vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention

Ramya Prabhu
Microsoft Research
Bengaluru, India

Ajay Nayak*
Indian Institute of Science
Bengaluru, India

Jayashree Mohan
Microsoft Research
Bengaluru, India

Ramchandran Ramjee
Microsoft Research
Bengaluru, India

Ashish Panwar
Microsoft Research
Bengaluru, India

Abstract

PagedAttention is a popular approach for dynamic memory allocation in LLM serving systems. It enables on-demand allocation of GPU memory to mitigate KV cache fragmentation — a phenomenon that crippled the batch size (and consequently throughput) in prior systems. However, in trying to allocate **physical** memory at runtime, PagedAttention ends up changing the **virtual** memory layout of the KV cache from contiguous to non-contiguous. Such a design leads to non-trivial programming and performance overheads.

We present vAttention — an approach that mitigates fragmentation in physical memory while retaining the virtual memory contiguity of the KV cache. We achieve this by decoupling the allocation of virtual and physical memory using CUDA virtual memory management APIs. We also introduce various LLM-specific optimizations to address the limitations of CUDA virtual memory support. Overall, vAttention is a simpler, portable, and performant alternative to PagedAttention: it supports various attention kernels out-of-the-box and improves LLM serving throughput by up to 1.23× compared to the use of PagedAttention-based kernels of FlashAttention-2 and FlashInfer.

CCS Concepts: • **Software and its engineering** → **Maintaining software**; *Virtual memory*; *Software design techniques*; • **General and reference** → **Performance**; • **Computing methodologies** → *Neural networks.*

Keywords: Large language models; KV cache; fragmentation; memory management

ACM Reference Format:
Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramchandran Ramjee, and Ashish Panwar. 2025. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3669940.3707256

*Contributed to this work as an intern at Microsoft Research India.

ASPLOS '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707256

System/library and issues related to PagedAttention
vLLM [51]: Pioneered PagedAttention. Despite being in an actively maintained code repository, vLLM's PagedAttention kernel is up to 2.8× slower than FlashAttention-2 (Table 7). Furthermore, changing block size changes the execution time of the kernel by as much as 1.9× (Figure 3).
FlashAttention-2 [42]: PagedAttention-based prefill kernel is up to 37% slower than the non-paged kernel (Figure 2) while the decode kernel is up to 12% slower. Initial attempts to add paging support failed unit tests [7].
FlashAttention-3 [67] / SDPA in cuDNN-9 [59]: State-of-the-art attention kernels for NVIDIA Hopper architecture did not support PagedAttention when released.
TensorRT-LLM [6]: Serving throughput dropped by more than 10% in a Python front-end [5]. Recommends using the C++ front-end. Even with C++, we observe up to 5% higher latency in some cases with PagedAttention.
FlashInfer [77]: PagedAttention-based prefill kernel is up to 42% slower than the non-paged kernel (Figure 2).

Table 1. The PagedAttention approach requires an application to explicitly manage dynamically allocated physical memory, including re-writing of attention kernels. These examples highlight the complexity, performance and maintenance challenges associated with this approach.

1 Introduction

Large Language Models (LLMs) are being deployed in a wide range of applications e.g., chat bots, search engines and coding assistants [9, 10, 18, 19, 25, 41, 61]. Given the size and scale of modern LLM deployments, optimizing inference has become extremely important [36, 46, 48, 49, 51, 63, 78, 82].

Batching is a powerful technique to boost LLM serving throughput [35, 51, 63, 78]. However, achieving a large batch size requires careful allocation of GPU memory. For each request, the serving framework stores the activations of all

the tokens processed so far in GPU memory and reuses them for generating subsequent tokens. This is called the KV cache [36, 63, 78] which accounts for a majority of GPU memory usage during inference. Efficiently allocating GPU memory for the KV cache is challenging for two reasons. First, the per-request KV cache grows slowly (one token per iteration), and second, a request's decode length (or its total KV cache size) is not known ahead of time.

Prior systems like Orca [78] and FasterTransformer [15] allocate memory for each request based on the maximum context length supported by the model (e.g., Yi-34B model supports context length of up to 200K [32]). However, the number of decode tokens generated are far less in practice, e.g., the average decode length for the chat-based sharegpt dataset is 415 tokens [35]. Therefore, static memory allocation could create severe internal fragmentation, limiting batch size and serving throughput.

Inspired by demand paging in OS-based virtual memory systems, vLLM introduced PagedAttention [51] that allocates small blocks of GPU memory on demand i.e., when previously allocated blocks are fully utilized and the model continues to generate more tokens. This approach provides a near-perfect solution for mitigating fragmentation and hence, PagedAttention has become the de facto standard for dynamic memory allocation in LLM serving systems, e.g., TensorRT-LLM, HuggingFace TGI, LightLLM [4, 6, 28] etc.

However, we show that PagedAttention faces a fundamental consequence of dynamic memory allocation: *dynamically allocated objects are not guaranteed to be contiguous*. Note that user-level objects are allocated in virtual memory. Therefore, in trying to enable dynamic allocation of **physical** memory, PagedAttention ends up changing the **virtual** memory layout of KV cache from contiguous to non-contiguous. We argue that this approach has several pitfalls (§3). First, it requires rewriting attention kernels, i.e., to enable de-referencing all tokens of the non-contiguous KV cache. Second, it forces developers to implement a memory manager in the serving framework, i.e., to stitch together dynamically allocated virtual memory blocks. Third, it adds runtime overhead in the critical path of both CPU and GPU execution. Table 1 provides empirical evidence and real-world experiences to support these arguments.

The fundamental issue with PagedAttention and prior systems is that they rely on the reservation-based memory allocation method exposed by the GPU runtime. In this method (used by `cudaMalloc`), the runtime allocates both virtual and physical memory on the GPU meaning that *physical memory is allocated even if the corresponding virtual memory is not accessed*. This is in stark contrast to OS-based demand paging [52, 62]. We show that separating the allocation of virtual and physical memory allows for more effective KV cache memory management. To support our claim, we introduce vAttention (§5) — an approach that stores KV cache in contiguous virtual memory without committing physical memory ahead-of-time. vAttention decouples the allocation of virtual and physical memory using the CUDA virtual memory management (VMM) APIs [12].

In building vAttention, we find that using CUDA VMM support for KV cache management poses two key efficiency challenges for an LLM serving system (§6). First, memory allocation using CUDA VMM APIs incurs high latency because each allocation involves a round-trip to the OS kernel. We tackle latency issues with several LLM-specific optimizations such as overlapping memory allocation with compute, opportunistically allocating pages ahead of time, and deferring memory reclamation. Second, CUDA supports memory allocation only at the granularity of large pages, i.e., in multiples of 2MB. Use of large pages can create significant fragmentation. We address this challenge by modifying the open-source CUDA unified virtual memory driver, adding support for smaller 64KB pages. Our evaluation shows that use of 64KB pages has no negative impact on the performance of attention kernels, i.e., we do not find any evidence of TLB thrashing. Together, these optimizations mitigate fragmentation while hiding the latency cost of on demand memory allocation, making vAttention a simpler, portable and performant alternative to PagedAttention.

Overall, we make the following contributions:

- We present vAttention – a memory management approach that retains the virtual contiguity of KV cache while enabling dynamic allocation of physical memory. Our implementation of vAttention in vLLM seamlessly adds dynamic memory allocation support to various unmodified attention kernels.
- We compare vAttention against PagedAttention-based alternatives of vLLM, FlashAttention-2 and FlashInfer on Yi-6B, Llama-3-8B and Yi-34B with 1-2 A100 GPUs. Using FlashAttention-2's non-paged attention kernel, vAttention outperforms vLLM by up to 1.99× in decode throughput. In long-context scenarios, it also improves the end-to-end LLM serving throughput by up to 1.18× and 1.23× over PagedAttention based kernels of FlashAttention-2 and FlashInfer
- We demonstrate the portability benefit of vAttention with the recently launched FlashAttention-3 kernel (FA3 [67]). FA3 is optimized for the NVIDIA Hopper architecture and was not released with PagedAttention support. vAttention supports FA3 out-of-the-box, leading to 1.26 – 1.5× higher throughout over PagedAttention based FlashAttention-2.

2 Background

2.1 Large Language Models

Given an input sequence, an LLM predicts the probability of an output sequence: a sequence is a set of tokens [51]. Each inference request begins with a prefill phase that processes

Symbol	Definition
N	Number of layers on a particular worker
H	Number of KV heads on a particular worker
D	Dimension of each attention head
P	Number of bytes needed to store one element
B	Maximum batch size
L	Maximum context length supported by the model
L'	Context length of a request seen thus far

Table 2. Notations used in the paper.

prompt tokens in parallel and produces the first output token of the request. Thereafter, the decode phase iteratively processes the output token generated in the previous step and produces the next output token in every iteration [36].

LLMs are built atop one of the variants of the transformer architecture [73]; an LLM consists of multiple transformer blocks. Internally, a transformer block contains two types of operators: position-wise and sequence-wise [78]. The former category includes feed-forward network, layer normalization, activation, embedding layer, output sampling layer, and residual connections whereas *attention* is a sequence-level operator. We primarily focus on attention since it is the primary consumer of GPU memory in LLM inference. Table 2 summarizes the notations used in the paper.

For the attention operator, the model computes the query, key and value vectors from a given sequence of tokens $(x_1, x_2, ..., x_K) \in \mathbb{R}^{K \times E}$ where E represents the embedding size of the model. For each x_i, query, key and value vectors are computed as follows:

$$q_i = W_q x_i, \quad k_i = W_k x_i, \quad v_i = W_v x_i \tag{1}$$

The resulting k_i and v_i are appended to the key and value vectors of the prior tokens of the corresponding request, producing two matrices $K, V \in \mathbb{R}^{L' \times (H \times D)}$ where L' represents the context length of the request seen so far, H is the number of KV heads of the model on a worker and D is the dimension of each KV head. Then, attention is computed as follows:

$$Attention(q_i, K, V) = softmax(\frac{q_i K^T}{scale})V \tag{2}$$

The attention score is computed separately for each request in the batch. A request executes until the model generates a special end-of-sequence token or reaches the maximum context length for the request. Note that in each iteration of a request, all its preceding k_i and v_i are needed to compute attention. Hence, an inference engine stores the k_i and v_i vectors in memory to reuse them across iterations. We refer to this state of all layers collectively as KV cache. In systems prior to PagedAttention, the K cache (or V cache) at each layer of a worker is typically allocated as a 4D tensor of shape $[B, L, H, D]$ where B refers to batch size and L refers to the maximum possible context length of a request.

2.2 Fragmentation and PagedAttention

Serving LLMs with high throughput requires careful allocation of GPU memory. This is challenging because the total context length of a request is not known in advance. Serving systems worked around this challenge by pre-reserving KV cache space assuming that each context is as long as the maximum length supported by the model (e.g., 200K for Yi-34B-200K). vLLM shows that this strategy is prone to severe internal fragmentation. In fact, vLLM showed that prior reservation is suboptimal even if the context lengths are known in advance. This is because the per-request KV cache grows one token at a time and hence prior reservation wastes memory over the entire lifetime of a request.

Inspired by the OS-based virtual memory systems, vLLM proposed PagedAttention to mitigate fragmentation by dynamically allocating memory for the KV cache. PagedAttention splits KV cache into fixed-sized blocks and allocates memory for one block at a time. This way, vLLM allocates only as much memory as a request needs, and only when required – not ahead-of-time.

GPUs and page sizes: NVIDIA GPUs support multiple page sizes in the hardware [22, 58, 65, 79]. A single call to a CUDA VMM API can allocate one or more physical pages, which we refer to as a page-group. We use page-groups to support multiple allocation granularities for the KV cache, similar to how multiple page sizes are commonly used in conventional OS-based virtual memory systems [52, 62].

3 Issues with the PagedAttention Approach

Despite being inspired by demand paging, the PagedAttention approach is different from it: *PagedAttention implements demand paging in user space whereas conventional demand paging is transparent to applications.* This section elaborates on issues that arise with such an approach.

3.1 Requires Re-writing the Attention Kernel

Conventional implementations of the attention operator assume that the two input tensors K and V (Equation 2) are stored in contiguous memory. By departing from the conventional memory layout, PagedAttention requires an implementation of the attention operator to be modified so as to compute attention scores over non-contiguous KV cache blocks. Writing correct and performant GPU kernels can be challenging for most programmers [7].

Being a fundamental building block of the transformer architecture, the attention operator has witnessed a tremendous pace of innovation in the systems and ML communities for performance optimizations [2, 37–40, 42, 43, 47, 49, 68, 77, 80], and this trend is likely to continue. In the PagedAttention model, keeping up with new research requires continued efforts in porting new optimizations to a PagedAttention-aware implementation. Production systems can therefore easily fall behind research, potentially losing performance

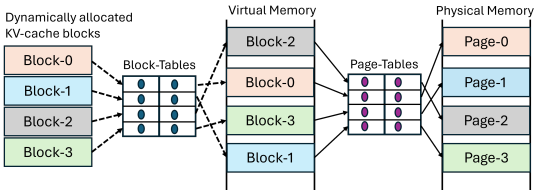


Figure 1. PagedAttention involves two layers of memory management: one in user space and one in OS kernel space.

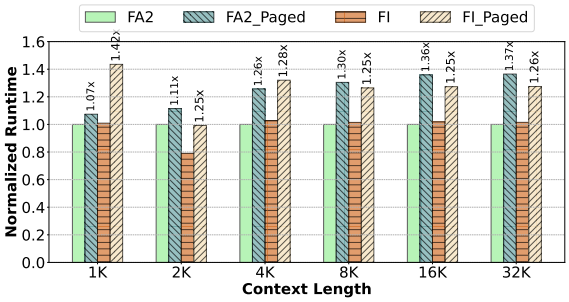


Figure 2. Overhead of PagedAttention in prefill kernels (model: Llama-3-8B, one A100 GPU). Numbers on top show overhead over the corresponding non-paged implementation of FlashAttention-2 (FA2) and FlashInfer (FI).

and competitive advantage. To provide an example, Table 7 shows that the paged kernel of vLLM is already up to 2.8× slower than the FlashAttention-2 kernel [37].

3.2 Adds Redundancy in the Serving Framework

PagedAttention makes an LLM serving system responsible for managing the mappings between KV cache and dynamically allocated memory blocks. For example, consider a request that allocates four KV cache blocks over time (left half of Figure 1). These blocks are usually non-contiguous in virtual memory. During the computation of Equation 2, PagedAttention kernel needs to access all the elements of the four KV cache blocks. To facilitate this, the serving system needs to track the virtual memory addresses of KV cache blocks and pass them to the attention kernel at runtime. This approach effectively requires duplicating what the operating system already does for enabling virtual-to-physical address translation (right half in Figure 1).

3.3 Performance Overhead

3.3.1 Runtime overhead on the GPU. PagedAttention slows down attention computation by adding extra code in the critical path of execution. For example, the vLLM paper acknowledges that the PagedAttention-based implementation was 20 – 26% slower than the corresponding none-paged FasterTransformer kernel, primarily due to the overhead of looking up Block-Tables and executing extra branches (see Figure 18a in [51]). In addition, Figure 2 shows that incorporating PagedAttention has also added a significant performance overhead in other state-of-the-art kernel libraries. For example, PagedAttention based prefill kernels of FlashAttention-2 and FlashInfer are up to 37% and 42% slower than the non-paged kernels in the corresponding libraries. Our analysis reveals that the number of instructions executed in PagedAttention kernels is 7 – 13% higher than the non-paged kernels. Caching page indices also increases register pressure, causing register spilling.

To highlight another example of difficulty involved in writing an efficient paged kernel, Figure 3 shows that the performance of vLLM's paged decode kernel is significantly worse with large block sizes of 64 and 128. Our analysis indicates that this is likely due to L1 cache efficiency: smaller

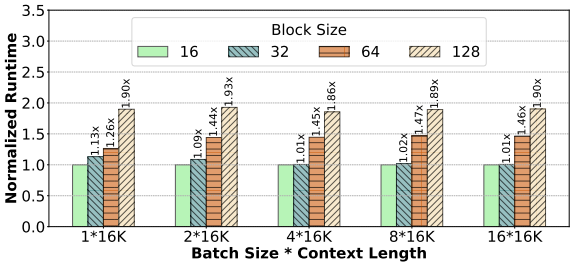


Figure 3. Latency of vLLM's paged decode kernel is sensitive to block size (model: Llama-3-8B, one A100 GPU).

blocks have a higher memory bandwidth utilization due to higher hit rates in the L1 cache.

3.3.2 Runtime overhead on the CPU. Implementing an additional memory manager can add performance issues in the CPU runtime of the serving system. We refer to a few real-world examples and our own observations on vLLM to corroborate this argument.

To enable PagedAttention, a serving system needs to supply Block-Tables to the attention kernel. In vLLM, the latency of preparing a Block-Table depends on batch composition and grows proportional to `max_num_blocks` × `batch_size` where `max_num_blocks` refers to the number of KV cache blocks in the longest request of the batch. This is because vLLM manages a Block-Table as a 2D tensor and aligns the number of KV cache blocks in each request by padding unoccupied slots with zeros. If a batch contains a few long and many short requests, such padding results in a significant overhead. In our earlier experiments, we observed that Block-Table preparation in vLLM was contributing 30% latency in decode iterations. While a recent fix [17] has mitigated some of this overhead, we find that it can still be as high as 10%. High overhead of PagedAttention has also been found in TensorRT-LLM, degrading throughput by 11% [5]. This issue was attributed to the Python runtime of TensorRT-LLM and

moving to a C++ runtime can mitigate the CPU overhead. However, doing so requires non-trivial programming effort.

Overall, this section shows that PagedAttention adds a significant programming burden while also being inefficient. In vAttention, we propose a more principled approach to dynamic KV cache memory management. However, before delving into vAttention, we first highlight some of the fundamental characteristics of LLM serving workloads from a memory management perspective.

4 Insights into LLM Serving Systems

To understand the memory allocation pattern of LLM serving systems, we experiment with Yi-6B running on a single A100 GPU, and Llama-3-8B and Yi-34B running on two A100 GPUs with tensor-parallelism (TP). We set the initial context length of each request to 1K tokens, vary the batch size from 1 to 320 and measure the throughput and memory requirement of the decode phase (see §6 for our discussion and optimizations for the prefill phase).

Observation-1: *KV cache memory requirement is predictable on a per-iteration basis.* Due to auto-regressive decoding, once a request enters the decode phase, its KV cache size increases uniformly by one token per iteration. This allows the serving system to determine in advance if additional memory will be required during the iteration's execution.[1]

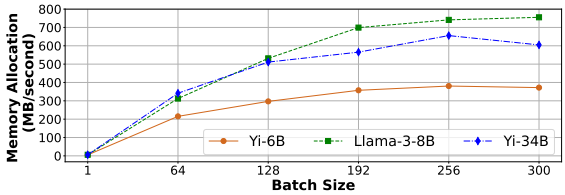
Observation-2: *KV cache does not require high memory allocation bandwidth.* The memory footprint of a single token across all layers is typically few 10s-100s of kilobytes of memory. For example, the per-token memory footprint of Yi-6B, Llama-3-8B and Yi-34B is 64KB, 128KB and 240KB, respectively. Further, each iteration runs for 10s-100s of milliseconds implying that a request requires at most a few megabytes of memory per second. While batching improves system throughput [35, 36, 63, 82], the number of tokens generated per second plateaus beyond a certain batch size (Figure 4a). This implies that the memory allocation bandwidth requirement also saturates at large batch sizes (e.g., at 256 for Yi-34B). For all the models we studied, we observe that the highest memory allocation rate is at most 750MB per second (Figure 4b). vAttention leverages these observations to optimize KV cache memory management.

5 vAttention: Design and Implementation

Our primary observation is that *physical memory fragmentation can be avoided without making KV cache non-contiguous in virtual memory.* To realize this, vAttention decouples the allocation of virtual memory from physical memory by leveraging system support for demand paging (instead of implementing demand paging in user space, as in PagedAttention).

[1]Note that this prediction is possible only at the granularity of individual iterations; the total KV cache requirement of a request remains unknown as it depends on the total number of output tokens in the request.

(a) Decode throughput.



(b) Rate of memory allocation.

Figure 4. Decode throughput (top) and the rate of physical memory allocation (bottom) saturate at large batch sizes.

5.1 Design Overview

vAttention employs distinct allocation policies for virtual and physical memory. Specifically, we allocate a large contiguous buffer for the KV cache in virtual memory ahead-of-time (similar to systems prior to PagedAttention) while deferring the allocation of physical memory to runtime (similar to PagedAttention). This design preserves virtual contiguity of KV cache without fragmenting physical memory. Note that this approach could fragment and waste virtual memory. However, this is not an issue since virtual memory is abundant, e.g., modern 64-bit systems provide a 128TB user-addressable virtual memory per process.[2]

5.1.1 Pre-reserving virtual memory. Since virtual memory is abundant, we pre-allocate it in size that is large enough to hold the KV cache of the maximum batch size (configurable) that needs to be supported. In doing so, we assume that each request's context length is same as the maximum supported by the model.

5.1.2 Number of virtual memory buffers. A serving framework maintains separate K and V tensors for each layer of the model. Therefore, we reserve $2 \times N$ buffers on a worker where N is the number of layers managed by that worker.

5.1.3 Size of a virtual memory buffer. The maximum size of a buffer is $BS = B \times S$ where B is the maximum batch size and S is the maximum size of a single request's per-layer K cache (or V cache) on a worker. Further, $S = L \times H \times D \times P$, where L is the maximum context length supported by the model, H is the number of KV heads on a worker, D is the dimension of each KV head and P is the

[2]64-bit systems use only 48 bits for virtual addresses today, providing a per-process virtual memory space of 256TB which is divided equally between the user space and (OS) kernel space.

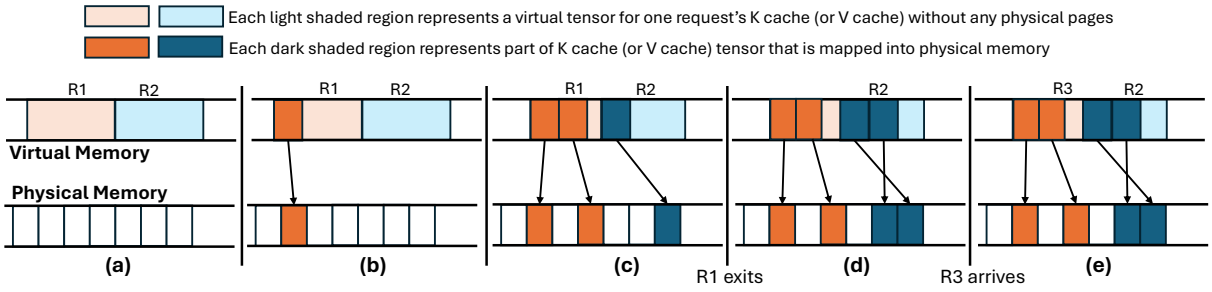


Figure 5. Dynamic memory management in vAttention for a single K cache (or V cache) tensor. (a) shows a virtual tensor for a batch of two requests with no physical memory allocation yet. (b) R1 is allocated one physical page. (c) R1 is allocated two pages and R2 is allocated one page. (d) R1 has completed but vAttention does not reclaim its memory (deferred reclamation). (e) when R3 arrives, vAttention assigns R1's tensor to it which is already backed by physical memory.

CUDA VM APIs	vAttention VM APIs	Description	Latency (microseconds) 64KB	128KB	256KB	2MB
cuMemAddressReserve *	vMemReserve *	Allocate a buffer in virtual memory	18	17	16	2
cuMemCreate *	vMemCreate *	Allocate a handle in physical memory	1.7	2	2.1	29
cuMemMap	vMemMap	Map a physical handle to a virtual buffer	8	8.5	9	2
cuMemSetAccess	-	Enable access to a virtual buffer	-	-	-	38
cuMemUnmap	-	Unmap physical handle from a virtual buffer	-	-	-	34
cuMemRelease *	vMemRelease *	Free physical pages of a handle	2	3	4	23
cuMemAddressFree *	vMemFree *	Free a virtual memory buffer	35	35	35	1

Table 3. CUDA VMM APIs. * represents APIs that we use once while instantiating or terminating the serving framework. Rest of the APIs are used for (un)mapping physical memory pages at runtime. CUDA APIs (prefixed with cu) support only 2MB pages, whereas our CUDA extension APIs (prefixed with v) support fine-grained allocations.

number of bytes based on model precision (e.g., P=2 for FP16/BF16). As an example, consider Yi-34B with FP16 and two-way tensor-parallelism (TP-2). In this case, $N=60, H=4, D=128, P=2$ (8 KV heads of Yi-34B are split evenly on two GPUs), and maximum supported context length $L = 200K$. For this configuration, $S = 200MB$ ($200K * 4 * 128 * 2$). Assuming $B = 500$, the maximum size of each buffer per-worker is $BS = 100GB$ ($500 \times 200MB$). Therefore, the total virtual memory requirement for 60 layers is 120 buffers of 100GB each (12TB total). Note that the size of virtual address space available grows with the number of workers, e.g., with two TP workers, the total size of user-addressable virtual address space is 256TB. Therefore, virtual memory is always plentiful to satisfy large allocations. Figure 5 shows an example of how vAttention allocates physical memory pages dynamically.

5.2 Leveraging CUDA Virtual Memory Support

The standard GPU memory allocation interface `cudaMalloc` does not support demand paging, i.e., it allocates virtual memory and physical memory at the same time. However, recent CUDA versions provide programmers a fine-grained control over managing virtual and physical memory, including support for decoupling their allocations [12, 45]. We leverage these low-level APIs.

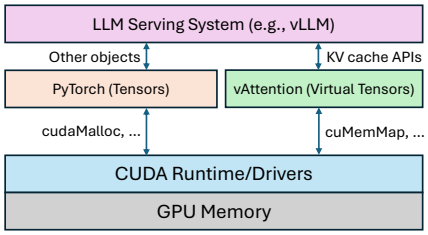


Figure 6. An LLM serving system interacts with vAttention for KV cache management with a set of simple APIs listed in Table 4. All other memory objects (e.g., activations) are allocated by the standard PyTorch caching allocator.

5.2.1 CUDA virtual memory APIs.

Table 3 provides an overview of CUDA VMM APIs that allow decoupling the allocation of virtual memory from physical memory. The allocation granularity depends on the page size used by the GPU. Further, the size of a virtual memory buffer or a physical memory handle must be a multiple of the physical memory allocation granularity. Physical memory pages can be allocated to (or de-allocated from) sub-regions in a virtual memory buffer independently of other sub-regions.

APIs	Description
`init`	Initializes vAttention with model parameters. arguments: N, B, L, H, D, P, `page-group-size`. return value: a list of KV cache tensors.
`alloc_reqid`	Allocates an unused `reqId` and marks it active arguments: None return value: an integer `reqId`
`free_reqid`	Frees a `reqId` and marks it inactive arguments: an integer `reqId` return value: None
`step`	Ensures KV cache tensors are backed by physical pages up to the current context length of each active request arguments: an array of size B containing sequence lengths of each `reqId` return value: 0 (success), -1 (failure).

Table 4. Key APIs that vAttention exposes to a serving framework for dynamic KV cache memory management.

5.2.2 Extending PyTorch caching allocator.

KV cache is a collection of tensors. In current deep learning frameworks such as PyTorch, a tensor allocated via APIs such as `torch.empty` comes with pre-allocated physical memory. This is because the PyTorch caching allocator relies on the `cudaMalloc` interface (Figure 6). Relying on the low-level API support from CUDA, we extend the PyTorch caching allocator to allow an application to reserve a virtual memory buffer for a tensor without committing physical memory ahead-of-time. We refer to tensors allocated via these APIs as virtual tensors.

5.2.3 Request-level KV cache indexing.

A virtual tensor represents the K cache (or V cache) of a layer for the maximum batch size B. In these tensors, different requests occupy different non-overlapping sub-regions (say sub-tensors). We locate the sub-tensor of a request with a unique integer identifier `reqId` that lies in the range of 0 to $B-1$ (note that at most B requests run simultaneously). The K cache (or V cache) offset of a request's sub-tensor in the virtual tensor of the entire batch is `reqId` $\times S$ where S is the maximum size of per-layer K cache (or V cache) of a request on a worker. The request identifier `reqId` is allocated by vAttention.

5.3 Serving LLMs with vAttention

We build vAttention as a Python library that internally uses a CUDA/C++ extension for interacting with CUDA drivers. Our library exposes a set of simple APIs to the serving framework (shown in Figure 6, Table 4 and Algorithm 1). For simplicity, we discuss the use of these APIs from a single worker's perspective; all workers behave the same.

5.3.1 Initial setup.

When the serving framework starts, each model worker loads the vAttention library and configures it with model parameters N, H, D, P, B and a preferred *page-group size* (§6.2) via the `init` API (line 4 in Algorithm 1).

Algorithm 1 Using vAttention in a serving framework.

```
1:  max_batch_size ← B
2:  cache_seq_len ← [0]*B
3:  req_batch_idx ← dict()
4:  vattention.init(config_params)
5:  while !request_pool.is_empty() do
6:      for R_i in new_requests do
7:          if can_schedule(R_i) then
8:              idx ← vattention.alloc_reqid()
9:              req_batch_idx[R_i] ← idx
10:             cache_seq_len[idx] ← prompt_len(R_i)
11:         end if
12:     end for
13:     vattention.step(cache_seq_len)
14:     model.forward()
15:     for R_i in active_requests do
16:         idx ← req_batch_idx[R_i]
17:         if is_complete(R_i) then
18:             cache_seq_len[idx] ← 0
19:             vattention.free_reqid(idx)
20:         else
21:             cache_seq_len[idx] + = 1
22:         end if
23:     end for
24: end while
```

Internally, vAttention reserves $2 \times N$ virtual tensors on the worker, as shown in Figure 5(a), where N is the number of layers hosted by the worker. These virtual tensors are reserved for the lifetime of the serving application. In addition, vAttention also pre-allocates physical memory pages at each worker during initialization. However, these pages are not mapped into the KV cache at this point.

5.3.2 Scheduling a new request.

When a new request is scheduled for the first time, the serving framework obtains a new `reqId` from vAttention via `alloc_reqid` (line 8). All subsequent memory management operations of the request are tagged with this `reqId`.

5.3.3 Model execution.

Before scheduling a batch for execution, the framework needs to ensure that the KV cache sub-tensors of each active request are backed by physical memory (Figure 5(b) and (c)). For this purpose, before dispatching the first kernel of an iteration to the GPU, the framework invokes the `step` API (line 13), specifying the current context length of each request (context length is set to 0 for each inactive `reqId`). Internally, vAttention ensures that enough physical pages are mapped for each active `reqId` before returning execution back to the framework. If vAttention cannot satisfy the memory demand, it returns with a failure in response to which a serving framework can preempt one or more requests to allow forward progress

(this is similar to vLLM's default behavior). We leave more sophisticated policies such as swapping out KV cache to CPU memory as future work.

Depending on whether a request is in the prefill phase or decode phase, different amount of physical memory may need to be mapped for a given iteration. The prefill phase processes the input tokens of a given prompt in parallel. Therefore, the amount of physical memory needed to be mapped depends on the number of prompt tokens being scheduled. If the total K cache size of all prompt tokens at one layer of the model is s and page-group size is t, then each worker needs to ensure that at least $(s+t-1)/t$ page-groups are mapped in each of the $2 \times N$ KV cache sub-tensors of the given `reqId`.

For a request in the decode phase, the number of new page-groups required is at most one per virtual tensor. This is because each iteration produces only one output token for a request. vAttention internally tracks the number of page-groups mapped for each request and maps new page-groups only when prior page-groups are about to be exhausted.

5.3.4 Request completion. A request terminates when it reaches user specified or the maximum context length supported by the model, or when the model produces a special end-of-sequence token. The framework notifies vAttention of a request's completion with `free_reqid` (line 19). Internally, vAttention may unmap the physical pages of a completed request or defer them to be freed later (§6.1.2).

Supporting continuous batching: Continuous batching poses one challenge in computing attention in our design. When a request from somewhere in the middle of a batch exits, it creates an unused hole in the virtual tensors of KV cache. This layout is not supported by implementations that expect the query (Q) and KV cache to be of the same size in batch (B) dimension. However, FlashAttention provides rich API support to address this issue; its argument `cache_batch_idx` allows Q and KV cache to have different batch sizes and also be arranged in arbitrary order (i.e., batch index 0 in Q can be mapped to batch index 1 in KV cache). vAttention benefits from this API support in terms of both performance and ease of programming; when the request composition of a batch changes, we update `cache_batch_idx` of running requests such that their Q tensors map to their respective KV cache based on their `reqId`.

6 Optimizations

There are two challenges in using CUDA virtual memory support for serving LLMs. First, invoking CUDA VMM APIs at runtime incurs high latency. Second, `cuMemCreate` currently allocates memory only at the granularity of large pages, i.e., multiples of 2MB. Use of large pages can waste physical memory due to internal fragmentation. This section details a set of simple-yet-effective optimizations that we introduce to overcome these challenges.

6.1 Hiding Latency of Memory Allocation

The serving framework invokes the `step` API in every iteration. The latency of `step` depends on the number of page-groups that need to be mapped into KV cache. Consider, for example, that the KV cache of one request needs to be extended for Yi-34B which has 60 layers. This requires 120 calls to `cuMemMap` + `cuMemSetAccess` each of which takes about 40 microseconds. Therefore, growing the KV cache of one request by new page-groups (two per layer) adds about 5 millisecond latency to the corresponding iteration. The latency overhead grows proportional to the number of requests that need new page-groups in a given iteration. We propose the following optimizations to hide this latency:

6.1.1 Overlapping memory allocation with compute (decode phase). We leverage the predictability of memory demand to overlap memory allocation with computation. In particular, note that each iteration produces a single output token for every decode request. Therefore, memory demand for a decode iteration is known ahead-of-time. Further, in the decode phase, a request requires at most one new page-group for each of its virtual tensors. vAttention keeps track of the current context length and how much physical memory is already mapped for each request. Using this information, it determines when a request would need more memory and uses a background thread to allocate new page-groups when the preceding iteration is executing. For example, consider that a request `R1` would require more physical memory in iteration `i`. When the serving framework invokes `step` API in iteration `i-1`, vAttention launches a background thread that maps page-groups for iteration `i`. Since per-iteration latency is typically in the range of 10s-100s of milliseconds, the background thread has enough time to prepare physical memory mappings for an iteration before it starts executing. This way, vAttention hides the latency of CUDA APIs by performing memory allocations out of the critical path.

6.1.2 Deferred reclamation + eager allocation (prefill phase). We observe that allocating physical memory for the prefill phase can be avoided in many cases. Consider that a request `R1` completed in iteration `i` and a new request `R2` joins the running batch in iteration `i+1`. To avoid allocating page-groups to `R2` from scratch, vAttention simply defers the reclamation of `R1`'s page-groups (Figure 5(d)) and assigns `R1`'s `reqId` to `R2`. This way, `R2` uses the same tensors for its KV cache that `R1` was using – which are already backed by physical pages (Figure 5(e)). Therefore, new allocations are required only if `R2`'s context length is higher than `R1`.

We further optimize the prefill phase by proactively allocating a small number of page-groups ahead of time. For this purpose, we try to keep a certain number of page-groups mapped into the virtual tensors of one of the inactive `reqId`. When a new request arrives, we allocate this `reqId`. At the same time, we identify a new `reqId` to be allocated next and

eagerly map physical page-groups for it. In most cases, these eager allocations obviate the need to allocate physical memory in the critical path of prefill execution. Finally, we trigger memory reclamation only when the number of page-groups cached in vAttention falls below a certain threshold (e.g., less than 10% of GPU memory). We delegate both deferred reclamation and eager allocation to the background thread that the `step` API spawns.

6.2 Mitigating Internal Fragmentation

We mitigate internal fragmentation by reducing the granularity of physical memory allocation. NVIDIA GPUs natively support at least three page sizes: 4KB, 64KB and 2MB [22, 58, 65, 79]. Therefore, in principal, physical memory can be allocated in any multiple of 4KB sizes. The simplest way to achieve this would be to extend the existing CUDA VMM APIs (listed in Table 3) to also support allocating smaller pages (similar to how `mmap` in Linux supports multiple page sizes [52, 62]). Unfortunately, the CUDA VMM APIs are implemented in the closed-source NVIDIA drivers which makes it impossible for us to modify their implementation.

Fortunately, some part of NVIDIA drivers (particularly related to unified memory management) is open-source. Therefore, we implement a new set of APIs in the open-source NVIDIA drivers to mimic the same functionality that existing CUDA APIs provide but with support for multiple page sizes. The second column in Table 3 shows our new APIs: most of our APIs have a one-to-one relationship with existing CUDA APIs except for `vMemMap` that combines the functionality of `cuMemMap` and `cuMemSetAccess`, and `vMemRelease` that combines the functionality of `cuMemUnmap` and `cuMemRelease` for simplicity. In contrast to CUDA VMM APIs, our APIs allocate physical memory in 64KB, 128KB and 256KB sized page-groups. A serving framework can configure a desired page-group size in vAttention while initializing it (we use the standard 2MB pages if the configured page-group size is 2MB). Table 3 shows the latency of each API with different page-group sizes.

7 Evaluation

Our evaluation answers the following questions:

- How does vAttention impact the performance of attention kernels and the overall performance of prefill and decode phases (§7.1, §7.2)?
- How does vAttention impact the end-to-end LLM serving throughput (§7.3, §7.4, §7.5)?
- What is the effect of each of our optimizations (§7.6)?

Models and hardware: We evaluate three models Yi-6B [33], Llama-3-8B [21] and Yi-34B [32]. We conduct most of our evaluation on a single NVIDIA A100 GPU for Yi-6B, and two NVLink-connected A100 GPUs for Llama-3-8B and Yi-34B (see Table 5). Each GPU has 80GB physical memory. We use tensor-parallelism degree of two (TP-2) for both Llama-3-8B and Yi-34B. To demonstrate the portability benefit of vAttention, we use 1–2 H100 GPUs.

Model	Hardware	# Q Heads	# KV Heads	# Layers
Yi-6B	1 A100	32	4	32
Llama-3-8B	2 A100s	32	8	32
Yi-34B	2 A100s	56	8	60

Table 5. Models and hardware used for evaluation.

Evaluation methodology: The computation and memory allocation pattern of the prefill and decode phases is substantially different [35, 46, 82]. Attention kernels used for these two phases are also different and hence we evaluate them separately. The prefill phase requires one time memory allocation potentially spanning multiple pages. In comparison, the decode phase requires incremental memory allocation over the lifetime of a request [51]. We define prefill throughput as the number of prompt tokens processed per second, and decode throughput as the number of tokens generated per second.

Serving framework: For a fair comparison, we use vLLM v0.2.7 as a common serving framework in all our experiments. We integrated state-of-the-art kernel libraries of both FlashAttention-2 v2.5.9 [1, 42, 43] and FlashInfer v0.4.0 [3] as attention back-ends into vLLM, and further added support for dynamic memory allocation via vAttention to their non-paged kernels. While FlashAttention-2 and FlashInfer are both based on the same underlying techniques (e.g., FlashDecoding [2]), they use different Block-Table formats; the former use a simple lookup table whereas the latter uses a compressed Block-Table to optimize lookups.

Baselines: We compare performance obtained by using the non-paged attention kernels (backed by vAttention for dynamic memory allocation), and their paged counterparts. In addition, we also compare against vLLM's decode kernel (note that vLLM does not have a paged prefill kernel). We also profiled each system to find its best performing configuration. Accordingly, we set KV cache block size to 16 for both vLLM and FlashInfer, and 256 for FlashAttention-2. Using a higher block size for vLLM increases its kernel latency by up to 1.9× as shown in Figure 3, and using a smaller block size for FlashAttention-2 paged kernel increases its latency by up to 9%. We find that the choice of page size does not affect vAttention as long as fragmentation is not a concern.

7.1 Prefill Evaluation

We evaluate 4 configurations for prefill: FA2_Paged, FI_Paged, FA2_vAttention and FI_vAttention. Configurations with the "_Paged" suffix represent the PagedAttention-based kernel and those with "_vAttention" use the non-paged kernels of the respective library. Figure 7 shows the prefill throughput for our models. We summarize key findings below.

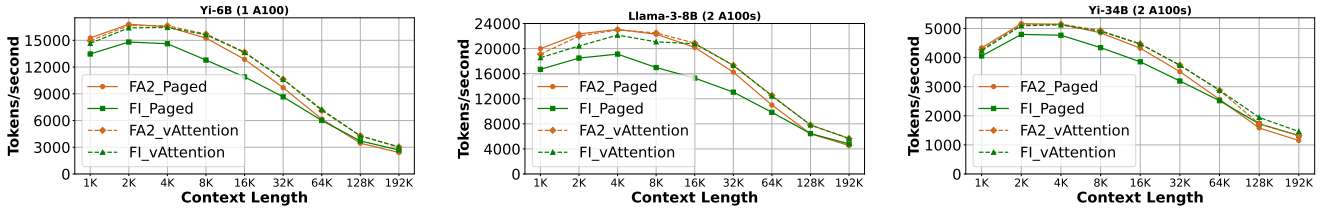


Figure 7. Prefill throughput. vAttention backed systems outperform the paged counterparts of both FlashAttention-2 and FlashInfer. Throughput for longer contexts is lower due to the quadratic complexity of prefill attention.

Model	Context Length	FlashAttention-2 Paged	FlashAttention-2 vAttention	FlashInfer Paged	FlashInfer vAttention
Yi-6B	64K	10.6 (7.0)	9.1 (5.5)	10.9 (6.0)	9.1 (5.4)
	128K	37.9 (30.3)	30.5 (23.1)	35.4 (25.4)	30.7 (23.3)
	192K	81.5 (70.0)	64.6 (53.6)	73.0 (58.3)	65.1 (53.6)
Llama-3 -8B	64K	6.0 (3.4)	5.2 (2.7)	6.7 (3.0)	5.3 (2.8)
	128K	20.4 (15.4)	16.8 (11.6)	20.3 (12.8)	16.8 (11.4)
	192K	43.3 (35.6)	34.8 (26.9)	40.9 (29.7)	34.7 (26.7)
Yi-34B	64K	25.5 (13.2)	22.8 (10.3)	26.0 (11.2)	22.7 (10.1)
	128K	82.8 (56.9)	68.4 (43.2)	76.0 (46.7)	67.4 (42.5)
	192K	170.7 (131.8)	136.9 (98.8)	148.8 (104.7)	134.6 (96.5)

Table 6. Prefill completion and attention (in parenthesis) time with different attention back-ends (unit: seconds).

Model	BS	vLLM	FA2_Paged	FI_Paged	FA2_vAttention
Yi-6B	16	32.3	11.5	15.2	11.3
	32	64.1	25.5	25.4	25.3
Llama-3 -8B	16	17.8	11.9	12.1	11.8
	32	35.3	25.4	23.23	25.3
Yi-34B	12	41.4	17.4	24.1	17.4
	16	55.1	21.7	28.8	21.8

Table 7. Total latency of attention kernel (sum of all layers) per decode iteration (in milliseconds, BS = batch size).

Small contexts: For small contexts, prefill cost is dominated by the linear operators, i.e., attention's contribution is relatively low [36]. Hence, even though vAttention helps speed up attention computation, the throughput of both paged and vAttention back-ends is nearly identical in FlashAttention-2. However, for FlashInfer, we find that using vAttention helps improve prefill throughput even for small contexts. This is because FlashInfer incurs various other sources of overhead in the paged version. First, appending a new K or V tensor to the KV cache requires a single tensor copy operation in vAttention, whereas in a paged implementation, it requires appending one block at a time (the copy operation has been optimized for FlashAttention-2 by vLLM [13]). Second, FlashInfer involves creation and deletion of a few objects for its compressed Block-Tables in every iteration. vAttention avoids such overheads because it maintains KV cache's virtual contiguity, eliminating the need for a Block-Table.

Long contexts: The contribution of attention computation becomes significant at 16K and higher context lengths in our experiments. Therefore, even for FlashAttention-2 back-end, vAttention outperforms the paged counterpart. For example, at context length 192K, FA2_vAttention outperforms FA2_Paged by 1.24×, 1.26× and 1.24× for Yi-6B, Llama-3-8B and Yi-34B, respectively. Similarly, for FlashInfer back-ends, FI_vAttention improves prefill throughput by up to 1.25× and 1.36× for Yi-6B and Llama-3-8B (context length 16K), and 1.17× for Yi-34B (context length 32K).

Attention time: vAttention's improvement in prefill throughput is primarily due to faster attention kernels enabled by a (virtually) contiguous KV cache. This is because the prefill phase of a long prompt is primarily dominated by attention computation, as can be observed by comparing the numbers inside parenthesis with total prefill completion time in Table 6. For FlashAttention-2, nearly all the gains of vAttention are due to faster attention kernels, e.g., prefill gains of 1.5 seconds, 7.4 seconds and 16.9 seconds for Yi-6B are all due to gains in attention computation. vAttention enabled kernels also help with the FlashInfer back-end. In addition, FI_Paged also has other sources of overheads, e.g., in the 14 seconds of total savings (Yi-34B, 192K context), only 7 seconds is due to attention and the rest is due to other sources.

7.2 Decode Evaluation

For decodes, in addition to FA2_Paged and FI_Paged, we also evaluate the throughput obtained with vLLM's decode kernel (the first ever kernel to support PagedAttention). For vAttention, we use FlashAttention-2's non-paged kernel. Unfortunately FlashInfer's non-paged decode kernel has significantly higher latency (up to 14.6×) compared to all these other kernels. Hence, while vAttention can support dynamic memory allocation for FlashInfer's non-paged decode kernel, we omit it for evaluation in this section.

Figure 8 shows the decode throughput of Yi-6B, Llama-3-8B and Yi-34B with varying batch size up to 32 (except for Yi-34B which runs out of memory for batch size 32). We set the initial context length of each request to 16K tokens and calculate decode throughput based on the mean latency of 400 decode iterations.

First, vAttention is on par with the best of PagedAttention as shown by FA2_Paged and FA2_vAttention in Figure 8. In comparison, FI_Paged has somewhat lower throughput and vLLM is the worst for all models and configurations.

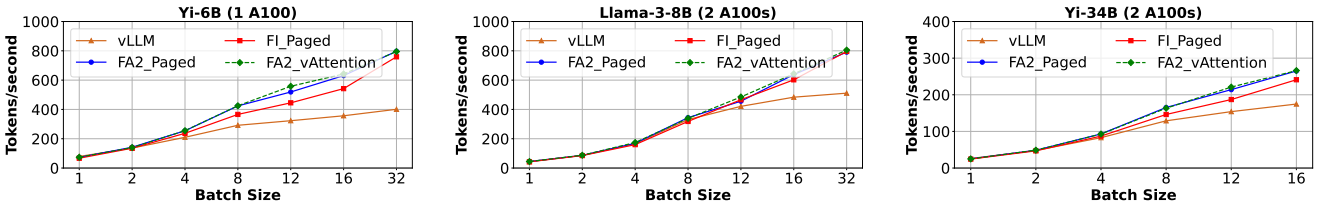


Figure 8. Decode throughput. FA2_vAttention is on par with FA2_Paged (note the overlapping lines) which is the best among all PagedAttention based alternatives, while outperforming FI_Paged and vLLM.

For example, FA2_Paged and FA2_vAttention outperform vLLM by up to 1.99×, 1.58× and 1.53× for Yi-6B, Llama-3-8B and Yi-34B, respectively. The primary reason is that vLLM's decode kernel has significantly higher latency than the FlashAttention-2 based kernels; while FlashAttention-2 has continuously adopted new optimizations (e.g., FlashDecoding [2]), vLLM has lagged behind. For example, Table 7 shows that vLLM's PagedAttention kernel incurs up to 2.8×, 1.5×, and 2.5× higher latency than FlashAttention-2 kernels for Yi-6B, Llama-3-8B and Yi-34B. This is despite vLLM being in an actively maintained open-source serving stack, as well as being used by various companies for serving LLMs. This is an important result that underlines the importance of low software complexity and portability.

Second, relative gains of FA2_vAttention and FA2_Paged increase over vLLM with the batch size, e.g., as batch size increases from 4 to 32 for Llama-3-8B, relative gains increase from 1.05× to 1.58×. This is because the latency of a decode attention kernel is proportional to the total number of tokens in the batch [34]. Therefore, the contribution of attention kernel in the overall latency – hence gains with a more efficient kernel – increase with the batch size. Further, for the same reasons as discussed in §7.1 (faster attention and lower CPU overhead), FA2_vAttention delivers up to 1.23× higher throughput than FI_Paged (Yi-6B, batch size 12).

Finally, note that vAttention is only as good as the state-of-the-art PagedAttention for decode throughput, as compared to prefills where it outperforms PagedAttention. This is because in case of decode attention, paged and non-paged kernels have similar latency as shown in Table 7. We believe this is due to memory bound nature of decode attention, i.e., memory stalls make it possible to hide the effect of additional compute that paging support requires. However, hiding compute overhead in a prefill attention kernel is hard because it is already compute bound (see Table 6 for vAttention's gains in the prefill kernel).

7.3 End-to-end Performance: Offline Scenarios

We measure the end-to-end system throughput in an offline scenario as the number of requests completed per minute. We evaluate a total of 427 long-context requests from the arXiv-Summarization workload trace wherein the total context length per-request varies from 64K tokens to 192K tokens

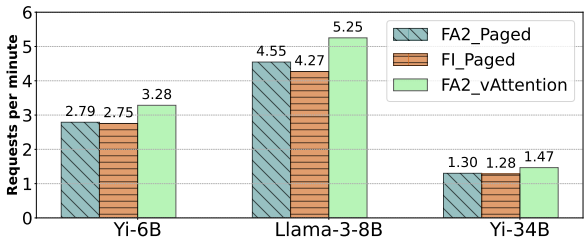


Figure 9. Offline inference throughput while serving long-context requests from the arXiv-Summarization dataset [11].

and the number of output tokens per-request varies from 17 to 5153. The mean prefill to decode token ratio is 356. Figure 9 shows throughput for different attention backends for all three models. FA2_vAttention outperforms FA2_Paged by 1.18×, 1.15× and 1.13×, and FI_Paged by 1.19×, 1.23×, and 1.14× for Yi-6B, Llama-3-8B and Yi-34B, respectively. In general, vAttention's performance gains are proportional to the context length and the prefill to decode token ratio (P:D ratio); these factors determine how much prefill attention contributes to the total runtime. A higher P:D ratio as well as longer context lengths indicate that the workload is more prefill bound. vAttention provides higher gains in such cases.

7.4 End-to-end Performance: Online Scenarios

We evaluate an online inference scenario serving long-context requests from arXiv-Summarization [11]. In total, we run 512 requests wherein the per-request input context length varies from 22K to 45K tokens (mean 29K), the number of decode tokens varies from 6 to 3250 (mean 348) and the mean P:D ratio is 129. We evaluate performance near the serving capacity of the system which denotes the maximum load a system can handle without incurring high queuing delays [35, 51]. We vary input load (queries-per-second or QPS) based on Poisson distribution and schedule requests in first-come-first-serve order. Figure 10 shows the CDF of the end-to-end request execution latency for this experiment. We see that vAttention consistently outperforms both baselines. For example, FA2_vAttention reduces the median request execution latency over FA2_Paged by up to 42% (QPS 0.25) for Yi-6B, by up to 28% (QPS 0.3) for Llama-3-8B and by up to 29% for Yi-34B (QPS 0.1). The primary reason for vAttention's

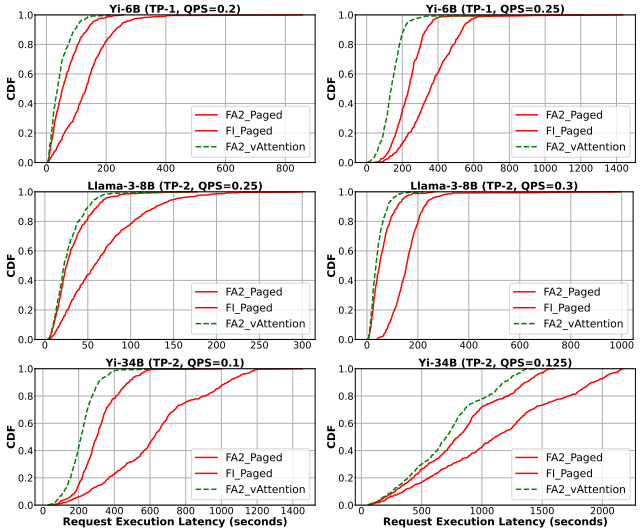


Figure 10. CDF of end-to-end request execution latency in online inference under varying load.

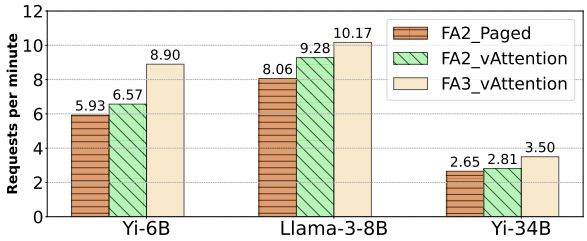


Figure 11. Offline inference throughput on H100 GPUs.

performance gain is that it can compute the prefill phase of new requests faster than the PagedAttention-based systems which substantially reduces queuing delays. Consistent with our prior results, FI_Paged is slower than FA2_Paged and hence our gains our FI_Paged are relatively higher compared to our gains over FA2_Paged.

7.5 Exemplifying Portability: FlashAttention-3

We demonstrate the portability benefit of vAttention with the recently released FA3 kernel [67]. FA3 is optimized for the NVIDIA Hopper architecture and did not support PagedAttention when released. Therefore, dynamic memory allocation via PagedAttention is not feasible for FA3 at the time of writing this paper (integration of FA3 into vLLM is also a work in progress [16, 26]). vAttention not only enables dynamic memory allocation with FA3, it also requires no code changes to deploy FA3. Figure 11 shows the offline inference throughput of our models with 1–2 H100 GPUs (Yi-6B deployed on a single GPU and the other models on two GPUs each) on the same arXiv-Summarization-based workload as evaluated in §7.3. FA3 with vAttention provides an additional

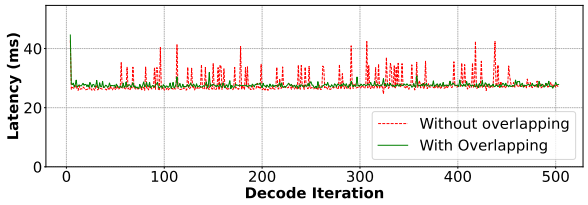


Figure 12. Latency of decode iterations with and without overlapping memory allocation with compute (batch size=32, context length=4K–8K, model: Llama-3-8B)

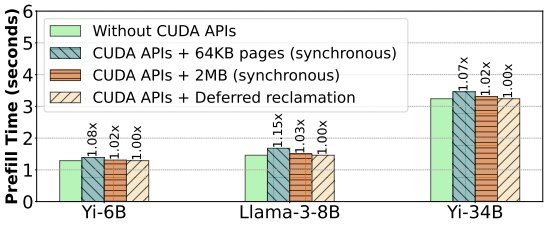


Figure 13. Prefill completion time of a single prompt of 16K tokens with different memory allocation strategies.

speedup of up to 1.35× (Yi-6B) over FA2_vAttention which is already up to 1.15× faster than FA2_Paged.

7.6 Ablation Studies

7.6.1 Hiding allocation latency. Figure 12 shows the latency of more than 500 decode iterations of Llama-3-8B. For this experiment, we used a batch of 32 requests with per request prefill context length varying between 2K to 3K tokens. It is evident that overlapping memory allocation with compute effectively hides the latency of CUDA VMM APIs. We used 2MB pages for this experiment to show that even the worst case memory allocation latency can be hidden by overlapping it with compute (Table 3 shows that 2MB pages incur highest latency). In contrast, allocating memory synchronously via CUDA APIs leads to frequent latency spikes of 5 to 15 milliseconds, depending on how many requests require physical memory allocation in a given iteration.

7.6.2 Deferred reclamation. Figure 13 shows that synchronous memory allocation incurs prefill overhead of up to 1.15× with 64KB pages and up to 1.03× with 2MB pages. In most cases, deferred reclamation eliminates the need to invoke CUDA VMM APIs for prefills because a newly arrived request can simply re-use physical memory allocated to a prior request. This way, deferred reclamation ensures that prefill latency is not affected by memory allocation.

7.6.3 Effect of page size. In many applications, use of smaller pages can potentially degrade performance due to TLB thrashing [52, 62, 64, 70]. We find that this is not the case for LLM inference. For example, Figure 14 shows that

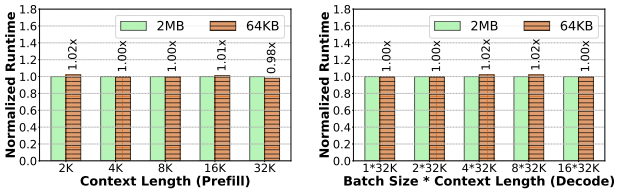


Figure 14. Effect of page size on the runtime of FlashAttention-2's prefill (left) and decode (right) attention kernels (model: Llama-3-8B).

Model	Block Size (# Tokens in a page-group)			
	64KB	128KB	256KB	2MB
Yi-6B (TP-1)	64	128	256	2048
Yi-6B (TP-2)	128	256	512	4096
Llama-3-8B (TP-1)	32	64	128	1024
Llama-3-8B (TP-2)	64	128	256	2048
Yi-34B (TP-1)	32	64	128	1024
Yi-34B (TP-2)	64	128	256	2048

Table 8. KV cache block size as a function of page-group size and degree of tensor parallelism.

the execution latency of an attention kernel remains largely unaffected when the KV cache is allocated using 64KB pages, compared to using 2MB pages. In a separate experiment, we find that these results are also consistent with very large models, e.g., Llama-3-70B and GPT-3-175B. We attribute this to the regular computation pattern of the attention operator as well as the hand-tuned implementations that explicitly try to avoid irregular memory accesses.

Table 8 shows the block size i.e., number of tokens per physical page-group. Smaller page-groups enable fine-grained allocation, approaching vLLM's recommended block size of 16–32 (the minimum block size in FlashAttention-2 is 256 – higher than vAttention). This shows that vAttention is as effective in reducing fragmentation as PagedAttention. In terms of end-to-end system performance, we find that 2MB pages are good enough for many online serving scenarios where latency constraints prevent use of very high batch sizes. In contrast, smaller page-groups are better for throughput-oriented scenarios where maximizing batch size is important to obtain peak performance. For example, using 2MB pages could serve batch sizes of up to 187, 203 and 56 for Yi-6B, Llama-3-8B and Yi-34B on a dynamic trace (dataset: OpenChat [74], load: 7 queries per seconds). In contrast, 64KB pages helped serve batch sizes of up to 240, 258 and 68 for our models as shown in Figure 15.

7.6.4 Memory allocation bandwidth. Table 9 shows that even with 64KB pages (our smallest), vAttention can allocate as much as 7.6GB physical memory per second per GPU. This is more than an order of magnitude higher than the maximum memory allocation rate of 750MB per second of

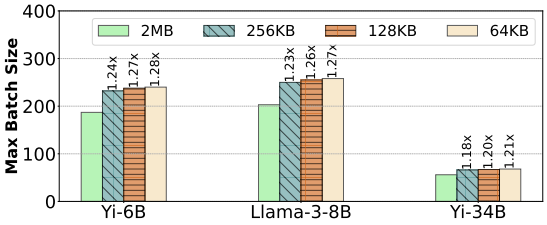


Figure 15. Maximum batch size obtained with different page-group sizes for a dynamic workload.

Config.	64KB	128KB	256KB	2MB
TP-1	7.59	14.56	27.04	35.17
TP-2	15.18	29.12	54.08	70.34

Table 9. Physical memory allocation bandwidth (GB per second) with varying allocation granularity.

decodes (Figure 4). Larger page-group sizes and higher TP dimensions increase the memory allocation rate proportionally. Therefore, memory allocation bandwidth of CUDA VMM APIs is more than sufficient for LLM inference.

8 Discussion

8.1 Managing KV cache via Unified Memory

To enable dynamic memory allocation, we also considered leveraging unified memory via `cudaMallocManaged` [58, 60]. However, we find that unified memory support is currently not suitable for serving LLMs. First, it does not support partial freeing, preventing reclamation of physical memory of individual requests. Second, it lacks support for memory aliasing which prevents de-duplication of KV cache content in physical memory (de-duplication is useful when requests share a common prefix [51]), consequently limiting batch size. `cudaMallocManaged` also allocates 2MB pages by default which can cause severe fragmentation. However, our changes in NVIDIA drivers are based on unified memory: we allocate virtual tensors using `cudaMallocManaged`, enabling support for partial freeing and page sharing with additional APIs. In addition, we introduce latency hiding optimizations and support for smaller pages. Therefore, one could consider our extensions to NVIDIA drivers as "unified memory optimized for LLM serving".

8.2 Reducing Fragmentation via Tensor Slicing

To reduce fragmentation caused by 2MB pages, we also provide an alternative method that does not require modifying NVIDIA drivers. In this method, we use a single 2MB page to store the KV cache tokens of all layers for a given request. This can be done by allocating one virtual tensor of shape $[B, L, N, H, D]$ for K cache (and one for V cache) and slicing it across all layers, instead of allocating $2 \times N$ virtual tensors of shape $[B, L, H, D]$. This reduces fragmentation to $1/N$ of the

Model	Block size (# Tokens in a 2MB page)	
	w/o Tensor Slicing	w/ Tensor Slicing
Yi-6B (TP-1)	2048	64
Yi-6B (TP-2)	4096	128
Llama-3-8B (TP-1)	1024	32
Llama-3-8B (TP-2)	2048	64
Yi-34B (TP-1)	1024	18
Yi-34B (TP-2)	2048	36

Table 10. KV cache block size with and without tensor slicing with 2MB pages.

```
-import flash_attn as fa
+import flashinfer as fi

-def flash_attn_prefill(q, k_cache, v_cache, kv_len):
-    k = k_cache[:, :kv_len, :, :]
-    v = v_cache[:, :kv_len, :, :]
-    return fa.flash_attn_func(q, k, v, causal=True)
+def flash_infer_prefill(q, k_cache, v_cache, kv_len):
+    q = q.squeeze(0)
+    k = k_cache.squeeze(0)[:kv_len, :, :]
+    v = v_cache.squeeze(0)[:kv_len, :, :]
+    return fi.single_prefill_with_kv_cache(q, k, v, causal=True)
```

Figure 16. Illustration of code changes needed to replace the prefill attention kernel of FlashAttention-2 by FlashInfer when using vAttention for memory allocation.

prior design as shown in Table 10. In this design, the K cache (or V cache) of a particular layer n is represented by slice $[B, L, n : n+1, H, D]$ of the original tensor, and can be passed to attention kernels for computation. However, note that with tensor slicing, the K cache (and V cache) of each layer is no longer contiguous. Computing attention over such a tensor slice is possible only if the the attention kernel supports memory addressing with strides. While many kernels (e.g., FlashAttention-2) support strides out-of-the-box, the earlier versions of FlashInfer lacked such support [14]. Therefore, relying on tensor slicing as the primary method of reducing fragmentation would have prevented vAttention from supporting FlashInfer kernels. Hence, to support unmodified FlashInfer kernels, we chose to add support for smaller pages in NVIDIA drivers. Importantly, our two solutions are compatible with each other; one could deploy smaller pages with tensor slicing to further reduce fragmentation, if required.

8.3 Programming Effort

PagedAttention requires significant programming effort. For example, integrating FlashInfer decode kernels in vLLM required more than 600 lines of code changes in over 15 files [24, 27, 30]. Implementing the initial paging support in FlashAttention-2 kernel also required ≈ 280 lines of code changes [20] and additional efforts to support small block sizes [8]. In contrast, vAttention makes it feasible to replace one attention kernel with another with only a few lines of code changes in the serving framework (Figure 16). Note that the example shown in Figure 16 has no interaction with memory management; this is precisely the goal of vAttention, i.e., when a slow kernel is replaced by a fast kernel, memory management should continue to work transparently.

9 Related Work

Optimizing LLM inference is an active area of research [44, 50, 53–57, 71, 72, 81, 83]. Various techniques have been proposed to improve different aspects of LLM serving like batching [35, 51, 78], disaggregation [48, 63, 82], scheduling [69, 75]. However, central to all these techniques is the need for efficient KV cache memory management. Since vLLM, PagedAttention has been adopted in various serving frameworks e.g., TensorRT-LLM [6], LightLLM [4], and libraries e.g., FlashAttention-2 [1] and FlashInfer [3]. Some other concurrent works have also proposed optimizations for attention kernels [29, 39, 47, 49, 66, 67, 76, 80]. Deploying new kernels for inference is hard with PagedAttention. vAttention offers an alternate – which we believe is also a more principled – approach to dynamic KV cache memory management that makes it easier to deploy new kernels.

In a recent work, GMLake [45] showed that using CUDA virtual memory support can mitigate fragmentation in DNN training jobs, increasing training batch size. In particular, GMLake uses CUDA support to coalesce multiple smaller physical memory pages into a single virtually contiguous object that can prevent out-of-memory errors. In contrast, vAttention targets optimizing inference workloads.

10 Conclusion

In this paper, we propose vAttention for dynamic KV cache memory management in LLM serving systems. The key highlight of vAttention is that it mitigates fragmentation in physical memory while retaining the contiguity of KV cache in virtual memory. We present various examples to highlight that vAttention reduces programming burden while improving portability and performance compared to the popular PagedAttention approach.

Acknowledgments

We thank our shepherd Sam Ainsworth, along with the anonymous ASPLOS reviewers, for their valuable feedback. We thank NVIDIA for useful discussions on CUDA VMM APIs and for sharing their experiences with vAttention. We also thank members of the open-source community (GitHub user names izhuhaoran and tongping) for trying to upstream the vAttention approach to vLLM [23, 31], and Zihao Ye for clarifying the role of register spilling in the runtime overhead of PagedAttention. Ajay Nayak is supported by the Prime Minister's Fellowship Scheme for Doctoral Research, co-sponsored by the Confederation of Indian Industry, the Government of India, and Microsoft Research India.

References

[1] 2022. FlashAttention. https://github.com/Dao-AILab/flash-attention.

[2] 2023. Flash-Decoding for long-context inference. https://crfm.stanford.edu/2023/10/12/flashdecoding.html.

[3] 2023. FlashInfer: Kernel Library for LLM Serving. https://github.com/flashinfer-ai/flashinfer.

[4] 2023. LightLLM: A Light and Fast Inference Service for LLM. https://github.com/ModelTC/lightllm.

[5] 2023. Performance decay when using paged attention. https://github.com/NVIDIA/TensorRT-LLM/issues/75.

[6] 2023. TensorRT-LLM: A TensorRT Toolbox for Optimized Large Language Model Inference. https://github.com/NVIDIA/TensorRT-LLM.

[7] 2023. Use optimized kernels for MQA/GQA. https://github.com/vllm-project/vllm/issues/1880.

[8] 2024. Add support for small page sizes. https://github.com/Dao-AILab/flash-attention/pull/824.

[9] 2024. Amazon CodeWhisperer. https://aws.amazon.com/codewhisperer/.

[10] 2024. Bing AI. https://www.bing.com/chat.

[11] 2024. ccdv/arxiv-summarization. https://huggingface.co/datasets/ccdv/arxiv-summarization.

[12] 2024. CUDA Toolkit Documentation: Virtual Memory Management. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__VA.html.

[13] 2024. Custom CUDA kernels for KV cache copy operations. https://github.com/vllm-project/vllm/blob/main/csrc/cache_kernels.cu.

[14] 2024. Custom strides to support non-contiguous KV cache. https://github.com/flashinfer-ai/flashinfer/commit/85b1878996a29814f674ee5000facb1e2e763d9a.

[15] 2024. Faster Transformer. https://github.com/NVIDIA/FasterTransformer.

[16] 2024. [Feature]: FlashAttention 3 support. https://github.com/vllm-project/vllm/issues/6348#issuecomment-2540969988.

[17] 2024. Fix eager mode performance. https://github.com/vllm-project/vllm/pull/2377.

[18] 2024. Github Copilot. https://github.com/features/copilot.

[19] 2024. Google Bard. https://bard.google.com.

[20] 2024. Implement Page KV Cache. https://github.com/Dao-AILab/flash-attention/commit/54e80a3829c6d2337570d01e78ebd9529c02d342.

[21] 2024. Meta-Llama-3-8B. https://huggingface.co/meta-llama/Meta-Llama-3-8B.

[22] 2024. Pascal MMU Format Changes. https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf.

[23] 2024. PoC of dAttention support (based on vAttention). https://github.com/vllm-project/vllm/pull/9078.

[24] 2024. Refactor Attention Take 2. https://github.com/vllm-project/vllm/pull/3462.

[25] 2024. Replit Ghostwriter. https://replit.com/site/ghostwriter.

[26] 2024. [Roadmap] vLLM Roadmap Q4 2024 #9006. https://github.com/vllm-project/vllm/issues/9006#issue-2559831134.

[27] 2024. Separate attention backends. https://github.com/vllm-project/vllm/pull/3005/.

[28] 2024. Text Generation Inference. https://huggingface.co/text-generation-inference.

[29] 2024. Tile primitives for speedy kernels. https://github.com/HazyResearch/ThunderKittens.

[30] 2024. Use FlashInfer for Decoding. https://github.com/vllm-project/vllm/pull/4353.

[31] 2024. VMM KV cache for NVIDIA GPUs. https://github.com/vllm-project/vllm/pull/6102.

[32] 2024. Yi-34B-200K. https://huggingface.co/01-ai/Yi-34B-200K.

[33] 2024. Yi-6B-200K. https://huggingface.co/01-ai/Yi-6B-200K.

[34] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. Vidur: A Large-Scale Simulation Framework For LLM Inference. *Proceedings of The Seventh Annual Conference on Machine Learning and Systems, 2024, Santa Clara* (2024).

[35] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[36] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. 2023. SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills. arXiv:2308.16369 [cs.LG]

[37] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. arXiv:2305.13245 [cs.CL]

[38] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. arXiv:2004.05150 [cs.CL]

[39] Ganesh Bikshandi and Jay Shah. 2023. A Case Study in CUDA Kernel Fusion: Implementing FlashAttention-2 on NVIDIA Hopper Architecture using the CUTLASS Library. arXiv:2312.11918 [cs.LG]

[40] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating Long Sequences with Sparse Transformers. arXiv:1904.10509 [cs.LG]

[41] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways. *CoRR* abs/2204.02311 (2022). https://doi.org/10.48550/arXiv.2204.02311 arXiv:2204.02311

[42] Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. arXiv:2307.08691 [cs.LG]

[43] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2024. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1189, 16 pages.

[44] Tyler Griggs, Xiaoxuan Liu, Jiaxiang Yu, Doyoung Kim, Wei-Lin Chiang, Alvin Cheung, and Ion Stoica. 2024. Mélange: Cost Efficient Large Language Model Serving by Exploiting GPU Heterogeneity. arXiv:2404.14527 [cs.DC] https://arxiv.org/abs/2404.14527

[45] Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, Minyi Guo, Hao Wu, Shouren Zhao, Junping Zhao, and Ke Zhang. 2024. GMLake: Efficient and Transparent GPU Memory Defragmentation for Large-scale DNN Training with Virtual Memory Stitching. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 450–466. https://doi.org/10.1145/3620665.3640423

[46] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. 2024. DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference. arXiv:2401.08671 [cs.PF]

[47] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, kangdi chen, Yuhan Dong, and Yu Wang. 2024. FlashDecoding++: Faster Large Language Model Inference with Asynchronization, Flat GEMM Optimization, and Heuristics. In *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 148–161. https://proceedings.mlsys.org/paper_files/paper/2024/file/5321b1dabcd2be188d796c21b733e8c7-Paper-Conference.pdf

[48] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. 2024. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. *arXiv preprint arXiv:2401.11181* (2024).

[49] Aditya K Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. 2024. POD-Attention: Unlocking Full Prefill-Decode Overlap for Faster LLM Inference. arXiv:2410.18038 [cs.LG] https://arxiv.org/abs/2410.18038

[50] Ferdi Kossmann, Bruce Fontaine, Daya Khudia, Michael Cafarella, and Samuel Madden. 2024. Is the GPU Half-Empty or Half-Full? Practical Scheduling Techniques for LLMs. arXiv:2410.17840 [cs.LG] https://arxiv.org/abs/2410.17840

[51] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[52] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 705–721. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon

[53] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. arXiv:2406.19707 [cs.LG] https://arxiv.org/abs/2406.19707

[54] Da Ma, Lu Chen, Situo Zhang, Yuxun Miao, Su Zhu, Zhi Chen, Hongshen Xu, Hanqi Li, Shuai Fan, Lei Pan, and Kai Yu. 2024. Compressing KV Cache for Long-Context LLM Inference with Inter-Layer Attention Similarity. arXiv:2412.02252 [cs.CL] https://arxiv.org/abs/2412.02252

[55] Saaduddin Mahmud, Mason Nakamura, and Shlomo Zilberstein. 2024. MAPLE: A Framework for Active Preference Learning Guided by Large Language Models. arXiv:2412.07207 [cs.LG] https://arxiv.org/abs/2412.07207

[56] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. 2024. Helix: Distributed Serving of Large Language Models via Max-Flow on Heterogeneous GPUs. arXiv:2406.01566 [cs.LG] https://arxiv.org/abs/2406.01566

[57] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2023. SpotServe: Serving Generative Large Language Models on Preemptible Instances. arXiv:2311.15566 [cs.DC] https://arxiv.org/abs/2311.15566

[58] Ajay Nayak, Pratheek B., Vinod Ganapathy, and Arkaprava Basu. 2021. (Mis)managed: A Novel TLB-based Covert Channel on GPUs. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (Virtual Event, Hong Kong) *(ASIA CCS '21)*. Association for Computing Machinery, New York, NY, USA, 872–885. https://doi.org/10.1145/3433210.3453077

[59] Matthew Nicely and NVIDIA. 2024. Accelerating Transformers with NVIDIA cuDNN 9. https://developer.nvidia.com/blog/accelerating-transformers-with-nvidia-cudnn-9/

[60] NVIDIA. 2024. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[61] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/arXiv.2303.08774 arXiv:2303.08774

[62] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-Grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 347–360. https://doi.org/10.1145/3297858.3304064

[63] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 118–132. https://doi.org/10.1109/ISCA59077.2024.00019

[64] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural support for address translation on GPUs: designing memory management units for CPU/GPUs with unified address spaces. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 743–758. https://doi.org/10.1145/2541940.2541942

[65] B Pratheek, Neha Jawalkar, and Arkaprava Basu. 2022. Designing Virtual Memory System of MCM GPUs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 404–422. https://doi.org/10.1109/MICRO56248.2022.00036

[66] Rya Sanovar, Srikant Bharadwaj, Renee St. Amant, Victor Rühle, and Saravan Rajmohan. 2024. Lean Attention: Hardware-Aware Scalable Attention Mechanism for the Decode-Phase of Transformers. arXiv:2405.10480 [cs.AR] https://arxiv.org/abs/2405.10480

[67] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. (2024).

[68] Noam Shazeer. 2019. Fast Transformer Decoding: One Write-Head is All You Need. arXiv:1911.02150 [cs.NE]

[69] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: high-throughput generative inference of large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 1288, 23 pages.

[70] Seunghee Shin, Michael LeBeane, Yan Solihin, and Arkaprava Basu. 2018. Neighborhood-Aware Address Translation for Irregular GPU Applications. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 352–363. https://doi.org/10.1109/MICRO.2018.00036

[71] Prajwal Singhania, Siddharth Singh, Shwai He, Soheil Feizi, and Abhinav Bhatele. 2024. Loki: Low-rank Keys for Efficient Sparse Attention. arXiv:2406.02542 [cs.LG] https://arxiv.org/abs/2406.02542

[72] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. 2024. DéjàVu: KV-cache Streaming for Fast, Fault-tolerant Generative LLM Serving. arXiv:2403.01876 [cs.DC] https://arxiv.org/abs/2403.01876

[73] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[74] Guan Wang, Sijie Cheng, Xianyuan Zhan, Xiangang Li, Sen Song, and Yang Liu. 2023. OpenChat: Advancing Open-source Language Models

with Mixed-Quality Data. arXiv:2309.11235 [cs.CL]

[75] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast Distributed Inference Serving for Large Language Models. arXiv:2305.05920 [cs.LG]

[76] Mengdi Wu, Xinhao Cheng, Oded Padon, and Zhihao Jia. 2024. A Multi-Level Superoptimizer for Tensor Programs. arXiv:2405.05751

[77] Zihao Ye, Lequn Chen, Ruihang Lai, Yilong Zhao, Size Zheng, Junru Shao, Bohan Hou, Hongyi Jin, Yifei Zuo, Liangsheng Yin, Tianqi Chen, and Luis Ceze. 2024. Accelerating Self-Attentions for LLM Serving with FlashInfer. https://flashinfer.ai/2024/02/02/introduce-flashinfer.html

[78] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[79] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. 2023. TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 960–974. https://doi.org/10.1145/3576915.3616672

[80] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang "Atlas" Wang, and Beidi Chen. 2023. H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 34661–34710. https://proceedings.neurips.cc/paper_files/paper/2023/file/6ceefa7b15572587b78ecfcebb2827f8-Paper-Conference.pdf

[81] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. arXiv:2312.07104 [cs.AI] https://arxiv.org/abs/2312.07104

[82] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

[83] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. 2024. NanoFlow: Towards Optimal Large Language Model Serving Throughput. arXiv:2408.12757 [cs.DC] https://arxiv.org/abs/2408.12757

A Artifact Appendix

A.1 Abstract

vAttention is a simpler, portable and performant alternative to PagedAttention for dynamic memory management in LLM serving systems. This artifact contains instructions to install vAttention and scripts to reproduce key results of our paper (Figures 2, 3, 4, 6, 7, 8, 9, 11). The scripts are configured to run three large language models: Yi-6B, Llama-3-8B and Yi-34B.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Dynamic memory management for serving LLMs.
- **Run-time environment:** CUDA 12.1, Python 3.10, PyTorch 2.3.0.
- **Hardware:** Two NVLink connected NVIDIA A100 GPUs with 80GB memory each.
- **Disk space required:** 200GB.
- **Time needed to prepare workflow:** 1 hour.
- **Time needed to complete experiments:** 1-2 days.
- **Publicly available?:** Yes
- **Archived DOI?:** https://doi.org/10.5281/zenodo.14048692

A.3 Description

A.3.1 How to access. Clone the artifact as follows:

```
$ git clone https://github.com/microsoft/vattention
```

A.3.2 Hardware dependencies. Running Yi-6B requires one NVIDIA A100 GPU with 80GB memory while the other two models require two NVLink-connected A100 GPUs with 80GB memory each.

A.3.3 Software dependencies. Requires PyTorch 2.3.0 and CUDA 12.1 (later CUDA versions may or may not work).

A.3.4 Expected runtime. Figures 2, 3, and 11 should take a few minutes each. Figures 4, 7 should take about 1 hour each. Figure 6 requires approximately 3 hours. Figures 8 and 9 may consume more than 12 hours each.

A.4 Installation

Move to the directory containing artifact scripts, create and activate a conda environment, then install the artifact as follows:

```
$ cd vattention/scripts/artifact_asplos25/
$ conda create -n vattn python=3.10
$ conda activate vattn
$ (vattn) ./install.sh
```

`vattention/scripts/artifact_asplos25/README.md` file provides detailed installation instructions.

A.5 Alternate Setup: Docker Image

We also provide a docker image for vAttention with all its dependencies pre-installed. To access the docker image, you need to have Docker and NVIDIA Docker installed on your system. You can then launch the docker container and navigate to the folder containing vAttention artifact, as follows:

```
$ docker run --gpus all -it \
  -p 8181:8181 --rm --ipc=host --cap-add=SYS_ADMIN \
  rnp1910/vattention:asplos_25_pytorch_run
$ cd /workspace/vattention/scripts/artifact_asplos25
```

Then follow the experiment workflow detailed in A.7

A.6 Accessing Llama-3-8B

Accessing Llama-3-8B requires logging into huggingface with the user's private token (`HF_TOKEN` below). Login as follows before any experiment:

```
$ (vattn) huggingface-cli login --token HF_TOKEN
```

A.7 Experiment workflow

You can launch all experiments at once or individually as:

```
$ (vattn) ./run_all.sh
          OR
$ (vattn) ./run_figure_2.sh
$ (vattn) ./run_figure_3.sh
```

A.8 Evaluation and expected results

The raw output logs and the final plots will be redirected to `./logs`. and `./plots/` subdirectories within the main artifact directory `vattention/scripts/artifact_asplos25/`. The plots are in the same format as the paper and can be compared directly. The expected result is that non-paged or vAttention based configurations would perform better than the PagedAttention counterparts, especially at longer context lengths. However, the exact numbers may differ from the paper depending on GPUs and software libraries installed on the experiment system.

A.9 Experiment customization

Reproducing Figures 8 and 9 of the paper requires running a large number of requests and hence these experiments can take more than 24 hours to complete. To reduce the experimental time, we have configured their scripts to run with a smaller number of requests (100 each) by default. If you want to run the full trace, execute these scripts with the `--full` argument, i.e., `./run_figure_8.sh --full` and `./run_figure_9.sh --full`. If two NVLink connected GPUs aren't available, update run scripts to avoid running Llama-3-8B and Yi-34B. You can also configure Llama-3-8B to run on a single GPU by setting `--model_tensor_parallel_degree` to 1 in the `./helpers/common.sh` file. Note that such changes would likely impact the absolute performance numbers considerably but vAttention is still expected to perform better than PagedAttention.