Project: A study of prefix sharing in LLM serving

Description

Prefix sharing is an important optimization in LLM serving. It enables reuse of KV cache across requests that share identical input prefixes. By avoiding redundant computation on repeated prefixes, prefix sharing reduces inference latency and improves system throughput. Modern serving engines such as vLLM have already integrated this optimization to enhance efficiency.

However, the performance benefits of prefix sharing depend on a variety of system-level factors, including cache capacity, block size, eviction policy, and workload characteristics. Despite its practical importance, there remains a lack of comprehensive analysis on how these factors interact and influence the effectiveness of prefix sharing---largely because this is a recently emerging research area.

In this project, we aim to fill this gap by conducting a systematic and quantitative study of prefix sharing in LLM serving systems, identifying key design trade-offs, and uncovering optimization opportunities for future serving frameworks.

Resource requirements

- A CPU-based machine. Could be your windows/mac laptop
- Git, GitHub
- Python programming
- AI coding assistant (optional), e.g., Cursor, Copilot, Claude code.

Project milestones

This project will have the following main milestones.

Prerequisite: vLLM setup

First, you need to install the most popular open-source inference engine: vLLM. To ensure that we all have the same vLLM version, we will checkout to a specific commit. Pleasing using the following way to clone vLLM:

```
git clone https://github.com/vllm-project/vllm.git
git checkout 94d545a1a18e20ea8763a6760194589b8a3c9065
```

Next, follow the vLLM installation instructions to set up the environment. Since we will later modify its implementation for this project, it is important to install vLLM from source rather than through a prebuilt package (e.g., pip install vllm). Installing from source allows direct access to the internal codebase, enabling you to experiment with and extend its components.

```
cd vllm
VLLM_USE_PRECOMPILED=1 uv pip install --editable .
```

Milestone 1: Change vLLM to a prefix sharing simulator

vLLM is primarily designed to serve LLMs on GPUs. However, since not all environments have GPU resources---and because studying prefix sharing does not depend on GPU computation---we will modify vLLM to bypass GPU execution.

In this milestone, you will hack the vLLM source code to bypass actual GPU inference. "Bypass" here means skipping the real GPU execution while keeping all other components functional, including continuous batching, scheduling, and request management, etc. In other words, your modified implementation should make vLLM believe that inference is running on GPUs, while in reality, the computation is skipped, and the inference outputs are generated by a simulator.

Specifically, vLLM adopts an iteration-level scheduling strategy: in each iteration, it either processes all tokens in the prompt (prefill phase) or generates one new token (decode phase). In this project, we will use pre-collected LLM inference traces (details to be provided in later milestones) that contain all prompts and their corresponding responses. This allows us to simulate the inference process using these traces. Instead of performing real token generation on GPUs, your modified vLLM will return the tokens directly from the trace data, effectively emulating end-to-end inference behavior while bypassing GPU execution.

To achieve this, consider implementing a dedicated Simulator class (in a separate file) that provides these simulation functions, and modify vLLM to invoke the corresponding methods in this class during execution.

The core you might need to change includes but is not limited to the following:

- vllm/core/scheduler.py
- vllm/engine/llm_engine.py
- vllm/engine/multiprocessing/engine.py

Note 1: vLLM has v0 and v1 two versions. In this project, we will all use v0. v0's code is directly under folder "vllm/vllm", while v1's code is under folder "vllm/vllm/v1".

Note 2: vLLM has online inference and offline batch inference two modes, which could have different engine entries. In this project, we will all use the online inference mode.

Please write a set of tests to verify that your modified vLLM implementation works correctly before proceeding to the next milestone.

Milestone 2: Prefix sharing for homogeneous chatbot workload

In this milestone, we will use the simulator developed earlier to evaluate the effectiveness of prefix sharing in one of the most common LLM workloads---chatbots. Specifically, we will utilize the ShareGPT dataset (https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered), which contains real-world multi-turn conversation traces collected from user-shared ChatGPT interactions. This dataset provides realistic conversational contexts that allow us to analyze how prefix sharing can improve inference efficiency in practical chatbot scenarios.

Task 1: Write a client simulator

First, we need to write a client simulator that replays the collected prompts in the trace as input for the models served by vLLM. There are several things needed to pay attention to

- **Timing:** Some traces have the request sending time. If so, the client simulator can directly use it. If not, let's use poisson distribution to simulate the request sending time.
- **Chat templates:** Some models have specific templates for their prompt. For example, Llama-3.2-1B-Instruct has the following template. The client simulator needs to take this into consideration and format the prompt using the right template.

```
>>> FROM TRANSFORMERS IMPORT AUTOTOKENIZER
>>> TOKENIZER = AUTOTOKENIZER.FROM_PRETRAINED("/ROOT/NFS/DOWNLOAD/LLAMA-3.2-1B-
INSTRUCT")
>>> PRINT(TOKENIZER.CHAT_TEMPLATE)
{{- BOS_TOKEN }}
{%- IF CUSTOM_TOOLS IS DEFINED %}
  {%- SET TOOLS = CUSTOM_TOOLS %}
{%- ENDIF %}
{%- IF NOT TOOLS_IN_USER_MESSAGE IS DEFINED %}
  {%- SET TOOLS_IN_USER_MESSAGE = TRUE %}
{%- ENDIF %}
{%- IF NOT DATE_STRING IS DEFINED %}
  {%- IF STRFTIME_NOW IS DEFINED %}
    {%- SET DATE_STRING = STRFTIME_NOW("%D %B %Y") %}
  {%- ELSE %}
    {%- SET DATE_STRING = "26 JUL 2024" %}
  {%- ENDIF %}
.......
```

Task 2: Replay the ShareGPT trace

Now, let's replay the ShareGPT trace to evaluate the effectiveness of prefix sharing. Consider replaying the trace under the following two settings:

- **Single-turn conversation:** For each conversation, replay only the first turn---that is, the first user prompt and its corresponding model response.
- **Multi-turn conversation:** Replay the entire conversation across all users and model turns.

While running these experiments, please measure the following prefix sharing-related statistics:

- The fraction of each request that benefits from prefix sharing.
- The number of hits per cache block.
- The time gap between reuses of each cache block.
- Any additional metrics you find interesting or relevant to prefix sharing.

When presenting the results, think carefully about how to visualize them---for example, using CDFs, line charts, or bar charts to best illustrate the observed trends.

Finally, analyze your results and discuss why the patterns occur.

Task 3: Tune the cache parameters

In the previous step, we used the default cache parameters. Now, we will tune the cache configurations to study how they influence prefix sharing behavior. The key parameters to explore include:

- Prefix cache size --- the amount of GPU memory allocated for prefix caching (KV cache)
- Block size --- the number of tokens in one block
- Eviction policy --- the strategy used to evict cache entries (e.g., LRU, LFU, FIFO).

Experiment with different values of these parameters and observe how they affect prefix sharing efficiency. Consider what figures or visualizations can best illustrate your findings.

Milestone 3: Cache eviction for heterogeneous workload

In this milestone, we will extend our study to multiple LLM workloads. Specifically, we aim to analyze the impact of different cache eviction policies across diverse workloads. For this experiment, you may fix the prefix cache size and block size while varying only the eviction policy.

Please explore different types of LLM workloads beyond the chatbot example used earlier. A few examples are provided below, but you are encouraged to discover and include additional workloads from HuggingFace (https://huggingface.co/datasets) or other public sources.

- AgentBank: https://huggingface.co/datasets/Solaris99/AgentBank. A large-scale trajectory tuning dataset featuring over 50,000 diverse high-quality interaction trajectories. It spans 16 tasks across five agent skill dimensions, each annotated with detailed chain-of-thought (CoT) rationales for every action step.
- CC-Bench-trajectorie: https://huggingface.co/datasets/zai-org/CC-Bench-trajectories. A benchmark designed to evaluate the agentic coding abilities of models like GLM-4.6 in realistic programming scenarios.
- qwen-bailian-usagetraces-anon: https://github.com/alibaba-edu/qwen-bailian-usagetraces-anon. This dataset is used in paper Optimizing KVCache cache design. (KVCache@ATC'25). It contains a two-hour anonymized KVCache trace collected from a Qwen model instance deployed on Aliyun Bailian.

Task 1: The optimal eviction policy for each workload

First, experiment with different cache eviction policies across the various workloads and determine which policy performs best for each workload. Here, you may want to report the effectiveness of prefix caching under different eviction strategies to identify the most suitable policy.

Task 2: The optimal eviction policy for mixed workload

In practice, an LLM often encounters a mixture of the above workloads. What would be the optimal eviction policy in such a scenario? In this task, please design an eviction policy that performs best under a mixed workload.

Things to consider:

- What is the difference in their KV cache reusing time interval?
- What is the difference in their KV cache reusing portion?
- How do I know a request belongs to which workload type?

Please compare the effectiveness of your designed eviction policy against the baselines (e.g., LRU and LFE for all requests.)

About teamwork

This is a team-based project. A team could have 3-4 members; other team size needs to discuss with the instructor. You can find team members offline or on Piazza or other approaches. The final team member information needs to be recorded here:
https://docs.google.com/spreadsheets/d/13-aiWNPrrwLhBXq58WWfx7LeP56eOJn9AliYA4S0Lko/edit?usp=sharing

Note that all team members will share the same grade.

Submission

The project must be submitted on time as specified on Canvas. Each team only needs to submit one copy of the project.

Your submission should include both the code and a technical report.

Code: Submit your code as a **.git folder**, ensuring that all commit histories are preserved so that changes can be tracked.

- At the root of the code directory, include a **README** file with clear instructions on how to set up and run your code.

Technical report (no longer than 6 pages): Submit your technical report as a pdf file.

- The technical report should talk about your design and results of the above milestones.
- No longer than 6 pages in total excluding references and appendix if you have.
- You don't have to write 6 pages if you feel shorter length is sufficient. We will not grade based on your pages.
- Use the MLSys 2025 latex format here: https://media.mlsys.org/Conferences/MLSYS2025/mlsys2025style.zip

Due: 12/10/2025 11:59pm. If this deadline differs from the time shown on Canvas, please follow the Canvas deadline.

About using AI

You are welcome to use any AI tools (e.g., ChatGPT, GitHub Copilot, Cursor, Claude, etc.) to assist with your project. However, you must fully understand the code generated by these tools.

In your technical report, please include a dedicated section describing how you used AI during the project. If certain AI prompts were particularly helpful (for example, they generated a significant portion of your code), please include those prompts in the technical report appendix.