ACCELGEN: Heterogeneous SLO-Guaranteed High-Throughput LLM Inference Serving for Diverse Applications

Haiying Shen and Tanmoy Sen
University of Virginia

Abstract

In this paper, we consider a mixed-prompt scenario for a large language model (LLM) inference serving system that supports diverse applications with both short prompts and long prompts and heterogeneous SLOs for iteration time. To improve throughput when handling long prompts, previous research introduces a chunking method, but has not addressed heterogeneous SLOs. To address the limitation, we propose ACCELGEN, a high-throughput LLM inference serving system with heterogeneous SLO guarantees for diverse applications. ACCELGEN introduces three core components: (1) SLO-guaranteed dynamic chunking, which dynamically adjusts chunk sizes to maximize GPU compute utilization while meeting iteration-level SLOs; (2) Iteration-level SLO-based task prioritization, which prioritizes tight-SLO requests and batches requests with similar SLOs; (3) Multi-resource-aware batching, which selects queued requests to maximize the utilizations of both GPU compute resource and key-value cache (KVC). Trace-driven real experiments demonstrate that ACCELGEN achieves 1.42-11.21× higher throughput, 1.43-13.71× higher goodput, 37-90% higher SLO attainment, and 1.61-12.22× lower response latency compared to the state-of-the-art approaches. It achieves performance near the *Oracle*, which optimally maximizes goodput.

1 Introduction

Transformer-based generative large-language models (LLMs) have garnered considerable attention in Natural Language Processing (NLP) applications such as text generation [1], text summarization [2], code generation [3], and chatbot [4]. In the model inference, each request begins with a user-entered prompt, comprising a sequence of tokens. The model processes this prompt in the first iteration to generate the initial token, and subsequently generates tokens in an autoregressive manner [5] in each iteration until the entire response text is produced. Requests are processed in batches, including the prompt processing (PP) tasks and the token generation (TG) tasks.

Given the diverse applications running on an LLM inference serving system, prompt lengths can vary significantly, ranging from just a few tokens for chat applications to 4K-100K tokens for more extensive content like electronic books used in text summarization, code generation, and legal document analysis [6]. Additionally, user experience is critical; users may have different iteration time service-level-objectives (SLOs) in online applications based on their reading speed and application requirements, with a typical reading speed around 0.1875 seconds per token [7–9]. Offline applications, on the other hand, may impose job completion time (JCT) SLOs. Thus, in this paper, we consider a mixed-prompt scenario, consisting of short prompts and long prompts (containing ≥4K tokens) and heterogeneous SLOs.

To improve throughput when handling long prompts, previous research introduces chunking [10–13]. The approaches select chunks or requests from the waiting queue until the target forward size (or token budget) – defined as the desired total number of tokens in a batch – is reached, to fully utilize the GPU compute resource. However, our trace-driven measurement analysis (§3) reveals that these systems fail to meet heterogeneous iteration-level SLOs, potentially degrade user experience, and fall short of maximizing GPU compute and key-value cache (KVC) utilization. Specifically, our findings indicate that:

(1) Heterogeneous iteration-level SLOs cannot be met in current LLM systems, making it necessary to account for SLO diversity in scheduling.

(2) Focusing solely on maximizing GPU compute resources does not guarantee maximum throughput for a workload. It is essential to jointly maximize both GPU compute utilization and KVC utilization at each iteration to improve throughput.

(3) We should limit the number of concurrently running long-prompt requests to free up KVC resources more quickly to improve throughput.

(4) The first-come-first-serve (FCFS) policy used in current LLM systems [12, 14] underutilizes GPU resources when

meeting iteration-level SLOs, but batching requests with similar SLOs enhances throughput.

(5) The varying available GPU compute resource and unallocated KVC space after each iteration, coupled with the diverse GPU and KVC demands of different prompts, provide an opportunity to identify prompts or prompt chunks to be added to the batch to maximize both GPU compute and KVC utilizations, thus improving throughput.

Building on these key insights, based on the chunking method of FastGen, we propose ACCELGEN, a high-throughput LLM inference serving system designed to meet heterogeneous SLOs while maximizing throughput for diverse applications. ACCELGEN incorporates the following methods to achieve the goal:

(1) **SLO-guaranteed dynamic chunking.** ACCELGEN determines the token budget and batches requests with similar SLOs to maximize GPU compute utilization as much as possible while satisfying the iteration-level SLOs of the requests in the batch. In addition, it limits concurrently running long-prompt requests to improve throughput.

(2) **Iteration-level SLO-based task prioritization.** ACCELGEN enables users to specify the SLOs for Time-To-First-Token (TTFT) in PP and for Time-Between-Tokens (TBT) in TG of a job, or specify the JCT SLO for a job. It converts a JCT SLO into iteration-level SLOs. Based on iteration-level SLOs, it orders the waiting requests to enhance SLO compliance and facilitate batching requests with similar SLOs to enhance throughput.

(3) **Multi-resource-aware batching.** After each iteration, ACCELGEN selects requests and adjusts prompt chunk lengths from the prioritized waiting queue to maximize both GPU compute utilization and KVC utilization in the subsequent iteration.

Our tace-driven real experiments show that ACCELGEN achieves 1.42-11.21× higher throughput, 1.43-13.71× higher goodput, 37-90% higher SLO attainment, and 1.61-12.22× lower response latency compared to the state-of-the-art approaches. This is the first paper on heterogeneous iteration-level SLOs, a topic of interest to industry for practical applications.

2 Background and Theoretical Founcation

2.1 Background

As a representative of the transformer model, we illustrate the OPT structure in Figure 1. OPT is a decoder-only transformer model and its layers fall into two categories: 1) forward computation layers (FCL), and 2) attention layer. The attention layer takes three input components: Query ($\mathbf{Q}$), Key ($\mathbf{K}$), and Value ($\mathbf{V}$). These components are obtained by projecting the embeddings ($\mathbf{E}$) using three learned matrices $\mathbf{W_q}$, $\mathbf{W_k}$, and $\mathbf{W_v}$ in the transformer:

$$\mathbf{Q} = \mathbf{W_q E}, \ \mathbf{K} = \mathbf{W_K E}, \ \mathbf{V} = \mathbf{W_v E}$$
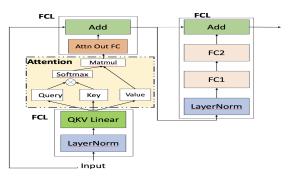

Figure 1: Transformer layer of OPT.

At the layer level, each matrix multiplication is a fully connected layer with input and output neurons. Then its number of operations equals:

$$2 \times S_f \times H^2, \tag{1}$$

where S_f is forward size and H is the model dimension.

The attention module includes the dot products of the query tokens in $\mathbf{Q}$ with all the keys of previous tokens in $\mathbf{K}$ to measure the similarity of previous tokens from the new token's perspective: $Similarity(\mathbf{Q}, \mathbf{K}) = \mathbf{QK}^T$. The attention operation produces the output:

$$Self-attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax(\frac{\mathbf{QK}^T}{\sqrt{d_k}})\mathbf{V}, \tag{2}$$

where d_k is the dimension of the $\mathbf{K}$ or $\mathbf{Q}$. The number of operations at the layer level equals:
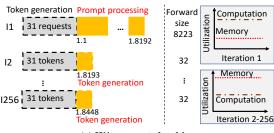
$$4 \times S_f^2 \times H. \tag{3}$$

The FC2 layer outputs logits, which are passed through the Add to generate the probabilities of words in the vocabulary. In the Transformer model, the above operations are executed by each attention head in parallel. Finally, the results from all attention heads are concatenated.
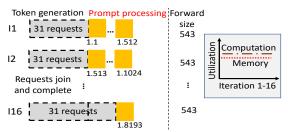
2.2 Theoretical Foundation

Throughput. For the Transformer layer, the primary throughput contributors are the FCL layers (QKV, Atten Out, FC1, FC2) and Attention. Neglecting the negligible operations of other layers like Add and LayerTransform, an FCL layer's number operations is given by Equation (1). Since each layer is based on a multiple of the model dimension, the total impact of each layer is $(3(QKV) + 1(Attenout) + 4(FC1) + 4(FC2)) = 12 \times 2 \times H^2$. Based on Equations (1) and (3), the total operations for a single transformer layer (assuming only one sequence) is $(24 \times S_f \times H^2 + 4 \times H \times S_f^2)$, which can be written as [15]:

$$24 \times S_f \times H^2 \times (1 + \frac{S_f}{(6 \times H)}). \tag{4}$$

The first addend comes from the FCL operations, and the second number is derived from the attention operations. For short sequence lengths ($S_f << 6 \times H$), it is reasonable to approximate the total operations by considering only this first addend.

| (a) W/o prompt chunking. | (b) W/ prompt chunking. |

Figure 2: Illustration of chunking a long prompt.

KVC allocation. In an iteration, the KV values of all the tokens from a step are calculated and stored in the KVC. In the subsequent iteration, only the KV tensors of the newly generated token need computation, while others are loaded from the KVC. The byte storage per token equals [16]:

$$2 \times 2 \times \text{\# of transformer layers} \times H. \quad (5)$$

A request i's sequence length (denoted by S_l^i) is the sum of its prompt length and the number of generated tokens so far. The vLLM approach [12] mitigates the KVC bottleneck problem in ORCA by using a block-based approach. Assume the block size is $b = 128$. When a token is generated, a block is created for this token and the next 127 tokens. Next, when the 129th token is generated, another 128-token block is created. The allocated KVC space for a batch **B** in ORCA equals: $|\mathbf{B}| \cdot S_{l_{max}}$, in which $S_{l_{max}}$ is the maximum sequence length and $|\mathbf{B}|$ is batch size. In vLLM, it equals: $\sum_{i \in \mathbf{B}} \lceil \frac{S_l^i}{b} \rceil \cdot b$. ACCELGEN is built based on the vLLM system.

2.3 Chunking on Long Prompts

Let's see an example to undertand the detrimental effects of long prompts and benefits of chunking. Figure 2 (a) illustrates the processing of long-prompt requests for an 8192-token prompt with 256 generation steps. In the figure, $x.y$ means the xth prompt's y^{th} token. The batch size is 32. In the batch, one request is PP, and the other 31 requests are either short PP or TG steps. The long prompt processing task consumes a significant amount of the GPU resource and takes a long time to complete, as per Equation (4). Consequently, the iteration time is prolonged, delaying other requests in the batch. Also, after the PP task becomes a TG task, each iteration has 32 forward size, leading to significant underutilization of GPU. Figure 2b illustrates the scenario where we divide the long prompt into multiple 512-token chunks. The first batch takes around $\times 16$ less time to process. The remaining chunks are added to the batches in subsequent iterations. The forward size of each batch is 543. Therefore, in each iteration, the GPU computation resource is more fully utilized compared to the no-chunking case. Chunking distributes the GPU load across 256 iterations, reducing iteration time for long prompts and increasing throughput.

3 Experiment Analysis

In this section, we experimentally analyze the performance of existing LLM systems. Our observations serve as motivation for our work and provide insights into the design of ACCELGEN.

3.1 Experiment Settings

Model settings. We ran the OPT-13B model [1] on one GPU, and the OPT-175B [1] model on eight GPUs, configuring the pipeline and tensor parallelism degrees to 8 and 2, respectively, as in [10]. Both used fp16-formatted model parameters and intermediate activations.

Machine settings. We conducted our experiments on an AWS p4d.24xlarge instance, equipped with 8 NVIDIA A100 GPUs, with each GPU having 80GB of memory. The GPUs are connected with a 600 GB/s NVSwitch.

LLM systems. We used the source code of Sarathi-Serve [14], FastGen [11] and vLLM [12], and implemented ORCA [5] by ourselves using FasterTransformer since its source code is not available. As in [12], we set the block size to 32 tokens. As in [5], we set the batch size to 8 and set the maximum sequence length to 8K in ORCA. We set KVC size to 12GB in the OPT-13B case and to 33.75GB in each GPU in the OPT-175B case.

Datasets. To create a mix-prompt scenario, we combined the Alpaca [17], ShareGPT [18] and BookCorpus [6] traces. Alpaca and ShareGPT have 52K and 90K requests, with up to 500 and 2K token prompt lengths, respectively. BookCorpus has 11K unpublished books, with longer prompt lengths up to 100K tokens. We used random selection with replacement to create a dataset of 20K requests from the traces, ensuring that 35% of the requests originated from BookCorpus.

Request settings. As in [12], each request can generate up to 2048 tokens. The requests arrive following a Poisson distribution with an arrival rate of 8 requests/s [5].

The resource utilization profiling was performed using the gpustat library [19] with a 0.1s time interval. *Token budget* (denoted as S_b) is referred to as the target forward size. In our experiment, as the forward size (S_f) increases, throughput improves and nearly saturates at a forward size of 768 and 1280 for OPT-13B and OPT-175B, respectively [20], referred to as *pivot forward size* (denoted by S_{pf}), S_{pf} is used as the token budget unless otherwise specified in our experiments.
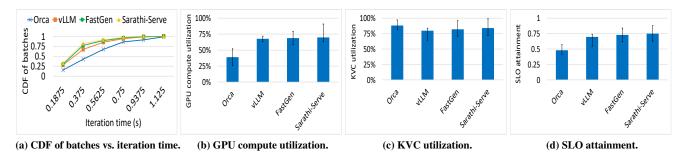
(a) CDF of batches vs. iteration time.   (b) GPU compute utilization.   (c) KVC utilization.   (d) SLO attainment.

Figure 3: Performance of different LLM systems for OPT-13B.

(a) CDF of batches vs. iteration time.   (b) GPU compute utilization.   (c) KVC utilization.   (d) SLO attainment.
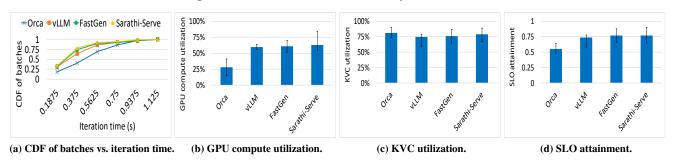
Figure 4: Performance of different LLM systems for OPT-175B.

This allows near-full GPU compute utilization without substantially increasing batch processing latency. In this paper, the figures with error bars report the average, the 5th and the 95th percentiles of the result values.

3.2 Measurement Results

Resource Utilization and SLO Performance. Figures 3a and 4a depict the cumulative distribution function (CDF) of batches (or iterations) versus iteration time for OPT-13B and OPT-175B, respectively. Only 16%-34% of batches on average across the two models in ORCA, vLLM, FastGen and Sarathi-Serve complete within 0.1875s, indicating that many iterations exceed the normal reading speed.

Figures 3b and 4b show the average GPU compute utilization per iteration, calculated as the ratio of the actual forward size to the token budget. ORCA, vLLM, FastGen, Sarathi-Serve achieve GPU compute utilization of 33%, 61%, 62% and 64%, respectively, on average for both models. KVC constraints and long prompt chunks hinder the full utilization of the remaining token budget. That is, when the KVC cannot host a chunk, the chunk won't be added to the batch. The results indicate significant potential for improvement in GPU compute utilization.

Figures 3c and 4c show the average KVC utilization per iteration, defined as the ratio of allocated KVC space to total KVC space in tokens. ORCA, vLLM, FastGen and Sarathi-Serve achieve KVC utilization of 88%, 79%, 80% and 83%, respectively, on average for both models. Although the systems add requests to the batch until the batch size is reached in ORCA or the KVC space is fully allocated, residual KVC space often remains that cannot accommodate the next request or chunk, resulting in incomplete KVC utilization.

A prompt or prompt chunk that fully utilizes both the remaining GPU compute and KVC resources is preferable over one that only fully utilizes the remaining GPU compute resource. This is because if the former is not selected for execution, it may later be unable to run due to insufficient KVC resources. Fully utilizing both resources better achieves the goal of maximizing throughput of a workload. However, the systems' exclusive focus on GPU compute utilization, combined with its FIFO policy, prevents it from prioritizing such optimal choices. Therefore, they fail to maximize both GPU compute and KVC utilizations.

We then measure the performance of the systems in meeting the SLO requirements. We grouped the prompts within the length range of 512 tokens, and measured the average prompt processing latency for each group using the Triton Inference Server with the Faster Transformer backend. As [21] that uses SLO-scale to determine SLO, we set a TTFT SLO as the product of the average prompt processing latency of its group and an SLO-scale [21] randomly selected from the range of [0.5,1.5]. The TBT SLO was set to 0.1875s × SLO scale, which was randomly selected from {0.25, 0.5, 1, 2} for each request. Figures 3d and 4d show the iteration SLO attainment, defined as the percentage of iterations that meet their iteration-level SLOs. We see that the systems fail to achieve high SLO attainment since they do not address heterogeneous SLOs.

We then set the system-wide SLO in Sarathi-Serve from 0.2s to 1s, increasing by 0.2s at each step as in [14]. Figures 5a and 5b show the iteration-SLO attainment and GPU compute
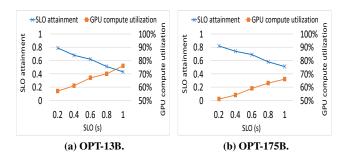
(a) OPT-13B.   **(b) OPT-175B.**

Figure 5: Performance of meeting SLOs in Sarathi-Serve.

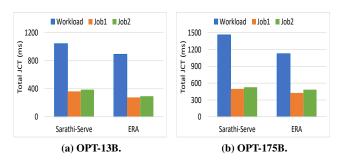**(a) OPT-13B.**   **(b) OPT-175B.**

Figure 6: Exclusive Request Allocation (ERA) vs. Sarathi-Serve.

utilization versus the system-wide SLO in Sarathi-Serve. A stringent system-wide SLO, while ensuring compliance with heterogeneous SLOs, reduces throughput. In contrast, a more relaxed system-wide SLO avoids throughput reduction but results in lower SLO attainment.

> **Observation 1.** Heterogeneous SLOs cannot be met in current LLM systems, making it necessary to account for SLO diversity in scheduling.

> **Observation 2.** An FCFS approach focused solely on maximizing GPU compute utilization underutilizes KVC resources, reducing throughput. Jointly optimizing GPU compute and KVC utilization per iteration is essential to improve throughput.

Order of Chunks in KVC Allocation. Sarathi-Serve allocates the token budget across chunks of different long prompts, increasing concurrent long-prompt requests. We argue that reducing concurrency can alleviate KVC limitations by fully allocating the token budget to one request before moving to the next, allowing for quicker KVC release. We call this method *Exclusive Request Allocation (ERA)*. To validate this, we compared ERA and Sarathi-Serve using two long prompts (10,214 and 10,252 tokens) and 20 short prompts (128-384 tokens). Figures 6a and 6b show that ERA reduces the JCT for both long-prompt jobs and the overall workload via minimizing concurrent KVC occupancy by long-prompt requests.
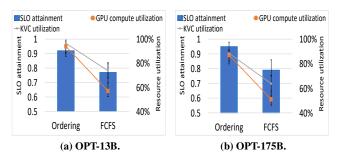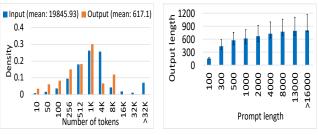

(a) OPT-13B.   **(b) OPT-175B.**

Figure 7: Importance of SLO-based ordering.

(a) Input and output length distribution.   **(b) Output length vs. prompt length.**

Figure 8: Features of the requests.

> **Observation 3.** Picking up chunks from different long-prompts increases their KVC occupying time and thus reduces throughput (reqs/s) compared to ERA.

Tradeoff Between Throughput and Heterogeneous SLO Compliance. Figure 7 compares iteration SLO attainment, GPU compute utilization, and KVC utilization in Sarathi-Serve using FIFO and request ordering based on the remaining time to their SLOs. When batching requests, we ensure the iteration time doesn't exceed the most stringent SLO in the batch, with iteration time estimated from profiled data. Compared to FIFO, the ordering method achieves 37% and 34% higher GPU compute and KVC utilization, respectively, and 16% higher SLO attainment. In FIFO, requests with large SLO variance can end up in the same batch, where stringent-SLO requests reduce batch size though loose-SLO requests allow for larger batch sizes. In contrast, the SLO-based ordering policy groups requests with similar SLOs, improving resource utilization ad throughput.

> **Observation 4.** The FCFS policy underutilizes GPU resources when meeting iteration time SLOs, but batching requests with similar SLOs enhances throughput.

Different Resource Demands of Requests. Figure 8a shows the density of the lengths of inputs and outputs, representing the probability of input or output lengths falling within specific ranges. We see that 34% of prompts fall within [10, 512], 52% within [512, 4K], and 14% exceed 4K. This highlights the diversity in prompt lengths and, consequently, in GPU compute and KVC resource demands. For outputs, 51% are in the [10, 512] range, 36% in [512, 4K], and 13% exceed 4K.

The varying lengths of outputs contribute to different KVC demands for distinct requests. Figure 8b shows the output length corresponding to each input length range. We see that the input and output lengths vary across different requests.

> **Observation 5.** The varying availability of GPU compute and KVC resources (Figures 3b, 4b, 3c and 4c) after each iteration, coupled with the diverse GPU compute and KVC demands of different prompts (Figure 8), provide an opportunity to identify prompts or prompt chunks to be added to the batch to maximize both GPU compute and KVC utilizations, thus improving throughput.

4 System Design of ACCELGEN

4.1 Overview

Based on the Observations (Os) from §3, we propose ACCEL-GEN. ACCELGEN consists of the following three methods:

(1) **SLO-guaranteed dynamic chunking (§4.2)**. Based on O1 and O3, it determines the target forward size (i.e., token budget) and batches requests with similar SLOs to maximize GPU compute utilization as much as possible while satisfying the iteration-level SLOs of the requests in the batch. In addition, it limits concurrently running long-prompt requests to improve throughput.

(2) **Iteration-level SLO-based task prioritization (§4.3)**. Based on O4, it orders requests by their iteration-level SLOs and groups those with similar SLOs for scheduling in each iteration.

(3) **Multi-resource-aware batching (§4.4)**. Based on O2, it concurrently considers both GPU compute and KVC resources in selecting prompts or creating prompt chunks based on their demands and available resources to maximize both GPU compute utilization and KVC utilization in the subsequent iteration.

Figure 9 shows the overview of ACCELGEN. In the figure, each block in the waiting queue is a token, and the tokens of the same color constitute a request. Method (2) is executed to order the requests based on their SLOs for each newly arrived request or when a portion of a long prompt is added to the batch (①). After an iteration finishes, the generated tokens of uncompleted requests (i.e., the rightmost three blocks) are returned to the scheduler (②). The scheduler then executes Methods (1) and (3) to select requests to maximize the GPU compute and KVC resources (③). Method (1) selects tokens sequentially from the waiting queue to fully utilize the GPU compute resource while meeting the requests' SLOs (③.1). Method (3) further simultaneously considers the KVC resources when selecting requests or creating prompt chunks, enabling the new batch to fully utilize both the GPU compute and KVC resources (③.2). In this example, green and blue tokens, and 502 tokens from the 6k long prompt shown in red are selected. Finally, the newly formed batch is sent to the execution engine (④).
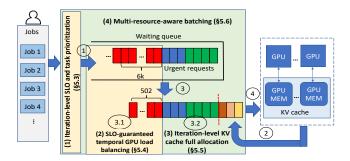

Figure 9: System architecture of ACCELGEN.

4.2 SLO-guaranteed Dynamic Chunking

Based on O1, we propose this method to maximize GPU compute utilization as much as possible while satisfying the heterogeneous SLOs of different requests. To achieve this objective, the method incorporates *token budget determination*, *limiting the number of concurrently running long-prompt requests* (O3), and *prompt selection with dynamic chunking*, as detailed below.

Token budget determination. For simplicity, unless otherwise specified, SLOs refer to iteration-level SLOs. The iteration time depends on the forward size. Therefore, we need to determine the token budget (S_b) for a batch to ensure high GPU compute utilization while satisfying the SLOs of the requests in the batch. Based on the GPU capacity of a server, we first find the pivot forward size, which saturates the GPU capacity. If the total number of operations for generating one token is x FLOPS (based on Equation (4)) and the maximum allowable FLOP/s by a GPU hardware is X, then the maximum number of processed tokens at a time equals X/x per second. GPU saturation throughput is usually lower than X [11], so we measure the pivot forward size (S_{pf}) by varying the forward size as described in §3, and measure the corresponding batch execution time, denoted by $\bar{T}_{pf}$. We can also calculate it by dividing the number of operations for S_{pf} by the GPU saturated throughput. In practice, the S_{pf} is model-specific and based on the allowable number of FLOP/s on the GPU. We can calculate S_{pf} for different GPUs belonging to the same GPU family using the non-linear regression method [22]. The model takes the x and X of the new GPU as inputs.

However, the inference time of a batch needs to satisfy the minimum SLO of the requests in a batch (SLO_{min}). Therefore, we determine the token budget by:

$$S_b = S_{pf} \cdot \frac{SLO_{min}}{\bar{T}_{pf}}. \tag{6}$$

If the scheduler uses FIFO, adding queued requests to the batch may change SLO_{min}, requiring an adjustment to S_b and the currently formed batch. Additionally, some requests in the batch may have SLOs significantly higher than SLO_{min}, resulting in their completion earlier than necessary and lower GPU compute utilization as loose-SLO requests can have a large batch. This is indicated in O4. ACCELGEN's *iteration-*

level SLO-based task prioritization method (§4.3) helps avoid the problem. It orders requests from the most stringent SLOs to the least stringent SLOs. As a result, the requests with similar SLOs will be together in the waiting queue. Since the first queued request has the most stringent SLO, we do not need to adjust S_b as more requests are added to the batch.

Limiting running long-prompt requests. When there is token budget, Sarathi-Serve picks up one chunk for the existing requests in the batch, and then picks up the chunks from a new request as much as possible. However, this may lead to the case that multiple long-prompt requests occupy KVC simultaneously, leaving smaller KVC for processing requests and increasing JCT (O3). To address this issue, ACCELGEN uses ERA, ensuring that processing of chunks from a new long prompt does not begin until all currently served long-prompt requests are completed. One might argue that this chunk selection strategy could limit the processing opportunities for waiting requests. However, in ACCELGEN, this is not an issue, as the system aims to meet the heterogeneous SLOs of all requests and selects requests that must run in the next iteration to satisfy their SLOs.

Prompt selection with dynamic chunking. After each iteration, ACCELGEN selects requests in the ordered waiting queue in sequence, aiming to reach S_b. The static chunking cannot achieve the goal of exactly reaching S_b since the number of uncompleted tasks in the batch (S_B) varies after an iteration, and hence cannot maximize the throughput as indicated in O1. To address this problem, we propose a dynamic chunking method. In this approach, when a batch completes an iteration, the ACCELGEN scheduler calculates the new S_b based on Equation (6). Subsequently, it sequentially selects tokens from the waiting queue until $S_f = S_b$, regardless of their associated requests. These selected tokens can be from one request or from different requests. If the last picked token is in the middle of a prompt, the prompt is chunked after that token. In the next iteration, the selection starts from the next token in this prompt. As a result, after all the tokens of the prompt are picked, this prompt is automatically chunked into partitions to be processed in different iterations.

The generation of a new token depends only on the logits of the preceding token. Consequently, when processing chunks, the logits operations for the entire prompt except the last token can be skipped. Consequently, only the last token of the final chunk is needed for calculating the logit value for generating the new token of the prompt.

For large models such as OPT-135B, model parallelism becomes necessary. However, adopting model parallelism introduces a tradeoff between peak compute throughput per GPU (which decreases with model parallelism) and forward size capacity (which increases). Consequently, the pivot forward size exceeds that of running the model on a single GPU.

4.3 Iteration-level SLO-based Task Prioritization

An inference engine may receive both latency-sensitive online requests like chat, and throughput-oriented offline requests like article writing. For latency-sensitive requests, users typically expect a smooth and rapid token generation based on their reading speeds and application requirements, while for throughput-oriented requests, they prioritize receiving the full response within a specific time frame. Sarathi-serve's single system-wide SLO setting fails to satisfy heterogeneous SLOs. ACCELGEN aims to meet the heterogeneous SLO requirements. It offers users the flexibility to define the iteration SLO for TTFT and for each TBT (with normal reading speed of 0.1875s/token as default) for online requests, and the JCT SLO for offline requests. In the following, we first demonstrate how ACCELGEN manages the SLOs of online applications, followed by the JCT SLOs of offline applications.

SLOs of online applications. A request can be a long prompt, a short prompt, a preempted TG task or a returned TG task. The SLO of a request defines the time period allowed to generate a token, beginning at time (t_0) when the request enters the queue – whether from a user or a preemption, or returns from a previous iteration. A request's remaining time is the period within which it must execute to meet its SLO. Requests in the waiting queue are ordered by ascending remaining time to prioritize requests with shorter remaining times.

Now, let's see how to calculate a request's remaining time. Recall that ACCELGEN may chunk a long prompt (details are in §4.2), so a long prompt completes pro-

| Waiting time | Remaining time | Execution time |

SLO

Figure 10: Remaining time to execute to satisfy the iteration-level SLO or the JCT SLO.

cessing only after the last chunk is processed. For a request with remaining token length S_r to be processed, the number of its remaining chunks to be processed can be estimated by $N_{ck} \approx \lceil S_r/L_c \rceil$, where L_c is the average chunk length. Note that for a TG task, $N_{ck} = 1$. The iteration time of a request in a batch is the batch execution time. We use T_{max} to denote the maximum batch execution time. Then, a request's remaining execution time can be estimated by $T_e \approx N_{ck} \cdot T_{max}$. As shown in Figure 10, the remaining time for a request to meet its SLO equals: $T_r = SLO - T_w - T_e$, where T_w is the waiting time since a new or preempted request entered the waiting queue and it equals to 0 for a returned TG task. When a request enters the waiting queue or a portion of a long prompt is added to the batch, its T_r is calculated, and its position in the queue is adjusted to maintain ascending T_r order among requests.

SLOs of offline applications. Users may specify a JCT SLO (denoted as SLO_{JCT}) for their requests in offline applications. Although JCT SLO is not our primary focus, we ensure that ACCELGEN can also accommodate it. Unlike the iteration-

level SLO, the JCT SLO applies to the entire job, encompassing prompt processing and all token generations. Consequently, ACCELGEN must estimate the total number of tokens to be generated (denoted as S_g), potentially using methods like that in [23]. Accurately predicting token generation remains a challenging research problem, though it is beyond the scope of this paper. Moreover, JCT SLOs for offline applications typically do not require strict adherence, so an approximate prediction should be sufficient.

A TG task may experience preemption due to KVC overflow, resulting in a preemption period before resuming execution. Therefore, the maximum time for a single token generation can be estimated as $(T_{max} + P_{max} \cdot P)$, where P_{max} represents the maximum duration of a preemption, and P represents the probability of a request being preempted after a token generation process. When such a prompt enters the waiting queue, its JCT is then calculated as $T_e \approx N_{ck} \cdot T_{max} + S_g \cdot (T_{max} + P_{max} \cdot P)$, where N_{ck} actually is the estimated total number of chunks. The total remaining waiting time is estimated as $SLO_{JCT} - T_e$. If this remaining waiting time is evenly distributed across all iterations of processing this prompt, the allowed waiting time per iteration is given by $T_r = (SLO_{JCT} - T_e)/(N_{ck} + S_g)$. For each iteration, the request's waiting time is initally set to T_r. However, this is not a hard deadline since the objective is to ensure that the JCT does not exceed the JCT SLO.

Given that the JCT SLO applies to the entire job, any over-utilized or under-utilized waiting time is propagated to subsequent iterations to maintain the total allowable waiting time. If a request completes d time units later than its allowed waiting time T_r, the allowed waiting time for the next iteration becomes $T_r = T_r - d$. If the next iteration actually runs for $T_r + d_1$, the allowed waiting time for the next iteration becomes $T_r - d_1$. This excess waiting time is propagated across subsequent iterations until it reaches zero. Conversely, if a request completes d time units earlier than T_r, the deadline for the next iteration is adjusted to $T_r + d$. This unused waiting time is also carried forward, allowing for extended waiting times in future iterations.

ACCELGEN prioritizes requests in ascending order of remaining time, which helps meet SLOs and enhances throughput by batching requests with similar remaining times, as indicated by O4. ACCELGEN classifies the requests to two categories: urgent requests and non-urgent requests. Urgent requests are those that must be executed in the next iteration to satisfy their SLOs. If a request's remaining time $T_r \approx T_{max}$, it is considered urgent.

4.4 Multi-resource-aware Batching

O2 shows it is important to jointly maximize both GPU compute utilization and KVC utilization at each iteration to improve throughput. O5 highlights that the varying resource demands of requests present an opportunity to select prompts or prompt chunks that can fully utilize both resources. Select-

---

Algorithm 1: Pseudocode of the batching algorithm.

Input : S_b, S_f
  A_{KV}: available (i.e., unallocated) KVC space
  U: urgent requests in waiting queue
  Q: waiting queue
Output : Batch **B** for the next iteration
1 $U \rightarrow B$ //Assign all urgent requests to the batch
2 **while** *Available GPU or KVC capacity < 0* **do**
3   Preempt the request$\in$ **B** with max T_r
4 **end while**
5 **if** *exists available GPU and KVC capacity* **then**
6   SelectRequests($S_b - S_f, A_{KV}, \mathbf{Q}, \mathbf{B}$)
7 **end if**

Algorithm 2: SelectRequests($A_{GPU}, A_{KV}, \mathbf{Q}, \mathbf{B}$).

Input : A_{GPU}: available GPU resource represented by # of tokens,
  A_{KV}, **Q** and **B**
Output : Batch **B** to fully utilize A_{KV} and A_{GPU}
1 $A_c = A_{GPU}$
2 $A_m = A_{KV}$
3 Get waiting requests with T_r within $T_r^1 + \gamma$
4 **for** *each long prompt in the selected requests* **do**
5   Create a shorter chunk with length either making $S_f = S_b$ or
      fully allocate KVC
6 **end for**
7 **while** *exists a request or chunk i with $A_c \geq D_c^i$ & $A_m \geq D_m^i$* **do**
8   Choose the request with min $\sqrt{(A_c - D_c^i)^2 + (A_m - D_m^i)^2}$
9   Add the request to **B**
10   Remove the request from **Q**
11     $A_c = A_c - D_c^i$
12     $A_m = A_m - D_m^i$
13 **end while**

---

ing requests using FCFS may fail to reach S_b due to KVC limits or may reach S_b without fully allocate the KVC. Thus, instead of using FCFS, after each iteration, ACCELGEN selectively chooses requests to be added to the batch to fully allocate KVC and also fully utilize GPU.

Requests with the same GPU compute demands can still have varying KVC demands. We use D_m^i to denote the KVC demand of a request i. If it is the first chunk or an entire prompt, then $D_m^i = \lceil \frac{S_l^i}{b} \rceil \times b$, where S_l^i denotes its sequence length and b denotes the block size. If it is a non-first chunk, some of its tokens may already have allocated KVC in the block of previous tokens. Thus, only the remaining tokens require KVC allocation, calculated as $D_m^i = \lceil \frac{remaining\ S_l^i}{b} \rceil \times b$. For a TG task (from a previous preemption), if a block is already assigned for its previous tokens including this token, then it does not need KVC allocation; otherwise, it needs one block of KVC. As a result, some requests may require less KVC space than others even though they have the same sequence length (or GPU compute demand). Therefore, it is important to identify the requests that can better utilize both resources to improve throughput.

After completing an iteration of a batch, the PP tasks in the batch transition to TG tasks from the next iteration onward. The ACCELGEN scheduler picks requests in the waiting

queue to achieve $S_f \approx S_b$. Algorithm 1 shows the pseudocode of the multi-resource aware batching algorithm. Urgent requests must be in the batch, and the previous TG requests should be retained in the batch as much as possible to allow them to complete and release KVC earlier (based on O3). We use **U** to denote the group of urgent requests. ACCELGEN first selects the requests from the urgent requests to the batch (line 1). If there are not enough resources to support an urgent request, the request with the highest T_r in the current batch will be preempted (Lines 2-4). After allocating urgent requests, if there are available GPU compute and KVC capacities, ACCELGEN selects requests from non-urgent queuing requests by calling function `SelectRequests()` (lines 5-7).

In the `SelectRequests()` function (shown in Algorithm 2), its goal is to select requests from the requests with the least remaining time to maximize GPU compute and KVC utilization in the next iteration. Let's use T_r^1 to denote the remaining time for the first non-urgent request. ACCELGEN selects several requests from the head of the queue that have T_r within $T_r^1 + \gamma$, where γ is a small value (Line 3). Then, if any long prompts exist, a chunk is created for each using the *SLO-guaranteed dynamic chunking method* described in §4.2, or by fully allocating KVC, whichever is shorter (Lines 4–6). This step aims to simultaneously maximize both GPU compute and KVC utilization. From this group, ACCELGEN first selects requests whose GPU and KVC demands (D_c^i and D_m^i) are the nearest to the available resources (A_c and A_m), i.e., those with the least Euclidean distance (Line 8). This ensures that the request that can use most of both the GPU and KVC resources is selected first. ACCELGEN repeats this process until the remaining resources cannot support more requests (Lines 7-12).

5 Implementation

We implemented ACCELGEN based upon the published source code of vLLM [12]. The ACCELGEN system is written in 9K lines of Python and 3K lines of C++ code. We developed the prioritization method using Python, and developed the dynamic chunking method using C++. We replaced the default FCFS scheduler in vLLM with ACCELGEN. In AC-CELGEN, the module for the multi-resource aware batching method communicates with the vLLM's block manager to obtain information regarding the current KVC state. ACCEL-GEN forwards the newly formed batch to the vLLM backend for execution. We added the *flash_attn_qkvpacked_func* from the FlashAttention2 [24] library in the vLLM engine for the parallelization of the attention operation. We directly use vLLM to handle KVC overflow and preemptions.

6 Performance Evaluation

6.1 Experiment Settings

The experiment settings are the same as in §3 unless otherwise specified. In addition to the mentioned LLM models in §3, we ran Llama-13B and Llama-3-70B model on one GPU and

four GPUs of one machine, respectively. They used 12GB and 135GB of GPU memory for the KVC, respectively. We set $\gamma = 0.75s$ empirically.

We set the TTFT SLO as explained in §3. To set the TBT SLO of a request, we randomly chose the SLO-scale from [0.75,1.25] and multiplied it by 0.1875s considering that some people may have a faster reading speed than the normal speed while others may have a slower reading speed.

Compared Methods. We compared ACCELGEN with vLLM, ORCA, FastGen and Sarathi-Serve (Sarathi in short). We also include FastServe [25] as a reference, which is built on ORCA. FastServe uses a skip-join multi-level feedback queue scheduler [26] to preempt the longest-running job to a lower-priority queue to minimize the average JCT. We set the number of queues to five. We also include the *Oracle* that knows the remaining time for job completion, the arrival times and output lengths of all requests, and schedules the chunking and batching to maximize the goodput. We use the Vidur simulator [27] to find the *Oracle*'s schedule solution.

6.2 Evaluation Results

In figures featuring both bars and lines, the bars represent the metric on the left axis, while the lines correspond to the metric on the right axis.

Throughput. Figure 11a shows the throughput in tokens/s and request/s of different systems. ACCELGEN has $1.42\times$-$11.21\times$, higher throughput in tokens/s and $1.54\times$-$10\times$ higher throughput in requests/s than other systems. ORCA, with iteration-level scheduling, and maximum memory allocation shows the minimum throughput. FastServe aims to handle head-of-line blocking in ORCA, thus improving its throughput. But its job preemption and swapping produce high latency and overhead. vLLM, FastGen, and Sarathi employ block-based KVC allocation to mitigate the KVC bottleneck and use chunking to enhance GPU compute utilization. However, by relying on FCFS and neglecting to optimize KVC utilization, they fail to consistently maximize GPU compute and KVC utilizations as shown in §3. ACCELGEN's SLO-guaranteed dynamic chunking method segments long prompts and batches requests with similar SLOs (facilitated by the iteration-level SLO-based task prioritization method) to fully utilize GPU while meeting the SLO requirements. Moreover, ACCELGEN's multi-resource-aware batching method simultaneously considers both GPU and KV cache resources when selecting requests to fully utilize both resources, thus increasing throughput. ACCELGEN shows 12% lower throughput than *Oracle*.

Goodput and SLO attainment. Figure 11b shows the goodput with T=1s of different systems. Figure 11c shows the iteration-level SLO and JCT SLO attainments of different systems. ACCELGEN improves the goodput of other systems by $1.43\times$-$13.71\times$, improves their SLO attainment by 37%-90%, and improve their JCT SLO attainment by 34%-93%, ACCELGEN accounts for heterogeneous SLOs in improv-
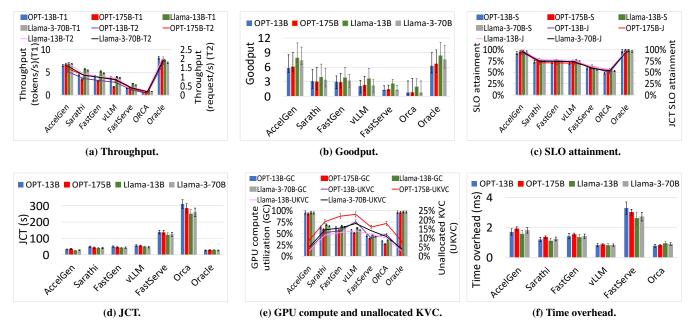
(a) Throughput.

(b) Goodput.

(c) SLO attainment.

(d) JCT.

(e) GPU compute and unallocated KVC.

(f) Time overhead.

Figure 11: Results of ACCELGEN and compared systems for different models.

ing throughput, resulting in significantly higher goodput and improved SLO attainment, which deliver a superior user experience. *Oracle* only shows 1.1% higher SLO and JCT SLO and 6% higher goodput than ACCELGEN.

JCT. Figure 11d shows the JCT of different systems. AC-CELGEN reduces reduces their JCT by 1.61×-12.22×. This is because ACCELGEN's high throughput (explained in Figure 11a) helps reduce request waiting time. *Oracle* shows 22% lower JCT compared to ACCELGEN.

GPU and KVC utilization. We present the average GPU compute utilization and unallocated KVC for all systems in Figure 11e. ACCELGEN shows 31%-63% higher GPU compute utilization than other systems. Moreover, ACCELGEN shows 7.40%-13.20% lower unallocated KVC compared to other systems. This is because ACCELGEN jointly optimizes the utilization of both resources. Selecting the request that better utilizes both resources reduces instances where one resource is insufficient for a request, thereby increasing overall GPU compute utilization. Oracle shows 2% higher GPU compute utilization and 1.5% lower unallocated KVC compared to ACCELGEN.

Time overhead. Figure 11f shows the average time overhead per iteration for each system. Compared to ACCELGEN, Fast-Serve has 95% higher time overhead, while other systems have 28%-93% lower time overhead. FastServe has a higher time overhead because it makes decisions on preemption and memory swapping. Other systems simply use FCFS. ACCEL-GEN, building upon vLLM, introduces additional methods that contribute to slightly longer time overhead. Despite this, ACCELGEN achieves a significant reduction of 1.61×-12.22× in JCT compared to these methods, highlighting the effective-

ness of ACCELGEN's approaches. Notably, ACCELGEN's time overhead remains 0.02% of the average JCT.

All systems perform better in OPT-175B than in OPT-13B due to the model parallelization. The parallelization allows memory-bound LLM models to access more available memory across eight GPUs, resulting in improved performance. However, the overall improvement for ACCELGEN is lower for the OPT-175B model compared to the OPT-13B model because of more available KVC for the OPT-175B model. All systems perform similarly on Llama-13B and Llama-3-70B, and ACCELGEN shows comparable performance improvements on both models due to the lower degree of parallelism.

6.3 Ablation Testing

We use *Chunk*, *Order*, *Batch* to represent the three methods in sequence in ACCELGEN, respectively. We use /Chunk, /Order, and /Batch to represent ACCELGEN without each method, respectively.

Figure 12a illustrates the throughput of individual variants of ACCELGEN. Notably, /Chunk has the most substantial effect, reducing throughput by 97% since *Chunk* leverages long prompts to mitigate their negative impact and enhance throughput. /Batch has the second-highest impact (61%), as *Batch* aims fully utilizing both resources. /Order exhibits the least impact, reducing throughput by 35%, as *Order* primarily focuses on ensuring that each request is completed within a designated time limit.

Figures 12b, and 12c show /Order, /Chunk, /Batch decrease the goodput of ACCELGEN by 52%, 44%, and 27%, respectively, reduce ACCELGEN's SLO attainment by 42%, 28%, and 24%, respectively, and they increase ACCELGEN's iter-
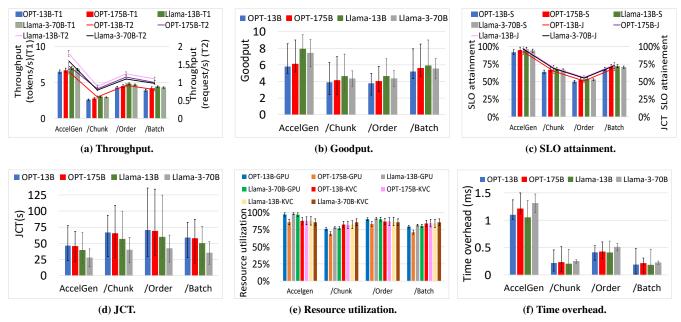
(a) Throughput.

(b) Goodput.

(c) SLO attainment.

(d) JCT.

(e) Resource utilization.

(f) Time overhead.

Figure 12: Results of AccelGen without a method for different models.

ation time by 56%, 39% and 29%, respectively. *Order* has the highest impact as it helps to satisfy the SLOs. *Chunk* has the second impact because it significantly improves GPU compute utilization and throughput while satisfying SLOs. *Batch* has lesser impacts as it aids in more fully utilizing GPU compute and KVC resources simultaneously.

Figure 12d depicts the JCT of each variant. /Order, /Chunk, and /Batch increase JCT by 17%, 31%, and 19%, respectively. *Chunk* stands out with the highest impact, followed by *Batch*.

In Figure 12e, /Order, /Chunk, and /Batch decrease GPU compute utilization by 8%, 28%, and 15%, respectively. Similarly, they decrease KVC utilization by 1%, 7%, and 11%, respectively. *Chunk* is most effective at fully utilizing both resources, leveraging long prompts, while *Batch* contributes significantly to resource utilization by finding requests that fully utilize both GPU and KVC resources simultaneously. *Order* has the least impact, focusing solely on SLO considerations.

Figure 12f shows the time overhead for AccelGen variants. /Order, /Chunk and /Batch have 63%, 81%, and 82% lower overhead than AccelGen. *Order* generates lower time overhead than others, as it primarily involves ordering requests in the queue. *Chunk* and *Batch* generate higher time overhead due to their more complex operations.

6.4 Sensitivity Testing

Figure 13 illustrates the influence of the percentage of long prompts and the arrival rate on the goodput. As the percentage of long prompts and the arrival rate increase, the goodput exhibits a continuous rise, with the increase rate decreasing toward the end as the system approaches its capacity limits. The results indicate AccelGen's ability to handle long
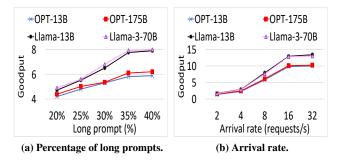

(a) Percentage of long prompts.

(b) Arrival rate.

Figure 13: Sensitivity testing.

prompts and high request arrival rates. This is achieved through effective utilization of various prompts, ensuring the full use of both GPU and KVC resources to enhance goodput.

6.5 Performance on Individual Dataset

Figure 14 shows the goodput performance for each of the three datasets. AccelGen shows the highest goodput improvement for the long-prompt BookCorpus dataset compared to Alpaca and ShareGPT. AccelGen improves the goodput of other systems by 2.57×-33× for BookCorpus 1.41-18× and 1.45-19× for Alpaca and ShareGPT, respectively. Long prompts offer greater opportunities to enhance goodput.

7 Limitations and Discussion

- If the number of generated tokens can be accurately predicted, AccelGen can be further optimized by fully leveraging this information to approach the performance of *Oracle*. For instance, it can avoid preempting jobs that are nearing completion, leading to better long-term scheduling of KVC usage. We will study this topic.
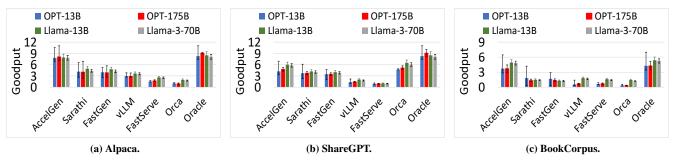
(a) Alpaca. **(b) ShareGPT.** **(c) BookCorpus.**

Figure 14: Results of ACCELGEN for individual dataset.

- Chunking long prompts has the potential to introduce delays in their processing. We aim to explore adaptive methods for determining the chunking size, ensuring that SLOs for long prompts are met.
- ACCELGEN currently incurs certain scheduling latency, and we plan to investigate approaches to minimize this latency.
- Exploring the impact of preempting long prompts and designing a strategy to select requests for preemption based on factors like sequence length and SLO is an important aspect that has not been thoroughly studied in ACCELGEN. We will investigate these topics.

8 Related Work

Job scheduling for transformer-based generative LLM inference systems has recently garnered significant attention. ORCA [5] integrates the scheduler and execution engine, employing iteration-level scheduling and selective batching. Despite these features, ORCA still faces challenges related to memory bottlenecks and GPU under-utilization. Kwon *et al.* proposed vLLM [12], which uses a block-based approach to overcome the problem of memory bottlenecks. The new vLLM version adopts the chunking method [13]. Sarathi-Serve [14] introduces chunked-prefill which splits a prefill request into near equal sized chunks and stall-free schedule that adds new requests into a batch without pausing ongoing decodes. Jin *et al.* [9] addressed GPU under-utilization by proposing S^3, a method that predicts the output sequence length for memory allocation. Wu *et al.* [25] proposed Fast-Serve, which uses preemptive scheduling to minimize JCT with a skip-join multi-level feedback queue scheduler. Zheng *et al.* [23] proposed a system that predicts the output sequence length of an input sequence and then groups queries with similar response lengths into micro-batches. Oh *et al.* [28] proposed *ExeGPT* which finds the execution schedule that maximizes inference throughput under a given latency constraint. The execution schedule includes the batch size, GPU count, and tensor parallelism degree. Splitwise [29] and Dist-Serve [30] assign prefill and decoding computation to different GPUs in order to maximize the throughput. Sheng *et al.* [31] considered fairness in scheduling based on a cost function that accounts for the number of input and output tokens processed in the inference server from each client. Zhao *et*

al. [32] introduced ALISA, a solution that combines a Sparse Window Attention algorithm to reduce the memory footprint of KVC with a three-phase token-level dynamic scheduling system. Llumnix [33] reschedules heterogeneous and unpredictable requests among multiple model instances to reduce tail latency.

Many other approaches have been proposed to improve performance. Sheng *et al.* [34] proposed FlexGen, which addresses memory bottlenecks by solving a linear programming problem to identify efficient patterns for storing and accessing tensors. Liu *et al.* [35] proposed asynchronous lookahead predictors that predict the sparsity for the attention head at the next layer and optimize the operations to speed up LLM inference in real time. Zheng *et al.* [36] proposed KVC reuse by storing KVC in a radix tree for multiple requests. Liu *et al.* [37] proposed compressing KVC by storing pivotal tokens based on attention scores. Compiler systems like those in [38, 39] address dynamic sparsity in LLM models by constructing GPU-efficient tiles to increase GPU compute utilization and low memory wastage.

However, none of the aforementioned schedulers handle the heterogeneous SLOs in the mixed-prompt scenarios. AC-CELGEN addresses this and aims to simultaneously maximize both the GPU compute and KVC utilization to improve throughput and goodput.

9 Conclusion

Existing schedulers for transformer-based LLM inference serving systems often fail to meet heterogeneous SLOs of different requests and maximize throughput in the mixed-prompt scenarios. To tackle this issue, we conducted a thorough measurement analysis of the performance of existing LLM inference systems in this scenario. Drawing from our observations, we propose ACCELGEN, which offers iteration-level SLO guarantees and maximizes both GPU compute and KVC resources in scheduling. Our trace-driven experiments show the superior performance of ACCELGEN in meeting heterogeneous SLOs hence delivering satisfactory user experience and improving throughput, goodput and JCT compared to the state-of-the-art approaches. Currently, ACCELGEN is only tested for the mentioned models and the GPU settings. We will test it on more different models and machine settings.

References

[1] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models. *ArXiv*, abs/2205.01068, 2022.

[2] Abigail See, Peter Liu, and Christopher Manning. Get to the point: Summarization with pointer-generator networks. In *Association for Computational Linguistics*, 2017.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021.

[4] Stephen Roller, Emily Dinan, Naman Goyal, Da Ju, Mary Williamson, Yinhan Liu, Jing Xu, Myle Ott, Kurt Shuster, Eric M Smith, et al. Recipes for building an open-domain chatbot. *arXiv preprint arXiv:2004.13637*, 2020.

[5] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[6] Sosuke Kobayashi. Homemade bookcorpus. https://github.com/soskek/bookcorpus, 2018.

[7] Marc Brysbaert. How many words do we read per minute? a review and meta-analysis of reading rate. *Journal of Memory and Language*, 109, 08 2019.

[8] Openai api documentation. https://help.openai.com/en/articles/ 4936856-what-are-tokens-and-how-to-count-them.

[9] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. S^3: Increasing gpu utilization during generative inference for higher throughput. *arXiv preprint arXiv:2306.06000*, 2023.

[10] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. Stablegen: Efficient llm inference with low tail latency. *Proc. of OSDI*, 2024.

[11] MicroSoft. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. https://github.com/microsoft/DeepSpeed/tree/master/blogs/deepspeed-fastgen, 2023.

[12] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[13] [feature] implement dynamic splitfuse #1562. https://github.com/vllm-project/vllm/issues/1562.

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

[15] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[16] kipply's blog. Transformer inference arithmetic. https://kipp.ly/transformer-inference-arithmetic/#kv-cache.

[17] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[18] ShareGPT. sharegpt-english. https://huggingface.co/datasets/theblackcat102/sharegpt-english, 2023.

[19] Jongwook Choi. gpustat. `https://github.com/woo kayin/gpustat`. Online; accessed 26 August 2023.

[20] Yu Emma Wang, Gu-Yeon Wei, and David M. Brooks. Benchmarking tpu, gpu, and cpu platforms for deep learning. *ArXiv*, abs/1907.10701, 2019.

[21] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Z. Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. In *USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[22] Sudipta Saha Shubha and Haiying Shen. Adainf: Data drift adaptive scheduling for accurate and slo-guaranteed multiple-model inference serving at edge servers. In *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023.

[23] Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. Response length perception and sequence scheduling: An LLM-empowered LLM inference pipeline. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[24] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *ArXiv*, abs/2307.08691, 2023.

[25] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *ArXiv*, abs/2305.05920, 2023.

[26] Shweta Jain and Saurabh Jain. Analysis of multi level feedback queue scheduling using markov chain model with data model approach. *International Journal of Advanced Networking and Applications*, 7:2915–2924, 06 2016.

[27] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S. Gulavani, Ramachandran Ramjee, and Alexey Tumanov. Vidur: A large-scale simulation framework for llm inference. *ArXiv*, abs/2405.05465, 2024.

[28] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du-seong Chang, and Jiwon Seo. Exegpt: Constraint-aware resource scheduling for llm inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024.

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

[30] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *USENIX Symposium on Operating Systems Design and Implementation*, 2024.

[31] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Fairness in serving large language models. In *Proceedings of OSDI*, 2024.

[32] Youpeng Zhao, Di Wu, and Jun Wang. Alisa: Accelerating large language model inference via sparsity-aware kv caching. *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 1005–1017, 2024.

[33] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 173–191, Santa Clara, CA, July 2024. USENIX Association.

[34] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez, et al. High-throughput generative inference of large language models with a single gpu. *Proc. of ICML*, 2023.

[35] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, and Beidi Chen. Deja vu: Contextual sparsity for efficient LLMs at inference time. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

[36] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, Ying Sheng. Fast and Expressive LLM Inference with RadixAttention and SGLang. `https://lmsys.org/blog/2024-01-17-sglang/`. Online; accessed in April 2024.

[37] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[38] Ningxin Zheng, Huiqiang Jiang, Quanlu Zhang, Zhenhua Han, Lingxiao Ma, Yuqing Yang, Fan Yang, Chengruidong Zhang, Lili Qiu, Mao Yang, and Lidong Zhou.

Pit: Optimization of dynamic sparse deep learning models via permutation invariant transformation. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

[39] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. SparTA: Deep-Learning model sparsity via Tensor-with-Sparsity-Attribute. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.