Bullet: Boosting GPU Utilization for LLM Serving via Dynamic Spatial-Temporal Orchestration

Zejia Lin
Sun Yat-sen University
Guangzhou, China
linzj39@mail2.sysu.edu.cn

Hongxin Xu
Sun Yat-sen University
Guangzhou, China
xuhx56@mail2.sysu.edu.cn

Guanyi Chen
Sun Yat-sen University
Guangzhou, China
felixlinker02@gmail.com

Xianwei Zhang*
Sun Yat-sen University
Guangzhou, China
zhangxw79@mail.sysu.edu.cn

Yutong Lu
Sun Yat-sen University
Guangzhou, China
luyutong@mail.sysu.edu.cn

Abstract

Modern LLM serving systems confront inefficient GPU utilization due to the fundamental mismatch between compute-intensive prefill and memory-bound decode phases. While current practices attempt to address this by organizing these phases into hybrid batches, such solutions create an inefficient tradeoff that sacrifices either throughput or latency, leaving substantial GPU resources underutilized. We identify two key root causes: 1) the prefill phase suffers from suboptimal compute utilization due to wave quantization and attention bottlenecks. 2) hybrid batches disproportionately prioritize latency over throughput, resulting in wasted compute and memory bandwidth. To mitigate the issues, we present `Bullet`, a novel spatial-temporal orchestration system that eliminates these inefficiencies through precise phase coordination. `Bullet` enables concurrent execution of prefill and decode phases, while dynamically provisioning GPU resources using real-time performance modeling. By integrating SLO-aware scheduling and adaptive resource allocation, `Bullet` maximizes utilization without compromising latency targets. Experimental evaluations on real-world workloads demonstrate that `Bullet` delivers 1.26× average throughput gains (up to 1.55×) over state-of-the-arts, while consistently meeting latency constraints.

1 Introduction

GPUs are the predominant computing platform for large language model (LLM) services [20, 23, 47, 63], empowering diverse applications with varying computational and latency requirements [1, 29, 30]. As these applications continue to advance, maximizing GPU utilization has become crucial for elevating serving quality [38, 71]. In response, a plethora of systems have been developed to optimize key aspects of LLM serving, such as kernel performance [19, 35, 50], scheduling strategies [4, 49, 70], and parallelization techniques [51, 60].

However, given the divergent computational characteristics in LLM inference workflow, achieving high resource utilization poses significant challenges [71]. In detail, the workflow involves a computationally intensive *prefill* phase that processes all inputs in parallel, succeeded by a memory-bound *decode* phase that generates tokens sequentially. These two contrasting phases lead to an imbalanced utilization of computational resources and memory bandwidth.

*Corresponding author.

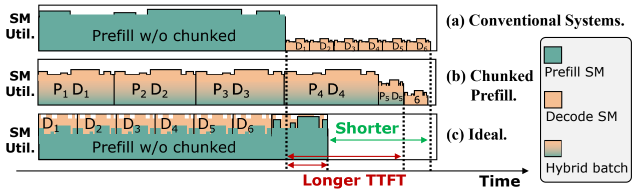


Figure 1: Request-level computation of conventional systems (top), chunked prefill (middle) and ideal (bottom).

The dynamism of workload patterns further widens the gap between the two phases, enforcing GPUs to switch between these complementary resource utilization states. Moreover, the distinct computational characteristics between the two phases introduces an unavoidable throughput-latency tradeoff [4]. For example, conventional systems [35] (Figure 1a) prioritize the prefill, which subsequently leads to a larger decode batch size. As a result, the strategy enhances throughput but incurs decode latency penalties, as the prefill operations exclusively occupy the GPU resources. Therefore, an ideal system necessitates fine-grained control to balance throughput, latency and GPU utilization, are non-trivial to achieve.

Prior attempts either isolate the two phases across different GPUs or coordinate in the same devices with *spatial* multiplexing. Prefill-decode disaggregation [20, 24, 49, 51, 70] implements the first strategy by physically separating the phases across dedicated GPU groups. However, such systems require careful determination of GPU counts for each phase, tailored to specific workload patterns [20], and may lack rapid reconfigurations for fluctuating request patterns [60]. Moreover, high-bandwidth interconnects are critically demanded to handle the significant state migration overhead [51, 70], greatly limiting the scenarios for deployment. In practice, chunked prefill [4, 5] (Figure 1b) has been pervasively employed in production systems [35, 43, 55, 69]. This technique leverages a fixed token budget to combine prefill and decode requests into *hybrid batches*, with longer sequences being partitioned into chunks to fit within capacity. While tunable chunk sizes enable latency control, the benefits inevitably compromise throughput [4, 50, 71]. The optimization presents inherent tensions, where smaller chunks sizes may operate below GPU saturation points, and larger ones sacrifice latency enhancements.

Despite throughput-latency tradeoff in LLM serving has been extensively studied [4, 50], we identify a critical limitation that, these approaches often lead to GPU under-utilization, resulting in suboptimal point stressing the balance of latency and hardware efficiency. **First**, the prefill phase achieves at most 75% of peak sustainable compute utilization regardless of sequence length. For short sequence, severe *wave quantization* effects [15] leaves an average of 19% computation unit cycles idle, while for long sequences, attention operations [19, 35], which maintain lower GPU utilization than linear layers dominates execution time. **Second**, the throughput-latency tradeoff in chunked prefill exhibits sub-linearly scaling with chunk sizes. Modest reductions in chunk size disproportionately degrade performance while yielding diminishing latency benefits, creating an inefficient optimization landscape. The key reason is that prefill and decode sequences are tightly coupled in the hybrid batches and executed in *lock-step* that under-utilizes resources, with an average of 64% compute and 16% memory bandwidth. **Third**, spatial sharing should properly leverage the natural complementarity between compute-intensive prefill and memory-bound decode phases through concurrent execution (Figure 1c) with minimal synchronization. However, the system still demands carefully orchestrating the concurrent phases to reduce contention and potential latency violation.

To mitigate the under-utilization issue and balance throughput-latency tradeoff, we propose `Bullet`, an LLM serving system to saturate GPU resources through spatial-temporal orchestration with precise provision. The system proactively monitors request progress and rapidly adjusts resource provision to maintain high utilization while adhering to latency requirements. `Bullet` achieves such efficient execution with four key components: 1) performance estimator (§3.2) that establishes a profile-augmented analytical model; 2) SLO-aware task scheduler (§3.3) to dynamically balance prefill and decode phases; 3) computational resource manager (§3.4) that offers lightning yet precise resource configuration; 4) concurrent execution engine (§3.5) for asynchronous CPU control flow and GPU execution.

In summary, the paper makes the following contributions:

- We identify the inefficiencies in maintaining high GPU utilization while navigating throughput-latency tradeoffs in existing LLM serving systems, and systematically analyze the chances for boosting the utilization.
- We establish accurate modeling for spatial-temporal shared phases, and propose fine-grained control to optimize resource provision aware of latency requirements.
- We develop an LLM serving system to effectively integrate the proposed techniques onto readily available frameworks.
- Experimental evaluations on real-world workloads show that `Bullet` outperforms state-of-the-arts in both latency and throughput while achieving higher GPU utilization.

2 Background and Motivation

2.1 LLM Computational Workflow

Recent LLMs [23, 47, 63] are mainly built by stacking Transformer blocks [57], with each containing four core components: QKV-projection layer, self-attention computation, Output-projection layer and multi-layer perception (MLP). These operators are primarily implemented through general matrix multiplications (GEMMs), with element-wise operations interspersed between layers. The self-attention specifically operates on query, key, and value matrices generated by the QKV-projection, with the computational efficiency significantly enhanced through optimized kernel implementations like FlashAttention [19].

The LLM inference pipeline operates through two distinct computational phases. During the initial prefill phase, the system processes all input tokens in parallel to generate the first output token, with the latency measured as time-to-first-token (TTFT). This compute-intensive stage performs full attention computations across the entire sequence while building the KV cache to store intermediate key-value states. Subsequently, the decode phase generates tokens sequentially, with each iteration consuming only the most recent output token to produce the next. The average iteration latency is termed time-per-output-token (TPOT). This memory-bound phase primarily reads from the KV cache while performing minimal computation for the current token's transformations.

2.2 GPU Utilization Principles

2.2.1 Execution Model and Theoretical Performance Bound. Modern GPUs employ a hierarchical architecture with hundreds of streaming multiprocessors (SMs), each containing general-purpose cores and specialized matrix units like Tensor Cores [17]. GPU's grid-block-thread programming model [18] aligns with this architecture, where kernels are organized into grids of thread blocks (TBs) that each manage thousands of cooperating threads. Upon kernel launch, it enters an asynchronous task queue (termed *stream* in CUDA[18]) for scheduling. The hardware scheduler retrieves kernels from these queues and dispatches them across SMs, enabling the concurrent kernel execution (CKE) [25, 46, 59] from different streams when required resources are satisfied.

During execution, multiple TBs can reside on the same SM to share the hardware resources. The SM executes the *warps* (32-thread groups) in a *wavefront* fashion to interleave different instructions and maximize hardware utilization. However, if the number of TBs is not evenly divisible by the number of SMs, a workload imbalance situation called *wave quantization* [15, 48] occurs. Some SMs finish early and remain idle while waiting for others to complete the tail wave. Formally, given a grid size of g TBs and a GPU with M SMs, the kernel demands $w = \lceil g/M \rceil$ waves to complete, with only $g - M \cdot (w - 1)$ SMs active in the final wave. Equation 1 computes the idle SM cycle ratio to the total number of cycles in a kernel.

$$s = \frac{M - tail}{Mw} = 1 - \frac{g}{M \cdot \lceil g/M \rceil} \tag{1}$$

GPU kernels typically use power-of-2 grid sizes to match data dimensions, but this conflicts with non-power-of-2 SM counts in GPUs. For example, 108 for Nvidia A100 [16] and 132 for H100 [17]. Therefore, wave quantization remains an open issue [34, 48] across diverse kernels. This inefficiency is particularly pronounced

Table 1: Theoretical SM idle ratio (%) caused by wave quantization effects, normalized to kernel/layer's execution time.

Sequence Length	QKV	Attn	OProj	MLP	Total
1024	11.1	21.0	40.7	13.0	19.4
2048	11.1	5.2	21.0	7.6	10.4
4096	11.1	5.2	5.2	7.6	9.1
16384	1.9	0.2	0.2	0.4	0.5

in Transformer's self-attentions [19, 57] and small-shaped GEMMs for short input sequences or small *chunked prefill-sizes* (§2.3.1) [4]. Additional under-utilization cause arises from memory-bound kernels like LLM's decode phases and element-wise operators, which idle compute resources during frequent memory accesses.

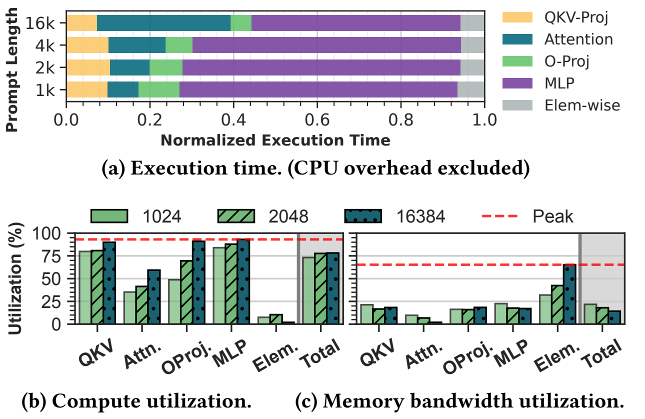
2.2.2 GPU Resource Sharing. Naturally, compute- and memory-bound kernels are suitable to co-execute on GPU, saturating both compute and bandwidth resources [9, 27, 39, 67]. The complementary nature of the prefill and decode phases makes them ideal for such concurrent exection. Since LLM serving systems generally necessitate adherence to service-level objectives (SLOs) of predefined latency requirements [4, 49, 70], predictable and controllable execution time over the two phases is demanded. However, current GPUs lack deterministic concurrent scheduling controls [8, 25, 33, 46], forcing users to carefully orchestrate kernel co-location to achieve reliable overlap [54, 67, 68]. Although modern GPUs provide resource partitioning through Nvidia's multi-process service (MPS) [44], these mechanisms still require precise kernel management to ensure effective resource sharing while meeting SLO requirements.

2.2.3 Kernel-Level Analysis. To characterize LLM serving performance and resource utilization, we analyze the execution time (Figure 2a), hardware utilization (Figure 2b,c) and theoretical waste caused by wave quantization (Table 1). The profiling methodology is detailed in §4.1, with CPU overheads excluded. While MLP operations achieve up to 92% compute utilization, complete Transformer layers sustain only 70%-76% due to compounding inefficiencies. For shorter sequences, severe wave quantization in GEMMs creates substantial under-utilization, as evidenced by the O-projection's measured 49% and 70% utilization, respectively. This result closely agrees with the theoretical bound of 59% and 79% ($100 - idle\%$ in Table 1). Furthermore, PagedAttention's [35] irregular memory access patterns degrade attention kernel performance [19], which is particularly problematic as attention dominates 34% of execution time for longer sequences. Together, the effects of wave quantization and memory access inefficiencies create persistent performance gaps between theoretical peak and achieved throughput. These gaps remain substantial regardless of input sequence length, inherently constraining overall system efficiency.

Takeway 1. *GPUs remain underutilized even during compute-intensive prefill. While co-locating prefill and decode saturate the resources, predictable control to orchestrate the two phases is demanded.*

2.3 Biased Utilization-Latency Tradeoff

2.3.1 Chunked Prefill Workflow. As shown in Figure 3a, chunked prefill [4] achieves low and stable TPOT by leveraging a fixed token budget to concatenate the prefill and decode tokens into a *hybrid batch*, executing in a lockstep fashion (❷). Given a chunk size of cs, the hybrid batch is first filled with ds active decode requests first, and allocates the remaining $cs - ds$ tokens to the prefill sequences. Sequences sl exceeding this residual capacity are split into chunks, leaving residual tokens processed in subsequent iterations. Therefore, the prefill completion requires $N = \lceil sl/(cs - ds) \rceil$ iterations. This forces $N \cdot (N + 1)/2$ times KV cache reloads as each new chunk *must* reprocess previous chunks' cached states.



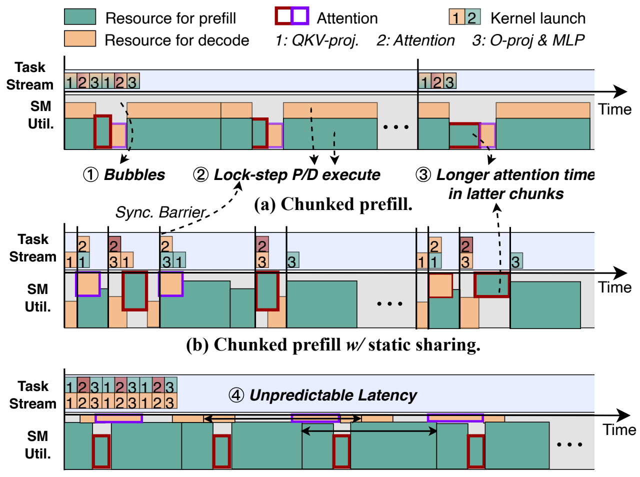
(a) Execution time. (CPU overhead excluded)

(b) Compute utilization. (c) Memory bandwidth utilization.

Figure 2: Breakdown of execution time and hardware utilization in the prefill phase of Llama-3.1-8B model on Nvidia A100 GPU. The aggregate utilization per layer remains below peak sustainable capacity (red line).

Due to the lock-step execution of the hybrid batch, a smaller chunk size effectively decreases TPOT at the cost of increased TTFT and degraded system throughput [4], while larger chunks exhibit the opposite effect. Previous works [4, 71] recognize these inherent tradeoffs and propose tuning chunk size based on workload's prefill-to-decode (P:D) time ratio through manual tuning [4, 50, 71] or automatic searching [3, 10, 55].

2.3.2 Sub-optimal Hardware Utilization. Despite the throughput-latency tradeoff has been extensively studied, we highlight that such tradeoff is biased, and the resulting performance degradation remains overlooked. **First**, as discussed in §2.2.3, chunked prefill typically uses suboptimal chunk sizes below GPU-saturating levels to prioritize low latency. This produces severe wave quantization effects [15], creating GPU bubbles (Figure 3a-❶). **Second**, redundant KV cache reloads required for long sequences significantly prolong

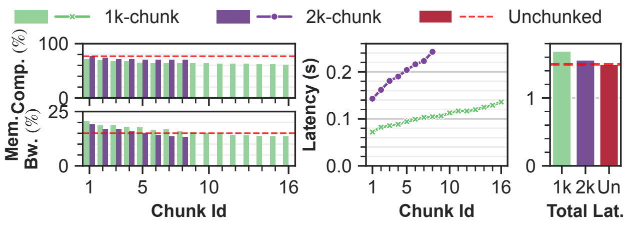


(a) Chunked prefill.

(b) Chunked prefill *w/* static sharing.

(c) Biased sharing.

Figure 3: Kernel-level computation workflow of existing systems featuring chunked prefill-based approach (a,b) and sub-optimal GPU sharing (c).



(a) Hardware utilization. (b) Per-chunk and total latency.

Figure 4: Per-chunk GPU utilization and latency across varying chunk sizes *w/o* hybrid batch on 16k-token chunked prefill (CPU overhead excluded).

attention computation time (❸), further reducing GPU utilization. **Third**, these factors collectively inflate TTFT, triggering a cascading congestion effect in which queued requests stall while awaiting prefill completion, degrading overall system throughput.

Figure 4 systematically quantify the performance degradation of chunked prefill of a 16k-token sequence prefill even *without* hybrid batching. For 1k chunk size, a progressive 10% drop in compute efficiency (from 71% to 61%) across successive chunks is witnessed in Figure 4a, which falls substantially below the 77% achievable peak. This under-utilization stems from redundant KV cache reloading in chunked attention, which also causes the final chunk's processing time to 1.9× than that of the initial chunk. Consequently, per-chunk latency scales linearly by chunk counts and increases total prefill latency by 1.13× compared to unchunked execution. While a larger chunk size of 2k partially mitigates utilization drops from -18% to -7%, the average per-chunk latency increases by 1.86×, significantly diminishing the TPOT improvements that motivated chunked prefill. This fundamental tension between maintaining high hardware utilization and minimizing TPOT creates an intractable optimization in chunked prefill.

Takeaway-2 *Chunked prefill inherently falls short in achieving high utilization due to 1) small chunk size that disproportionately trades throughput for latency. 2) Redundant memory access. 3) cascading congestion on pending requests.*

2.4 Existing Optimizations

Canonical chunked prefill [4] sequentially computes decode and prefill attention, leading to severe bubbles. While PodAttention [50] mitigates this inefficiency by fusing the two attention kernels, such operator-specific optimization fails to address the inherent limitations of chunked prefill (§2.3.2). Nanoflow [71] optimizes the hybrid batch through finer-grained nano batches (Figure 3b). The system establishes a fixed-size processing pipeline through careful configuration of the kernel's *grid size* and inter-kernel synchronization. Using CUDA streams [18], the technique achieves partial overlap among compute-intensive, memory-bound, and network communication operators across distinct nano batches. However, similar to chunked prefill, the attention duration grows by successive chunks. This gap eventually eliminates the chances of overlapping due to kernel dependencies and fixed pipelines.

Apart from chunked prefill-based approaches, MuxServe [22] (Figure 3c) decouples prefill and decode phases into separate processes, leveraging MPS [44] with manually configured, fixed SM quotas to enable spatial sharing. However, this coarse-grained partitioning relinquishes execution control to the device after launching the model, which consists of hundreds of kernels. This leads to unpredictable latency despite GPU saturation (§2.2.2). Furthermore, the inflexible SM allocation cannot adapt to dynamic workloads, compromising serving quality. Thus, neither solution fully resolves the utilization-latency tradeoff under varying serving demands.

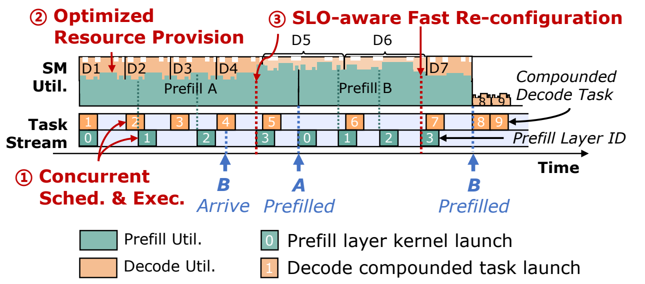

② Optimized Resource Provision
③ SLO-aware Fast Re-configuration
SM Util.
Task Stream
Prefill A
Prefill B
Compounded Decode Task
Prefill Layer ID
Time
① Concurrent Sched. & Exec.
B Arrive
A Prefilled
B Prefilled
Prefill Util.
Decode Util.
Prefill layer kernel launch
Decode compounded task launch

Figure 5: Dynamic spatial-temporal orchestration of concurrently executed prefill and decode tasks.

2.5 Opportunities and Challenges

To address the inefficiencies discussed above, a concurrent yet predictable execution mechanism is urgently demanded to maximize GPU utilization in LLM serving while adhering to latency constraints, as illustrated in Figure 5.

Concurrent Schedule and Execution. Despite the concurrent execution of prefill and decode improving performance [22], maintaining latency targets requires fine-grained request monitoring and kernel scheduling based on real-time progress and system status. A layer-level SLO-aware scheduler (§3.3) may effectively address these challenges, but must be designed to with minimal overhead and synchronization costs (§3.5).

Optimized Resource Provision. As discussed in §2.2.3, neither short nor long input sequences saturate GPU resources during the prefill phase. By carefully partitioning available SM resources (§3.4), memory-bound decode kernels may efficiently co-execute with compute-intensive prefill kernels. However, the multi-dimensional nature of input parameters and dynamic inter-kernel contention create a complex optimization space. This necessitates precise execution time modeling (§3.2) across SM allocations and efficient exploration algorithms (§3.3) to identify optimal configurations.

SLO-aware Fast Re-configuration. In response to runtime status dynamics, including new incoming requests or potential SLO violations, resource allocation between prefill and decode phases

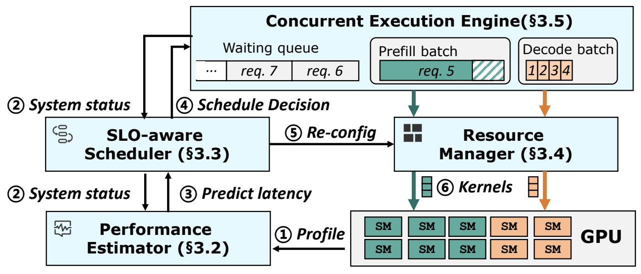


Figure 6: Workflow of `Bullet`. (Numbers: dataflow order.)

must be re-configured. These frequent adjustments demand near zero-overhead reconfiguration to maintain efficiency. Although MPS [44] offers context APIs to adjust different SM quotas, we explore an even more lightweight approach (§3.4) to maintain multiple pre-configured execution states for instantaneous switching between optimized resource partitions.

Figure 5 illustrates dynamic resource scheduling in a 4-layer LLM serving scenario. Initially, ❶ prefill and decode run concurrently with ❷ resource provision at a balanced point. When request B arrives, request A's prefill is ❸ prioritized by allocating more resources while avoiding suspending active decodes. Once request B begins its prefill phase, the scheduler ❸ dynamically adjusts quotas again to maintain the optimal configuration that satisfies both SLO requirements. Overall, this approach enables fine-grained control of task execution for dynamic workloads.

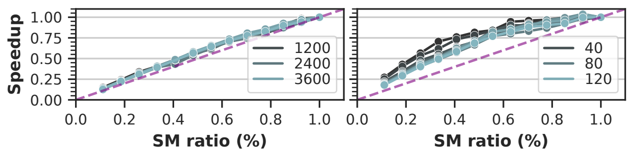
3 Design

3.1 Overview

Figure 6 illustrates Bullet's workflow for concurrent prefill and decode execution with optimal resource provision dynamically. Bullet comprises four key components around the scheduling, resource partition and execution: performance estimator, SLO-aware task scheduler, computational resource manager and concurrent execution engine. The *performance estimator* (§3.2) ❶ first builds an analytical model for the served LLM to facilitate profiling and collects profile data to improve the prediction accuracy. The enhanced model precisely estimates task latency across different configurations of co-executing prefill and decode batch sizes and allocated computing resources. During runtime, *SLO-aware task scheduler* (§3.3) acts as a critical coordinator to enable GPU sharing with prefill and decode tasks via spatial-temporal scheduling. At every layer-wise scheduling cycle, the scheduler ❷ proactively retrieves system status from *concurrent execution engine* (§3.5), monitoring request progress and evaluating potential SLO violations through a combination of *performance estimator* and historical statistics. The scheduler rapidly searches the optimal resource configuration and ❸ scheduling decision that maximizes throughput while adhering to SLO requirements. *Computational resource manager* is ❹ triggered for lightning resource re-configuration when necessary. Finally, the prefill and decode kernels are ❺ launched concurrently on the provisioned SMs. Bullet dynamically balancing the competing demands of TTFT and TPOT maintains high utilization.

3.2 Performance Estimator

3.2.1 Performance Modeling. Accurate performance estimation is critical for SLO-aware scheduling (§3.3) to determine optimal resource configurations efficiently. While predicting LLM operators' latency can be achieved using the roofline model [3, 65], non-trivial effects arise when considering SMs are partitioned, wave quantization effects and inter-kernel contention. These factors make purely analytical approaches insufficient. Additionally, the exponential combinations of potential kernel overlaps and input dimensions render exhaustive profiling [54, 67] impractical. We analytically model these effects, then leverage offline profiling to refine the model with minimizes required data points while maintaining accuracy.



(a) Prefill sequence lengths. (b) Decode batch sizes. (CL: 2048.)

Figure 7: Speedup of using partial SMs normalized to using full GPU. (Purple dotted line: linear scale.)

Consider a kernel with c_i flops of operations and access b_i bytes of memory, we first estimate the execution time utilizing the roofline model [3, 65]. When the kernel is restricted to use m_i SMs on an M SMs GPU, the available compute resource and memory bandwidth non-linear scales to the ratio of m_i/M, as illustrated in Figure 7. Compute intensive prefill kernels are susceptible to resource restrictions, while the memory-bound decode scales super-linear with the SMs. To model such behavior, we suppose the peak compute flops C and bandwidth B with decaying factors d_c and d_b, respectively. These factors are obtained via the subsequent profiling. Similarly, as co-located prefill and decode kernels compete for compute and bandwidth resources, we denote the decaying factor as p_c and p_d, respectively. Finally, by introducing the ideal SM ratio s_i of wave quantization in Equation 1, we formulate the estimation as:

$$t_i = \frac{t_i^m}{(1-s_i)} = \max\left(\frac{c_i}{C}\cdot\frac{M}{m_i d_c p_c}, \frac{b_i}{B}\cdot\frac{M}{m_i d_b p_b}\right)\cdot(1-s_i)^{-1} \quad (2)$$

3.2.2 Offline Profiling. While the model characterizes isolated kernel performance, analyzing inter-kernel contention between co-located prefill and decode tasks presents combinatorial complexity. Therefore, we focus on Transformer layer latency rather than individual kernels, namely, $T_m = \sum t_i$. Through profiling, we realize the parameters in Equation 2 to capture layer-level behavior, enabling predictable latency analysis without explicitly tracking individual kernel interactions during concurrent prefill and decode execution. The profiling records latency varied by five parameters, prefill sequence length (*sl*), decode batch size (*bs*), average context length in decode batch (*cl*), number of SMs for prefill (*pm*), and number of SMs for decode (*dm*). The process begins with isolated kernel measurements to establish d_c and d_b, followed by co-located execution to fit contention parameters p_c and p_b across prefill and decode. Since the parameter space is highly complex, we sample data at steps, enumerating *sl*, *bs*, *cl* and SM counts at a step of 1024, 8, 1024 and 6 while keeping $bs \cdot cl$ within KV cache capacity. This sampling effectively reduces the number of trails to approximately 12k while preserving coverage. Finally, the augmented model interpolates unsampled configurations using derived scaling factors p_c and p_b. The profiling process completes within approximately two hours, with model accuracy validated on real-world workloads in §4.5.2.

3.3 SLO-aware Task Scheduler

3.3.1 Scheduling Workflow. The scheduler operates independently for prefill and decode tasks within their respective *concurrent execution engines* (§3.5). At each scheduling cycle, the scheduler retrieves global system status from the shared *metadata buffer* (§3.5.2) to inform the decisions. Figure 8a demonstrates a workflow on a 12-SM

GPU executing a 4-layer LLM, with layers in the prefill phase denoted as P1-P4 and a decode step marked as D1-D7. For prefill tasks, the scheduler launches a fixed number of layers (1 in the example) and synchronizes GPU to make subsequent scheduling decisions. This enables real-time perception (§3.3.2) of prefill progress and rapid adaptation to system fluctuations. In contrast, decode tasks are dispatched as a single compounded operation via CUDA Graph [18] to minimize kernel launch overhead, with the scheduler invoked before each iteration. Each scheduling decision estimates latency through *performance estimator*, thereby perceiving the request progress. Based on the predicted latency, §3.3.3 describes the primary scheduling objective, which is to meet stringent SLO requirements while maximizing throughput. If the estimated latency violates SLOs with current resource configurations, the scheduler employs sophisticated rules (Algorithm 1, 2) to rapidly determine the optimal resource partition configurations. Finally, the scheduler triggers the *computational resource manager* to switch resources configuration, ensuring compliance with latency guarantees.

3.3.2 Execution Progress Tracking. Scheduling decisions depend on the progress of the executing kernels, and the all requests' latency. The system state at scheduling iteration k is defined as $S_k = (P_k, D_k, R_k)$, where P_k represents the progress of the running prefill operation, D_k denotes the aggregated TPOT in the decode batch, and R_k specifies the resource allocation configuration. The prefill state $P_k = (l_k, n_p, p_k, q_i, w_k)$, in which l_k indicates the number of executed layers, n_p is the total number of tokens in the prefill batch, *i.e.*, the sum of sequence lengths of batched requests. The p_k reflects the elapsed time since prefill starts, q_k is the queuing time for the i-th request, and w_k is the number of pending requests in the waiting queue at step k. Decode batch status $D_k = (n_d, o_i, d_i)$ tracks the batch size n_d, output token count o_i and decode latency d_i for request i, with TPOT calculated as d_i/o_i. Resource configuration $R_k = (u_k, v_k)$ consists of the number of SMs allocated to prefill and decode, respectively. This comprehensive state representation enables the scheduler to jointly optimize for both immediate task progress and long-term SLO compliance while accounting for resource utilization efficiency.

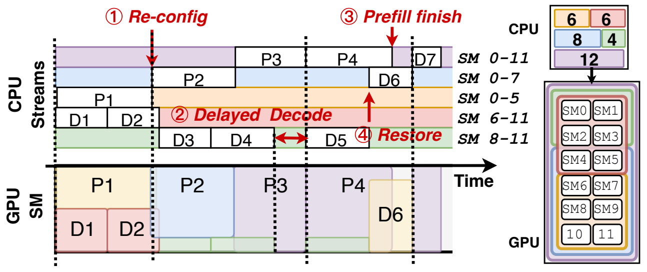
Algorithm 1 outlines the scheduling process. First, the algorithm monitors prefill request progress and updates the system states (line 2-6). Also, decode tasks update per-request latency metrics o_i/d_i to compute real-time TPOT values (line 8-10). This dual-level estimation captures both immediate system status and holistic request progression. Notably, the estimator refines its accuracy by correlating deviations between predicted latency and observed execution times across scheduling cycles, using the inter-cycle duration $t_{k+1} - t_k$ as dynamic feedback to adjust future predictions.

Algorithm 1: SLO-aware Scheduling

```
    Input: System state: $S_k$,   SLO: $\Gamma_{ttft}, \Gamma_{tpot}$
    Output: Updated state: $S_{k+1}$
1   Function MyFunction(Schedule):
        /* Track request status                              */
2       rem_time ← EstimPrefillTime($S_k$)×(total_layers−$l_k$)
3       for $i$ ← 0 to $n_p$ do
4           $ttft[i]$ ← rem_time + $q_i$ // Estimate TTFT
5       for $i$ ← 0 to $w_k$ do
6           $q_i$ ← $l_i$ + rem_time // Estimate queuing delay
7       sort(pending_reqeusts) // Reorder pending reqs.
8       for $i$ ← 0 to $n_d$ do
9           $o_i$ ← $t_k$, $d_i$ ← $d_i$ + 1
10          $tpot[i]$ ← $o_i/d_i$ // Update TPOT
        /* Find optimal SM configuration                     */
11      if $P90(ttft) \le \Gamma_{ttft}$ and $P90(tpot) \le \Gamma_{tpot}$ then
12          return ReduceDecodeSM($S_k, ttft, tpot$)
13      else if $P90(ttft) > \Gamma_{ttft}$ and $P90(tpot) > \Gamma_{tpot}$ then
14          return SetBalancedSM($S_k, ttft, tpot$)
15      else if $P90(tpot) > \Gamma_{tpot} \le W_{max}$ then
16          return ReducePrefillSM($S_k, ttft, tpot$)
17      else
18          return ReduceDecodeSM($S_k, ttft, tpot$)
```

3.3.3 Scheduling Space and Objectives. The primary scheduling objective is to balance between TTFT and TPOT and prioritize prefill if decode latency is not violated, which helps optimize throughput. Based on the real-time progress tracking, the scheduler explores the optimization space to determine the optimal resource allocation R_k and the number of concurrent decode steps that co-executes with prefill operations. When the estimated TTFT and TPOT both violate the requirements (lines 10-11), the scheduler strategically reduces the number of SMs allocated to decode tasks.This reallocation accelerates prefill execution while maintaining acceptable decode performance downgrade. However, if satisfying both constraints becomes infeasible due to a high request rate (line 12-17), the system dynamically balances resource allocation to prevent excessive latency from occurring in both stages.

For instance, when TTFT violation is detected (line 14, Figure 8a-❶), the scheduler reduces decode SMs to accelerate prefill. The search algorithm iteratively decreases decode SMs from the current v_k to a minimum value v_{min}, while estimating delays each step to prevent future violations. The algorithm returns when a new decode SM count is found. However, if setting v_{min} SMs still violates TTFT while adhering to TPOT, the system prioritizes prefill by



(a) Kernel launches to SM-masked streams for spatial-temporal sharing. (P1-P4: prefill layers, D1-D7: decode iterations.)

(b) Pre-configured stream-to-SM mappings.

Figure 8: Fine-grained schedule with SLO-awareness and instant resource re-configuration.

temporarily pausing decode (❷) until the next cycle (Figure 8a). This indicates significant potential to improve prefill performance by temporarily borrowing resources from decode tasks. The process repeats until find an optimal configuration, balancing prefill speed while avoiding excessive decode suspension.

3.4 Computational Resource Manager

3.4.1 Computation Unit Partitioning. Current approaches for dynamic SM partition inevitably involve intensive kernel code modification [71] to achieve precise management. Alternatively, several non-intrusive methods [9, 22, 67] use MPS context [44] to share GPU among different processes, which are tagged with maximum SM limitations. However, simply specifying the context offers no guarantee on which SMs would be allocated for concurrent kernels, potentially aggravating contention [33, 46]. To gain better control of SM allocation, we employ an SM mask technique [7, 8] upon the MPS. The `libsmctrl_set_stream_mask` API enforces kernel execution on designated SMs by configuring available SM constraints for the task queue, namely CUDA stream [18]. Specifically, the API supports allocation granularity of 2 SMs and support for arbitrary SM group configurations (Figure 8b). Therefore, by submitting tasks to the masked streams, resource partition can be realized at a fine-grained level while maintaining the benefits of multi-process GPU sharing through MPS. Despite targeting CUDA, the design can also be extended to other hardware supporting concurrent task queues and resource partition [41].

3.4.2 Instant Layer-wise Re-configuration. To achieve agile resource re-configuration layer-wise (§3.3), `Bullet` creates multiple CUDA streams with different SM partitions beforehand. During execution, the task scheduler instantly selects the pre-configured stream for kernel launches, achieving efficient spatial-temporal resource sharing through rapid stream switching. Moreover, such flexible configuration enables non-strictly isolated configurations to allow for kernels sharing intra-SM resources.

As depicted in Figure 8, a set of streams is masked with various SM counts in advance. Initially, first prefill layer and two decode iterations onto strictly partitioned streams with 6 SMs each. Then, the scheduler estimates a potential prefill timeout and ❶ instructs the second layer to launch on a stream configured with 8 SMs. In contrast, the decode task is launched with fewer SMs. In the next scheduling cycle, decode execution is ❷ temporarily delayed to accelerate TTFT without violating TPOT constraints, delaying decode to the next cycle. At the sixth decode step, the scheduler ❸ anticipates prefill completion and ❹ allocates 8 SMs to decode, enabling controlled SM sharing with prefill. This intentional resource contention facilitates a smooth transition between co-running prefill/decode and decode-only, trading marginal throughput reduction for significantly improved hardware utilization.

3.5 Concurrent Execution Engine

3.5.1 Execution Workflow. Figure 9 shows the concurrent execution engine for prefill (green boxed) and decode (orange), isolated in separate processes and operating independently with decentralized schedulers. We exemplify a scenario in which the prefill engine computes the last groups of layers for request 5 while the decode engine executes requests 1-4. Taking the control and dataflow for

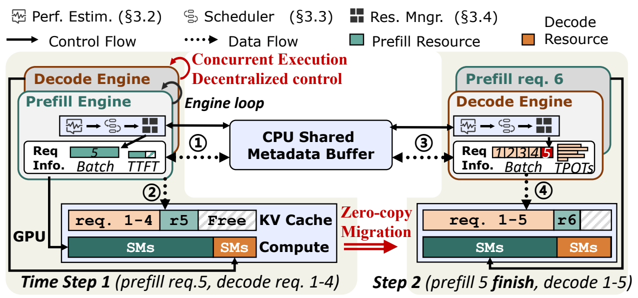


Figure 9: The processing framework of `Bullet`, demonstrating an example of concurrent prefill and decode execution, with metadata transition between the engines.

request 5 as an example, the execution begins with the scheduling workflow depicted in §3.3.1, illustrated as the purple box in the engine. Initially, in time step 1, system status from the decode engine is ❶ retrieved from the CPU metadata buffer, enabling the prefill engine to perceive holistic information for scheduling, including decode batch size, real-time progress and latencies. Conversely, the request metadata, such as KV cache indices [69] and progress, are sent to the buffer to share with decode engine. After that, the SLO-aware scheduler determines the optimal resource configuration and ❷ launches kernels from the layer group to the designated SM partition. Note that the decode engine is performing the same process and executing decoding concurrently with the prefill. When the prefill kernels finish, migrating request 5 to the decode engine is copy-free, since the KV cache and index table are shared by the two engines with *GPU memory pool* (§3.5.2). In time step 2, when the decode engine finishes an iteration and starts to schedule, the engine ❸ fetches system status from the buffer, similar to the process in ❶. Therefore, request 5's metadata is received as a component of the status by the decode engine, allowing for the scheduler to batch request 5 with existing decoding requests. Finally, the scheduler follows the same workflow for ❹ resource determination and kernel launch, with prefill engine computing on request 6 concurrently.

`Bullet`'s scheduler works independently while proactively communicating through the shared metadata buffer, unblocking both CPU and GPU execution. This decentralized architecture allows kernels to submit to GPU while eliminate the need for frequent synchronization compared with a centralized controller.

3.5.2 Memory Space Sharing. To facilitate inter-engine communication, `Bullet` designs a shared CPU buffer and unified GPU memory pool. The CPU buffer is implemented as OS-managed shared memory, storing system status and request metadata. During runtime initialization, the prefill engine creates the shared memory, and the decode engine maps to the region. This allows engines to perform direct read/write operations on the buffer with low-latency. `Bullet` also defines several control bits in the buffer to indicate the data availability. For GPU memory management, `Bullet` employs a dedicated initialization process that allocates model weights and KV cache [35] prior to engine launch. The GPU memory region is exported as a handle using CUDA's `cudaIpcGetMemHandle` API [18], which is then transmitted to the engines via ZeroMQ [66]. This process is lightweight and causes no adverse effects on the memory allocation, as documented. Upon receiving the handle, each engine maps the memory using cudaIpcOpenMemHandle, gaining direct access to model weights and KV cache within its own memory space without performance penalty, which is similar to OS-managed shared memory.

4 Experimental Evaluation

4.1 Methodology

Platforms. We conduct experiments on a server equipped with Nvidia A100-PCIe-80GB GPUs, featuring 108 SMs and 2TB/s of HBM bandwidth, with GPU clocks locked to the maximum stable frequency of 1410MHz. The CPU is Intel Xeon Gold 6150 CPU, 256GB of DRAM, and runs Debian Linux 5.10.0-33. The version of the Nvidia driver is 535.183.06 and CUDA Toolkit is 12.2.

Implementation. We implement Bullet on top of SGLang [69] v0.3.0 and PyTorch 2.4.0 with 4100 lines of Python code, and integrate a modified SM partitioning technique [7, 8] to optimize GPU resource allocation within the serving engine. The prefill and decode engines are implemented as SGLang's workers launched in separate processes, with MPS [44] enabled for spatial sharing. The end-to-end processing pipeline begins with SGLang's built-in tokenizer handling incoming requests. Tokenized requests are routed to the prefill engine with metadata pushed to the shared CPU buffer. Then the standard execution workflow (§3.5) starts, with generated tokens passing through SGLang's native detokenizer, and finally delivers to the front-end server.

Evaluated Schemes. We compare Bullet against several state-of-the-arts implementations of chunked prefill-based optimization: vLLM [35] V1 (chunk size: 1024), SGLang [69] v0.3.0 (chunk sizes: 1024/2048), and Nanoflow [71] (chunk size: 1024).

Models and Datasets. We evaluate the system using the Llama-3.1-8B model [23] across three representative datasets, which are widely used in previous works [4, 35, 71], and the arrival distribution of requests also follows a Poisson process. Figure 10 details the cumulative distribution function (CDF) of the datasets.The ShareGPT [56] contains real-world conversational data, Azure-Code [49] is production code completions by Azure, and arXiv-Summary [13, 14] is long-context summarization.

Metrics We report TTFT, TPOT and throughput aligned with previous works [4, 35, 71]. For SLO compliance (goodput) measurement, we define as prefill and decode latency satisfy both the constraints, as listed in Table 2. Due to the variation in sequence length, we use *normalized input latency* (norm. TTFT) established in previous work [60]. We collect hardware metrics using Nvidia Nsight Systems 2023.2.3 [2], a low-overhead profiling tool.

Table 2: Workload latency requirements.

	ShareGPT	Azure-Code	arXiv-Sum
norm. TTFT (ms)	3.0	1.5	1.5
TPOT (ms)	150	200	175

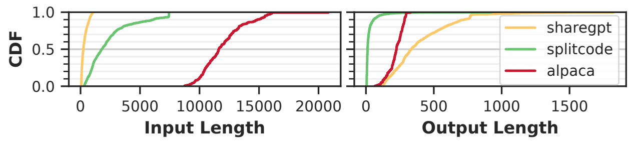


Figure 10: Workload input/output length CDF.

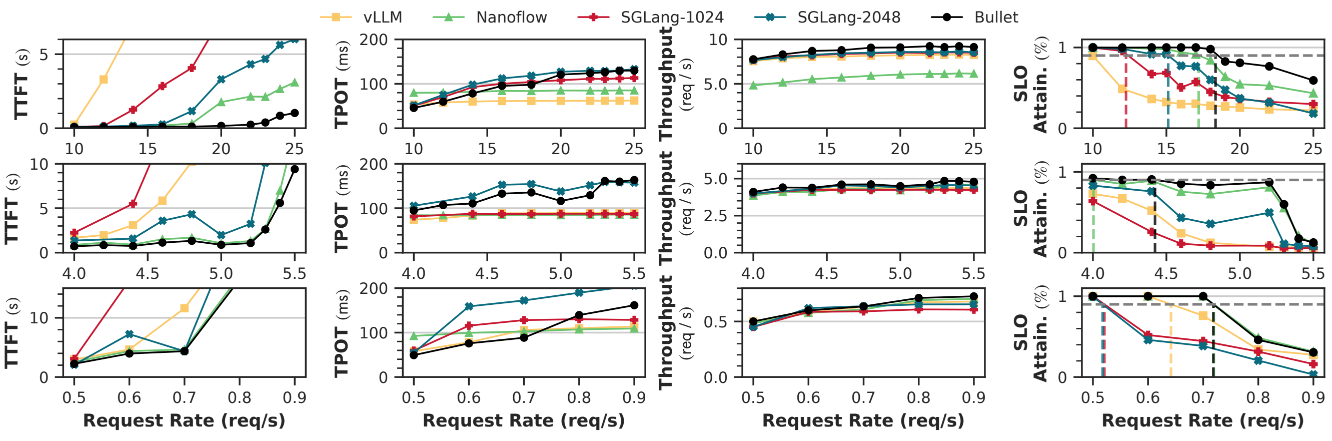
4.2 End-to-End Performance

We evaluate Bullet against four baselines across three production workloads with varying request rates (Figure 11). Bullet demonstrates consistent throughput improvements, achieving 1.09× average and up to 1.20× higher throughput compared to SGLang-1024. Overall, Bullet maintains superior prefill latency enhancement (13.5×) with acceptable TPOT (0.94×) in average through dynamic SM allocation, translating to 1.86× end-to-end speedup compared with SGLang-1024. In contrast, all the chunked prefill-based systems exhibit unacceptable TTFT degradation even in a low request rate. For example, on the ShareGPT workload, with request rate of 20req/s, the prefill latency ranges from 1.8s (Nanoflow) to 14.9s (vLLM). The key reason is that chunk prefill limits the system's capability to process the prefill phase (§2.3.2) and creates a cascading congestion for pending requests. By eliminating these bottlenecks through intelligent SM allocation and adaptive scheduling, Bullet achieves mean TTFT of 0.16s and P90 tail latency of 0.31s, which is 54.9× and 78.5× better to SGLang-1024. The lower prefill latency remarkably translates to larger decode batches that boost overall throughput and SLO compliance (1.49×).

Bullet effectively balances the throughput-latency tradeoff that skewed in chunked prefill systems. While SGLang-2048 demonstrates a 3.20× TTFT improvement and 1.18× higher throughput than SGLang-1024, the 2048 chunk suffers from 0.78× worse TPOT in average, confirming the imbalanced tradeoff inherent in chunked prefill. Bullet breaks this paradigm by simultaneously achieving both lower TTFT (4.2×) and better TPOT (1.20×) than SGLang-2048 through exploiting GPU utilization via concurrent prefill-decode execution. While Nanoflow achieves 2.37× longer TTFT and 0.86× shorter TPOT than Bullet via static kernel overlapping pipeline, it still suffers from the inherent limitations of chunked prefill that inflate TTFT tail latency. This results in 5.2% lower SLO compliance and 20.0% lower throughput compared to Bullet.

4.3 Performance Breakdown

4.3.1 Dynamic SM Partition. We break down the effectiveness of dynamic resource provisioning in Figure 12 by timeline with the Azure-Code workload (5.0 req/s). The top row in Figure 12a demonstrates the number of SMs provisioned for the prefill phase, with each bar showing the SM count and duration. On request rate bursts (spikes in Figure 12-bottom), Bullet adaptively sets the number of prefill SMs to use full GPU, and may temporarily delay decode requests. This enables the rapid response for queuing requests, avoiding excessive waiting time. After processing the pending requests, Bullet quickly reconfigurates the resources to a balance point for both phases. Instead, chunked prefill-based systems are not able to optimize such a scenario, since redundant KV cache access inflates the prefill speed. Moreover, decode batch shares the token budget with prefill tokens (Figure 12b), enforcing more iterations to complete a prefill, which further degrades performance. These factors jointly result in 4.17× longer queuing delay for SGLang-2048. Bullet exempts from the token budget (Figure



(a) ShareGPT dataset.

(b) Azure-Code dataset.

(c) arXiv-Summary dataset.

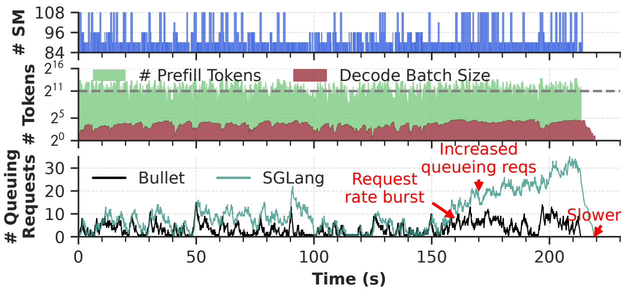
Figure 11: Average latency, throughput and SLO attainment rate of different LLM serving systems. Bullet achieves the highest throughput and SLO compliance in all workloads, attributed to remarkably decreased TTFT with modest TPOT increments.

12-middle) and orchestrates the two phases with fine-grained resource control to saturate GPU, significantly decreasing both TTFT and TPOT by 9.15× and 1.33×, respectively.

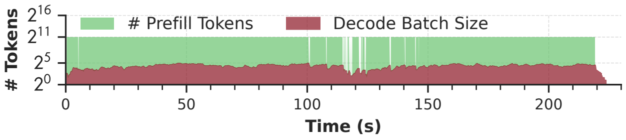
4.3.2 Overheads. We quantify the CPU overhead of Bullet's key components in Table 3 Sending and receiving metadata only necessitates serialization and de-serialization of Python objects, therefore achieving mean latency of 0.21*ms*. The performance prediction module incurs merely 10.2μs overhead by simply invoking the analytical model. For GPU resource management, pre-allocated CUDA streams with configured SM partitions enable instantaneous reconfiguration, maintaining negligible runtime overhead. These design choices jointly ensure Bullet's control plane adds minimal latency while enabling fine-grained resource optimization.

Table 3: Overheads in Bullet.

	Mean	Std.	P90	P99
Metadata Send/Recv (ms)	0.21	0.44	0.89	1.54
Performance Predict (μs)	10.2	5.1	24.5	25.8
Resource Re-config (μs)	4.1	0.79	4.2	5.9



(a) SM allocation for prefill changes dynamically according to system load (top). Concurrently processing tokens/batch size in Bullet (middle). Number of requests waiting for prefill (bottom). Bullet adaptively avoids request congestion on burst.



(b) SGLang-2048's hybrid batch status. Decode requests occupy chunk budget and degrades prefill efficiency, incurring severe queuing time.

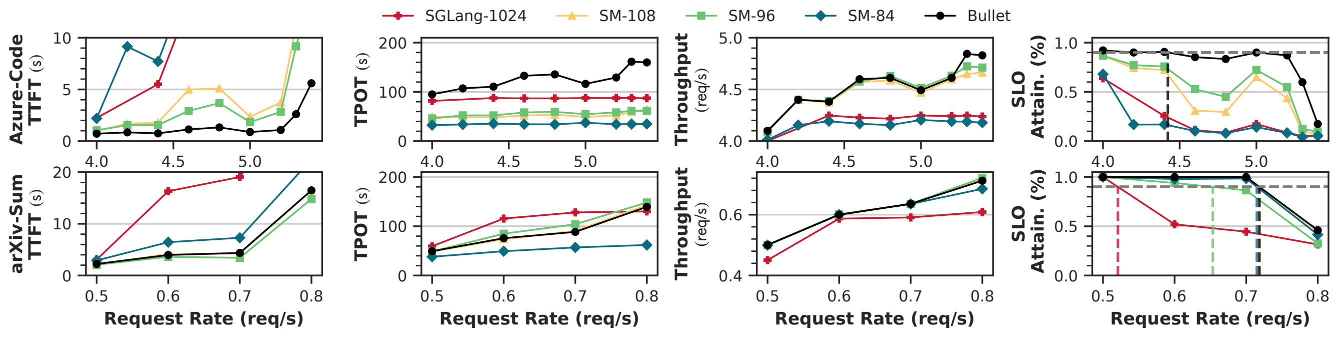
Figure 12: System serving status of the Azure-Code workload in timeline view (request rate: 5.0 req/s).

4.4 Sensitivity Studies

To study the effect of dynamic resource provisioning, we run the workloads under fixed SM configurations for prefill and allow decode to use all SMs. The results are reported in Figure 13. The SM-108 configuration (no partitioning) demonstrates severe imbalance in the Azure-Code workload. While achieving low TPOT, SM-108 suffers 1.20× higher TTFT in average and 1.19× worse P90 tail latency compared to Bullet, ultimately reducing throughput and SLO attainment by 13%. Smaller static partitions, like 84 SMs, prove even more problematic, exacerbating latency imbalance with 1.78× worse TTFT even than chunked prefill baselines and reducing throughput by 5.9%. Therefore, there is no optimal fixed SM allocation, as smaller partitions improve TPOT but degrade TTFT and introduce tail latency violations, and vice versa. These results confirm Bullet's dynamic approach, which adaptively orchestrates resources to simultaneously maintain balanced latency targets and maximize throughput.

4.5 Ablation Studies

4.5.1 Effect of different components. Figure 14 analyzes the contributions of different components in Bullet. Three comparison systems isolate part of the design. Naive: concurrent execution without resource provisioning or scheduling. *w*/Partition: adds resource provision only. *w*/Scheduler: Incorporates only request reordering and delayed decode. The Naive design exhibits the expected latency imbalance (§4.4), high TPOT and low TTFT attributed to unpartitioned resource contention. The *w*/Partition variant improves TPOT



(a) Avg. input token latency. (b) Avg output token latency. (c) Request throughput. (d) P90 goodput.

Figure 13: Sensitivity study of setting different fixed number of SMs for prefill. The static configurations result in latency-throughput imbalance. More SMs reduce TTFT but hurt TPOT and goodput. Dynamic SM tuning optimizes both metrics.

for Azure-Code but suffers unacceptable TTFT degradation from its inability to reorder pending requests. Conversely, *w*/Scheduler maintains comparable latency while reducing Azure-Code TPOT through contention alleviation. Only with the complete Bullet design, incorporating both partitioning and scheduling, achieve balanced latency across all workloads.

4.5.2 Performance Estimator Accuracy. We evaluate the precision of the profile-augmented analytical model in Figure 15. On average, the model achieves 88% accuracy in SLO compliance estimation, *i.e.*, classifying whether requests will meet or violate latency targets. While the mean relative error between predicted and actual durations is 19.1%, this absolute error is inconsequential for scheduling decisions. The scheduler only requires reliable identification of SLO violations, which our model delivers through its high compliance prediction accuracy. This confirms that the model is fully sufficient for guiding Bullet's real-time scheduling.

5 Related Works

LLM Serving Systems. Optimizations for LLM serving have been extensively explored from multiple system layers. At the kernel level, current systems heavily rely on high-performance operator libraries, especially hardware vendor-optimized GEMM implementations [6, 45, 48] and specialized attention kernels [19, 50, 52] . While achieving near-peak performance for individual operators, the dynamic computational demands of end-to-end LLM serving create system-level utilization gaps. Bullet builds upon these kernel optimizations while introducing adaptive phase coordination to address these gaps. Scheduling innovations like continuous batching [64] and chunked prefill [4] improve batching efficiency but maintain lockstep prefill and decode phase execution that underutilizes hardware. In contrast, Bullet fully parallelizes the phases to maximize GPU saturation. This approach also enhances emerging intra-device parallelism techniques [22, 71], further validating its effectiveness. Recent works disaggregate prefill and decode phases on separate replicas [20, 24, 31, 49, 51, 70] to eliminate the interference, but require costly KV cache migration and workload-specific GPU allocation, limiting the implementation in high bandwidth interconnect clusters. As such, Bullet is orthogonal to these works and offers complementary advantages in the transitional mixed instances required by disaggregated systems when switching between prefill and decode modes.

Concurrent Kernel Execution. Co-executing kernels with complementary resource demands have been proven to improve GPU utilization and performance [9, 39, 54, 67]. Spatial-temporal multiplexing techniques [9, 11, 12, 21, 26, 36, 54] typically classify kernels by computational characteristics and rely on CUDA streams [18] or MPS [44] for kernel submission. However, these approaches treat the hardware scheduler as a blackbox, providing limited control [25, 33, 46] over actual execution. Alternative solutions employ static kernel fusion [27, 34, 37, 58, 59] and explore optimization space with compiler [32, 40, 62, 72] to control the execution precisely. Nonetheless, the kernel-level modifications lack adaptability for online dynamic workloads. While some systems use precise kernel timing and resource provisioning [28, 67, 68], they focus on traditional DNN workloads and fail to address LLM-specific requirements, including prefill-decode phase coordination, KV cache management and iteration-level batching. Bullet distinguishes itself from previous works by fine-grained orchestration of LLM's phases and layer-level precise resource management.

Task Scheduling. Concurrent kernel execution requires effective scheduling to enhance performance. Several works [9, 39]

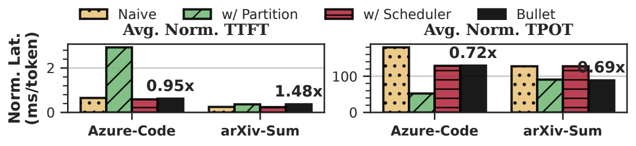


Figure 14: Ablation study on the contribution of the components. Relying solely on a component fails to optimize both metrics across various workloads.

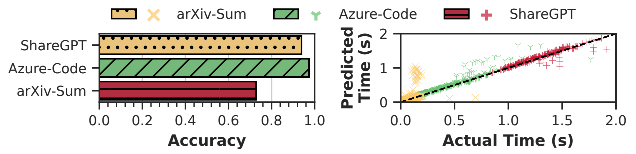


Figure 15: Accuracy of SLO latency compliance classification (left) and actual vs. predicted duration distribution (right).

schedule concurrent kernels based on inter-kernel dependencies, while others [27, 53, 54] classify tasks as real-time or best-effort and apply prioritized scheduling. Latency model-based solutions [28, 42, 61, 67, 68] schedule kernels within latency constraints but rely on excessive offline profiling, which is unsuitable for the dynamic workloads in LLM serving. Although estimating standalone LLM inference is well-established [3, 65], the complex interactions between concurrent prefill and decode phases remain unstudied. Bullet addresses this gap by developing a profile-augmented analytical model for performance estimation and refining kernel scheduling mechanisms for LLM serving.

6 Conclusion

This paper proposes Bullet to optimize LLM serving through precise GPU resource provisioning. By carefully orchestrating spatial-temporal execution between prefill and decode phases, Bullet effectively saturates wasted GPU resources. On runtime, Bullet employs real-time task progress estimation to optimize scheduling decisions and resource allocation, enabling precise and fine-grained control over layer-level resource quotas. Experimental results demonstrate that Bullet effectively boosts performance and guarantees latency constraints.

References

[1] 2022. LlamaIndex. https://github.com/jerryjliu/llama_index.

[2] 2025. Nsight Systems Documentation. https://docs.nvidia.com/nsight-systems/UserGuide/index.html.

[3] Amey Agrawal, Nitin Kedia, Jayashree Mohan, et al. 2024. VIDUR: A Large-Scale Simulation Framework for LLM Inference. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*, Phillip B. Gibbons, Gennady Pekhimenko, and Christopher De Sa (Eds.). mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2024/hash/b74a8de47d2b3c928360e0a011f48351-Abstract-Conference.html

[4] Amey Agrawal, Ashish Panwar, Jayashree Mohan, et al. 2023. SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills. *CoRR* abs/2308.16369 (2023). doi:10.48550/ARXIV.2308.16369 arXiv:2308.16369

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

[6] AMD. 2025. hipBLAS, the Basic Linear Algebra Subroutine library. https://github.com/ROCmSoftwarePlatform/hipBLAS.

[7] Joshua Bakita and James H. Anderson. 2023. Hardware Compute Partitioning on NVIDIA GPUs. In *29th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2023, San Antonio, TX, USA, May 9-12, 2023.* IEEE, 54–66. doi:10.1109/RTAS58335.2023.00012

[8] Joshua Bakita and James H. Anderson. 2024. Demystifying NVIDIA GPU Internals to Enable Reliable GPU Management. In *30th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2024, Hong Kong, May 13-16, 2024.* IEEE, 294–305. doi:10.1109/RTAS61025.2024.00031

[9] Chao Chen, Chris Porter, and Santosh Pande. 2022. CASE: a compiler-assisted SchEduling framework for multi-GPU systems. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 17–31. doi:10.1145/3503221.3508423

[10] Ke Cheng, Zhi Wang, Wen Hu, Tiannuo Yang, Jianguo Li, and Sheng Zhang. [n. d.]. SCOOT: SLO-Oriented Performance Tuning for LLM Inference Engines. In *THE WEB CONFERENCE 2025.*

[11] Seungbeom Choi, Sunho Lee, Yeonjae Kim, et al. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 199–216. https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[12] Marcus Chow, Ali Jahanshahi, and Daniel Wong. 2023. KRISP: Enabling Kernel-wise RIght-sizing for Spatial Partitioned GPU Inference Servers. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023.* IEEE, 624–637. doi:10.1109/HPCA56546.2023.10071121

[13] Arman Cohan, Franck Dernoncourt, Doo Soon Kim, et al. 2018. A Discourse-Aware Attention Model for Abstractive Summarization of Long Documents. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, Marilyn A. Walker, Heng Ji, and Amanda Stent (Eds.). Association for Computational Linguistics, 615–621. doi:10.18653/V1/N18-2097

[14] Arman Cohan, Franck Dernoncourt, Doo Soon Kim, et al. 2018. A Discourse-Aware Attention Model for Abstractive Summarization of Long Documents. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers).* Association for Computational Linguistics, New Orleans, Louisiana, 615–621. doi:10.18653/v1/N18-2097

[15] NVIDIA Corporation. [n. d.]. NVIDIA Deep Learning Performance. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html.

[16] Nvidia Corporation. 2021. Nvidia A100 tensor core GPU architecture. https://resources.nvidia.com/en-us-genomics-ep/ampere-architecture-white-paper.

[17] NVIDIA Corporation. 2022. NVIDIA H100 Tensor Core GPU Architecture Overview. https://resources.nvidia.com/en-us-tensor-core.

[18] NVIDIA Corporation. 2025. CUDA Runtime API Documentation. https://docs.nvidia.com/cuda/cuda-runtime-api.

[19] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135 [cs.LG] https://arxiv.org/abs/2205.14135

[20] DeepSeek-AI, Aixin Liu, Bei Feng, et al. 2024. DeepSeek-V3 Technical Report. *CoRR* abs/2412.19437 (2024). doi:10.48550/ARXIV.2412.19437 arXiv:2412.19437

[21] Aditya Dhakal, Sameer G. Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: controlled spatial sharing of GPUs for a scalable inference platform. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 492–506. doi:10.1145/3419111.3421284

[22] Jiangfei Duan, Runyu Lu, Haojie Duanmu, et al. 2024. MuxServe: Flexible Spatial-Temporal Multiplexing for Multiple LLM Serving. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024.* OpenReview.net. https://openreview.net/forum?id=R0SoZvqXyQ

[23] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, et al. 2024. The Llama 3 Herd of Models. *CoRR* abs/2407.21783 (2024). doi:10.48550/ARXIV.2407.21783 arXiv:2407.21783

[24] Bin Gao, Zhuomin He, Puru Sharma, et al. 2024. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, Saurabh Bagchi and Yiying Zhang (Eds.). USENIX Association, 111–126. https://www.usenix.org/conference/atc24/presentation/gao-bin-cost

[25] Guin Gilman, Samuel S. Ogden, Tian Guo, and Robert J. Walls. 2020. Demystifying the Placement Policies of the NVIDIA GPU Thread Block Scheduler for Concurrent Kernels. *SIGMETRICS Perform. Evaluation Rev.* 48, 3 (2020), 81–88. doi:10.1145/3453953.3453972

[26] Jianfeng Gu, Yichao Zhu, Puxuan Wang, et al. 2023. FaST-GShare: Enabling Efficient Spatio-Temporal GPU Sharing in Serverless Computing for Deep Learning Inference. In *Proceedings of the 52nd International Conference on Parallel Processing, ICPP 2023, Salt Lake City, UT, USA, August 7-10, 2023.* ACM, 635–644. doi:10.1145/3605573.3605638

[27] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[28] Ziyi Han, Ruiting Zhou, Chengzhong Xu, Yifan Zeng, and Renli Zhang. 2024. InSS: An Intelligent Scheduling Orchestrator for Multi-GPU Inference With Spatio-Temporal Sharing. *IEEE Transactions on Parallel and Distributed Systems* 35, 10 (2024), 1735–1748. doi:10.1109/TPDS.2024.3430063

[29] Sirui Hong, Mingchen Zhuge, Jonathan Chen, et al. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024.* OpenReview.net. https://openreview.net/forum?id=VtmBAGCN7o

[30] Sirui Hong, Mingchen Zhuge, Jonathan Chen, et al. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024.* OpenReview.net. https://openreview.net/forum?id=VtmBAGCN7o

[31] Cunchen Hu, Heyang Huang, Liangliang Xu, et al. 2024. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. *CoRR* abs/2401.11181 (2024). doi:10.48550/ARXIV.2401.11181 arXiv:2401.11181

[32] Changho Hwang, KyoungSoo Park, Ran Shu, et al. 2023. ARK: GPU-driven Code Execution for Distributed Deep Learning. In *20th USENIX Symposium on*

Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 87–101. https://www.usenix.org/conference/nsdi23/presentation/hwang

[33] Paras Jain, Xiangxi Mo, Ajay Jain, et al. 2019. Dynamic Space-Time Scheduling for GPU Inference. *CoRR* abs/1901.00041 (2019). arXiv:1901.00041 http://arxiv.org/abs/1901.00041

[34] Abhinav Jangda, Saeed Maleki, Maryam Mehri Dehnavi, Madan Musuvathi, and Olli Saarikivi. 2024. A Framework for Fine-Grained Synchronization of Dependent GPU Kernels. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2024, Edinburgh, United Kingdom, March 2-6, 2024*, Tobias Grosser, Christophe Dubach, Michel Steuwer, Jingling Xue, Guilherme Ottoni, and ernando Magno Quintão Pereira (Eds.). IEEE, 93–105. doi:10.1109/CGO57630.2024.10444873

[35] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, et al. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 611–626. doi:10.1145/3600006.3613165

[36] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. 2020. Nimble: Lightweight and Parallel GPU Task Scheduling for Deep Learning. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/5f0ad4db43d8723d18169b2e4817a160-Abstract.html

[37] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2022. Automatic Horizontal Fusion for GPU Kernels. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022*, Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman (Eds.). IEEE, 14–27. doi:10.1109/CGO53902.2022.9741270

[38] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, et al. 2024. Parrot: Efficient Serving of LLM-based Applications with Semantic Variable. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 929–945. https://www.usenix.org/conference/osdi24/presentation/lin-chaofan

[39] Zejia Lin, Zewei Mo, Xuanteng Huang, et al. 2023. KeSCo: Compiler-based Kernel Scheduling for Multi-task GPU Applications. In *41st IEEE International Conference on Computer Design, ICCD 2023, Washington, DC, USA, November 6-8, 2023.* IEEE, 247–254. doi:10.1109/ICCD58817.2023.00046

[40] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, et al. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020.* USENIX Association, 881–897. https://www.usenix.org/conference/osdi20/presentation/ma

[41] AMD MxGPU. 2016. Hardware-based virtualization.

[42] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency Model Serving with Software-defined GPU Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, Jason hFlinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 595–610. doi:10.1145/3600006.3613163

[43] NVIDIA. 2023. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM.

[44] NVIDIA. 2025. Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html.

[45] NVIDIA Corporation. 2025. *cuBLAS: NVIDIA CUDA Basic Linear Algebra Subroutines.* https://developer.nvidia.com/cublas

[46] Ignacio Sanudo Olmedo, Nicola Capodieci, Jorge Luis Martinez, et al. 2020. Dissecting the CUDA scheduling hierarchy: a Performance and Predictability Perspective. In *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2020, Sydney, Australia, April 21-24, 2020.* IEEE, 213–225. doi:10.1109/RTAS48715.2020.000-5

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

[48] Muhammad Osama, Duane Merrill, Cris Cecka, et al. 2023. 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 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*, Maryam Mehri Dehnavi, Milind Kulkarni, and Sriram Krishnamoorthy (Eds.). ACM, 429–431. doi:10.1145/3572848.3577479

[49] Pratyush Patel, Esha Choukse, Chaojie Zhang, et al. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024.* IEEE, 118–132. doi:10.1109/ISCA59077.2024.00019

[50] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, et al. 2025. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, Lieven Eeckhout, Georgios Smaragdakis, Kaitai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach (Eds.). ACM, 1133–1150. doi:10.1145/3669940.3707256

[51] Ruoyu Qin, Zheming Li, Weiran He, et al. 2025. Mooncake: Trading More Storage for Less Computation - A KVCache-centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies, FAST 2025, Santa Clara, CA, February 25-27, 2025*, Haryadi S. Gunawi and Vasily Tarasov (Eds.). USENIX Association, 155–170. https://www.usenix.org/conference/fast25/presentation/qin

[52] Rya Sanovar, Srikant Bharadwaj, Renée St. Amant, Victor Rühle, and Saravan Rajmohan. 2024. Lean Attention: Hardware-Aware Scalable Attention Mechanism for the Decode-Phase of Transformers. *CoRR* abs/2405.10480 (2024). doi:10.48550/ARXIV.2405.10480 arXiv:2405.10480

[53] Sudipta Saha Shubha, Haiying Shen, and Anand P. Iyer. 2024. USHER: Holistic Interference Avoidance for Resource Optimized ML Inference. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 947–964. https://www.usenix.org/conference/osdi24/presentation/shubha

[54] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024.* ACM, 1075–1092. doi:10.1145/3627703.3629578

[55] DeepSpeed Team. 2025. DeepSpeed-MII: Enabling Low-Latency, High-Throughput Inference. https://github.com/deepspeedai/DeepSpeed-MII.

[56] ShareGPT Teams. 2023. Share your wildest ChatGPT conversations with one click. https://sharegpt.com/

[57] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[58] Guibin Wang, Yisong Lin, and Wei Yi. 2010. Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications, GreenCom 2010, & Int'l Conference on Cyber, Physical and Social Computing, CPSCom 2010, Hangzhou, China, December 18-20, 2010*, Peidong Zhu, Lizhe Wang, Feng Xia, Huajun Chen, Ian McLoughlin, Shiao-Li Tsao, Mitsuhisa Sato, Sun-Ki Chai, and Irwin King (Eds.). IEEE Computer Society, 344–350. doi:10.1109/GREENCOM-CPSCOM.2010.102

[59] Zhenning Wang, Jun Yang, Rami G. Melhem, et al. 2016. Simultaneous Multikernel: Fine-Grained Sharing of GPUs. *IEEE Comput. Archit. Lett.* 15, 2 (2016), 113–116. doi:10.1109/LCA.2015.2477405

[60] Bingyang Wu, Shengyu Liu, Yinmin Zhong, et al. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*, Emmett Witchel, Christopher J. Rossbach, Andrea C. Arpaci-Dusseau, and Kimberly Keeton (Eds.). ACM, 640–654. doi:10.1145/3694715.3695948

[61] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 69–85. https://www.usenix.org/conference/nsdi23/presentation/wu

[62] Kan Wu, Zejia Lin, Mengyue Xi, et al. 2025. GoPTX: Fine-grained GPU Kernel Fusion by PTX-level Instruction Flow Weaving. In *Proceedings of the 62nd ACM/IEEE Design Automation Conference, DAC 2025, Moscone Center West, San Francisco, CA, USA, June 22, 2025.* ACM, TBD.

[63] An Yang, Baosong Yang, Binyuan Hui, et al. 2024. Qwen2 Technical Report. *CoRR* abs/2407.10671 (2024). doi:10.48550/ARXIV.2407.10671 arXiv:2407.10671

[64] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, et al. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[65] Zhihang Yuan, Yuzhang Shang, Yang Zhou, et al. 2024. LLM Inference Unveiled: Survey and Roofline Model Insights. *CoRR* abs/2402.16363 (2024). doi:10.48550/ARXIV.2402.16363 arXiv:2402.16363

[66] ZeroMQ authors. [n. d.]. ZeroMQ: An open-source universal messaging library. https://zeromq.org/

[67] Shulai Zhang, Quan Chen, Weihao Cui, Han Zhao, Chunyu Xue, Zhen Zheng, Wei Lin, and Minyi Guo. 2025. Improving GPU Sharing Performance through Adaptive Bubbleless Spatial-Temporal Sharing. In *Proceedings of the Twentieth European Conference on Computer Systems.* 573–588.

[68] Yongkang Zhang, Haoxuan Yu, Chenxia Han, et al. 2025. SGDRC: Software-Defined Dynamic Resource Control for Concurrent DNN Inference on NVIDIA

GPUs. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2025, Las Vegas, NV, USA, March 1-5, 2025*. ACM, 267–281. doi:10.1145/3710848.3710863

[69] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, et al. 2023. Efficiently Programming Large Language Models using SGLang. *CoRR* abs/2312.07104 (2023). doi:10.48550/ARXIV.2312.07104 arXiv:2312.07104

[70] Yinmin Zhong, Shengyu Liu, Junda Chen, et al. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

[71] Kan Zhu, Yilong Zhao, Liangyu Zhao, et al. 2024. NanoFlow: Towards Optimal Large Language Model Serving Throughput. *CoRR* abs/2408.12757 (2024). doi:10.48550/ARXIV.2408.12757 arXiv:2408.12757

[72] Donglin Zhuang, Zhen Zheng, Haojun Xia, Xiafei Qiu, Junjie Bai, Wei Lin, and Shuaiwen Leon Song. 2024. {MonoNN}: Enabling a New Monolithic Optimization Space for Neural Network Inference Tasks on Modern {GPU-Centric} Architectures. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 989–1005.