BLENDSERVE: OPTIMIZING OFFLINE INFERENCE FOR AUTO-REGRESSIVE LARGE MODELS WITH RESOURCE-AWARE BATCHING

Yilong Zhao [*1] **Shuo Yang** [*1] **Kan Zhu** [2] **Lianmin Zheng** [3] **Baris Kasikci** [2] **Yang Zhou** [1] **Jiarong Xing** [1] **Ion Stoica** [1]

ABSTRACT

Offline batch inference, which leverages the flexibility of request batching to achieve higher throughput and lower costs, is becoming more popular for latency-insensitive applications. Meanwhile, recent progress in model capability and modality makes requests more diverse in compute and memory demands, creating unique opportunities for throughput improvement by resource overlapping. However, a request schedule that maximizes resource overlapping can conflict with the schedule that maximizes prefix sharing, a widely-used performance optimization, causing sub-optimal inference throughput. We present BlendServe, a system that maximizes resource utilization of offline batch inference by combining the benefits of resource overlapping and prefix sharing using a resource-aware prefix tree. BlendServe exploits the relaxed latency requirements in offline batch inference to reorder and overlap requests with varied resource demands while ensuring high prefix sharing. We evaluate BlendServe on a variety of synthetic multi-modal workloads and show that it provides up to $1.44\times$ throughput boost compared to widely-used industry standards, vLLM and SGLang.

1 INTRODUCTION

Offline batch inference is becoming increasingly popular as a cost-effective solution for Large Language Model (LLM) inference, especially in applications that are not latency-sensitive. Compared to online inference, offline batch inference services have relaxed service-level objectives (SLOs), such as the 24-hour response windows in OpenAI and Anthropic's batch APIs (OpenAI, 2024b; Anthropic, 2024). This flexibility enables more efficient request batching and scheduling, resulting in higher throughput and lower inference costs. As a result, many service providers now offer offline batch inference services (OpenAI, 2024b; Anthropic, 2024; AWS, 2024; Anyscale, 2024; Databricks, 2024).

At the same time, advances in model capabilities and modalities resulted in an increasingly diverse and challenging set of workloads that inference services must handle. With enhanced long-context and reasoning capability, modern LLMs like OpenAI O1 (OpenAI, 2024a) can generate outputs across a wide range of lengths, from a few to thousands of tokens (OpenAI, 2024a). Additionally, the rapid development of multi-modal models, which support a mix of input and output types (e.g., text, image, and video), introduces

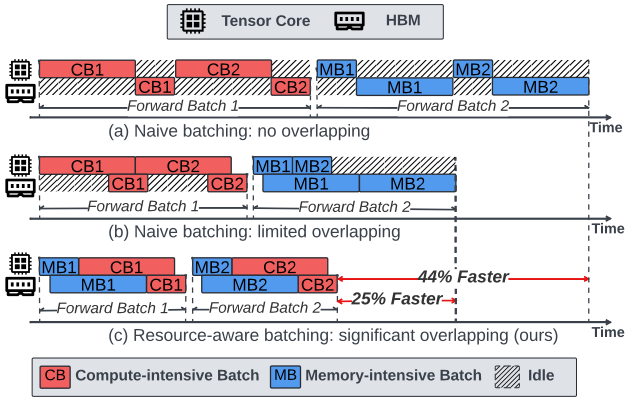

Figure 1. Three paradigms for scheduling compute- and memory-intensive requests in offline batch inference. For simplicity, the utilization of GPU tensor cores and HBM bandwidth across different operators within a batch is combined into one rectangle.

even more diverse workloads (Wang et al., 2024a; Wu et al., 2024c; Wang et al., 2024c; Wu et al., 2024b). For instance, video understanding tasks may take extensive input tokens but yield fewer output tokens, whereas video generation requests can produce a substantial number of output tokens.

The diverse workloads result in varied resource demands during inference: some requests are compute-intensive (e.g., vision understanding with large input length), while others are memory-intensive (e.g., video generation or long output length with complex reasoning). This diversity in resource demand presents a unique opportunity to enhance hardware

[*]Equal contribution [1]University of California, Berkeley [2]University of Washington [3]xAI. Correspondence to: Yilong Zhao <yilongzhao@berkeley.edu>, Shuo Yang <andy_yang@berkeley.edu>.

Preprint, under review.

utilization in serving systems—by leveraging offline batch inference's flexibility, we can reorder requests to maximize the overlapping between compute- and memory-intensive tasks. As illustrated in Figure 1, sequential execution of requests with different resource usages can lead to substantial under-utilization. Existing work like NanoFlow has exploited resource overlapping to improve GPU utilization, but it batches requests without considering their diverse resource demands, limiting the potential for overlapping (Zhu et al., 2024). By fully utilizing the diversity in requests' resource demands, we can create batches with blended compute- and memory-intensive requests, significantly improving the efficacy of resource overlapping. This can further increase GPU utilization, reducing the batch inference time and lowering the cost.

The key challenge here is determining the request order in batches that can maximize resource overlapping and minimize overall inference time. However, strategies for maximizing resource overlapping may conflict with another common technique for accelerating inference—prefix sharing (Zheng et al., 2024b; Lin et al., 2024; Juravsky et al., 2024). Prefix sharing involves group requests with shared prefixes, allowing the shared prefix to be computed only once. This approach, however, can be at odds with the reordering needed to maximize resource overlapping. A request order that achieves high prefix sharing does not necessarily yield high resource utilization, and vice versa. Therefore, we must consider both factors together and develop a strategy that effectively combines their advantages.

To address this challenge, we propose BlendServe, a system that maximizes hardware resource utilization by reordering and overlapping requests with complimentary resource requirements under the relaxed SLOs of offline batch inference. Meanwhile, BlendServe maintains a high prefix sharing ratio, thus combining the benefits of both techniques. The core design of BlendServe is based on a resource-aware prefix tree where each node encodes the compute density of all requests represented in its subtree. Based on this, BlendServe sorts the tree nodes according to their density values, arranging compute-intensive nodes to the left and memory-intensive nodes to the right. The sorted tree preserves the structure of the prefix tree, so it inherits the benefit of prefix sharing via a DFS order. To determine the request order for batching, BlendServe employs a dual scanner algorithm, which scans the tree leaves from left and right simultaneously, effectively batching compute-intensive requests with memory-intensive requests for overlapping.

We prototype BlendServe by leveraging the prefix tree implementation from SGLang (Zheng et al., 2024b) and implement a backend engine based on NanoFlow (Zhu et al., 2024). We evaluate BlendServe on a wide range of datasets featuring different prefix-sharing ratios and resource demands, including chat, benchmark, and multi-modal workloads. We compare our performance against state-of-the-art serving engines including vLLM (Kwon et al., 2023), SGLang, and two variants of NanoFlow: NanoFlow-DFS, which maximizes prefix sharing, and NanoFlow-Balance, which randomly shuffles requests for resource overlapping. BlendServe achieves 19.34% to 22.65% throughput gains compared to the best baseline, NanoFlow-DFS, by achieving near-optimal prefix sharing and stable resource usage. Compared to the industry standard vLLM and SGLang, BlendServe achieves up to $1.44\times$ throughput speedup.

In summary, our main contributions are:

- We conduct a detailed analysis of offline serving workloads, revealing the opportunities brought by diverse input and output lengths.
- We design BlendServe, an offline serving system that batches requests with diverse resource demands for overlapping while maintaining high prefix sharing.
- We build a prototype and evaluate it comprehensively, achieving significant throughput improvement compared to state-of-the-art serving frameworks.

2 Background and Motivation

In this section, we provide a primer for transformer-based large model inference (§ 2.1) and illustrate the evolving diversity of inference workloads (§ 2.2). Then we motivate our work by explaining the diverse resource demands across requests (§ 2.3), and introduce our research goal and challenge with a performance analysis (§ 2.4).

2.1 Transformer-based large model inference

LLM inference. LLM inference involves two main phases: *prefill* and *decode*. The prefill phase processes the initial input sequence (i.e., prompt) in parallel, which is compute-intensive due to the large parallelism. After that, the decode phase generates output tokens in an *auto-regressive* manner, generating one token at a time, with each token prediction conditioned on all previously generated tokens (Vaswani et al., 2023). These two phases share the same series of operations including KQV generations, O projection, self-attention, feed-forward network (FFN), and others, which show distinct resource usage of compute and memory in different phases. For example, KQV generation, O projection, and FFN consist of general matrix-matrix multiplication (GEMM), which is memory-intensive in the decode phase due to the low parallelism but compute-intensive in the prefill phase (Zhao et al., 2024c). Existing studies analyze operator-level resource usage in different phases and improve resource utilization by techniques such as continuous batching or chunked prefill (Yu et al., 2022; Agrawal et al.,

2024; Lin et al., 2023). However, they do not consider request-level resource usage, missing the holistic opportunities by overlapping requests consuming different resources.

Online vs. offline serving. There are two primary LLM serving scenarios: *online serving* and *offline serving*. In online serving, LLMs are deployed to handle user requests in real-time, such as in chatbots, code assistants, or interactive applications (OpenAI, 2022; Microsoft, 2023). In this context, SLOs are critical and strict to guarantee user experience. Consequently, online serving frameworks often use a first-come-first-serve (FCFS) strategy to meet SLO requirements (Yu et al., 2022; Kwon et al., 2023). Conversely, offline serving processes large volumes of requests where latency for individual requests is less critical, with a focus on *maximizing overall throughput*. Examples include data cleaning tasks (Zheng et al., 2023), synthetic data generation (Allal et al., 2024), and model checkpoint evaluations (Meta-Team, 2024). Since immediate responses are not required, requests can be scheduled with an arbitrary order and batched more flexibly. We leverage this flexibility to reorder requests for resource overlapping and prefix sharing, enhancing offline inference throughput.

2.2 Evolving diversity of inference workloads

The growing complexity of LLM inference paradigms leads to greater *workload diversity*, i.e., input/output token length. First, the range of decoding length of modern LLMs has increased significantly due to the recent inference advances (Snell et al., 2024; Qin et al., 2024). As test-time compute is getting more attention, complex inference like Tree of Thoughts, Chain of Thought, and Agent Systems is used to enhance the model's capability for solving hard problems, leading to greatly increased output length (Yao et al., 2023; Wei et al., 2023; Tang et al., 2024). For instance, OpenAI O1 has complex reasoning steps during inference, which generates numerous intermediate tokens (counted in the output length even though they are invisible to users) before generating the first output tokens (OpenAI, 2024a).

On the other hand, recent progress in transformer-based auto-regressive multi-modal models (e.g., LWM (Liu et al., 2024), Unified-IO (Lu et al., 2023), EMU (Wang et al., 2024a), MIO (Wang et al., 2024c), and VILA-U (Wu et al., 2024c)) demonstrates the potential of extending language models to handle diverse types of tasks within a unified architecture. These models typically share a common base model architecture: a transformer-based LLM augmented with several modality-specific adapters. These adapters convert inputs from various modalities into a format that the base model can process and translate its outputs back into the desired modality. Multi-modal models further exacerbate the variances of input/output token length. For example, text-only chatbot requests typically have hundreds of tokens as context length (Zheng et al., 2024a), while a video generation request can generate up to thousands output length.

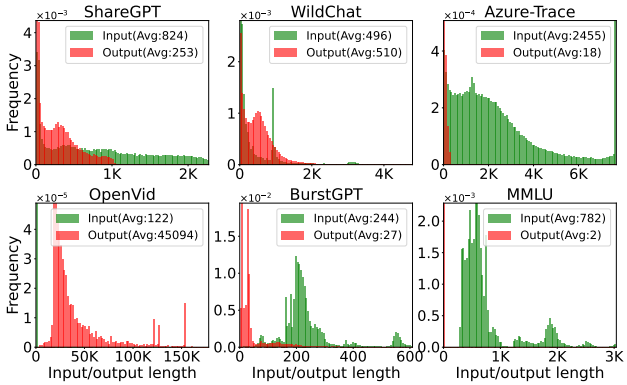

Figure 2. Request input/output length distribution from 6 well-known open-sourced traces, including chatbot ShareGPT, WildChat, and BurstGPT (Zhao et al., 2024a; Zheng et al., 2024a; Wang et al., 2024b), video datasets OpenVid (Nan et al., 2024), benchmark traces MMLU (Hendrycks et al., 2021) and production traces Azure-Trace (Stojkovic et al., 2024). Requests from different traces demonstrate distinct length distributions.

To showcase the workload diversity, we illustrate the variance in request length distributions in different use cases in Figure 2. While chatbot requests typically have less than 1K output length, the long generation and video traces can easily generate more than 5K tokens.

2.3 Diverse resource demands across requests

These diverse requests consume different types of GPU resources (i.e., compute and memory bandwidth). For example, requests with long output length tend to consume more GPU memory bandwidth due to the quadratically increased memory access size, while requests with shorter output length tend to demand more compute because of the intensive GEMMs. To better understand this problem, we construct a resource usage model for a request with input length p and output length d. Given a model of P_{model} parameters, H hidden dimension of model width, H_{kv} feature dimension for each KV head, and L decoder layers, and a hardware configuration of `compute` peak FP16 GFlops and `bandwidth` GB/s memory bandwidth, the total time for compute-bound operators of a single request r can be approximated by total computation amount of GEMM operators and the self-attention in prefill phase divided by hardware compute capability:

$$\text{Comp}(r) \approx \frac{(p+d) \cdot P_{model} \cdot 2 + p^2 \cdot H \cdot L \cdot 4}{\texttt{compute}}$$

where $(p+d)$ is the number of tokens processed by all GEMM operators. Since parameters of GEMM (QKV

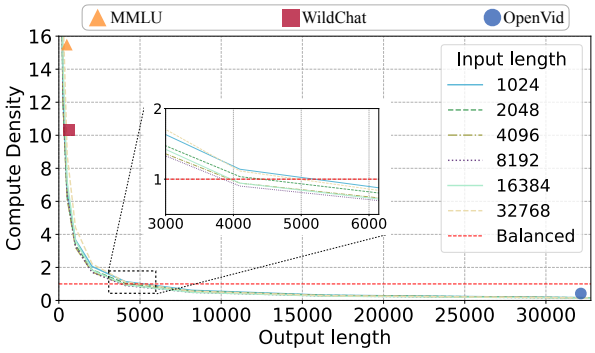

Figure 3. Compute density of requests with different input/output lengths (Llama-3-70B on 8xA100 80GB GPUs). Compute density is calculated as compute usage divided by memory usage, following the definition in § 2.3. The results show that requests with different output length leads to diverse resource demands.

generation + FFN) occupy most of the model parameter, the computation amount can be effectively approximated by `model_size` and $(p+d)$ (Zhu et al., 2024). The p^2 of the second terms comes from the quadratic computation of self-attention in the prefill phase, which is much smaller than first term with a p less than 64K. The total time for memory-bound operators can be approximated as:

$$\begin{aligned}\text{Mem}(r) &\approx \frac{\sum_{i=1}^{d}(p+i)\cdot H_{kv}\cdot L\cdot 2\cdot 2}{\texttt{bandwidth}} \\ &\approx \frac{(p\cdot d+\frac{1}{2}\cdot d^2)\cdot H_{kv}\cdot L\cdot 4}{\texttt{bandwidth}}\end{aligned}$$

where $\sum_{i=1}^{d}(p+i)$ calculates the total number of loaded tokens by self-attention during the d steps of the auto-regressive generation process, and 4 comes from Key and Value vectors stored in FP16 for each token.

Based on this cost model, we calculate the theoretical usage of GPU compute and memory for processing requests with various input length and output length. Note that since multi-modal adapters add very minimal overhead compared to their transformer-based backbone—e.g., less than 0.1% when generating a short 30-frame video with LWM on an NVIDIA A100 GPU[1]—we only consider the resource usage of their transformer-based backbone models. As shown in Figure 3, the output length determines whether a request is compute- or memory-intensive. When output length is larger than 4K, a typical case in video generations, the requests are completely memory-intensive. In contrast, the chatbot and benchmark requests from WildChat (Zhao et al., 2024a) and MMLU (Hendrycks et al., 2021) are compute-intensive due to their short output length. The great diversity of workloads leads to diverse resource demands, posing a great opportunity for resource overlapping.

[1]We profile LWM-Chat-32K-Jax on a single A100 80GB SXM with the official implementation. To generate a 30-frame video with 256 tokens per frame, the backbone model takes 661.7s and the adaptor, using VQGAN to generate video, takes only 0.43s, which is less than 0.1% of the total inference time.

2.4 Optimizing offline inference throughput

Opportunity: request-level resource overlapping. The request diversity presents a unique opportunity for optimizing offline inference performance. With relaxed latency requirements in offline batch inference, we can reorder requests and create inference batches that blend compute- and memory-intensive requests. This enables us to collocate compute and memory usages, enhancing GPU resource utilization and achieving higher throughput.

Challenge: resource overlapping vs. prefix sharing. Resource overlapping can interfere with another key optimization: prefix sharing (Lin et al., 2024; Zheng et al., 2024b). This technique caches the computed prompts from previously processed requests, which are looked up before the process of incoming requests. Once cache hits, the common prefix of prompts can be used without recomputation, thus significantly boosting throughput (Ye et al., 2024). For example, certain workloads can save up to 80% computation usage with prefix sharing (Zheng et al., 2024b). Therefore, many serving frameworks leverage this technique (Kwon et al., 2023; LMDeploy, 2023). Similar to the classic cache management, different replacement policies of the prefix cache can lead to different cache hit rates, which is denoted as *prefix sharing ratio* in this work. To achieve the highest prefix sharing ratio, requests need to be reordered following DFS order to batch similar requests together (Zheng et al., 2024b; Srivatsa et al., 2024a); however, this reordering can be at odds with the reordering needed to maximize resource overlapping. The performance of request ordering with a high prefix-sharing ratio may be constrained by imbalanced resource demands across requests. As a result, we must consider resource overlapping and prefix sharing simultaneously to achieve the best of both.

Unfortunately, existing request reordering fails to achieve the best of both, as summarized in Table 1. (1) *First-Come-First-Serve (FCFS)* is mostly used by online inference due to the latency constraint (Yu et al., 2022; Kwon et al., 2023; Zheng et al., 2024b; Zhu et al., 2024; Patel et al., 2024), thus it does not consider resource overlapping and cannot share the same prompts of inconsecutive requests. (2) *Depth-First-Search (DFS)* is used in a Trie Tree to find the best order that can maximize the prefix sharing ratio (Zheng et al., 2024b; Srivatsa et al., 2024a). However, it tends to batch requests with similar prompts together which potentially have similar resource usage without considering resource overlapping. (3) *Random* shuffled requests can uniformly batch diverse requests to balance compute and memory bandwidth resource usage for overlapping, but it significantly disrupts prefix sharing, causing expensive recomputation.

Policy	Prefix sharing	Resource overlapping
FCFS	✗	✗
DFS	✓	✗
Random	✗	✓
Ours	✓	✓

Table 1. Existing schedule policies cannot achieve both high prefix sharing and request-level resource overlapping.

Our goal: combining both for throughput optimization. This work aims to optimize the throughput for offline batch inference by taking advantage of both resource overlapping and prefix sharing. We formulate this problem by deriving the optimal throughput as the following. Consider a set of offline inference requests R, which requires certain amounts of compute and memory bandwidth resources:

$$T_{\text{comp}} = \sum_{r \in R} \text{Comp}(r), \quad T_{\text{mem}} = \sum_{r \in R} \text{Mem}(r)$$

where T_{comp} and T_{mem} are processing time for compute- and memory-intensive operators based on the performance model in § 2.3. Considering a prefix sharing ratio $s \in [0, 1]$, which means s of the total compute T_{comp} are saved, the compute time will be reduced to $(1 - s) \cdot T_{\text{comp}}$. We do not enable cascade attention as described in § 6.2. Therefore, the end-to-end processing time T will be:

$$T = f((1 - s) \cdot T_{\text{comp}}, T_{\text{mem}})$$

where f depends on the scheduling policy. For example, using an FCFS policy with sequential operator execution, f will be $sum(\cdot, \cdot)$. To achieve a minimal processing time T_o, we require perfect overlapping, namely $f = max(\cdot, \cdot)$, and the highest prefix-sharing ratio s_o:

$$T_o = \max((1 - s_o) \cdot T_{\text{comp}}, T_{\text{mem}})$$

In the next section, we will describe how we approach T_o through a scheduling algorithm that achieves both an effective overlapping f and a high prefix sharing ratio s.

3 BlendServe Design

Overview. Figure 4 shows the end-to-end workflow of BlendServe. Given a set of requests upfront with known prompts, BlendServe first constructs a prefix tree to capture the shared prefix among requests (①, § 3.1). Next, BlendServe calculates compute density for each node, which involves estimating request output length by sampling over the prefix tree (②, § 3.2). With compute density, requests are characterized as compute- or memory-intensive and sorted based on their resource usage, resulting in a sorted tree where most compute-intensive requests are on the left and most memory-intensive requests are on the right. This process preserves the structure of the prefix tree (③, § 3.3). Therefore, BlendServe can efficiently find a request order by sweeping the tree from left and right simultaneously. This order can balance compute-memory demand for resource overlapping and has high prefix sharing (④, § 3.4). Finally, the ordered requests are batched and fed into a backend engine for inference. In the rest of this section, we present detailed descriptions of these key components.

3.1 Key data structure: resource-aware prefix tree

Requests in BlendServe are organized based on a prefix tree, which is implemented following the Trie Tree (Zheng et al., 2024b). Each leaf node in this tree represents an actual request while each internal node is a segment of the prefix shared by all its descendants. Therefore, a path from the root node to the leaf node represents the longest shared prefix of this request. By walking through this prefix tree in a DFS order, we can obtain a request sequence that maximizes the prefix sharing ratio. However, this will neglect diverse resource demands across requests and miss the opportunity for resource overlapping. Conversely, if we randomly pick up requests from the tree, we can achieve an expected resource demand balance across requests for overlapping but sacrifice the benefit of prefix sharing.

To address this problem, we enhance the prefix tree with resource usage information for each node, making it a resource-aware prefix tree. At a high level, we calculate a metric called *compute density* for each node, representing its compute and memory usage. Nodes with higher metrics correspond to compute-intensive requests, while lower metrics indicate memory-intensive ones. With this, we can sort the prefix tree, i.e., compute-intensive requests on the left-hand side and memory-intensive requests on the right-hand side. The sorted prefix tree enables an efficient search for a request order that overlaps distinct resource demands while maximizing prefix sharing.

3.2 Enriching prefix tree with compute density

Compute density definition. The compute density $\rho(r)$ of a request r is the total compute time divided by the total time of memory-bound operators, following the similar intuition of arithmetic intensity (Williams et al., 2009):

$$\rho(r) = \frac{\text{Comp}(r)}{\text{Mem}(r)}$$

where $\rho(r)$ is calculated using the cost model described in § 2.3. With the same notations in § 2.4, compute density for a set of requests R with shared prefix is:

$$\rho(R) = \frac{(1 - s) \cdot T_{\text{comp}}}{T_{\text{mem}}}$$

BlendServe calculates compute density for each node in the prefix tree. For a leaf node, the compute density represents

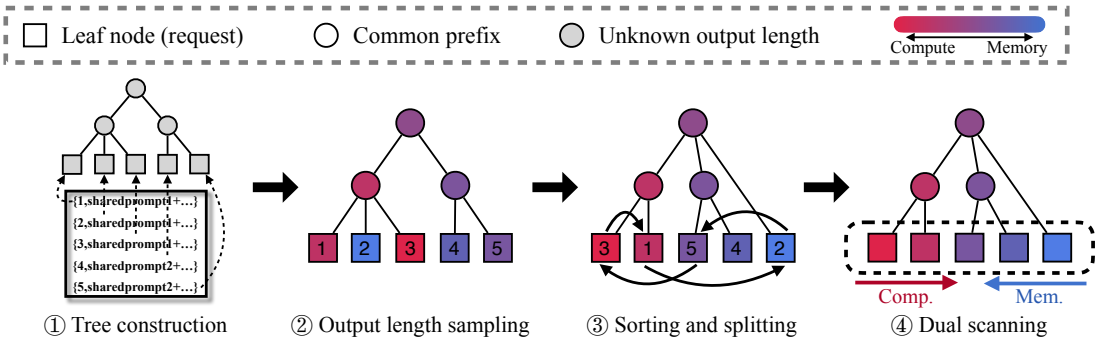

Figure 4. Overview of BlendServe's design. Leaf nodes in the prefix tree are actual requests while others represent the common prefix in prompts. The color of nodes represents the performance characterization of all requests within the sub-tree, which is more compute-intensive at red and memory-intensive at blue. Give a set of requests, preprocess ahead of GPU running is performed, which consists of prefix tree construction, output length sampling, and transformation including tree sorting and node splitting (①,②, and ③). Then dual scanner forms the runtime batch from most compute- and memory-intensive nodes, which is consumed by the backend engine (④). Preprocess consumes less than 1% time compared to the end-to-end process, which is negligible.

the resource usage of the actual request. For an internal node, the compute density is calculated over all requests within the sub-tree rooted at it. The larger the compute density is, the more compute-intensive these requests are.

Output length sampling for compute density. Request output length is needed for calculating compute density, and it is the key factor in determining whether a request is compute- or memory-intensive, as memory demands increase quadratically with output length. However, output length is unknown beforehand; therefore, precise estimation of output length is critical for accurate compute density calculation. Instead of relying on statistical analysis of input and output length or model-based prediction that incurs overhead, our key insight is that output length is more related to the prompt semantics rather than its length (Fu et al., 2024; Stojkovic et al., 2024). For example, benchmark requests of MMLU all have an output length of 2 tokens (Hendrycks et al., 2021), while chatbot requests of ShareGPT follow a unimodal distribution (Chen et al., 2023; ShareGPT, 2023).

In the prefix tree, requests sharing similar prompts are naturally grouped under specific sub-trees. These requests have a similar distribution of output length due to their shared semantics of similar prefixes. To estimate output length, BlendServe employs a sampling strategy, where a subset of requests is selected with a sampling probability p. Each sub-tree uses the average output length of its sampled requests as an estimation for unsampled requests within the same sub-tree. If a sub-tree t_1 is not sampled at all, it will adopt the average sampled output length of its sibling sub-tree t_2 since t_1 and t_2 share the longest common prefix and tend to have a similar distribution of output length.

Notably, image- or video-generation requests inherently have a known output length due to the preset quality and frame parameters (Liu et al., 2024; Lu et al., 2022), which simplifies the estimation process. In our evaluation, we sampled 1% of the total requests and found that this approach can achieve performance comparable to full sampling.

3.3 Resource-aware prefix tree sorting

Now we can sort the prefix tree based on each node's compute density value. BlendServe performs a layer-wise sorting of nodes based on their compute density in descending order, which only reorders nodes within the same depth and ancestor as shown in Algorithm 1. This sorting maintains the hierarchical structure of the prefix tree. After sorting, the tree exhibits a global pattern with compute-intensive nodes on the left and memory-intensive nodes on the right. However, local outliers that do not conform to this trend may still exist. For example, in the first tree of Figure 4, request #2 which has low compute density should be separated from requests #1 and #3 and repositioned to the right.

To address this issue, BlendServe introduces a conditional node splitting technique as shown in Algorithm 2, to relocate outliers to new nodes, e.g., request #2 gets relocated from the leftmost to the rightmost. This technique employs a heuristic threshold t. If the potential recomputation waste for the relocation is below t, the node is repositioned to adhere to the descending order of compute density. This leverages a trade-off between prefix sharing and request-level resource overlap by sacrificing a tolerable amount of prefix sharing for better resource balance.

3.4 Request order search: heuristic dual scanner

Finally, BlendServe must derive a request order from the prefix tree, ensuring a high prefix sharing ratio and balanced compute and memory usage across requests for overlapping.

Algorithm 1 Layer-wise sorting

Function layer_sort(ptr: tree node)
if ptr is not leaf node **then**
  Sort $ptr.childList$ based on compute density
  for $cptr$ in $ptr.childList$ **do**
    Call layer_sort($cptr$)
  end for
end if
EndFunction

Algorithm 2 Node splitting

Initialize $leaf\_list \leftarrow \{\}$
Function node_split(ptr: tree node, t: threshold)
$ptr.len_{prefix} \leftarrow$ length of prefix to ptr
if $ptr.len_{prefix} \cdot len(ptr.childList) > t$ **then**
  Append ptr to $leaf\_list$
else
  for $cptr$ in $ptr.childList$ **do**
    Call node_split($cptr$, $t/len(ptr.childList)$)
  end for
end if
if ptr is root node **then**
  Sort $leaf\_list$ based on compute density
end if
EndFunction

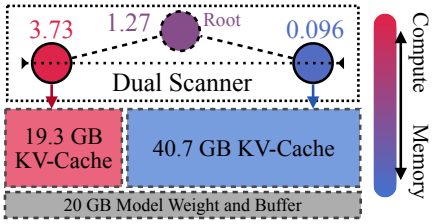

Figure 5. Example of memory partition on Llama-3.1-8B with an A100 80GB. The left node has requests with 512 input length and 256 output length which lead to a compute density of 3.73, while the right node is memory-intensive with 256 input length and 16384 output length. The dual scanner will reserve 20GB for model weights and temporary buffers, then partition the rest of memory according to the compute density.

However, searching for an optimal request order is NP-hard. For each scheduling step, the search problem can be reduced to a Knapsack Problem (Cacchiani et al., 2022) where a set of requests of different lengths fills the GPU memory for maximal utilization. Furthermore, since requests need to be decoded in multiple steps due to the auto-regressive inference, scheduling in different steps is dependent, further complicating the problem. Given the large scale of request numbers and scheduling steps, the optimal solution is hard to solve in a useful time, necessitating an approximation.

To address this problem more efficiently, BlendServe employs a heuristic algorithm by scanning the leaf nodes of the prefix tree concurrently from left to right and right to left with a dual scanner, progressively adding requests into the on-the-fly batch from two leaf nodes. To determine how many requests should be selected from compute-intensive node R_L and memory-intensive node R_R, BlendServe logically partitions the GPU memory M into two parts M_L and M_R, where the partition sizes M_L and M_R are dynamically calculated by the following theoretical modeling:

$$\begin{cases} M_L + M_R = M & \text{(Memory)} \\ M_L \cdot \rho(R_L) + M_R \cdot \rho(R_R) = M \cdot \rho(rt) & \text{(Compute)} \end{cases}$$

These two equations represent the memory and compute demands, respectively. Here, M is a constant denoting GPU memory size. $\rho(rt)$ is the compute density of the tree root node, remaining constant for the current request set. Similarly, $\rho(R_L)$ and $\rho(R_L)$ are the compute densities of the compute- and memory-intensive nodes, which are also constants when the scanner reaches a specific node. Thus, M_L and M_R can be derived from these two equations. This memory partition ensures that the compute density of the blended compute- and memory-intensive requests approximates $\rho(rt)$, allowing the memory access time to be fully overlapped with the compute time (when $\rho(rt) > 1$).

Within each memory partition, BlendServe adopts a regular scheduler used by prior works (Zhu et al., 2024; Agrawal et al., 2024), to manage the admission, retraction, and competence of requests. Given the assigned memory size, the scheduler calculates the desired on-the-fly batch size and feeds requests via continuous batching (Yu et al., 2022). For example, when processing two nodes with compute density of 3.73 and 0.096 as shown in Figure 5, the dual scanner will logically split available GPU memory into 19.3GB and 40.7GB based on the target compute density of 1.27. Note that we can add more memory-intensive requests here; as long as the compute density of the blended requests remains close to one, we can overlap compute and memory access time. However, this may cause fluctuations in subsequent batches (more compute-intensive in this example). To maintain a stable compute density and resource overlap ratio, we consistently target $\rho(rt)$ for the blended requests.

This strategy also ensures high prefix sharing ratio ($> 99\%$ compared to optimal sharing (§ 5.3)), as the dual scanning method essentially performs two local DFS orders over the left and right subtrees concurrently. Prior work has demonstrated that a global DFS can achieve the upper bound of the prefix sharing ratio due to the minimal memory footprint as only one path needs to be cached for reuse (Zheng et al., 2024b). Compared to the global DFS over the entire tree, introducing two local DFS orders only leads to extra memory for a single path, which is typically much smaller compared to GPU memory. For example, while one A100 80GB can

hold nearly 500K tokens for an Llama-3.1-8B, the longest shared prefix in MMLU is less than 1K tokens. Therefore, our dual scanning sacrifices minimal prefix sharing benefits.

4 IMPLEMENTATION

We introduce additional noteworthy details of our implementation in BlendServe here. We will release our system prototype to the public upon acceptance.

Offline prefix tree. We preprocess all requests and construct a prefix tree following a Trie Tree to capture their shared prefixes before serving. After compute density calculation and node sorting, we merge sub-trees into single nodes if doing so does not hurt the prefix sharing ratio. This merging reduces fragmentation that would cause fluctuation during the dual scanner process.

Runtime prefix tree. The runtime prefix tree in BlendServe is implemented based on SGLang (Zheng et al., 2024b). It manages runtime information related to prefix sharing, including a dynamic Trie Tree and a mapping between the physical memory and key-value tokens. We also employ intra-batch prefix sharing, enabling exactly-once computation of shared prefixes for a single batch, which is particularly beneficial for offline processing using a DFS order.

Batch scheduler. The batch scheduler within the dual scanner is implemented following NanoFlow (Zhu et al., 2024). It strictly enforces batch sizes in multiples of 128 to ensure higher hardware utilization. We also incorporate chunked prefill and continuous batching following state-of-the-art serving systems (Agrawal et al., 2024; Yu et al., 2022).

Backend engine. Our backend engine is built in C++ following NanoFlow's operator-level overlapping approach, which enables simultaneous execution of compute-intensive operators like GEMM and memory-intensive operators like self-attention (Zhu et al., 2024). Based on the operator-level overlapping, BlendServe overlaps operators from requests with distinct resource usages.

5 EVALUATION

5.1 Experiment setup

Workload synthesizing. To the best of our knowledge, there is no open-sourced trace available for offline batch inference on multi-modal models. Instead, we synthesize our workloads by combining existing well-known single-modal traces, including WildChat (Zhao et al., 2024a) of OpenAI GPT4 chatbot traces, BurstGPT (Wang et al., 2024b) of ChatGPT workloads from Azure API, ShareGPT (ShareGPT, 2023), Azure-Trace (Stojkovic et al., 2024) collected from Azure's production LLM workloads,

	High Prefix Sharing	Low Prefix Sharing
Compute-intensive	Trace#1 (1.4, 35%)	Trace#3 (1.4, 5%)
Memory-intensive	Trace#2 (0.9, 35%)	Trace#4 (0.9, 5%)

Table 2. Four representative synthesized workloads. Trace#X (A,B%) has a compute density of A, with a prefix sharing ratio of B%. For example, Trace#1 is compute-intensive with high prefix sharing, which has a compute density of 1.4 larger than 1 and a prefix sharing ratio of 35%. Note that 35% is a high prefix sharing ratio as most workloads have less than 20% as shown in Table 3. Without losing genericity, Figure 9 shows more trace combinations and reports BlendServe's performance on them.

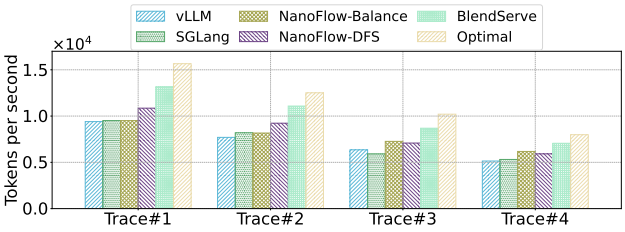

Figure 6. End-to-end throughput comparison (GPU-time only). BlendServe consistently outperforms baselines at all settings, with an average 20.84% speedup compared to the state-of-the-art baseline, NanoFlow-DFS. Notably, BlendServe achieves 86.55% of optimal throughput on average.

OpenVid (Nan et al., 2024) of video generation traces[2], and MMLU (Hendrycks et al., 2021) of model benchmark traces. Figure 2 illustrates the length distributions of individual traces. These single-modal traces have different representative characteristics: WildChat, ShareGPT, BurstGPT, and Azure-Trace requests are highly compute-intensive, OpenVid requests are memory-intensive, and MMLU requests have high prefix sharing. Therefore, by combining different ratios of traces, we could synthesize a variety of multi-modal workloads with different prefix sharing ratio and compute density, based on which we demonstrate the effectiveness and generality of proposed BlendServe. Detailed methodology of synthetic workloads is described in Appendix § A.1.

Table 2 shows the four most representative workloads we mainly use in evaluation, which have different resource demands and prefix sharing ratio. Each synthesized workload is made from BurstGPT, MMLU, OpenVid and contains at least $400,000$ requests, which require 3 to 6 A100 GPU hours and are large enough to reach a stable performance. Evaluation results on more ratios are presented in § 5.4. We also present results with more source traces including ShareGPT, WildChat, and Azure-Trace in § A.2.

Models and hardware configurations. As we focus on

[2]We calculate the output length of a video generation request using the frames and quality of the videos in OpenVid.

improving the throughput of transformer-based backbone models, we evaluate one of the most widely-used models, Llama-3.1-8B (Meta-Team, 2024). Other language models such as Mixtral (Jiang et al., 2024), DeepSeek (DeepSeek-Team, 2024), and Qwen (Bai et al., 2023), along with auto-regressive multi-modal models like EMU3 (Wang et al., 2024a) and VILA-U (Wu et al., 2024c), all share a similar backbone architecture. For hardware, we use one of the frontier data center GPUs, A100 80GB SXM, to serve the single Llama-3.1-8B (NVIDIA, 2020). We argue that our proposed method can be extended to distributed settings including both tensor and data parallelism for large-scale serving in § 6.1 (Shoeybi et al., 2020).

Baseline frameworks. We use two widely used industry standard frameworks, vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024b), and a recently released throughput-oriented framework, NanoFlow (Zhu et al., 2024)[3]. We do not evaluate frameworks that are designed for resource-constrained settings, e.g., FlexGen (Sheng et al., 2023) and HeteGen (Zhao et al., 2024b). For vLLM and SGLang, we enable prefix caching for both and reorder each workload trace into a DFS order, which can achieve a high prefix sharing ratio. For NanoFlow, we add prefix caching support for fair comparison. For each workload trace, we evaluate the performance of NanoFlow using both DFS (NanoFlow-DFS) and random ordering (NanoFlow-Balance). The improvement of BlendServe over NanoFlow-DFS would demonstrate the advantage of achieving better request-level resource balance, while the improvement over NanoFlow-Balance would highlight the benefit of a higher prefix sharing ratio. We use the max sampler and do not enable the CUDA graph for all baselines and BlendServe. We adopt a 0.01 sampling probability p for all evaluations of BlendServe without explicitly mentioning.

As BlendServe focuses on improving GPU utilization, we do not measure CPU time including tokenizations, metadata transferring (to the GPU), and scheduling (Srivatsa et al., 2024b). We discuss the CPU overhead in § 5.5.

5.2 End-to-end throughput

Compared to existing frameworks. We measure the end-to-end throughput of BlendServe and all baselines, including vLLM-DFS, SGLang-DFS, NanoFlow-Balance, and NanoFlow-DFS. We define end-to-end throughput as all processed tokens (including both input and output tokens) divided by the total processing time. As shown in Figure 6, with a small prefix sharing ratio (i.e., Trace#3 and #4), NanoFlow-Balance works better than NanoFlow-DFS since resource overlapping contributes to more throughput gain. However, with a large prefix sharing ratio, NanoFlow-DFS achieves state-of-the-art throughput in existing frameworks due to its operator-level resource overlapping and high prefix sharing ratio. Since BlendServe is designed to leverage the best of both, it consistently outperforms the best baseline, NanoFlow-DFS, in all settings from 19.34% to 22.65%. Compared with vLLM-DFS, BlendServe achieves up to $1.44\times$ throughput speedup.

[3]We use vLLM v0.6.3.post2.dev102 (commit: e26d37a1) and SGLang v0.3.4.post1 (commit: 3f5ac88).

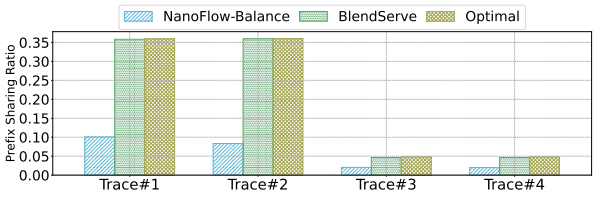

Figure 7. Prefix sharing ratio of four selected traces in the end-to-end evaluation. Note that the optimal value is measured via a DFS order of the prefix tree. BlendServe consistently maintains the benefit of prefix sharing, achieving 97% of optimal values.

Performance gap between optimal throughput. To assess how closely BlendServe's throughput approaches the optimal, we calculate optimal throughput with T_o defined in § 2.4. Due to the well-known performance interference issue in GPU hardware during spatial sharing (Zhu et al., 2024; Strati et al., 2024), deriving T_o with $\max(T_{comp}, T_{mem})$ is both impractical and unachievable. Therefore, to estimate an *achievable optimal throughput* on the A100 GPU hardware, we first use NanoFlow's implementation of operator-level spatial sharing to profile the performance interference, and then we apply a polynomial fitting to get an achievable T_o.

As shown in Figure 6, BlendServe achieves an average 86.55% of the achievable optimal throughput, leaving only a 13% performance gap. This performance gap is mainly caused by two reasons: (1) the imperfect output length estimations, and (2) the heuristics-based request order search. Nevertheless, compared to state-of-the-art serving frameworks, BlendServe is much closer to the optimal, demonstrating its effectiveness in achieving both high prefix sharing ratio and request-level resource overlapping.

5.3 Performance analysis

We now analyze the key factors contributing to BlendServe's performance improvement by showing prefix sharing ratio and hardware resource usage over time, corresponding to the two key design points introduced in § 2.4.

Prefix sharing ratio. To illustrate that BlendServe can achieve nearly optimal prefix sharing ratio, we collect the achieved prefix sharing ratio along with the optimal values. We manually exclude prefix sharing related to the recomputation of retracted requests. As shown in Figure 7, BlendServe achieves over 97% of the optimal prefix sharing

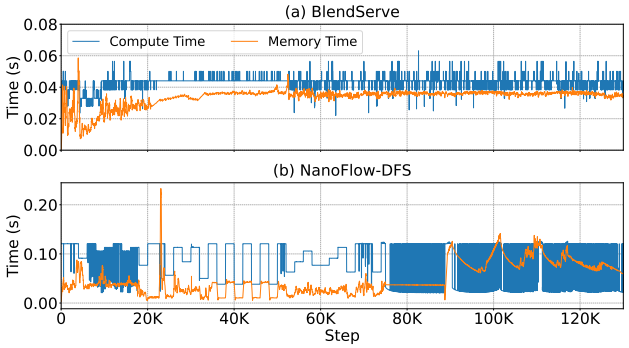

Figure 8. Compute and memory usages along with first 130K inference steps on Trace#2. Large variance of output length causes substantial fluctuations in NanoFlow-DFS, incurring under-utilization of at least one resource at each step.

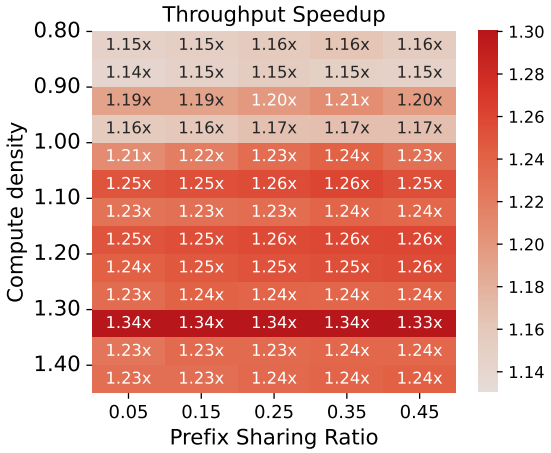

Figure 9. Simulated throughput improvement of BlendServe compared to NanoFlow-DFS on workloads synthesized from BurstGPT, MMLU, and OpenVid. We use different numbers of requests from these traces to compose workloads with different compute density and prefix sharing ratio. BlendServe consistently surpasses baselines, with an average of 1.23× throughput speedup.

ratio, while NanoFlow-Balance fails below 30%. Besides, BlendServe demonstrates an average 1.36× throughput improvement compared to NanoFlow-Balance with Trace#1 and #2, highlighting the practical usage of prefix sharing.

Hardware resource usage. To demonstrate how effectively BlendServe balances resource usage for overlapping, we visualize the compute and memory usage of BlendServe and NanoFlow-DFS in Figure 8. We select Trace#2 due to its intensive memory usage, which leads to significant request-level resource imbalance. For each step, we collect the input batch sizes of GEMM and memory loading size of active KV-cache, which are used to calculate compute and memory time based on the cost model in § 2.3. BlendServe maintains stable compute and memory usage, whereas NanoFlow-DFS exhibits significant fluctuations, resulting in resource under-utilization. To maximize tensor core utilization, BlendServe schedules batch sizes in multiples of 128, which needs to alternately round up/down to achieve the desired batch size by amortization, leading to slight fluctuations.

5.4 Sensitivity study

To demonstrate the generality of BlendServe in real-world scenarios, we then evaluate on more diverse synthetic workloads, with a large range of compute density and prefix sharing ratio. In addition to the four most representative workloads shown in Table 2, we conduct a grid search of compute density from 0.80 to 1.40 and prefix sharing ratio from 0.05 to 0.45 with step sizes 0.05 and 0.10, respectively. In total, we synthesize 65 traces and evaluate BlendServe and the best-performed baseline NanoFlow-DFS on these traces. Due to limited GPU resources, we use the frontend scheduler of BlendServe to generate practical batch schedules that are the same as running on real GPUs, which are then fed into a *simulated GPU backend* to get the estimated inference time. For the backend simulation, we use polynomial fit to estimate the GPU runtime when given a certain amount of compute and memory usage. Results show that only a 0.91% difference exists between the real and simulation speedup over the four representative workloads on average. Therefore, our simulation results practically reflect real performance.

As shown in Figure 9, BlendServe consistently outperforms the baseline in all workloads by 14% to 34%, with an average speedup of 22.53%, demonstrating the generality of the proposed method. Since both BlendServe and NanoFlow-DFS can achieve the highest prefix sharing ratio with DFS ordering, the inference throughput remains stable when varying different prefix sharing ratio. However, we find that the advantages of BlendServe from resource overlapping tend to shrink with smaller compute density, which is caused by the larger GPU interference with memory-intensive workloads. At the same time, the relative speedup achieves the maximal 1.34× when the compute density is around 1.30, potentially because 1.30 is the best resource ratio for overlapping.

5.5 Scheduling overhead of BlendServe

As described in § 3, BlendServe has two scheduling overhead: 1) preprocessing all token ids of requests prompt to construct the prefix tree, followed by a series of tree transformations; and 2) runtime scheduling request batches based on the double scanner algorithm and the prefix tree to manage KV-cache memory. We now demonstrate that these two parts have minimal overhead compared to the GPU time.

Preprocessing overhead. There is no additional overhead for tokenization, since it is also necessary for model infer-

ence, and the storage for generated token ids is at the same magnitude as the input strings. Assuming N requests with T tokens in the prompts, for the trie tree construction with D max depth, the time complexity $O(N \times D)$. Since requests' prompts diverge quickly, D is typically small. In our evaluations, this process typically takes several minutes, which is negligible compared to hours of GPU inference.

Runtime scheduling overhead. Since the runtime batch size is typically at the magnitude of thousands, the runtime prefix tree is much smaller compared to the offline prefix tree built during preprocessing. Based on our measurement in evaluations, the operations on the runtime prefix tree take 0.08 ms on average, with a P99 latency of 0.23 ms, which is generally less than 10% compared to the GPU time. Such small runtime scheduling overhead can be effectively overlapped with asynchronous CPU scheduling, incurring zero overhead in end-to-end performance (Zhu et al., 2024).

6 Discussion

6.1 Extend BlendServe to distributed settings

Tensor parallelism. Although BlendServe is primarily evaluated on an 8B model and a single GPU, it can be easily extended to distributed settings with larger models. Previous studies have demonstrated that communication operators involved in the tensor parallelism can be overlapped via specific operator-level pipeline designs (Zhu et al., 2024; Chen et al., 2024). This approach is orthogonal to our request-level scheduling and feasible to be integrated.

Data parallelism. Given a large request pool that needs data parallelism (DP) for parallel computing, BlendServe is feasible to split the global prefix tree into parallelized trees, without compromising resource overlapping and prefix sharing. Following the double scanner design, BlendServe could split continuous requests from both sides of the prefix tree to different DP nodes. Each block of requests has the same amount of resource usage and maintains the same compute density as the root node. Since only the prompts along the split paths are recalculated, the waste of prefix sharing is linear to the number of DP nodes, making it negligible.

6.2 Cascade attention kernels

Recent progress in *Cascade Attention* further enhances the memory efficiency of prefix sharing by reusing the KV-cache access shared by the common prefix (Ye et al., 2024; Juravsky et al., 2024). We can integrate it into BlendServe by changing the formulation of memory into $(1-s) \cdot T_{mem}$ (§ 3.2) and adopting cascade backend. However, this requires a significant amount of engineering efforts to tune the kernel performance. For example, at least 128 shared tokens are needed so that the benefit of saving memory bandwidth is larger than the kernel launch overhead. We leave the trade-off exploration between memory bandwidth saving and prefix sharing as future work.

7 Related Work

LLM serving optimizations. Efficient LLM serving has been extensively studied for both online and offline scenarios. For online scenarios, Orca (Yu et al., 2022), vLLM (Kwon et al., 2023), SGLang (Zheng et al., 2024b), DistServe (Zhong et al., 2024), FastServe (Wu et al., 2024a), and VTC (Sheng et al., 2024) propose continuous batching, paged attention, prefix sharing, prefill-decode disaggregation, Multi-Level Feedback Queue scheduling, and Virtual Token Counter scheduling, respectively, to improve performance and/or fairness. For offline scenarios, FlexGen (Sheng et al., 2023), PowerInfer (Song et al., 2023), TwinPilots (Yu et al., 2024), HeteGen (Zhao et al., 2024b), Fiddler (Kamahori et al., 2024), and FastDecode (He & Zhai, 2024) extensively leverage CPUs to offload model weights, activations, KV-cache, and computation. Different from all of them, BlendServe focuses on optimizing GPU-only offline inference with resource-aware batching.

Resource overlapping techniques. Prior works have extensively explored resource overlapping techniques to enhance GPU hardware utilization. Rammer (Ma et al., 2020) introduces operator-level overlapping techniques for deep neural networks. NanoFlow (Zhu et al., 2024) extends operator-level overlapping to LLM serving. Sarathi-Serve (Agrawal et al., 2024) and FastGen (Holmes et al., 2024) apply phase-level overlapping to LLM serving. MuxServe (Duan et al., 2024) collocates models based on their popularity and resource characteristics, targeting resource-limited scenarios. ConServe (Qiao et al., 2024) collocates offline requests with online requests to increase the hardware utilization. Compared to them, BlendServe is the first to exploit request-level resource overlapping with request reordering.

8 Conclusion

We analyze the characteristics of offline serving workloads for LLMs and find that requests exhibit diverse compute and memory demands, revealing the opportunity for resource overlapping. We then build BlendServe, an offline batch inference system that maximizes hardware utilization by overlapping requests with distinct resource usages while maintaining a high prefix sharing ratio. BlendServe exploits the relaxed SLOs in offline batch inference to reorder requests through a resource-aware prefix tree and a dual scanner searching algorithm. BlendServe achieves 19.34% to 22.65% throughput gains over state-of-the-art research systems, and up to $1.44\times$ speedup over vLLM and SGLang.

REFERENCES

Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., Tumanov, A., and Ramjee, R. Taming throughput-latency tradeoff in llm inference with sarathi-serve, 2024. URL https://arxiv.org/abs/2403.02310.

Allal, L. B., Lozhkov, A., and van Strien, D. Cosmopedia: how to create large-scale synthetic data for pre-training Large Language Models — huggingface.co. https://huggingface.co/blog/cosmopedia, 2024. [Accessed 25-10-2024].

Anthropic. Introducing the Message Batches API — anthropic.com. https://www.anthropic.com/news/message-batches-api, 2024. [Accessed 20-10-2024].

Anyscale. LLM offline batch inference with Ray Data and vLLM — Anyscale Docs — docs.anyscale.com. https://docs.anyscale.com/examples/batch-llm/, 2024. [Accessed 26-10-2024].

AWS. Supported Regions and models for batch inference - Amazon Bedrock — docs.aws.amazon.com. https://docs.aws.amazon.com/bedrock/latest/userguide/batch-inference-supported.html, 2024. [Accessed 26-10-2024].

Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., Fan, Y., Ge, W., Han, Y., Huang, F., Hui, B., Ji, L., Li, M., Lin, J., Lin, R., Liu, D., Liu, G., Lu, C., Lu, K., Ma, J., Men, R., Ren, X., Ren, X., Tan, C., Tan, S., Tu, J., Wang, P., Wang, S., Wang, W., Wu, S., Xu, B., Xu, J., Yang, A., Yang, H., Yang, J., Yang, S., Yao, Y., Yu, B., Yuan, H., Yuan, Z., Zhang, J., Zhang, X., Zhang, Y., Zhang, Z., Zhou, C., Zhou, J., Zhou, X., and Zhu, T. Qwen technical report, 2023. URL https://arxiv.org/abs/2309.16609.

Cacchiani, V., Iori, M., Locatelli, A., and Martello, S. Knapsack problems — an overview of recent advances. part i: Single knapsack problems. *Computers & Operations Research*, 143:105692, 2022. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2021.105692. URL https://www.sciencedirect.com/science/article/pii/S0305054821003877.

Chen, C., Li, X., Zhu, Q., Duan, J., Sun, P., Zhang, X., and Yang, C. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, pp. 178–191, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703867. doi: 10.1145/3620666.3651379. URL https://doi.org/10.1145/3620666.3651379.

Chen, L., Ye, Z., Wu, Y., Zhuo, D., Ceze, L., and Krishnamurthy, A. Punica: Multi-tenant lora serving, 2023. URL https://arxiv.org/abs/2310.18547.

Databricks. Introducing Simple, Fast, and Scalable Batch LLM Inference on Mosaic AI Model Serving — databricks.com. https://www.databricks.com/blog/introducing-simple-fast-and-scalable-batch-llm-in 2024. [Accessed 26-10-2024].

DeepSeek-Team. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024. URL https://arxiv.org/abs/2405.04434.

Duan, J., Lu, R., Duanmu, H., Li, X., Zhang, X., Lin, D., Stoica, I., and Zhang, H. Muxserve: Flexible spatial-temporal multiplexing for multiple llm serving, 2024. URL https://arxiv.org/abs/2404.02015.

Fu, Y., Zhu, S., Su, R., Qiao, A., Stoica, I., and Zhang, H. Efficient llm scheduling by learning to rank, 2024. URL https://arxiv.org/abs/2408.15792.

He, J. and Zhai, J. Fastdecode: High-throughput gpu-efficient llm serving using heterogeneous pipelines, 2024. URL https://arxiv.org/abs/2403.11421.

Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., and Steinhardt, J. Measuring massive multitask language understanding, 2021. URL https://arxiv.org/abs/2009.03300.

Holmes, C., Tanaka, M., Wyatt, M., Awan, A. A., Rasley, J., Rajbhandari, S., Aminabadi, R. Y., Qin, H., Bakhtiari, A., Kurilenko, L., and He, Y. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference, 2024. URL https://arxiv.org/abs/2401.08671.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., de las Casas, D., Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M.-A., Stock, P., Subramanian, S., Yang, S., Antoniak, S., Scao, T. L., Gervet, T., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mixtral of experts, 2024. URL https://arxiv.org/abs/2401.04088.

Juravsky, J., Brown, B., Ehrlich, R., Fu, D. Y., Ré, C., and Mirhoseini, A. Hydragen: High-throughput llm inference with shared prefixes, 2024. URL https://arxiv.org/abs/2402.05099.

Kamahori, K., Gu, Y., Zhu, K., and Kasikci, B. Fiddler: Cpu-gpu orchestration for fast inference of mixture-of-experts models, 2024. URL https://arxiv.org/abs/2402.07033.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

Lin, C., Han, Z., Zhang, C., Yang, Y., Yang, F., Chen, C., and Qiu, L. Parrot: Efficient serving of llm-based applications with semantic variable, 2024. URL https://arxiv.org/abs/2405.19888.

Lin, J., Tang, J., Tang, H., Yang, S., Dang, X., and Han, S. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

Liu, H., Yan, W., Zaharia, M., and Abbeel, P. World model on million-length video and language with blockwise ringattention, 2024. URL https://arxiv.org/abs/2402.08268.

LMDeploy. Lmdeploy: A toolkit for compressing, deploying, and serving llm. https://github.com/InternLM/lmdeploy, 2023.

Lu, J., Clark, C., Zellers, R., Mottaghi, R., and Kembhavi, A. Unified-io: A unified model for vision, language, and multi-modal tasks, 2022. URL https://arxiv.org/abs/2206.08916.

Lu, J., Clark, C., Lee, S., Zhang, Z., Khosla, S., Marten, R., Hoiem, D., and Kembhavi, A. Unified-io 2: Scaling autoregressive multimodal models with vision, language, audio, and action, 2023. URL https://arxiv.org/abs/2312.17172.

Ma, L., Xie, Z., Yang, Z., Xue, J., Miao, Y., Cui, W., Hu, W., Yang, F., Zhang, L., and Zhou, L. Rammer: Enabling holistic deep learning compiler optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 881–897. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/ma.

Meta-Team. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Microsoft. GitHub Copilot · Your AI pair programmer — github.com. https://github.com/features/copilot, 2023. [Accessed 28-10-2024].

Nan, K., Xie, R., Zhou, P., Fan, T., Yang, Z., Chen, Z., Li, X., Yang, J., and Tai, Y. Openvid-1m: A large-scale high-quality dataset for text-to-video generation, 2024. URL https://arxiv.org/abs/2407.02371.

NVIDIA. Nvidia a100 tensor core gpu architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020. [Accessed 25-10-2024].

OpenAI. Introducing chatgpt. https://openai.com/index/chatgpt/, 2022. [Accessed 20-10-2024].

OpenAI. Introducing openai gpt4-o1-preview. https://openai.com/index/introducing-openai-o1-preview/, 2024a. [Accessed 20-10-2024].

OpenAI. Introducing batch api. https://platform.openai.com/docs/guides/batch, 2024b. [Accessed 20-10-2024].

Patel, P., Choukse, E., Zhang, C., Shah, A., Íñigo Goiri, Maleki, S., and Bianchini, R. Splitwise: Efficient generative llm inference using phase splitting, 2024. URL https://arxiv.org/abs/2311.18677.

Qiao, Y., Anzai, S., Yu, S., Ma, H., Wang, Y., Kim, M., and Xu, H. Conserve: Harvesting gpus for low-latency and high-throughput large language model serving, 2024. URL https://arxiv.org/abs/2410.01228.

Qin, Y., Li, X., Zou, H., Liu, Y., Xia, S., Huang, Z., Ye, Y., Yuan, W., Liu, H., Li, Y., and Liu, P. O1 replication journey: A strategic progress report – part 1, 2024. URL https://arxiv.org/abs/2410.18982.

ShareGPT. Sharegpt. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered, 2023.

Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Fu, D. Y., Xie, Z., Chen, B., Barrett, C., Gonzalez, J. E., Liang, P., Ré, C., Stoica, I., and Zhang, C. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023. URL https://arxiv.org/abs/2303.06865.

Sheng, Y., Cao, S., Li, D., Zhu, B., Li, Z., Zhuo, D., Gonzalez, J. E., and Stoica, I. Fairness in serving large language models, 2024. URL https://arxiv.org/abs/2401.00588.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020. URL https://arxiv.org/abs/1909.08053.

Snell, C., Lee, J., Xu, K., and Kumar, A. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Song, Y., Mi, Z., Xie, H., and Chen, H. Powerinfer: Fast large language model serving with a consumer-grade gpu, 2023. URL https://arxiv.org/abs/2312.12456.

Srivatsa, V., He, Z., Abhyankar, R., Li, D., and Zhang, Y. Preble: Efficient distributed prompt scheduling for llm serving, 2024a. URL https://arxiv.org/abs/2407.00023.

Srivatsa, V., Li, D., Zhang, Y., and Abhyankar, R. MLSys @ WukLab - Can Scheduling Overhead Dominate LLM Inference Performance? A Study of CPU Scheduling Overhead on Two Popular LLM Inference Systems — mlsys.wuklab.io. https://mlsys.wuklab.io/posts/scheduling_overhead/, 2024b. [Accessed 25-10-2024].

Stojkovic, J., Zhang, C., Íñigo Goiri, Torrellas, J., and Choukse, E. Dynamollm: Designing llm inference clusters for performance and energy efficiency, 2024. URL https://arxiv.org/abs/2408.00741.

Strati, F., Ma, X., and Klimovic, A. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, pp. 1075–1092, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704376. doi: 10.1145/3627703.3629578. URL https://doi.org/10.1145/3627703.3629578.

Tang, J., Zhao, Y., Zhu, K., Xiao, G., Kasikci, B., and Han, S. Quest: Query-aware sparsity for efficient long-context llm inference, 2024. URL https://arxiv.org/abs/2406.10774.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

Wang, X., Zhang, X., Luo, Z., Sun, Q., Cui, Y., Wang, J., Zhang, F., Wang, Y., Li, Z., Yu, Q., Zhao, Y., Ao, Y., Min, X., Li, T., Wu, B., Zhao, B., Zhang, B., Wang, L., Liu, G., He, Z., Yang, X., Liu, J., Lin, Y., Huang, T., and Wang, Z. Emu3: Next-token prediction is all you need, 2024a. URL https://arxiv.org/abs/2409.18869.

Wang, Y., Chen, Y., Li, Z., Kang, X., Tang, Z., He, X., Guo, R., Wang, X., Wang, Q., Zhou, A. C., and Chu, X. Burstgpt: A real-world workload dataset to optimize llm serving systems, 2024b.

Wang, Z., Zhu, K., Xu, C., Zhou, W., Liu, J., Zhang, Y., Wang, J., Shi, N., Li, S., Li, Y., Que, H., Zhang, Z., Zhang, Y., Zhang, G., Xu, K., Fu, J., and Huang, W. Mio: A foundation model on multimodal tokens, 2024c. URL https://arxiv.org/abs/2409.17692.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

Williams, S., Waterman, A., and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL https://doi.org/10.1145/1498765.1498785.

Wu, B., Zhong, Y., Zhang, Z., Liu, S., Liu, F., Sun, Y., Huang, G., Liu, X., and Jin, X. Fast distributed inference serving for large language models, 2024a. URL https://arxiv.org/abs/2305.05920.

Wu, C., Chen, X., Wu, Z., Ma, Y., Liu, X., Pan, Z., Liu, W., Xie, Z., Yu, X., Ruan, C., and Luo, P. Janus: Decoupling visual encoding for unified multimodal understanding and generation, 2024b. URL https://arxiv.org/abs/2410.13848.

Wu, Y., Zhang, Z., Chen, J., Tang, H., Li, D., Fang, Y., Zhu, L., Xie, E., Yin, H., Yi, L., Han, S., and Lu, Y. Vila-u: a unified foundation model integrating visual understanding and generation, 2024c. URL https://arxiv.org/abs/2409.04429.

Xue, F., Chen, Y., Li, D., Hu, Q., Zhu, L., Li, X., Fang, Y., Tang, H., Yang, S., Liu, Z., He, E., Yin, H., Molchanov, P., Kautz, J., Fan, L., Zhu, Y., Lu, Y., and Han, S. Longvila: Scaling long-context visual language models for long videos, 2024. URL https://arxiv.org/abs/2408.10188.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models, 2023. URL https://arxiv.org/abs/2305.10601.

Ye, Z., Lai, R., Lu, B.-R., Lin, C.-Y., Zheng, S., Chen, L., Chen, T., and Ceze, L. Cascade inference: Memory bandwidth efficient shared prefix batch decoding, February 2024. URL https://flashinfer.ai/2024/02/02/cascade-inference.html.

Yu, C., Wang, T., Shao, Z., Zhu, L., Zhou, X., and Jiang, S. Twinpilots: A new computing paradigm for gpu-cpu parallel llm inference. In *Proceedings of the 17th ACM*

International Systems and Storage Conference, pp. 91–103, 2024.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Zhao, W., Ren, X., Hessel, J., Cardie, C., Choi, Y., and Deng, Y. Wildchat: 1m chatgpt interaction logs in the wild, 2024a. URL https://arxiv.org/abs/2405.01470.

Zhao, X., Jia, B., Zhou, H., Liu, Z., Cheng, S., and You, Y. Hetegen: Heterogeneous parallel inference for large language models on resource-constrained devices, 2024b. URL https://arxiv.org/abs/2403.01164.

Zhao, Y., Lin, C.-Y., Zhu, K., Ye, Z., Chen, L., Zheng, S., Ceze, L., Krishnamurthy, A., Chen, T., and Kasikci, B. Atom: Low-bit quantization for efficient and accurate llm serving, 2024c. URL https://arxiv.org/abs/2310.19102.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., and Stoica, I. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023. URL https://arxiv.org/abs/2306.05685.

Zheng, L., Chiang, W.-L., Sheng, Y., Li, T., Zhuang, S., Wu, Z., Zhuang, Y., Li, Z., Lin, Z., Xing, E. P., Gonzalez, J. E., Stoica, I., and Zhang, H. Lmsys-chat-1m: A large-scale real-world llm conversation dataset, 2024a. URL https://arxiv.org/abs/2309.11998.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. Sglang: Efficient execution of structured language model programs, 2024b. URL https://arxiv.org/abs/2312.07104.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024. URL https://arxiv.org/abs/2401.09670.

Zhu, K., Zhao, Y., Zhao, L., Zuo, G., Gu, Y., Xie, D., Gao, Y., Xu, Q., Tang, T., Ye, Z., Kamahori, K., Lin, C.-Y., Wang, S., Krishnamurthy, A., and Kasikci, B. Nanoflow: Towards optimal large language model serving throughput, 2024. URL https://arxiv.org/abs/2408.12757.

A Appendix

A.1 Methodolody of workload synthesize

To synthesize workloads that reflect real use cases, we collect a variety of open-source inference traces that have distinct characterization, including compute density, prefix sharing ratio, and modalities. We illustrate their length distribution in Figure 2. For each set of traces, we add a unique system prompt ahead of prompts as it is not collected. For traces without detailed prompt content, we randomize their prompts' token ids corresponding to their prompt length. For video generation requests, we use OpenVid (Nan et al., 2024) and treat the videos in training datasets as their auto-regressive generation output. For each video, we collect its output length by counting the number of frames and multiplying it by 256, which represents the number of tokens per frame observed in normal videos (Xue et al., 2024; Liu et al., 2024). Additionally, we normalize the average output length of OpenVid to 16K as the original 45K is too large for evaluation of Llama-3.1-8B on a single A100 GPU. We also normalize the average output length of WildChat (Zhao et al., 2024a) to 256 for a more compute-intensive workload while maintaining the length variance. We calculate the resource characterization in Table 3.

	ShareGPT	WildChat	Azure-Trace	OpenVid	BurstGPT	MMLU
Prefix sharing	0.02	0.19	0.01	0.00	0.02	0.86
Compute density	3.12	2.13	33.2	0.05	17.78	54.91

Table 3. Prefix sharing ratio and compute density of collected traces. OpenVid is memory-intensive due to its large output length, while MMLU has a high prefix sharing ratio of 86.46%. Others are compute-intensive with less prefix sharing ratio.

To cover the real cases in offline batch inference, we conduct a grid search of synthetic workloads with different compute density and prefix sharing ratio. To reach the desired compute density t, we combine one compute-intensive trace among ShareGPT, Azure-Trace, WildChat, and BurstGPT, and a memory-intensive video generation trace OpenVid. Based on t and compute density of selected traces, we calculate the required request number of each trace, with a total number of $40,000$ requests. Then we mix requests from MMLU to reach the desired number of prefix sharing ratio to get the synthetic workload. Such a synthetic workload has a diverse request length and various resource characterization, which is similar to real-world cases.

A.2 Extensive evaluation of synthetic workloads

In addition to the main evaluations conducted on BurstGPT, MMLU, and OpenVid in § 5, we also evaluate BlendServe on Azure-Trace (Figure 10), ShareGPT (Figure 11), and WildChat (Figure 12) to demonstrate the generality of proposed methods over diverse workloads, following the same

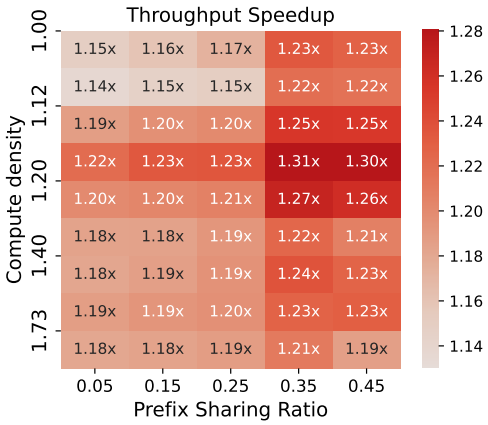

Figure 10. Simulated throughput improvement of BlendServe compared to NanoFlow-DFS on workloads synthesized from Azure-Trace, MMLU, and OpenVid. BlendServe achieves up to 31% throughput gain compared to baselines.

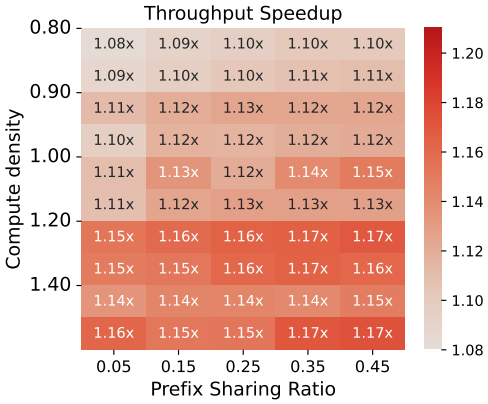

Figure 11. Simulated throughput improvement of BlendServe compared to NanoFlow-DFS on workloads synthesized from ShareGPT, MMLU, and OpenVid. BlendServe consistently surpasses baselines by up to 17% throughput.

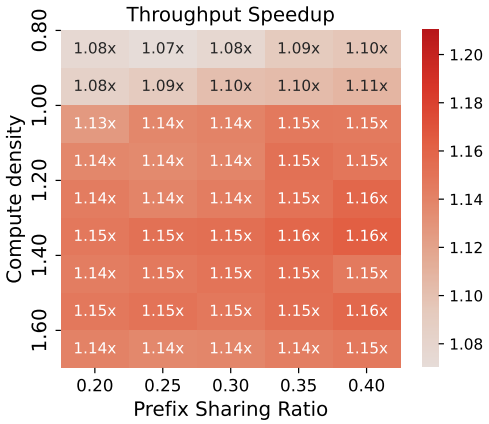

Figure 12. Simulated throughput improvement of BlendServe compared to NanoFlow-DFS on workloads synthesized from WildChat, MMLU, and OpenVid.

experiment setup (§ 5.1).

Results show that BlendServe consistently surpasses baselines by $1.08\times$ to $1.31\times$ in different workloads. We find that BlendServe works better on BurstGPT and Azure-Trace due to their smaller variance of output length. When the output length variance is large in ShareGPT and WildChat, the sampling strategy works less effectively, leading to sub-optimal performance. We leave the better strategy for workloads with large variance output length that cannot be effectively captured by the prefix tree for future work.