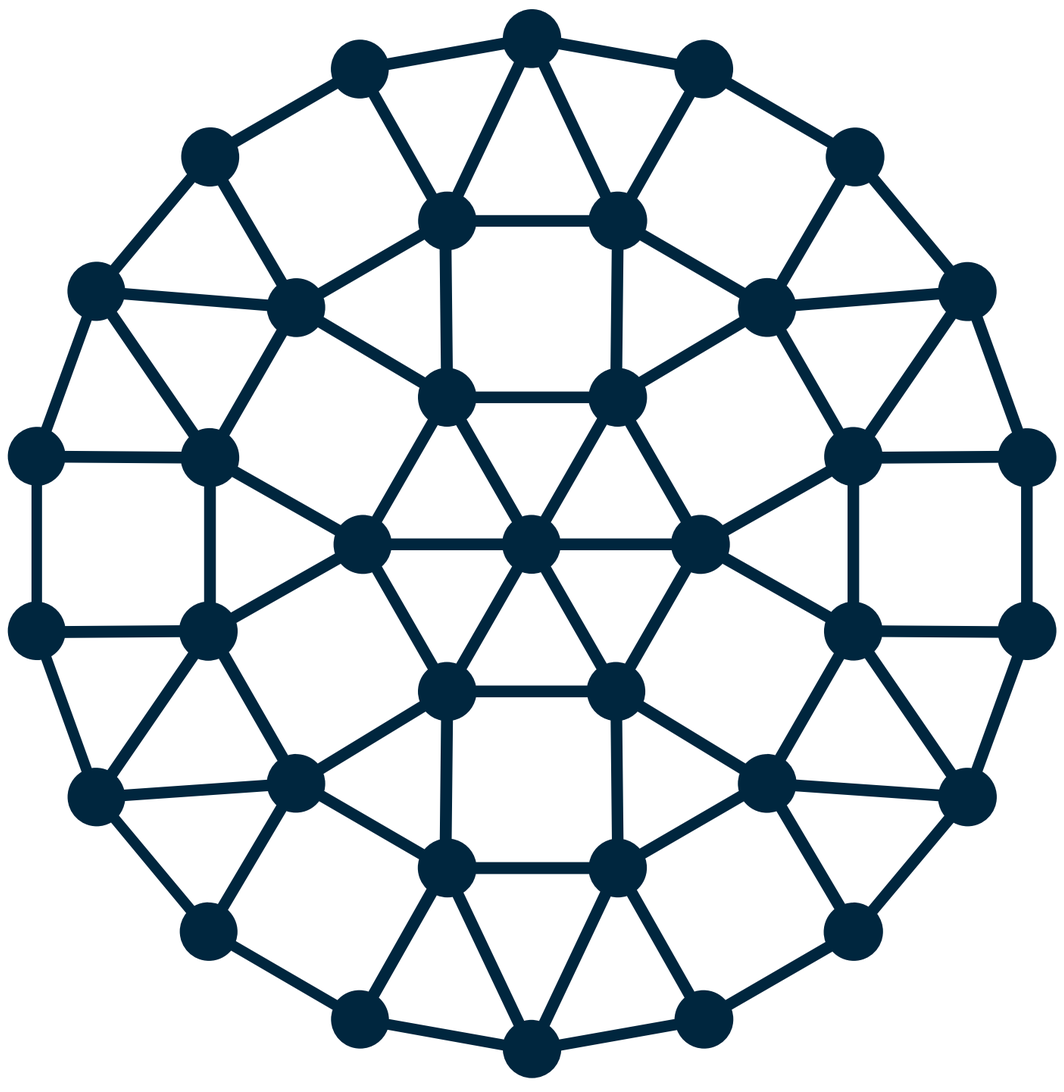
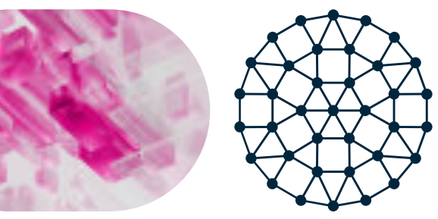
run:ai

A Benchmarking Study:

Which serving technology to choose for LLMs?



January 2024 | Omer Dayan · Ekin Karabulut · Noa Neria



Abstract

This benchmarking study assesses the performance of large language models (LLMs) through an exploration of model-serving frameworks, focusing on the critical aspect of throughput. Evaluating engines such as TensorRT-LLM, vLLM, and inference servers like RayLLM with RayServe, TGI, and TensorRT-LLM with Triton, the study aims to provide valuable insights for machine learning practitioners in optimizing their projects. Our experiments uncover the importance of strategic preemption mechanism in the management of the KV cache memory. The importance of this strategy stems from the generation process being memory bound, sequence length considerations specific to engines and the impact of the model size on the throughput. Besides comparing different servers and engines, our experiments were designed to compare the performance of two strategies for management of the KV cache: preemption versus strict reservation. Our main finding is that for a limited amount of GPU memory and varied lengths of output size, the best strategy was proved to be preemption. Other notable findings include the delicate trade-offs in memory allocation, vLLM's remarkable concurrency handling, and the influence of server selection on overall throughput, as TensorRT-LLM with Triton outperforms TensorRT-LLM alone in high Query per Second (QPS) rates. The study offers publicly accessible benchmarking scripts and a detailed analysis, making it a valuable asset for the community looking to enhance LLM deployment in production environments. We encourage everyone to contribute with new tools and utilize our publicly available scripts for their own performance experimentation.

Table of contents

Introduction

The rapidly growing interest in leveraging large language models (LLMs) for a wide array of applications has led to extensive exploration within both the industry and the open-source community. As the demand for these models in production environments continues to rise, the need to assess their performance, alongside the available tools and frameworks, becomes increasingly significant.

This benchmarking study takes a deep dive into model-serving frameworks with a specific focus on the critical performance aspect of throughput. By examining various precisions, input sizes, and output sizes, and evaluating inference engines such as TensorRT-LLM, vLLM and inference servers such as RayLLM with RayServe, TGI and TensorRT-LLM + Triton, we aim to provide ML practitioners with some insights to make informed decisions for their own projects.

To complement this benchmarking study, we have published an accompanying blog post that explains the generation process of LLMs, covering state-of-the-art techniques employed in the tools assessed in this study. This resource serves as a detailed reference for readers seeking a deeper understanding of the methodologies employed. We encourage readers who wish to dive deeper into the details of these techniques to explore the blog post first.

To further advance the transparency and collaboration, we have publicly released the benchmarking scripts used in our experiments. This repository allows others not only to verify and replicate our results but also to build upon this research by introducing new frameworks or techniques into the evaluation.

Components of LLM serving

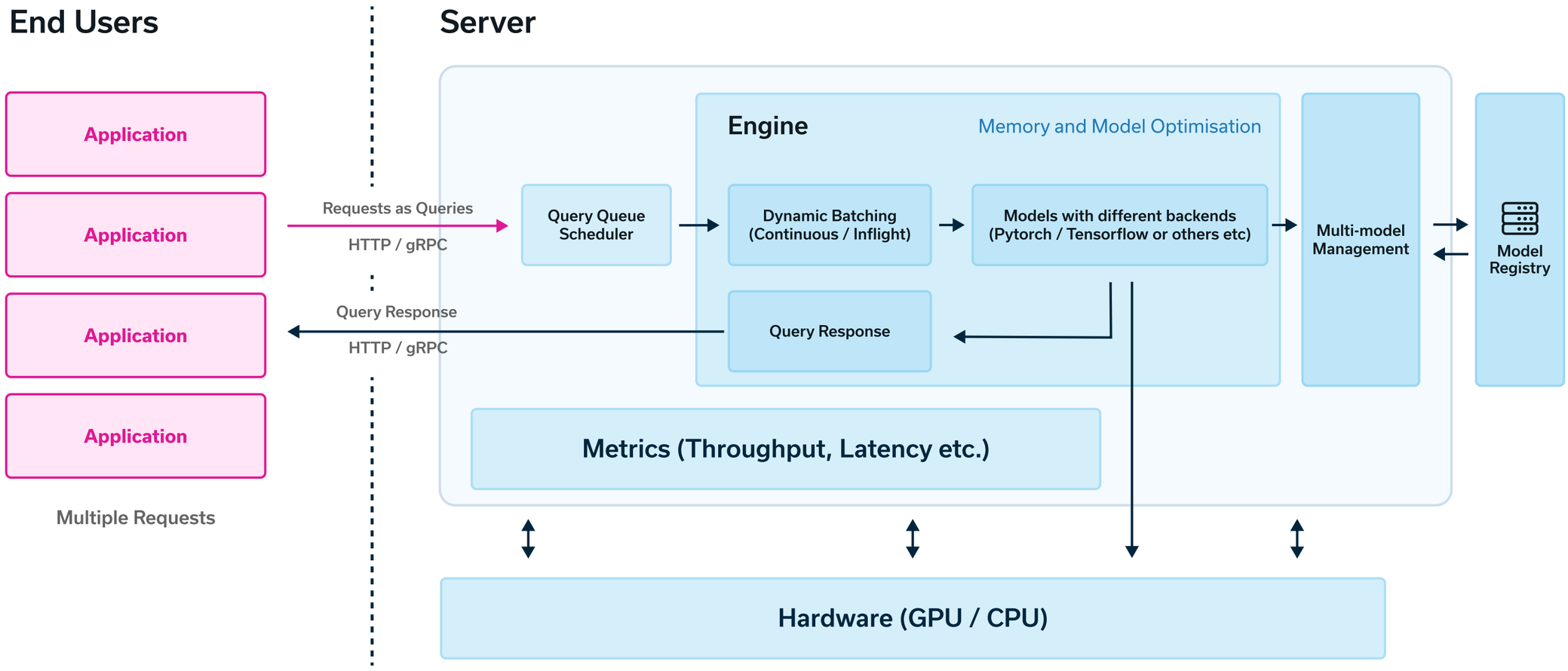


Figure 1: *Architecture of servers and engines*

When it comes to serving LLM based applications, model serving can be divided into two components: engine and server. At high level, the engine handles everything about the models and batching the requests, while the server handles forwarding the user requests.

Engines

Engines are what run the models and everything that we covered so far about the generation process with different types of optimization techniques. In their core, these are Python or C++ libraries. They handle batching of the requests that are coming from users to our chatbot and generating the response for those requests.

Servers

Servers are responsible for orchestrating the HTTP/gRPC requests coming in from the users. In real world applications, we will have many users asking questions to our chatbot at different times of the day. Servers queue these requests and forward them to the engine for the generation of the response. Servers also bring the metrics such as throughput and latency, which are important to track for model serving.

	Capabilities	Frameworks
Engines	• Memory management of KV-cache (preemption, reservation) • Memory optimization • Model specific optimization • Batching support	• TensorRT-LLM • vLLM • Text Generation Inference (TGI)
Servers	• HTTP/gRPC APIs • Request queuing • Multi-model serving • Multi-engine support	• NVIDIA Triton Inference Server • Text Generation Inference (TGI) • RayLLM with RayServe

Table 1: *An overview of engine & server capabilities together with common frameworks that offer these capabilities*

Note: In this whitepaper, we expect the readers to know the basics of the LLM generation process and the concepts such as continuous batching, KV-cache and PagedAttention. Please refer to our blogpost first, if you need an overview of these state-of-art concepts.

Memory Management of KV cache

Memory management of KV cache is a crucial aspect of LLM serving. It is handled by engines. There are 2 main policies; preemption and reservation.

Preemption Mechanism

Preemption is a central technique in the KV cache management, particularly when system capacity is overwhelmed with requests. The engines with this mechanism adopt a first-come-first-serve (FCFS) policy, prioritizing the processing of requests based on their arrival time, which mitigates the risk of any request being perpetually delayed (starvation). When the GPU cache becomes saturated with data, the system must decide which blocks to evict to make room for new ones. As an example, an all-or-nothing eviction policy can be utilized, which means that all blocks related to a particular sequence, or a related group of sequences, are evicted or maintained together to account for potential data dependencies [1]. To manage evicted blocks, techniques like swapping—copying blocks to CPU memory—or recomputation—regenerating KV cache data when needed—are deployed. The strategy ensures effective utilization of available GPU memory resources while considering the interdependencies between sequence groups and the need to recover evicted data efficiently.

Reservation

The reservation strategy differs from preemption in that it aims to avoid the necessity of evicting KV cache by allocating sufficient resources ahead of time. Through reservation, a portion of the GPU memory is allocated specifically for the KV cache needs of a particular sequence. This approach seeks to guarantee that essential data resides in the GPU memory as long as required, thus avoiding the latency associated with eviction and subsequent data recovery. By predicting the KV cache size needed for a given length of input and reserving that space in advance, the engine can ensure a smoother execution flow. However, the reservation method must be well-calibrated to optimize memory utilization and prevent over or under-provisioning, which could lead to inefficiencies or a need for eventual preemption respectively.

Frameworks

In this section, we delve into the details of the key frameworks used in our benchmarking study. We only choose a limited amount of frameworks for our experiments. Each framework has a unique value in optimizing and enhancing the performance of Large Language Models (LLMs) during inference. We'll explore the innovative features and capabilities of each framework, shedding light on how they contribute to the state of the art in model serving. To start off; we will present them in two categories: engines and servers.

Engines

NVIDIA TensorRT-LLM (TRT-LLM)

- An open-source library designed to accelerate and optimize inference performance on the latest LLMs using NVIDIA Tensor Core GPUs [2][3].
- Wraps TensorRT's Deep Learning Compiler, optimized kernels from FasterTransformer, pre- and post-processing, and multi-GPU/multi-node communication in a simple, open-source Python API for defining, optimizing and executing LLMs in production
- Utilizing tensor parallelism, TensorRT-LLM allows for efficient inference at scale across multiple GPUs and servers without the need for extensive developer intervention.
- Includes highly optimized, ready-to-run versions of popular LLMs, such as Meta Llama 2, OpenAI GPT-2 and GPT-3, Falcon, Mosaic MPT, and more.
- Provides a C++ runtime for executing LLM engines, offering features like PagedAttention, token sampling and KV cache management, further enhancing the efficiency of inference.
- Supports in-flight batching, also known as continuous batching or iteration-level batching. This is a technique that aims at reducing wait times in queues, eliminating the need for padding requests, and making higher GPU utilization possible.
- Aims to simplify the process of building and experimenting with new LLMs, providing peak performance and customization without requiring deep knowledge of C++ or NVIDIA CUDA.

vLLM

- A high-performance library tailored for LLM inference and serving, emphasizing state-of-the-art serving throughput and efficient management of attention [1][4].
- Memory efficiency and high throughput are at the core of vLLM, thanks to its innovative PagedAttention mechanism. This approach optimizes memory allocation and allows for non-contiguous KV cache, translating into higher batch sizes and cost-effective serving [5].
- Includes support for continuous batching, GPU parallelism, streaming output, and OpenAI compatibility.
- Provides a Python API for conducting offline batched inference on datasets, establishing API servers for LLMs, and launching OpenAI-compatible API servers.

Servers

RayLLM

- Built on Ray Serve, RayLLM benefits from a distributed compute framework that provides specialized libraries for data streaming, training, fine-tuning, hyperparameter tuning, and serving, simplifying the development and deployment of large-scale AI models [6].
- Supports deployment of multi model endpoint.
- It provides server capabilities while engine capabilities are provided by integrations such as continuous batching, paged attention, and other optimization techniques through TGI and vLLM integration.

Note: In this whitepaper, we benchmark RayLLM with vLLM engine

NVIDIA Triton Inference Server with TensorRT-LLM

- An open-source inference serving software that provides the ability to deploy models at scale in production environments. It supports various machine learning frameworks and is designed for high throughput and low latency inference workloads.
- Triton complements TensorRT-LLM by offering a solution for optimizing, deploying, and running Large Language Models (LLMs).Triton, as the server, interacts seamlessly with the TensorRT-LLM engine, utilizing techniques such as in-flight batching and paged KV-caching, while leveraging the advantages of TensorRT-LLM for rapid inference execution [7].

Text Generation Inference (TGI)

- A Rust, Python, gRPC server, used at HuggingFace to power HuggingChat, the Inference API and Inference Endpoint.
- Utilizes tensor parallelism (Accelerate) for faster inference on multiple GPUs
- Supports continuous batching for increased throughput, quantization, Paged and FlashAttention, token streaming using Server-Sent Events (SSE) and many more
- Logits warper (different parameters such as temperature,repetition penalty, top-k, top-n, etc.)
- Supports optimized set of specific LLMs

To provide an overview of the frameworks' capabilities, we present Table 2, which compares the engine and server features of the selected frameworks.

Engines	PagedAttention	C++ runtime	OpenAI compatibility	GPU parallelism	KV-cache management policy	Continuous batching
TensorRT-LLM	+	+	+	+ (pipeline & tensor parallelism)	reservation	+
vLLM	+	–	+	+ (tensor parallelism)	preemption	+
Servers						
RayLLM (with vLLM)	+	–	+	+	preemption	+
Triton (with TensorRT-LLM)	+	+	+	+ (pipeline & tensor parallelism)	reservation	+
Engine & Server						
TGI	+	–	–	+	reservation	+

Table 2: *Comparison of the engine and server capabilities*

Metrics for LLM Serving

When it comes to serving these models, there are two important metrics that needs to be underlined [9]:

Throughput

Throughput stands for the number of generated tokens per second by the inference server throughout the multiple requests by the users. The higher the throughput, the better.

Latency

Latency represents the time taken by the server and model to generate the whole output in the output sequence. If the generated output is streamed to the end user, then it represents the time taken by the inference server to generate the very first token (also called time to first token (TTFT)).

While latency holds significance as an inference metric, it falls outside the scope of this whitepaper's focus. Our emphasis remains on understanding and optimizing throughput for effective language model serving.

Experiment Setup

In this study, we emphasize the critical aspect of throughput, a key metric for assessing real-time performance in the deployment of large language models. To address it comprehensively, we conduct a series of experiments, each configured to explore a different facet of inference performance. Here's a concise overview of each experiment:

Experiment #1 **Throughput Dynamics with QPS and Batch Size**

Explored throughput dynamics of vLLM and TensorRT-LLM with varying Query Per Second (QPS) and batch size.

Experiment #2 **Preemption Mechanisms Analysis**

Analyzed preemption mechanisms in vLLM by varying batch sizes. Explored factors contributing to throughput peaks and degradation observed in Experiment #1.

Experiment #3 **Throughput with Various Query Rates and Variant Input**

exploration to multiple frameworks, examining throughput under different query rates and varied input. Comparing performance of vLLM, TensorRT-LLM, TGI, RayLLM, and TensorRT-LLM with Triton.

Experiment #4 **Throughput Analysis of vLLM and TGI from a Memory Perspective**

Investigated the effect of available memory for the KV-cache on throughput. Explored memory constraints' impact on vLLM, TensorRT-LLM, and TGI.

Experiment #5 **Real-Life Scenario: Impact of Model Size Increase & Variant Input/Output**

Replicated a real-life scenario by increasing model size to Llama-2-13b-chat and introducing variant input/output lengths. Evaluated throughput performance of vLLM, TensorRT-LLM, RayLLM, TGI, and TensorRT-LLM with Triton.

All experiments detailed in this paper are conducted on a single NVIDIA A100 GPU with 40 GB of memory. The tools under consideration include TensorRT-LLM (max utilization as scheduling policy), vLLM, RayLLM with vLLM, Triton Inference Server with TensorRT-LLM, and TGI. We conduct all experiments with Llama-2 models, specifically Llama-2-7b, Llama-2-7b-chat and Llama-2-13b-chat. For the experiments, where we examine the throughput with variant input/output sequence length, we use ShareGPT dataset.

Notably, for TensorRT-LLM, a crucial step involved the creation of an engine before serving the model. This engine requires input and output sequence length, as well as batch size parameters, and is tailored accordingly. In experiments where the sequence length is known, we employed the specific sequence length for engine creation. Conversely, for experiments with variant sequence lengths, we crafted the engine using the highest sequence length parameters offered by TensorRT-LLM, which is 2048.

Due to the fact that the details of the experiments vary depending on our findings in each experiment, we also provide a table with the setting in the next chapter. Please refer to the tables for a better overview on the setting of the specific experiment.

To ensure the robustness of our findings, each experiment is repeated ten times, and the results are reported as the average of these ten iterations to minimize variance. Furthermore, the scripts and input sequences used in these experiments are publicly available in our repository, facilitating transparency and reproducibility.

Experiment Results

Experiment #1 Throughput Dynamics with QPS and Batch Size

In the initial experiment, we investigated throughput dynamics on engines by varying Query Per Second (QPS) and batch size for vLLM and TensorRT-LLM. Our focus was on understanding how these engines handle continuous batching under a heavy load of user requests, using a fixed input/output sequence length.

Experiment Setting

Model	Llama-2-7b
Maximum batch size Max-num-seq (vLLM)	[64, 96, 128, 160, 192, 224, 256]
Query per second (QPS)	[32, 64, 96, 128]
Input token size	128
Output token size	128
Frameworks	TensorRT-LLM & vLLM

Table 3: *Experiment Setting to measure throughput with QPS and batch size. We measure token generation throughput with continuous batching as a function of query rate and number of parallel sequences.*

QPS Batch	32(QPS)	64(QPS)	96(QPS)	128(QPS)
64	2272	2295	2315	2390
96	2492	2521	2608	2655
128	2691	2727	2730	2732
160	2584	2615	2621	2637
192	2833	**2852**	2761	2782
224	2665	2666	2670	2680
256	2596	2636	2598	2615

Table 4: vLLM throughput results

QPS Batch	32(QPS)	64(QPS)	96(QPS)	128(QPS)
64	2485	2596	2612	2585
96	2866	2892	2861	2812
128	3246	3344	**3345**	3343
160	3146	3222	3206	3181
192	3223	3166	3185	3164
224	3223	3043	2971	2943
256	3230	3028	2949	2926

Table 5: TensorRT-LLM throughput results

A noteworthy finding emerges: Both engines exhibit a throughput peak for a specific batch size, with degradation observed beyond this peak. The underlying reasons for this behavior are explored in the next experiment.

Experiment #2 Preemption Mechanisms Analysis

Building on the insights from the previous experiment, we hypothesize that preemption mechanisms in vLLM contribute to the observed behavior. To investigate, we use vLLM with the configuration that yielded peak throughput in the prior experiment—varying batch sizes from 128 to 288 (also called max-num-seq in vLLM terms), and send 64 queries per second. Preemptions start occuring at a batch size of 192, increasing with larger batch sizes, a value coinciding with the throughput peak observed in the earlier experiment. Beyond this threshold, as we increased the batch size further, a decrease in throughput was noted. In the result of this experiment, we can see that the number of preemptions keep increasing when we increase the batch size further than 192.

Note: This experiment is exclusive to vLLM due to the limited open-source nature of TensorRT-LLM.

Experiment Setting

Model	Llama-2-7b
Max-num-seq (vLLM)	[128, 144, 160, 176, 192, 208, 224, 240, 256, 272, 288]
Query per second (QPS)	64
Input token size	128
Output token size	128
Frameworks	vLLM

Table 6: *Experiment setting to measure preemption rate as a function of batch size*

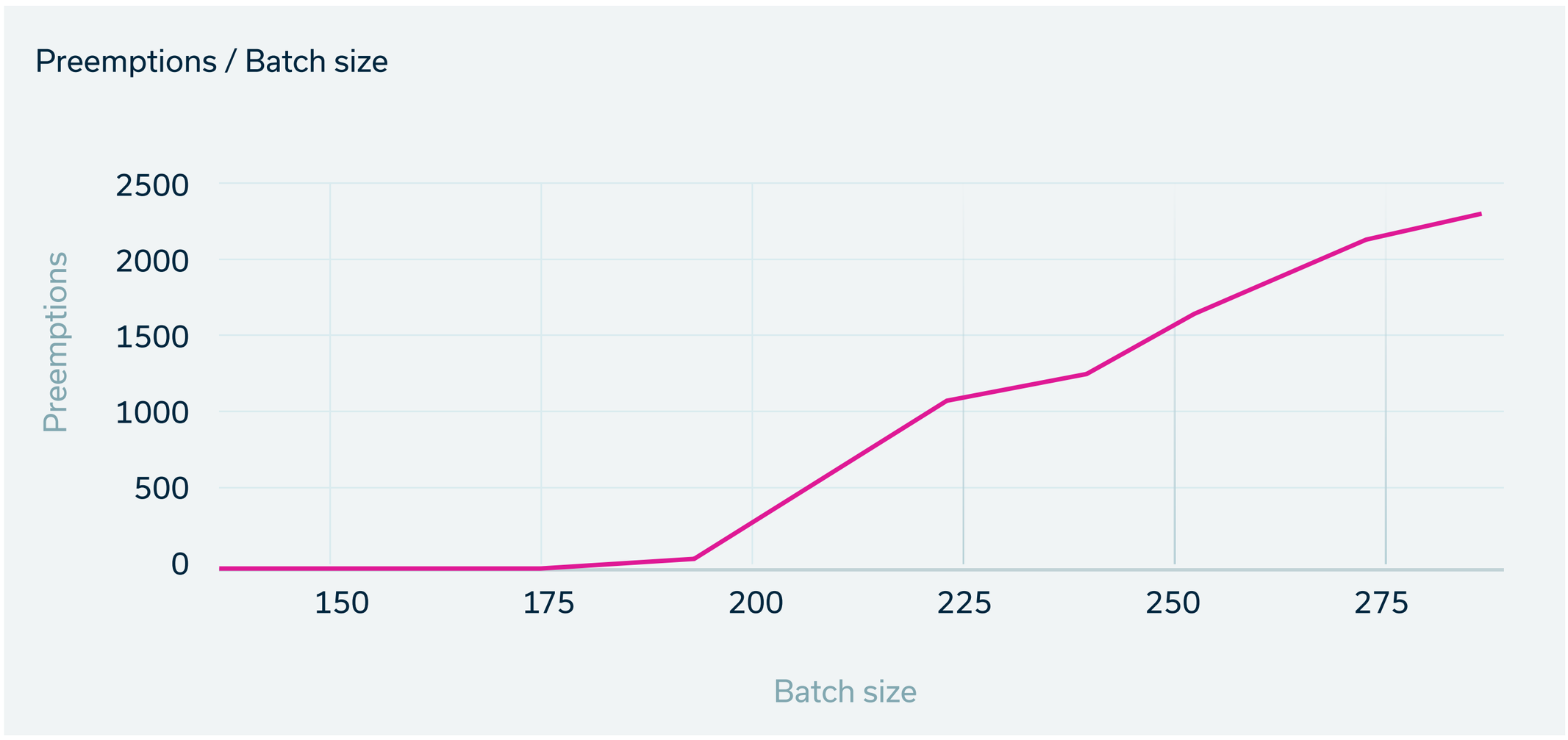


Figure 2: *Number of preemption actions taken dependent on the batch size for user requests. See Appendix for the table of results*

Experiment #3 Throughput with various query rates and variant input for servers and engines

Experiment Setting

Model	Llama-2-7b
Query per second (QPS)	[1, 2, 4, 8, 16, 32, 64, 128]
Input token size	Variant
Output token size	Predefined in each request, derived from the shareGPT prompt
Frameworks	vLLM & TensorRT-LLM & RayLLM & TensorRT-LLM with Triton & TGI

__Table 7:__ Experiment setting to evaluate the throughput of vLLM, TensorRT-LLM, RayLLM, TensorRT-LLM with Triton and TGI

	Engines		**Engine & Server**		
QPS	vLLM	TensorRT-LLM	TGI	RayLLM with vLLM	TensorRT-LLM with Triton
1	228	244	239	2596	2596
2	444	480	478	474	475
4	890	944	341	326	930
8	1454	1242	1509	1199	1064
16	1548	1498	1514	1206	170
32	1552	1500	1521	1215	1093
64	1557	1490	1524	1211	1076
128	1557	1490	1526	1206	1090

__Table 8:__ The throughput results with various QPS rates

In this experiment, we compare vLLM, TRT-LLM, TGI, vLLM with RayLLM and TRT-LLM with Triton Inference Server. Our goal is again to examine the throughput when servers and engines are fully loaded with user requests (QPS) and create a queue of requests. As batch size of vLLM and TRT-LLM, we perform an intermediate experiment to determine the optimal batch sizes and choose the best performed values from that experiment, namely 128 for vLLM and 64 for TensorRT-LLM. We see similar results for each server at both high and low QPS values. At QPS=8, TensorRT-LLM performs slightly worse than TGI and vLLM, but the values converge as QPS increases to 16. We see that all frameworks exhibit a backlog of requests between a QPS of 8 and 16. Apart from that, we also observe overhead when engines are used with servers. It is worth noting that the throughput is getting stabilized after a point, no matter which engine or server is used.

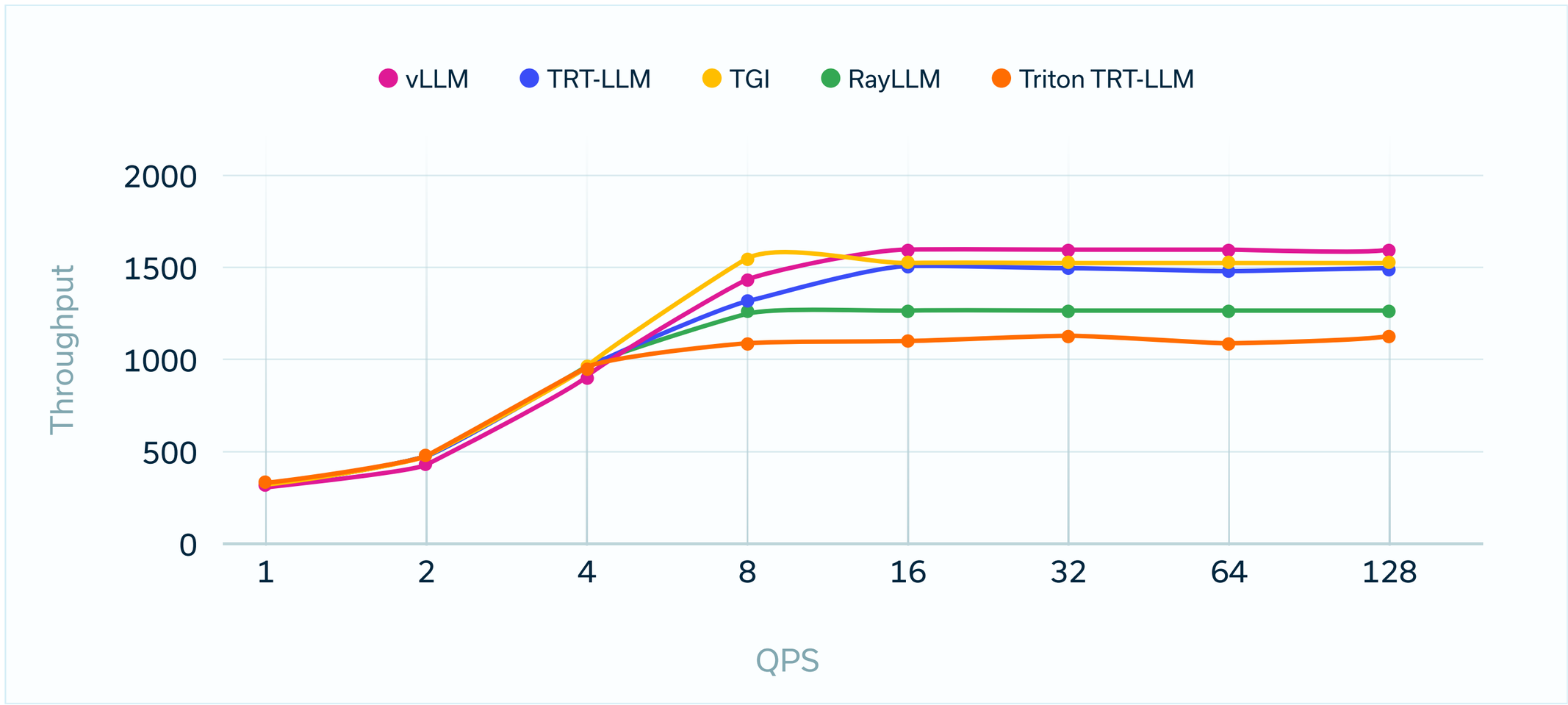


Figure 3: *The throughput results with various QPS rates*

Experiment #4 Throughput analysis of vLLM, TensorRT-LLM and TGI from memory perspective

Experiment Setting

Model	Llama-2-7b Llama-2-7b-chat
Approximate GPU Memory fraction for KV cache	[1, 2, 4, 8, 16, 32, 64, 128]
Query per second (QPS)	32
Input token size	Variant
Output token size	Not specified, model generates EOS
Frameworks	vLLM & TensorRT-LLM & TGI

Table 9: *Experiment setting for memory analysis*

In this experiment, we investigated the impact of available memory for the KV-cache on throughput. To evaluate this, we use two different versions of Llama-2-7b:

- Llama-2-7b, which generates relatively long output sequences until it hits the EOS token.
- Llama-2-7b-chat, which generates relatively short output sequences until it hits EOS token in comparison to Llama-2-7b.

The experiment also involved limiting GPU memory, used for KV cache. Parameters were adjusted using the kv_cache_free_gpu_mem_fraction parameter for TensorRT-LLM, cuda_memory_fraction for TGI, and gpu-memory-utilization for vLLM.

We observed that some engines did not perfectly align with the specified memory fractions, so we manually noted the GPU memory each framework allocated. The model parameters took approximately 15 GB of memory in half precision (float16) on the GPU. The table below displays the GPU memory and approximate fraction that parameters and the KV cache took together.

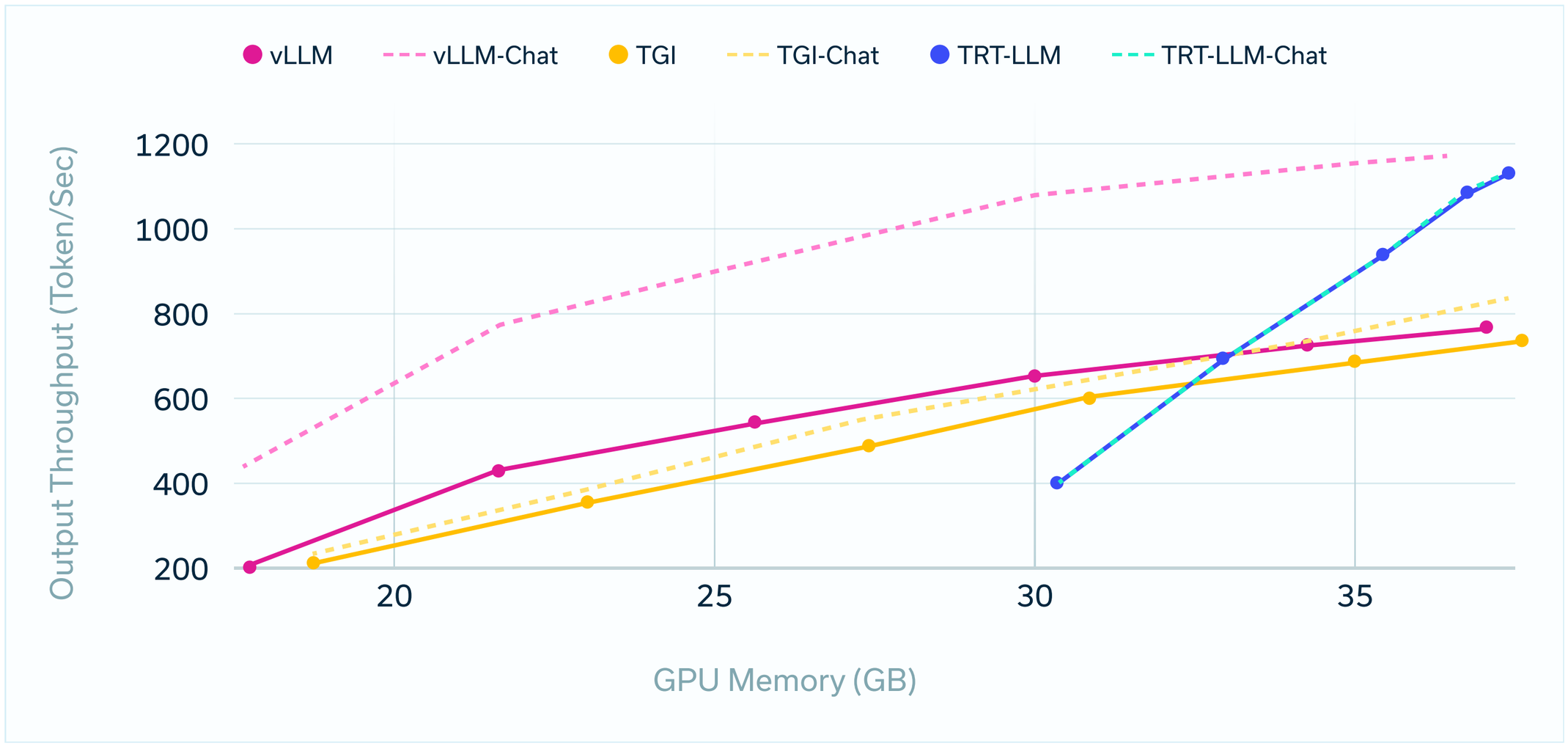


Figure 4: *The output throughput of Llama-2-7b and Llama-2-7b-chat using vLLM, TGI and TensorRT-LLM when GPU memory allocation is limited.*

Examining how memory allocation affects KV cache, we consistently observed improved throughput as memory increased. In the context of the Llama-2-7b model, vLLM exhibits slightly better throughput performance than TGI across various GPU memory fractions. Transitioning to the Llama-2-7b-chat scenario, a notable disparity in throughput performance between TGI and vLLM becomes evident. Here, vLLM outperforms TGI across all fractional memory allocations for KV cache, showcasing superior performance. Importantly, even when we limit memory for the KV cache (17.7 GB), vLLM surpasses its performance with Llama-2-7b, showcasing superior throughput.

When utilizing TensorRT-LLM with both models, we see almost no difference in throughput and memory consumption. While it performs nearly on par with vLLM for higher GPU memory availability in both models, TensorRT-LLM underperforms compared to both engines when the KV cache memory is smaller, such as with a 30 GB allocation. Under these conditions of significantly reduced memory, TensorRT-LLM is unable to generate responses for user requests and instead issues an out-of-memory message.

Experiment #5 Creating a real life scenario: Impact of model size increase & variant input/output sequence length to throughput

Experiment Setting

Model	Llama-2-13b-chat
Batch size	[32, 64, 96, 128, 256] for vLLM [4, 8, 16, 32, 64] for TensorRT-LLM
Query per second (QPS)	[1, 2, 4, 8, 16, 32, 64, 128]
Input token size	Variant
Output token size	Not specified, model generates EOS
Frameworks	vLLM & TensorRT-LLM & RayLLM & TGI & TensorRT-LLM with Triton

Table 10: *Experiment Setting for examining the throughput on a real life scenario.*

In this experiment, we aimed to simulate a real-life scenario by exploring the impact of increased model size, varying input/output sequence lengths, and their influence on throughput. Our objective was to understand the engines' performance when fully loaded with user requests (QPS), creating a queue of requests in a realistic use case. To challenge our memory constraints, we increased the model size to Llama-2-13b-chat and waited until the model generated the end-of-sequence token (EOS).

We initiated the experiment by determining optimal batch sizes for both vLLM and TensorRT-LLM, surpassing the processing speed, leading to a queue of requests.

Batch	TensorRT-LLM
4	261
8	345
16	328
32	355
64	OOM

Table 11: *Throughput of TensorRT-LLM depending on various batch sizes (QPS = 32)*

Batch	vLLM
32	525
64	561
96	561
128	560
256	509

Table 12: *Throughput of vLLM depending on various batch sizes (QPS = 32)*

Due to an out-of-memory error at a batch size of 64 for TensorRT-LLM, further increments in batch size were not included. The experiment proceeded with throughput measurements, utilizing batch sizes of 64 for vLLM and 8 for TensorRT-LLM. For TGI, the highest max-batch-total-tokens fitting into machine memory was employed.

Next, we move to the throughput experiment. Notably, TensorRT-LLM with Triton uses TensorRT-LLM and RayLLM uses vLLM as engines. Therefore, we use the batch sizes of 64 and 16 in those experiments as well. Due to memory restrictions, QPS values of 0.5, 1, 1.5, 2, 4, and 8 were tested.

	Engines		Engine & Server		
QPS	vLLM	TensorRT-LLM	TGI	RayLLM with vLLM	TensorRT-LLM with Triton
0.5	187	209	176	178	122
1	321	398	202	346	240
1.5	423	434	206	478	375
2	550	436	205	479	463
4	563	436	205	470	482
8	535	440	203	468	486

Table 13: *Throughput results of frameworks with various QPS*

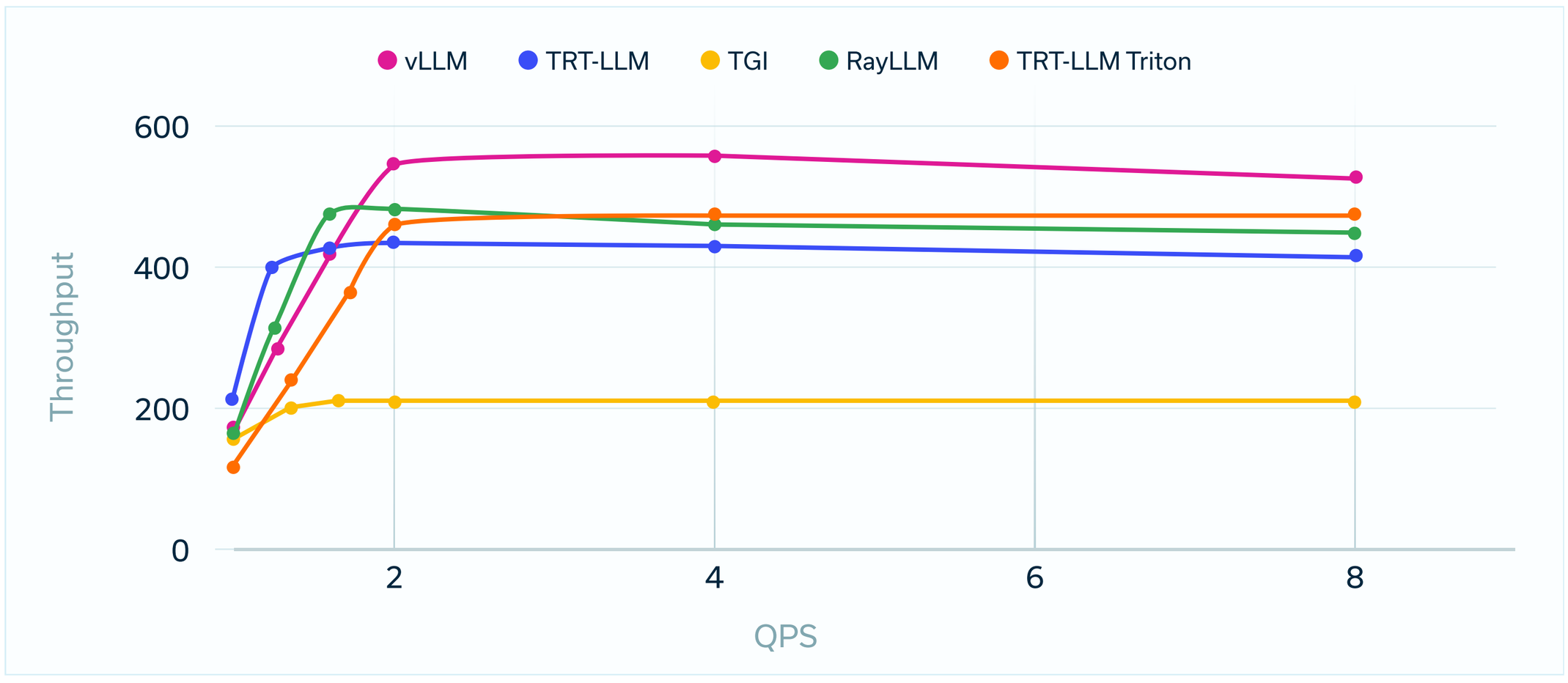


Figure 5: *: Throughput of various frameworks using Llama-2-13b with various QPS rates*

Results indicated that vLLM outperformed other frameworks, followed by TensorRT-LLM with Triton and vLLM-powered RayLLM for higher query rates. Around 2 QPS, all curves began stabilizing, suggesting a backlog of user requests. Notably, throughput reached a stable state after a certain point, irrespective of the framework used.

Discussion & Conclusion

Memory Allocation: A Critical Consideration

Serving a Large Language Model (LLM) involves managing two primary memory components: model parameters and the Key-Value (KV) cache. While model parameters allocate static memory, the KV cache dynamically expands and shrinks based on the number of requests and sequence lengths. The autoregressive nature of LLMs and the large number of weights makes the generation process memory-bound rather than compute-bound. Hence, achieving efficient LLM performance requires careful consideration of memory allocation and consumption, recognizing them as essential optimization metrics.

Preemptions: A Strategic Trade-off

Our investigation reveals a shared characteristic among engines—specifically, vLLM and TensorRT-LLM — employing preemptions to address memory constraints. In Experiment #2, vLLM strategically preempted the KV cache under heavy request loads, mitigating memory challenges at the expense of increased computation. Although TensorRT-LLM's closed-source nature prevented direct exploration in Experiment #2, its behavior in Experiment #1 looks like it implies a similar preemption mechanism at first sight. However, the results of the Experiment #4 implies that there is no such preemption mechanism implemented yet. The strategic approach of preemptions underscores the delicate balance required for memory optimization and computational efficiency in model serving.

Sequence Length Insight for Specific Engines

Certain engines, such as TensorRT-LLM and TGI, allocate memory on the GPU based on the expected KV cache size for upcoming batch sizes and sequence lengths (reservation policy). The requirement for input and output sequence length arguments (e.g., max-batch-total-tokens for TGI) and batch size is important. As an example, if the maximum output isn’t specific in the request, TGI assumes the worst-case scenario (max_total_tokens), which takes up a significant memory on the device. Experiment #3 demonstrates that when the output sequence length matches the expected length, vLLM, TensorRT-LLM, and TGI exhibit similar throughput performances. However, Experiment #4, which involves generating end-of-sequence tokens with variant input sequences, reveals vLLM's superior throughput and ability to serve efficiently while being memory-bound.

vLLM distinguishes itself by handling more requests concurrently, especially when the output is shorter, as seen in Experiment #4 with the Llama-2-7b-chat model. This advantage positions vLLM for better performance in processing multiple requests swiftly, particularly with shorter outputs.

Model Size's Influence on Throughput

An additional consideration applicable to all engines is the impact of model size on the throughput curve. As the model size increases, there is an expected shift to the left in the throughput curve in Figure 4 due to limited memory for batch size, which is needed for KV cahce (memory boundness). More GPU memory or a smaller model allows for exploiting the remaining memory for higher batch sizes, leading to improved throughput. However, it is crucial to recognize that this improvement reaches a threshold. Beyond a certain point, additional GPU memory no longer contributes to higher throughput. This observation emphasizes the intricate interplay between model size, GPU memory utilization, and achievable throughput in the context of language model serving.

Impact of server selection

Experiment #5 introduces the real-world scenario of server selection's influence on throughput. While Experiment #3 hinted at potential overhead when using servers, Experiment #5 surprisingly reveals that TensorRT-LLM with Triton outperforms TensorRT-LLM alone. This finding emphasizes the importance of strategic server selection in enhancing engine performance and influencing overall throughput. A deeper exploration of engine-server combinations promises more detailed insights into optimizing language model serving setups.

Future Work

As we conclude this benchmarking study, several topics present themselves for future exploration and refinement of our findings. The following areas stand out as promising directions for advancing our understanding of large language model (LLM) serving performance:

GPU Parallelism on a single node (Distributed Serving Techniques)

While we used single node and single GPU in our experiments, a deeper exploration into distributed serving techniques and their impact on throughput and latency could offer a more comprehensive view. Investigating how different frameworks and engines scale in a distributed setting may uncover optimizations for achieving even higher throughput and lower latency.

Multi-Node Setting

Extending our experiments to a multi-node setting could provide valuable insights into the scalability of the evaluated frameworks. Assessing the performance of model-serving frameworks across multiple nodes will be crucial for applications demanding distributed and highly available LLMs.

Deeper Dive into Latency

While our benchmarking study primarily focused on throughput as a key performance metric, a deeper exploration into latency considerations is warranted. Understanding the time taken by the server and model to generate output, particularly in real-time scenarios, will be essential for applications where responsiveness is critical.

Engine-Server Combination Discovery

A deeper exploration of engine-server combinations promises more detailed insights into optimizing language model serving setups.

References

1. https://github.com/vllm-project/vllm
2. https://developer.nvidia.com/blog/nvidia-tensorrt-llm-supercharges-large-language-model-inference-on-nvidia-h100-gpus/
3. https://developer.nvidia.com/blog/optimizing-inference-on-llms-with-tensorrt-llm-now-publicly-available/
4. https://developer.nvidia.com/blog/optimizing-inference-on-llms-with-tensorrt-llm-now-publicly-available/
5. https://vllm.readthedocs.io/en/latest/getting_started/quickstart.html
6. https://www.anyscale.com/blog/continuous-batching-llm-inference
7. https://docs.ray.io/en/latest/ray-overview/use-cases.html
8. https://github.com/triton-inference-server/tensorrtllm_backend
9. https://nvidia.github.io/TensorRT-LLM/architecture.html
10. https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices

Appendix

Batch Size	Preemption Count	Time (s)	Rate (p/s)
128	0	Not relevant	0
144	0	Not relevant	0
160	0	Not relevant	0
176	0	Not relevant	0
192	26	238	0.11
208	485	249	1.95
224	1024	253	4.04
240	1294	257	5.03
256	1750	260	6.73
272	2107	266	7.92
288	2355	268	8.8

Table 14: *Detailed overview of the preemption counts using vLLM (Experiment #2)*