

vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention

Ramya Prabhu
Microsoft Research India

Ajay Nayak*
Indian Institute of Science

Jayashree Mohan
Microsoft Research India

Ramachandran Ramjee
Microsoft Research India

Ashish Panwar
Microsoft Research India

Abstract

Efficient management of GPU memory is essential for high throughput LLM inference. Prior systems used to reserve KV-cache memory ahead-of-time that resulted in wasted capacity due to internal fragmentation. Inspired by demand paging, vLLM proposed PagedAttention to enable dynamic memory allocation for KV-cache. This approach eliminates fragmentation and improves serving throughout. However, to be able to allocate physical memory dynamically, PagedAttention *changes the layout of KV-cache from contiguous virtual memory to non-contiguous virtual memory*. As a consequence, one needs to rewrite the attention kernels to support paging, and implement a memory manager in the serving framework. This results in both performance and programming overheads, as well as portability challenges in adopting state-of-the-art attention kernels.

In this paper, we propose vAttention, a new approach for dynamic KV-cache memory management. In contrast to PagedAttention, vAttention stores KV-cache in *contiguous virtual memory* and leverages OS support for on-demand allocation of physical memory. vAttention thus enables one to use state-of-the-art attention kernels out-of-the-box by adding support for dynamic allocation of physical memory without having to re-write their code. We implement vAttention in the vLLM serving stack to show that it also helps improve decode throughput by up to 1.99 \times over vLLM, and the end-to-end serving throughput by up to 1.22 \times and 1.29 \times , compared to using the state-of-the-art PagedAttention based kernels of FlashAttention and FlashInfer.

1 Introduction

Large Language Models (LLMs) are deployed in a wide range of applications e.g., chat bots, search engines and coding assistants [1–5, 37, 47]. Given the size and scale of modern LLM deployments, optimizing inference has become extremely important [31, 41, 43, 44, 49, 61, 65]. LLM inference happens in two phases: a compute-bound *prefill* phase, followed by several iterations of memory-bound *decode* phase. Batching is a powerful technique to boost LLM serving throughput [30, 44, 49, 61], as it amortizes the cost of fetching model weights from HBM to the on-device SRAM during decodes.

System/library and issues related to PagedAttention

vLLM: Pioneered PagedAttention. Despite being in an actively maintained code repository, vLLM’s PagedAttention kernel is up to 2.8 \times slower than FlashAttention (Table 6). Further, changing block size changes the execute latency of the kernel by as much as 3 \times (Figure 3).

FlashAttention: PagedAttention based prefill kernel is up to 28% 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 [13].

TensorRT-LLM: Serving throughput dropped by more than 10% in a Python front-end [11]. Recommends using the C++ front-end. Even with C++, we observe up to 5% higher latency in some cases with PagedAttention.

FlashInfer: PagedAttention based prefill kernel is up to 24% 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.

Careful allocation of GPU memory is key to enabling larger batch sizes. For each request, the serving framework stores the activations of all the tokens processed so far in GPU memory and reuses them for the subsequent token generation. This is called the KV-cache [31, 49, 61] which accounts for a majority of GPU memory usage during inference. Efficiently allocating GPU memory for the KV-cache of a batch of requests 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 [61] and FasterTransformer [18] allocate GPU 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 [27]). However, in practice, the number of decode tokens generated are far less; the average decode length for the chat-based sharegpt dataset is 415 tokens [30]. Therefore, static memory allocation results in severe internal fragmentation, limiting batch size and serving throughput.

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

Inspired by demand paging in OS-based virtual memory systems, vLLM introduced PagedAttention [44] to mitigate KV-cache related memory fragmentation. Instead of reserving the maximum sequence length of KV-cache memory ahead-of-time, vLLM 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., in TensorRT-LLM [12], HuggingFace TGI [6], LightLLM [10] etc.

However, the PagedAttention approach faces a fundamental consequence of dynamic memory allocation: *dynamically allocated memory blocks 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 layout of KV-cache from contiguous to non-contiguous **virtual** memory. We argue that this approach has several pitfalls (see Table 1 for empirical evidence and real-world experiences):

1. Requires re-writing the attention kernel (GPU code). The elements of a virtually contiguous object can be accessed using index-based lookup which is both simple and efficient. Many implementations of deep learning operators also expect virtual contiguity of input tensors. However, by storing KV-cache in non-contiguous virtual memory, PagedAttention mandates re-writing GPU code so that the attention kernel can de-reference all the elements of KV-cache. The need to re-write code is a major barrier to using new attention optimizations in production settings.

2. Adds software complexity and redundancy (CPU code). To stitch together dynamically allocated virtual memory blocks, PagedAttention forces developers to implement a memory manager in the serving framework. Typically, this requires a Block-Table manager in user space which is essentially a re-implementation of demand paging (an OS functionality) in user code.

3. Introduces performance overhead. PagedAttention adds runtime overhead in the critical path of execution in two ways. First, GPU kernels that compute attention over a non-contiguous KV-cache need to execute additional instructions. In many cases, this slows down an attention kernel by more than 20%, compared to when the KV-cache is virtually contiguous. Second, the user space memory manager can add CPU overhead contributing up to another 10% cost (§3.3).

In this paper, we instead advocate for leveraging the OS support of virtual memory and demand paging for dynamic KV-cache memory management. This approach alleviates the aforementioned issues that appear with PagedAttention.

To support our claims, we present the design and implementation of vAttention. Leveraging OS support, vAttention *stores KV-cache in contiguous virtual memory without committing physical memory ahead-of-time*, thereby enabling the use

of state-of-the-art high-performant attention kernels out-of-the-box. We achieve this by using CUDA support of low-level virtual memory APIs which expose distinct interfaces for allocating virtual and physical memory (§5).

Leveraging OS support for memory allocation in an LLM serving system poses two key efficiency challenges (§5.4). First, the minimum physical memory allocation granularity supported by CUDA is 2MB. This can result in significant wasted capacity and fragmentation. We address this challenge by modifying the open-source CUDA unified virtual memory driver, adding support for finer-grained physical memory allocations in multiples of 64KB. Our evaluation shows that using 64KB pages as the granularity of virtual-to-physical address translation does not affect the execution latency of attention kernels i.e., we do not find any evidence of TLB thrashing. Second, memory allocation using CUDA APIs incurs high latency because each allocation involves a round-trip to the OS kernel. We introduce several LLM-specific optimizations such as overlapping memory allocation with compute, opportunistically allocating pages ahead of time, and deferring memory reclamation. These optimizations hide the latency cost of memory allocation from end-users, making vAttention an efficient KV-cache memory manager.

Overall, we make the following contributions in this paper:

- We present vAttention – a memory management approach that retains the virtual contiguity of KV-cache while enabling dynamic physical memory allocation.
- We implement vAttention in vLLM serving stack to show that it seamlessly adds dynamic memory management support to unmodified attention kernels of FlashAttention [7] and FlashInfer [9].
- We compare vAttention against PagedAttention based alternatives of vLLM, FlashAttention and FlashInfer by evaluating Yi-6B, Llama-3-8B and Yi-34B on 1-2 A100 GPUs. Using FlashAttention’s non-paged kernel, vAttention outperforms vLLM by up to 1.99× in decode throughput, while improving the end-to-end LLM serving throughput by up to 1.22× and 1.29×, compared to using the state-of-the-art PagedAttention based kernels of FlashAttention and FlashInfer.

2 Background

2.1 Large Language Models

Given an input sequence, an LLM predicts the probability of an output sequence wherein a sequence is a set of tokens [44]. Each inference request begins with a prefill phase that processes all its prompt tokens in parallel. The prefill phase produces the first output token of a request. Thereafter, the decode phase iteratively processes the output token generated in the previous step and produces the next output token in every iteration [31].

LLMs are built atop one of variants of the transformer architecture [56]. A transformer block contains two types of operators: position-wise and sequence-wise. 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. In this paper, we primarily focus on attention since it is the primary consumer of GPU memory in LLM inference.

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

$$q_i = W_q x_i, \quad k_i = W_k x_i, \quad v_i = W_v x_i \quad (1)$$

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 and D is the dimension of each KV head. Then, attention is computed as follows:

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

The attention score is computed separately for each request in the batch. 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 as KV-cache. A request is executed until the model generates a special end-of-sequence token or reaches the maximum context length for the request.

Structure of the KV-cache and terminology: An LLM consists of multiple layers of the transformer block and each layer maintains its own cache of keys and values. In this paper, we refer to the cache of all transformer blocks collectively as KV-cache while using the term K-cache or V-cache for keys and values, respectively. In deep learning frameworks, the K-cache (or V-cache) at each layer is typically represented 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. We refer to a kernel implementation that computes attention scores over contiguously stored K and V as a non-paged kernel.

2.2 Fragmentation and PagedAttention

To improve serving throughput, production systems rely on batching which 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

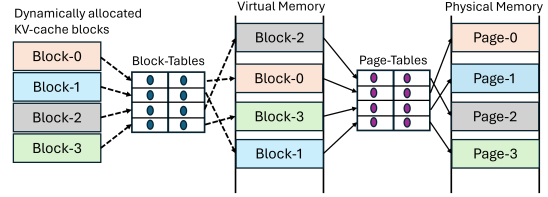


Figure 1. PagedAttention involves two layers of memory management: one in user space and one in OS kernel space.

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.

3 Issues with the PagedAttention Model

Despite being inspired by demand paging, the PagedAttention approach is different from it: *PagedAttention mandates an implementation of 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

PagedAttention necessitates re-writing the attention kernel. This is because 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 [13].

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 [8, 32, 34–36, 38, 39, 42, 53, 60, 64], 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 and competitive advantage. To provide an example, Table 6 shows that the paged kernel of vLLM is already up to 2.8× slower than the FlashAttention kernel [32].

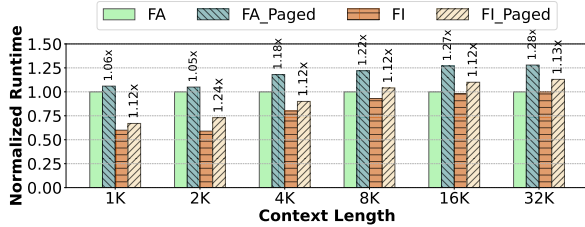


Figure 2. Overhead of PagedAttention in prefill kernels (model: Llama-3-8B, one A100 GPU). Numbers on top show overhead over the corresponding non-paged implementation of FlashAttention (FA) and FlashInfer (FI).

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. For example, vLLM acknowledges that their PagedAttention-based implementation was 20 – 26% slower than the original FasterTransformer kernel, primarily due to the overhead of looking up Block-Tables and executing extra branches (see Figure 18a in [44]).

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 and FlashInfer are up to 28% and 24% 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.

To highlight another example of difficulty involved in writing an efficient attention kernel with paging support, Figure 3 shows that the performance of vLLM’s paged decode kernel is significantly worse with larger block sizes of 64 and 128. Our analysis indicates that this is likely due to L1 cache efficiency: smaller blocks have a higher memory bandwidth utilization due to higher hit rates in L1 cache.

3.3.2 Runtime overhead on the CPU. Implementing an additional memory manager can add performance issues in

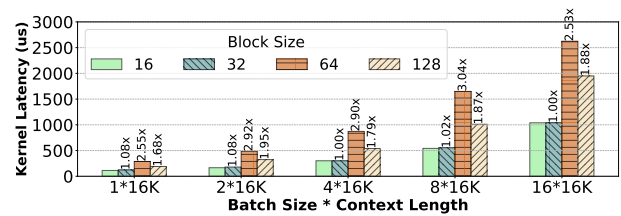


Figure 3. Latency of vLLM’s paged decode kernel is sensitive to block size (model: Llama-3-8B, one A100 GPU).

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 $\text{max_num_blocks} \times \text{batch_size}$ 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 [19] 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%, from 412 to 365 tokens/sec to 365 tokens/sec [11]. 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 would be an unwelcome change for most programmers.

Overall, this section shows that the PagedAttention model adds a significant programming burden while also being inefficient. vAttention introduces a more systematic approach to dynamic KV-cache memory management by leveraging the existing system support for demand paging. 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 highlight 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. 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 §5.4 for our discussion and optimizations for the prefill phase).

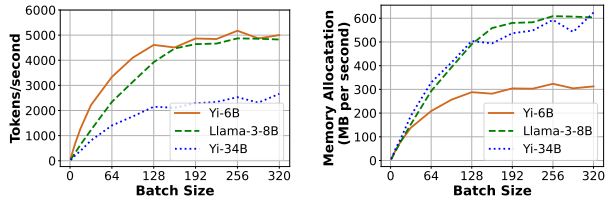


Figure 4. Decode throughput (left) and the rate of physical memory allocation (right) saturate at large batch sizes.

Observation-1: *On a per-iteration basis, KV-cache memory requirement is known in advance.* This is a dream property for a memory allocator, thanks to the auto-regressive decoding wherein every forward pass of the model generates one token per request. Therefore, with every iteration, the KV-cache memory footprint of a request grows uniformly by one token as long as the request is active.

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 [30, 31, 49, 65], the number of tokens generated per second plateaus beyond a certain batch size (Figure 4, left). This implies that the memory allocation bandwidth requirement also saturates at large batch sizes (e.g., at 128 for Yi-34B). For all the models we studied, we observe that the highest memory allocation rate is at most 600MB per second (Figure 4, right).

In vAttention, we leverage these observations to implement an efficient dynamic memory management system for KV-cache. In the next section, we begin with a high-level overview of vAttention (§5.1), then discuss how vAttention is used for serving LLMs (§5.3) and finally describe our optimizations (§5.4).

5 vAttention: Design and Implementation

To facilitate dynamic allocation of physical memory to unmodified attention kernels, vAttention leverages system support for demand paging instead of implementing it in user space.

5.1 Design Overview

vAttention builds on the ability to allocate virtual memory and physical memory separately. Specifically, we allocate a large contiguous buffer for the KV-cache in virtual memory ahead-of-time (similar to reservation-based allocators) while deferring the allocation of physical memory to runtime (similar to PagedAttention). This way, vAttention preserves

virtual contiguity of KV-cache without wasting physical memory. This approach is feasible because memory capacity and fragmentation are limiting factors only for physical memory whereas virtual memory is abundant e.g., modern 64-bits systems provide a 128TB user-managed virtual memory per process¹.

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 would generate as many tokens as the maximum sequence length supported by the model.

Number of virtual memory buffers. A serving framework typically maintains separate K and V tensors for each layer of the model: we refer to them individually as K-cache and V-cache. We allocate separate virtual memory buffers for K-cache and V-cache. For a single GPU job, this requires pre-reserving $2 \times N$ buffers where N is the number of layers in the model. In a multi-GPU job, each worker reserves $2 \times N'$ buffers where N' is the number of layers managed by that worker ($N' = N$ with tensor-parallelism whereas $N' < N$ with pipeline-parallelism).

Size of a virtual memory buffer. The maximum size of a buffer is $BS = B \times L \times S$ where B is the maximum batch size, L is the maximum context length supported by the model, and S is the size of a single token’s per-layer K-cache (or V-cache) on a worker. Further, $S = H \times D \times P$, where H is the number of KV heads on a worker, D is the dimension of each KV head and P is the number of bytes based on model precision (e.g., $P=2$ for FP16/BF16). Note that S is constant for a given model configuration.

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 model, the maximum size of K-cache (or V-cache) per-worker per-layer is $S = 200MB$ ($200K * 4 * 128 * 2$). 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 amount of virtual address space available grows with the number of GPUs e.g., with two TP workers, the amount of virtual address space available is 256TB. Therefore, virtual memory allocations can be satisfied easily. Figure 5 shows how vAttention allocates physical memory pages dynamically.

¹64-bits 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.

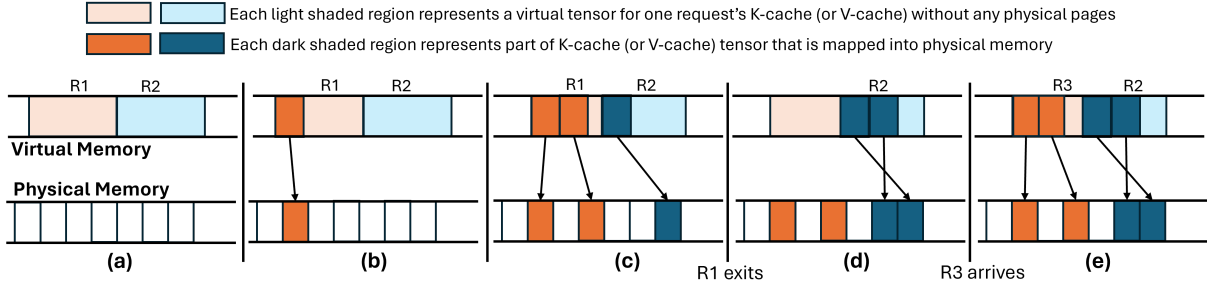


Figure 5. Dynamic memory management in vAttention for a single K-cache (or V-cache) tensor. (a) shows a virtual tensor for a batch of two requests with no physical memory allocation yet. (b) R1 is allocated one physical page. (c) R1 is allocated two pages and R2 is allocated one page. (d) R1 has completed 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 2. CUDA VM (virtual memory) 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 allocation sizes, whereas our CUDA extension APIs (prefixed with v) support fine-grained allocations.

5.2 Leveraging Low-level CUDA 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 virtual and physical memory [16, 40]. We leverage these low-level APIs in vAttention.

5.2.1 CUDA virtual memory APIs. Table 2 provides an overview of CUDA APIs that allow separating 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. It is also important to highlight that physical memory pages can be allocated (or de-allocated) to sub-regions in a virtual memory buffer independently of other sub-regions. For simplicity, we refer to the granularity of physical memory allocation as *page size*.

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 uses the `cudaMalloc` interface to allocate GPU memory (both virtual

and physical). 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. Note that each 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 $\text{reqId} \times S$ where S is the maximum K-cache (or V-cache) size 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 (see Table 3 and Algorithm 1).

APIs	Description
init	Initializes vAttention with model parameters. arguments: $N', B, L, H', P, \text{page_size}$. 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 physical memory pages are mapped arguments: an array of size B containing sequence lengths of each reqId return value: 0 (success), -1 (failure).

Table 3. Key APIs that vAttention exposes to a serving framework for dynamic KV-cache memory management.

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 size* via the `init` API. Internally, vAttention reserves $2 \times N'$ virtual tensors (using our modified PyTorch caching allocator) for the KV-cache at each worker. These virtual tensors are reserved for the lifetime of the serving application. In addition, vAttention also pre-allocates physical memory pages during initialization. However, these pages are not mapped into the KV-cache yet.

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`. 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. For this purpose, before dispatching the first kernel of an iteration to the GPU, the framework invokes the `step` API, 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 number of physical memory pages may need to be mapped for a given iteration. The prefill phase processes the input tokens of given prompt in parallel and populates one slot in the K-cache (and V-cache) of the request at each layer of the model. Therefore, the number

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_cache(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

```

of pages 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 size is t , then each worker needs to ensure that at least $(s + t - 1)/t$ physical memory pages 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 pages required is at most one per request. This is because each iteration produces only one output token for a request. vAttention internally tracks the number of pages mapped for each request and maps a new page only when the last page allocated to that request is fully utilized.

5.3.4 Request completion. A request terminates when a user specified context length or the maximum context length supported by the model is reached, or when the model produces a special end-of-sequence token. The framework notifies vAttention of a request’s completion with `free_reqid`. Internally, vAttention may unmap the pages of a completed request or defer them to be freed later.

5.4 Optimizations

There are two primary challenges in using CUDA virtual memory support for serving LLMs. First, `cuMemCreate` currently allocates a minimum of 2MB physical memory page. Large pages can waste physical memory due to internal fragmentation. Second, invoking CUDA APIs incurs high latency.

This section details a set of simple-yet-effective optimizations that we introduce to overcome these limitations.

5.4.1 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, 33, 46, 62]. 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 virtual memory APIs (listed in Table 2) to also support allocating smaller pages (similar to how `mmap` in Linux supports multiple page sizes). Unfortunately, the CUDA 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 2 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 APIs, our APIs can allocate memory in 64KB, 128KB and 256KB page sizes. A serving framework can configure a desired page size in `vAttention` while initializing it. The last set of columns in Table 2 shows the latency of each API with different page sizes.

5.4.2 Hiding memory allocation latency. The serving framework invokes the `step` API in every iteration. The latency of `step` depends on how many new pages need to be mapped in the virtual tensors of 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 pages (at all layers) adds about 5 millisecond latency to the corresponding iteration. Further, the latency overhead grows proportional to the number of requests that need new pages in a given iteration. We propose the following optimizations to hide the latency of allocation:

Overlapping memory allocation with compute. 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. `vAttention` keeps track of the current context length and how many physical memory pages are already mapped for each request. Using this information, it determines when a request would need a new page

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 4. Models and hardware used for evaluation.

and uses a background thread to allocate a new page when the preceding iteration is executing. For example, consider that a request R1 would require a new page in iteration i . When the serving framework invokes `step` API in iteration $i-1$, `vAttention` launches a background thread that maps physical memory pages for iteration i . Since 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 mapping physical pages in the KV-cache tensors out of the critical path. Note that in every iteration, `step` API still needs to ensure that physical pages required for the current iteration are actually mapped. If not, required pages are mapped synchronously.

Deferred reclamation + eager allocation. We observe that allocating physical memory for a new request 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 new pages to R2 from scratch, `vAttention` simply defers the reclamation of R1’s pages 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 (for security reasons, we still zero-fill such pages before allowing a new request to use them). Therefore, new pages are required only if the context length of R2 is bigger than that of R1.

We further optimize memory allocation by proactively allocating a small number of physical pages ahead of time. For this purpose, we try to keep a certain number of pages (configurable based on expected workload) mapped into the virtual tensors of one of the inactive `reqId`. When a new request arrives, we allocate this `reqId` and identify a new `reqId` to be allocated next and map physical pages for it. In most cases, these eager allocations obviate the need to allocate physical pages in the critical path of prefill execution. Finally, we trigger memory reclamation only when the number of physical pages 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 Evaluation

Our evaluation seeks to answer the following questions:

- How does `vAttention` perform for prefill and decode phases (§6.1, §6.2)?

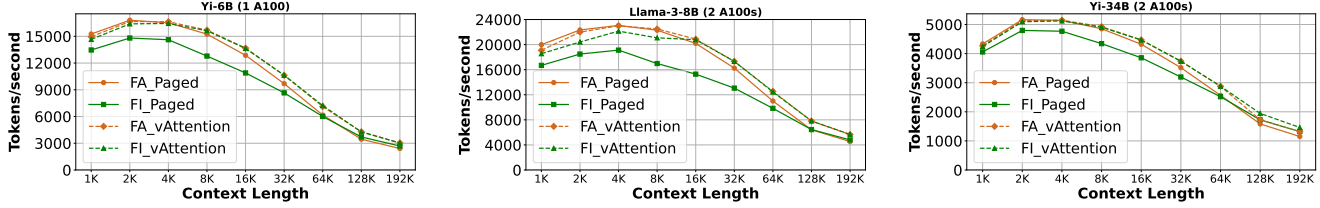


Figure 6. Prefill throughput. vAttention backed systems outperform the paged counterparts of both FlashAttention and FlashInfer. Throughput for longer contexts is lower due to the quadratic complexity of prefill attention.

- How vAttention impacts LLM serving throughput (§6.3)?
- What is the effect of each of our optimizations (§6.4)?

Models and hardware: We evaluate three models Yi-6B [28], Llama-3-8B [21] and Yi-34B [27], using a single NVIDIA A100 GPU for Yi-6B, and two NVLink-connected A100 GPUs for Llama-3-8B and Yi-34B (see Table 4). Each GPU has 80GB physical memory. We use tensor-parallelism degree of two (TP-2) for both Llama-3-8B and Yi-34B.

Evaluation methodology: The computation and memory allocation pattern of the prefill and decode phases is substantially different [30, 41, 65]. Attention kernels used for these two phases are also different and hence we also 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 [44]. We define throughput as the number of tokens processed (or 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 v2.5.9 [7, 38, 39] and FlashInfer v0.4.0 [9] as attention back-ends into vLLM, and further added support for dynamic memory allocation via vAttention to their non-paged kernels. While FlashAttention and FlashInfer are both based on the same underlying techniques (e.g., FlashDecoding [8]), they use different Block-Table formats; the former use a simple lookup table whereas the latter uses a compressed Block-Table to aid fetching KV-cache block locations.

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). For a fair comparison, 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. Using a higher block size for vLLM increases its kernel latency by up to 3× as shown in Figure 3, and using a smaller block size for FlashAttention paged kernel increases its latency by up to 9% (Table 9). We find that the

Model	Context Length	FA_Paged	FA_vAttention	FI_Paged	FI_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 5. Prefill completion and attention (in parenthesis) time with different attention back-ends (unit: seconds).

choice of page size does not affect vAttention as long as fragmentation is not a concern (Figure 11).

6.1 Prefill Evaluation

We evaluate four configs for prefill: FA_Paged, FI_Paged, FA_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 6 shows the prefill throughput for Yi-6B, Llama-3-8B and Yi-34B. We summarize our key findings below.

Small contexts: For small contexts, prefill cost is dominated by the linear operators i.e., attention’s contribution is relatively low [31]. Hence, even though vAttention helps speed up attention computation, the throughput of both paged and vAttention back-ends is nearly identical in FlashAttention. 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 by vLLM [17]). 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

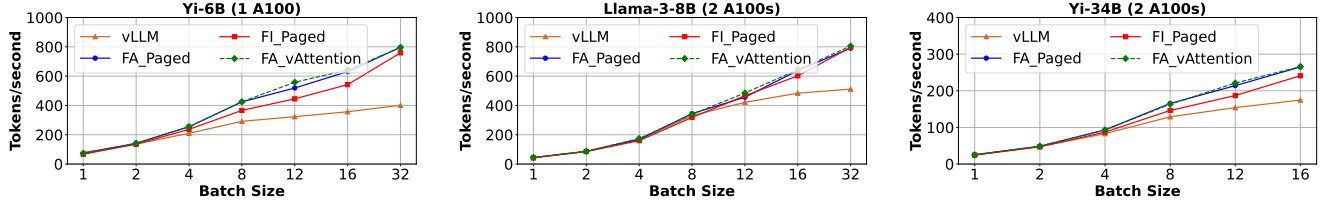


Figure 7. Decode throughput. FA_vAttention is on par with FA_Paged (note the overlapping lines) which is the best among all PagedAttention based alternatives, while outperforming FI_Paged and vLLM.

our experiments. Therefore, even for FlashAttention backend, vAttention outperforms the paged counterpart. For example, at context length 192K, FA_vAttention outperforms FA_Paged by 1.24 \times , 1.26 \times and 1.24 \times 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 \times and 1.36 \times for Yi-6B and Llama-3-8B (context length 16K), and 1.17 \times 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 the total prefill completion time in Table 5. For FlashAttention, 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 almost entirely 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.

6.2 Decode Evaluation

For decodes, in addition to FA_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’s non-paged kernel. Unfortunately FlashInfer’s non-paged decode kernel has significantly higher latency compared to all the other kernels we used in these experiments (up to 14.6 \times , Table 10). Hence, while vAttention supports dynamic memory allocation for FlashInfer’s non-paged decode kernel, we omit it for evaluation in this section (note that vAttention is responsible only for allocating memory – it cannot make a slow kernel fast).

Figure 7 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 (see Figure 7).

Model	BS	vLLM	FA_Paged	FI_Paged	FA_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 6. Total latency of attention kernel (sum of all layers) per decode iteration (in milliseconds, BS = batch size).

First, vAttention is on par with the best of PagedAttention as shown by FA_Paged and FA_vAttention in Figure 7. In comparison, FI_Paged has somewhat lower throughput and vLLM is the worst for all models and configurations. For example, FA_Paged and FA_vAttention outperform vLLM by up to 1.99 \times , 1.58 \times and 1.53 \times 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 based kernels; while FlashAttention has continuously adopted new optimizations (e.g., FlashDecoding [8]), vLLM has lagged behind. For example, Table 6 shows that vLLM’s PagedAttention kernel incurs up to 2.8 \times , 1.5 \times , and 2.5 \times higher latency than FlashAttention 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 adopting new optimizations.

Second, relative gains of FA_vAttention and FA_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 \times to 1.58 \times . This is because the latency of a decode attention kernel is proportional to the total number of tokens in the batch [29]. Therefore, the contribution of attention kernel in the overall latency – and consequently gains with a more efficient kernel – increase with the batch size. Further, for the same reasons as discussed in §6.1 (faster attention and lower CPU overhead), FA_vAttention delivers up to 1.23 \times 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. We believe

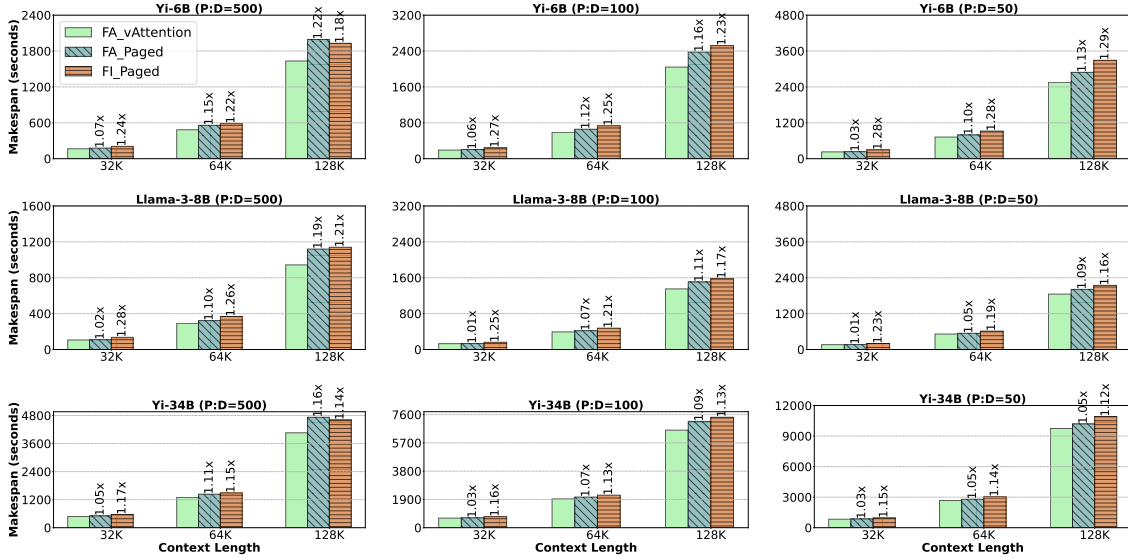


Figure 8. Makespan of serving a set of 50 requests with varying context lengths and the ratio of prefill to decode tokens.

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. This is evident from Table 6 wherein FA_Paged and FA_vAttention have similar latency. However, hiding compute overhead in a prefill attention kernel is hard because it is already compute bound (see Table 5 for latency gains in the prefill attention kernel).

6.3 End-to-end Throughput Evaluation

For end-to-end performance, we measure the makespan of serving a static trace of 50 requests for long context scenarios. To understand system performance for different workloads, we vary the initial context length of requests from 32K to 128K, and the ratio of prefill to decode tokens (referred as P:D) from 500 to 50. Note that a higher P:D as well as a longer context indicates that the workload is more prefill bound; we expect vAttention to improve performance in such cases.

Figure 8 shows the makespan for all the three models. For Yi-6B, FA_vAttention is up to 1.22x, 1.16x and 1.13x faster than FA_Paged for P:D ratio of 500, 100 and 50. FA_vAttention is also up to 1.19x faster for Llama-3-8B and up to 1.16x faster for Yi-34B, compared to FA_Paged. Our gains over FA_Paged are higher at higher P:D ratios, and for a fixed P:D, our gains increase as the context length grows. These results are consistent with our expectation since a higher P:D and context lengths both mean that the system spends more time processing prefills where use of non-paged attention kernels outperform paged kernels.

Interestingly, we observe that despite being faster than FA_Paged in prefills, FI_Paged under-performs FA_Paged in

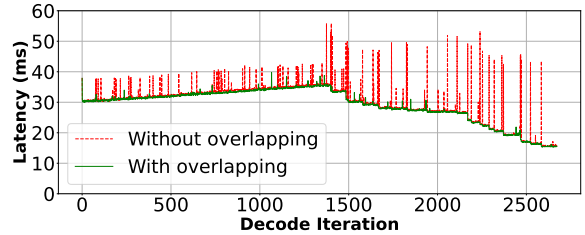


Figure 9. Latency of decode iterations with and without overlapping memory allocation with compute (batch size=32, context length=4K-8K, model: Llama-3-8B)

end-to-end throughput in most cases (due to inefficient decodes as shown in Figure 7). Hence, our gains over FI_Paged are relatively higher (up to 1.29x), except for some cases where FI_Paged outperforms FA_Paged (e.g., Yi-6B and Yi-34B, context length=128K, P:D=500).

6.4 Ablation Studies

6.4.1 Hiding allocation latency. Figure 9 shows the latency of more than 2500 decode iterations of Llama-3-8B. For this experiment, we used a batch of 32 requests and initialized the prefill context length of each request to be in between 4K-8K (chosen randomly). It is evident that overlapping memory allocation with compute effectively hides the latency of allocating memory via CUDA APIs. We used 2MB pages for this experiment to show that even the worst case latency can be hidden (Table 2 shows that 2MB pages incur highest latency). In contrast, allocating memory synchronously via CUDA APIs leads to frequent latency spikes

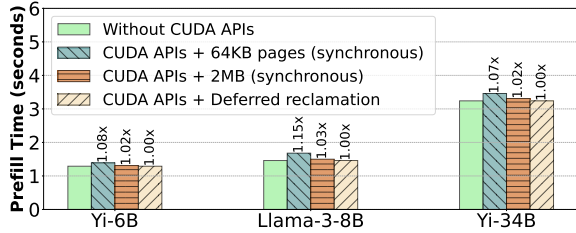


Figure 10. Prefill completion time of a single prompt of 16K tokens with different memory allocation strategies.

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

of 5ms–20ms, depending on how many requests need new physical memory pages.

6.4.2 Deferred reclamation. Figure 10 shows that synchronous memory allocation using CUDA APIs for prefills incurs 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 APIs for prefills because a newly arrived request can simply re-use physical memory pages that were allocated to a prior request. This way, deferred reclamation ensures that prefill latency is not affected by memory allocation.

6.4.3 Memory allocation bandwidth. Table 7 shows that even with 64KB (our smallest) page size, vAttention can allocate as much as 7.6GB per second per GPU. This is more than an order of magnitude higher than the maximum memory allocation rate of 600MB per second of decodes (Figure 4). Larger page sizes and higher TP dimensions increase the memory allocation rate proportionally. Therefore, memory allocation bandwidth of CUDA APIs is more than enough for LLM inference.

6.4.4 Effect of page size. In many applications, use of smaller pages can potentially degrade performance due to TLB thrashing [45, 48, 50, 55]. We find that this is not the case for LLM inference. For example, Figure 11 shows that the execution latency of an attention kernel remains largely unaffected when the page size for KV-cache is reduced to 64KB, from the original 2MB. 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.

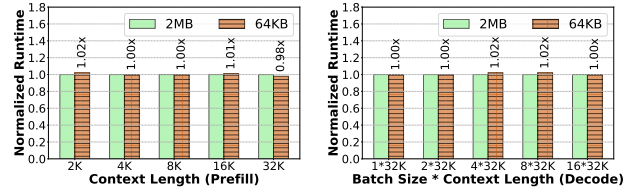


Figure 11. Effect of page size on the performance of FlashAttention’s prefill (left) and decode (right) attention kernels (model: Llama-3-8B).

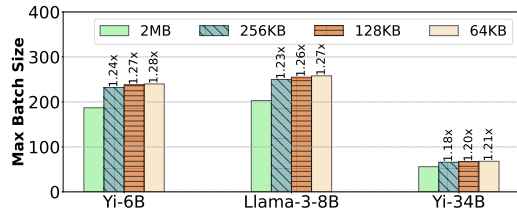


Figure 12. Maximum batch size obtained with different page sizes for a dynamic workload. Large pages limit batch size due to internal fragmentation.

Further, we find that 2MB pages are good enough for many online serving scenarios where latency constraints limit the maximum batch size. However, for throughput-oriented scenarios, smaller pages may be preferred. 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 [57], load: 7 queries per seconds). In contrast, 64KB pages helped serve batch sizes of up to 240, 258 and 68 for our models (see Figure 12).

6.5 Programming Effort

vAttention makes it feasible to replace one attention kernel with another with only a few lines of code changes in the serving framework (Figure 13). In contrast, in PagedAttention, a developer first needs to write a paged kernel and then make significant changes in the serving framework. For example, integrating FlashInfer decode kernels in vLLM required more than 600 lines of code changes in over 15

```

-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 13. Illustration of code changes needed to replace the prefill attention kernel of FlashAttention by FlashInfer when using vAttention for dynamic physical memory allocation.

Model	# Tokens in a physical memory block				Max memory waste per request			
	64KB	128KB	256KB	2MB	64KB	128KB	256KB	2MB
Yi-6B (TP-1)	64	128	256	2048	4MB	8MB	16MB	128MB
Yi-6B (TP-2)	128	256	512	4096	8MB	16MB	32MB	256MB
Llama-3-8B (TP-1)	32	64	128	1024	4MB	8MB	16MB	128MB
Llama-3-8B (TP-2)	64	128	256	2048	8MB	16MB	32MB	256MB
Yi-34B (TP-1)	32	64	128	1024	7.5MB	15MB	30MB	240MB
Yi-34B (TP-2)	64	128	256	2048	15MB	30MB	60MB	480MB

Table 8. Block size (number of tokens in a physical memory page) as a function of the page size and degree of tensor parallelism. Columns on the right show how much physical memory can be wasted per-request in the worst-case.

Model	Batch Size	FA_Paged (Block Size = 256)	FA_Paged (Block Size = 16)	Ratio
Yi-6B	1	0.085	0.085	1.00×
	2	0.093	0.094	1.01×
	4	0.109	0.115	1.05×
Llama-3-8B	1	0.076	0.082	1.08×
	2	0.097	0.105	1.07×
	4	0.136	0.143	1.05×
Yi-34B	1	0.070	0.076	1.09×
	2	0.088	0.095	1.08×
	4	0.116	0.117	1.01×

Table 9. Impact of varying block size on the execution latency (in milliseconds) of FlashAttention decode kernel.

Context Length	Batch Size	FI_NonPaged	FI_Paged	Ratio
8K	4	0.902	0.071	12.7×
8K	8	0.931	0.128	7.3×
8K	16	0.943	0.229	4.2×
16K	4	0.902	0.13	7.2×
16K	8	1.821	0.233	7.8×
16K	16	1.875	0.450	4.2×
32K	4	3.398	0.233	14.6×
32K	8	3.291	0.458	7.2×
32K	16	3.365	0.891	3.8×

Table 10. Latency (in milliseconds) of paged and non-paged decode attention kernels in FlashInfer. Ratio denotes the latency of non-paged kernel divided by the latency of paged kernel (model: Llama-3-8B, 2 A100 GPUs).

files [23, 24, 26]. Implementing the initial paging support in FlashAttention kernel also required ≈ 280 lines of code changes [20] and additional efforts to enable support for smaller block sizes [14].

7 Discussion

7.1 Further Analysis of Internal Fragmentation

vAttention uses page size as the unit of physical memory allocation as opposed to the PagedAttention’s unit of KV-cache block size. Therefore, it is natural to ask *what is the relationship between page size and block size*. To compute this relationship, let S be the size of a single token’s per-layer K-cache (or V-cache) on a worker (see §5.1). Then, page size t translates to KV-cache block size = t/S . Note that $S = H \times D \times P$, where H is the number of KV heads on a worker, D is the dimension of each KV head and P is the number of bytes based on model precision (e.g., $P=2$ for FP16/BF16). Therefore, block size = $t/(H \times D \times P)$.

Table 8 shows the block size for different model configurations and physical memory allocation sizes; we show each model under two TP configurations – TP-1 and TP-2 – to highlight the effect of TP dimension on block size. In addition, the table also shows how much physical memory can be (theoretically) wasted by vAttention due to internal fragmentation in the worst-case. The worst-case occurs when a new page is allocated but remains completely unused.

vAttention allocates physical memory equivalent to the page size on each TP worker whereas the per-token physical memory requirement of a worker goes down as TP dimension increases (because KV heads get split across TP workers). Therefore, block size increases proportionally with the TP dimension. Table 8 shows that using 2MB pages leads to large KV-cache block sizes of 1024 (Llama-3-8B and Yi-34B, TP-1) to 4096 (Yi-6B, TP-2) which could waste significant amount of physical memory (100s of MBs, per-request). Use of smaller (e.g., 64KB) pages reduces internal fragmentation proportionally, resulting in KV-cache block size of 32 (Yi-34B and Llama-3-8B, TP-1) to 128 (Yi-6B, TP-2), which in turn limit the maximum theoretical waste of only 4-15MB physical memory, per request. This shows that controlling the granularity of physical memory allocation is essential for reducing fragmentation. If required, page size can be reduced further to as low as 4KB which is the minimum page size supported in almost all architectures today, including NVIDIA GPUs.

7.2 Effect of Block Size on FlashAttention Kernel

We find that varying the block size of KV-cache also affects the execution latency of FlashAttention’s decode kernel (in addition to vLLM wherein using a larger block size increase the latency of decode kernel by up to 3× as shown in Figure 3). For example, Table 9 shows that using a smaller block

size of 16 increases the latency of FlashAttention decode kernel by up to 5% for Yi-6B, up to 8% for Llama-3-8B and up to 9% for Yi-34B. This is likely because work distribution and shared memory usage in a paged kernel needs to explicitly account for block size. Hence, writing an implementation that is equally efficient for all block sizes could be non-trivial. In fact, it is interesting to see that vLLM decode kernel performs better with small block sizes whereas FlashAttention decode kernel performs better with large block sizes. While the impact of block size on kernel latency is much less pronounced in FlashAttention compared to vLLM, a degradation in performance is still undesirable for an operator as important as attention. vAttention is free of such implementation challenges because it does not require a block table.

7.3 FlashInfer Non-paged Decode Attention Kernel

Note that we did not include FlashInfer non-paged decode kernel in our evaluation (§6). There are two reasons for this. First, the non-paged kernel has a much higher latency than the paged counterpart of FlashInfer e.g., Table 10 shows that the latency of non-paged kernel is up to 14.6× higher than the paged kernel. Further, the latency of the paged kernel follows an expected trend i.e., increasing the context length or batch size leads to a proportional increase in the execution latency. In contrast, increasing the batch size does not affect the latency of the non-paged kernel (e.g., context lengths of 8K and 32K). This behavior of the non-paged kernel is counter-intuitive and likely due to un-optimized implementation. Note that the job of a memory allocator is to simply allocate memory – it cannot accelerate a kernel. Second, the non-paged decode kernel of FlashInfer does not support variable sequence lengths – a fundamental requirement for LLM inference.

8 Related Work

Optimizing LLM inference is an active area of research. Various techniques have been proposed to improve different aspects of LLM serving like batching [30, 44, 61], disaggregation [43, 49, 65], scheduling [54, 58]. 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 [12], LightLLM [10], and libraries e.g., FlashAttention [7] and FlashInfer [9]. In contrast, vAttention offers an alternate – which we believe is also a more principled – approach to dynamic KV-cache memory management.

In a recent work, GMLake [40] 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 for large object allocations. In contrast, vAttention is focused on avoiding

fragmentation for LLM inference. Different from training, LLM inference is latency sensitive and requires smaller granularity allocations. We proposed various LLM inference specific optimizations to meet these requirements.

Similar to our approach, PyTorch also recently added support for expandable segments [15] that dynamically attach physical memory pages to a pre-reserved virtual memory buffer, if enabled. However, PyTorch uses expandable segments opportunistically, using synchronous memory allocation, and allocates physical memory pages only in multiples of 2MB granularity. In contrast, vAttention caters to the specific requirements of the KV-cache.

Some other concurrent works are also motivated to optimize attention kernels [25, 35, 42, 51, 52, 59, 63]. As one would expect, new research ideas do not start with paging support in mind. vAttention would make it easier to deploy them.

9 Conclusion

In this paper, we propose vAttention for dynamic memory management in LLM serving systems. The key highlight of vAttention is that it leverages system support for demand paging instead of implementing it in user space. We present various examples to show that the vAttention approach reduces programming burden while improving portability and performance compared to the popular PagedAttention approach that many LLM serving systems use today.

References

- [1] Amazon codewhisperer. <https://aws.amazon.com/codewhisperer/>.
- [2] Bing ai. <https://www.bing.com/chat>.
- [3] Github copilot. <https://github.com/features/copilot>.
- [4] Google bard. <https://bard.google.com>.
- [5] Replit ghostwriter. <https://replit.com/site/ghostwriter>.
- [6] Text generation inference. <https://huggingface.co/text-generation-inference>.
- [7] FlashAttention. <https://github.com/Dao-AI/flash-attention>, 2022.
- [8] Flash-Decoding for long-context inference. <https://crfm.stanford.edu/2023/10/12/flashdecoding.html>, 2023.
- [9] FlashInfer: Kernel Library for LLM Serving. <https://github.com/flashinfer-ai/flashinfer>, 2023.
- [10] Lightllm: A light and fast inference service for llm. <https://github.com/ModelTC/lightllm>, 2023.
- [11] Performance decay when using paged attention. <https://github.com/NVIDIA/TensorRT-LLM/issues/75>, 2023.
- [12] Tensorrt-llm: A tensorrt toolbox for optimized large language model inference. <https://github.com/NVIDIA/TensorRT-LLM>, 2023.
- [13] Use optimized kernels for MQA/GQA. <https://github.com/vllm-project/vllm/issues/1880>, 2023.
- [14] Add support for small page sizes. <https://github.com/Dao-AI/flash-attention/pull/824>, 2024.
- [15] CUDA semantics. <https://pytorch.org/docs/stable/notes/cuda.html>, 2024.
- [16] CUDA Toolkit Documentation: Virtual Memory Management. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__VA.html, 2024.
- [17] Custom CUDA kernels for KV cache copy operations. https://github.com/vllm-project/vllm/blob/main/csrc/cache_kernels.cu, 2024.

- [18] Faster Transformer. <https://github.com/NVIDIA/FasterTransformer>, 2024.
- [19] Fix eager mode performance. <https://github.com/vllm-project/vllm/pull/2377>, 2024.
- [20] Implement Page KV Cache. <https://github.com/Dao-AILab/flash-attention/commit/54e80a3829c6d2337570d01e78ebd9529c02d342>, 2024.
- [21] Meta-Llama-3-8B. <https://huggingface.co/meta-llama/Meta-Llama-3-8B>, 2024.
- [22] Pascal MMU Format Changes. <https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf>, 2024.
- [23] Refactor Attention Take 2. <https://github.com/vllm-project/vllm/pull/3462>, 2024.
- [24] Separate attention backends. <https://github.com/vllm-project/vllm/pull/3005/>, 2024.
- [25] Tile primitives for speedy kernels. <https://github.com/HazyResearch/ThunderKittens>, 2024.
- [26] Use FlashInfer for Decoding. <https://github.com/vllm-project/vllm/pull/4353>, 2024.
- [27] Yi-34B-200K. <https://huggingface.co/01-ai/Yi-34B-200K>, 2024.
- [28] Yi-6B-200K. <https://huggingface.co/01-ai/Yi-6B-200K>, 2024.
- [29] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S Gulavani, Ramachandran Ramjee, and Alexey Tumanov. Vidur: A large-scale simulation framework for llm inference. *Proceedings of The Seventh Annual Conference on Machine Learning and Systems, 2024, Santa Clara, 2024*.
- [30] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve, 2024.
- [31] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills, 2023.
- [32] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.
- [33] Pratheek B, Neha Jawalkar, and Arkaprava Basu. Designing virtual memory system of mcm gpus. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '22*, page 404–422. IEEE Press, 2023.
- [34] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020.
- [35] Ganesh Bikshandi and Jay Shah. A case study in cuda kernel fusion: Implementing flashattention-2 on nvidia hopper architecture using the cutlass library, 2023.
- [36] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers, 2019.
- [37] 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 Sankaranarayanan 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. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022.
- [38] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.
- [39] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.
- [40] Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, Minyi Guo, Hao Wu, Shouren Zhao, Junping Zhao, and Ke Zhang. 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, ASPLOS '24*, page 450–466, New York, NY, USA, 2024. Association for Computing Machinery.
- [41] 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. DeepSpeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference, 2024.
- [42] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. Flashdecoding++: Faster large language model inference on gpus, 2024.
- [43] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024.
- [44] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *SOSP '23*, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.
- [45] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 705–721, Savannah, GA, November 2016. USENIX Association.
- [46] Ajay Nayak, Pratheek B., Vinod Ganapathy, and Arkaprava Basu. (mis)managed: A novel tlb-based covert channel on gpus. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security, ASIA CCS '21*, page 872–885, New York, NY, USA, 2021. Association for Computing Machinery.
- [47] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.
- [48] Ashish Panwar, Sorav Bansal, and K. Gopinath. 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, ASPLOS '19*, page 347–360, New York, NY, USA, 2019. Association for Computing Machinery.
- [49] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting, 2023.
- [50] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 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, ASPLOS '14*, page 743–758, New York, NY, USA, 2014. Association for Computing Machinery.
- [51] Rya Sanovar, Srikant Bharadwaj, Renee St. Amant, Victor Rühle, and Saravan Rajmohan. Lean attention: Hardware-aware scalable attention mechanism for the decode-phase of transformers, 2024.
- [52] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. 2024.
- [53] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

- [54] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.
- [55] Seunghee Shin, Michael LeBeane, Yan Solihin, and Arkaprava Basu. Neighborhood-aware address translation for irregular gpu applications. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 352–363, 2018.
- [56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.
- [57] Guan Wang, Sijie Cheng, Xianyuan Zhan, Xiangang Li, Sen Song, and Yang Liu. Openchat: Advancing open-source language models with mixed-quality data, 2023.
- [58] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models, 2023.
- [59] Mengdi Wu, Xinhao Cheng, Oded Padon, and Zhihao Jia. A multi-level superoptimizer for tensor programs, 2024.
- [60] 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. Accelerating self-attentions for llm serving with flash-infer, February 2024.
- [61] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.
- [62] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. 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, CCS '23*, page 960–974, New York, NY, USA, 2023. Association for Computing Machinery.
- [63] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Re, Clark Barrett, Zhangyang Wang, and Beidi Chen. \$h_2o\$: Heavy-hitter oracle for efficient generative inference of large language models. In *Conference on Parsimony and Learning (Recent Spotlight Track)*, 2023.
- [64] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H₂o: Heavy-hitter oracle for efficient generative inference of large language models, 2023.
- [65] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.