LLMs on Affordable GPUs: A Benchmarking Study

David Georg Reichelt
Lancaster University / URZ Leipzig

Daniel Abitz
URZ Leipzig

Jonathan Gross
URZ Leipzig

Stefan Kühne
URZ Leipzig

Abstract

A variety of Large Language Models (LLMs) exist, which have different characteristics regarding the quality of their answers and their resource and time consumption. For most of the LLMs, the required hardware is costly. Nevertheless, to achieve data sovereignty and privacy, hosting LLMs locally is necessary. Due to the high costs, it is crucial to choose LLMs and configurations that are suitable for the given environment. In this benchmarking study, we examine how to provide LLMs for researchers in the context of a local university computing centre. To do so, we extended an implementation of the LLM-as-a-judge principle, and executed it on our infrastructure. Our extended implementation can be used to decide what to host on different hardware, and to decide which new models may fit better.

1 Introduction

Inference in LLMs, i.e., the generation of outputs from a pre-trained model, requires a substantial amount of computing power. If data privacy or sovereignty should be achieved, the models need to be hosted locally. Modern graphics cards, such as the Nvidia H200 GPU with 141 GB RAM, cost around 30,000 Euros at the time of writing. Therefore, they are not affordable for most providers. This requires providers to make a trade-off between costs, the quality of the results, and the response time of the LLM.

In our context, we have 8 nodes with Nvidia L40S with 48 GB available. Their inference capabilities should be made available to researchers. For this purpose, a stable environment with good quality and response time needs to be provided. To parameterize and test our environment, we perform a benchmarking study [2] to parameterize our setup. The goal of this study is to examine the quality and response times of a set of LLMs that has been *predefined* by the users of our research cluster. Models were selected if they are fitting into the VRAM of our cluster and available in a license that is usable on our cluster. A comparison of the concrete VRAM usage or the energy consumption was out of the scope of this research.

For our study, we extend an existing LLM-as-a-judge implementation using the MT-bench benchmark data. Based on these benchmarking data, we compare the different models and examine how on-demand scaling works and how two different execution engines work. We find that (1) Model `vllm-llama-3-3-nemotron-super-49b-v1` gives the highest quality over all used models, (2) on-demand scaling works, but the `targetRequests` needs to be parametrized based on the model and runtime, and (3) the LLM runtime vllm[1] has better performance for parallel use cases than the LLM runtime ollama[2].

The remainder of this paper is organized as follows: First, we give an overview of LLM benchmarks and existing benchmarking approaches in this field. Based on them, we describe our extension of the existing LLM-as-a-judge code. Using this extended benchmark, we present our benchmarking results. Finally, we give a summary and an outlook.

2 LLM Benchmarks

In the following, we first describe how quality benchmarks for LLMs work. Based on this, we describe how resource consumption of LLMs is typically measured. Finally, we discuss existing LLM benchmarking studies of quality and resource consumption.

2.1 Quality Benchmarks

For quality benchmarking, the standard approach is LLM-as-a-judge [4]: Each model is asked to answer a set of questions, for example the questions from the MT-bench benchmark (also introduced in [4]). After the answers are collected, a prompt is used to ask another LLM for a grading for each answer regarding a selection of criteria. These gradings can be done comparing the models pairwise, grading single answers, or grading the answers based on a set of criteria. Finally, the results are used to estimate the average quality of answers of a model in a certain field. This process is summarized in Figure 1.

2.2 Resource Consumption Metrics

A simple response time analysis is not appropriate for LLMs, since the complexity of the request correlates with the response time: A longer prompt results in much longer response times. To overcome this, benchmarking of LLMs usually determines the Time To First Token (TTFT) and the Tokens Per Second (TPS) [3]. The time to first token is crucial for the

[1] https://github.com/vllm-project/vllm
[2] https://ollama.com/

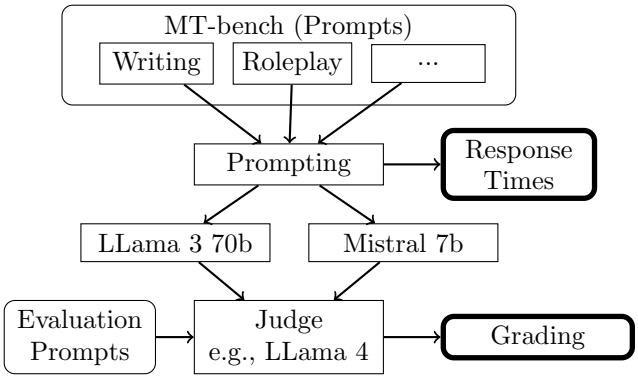

Figure 1: LLM-as-a-Judge Process

users perceived performance, because when the first token is available, the user can start reading the output. The TPS give an indication of how fast the response is generated; if the generation speed is lower than the reading speed of the user, the LLM appears to be slow.

2.3 Existing Comparisons

Tuggener et al. [6] use MT-bench to examine VRAm usage, TPS and power consumption. They examine quantization, low-rank approximators and model pruning to make models runnable with less hardware; they find that according to MT-bench, this causes very small changes to the result quality.

Jie et al. [7] examine how LLMs can be executed on smartphone hardware. They find that LLama 7b with 4 bit quantization, depending on the phone and setup, can have roughly 10 TPS; however, due to the small model size, the quality can be expected to be worse than within our benchmarking study.

Chitty et al. [5] evaluate LLMs on high-end hardware, e.g., NVidia H100 GPUs. Depending on the model size, they find strongly varying TPS values, reaching up to 3 000 TPS with 7B models. In contrast to this work, they use different benchmarks for different environments, e.g., llama-bench for llama.cpp.

3 Benchmark Extension: LLM-as-a-Judge-And-Resource-Measurement

The basic LLM-as-a-judge implementation only benchmarks the response quality, not taking into account the response times.[3] To host the models, it is also necessary to obtain TTFT and TPS, and compare them for different models and configurations.

To do so, we extended the LLM-as-a-judge implementation to also measure response times, TTFT and TPS. Furthermore, we obtained the count of running pods of one model, to check the autoscaler. This required three extensions of the existing implementation: (1) To get TTFT, we switched the implementation to using the streaming API instead of the batch

[3] `https://github.com/lm-sys/FastChat/tree/main/fastchat/llm_judge`

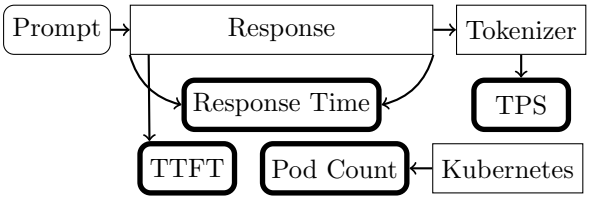

Figure 2: Observability Data Gathering

API. After the first token is received, the time is measured, and by the difference to the start time, the TTFT is calculated. Afterwards, to still store the whole response, all tokens of the streaming response are combined. (2) To obtain the TPS, we tokenize the input and divide the token count by the overall time. (3) To get the pod count, we analyzed the output of `kubectl get pods` for the count of pods of the model that is currently tested. This process is summarized in Figure 2. Furthermore, the original implementation of LLM-as-a-judge requires access to OpenAIs ChatGPT interface. Therefore, we extended the implementation to use any OpenAI compatible REST interface specified by `--openai-api-base`.

4 Result Analysis

We used the extended benchmark for three experiments: The comparison of the quality of different models, the comparison of the Ollama vs. the vllm runtime and the configuration of scaling the pods. All experiments were executed on CentOS 10 using Kernel 5.14.0-503, running on the aforementioned cluster. Our code and result data are available.[4]

4.1 Model Quality Comparison

To compare the quality, we asked ollama-llama3-3-70b to judge all models answers. Figure 3 shows us the results of selected models: Most models only have slight differences, but the deepseek-coder-33b-instruct has worse quality for every aspect excluding coding.

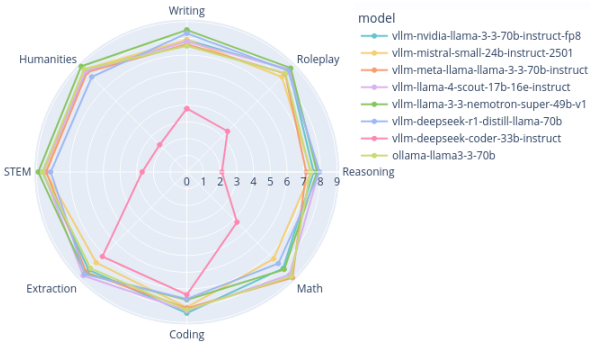

Figure 3: Quality Comparison of Selected Models

Even though the prompt response process is a random process, the result quality is stable: We repeated each prompt 12 times, and compared the grades

[4] `https://doi.org/10.5281/zenodo.16037259`

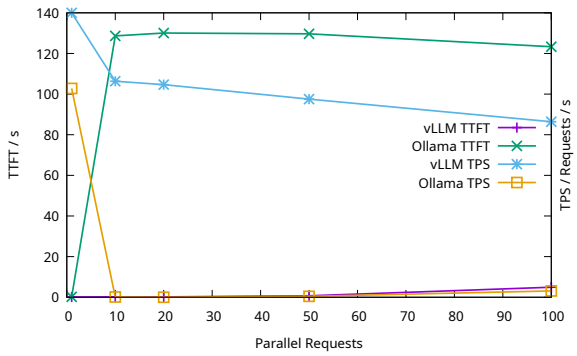

Figure 4: Comparison of the Behaviour of llama3 With ollama and vLLM

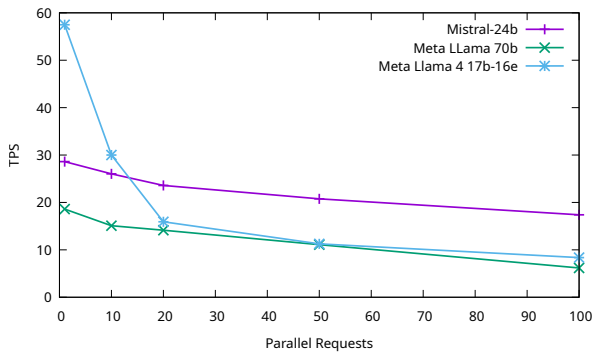

Figure 5: TPS of Models Before Scaling

given by the judge. The maximum relative standard deviation between the different runs was 1.8 % for `vllm-deepseek-coder-33b-instruct-2gpus`.

4.2 Ollama vs. vllm

The LLM runtimes ollama and vllm are tailored for different use cases: ollama is built for local execution on one device and vllm is built for high performance LLM-servers. We evaluated whether one of both is more suitable for our use case by running the same model–llama-3.3-70b–on both runtimes. For this model, Ollama requires one GPU and vllm requires 4 GPUs due to different quantization. The different setup leads to different response times, which is shown in Figure 4: While Ollamas TTFT increases with parallel requests, vLLMs TTFT nearly stays close to 0. At the same time, Ollamas TPS decrease faster than vLLMs TPS. Even with more Ollama pods, Ollama still has higher TTFT and lower TPS. Therefore, for handling parallel requests, using vLLM is more suitable than using Ollama.

4.3 On-Demand Scaling

To examine the scaling behavior, we started the benchmarks with different count of parallel requests. Figure 5 shows that with increasing count of parallel requests, the TPS decrease. The KubeAI autoscaler is mainly configured by the `targetRequests` parameter,[5] which is the average number of active requests that the autoscaler tries to maintain by scaling up or down. If `targetRequests` is set to the default value of 10, the TPS reach the value they have at 10 requests after enough pods are started. However, it takes a few minutes (depending on the model) to start these pods. Decreasing `targetRequests` leads to more pods being started and the pods being started earlier; however, it still takes the same time for the pods to be usable.

vllm is not able to scale to increase the size of one pod, i.e., to scale horizontally. Increased pod sizes, especially more GPUs per pod, might achieve better performance metrics, at the cost of less flexibility to scale single models. Since the current performance metrics are sufficient, we did not examine this option.

[5] `https://www.kubeai.org/reference/kubernetes-api/`

5 Summary and Outlook

In this work, we presented how we benchmarked multiple LLM models in order to have them running locally on our cluster. Our benchmark extension and experiment setup can be used for evaluating additional LLMs, or evaluating similar LLMs on different hardware. For extension of this work, we see two main directions: Additional observability metrics like current VRAM usage or system TPS should be included. For analysis also in production settings, automated analysis of these observability data is necessary, for example by extension of the Kieker framework [8].

Second, using fine-tuning or RAG [1], models can be made more efficient for specific tasks. By extending the setup, the benchmark could be used for models that are adapted for specific purposes.

Acknowledgements Computations for this work were done (in part) using resources of the Leipzig University Computing Center.

References

[1] P. Lewis et al. “Retrieval-augmented generation for knowledge-intensive nlp tasks”. In: *Advances in neural information processing systems* 33 (2020). DOI: 10.48550/arXiv.2005.11401.

[2] W. Hasselbring. “Benchmarking as empirical standard in software engineering research”. In: *Proceedings of 25th ICEASE*. 2021. DOI: 10.1145/3463274.3463361.

[3] X. Miao et al. “Towards efficient generative large language model serving: A survey from algorithms to systems”. In: (2023). DOI: 10.48550/arXiv.arXiv.2312.15234.

[4] L. Zheng et al. “Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena”. In: *Advances in Neural Information Processing Systems* 36 (2023). DOI: 10.48550/arXiv.2306.05685.

[5] K. T. Chitty-Venkata et al. “LLM-Inference-Bench: Inference Benchmarking of Large Language Models on AI Accelerators”. In: *SC24-W: Workshops of the ICHPCNSA*. IEEE. 2024, pp. 1362–1379.

[6] L. Tuggener et al. “So you want your private LLM at home? A survey and benchmark of methods for efficient GPTs”. In: *2024 11th IEEE SDS*. 2024, pp. 205–212. DOI: 10.1109/SDS60720.2024.00036.

[7] J. Xiao et al. “Understanding Large Language Models in Your Pockets: Performance Study on COTS Mobile Devices”. In: (2024). DOI: 10.48550/arXiv.2410.03613.

[8] S. Yang et al. “The Kieker Observability Framework Version 2”. In: *Companion of the 16th ACM/SPEC ICPE*. 2025, pp. 11–15. DOI: 10.1145/3680256.3721972.