Prefill-Only Optimizations for Prefill–Decode Disaggregation in vLLM

Sejal Agarwal [*1] **Maksym Bidnyi** [*1] **Joshua Caiata** [*1] **Gavin Deane** [*1]

Abstract

Disaggregating the prefill and decode steps in Large Language Model (LLM) inference has allowed for optimizing throughput and latency separately. Prior work has shown that hybrid prefilling and Job Completion Time (JCT)-aware scheduling can accelerate prefill-only workloads. This project considers whether these prefill-only optimizations can be used together with disaggregated prefill-decode, and what challenges exist in trying to use prefill-only optimizations in the disaggregated setting. We implement both techniques in vLLM's prefill path and benchmark their performance against the standard disaggregated baseline. Across all loads, these changes underperform the baseline in request throughput, token throughput, and time-to-first-token. Our research reveals that while PrefillOnly-style gains are transferable in theory, they conflict with the coordination, memory behaviour, and compilation patterns of vLLM's disaggregated architecture. Our key takeaway is that prefill specialization, while compelling in theory, is difficult to transfer in practice. We highlight where the combination of these two approaches breaks down and provide insight for potential avenues for improvement.

1. Introduction

As large language models (LLMs) have grown in size and in prevalence, there has been increased pressure on inference infrastructure, particularly with a focus on latency and throughput. LLM inference consists of two computationally distinct phases: *prefill* and *decode*. In the prefill phase, the input prompt is processed to generate the first token, and in the decode phase, the remaining output tokens are generated. Traditionally, both phases are completed on the same GPU, which leads to underutilization of resources. For example, long prefill jobs will delay decoding, inflating latency for decode requests.

Disaggregating these two phases across GPUs has been a popular and promising architectural shift that improves throughput and reduces idle GPU time. It also allows each phase to be optimized independently. However, most existing implementations treat the prefill step as a generic inference worker. This overlooks unique opportunities for extracting efficiency gains: the prefill stage has deterministic runtimes, high memory usage, and is often the primary bottleneck for longer prompts.

In this project, we specialize the prefill stage in a disaggregated LLM serving pipeline. Specifically, we implement two optimizations, which have been introduced as a way to improve the prefill stage [9], into an existing disaggregated system:

1. Hybrid prefilling, which processes non-attention layers in memory-efficient chunks to support longer input contexts.
2. JCT-aware scheduling, which reorders requests by expected completion time, reducing queuing delays and improving GPU utilization.

Our implementation integrates these design features into vLLM's existing disaggregated path [11] [1]. In this project, we aim to understand whether PrefillOnly heuristics transfer to the disaggregated LLM setting and how they interact with the disaggregated environment.

The result of this project is that, while the theoretical advantages of hybrid prefilling and JCT-aware scheduling are compelling, our implementation and experiments indicate that these optimizations are not cleanly transferable to a disaggregated vLLM pipeline. The prefill-optimized path underperformed the baseline across request throughput and time-to-first-token, even at low QPS. These results suggest that disaggregation introduces new interactions with memory allocation and scheduling behaviour that negate the expected benefits. While prefill specialization is compelling and promising in theory, it is far from plug-and-play in

[1]David R. Cheriton School of Computer Science, University of Waterloo, Canada. Correspondence to: Sejal Agarwal <s97agarw@uwaterloo.ca>, Maksym Bidnyi <mbidnyi@uwaterloo.ca>, Joshua Caiata <jcaiata@uwaterloo.ca>, Gavin Deane <gdeane@uwaterloo.ca>.

[1]Our vLLM GitHub repository fork is available at: https://github.com/GDeane/vllm/tree/prefillOnlyOptimizations

practice. Our work highlights this insight and provides a roadmap for future improvements.

2. Related Work

In order to serve LLMs at scale, optimizing both throughput and latency across prefill and decode is imperative. Recent work has made progress in this area by proposing architectural and scheduling strategies to improve system performance. In light of this, disaggregated execution and trying to deal with prefill-specific bottlenecks have been a spotlight of the field, and are the main point of concern for this work.

DistServe [14] introduced the architectural design of disaggregation by decoupling prefill and decode on separate GPUs. While effective, DistServe treated the prefill node as a generic inference worker and does not leverage unique prefill workload characteristics to extract optimizations and performance improvements. Further, PrefillOnly [9] extends this line of thinking by identifying two opportunities to improve the prefill stage: hybrid prefilling and JCT-aware scheduling. Hybrid prefilling chunks the execution of non-attention layers, which allows longer input contexts without using up all of the GPU memory. Additionally, JCT-aware scheduling uses the deterministic nature of the prefill step to reduce latency and improve throughput. PrefillOnly focused on single-token output tasks, but the techniques introduced are, in theory, generalizable to different architectures and designs and serve as the backbone to this project. Our work builds on top of this by integrating PrefillOnly optimizations into the disaggregated execution model, which PrefillOnly does not do and suggests as future work.

Furthermore, Sarathi-Serve [7] introduced chunked prefilling and stall-free token-level scheduling to improve throughput-latency tradeoffs in non-disaggregated serving architectures. Our approach uses a similar chunked execution strategy; however, we extend it to the disaggregated architecture and complement it with JCT-based prioritization. Semi-PD is another related work [10], which introduced disaggregation at the compute level while maintaining unified storage to eliminate key-value (KV)-transfer overhead. Our implementation, on the other hand, operates within vLLM's existing PD pipeline and accepts KV cache transfer as a design constraint.

Our project unifies different perspectives on serving LLMs by specializing the prefill phase within a PD-disaggregated framework to extend maximum prompt length and increase throughput.

3. Implementation

Our implementation is based on the recently introduced disaggregated prefill framework in vLLM, which separates the prefill and decode phases of LLM inference between different worker nodes to improve utilization and throughput [12]. We integrate two key optimizations: hybrid prefilling and JCT-aware scheduling, which are inspired by the *PrefillOnly* design paradigm proposed in recent work [9]. *PrefillOnly* demonstrated that we could drastically reduce memory overhead and increase request concurrency for longer prompts by applying these optimizations to the prefill step.

In our design, we extend vLLM's existing disaggregated infrastructure to support hybrid prefill execution, where prefill requests can be dynamically distributed between dedicated prefill workers depending on workload characteristics and the state of the KV cache. This hybridization improves system flexibility, allowing for more efficient utilization under mixed request sizes. Additionally, we implement JCT-aware scheduling to prioritize request execution based on expected job completion time, thereby improving both average latency and fairness under high concurrency. Together, these components form a cohesive scheduling and execution pipeline that optimizes the prefill phase for large-context workloads while maintaining compatibility with vLLM's modular architecture.

3.1. Disaggregated Prefilling

The disaggregated prefilling feature in vLLM enables the prefill and decode phases of LLM inference to run on separate vLLM instances, allowing independent tuning and parallelization of these stages [12]. This architectural separation provides the flexibility to optimize time-to-first-token (TTFT) and inter-token latency (ITL) individually, which are two key performance metrics in serving large models interactively.

Without disaggregation, vLLM executed prefill and decode tasks within the same process, which can lead to resource contention. Prefill-heavy requests, especially those with long prompts, may introduce idle GPU cycles and increase tail latency when interleaved with decode-heavy requests. By running two coordinated instances, a prefill instance and a decode instance, disaggregated prefilling ensures that each stage can use its own optimized tensor and pipeline parallelism strategies. This design also enables better control over tail ITL, which is otherwise difficult to tune through chunked prefill alone.

3.1.1. Architecture and Workflow

At a high level, the system operates as follows (illustrated in Figure 1):

1. The Proxy API server receives an incoming request and sets `max_tokens=1` before sending it to the vLLM prefill instance.

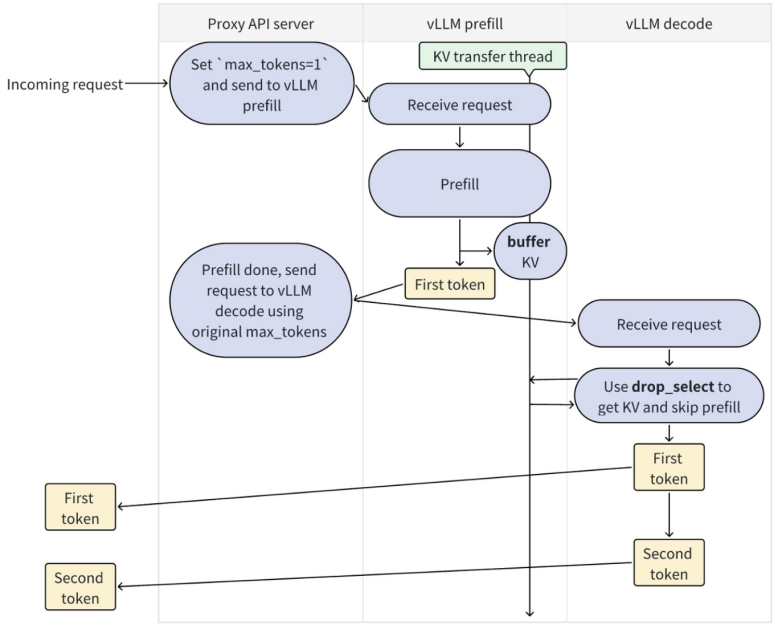

Figure 1. Workflow of disaggregated prefilling in vLLM. The prefill and decode instances operate independently, exchanging KV caches through the connector interface [12].

2. The prefill worker processes the prompt tokens, computes the attention KV caches, and stores them temporarily using a connector interface.

3. Once the token is generated, the request is forwarded to the decode instance, which retrieves the prefilled KV tensors and continues token generation without repeating the prefill step.

4. The KV transfer thread facilitates communication between the two instances, ensuring that the cached tensors are correctly synchronized and released once decoding completes.

This modular design allows independent scaling of the prefill decode clusters; thus, we can apply our prefill optimizations without modifying the decode node.

3.1.2. Connectors and Shared Storage

Communication between the prefill and decode stages is abstracted through the Connector interface, which defines how KV tensors are transmitted. vLLM currently supports several connector types.

Our implementation leverages the SharedStorageConnector, which enables the transfer of KV caches via a shared memory [13]. This connector provides high reliability and ease of deployment across nodes, as it eliminates the need for direct GPU-to-GPU network transfer. The key abstractions used internally include:

- Connector: Defines producer–consumer relationships for KV cache transfer.

- LookupBuffer: Provides insert and `drop_select` APIs for inserting or retrieving KV tensors, similar to SQL semantics.

- Pipe: Implements FIFO tensor transmission, supporting `send_tensor` and `recv_tensor` operations for efficient data streaming.

3.1.3. Benefits

Disaggregated prefilling does not directly improve throughput but instead provides architectural flexibility and latency control. It allows for

- Independent optimization of TTFT and ITL through tailored parallelism.

- Reduced tail latency under mixed workloads.

- Better system utilization for heterogeneous request distributions (short vs. long prompts).

This separation also sets the foundation for our hybrid prefilling and JCT-aware scheduling extensions, which further enhance throughput and fairness across concurrent users.

3.2. Hybrid Prefilling

To further optimize memory efficiency in the prefill stage, we integrate hybrid prefilling, the key optimization introduced in PrefillOnly [9]. Hybrid prefilling addresses the memory bottleneck caused by the large intermediate tensors generated by linear (non-attention) layers during prefill. In conventional prefilling, the full sequence of tokens is processed at once, which leads to memory spikes as all intermediate activations of multilayer perceptrons (MLPs) are held in memory simultaneously. One may note that chunking the entire prefill accomplishes the same goal (as is the default in vLLM [2]). However, this forces the KV-cache of all attention layers to be stored for subsequent chunk processing [9]. If a prefill request has only one output token, there is little need to keep all of the KV cache in the prefill node's memory. Hybrid prefill allows us to do precisely that by processing the entire attention layer at once, then directly offloading cache values to save on precious memory.

These activations are significantly larger than the per-layer KV cache, resulting in limited memory isolation and reduced model instance-level parallelism (MIL).

Hybrid prefilling mitigates this issue by executing non-attention layers chunk-by-chunk while continuing to prefill attention layers normally. Because linear operations are independent across token chunks, this approach preserves correctness while substantially lowering peak memory usage. We implement this mechanism using `torch.compile`, grouping consecutive linear operations into virtual layers and streaming their computation incrementally. Additional optimizations, such as output preallocation and in-place computation, further reduce memory duplication. By incorporating hybrid prefilling into vLLM's disaggregated prefill pipeline, we enable higher concurrency for large prompts without exceeding GPU memory limits, effectively improving throughput and stability under memory-bound workloads.

3.3. JCT-aware Scheduling

To complement hybrid prefilling, we implement JCT-aware scheduling to improve fairness and efficiency across concurrent inference requests [9]. Traditional first-in-first-out (FIFO) or round-robin schedulers in LLM serving systems can lead to head-of-line blocking, where long prompts delay shorter interactive queries. JCT-aware scheduling mitigates this by estimating each request's expected completion time based on prompt length and prefill/decode cost models, then prioritizing requests to minimize overall mean JCT. Normally, the nondeterministic-length decode step in LLM serving prevents reliable JCT estimates, but PrefillOnly showed that for requests with a single output token, JCT could be estimated by subtracting the number of prefix cache hits in the prompt from the number of tokens in the prompt. In our implementation, the scheduler operates at the request queue level of the disaggregated prefill pipeline, dynamically reordering incoming requests before assignment to prefill or decode workers. This policy ensures that shorter requests receive timely service while maintaining high GPU utilization for longer prefill jobs, achieving a balanced trade-off between throughput and latency.

4. Results

To evaluate our optimizations, we adapted existing vLLM benchmark scripts used for evaluating prefill-decode disaggregation [4]. The existing script [5] compared the disaggregated prefill-decode configuration to a standard chunked prefill configuration. We modified this script to instead compare a prefill-decode configuration with our optimizations to one without.

Each experiment uses two L40 GPUs [1] on a single host. The driver builds a long synthetic dataset (`sonnet_8x.txt`) so that requests are long enough to put pressure on the prefill step. We benchmark four loads (queries-per-second (QPS) $\in \{2, 4, 6, 8\}$) with 100 prompts per load. Unless otherwise noted, the following configuration is used:

- **Model**: Meta-Llama-3.1-8B-Instruct.
- **Max context**: 10000.
- **Prompt/output**: input_len = 8192, output_len = 6, shared prefix_len = 50.
- **Disaggregation**: SharedStorageConnector, kv_parallel_size=2, kv_buffer_size $\approx$ 5 GB.
- **Resource limit**: –gpu-memory-utilization 0.6.
- **Traffic**: vllm bench serve with –request-rate set to the target QPS and –num-prompts=100.

Although our primary evaluation focuses on long-context prompts (input_len = 8192) to explicitly stress the prefill phase, we also ran an additional set of benchmarks with shorter prompts (input_len = 1024, output_len = 6, prefix_len = 50) under the same PD configuration and QPS settings. These short-prompt runs were much less prefill-bound and did not significantly stress GPU memory, so the relative differences between the baseline and prefill-mode configurations were not notable in terms of throughput and TTFT. For completeness, we report these results and the corresponding hardware configuration in Section A, but we base our main analysis on the 8k-token setting described above.

4.1. Compared Variants

Optimized (using vllm prefill). The first GPU runs `vllm prefill` (prefill mode enabled); the second GPU runs `vllm serve` as a standard decode node. This utilizes the specialized prefill-only path that avoids chunked prefill.

Baseline (using vllm serve). Both sides run `vllm serve`. The prefill side behaves like a generic server (i.e., the chunked-prefill style path); KV is still disaggregated via shared storage. All other config is identical.

The only conceptual difference is on the prefill node, where our optimizations use prefill mode (hybrid prefilling and JCT-aware scheduling) and the baseline uses chunked prefilling.

4.2. Metrics and Reporting

We report three primary metrics that the benchmark harness emits per run and aggregate across 100 prompts at each QPS. These metrics are used because they are most strongly affected (or bottlenecked) by the prefill step we are optimizing:

- **Request throughput** (req/s): completed requests per second.
- **Token throughput** (tok/s): total output tokens per second.
- **Time-To-First-Token (TTFT)** (ms): latency from request admission at the proxy to the first decoded token.

4.3. Summary of Observations

Our results (see Figure 2) show a trend where, at low QPS, the chunked-prefill baseline for prefill-decode disaggregation in vLLM severely outperforms our prefill-only optimizations for prefill-decode. That is, the prefill-only optimizations range from having 27× the baseline's TTFT at 2 QPS, to 1.8× the baseline's TTFT at 8 QPS. In addition, the prefill-only optimizations range from 0.34× to 0.75× the baseline's throughput. At higher QPS values, the two become more in line, though the chunked prefill is still performing better.

We reflect that in the baseline, the prefill node runs the standard serve and chunked-prefill path in vLLM. This path uses mature, highly optimized kernels and graphs based on requests being chunked into a standard maximum chunked length. In contrast, the prefill pass in the prefill-only optimization run is not chunked by design, so it may have less optimized graph captures associated with larger request shapes.

These trends persist across all loads despite both variants suffering from reduced decode-side efficiency due to a SharedStorageConnector bug (see Section 5.3). We also hypothesize that the prefill-mode path is further penalized by additional compilation and graph-shape constraints (see Section 5.2).

Some other reasons for prefill-only optimization performance degradations we hypothesize as follows:

- **Prefill mode network burstiness**: Hybrid prefilling causes a single large prefill pass compared to the chunked baseline. This means KV values get written to the connector in a single large burst, increasing burstiness and potentially bottlenecking the network.
- **JCT-aware scheduling overhead**: The implementation of JCT-aware scheduling involves continuously checking requests in the queue to see which ones are the shortest based on the current cache state. For some workloads, this overhead may be worth it, but for our workload with the common shared prefix, that overhead may be redundant compared to first-come, first-serve scheduling.
- **Memory Pressure**: Prefill mode has significantly larger memory for prefill in contrast to chunked prefilling. This can stress GPU memory pressure and throttle the system, particularly when larger models with long prompts are loaded.

Even though these optimizations did not work, our work reveals an important insight about the underlying architectures of these two systems: these approaches are not compatible in practice as they stand right now.

5. Discussion

Here, we discuss challenges associated with working in vLLM, implementing our optimizations, and setting up our evaluation.

5.1. Challenges in storing KV caches to CPU during forward pass

In PrefillOnly [9], it is mentioned that one of the main reasons to implement hybrid prefilling (chunking only inputs to non-attention layers during the prefill) is so that the KV cache for intermediate layers can be offloaded rather than being kept for subsequent chunks.

This was part of our original plan to reduce memory usage during the prefill step. However, when it came time to implement this, we discovered the following about memory allocation in vLLM.

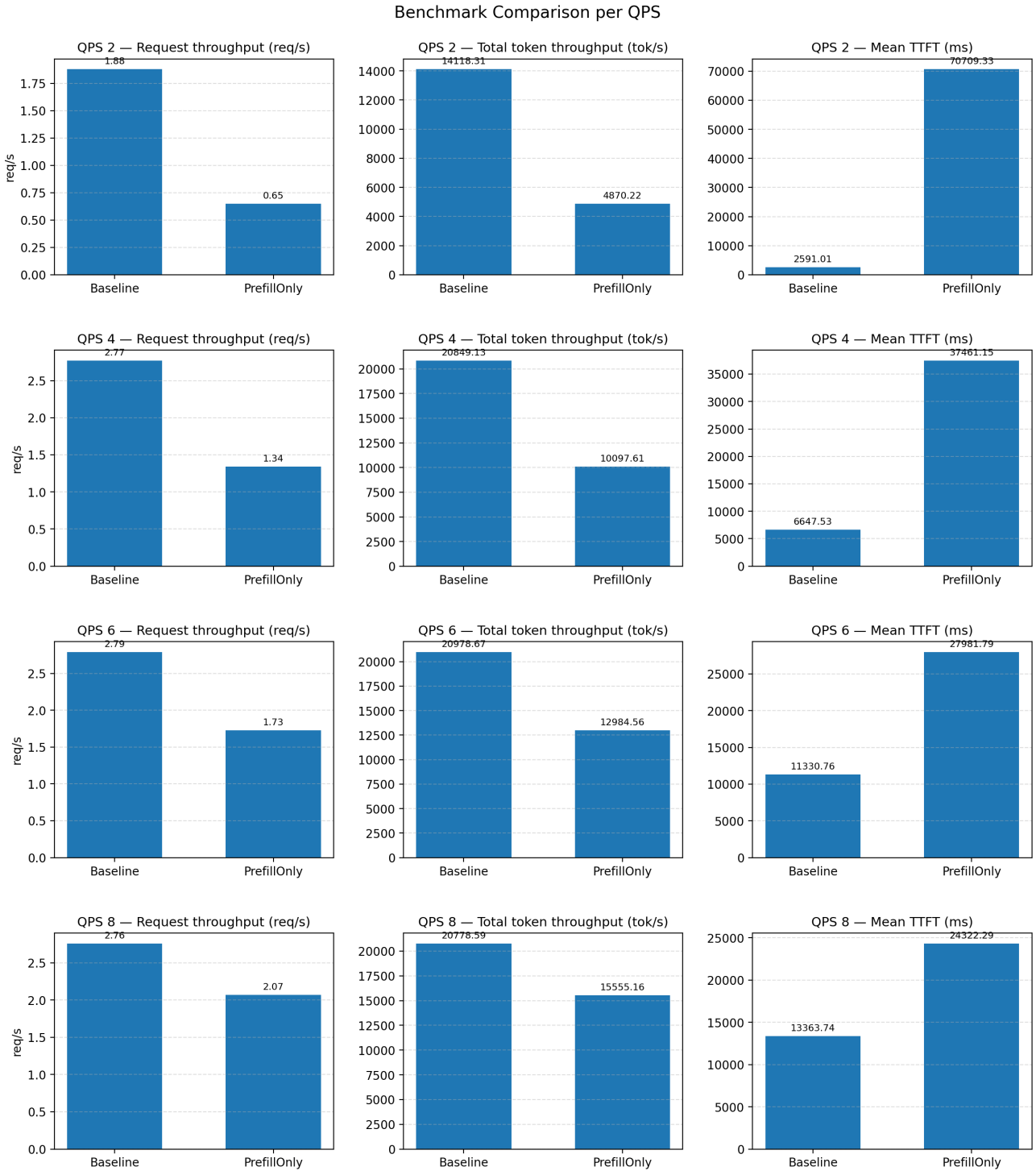

Figure 2. Throughput and TTFT at QPS {2,4,6,8}. Each row is a QPS; columns show request throughput (left), token throughput (middle), and TTFT (right), comparing the chunked-prefill baseline to the optimized prefill-mode path (hybrid prefill).

In prefill-decode disaggregation, the connectors already stream per-layer KV tensors to disk, but the Paged KV cache stays allocated because the attention stack relies on the global block pool for the entire forward pass. To "spill immediately" between attention layers, we would need to change how blocks are allocated and reused between attention layers. At the moment, the scheduler allocates once per request before the forward begins, and the attention modules expect those blocks to stay valid until the request finishes. That is, paged memory is allocated and freed at the request level, not the layer level. Reclaiming them right after an attention layer would break the subsequent layers because they would still be reading from freed pages.

Because of this, implementing KV-spilling after attention layers would require modifying how PagedAttention works in vLLM to break the assumption that allocated pages for a request stay allocated. Therefore, we elected not to implement this optimization.

We note that this does not necessarily make hybrid prefilling useless, since the intermediate tensors generated by non-attention layers make up a significant part of memory spikes during LLM inference [9]. Chunking these intermediate tensors should still reduce peak memory in theory. However, the challenges faced in this project indicate that these approaches are incompatible out of the box and may need structural changes to be able to actually see an improvement.

5.2. torchInductor negative Buffer error

An error that stymied our evaluation came from the way vLLM uses TorchInductor and TorchDynamo [3; 6; 8]. In short, TorchDynamo is a PyTorch just-in-time compiler that intercepts Python's frame execution, rewrites the computation graph, and sends that computation graph to an optimized backend [8].

During initialization in vLLM, torchInductor captures a CUDA graph for a representative batch shape. This is useful for having the CUDA kernels ready on demand later. TorchInductor infers the shapes of tensors in this graph by subtracting a fixed chunk size (used for decoding) from the total token count. However, if the representative batch size is smaller than the fixed chunk size, the buffers for these tensors have a negative size.

In our implementation, hybrid-prefill mode uses tiny decode slices, as max_tokens is set to 1 for prefill-only requests. As a result of this, the warmup batch ends up much smaller than what the CUDA-graph template expects, so when the CUDA-graph template subtracts the decode chunk size, it goes negative.

In order to work around this issue and run our evaluation, we modified the CUDA-compile warm-up run for vLLM to run with both a short token count and the default token count, thus producing CUDA graphs for both shapes. This has the tradeoff of introducing a second compilation pass, increasing startup, but it allowed us to run our evaluation without crashes. We suspect changing the warm-up run for our specific evaluation reduced the optimizations of the CUDA graphs for the prefill-only runs, likely contributing to the reduced performance compared to the chunked-prefill baseline.

5.3. SharedStorageConnector bug

In our implementation, we made use of the SharedStorageConnector [12] to transfer the KV cache from the prefill to the decode node. This choice was made due to the relative simplicity and lack of special setup requirements. In addition, we justify that since both the baseline and the optimized setup use the SharedStorageConnector, it should not adversely affect our evaluation.

During evaluation, we discovered a bug where the decode node, attempting to load from the ShareStorageConnector, could not load the KV cache because the passed attention is none. It turns out this is a known bug with the SharedStorageConnector in vLLM [**?**].

In our evaluation, this bug occurred both when running the baseline and when running with prefill mode optimizations. This means that in both cases, the decode node was not able to read from the shared KV cache sent by the prefill node, and presumably had to recompute the KV-cache values.

The implication is that the throughput of both the baseline and the prefill mode optimizations was greatly reduced due to the lack of KV cache reuse in the decode node. However, this alone does not explain the degradation of throughput found with prefill mode when compared to the baseline, as this issue would affect both decode instances equally.

6. Future Work

Future work on this system can proceed in several directions. A primary challenge encountered in our implementation was the interaction between hybrid prefilling and TorchInductor's CUDA graph capture. As discussed earlier, TorchInductor infers tensor shapes using fixed decode chunk sizes, which can result in negative buffer allocations when `max_tokens` is restricted to one. Addressing this issue is necessary to support reliable warmup for hybrid-prefill execution. Potential solutions include adjusting representative batch sizes during initialization, adding shape guards to prevent invalid allocations, or selectively disabling CUDA graph capture for prefill-only workloads.

A second direction involves conducting a more comprehensive evaluation under realistic serving conditions. While our prototype integrates hybrid prefilling and JCT-aware

scheduling into the disaggregated vLLM pipeline, full-system behaviour under high concurrency, multi-worker scaling, and long-context prompts remain unexplored. Such experiments would provide deeper insight into throughput, fairness, and tail latency improvements enabled by these optimizations.

Additionally, extending the design to support alternative architectural choices, such as Semi-PD's unified KV storage model, may reduce KV transfer overhead and improve compatibility with large-scale deployments. Finally, refining the JCT estimator and exploring adaptive chunk sizing for hybrid prefilling may yield more efficient scheduling policies and improved memory utilization. Together, these directions outline a path toward a more robust and scalable prefill-optimized serving system.

7. Conclusion

In this work, we integrated hybrid prefilling and JCT-aware scheduling into vLLM's disaggregated prefill pipeline to evaluate whether PrefillOnly-style optimizations could be applied in a phase-disaggregated architecture. While the techniques fit conceptually within vLLM's modular design, our empirical evaluation and debugging experience show that their practical integration is substantially more complex than expected. Across all measured loads, the prefill-optimized path underperformed the chunked-prefill baseline in both throughput and time-to-first-token, even after controlling for known issues in SharedStorageConnector and TorchInductor. These findings suggest that PrefillOnly heuristics interact poorly with the coordination, memory behaviour, and compilation assumptions of disaggregated execution. Rather than validating the transferability of these optimizations, our results highlight fundamental compatibility gaps, particularly around graph capture, KV-cache handling, and burstiness during prefill, which limits their effectiveness in practice. This project, therefore, serves primarily to surface where the theoretical appeal of prefill specialization conflicts with the realities of the current LLM serving stacks, and clarifies what architectural changes would be required for these approaches to become viable in disaggregated systems.

8. Compute Acknowledgement

We would like to acknowledge that this research was enabled in part by GPUs provided by the Vector Institute (https://www.alliancecan.ca/en/services/compute/killarney) and the Digital Research Alliance of Canada (alliancecan.ca).

References

[1] [n. d.]. NVIDIA L40 GPU for Data Center. https://www.nvidia.com/en-us/data-center/l40/

[2] [n. d.]. Optimization and Tuning - vLLM. https://docs.vllm.ai/en/stable/configuration/optimization/#chunked-prefill

[3] [n. d.]. torch.compiler — PyTorch 2.9 documentation. https://docs.pytorch.org/docs/stable/torch.compiler.html

[4] [n. d.]. vllm bench serve - vLLM. https://docs.vllm.ai/en/latest/cli/bench/serve/

[5] [n. d.]. vllm/benchmarks at main · vllm-project/vllm. https://github.com/vllm-project/vllm/tree/main/benchmarks

[6] 2022. TorchInductor: a PyTorch-native Compiler with Define-by-Run IR and Symbolic Shapes - compiler. https://dev-discuss.pytorch.org/t/torchinductor-a-pytorch-native-compiler-with-define-by-run-ir-and-symbolic-shapes/747 Section: compiler.

[7] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. arXiv:2403.02310 [cs.LG] https://arxiv.org/abs/2403.02310

[8] Aviralgoel. 2025. What is TorchDynamo and TorchInductor? (for Dummies). https://medium.com/@goelaviral/what-is-torchdynamo-and-torchinductor-for-dummies-c609b851aaff

[9] Kuntai Du, Bowen Wang, Chen Zhang, Yiming Cheng, Qing Lan, Hejian Sang, Yihua Cheng, Jiayi Yao, Xiaoxuan Liu, Yifan Qiao, Ion Stoica, and Junchen Jiang. 2025. PrefillOnly: An Inference Engine for Prefill-only Workloads in Large Language Model Applications. arXiv:2505.07203 [cs.DC] https://arxiv.org/abs/2505.07203

[10] Ke Hong, Lufang Chen, Zhong Wang, Xiuhong Li, Qiuli Mao, Jianping Ma, Chao Xiong, Guanyu Wu, Buhe Han, Guohao Dai, Yun Liang, and Yu Wang. 2025. semi-PD: Towards Efficient LLM Serving via Phase-Wise Disaggregated Computation and Unified Storage. arXiv:2504.19867 [cs.CL] https://arxiv.org/abs/2504.19867

[11] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. arXiv:2309.06180 [cs.LG] https://arxiv.org/abs/2309.06180

[12] vLLM Contributors. 2024. Disaggregated Prefill in vLLM. https://docs.vllm.ai/en/stable/features/disagg_prefill/. Accessed: 2025-11-27.

[13] vLLM Contributors. 2025. vLLM SharedStorageConnector — KV transfer connector for disaggregated prefill. https://docs.vllm.ai/en/latest/api/vllm/distributed/kv_transfer/kv_connector/v1/shared_storage_connector/. Accessed: 2025-11-27.

[14] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. arXiv:2401.09670 [cs.DC] https://arxiv.org/abs/2401.09670

A. Additional Benchmark with Shorter Prompts

In addition to the long-context experiments described in Section 4, we ran an auxiliary benchmark using shorter prompts to evaluate the impact of prefill-only optimizations when the prefill phase is not the dominant bottleneck.

A.1. Configuration

This experiment uses the same disaggregated vLLM setup as in the main results:

- **Model**: Meta-Llama-3.1-8B-Instruct.
- **Hardware**: Two NVIDIA L40 GPUs on a single host.
- **Disaggregation**: SharedStorageConnector with kv_parallel_size = 2 and kv_buffer_size $\approx$ 5 GB.
- **Resource limit**: –gpu-memory-utilization 0.6.
- **Traffic**: vllm bench serve with QPS $\in \{2, 4, 6, 8\}$ and 100 prompts per load.

The key difference from the main experiment is the prompt length:

- **Prompt/output**: input_len = 1024, output_len = 6, shared prefix_len = 50.

All other prefill/decode configuration, including the use of `vllm prefill` on the prefill node and `vllm serve` on the decode node, matches Section 4.

A.2. Results

As shown in Figure 3, the baseline and prefill-mode configurations achieve very similar request and token throughput and less notable TTFT differences at all investigated QPS levels. Because the shorter prompts place substantially less load on the prefill phase and GPU memory, these runs do not reveal the same performance gaps observed in the 8k-token setting. Given that the prefill-only optimizations are focused on prefill-dominated workloads, we do not place emphasis on results from these short-prefill runs.

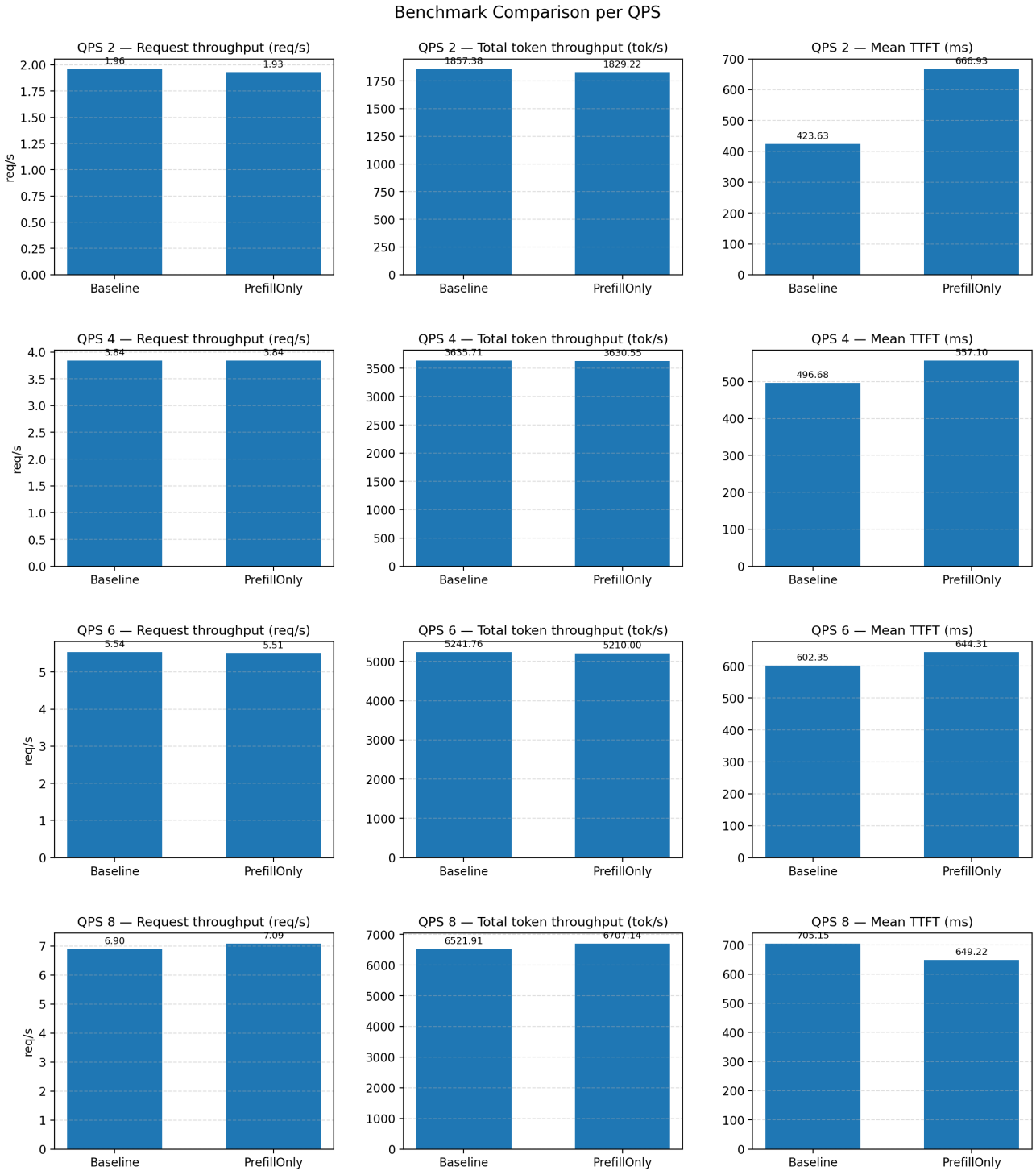

Figure 3. Comparison of baseline and prefill-mode configurations for input_len = 1024 across QPS $\{2, 4, 6, 8\}$. Each row corresponds to a QPS and reports request throughput (left), total token throughput (middle), and mean TTFT (right).