



# Nova: Real-Time Agentic Vision-Language Model Serving with Adaptive Cross-Stage Parallelization

Yuhang Xu<sup>†</sup>, Shengzhong Liu<sup>\*†</sup>, Dong Zhang<sup>§</sup>, Bingheng Yan<sup>§</sup>, Fan Wu<sup>†</sup>, Guihai Chen<sup>†</sup>

<sup>†</sup>Shanghai Jiao Tong University, <sup>§</sup>Inspur Data Co.,Ltd.

Email: {xuyuhangtmx, shengzhong}@sjtu.edu.cn, {zhangdong, yanbh}@inspur.com, {fwu, gchen}@cs.sjtu.edu.cn

**Abstract**—This paper presents Nova, a real-time scheduling framework for serving agentic vision-language models (VLMs) on a single GPU with balanced per-request latency and overall request process throughput. Our design begins by enabling effective pipelining across vision encode, LLM prefill, and LLM decode stages of VLMs, by exploiting their heterogeneous resource demands during execution and incorporating elastic GPU spatial partitioning among stages to maximally utilize the compute and memory resources. Building on this, we introduce a real-time scheduling algorithm that adaptively calibrates resource allocation among stages based on a Pareto-optimal analysis of the latency–throughput trade-off, allowing the system to sustain responsiveness and resource efficiency under dynamic request loads. To further alleviate GPU memory pressure, we design a lightweight weight offloading strategy for vision encoders that preserves inference efficiency with minimized memory overhead. Extensive evaluations on both synthetic and real-world agent workloads demonstrate that Nova consistently outperforms the state-of-the-art baselines, improving the maximum latency by up to 23.3%, while keeping competitive throughput.

## I. INTRODUCTION

Recent advances in vision-language models (VLMs) have enabled a new class of interactive agents that can perceive visual environments and generate language-based actions [1]–[7]. Such VLM agents, capable of interpreting graphical user interfaces (GUIs) and textual user instructions, offer transformative capabilities across application domains, including mobile automation, remote control, and digital assistance [8]–[10]. Different from standard LLM applications like chat and summarization, VLM agents must operate under tight latency constraints to ensure smooth and reactive behaviors. As a result, responsive serving becomes a critical requirement for deploying VLM-based agents in practice.

This paper considers the scenario of serving agentic VLMs on a single GPU, designed for data-sensitive application domains (*e.g.*, banking, healthcare, government) that prohibit offloading GUI images to cloud servers. Unlike cloud clusters consisting of massive GPUs, where different stages of VLM inference, including vision encode, LLM prefill, and autoregressive LLM decode, may be distributed across separate GPUs [11], [12], edge serving executes all stages on a single device. These stages exhibit heterogeneous resource demands, inducing significant challenges to GPU resource management. Besides, the short but bursty nature of agentic workloads

makes it difficult to balance per-request latency with overall system throughput.

Heterogeneous VLM stages pose significant challenges to balancing throughput and latency, as prioritizing the vision encode or LLM prefill can severely slow down token generation, while favoring LLM decode may lead to severe GPU underutilization. To address this issue, techniques like chunked prefill [13] are proposed to divide LLM prefill into multiple encode iterations by encoding one chunk at a time, parallelize prefill with decode in a hybrid batch, and leverage their shared model architecture and weights with homogeneous operators, such that later requests in the waiting queue can receive their first response within shorter time to first token (TTFT). However, they fall short in supporting agentic VLM workloads, as the vision encoder and LLM modules are structurally separated, hindering data-level cross-stage batching. Furthermore, vision encode takes significantly longer than LLM prefill, *e.g.*, over 2 $\times$  in our measurements, making it the new bottleneck beyond batching. Even operator-level optimizations like POD-Attention [14] that do not require shared weights become ineffective for overlapping vision and language stages. Under high request load, this mismatch can cause even decode-prioritized approaches, such as chunked prefill, to suffer from decode starvation and resource underutilization.

These observations motivate us to rethink the system design for agentic VLM serving. To this end, we propose Nova, a pipeline-parallel, resource-aware serving framework designed to address the stage heterogeneity and latency sensitivity of agentic VLM inference. We begin by estimating the feasibility of cross-stage pipelining through multi-kernel co-execution. Unlike operator-level batching depending on shared model weights [13], [14], kernel co-execution allows concurrent execution across structurally independent stages. While prior work, such as NanoFlow [15], enables such parallelism via fine-grained kernel re-implementation and customized scheduling, we adopt a hardware-centric approach by exploring GPU spatial sharing [16]–[18] as a more generalizable and elastic solution. Our key insight is that the primary barrier to parallelization arises from resource contention between heterogeneous kernel launches. To address this, we exploit streaming multiprocessor (SM) partitioning to enable multi-stage co-execution on the GPU. Specifically, vision encode and LLM decode often contend for limited registers and shared memory, which inhibits concurrent kernel launches

<sup>\*</sup>Shengzhong Liu is the corresponding author.

despite their complementary compute and memory demands. By assigning disjoint SM subsets to these inference stages, their contention is alleviated and co-execution efficiency is significantly improved.

Previous GPU sharing efforts have primarily focused on maximizing the overall system throughput [18], [19] but overlooked per-request inference latency. We instead propose an adaptive SM allocation strategy that navigates the trade-off between these two metrics in real time. Unlike static partitioning that fails to accommodate real-time workload variations, we model the resource demands of each stage and dynamically calibrate GPU allocation based on the runtime request load. Specifically, we leverage Pareto-optimal analysis to guide SM partitioning that minimizes end-to-end latency across varying conditions. This is particularly crucial under bursty request patterns, where decode batching and front-stage congestion must be jointly managed. Therefore, SM partitioning not only achieves pipeline parallelism to reduce latency but also exposes resource-level concurrency to improve throughput. The stage-level scheduling aligns better with the modular VLM structures than operator-level techniques and achieves consistent improvement across varying workloads.

We further identify the GPU memory optimization space in vision encoder inference in agentic VLMs, which compete with the pre-allocated KV cache of LLM decode. This contention can lead to cache eviction or recomputation [20], which is problematic in memory-constrained serving scenarios. To address this, we introduce a lightweight weight offloading mechanism that asynchronously swaps vision encoder layers between CPU and GPU memory. This approach alleviates memory pressure with negligible latency overhead, preserving KV cache space and improving system generalizability across hardware configurations.

We implement Nova and evaluate its performance on multiple platforms using both synthetic workloads and real-world workloads. The results show that Nova consistently achieves lower end-to-end latency, outperforming the SOTA baselines by up to 14.6% in average latency and 23.3% in maximum latency, while sustaining no smaller throughput.

Overall, our main contributions are summarized as:

- We identify and address the unique system challenges in serving agentic VLMs on a single GPU, which involve heterogeneous and stage-wise workloads different from standard LLM serving systems;
- We propose a stage-parallel execution framework based on inter-SM GPU sharing, combined with an adaptive SM partitioning strategy guided by Pareto-optimal analysis, to dynamically balance per-request latency and overall throughput under varying workloads;
- We design a lightweight weight offloading mechanism for large vision encoders, mitigate GPU memory pressure with minimal overhead;
- We conduct extensive experiments using both synthetic and real-world datasets on multiple platforms to demonstrate the effectiveness and scalability of our design.

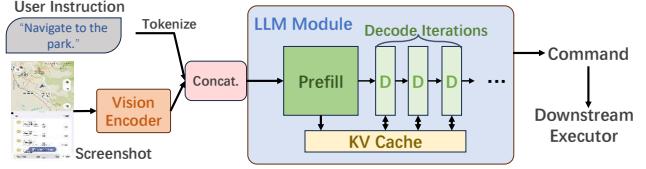


Fig. 1: Framework of VLM-based GUI Agent.

TABLE I: Forward latency of different stages in VLM inference.

Stage	Vision Encode	LLM Prefill	LLM Decode (Single Iter)
Duration	806.8 ms	324.1 ms	28.9 ms

## II. BACKGROUND AND MOTIVATIONS

### A. GUI-based Agentic Vision-Language Model

A Vision-Language Model (VLM)-based agent is a multimodal large language model (LLM) designed to interpret graphical user interface (GUI) screens and generate executable actions based on natural language instructions [2]. Its input typically includes a screenshot of the current GUI state and a user-issued instruction. The output is a structured action command, such as `<click button>`, `<scroll down>`, which an external controller can execute.

The architecture of a VLM-based agent, as shown in Figure 1, consists of two main components: a vision module and an LLM module. The vision module uses a transformer-based encoder (typically CLIP-pretrained Vision Transformer (ViT) [21], [22]) to transform the image into visual embeddings, which are adapted to the LLM’s input format by a lightweight adapter. Next, the LLM module concatenates the visual embeddings with the tokenized instruction and performs autoregressive generation to produce the textual responses (*i.e.*, sequence of text tokens). This process includes a prefill stage, where the multi-modal input is processed to build a key-value (KV) cache as context, and a decode stage, where tokens are generated one at a time sequentially based on the encoded context. The involvement of the KV cache improves computation efficiency by avoiding repetitive computation during sequential decode [20], [23]. As a reference, we report the forward latency of different stages in VLM inference in Table I, testing using cogAgent [2] on an NVIDIA RTX A6000 GPU. The results show that the forward latency of the vision stage is significantly longer than that of the prefill stage. Although a single decode iteration takes relatively little time, the decode stage still accounts for the majority of the total end-to-end latency (typically more than 50%) due to the repeated execution over tens of token generation steps.

**Difference from General-Purpose VLMs:** To ensure the quality of decision making, agentic VLMs differ from general-purpose VLMs in the following aspect: Agentic VLMs typically accept higher-resolution images and employ larger model sizes, as shown in Table II, to capture fine-grained visual details, perform precise pixel-level grounding, and generate reliable action commands for downstream execution. As a result, the vision encoder of agentic VLMs incurs significantly longer image processing time and places a substantially higher workload on limited GPU memory.

TABLE II: Comparisons of model parameters and input sizes of VLMs.

VLM	Type	LLM Size	VE Size	Image Size
InternVL3-8B [24]	General	7 Billions	0.3 Billions	448×448
LLAVA-Next-8B [25]	General	8 Billions	0.5 Billions	672×672
CogAgent-9B [2]	Agentic	9 Billions	4 Billions	1120×1120

TABLE III: Resource utilization profiles of different stages, measured with Nsight Compute. The two values in each cell denote compute throughput (%) and memory throughput (%)<sup>1</sup>.

Stage	Kernel				
	Linear (QKV)	Linear (O)	Linear (UG)	Linear (D)	Attention
Vision	83.5 / 57.0	73.7 / 52.0	87.7 / 59.2	89.9 / 58.7	74.4 / 36.3
Prefill	74.5 / 58.5	87.4 / 73.9	92.3 / 58.9	88.0 / 70.1	73.8 / 34.9
Decode	26.0 / 86.4	26.6 / 88.2	26.6 / 90.9	27.8 / 92.3	17.0 / 53.1

### B. Characterizing Agentic VLM Inference Workload

The inference workload of a VLM-based GUI agent consists of three distinct stages: (1) vision encode, (2) LLM prefill, and (3) LLM decode. Each stage exhibits heterogeneous compute-memory characteristics that influence the hardware utilization and system scheduling strategy.

a) **VLM Computation Stages:** Both the vision encode and LLM prefill stages exhibit compute-intensive behavior. The vision stage processes high-resolution GUI screenshots using a ViT model [22], requiring heavy computation due to high image resolutions and model complexity. The LLM prefill stage corresponds to the initial forward pass of the language encoder with both visual and textual embeddings, involving dense matrix multiplications and attention over a long context, making the stage similarly compute-bound. In contrast, the LLM decode stage is mostly memory-bound. It performs auto-regressive token generation, where the LLM predicts one token at a time using cached key-value (KV) pairs from the prefill stage. Although the input length during decode is much shorter, each iteration still requires loading the full LLM weights from GPU memory to on-chip compute units, *e.g.*, streaming multiprocessors (SMs) on NVIDIA GPUs, as well as accessing KV-cache, especially costly when serving multiple concurrent user requests.

We report the resource utilization of four linear kernels and the attention kernel in the Transformer [26] layers across the three stages using nsight-compute [27], as shown in Table III. It can be observed that the decode stage exhibits significantly higher memory throughput compared to compute throughput, whereas the vision and prefill stages show the opposite trend, indicating highly heterogeneous resource demands across different stages.

b) **Agentic VLM Serving on a Single GPU:** We focus on deploying an agentic VLM serving system on an edge server with a single GPU, motivated by the high bandwidth cost of uploading screenshots and the need to preserve their privacy. Unlike cloud settings where VLM inference stages can

<sup>1</sup>Both compute and memory throughput are aggregated metrics in Nsight Compute. In our setting, they correspond to the tensor pipeline active rate and the DRAM active rate, respectively.

be distributed across GPUs via high-bandwidth interconnects such as NVLink [11], [28], all stages in our setting run on a single GPU. Each request from a user device includes an instruction text (prompt) and a screenshot image. For multi-step tasks, the instruction may also contain historical interactions to maintain continuity across steps.

Our serving system can be formulated as a multi-stage queuing system. The objective is to minimize the end-to-end latency of requests, which consists of the queuing delay and processing time at each stage. Both average and tail latencies affect the quality of service (QoS), with the latter primarily caused by prolonged queuing delays directly tied to system throughput. Since bounding the worst-case latency under unpredictable request arrivals is challenging, we focus on balancing per-request processing time and overall throughput, thereby achieving favorable average and tail latencies.

c) **Scheduling Implications:** The heterogeneous characteristics of VLM inference introduce a fundamental trade-off between system throughput and per-request latency. Prioritizing the vision and prefill stages first, followed by batching decode requests, can significantly improve the overall request processing throughput by maximizing compute utilization and amortizing decode overhead, while reducing time-to-first-token latency (TTFT). However, this strategy may lead to increased or unstable time-between-token latency (TBT) in the decode stage (*i.e.*, interrupted token generation), which can degrade the user-perceived response smoothness. Therefore, achieving an optimal balance between overall throughput and per-request latency is a key challenge in scheduling agentic VLM workloads on a single GPU.

### C. Limitations and Opportunities of LLM Serving Systems

1) **Limitations of Chunked Prefill for VLM Inference:** Chunked prefill [13] is a widely adopted technique in LLM serving systems aimed at reducing TBT and ensuring smooth generations. It works by splitting the user-provided prompt into smaller blocks, encoding them in multiple iterations, and combining the prefill requests and additional decode requests in a single batch, called hybrid batching. By doing so, it implicitly leverages the complementary resource demands of the prefill and decode stages to improve the overall GPU utilization. This strategy is effective in LLMs because prefill and decode stages share the same model weights, while Transformer architectures [26] support batching of mixed operations, including token-wise computation in linear layers and optimized request-wise self-attention (*e.g.*, POD-Attention [14]).

However, chunked prefill faces fundamental limitations when applied to agentic VLM inference. The vision encoder and LLM modules in VLMs correspond to separate model parameters, making it infeasible to combine the vision encode and LLM decode stages in a single batch. Moreover, the vision encode stage runs significantly longer than LLM prefill, further diminishing the benefits of chunked execution.

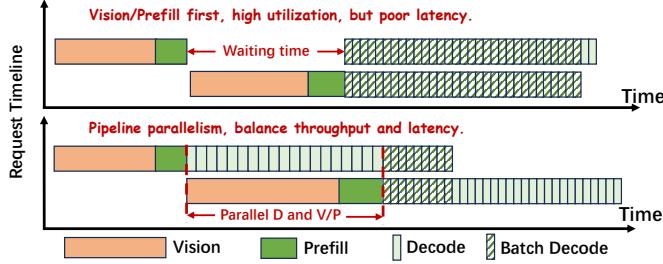


Fig. 2: Timelines of two requests under different scheduling strategies. By leveraging pipeline parallelization between the decode and vision/prefill stages, the system achieves a better balance between GPU throughput and request-level latency.

**Insight 1:** Existing LLM serving frameworks overlook the unique execution patterns of agentic VLMs and struggle to balance their overall throughput and per-request latency.

**2) Opportunities in Cross-Stage Parallelization:** Unlike operator-level parallelization techniques like chunked prefetch, we argue that cross-stage pipeline parallelization is more effective for agentic VLM serving. First, it supports co-running across any inference stages without restrictions on their weight sharing. Second, it frees the scheduler from complex, manual request-level splitting and merging. Third, it better exploits the heterogeneous resource demands of different stages, leading to improved GPU utilization.

With properly designed pipeline parallelization across stages, it is possible to achieve a more favorable trade-off between throughput and latency. As illustrated in Figure 2, compared to traditional scheduling strategies that prioritize vision encode and LLM prefetch stages, overlapping LLM decode with long-running vision or prefetch stages eliminates part of the decode waiting time for the first request, thereby significantly reducing its TBT. Moreover, by intra-device parallelization and maintaining batching opportunities among decode requests, the impact on request processing throughput is reduced.

**Insight 2:** Pipeline parallelization across VLM stages offers a superior balance between throughput and latency compared to rigid prefetch-first or chunked prefetch strategies.

**3) Cross-Stage Parallelization and GPU Sharing:** Efficient stage parallelization relies on fine-grained GPU sharing techniques. Recent works [17], [29] have explored GPU sharing via spatial multiplexing of low-level resources such as streaming multiprocessors (SMs), aiming to improve throughput by co-locating multiple kernels or tasks on a single GPU. However, most of these methods focus solely on maximizing resource utilization but fail to address the latency requirements of agentic VLM serving. Besides, integrating such low-level resource scheduling with the complex request-level coordination in serving systems poses additional challenges.

In agentic VLM serving, cross-stage GPU sharing plays two essential roles. First, it enables high-level request parallelism, helping to reduce overall end-to-end latency. Second, it promotes low-level GPU resource multiplexing by co-locating heterogeneous kernels from different stages. This co-location mitigates the throughput degradation caused by reduced batching opportunities in the decode stage under

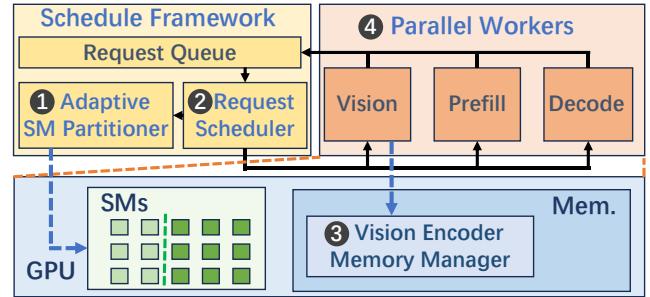


Fig. 3: System Overview of Nova.

pipeline parallelism, as illustrated in Figure 2. Therefore, to achieve low latency across varying workloads, effective GPU sharing must account for both dynamic request fluctuations and the distinct compute and memory characteristics of each stage at the kernel level.

**Insight 3:** Effectively combining GPU sharing techniques with VLM serving scheduling demands fine-grained, latency-aware resource coordination and remains a non-trivial problem.

### III. SYSTEM DESIGN

#### A. System Overview

Figure 3 illustrates the overall architecture of Nova. The system is designed to efficiently serve agentic VLM requests by optimizing GPU resource sharing, execution pipeline, and memory management. The key components and design principles are summarized as follows:

- 1 Adaptive GPU resource partitioner:** Nova employs inter-SM (Streaming Multiprocessor) co-running to enable GPU spatial sharing between VLM stage workers. By co-locating compute-intensive and memory-bound kernels, and adaptively adjusting SM allocations based on workload, Nova improves heterogeneous resource utilization and overall throughput.
- 2 Runtime pipelined request scheduler:** A centralized request scheduler is responsible for tracking the states of active requests. It determines the request priorities and batching strategies among requests to balance per-request latency and overall throughput, particularly under varying runtime workloads. The scheduler runs in a separate control thread and communicates with each model worker via asynchronous message queues.
- 3 Vision encoder memory manager:** To address the high memory demands imposed by the large vision encoders in agentic VLMs, Nova implements a layer-wise parameter swapping strategy. This technique efficiently loads vision encoder layers into GPU memory only when needed, thus leaving a significantly smaller memory footprint.
- 4 Parallel model workers:** The system consists of three dedicated model workers: a vision encode worker, an LLM prefetch worker, and an LLM decode worker. These workers operate concurrently in separate threads. Each worker runs an event loop that continuously processes incoming requests dispatched by the central scheduler.

TABLE IV: Solo/Corun time and kernel-level statistics of two linear operators from the vision encoder and LLM decode.

Kernel	DRAM Band. (GB/s)	SM Throughput (%)	Blocks Per SM	Total Blocks	Duration (ms)	Completion Time (ms)
						Solo Corun
Decode Linear	639.78	33.58	1	448	0.35	699.8 1241.8
Vision Linear	273.14	87.63	1	3000	2.97	588.3 680.6

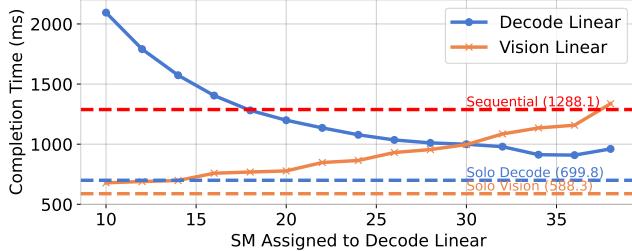


Fig. 4: Kernel completion time varies with different SM allocations. Throughput improves over sequential execution when the decode linear kernel is assigned between 18 and 36 SMs.

Note that the LLM prefill and LLM decode workers share the same model parameters.

### B. Spatial GPU Sharing For Different Stages

To enable parallel execution across VLM stages, we first explore GPU spatial sharing techniques that allow different kernels to co-run efficiently. This section focuses on the underlying SM-level partitioning technique as a foundation for our adaptive scheduling strategy.

1) **Infeasibility with multi-stream inference:** A straightforward approach to exploit spatial sharing across different inference stages is to use CUDA streams, which allow kernels from different streams to execute concurrently on the GPU. However, directly leveraging CUDA streams can lead to severe GPU resource contention and unpredictable latency fluctuations.

To demonstrate this, we select the two up-projection linear kernels—the largest linear kernels—from the vision encode and LLM decode, and execute them concurrently using CUDA streams on RTX A6000. The LLM decode linear kernel is repeated 2000 times, while the vision linear kernel is repeated 200 times. We also use `nvidia-compute` to profile the kernel-level statistics of these two operations. The results are summarized in Table IV. These two kernels are chosen because linear operations constitute the majority of execution time in both the vision encode and the LLM decode forward pass—approximately 75% for LLM decode and 60% for vision encode. The decode batch size is set to 3, as the output sequences of agent-oriented VLMs are typically short, which in turn limits the number of concurrent decode requests.

As shown in Table IV, the two linear operators exhibit different run-time characteristics: the decode linear kernel achieves high memory throughput, utilizing 83% of the A6000’s peak memory bandwidth (768 GB/s), but shows relatively low SM throughput—a metric that reflects compute intensity, primarily indicating tensor core utilization in this case. In contrast, the vision linear kernel is more compute-

intensive. However, running them concurrently results in only marginal overall throughput improvement (1241.8 ms compared to 1288.1 ms for sequential execution), while the decode linear kernel experiences significant slowdowns.

The key reason for this performance limitation is severe **launch resource contention** between the two kernels. As shown in the table, both kernels launch significantly more thread blocks than the number of available SMs on the A6000 (84), which limits inter-SM parallelism. Additionally, each block consumes a substantial amount of per-SM resources (e.g., registers and shared memory), allowing at most one block to be scheduled per SM. This further restricts intra-SM parallelism and reduces overall concurrency.

2) **GPU spatial sharing via SM control:** To overcome this bottleneck, we adopt SM partitioning techniques to limit the number of SMs visible to each kernel, thereby ensuring that they can be launched concurrently. We use `libsmctrl` [30] to control the SM allocation for different kernels, as it offers a more flexible and lightweight solution compared to other SM partitioning methods such as MIG [31] and Green Context [32]. We vary the number of SMs assigned to the two linear kernels and use CUDA events to record their respective completion times, as shown in Figure 4.

By leveraging inter-SM parallelism, we observe throughput improvement: when the number of SMs allocated to the decode linear kernel is between 18 and 36, the completion time of the two co-running kernels is shorter than that of sequential execution. The main reason for this improvement is that memory-bound kernels like decode linear spend a significant portion of their execution time waiting for memory reads to complete. Since DRAM bandwidth and L2 cache—two critical resources for memory access—are shared among SMs, allocating too many SMs to memory-bound kernels leads to contention among thread blocks from the same kernel for memory access, causing frequent instruction stalls.

Using `nvidia-compute`, we observe that over 40% linear kernel execution cycles of LLM decode are stalled waiting for global memory access (as indicated by the Stall Long Scoreboard metric). As a result, its execution latency increases more slowly as the number of assigned SMs decreases, compared to compute-bound kernels like the vision linear. Therefore, co-running these two kernels can improve overall throughput. Besides, we also observe that in corun scenarios, the latency of both kernels is higher than their respective solo runs, even when sufficient SMs are allocated. This is due to contention for memory resources, such as DRAM bandwidth and L2 cache. Thus, inter-SM parallelism improves throughput at the cost of potential latency degradation.

3) **Why not intra-SM parallelization for spatial sharing:** Many prior works [19], [33] leverage intra-SM parallelism to alleviate launch resource contention between GPU kernels. Inspired by these efforts, we apply two techniques to reduce the launch resource consumption of linear kernels: For vision linear, we use CUTLASS [34] to tune tile sizes and pipeline stages, significantly reducing register and shared memory usage with negligible performance degradation. For

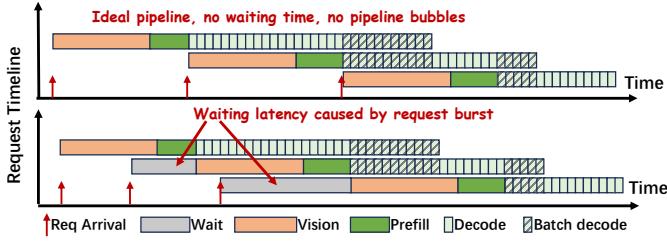


Fig. 5: Waiting latency caused by request burst.

decode linear, which typically operate on small batch sizes, we reimplement them using CUDA without shared memory allocation and configure them to use CUDA cores instead of tensor cores, thereby avoiding compute unit contention with vision linear layers. With these kernel-level optimizations, we observe throughput improvements even slightly exceeding that of inter-SM parallelism when co-running kernels.

However, intra-SM parallelism depends heavily on the GPU’s internal scheduling behavior, which results in unstable kernel completion times due to the lack of control over kernel launch and collocation within the same SM. This instability persists even when combined with inter-SM techniques, making it hard to perform dynamic resource partitioning. Therefore, we argue that intra-SM parallelism is more suitable for throughput-oriented applications, but is less appropriate for our scenario, where a fine-grained balance between throughput and latency is critical. For this reason, we opt to only use the inter-SM parallelization strategy in this paper.

### C. Adaptive SM Partition

Another important question is how to partition SMs among requests in different stages to achieve the best latency-throughput trade-off. We first consider this problem in an idealized scenario, *i.e.*, all requests are scheduled with perfect pipeline parallelism, without any waiting time in the vision encode/LLM prefill stages or pipeline bubbles, as illustrated in the upper example of Figure 5.

1) **End-to-end request latency and SM partition**: Assume the SM partitions for decode-vision and decode-prefill co-running are  $P_v$  and  $P_p$ , respectively. Let the forward duration of LLM decode under these two co-running configurations be  $t_d(P_v)$  and  $t_d(P_p)$ , and let  $t_v(P_v)$  and  $t_p(P_p)$  denote the durations for the vision encode and LLM prefill stages when co-running with decode, respectively. Assuming an token generation length of  $L$ , the expectation of the end-to-end latency  $t_{e2e}$  in this ideal scenario is:

$$\mathbb{E}[t_{e2e}] = t_v(P_v) + t_p(P_p) + (prop_v \cdot t_d(P_v) + prop_p \cdot t_d(P_p)) \cdot L, \quad (1)$$

where  $prop_v$  and  $prop_p$  denote the proportion of co-running time with vision encode or LLM prefill during the entire decode iterations:

$$prop_v = t_v(P_v)/(t_v(P_v) + t_p(P_p)), \quad (2)$$

$$prop_p = t_p(P_p)/(t_v(P_v) + t_p(P_p)).$$

After profiling the forward durations under different SM partitions and the average output length for token generation,

we perform an enumeration search to determine the optimal SM partitions  $P_v$  and  $P_p$  that minimize the end-to-end latency in the ideal scenario, *i.e.*,

$$\begin{aligned} \min_{P_v, P_p} & \mathbb{E}[t_{e2e}], \\ \text{s.t. } & P_v, P_p \in \mathcal{P}_{\text{avai}}, \end{aligned} \quad (3)$$

where  $\mathcal{P}_{\text{avai}}$  denotes the set of valid SM partitions<sup>2</sup>. Assuming the total number of SMs is  $N$ , the time complexity of profiling is  $O(N)$ , and the complexity of searching for the optimal partition configuration is  $O(N^2)$ , which is lightweight given that modern GPUs typically have between tens and low hundreds of SMs.

2) **Adaptive SM partition strategy**: However, static SM partitioning is inadequate for handling dynamic workloads, particularly during request surges. The optimal static configuration typically allocates relatively more SMs to the decode stage, though still fewer than those required to fully accelerate the vision encode or LLM prefill stages, to reduce iterative token generation latency. However, this leads to longer execution times for the vision encode and LLM prefill stages. As explained in the lower example of Figure 5, during burst periods, the waiting time for requests in the vision encode and LLM prefill stages is significantly prolonged, as these stages exhibit no acceleration with request batching.

This motivates us to further analyze the throughput-latency trade-off introduced by SM partitioning. The end-to-end latency under a given SM partition configuration is given in Equation (1), while the corresponding throughput  $Thr$  can be approximated as:

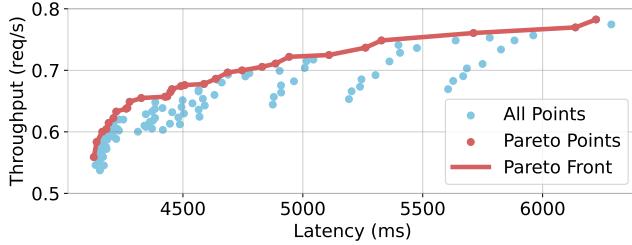
$$Thr = \frac{1}{t_v(P_v) + t_p(P_p)}. \quad (4)$$

This is because the vision encode and LLM prefill are executed sequentially, whereas decode requests can be batched and executed in parallel with prefill and vision. Moreover, the latency of a decode iteration remains relatively stable as batch size increases. For instance, when the batch size increases from 1 to 10, the latency only rises slightly from 28.9 ms to 30.6 ms. In our serving scenarios, the decode batch size typically remains below 5 (reported in Section V-C). Therefore, the ideal throughput is largely unaffected by the decode latency.

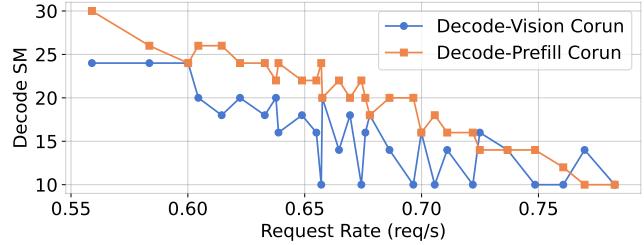
We report the latency and throughput under different SM partition configurations in Figure 6(a)<sup>3</sup>, and plot the Pareto frontier to identify the optimal trade-offs between latency and throughput. Furthermore, we illustrate the optimal SM partition configurations on the Pareto frontier corresponding to different throughput levels—*i.e.*, request rates in serving scenarios—in Figure 6(b). It can be observed that as the request rate increases, the optimal strategy gradually allocates

<sup>2</sup>We only consider assigning SMs with continuous indices, as non-contiguous partitions offer no performance benefit.

<sup>3</sup>We use the average output length on the dataset, which are typically between 40-60 due to structured formatting in agentic VLM responses, justifying the use of the average.



(a) Latency-Throughput trade-off with Pareto front.



(b) SM partition configurations under Pareto front.

Fig. 6: Latency-Throughput trade-off under Pareto front.

fewer SMs to the decode stage, following an approximately linear decreasing trend.

Since the real-time request rate is difficult to measure directly, we instead use the number of *pending requests*, *i.e.*, those currently in the vision encode or LLM prefill stages, as an estimate of the current request rate and system load. Based on this, we design a dynamic strategy that adjusts the SM partitioning according to the number of pending requests: more SMs are allocated to the vision and prefill stages when the number of pending requests increases. The number of SMs assigned to the LLM decode stage is determined as follows:

$$SM_{dec} = \max(SM_{min}, SM_{op} - \alpha \cdot (N_{pend} - 1)), \quad (5)$$

where  $SM_{op}$  (calculated as equation (3)) denotes the default number of SMs allocated to the decode worker when only one request is active in the vision or prefill stage, and  $N_{pend}$  is the number of requests currently in those stages. A minimum threshold  $SM_{min}$  is enforced to bound the TBT and prevent excessive variability in token generation latency.

With this adaptive SM partitioning strategy, the scheduler can better balance overall throughput and per-request latency metrics. When the number of pending requests is high, *i.e.*, under heavy system load, reducing the resources allocated to decode and accelerating the processing of vision encode/LLM prefill stages provides two key benefits:

- It reduces the waiting time for requests in the vision encode and LLM prefill queues, preventing severe accumulation of pending requests.
- It allows earlier LLM decode requests to “wait” for later ones, thereby increasing opportunities for batching decode requests and improving overall system throughput, especially advantageous under high load conditions.

#### D. Pipeline Request Scheduling

We now introduce our request pipeline scheduling algorithm with adaptive inter-SM spatial GPU sharing. The scheduler decides the priorities of different requests, the batch choices among requests, and the number of SMs assigned to model workers of different VLM stages to balance per-request latency and the overall system throughput.

**Priority of Requests:** For requests within the same stage, we adopt a FIFO scheduling policy. Between stages, requests in the LLM decode stage are assigned the highest priority and are scheduled promptly. They can be executed in

---

#### Algorithm 1: Adaptive Request Scheduling Algorithm

---

**Data:** Request queue  $Q$ ; Suspending decode request queue  $Q_d$ ; Suspending vision request queue  $Q_v$ .

```

1 while True do
2   req := get new request from  $Q$ ;
3   Update the number of requests in different stages;
4   Adjust the SM partition according to eq (5);
5   if req is finished then send req to the output queue;
6   if req is prefill then send req to the prefill worker;
7   if req is decode then
8     if there is decode running then  $Q_d$ .append(Req);
9     else send Merge( $Q_d$ , req) to decode worker;
10  end
11 if req is vision then
12   if no running vision and prefill requests then
13     Send req to vision worker;
14   else  $Q_v$ .append(req);
15  end
16 end

```

---

parallel with vision encode or LLM prefill requests to ensure consistent token generation and maximize GPU utilization. In contrast, vision encode and LLM prefill requests do not run in parallel, as both are compute-bound and would contend for GPU resources. We prioritize LLM prefill requests over vision encode requests because of shorter durations. Moreover, completing the LLM prefill stage enables the generation of new LLM decode requests, which can then be batched on the fly to improve overall throughput.

**Batch Decision:** We do not batch LLM prefill or vision encode requests, as both stages are highly compute-bound. Batching them does not improve throughput but instead increases the per-request latency. In contrast, we always batch LLM decode requests, whose batching significantly improves request throughput with minimal impact on per-request latency. Similar to existing LLM serving systems, we adopt an in-flight batching strategy [23]. When a new LLM decode request arrives, if there is an ongoing LLM decode batch, the request waits until the current running batch finishes and then joins the next batch to be dispatched. This wait time is typically short, as LLM decode latency is generally low.

The request scheduling loop of Nova is detailed in Algorithm 1. The request queue contains both newly arriving requests and those already being processed by model workers. The scheduler maintains two waiting queues: decode requests may be suspended to wait for forming an in-flight batch, and vision requests may be suspended to avoid blocking

prefill requests. At the start of each scheduling iteration, the scheduler checks the status of active requests and adjusts the SM partition accordingly (Lines 2 to 4). For requests at different stages, the scheduler decides whether to dispatch them to model workers immediately or suspend them based on their priority and batching decisions (Lines 5 to 15).

We next describe the granularity of SM assignment calibration. Since the request scheduler and model workers run concurrently on separate threads, each SM re-assignment requires synchronization. Possible granularities include forward-pass-level (*i.e.*, adjusting SM allocation per forward pass) and kernel-level (*i.e.*, adjusting before each GPU kernel launch). While finer-grained control allows more precise partitioning, it also incurs higher synchronization overhead. As most kernels are short, frequent reassessments can cause significant CPU blocking and hinder asynchronous GPU execution. Therefore, we adopt a coarse granularity by adjusting SM partitions before each forward pass within the model worker.

**Queueing Behavior:** Under Nova scheduling, an incoming request first waits in the vision worker’s queue. After vision encoding, it is immediately processed by the prefill worker without interruption from other vision-stage requests. The decode worker runs continuously, and a request only waits in its queue during the first iteration to join the in-flight batch—a typically negligible delay. Thus, the queueing delay is dominated by the vision worker’s queue, which can be modeled as a single-capacity queue, since requests in vision encoding and LLM prefill are not parallelized or batched. Assuming that request arrivals follow a Poisson process, the queueing delay can be modeled with an M/G/1 queue, where the service time  $T$  consists of the vision encode and LLM prefill durations, *i.e.*,  $T = t_v + t_p$ . Under an arrival rate  $\lambda$ , the expected queueing delay  $W_q$  is given by:

$$\mathbb{E}[W_q] = \frac{\lambda \mathbb{E}[T^2]}{2 \times (1 - \lambda \mathbb{E}[T])}. \quad (6)$$

### E. Efficient Weight Offloading for Vision Encoder

As discussed in Section II-A, agentic VLMs have significantly larger vision encoder model sizes compared to general-purpose VLMs, causing higher memory pressure on GPU devices. Existing LLM serving systems typically preallocate memory for KV-cache storage [20]; thus, the enlarged model size reduces the available KV-cache capacity, potentially degrading both system throughput and serving quality. To address this challenge, we propose an efficient layer-wise weight offloading and swap-in strategy between GPU memory and CPU memory, leveraging compute and data transfer overlapping to minimize performance overhead.

Mainstream vision encoders typically adopt the ViT [22] architecture, comprising a stack of Transformer layers. Existing serving systems load all  $L$  layers into GPU memory at initialization, leading to significant memory waste. In contrast, Nova only allocates GPU memory for  $K$  layers (called physical layers), where  $2 \leq K \ll L$ , and initializes weights for only the first  $K$  layers. Each physical layer acts as a reusable

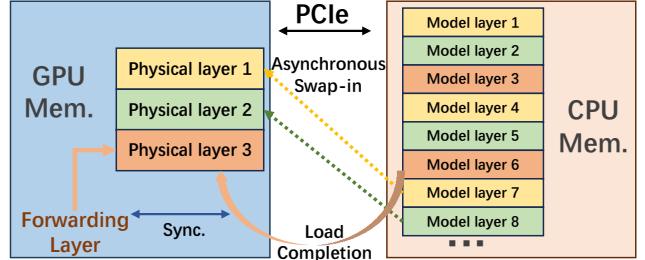


Fig. 7: Weight swap-in between CPU and GPU memory. Physical layers and their corresponding model layers are marked with matching colors. The system is currently forwarding physical layer 3, which corresponds to model layer 6, while model layers 7 and 8 are being asynchronously swapped in.

buffer that holds the weights of multiple logical layers over time, as illustrated in Figure 7. During inference, once a model layer completes the forward pass, we asynchronously swap in the weights for the next logical model layer. Assume the current logical model layer ID stored in the physical layer is  $cur\_layer$ , the next logical layer to be loaded is:

$$nxt\_layer = (cur\_layer + K) \bmod L. \quad (7)$$

Directly adopting weight offloading may incur significant memory transfer overhead. To reduce this, we apply three optimizations: (1) Forward computation and weight swap-in are placed in separate CUDA streams to enable compute–transfer overlap. (2) Pinned (page-locked) CPU memory is used for offloaded weights, allowing zero-copy DMA transfers between host and device. (3) Synchronization between the forward and weight-loading streams is coordinated using CUDA events. Specifically, each physical layer is associated with a CUDA event, and synchronization is performed before the layer is either executed or its weights are swapped in.

**Condition Analysis:** We now analyze the latency introduced by model weight loading and derive the memory bandwidth required for zero-overhead swap-in. Let  $S$  be the total model size,  $T$  be the single forward latency, and  $B$  be the memory bandwidth. During a forward pass, the first  $K$  layers are preloaded into GPU memory. The initial preload occurs at system startup, while subsequent asynchronous swaps are overlapped with computation to hide their latency. When forwarding the  $l$ -th layer ( $l > K$ ), at least  $\frac{l-K}{L} \cdot S$  bytes must have been loaded into GPU memory. The available time window is  $\frac{l-2}{L} \cdot T$ , since swap-in starts only after the first layer’s execution completes. To avoid stalling, the required memory bandwidth  $B$  must satisfy:

$$\frac{l-2}{L} \cdot T \geq \frac{S}{B} \cdot \frac{l-K}{L}. \quad (8)$$

This condition must hold for all  $l \in [K+1, L]$ , yielding that  $B \geq \frac{S}{T} \cdot \frac{L-K}{L-2} \approx \frac{S}{T}$  when  $L \gg K$ . The analysis for consecutive forward passes yields a similar result, indicating that the required bandwidth is independent of  $K$ . In practice, CogAgent’s vision encoder (8 GB) takes 500–800 ms per forward pass on an RTX A6000 or RTX 4090, requiring at most 16 GB/s bandwidth—well below the typical peak bandwidth of PCIe 4.0×16 (32 GB/s), as commonly used

for GPU interconnect. Therefore, the runtime weight swap-in introduces negligible latency overhead.

**Limitations and Potential Extensions:** Weight offloading is less effective for the LLM module due to the shared model weights between prefill and decode stages. Since each decode step is short, there is little opportunity to overlap weight loading with computation. However, prior work has proposed disaggregating prefill and decode [11], [12], allowing offloading to be applied on the prefill side. Additionally, on compute-rich but memory-constrained devices, increasing batch size can extend forward latency and help hide memory transfer overhead.

#### IV. IMPLEMENTATION

We implemented the Nova framework with  $\sim 2K$  lines of Python code, using PyTorch [35] as the backend library. The scheduler and the three parallel model workers run in separate threads and communicate via message queues. Each model worker performs forward passes in its own CUDA stream, with all streams configured to the same priority level. SM partitioning is implemented by adding a Python interface to `libsmctrl` [30], assigning SMs to each stream with contiguous indices to each specific stage. We apply kernel fusion techniques for operators such as RoPE [36] and RMSNorm [37], and use FlashInfer [38] for attention kernels.

#### V. EVALUATION

##### A. Experimental Setup

**1) Hardware Platform:** We mainly conduct experiments on a server equipped with an AMD EPYC 7K62 48-core CPU, an NVIDIA RTX A6000 GPU, and 256 GB of host memory. We also test Nova on an RTX 4090 GPU. The RTX A6000 features 84 SMs and 48 GB of VRAM, whereas the RTX 4090 offers 128 SMs and 24 GB of VRAM.

**2) VLM Model:** We use cogAgent [2], a state-of-the-art agentic VLM. The model consists of a vision encoder, implemented as a 64-layer ViT [22] with 4B parameters, and an LLM module, which is a 40-layer Transformer model with 9B parameters. The model precision is set to `bf16`, requiring approximately 9 GB of GPU memory for vision encoding and 18 GB for LLM prefill/decode. The KV-cache for a single request occupies about 70 MB of GPU memory.

**3) Dataset:** We adopt the Android Instruction Dataset from AndroidLab [39], which contains GUI interaction data collected from 138 tasks across nine Android applications running on virtual devices. Each sample includes a screenshot and a corresponding user instruction. The typical output length ranges from 30 to 80 tokens.

**4) Workloads:** Due to the lack of public agentic VLM serving traces, we simulate request arrivals using a Poisson distribution. By adjusting the parameter  $\lambda$ , we emulate various average request rates. Since the vision and prefill stages of each request take approximately 1.1 s to complete, we limit the maximum average arrival rate to 0.8 requests per second to avoid severe system overload. Additionally, for experiments on the RTX 4090, we use private trace data collected from

real-world serving scenarios to evaluate performance under practical workloads. To simulate varying load conditions, we scale the request arrival intervals of the trace to generate different request rates.

**5) Hyperparameter Setting:** As introduced in Section III-C, we determine  $SM_{op}$  through enumeration based on profiling results under various SM partition configurations (reported in Section V-B). The optimal  $SM_{op}$  values are 24 for decode–vision co-running and 30 for decode–prefill co-running. For both cases, we set  $SM_{min}$  to 12, which ensures that the theoretical maximum TBT remains below 80 ms. We set the  $\alpha$  values for decode–vision and decode–prefill co-running to 4 and 6, respectively<sup>4</sup>, meaning that  $SM_{dec}$  will drop to  $SM_{min}$  when the number of pending requests reaches 4. The values of  $\alpha$  are determined through offline profiling. We evaluated various  $\alpha$  settings between 2 and 8 and observed stable performance across these configurations.

**6) Metrics:** We focus on two main categories of metrics.

- **Latency Metrics:** We primarily consider end-to-end (E2E) request latency, as it directly impacts the responsiveness perceived by downstream executors in agent scenarios. We report both the average and maximum E2E latencies observed in our experiments. Additionally, we report latency breakdown metrics, including time-to-first-token (TTFT) and time-between-tokens (TBT), as commonly adopted in standard LLM serving systems.
- **Throughput Metrics:** While prior works on LLM serving typically measure throughput using the rate of processed or generated tokens [15], this approach is less suitable for VLM agent serving, where the vision stage accounts for a significant portion of the total execution time but does not involve token generation. Therefore, we instead adopt request throughput (*i.e.*, number of processed requests per second) as our throughput metric.

**7) Baselines:** Nova is compared with three approaches:

- **PF-Limit:** Prefill-first scheduling is a widely adopted strategy in existing LLM serving systems such as vLLM [20] and SGLang [40], which prioritizes the vision encode and LLM prefill stages to maximize system throughput. Since naive prefill-first scheduling suffers from poor latency performance due to prolonged waiting times in the LLM decode queue, we introduce a threshold-based modification: LLM decode requests are scheduled when the number of waiting LLM decode requests exceeds a predefined threshold (set to 5 in our experiments).
- **Chunk [13]:** Chunked prefill splits a LLM prefill request into several chunks and batches these chunks with LLM decode requests, thus reducing LLM decode latency. It also achieves better GPU utilization by locating compute-bound LLM prefill and memory-bound LLM decode requests in the same batch [41]. We evaluate various token budget settings and select 128 as the optimal value.
- **Multi-Stream:** Similar to Nova, multi-stream scheduling allows kernels from different stages to execute concurrently

<sup>4</sup>`libsmctrl` only supports SM adjustments in units of 2, as each TPC (thread processing cluster) on the RTX A6000 consists of 2 SMs.

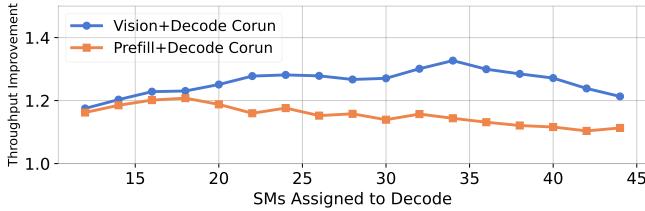


Fig. 8: Corun throughput improvement under different SM allocations.

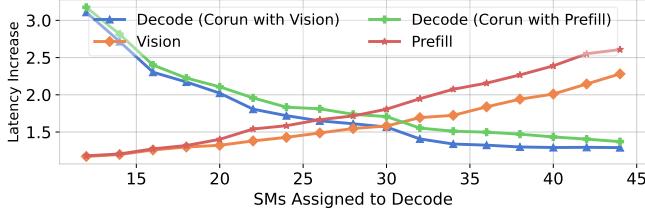


Fig. 9: Corun latency increase under different SM allocations.

on the GPU. However, it relies on CUDA’s default multi-stream scheduling policy for GPU resource partitioning.

#### B. Impact of SM Partition

We first evaluate the performance of co-running requests from different stages under various SM partition configurations. We report both the overall throughput improvements and the corresponding single-pass latency increase introduced by co-running in Figure 8 and Figure 9. The main observations include: First, co-running LLM decode requests with either vision encode or LLM prefill requests improves overall throughput, with moderate variation across different SM allocations. We observe that its co-running with vision encode yields higher throughput improvement, because of the higher memory bandwidth contention between LLM prefill and LLM decode stages. Although the LLM prefill is shorter in duration than vision encode, it involves loading larger model weights, resulting in greater bandwidth usage. Second, the latency increase for both decode–vision and decode–prefill co-running exhibits similar trends. However, the vision stage requires more SMs to maintain low latency, as it exhibits more compute-intensive properties.

#### C. Overall Performance

Next, we compare the end-to-end performance under different request rates. The request trace length is set to 500, spanning tens of minutes and capturing the stationary system behavior under sustainable request rates. We report both end-to-end latency and throughput in Figure 10, and provide a latency breakdown in Figure 11. Nova demonstrates the best performance in both average and maximum end-to-end latencies, critical to downstream command executors. As shown in the latency breakdown, Nova maintains a low TBT through steady token generation by parallelizing LLM decode requests with other request types. While chunked prefill enables parallelization between LLM prefill and decode stages, overlooking vision encode requests can still block LLM decode requests. Multi-Stream relies on internal black-box GPU scheduling, tending to prioritize LLM prefill/vision encode over LLM

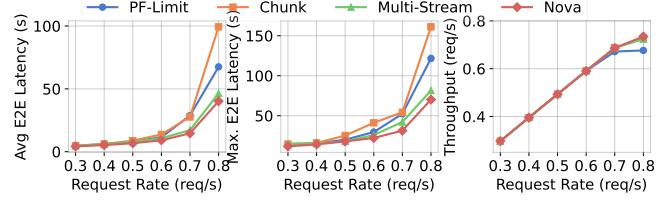


Fig. 10: E2E latency and throughput under different request rates.

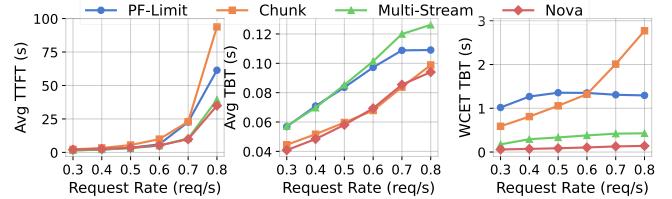


Fig. 11: Latency breakdown under different request rates.

decode as they launch significantly more thread blocks. The phenomenon results in severe blocking of decode execution, yielding low TTFT but high TBT.

To better understand the throughput gain brought by spatial gpu sharing, we report the average LLM decode batch sizes under different scheduling approaches in Figure 12. Chunked prefill maintains throughput by creating more opportunities to batch LLM decode requests and batching LLM prefill chunks with LLM decode requests, thereby avoiding the SM under-utilization caused by solo LLM decode runs. Nova exhibits the lowest average decode batch size due to its consistent token generation strategy, which keeps the number of active decode requests low. Nevertheless, it still achieves comparable throughput because of its integrated GPU spatial sharing strategy.

We also report the queueing delays under Nova scheduling and compare them with the theoretical predictions from the M/G/1 model given in equation (6), as summarized in Table V. The system utilization is defined as  $\lambda T$ . At a request rate of 0.8, the system is overloaded with utilization exceeding 1. It can be observed that the measured queueing delays closely match the theoretical predictions, indicating that Nova scheduling aligns well with the M/G/1 queue model. At a request rate of 0.7, the discrepancy is relatively larger, as the system is nearly saturated.

#### D. Impact of Weight Offloading

Here we report the impact of weight offloading on the vision encoder. We evaluate on both the single forward pass and the end-to-end impact when integrated into request scheduling. As summarized in Table VI, only 2 physical layers in GPU memory are sufficient to hide the latency of weight swapping, enabling over 90% memory reduction for the vision encoder, with negligible overhead in both single-pass and end-to-end serving scenarios. Increasing the number of physical layers  $K$  slightly raises the overhead, because of the increased synchronization complexity between model computing and weight swapping, as each physical layer requires a dedicated CUDA event for synchronization.

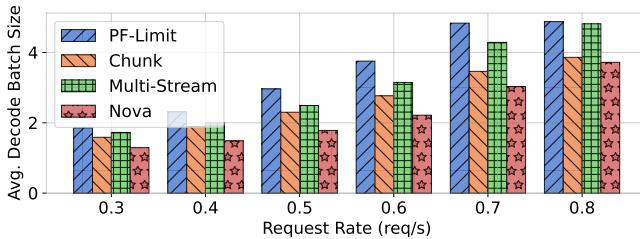


Fig. 12: Average decode batch size under different approaches.

TABLE V: Measured queueing delays under Nova scheduling versus theoretical predictions from the M/G/1 model.

Request Rate $\lambda$	0.3	0.4	0.5	0.6	0.7	0.8
Average Processing Time $T$ (s)	1.39	1.44	1.43	1.42	1.38	1.35
System Utilization	0.42	0.58	0.72	0.86	0.97	1.08
Average Measured Queueing Latency (s)	0.60	1.14	2.06	3.82	8.49	33.57
Theoretical M/G/1 Queueing Latency Prediction (s)	0.51	1.00	1.86	4.32	20.99	\

TABLE VI: Weight offloading impact on compute and memory efficiency.

Model	Raw	Vision Encode w/ Layer Offload			
		K=2	K=3	K=4	K=5
Single Forward	Forward Latency (ms)	797.2	798.6	802.2	804.8
	GPU Mem. Usage (MB)	8595.2	692.1	821.6	951.1
Serving Scenario	Avg E2E Latency (s)	11.96	11.85	12.35	12.21
	Maximum E2E Latency (s)	21.94	21.83	22.62	22.39

### E. Contribution of Adaptive SM Partition

We further analyze the adaptive SM partition strategy by comparing Nova with static SM partition strategies, where fixed SMs are allocated to LLM decode requests when co-running with LLM prefill or vision encode stages. As shown in Figure 13, the proposed adaptive strategy consistently outperforms static counterparts across varying request rates, particularly when the request rate increases and request bursts become more frequent. Under high load, allocating too many SMs to LLM decode causes LLM prefill and vision encode requests to experience longer waiting times, further increasing end-to-end latency. In addition, static SM partitioning struggles to globally balance the system throughput between cross-stage co-running and LLM decode request batching. Over-allocating SMs to LLM decode reduces batching opportunities, as early LLM decode requests finish soon to wait for others.

We also visualize the scheduling timeline of Nova during a request burst in Figure 14. The figure shows the latency changes of individual forward passes for LLM decode and vision encode requests, along with the fluctuation in the number of pending requests. Latency values are normalized between 0 and 1 for comparability. We find that when the number of pending requests increases, the scheduler dynamically allocates more SMs to vision encode requests to accelerate their processing. This adaptive behavior helps prevent the accumulation of pending requests and mitigates excessive waiting

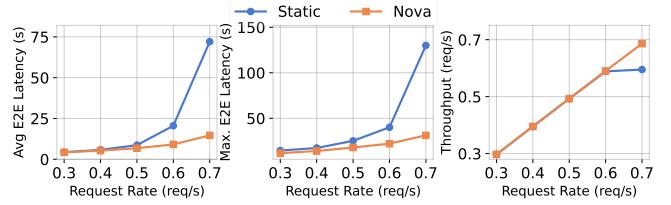


Fig. 13: E2E latency and throughput comparison between adaptive and static SM partition.

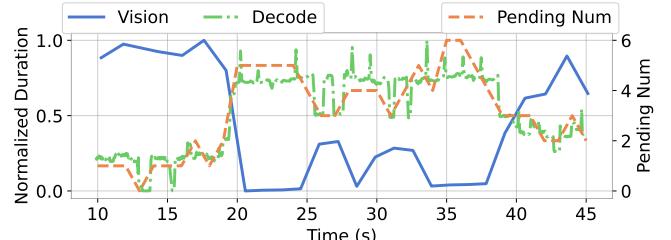


Fig. 14: Practical adaptive SM partition example when requests burst.

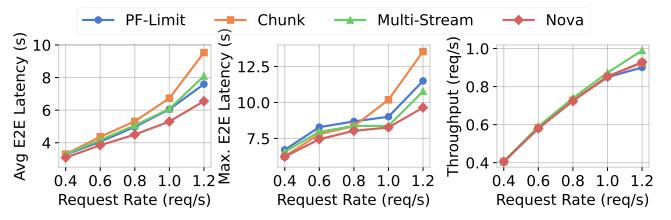


Fig. 15: Performance on RTX 4090 using real-world trace data.

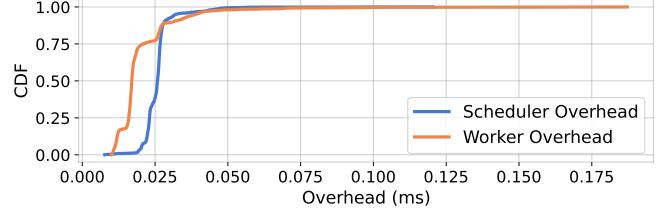


Fig. 16: Overhead CDF of scheduler and model worker.

delays, ultimately improving overall system responsiveness.

### F. Scalability and Generalization Test

To evaluate the generalization of Nova into GPU models with smaller memories and real-world request distributions, we conduct additional experiments on an RTX 4090 using real-world traces from a production VLM agent, whose results are given in Figure 15. Note CogAgent has 13 B model parameters, and the model alone consumes approximately 27 GB of GPU memory in BF16 precision, exceeding the 24 GB memory capacity of RTX 4090. Therefore, we adopt weight offloading for the vision encoder in all approaches. The results show a similar trend to those on RTX A6000: Nova consistently outperforms other baselines in terms of end-to-end latency while maintaining comparable throughput, demonstrating its scalability and cross-platform compatibility.

### G. Overhead Analysis

We finally report the scheduling overhead of Nova, which mainly comes from the synchronization between the scheduler

and model workers for SM partition adjustments. As shown in Figure 16, we present the cumulative distribution function (CDF) of overhead. The overhead remains lightweight compared to the overall model inference latency. Specifically, the model worker performs synchronization and performs the SM re-partition only once before each model forward pass. Moreover, since the scheduler runs in a separate thread and the model worker continues inference concurrently during request scheduling, the scheduling overhead is further minimized.

## VI. DISCUSSION

**Extensions to Additional VLMs and Edge Devices:** Typical VLMs and other multi-modal LLMs use separate modality encoders with LLM modules [42], [43]. Our hardware-level spatial sharing and pipelining approach is broadly applicable: Prefill-decode co-running remains effective, while vision-decode co-running benefits depend on model size and image resolution, which affect hyperparameter tuning. When deploying large-scale VLMs on resource-constrained devices, the efficiency of weight offloading is primarily determined by the ratio between device compute capability and main memory bandwidth. Our experiments indicate that PCIe 3.0 bandwidth is sufficient to hide weight transfer latency. Nevertheless, further improvements in model quantization [44]–[46] are needed, since quantizing the vision encoder can lead to significant accuracy degradation. Complementary strategies, such as self-speculative decoding [47]–[49], may also be employed to improve LLM inference efficiency.

**Exploiting Frame Similarity:** One promising direction to improve efficiency is exploiting frame similarity across requests, especially when Nova is employed in video processing with high temporal redundancy. Currently, Nova does not use similarity-based batching or region skipping, as cross-attention over all image patches or tokens makes reusing intermediate features potentially harmful to accuracy. However, there are two potential ways to exploit. First, the vision encoder can be accelerated by skipping redundant regions with high similarity, for example by merging or pruning tokens that are highly similar [50]–[52]. Second, the KV-cache of image tokens corresponding to similar regions can be reused in the LLM, while carefully retaining important tokens for recomputation to preserve accuracy [53], [54].

## VII. RELATED WORKS

**Efficient LLM Serving Systems.** Prior work on LLM serving falls broadly into three categories. First, *memory efficiency*, particularly targeting at alleviating KV-cache bottlenecks [20], [40], [55]–[62]. For example, PagedAttention [20] reduces fragmentation via block-level cache management, while SGLang [40] improves cache reuse with RadixCache. Second, *GPU utilization and throughput*, targeting better resource efficiency [11]–[15], [23], [63], [64]. DistServe [11] disaggregates prefill and decode across devices, and NanoFlow [15] overlaps heterogeneous kernels via fine-grained mini-batching. Third, *quality of service*, which aims to meet latency metrics such as TTFT and TBT [65]–[68].

Despite these efforts, most approaches overlook the distinct execution patterns of multi-modal LLMs, especially agentic VLMs. Specifically, they do not account for the heterogeneous compute and memory demands, nor the stage-level modularity, present in VLM agent workloads. To the best of our knowledge, no dominant serving framework has yet been proposed to systematically address the challenges of efficient and low-latency agentic VLM inference.

**GPU Resource Sharing.** GPU sharing techniques aim to improve utilization and support concurrent execution. For DNN inference, both temporal and spatial sharing have been explored to reduce idleness and enable parallelism [17], [18], [29], [69]–[72], typically by leveraging heterogeneous resource demands for co-scheduling. Low-level GPU partitioning has also been studied to support kernel co-execution [19], [30]–[33], [73]–[76], and can be categorized into inter-SM and intra-SM approaches. Intra-SM techniques, such as elastic kernels [19], [33], provide fine-grained sharing but often suffer from unstable performance due to opaque scheduling. Moreover, intra-SM partitioning tends to prioritize throughput improvements without providing sufficient flexibility for fine-grained, latency-aware scheduling, making it less suitable for dynamic, multi-stage VLM serving workloads. In contrast, inter-SM partitioning offers more stable performance and better compatibility with adaptive scheduling policies. Therefore, we adopt inter-SM sharing techniques to support adaptive, latency-aware resource allocation in VLM agent serving.

## VIII. CONCLUSION

We presented Nova, a real-time scheduling framework tailored for multi-stage agentic vision-language model serving on a single GPU. Nova integrates adaptive cross-stage pipeline parallelization with fine-grained SM partitioning to fully exploit GPU resources, and employs a lightweight weight offloading strategy to alleviate memory constraints by high-resolution vision encoders. Through dynamic scheduling based on a latency–throughput Pareto frontier, Nova ensures consistent responsiveness under diverse and bursty workloads. Extensive experiments on both synthetic and real-world agent tasks demonstrate Nova’s superiority in terms of end-to-end latency, throughput, and GPU memory efficiency.

## ACKNOWLEDGEMENT

This work was sponsored in part by the National Key R&D Program of China (No. 2022ZD0119100), in part by China NSF grant No. 62472278, 62025204, 62432007, 62441236, 62332014, and 62332013, in part by the Taishan Industrial Experts Program, in part by Alibaba Group through Alibaba Innovation Research Program, and in part by Tencent Rhino Bird Key Research Project. This work was partially supported by SJTU Kunpeng & Ascend Center of Excellence. The opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies or the government.

## REFERENCES

[1] J. Wang, H. Xu, J. Ye, M. Yan, W. Shen, J. Zhang, F. Huang, and J. Sang, “Mobile-agent: Autonomous multi-modal mobile device agent with visual perception,” *arXiv preprint arXiv:2401.16158*, 2024.

[2] W. Hong, W. Wang, Q. Lv, J. Xu, W. Yu, J. Ji, Y. Wang, Z. Wang, Y. Dong, M. Ding *et al.*, “Cogagent: A visual language model for gui agents,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 14281–14290.

[3] J. Zhang, J. Huang, S. Jin, and S. Lu, “Vision-language models for vision tasks: A survey,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

[4] D. Zhu, J. Chen, X. Shen, X. Li, and M. Elhoseiny, “Minigpt-4: Enhancing vision-language understanding with advanced large language models,” *arXiv preprint arXiv:2304.10592*, 2023.

[5] R. Niu, J. Li, S. Wang, Y. Fu, X. Hu, X. Leng, H. Kong, Y. Chang, and Q. Wang, “Screenagent: A vision language model-driven computer control agent,” *arXiv preprint arXiv:2402.07945*, 2024.

[6] S. Zhai, H. Bai, Z. Lin, J. Pan, P. Tong, Y. Zhou, A. Suhr, S. Xie, Y. LeCun, Y. Ma *et al.*, “Fine-tuning large vision-language models as decision-making agents via reinforcement learning,” *Advances in neural information processing systems*, vol. 37, pp. 110935–110971, 2024.

[7] M. Zhao, S. Liu, F. Wu, and G. Chen, “Responsive dnn adaptation for video analytics against environment shift via hierarchical mobile-cloud collaborations,” in *Proceedings of the 23rd ACM Conference on Embedded Networked Sensor Systems*, 2025, pp. 317–331.

[8] Y. Li, H. Wen, W. Wang, X. Li, Y. Yuan, G. Liu, J. Liu, W. Xu, X. Wang, Y. Sun *et al.*, “Personal llm agents: Insights and survey about the capability, efficiency and security,” *arXiv preprint arXiv:2401.05459*, 2024.

[9] M. A. Ferrag, N. Tihanyi, and M. Debbah, “From llm reasoning to autonomous ai agents: A comprehensive review,” *arXiv preprint arXiv:2504.19678*, 2025.

[10] X. Dong, X. Zhang, W. Bu, D. Zhang, and F. Cao, “A survey of llm-based agents: Theories, technologies, applications and suggestions,” in *2024 3rd International Conference on Artificial Intelligence, Internet of Things and Cloud Computing Technology (AIoTC)*. IEEE, 2024, pp. 407–413.

[11] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. 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)*, 2024, pp. 193–210.

[12] P. Patel, E. Choukse, C. Zhang, A. Shah, I. Goiri, S. Maleki, and R. Bianchini, “Splitwise: Efficient generative llm inference using phase splitting,” in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 118–132.

[13] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee, “Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarahi-Serve},” in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 117–134.

[14] A. K. Kamath, R. Prabhu, J. Mohan, S. Peter, R. Ramjee, and A. Panwar, “Pod-attention: Unlocking full prefill-decode overlap for faster llm inference,” in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 897–912.

[15] K. Zhu, Y. Zhao, L. Zhao, G. Zuo, Y. Gu, D. Xie, Y. Gao, Q. Xu, T. Tang, Z. Ye *et al.*, “Nanoflow: Towards optimal large language model serving throughput,” *arXiv preprint arXiv:2408.12757*, 2024.

[16] P. Yu and M. Chowdhury, “Fine-grained gpu sharing primitives for deep learning applications,” *Proceedings of Machine Learning and Systems*, vol. 2, pp. 98–111, 2020.

[17] F. Strati, X. Ma, and A. Klimovic, “Orion: Interference-aware, fine-grained gpu sharing for ml applications,” in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 1075–1092.

[18] B. Wu, Z. Zhang, Z. Bai, X. Liu, and X. Jin, “Transparent {GPU} sharing in container clouds for deep learning workloads,” in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 69–85.

[19] H. Zhao, W. Cui, Q. Chen, J. Zhao, J. Leng, and M. Guo, “Exploiting intra-sm parallelism in gpus via persistent and elastic blocks,” in *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, 2021, pp. 290–298.

[20] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, “Efficient memory management for large language model serving with pagedattention,” in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[21] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark *et al.*, “Learning transferable visual models from natural language supervision,” in *International conference on machine learning*. PMLR, 2021, pp. 8748–8763.

[22] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, “An image is worth 16x16 words: Transformers for image recognition at scale,” *arXiv preprint arXiv:2010.11929*, 2020.

[23] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, “Orca: A distributed serving system for {Transformer-Based} generative models,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 521–538.

[24] Z. Chen, J. Wu, W. Wang, W. Su, G. Chen, S. Xing, M. Zhong, Q. Zhang, X. Zhu, L. Lu *et al.*, “Internvl: Scaling up vision foundation models and aligning for generic visual-linguistic tasks,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2024, pp. 24185–24198.

[25] F. Li, R. Zhang, H. Zhang, Y. Zhang, B. Li, W. Li, Z. Ma, and C. Li, “Llava-next-interleave: Tackling multi-image, video, and 3d in large multimodal models,” *arXiv preprint arXiv:2407.07895*, 2024.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.

[27] “NVIDIA Nsight Compute — developer.nvidia.com,” <https://developer.nvidia.com/nsight-compute>.

[28] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, “Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2019.

[29] B.-S. Han, T. Paul, Z. Liu, and A. Gandhi, “Kace: Kernel-aware colocation for efficient gpu spatial sharing,” in *Proceedings of the 2024 ACM Symposium on Cloud Computing*, 2024, pp. 460–469.

[30] J. Bakita and J. H. Anderson, “Hardware compute partitioning on nvidia gpus,” in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2023, pp. 54–66.

[31] NVIDIA, “NVIDIA Multi-Instance GPU (MIG) — nvidia.com,” <https://www.nvidia.com/en-us/technologies/multi-instance-gpu/>, 2025, [Accessed 09-05-2025].

[32] NVIDIA, “CUDA Driver API :: CUDA Toolkit Documentation — docs.nvidia.com,” [https://docs.nvidia.com/cuda/cuda-driver-api/group\\_CUDA\\_GREEN\\_CONTEXTS.html](https://docs.nvidia.com/cuda/cuda-driver-api/group_CUDA_GREEN_CONTEXTS.html), [Accessed 09-05-2025].

[33] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, “Improving gppu concurrency with elastic kernels,” *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 407–418, 2013.

[34] V. Thakkar, P. Ramani, C. Cecka, A. Shivam, H. Lu, E. Yan, J. Kosaian, M. Hoemmen, H. Wu, A. Kerr, M. Nicely, D. Merrill, D. Blasig, F. Qiao, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, and M. Gupta, “CUTLASS,” Jan. 2023. [Online]. Available: <https://github.com/NVIDIA/cutlass>

[35] A. Paszke, “Pytorch: An imperative style, high-performance deep learning library,” *arXiv preprint arXiv:1912.01703*, 2019.

[36] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu, “Roformer: Enhanced transformer with rotary position embedding,” *Neurocomputing*, vol. 568, p. 127063, 2024.

[37] B. Zhang and R. Sennrich, “Root mean square layer normalization,” *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[38] Z. Ye, L. Chen, R. Lai, W. Lin, Y. Zhang, S. Wang, T. Chen, B. Kasikci, V. Grover, A. Krishnamurthy, and L. Ceze, “Flashinference: Efficient and customizable attention engine for llm inference serving,” *arXiv preprint arXiv:2501.01005*, 2025. [Online]. Available: <https://arxiv.org/abs/2501.01005>

[39] Y. Xu, X. Liu, X. Sun, S. Cheng, H. Yu, H. Lai, S. Zhang, D. Zhang, J. Tang, and Y. Dong, “Androidlab: Training and systematic benchmarking of android autonomous agents,” *arXiv preprint arXiv:2410.24024*, 2024.

[40] L. Zheng, L. Yin, Z. Xie, C. L. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez *et al.*, “Sglang: Efficient execution of structured language model programs,” *Advances in Neural Information Processing Systems*, vol. 37, pp. 62557–62583, 2024.

[41] “Performance and tuning 2014; vllm — docs.vllm.ai,” <https://docs.vllm.ai/en/v0.4.2/models/performance.html>.

[42] C. Lyu, M. Wu, L. Wang, X. Huang, B. Liu, Z. Du, S. Shi, and Z. Tu, “Macaw-llm: Multi-modal language modeling with image, audio, video, and text integration,” *arXiv preprint arXiv:2306.09093*, 2023.

[43] F. Shu, L. Zhang, H. Jiang, and C. Xie, “Audio-visual llm for video understanding,” *arXiv preprint arXiv:2312.06720*, 2023.

[44] E. Frantar, S. Ashkboos, T. Hoefer, and D. Alistarh, “Gptq: Accurate post-training quantization for generative pre-trained transformers,” *arXiv preprint arXiv:2210.17323*, 2022.

[45] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, “Awq: Activation-aware weight quantization for on-device llm compression and acceleration,” *Proceedings of machine learning and systems*, vol. 6, pp. 87–100, 2024.

[46] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, “Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale,” *Advances in neural information processing systems*, vol. 35, pp. 30318–30332, 2022.

[47] Z. Zhou, X. Ning, K. Hong, T. Fu, J. Xu, S. Li, Y. Lou, L. Wang, Z. Yuan, X. Li *et al.*, “A survey on efficient inference for large language models,” *arXiv preprint arXiv:2404.14294*, 2024.

[48] J. Zhang, J. Wang, H. Li, L. Shou, K. Chen, G. Chen, and S. Mehrotra, “Draft & verify: Lossless large language model acceleration via self-speculative decoding,” *arXiv preprint arXiv:2309.08168*, 2023.

[49] M. Elhoushi, A. Shrivastava, D. Liskovich, B. Hosmer, B. Wasti, L. Lai, A. Mahmoud, B. Acun, S. Agarwal, A. Roman *et al.*, “Layerskip: Enabling early exit inference and self-speculative decoding,” *arXiv preprint arXiv:2404.16710*, 2024.

[50] Z. Song, C. Qi, F. Liu, N. Jing, and X. Liang, “Cmc: Video transformer acceleration via codec assisted matrix condensing,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 201–215.

[51] S. Lee, S. H. Hwang, S. Oh, B. J. Park, and Y. Cho, “Multi-input vision transformer with similarity matching,” in *International Workshop on PRedictive Intelligence In MEdicine*. Springer, 2023, pp. 184–193.

[52] S. Black, A. Stylianou, R. Pless, and R. Souvenir, “Visualizing paired image similarity in transformer networks,” in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2022, pp. 3164–3173.

[53] J. Yao, H. Li, Y. Liu, S. Ray, Y. Cheng, Q. Zhang, K. Du, S. Lu, and J. Jiang, “Cacheblend: Fast large language model serving for rag with cached knowledge fusion,” in *Proceedings of the Twentieth European Conference on Computer Systems*, 2025, pp. 94–109.

[54] J. Yang, B. Hou, W. Wei, Y. Bao, and S. Chang, “Kvlink: Accelerating large language models via efficient kv cache reuse,” *arXiv preprint arXiv:2502.16002*, 2025.

[55] M. Ji, S. Yi, C. Koo, S. Ahn, D. Seo, N. Dutt, and J.-C. Kim, “Demand layering for real-time dnn inference with minimized memory usage,” in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 291–304.

[56] I. Rehg, “Kv-compress: Paged kv-cache compression with variable compression rates per attention head,” *arXiv preprint arXiv:2410.00161*, 2024.

[57] W. Lee, J. Lee, J. Seo, and J. Sim, “{InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management,” in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 155–172.

[58] B. Lin, C. Zhang, T. Peng, H. Zhao, W. Xiao, M. Sun, A. Liu, Z. Zhang, L. Li, X. Qiu *et al.*, “Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache,” *arXiv preprint arXiv:2401.02669*, 2024.

[59] X. Li, Y. Li, Y. Li, T. Cao, and Y. Liu, “Flexnn: Efficient and adaptive dnn inference on memory-constrained edge devices,” in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 709–723.

[60] W. Kang, J. Lee, Y. Lee, S. Oh, K. Lee, and H. S. Chwa, “Rt-swap: Addressing gpu memory bottlenecks for real-time multi-dnn inference,” in *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2024, pp. 373–385.

[61] R. Giannessi, A. Biondi, and A. Biasci, “Rt-mimalloc: A new look at dynamic memory allocation for real-time systems,” in *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2024, pp. 173–185.

[62] R. Prabhu, A. Nayak, J. Mohan, R. Ramjee, and A. Panwar, “vattention: Dynamic memory management for serving llms without pagelatency,” in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 1133–1150.

[63] J. Bakita and J. H. Anderson, “Demystifying nvidia gpu internals to enable reliable gpu management,” in *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2024, pp. 294–305.

[64] J. Chen, S. Bai, Z. Wang, S. Wu, C. Du, H. Yang, R. Gong, S. Liu, F. Wu, and G. Chen, “Pre<sup>3</sup>: Enabling deterministic pushdown automata for faster structured LLM generation,” in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds. Vienna, Austria: Association for Computational Linguistics, Jul. 2025, pp. 11253–11267. [Online]. Available: <https://aclanthology.org/2025.acl-long.551/>

[65] K. Cheng, Z. Wang, W. Hu, T. Yang, J. Li, and S. Zhang, “Towards slo-optimized llm serving via automatic inference engine tuning,” *arXiv preprint arXiv:2408.04323*, 2024.

[66] Y. Lin, S. Peng, S. Wu, Y. Li, C. Lu, C. Xu, and K. Ye, “Planck: Optimizing llm inference performance in pipeline parallelism with fine-grained slo constraint,” in *2024 IEEE International Conference on Web Services (ICWS)*. IEEE, 2024, pp. 1306–1313.

[67] Y. Li, Z. Li, W. Yang, and C. Liu, “Rt-lm: Uncertainty-aware resource management for real-time inference of language models,” in *2023 IEEE Real-Time Systems Symposium (RTSS)*, 2023, pp. 158–171.

[68] R. Wang, H. Liu, J. Qiu, M. Xu, R. Guérin, and C. Lu, “Progressive neural compression for adaptive image offloading under timing constraints,” in *2023 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2023, pp. 118–130.

[69] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, “{AntMan}: Dynamic scaling on {GPU} clusters for deep learning,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 533–548.

[70] P. Yu and M. Chowdhury, “Fine-grained gpu sharing primitives for deep learning applications,” *Proceedings of Machine Learning and Systems*, vol. 2, pp. 98–111, 2020.

[71] M. Han, H. Zhang, R. Chen, and H. Chen, “Microsecond-scale pre-emption for concurrent {GPU-accelerated}{DNN} inferences,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 539–558.

[72] Y. Xu, Z. Liu, X. Fu, S. Liu, F. Wu, and G. Chen, “Flex: Adaptive task batch scheduling with elastic fusion in multi-modal multi-view machine perception,” in *2024 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2024, pp. 294–307.

[73] M. Chow, A. Jahanshahi, and D. Wong, “Krisp: Enabling kernel-wise right-sizing for spatial partitioned gpu inference servers,” in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 624–637.

[74] K. Gupta, J. A. Stuart, and J. D. Owens, *A study of persistent threads style GPU programming for GPGPU workloads*. IEEE, 2012.

[75] NVIDIA, “Multi-Process Service — docs.nvidia.com,” <https://docs.nvidia.com/deploy/mps/index.html>, 2025, [Accessed 09-05-2025].

[76] S. Jain, I. Baek, S. Wang, and R. Rajkumar, “Fractional gpus: Software-based compute and memory bandwidth reservation for gpus,” in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019, pp. 29–41.