Dynamic Micro-Batch and Token-Budget Scheduling for IoT-Scale Pipeline-Parallel LLM Inference

Juncheol Ahn , Yubin Son , Daemin Kim , Sejin Park *

Posted Date: 9 December 2025

doi: 10.20944/preprints202512.0788.v1

Keywords: large language models; IoT; edge computing; cloud inference; pipeline parallelism; micro-batching; GPU scheduling

 doi:10.20944/preprints202512.0788.v1

Article

Dynamic Micro-Batch and Token-Budget Scheduling for IoT-Scale Pipeline-Parallel LLM Inference

Juncheol Ahn, Yubin Son, Daemin Kim and Sejin Park *

System Software Laboratory, Department of Computer Engineering, Keimyung University, Daegu 42601, Republic of Korea
* Correspondence: baksejin@kmu.ac.kr

Abstract

Large language models (LLMs) are increasingly integrated into IoT–edge–cloud systems, where real-time analytics and natural-language interaction demand both high throughput and stable latency. However, IoT workloads are inherently bursty and heterogeneous: prompt and generation lengths vary widely, and prefill- and decode-heavy requests coexist. When served via pipeline-parallel LLM inference, these characteristics amplify micro-batch imbalance and communication stalls, leading to substantial GPU idle time and degraded TTFT/ITL service-level objectives (SLOs). We propose a runtime-adaptive scheduling framework that combines *Dynamic Token-Budget Estimation* with *Dynamic Micro-Batch Scheduling*. Unlike static token-budget settings—which act primarily as latency knobs—our approach dynamically adjusts token budgets to balance prefill/decode workloads across micro-batches, while selecting the optimal number of micro-batches to minimize pipeline bubbles under varying compute and network conditions. Implementing the framework on a four-node RTX 4070 cluster running pipeline-parallel Llama-2-13b-chat with vLLM, we evaluate both synthetic offline workloads and online Poisson-arrival workloads. The combined scheme reduces GPU idle time by up to **55%** and improves throughput (completion time) by up to **1.61**× compared with the baseline, while significantly increasing TTFT/ITL SLO satisfaction under bursty conditions. These results demonstrate that dynamic, workload-aware scheduling is essential for scalable and latency-stable LLM inference in IoT–edge–cloud environments.

Keywords: large language models; IoT; edge computing; cloud inference; pipeline parallelism; micro-batching; GPU scheduling

1. Introduction

Large language models (LLMs) are increasingly embedded into IoT, edge, and cloud services to support real-time analytics, semantic understanding, and interactive decision-making. Because IoT and edge devices lack sufficient compute capacity, their inference workloads are typically offloaded to cloud clusters, where pipeline parallelism is used to serve large numbers of concurrent requests efficiently.

However, real-world IoT workloads exhibit two challenging properties: (1) *burstiness*, where arrival rates fluctuate unpredictably, and (2) *heterogeneity*, where prompt and generation lengths vary significantly across devices and tasks. These characteristics create highly imbalanced mixtures of prefill- and decode-heavy requests. When such workloads flow through a pipeline-parallel LLM, even small imbalances across micro-batches propagate into substantial pipeline bubbles, inflating GPU idle time and destabilizing TTFT and inter-token latency (ITL).

A central—but often misunderstood—component of modern inference systems such as vLLM is the *token-budget* parameter. Contrary to common intuition, the token-budget is not a throughput-optimization lever; rather, it is a *latency control knob* governing the TTFT–ITL trade-off. A small budget improves ITL but harms TTFT, while a large budget improves TTFT but increases ITL and prefill–decode interference. Static token-budget configurations therefore struggle to meet SLOs under dynamic IoT traffic and frequently exacerbate pipeline imbalance.

 doi:10.20944/preprints202512.0788.v1

From a throughput perspective, the optimal condition would balance total computation across all micro-batches, allowing pipeline stages to progress in lockstep with minimal idle time. Yet this ideal is unattainable in practice due to unpredictable workload variability and strict TTFT/ITL SLO requirements that constrain how batches can be reshaped.

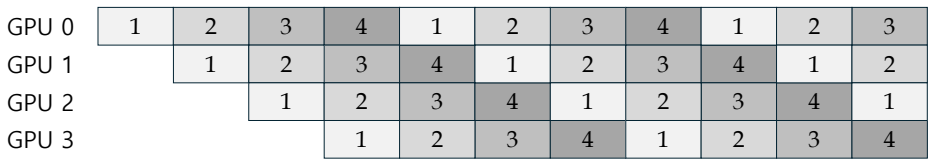
These limitations motivate the need for a scheduling mechanism that is both *dynamic* and *workload-aware*. In this work, we introduce two runtime-adaptive techniques: (1) *Dynamic Token-Budget Estimation*, which balances prefill and decode workloads across micro-batches while preserving TTFT/ITL constraints, and (2) *Dynamic Micro-Batch Scheduling*, which selects the micro-batch count that minimizes expected pipeline stalls under empirical compute and communication models.

We implement the proposed framework on a four-node RTX 4070 cluster using pipeline-parallel Llama-2-13b-chat with vLLM, and evaluate it under both synthetic offline workloads and realistic online Poisson-arrival workloads. Our approach reduces GPU idle time by up to **55%**, improves throughput by up to **1.61**×, and achieves consistently higher TTFT/ITL SLO satisfaction compared with the baseline. These results demonstrate that dynamic scheduling is crucial for scalable and latency-sensitive LLM inference in IoT–edge–cloud environments.

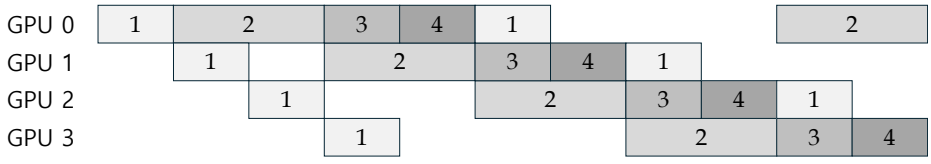
2. Background

Maximizing GPU utilization in LLM inference requires a precise understanding of how workloads flow through the pipeline and where pipeline bubbles arise. Modern inference engines such as `vLLM` [1] typically employ pipeline parallelism to split a large model into multiple stages across GPUs, and micro-batching to keep those stages busy. However, when the amount of work in each micro-batch is imbalanced, or when inter-GPU communication latency becomes non-negligible, some stages finish earlier than others and wait idly, degrading end-to-end throughput.

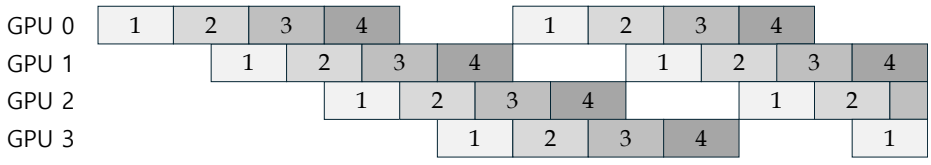
Figure 1 summarizes these effects with three representative scenarios: (a) an ideal pipeline where all micro-batches carry similar workloads, (b) a pipeline suffering from micro-batch imbalance, and (c) a pipeline where communication latency between stages creates additional idle time. In the ideal case, micro-batches move like items on a conveyor belt and each GPU stays busy during almost the entire pipeline cycle. In contrast, a single compute-heavy micro-batch or high communication latency can stall downstream stages and introduce long idle periods, even when total GPU compute capacity is sufficient.



(**a**) Ideal Pipeline: balanced micro-batch workloads, no idle.



(**b**) Imbalanced Micro-Batch: micro-batch with higher computation causes idle.



(**c**) Communication Latency: inter-GPU transfer delays create pipeline idle.

Figure 1. Pipeline execution scenarios with four micro-batches across four GPU ranks: (a) ideal, (b) with workload imbalance, and (c) with communication latency.

 doi:10.20944/preprints202512.0788.v1

These issues become more pronounced in emerging IoT/edge/cloud deployments. IoT devices and edge nodes generate requests with highly variable prompt lengths and bursty arrival patterns [2,3]. Inference is typically offloaded to a cloud or edge GPU cluster, where many devices share the same pipeline-parallel LLM backend. From the perspective of the pipeline, IoT traffic appears as a sequence of mixed-length prefill and decode workloads that arrive at irregular intervals. Consequently, it is difficult to maintain balanced micro-batches and fully utilized GPUs unless the scheduler explicitly accounts for these characteristics. The rest of this section reviews the key architectural features that interact with our proposed scheduling method: the asymmetry between prefill and decode, the behavior of the KV-cache, and the role of the token-budget parameter.

2.1. Prefill and Decode Characteristics

LLM inference comprises two distinct computational phases:

- **Prefill.** The model processes the entire prompt and builds the key–value (KV) cache for all input tokens across all layers. This phase is highly parallel and compute-intensive because self-attention must consider all previous tokens in the prompt. As a result, its *computational* demand and memory footprint are substantial.
- **Decode.** Leveraging the cached KV tensors, the model generates one token at a time in an autoregressive manner. Each decode step only processes the newly generated token while attending to the existing KV cache. Thus, per-step computation and memory usage are significantly smaller than in the prefill phase, but must be repeated many times.

This structural asymmetry causes significant *variation* in per-request workload. Requests with long prompts but short outputs are dominated by prefill cost, whereas chat-like sessions with many generated tokens are dominated by decode. In pipeline-parallel execution, multiple requests and micro-batches are often processed concurrently on different pipeline stages. If the mixture of prefill- and decode-heavy requests is skewed across micro-batches, some stages finish much earlier than others and wait for the slowest micro-batch to complete, creating pipeline bubbles as illustrated in Figure 1.

2.2. KV-Cache and Memory Preemption Issue

Transformer-based LLMs retain the KV cache in GPU memory to avoid recomputing attention for past tokens during decoding. While this design is essential for throughput, it also introduces complex memory-management challenges. The KV cache grows with both prompt length and the number of generated tokens. When many long-context or long-generation requests coexist, the aggregate KV-cache footprint can approach or exceed the available GPU memory capacity.

To cope with this pressure, inference engines rely on block-level KV-cache management policies, such as LRU or LFU-style eviction, and sometimes offload rarely used blocks to host memory or remote GPUs. However, these mechanisms are fundamentally reactive: once a block is evicted or migrated, some of the previously computed work is partially lost and must be recomputed if the corresponding request becomes active again. From a pipeline perspective, frequent KV-cache eviction or migration can propagate as additional stalls and bubbles, because downstream stages must wait for recomputation or data transfer to complete before proceeding.

In IoT and edge scenarios, where request burstiness and context lengths vary substantially across time and devices, KV-cache usage tends to fluctuate strongly. This makes it even more important to design scheduling strategies that avoid creating extreme imbalances in per-micro-batch KV-cache usage, since such imbalances amplify the likelihood of eviction and the associated recomputation overhead.

2.3. Token-Budget: A Latency Control Parameter for Maintaining TTFT–ITL SLOs

Modern LLM inference systems such as `vLLM` [1] expose the *token-budget* parameter, which limits the maximum amount of computation allowed per micro-batch during decoding. Importantly, the token-budget is **not a throughput-optimization mechanism**. Instead, it is a **latency-oriented control**

 doi:10.20944/preprints202512.0788.v1

knob designed to regulate the trade-off between *Time To First Token* (TTFT) and *Inter-Token Latency* (ITL), enabling the system to satisfy application-level SLOs.

Operationally, the token-budget constrains how many tokens from all active requests can be processed in a single micro-batch. A smaller budget forces the scheduler to process fewer tokens per batch, whereas a larger budget allows more tokens to be advanced at once. This leads to the following fundamental latency behaviors:

- **Small token-budget (ITL-friendly, TTFT-unfriendly).** With fewer tokens per micro-batch, each decode iteration is lightweight, reducing ITL. Furthermore, when a prefill request arrives, the scheduler can insert its prefill chunk between decode batches with minimal disruption, mitigating *Prefill–Decode interference*. However, because each micro-batch carries little work, the first token arrives later, increasing TTFT.
- **Large token-budget (TTFT-friendly, ITL-unfriendly).** Larger budgets enable more prefill tokens to be processed at once, improving TTFT for long-prompt requests. However, inserting large prefill chunks into an ongoing decode stream causes significant Prefill–Decode interference, increasing ITL for existing decoding sessions and potentially amplifying queueing delays.

From a throughput perspective, the ideal scenario is to achieve *perfectly balanced workload distribution* across all micro-batches. If the total computational load of prefill and decode were evenly partitioned across micro-batches, each pipeline stage would process similar amounts of work per cycle, eliminating pipeline imbalance and maximizing GPU utilization. However, this idealized scenario is unattainable in real systems for two reasons:

1. **Real-time workloads fluctuate unpredictably.** IoT and edge applications generate highly variable prompt lengths, decode lengths, and arrival patterns, causing sudden shifts in workload composition.
2. **TTFT and ITL must remain within strict SLO bounds.** Allowing unrestricted batch growth destabilizes both TTFT and ITL, especially under bursty or prefill-heavy workloads.

Thus, in existing systems, a *static* token-budget is chosen as a compromise to balance TTFT and ITL. However, static budgets fail to adapt to real-time changes in workload characteristics, making them suboptimal both for SLO satisfaction and throughput stability.

This limitation motivates our approach. By dynamically estimating token budgets based on the current mixture of prefill and decode workloads, the proposed scheduler maintains TTFT/ITL SLOs **while also improving throughput**, reducing pipeline imbalance and minimizing GPU idle time. In other words, dynamic token-budget estimation transforms what was previously a static latency knob into an adaptive mechanism that simultaneously preserves SLOs and enhances effective system throughput under IoT-scale workloads.

3. Related Work

A significant body of prior work has investigated task placement, resource management, and inference scheduling for IoT and LLM systems. In this section we summarize the most relevant directions and highlight the remaining gap that motivates our work.

3.1. IoT and Edge–Cloud Computing Architectures

The IoT ecosystem increasingly relies on multi-tier architectures in which sensing, preprocessing, and decision-making operations are distributed across devices, edge nodes, and cloud servers. Edge computing is widely adopted to reduce the communication overhead and latency associated with offloading all computation to the cloud. Hamdan et al. survey edge-computing architectures for IoT applications and emphasize the importance of balancing latency, bandwidth savings, and computational capacity across layers of the system [2]. Andriulo et al. review edge–cloud hybrid computing frameworks and highlight that IoT workloads often exhibit high variability, requiring adaptive mechanisms to maintain efficiency and responsiveness [3]. These studies focus primarily on

 doi:10.20944/preprints202512.0788.v1

where tasks are executed within the IoT–edge–cloud hierarchy. Our work, in contrast, focuses on how to optimize the internal scheduling of LLM inference once the computation is already offloaded to a cloud or edge cluster.

3.2. Scheduling and Offloading in IoT and MEC Systems

Many prior works investigate task scheduling, resource allocation, and latency mitigation for IoT applications at edge or fog computing sites. Lim proposes a latency-aware scheduling method using AI-based task partitioning to optimize end-to-end inferencing delay in small-scale fog environments [4]. Eang et al. present a joint offloading and resource-allocation strategy in mobile edge computing (MEC) to balance latency and cost under real-time IoT constraints [5]. Saeik et al. survey mathematical, AI-based, and control-theoretic approaches for IoT task offloading across edge and cloud infrastructures [6]. These studies focus on global, multi-layer scheduling (device $\rightarrow$ edge $\rightarrow$ cloud) to reduce total response time and network overhead. However, none of them address the fundamental computational bottleneck introduced by large language model inference inside edge/cloud clusters themselves. As IoT applications increasingly adopt LLM-powered analytics, the ability to serve inference requests at high throughput becomes essential. Our work is complementary to IoT offloading research: instead of deciding *where* to run tasks, we propose a method to improve *how* those tasks are executed inside the LLM-serving pipeline.

3.3. LLM Inference Systems and Pipeline Parallelism

Large language model inference introduces unique computational challenges due to the separation between the prefill and decode phases and the variability in request lengths. Systems such as vLLM [7], Orca [8], Sarathi [9,10], DistServe [11], and Splitwise [12] aim to reduce inference overhead through optimized batching, KV-cache management, and phase separation. Disaggregation-based approaches such as TetriInfer [13] and HexGen [14,15] explore architectural redesigns that separate model components to improve throughput under heterogeneous or mixed workloads. Pipeline parallelism is commonly employed to scale inference across multiple GPUs. Prior works such as GPipe [16] focus on pipeline efficiency in the training setting, but similar principles apply to inference. More recent efforts—including H2O [17], StreamingLLM [18], and FlashAttention variants [19–21]— optimize kernel execution, memory access, or attention mechanisms, yet do not address system-level pipeline scheduling under highly variable request patterns. Our work differs from these approaches by targeting pipeline-stage imbalance caused by heterogeneous prompt lengths and decode workloads, a phenomenon particularly exacerbated by bursty IoT request patterns.

3.4. Throughput-Oriented Scheduling for LLM Inference

Recently, researchers have begun exploring throughput-driven resource management for LLM serving. POD-Attention improves prefill–decode overlap [22], while ExeGPT introduces constraint-aware scheduling for multi-tenant inference [23]. DeepSpeed-FastGen and Sequoia propose enhancements for speculative decoding [24,25], but their focus remains token-level efficiency rather than pipeline-level throughput stabilization. Our method complements these by addressing a different dimension: the alignment of micro-batches and token budgets across pipeline stages to minimize GPU idle time in a multi-GPU setting. Whereas speculative decoding or kernel optimization improves per-token efficiency, our scheduler improves per-step pipeline utilization— a critical requirement when serving high-volume IoT workloads.

3.5. Summary and Research Gap

Across IoT system research and LLM inference optimization, a clear gap emerges. IoT and MEC studies optimize where computation should occur (device $\leftrightarrow$ edge $\leftrightarrow$ cloud). LLM inference research optimizes computation inside the model. However, none address how to maintain stable throughput under IoT-scale workload variability in a pipeline-parallel LLM-serving cluster. Our work fills this gap by introducing a dynamic token-budget and micro-batch scheduling framework that improves

 doi:10.20944/preprints202512.0788.v1

internal pipeline utilization and consequently enhances SLO stability for IoT applications relying on cloud-hosted LLMs.

4. Proposed Method

This section introduces two **runtime-adaptive scheduling techniques** designed to minimize **GPU Idle Time** in pipeline-parallel LLM inference. Our approach preserves the conventional pipeline execution model but enhances it with two adaptive mechanisms: (1) *Dynamic Token-Budget Estimation*, which balances compute load across micro-batches, and (2) *Dynamic Micro-batch Scheduling*, which selects the optimal number of micro-batches to reduce communication-induced stalls. Together, these techniques maintain the TTFT/ITL trade-off while improving end-to-end throughput without modifying the underlying model or decode semantics.

4.1. Dynamic Token-Budget Estimation

Schedulers that rely on a *fixed* **token-budget** often distribute prefill and decode workloads unevenly, resulting in micro-batch imbalance. Under prefill-heavy workloads, this imbalance increases the likelihood of **preemption**, where an in-flight prefill is aborted and its KV-cache discarded, wasting already-computed work and exacerbating pipeline stalls.

To mitigate this, we *dynamically adjust the token-budget per micro-batch based on estimated compute cost*. Unlike static pre-chunking strategies, our method assigns at most *one* prefill chunk to each micro-batch and distributes the overall prefill workload evenly across `num_micro_batch`. This choice is motivated by two key observations:

- Each prefill generates KV-cache tensors that occupy GPU memory. Scheduling many small prefills concurrently increases memory pressure and raises the risk of preemption. A dynamic token-budget naturally limits concurrent prefill inflight size, lowering this risk.
- Splitting a prefill into numerous small chunks can benefit decode-heavy workloads, but excessively small chunks inflate TTFT and make GPU compute saturation difficult. By dividing the overall prefill cost across exactly `num_micro_batch`, we maintain a consistent TTFT/ITL balance while improving pipeline utilization.

The effect on pipeline behavior is intuitive: if the first micro-batch requires x seconds and the next requires $y > x$, then a k-stage pipeline may incur up to $(k-1)(y-x)$ idle time (Figure 1(b)). Reducing the compute gap between consecutive micro-batches—by equalizing their workloads—significantly decreases such stage-amplified idle propagation.

By introducing dynamic token-budget estimation, TTFT can be reduced without sacrificing ITL, because the micro-batch count continues to regulate the TTFT/ITL trade-off. Algorithm 1 formalizes this estimation procedure, separating prefill and decode workloads and balancing them across micro-batches to minimize GPU Idle Time.

Algorithm 1: Dynamic Token-Budget Estimation

Input: num_micro_batch, total_num_decode_tokens, num_new_scheduled_prefill_tokens, prefill_budget_queue
Output: token_budget for current micro_batch

while *prefill_budget_queue.size()* < *num_micro_batch* **do**
    prefill_budget_queue.append(0);
if *num_new_scheduled_prefill_tokens* > *0* **then**
    total_tokens ← sum(prefill_budget_queue) + num_new_scheduled_prefill_tokens;
    base, rem ← divmod(total_tokens, num_micro_batch);
    for *i* ← *0* **to** *num_micro_batch - 1* **do**
        prefill_budget_queue[i] ← base;
    for *i* ← *0* **to** *rem - 1* **do**
        prefill_budget_queue[i] += 1;
token_budget ← prefill_budget_queue.pop() + $\lceil \frac{\text{total\_num\_decode\_tokens}}{\text{num\_micro\_batch}} \rceil$;
return token_budget;

 doi:10.20944/preprints202512.0788.v1

4.2. Dynamic Micro-Batch Scheduling

Even with balanced micro-batches, pipeline efficiency still depends critically on the *number of micro-batches*. Given empirical models of per-batch computation and communication cost, the scheduler must choose the value of `num_micro_batch` that minimizes expected pipeline stalls.

Assuming balanced workloads—achieved by Dynamic Token-Budget Estimation—the following two timing components characterize pipeline efficiency:

- **First-Token Generation Time**

$$T_{\text{first\_token}} = comp\_\text{time} \times pp\_\text{size} + comm\_\text{time} \times (pp\_\text{size} - 1) \tag{1}$$

  representing how long it takes for the first micro-batch to traverse the entire pipeline.
- **Rank-0 GPU Total Time**

$$T_{\text{rank0}} = comp\_\text{time} \times num\_micro\_batch \tag{2}$$

  representing how long the first GPU remains busy processing all micro-batches.

The absolute difference

$$|T_{\text{first\_token}} - T_{\text{rank0}}|$$

captures how much idle time is created on downstream pipeline stages. **Dynamic Micro-batch Scheduling** selects the number of micro-batches that minimizes this gap. This procedure is shown in Algorithm 2.

Algorithm 2: Dynamic Micro-batch Scheduling

Input: pp_size, max_num_micro_batch, computation_times(), communication_times(), num_total_tokens
Output: best_num_micro_batch

```
best_nb ← pp_size;
best_gap ← ∞;
for num_batch ← pp_size to max_num_micro_batch do
    batch_size ← num_total_tokens / num_batch;
    comp_time ← computation_times(batch_size);
    comm_time ← communication_times(batch_size);
    total_pipeline_time ← pp_size × comp_time + (pp_size - 1) × comm_time;
    gap ← abs(total_pipeline_time - (num_batch × comp_time));
    if gap < best_gap then
        best_gap ← gap;
        best_nb ← num_batch;
return best_nb;
```

When computation dominates or network latency is relatively small, the algorithm tends to select fewer micro-batches to avoid unnecessary fragmentation. When communication latency is significant, it increases the micro-batch count to improve overlap between computation and communication. This adaptive strategy sustains robust performance across varying workload conditions and pipeline depths.

5. Experiments

In this section, we evaluate the proposed **Dynamic Token-Budget** and **Dynamic Micro-batch** scheduling techniques on a real pipeline-parallel LLM inference system. We first describe the experimental environment and compared configurations, then present results for two scenarios: an

 doi:10.20944/preprints202512.0788.v1

offline setting with controlled synthetic workloads and an *online* setting that emulates real-time service conditions. Our analysis focuses on *GPU Idle Time*, *throughput* (completion time), and *latency SLO satisfaction*.

5.1. Experimental Setup

We evaluated the proposed techniques on a four-node **LAN cluster**. Each node is configured as follows: **CPU** Intel Core i9-13900, **GPU** NVIDIA RTX 4070 (12 GB), **Memory** 64 GB RAM, **OS** Ubuntu 22.04 LTS, **NVIDIA driver** 570, and **CUDA** 12.8, using a Docker-based runtime with Python 3.12.10. The inter-node network bandwidth is **100 Mbps**. We use `vLLM` v0.9.0.1 orchestrated with `Ray` 2.46.0 and `Transformers` 4.52.4 (base image: `vllm/vllm-openai:v0.9.0.1`). The model under test is `meta-llama/Llama-2-13b-chat-hf`, partitioned into *four* pipeline stages—one stage per node.

We compare three configurations:

- **Baseline.** The stock implementation of `vLLM` v0.9.0.1, using its default static token-budget and micro-batch scheduling policy.
- **Dynamic Token-Budget.** Baseline augmented with *Dynamic Token-Budget Estimation* only.
- **Dynamic Micro-Batch.** The full proposed method, combining *Dynamic Token-Budget Estimation* and *Dynamic Micro-batch Scheduling*.

This setup reflects a realistic multi-node cloud/edge deployment where a pipeline-parallel LLM is served over a relatively constrained network.

5.2. Scenarios and Evaluation Metrics

We evaluate the three configurations in two complementary scenarios:

1. **Offline scenario.** Synthetic workloads with controlled request parameters are used to examine how the methods behave under different sequence lengths and concurrency levels.
2. **Online scenario.** Requests arrive according to a Poisson process, mimicking real-time service conditions with heterogeneous input/output lengths and time-varying load.

For each run we record three time-based metrics:

- **Completion Time** – total wall-clock time from the arrival of the first request until all requests are completed. This directly reflects throughput.
- **Processing Time (Proc)** – aggregated time during which GPUs are actively executing kernels.
- **Idle Time** – cumulative time during which at least one GPU has no work to execute (i.e., pipeline bubbles).

5.2.1. Offline Scenario

In the offline scenario, we configure multiple workloads by varying the number of requests, input token lengths, and output token lengths. For each workload, we measure **Completion**, **Processing**, and **Idle** times to compare the three scheduling methods. Table 1 summarizes the results, along with the improvement of the full **Dynamic Token-Budget & Micro-batch** scheme over the **Baseline**.

Across *all* workloads, the two dynamic schedulers outperform the fixed Baseline. For moderate request sizes such as 64 / 256 / 256, the full scheme reduces **Completion** time from 130.3 s to 111.6 s (a **1.17× speed-up**) while cutting **Idle** time from 66.5 s to 47.7 s (–28%). As sequence length increases, the benefits become more pronounced: for 256 / 256 / 256, Completion decreases from 616.2 s to 381.9 s (**1.61**×) and Idle drops by 55%. The longest prompt configuration 64 / 2048 / 2048 similarly improves from 2468.7 s to 2113.0 s (1.17×) while reducing Idle by 16%.

 doi:10.20944/preprints202512.0788.v1

Table 1. Completion (**Completion**), Processing (**Proc**), and GPU-idle (**Idle**) times for each workload. The last column shows the gain of **Dynamic Token-Budget & Micro-batch** over the **Baseline**: multiplicative speed-up for **Completion** (↑ is better) and percentage reduction for **Proc** and **Idle** (↓ is better).

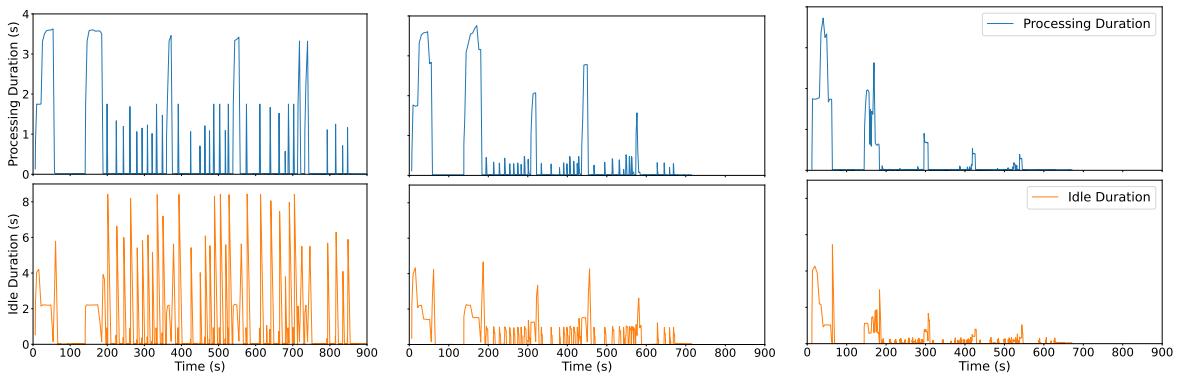
Workload (Req/In/Out)	Metric	Baseline	Only Dynamic Token-Budget	Dynamic Token-Budget & Micro-batch	Improvement
64 / 256 / 256	Completion	130.3 s	123.8 s	**111.6 s**	**1.17×**
	Proc	63.3 s	67.1 s	**62.4 s**	**1%**
	Idle	66.5 s	56.2 s	**47.7 s**	**28%**
128 / 256 / 256	Completion	290.9 s	226.4 s	**211.0 s**	**1.38×**
	Proc	112.6 s	110.8 s	**122.4 s**	**–9%**
	Idle	177.7 s	115.0 s	**88.0 s**	**50%**
256 / 256 / 256	Completion	616.2 s	441.0 s	**381.9 s**	**1.61×**
	Proc	206.3 s	197.2 s	**197.9 s**	**4%**
	Idle	409.3 s	243.2 s	**182.6 s**	**55%**
32 / 512 / 512	Completion	152.0 s	147.5 s	**139.0 s**	**1.09×**
	Proc	88.2 s	91.8 s	**87.4 s**	**1%**
	Idle	63.2 s	55.2 s	**51.1 s**	**19%**
64 / 512 / 512	Completion	319.3 s	268.7 s	**256.5 s**	**1.24×**
	Proc	158.8 s	158.4 s	**170.7 s**	**–8%**
	Idle	160.0 s	109.7 s	**85.2 s**	**47%**
128 / 512 / 512	Completion	673.5 s	499.9 s	**471.7 s**	**1.43×**
	Proc	295.0 s	266.0 s	**282.8 s**	**4%**
	Idle	377.9 s	233.4 s	**188.3 s**	**50%**
32 / 1024 / 1024	Completion	420.6 s	366.5 s	**349.6 s**	**1.20×**
	Proc	255.7 s	255.0 s	**241.6 s**	**6%**
	Idle	164.4 s	110.9 s	**107.4 s**	**35%**
64 / 1024 / 1024	Completion	891.8 s	708.8 s	**660.5 s**	**1.35×**
	Proc	475.6 s	447.5 s	**424.9 s**	**11%**
	Idle	415.7 s	260.7 s	**234.2 s**	**44%**
32 / 2048 / 2048	Completion	1262.9 s	1142.3 s	**1092.5 s**	**1.16×**
	Proc	833.1 s	782.9 s	**738.6 s**	**11%**
	Idle	429.3 s	358.8 s	**353.4 s**	**18%**
64 / 2048 / 2048	Completion	2468.7 s	2222.2 s	**2113.0 s**	**1.17×**
	Proc	1684.4 s	1567.0 s	**1451.1 s**	**14%**
	Idle	783.8 s	654.5 s	**660.4 s**	**16%**

The main driver of improvement is reduced **GPU Idle Time**. In the 64 / 1024 / 1024 and 64 / 2048 / 2048 workloads, the Baseline experiences repeated *chunking* and **preemption**, inflating Idle to 429.3 s and 415.7 s, respectively. The dynamic scheme limits preemption, reducing Idle to 353.4 s and 234.2 s. Longer sequences also reveal the impact on redundant computation: for 64 / 1024 / 1024, our scheduler cuts **Proc** from 475.6 s to 424.9 s (–11%), and for 64 / 2048 / 2048 from 1684.4 s to 1451.1 s (–14%) by preventing repeated prefill re-execution.

Overall, the proposed schedulers reduce both compute imbalance and re-execution overhead, yielding up to **1.61**× faster completion and as much as **55%** less GPU Idle Time without penalizing processing time.

To illustrate the effect more concretely, Figure 2 shows the timelines of GPU processing and idleness for the large-input workload `64 / 1024 / 1024`. The Baseline exhibits long, contiguous idle intervals, whereas the proposed methods progressively shrink idle regions and shorten overall completion time.

 doi:10.20944/preprints202512.0788.v1



(**a**) Baseline (**b**) Dynamic Token-Budget (**c**) Dynamic Token-Budget & Micro-Batch

Figure 2. Processing and idle time timelines for the `64 / 1024 / 1024` scenario.

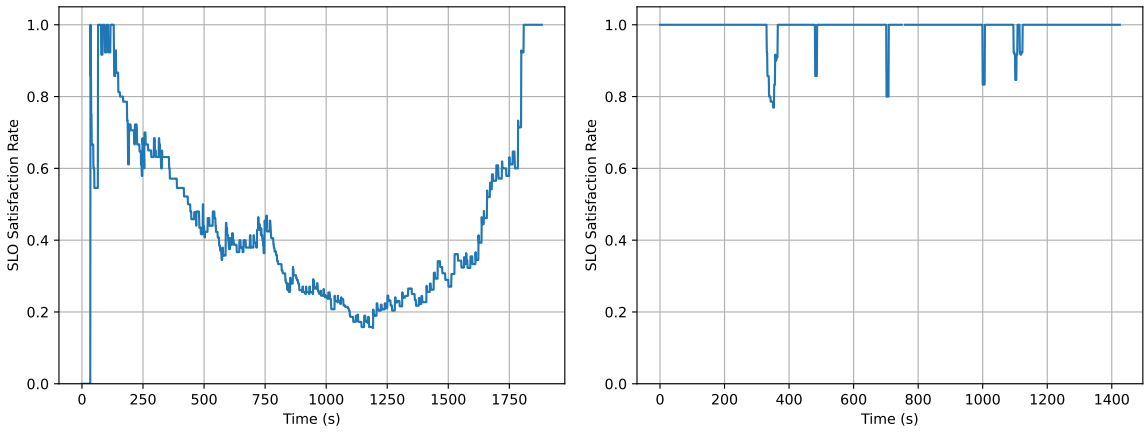
5.2.2. Online Scenario

To emulate a realistic service workload, we generate **128 requests** whose input and output lengths are drawn *uniformly at random* (input: 512–2048 tokens, output: 256–1024 tokens). Requests arrive according to a **Poisson process** with rate $\lambda = 0.1$ requests, and are dispatched in order of arrival. This setting allows us to assess how well each scheduler preserves *latency quality*—i.e., TTFT and ITL—under heterogeneous, time-varying load.

Per-request performance is measured using an **SLO satisfaction rate** based on two latency metrics: TTFT and ITL. Thresholds are set to reflect production responsiveness requirements; during each run we record the fraction of in-flight requests that satisfy both TTFT and ITL targets.

Figure 3 plots the SLO satisfaction ratio over time. Both dynamic schedulers maintain a **higher SLO ratio** than the Baseline, especially during **network bottlenecks** or **request bursts**, where the fixed scheduler becomes unstable.

- **Dynamic Token-Budget** alleviates excessive compute imbalance, shortens **GPU Idle Time**, and consequently lowers ITL while keeping TTFT within target.
- **Dynamic Micro-Batch** adapts the micro-batch count to rising network latency, sustaining the fraction of requests that meet the ITL SLO under congestion.



(**a**) Baseline (**b**) Dynamic Micro-Batch

Figure 3. SLO satisfaction rates over time for baseline (top) and proposed method (bottom).

Overall, the dynamic schemes noticeably improve SLO attainment, maintaining higher TTFT/ITL satisfaction ratios than the Baseline, particularly during congestion periods.

 doi:10.20944/preprints202512.0788.v1

5.3. Results and Analysis

The experimental results confirm that the proposed **Dynamic Token-Budget** and **Dynamic Micro-batch** techniques significantly enhance pipeline-parallel LLM inference. When combined, they reduce **GPU Idle Time** by up to **55%** and improve completion time (throughput) by up to **1.61**×. On average across all offline workloads, idle time drops by **36.2%** and throughput increases by **1.28**×.

These gains indicate that a runtime-aware model of computation and communication enables the scheduler to adapt to diverse token lengths and dynamic network delays. The improved SLO satisfaction observed in the online scenario further shows that our approach not only raises efficiency but also maintains real-time latency requirements, making it well-suited for latency-sensitive LLM services deployed in IoT–edge–cloud environments.

Finally, the two methods are *synergistic*: balancing compute load (**Dynamic Token-Budget**) and adapting to communication constraints (**Dynamic Micro-batch**) together deliver higher performance than either technique alone.

6. Discussion

We now discuss the implications of our results for IoT-scale deployments and position our throughput-first design relative to latency-oriented approaches.

6.1. Effectiveness for IoT-Scale Workloads

IoT traffic is characterized by intermittent bursts and heterogeneous request sizes. Our results confirm that token-budget estimation effectively mitigates compute imbalance caused by diverse prompt lengths, while adaptive micro-batch scheduling smooths traffic variability. Pipeline stability is significantly improved: GPU idle time is reduced, and per-stage utilization remains high even as traffic patterns shift. These properties are essential for real-time IoT services that run LLM inference in the cloud, where sudden spikes in event-triggered requests can otherwise cause severe performance degradation.

6.2. Throughput-First vs. Latency-First Optimization

Traditional inference schedulers often target token-level latency metrics, such as minimizing TTFT or per-token generation delay for individual sessions. However, IoT backends typically need to sustain a large volume of concurrent data streams and maintain SLOs at the *system* level. In this context, throughput stability—rather than best-effort latency for a single request—is the primary requirement.

Our throughput-first design supports more devices per cluster, ensures more predictable latency under high load, and improves system capacity without additional hardware. By explicitly maximizing pipeline utilization and reducing bubbles, the scheduler reduces queueing delay for many requests simultaneously. This perspective aligns naturally with IoT service requirements and distributed edge–cloud computing paradigms, where compute and network resources must be shared across large numbers of devices.

6.3. Impact of Network Bandwidth.

Our experimental cluster operates with a relatively constrained 100 Mbps inter-node bandwidth, reflecting realistic edge–cloud deployments where high-speed datacenter networking is not always available.

While limited bandwidth amplifies communication-induced pipeline stalls, our results show that the proposed scheduling framework remains effective even under such conditions, significantly reducing idle time and stabilizing throughput.

This suggests that dynamic micro-batch selection is particularly valuable when network latency is a dominant bottleneck.

 doi:10.20944/preprints202512.0788.v1

Nevertheless, future work should evaluate the framework under both higher-bandwidth datacenter interconnects and heterogeneous wide-area network settings to fully characterize its performance across diverse deployment environments.

6.4. Generalization, Limitations, and Future Work

Both Algorithm 1 and 2 are lightweight and easily integrated into existing inference engines. Their design is model-agnostic, making them applicable to a variety of LLM architectures, pipeline depths, and deployment scales, including future edge–cloud deployments and real production IoT workloads.

Nevertheless, this study has limitations. Our experiments are conducted on a homogeneous GPU cluster with relatively stable intra-cluster network latency. Although this setting is representative of many cloud deployments, real-world edge–cloud systems often exhibit heterogeneous compute capabilities and non-negligible wide-area network delays. In such environments, the interaction between pipeline scheduling, network latency, and cross-layer offloading decisions becomes more complex.

Despite these constraints, our results suggest that the proposed method already improves throughput in settings where network performance is a relative bottleneck, by shortening pipeline idle time and reducing unnecessary micro-batch stalls. A promising direction for future work is to extend the framework to heterogeneous clusters with large network delays and diverse GPU capabilities, jointly optimizing micro-batch scheduling, token budgets, and cross-node placement to maximize end-to-end throughput and SLO satisfaction.

7. Conclusions

As large language models are increasingly deployed in IoT–edge–cloud environments, maintaining both high throughput and stable latency becomes a central challenge. Bursty, heterogeneous workloads with mixed prefill- and decode-heavy requests easily amplify pipeline-stage imbalance in pipeline-parallel LLM inference, causing GPU idle time to grow and making it difficult to meet TTFT and ITL service-level objectives (SLOs) even when sufficient raw compute capacity is available. Static batching and token-budget configurations are ill-suited to such dynamic conditions, as they cannot react to real-time changes in workload composition or network behavior.

In this work, we presented a runtime-adaptive scheduling framework that combines *Dynamic Token-Budget Estimation* with *Dynamic Micro-batch Scheduling*. The proposed method uses token-budget estimation to balance prefill and decode workloads across micro-batches, and selects the number of micro-batches to minimize pipeline bubbles based on empirical compute and communication models. Crucially, we do not treat the token-budget as a fixed throughput knob; instead, we reinterpret it as a latency control parameter that can be adjusted at runtime to preserve TTFT/ITL SLOs while improving GPU utilization.

Our implementation on a four-node RTX 4070 cluster running pipeline-parallel `Llama-2-13b-chat` with `vLLM` demonstrates that the combined scheme reduces GPU idle time by up to **55%** and improves throughput (completion time) by up to **1.61**$\times$ compared with the baseline static scheduler. On average across offline workloads, idle time is reduced substantially, and throughput increases by more than 1.2$\times$. Furthermore, in an online Poisson-arrival scenario, the proposed framework maintains a higher TTFT/ITL SLO satisfaction ratio, especially during bursty periods and under network bottlenecks.

These results suggest that dynamic, workload-aware scheduling is a practical and effective way to unlock additional performance from existing LLM inference engines without modifying model architectures or kernels. Future work includes extending the framework to heterogeneous GPU clusters, incorporating richer network feedback, and jointly optimizing scheduling decisions with cross-layer offloading policies in IoT–edge–cloud systems.

Author Contributions: Conceptualization, J.A. and S.P.; methodology, J.A.; software, J.A.; validation, J.A., Y.S., and D.K.; formal analysis, J.A.; investigation, J.A.; resources, S.P.; data curation, J.A. and Y.S.; writing—original draft preparation, J.A.; writing—review and editing, J.A., D.K., and S.P.; visualization, J.A.; supervision, S.P.;

 doi:10.20944/preprints202512.0788.v1

project administration, S.P.; funding acquisition, S.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Daegu Digital Innovation Promotion Agency(DIP) OF FUNDER grant number 25DIH-32.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: No new data were created in this study. Data sharing is not applicable to this article.

Acknowledgments: This work was supported by the Digital Innovation Hub project supervised by the Daegu Digital Innovation Promotion Agency(DIP) grant funded by the Korea government(MSIT and Daegu Metropolitan City) in 2025(No.25DIH-32 / Development of a Vehicle Operation Data Recording and Trust Verification Platform Based on Micro-Blockchain Technology).

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. vLLM Contributors. vLLM: Open-source LLM Inference Library. https://github.com/vllm-project/vllm, 2025. Version v0.9.0.1, commit 5fbbfe9.
2. Hamdan, S.; Ayyash, M.; Almajali, S. Edge-Computing Architectures for Internet of Things Applications: A Survey. *Sensors* **2020**, *20*, 6441. https://doi.org/10.3390/s20226441.
3. Andriulo, F.C.; Fiore, M.; Mongiello, M.; Traversa, E.; Zizzo, V. Edge Computing and Cloud Computing for Internet of Things: A Review. *Informatics* **2024**, *11*, 71. https://doi.org/10.3390/informatics11040071.
4. Lim, J. Latency-Aware Task Scheduling for IoT Applications Based on Artificial Intelligence with Partitioning in Small-Scale Fog Computing Environments. *Sensors* **2022**, *22*, 7326. https://doi.org/10.3390/s22197326.
5. Eang, C.; Ros, S.; Kang, S.; Song, I.; Tam, P.; Math, S.; Kim, S. Offloading Decision and Resource Allocation in Mobile Edge Computing for Cost and Latency Efficiencies in Real-Time IoT. *Electronics* **2024**, *13*, 1218. https://doi.org/10.3390/electronics13071218.
6. Saeik, F.; Avgeris, M.; Spatharakis, D.; Santi, N.; Dechouniotis, D.; Violos, J.; Leivadeas, A.; Athanasopoulos, N.; Mitton, N.; Papavassiliou, S. Task Offloading in Edge and Cloud Computing: A Survey on Mathematical, Artificial Intelligence and Control Theory Solutions. *Computer Networks* **2021**, *195*, 108177. https://doi.org/10.1016/j.comnet.2021.108177.
7. Kwon, W.; Li, Z.; Zhuang, S.; Sheng, Y.; Zheng, L.; Yu, C.H.; Gonzalez, J.; Zhang, H.; Stoica, I. Efficient Memory Management for Large Language Model Serving with PagedAttention. In Proceedings of the Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 2023). ACM, 2023, pp. 611–626. https://doi.org/10.1145/3600006.3613165.
8. Yu, G.; Jeong, J.S.; Kim, G.; Kim, S.; Chun, B. Orca: A Distributed Serving System for Transformer-Based Generative Models. In Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022). USENIX Association, 2022, pp. 521–538.
9. Agrawal, A.; Panwar, A.; Mohan, J.; Kwatra, N.; Gulavani, B.S.; Ramjee, R. SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills. *arXiv* **2023**, *abs/2308.16369*, [2308.16369]. https://doi.org/10.48550/ARXIV.2308.16369.
10. Agrawal, A.; Kedia, N.; Panwar, A.; Mohan, J.; Kwatra, N.; Gulavani, B.S.; Tumanov, A.; Ramjee, R. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2024). USENIX Association, 2024, pp. 117–134.
11. Zhong, Y.; Liu, S.; Chen, J.; Hu, J.; Zhu, Y.; Liu, X.; Jin, X.; Zhang, H. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2024). USENIX Association, 2024, pp. 193–210.
12. Patel, P.; Choukse, E.; Zhang, C.; Shah, A.; Goiri, Í.; Maleki, S.; Bianchini, R. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In Proceedings of the 51st ACM/IEEE International Symposium on Computer Architecture (ISCA 2024). IEEE, 2024, pp. 118–132. https://doi.org/10.1109/ISCA59077.2024.00019.

 doi:10.20944/preprints202512.0788.v1

13. Hu, C.; Huang, H.; Xu, L.; Chen, X.; Xu, J.; Chen, S.; Feng, H.; Wang, C.; Wang, S.; Bao, Y.; et al. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. *arXiv* **2024**, *abs/2401.11181*, [2401.11181]. https://doi.org/10.48550/ARXIV.2401.11181.
14. Jiang, Y.; Yan, R.; Yao, X.; Zhou, Y.; Chen, B.; Yuan, B. HexGen: Generative Inference of Large Language Model over Heterogeneous Environment. In Proceedings of the Proceedings of the 41st International Conference on Machine Learning (ICML 2024). OpenReview.net, 2024.
15. Jiang, Y.; Yan, R.; Yuan, B. HexGen-2: Disaggregated Generative Inference of LLMs in Heterogeneous Environment. In Proceedings of the The Thirteenth International Conference on Learning Representations (ICLR 2025). OpenReview.net, 2025.
16. Huang, Y.; Cheng, Y.; Bapna, A.; Firat, O.; Chen, D.; Chen, M.X.; Lee, H.; Ngiam, J.; Le, Q.V.; Wu, Y.; et al. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In Proceedings of the Advances in Neural Information Processing Systems 32 (NeurIPS 2019), 2019, pp. 103–112.
17. Zhang, Z.; Sheng, Y.; Zhou, T.; Chen, T.; Zheng, L.; Cai, R.; Song, Z.; Tian, Y.; Ré, C.; Barrett, C.W.; et al. H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. In Proceedings of the Advances in Neural Information Processing Systems 36 (NeurIPS 2023), 2023.
18. Xiao, G.; Tian, Y.; Chen, B.; Han, S.; Lewis, M. Efficient Streaming Language Models with Attention Sinks. In Proceedings of the The Twelfth International Conference on Learning Representations (ICLR 2024). OpenReview.net, 2024.
19. Dao, T.; Fu, D.Y.; Ermon, S.; Rudra, A.; Ré, C. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In Proceedings of the Advances in Neural Information Processing Systems 35 (NeurIPS 2022), 2022.
20. Dao, T. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In Proceedings of the The Twelfth International Conference on Learning Representations (ICLR 2024). OpenReview.net, 2024.
21. Shah, J.; Bikshandi, G.; Zhang, Y.; Thakkar, V.; Ramani, P.; Dao, T. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. In Proceedings of the Advances in Neural Information Processing Systems 38 (NeurIPS 2024), 2024.
22. Kamath, A.K.; Prabhu, R.; Mohan, J.; Peter, S.; Ramjee, R.; Panwar, A. POD-Attention: Unlocking Full Prefill-Decode Overlap for Faster LLM Inference. In Proceedings of the Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2025). ACM, 2025, pp. 897–912. https://doi.org/10.1145/3676641.3715996.
23. Oh, H.; Kim, K.; Kim, J.; Kim, S.; Lee, J.; Chang, D.; Seo, J. ExeGPT: Constraint-Aware Resource Scheduling for LLM Inference. In Proceedings of the Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2024). ACM, 2024, pp. 369–384. https://doi.org/10.1145/3620665.3640383.
24. Holmes, C.; Tanaka, M.; Wyatt, M.; Awan, A.A.; Rasley, J.; Rajbhandari, S.; Aminabadi, R.Y.; Qin, H.; Bakhtiari, A.; Kurilenko, L.; et al. DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference. *arXiv* **2024**, *abs/2401.08671*, [2401.08671]. https://doi.org/10.48550/ARXIV.2401.08671.
25. Chen, Z.; May, A.; Svirschevski, R.; Huang, Y.; Ryabinin, M.; Jia, Z.; Chen, B. Sequoia: Scalable, Robust, and Hardware-aware Speculative Decoding. *arXiv* **2024**, *abs/2402.12374*, [2402.12374]. https://doi.org/10.48550/ARXIV.2402.12374.