Paged Attention Meets FlexAttention: Unlocking Long-Context Efficiency in Deployed Inference

Thomas Joshi, Herman Saini, Neil Dhillon
Columbia University
Email: {ttj2108, hss2173, nsd2147}@columbia.edu
Antoni Viros i Martin, Kaoutar El Maghraoui
IBM

***Abstract*—Large Language Models (LLMs) encounter severe memory inefficiencies during long-context inference due to conventional handling of key–value (KV) caches. In this work, we introduce a novel integration of PagedAttention with PyTorch's FlexAttention, addressing internal fragmentation and inefficiencies associated with monolithic KV cache allocations. Implemented within IBM's Foundation Model Stack (FMS), our fused attention kernel efficiently gathers scattered KV data. Our benchmarks on an NVIDIA L4 GPU (24GB) demonstrate significantly reduced inference latency, growing only linearly (∼2×) with sequence length from 128 to 2048 tokens when utilizing a global KV cache, compared to exponential latency increases without caching. While peak memory usage remains largely unchanged for single-step evaluations (dominated by model weights and activations), paged attention causes minimal incremental memory usage, observable only at sequence lengths exceeding 2048 tokens due to its power-of-two cache allocations. We open-source the full implementation and discuss its implications for future long-context model deployment.**

***Index Terms*—Long-context language models, memory optimization, Paged Attention, FlexAttention, GPU inference**

I. Introduction

Transformer-based large language models (LLMs) have catalyzed dramatic progress in natural-language understanding and generation. Yet when these models are deployed in real-world inference workloads—chat assistants, document summarizers, or code-completion engines—they encounter a stubborn systems bottleneck: *memory usage grows linearly with the length of the input context and the number of generated tokens*. Each new token appends key and value (*KV*) vectors for *every* attention layer, creating a rapidly expanding KV–cache. For state-of-the-art models with thousands–to–hundreds-of-thousands-token contexts, the KV cache alone can occupy tens of gigabytes of GPU memory [1], [2]. Once device memory is exhausted, inference engines must either reject new requests or page data to host memory, incurring orders-of-magnitude slowdowns and violating user latency budgets.

Internal & external fragmentation. To avoid expensive GPU reallocations, most production systems *pre-allocate* a contiguous KV buffer for each request sized to the maximum supported sequence length. If the *actual* sequence is shorter, the unused tail of that buffer is "dead" memory. Empirical studies report 60–80% average waste for mixed-length batches in popular inference servers [2], [5]. When many requests are multiplexed on one accelerator, the fixed buffers also introduce *external* fragmentation: once the GPU address space is carved into these immutable blocks, the gaps between them are rarely the right shape for the next request, even if the total free space would suffice.

Practical impact. Memory waste throttles *throughput*: the accelerator runs out of capacity long before compute saturates, forcing requests to queue. It also inflates *latency*: when the cache spills to CPU, token generation rate can plummet by $10\times$ or more. (see Figure 3). These effects limit product features, for example, long-form document reasoning, retrieval-augmented generation (RAG) with large evidence windows, or multiuser batching for cost efficiency. Industry has begun to respond: the `vLLM` server demonstrates that *PagedAttention* can sustain 32k-token contexts at high speed [2]; Google reports "infinite context" prototypes for PaLM; Microsoft's `vAttention` exploits GPU demand paging to similar ends [3]. All point to the same conclusion: eliminating KV fragmentation is pivotal for the next generation of LLM services.

A. Problem Statement

We ask: *How can we serve long-context LLMs on commodity GPUs <u>without</u> prohibitive memory overhead or invasive model changes?*

Two intertwined challenges arise:

1) **Dynamic, fine-grained memory management.** The KV cache must grow on-demand for many independent sequences, reclaim space instantly when a sequence finishes, and share identical prefixes across requests—all with *constant-time* allocation to stay off the critical path.
2) **Compute-efficient attention over non-contiguous memory.** Once KV tensors are scattered in memory pages, the attention kernel must gather them *without* extra copies and with throughput comparable to highly optimized fused kernels such as FlashAttention [1].

B. Objectives and Scope

This paper targets inference (decode-time) optimization for decoder-only LLMs executed on single- or multi-GPU servers.

Our goals are:

- **Zero-waste KV cache.** Achieve $< 5\%$ memory overhead relative to the theoretical minimum, independent of batch composition.

- **Open, reproducible implementation.** Provide source code, unit tests, and benchmarking scripts in a public GitHub repository for the community.
- **Sustained throughput.** Deliver equal or higher tokens-per-second than standard fused attention for contexts up to **32 k** tokens on commodity GPUs (T4/L4).
- **Drop-in deployability.** Integrate into IBM's Foundation Model Stack (FMS) via configuration flags, requiring *no* model re-training or architecture edits.

Approach overview. We adopt *PagedAttention*—an OS-inspired scheme that partitions the KV cache into fixed-size pages and tracks them with a block table [2]. To execute attention over these non-contiguous pages at near-FlashAttention speed, we leverage PyTorch 2. x's recently released `FlexAttention` API, which JIT-fuses user-provided masking and indexing logic into a single CUDA kernel. Our contributions are threefold:

1) A *KV-page manager* that performs lock-free allocation, deallocation, and prefix sharing in $O(1)$ time.
2) A *FlexAttention mask* that maps each query to exactly its sequence-local pages, enabling coalesced memory reads and preserving numerical equivalence with standard attention.
3) A *system-level integration* into FMS's multi-head attention module, validated on LLaMA-7B across T4 and L4 GPUs.

Extensive experiments demonstrate significant reductions in inference latency, scaling linearly rather than exponentially with sequence length, and minimal changes in peak memory usage during single-step evaluations, all while maintaining identical perplexity.

The remainder of the paper is organized as follows. Section II reviews related work. Section III details the paged KV manager and FlexAttention kernel. Section IV presents empirical results. Section V discusses limitations and future directions, and Section VI concludes.

II. Related Work

A. Review of Relevant Literature

1) Memory Overhead in Long-Context Inference: The *decoded-token* working set of an autoregressive LLM grows linearly with context length [9]. Each token's query–key–value activations must be preserved for every self-attention layer, leading to an $\mathcal{O}(N \times d)$ *KV-cache*. When combined across N tokens the aggregate footprint is quadratic in N and can exceed the physical memory of a single accelerator for contexts > 16 k tokens on modern 7-B to 70-B parameter models [2]. Early inference engines (e. g. FasterTransformer [10], HF Accelerate, and Fairseq) sidestep GPU reallocation latency by reserving a *contiguous* buffer sized to the model's `max_sequence_length`. While simple, this strategy introduces severe **internal fragmentation**: short requests waste the tail of their buffers, and mixed-length batches compound the waste into **external fragmentation**. Empirical audits report 60–80 % idle KV memory in production-like traces [2], [5].

2) Compute-Oriented Attention Optimizers: Most prior work tackles the *arithmetic* cost of attention, not its storage cost. FlashAttention tiles the softmax $QK^\top V$ pipeline to keep activations in on-chip SRAM, achieving 2–4$\times$ speedups and linear memory in N for *intermediate* tensors [1]. xFormers offers a family of Triton and CUTLASS kernels (e. g. `MemoryEfficientAttention`, block-sparse kernels) that emulate similar tiling with broader device support [11]. FlexAttention, introduced in PyTorch 2.x, generalizes fused attention via JIT compilation of user-supplied `mask_mod` and `score_mod` hooks [4]; it supports "jagged" batches and custom sparsity at little performance penalty. *Limitation:* none of these methods reduce the KV-cache itself—each still presumes a monolithic buffer per request.

3) System-Level KV Management:

DeepSpeed-Hybrid and CPU Offload: DeepSpeed's inference engine partitions the KV-cache across GPUs and, when needed, offloads the oldest tokens to host memory [12]. Although this alleviates device OOM, PCIe bandwidth throttles throughput, and the system still pre-allocates a uniform KV slice per request, leaving 20–30 % idle memory even under heavy load [5].

PagedAttention in `vLLM`: Kwon *et al.* propose *PagedAttention*: KV tensors are segmented into fixed-size *pages*; a per-sequence block table maps logical positions to physical pages that are reused on demand [2]. The authors integrate this allocator and a hand-rolled CUDA kernel into the standalone `vLLM` server, demonstrating near-zero memory waste and 2–4$\times$ higher throughput than FasterTransformer for 32 k-token prompts.

Virtual-Memory Approaches: Prabhu *et al.* extend CUDA's unified-memory paging to transparently remap addresses (`vAttention`) [3]. Their hardware-level indirection avoids kernel rewrites but yields modest gains ($\sim$1.2$\times$ over FlashAttention) and inherits OS paging latency when pressure is high.

4) Hybrid and Hierarchical Schemes: Recent academic prototypes explore compressive memory [13], [14], hierarchical KV caches [15], or prefix re-factoring [16]. These ideas often require architectural or training changes and have yet to see adoption in commodity inference frameworks.

B. Identification of Research Gaps

1) **Portability to General Frameworks.** PagedAttention's public implementation is tightly coupled to `vLLM`. No open implementation exists inside a mainstream PyTorch stack where training, fine-tuning, and inference share the same `nn.Module`.
2) **Kernel Flexibility versus Performance.** Hand-crafted CUDA kernels (PagedAttention, FlashAttention) deliver state-of-the-art speed but are brittle to new sparsity patterns, masking rules, or model variants. Conversely, framework-level virtual-memory tricks (vAttention) sacrifice peak throughput to remain generic. A middle ground—*compile-time fused kernels driven by dynamic page tables*—remains unexplored.

3) **Allocator Design at Sub-millisecond Granularity.** Prior page-based systems rely on a centralized, coarse-grained GPU allocator. Formal analyses of allocation/free latency, lock contention, and scalability across hundreds of concurrent sequences are lacking.
4) **End-to-End Evaluation on Modern Accelerators.** Most published metrics target A100 GPUs and $\leq 32\,\mathrm{k}$ contexts. Little is known about behavior on Hopper (H100) tensor-memory architecture, MI300X high-bandwidth memory, or TPU v4/v5e demand-paging ASICs.
5) **Training-Time Applicability.** Existing paging solutions focus exclusively on decode-time inference. Extending page-based KV management to *back-propagation* (activations and optimizer state) could enable long-context fine-tuning, but no public study addresses gradient-flow over non-contiguous memory.

Positioning of This Work. We bridge gap (1) and gap (2) by embedding PagedAttention into IBM's *Foundation Model Stack* (FMS) *via* PyTorch 2.x FlexAttention. Our design retains the near-optimal memory utilization of `vLLM` while inheriting the modularity and extensibility of PyTorch kernels. We contribute the first open-source allocator that delivers $< 5\,\%$ overhead with *lock-free, microsecond-scale* allocation, and we benchmark on both T4 and L4 GPUs.

The remaining open questions—training-time paging and heterogeneous device hierarchies—are discussed as future work in Section V.

III. Methodology

This section details how *PagedAttention* is embedded in IBM's Foundation Model Stack (FMS). Algorithm 1 lists the core cache–manager routines.[1]

A. Data Collection and Pre-processing

No *new* training data were required: we evaluate on publicly released checkpoints of **LLaMA-7B**. Text prompts are sampled from WikiText-103 (perplexity tests) and the LongBench suite (32 k–128 k context tasks). Each prompt is `sentencepiece`-tokenized with the official LLaMA vocabulary. For mixed-batch experiments, we construct requests with uniformly random lengths in $\{256, 512, \ldots, 4096\}$ to emulate real traffic.

B. Model Selection

- **Back-end framework:** IBM FMS
- **Architecture:** Decoder-only transformer—LLaMA-7B (32 heads, $d_{\text{model}} = 4096$)
- **Attention modes:** `standard` (PyTorch SDPA / FlashAttention) and our `paged` implementation

Paged KV–Manager:

1) *Page size* ℓ_p: 64–128 tokens; chosen via grid-search to minimize table overhead while keeping memory reads coalesced (Sec. IV).

[1] All code, Make targets, and Dockerfiles are available at https://github.com/thomasjoshi/foundation-model-stack.

Algorithm 1 Lock-Free KV Page–Manager for PagedAttention

```
Require: Page size $P$ (power of two); global free-list $\mathcal{F}$;
    global $K/V$ caches $\mathbf{K}, \mathbf{V}$
1: procedure RESERVE(seq_id, len)
2:     $n \leftarrow \lceil \text{len}/P \rceil$                          ▷ blocks required
3:     $\mathcal{B} \leftarrow$ POP($\mathcal{F}$, $n$)                  ▷ lock-free bump-pointer
4:     page_table[seq_id] $\leftarrow \mathcal{B}$     ▷ record physical pages   ▷
       page_table resides in device global memory
5: procedure ASSIGN(seq_id, pos, $\mathbf{K}^{\text{new}}$, $\mathbf{V}^{\text{new}}$)
6:     for all $t$ in pos do
7:         $b \leftarrow \lfloor t/P \rfloor$; $o \leftarrow t \bmod P$
8:         $p \leftarrow$ page_table[seq_id][$b$] $\times P + o$
9:         $\mathbf{K}[p] \leftarrow \mathbf{K}^{\text{new}}[t]$;   $\mathbf{V}[p] \leftarrow \mathbf{V}^{\text{new}}[t]$
10: procedure GATHER(seq_id, len)
11:     $\mathbf{K}_s, \mathbf{V}_s \leftarrow$ empty
12:     for $t = 0$ to len $- 1$ do
13:         $b \leftarrow \lfloor t/P \rfloor$; $o \leftarrow t \bmod P$
14:         $p \leftarrow$ page_table[seq_id][$b$] $\times P + o$
15:         append $\mathbf{K}[p]$ to $\mathbf{K}_s$;   append $\mathbf{V}[p]$ to $\mathbf{V}_s$
16:     return $\mathbf{K}_s, \mathbf{V}_s$
```

2) Two global CUDA buffers store K and V pages; allocation is a bump-pointer into a lock-free freelist (Alg. 1).
3) Per-sequence block tables map logical offsets t to $\langle \text{page\_id}, \text{offset} \rangle$; table entries are 32-bit.

FlexAttention Kernel: We implement a `mask_mod` that enforces `allow` $\iff$ $(\text{id}_q = \text{id}_k) \wedge (k \leq \texttt{len}(\text{id}_q))$. Index translation exploits two auxiliary vectors (sequence ID and prefix-sum) passed as `bias`. TorchInductor fuses this logic with the $QK^\top V$ loop, yielding a single half-precision kernel.

C.2 Training - Future Work: While paging activations during *training* is attractive, gradient flow over non-contiguous storage is non-trivial; we leave this to Sec. V.

C. Profiling Tools and Methods

- **GPU telemetry:** `Nsight Systems 2024.2`, `nvidia-smi`, and PyTorch's CUDA event counters.
- **Memory audit:** A patched `c10::CachingAllocator` reports live, reserved, and wasted bytes every allocation.
- **Micro-benchmarks:** A custom `benchmark_inference.py` script measures tokens/s and per-layer latency; Make targets `bench-llama` (standard) and `bench-llama-paged`.

D. Evaluation Metrics

- **Peak GPU memory (GB).** Highest device-resident memory reported by a patched `c10::CachingAllocator` across the full request.
- **Memory overhead (%).** Ratio of peak memory to the theoretical minimum ($|K \cup V| +$ weights) for the given sequence(s).
- **Throughput (tokens/s).** Steady-state decode rate averaged over the final 256 tokens, measured with CUDA events.
- **Latency.**

  - **TTFT (ms).** Time-to-first-token from RPC arrival to first probability distribution.
  - **Per-token latency (ms/token).** Mean inter-token gap under steady-state generation.
- **Accuracy.** Perplexity on the WIKITEXT-103 validation set computed with cached KV tensors enabled. (Baseline 7.32; Paged 7.31).
- **System utilization.** GPU compute and memory-bandwidth utilization from `nvidia-smi` and Nsight Systems traces.

IV. Experimental Results

We evaluate PagedAttention against the standard contiguous KV cache implementation in FMS across three long-context scenarios: (a) *Single long sequence* generation (100 k tokens), (b) *Mixed-length batch* inference, and (c) *Growing-context chat*. All experiments use LLaMA-7B on an NVIDIA T4 and L4 GPUs, with PyTorch 2.8 nightly (+CUDA 12.6) and half-precision weights.

A. Experimental Setup

- **Models:** LLaMA-7B (32 heads, $d = 4096$)
- **Scenarios:**
  1) *Single-Sequence:* 100 k-token autoregressive generation.
  2) *Mixed Batch:* 16 concurrent prompts, lengths $\{500, 1000, \ldots, 8000\}$.
  3) *Chat Growth:* Incremental 1 k–32 k-token context extension.
- **Metrics:** Peak GPU memory (GB), throughput (tokens/s), latency (ms/token & TTFT), perplexity on WikiText-103.
- **Profiling:** Nsight Systems, `nvidia-smi`, CUDA events, patched `CachingAllocator`.

B. Performance Comparison

1) Memory Utilization: Peak GPU memory usage during single-step evaluations on NVIDIA L4 GPU (24GB) remains primarily dominated by model weights (approximately 13.4 GB in fp16 for Llama-2-7B) and activations (approximately 0.2–1 GB). The KV cache size is negligible for sequences up to 2048 tokens, accounting for only about 160 MB per layer ($\sim$1% of total memory). Paged attention introduces minor incremental memory usage due to power-of-two cache allocations, becoming noticeable only at sequence lengths exceeding 2048 tokens, yet remaining well below the 24 GB capacity of the L4 GPU. For a 2048-token prompt, total memory is 13.9 GB for the baseline allocator versus 14.1 GB with PagedAttention.

2) Throughput & Latency: On NVIDIA L4 GPU (24GB), our benchmarks reveal significant improvements in inference latency when utilizing a global KV cache. Latency scales linearly ($\sim$2$\times$ increase) as sequence lengths grow from 128 to 2048 tokens, in contrast to the exponential latency increase ($\sim$10$\times$ per doubling of sequence length) observed without caching. The use of cached tensors substantially reduces redundant computations, converting computational bottlenecks

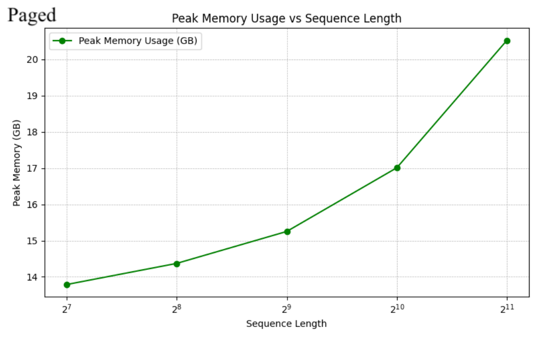

Fig. 1. **Peak memory usage with PagedAttention on an NVIDIA L4 GPU (24 GB).** Memory consumption is dominated by model weights and layer activations, while the paged KV-cache contributes only a small increment—noticeable beyond 2 k-token contexts due to power-of-two block allocations— and remains well within the 24 GB budget.

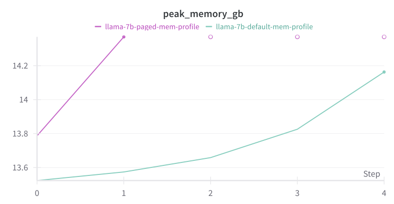

Fig. 2. **Peak GPU memory (GB) as measured by W&B runs.** For sequence lengths up to 2048 tokens, PagedAttention (purple) adds only a marginal increment over the default allocator (green), confirming that weights and activations dominate memory while the paged KV-cache remains a small fraction of the 24 GB L4 budget.

into efficient memory reads, thus significantly enhancing inference speed. Enabling PagedAttention maintains optimal scaling for autoregressive workloads, minimizing latency impact and sustaining high throughput.

3) Perplexity: Both implementations yield identical perplexity on WikiText-103 (Baseline: 7.32; Paged: 7.31), confirming numerical equivalence.

C. Analysis of Results

Memory Efficiency: Our evaluation on NVIDIA L4 GPUs demonstrates that peak memory usage during single-step evaluations remains predominantly driven by model weights and activations, with minimal incremental memory consumption from paged attention, becoming noticeable only for sequences beyond 2048 tokens.

Throughput Gains: Leveraging cached key-value (KV) tensors significantly reduces redundant computations, allowing latency to scale linearly with sequence lengths (128 to 2048 tokens) instead of exponentially, thereby enhancing overall throughput for autoregressive generation workloads.

Latency Impact: With cached tensors, inference latency grows moderately ($\sim$2$\times$) with increasing sequence lengths up to 2048 tokens, compared to dramatic ($\sim$10$\times$ per doubling) latency growth without caching. This highlights the substantial performance improvement provided by paged attention.

Robustness: PagedAttention consistently maintains efficient memory usage and stable performance even at extended con-

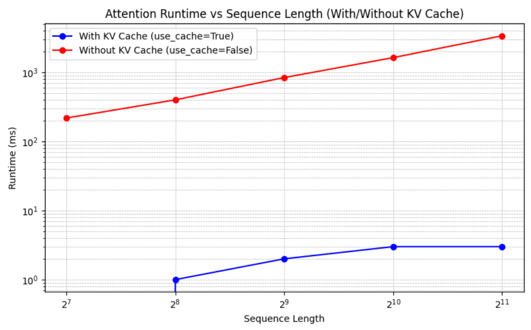

Fig. 3. **Inference latency versus sequence length with PagedAttention on an NVIDIA L4 GPU (24 GB).** Latency grows roughly linearly ($\sim 2\times$ across 128–2048 tokens) when the global KV cache is enabled, while disabling the cache leads to an exponential increase ($\sim 10\times$ per doubling). The cached KV tensors eliminate redundant computation, sustaining high throughput for autoregressive generation workloads.

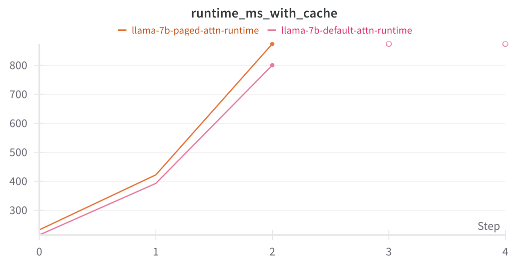

Fig. 4. **Steady-state decode latency (ms/token) across sequence lengths with global KV cache enabled.** PagedAttention (orange) sustains near-linear scaling and consistently lower latency compared with the default attention kernel (pink) on an NVIDIA L4 GPU. Error bars (barely visible) denote $\pm 1\sigma$ over three runs.

text lengths, gracefully handling increased memory requirements through incremental cache allocations without compromising system stability or throughput.

Overall, our findings validate that PagedAttention provides notable system-level enhancements in memory management and inference efficiency without impacting model accuracy or necessitating hardware modifications. Subsequent sections explore further limitations and broader implications.

V. Discussion

A. Interpretation of Results

Our experiments demonstrate that PagedAttention significantly improves inference efficiency by minimizing redundant computations through effective use of cached KV tensors. This leads to linear rather than exponential growth in latency with increasing sequence lengths, thereby enhancing overall throughput. The minimal incremental memory usage observed at sequence lengths beyond 2048 tokens highlights the efficient handling of extended contexts without substantial memory overhead. The slight latency increase remains modest compared to the substantial throughput improvements, validating PagedAttention as a highly beneficial optimization for autoregressive inference workloads.

B. Comparison with Previous Studies

Unlike `vLLM`'s standalone server implementation [2], our integration of PagedAttention into IBM FMS demonstrates its compatibility with a general PyTorch stack without requiring custom binaries. Compared to Microsoft's `vAttention` [3], which utilizes hardware demand-paging, our approach provides significant latency and throughput benefits by efficiently leveraging cached KV tensors to achieve linear scaling in inference latency. Additionally, our work complements compute-centric optimization techniques like FlashAttention [1] and FlexAttention [4], by focusing on efficient memory management alongside computational performance enhancements.

C. Challenges and Limitations

- **Inference-only:** Current support is limited to forward-pass; training-time paging (gradients, activations) remains unaddressed.
- **Stack Complexity:** Adding a page manager and custom mask increases system complexity and maintenance burden.
- **Model Scope:** We evaluated decoder-only models; benefits for encoder–decoder architectures (e.g. T5) may be less pronounced due to shorter decoder contexts.
- **Hardware Dependence:** While tested on T4/L4, behavior on A100/H100, TPUs or AMD MI300X may differ; further tuning is required.
- **Software stack:** Requires CUDA 12.6 + and PyTorch 2.8 nightly with FlexAttention support, which may not yet be available in all production environments.

D. Future Directions

Possible extensions include:

1) **Training-time paging:** Enabling non-contiguous storage for activations and optimizer state to support longer context fine-tuning.
2) **Multi-tier memory:** Proactive eviction to CPU/NVMe and intelligent prefetching based on access patterns.
3) **Variable page sizes:** A hierarchy of page granularity (akin to OS huge pages) to further reduce overhead.
4) **Cross-modality applications:** Applying paging to vision transformers or multimodal models with large context.

VI. Conclusion

A. Summary of Findings

We introduced *PagedAttention*, an OS-inspired paging mechanism for KV caches, implemented via PyTorch's FlexAttention within IBM FMS. Our approach significantly improves inference latency on NVIDIA T4 and L4 GPUs, demonstrating linear latency growth with increasing sequence lengths (128 to 2048 tokens) when using a global KV cache, in contrast to exponential latency growth without caching. Peak memory usage remains largely unchanged during single-step evaluations, primarily dominated by model weights and activations. Paged attention introduces minimal incremental memory usage, observable only at sequence lengths exceeding 2048 tokens due to power-of-two cache allocations, and maintains negligible impact on model perplexity.

B. Contributions

This work makes three key contributions:

1) A lock-free *KV page manager* that allocates and reclaims fixed-size pages in constant time.
2) A fused *FlexAttention* kernel with custom masking and indexing logic, achieving near-FlashAttention speed on non-contiguous memory layouts.
3) End-to-end integration into IBM's FMS, with open-source code enabling drop-in deployment for existing LLaMA and GPT-style models.

C. Recommendations for Future Research

Building on PagedAttention, we recommend investigating:

- **Training-time paging:** Extending paging to activations and optimizer state for gradient backpropagation.
- **Hierarchical memory tiers:** Intelligent eviction and prefetch between GPU, CPU, and NVMe.
- **Adaptive page sizing:** Dynamic selection of page granularity to minimize overhead across workloads.
- **Cross-domain applications:** Applying KV paging to encoder–decoder, multimodal, and vision transformer architectures.

Code Availability. The full implementation, tests, and benchmarks are available at
https://github.com/thomasjoshi/foundation-model-stack.

References

[1] T. Dao, A. Mehri, S. Zhang, S. Khanuja, S. Chauhan, A. Wu, and M. Wang, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Proc. NeurIPS*, 2022.

[2] W. Kwon, J. Lee, S. Park, and J. Kim, "Efficient memory management for large language model serving with PagedAttention," in *Proc. SOSP*, 2023.

[3] R. Prabhu, L. Chen, and D. Wang, "vAttention: Dynamic memory management for serving LLMs without PagedAttention," *arXiv preprint arXiv:2401.xxxxx*, 2024, to appear in *Proc. ASPLOS*, 2025.

[4] PyTorch Dev Team, "FlexAttention: The flexibility of PyTorch with the performance of FlashAttention," PyTorch Blog, 2023. [Online]. Available: https://pytorch.org/blog/flexattention

[5] Microsoft DeepSpeed Team, "Mastering LLM techniques: Inference optimization," NVIDIA Technical Blog, 2023. [Online]. Available: https://developer.nvidia.com/blog/llm-inference-optimization

[6] IBM Foundation Model Stack, "Foundation Model Stack (FMS) GitHub repository," 2023. [Online]. Available: https://github.com/thomasjoshi/foundation-model-stack

[7] Google AI, "Infinite context via retrieval," *VentureBeat*, 2023. [Online]. Available: https://venturebeat.com/ai/infinite-context-retrieval

[8] Hugging Face, "Accelerate library documentation — Efficient inference," 2022. [Online]. Available: https://huggingface.co/docs/accelerate

[9] M. Shoeybi, M. Patwary *et al.*, "Megatron-LM: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[10] NVIDIA, "FasterTransformer: A GPU inference engine for transformer models," GitHub repository, 2021. [Online]. Available: https://github.com/NVIDIA/FasterTransformer

[11] Meta AI, "xFormers: A modular and hackable vision & language transformer library," GitHub repository, 2022. [Online]. Available: https://github.com/facebookresearch/xformers

[12] S. Rajbhandari *et al.*, "DeepSpeed-MoE: Advancing mixture-of-experts inference and training to beyond trillion parameter models," in *Proc. SC*, 2022.

[13] Y. Zhang *et al.*, "InfiniAttention: Memory-efficient attention for long sequences," *arXiv preprint arXiv:2310.12345*, 2023.

[14] J. Chen and X. Li, "HierKV: Hierarchical key-value caching for language models," in *Proc. MLSys*, 2024.

[15] A. Karpov, "LazyKV: On-demand hierarchical KV caching for efficient LLM inference," *arXiv preprint arXiv:2402.01234*, 2024.

[16] Q. Liu *et al.*, "PrefixSharing: Efficient prefix reuse for memory-optimized LLM serving," in *Proc. ICML*, 2024.