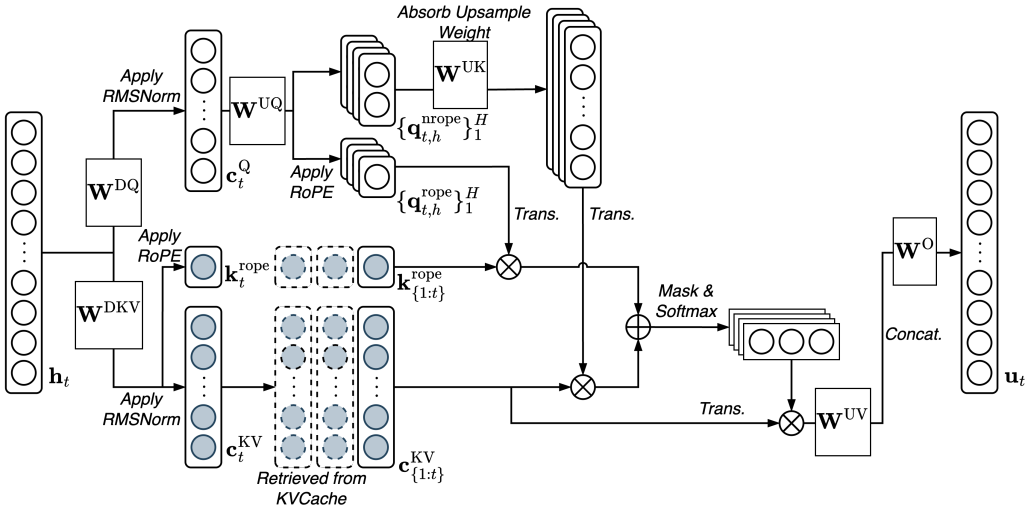

Figure 1. Efficient MLA implementation with absorbed KV up-projection weights for memory and computation savings.

queries, to save the activation memory, MLA also performed a low-rank compression,

$$\mathbf{q}'_{\mathrm{t}} = W^{\mathrm{UQ}}\mathrm{RMSNorm}(W^{\mathrm{DQ}}\mathbf{h}_t), \tag{1}$$

where $W^{\mathrm{DQ}} \in \mathbb{R}^{d_{\mathrm{c}}^{\mathrm{q}} \times d}$ and $W^{\mathrm{UQ}} \in \mathbb{R}^{n_{\mathrm{h}} \times (d_{\mathrm{h}}+d_{\mathrm{r}}) \times d_{\mathrm{c}}^{\mathrm{q}}}$. RMSNorm (Zhang & Sennrich, 2019) is a variant of layer normalization without centering. For an input vector $\mathbf{x} \in \mathbb{R}^d$, it is defined as:

$$\mathrm{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} \mathbf{x}_i^2 + \epsilon}} \odot \mathbf{w}, \tag{2}$$

where $\mathbf{w} \in \mathbb{R}^d$ is a learnable scale parameter and ϵ is a small constant for numerical stability. $\mathbf{q}_t$ is split into two branches for non-RoPE and RoPE computing, which saves the computation burden for separately upsampling,

$$\mathbf{q}'_{\mathrm{t}} = [\mathbf{q}_{\mathrm{t}}^{\mathrm{nr}}; \mathbf{q}_{\mathrm{t}}^{\mathrm{r}'}], \tag{3}$$

where $\mathbf{q}_{\mathrm{t}}^{\mathrm{nr}} \in \mathbb{R}^{n_{\mathrm{h}} \times d_{\mathrm{h}}}$, $\mathbf{q}_{\mathrm{t}}^{\mathrm{r}} \in \mathbb{R}^{n_{\mathrm{h}} \times d_{\mathrm{r}}}$. Then the RoPE embedding of query is

$$\mathbf{q}_{\mathrm{t}}^{\mathrm{r}} = \mathrm{RoPE}(\mathbf{q}_{\mathrm{t}}^{\mathrm{r}'}). \tag{4}$$

RoPE (Su et al., 2021) (Rotary Position Embedding) is defined as:

$$\mathrm{RoPE}(\mathbf{x}) = \mathrm{Re}\left(\mathbf{x}^{\mathbb{C}} \odot \mathbf{f}_t^{\mathbb{C}}\right), \tag{5}$$

$$\mathbf{f}_t^{\mathbb{C}} = \left[e^{i\theta_0 t}, e^{i\theta_1 t}, \ldots, e^{i\theta_{d/2-1} t}\right], \tag{6}$$

$$\theta_k = \frac{1}{\theta^{\frac{2k}{d}}}, \quad k = 0, 1, \ldots, \frac{d}{2} - 1, \tag{7}$$

where $\mathbf{x} \in \mathbb{R}^d$, $\mathbf{x}^{\mathbb{C}}$ and $\mathbf{f}_t^{\mathbb{C}} \in \mathbb{C}^{d/2}$ are the complex representations of the input and rotary frequencies, $\odot$ denotes element-wise complex multiplication, and $\theta = 1/10000$. In practice, we store the rotary frequencies as real-valued sine and cosine pairs and apply them via element-wise complex multiplication implemented through real-imaginary decomposition. Finally, we get the query

$$\mathbf{q}_{\mathrm{t}} = [\mathbf{q}_{\mathrm{t}}^{\mathrm{nr}}; \mathbf{q}_{\mathrm{t}}^{\mathrm{r}}], \tag{8}$$

where $\mathbf{q}_{\mathrm{t}}^{\mathrm{nr}} \in \mathbb{R}^{n_{\mathrm{h}} \times (d_{\mathrm{h}}+d_{\mathrm{r}})}$.

Then we downsample the input to latent key-value vector and key RoPE embedding jointly,

$$[\mathbf{c}_t^{\mathrm{KV}'}; \mathbf{k}_t^{\mathrm{r}'}] = W^{\mathrm{DKV}}\mathbf{h}_t, \tag{9}$$

$$\mathbf{c}_t^{\mathrm{KV}} = \mathrm{RMSNorm}(\mathbf{c}_t^{\mathrm{KV}'}) \tag{10}$$

$$\mathbf{k}_t^{\mathrm{r}} = \mathrm{RoPE}(\mathbf{k}_t^{\mathrm{r}'}) \tag{11}$$

where $W^{\mathrm{DKV}} \in \mathbb{R}^{(d_{\mathrm{c}}+d_{\mathrm{r}}) \times d}$, $\mathbf{c}_t^{\mathrm{KV}} \in \mathbb{R}^{d_{\mathrm{c}}}$, and $\mathbf{k}_t^{\mathrm{r}} \in \mathbb{R}^{d_{\mathrm{r}}}$. $\mathbf{c}_t^{\mathrm{KV}}$ and $\mathbf{k}_t^{\mathrm{r}}$ are stored in the KV cache. To alleviate the memory cost of directly upsampling key-value latent vectors, the upsampling weights are fused into the non-RoPE branch of the query projection,

$$\mathbf{q}_{t,h}^{\mathrm{nrabs}} = \mathbf{q}_{t,h}^{\mathrm{nr}} W_h^{\mathrm{UK}}, \tag{12}$$

where $\mathbf{q}_{t,h}^{\mathrm{nr}} \in \mathbb{R}^{d_{\mathrm{h}}}$, $W^{\mathrm{UK}} \in \mathbb{R}^{d_{\mathrm{h}} \times d_{\mathrm{c}}}$, and $\mathbf{q}_{t,h}^{\mathrm{nrabs}} \in \mathbb{R}^{d_{\mathrm{c}}}$.

Finally, the attention process is:

$$\mathbf{o}_{t,h} = \sum_{j=1}^{t} \mathrm{Softmax}\left(\frac{\mathbf{q}_{t,h}^{\mathrm{nrabs}\top}\mathbf{c}_j^{\mathrm{KV}} + \mathbf{q}_{t,h}^{\mathrm{r}\top}\mathbf{k}_j^{\mathrm{r}}}{\sqrt{d_{\mathrm{h}} + d_{\mathrm{r}}}}\right)\mathbf{c}_j^{\mathrm{KV}} W^{\mathrm{UV}}, \tag{13}$$

$$\mathbf{u}_t = W^{\mathrm{O}}[\mathbf{o}_{t,1}; \ldots; \mathbf{o}_{t,n_{\mathrm{h}}}] \tag{14}$$

where $\mathbf{c}_{[1:t]}^{\mathrm{KV}}$ and $\mathbf{k}_{[1:t]}^{\mathrm{r}}$ are loaded from the cache, $W^{\mathrm{U,V}} \in \mathbb{R}^{d_{\mathrm{c}} \times d_{\mathrm{h}}^{\mathrm{v}}}$, $\mathbf{o}_{t,h} \in \mathbb{R}^{d_{\mathrm{h}}^{\mathrm{v}}}$, $W^{\mathrm{O}} \in \mathbb{R}^{d \times n_{\mathrm{h}} d_{\mathrm{h}}^{\mathrm{v}}}$ is the output projection weight, and $\mathbf{u}_t \in \mathbb{R}^d$ is the final output of MLA.

3.2 Kernel Fusion

To improve runtime efficiency, we apply kernel fusion to reduce launch overhead and minimize global memory traffic caused by intermediate results. Specifically, we will implement the following kernel fusion strategies: (1) fusion of attention score computation; (2) fusion of softmax; (3) fusion of RMSNorm; (4) fusion of rotary position embedding application.

3.2.1 Attention Score

The attention score computation in MLA, i.e.,

$$\frac{\mathbf{q}_{t,h}^{\text{nrabs}\top}\mathbf{c}_j^{\text{KV}} + \mathbf{q}_{t,h}^{\text{r}\top}\mathbf{k}_j^{\text{r}}}{\sqrt{d_{\text{h}} + d_{\text{r}}}}, \tag{15}$$

requires two matrix multiplications and one matrix addition. In a naive PyTorch implementation, each operation would load and store data independently, leading to significant memory access overhead.

To mitigate this, we implement a fused Triton kernel that loads the query and cache vectors only once and writes the final result to memory in a single pass. The kernel parallelizes over query positions, key positions, and attention heads. Each program instance computes a $\text{BLOCK}_\text{L} \times \text{BLOCK}_\text{T}$ tile of the attention score matrix for a fixed (b, h) pair, where b and h denote the batch index and head index, respectively, and BLOCK_L and BLOCK_T are block sizes along query sequence length and key sequence length, respectively. Within each tile, the kernel performs (1) a block-wise matrix multiplication between non-RoPE queries and latent key-value vectors, (2) a block-wise matrix multiplication between RoPE queries and RoPE key embeddings, and (3) an accumulation of both products, followed by a scaling operation. Each operand is loaded from memory only once, and intermediate results are kept in on-chip memory until the final write. This design minimizes global memory access and eliminates redundant kernel launches compared to naive PyTorch implementations.

3.2.2 Softmax

In a naive PyTorch implementation, the masked softmax operation over the attention scores typically involves three separate passes over the data: (1) adding the attention mask to the scores, (2) computing the exponentials and their sum for normalization, and (3) writing back the normalized results. Each step requires independent memory loads and stores, introducing significant global memory access overhead and kernel launch latency.

To address this inefficiency, we implement a fused Triton kernel that performs all three steps in a single pass. For each row corresponding to a (b, l, h) tuple in the attention score tensor of shape $[B, L, H, T]$, where b, l, and h denote the batch index, query sequence index, and head index, respectively, and B, L, H, and T denote the batch size, query sequence length, number of heads, and key sequence length, respectively, the kernel loads the relevant slice of the score matrix and the corresponding attention mask from memory. It then computes the maximum value in the row for numerical stability, accumulates the exponential values, and finally writes back the normalized softmax outputs—all within on-chip memory.

The kernel is parallelized over (b, l, h) triplets, and each program instance processes a tile of length $\texttt{BLOCK}_T$ along the T dimension. By loading each operand only once and eliminating intermediate memory allocations, the fused implementation significantly reduces global memory traffic and improves runtime efficiency over standard PyTorch implementations. The block size $\texttt{BLOCK}_T$ is autotuned at runtime to maximize hardware utilization.

3.2.3 RMSNorm

In standard PyTorch implementations, RMSNorm is typically implemented in multiple steps involving separate kernel launches for squared-sum accumulation, scaling, and element-wise multiplication with the weight vector. These redundant memory loads and stores increase latency and memory bandwidth consumption, especially for long sequence lengths and large batch sizes.

To address this, we implement a fused Triton kernel that parallelizes across the batch and sequence dimensions. The kernel tiles along the normalized dimension and processes a BLOCK_K-wide segment per thread block. Within each tile, it performs two passes: the first computes the sum of squared input elements to derive the inverse root mean square (RMS), and the second applies normalization and scaling using the corresponding weights. Each element is loaded from global memory only once, and all intermediate results are kept in registers until the final output is written back. By eliminating multiple memory accesses and avoiding unnecessary kernel launches, this fused implementation significantly improves the runtime efficiency of RMSNorm.

3.2.4 Rotary Position Embedding

Given precomputed complex frequencies, the RoPE operation applies them to query or key vectors (see Equations 4 and 11) via a real-valued decomposition of complex multiplication. The naive PyTorch implementation involves multiple `view`, `unsqueeze`, and `flatten` operations, which introduce unnecessary memory overhead and degrade runtime efficiency.

To address this, we implement a fused Triton kernel that directly applies RoPE on query or key vectors without reshaping. The kernel parallelizes over the (b, l, h) dimensions,

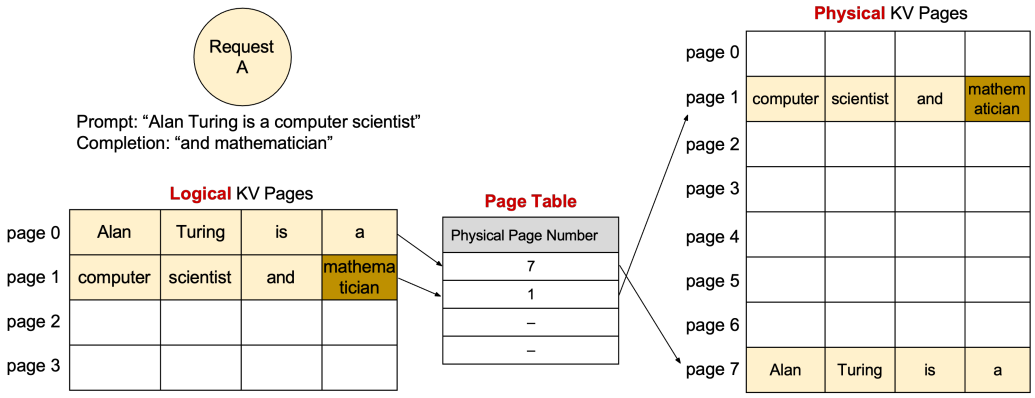

Figure 2. Paged KV cache management during inference. The system dynamically allocates memory at the granularity of pages and maintains a page table to map token positions to physical memory locations for key-value latent vectors and RoPE embeddings. This design minimizes memory fragmentation.

where b, l, and h denote the batch, sequence, and head indices, respectively, and processes a BLOCK_D tile per thread block. For each block, the kernel loads the interleaved real and imaginary parts of the query or key vector, along with the corresponding precomputed cosine and sine values of the complex rotary frequencies. It then performs the rotation via element-wise complex multiplication and writes the result back to global memory in an interleaved real-imaginary pair format. This fused implementation eliminates the overhead of repeated tensor reshaping and intermediate allocations, significantly reducing memory bandwidth consumption and improving runtime efficiency.

3.3 Paged KV Cache Management

We propose a paged KV cache manager, inspired by Kwon et al. (2023), that manages KV cache through dynamic memory allocation and page table indexing. In conventional implementations, the KV cache memory is preallocated based on the maximum sequence length, regardless of the actual sequence length during inference. Such static allocation leads to significant memory inefficiency, as a large portion of the reserved space often remains unused. In contrast, our paged design allocates and manages memory on demand at the granularity of pages, improving memory utilization and reducing fragmentation during decoding. Specifically, during inference, a page table is maintained for each batch entry, as shown in Figure 2. For a given token position, we first compute the logical page index and the offset within the page based on a predefined page size. The logical page index is then mapped to a physical page index, which corresponds to the actual memory region storing the latent key-value vectors and the rotary positional embeddings (RoPE) for the key. If the required logical page does not yet exist in the page table, a new physical page is allocated from the free page pool, and the mapping is updated accordingly. This dynamic page allocation mechanism allows the system to maintain a compact memory footprint throughout generation. When accessing cached information during attention computation, the system retrieves the corresponding key-value vectors and RoPE embeddings through the page table, ensuring efficient and transparent memory access.

We first develop a basic page cache manager that handles each token position sequentially: for every sequence in a batch, it iterates over each position to update and retrieve the corresponding key-value latent vectors and RoPE embeddings. While this naive implementation ensures functional correctness, it incurs significant overhead due to the sequential memory accesses and frequent indexing operations. To address this bottleneck and fully leverage GPU parallelism, we design custom Triton kernels that perform batched cache updates and retrievals in parallel. Specifically, our kernels organize threads such that each thread is responsible for loading or storing a single token position across the batch. By parallelizing memory operations across sequences and batch elements, this optimization substantially improves the memory throughput, reduces latency, and scales better with longer sequence lengths during decoding.

4 Experiments

In our experiments, we used parameter configurations based on the DeepSeek-V3 (DeepSeek-AI, 2024) architecture to ensure real-world relevance. Our implementation uses the following model parameters: $d = 2048$, $n_\text{h} = 16$, $d_\text{h} = 128$, $d_\text{h}^\text{v} = 128$, $d_\text{c} = 512$, $d_\text{c}^\text{q} = 512$, $d_\text{r} = 64$. Furthermore, we performed experiments with sequence lengths of 1024 and 4096 tokens and batch sizes of 4 and 8, which are representative of real-world inference scenarios. The experiments for kernel fusion were performed on a single NVIDIA GeForce RTX 4090 GPU with CUDA 12.6 and PyTorch 2.5.1, and

the experiments for paged KV cache management is performed on NVIDIA H200 with CUDA 12.4 and PyTorch 2.6.0.

4.1 Evaluation Metrics

To evaluate the efficiency of our optimized MLA implementation, we primarily focus on *throughput*, measured in tokens per second. This metric represents the number of tokens that can be processed in one second and is calculated as:

$$\text{Throughput} = \frac{\text{Batch Size} \times \text{Sequence Length}}{\text{Average Forward Pass Time}} \tag{16}$$

Higher throughput values indicate better performance. We measure the average forward pass time over 10 runs after 5 warm-up iterations to ensure stable measurements.

To evaluate the memory savings after applying the paged KV cache manager, we report the *Memory Saving Ratio*:

$$\text{Memory Saving Ratio} = 1 - \frac{\text{Paged Cache Usage}}{\text{No Paged Cache Usage}} \tag{17}$$

where the *Paged Cache Usage* and *No Paged Cache Usage* are computed as

$$\text{Paged Cache Usage} = \#\text{Used Pages} \times \text{Page Size} \tag{18}$$

$$\text{No Paged Cache Usage} = \text{Max Batch Size} \times \text{Max Seq Len}. \tag{19}$$

Here, *#Used Pages* denotes the number of physical memory pages currently occupied during inference, *Page Size* represents the fixed capacity of each memory page, and *Max Batch Size* and *Max Seq Len* correspond to the predefined maximum batch size and maximum sequence length respectively under static memory preallocation.

4.2 Kernel Fusion

To comprehensively evaluate our optimizations through kernel fusion, we conduct experiments in two main directions:

4.2.1 Performance Comparison

We measure the throughput of our implementation compared to the official DeepSeek-V3 implementation [2] across different batch sizes (4 and 8) and sequence lengths (512, 1024, 2048 and 4096). This allows us to assess the overall performance gains achieved by our optimizations.

4.2.2 Ablation Study

To understand the contribution of each optimization technique, we perform ablation studies by selectively enabling different kernel fusion strategies:

- **Attention Score Fusion (ASF)**: Fusion of the attention score computation.
- **Softmax Fusion (SF)**: Fusion of the masked softmax operation.
- **RMSNorm Fusion (RF)**: Fusion of the RMSNorm operation.
- **Rotary Position Embedding Fusion (RPEF)**: Fusion of the RoPE computation.

We test different combinations of these optimizations to analyze their individual and combined impacts on throughput.

[2] https://github.com/deepseek-ai/DeepSeek-V3

Table 2. Throughput (token/s) comparison using scientific notation for batch size–sequence length (B–L) combinations. Gain is relative to DeepSeek-V3. OOM denotes out-of-memory under this setting.

B-L	DeepSeek	Ours	Improvement
4-512	7.25×10^5	1.02×10^6	+40.24%
4-1024	4.83×10^5	7.41×10^5	+53.29%
4-2048	2.88×10^5	4.67×10^5	+62.01%
4-4096	1.53×10^5	2.65×10^5	**+73.02%**
8-512	7.35×10^5	1.03×10^6	+40.78%
8-1024	4.84×10^5	7.37×10^5	+52.32%
8-2048	2.87×10^5	4.67×10^5	+62.84%
8-4096	OOM	OOM	OOM

4.3 Paged KV Cache Management

To evaluate the effectiveness of paged KV cache management in optimizing memory usage, we compare the memory saving ratio across different batch sizes (4 and 8) and sequence lengths (512, 1024, 2048, and 4096). The system's predefined maximum batch size and maximum sequence length are 32 and 16384, respectively. The page size for the paged KV cache manager is set to 1024 tokens. The memory saving ratio is evaluated based on the basic implementation of the paged KV cache without Triton optimization.

We further evaluate the throughput improvements brought by optimizing the cache update and retrieval processes with Triton kernels. The baseline is an MLA model equipped with Triton-optimized attention, softmax, RMSNorm, RoPE, and a basic (non-optimized) paged KV cache manager. On top of this baseline, we integrate the Triton-optimized paged KV cache manager to assess the additional throughput gains. The evaluation is conducted using the same batch size and sequence length settings as in the memory usage experiments.

Table 3. Throughput improvement from individual and combined kernel fusion strategies (seq len = 4096, batch size = 4), relative to the DeepSeek-V3 baseline.

Fusion Strategy	Throughput (token/s)	Improvement
DeepSeek-V3	1.53×10^5	–
+ ASF	2.24×10^5	+46.23%
+ SF	1.68×10^5	+9.37%
+ RF	1.54×10^5	+0.25%
+ RPEF	1.54×10^5	+0.61%
All	2.65×10^5	**+73.02%**

Table 4. Memory saving ratios achieved by paged KV cache management under different batch sizes and sequence lengths.

B-L	Memory Saving Ratio
4-512	99.29%
4-1024	99.29%
4-2048	98.58%
4-4096	97.16%
8-512	98.58%
8-1024	98.58%
8-2048	97.16%
8-4096	94.33%

Table 5. Throughput (token/s) comparison between baseline paged KV cache and Triton optimized paged KV cache.

B-L	Baseline	Optimized	Improvement
4-512	4.32×10^4	5.66×10^4	+30.91%
4-1024	4.36×10^4	5.66×10^4	+29.95%
4-2048	4.37×10^4	5.62×10^4	+28.59%
4-4096	4.32×10^4	5.52×10^4	+27.92%
8-512	4.35×10^4	5.68×10^4	+30.55%
8-1024	4.38×10^4	5.67×10^4	+29.56%
8-2048	4.37×10^4	5.62×10^4	+28.60%
8-4096	4.30×10^4	5.54×10^4	+28.99%

5 RESULTS

5.1 Kernel Fusion

5.1.1 Performance Comparison

Table 2 presents the throughput of our optimized MLA implementation compared to the official DeepSeek-V3 implementation across different sequence lengths and batch sizes.

Our optimized implementation achieves significant throughput improvements over the official DeepSeek-V3 implementation across all tested configurations. For sequence length 512 with batch size 4, we observe a throughput improvement of 40.24%. For longer sequences (4096 tokens), the improvement is even more pronounced, with a speedup of 73.02%. For batch size 4, the average improvement across all sequence length configurations is 57.14%, demonstrating the effectiveness of our kernel fusion strategies in enhancing inference efficiency.

Furthermore, our implementation demonstrates more efficient scaling behavior, particularly at larger batch sizes and sequence lengths. As shown in Table 2, the performance advantage increases as the sequence length increases (from 40. 24% at 512 tokens to 73.02% at 4096 tokens with batch size 4), highlighting the scalability of our approach. This suggests that our optimizations are particularly effective for long-context inference scenarios.

5.1.2 Ablation Study

Table 3 shows the throughput achieved by enabling individual kernel fusion strategies. We report the improvement relative to the baseline DeepSeek-V3 implementation for a sequence length of 4096 and batch size of 8.

The ablation study reveals that Attention Score Fusion (ASF) contributes the most significant individual improvement, with a 46.23% throughput increase over the baseline. Softmax Fusion (SF) provides a moderate improvement of 9.37%, while RMSNorm Fusion (RF) and Rotary Position Embedding Fusion (RPEF) contribute relatively minor improvements of 0.25% and 0.61%, respectively.

Interestingly, when all fusion strategies are combined, we observe a throughput improvement of 73.02%, which is greater than the sum of individual improvements (56.46%). We are still investigating the specific mechanisms behind this non-linear improvement.

5.2 Paged KV Cache Management

As shown in Table 4, the paged KV cache manager dramatically reduces memory consumption compared to conventional static preallocation, highlighting its effectiveness in improving memory efficiency during inference.

As shown in Table 5, the Triton-optimized paged KV cache consistently improves throughput across different settings, demonstrating the effectiveness of parallelized batched cache updates and retrievals.

6 CONCLUSION

In this project, we presented an optimized implementation of Multi-head Latent Attention (MLA) to improve inference efficiency. Our work focused on four key aspects: (1) efficient implementation of MLA by absorbing the upsampling weights into queries, (2) performing system-level kernel fusion for critical MLA components using Triton, (3) developing a paged KV cache management system for dynamic

memory allocation, and (4) optimizing cache updates and retrievals through batched parallel operations with Triton kernels. Experimental results demonstrate that our optimized MLA achieves up to a 73% improvement in memory throughput. Furthermore, paged KV cache management reduces memory usage by over 90% compared to static preallocation. With additional Triton-based batched updates and retrievals, the memory throughput is further improved by approximately 30% relative to the basic implementation.

7 Limitation and Future Work

One limitation of our study is that, due to computational resource constraints, we evaluated the optimizations primarily on standalone modules rather than integrating them into a full-scale model and benchmarking in a realistic inference environment. Additionally, while the paged KV cache manager with batched parallel operations improves memory efficiency, its runtime performance still lags behind the preallocated cache implementation without page table management. Future work includes extending the evaluation to full model inference with large-scale pretrained models, and further improve the efficiency of the paged KV cache.

8 Contribution

The contributions of each team member are summarized as follows:

- **Yike Tan**:
  - Implemented the RMSNorm kernel and paged KV cache management.
  - Fixed errors in the Softmax kernel and contributed to test code development.
  - Contributed to the kernel fusion and tiling parts of the proposal; refined the overall proposal.
  - Contributed to RMSNorm kernel fusion and experiments for the midterm report.
  - Reviewed the final report; contributed to paged attention management, experiments, poster template, and printing.
- **Qingzheng Wang**:
  - Implemented the basic MLA and benchmark code.
  - Developed the Attention Score and RoPE kernels.
  - Optimized paged cache updates and retrievals.
  - Contributed to the MLA, evaluation plan, and timeline in the proposal.
  - Authored sections on MLA, Attention Score, RoPE kernel fusion, and introduction for the midterm report.
  - Contributed to paged KV management, experiments, and figures/tables for the final report.
  - Designed the figures for the poster.
- **Xun Wang**:
  - Implemented the Softmax kernel.
  - Contributed to the introduction section of the proposal.
  - Contributed to Softmax kernel fusion and experiments for the midterm report.
  - Contributed to related works and contribution sections of the final report.
  - Contributed to motivation, MLA implementation, and kernel fusion sections of the poster.

References

Ainslie, J., Lee-Thorp, J., De Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. GQA: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

DeepSeek-AI. Deepseek-v3 technical report, 2024.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Shao, Z., Dai, D., Guo, D., Liu, B., Wang, Z., and Xin, H. DeepSeek-V2: A strong, economical, and efficient mixture-of-experts language model, 2024.

Shazeer, N. Fast Transformer decoding: One write-head is all you need, 2019.

Su, J., Lu, Y., Pan, S., Wen, B., and Liu, Y. RoFormer: Enhanced transformer with rotary position embedding, 2021.

Tillet, P., Kung, H.-T., and Cox, D. D. Triton: An intermediate language and compiler for tiled neural network computations. *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Zhang, B. and Sennrich, R. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.