# POD-Attention: Unlocking Full Prefill-Decode Overlap for Faster LLM Inference

Aditya K Kamath*
University of Washington

Ramya Prabhu
Microsoft Research India

Jayashree Mohan
Microsoft Research India

Simon Peter
University of Washington

Ramachandran Ramjee
Microsoft Research India

Ashish Panwar
Microsoft Research India

## Abstract

Each request in LLM inference goes through two phases: compute-bound *prefill* and memory-bandwidth-bound *decode*. To improve GPU utilization, recent systems use hybrid batching that combines the prefill and decode phases of different requests into the same batch. Hybrid batching works well for linear operations as it amortizes the cost of loading model weights from HBM. However, attention computation in hybrid batches remains inefficient because existing attention kernels are optimized for either prefill or decode.

In this paper, we present POD-Attention[1] — the first GPU kernel that efficiently computes attention for hybrid batches. POD-Attention aims to maximize the utilization of both compute and memory bandwidth by carefully allocating the GPU's resources such that prefill and decode operations happen concurrently on the same multiprocessor. We integrate POD-Attention in a state-of-the-art LLM inference scheduler Sarathi-Serve. POD-Attention speeds up attention computation by up to 75% (mean 28%) and increases LLM serving throughput by up to 22% in offline inference. In online inference, POD-Attention enables lower time-to-first-token (TTFT), time-between-tokens (TBT), and request execution latency versus Sarathi-Serve.

## 1 Introduction

The serving infrastructure of large language models (LLMs) is expanding to meet growing demands [10, 19]. Large-scale service providers often depend on expensive high-end GPUs to meet peak demand or latency targets [41]. Therefore, optimizing LLM serving systems has become crucial [23, 25, 37, 50, 55, 58, 59]. Importantly, the overall efficiency of a deployment depends on how well GPU resources are utilized.

From a resource utilization perspective, LLM inference is a challenging workload because different phases require different resources at different times [24–26, 59]. The processing of an LLM request begins with a highly parallel (hence compute-bound) prefill phase which is then followed by a memory-bound decode phase [26]. Serving LLMs efficiently, therefore, requires both high compute and high memory bandwidth. An ideal system would strive to maximize the utilization of both compute and memory. However, doing so is non-trivial because, for a given request, the prefill and decode phases occur at different times.

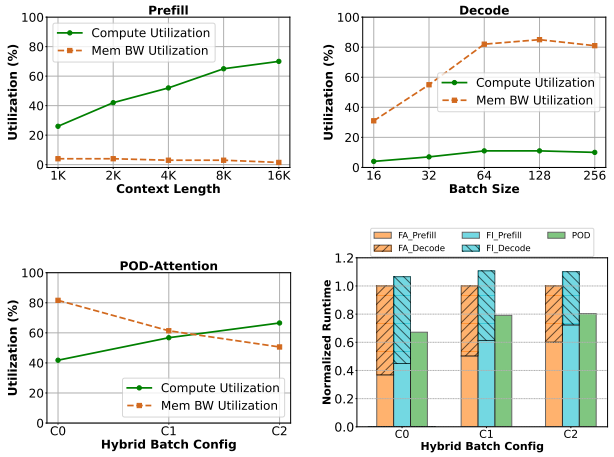



**Figure 1.** State-of-the-art attention kernels utilize either compute or memory (FA: FlashAttention, FI: FlashInfer). POD-Attention utilizes both compute and memory to accelerate attention computation in hybrid batches (see Table 1 for configurations. Model: Llama-3-8B on 2 A100 GPUs).

State-of-the-art LLM serving systems deal with this challenge by combining the inputs of prefill and decode phases of different requests into the same batch [26, 31, 55] — a technique we refer to as *hybrid batching*. Hybrid batching avoids the need to fetch model weights from GPU high-bandwidth memory (HBM) separately for prefill and decode tokens. Instead, it allows the GPU to fetch model weights once and use them to compute over both prefill and decode inputs. Hybrid batching also helps reduce tail latency: to limit the runtime of each iteration, the scheduler can divide long input prompts (prefill inputs) into multiple smaller chunks, then combine ongoing decodes with a new prefill chunk every iteration [25, 31]. As such, use of hybrid batching is common in various LLM serving systems today [25, 31, 37, 55, 59].

While prior work has focused on optimizing the linear operations [25, 31, 55], they do not optimize the attention computation of a hybrid batch. This is reasonable for a system that primarily deals with small context lengths since

*Work done as an intern at Microsoft Research India.
[1]Code available at https://github.com/microsoft/vattention/pod.

| Config. | Prefill | | | Decode | | Resource requirement |
|---|---|---|---|---|---|---|
| | BS | CS | CL | BS | CL | |
| C0 | 1 | 1K | 12K | 80 | 12K | memory-bound |
| C1 | 1 | 12K | 12K | 220 | 12K | balanced |
| C2 | 1 | 16K | 16K | 250 | 12K | compute-bound |

**Table 1.** Details of hybrid batches evaluated in Figure 1 (BS: batch size, CS: chunk size, CL: context length).

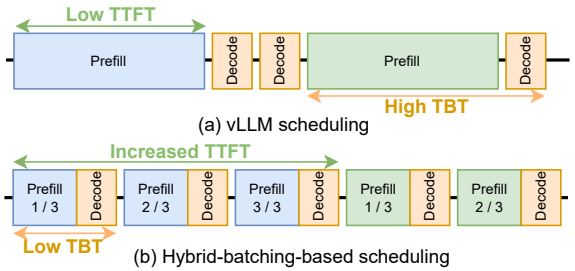



**Figure 2.** Impact of scheduling strategies on TTFT and TBT.

linear operations dominate run time in this setting [55, 59]. In contrast, as the context length increases, attention computation becomes the primary performance bottleneck (Figure 4).

Some recent works have also tried to optimize attention computation [8, 30, 32, 43], but current solutions address prefill and decode operations separately — maximizing compute utilization for prefills and bandwidth utilization for decodes, as shown in Figure 1. In this paper, we show that such an approach is suboptimal as it leaves critical GPU resources underutilized in different parts of computation. For example, Figure 1 illustrates that memory bandwidth utilization of the prefill attention kernel is often below 5%, while compute utilization of the decode attention kernel is under 10%. The effect of using independently optimized kernels is particularly noticeable with hybrid batching because prefill and decode kernels execute immediately one after the other, leading to periods of high demand of a resource immediately followed by low utilization of the same resource.

To improve the efficiency of hybrid batching, we present POD-Attention — the first GPU kernel, to the best of our knowledge, that efficiently batches the computation of prefill and decode attention. In doing so, we first show (§3) that existing techniques do not provide adequate performance in fusing attention computation due to various limitations such as straggler threads, synchronization barriers and lack of guaranteed SM-level co-location of different Cooperative Thread Arrays (CTAs) on GPU Streaming Multiprocessors (SMs). POD-Attention addresses these issues by fusing the computation in a CTA-parallel manner, introducing SM-aware software-based CTA scheduling within the GPU (§4). Building on state-of-the-art FlashAttention kernels [7], POD-Attention significantly accelerates attention computation by utilizing both compute and memory resources as per the requirement of a given batch of requests (see Figure 1).

We also integrate POD-Attention in a state-of-the-art LLM inference scheduler Sarathi-Serve [25]. Our experiments show that POD-Attention computes attention up to 75% faster (mean 28%) than the prefill and decode attention kernels of FlashAttention and FlashInfer. In terms of the end-to-end LLM inference performance, POD-Attention improves throughput by up to 22% while also reducing crucial latency metrics such as time-to-first-token (TTFT), time-between-tokens (TBT) and the end-to-end request execution latency over Sarathi-Serve.

**Contributions:** We make the following contributions:

- We highlight that independently optimizing prefill and decode attention kernels is not performant with hybrid-batching-based LLM inference.
- We present POD-Attention — a single GPU kernel that computes prefill and decode attention concurrently to try and utilize both compute and memory bandwidth simultaneously.
- We integrate POD-Attention in Sarathi-Serve and show that it enables high throughput and low latency LLM inference compared to current systems.

## 2 Background and Motivation

We first discuss why LLM serving systems use hybrid batching and then motivate the need to optimize attention computation. Finally, we provide an overview of GPU execution.

### 2.1 Large Language Model (LLM) Inference

LLMs process user inputs and outputs as tokens, internally represented as vectors. Each request during inference goes through two phases — prefill and decode [55]. The prefill phase processes the tokens of a user's prompt in parallel and produces the first output token, whose latency is called time-to-first-token (TTFT). Subsequently, the decode phase generates one output token (per-request) per-iteration autoregressively. The latency taken to generate each output token is called time-between-tokens (TBT). The prefill phase is highly parallel and compute bound while the decode phase is memory bound. Due to the parallel processing of a large number of tokens, the latency of a prefill iteration is generally higher than that of a decode iteration.

The distinct computational characteristics of prefill and decode operations create a throughput-latency tradeoff in LLM inference [25, 33, 41, 58], as illustrated in Figure 2. Since decoding is memory bound, using a large batch size improves throughput. The original vLLM scheduler [37] uses prefill-prioritizing scheduling to maximize the decode batch size (Figure 2(a)). This approach provides low TTFT, but at the cost of high TBT because a new request's prefill can pause ongoing decodes, causing *generation stalls* [25]. High TBT

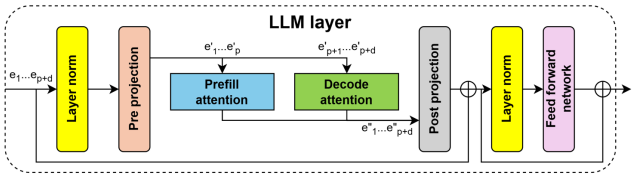



**Figure 3.** Computation in hybrid batches. Current systems compute prefill inputs ($e_1 \ldots e_p$) and decode inputs ($e_{p+1} \ldots e_{p+d}$) together for linear operations. However, they compute prefill and decode attention separately using specialized kernels.

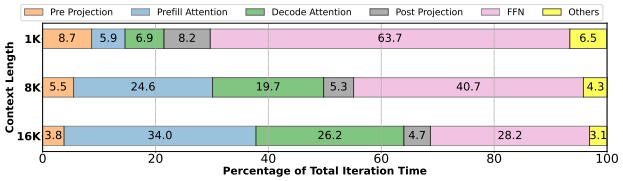



**Figure 4.** Contribution of different operations in iteration runtime with hybrid batching (model: Llama-3-8B, batch size: 60, chunk size: 1K). For each context length, we show runtime of iteration that processes the last chunk of a prompt.

is especially problematic in long-context scenarios, where each generation stall can last several seconds.

The issue of high TBT has been acknowledged in real-world deployments [17]. Sarathi-Serve [25] proposed *chunked-prefills* coupled with *continuous hybrid batching* [55] — a technique that divides the prefill tokens of a request into multiple smaller chunks and schedules one prefill chunk per-iteration with on-going decodes (Figure 2(b)). This way, Sarathi-Serve enables increasing batch size while avoiding generation stalls, improving both performance and user interactivity. Various LLM serving systems have incorporated this technique [9, 57, 59] including vLLM [21]. However, a downside is that it increases TTFT because each prefill chunk experiences interference from co-running decodes [26]. Since longer contexts are split into more chunks, the TTFT of long context requests can become significantly higher.

Figure 3 shows how hybrid batching works. Except attention, all other operations are linear i.e., computed element-wise. Linear operations obey the rule f(x + y) = f(x) + f(y) so inputs for a linear operation can be combined, computed upon by the same model weights to reduce memory accesses, and then separated. In contrast, attention is a sequence-level operator that is computed between three representations Q (query, of the current tokens being processed), and K/V (key/value, of all tokens in the sequence seen so far) as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{scale}\right)V$$

The QKV representations are further divided among multiple query heads and K/V heads, each assigned to a group [27]. Attention is computed in parallel for each Q head and K/V head pair. Since resource requirements of prefill and decode attention are different, state-of-the-art libraries such as FlashAttention (FA) [29, 30, 44] and FlashInfer (FI) [54] provide specialized kernel APIs, optimized separately for each phase. Use of these kernels works well in small context length scenarios where attention computation is a small fraction of the total inference time [26, 55].

However, the context length in many real-world LLM applications continues to grow [23, 50]. In such scenarios, attention computation dominates, becoming more than 60% of the total inference time in many cases as shown in Figure 4 (context length 16K). Note that prefill and decode attention are computed immediately one after the other in hybrid batches (see Figure 3). Therefore, *when independently optimized attention kernels are used, GPU execution goes through periods of high demand of a resource followed by low utilization of the same resource.* For example, the prefill kernel requires high compute but compute is (mostly) idle when the decode kernel executes.

We posit that concurrently computing prefill and decode attention can improve performance as it would utilize both compute and memory simultaneously. However, current techniques have several limitations with attention computation. To delve deeper into this, we first explain how GPUs operate and then present a case study of existing methods for executing different operations concurrently on GPUs (§3).

### 2.2 GPU Execution Model

The GPU's hardware is arranged in a hierarchy that supports execution at a scale of hundreds of thousands of parallel threads, depicted in Figure 5 [2]. The main processor unit of a GPU is a *Streaming Multiprocessor (SM)*, with modern GPUs containing around a hundred SMs. Each SM has an L1 cache and *shared memory* along with tensor cores for accelerated general matrix multiplication (GEMM) and execution units for integer/floating point operations. The shared memory is a user-addressable partition of the L1 cache. The GPU memory is accessed by SMs through the shared L2 cache.

GPU programming languages expose a hierarchy of threads that mimic the hardware hierarchy. The smallest unit of execution is a thread, while a group of 32 threads make up a *warp*, which typically execute concurrently in lockstep. To maximize throughput, GPU programmers ensure that threads within a warp execute the same code path. A *Cooperative Thread Array (CTA)* [3] is a group of warps that share the L1 cache and shared memory. All warps in a CTA are guaranteed to execute within a single SM.

Users launch GPU *kernels*, or GPU-executed functions, specifying the number of threads in the CTA, the number of CTAs in the kernel, as well as the required shared memory per CTA. This launch is then queued in a *stream*; operations

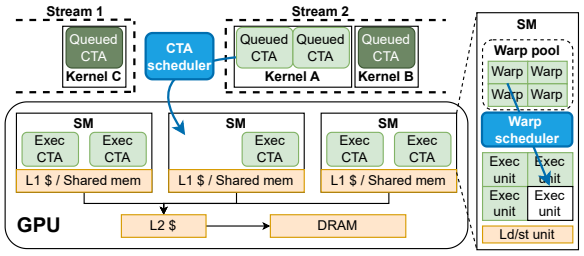



**Figure 5.** GPU execution model.

within a stream are serialized but different streams can execute in parallel in any order. The *CTA scheduler* selects CTAs from streams and assigns them to SMs when sufficient execution resources (e.g., threads, shared memory and registers) are available within the SM.

Central to the GPU's massive throughput is the fast, cycle-level *warp scheduler* baked into the hardware. Every clock cycle, the warp scheduler dispatches eligible warps for execution; a warp is eligible if its threads aren't stalled (e.g., waiting for memory access). This allows each SM to context switch at every clock cycle if required, effectively utilizing all its execution resources.

## 3 A Case Study on Concurrent Execution

The simplest way to compute prefill and decode attention together is to pass both inputs to an existing attention kernel. Some LLM serving systems prefer this method for computing attention in hybrid batches [12, 20]. In §5.1, we show that this is counter-productive and slower than serial execution.

In this section, we focus on GPU methods for concurrent execution e.g., running kernels in parallel or fusing their operations into a single kernel. We quantitatively analyze their performance and highlight key limitations that motivated us to develop a specialized attention kernel.

### 3.1 Methods of Concurrent Execution

Each level of the execution hierarchy in a GPU offers potential for concurrent execution (see Table 2).

1. **Kernel-parallel.** Streams can potentially execute different GPU kernels concurrently. This approach is easy to implement as it only requires submitting existing kernels to different streams; all other approaches require fusing different operations into a single kernel. Unfortunately, streams alone guarantees neither concurrency nor SM-level co-location of different operations [40, 56].
2. **CTA-parallel.** In this scheme, the CTAs in the kernel are split across operations in a predetermined manner. CTA-parallel enables better load-balancing: when one CTA finishes execution, the GPU scheduler can deploy the next CTA to the SM. However, similar to streams, CTA-parallel does not guarantee SM-level co-location.

| Execution method | GC | WQ | Notes |
|---|---|---|---|
| Streams [40] | × | ✓ | Easiest to implement |
| CTA | × | ✓ | Easy load balancing |
| Warp (e.g., HFuse [38]) | ✓ | × | Suffers from straggler problem |
| Intra-thread [48, 53] | ✓ | × | Cannot overlap with CTA barriers |
| SM-aware CTA (Ours) | ✓ | ✓ | Minimizes operation interference |

**Table 2.** Methods of concurrently executing or fusing different operations along different levels of the GPU execution hierarchy (GC=guarantees op co-location, WQ=reduces wave quantization).

| Config. | Description |
|---|---|
| FA_Serial | Serial execution with FA kernels |
| FA_Streams | Parallel execution via streams with FA kernels |
| FA_HFuse | Horizontally fused FA kernels with HFuse [38] |
| POD (Ours) | Optimized fused computation with our kernel |

**Table 3.** Different methods of computing attention in hybrid batches (FA: FlashAttention).

3. **Warp-parallel.** Here, warps within each CTA are split across operations, as proposed in horizontal fusion (HFuse [38]). This apprach guarantees co-location since all warps in a CTA are guaranteed to reside within the same SM. Unfortunately, warp-parallel fusion suffers from the straggler problem: a CTA prevents other CTAs from being scheduled on the same SM if one or more of its threads/warps are lagging behind others.
4. **Intra-thread.** In intra-thread fusion, each thread alternates between executing instructions of different operations [48, 53]. In simple cases, this strategy provides the maximum opportunity to overlap different operations. However, attention kernels use CTA-level sync barriers to coordinate fetching data into shared memory. These barriers limit intra-thread fusion as instructions before a barrier cannot be overlapped with those after the barrier.

We now quantitatively analyze the performance of different methods. Unfortunately, no readily available implementation exists for CTA-parallel and intra-thread fusion. Hence, we first analyze kernel-parallel and warp-parallel methods on attention kernels and then investigate other methods.

### 3.2 Analysis of Readily Available Methods

For kernel-parallel execution, shown as FA_Streams in Figure 6, we run FA's prefill and decode kernel on two different CUDA streams. For warp-parallel execution (FA_HFuse), we fuse FA's kernels using the toolchain provided by [38]. Figure 6 compares their performance against serial execution of FA's prefill and decode attention kernels (FA_Serial). Our experiment shows the per-layer attention computation time of Yi-6B for 32 chunks of a 16K prompt (chunk size 512), each co-scheduled with decodes of 16K context length each.

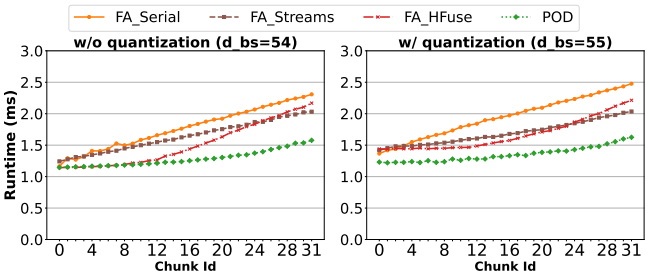



**Figure 6.** Per layer attention runtime of 32 hybrid batches corresponding to chunked prefills of a request of 16K tokens (chunk size: 512, model: Yi-6B, d_bs: decode batch size).

Note that if the number of CTAs in a kernel is not divisible by the number of GPU SMs, some of the SMs in the last wave of scheduling can remain idle — a phenomenon known as *wave quantization* [36, 39]. In the worst case, a marginal increase in work can double the latency of a kernel due to wave quantization. Therefore, to fully understand the benefit of concurrent execution, we evaluate performance with and without wave quantization. Each decode request uses 4 CTAs in our experiment (one CTA per KV head). Hence a decode batch size of 54 uses 216 CTAs having no wave quantization on our NVIDIA A100 GPU (108 SMs). In contrast, a batch size of 55 uses 220 CTAs leaving 4 quantized CTAs.

FA_Streams provides some speed up over FA_Serial and its gains are higher (up to 20%) when serial execution suffers from wave quantization. This is because streams run kernels in parallel to fill GPU SMs that would otherwise remain idle. This effect can be seen in Figure 6 where FA_Streams take roughly the same amount of time for both batch sizes while the time taken by FA_Serial increases at batch size 55; in particular, decode time increases by more than 25% in FA_Serial when batch size goes from 54 to 55 which increases the total attention time of prefill and decode by up to 17%. FA_HFuse outperforms FA_Streams is some cases but its performance degrades quickly due to straggler effect in the later chunks that are dominated by prefill. This happens because the prefill cost increases with each successive chunk but decode cost is same in all hybrid batches. Overall, FA_Streams and FA_HFuse both perform better than FA_Serial but still leave significant performance on the table as shown by POD-Attention which outperforms both methods by a significant margin.

### 3.3 Analysis of Other Methods

For complex kernels, such as attention, efficiently implementing fine-grained fusion schemes is non-trivial and prone to errors. Therefore, we analyze the performance of CTA-parallel and intra-thread fusion methods with a simple micro-benchmark consisting of a compute-bound kernel that repeatedly multiplies array elements with a scalar, and a memory-bound kernel that repeatedly adds three arrays. Each thread

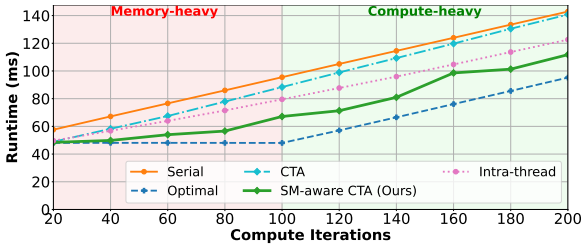



**Figure 7.** Fine-grained fusion versus serial computation.

executes a barrier after each operation. We vary the number of compute iterations to evaluate performance under varying compositions of compute-bound and memory-bound operations. Figure 7 shows the runtime of different fusion methods applied on these two functions. At 100 compute iterations, both operations consume equal time when executed serially. To the left of this point, memory bound is more dominant. To the right, it is compute bound. Figure 7 also shows the runtime achievable with an ideal oracle (i.e., perfect overlap).

CTA-parallel cannot guarantee SM-level co-location of compute-bound and memory-bound operations and hence provides only marginal average improvement of 8% over serial execution. Intra-thread fusion outperforms both serial and CTA-parallel execution, on average by 19% and 10%. However, the benefit of intra-thread fusion is limited due to sync barriers that hinder concurrent execution.

In summary, current methods for concurrently executing heterogeneous operations face several challenges, such as stragglers, barrier-induced delays, and the inability to guarantee SM-level co-location. In the following sections, we demonstrate how a specialized fused kernel, designed to leverage the characteristics of prefill and decode phases, can overcome these challenges.

## 4 POD-Attention

We introduce POD-Attention — a single GPU kernel that efficiently computes both prefill and decode attention. Our primary goal is to ensure that each GPU SM computes both operations simultaneously while minimizing resource contention between them. We build our kernel atop FA v2.6.1 [29].

To achieve our goal, we fuse computation along the CTA dimension that helps avoid the pitfalls of finer-grained warp-parallel and intra-thread fusion. In particular, CTA-parallel fusion offers three advantages: 1) it allows different CTAs to start and finish at different times independently of others, 2) ensures that sync barriers do not affect other parts of the computation since the effect of a barrier is limited to within its CTA, and 3) it is easier to program (§4.3). However, naive CTA-parallel fusion cannot guarantee that prefill and decode will be co-located on GPU SMs. To overcome this limitation, we introduce *software-based SM-aware CTA scheduling wherein each CTA decides whether to compute prefill or decode after it has been dispatched to an SM.*

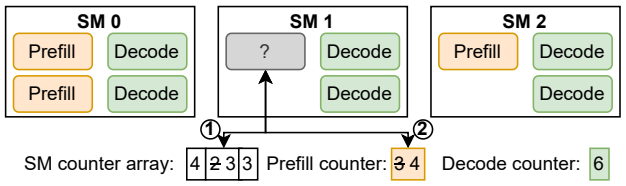



**Figure 8.** SM-aware CTA scheduling.

### 4.1 SM-aware CTA Scheduling

SM-aware CTA scheduling co-locates prefill and decode CTAs through "runtime operation binding". Here, a CTA decides whether to perform prefill or decode at runtime, after checking: 1) which SM it got launched on [52], and 2) what other CTAs running on the same SM are doing. This allows the kernel to remain completely agnostic to how the hardware scheduler assigns SMs to CTAs.

To do this, before launching the kernel, we determine how many CTAs are required for prefill and decode independently, and launch the kernel with CTAs matching the sum of both. Each SM has a counter keeping track of the number of CTAs launched on it along with 2 more counters that track the number of prefill and decode CTAs executed on it so far.

Figure 9 shows a simple code snippet of SM-aware CTA scheduling. When the hardware scheduler schedules a new CTA on an SM, a leader thread of the CTA (e.g., thread 0) reads the SMID hardware counter [4] that contains the unique ID of the SM it was launched on (lines 2 - 3). The thread then performs an atomic add operation on the SM counter to obtain a ticket (line 6). This ticket informs the thread as to which operation it should perform i.e., prefill or decode (lines 7 - 8), depending on the scheduling policy. The thread also increments the CTA counter for the operation (line 10). If this exceeds the maximum CTAs for that operation, it switches operations (line 12 - 18). Finally, it writes this information to shared memory so that the other threads in the CTA can begin execution accordingly (lines 20 - 30). We examined two scheduling policies: 50:50 and proportional. In the 50:50 policy, subsequent CTAs on an SM alternate between prefill and decode. In contrast, the proportional policy (line 5) allocates CTAs based on the ratio of prefill and decode CTAs in the current batch.

### 4.2 Performance Optimizations

Simply co-locating prefill and decode operations does not yield optimal performance. In this subsection, we introduce various optimizations to maximize the benefit of fusing prefill and decode attention computation.

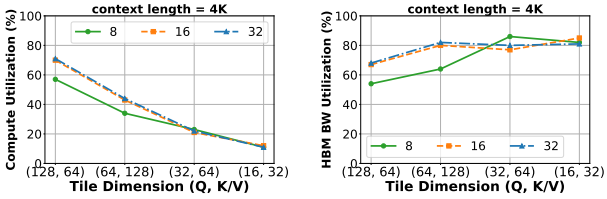
**4.2.1 Tile Sizes.** Data tiling is necessary to make effective use of tensor cores, which provide ~8× higher throughput than their CUDA core counterpart [6]. Tiling also helps improve shared memory usage. However, the benefit of tiling

```
1  if (threadIdx.x == 0) { // Leader thread finds assignment
2      int sm_id; // Find which SM this CTA is on
3      asm volatile("mov.u32 %0, %smid;" : "=r"(sm_id));
4      // For this SM, what do we want to run?
5      const int ratio = (prefill_ratio + decode_ratio);
6      int op, ticket = (atomicAdd(&sm_ctr[sm_id], 1) % ratio);
7      if(ticket < prefill_ratio) op = PREFILL;
8      else op = DECODE;
9      // Get the next CTA for operation
10     int cta_id = atomicAdd(&cta_assign[op], 1);
11     // If the CTA exceeds the max CTA for that op switch ops
12     if (op == PREFILL && cta_id >= prefill_ctas) {
13         op = DECODE;
14         cta_id = atomicAdd(&cta_assign[op], 1);
15     } else if (op == DECODE && cta_id >= decode_ctas) {
16         op = PREFILL;
17         cta_id = atomicAdd(&cta_assign[op], 1);
18     }
19     // Write the CTA ID and operation to shared memory
20     shared_mem[0] = cta_id;
21     shared_mem[1] = op;
22 }
23 __syncthreads(); // Barrier: waits for scheduling to finish
24 // Fetch the assigned CTA and operation.
25 int cta_id = shared_mem[0];
26 const int op = shared_mem[1];
27 __syncthreads();
28 // Perform the appropriate operation
29 if (op == PREFILL) prefill_op(cta_id);
30 else decode_op(cta_id)
```

**Figure 9.** CUDA code for SM-aware CTA scheduling.

is not uniform across operations. Decode operates on a single token per request, having a tile length of one across the query sequence length (QSL) dimension. In Group Query Attention [27], this length increases to the ratio between query and KV heads, typically 2 – 8. Due to this small dimension length, data reuse is insignificant, and performance is limited by memory bandwidth.

FlashAttention uses tile lengths of 64 – 128 for the QSL dimension. The side-effect of using such large tile sizes is that decodes end up zero padded, causing redundant compute [32]. For example, Figure 10a shows that compute utilization of the decode attention kernel is proportional to tile sizes, reaching up to 70% at QSL tile dimension of 128, compared to 10% with tile dimension of 16. However, note that decode attention is memory bound and hence, the primary objective of a decode kernel is to try and saturate memory bandwidth. Figure 10b shows that even at a relatively large QSL tile dimension of 64, the decode kernel is able to maximize memory bandwidth utilization. Hence, for




**(a)** Compute utilization. **(b)** DRAM BW utilization.

**Figure 10.** Impact of decode tile size on compute and HBM BW utilization for batch sizes 8, 16 and 32.

a decode-only attention kernel, there is little incentive to reducing tile sizes further.

In contrast, using large tile sizes for decodes is counterproductive in a fused kernel: any redundant compute performed by decodes interferes with co-located prefills since tensor cores are shared between them. If we reduce unnecessary computation, prefill can make better use of the tensor cores. To do so, we use a decode tile length of 16 for QSL, the minimum needed by CUTLASS [16] for A100 tensor operations. This drops the compute utilization of decodes to ~10%, freeing up tensor cores for prefill. Figure 10b shows that reducing tile size has no adverse impact on decode performance at large batch sizes.

**4.2.2 Concurrent CTAs per SM.** The number of CTAs running concurrently on an SM dictates the amount of resources (e.g., shared memory) each CTA can have. More CTAs per SM implies less resources per CTA, but more opportunities for fine-grained scheduling and co-location, i.e., with 2 CTAs per SM we can only co-locate prefills and decodes in a 1:1 ratio, but with 4 CTAs per SM, we can allocate CTAs to prefill and decode in different proportion depending on batch composition e.g., 3 CTAs to prefill and 1 CTA to decode. In general, prefills benefit from fewer CTAs per SM as it allows each CTA access to more shared memory, enabling use of larger tile sizes. In contrast, decodes do not benefit from larger tile sizes and therefore using more CTAs per SM can be beneficial since it allows fine-grained scheduling.

To achieve the best of both worlds, POD-Attention supports two configurations: 2 CTAs per SM for prefill-dominant hybrid batches and 4 CTAs per SM otherwise. Based on the desired configuration, we modify the tile lengths and number of threads used for prefill and decode. We also explored if 8 CTAs per SM can further improve performance and found that it only marginally improves performance in a few cases while under-performing in most cases. POD-Attention automatically picks the most suitable configuration at runtime.

**4.2.3 Virtual Decode CTAs.** The amount of shared memory provided to each prefill and decode CTA must be same in the fused kernel. However, because decode uses smaller tile sizes, the shared memory requirement of decode is a quarter of the prefill requirement. To avoid over-allocating shared memory to decodes, we divide each decode CTA into virtual CTAs containing a warp of threads. If the original decode CTA has four warps, each virtual CTA contains one warp which uses a quarter of the shared memory of the original CTA. The sum of shared memory used by all the virtual CTAs in each regular CTA is close to the shared memory used by prefill. This way, virtual decode CTAs balance the shared memory used by prefill and decode.

**4.2.4 Limiting Prefill Splits.** FlashAttention parallelizes computation across the query heads and QSL tile dimension. FlashDecoding [8], designed for decode which has a QSL of one, further splits the computation across the K/V dimension when there is not enough parallelism to fill the SMs of the GPU. The side-effect of this approach is that different CTAs fetch the same query tensor from memory independently of each other, proportional to the number of splits. Consequently, splitting the computation increases memory bandwidth utilization. While splitting along the key/value dimension is not required for prefills when the input contains enough tokens, chunked-prefills limit the number of tokens processed per-iteration *by design* (to minimize TBT). Therefore, FlashAttention also uses the FlashDecoding technique to accelerate the chunked-prefill attention computation. This scheme works well for a prefill-only kernel as increased parallelism can easily offset the cost of extra memory reads.

However, in a fused kernel, using a large number of splits for chunked-prefills can cause memory bandwidth contention between prefill and decode CTAs, potentially negating the benefit of fusion. To balance this trade-off, we limit the number of splits for a chunked-prefill to fill at most two full waves (determined empirically). This allows a chunked-prefill to use more CTAs when required, while ensuring that the number of splits do not get excessive and harm concurrent decodes.

### 4.3 Implementing CTA-parallel Fusion

To fuse the two kernels, we first convert them into generic device functions callable from within GPU code while removing all references to the CUDA-provided CTA ID (i.e., *blockIdx*), instead passing this as a function parameter. We build a wrapper kernel that calls these different functions using a calculated CTA ID. The prefill and decode operations execute as if the supplied CTA ID was their actual ID. This enables flexible remapping of CTA IDs, e.g., CTA 0 of the fused kernel can invoke prefill with CTA ID 0, CTA 1 can call decode with ID 0, CTA 2 can call prefill with ID 1, and so on. The amount of shared memory each CTA gets is fixed at kernel launch time, and prefill and decode operations have different requirements. To manage this, we hand-tune the shared memory usage of both prefill and decode operations to balance their requirements while minimizing performance degradation. We launch our fused kernel with enough shared memory for the maximum needed by either operation. To implement virtual CTAs, we modify the decode function replacing all CTA-level barriers with warp-level barriers. The decode function in the fused kernel is called with the appropriate virtual CTA ID, instead of the assigned CTA ID.

## 5 Evaluation

Our evaluation answers the following questions:

- What is the effect of POD-Attention on attention computation latencies?
- How does POD-Attention affect end-to-end LLM inference performance?

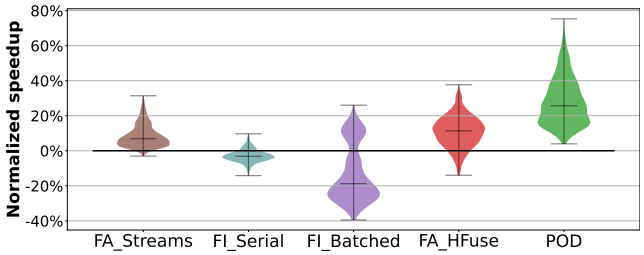



**Figure 11.** Distribution of speedup in attention computation with different mechanisms compared to FA_Serial.

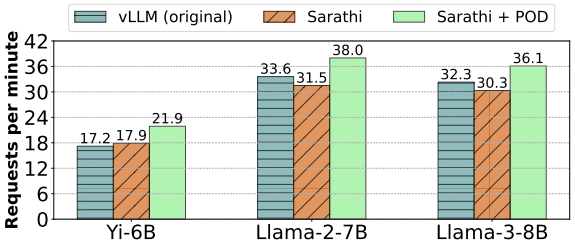



**Figure 12.** Serving throughput in offline inference.

- What is the impact of different optimizations and design choices employed in POD-Attention?

**Models and environment:** We evaluate POD-Attention with Yi-6B (4 KV heads [22]), Llama-2-7B (32 KV heads [11]) and Llama-3-8B (8 KV heads [13]), deploying Yi-6B on one A100 GPU, and others on two A100 GPUs with tensor parallelism. Each model has 32 query heads. Each GPU has 80GB memory.

**Workloads and metrics:** We evaluate both offline and online inference scenarios. For offline inference, we report the number of requests processed per minute. For online inference, we report TTFT, TBT and request execution latency on two workloads consisting of 2K requests each, and context length ranging from 4K to 32K tokens per-request. One of the workloads is an internal enterprise workload (mean context length of 10.5K tokens, per-request prefill to decode token ratio i.e., P:D in the range of 0 – 40) and the other is based on arXiv-Summarization [1] (mean context length of 9.5K tokens, P:D ratio of 0-50). On average, the number of decode tokens in arXiv workload is 42% higher (470) than the internal workload (331).

**Serving system baselines:** Our experiments use Sarathi-Serve [18] as the serving framework, which is built atop vLLM [5] . We evaluate two baselines: 1) the original vLLM scheduler [37] that runs prefills and decodes in separate batches, prioritizing prefills over decodes and 2) Sarathi-Serve [25]. Both baselines use FlashAttention kernels (v2.6.1) for attention computation. We integrate POD-Attention into Sarathi-Serve to evaluate the benefits of our optimizations. For simplicity, we refer to Sarathi-Serve without and with POD-Attention as Sarathi and Sarathi+POD.

### 5.1 Accelerating Attention with POD-Attention

Figure 6 illustrates a specific instance where POD-Attention accelerates attention computation, outperforming the next best alternative by up to 29%. To demonstrate the broad applicability of POD-Attention, we conducted a comprehensive sweep across over a thousand hybrid batches on our models. In these experiments, we varied the context length from 4K to 20K and the prefill chunk size from 512 to 2K. We focused on scenarios where prefill and decode attention account for at least 20% of the serial runtime, as other cases offer limited potential for optimization through operation fusion.

In addition to FlashAttention kernels, we also compare the runtime of FlashInfer (FI) kernels [54] in two configurations: FI_Serial and FI_Batched. FI_Batched computes prefill and decode attention using the prefill kernel of FlashInfer. We compare against FI_Batched for two reasons: 1) this strategy is the easiest way to compute prefill and decode attention together, and 2) some systems prefer this method e.g., Sarathi used FI_Batched in its default attention back-end [12], and a similar feature is requested in vLLM [20]. However, we show that this strategy is inefficient e.g., when FI_Batched uses a prefill-optimized kernel, it leads to redundant compute in decode computation due to use of larger tile sizes (§4.2.1). This redundant computation interferes with co-running prefill. Similarly, interference would occur on memory-bandwidth if FI_Batched uses a decode-optimized kernel.

Figure 11 shows the relative speedup for different mechanisms compared to FA_Serial. FA_Streams provides limited speedup as it cannot guarantee SM-level overlap of operations. In rare cases, we find that the overhead of stream synchronization can also negate its benefits. FI_Serial is mostly on par with FA_Serial but FlashInfer's prefill kernel is somewhat slower than that of FlashAttention [42]. FI_Batched improves performance at low context lengths, but degrades at higher lengths by up to 40% due to redundant computation for decodes. FA_HFuse is the strongest baseline as it guarantees operation overlap, improving median performance by 11%. However, FA_HFuse is susceptible to the straggler effect due to which it is slower by up to 16% in some cases compared to FA_Serial. The straggler effect can also be seen in Figure 6 towards the later chunks where prefill becomes more dominant, making it hard to achieve perfect utilization.

POD-Attention reaches a peak speedup of 75%, and a mean of 28% — higher than all alternatives. We found that in 18% of cases, it also reaches within 10% of the theoretical peak speedup, signifying near-perfect overlap. Furthermore, unlike other alternatives, POD-Attention never under-performs serial execution. These results underline the importance of a specialized attention kernel for hybrid batching based LLM inference.

| QPS | System | TTFT | | TBT | | Request Latency | | % Requests with Stalls | |
|---|---|---|---|---|---|---|---|---|---|
| | | P50 | P99 | P50 | P99 | P50 | P99 | 200ms | 500ms |
| 1.1 | vLLM (original) | 0.67 | 10.11 | 0.04 | 1.13 | 25.05 | 91.01 | 99.95 | 97.8 |
| | Sarathi | 2.2 | 12.58 | 0.10 | 0.15 | 26.83 | 92.24 | 2.05 | 0 |
| | Sarathi+POD | 1.9 | 12.26 | 0.10 | 0.14 | 24.70 | 79.04 | 3.17 | 0 |
| 1.2 | vLLM (original) | 0.94 | 12.70 | 0.07 | 1.76 | 42.73 | 151.8 | 99.95 | 99.6 |
| | Sarathi | 25.44 | 57.83 | 0.12 | 0.16 | 67.12 | 140.5 | 5.07 | 2.63 |
| | Sarathi+POD | 7.49 | 23.78 | 0.11 | 0.15 | 38.69 | 106.8 | 2.29 | 0 |

**Table 4.** Internal workload. Latency numbers in seconds.

| QPS | System | TTFT | | TBT | | Request Latency | | % Requests with Stalls | |
|---|---|---|---|---|---|---|---|---|---|
| | | P50 | P99 | P50 | P99 | P50 | P99 | 200ms | 500ms |
| 0.85 | vLLM (original) | 0.55 | 6.26 | 0.03 | 0.82 | 20.53 | 234.93 | 99.9 | 97.8 |
| | Sarathi | 2.68 | 14.89 | 0.08 | 0.13 | 27.87 | 281.07 | 4.15 | 2.05 |
| | Sarathi+POD | 1.85 | 12.71 | 0.08 | 0.11 | 24.31 | 255.75 | 1.85 | 1.61 |
| 0.95 | vLLM | 0.71 | 8.25 | 0.06 | 1.36 | 36.86 | 401.2 | 99.9 | 99.45 |
| | Sarathi | 46.22 | 144.2 | 0.1 | 0.14 | 90.12 | 417.6 | 4.44 | 1.9 |
| | Sarathi+POD | 11.74 | 27.38 | 0.09 | 0.12 | 40.6 | 333.0 | 2.2 | 2.1 |

**Table 5.** arXiv-based workload. Latency numbers in seconds.

### 5.2 Evaluating Throughput in Offline Inference

For evaluating offline inference scenarios, we run long context requests of 16K tokens each. We use chunk size 512 for Yi-6B, and 1K for both Llama-2-7B and Llama-3-8B, chosen in a way that chunking a prompt does not reduce the performance of linear operations (as recommended by Sarathi [25, 26]). We run 1K total requests for Yi-6B, and 2K requests each for Llama-2-7B and Llama-3-8B such that the total runtime of a single configuration is about one hour. The number of output tokens per-request is set to 2K for Yi-6B, 1K for Llama-3-8B and 256 for Llama-2-7B; we study the effect of varying prefill to decode token ratio (P:D ratio) in §5.4.4.

Figure 12 shows that Sarathi+POD delivers the best throughput: 22%, 20% and 19% higher than Sarathi, and 27%, 13% and 12% higher than vLLM, for the three models. It is worth highlighting that chunked-prefills and hybrid batching involves a tradeoff. Chunking a prompt increases attention computation time due to repeated KV cache loads: computing attention of a prefill chunk requires reading KV cache of all prior chunks [58]. At the same time, fusing decode tokens with prefills helps execute linear operations more efficiently: model weights need not be read separately for prefills and decodes. Therefore, the relative performance of vLLM and Sarathi can vary depending on workload, model configuration and chunk size. In our experiments, Sarathi improves throughput slightly over vLLM for Yi-6B but underperforms it for Llama-2-7B and Llama-3-8B. Sarathi+POD fuses prefills and decodes in all operations to improve GPU resource utilization, thereby outperforms both baselines.

### 5.3 Evaluating Latency in Online Inference

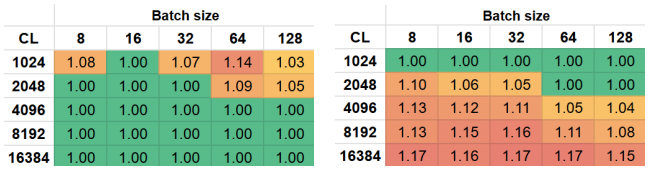
We evaluate Llama-3-8B on the internal and arXiv-based workloads near the serving capacity of the system: the maximum load a system can handle while avoiding high queuing delays [25]. We evaluate 2048 requests in each workload by varying the input load based on Poisson distribution. For Sarathi and Sarathi+POD, we use chunk size of 1024 for the arXiv-based workload, and 1536 for the internal workload which is more prefill-heavy. We discuss performance on important LLM-specific latency metrics of TTFT, TBT, and end-to-end request execution latency.

Note that there is an inherent trade-off between these metrics [25] and optimizing for one metric can severely compromise the others. For example, as will see below, vLLM prioritizes prefills and thus achieves low TTFT but sacrifices TBT, resulting in 95+% of user requests experiencing one or more stalls during decode generation. On the other hand, Sarathi reduces the stalls to a small % of user requests but significantly increases TTFT compared to vLLM.

**5.3.1 TTFT.** vLLM provides the lowest TTFT as it schedules a prefill on the first available opportunity. In comparison, Sarathi increases TTFT because the ongoing decodes interfere with prefills. TTFT in Sarathi further increases with the load, particularly due to higher queuing delays, e.g., the median TTFT goes to 25.4 and 46.2 seconds for the internal and arXiv-based workloads, compared to 0.94 and 0.71 seconds of vLLM. Sarathi+POD significantly reduces TTFT over Sarathi, bringing the median TTFT down to 7.5 and 11.74 seconds at higher load. Sarathi+POD also reduces the P99 TTFT by up to 4.3× over Sarathi.

| Latency Metric | vLLM (original) | Sarathi+POD 1024 | 1536 | 2048 |
|---|---|---|---|---|
| TTFT (P50) | 0.67 | 6.29 | 1.9 | 1.59 |
| TTFT (P99) | 10.11 | 18.99 | 12.26 | 12.40 |
| TBT (P50) | 0.04 | 0.08 | 0.10 | 0.08 |
| TBT (P99) | 1.13 | 0.11 | 0.14 | 0.18 |

**Table 6.** TTFT and TBT of Sarathi+POD with different chunk sizes versus vLLM (internal workload, QPS 1.1).




**(a)** 2 CTAs per SM. **(b)** 4 CTAs per SM.

**Figure 13.** POD-Attention with varying CTA configs.

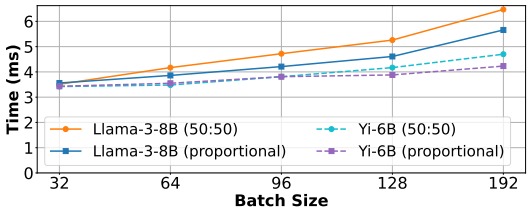



**Figure 14.** Effect of scheduling policy in POD-Attention.

#### 5.3.2 TBT and Stalls.

vLLM induces generation stalls by pausing on-going decodes whenever a new prefill is scheduled, resulting in poor interactivity with the LLM service. These generation stalls are reflected as high tail TBT latency, e.g., the P99 TBT of vLLM reaches up to 1.76 seconds (internal workload) and 1.36 seconds (arXiv-based workload). In the worst-case, we observe that the highest TBT latency reaches up to 8 seconds in vLLM when it computes multiple prefills consecutively. In comparison, Sarathi ensures that ongoing decodes do not get affected by a new prefill. Therefore, Sarathi provides significantly lower tail TBT latency compared to vLLM e.g., the P99 TBT of Sarathi is at most 0.16 seconds (10× lower than vLLM). Sarathi+POD further minimizes tail TBT over Sarathi by 10 – 20%. Crucially, since a single response results in a large number of decodes, *high TBT tail latency affects nearly all requests in vLLM*, signifying poor interactive experience for almost all users. Even if the TBT SLO is raised to 500ms, more than 97% of the total requests experience at least one stall in vLLM. In contrast, very few requests (<5%) observe a stall in Sarathi, which Sarathi+POD further reduces in most cases.

#### 5.3.3 End-to-end Request Latency.

Request latency can be used to approximate system throughput in online inference. Sarathi reduces P99 request latency over vLLM by 8% for the internal workload at QPS 1.2, but increase it by up to 24% over vLLM for the arXiv-based workload (QPS 0.85). Sarathi+POD is not only better than Sarathi in all cases, but also outperforms vLLM in many cases e.g., it reduces the P99 request execution latency by up to 42% over vLLM for the internal workload (106.8 seconds vs 151.8 seconds at QPS 1.2) and by up to 17% for the arXiv-based workload (333 seconds vs 401.2 seconds at QPS 0.95).

These results demonstrate that Sarathi enhances interactivity by reducing tail TBT and minimizing stalls, albeit with increased TTFT and some throughput reduction compared to vLLM. POD-Attention optimizes Sarathi 's performance across all metrics, effectively balancing the throughput-latency tradeoff. Table 6 shows that the chunk size in Sarathi+POD can be tuned further to navigate the TTFT and TBT trade-off, e.g., using a larger chunk size of 2K tokens lowers the median TTFT from 6.3 seconds to 1.6 seconds at the cost of higher TBT (P99 0.18 seconds vs 0.11 seconds).

### 5.4 Sensitivity Studies

#### 5.4.1 CTAs per SM.

Figure 13 shows the performance of POD-Attention with different numbers of CTAs running concurrently on an SM, varying batch sizes (horizontally) and context lengths (vertically) for Llama-3-8B. For each (context length, batch size) data point, we normalize the runtime to the best among the two configurations. In general, for long contexts where prefill cost dominates, 2 CTAs per SM performs better as it allows for larger tile sizes. As the context length decreases, the decode cost starts demonating and hence 4 CTAs per SM starts performing better: more CTAs per SM allows packing more decodes with fewer prefills, e.g., 1 prefill CTA and 3 decode CTAs.

#### 5.4.2 Scheduling Policy.

We explore two CTA scheduling policies within an SM, namely 50:50 allocation and Proportional allocation. In 50:50 allocation, CTAs launched on an SM alternate between prefill and decodes, i.e., the first CTA performs prefill, the next decode, and so on. This policy is agnostic to the total number of prefill and decode CTAs in the kernel. In Proportional allocation, the CTAs pick whether to perform prefill or decode depending on the total number of CTAs in the kernel. For example, if 50 prefill and 100 decode CTAs are required, the first CTA on each SM will perform prefill, the next two CTAs will perform decode, then repeat. Figure 14 shows the latency of POD-Attention with these policies for 8K context length and varying decode batch sizes on Yi-6B and Llama-3-8B. We notice that as the load increases (greater batch size), the performance of Proportional improves over 50:50 allocation. Proportional allocation spreads out the less frequent operations allowing better operational overlap and reduced resource contention, performing up to 14% better than a 50:50 allocation scheme.

| Chunk Id | FA_Serial | POD-Attention | |
|---|---|---|---|
| | | Vanilla split | Limited split [Ours] |
| 28 | 1.93 | 1.68 (0.87×) | 1.45 (0.75×) |
| 29 | 1.96 | 1.69 (0.86×) | 1.45 (0.74×) |
| 30 | 1.98 | 1.71 (0.86×) | 1.45 (0.73×) |
| 31 | 1.99 | 1.71 (0.86×) | 1.46 (0.73×) |

**Table 7.** Per-layer attention runtime (ms) of last four prefill chunks of a prompt, co-running with decode batch size 64 (model: Llama-3-8B, context length: 16K, chunk size: 512).

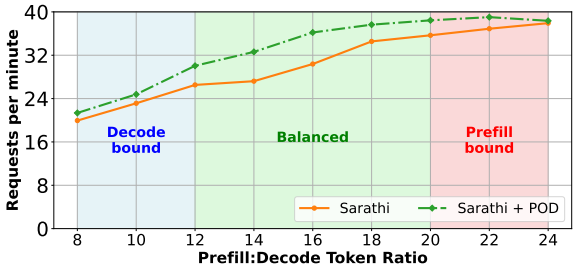



**Figure 15.** Request processing throughput under varying workload distribution (model: Llama-3-8B, TP-2).

#### 5.4.3 Limiting Prefill Splits.

POD-Attention reduces attention computation time with the default FlashDecoding-style splitting along the KV dimension. However, limiting the number of splits further improves performance. For example, Table 7 shows that in the last four chunks of a 16K prompt, co-running with 64 decode requests of the same context length, limiting the number of splits in prefill attention computation nearly doubles the speedup of POD-Attention over FA_Serial.

#### 5.4.4 Sensitivity to Workload.

POD-Attention accelerates the execution of hybrid batches and hence its impact on overall performance depends on how many iterations consist of hybrid batches in a given workload. A workload that is highly dominated by either prefills (high P:D ratio) or decodes (low P:D ratio) is likely to experience little benefit with POD-Attention. To understand the effect of varying P:D ratio, we benchmark Llama-3-8B with a total of 2048 requests, each consisting of ≈ 16.5K tokens, but with varying P:D ratio (in the range of 8 to 24) e.g., if the P:D is 10, then a request contains ≈15K prefill tokens and ≈1.5K decode tokens. Figure 15 shows that Sarathi+POD outperforms Sarathi over varying workload mixes. The peak gains occur in the P:D range of 12 to 18 because most batches are hybrid batches in this regime. In contrast, many iterations run decode-only batches when P:D ratio is lower than 12 (or prefill-only batches when P:D ratio is higher than 18).

## 6 Related Work

**Optimizing Attention Computation:** FlashAttention [30] introduced the first specialized implementation of attention, fusing all its operations into a single kernel with tile-based computation. FA-2 [29] improved it further with better work partitioning and load balancing. FlashDecoding [8] accelerates decode attention by splitting computation along the KV dimension. FlashDecoding++ [32] uses asynchronized softmax, double-buffered flat GEMM optimizations, and dataflow-based hardware resource adaptation to accelerate decode. LeanAttention [43] follows Stream-K reduction [39] of tiled calculation to enable better load distribution across SMs for decodes. FlashInfer [54] introduced shared-prefix based optimized attention kernels. Compared to works that separately handle prefill and decode, POD-Attention jointly optimizes and fuses them into a single kernel.

FA-3 [44] is a recent addition to the FlashAttention family of kernels. It leverages new features available in the NVIDIA Hopper architecture, exploiting the asynchrony of Tensor Cores, the Tensor Memory Accelerator, and the Special Function Units. FA-3 is still under active development at the time of this writing, and many optimizations like ping-pong scheduling were unavailable when POD-Attention was created. We leave extending POD-Attention support to FA-3 and Hopper architecture for future work.

**Operation Fusion:** Kernel fusion is a commonly used technique for improving GPU performance. Elastic kernels [40] proposes restricting resources to enable running multiple kernels concurrently. However, this method provides no guarantee of intra-SM co-location. To overcome this, ISPA [56] deploys a predetermined number of CTAs for each kernel, less than the number of CTAs that run concurrently on the GPU. Significant a priori profiling is used to determine the appropriate CTA sizes to allow for both kernels to execute concurrently. This can be tedious for attention kernels with dynamically changing input sizes, and makes load balancing between the prefill and decode operations difficult, as one operation completing early leaves resources underutilized. HFuse [38] fuses operations in warp-parallel fashion, providing source-to-source compilation tools to fuse kernels. SM-centric scheduling [52] uses the SM counter to assign work to CTAs, which we leverage in POD-Attention. We do not consider NVIDIA support for MPS [14] and MIG [15] in this paper since these features cater to multi-process scenarios which is different from our use case.

**Optimizing LLM Inference:** Optimizing LLM serving systems is an active area of research [25, 31, 34, 37, 41, 45–47, 50, 51, 55]. Orca [55] introduced iteration-level scheduling to eliminate compute fragmentation when requests of different lengths are batched together. PagedAttention [37] and vAttention [42] proposed different techniques for dynamic memory management for LLM inference. Sarathi-Serve [25] leverages chunked prefills to enable stall-free batching. In

contrast, Splitwise [41], DistServe [58] and TetriInfer [33] disaggregate the prefill and decode phases onto different GPU nodes to avoid interference between these phases. Various recent works have also proposed overlapping compute with communication to improve resource utilization [28, 35, 49].

Similar to POD-Attention, NanoFlow [59] also targets improving intra-device resource utilization, albeit with a contrasting approach. NanoFlow divides a batch into smaller operation-level nano-batches and schedules them in a way that overlaps operations with complementary resource profiles via CUDA streams. In contrast, POD-Attention tries to maximize resource-utilization within a given batch by fusing prefill and decode attention computation. While NanoFlow requires large batch sizes in order to benefit from batch splitting, POD-Attention is useful when attention consumes a significant amount of time. Therefore, NanoFlow seems more suitable for small-context scenarios whereas POD-Attention targets long-context scenarios that depend on hybrid batching for efficient LLM serving.

## 7 Conclusion

We introduce POD-Attention — the first attention kernel specialized to compute prefill and decode attention in parallel such that both compute and memory bandwidth can be utilized simultaneously. POD-Attention enables efficient hybrid batching based LLM inference by accelerating attention computation by up to 75% (mean 28%) compared to using independently optimized prefill and decode attention kernels. POD-Attention also improves the end-to-end serving throughput by up to 22%, while significantly reducing latency over state-of-the-art LLM serving systems Sarathi-Serve and vLLM.

## References


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

[2] CUDA C Programming Guide – Hardware Implementation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#hardware-implementation.

[3] Parallel Thread Execution ISA Version 8.5 – Cooperative Thread Arrays. https://docs.nvidia.com/cuda/parallel-thread-execution/#cooperative-thread-arrays.

[4] Parallel Thread Execution ISA Version 8.5 – Special Registers: %smid. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#special-registers-smid.

[5] vllm: Easy, fast, and cheap llm serving for everyone. https://github.com/vllm-project/vllm.

[6] Programming Tensor Cores in CUDA 9. https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/, 2017.

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

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

[9] Tensorrt-llm: A tensorrt toolbox for optimized large language model inference. https://github.com/NVIDIA/TensorRT-LLM, 2023.

[10] AI Infrastructure Spending Forecast to Be Over a Trillion Dollars Over the Next Five Years. https://www.delloro.com/news/ai-infrastructure-spending-forecast-to-be-over-a-trillion-dollars-over-the-next-five-years/, 2024.

[11] Llama-2-7B. https://huggingface.co/meta-llama/Llama-2-7b-hf, 2024.

[12] Merged PR 1865: Critical bug fixes related to sampling. https://github.com/microsoft/sarathi-serve/commit/50e59c51b85b1157e001bb8ee7a1b049d551955d#diff-450b0de5cce8a2341140afed859dc5dd3b913fa6e62d27988fccefeacc7b33ec, 2024.

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

[14] Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html, 2024.

[15] NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2024.

[16] NVIDIA/cutlass: CUDA Templates for Linear Algebra Subroutines. https://github.com/NVIDIA/cutlass, 2024.

[17] Performance and Tuning. https://docs.vllm.ai/en/v0.6.0/models/performance.html, 2024.

[18] Sarathi-Serve. https://github.com/microsoft/sarathi-serve, 2024.

[19] The State of AI Infrastructure at Scale 2024. https://ai-infrastructure.org/wp-content/uploads/2024/03/The-State-of-AI-Infrastructure-at-Scale-2024.pdf, 2024.

[20] Unify the kernel used in flash attention backend. https://github.com/vllm-project/vllm/pull/6052, 2024.

[21] Upstream Chunked Prefill. https://github.com/vllm-project/vllm/issues/3130, 2024.

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

[23] Amey Agrawal, Junda Chen, Íñigo Goiri, Ramachandran Ramjee, Chaojie Zhang, Alexey Tumanov, and Esha Choukse. Mnemosyne: Parallelization strategies for efficiently serving multi-million context length llm inference requests without approximations, 2024.

[24] 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.

[25] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, Santa Clara, CA, July 2024. USENIX Association.

[26] 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.

[27] 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.

[28] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Ziheng Jiang, Haibin Lin, Xin Jin, and Xin Liu. Flux: Fast software-based communication overlap on gpus through kernel fusion, 2024.

[29] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.

[30] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[31] 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.

[32] 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.

[33] 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.

[34] Haiyang Huang, Newsha Ardalani, Anna Sun, Liu Ke, Hsien-Hsin S. Lee, Anjali Sridhar, Shruti Bhosale, Carole-Jean Wu, and Benjamin Lee. Towards moe deployment: Mitigating inefficiencies in mixture-of-expert (moe) inference, 2023.

[35] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 402–416, New York, NY, USA, 2022. Association for Computing Machinery.

[36] Abhinav Jangda, Saeed Maleki, Maryam Mehri Dehnavi, Madan Musuvathi, and Olli Saarikivi. A framework for fine-grained synchronization of dependent gpu kernels. In *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '24, page 93–105. IEEE Press, 2024.

[37] 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.

[38] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. Automatic horizontal fusion for gpu kernels. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 14–27, 2022.

[39] Muhammad Osama, Duane Merrill, Cris Cecka, Michael Garland, and John D. Owens. Stream-k: Work-centric parallel decomposition for dense matrix-matrix multiplication on the gpu. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 429–431, New York, NY, USA, 2023. Association for Computing Machinery.

[40] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 407–418, New York, NY, USA, 2013. Association for Computing Machinery.

[41] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *ISCA*, June 2024.

[42] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. vattention: Dynamic memory management for serving llms without pagedattention, 2024.

[43] 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.

[44] 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.

[45] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 965–988, Santa Clara, CA, July 2024. USENIX Association.

[46] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu, 2023.

[47] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. Dynamollm: Designing llm inference clusters for performance and energy efficiency, 2024.

[48] Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound gpu applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 191–202, 2014.

[49] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 93–106, New York, NY, USA, 2022. Association for Computing Machinery.

[50] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism, 2024.

[51] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models, 2023.

[52] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 119–130, New York, NY, USA, 2015. Association for Computing Machinery.

[53] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 107–118, 2012.

[54] 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 flashinfer, February 2024.

[55] 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.

[56] Han Zhao, Weihao Cui, Quan Chen, and Minyi Guo. Ispa: Exploiting intra-sm parallelism in gpus via fine-grained resource management. *IEEE Transactions on Computers*, 72(5):1473–1487, 2023.

[57] 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. Sglang: Efficient execution of structured language model programs, 2024.

[58] 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. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, Santa Clara, CA, July 2024. USENIX Association.

[59] 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. Nanoflow: Towards optimal large language model serving throughput, 2024.