lmdeploy Documentation

Release 0.4.1

LMDeploy Contributors

May 17, 2024

GET STARTED

CHAPTER

ONE

GET STARTED

LMDeploy offers functionalities such as model quantization, offline batch inference, online serving, etc. Each function can be completed with just a few simple lines of code or commands.

1.1 Installation

Install lmdeploy with pip (python 3.8+) or *from source*

```
pip install lmdeploy
```

The default prebuilt package is compiled on **CUDA 12**. However, if CUDA 11+ is required, you can install lmdeploy by:

```
export LMDEPLOY_VERSION=0.3.0
export PYTHON_VERSION=38
pip install https://github.com/InternLM/lmdeploy/releases/download/v${LMDEPLOY_VERSION}/
↪lmdeploy-${LMDEPLOY_VERSION}+cu118-cp${PYTHON_VERSION}-cp${PYTHON_VERSION}-
↪manylinux2014_x86_64.whl --extra-index-url https://download.pytorch.org/whl/cu118
```

1.2 Offline batch inference

```
import lmdeploy
pipe = lmdeploy.pipeline("internlm/internlm-chat-7b")
response = pipe(["Hi, pls intro yourself", "Shanghai is"])
print(response)
```

For more information on inference pipeline parameters, please refer to *here*.

1.3 Serving

LMDeploy offers various serving methods, choosing one that best meet your requirements.

- Serving with openai compatible server
- Serving with docker
- Serving with gradio

1.4 Quantization

LMDeploy provides the following quantization methods. Please visit the following links for the detailed guide

- *4bit weight-only quantization*
- *k/v quantization*
- *w8a8 quantization*

1.5 Useful Tools

LMDeploy CLI offers the following utilities, helping users experience LLM features conveniently

1.5.1 Inference with Command line Interface

```
lmdeploy chat internlm/internlm-chat-7b
```

1.5.2 Serving with Web UI

LMDeploy adopts gradio to develop the online demo.

```
# install dependencies
pip install lmdeploy[serve]
# launch gradio server
lmdeploy serve gradio internlm/internlm-chat-7b
```

CHAPTER

TWO

BUILD FROM SOURCE

LMDeploy provides prebuilt package that can be easily installed by `pip install lmdeploy`.

If you have requests to build lmdeploy from source, please clone lmdeploy repository from GitHub, and follow instructions in next sections

```
git clone --depth=1 https://github.com/InternLM/lmdeploy
```

2.1 Build in Docker (recommended)

We highly advise using the provided docker image for lmdeploy build to circumvent complex environment setup.

The docker image is `openmmlab/lmdeploy-builder:cuda11.8`. Make sure that docker is installed before using this image.

In the root directory of the lmdeploy source code, please run the following command:

```
# the home folder of lmdeploy source code
cd lmdeploy
bash builder/manywheel/build_all_wheel.sh
```

All the wheel files for lmdeploy under py3.8 - py3.11 will be found in the `builder/manywheel/cuda11.8_dist` directory, such as,

```
builder/manywheel/cuda11.8_dist/
├── lmdeploy-0.0.12-cp310-cp310-manylinux2014_x86_64.whl
├── lmdeploy-0.0.12-cp311-cp311-manylinux2014_x86_64.whl
├── lmdeploy-0.0.12-cp38-cp38-manylinux2014_x86_64.whl
└── lmdeploy-0.0.12-cp39-cp39-manylinux2014_x86_64.whl
```

If the wheel file for a specific Python version is required, such as py3.8, please execute:

```
bash builder/manywheel/build_wheel.sh py38 manylinux2014_x86_64 cuda11.8 cuda11.8_dist
```

And the wheel file will be found in the `builder/manywheel/cuda11.8_dist` directory.

You can use `pip install` to install the wheel file that matches the Python version on your host machine.

2.2 Build in localhost (optional)

Firstly, please make sure gcc version is no less than 9, which can be conformed by `gcc --version`.

Then, follow the steps below to set up the compilation environment:

- install the dependent packages:

```
pip install -r requirements.txt
apt-get install rapidjson-dev
```

- install nccl, and set environment variables:

```
export NCCL_ROOT_DIR=/path/to/nccl
export NCCL_LIBRARIES=/path/to/nccl/lib
```

- install openmpi from source:

```
wget https://download.open-mpi.org/release/open-mpi/v4.1/openmpi-4.1.5.tar.gz
tar xf openmpi-4.1.5.tar.gz
cd openmpi-4.1.5
./configure --prefix=/usr/local/openmpi
make -j$(nproc) && make install
export PATH=$PATH:/usr/local/openmpi/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/openmpi/lib
```

- build and install lmdeploy libraries:

```
# install ninja
apt install ninja-build
# the home folder of lmdeploy
cd lmdeploy
mkdir build && cd build
sh ../generate.sh
ninja -j$(nproc) && ninja install
```

- install lmdeploy python package:

```
cd ..
pip install -e .
```

CHAPTER

THREE

PROFILE TOKEN LATENCY AND THROUGHPUT

We profile the latency and throughput of generated tokens with fixed batch size and fixed input/output token.

The profiling script is `profile_generation.py`. Before running it, please install the lmdeploy precompiled package and download the profiling script:

```
pip install lmdeploy
git clone --depth=1 https://github.com/InternLM/lmdeploy
```

3.1 Metrics

LMDeploy records test results like first token latency, token throughput (tokens/s), percentile data of each token's latency (P50, P75, P95, P99), GPU mem, etc.

`first_token_latency` is only reported in the case of streaming inference.

The formula for calculating `throughput` is:

$$ TokenThroughput = Number\ of\ generated\ tokens/TotalTime $$

Total time includes prefill time.

During the test process, all graphics cards on the node should not run any other programs, otherwise the statistics of GPU mem would be inaccurate.

3.2 Profile

In this section, we take internlm/internlm-7b as an example to show how to profile the inference engines of LMDeploy.

3.2.1 Profile turbomind engine

```
cd lmdeploy/benchmark
python3 profile_generation.py internlm/internlm-7b
```

3.2.2 Profile pytorch engine

```
cd lmdeploy/benchmark
python3 profile_generation.py internlm/internlm-7b --backend pytorch
```

For detailed argument specification of `profile_generation.py`, such as batch size, input and output token number an so on, please run the help command `python3 profile_generation.py -h`.

CHAPTER

FOUR

PROFILE REQUEST THROUGHPUT

In the applications, the length of the user's input prompt and the size of generated tokens are dynamic. The static inference performance is insufficient to reflect the inference engine's ability to handle the dynamic characteristics.

Therefore, it is necessary to use real dialogue data to evaluate the dynamic inference capabilities of the inference engine. This article will introduce how to test the dynamic inference performance of LMDeploy on localhost.

The profiling script is `profile_throughput.py`. Before running it, please install the lmdeploy precompiled package, download the profiling script and the test dataset:

```
pip install lmdeploy
git clone --depth=1 https://github.com/InternLM/lmdeploy
cd lmdeploy/benchmark
wget https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered/resolve/
↪main/ShareGPT_V3_unfiltered_cleaned_split.json
```

4.1 Metrics

LMDeploy records the performance metrics like first token latency, token throughput (tokens/s) and request throughput (RPM)

`first_token_latency` is only reported in the case of streaming inference.

The formula for calculating `token throughput` is:

$$ TokenThroughput = Number\ of\ generated\ tokens/TotalTime $$

And the formula for calculating `request throughput` is:

$$ RPM(request\ per\ minute) = Number\ of\ prompts/TotalTime * 60 $$

Total time includes prefill time.

4.2 Profile

In this section, we take internlm/internlm-7b as an example to show how to profile the inference engines of LMDeploy.

4.2.1 Profile turbomind engine

```
python3 profile_throughput.py ./ShareGPT_V3_unfiltered_cleaned_split.json internlm/
↪internlm-7b
```

4.2.2 Profile pytorch engine

```
python3 profile_throughput.py ./ShareGPT_V3_unfiltered_cleaned_split.json internlm/
↪internlm-7b  --backend pytorch
```

For detailed argument specification of `profile_throughput.py`, such as request concurrency, sampling parameters, k/v cache memory percentage an so on, please run the help command `python3 profile_throughput.py -h`.

CHAPTER

FIVE

PROFILE API SERVER

The way to profiling `api_server` performance is similar to the method for *profiling throughput*. The difference is `api_server` should be launched successfully before testing.

The profiling script is `profile_restful_api.py`. Before running it, please install the lmdeploy precompiled package, download the script and the test dataset:

```
pip install lmdeploy
git clone --depth=1 https://github.com/InternLM/lmdeploy
cd lmdeploy/benchmark
wget https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered/resolve/
↪main/ShareGPT_V3_unfiltered_cleaned_split.json
```

5.1 Metrics

LMDeploy records the performance metrics like first token latency, token throughput (tokens/s) and request throughput (RPM)

`first_token_latency` is only reported in the case of streaming inference.

The formula for calculating `token throughput` is:

$$ TokenThroughput = Number\ of\ generated\ tokens/TotalTime $$

And the formula for calculating `request throughput` is:

$$ RPM(request\ per\ minute)=Number\ of\ prompts/TotalTime * 60 $$

Total time includes prefill time.

5.2 Profile

In this section, we take internlm/internlm-7b as an example to show the benchmark procedure.

5.2.1 Launch api_server

```
lmdeploy serve api_server internlm/internlm-7b
```

If you would like to change the server's port or other parameters, such as inference engine, max batch size and etc., please run `lmdeploy serve api_server -h` or read *this* guide to get the detailed explanation.

5.2.2 Profile

```
python3 profile_restful_api.py http://0.0.0.0:23333 internlm/internlm-7b ./ShareGPT_V3_
↪unfiltered_cleaned_split.json
```

For detailed argument specification of `profile_restful_api.py`, such as request concurrency, sampling parameters an so on, please run the help command `python3 profile_restful_api.py -h`.

CHAPTER

SIX

PROFILE TRITON INFERENCE SERVER

Triton Inference Server (TIS) is another serving method supported by LMDeploy besides `api_server`. Its performance testing methods and metrics are similar to those of *api_server*.

The profiling script is `profile_serving.py`. Before running it, please install the lmdeploy precompiled package, download the profiling script and the test dataset:

```
pip install 'lmdeploy[serve]'
git clone --depth=1 https://github.com/InternLM/lmdeploy
cd lmdeploy/benchmark
wget https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered/resolve/
↪main/ShareGPT_V3_unfiltered_cleaned_split.json
```

6.1 Metrics

LMDeploy records the performance metrics like first token latency, token throughput (tokens/s) and request throughput (RPM)

`first_token_latency` is only reported in the case of streaming inference.

The formula for calculating `token throughput` is:

$$ TokenThroughput=Number\ of\ generated\ tokens/TotalTime $$

And the formula for calculating `request throughput` is:

$$ RPM(request\ per\ minute)=Number\ of\ prompts/TotalTime * 60 $$

Total time includes prefill time.

6.2 Profile

In this section, we take internlm/internlm-7b as an example to show the benchmark procedure.

6.2.1 Launch triton inference server

Before launching the server, the LLM model must be converted to the turbomind format in advance.

```
lmdeploy convert internlm internlm/internlm-7b --dst-path ./internlm-7b --trust-remote-
↪code
```

Then, the triton inference server can be launched by:

```
bash ./internlm-7b/service_docker_up.sh
```

6.2.2 Profile

```
python3 profile_serving.py 0.0.0.0:33337 ./internlm-7b/triton_models/tokenizer ./
↪ShareGPT_V3_unfiltered_cleaned_split.json
```

For detailed argument specification of `profile_serving.py`, such as request concurrency, sampling parameters an so on, please run the help command `python3 profile_serving.py -h`.

CHAPTER

SEVEN

EVALUATE LLMS WITH OPENCOMPASS

The LLMs accelerated by lmdeploy can be evaluated with OpenCompass.

7.1 Setup

In this part, we are going to setup the environment for evaluation.

7.1.1 Install lmdeploy

Install lmdeploy through pip (python 3.8+). If you want to install from source, you can refer to *build.md*.

```
pip install lmdeploy
```

7.1.2 Install OpenCompass

Install OpenCompass from source. Refer to installation for more information.

```
git clone https://github.com/open-compass/opencompass.git
cd opencompass
pip install -e .
```

At present, you can check the Quick Start to get to know the basic usage of OpenCompass.

7.1.3 Download datasets

Download the core datasets

```
# Run in the OpenCompass directory
cd opencompass
wget https://github.com/open-compass/opencompass/releases/download/0.1.8.rc1/
↪OpenCompassData-core-20231110.zip
unzip OpenCompassData-core-20231110.zip
```

7.2 Prepare Evaluation Config

OpenCompass uses the configuration files as the OpenMMLab style. One can define a python config and start evaluating at ease. OpenCompass has supported the evaluation for lmdeploy's TurboMind engine using python API.

7.2.1 Dataset Config

In the home directory of OpenCompass, we are writing the config file `$OPENCOMPASS_DIR/configs/eval_lmdeploy.py`. We select multiple predefined datasets and import them from OpenCompass base dataset configs as `datasets`.

```
from mmengine.config import read_base


with read_base():
    # choose a list of datasets
    from .datasets.mmlu.mmlu_gen_a484b3 import mmlu_datasets
    from .datasets.ceval.ceval_gen_5f30c7 import ceval_datasets
    from .datasets.SuperGLUE_WiC.SuperGLUE_WiC_gen_d06864 import WiC_datasets
    from .datasets.SuperGLUE_WSC.SuperGLUE_WSC_gen_7902a7 import WSC_datasets
    from .datasets.triviaqa.triviaqa_gen_2121ce import triviaqa_datasets
    from .datasets.gsm8k.gsm8k_gen_1d7fe4 import gsm8k_datasets
    from .datasets.race.race_gen_69ee4f import race_datasets
    from .datasets.crowspairs.crowspairs_gen_381af0 import crowspairs_datasets
    # and output the results in a chosen format
    from .summarizers.medium import summarizer

datasets = sum((v for k, v in locals().items() if k.endswith('_datasets')), [])
```

7.2.2 Model Config

This part shows how to setup model config for LLMs. Let's check some examples:

internlm-20b

internlm-chat-20b

```
from opencompass.models.turbomind import TurboMindModel

internlm_20b = dict(
        type=TurboMindModel,
        abbr='internlm-20b-turbomind',
        path="internlm/internlm-20b",  # this path should be same as in huggingface
        engine_config=dict(session_len=2048,
                           max_batch_size=8,
                           rope_scaling_factor=1.0),
        gen_config=dict(top_k=1, top_p=0.8,
                        temperature=1.0,
                        max_new_tokens=100),
        max_out_len=100,
        max_seq_len=2048,
        batch_size=8,
```

(continues on next page)

(continued from previous page)

```
        concurrency=8,
        run_cfg=dict(num_gpus=1, num_procs=1),
    )

models = [internlm_20b]
```

For Chat models, you have to pass `meta_template` for chat models. Different Chat models may have different `meta_template` and it's important to keep it the same as in training settings. You can read meta_template for more information.

```
from opencompass.models.turbomind import TurboMindModel

internlm_meta_template = dict(round=[
    dict(role='HUMAN', begin='<|User|>:', end='\n'),
    dict(role='BOT', begin='<|Bot|>:', end='<eoa>\n', generate=True),
],
                              eos_token_id=103028)

internlm_chat_20b = dict(
    type=TurboMindModel,
    abbr='internlm-chat-20b-turbomind',
    path='internlm/internlm-chat-20b',
    engine_config=dict(session_len=2048,
                       max_batch_size=8,
                       rope_scaling_factor=1.0),
    gen_config=dict(top_k=1,
                    top_p=0.8,
                    temperature=1.0,
                    max_new_tokens=100),
    max_out_len=100,
    max_seq_len=2048,
    batch_size=8,
    concurrency=8,
    meta_template=internlm_meta_template,
    run_cfg=dict(num_gpus=1, num_procs=1),
    end_str='<eoa>'
)

models = [internlm_chat_20b]
```

Note

- If you want to pass more arguments for `engine_configgen_config` in the evaluation config file, please refer to TurbomindEngineConfig and EngineGenerationConfig

7.3 Execute Evaluation Task

After defining the evaluation config, we can run the following command to start evaluating models. You can check Execution Task for more arguments of `run.py`.

```
# in the root directory of opencompass
python3 run.py configs/eval_lmdeploy.py --work-dir ./workdir
```

CHAPTER

EIGHT

SUPPORTED MODELS

8.1 Models supported by TurboMind

"-" means not verified yet.

Note: The TurboMind engine doesn't support window attention. Therefore, for models that have applied window attention and have the corresponding switch "use_sliding_window" enabled, please choose the PyTorch engine for inference.

8.2 Models supported by PyTorch

CHAPTER

NINE

LLM OFFLINE INFERENCE PIPELINE

In this tutorial, We will present a list of examples to introduce the usage of `lmdeploy.pipeline`.

You can overview the detailed pipeline API in this guide.

9.1 Usage

- **An example using default parameters:**

```
from lmdeploy import pipeline

pipe = pipeline('internlm/internlm2-chat-7b')
response = pipe(['Hi, pls intro yourself', 'Shanghai is'])
print(response)
```

In this example, the pipeline by default allocates a predetermined percentage of GPU memory for storing k/v cache. The ratio is dictated by the parameter `TurbomindEngineConfig.cache_max_entry_count`.

There have been alterations to the strategy for setting the k/v cache ratio throughout the evolution of LMDeploy. The following are the change histories:

1. `v0.2.0 <= lmdeploy <= v0.2.1`

   `TurbomindEngineConfig.cache_max_entry_count` defaults to 0.5, indicating 50% GPU **total memory** allocated for k/v cache. Out Of Memory (OOM) errors may occur if a 7B model is deployed on a GPU with memory less than 40G. If you encounter an OOM error, please decrease the ratio of the k/v cache occupation as follows:

   ```
   from lmdeploy import pipeline, TurbomindEngineConfig

   # decrease the ratio of the k/v cache occupation to 20%
   backend_config = TurbomindEngineConfig(cache_max_entry_count=0.2)

   pipe = pipeline('internlm/internlm2-chat-7b',
                   backend_config=backend_config)
   response = pipe(['Hi, pls intro yourself', 'Shanghai is'])
   print(response)
   ```

2. `lmdeploy > v0.2.1`

   The allocation strategy for k/v cache is changed to reserve space from the **GPU free memory** proportionally. The ratio `TurbomindEngineConfig.cache_max_entry_count` has been adjusted to 0.8 by default. If OOM

error happens, similar to the method mentioned above, please consider reducing the ratio value to decrease the memory usage of the k/v cache.

- **An example showing how to set tensor parallel num**:

```
from lmdeploy import pipeline, TurbomindEngineConfig

backend_config = TurbomindEngineConfig(tp=2)
pipe = pipeline('internlm/internlm2-chat-7b',
                backend_config=backend_config)
response = pipe(['Hi, pls intro yourself', 'Shanghai is'])
print(response)
```

- **An example for setting sampling parameters:**

```
from lmdeploy import pipeline, GenerationConfig, TurbomindEngineConfig

backend_config = TurbomindEngineConfig(tp=2)
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
pipe = pipeline('internlm/internlm2-chat-7b',
                backend_config=backend_config)
response = pipe(['Hi, pls intro yourself', 'Shanghai is'],
                gen_config=gen_config)
print(response)
```

- **An example for OpenAI format prompt input:**

```
from lmdeploy import pipeline, GenerationConfig, TurbomindEngineConfig

backend_config = TurbomindEngineConfig(tp=2)
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
pipe = pipeline('internlm/internlm2-chat-7b',
                backend_config=backend_config)
prompts = [[{
    'role': 'user',
    'content': 'Hi, pls intro yourself'
}], [{
    'role': 'user',
    'content': 'Shanghai is'
}]]
response = pipe(prompts,
                gen_config=gen_config)
print(response)
```

- **An example for streaming mode:**

```
from lmdeploy import pipeline, GenerationConfig, TurbomindEngineConfig

backend_config = TurbomindEngineConfig(tp=2)
```

(continues on next page)

(continued from previous page)

```
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
pipe = pipeline('internlm/internlm2-chat-7b',
                backend_config=backend_config)
prompts = [[{
    'role': 'user',
    'content': 'Hi, pls intro yourself'
}], [{
    'role': 'user',
    'content': 'Shanghai is'
}]]
for item in pipe.stream_infer(prompts, gen_config=gen_config):
    print(item)
```

- **Below is an example for pytorch backend. Please install triton first.**

```
pip install triton>=2.1.0
```

```
from lmdeploy import pipeline, GenerationConfig, PytorchEngineConfig

backend_config = PytorchEngineConfig(session_len=2048)
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
pipe = pipeline('internlm/internlm-chat-7b',
                backend_config=backend_config)
prompts = [[{
    'role': 'user',
    'content': 'Hi, pls intro yourself'
}], [{
    'role': 'user',
    'content': 'Shanghai is'
}]]
response = pipe(prompts, gen_config=gen_config)
print(response)
```

- **An example for slora.**

```
from lmdeploy import pipeline, GenerationConfig, PytorchEngineConfig

backend_config = PytorchEngineConfig(session_len=2048,
                                     adapters=dict(lora_name_1='chenchi/lora-chatglm2-6b-
↪guodegang'))
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
pipe = pipeline('THUDM/chatglm2-6b',
                backend_config=backend_config)
```

(continues on next page)

(continued from previous page)

```
prompts = [[{
    'role': 'user',
    'content': ''
}]]
response = pipe(prompts, gen_config=gen_config, adapter_name='lora_name_1')
print(response)
```

9.2 FAQs

- **RuntimeError: An attempt has been made to start a new process before the current process has finished its bootstrapping phase**.

  If you got this for tp>1 in pytorch backend. Please make sure the python script has following

  ```
  if __name__ == '__main__':
  ```

  Generally, in the context of multi-threading or multi-processing, it might be necessary to ensure that initialization code is executed only once. In this case, `if __name__ == '__main__':` can help to ensure that these initialization codes are run only in the main program, and not repeated in each newly created process or thread.

- To customize a chat template, please refer to *chat_template.md*.

- If the weight of lora has a corresponding chat template, you can first register the chat template to lmdeploy, and then use the chat template name as the adapter name.

CHAPTER

TEN

VLM OFFLINE INFERENCE PIPELINE

LMDeploy abstracts the complex inference process of multi-modal Vision-Language Models (VLM) into an easy-to-use pipeline, similar to the Large Language Model (LLM) inference *pipeline*.

Currently, it supports the following models.

- Qwen-VL-Chat
- LLaVA series: v1.5, v1.6
- Yi-VL

We genuinely invite the community to contribute new VLM support to LMDeploy. Your involvement is truly appreciated.

This article showcases the VLM pipeline using the liuhaotian/llava-v1.6-vicuna-7b model as a case study. You'll learn about the simplest ways to leverage the pipeline and how to gradually unlock more advanced features by adjusting engine parameters and generation arguments, such as tensor parallelism, context window sizing, random sampling, and chat template customization. Moreover, we will provide practical inference examples tailored to scenarios with multiple images, batch prompts etc.

10.1 A 'Hello, world' example

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('liuhaotian/llava-v1.6-vicuna-7b')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

If `ImportError` occurs while executing this case, please install the required dependency packages as prompted.

In the above example, the inference prompt is a tuple structure consisting of (prompt, image). Besides this structure, the pipeline also supports prompts in the OpenAI format:

```
from lmdeploy import pipeline

pipe = pipeline('liuhaotian/llava-v1.6-vicuna-7b')

prompts = [
```

(continues on next page)

(continued from previous page)

```
    {
        'role': 'user',
        'content': [
            {'type': 'text', 'text': 'describe this image'},
            {'type': 'image_url', 'image_url': {'url': 'https://raw.githubusercontent.
↪com/open-mmlab/mmdeploy/main/tests/data/tiger.jpeg'}}
        ]
    }
]
response = pipe(prompts)
print(response)
```

10.1.1 Set tensor parallelism

Tensor paramllelism can be activated by setting the engine parameter `tp`

```
from lmdeploy import pipeline, TurbomindEngineConfig
from lmdeploy.vl import load_image

pipe = pipeline('liuhaotian/llava-v1.6-vicuna-7b',
                backend_config=TurbomindEngineConfig(tp=2))

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

10.1.2 Set context window size

When creating the pipeline, you can customize the size of the context window by setting the engine parameter `session_len`.

```
from lmdeploy import pipeline, TurbomindEngineConfig
from lmdeploy.vl import load_image

pipe = pipeline('liuhaotian/llava-v1.6-vicuna-7b',
                backend_config=TurbomindEngineConfig(session_len=8192))

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

10.1.3 Set sampling parameters

You can change the default sampling parameters of pipeline by passing `GenerationConfig`

```
from lmdeploy import pipeline, GenerationConfig, TurbomindEngineConfig
from lmdeploy.vl import load_image

pipe = pipeline('liuhaotian/llava-v1.6-vicuna-7b',
                backend_config=TurbomindEngineConfig(tp=2, session_len=8192))
gen_config = GenerationConfig(top_k=40, top_p=0.8, temperature=0.6)
image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image), gen_config=gen_config)
print(response)
```

10.1.4 Set chat template

While performing inference, LMDeploy identifies an appropriate chat template from its builtin collection based on the model path and subsequently applies this template to the input prompts. However, when a chat template cannot be told from its model path, users have to specify it. For example, liuhaotian/llava-v1.5-7b employs the 'vicuna' chat template, but the name 'vicuna' cannot be ascertained from the model's path. We can specify it by setting 'vicuna' to `ChatTemplateConfig` as follows:

```
from lmdeploy import pipeline, ChatTemplateConfig
from lmdeploy.vl import load_image
pipe = pipeline('liuhaotian/llava-v1.5-7b',
                chat_template_config=ChatTemplateConfig(model_name='vicuna'))

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

For more information about customizing a chat template, please refer to *this* guide

10.2 Multi-images inference

When dealing with multiple images, you can put them all in one list. Keep in mind that multiple images will lead to a higher number of input tokens, and as a result, the size of the *context window* typically needs to be increased.

```
from lmdeploy import pipeline, TurbomindEngineConfig
from lmdeploy.vl import load_image

pipe = pipeline('liuhaotian/llava-v1.6-vicuna-7b',
                backend_config=TurbomindEngineConfig(session_len=8192))

image_urls=[
    'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/demo/resources/human-
↪pose.jpg',
    'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/demo/resources/det.jpg'
]
```

(continues on next page)

(continued from previous page)

```
images = [load_image(img_url) for img_url in image_urls]
response = pipe(('describe these images', images))
print(response)
```

10.3 Batch prompts inference

Conducting inference with batch prompts is quite straightforward; just place them within a list structure:

```
from lmdeploy import pipeline, TurbomindEngineConfig
from lmdeploy.vl import load_image

pipe = pipeline('liuhaotian/llava-v1.6-vicuna-7b',
                backend_config=TurbomindEngineConfig(session_len=8192))

image_urls=[
    "https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/demo/resources/human-
↪pose.jpg",
    "https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/demo/resources/det.jpg"
]
prompts = [('describe this image', load_image(img_url)) for img_url in image_urls]
response = pipe(prompts)
print(response)
```

10.4 Multi-turn conversation

There are two ways to do the multi-turn conversations with the pipeline. One is to construct messages according to the format of OpenAI and use above introduced method, the other is to use the `pipeline.chat` interface.

```
from lmdeploy import pipeline, TurbomindEngineConfig, GenerationConfig
from lmdeploy.vl import load_image

pipe = pipeline('liuhaotian/llava-v1.6-vicuna-7b',
                backend_config=TurbomindEngineConfig(session_len=8192))

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/demo/
↪resources/human-pose.jpg')
gen_config = GenerationConfig(top_k=40, top_p=0.8, temperature=0.8)
sess = pipe.chat(('describe this image', image), gen_config=gen_config)
print(sess.response.text)
sess = pipe.chat('What is the woman doing?', session=sess, gen_config=gen_config)
print(sess.response.text)
```

CHAPTER

ELEVEN

SERVING LLM WITH OPENAI COMPATIBLE SERVER

This article primarily discusses the deployment of a single LLM model across multiple GPUs on a single node, providing a service that is compatible with the OpenAI interface, as well as the usage of the service API. For the sake of convenience, we refer to this service as `api_server`. Regarding parallel services with multiple models, please refer to the guide about *Request Distribution Server*.

In the following sections, we will first introduce two methods for starting the service, choosing the appropriate one based on your application scenario.

Next, we focus on the definition of the service's RESTful API, explore the various ways to interact with the interface, and demonstrate how to try the service through the Swagger UI or LMDeploy CLI tools.

Finally, we showcase how to integrate the service into a WebUI, providing you with a reference to easily set up a demonstration demo.

11.1 Launch Service

Take the internlm2-chat-7b model hosted on huggingface hub as an example, you can choose one the following methods to start the service.

11.1.1 Option 1: Launching with lmdeploy CLI

```
lmdeploy serve api_server internlm/internlm2-chat-7b --server-port 23333
```

The arguments of `api_server` can be viewed through the command `lmdeploy serve api_server -h`, for instance, `--tp` to set tensor parallelism, `--session-len` to specify the max length of the context window, `--cache-max-entry-count` to adjust the GPU mem ratio for k/v cache etc.

11.1.2 Option 2: Deploying with docker

With LMDeploy official docker image, you can run OpenAI compatible server as follows:

```
docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
    -p 23333:23333 \
    --ipc=host \
    openmmlab/lmdeploy:latest \
    lmdeploy serve api_server internlm/internlm2-chat-7b
```

The parameters of `api_server` are the same with that mentioned in "*option 1*" section

11.1.3 Option 3: Deploying to Kubernetes cluster

Connect to a running Kubernetes cluster and deploy the internlm2-chat-7b model service with kubectl command-line tool (replace <your token> with your huggingface hub token):

```
sed 's/{{HUGGING_FACE_HUB_TOKEN}}/<your token>/' k8s/deployment.yaml | kubectl create -f␣
↪- \
    && kubectl create -f k8s/service.yaml
```

In the example above the model data is placed on the local disk of the node (hostPath). Consider replacing it with high-availability shared storage if multiple replicas are desired, and the storage can be mounted into container using PersistentVolume.

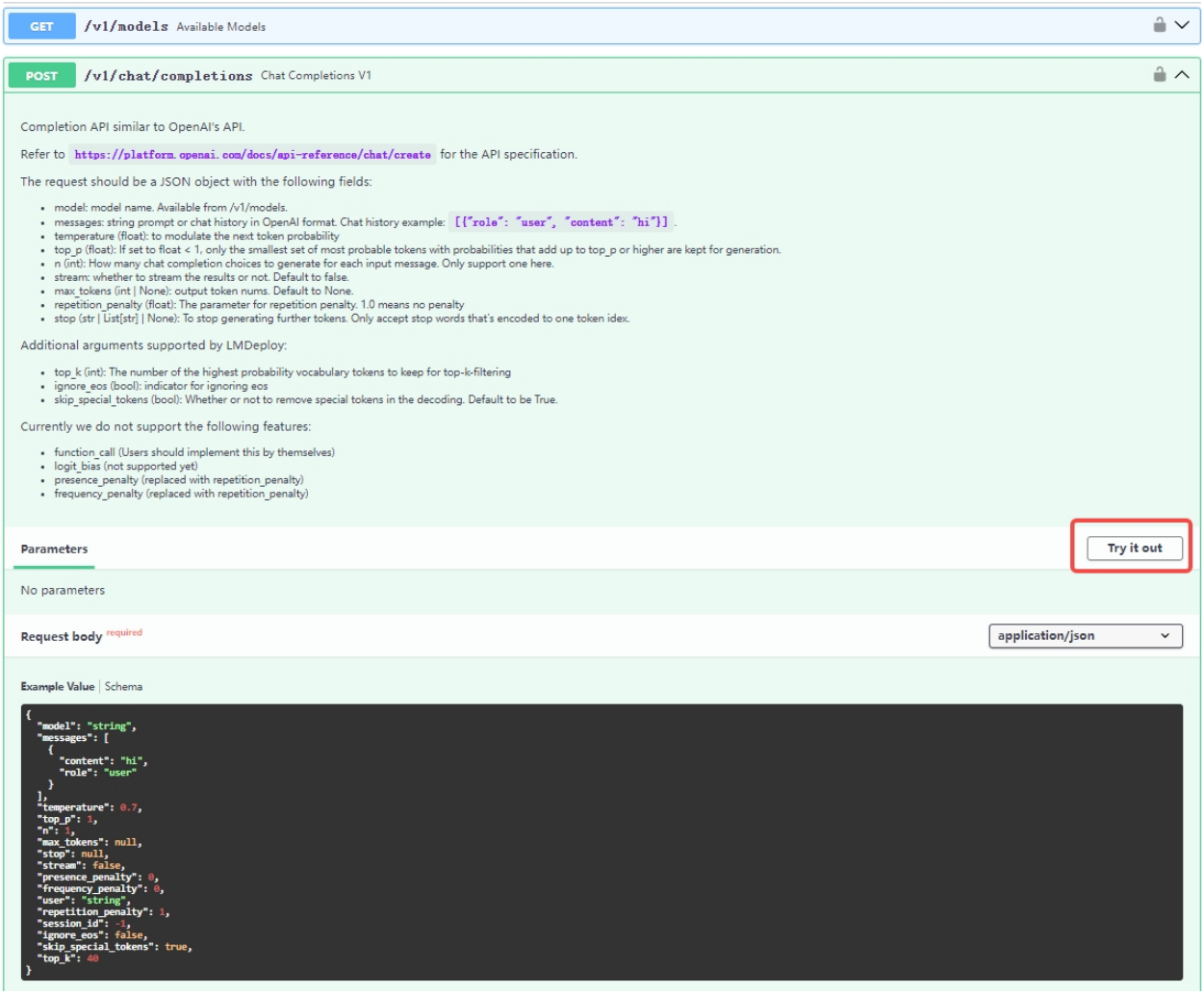
11.2 RESTful API

LMDeploy's RESTful API is compatible with the following three OpenAI interfaces:

- /v1/chat/completions
- /v1/models
- /v1/completions

Additionally, LMDeploy also defines `/v1/chat/interactive` to support interactive inference. The feature of interactive inference is that there's no need to pass the user conversation history as required by `v1/chat/completions`, since the conversation history will be cached on the server side. This method boasts excellent performance during multi-turn long context inference.

You can overview and try out the offered RESTful APIs by the website `http://0.0.0.0:23333` as shown in the below image after launching the service successfully.



Or, you can use the LMDeploy's built-in CLI tool to verify the service correctness right from the console.

```
# restful_api_url is what printed in api_server.py, e.g. http://localhost:23333
lmdeploy serve api_client ${api_server_url}
```

If you need to integrate the service into your own projects or products, we recommend the following approach:

11.2.1 Integrate with `OpenAI`

Here is an example of interaction with the endpoint `v1/chat/completions` service via the openai package. Before running it, please install the openai package by `pip install openai`

```
from openai import OpenAI
client = OpenAI(
    api_key='YOUR_API_KEY',
    base_url="http://0.0.0.0:23333/v1"
)
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
  model=model_name,
```

(continues on next page)

(continued from previous page)

```
  messages=[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": " provide three suggestions about time management"},
  ],
    temperature=0.8,
    top_p=0.8
)
print(response)
```

If you want to use async functions, may try the following example:

```
import asyncio
from openai import AsyncOpenAI

async def main():
    client = AsyncOpenAI(api_key='YOUR_API_KEY',
                         base_url='http://0.0.0.0:23333/v1')
    model_cards = await client.models.list()._get_page()
    response = await client.chat.completions.create(
        model=model_cards.data[0].id,
        messages=[
            {
                'role': 'system',
                'content': 'You are a helpful assistant.'
            },
            {
                'role': 'user',
                'content': ' provide three suggestions about time management'
            },
        ],
        temperature=0.8,
        top_p=0.8)
    print(response)

asyncio.run(main())
```

You can invoke other OpenAI interfaces using similar methods. For more detailed information, please refer to the OpenAI API guide

11.2.2 Integrate with lmdeploy `APIClient`

Below are some examples demonstrating how to visit the service through `APIClient`

If you want to use the `/v1/chat/completions` endpoint, you can try the following code:

```
from lmdeploy.serve.openai.api_client import APIClient
api_client = APIClient('http://{server_ip}:{server_port}')
model_name = api_client.available_models[0]
messages = [{"role": "user", "content": "Say this is a test!"}]
for item in api_client.chat_completions_v1(model=model_name, messages=messages):
    print(item)
```

For the `/v1/completions` endpoint, you can try:

```
from lmdeploy.serve.openai.api_client import APIClient
api_client = APIClient('http://{server_ip}:{server_port}')
model_name = api_client.available_models[0]
for item in api_client.completions_v1(model=model_name, prompt='hi'):
    print(item)
```

As for `/v1/chat/interactive`we disable the feature by default. Please open it by setting `interactive_mode = True`. If you don't, it falls back to openai compatible interfaces.

Keep in mind that `session_id` indicates an identical sequence and all requests belonging to the same sequence must share the same `session_id`. For instance, in a sequence with 10 rounds of chatting requests, the `session_id` in each request should be the same.

```
from lmdeploy.serve.openai.api_client import APIClient
api_client = APIClient(f'http://{server_ip}:{server_port}')
messages = [
    "hi, what's your name?",
    "who developed you?",
    "Tell me more about your developers",
    "Summarize the information we've talked so far"
]
for message in messages:
    for item in api_client.chat_interactive_v1(prompt=message,
                                               session_id=1,
                                               interactive_mode=True,
                                               stream=False):
        print(item)
```

11.2.3 Integrate with Java/Golang/Rust

May use openapi-generator-cli to convert `http://{server_ip}:{server_port}/openapi.json` to java/rust/golang client. Here is an example:

```
$ docker run -it --rm -v ${PWD}:/local openapitools/openapi-generator-cli generate -i /
↪local/openapi.json -g rust -o /local/rust

$ ls rust/*
rust/Cargo.toml  rust/git_push.sh  rust/README.md

rust/docs:
ChatCompletionRequest.md  EmbeddingsRequest.md  HttpValidationError.md  LocationInner.md␣
↪ Prompt.md
DefaultApi.md             GenerateRequest.md    Input.md                Messages.md     ␣
↪ ValidationError.md

rust/src:
apis  lib.rs  models
```

11.2.4 Integrate with cURL

cURL is a tool for observing the output of the RESTful APIs.

- list served models `v1/models`

```
curl http://{server_ip}:{server_port}/v1/models
```

- chat `v1/chat/completions`

```
curl http://{server_ip}:{server_port}/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "internlm-chat-7b",
    "messages": [{"role": "user", "content": "Hello! How are you?"}]
  }'
```

- text completions `v1/completions`

```
curl http://{server_ip}:{server_port}/v1/completions \
  -H 'Content-Type: application/json' \
  -d '{
  "model": "llama",
  "prompt": "two steps to build a house:"
}'
```

- interactive chat `v1/chat/interactive`

```
curl http://{server_ip}:{server_port}/v1/chat/interactive \
  -H "Content-Type: application/json" \
  -d '{
    "prompt": "Hello! How are you?",
    "session_id": 1,
    "interactive_mode": true
  }'
```

11.3 Integrate with WebUI

LMDeploy utilizes `gradio` or OpenAOE to integrate a web ui for `api_server`

11.3.1 Option 1: gradio

```
# api_server_url is what printed in api_server.py, e.g. http://localhost:23333
# server_ip and server_port here are for gradio ui
# example: lmdeploy serve gradio http://localhost:23333 --server-name localhost --server-
↪port 6006
lmdeploy serve gradio api_server_url --server-name ${gradio_ui_ip} --server-port $
↪{gradio_ui_port}
```

11.3.2 Option 2: OpenAOE

```
pip install -U openaoe
openaoe -f /path/to/your/config-template.yaml
```

Please refer to the guidance for more deploy information.

11.4 FAQ

1. When user got `"finish_reason":"length"`, it means the session is too long to be continued. The session length can be modified by passing `--session_len` to api_server.
2. When OOM appeared at the server side, please reduce the `cache_max_entry_count` of `backend_config` when lanching the service.
3. When the request with the same `session_id` to `/v1/chat/interactive` got a empty return value and a negative `tokens`, please consider setting `interactive_mode=false` to restart the session.
4. The `/v1/chat/interactive` api disables engaging in multiple rounds of conversation by default. The input argument `prompt` consists of either single strings or entire chat histories.
5. Regarding the stop words, we only support characters that encode into a single index. Furthermore, there may be multiple indexes that decode into results containing the stop word. In such cases, if the number of these indexes is too large, we will only use the index encoded by the tokenizer. If you want use a stop symbol that encodes into multiple indexes, you may consider performing string matching on the streaming client side. Once a successful match is found, you can then break out of the streaming loop.
6. To customize a chat template, please refer to *chat_template.md*.

CHAPTER

TWELVE

SERVING VLM WITH OPENAI COMPATIBLE SERVER

This article primarily discusses the deployment of a single large vision language model across multiple GPUs on a single node, providing a service that is compatible with the OpenAI interface, as well as the usage of the service API. For the sake of convenience, we refer to this service as `api_server`. Regarding parallel services with multiple models, please refer to the guide about *Request Distribution Server*.

In the following sections, we will first introduce two methods for starting the service, choosing the appropriate one based on your application scenario.

Next, we focus on the definition of the service's RESTful API, explore the various ways to interact with the interface, and demonstrate how to try the service through the Swagger UI or LMDeploy CLI tools.

Finally, we showcase how to integrate the service into a WebUI, providing you with a reference to easily set up a demonstration demo.

12.1 Launch Service

Take the llava-v1.6-vicuna-7b model hosted on huggingface hub as an example, you can choose one the following methods to start the service.

12.1.1 Option 1: Launching with lmdeploy CLI

```
lmdeploy serve api_server liuhaotian/llava-v1.6-vicuna-7b --server-port 23333
```

The arguments of `api_server` can be viewed through the command `lmdeploy serve api_server -h`, for instance, `--tp` to set tensor parallelism, `--session-len` to specify the max length of the context window, `--cache-max-entry-count` to adjust the GPU mem ratio for k/v cache etc.

12.1.2 Option 2: Deploying with docker

With LMDeploy official docker image, you can run OpenAI compatible server as follows:

```
docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
    -p 23333:23333 \
    --ipc=host \
    openmmlab/lmdeploy:latest \
    lmdeploy serve api_server liuhaotian/llava-v1.6-vicuna-7b
```

The parameters of `api_server` are the same with that mentioned in "*option 1*" section

Each model may require specific dependencies not included in the Docker image. If you run into issues, you may need to install those yourself on a case-by-case basis. If in doubt, refer to the specific model's project for documentation.

For example, for Llava:

```
FROM openmmlab/lmdeploy:latest

RUN apt-get update && apt-get install -y python3 python3-pip git

WORKDIR /app

RUN pip3 install --upgrade pip
RUN pip3 install timm
RUN pip3 install git+https://github.com/haotian-liu/LLaVA.git --no-deps

COPY . .

CMD ["lmdeploy", "serve", "api_server", "liuhaotian/llava-v1.6-34b"]
```

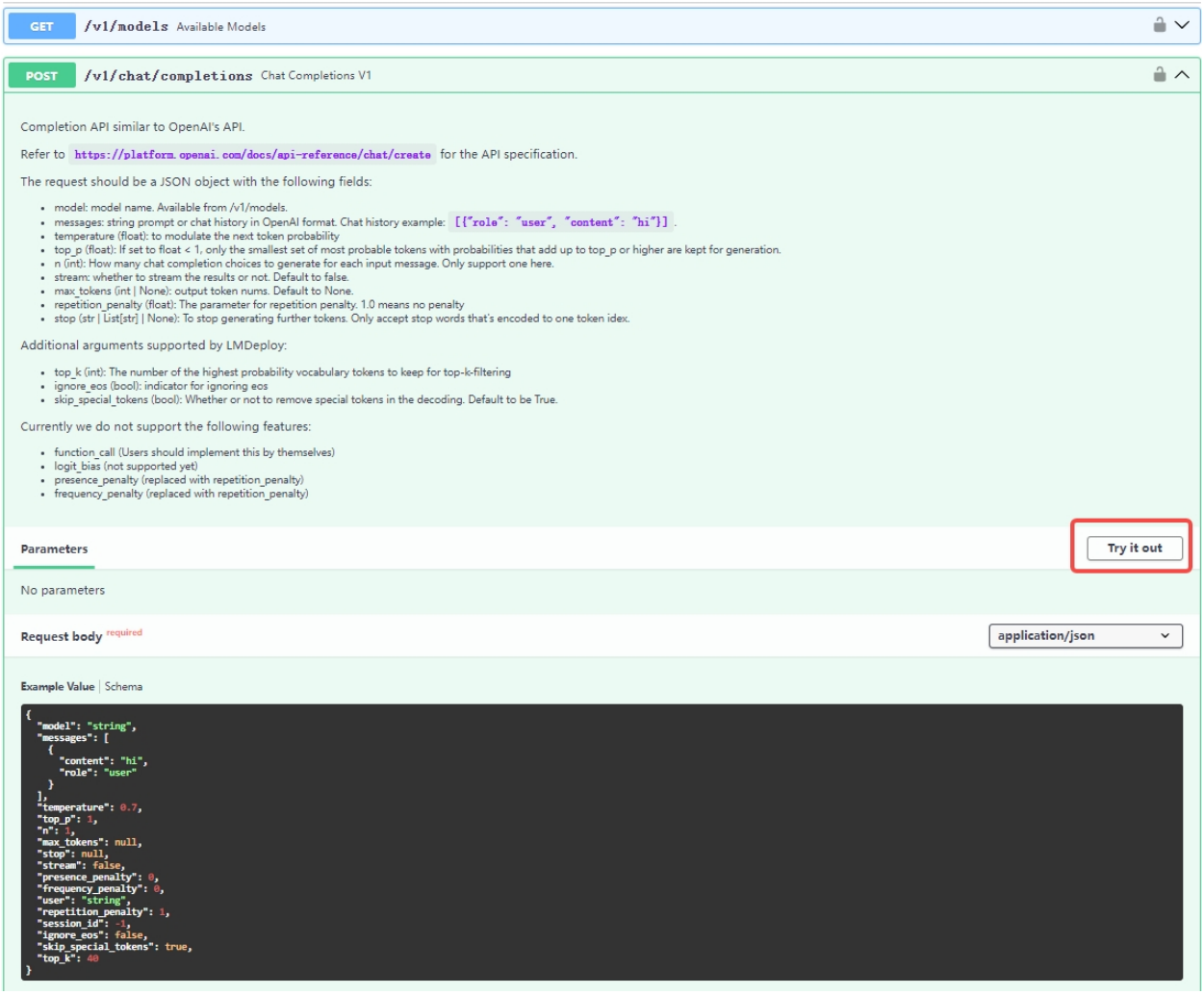
12.2 RESTful API

LMDeploy's RESTful API is compatible with the following three OpenAI interfaces:

- /v1/chat/completions
- /v1/models
- /v1/completions

The interface for image interaction is `/v1/chat/completions`, which is consistent with OpenAI.

You can overview and try out the offered RESTful APIs by the website `http://0.0.0.0:23333` as shown in the below image after launching the service successfully.



If you need to integrate the service into your own projects or products, we recommend the following approach:

12.2.1 Integrate with `OpenAI`

Here is an example of interaction with the endpoint `v1/chat/completions` service via the openai package. Before running it, please install the openai package by `pip install openai`

```
from openai import OpenAI

client = OpenAI(api_key='YOUR_API_KEY', base_url='http://0.0.0.0:23333/v1')
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role':
        'user',
        'content': [{
            'type': 'text',
            'text': 'Describe the image please',
        }, {
```

(continues on next page)

(continued from previous page)

```
            'type': 'image_url',
            'image_url': {
                'url':
                'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/data/
↪tiger.jpeg',
            },
        }],
    }],
    temperature=0.8,
    top_p=0.8)
print(response)
```

12.2.2 Integrate with lmdeploy `APIClient`

Below are some examples demonstrating how to visit the service through `APIClient`

If you want to use the `/v1/chat/completions` endpoint, you can try the following code:

```
from lmdeploy.serve.openai.api_client import APIClient

api_client = APIClient(f'http://0.0.0.0:23333')
model_name = api_client.available_models[0]
messages = [{
    'role':
    'user',
    'content': [{
        'type': 'text',
        'text': 'Describe the image please',
    }, {
        'type': 'image_url',
        'image_url': {
            'url':
            'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/data/tiger.
↪jpeg',
        },
    }]
}]
for item in api_client.chat_completions_v1(model=model_name,
                                           messages=messages):
    print(item)
```

12.2.3 Integrate with Java/Golang/Rust

May use openapi-generator-cli to convert `http://{server_ip}:{server_port}/openapi.json` to java/rust/golang client. Here is an example:

```
$ docker run -it --rm -v ${PWD}:/local openapitools/openapi-generator-cli generate -i /
↪local/openapi.json -g rust -o /local/rust

$ ls rust/*
```

(continues on next page)

(continued from previous page)

```
rust/Cargo.toml  rust/git_push.sh  rust/README.md

rust/docs:
ChatCompletionRequest.md  EmbeddingsRequest.md  HttpValidationError.md  LocationInner.md␣
↪ Prompt.md
DefaultApi.md             GenerateRequest.md    Input.md                Messages.md     ␣
↪ ValidationError.md

rust/src:
apis  lib.rs  models
```

CHAPTER

THIRTEEN

SERVING WITH GRADIO

Starting an LLM model's gradio service with LMDeploy and interacting with the model on the WebUI is incredibly simple.

```
pip install lmdeploy[serve]
lmdeploy serve gradio {model_path}
```

All it takes is one-line command, with the `{model_path}` replaced by the model ID from huggingface hub, such as `internlm/internlm2-chat-7b`, or the local path to the model.

For detailed parameters of the command, please turn to `lmdeploy serve gradio -h` for help.

13.1 Create a huggingface demo

If you want to create an online demo project for your model on huggingface, please follow the steps below.

13.1.1 Step 1: Create space

First, register for a Hugging Face account. After successful registration, click on your profile picture in the upper right corner and select "New Space" to create one. Follow the Hugging Face guide to choose the necessary configurations, and you will have a blank demo space ready.

13.1.2 Step 2: Develop demo's entrypoint `app.py`

Replace the content of `app.py` in your space with the following code:

```
from lmdeploy.serve.gradio.turbomind_coupled import run_local
from lmdeploy.messages import TurbomindEngineConfig

backend_config = TurbomindEngineConfig(max_batch_size=8)
model_path = 'internlm/internlm2-chat-7b'
run_local(model_path, backend_config=backend_config, server_name="huggingface-space")
```

Create a `requirements.txt` file with the following content:

```
lmdeploy
```

13.2 FAQs

- ZeroGPU compatibility issue. ZeroGPU is not suitable for LMDeploy turbomind engine. Please use the standard GPUs. Or, you can change the backend config in the above code to `PyTorchEngineConfig` to use the ZeroGPU.
- Gradio version issue, versions above 4.0.0 are currently not supported. You can modify this in `app.py`, for example:

```
import os
os.system("pip uninstall -y gradio")
os.system("pip install gradio==3.43.0")
```

CHAPTER

FOURTEEN

REQUEST DISTRIBUTOR SERVER

The request distributor service can parallelize multiple api_server services. Users only need to access the proxy URL, and they can indirectly access different api_server services. The proxy service will automatically distribute requests internally, achieving load balancing.

14.1 Startup

Start the proxy service:

```
python3 -m lmdeploy.serve.proxy.proxy --server_name {server_name} --server_port {server_
↪port} --strategy "min_expected_latency"
```

After startup is successful, the URL of the proxy service will also be printed by the script. Access this URL in your browser to open the Swagger UI.

14.2 API

Through Swagger UI, we can see multiple APIs. Those related to api_server node management include:

- /nodes/status
- /nodes/add
- /nodes/remove

They respectively represent viewing all api_server service nodes, adding a certain node, and deleting a certain node.

APIs related to usage include:

- /v1/models
- /v1/chat/completions
- /v1/completions

The usage of these APIs is the same as that of api_server.

14.3 Dispatch Strategy

The current distribution strategies of the proxy service are as follows:

- random dispatches based on the ability of each api_server node provided by the user to process requests. The greater the request throughput, the more likely it is to be allocated. Nodes that do not provide throughput are treated according to the average throughput of other nodes.
- min_expected_latency allocates based on the number of requests currently waiting to be processed on each node, and the throughput capability of each node, calculating the expected time required to complete the response. The shortest one gets allocated. Nodes that do not provide throughput are treated similarly.
- min_observed_latency allocates based on the average time required to handle a certain number of past requests on each node. The one with the shortest time gets allocated.

CHAPTER

FIFTEEN

W4A16 QUANTIZATION

LMDeploy adopts AWQ algorithm for 4bit weight-only quantization. By developed the high-performance cuda kernel, the 4bit quantized model inference achieves up to 2.4x faster than FP16.

LMDeploy supports the following NVIDIA GPU for W4A16 inference:

- Turing(sm75): 20 series, T4
- Ampere(sm80,sm86): 30 series, A10, A16, A30, A100
- Ada Lovelace(sm90): 40 series

Before proceeding with the quantization and inference, please ensure that lmdeploy is installed.

```
pip install lmdeploy[all]
```

This article comprises the following sections:

- *Quantization*
- *Evaluation*
- *Inference*
- *Service*
- *Performance*

15.1 Quantization

A single command execution is all it takes to quantize the model. The resulting quantized weights are then stored in the $WORK_DIR directory.

```
export HF_MODEL=internlm/internlm2-chat-7b
export WORK_DIR=internlm/internlm2-chat-7b-4bit

lmdeploy lite auto_awq \
   $HF_MODEL \
  --calib-dataset 'ptb' \
  --calib-samples 128 \
  --calib-seqlen 2048 \
  --w-bits 4 \
  --w-group-size 128 \
  --batch-size 1 \
```

(continues on next page)

(continued from previous page)

```
  --search-scale False \
  --work-dir $WORK_DIR
```

Typically, the above command doesn't require filling in optional parameters, as the defaults usually suffice. For instance, when quantizing the internlm/internlm2-chat-7b model, the command can be condensed as:

```
lmdeploy lite auto_awq internlm/internlm2-chat-7b --work-dir internlm2-chat-7b-4bit
```

Note:

- We recommend that you specify the –work-dir parameter, including the model name as demonstrated in the example above. This facilitates LMDeploy in fuzzy matching the –work-dir with an appropriate built-in chat template. Otherwise, you will have to designate the chat template during inference.
- If the quantized model's accuracy is compromised, it is recommended to enable –search-scale for re-quantization and increase the –batch-size, for example, to 8. When search_scale is enabled, the quantization process will take more time. The –batch-size affects the amount of memory used, which can be adjusted according to actual conditions as needed.

Upon completing quantization, you can engage with the model efficiently using a variety of handy tools.

For example, you can initiate a conversation with it via the command line:

```
lmdeploy chat ./internlm2-chat-7b-4bit --model-format awq
```

Alternatively, you can start the gradio server and interact with the model through the web at `http://{ip_addr}:{port`

```
lmdeploy serve gradio ./internlm2-chat-7b-4bit --server_name {ip_addr} --server_port
↪{port} --model-format awq
```

15.2 Evaluation

Please overview this guide about model evaluation with LMDeploy.

15.3 Inference

Trying the following codes, you can perform the batched offline inference with the quantized model:

```
from lmdeploy import pipeline, TurbomindEngineConfig
engine_config = TurbomindEngineConfig(model_format='awq')
pipe = pipeline("./internlm2-chat-7b-4bit", backend_config=engine_config)
response = pipe(["Hi, pls intro yourself", "Shanghai is"])
print(response)
```

For more information about the pipeline parameters, please refer to *here*.

In addition to performing inference with the quantized model on localhost, LMDeploy can also execute inference for the 4bit quantized model derived from AWQ algorithm available on Huggingface Hub, such as models from the lmdeploy space and TheBloke space

```
# inference with models from lmdeploy space
from lmdeploy import pipeline, TurbomindEngineConfig
pipe = pipeline("lmdeploy/llama2-chat-70b-4bit",
                backend_config=TurbomindEngineConfig(model_format='awq', tp=4))
response = pipe(["Hi, pls intro yourself", "Shanghai is"])
print(response)

# inference with models from thebloke space
from lmdeploy import pipeline, TurbomindEngineConfig, ChatTemplateConfig
pipe = pipeline("TheBloke/LLaMA2-13B-Tiefighter-AWQ",
                backend_config=TurbomindEngineConfig(model_format='awq'),
                chat_template_config=ChatTemplateConfig(model_name='llama2')
                )
response = pipe(["Hi, pls intro yourself", "Shanghai is"])
print(response)
```

15.4 Service

LMDeploy's `api_server` enables models to be easily packed into services with a single command. The provided RESTful APIs are compatible with OpenAI's interfaces. Below are an example of service startup:

```
lmdeploy serve api_server ./internlm2-chat-7b-4bit --backend turbomind --model-format awq
```

The default port of `api_server` is `23333`. After the server is launched, you can communicate with server on terminal through `api_client`:

```
lmdeploy serve api_client http://0.0.0.0:23333
```

You can overview and try out `api_server` APIs online by swagger UI at `http://0.0.0.0:23333`, or you can also read the API specification from *here*.

15.5 Performance

We benchmarked the Llama-2-7B-chat and Llama-2-13B-chat models with 4-bit quantization on NVIDIA GeForce RTX 4090 using profile_generation.py. And we measure the token generation throughput (tokens/s) by setting a single prompt token and generating 512 tokens. All the results are measured for single batch inference.

CHAPTER

SIXTEEN

KEY-VALUE(KV) CACHE QUANTIZATION

Since v0.4.0, LMDeploy has supported **online** key-value (kv) cache quantization with int4 and int8 numerical precision, utilizing an asymmetric quantization method that is applied on a per-head, per-token basis. The original kv offline quantization method has been removed.

Intuitively, quantizing the kv cache is beneficial for reducing memory usage. Compared to FP16, the memory for int4/int8 kv can be reduced to 1/4 and 1/2, respectively. This means that under the same memory conditions, the system can support a significantly increased number of concurrent operations after kv quantization, thereby ultimately enhancing throughput.

However, quantization typically brings in some loss of model accuracy. We have used OpenCompass to evaluate the accuracy of several models after applying int4/int8 quantization. int8 kv keeps the accuracy while int4 kv has slight loss. The detailed results are presented in the Evaluation section. You can refer to the information and choose wisely based on your requirements.

LMDeploy inference with quantized kv supports the following NVIDIA GPU models:

- Volta architecture (sm70): V100
- Turing architecture (sm75): 20 series, T4
- Ampere architecture (sm80, sm86): 30 series, A10, A16, A30, A100
- Ada Lovelace architecture (sm89): 40 series
- Hopper architecture (sm90): H100, H200

In summary, LMDeploy kv quantization has the following advantages:

1. data-free online quantization
2. Supports all nvidia GPU models with Volta architecture (sm70) and above
3. KV int8 quantization has almost lossless accuracy, and KV int4 quantization accuracy is within an acceptable range
4. Efficient inference, with int8/int4 kv quantization applied to llama2-7b, RPS is improved by round 30% and 40% respectively compared to fp16

In the next section, we will take `internlm2-chat-7b` model as an example, introducing the usage of kv quantization and inference of lmdeploy. But before that, please ensure that lmdeploy is installed.

```
pip install lmdeploy
```

16.1 Usage

Applying kv quantization and inference via LMDeploy is quite straightforward. Simply set the `quant_policy` parameter.

LMDeploy specifies that `quant_policy=4` stands for 4-bit kv, whereas `quant_policy=8` indicates 8-bit kv.

16.1.1 Offline inference

```
from lmdeploy import pipeline, TurbomindEngineConfig
engine_config = TurbomindEngineConfig(quant_policy=8)
pipe = pipeline("internlm/internlm2-chat-7b", backend_config=engine_config)
response = pipe(["Hi, pls intro yourself", "Shanghai is"])
print(response)
```

16.1.2 Serving

```
lmdeploy serve api_server internlm/internlm2-chat-7b --quant-policy 8
```

16.2 Evaluation

We apply kv quantization of LMDeploy to several LLM models and utilize OpenCompass to evaluate the inference accuracy. The results are shown in the table below:

For detailed evaluation methods, please refer to *this* guide. Remember to pass `quant_policy` to the inference engine in the config file.

16.3 Performance

The performance data is obtained by `benchmark/profile_throughput.py`

CHAPTER

SEVENTEEN

W8A8 LLM MODEL DEPLOYMENT

LMDeploy provides functions for quantization and inference of large language models using 8-bit integers.

Before starting inference, ensure that lmdeploy and openai/triton are correctly installed. Execute the following commands to install these:

```
pip install lmdeploy
pip install triton>=2.1.0
```

17.1 8-bit Weight Model Inference

For performing 8-bit weight model inference, you can directly download the pre-quantized 8-bit weight models from LMDeploy's model zoo. For instance, the 8-bit Internlm-chat-7B model is available for direct download from the model zoo:

```
git-lfs install
git clone https://huggingface.co/lmdeploy/internlm-chat-7b-w8 (coming soon)
```

Alternatively, you can manually convert original 16-bit weights into 8-bit by referring to the content under the *"8bit Weight Quantization"* section. Save them in the internlm-chat-7b-w8 directory, using the command below:

```
lmdeploy lite smooth_quant internlm/internlm-chat-7b --work-dir ./internlm-chat-7b-w8
```

Afterwards, use the following command to interact with the model via the terminal:

```
lmdeploy chat ./internlm-chat-7b-w8 --backend pytorch
```

17.2 Launching gradio service

Coming soon...

17.3 Inference Speed

Coming soon...

17.4 8bit Weight Quantization

Performing 8bit weight quantization involves three steps:

1. **Smooth Weights**: Start by smoothing the weights of the Language Model (LLM). This process makes the weights more amenable to quantizing.
2. **Replace Modules**: Locate DecoderLayers and replace the modules RSMNorm and nn.Linear with QRSMNorm and QLinear modules respectively. These 'Q' modules are available in the lmdeploy/pytorch/models/q_modules.py file.
3. **Save the Quantized Model**: Once you've made the necessary replacements, save the new quantized model.

The script `lmdeploy/lite/apis/smooth_quant.py` accomplishes all three tasks detailed above. For example, you can obtain the model weights of the quantized Internlm-chat-7B model by running the following command:

```
lmdeploy lite smooth_quant internlm/internlm-chat-7b --work-dir ./internlm-chat-7b-w8
```

After saving, you can instantiate your quantized model by calling the from_pretrained interface.

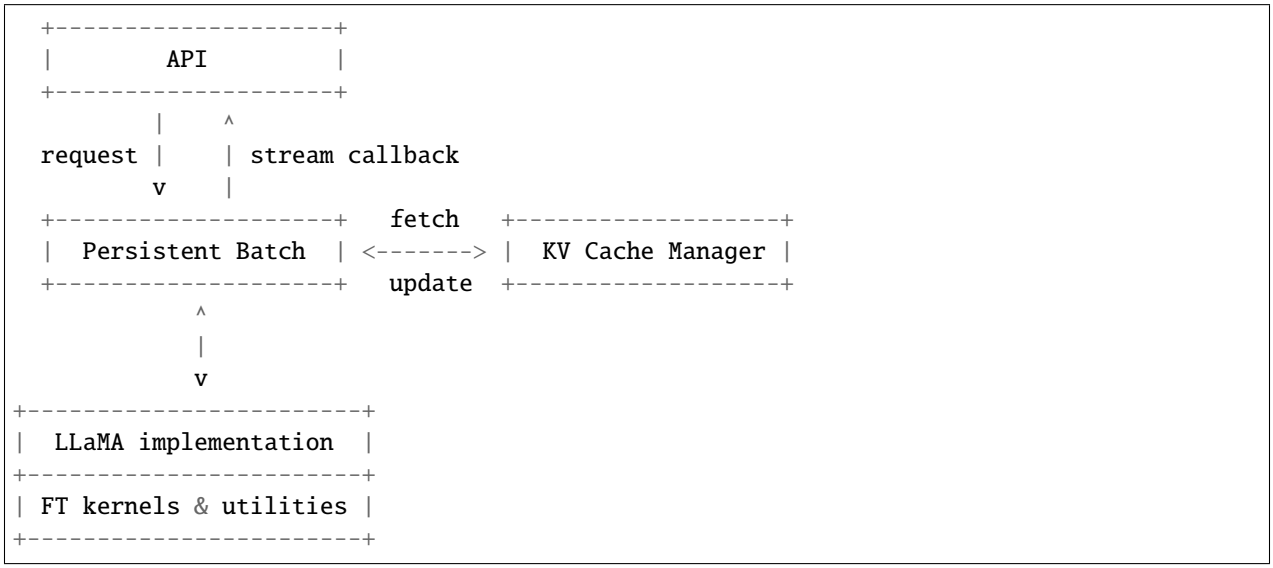
CHAPTER

EIGHTEEN

ARCHITECTURE OF TURBOMIND

TurboMind is an inference engine that supports high throughput inference for conversational LLMs. It's based on NVIDIA's FasterTransformer. Major features of TurboMind include an efficient LLaMa implementation, the persistent batch inference model and an extendable KV cache manager.

18.1 High level overview of TurboMind



18.2 Persistent Batch

You may recognize this feature as "continuous batching" in other repos. But during the concurrent development of the feature, we modeled the inference of a conversational LLM as a persistently running batch whose lifetime spans the entire serving process, hence the name "persistent batch". To put it simply

- The persistent batch as N pre-configured batch slots.
- Requests join the batch when there are free slots available. A batch slot is released and can be reused once the generation of the requested tokens is finished.
- **On cache-hits (see below), history tokens don't need to be decoded in every round of a conversation; generation of response tokens will start instantly.**
- The batch grows or shrinks automatically to minimize unnecessary computations.

18.3 KV Cache Manager

The KV cache manager of TurboMind is a memory-pool-liked object that also implements LRU policy so that it can be viewed as a form of **cache of KV caches**. It works in the following way

- All device memory required for KV cache is allocated by the manager. A fixed number of slots is pre-configured to match the memory size of the system. Each slot corresponds to the memory required by the KV cache of a single sequence. Allocation chunk-size can be configure to implement pre-allocate/on-demand style allocation policy (or something in-between).
- When space for the KV cache of a new sequence is requested but no free slots left in the pool, the least recently used sequence is evicted from the cache and its device memory is directly reused by the new sequence. However, this is not the end of the story.
- Fetching sequence currently resides in one of the slots resembles a *cache-hit*, the history KV cache is returned directly and no context decoding is needed.
- Victim (evicted) sequences are not erased entirely but converted to its most compact form, i.e. token IDs. When the same sequence id is fetched later (*cache-miss*) the token IDs will be decoded by FMHA backed context decoder and converted back to KV cache.
- The eviction and conversion are handled automatically inside TurboMind and thus transparent to the users. **From the user's aspect, system that use TurboMind has access to infinite device memory.**

18.4 LLaMa implementation

Our implementation of the LLaMa family models is modified from Gpt-NeoX model in FasterTransformer. In addition to basic refactoring and modifications to support the LLaMa family, we made some improvements to enable high performance inference of conversational models, most importantly:

- To support fast context decoding in multi-round conversations. We replaced the attention implementation in context decoder with a cutlass-based FMHA implementation that supports mismatched Q/K lengths.
- We introduced indirect buffer pointers in both context FMHA and generation FMHA to support the discontinuity in KV cache within the batch.
- To support concurrent inference with persistent batch, new synchronization mechanism was designed to orchestrate the worker threads running in tensor parallel mode.
- To maximize the throughput, we implement INT8 KV cache support to increase the max batch size. It's effective because in real-world serving scenarios, KV cache costs more memory and consumes more memory bandwidth than weights or other activations.
- We resolved an NCCL hang issue when running multiple model instances in TP mode within a single process, NCCL APIs are now guarded by host-side synchronization barriers.

18.5 API

TurboMind supports a Python API that enables streaming output and tensor parallel mode.

The ability to use tritonserver for serving is also inherited from FasterTransformer. However, to support submitting concurrent requests into our persistent batch model, we no longer use sequence batching or dynamic batching as FasterTransformer does. The bookkeeping of request and sequence states are managed by TurboMind instead.

18.6 Difference between FasterTransformer and TurboMind

Apart of the features described above, there are still many minor differences that we don't cover in this document. Notably, many capabilities of FT are dropped in TurboMind because of the difference in objectives (e.g. prefix prompt, beam search, context embedding, sparse GEMM, GPT/T5/other model families, etc)

18.7 FAQ

18.7.1 Supporting Huggingface models

For historical reasons, TurboMind's weight layout is based on the original LLaMa implementation (differ only by a transpose). The implementation in huggingface transformers uses a different layout for `W_q` and `W_k` which is handled in deploy.py.

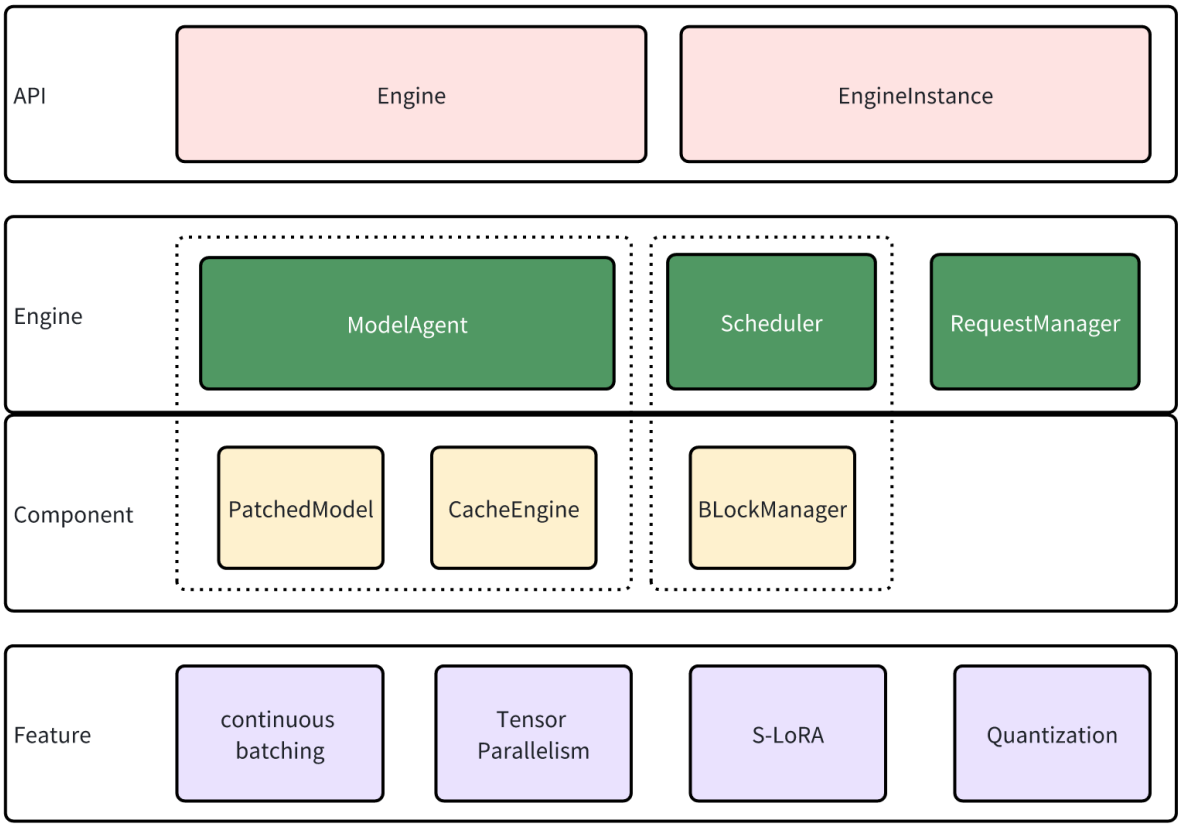
CHAPTER

NINETEEN

ARCHITECTURE OF LMDEPLOY.PYTORCH

`lmdeploy.pytorch` is an inference engine in LMDeploy that offers a developer-friendly framework to users interested in deploying their own models and developing new features.

19.1 Design



19.2 API

`lmdeploy.pytorch` shares service interfaces with `Turbomind`, and the inference service is implemented by `Engine` and `EngineInstance`.

`EngineInstance` acts as the sender of inference requests, encapsulating and sending requests to the `Engine` to achieve streaming inference. The inference interface of `EngineInstance` is thread-safe, allowing instances in different threads to initiate requests simultaneously. The `Engine` will automatically perform batch processing based on the current system resources.

Engine is the request receiver and executor. It contain modules:

- `ModelAgent` serves as a wrapper for the model, handling tasks such as loading model/adapters, managing the cache, and implementing tensor parallelism.
- The `Scheduler` functions as the sequence manager, determining the sequences and adapters to participate in the current step, and subsequently allocating resources for them.
- `RequestManager` is tasked with sending and receiving requests. acting as the bridge between the `Engine` and `EngineInstance`.

19.3 Engine

The Engine responses to requests in a sub-thread, following this looping sequence:

1. Get new requests through `RequestManager`. These requests are cached for now.
2. The `Scheduler` performs scheduling, deciding which cached requests should be processed and allocating resources for them.
3. `ModelAgent` swaps the caches according to the information provided by the Scheduler, then performs inference with the patched model.
4. The `Scheduler` updates the status of requests based to the inference results from `ModelAgent`.
5. `RequestManager` responds to the sender (`EngineInstance`), and the process return to step 1.

Now, Let's delve deeper into the modules that participate in these steps.

19.3.1 Scheduler

In LLM inference, caching history key and value states is a common practice to prevent redundant computation. However, as history lengths vary in a batch of sequences, we need to pad the caches to enable batching inference. Unfortunately, this padding can lead to significant memory wastage, limiting the transformer's performance.

vLLM employs a paging-based strategy, allocating caches in page blocks to minimize extra memory usage. Our Scheduler module in the Engine shares a similar design, allocating resources based on sequence length in blocks and evicting unused blocks to support larger batching and longer session lengths.

Additionally, we support S-LoRA, which enables the use of multiple LoRA adapters on limited memory.

19.3.2 ModelAgent

`lmdeploy.pytorch` supports Tensor Parallelism, which leads to complex model initialization, cache allocation, and weight partitioning. ModelAgent is designed to abstract these complexities, allowing the Engine to focus solely on maintaining the pipeline.

ModelAgent consists of two components:

1. `**patched_model**: : This is the transformer model after patching. In comparison to the original model, the patched model incorporates additional features such as Tensor Parallelism, quantization, and high-performance kernels.
2. **cache_engine**: This component manages the caches. It receives commands from the Scheduler and performs host-device page swaps. Only GPU blocks are utilized for caching key/value pairs and adapters.

19.4 Patching

In order to facilitate the deployment of a new model, we have developed a tool to patch the modules.

For example, if we want to reimplement the forward method of `LlamaAttention`:

```
class CustomLlamaAttention(nn.Module):
    def forward(self, ...):
        # custom forward
```

We register the implementation above into `lmdeploy.pytorch.models.module_map`:

```
MODULE_MAP.update({
'transformers.models.llama.modeling_llama.LlamaAttention':
'qualname.to.CustomLlamaAttention'})
```

`ModelAgent` would then load and patch `LlamaAttention` with `CustomLlamaAttention` while leaving everything else unchanged. You can perform inference with the new implementation. For more detail about model patching, please refer to *support new model* .

19.5 Features

`lmdeploy.pytorch` supports new features including:

- **Continuous Batching**: As the sequence length in a batch may vary, padding is often necessary for batching inference. However, large padding can lead to additional memory usage and unnecessary computation. To address this, we employ continuous batching, where all sequences are concatenated into a single long sequence to avoid padding.
- **Tensor Parallelism**: The GPU memory usage of LLM might exceed the capacity of a single GPU. Tensor parallelism is utilized to accommodate such models on multiple devices. Each device handles parts of the model simultaneously, and the results are gathered to ensure correctness.
- **S-LoRA**: LoRA adapters can be used to train LLM on devices with limited memory. While it's common practice to merge adapters into the model weights before deployment, loading multiple adapters in this way can consume a significant amount of memory. We support S-LoRA, where adapters are paged and swapped in when necessary. Special kernels are developed to support inference with unmerged adapters, enabling the loading of various adapters efficiently.

- **Quantization**: Model quantization involves performing computations with low precision. `lmdeploy.pytorch` supports w8a8 quantization. For more details, refer to *w8a8*.

CHAPTER

TWENTY

HOW TO SUPPORT NEW MODEL IN LMDEPLOY.PYTORCH

lmdeploy.pytorch is designed to ease new model deployment and prototype verification. If you are willing to use our engine, here is the tutorial.

20.1 Support New Model

Let's begin with Llama.

Before delving into the details, it's essential to acquaint ourselves with the input specifications of the model. In order to accommodate new features within our engine, there are some deviations from the typical transformer inputs.

1. To circumvent the need for batch padding, continuous batching is employed. Consequently, the `input_ids` now represents the concatenation of all input sequences in the batch, followed by a `unsqueeze(0)` operation to align with the original `input_ids` dimension.
2. In an effort to optimize memory usage for the key/value cache, we implement paged attention. This transforms the `past_key_value` into a substantial tensor with dimensions `[num_blocks, block_size, num_heads, head_dim]`. Here, `num_blocks` denotes the number of page blocks, and `block_size` indicates the size of each block.
3. Accompanying these changes, additional inputs are imperative to support the modified inputs described above. These include the block table and history length. It's important to note that these supplementary inputs are not explicitly listed as arguments in the original forward method. Instead, a context object is utilized to furnish this essential information.

Due to the alterations in the input structure mentioned earlier, the forward methods for both `LlamaModel` and `LlamaAttention` modules need to be adjusted. Below are the modified implementations:

For `LlamaModel`:

```
# lmdeploy/pytorch/models/llama.py

class LlamaModel(nn.Module):
    def forward(
        self,
        input_ids: torch.LongTensor = None,
        attention_mask: Optional[torch.Tensor] = None,
        position_ids: Optional[torch.LongTensor] = None,
        past_key_values: Optional[List[torch.FloatTensor]] = None,
        inputs_embeds: Optional[torch.FloatTensor] = None,
        use_cache: Optional[bool] = None,
        output_attentions: Optional[bool] = None,
        output_hidden_states: Optional[bool] = None,
```

(continues on next page)

(continued from previous page)

```
        return_dict: Optional[bool] = None,
    ) -> Union[Tuple, BaseModelOutputWithPast]:
        """Rewrite implementation of LlamaModel.forward."""
        inputs_embeds = self.embed_tokens(input_ids)
        hidden_states = inputs_embeds

        # decoder layers
        for idx, decoder_layer in enumerate(self.layers):
            past_key_value = past_key_values[idx]
            layer_outputs = decoder_layer(
                hidden_states,
                attention_mask=attention_mask,
                position_ids=position_ids,
                past_key_value=past_key_value,
                output_attentions=output_attentions,
                use_cache=use_cache,
            )
            hidden_states = layer_outputs[0]
        hidden_states = self.norm(hidden_states)

        return BaseModelOutputWithPast(
            last_hidden_state=hidden_states,
            past_key_values=past_key_values,
            hidden_states=None,
            attentions=None,
        )
```

For LlamaAttention:

```
# lmdeploy/pytorch/models/llama.py
from lmdeploy.pytorch.kernels import apply_rotary_pos_emb, fill_kv_cache, paged_
↪attention_fwd

class LlamaAttention(nn.Module):
    def forward(
        self,
        hidden_states: torch.Tensor,
        attention_mask: Optional[torch.Tensor] = None,
        position_ids: Optional[torch.LongTensor] = None,
        past_key_value: Optional[Tuple[torch.Tensor]] = None,
        output_attentions: bool = False,
        use_cache: bool = False,
    ) -> Tuple[torch.Tensor, Optional[torch.Tensor],
               Optional[Tuple[torch.Tensor]]]:
        """Rewrite of LlamaAttention.forward."""
        context = self.context.context
        history_lengths = context.history_lengths
        position_ids_1d = context.position_ids_1d
        block_offsets = context.block_offsets

        # qkv proj
        query_states = q_proj(hidden_states)
```

(continues on next page)

(continued from previous page)

```
        key_states = k_proj(hidden_states)
        value_states = v_proj(hidden_states)
        query_states = query_states.view(-1, num_heads, head_dim)
        key_states = key_states.view(-1, num_kv_heads, head_dim)
        value_states = value_states.view(-1, num_kv_heads, head_dim)

        # rotary embedding
        max_seq_len = position_ids.size(-1)
        kv_seq_len = max_seq_len + max(history_lengths)
        if kv_seq_len >= self.rotary_emb.max_seq_len_cached:
            cos, sin = self.rotary_emb(value_states,
                                       seq_len=kv_seq_len + 128)
        query_states, key_states = apply_rotary_pos_emb(
            query_states,
            key_states,
            self.rotary_emb.cos_cached,
            self.rotary_emb.sin_cached,
            position_ids,
            position_ids_1d,
            q_embed=query_states,
            k_embed=key_states)

        # fill kv cache
        kv_seq_length = context.kv_seq_length
        q_seq_length = context.q_seq_length
        q_start_loc = context.q_start_loc
        fill_kv_cache(key_states,
                      value_states,
                      past_key_value[0],
                      past_key_value[1],
                      q_start_loc,
                      q_seq_length,
                      block_offsets=block_offsets,
                      history_lengths=history_lengths,
                      context=context)

        # attention
        attn_output = query_states
        block_size = past_key_value[0].size(1)
        paged_attention_fwd(
            query_states,
            past_key_value[0],
            past_key_value[1],
            attn_output,
            block_offsets,
            q_start_loc=q_start_loc,
            q_seqlens=q_seq_length,
            kv_seqlens=kv_seq_length,
            max_seqlen=max_seq_len,
        )
        hidden_size = num_heads * head_dim
        attn_output = attn_output.reshape(*hidden_states.shape[:-1], hidden_size)
```

(continues on next page)

(continued from previous page)

```
        # o proj
        attn_output = o_proj(attn_output)
        return attn_output, None, past_key_value
```

Note: The additional arguments like `history_lengths` and `block_offsets` are accessed from the `context` object, which acts as a container for the necessary inputs required by continuous batching and paged attention. Refer to the *context info* for more detail about `context` object.

We have replaced certain operations with our custom Triton kernel for two reasons:

1. The custom Triton kernel allows us to incorporate new features, such as `paged_attention_fwd`.
2. Fused kernels offer superior performance compared to the pure PyTorch implementation.

Now that we have the updated implementations for the two modules, let's register them in `lmdeploy/pytorch/models/module_map.py`.

```
# lmdeploy/pytorch/models/module_map.py
MODEL_MAP.update({
    'transformers.models.llama.modeling_llama.LlamaAttention':
    'lmdeploy.pytorch.models.llama.LlamaAttention',
    'transformers.models.llama.modeling_llama.LlamaModel':
    'lmdeploy.pytorch.models.llama.LlamaModel'
})
```

In this mapping, the revised modules are associated with their original counterparts. When creating an `Engine`, the `ModelAgent` will automatically patch the model. Subsequently, we can conduct inference using these updated implementations.

20.2 Support Tensor Parallelism

If we aim to enable tensor parallelism (TP), it is necessary to partition the weights in the model. Let's build upon the previously mentioned modifications to accommodate TP in the Llama model:

In Llama (as well as in most Language Model models), the weight partition primarily affects the Linear layers. Specifically, for the following components:

- In `LlamaAttention`: `q_proj`, `k_proj`, `v_proj` require column-wise partitioning, while `o_proj` necessitates row-wise partitioning.
- In `LlamaMLP`: `gate_proj` and `up_proj` require column-wise partitioning, while `down_proj` requires row-wise partitioning.

We can implement the _distribution_partition_fn in each of the rewritten modules:

```
# lmdeploy/pytorch/models/llama.py
from ..dist_utils import (colwise_parallelize_linear_fn,
                          rowwise_parallelize_linear_fn)

class LlamaAttention(nn.Module):
    @classmethod
    def _distribute_partition_fn(cls, mod_name: str, mod: nn.Module,
                                 device_mesh: DeviceMesh):
        """Distribution partition callback."""
```

(continues on next page)

(continued from previous page)

```
        if mod_name in ['q_proj', 'k_proj', 'v_proj']:
            colwise_parallelize_linear_fn(mod,
                                          device_mesh=device_mesh,
                                          to_local=True)
        elif mod_name in ['o_proj']:
            rowwise_parallelize_linear_fn(mod,
                                          device_mesh=device_mesh,
                                          to_local=True)

class LlamaMLP(nn.Module):
    @classmethod
    def _distribute_partition_fn(cls, mod_name: str, mod: nn.Module,
                                 device_mesh: DeviceMesh):
        """Distribution partition callback."""
        if mod_name in ['gate_proj', 'up_proj']:
            colwise_parallelize_linear_fn(mod,
                                          device_mesh=device_mesh,
                                          to_local=True)
        elif mod_name in ['down_proj']:
            rowwise_parallelize_linear_fn(mod,
                                          device_mesh=device_mesh,
                                          to_local=True)
```

In the process of loading model weights, the `_distribute_partition_fn` is called to distribute the weights of specific modules across different devices. Following the weight partitioning, it becomes necessary to perform `all_reduce` on the output tensors of `o_proj` and `down_proj`. While one option is to include `all_reduce` directly in the forward method, an alternative approach is to introduce the `_distribute_output_fn` call:

```
# lmdeploy/pytorch/models/llama.py
import torch.distributed as dist

class LlamaAttention(nn.Module):
    @classmethod
    def _distribute_output_fn(cls, outputs, device_mesh: DeviceMesh):
        """Distribution output hook."""
        dist.all_reduce(outputs[0])
        return outputs

class LlamaMLP(nn.Module):
    @classmethod
    def _distribute_output_fn(cls, outputs, device_mesh: DeviceMesh):
        """Distribution output hook."""
        dist.all_reduce(outputs)
        return outputs
```

It is essential to remember to add `LlamaMLP` to the `module_map`:

```
# lmdeploy/pytorch/models/module_map.py
MODEL_MAP.update({
    'transformers.models.llama.modeling_llama.LlamaMLP':
    'lmdeploy.pytorch.models.llama.LlamaMLP'
})
```

With these adjustments, the model is now capable of utilizing multiple GPUs for deploying Large Language Models (LLM). This enables efficient distribution of computations across different devices in a parallelized manner.

20.3 Debug Module

When the output of the model does not meet expectations, we would like to debug a specific module to determine if the added rewrite is correct. `lmdeploy.pytorch` provides some tools to assist with accuracy alignment. Let's take `LlamaAttention` module as an example.

First, create an instance of the module that we want to debug:

```
import torch
from transformers import AutoModelForCausalLM

# get module
model_path = 'meta-llama/Llama-2-7b-chat-hf'
dtype = torch.float16
model = AutoModelForCausalLM.from_pretrained(model_path).to(torch.float16).cuda()
self_attn = model.model.layers[0].self_attn
```

Extract the inputs/outputs with `ModuleIOExtractor`.

```
from lmdeploy.pytorch.tools.make_inputs import ModuleIOExtractor

# extract module input/output
input_ids = torch.tensor([[1, 2, 3, 4, 5]]).cuda()
extractor = ModuleIOExtractor(model, self_attn)
attn_args, attn_kwargs, attn_output = extractor.extract(input_ids)
```

The inputs of rewrite module are different from the inputs of origin module:

1. Module requires some special inputs, which are passed through `StepContext`. We can create one with `make_step_context`.
2. `input_ids`, `hidden_states` should be continuous. We can use `continuous_tensor` to do the process.
3. `past_key_value` should be paged to meet the demand of paged attention.

Based on the reason above, the input should be updated:

```
from lmdeploy.pytorch.tools.make_inputs import make_step_context
from lmdeploy.pytorch.tools.layout_convert import continuous_tensor

# create patched input/output
context = make_step_context(input_ids,
                            kv_cache_dtype=dtype,
                            num_key_value_heads=32)
seq_length = context.q_seq_length
attn_kwargs['hidden_states'] = continuous_tensor(
    attn_kwargs['hidden_states'],
    seq_length)
attn_kwargs['past_key_value'] = context.kv_caches[0]
```

Then you can start the rewrite and compare the correctness of the results.

```
from lmdeploy.pytorch.models import patch

# patch and test
patched_self_attn = patch(self_attn, extra_args=['context'])
with torch.inference_mode():
    patched_output = patched_self_attn.patched_forward(*attn_args,
                                                       **attn_kwargs,
                                                       context=context)
torch.testing.assert_close(patched_output[0],
                           continuous_tensor(attn_output[0], seq_length))
```

Adjust the rewrite module until the output can be aligned.

20.4 Appendix

20.4.1 context info

```
@dataclass
class StepContext:
    """context of Model.
    """
    inputs: ModelInputs
    block_offsets: torch.LongTensor
    position_ids: torch.LongTensor
    position_ids_1d: torch.LongTensor
    q_start_loc: torch.LongTensor
    history_lengths: torch.LongTensor
    seq_length: torch.LongTensor
    max_seq_length: int
    kv_seq_length: torch.LongTensor
    kv_caches: List
    is_decoding: bool
    world_size: int = 1
    json_config: Dict = None
    local_adapter_ids: torch.LongTensor = None
    global_adapter_ids: torch.LongTensor = None
    adapter_offsets: torch.LongTensor = None
    max_rank: int = 0
```

20.4.2 FAQ

- **How to invoke the original forward method?**

A common approach is to add hooks to a method rather than performing a complete rewrite. To access the unpatched module, you can utilize self.origin_mod within the rewritten method.

- **How to register modules in remote code?**

For modules located in remote code, pinpointing them via `qualname` might be challenging. `lmdeploy.pytorch` facilitates registration using abbreviations for such modules:n:

```
MODULE_MAP.update({
    'modeling_internlm.InternLMAttention':
    'lmdeploy.pytorch.models.internlm.PatchedInternLMAttention',
})
```

[!NOTE]

Although abbreviations are supported, they tend to have lower priority. It is advisable to register modules using their complete `qualname` for more robust and accurate mapping.

- **How to support different modules with the same name?**

You can accommodate multiple modules with the same name within a single rewrite module by providing distinct implementations based on their attributes. For instance, consider `baichuan2` 7b/13b:

```
class BaichuanModel(nn.Module):
    def forward(self, ...):
        if self.config.num_hidden_layers == 32:
            return forward_7b(...)
        else:
            return forward_default(...)
```

- **How to perform post-initialization for a rewrite module?**

To execute tasks after model weight loading, introduce a `_update_model_fn` method in your rewrite module. This method will be automatically called post-initialization:

```
class LlamaAttention:
    def _update_model_fn(self):
        # ADD YOUR CODE HERE
```

Here, you can include any additional post-initialization steps or configurations needed for your specific use case.

CHAPTER

TWENTYONE

CONTEXT LENGTH EXTRAPOLATION

Long text extrapolation refers to the ability of LLM to handle data longer than the training text during inference. TurboMind engine now support LlamaDynamicNTKScalingRotaryEmbedding and the implementation is consistent with huggingface.

21.1 Usage

You can enable the context length extrapolation abality by modifying the TurbomindEngineConfig. Edit the `session_len` to the expected length and change `rope_scaling_factor` to a number no less than 1.0.

Here is an example:

```
from lmdeploy import pipeline, GenerationConfig, TurbomindEngineConfig

backend_config = TurbomindEngineConfig(rope_scaling_factor=2.0, session_len=160000)
pipe = pipeline('internlm/internlm2-chat-7b', backend_config=backend_config)
prompt = 'Use a long prompt to replace this sentence'
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
response = pipe(prompt, gen_config=gen_config)
print(response)
```

21.2 Evaluation

We use several methods to evaluate the long-context-length inference ability of LMDeploy, including *passkey retrieval*, *needle in a haystack* and computing *perplexity*

21.2.1 Passkey Retrieval

You can try the following code to test how many times LMDeploy can retrieval the special key.

```
import numpy as np
from lmdeploy import pipeline
from lmdeploy import TurbomindEngineConfig

session_len = 160000
backend_config = TurbomindEngineConfig(rope_scaling_factor=2.0, session_len=session_len)
pipe = pipeline('internlm/internlm2-chat-7b', backend_config=backend_config)


def passkey_retrival(session_len, n_round=5):
    # create long context input
    tok = pipe.tokenizer
    task_description = 'There is an important info hidden inside a lot of irrelevant␣
↪text. Find it and memorize them. I will quiz you about the important information there.
↪'
    garbage = 'The grass is green. The sky is blue. The sun is yellow. Here we go. There␣
↪and back again.'

    for _ in range(n_round):
        n_times = (session_len - 1000) // len(tok.encode(garbage))
        n_garbage_prefix = np.random.randint(0, n_times)
        n_garbage_suffix = n_times - n_garbage_prefix
        garbage_prefix = ' '.join([garbage] * n_garbage_prefix)
        garbage_suffix = ' '.join([garbage] * n_garbage_suffix)
        pass_key = np.random.randint(1, 50000)
        information_line = f'The pass key is {pass_key}. Remember it. {pass_key} is the␣
↪pass key.'  # noqa: E501
        final_question = 'What is the pass key? The pass key is'
        lines = [
            task_description,
            garbage_prefix,
            information_line,
            garbage_suffix,
            final_question,
        ]

        # inference
        prompt = ' '.join(lines)
        response = pipe([prompt])
        print(pass_key, response)


passkey_retrival(session_len, 5)
```

21.2.2 Needle In A Haystack

OpenCompass offers very useful tools to perform needle-in-a-haystack evaluation. For specific instructions, please refer to the guide.

21.2.3 Perplexity

The following codes demonstrate how to use LMDeploy to calculate perplexity.

```
from datasets import load_dataset
from lmdeploy import TurbomindEngineConfig
from lmdeploy.turbomind import TurboMind
import numpy as np

# load model and tokenizer
engine_config = TurbomindEngineConfig(rope_scaling_factor=2.0, session_len=160000)
engine = TurboMind.from_pretrained('internlm/internlm2-chat-7b', engine_config)
tokenizer = engine.tokenizer
generator = engine.create_instance()

# get perplexity
text = 'Use a long prompt to replace this sentence'
input_ids = tokenizer.encode(text)
loss = generator.get_ppl(input_ids)[0]
ppl = np.exp(loss)
```

CHAPTER

TWENTYTWO

CUSTOMIZED CHAT TEMPLATE

The effect of the applied chat template can be observed by **setting log level** `INFO`.

LMDeploy supports two methods of adding chat templates:

- One approach is to utilize an existing conversation template by directly configuring a JSON file like the following.

```
{
    "model_name": "your awesome chat template name",
    "system": "<|im_start|>system\n",
    "meta_instruction": "You are a robot developed by LMDeploy.",
    "eosys": "<|im_end|>\n",
    "user": "<|im_start|>user\n",
    "eoh": "<|im_end|>\n",
    "assistant": "<|im_start|>assistant\n",
    "eoa": "<|im_end|>",
    "separator": "\n",
    "capability": "chat",
    "stop_words": ["<|im_end|>"]
}
```

`model_name` is a required field and can be either the name of an LMDeploy built-in chat template (which can be viewed through `lmdeploy list`), or a new name. Other fields are optional.

1. When `model_name` is the name of a built-in chat template, the non-null fields in the JSON file will override the corresponding attributes of the original chat template.
2. However, when `model_name` is a new name, it will register `BaseChatTemplate` directly as a new chat template. The specific definition can be referred to BaseChatTemplate.

The new chat template would be like this:

```
{system}{meta_instruction}{eosys}{user}{user_content}{eoh}{assistant}{assistant_
↪content}{eoa}{separator}{user}...
```

When using the CLI tool, you can pass in a custom chat template with `--chat-template`, for example.

```
lmdeploy serve api_server internlm/internlm2-chat-7b --chat-template ${JSON_FILE}
```

You can also pass it in through the interface function, for example.

```
from lmdeploy import ChatTemplateConfig, serve
serve('internlm/internlm2-chat-7b',
      chat_template_config=ChatTemplateConfig.from_json('${JSON_FILE}'))
```

- Another approach is to customize a Python chat template class like the existing LMDeploy chat templates. It can be used directly after successful registration. The advantages are a high degree of customization and strong controllability. Below is an example of registering an LMDeploy chat template.

```
from lmdeploy.model import MODELS, BaseChatTemplate


@MODELS.register_module(name='customized_model')
class CustomizedModel(BaseChatTemplate):
    """A customized chat template."""

    def __init__(self,
                 system='<|im_start|>system\n',
                 meta_instruction='You are a robot developed by LMDeploy.',
                 user='<|im_start|>user\n',
                 assistant='<|im_start|>assistant\n',
                 eosys='<|im_end|>\n',
                 eoh='<|im_end|>\n',
                 eoa='<|im_end|>',
                 separator='\n',
                 stop_words=['<|im_end|>', '<|action_end|>']):
        super().__init__(system=system,
                         meta_instruction=meta_instruction,
                         eosys=eosys,
                         user=user,
                         eoh=eoh,
                         assistant=assistant,
                         eoa=eoa,
                         separator=separator,
                         stop_words=stop_words)


from lmdeploy import ChatTemplateConfig, pipeline

messages = [{'role': 'user', 'content': 'who are you?'}]
pipe = pipeline('internlm/internlm2-chat-7b',
                chat_template_config=ChatTemplateConfig('customized_model'))
for response in pipe.stream_infer(messages):
    print(response.text, end='')
```

In this example, we register a LMDeploy chat template that sets the model to be created by LMDeploy, so when the user asks who the model is, the model will answer that it was created by LMDeploy.

CHAPTER

TWENTYTHREE

HOW TO DEBUG TURBOMIND

Turbomind is implemented in C++, which is not as easy to debug as Python. This document provides basic methods for debugging Turbomind.

23.1 Prerequisite

First, complete the local compilation according to the commands in *Build in localhost*.

23.2 Configure Python debug environment

Since many large companies currently use Centos 7 for online production environments, we will use Centos 7 as an example to illustrate the process.

23.2.1 Obtain `glibc` and `python3` versions

```
rpm -qa | grep glibc
rpm -qa | grep python3
```

The result should be similar to this:

```
[username@hostname workdir]# rpm -qa | grep glibc
glibc-2.17-325.el7_9.x86_64
glibc-common-2.17-325.el7_9.x86_64
glibc-headers-2.17-325.el7_9.x86_64
glibc-devel-2.17-325.el7_9.x86_64

[username@hostname workdir]# rpm -qa | grep python3
python3-pip-9.0.3-8.el7.noarch
python3-rpm-macros-3-34.el7.noarch
python3-rpm-generators-6-2.el7.noarch
python3-setuptools-39.2.0-10.el7.noarch
python3-3.6.8-21.el7_9.x86_64
python3-devel-3.6.8-21.el7_9.x86_64
python3.6.4-sre-1.el6.x86_64
```

Based on the information above, we can see that the version of `glibc` is `2.17-325.el7_9.x86_64` and the version of `python3` is `3.6.8-21.el7_9.x86_64`.

23.2.2 Download and install debuginfo library

Download glibc-debuginfo-common-2.17-325.el7.x86_64.rpm, glibc-debuginfo-2.17-325.el7.x86_64.rpm, and python3-debuginfo-3.6.8-21.el7.x86_64.rpm from http://debuginfo.centos.org/7/x86_64.

```
rpm -ivh glibc-debuginfo-common-2.17-325.el7.x86_64.rpm
rpm -ivh glibc-debuginfo-2.17-325.el7.x86_64.rpm
rpm -ivh python3-debuginfo-3.6.8-21.el7.x86_64.rpm
```

23.2.3 Upgrade GDB

```
sudo yum install devtoolset-10 -y
echo "source scl_source enable devtoolset-10" >> ~/.bashrc
source ~/.bashrc
```

23.2.4 Verification

```
gdb python3
```

The output should be similar to this:

```
[username@hostname workdir]# gdb python3
GNU gdb (GDB) Red Hat Enterprise Linux 9.2-10.el7
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
   <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from python3...
(gdb)
```

If it shows `Reading symbols from python3`, the configuration has been successful.

For other operating systems, please refer to DebuggingWithGdb.

23.3 Set up symbolic links

After setting up symbolic links, there is no need to install it locally with `pip` every time.

```
# Change directory to lmdeploy, e.g.
cd /workdir/lmdeploy

# Since it has been built in the build directory
# Link the lib directory
cd lmdeploy && ln -s ../build/lib . && cd ..
# (Optional) Link compile_commands.json for clangd index
ln -s build/compile_commands.json .
```

23.4 Start debugging

```
# Use gdb to start the API server with Llama-2-13b-chat-hf, e.g.
gdb --args python3 -m lmdeploy serve api_server /workdir/Llama-2-13b-chat-hf

# Set directories in gdb
Reading symbols from python3...
(gdb) set directories /workdir/lmdeploy

# Set a breakpoint using the relative path, e.g.
(gdb) b src/turbomind/models/llama/BlockManager.cc:104

# When it shows
# ```
# No source file named src/turbomind/models/llama/BlockManager.cc.
# Make breakpoint pending on future shared library load? (y or [n])
# ```
# Just type `y` and press enter

# Run
(gdb) r

# (Optional) Use https://github.com/InternLM/lmdeploy/blob/main/benchmark/profile_
↪restful_api.py to send a request

python3 profile_restful_api.py --server_addr 127.0.0.1:23333 --tokenizer_path /workdir/
↪Llama-2-13b-chat-hf --dataset /workdir/ShareGPT_V3_unfiltered_cleaned_split.json --
↪concurrency 1 --num_prompts 1
```

23.5 Using GDB

Refer to GDB Execution Commands and happy debugging.

CHAPTER

TWENTYFOUR

LMDEPLOY-QOS INTRODUCE AND USAGE

24.1 Background

With the rise of Large Language Model (LLM) and Artificial General Intelligence (AGI), numerous inference frameworks have emerged. These frameworks deliver scalable and high-performance services by serving online workloads with language models. However, these workloads often come from multiple user groups, exhibiting rapid changes in workload patterns within short periods. Many inference frameworks struggle to meet the demands of such multi-tenancy traffic patterns and fail to effectively shape user behaviors. Therefore, we believe that systematically considering these issues in LLM inference framework is both valuable and necessary.

24.2 User Categorizations for Multi-tenancy Handling

LMDeploy-QoS is part of LMDeploy, offering a range of multi-tenancy functionalities. It requires users to tag their inference requests with appropriate user identifications (user_id in configuration or codebase). The system operates based on a dictionary-like configuration that serves as a multi-tenancy policy. In this configuration, users are mapped to different classes, known as "user groups", each configured with a ratio value. Our multi-tenancy strategy reads this configuration and schedules user inference requests according to class priority and the difference between the predefined ratio and real-time allocation ratio. Extensive testing shows that LMDeploy-QoS significantly enhances LLM serving reliability and GPU resource utilization for real-world large language model inference workloads.

We categorize LMDeploy users into four groups:

- Platinum
- Gold
- Silver
- Bronze

Based on our experiences in delivering LLM services, we can map the following four types of users to these user groups:

- Platinum: VIP or administrative users. Examples include service inspectors or product demo presenters who require uninterrupted online services. Their workloads are typically at a low frequency and require limited resources.
- Gold: Contracted business user groups requiring specific quantities of reliable services. For instance, Company A signs a contract with the LLM service provider to secure X requests/sec service capability with Z% availability for its employees at the cost of Y million dollars per year.
- Silver: The vast majority of users fall under this category. Most trial or monthly subscribed users are included in this group. They need a relatively small quantity of services, but their user experiences significantly affect the LLM service reputation.

- Bronze: Heavy users who pay minimal fees to LLM providers.

The above user group categorization is intended for guidance rather than as a recommendation for all LMDeploy users, as it may not be suitable for all LLM service providers. Users can develop their own method of categorizing users based on their observations of daily workloads.

Next, we will discuss how LMDeploy schedules requests based on these categorizations.

24.3 Multi-tenancy Strategies

24.3.1 Strategy 1: prioritized scheduling between groups

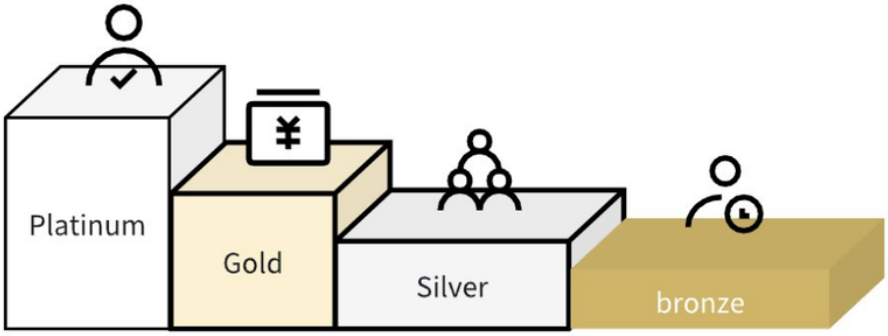
This strategy works as simple as its title suggests.

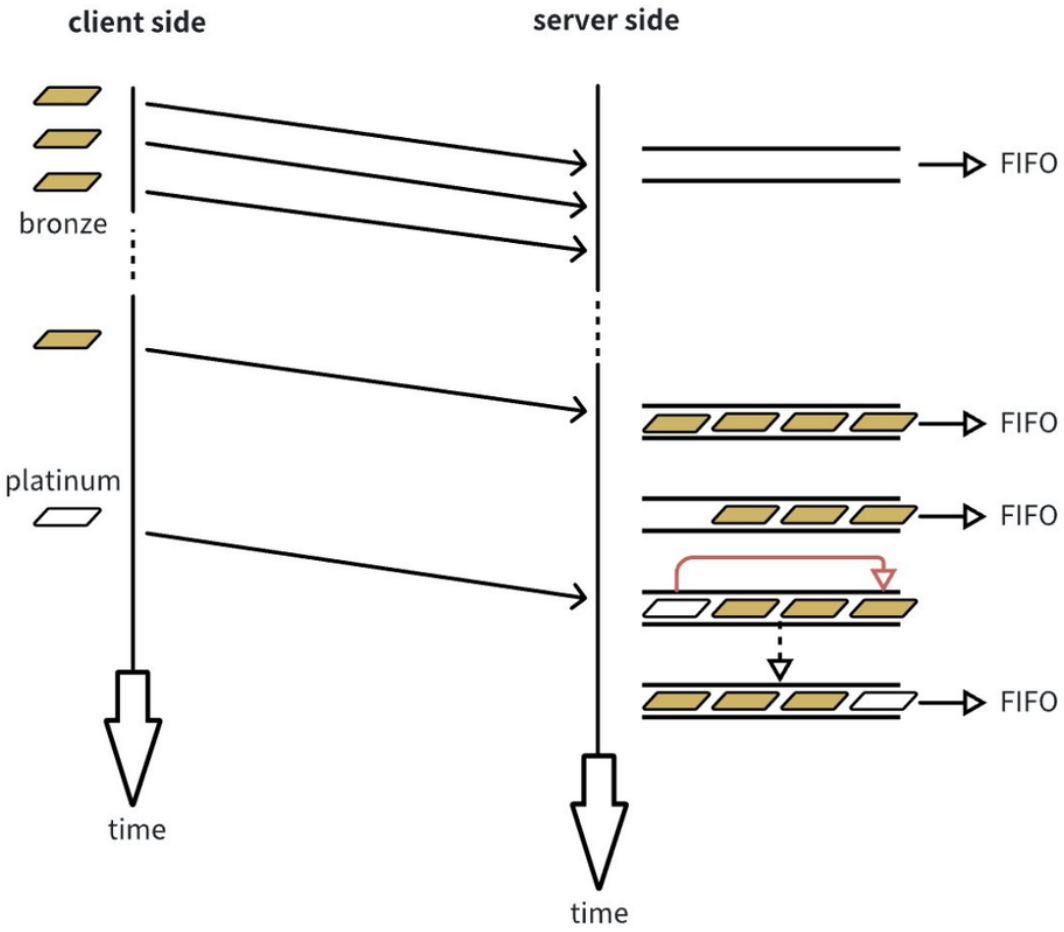
User groups are introduced for this strategy, with users in each group to be specified. Recommended user groups are as follows:

- Platinum
- Gold
- Silver
- Bronze

The priority of each group decreases sequentially. Requests with higher priority are always given precedence for inference. Be noted that the scheduling is performed at the time of request reception, so lower-priority requests will not be withdrawn from the GPU if they are already under inference.

The below diagram shows how the prioritization works. As you can see, the platinum request is reprioritized and moved to the queue head.

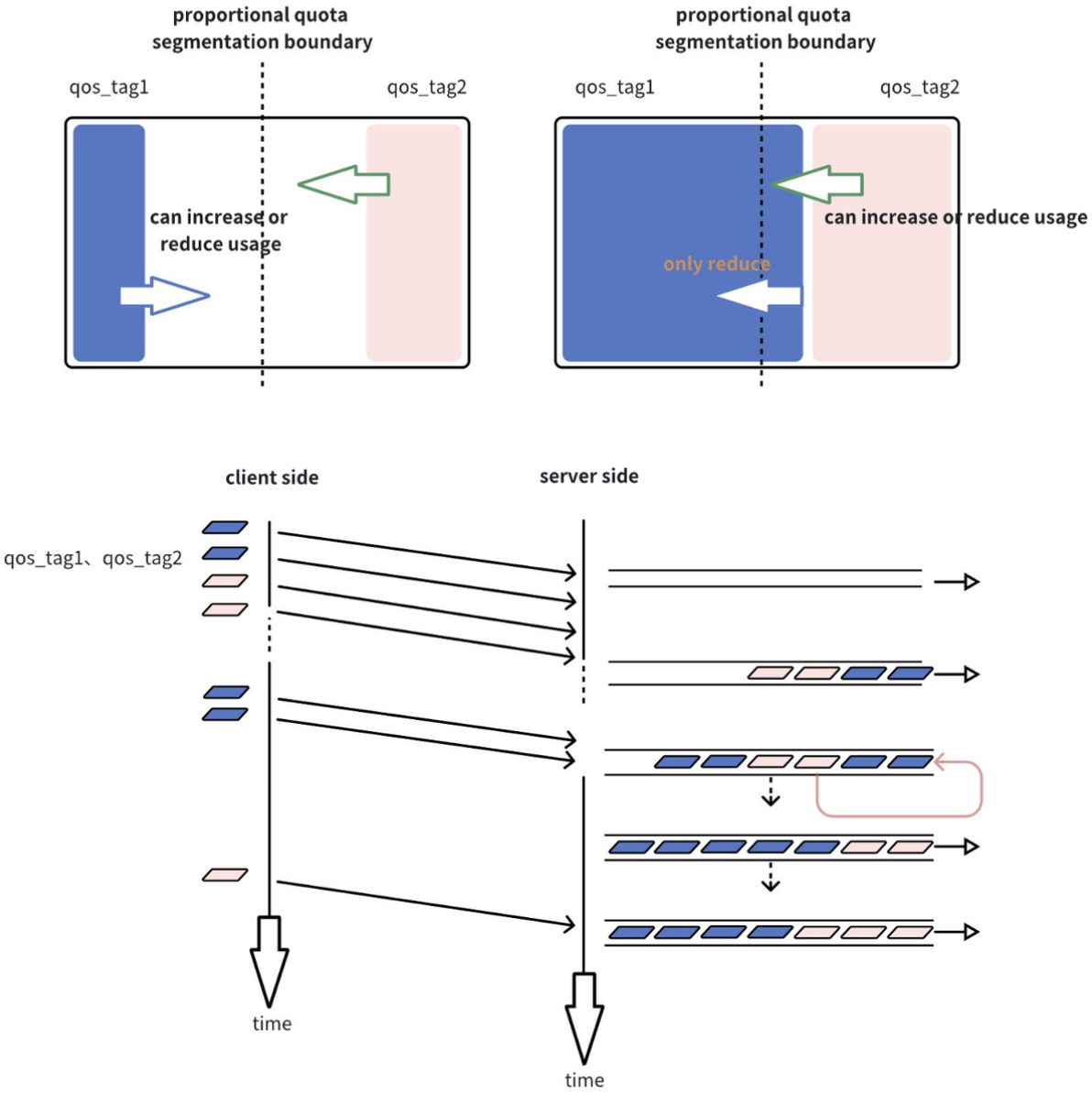

Platinum
Gold
Silver
bronze


client side
server side
bronze
platinum
FIFO
FIFO
FIFO
FIFO
time
time

24.3.2 Strategy 2: proportionally rated scheduling with a pre-defined ratio within user group

This strategy works only within the user group. We introduce a within-group user quota configuration table. This table defines users' "ideal share ratio" with a sum value of 100% GPU resource. Each "user" appears in the list as a user_id, and a user can only belong to one user group. Requests from different users will be scheduled according to each user's "ideal share ratio". To be specific, users with their real-time usage ratio lower than their quota ratio will have priority over users whose real-time usage ratio is higher than their quota ratio. It is worth noting that the scheduling only considers users in the request queue, ignoring any absent users from the configuration table.

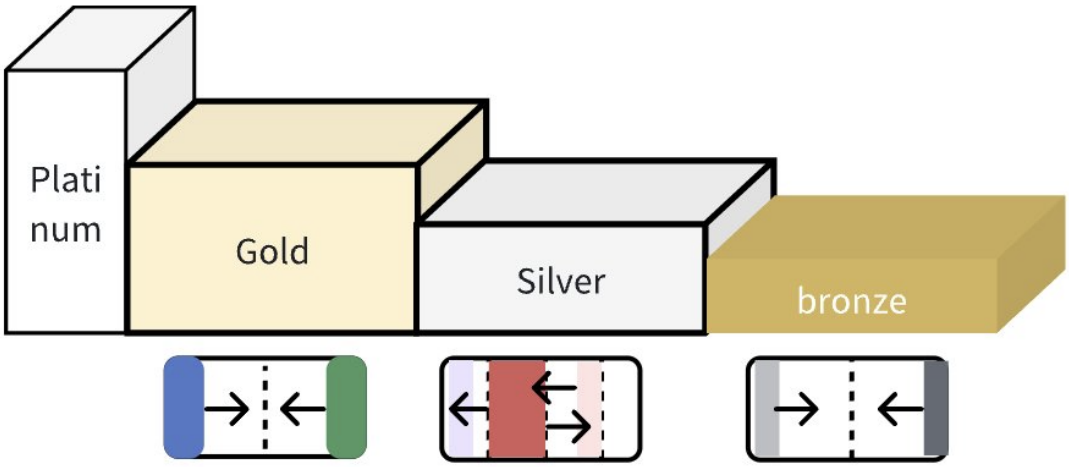
The below diagram shows a typical example of how this strategy works.



24.3.3 Strategy 3: a combination strategy of 1 and 2

We can call it a hybrid strategy. The way we hybrid these 2 strategies is fairly simple: we adopt strategy 1 in between user groups, and adopt strategy 2 within a user group. So users belonging to different groups with different priorities will only obey strategy 1 to determine their privilege in resource allocation. That is, when both strategies are applied, the first strategy will overpower the second. When it comes to a situation that no cross-group requests are waiting for serving, the within-group strategy 2 comes into play.

Below is a diagram showing it.



To be noted, there could be other ways of hybrid strategies 1 & 2, and this doc only introduces one method that works well in our scenario. Considering that prioritization and pro-rated sharing are obviously conflicting strategies, there is no easy way to mix them to work within a single dimension.

24.4 A Sample QoS Configuration

The configuration will be specified by the `--qos-config-path` flag, and will be loaded by program upon startup.

```
{
    "enable_user_qos": true,
    "user_groups": [
        "Platinum",
        "Gold",
        "Silver",
        "Bronze"
    ],
    "user_group_map": {
        "Platinum": [
            {
                "id": "user_id0",
```

(continues on next page)

(continued from previous page)

```
                "quota_pct": 100
            },
            {
                "id": "default",
                "quota_pct": 0
            }
        ],
        "Gold": [
            {
                "id": "user_id1",
                "quota_pct": 50
            },
            {
                "id": "user_id2",
                "quota_pct": 50
            }
        ],
        "Silver": [
            {
                "id": "user_id3",
                "quota_pct": 5
            },
            {
                "id": "default",
                "quota_pct": 95
            }
        ],
        "Bronze": [
            {
                "id": "user_id4",
                "quota_pct": 30
            },
            {
                "id": "user_id5",
                "quota_pct": 30
            },
            {
                "id": "user_id6",
                "quota_pct": 40
            },
            {
                "id": "default",
                "quota_pct": 0
            }
        ]
    }
}
```

24.5 How to perform inference job with Lmdeploy-QoS aware

We provide the code link below to show how to call infer requests with multi-tenancy strategy awarded. What the qos related argument appears as in http body

/v1/chat/interactive_qos

```
curl -X POST http://localhost/v1/chat/interactive_qos \
  -H "Content-Type: application/json" \
  -d '{
  "prompt": "Hello,Hello",
  "session_id": -1,
  "interactive_mode": false,
  "stream": false,
  "stop": false,
  "request_output_len": 512,
  "top_p": 0.8,
  "top_k": 40,
  "temperature": 0.8,
  "repetition_penalty": 1,
  "ignore_eos": false,
  "user_id": "user_id0"
}'
```

/v1/chat/completions_qos

```
curl -X POST http://localhost/v1/chat/completions_qos \
  -H "Content-Type: application/json" \
  -d '{
  "model": "internlm-chat-7b",
  "messages": "Hello,Hello",
  "temperature": 0.7,
  "top_p": 1,
  "n": 1,
  "max_tokens": 512,
  "stop": false,
  "stream": false,
  "presence_penalty": 0,
  "frequency_penalty": 0,
  "repetition_penalty": 1,
  "session_id": -1,
  "ignore_eos": false,
  "user_id": "user_id0"
}'
```

/v1/completions_qos

```
curl -X POST http://localhost/v1/completions_qos \
  -H "Content-Type: application/json" \
  -d '{
  "model": "internlm-chat-7b",
  "prompt": "Hello,Hello",
  "suffix": "string",
  "temperature": 0.7,
```

(continues on next page)

(continued from previous page)

```
  "n": 1,
  "max_tokens": 16,
  "stop": "string",
  "stream": false,
  "top_p": 1,
  "repetition_penalty": 1,
  "session_id": -1,
  "ignore_eos": false,
  "user_id": "user_id0"
}'
```

24.6 File Configuration Modification

The template of the configuration file is located at: `lmdeploy/server/qos_engine/qos_config.json.template`. Add the necessary users based on actual requirements, ensure correct priority assignment, and set appropriate quota values.

24.7 Passing Configuration Parameters

Upon starting the api_server, pass the configuration file and its path using the `--qos-config-path` flag. An example is illustrated below:

```
CUDA_VISIBLE_DEVICES=0 lmdeploy serve api_server internlm/internlm-chat-7b --server-port␣
↪8000 --qos-config-path lmdeploy/serve/qos_engine/qos_config.json.template
```

24.8 Contributor

Eric, sallyjunjun, sfireworks, Dofgal, shadow

CHAPTER

TWENTYFIVE

INFERENCE PIPELINE

25.1 pipeline

lmdeploy.**pipeline**(*model_path: str*, *model_name: Optional[str] = None*, *backend_config: Optional[Union[*lmdeploy.messages.TurbomindEngineConfig, lmdeploy.messages.PytorchEngineConfig*]] = None*, *chat_template_config: Optional[*lmdeploy.model.ChatTemplateConfig*] = None*, *log_level='ERROR'*, ***kwargs*)

Parameters

- **model_path** (*str*) – the path of a model. It could be one of the following options:
  - i) A local directory path of a turbomind model which is

    converted by *lmdeploy convert* command or download from ii) and iii).

  - ii) The model_id of a lmdeploy-quantized model hosted

    inside a model repo on huggingface.co, such as "InternLM/internlm-chat-20b-4bit", "lmdeploy/llama2-chat-70b-4bit", etc.

  - iii) The model_id of a model hosted inside a model repo

    on huggingface.co, such as "internlm/internlm-chat-7b", "Qwen/Qwen-7B-Chat ", "baichuan-inc/Baichuan2-7B-Chat" and so on.

- **model_name** (*str*) – needed when model_path is a pytorch model on huggingface.co, such as "internlm/internlm-chat-7b", "Qwen/Qwen-7B-Chat ", "baichuan-inc/Baichuan2-7B-Chat" and so on.
- **backend_config** (*TurbomindEngineConfig | PytorchEngineConfig*) – backend config instance. Default to None.
- **chat_template_config** (ChatTemplateConfig) – chat template configuration. Default to None.
- **log_level** (*str*) – set log level whose value among [CRITICAL, ERROR, WARNING, INFO, DEBUG]

Examples

```
>>> # LLM
>>> import lmdeploy
>>> pipe = lmdeploy.pipeline('internlm/internlm-chat-7b')
>>> response = pipe(['hi','say this is a test'])
>>> print(response)
>>>
>>> # VLM
>>> from lmdeploy.vl import load_image
>>> from lmdeploy import pipeline, TurbomindEngineConfig, ChatTemplateConfig
>>> pipe = pipeline('liuhaotian/llava-v1.5-7b',
...                 backend_config=TurbomindEngineConfig(session_len=8192),
...                 chat_template_config=ChatTemplateConfig(model_name='vicuna'))
>>> im = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/
↪demo/resources/human-pose.jpg')
>>> response = pipe([('describe this image', [im])])
>>> print(response)
```

25.2 serving

lmdeploy.**serve**(*model_path: str*, *model_name: Optional[str] = None*, *backend: Literal[turbomind, pytorch] = 'turbomind'*, *backend_config: Optional[Union[*lmdeploy.messages.TurbomindEngineConfig, lmdeploy.messages.PytorchEngineConfig*]] = None*, *chat_template_config: Optional[*lmdeploy.model.ChatTemplateConfig*] = None*, *server_name: str = '0.0.0.0'*, *server_port: int = 23333*, *log_level: str = 'ERROR'*, *api_keys: Optional[Union[str, List[str]]] = None*, *ssl: bool = False*, ***kwargs*)

This will run the api_server in a subprocess.

Parameters

- **model_path** (*str*) – the path of a model. It could be one of the following options:
  - i) A local directory path of a turbomind model which is

    converted by *lmdeploy convert* command or download from ii) and iii).

  - ii) The model_id of a lmdeploy-quantized model hosted

    inside a model repo on huggingface.co, such as "InternLM/internlm-chat-20b-4bit", "lmdeploy/llama2-chat-70b-4bit", etc.

  - iii) The model_id of a model hosted inside a model repo

    on huggingface.co, such as "internlm/internlm-chat-7b", "Qwen/Qwen-7B-Chat ", "baichuan-inc/Baichuan2-7B-Chat" and so on.

- **model_name** (*str*) – needed when model_path is a pytorch model on huggingface.co, such as "internlm/internlm-chat-7b", "Qwen/Qwen-7B-Chat ", "baichuan-inc/Baichuan2-7B-Chat" and so on.
- **backend** (*str*) – either *turbomind* or *pytorch* backend. Default to *turbomind* backend.

- **backend_config** (*TurbomindEngineConfig | PytorchEngineConfig*) – backend config instance. Default to none.
- **chat_template_config** (ChatTemplateConfig) – chat template configuration. Default to None.
- **server_name** (*str*) – host ip for serving
- **server_port** (*int*) – server port
- **log_level** (*str*) – set log level whose value among [CRITICAL, ERROR, WARNING, INFO, DEBUG]
- **api_keys** (*List[str] | str | None*) – Optional list of API keys. Accepts string type as a single api_key. Default to None, which means no api key applied.
- **ssl** (*bool*) – Enable SSL. Requires OS Environment variables 'SSL_KEYFILE' and 'SSL_CERTFILE'.

Returns A client chatbot for LLaMA series models.

Return type APIClient

Examples

```
>>> import lmdeploy
>>> client = lmdeploy.serve('internlm/internlm-chat-7b', 'internlm-chat-7b')
>>> for output in client.chat('hi', 1):
...     print(output)
```

lmdeploy.**client**(*api_server_url: str = 'http://0.0.0.0:23333'*, *api_key: Optional[str] = None*, ***kwargs*)

Parameters

- **api_server_url** (*str*) – communicating address 'http://<ip>:<port>' of api_server
- **api_key** (*str | None*) – api key. Default to None, which means no api key will be used.

Returns Chatbot for LLaMA series models with turbomind as inference engine.

25.3 PytorchEngineConfig

class lmdeploy.**PytorchEngineConfig**(*model_name: str = ''*, *tp: int = 1*, *session_len: Optional[int] = None*, *max_batch_size: int = 128*, *cache_max_entry_count: float = 0.8*, *eviction_type: str = 'recompute'*, *prefill_interval: int = 16*, *block_size: int = 64*, *num_cpu_blocks: int = 0*, *num_gpu_blocks: int = 0*, *adapters: Optional[Dict[str, str]] = None*, *max_prefill_token_num: int = 4096*, *thread_safe: bool = False*, *enable_prefix_caching: bool = False*, *download_dir: Optional[str] = None*, *revision: Optional[str] = None*)

PyTorch Engine Config.

Parameters

- **model_name** (*str*) – name of the given model.
- **tp** (*int*) – Tensor Parallelism. default 1.
- **session_len** (*int*) – Max session length. Default None.

- **max_batch_size** (*int*) – Max batch size. Default 128.
- **cache_max_entry_count** (*float*) – the percentage of gpu memory occupied by the k/v cache. For lmdeploy versions greater than *v0.2.1*, it defaults to 0.8, signifying the percentage of FREE GPU memory to be reserved for the k/v cache
- **eviction_type** (*str*) – What action to perform when kv cache is full, [‘recompute’, ‘copy’], Default ‘recompute’.
- **prefill_interval** (*int*) – Interval to perform prefill, Default 16.
- **block_size** (*int*) – paging cache block size, default 64.
- **num_cpu_blocks** (*int*) – Num cpu blocks. If num is 0, cache would be allocate according to current environment.
- **num_gpu_blocks** (*int*) – Num gpu blocks. If num is 0, cache would be allocate according to current environment.
- **adapters** (*dict*) – The path configs to lora adapters.
- **max_prefill_token_num** (*int*) – tokens per iteration.
- **thread_safe** (*bool*) – thread safe engine instance.
- **enable_prefix_caching** (*bool*) – Enable token match and sharing caches.
- **download_dir** (*str*) – Directory to download and load the weights, default to the default cache directory of huggingface.
- **revision** (*str*) – The specific model version to use. It can be a branch name, a tag name, or a commit id. If unspecified, will use the default version.

25.4 TurbomindEngineConfig

class lmdeploy.**TurbomindEngineConfig**(*model_name: Optional[str] = None*, *model_format: Optional[str] = None*, *tp: int = 1*, *session_len: Optional[int] = None*, *max_batch_size: int = 128*, *cache_max_entry_count: float = 0.8*, *cache_block_seq_len: int = 64*, *enable_prefix_caching: bool = False*, *quant_policy: int = 0*, *rope_scaling_factor: float = 0.0*, *use_logn_attn: bool = False*, *download_dir: Optional[str] = None*, *revision: Optional[str] = None*, *max_prefill_token_num: int = 8192*, *num_tokens_per_iter: int = 0*, *max_prefill_iters: int = 1*)

TurboMind Engine config.

Parameters

- **model_name** (*str*) – the name of the deployed model, deprecated and has no effect when version > 0.2.1
- **model_format** (*str*) – the layout of the deployed model. It can be one of the following values [hf, llama, awq], *hf* meaning *hf_llama*, *llama* meaning *meta_llama*, *awq* meaning the quantized model by AWQ.
- **tp** (*int*) – the number of GPU cards used in tensor parallelism, default to 1
- **session_len** (*int*) – the max session length of a sequence, default to None
- **max_batch_size** (*int*) – the max batch size during inference, default to 128

- **cache_max_entry_count** (*float*) – the percentage of gpu memory occupied by the k/v cache. For versions of lmdeploy between *v0.2.0* and *v0.2.1*, it defaults to 0.5, depicting the percentage of TOTAL GPU memory to be allocated to the k/v cache. For lmdeploy versions greater than *v0.2.1*, it defaults to 0.8, signifying the percentage of FREE GPU memory to be reserved for the k/v cache
- **cache_block_seq_len** (*int*) – the length of the token sequence in a k/v block, default to 64
- **enable_prefix_caching** (*bool*) – enable cache prompts for block reuse, default to False
- **quant_policy** (*int*) – default to 0. When k/v is quantized into 8 bit, set it to 4
- **rope_scaling_factor** (*int*) – scaling factor used for dynamic ntk, default to 0. TurboMind follows the implementation of transformer LlamaAttention
- **use_logn_attn** (*bool*) – whether or not to use log attn: default to False
- **download_dir** (*str*) – Directory to download and load the weights, default to the default cache directory of huggingface.
- **revision** (*str*) – The specific model version to use. It can be a branch name, a tag name, or a commit id. If unspecified, will use the default version.
- **max_prefill_token_num** (*int*) – the number of tokens each iteration during prefill, default to 8192
- **num_tokens_per_iter** (*int*) – the number of tokens processed in each forward pass. Working with *max_prefill_iters* enables "Dynamic SplitFuse"-like scheduling
- **max_prefill_iters** (*int*) – the max number of forward pass during prefill stage

25.5 GenerationConfig

class lmdeploy.**GenerationConfig**(*n: int = 1*, *max_new_tokens: int = 512*, *top_p: float = 1.0*, *top_k: int = 1*, *temperature: float = 0.8*, *repetition_penalty: float = 1.0*, *ignore_eos: bool = False*, *random_seed: Optional[int] = None*, *stop_words: Optional[List[str]] = None*, *bad_words: Optional[List[str]] = None*, *min_new_tokens: Optional[int] = None*, *skip_special_tokens: bool = True*, *logprobs: Optional[int] = None*)

generation parameters used by inference engines.

Parameters

- **n** (*int*) – Define how many chat completion choices to generate for each input message
- **max_new_tokens** (*int*) – The maximum number of tokens that can be generated in the chat completion
- **top_p** (*float*) – An alternative to sampling with temperature, called nucleus sampling, where the model considers the results of the tokens with top_p probability mass
- **top_k** (*int*) – An alternative to sampling with temperature, where the model considers the top_k tokens with the highest probability
- **temperature** (*float*) – Sampling temperature
- **repetition_penalty** (*float*) – Penalty to prevent the model from generating repeated words or phrases. A value larger than 1 discourages repetition
- **ignore_eos** (*bool*) – Indicator to ignore the eos_token_id or not

- **random_seed** (*int*) – Seed used when sampling a token
- **stop_words** (*List[str]*) – Words that stop generating further tokens
- **bad_words** (*List[str]*) – Words that the engine will never generate
- **min_new_tokens** (*int*) – The minimum numbers of tokens to generate, ignoring the number of tokens in the prompt.
- **skip_special_tokens** (*bool*) – Whether or not to remove special tokens in the decoding. Default to be True.
- **logprobs** (*int*) – Number of log probabilities to return per output token.

25.6 ChatTemplateConfig

class lmdeploy.**ChatTemplateConfig**(*model_name: str*, *system: Optional[str] = None*, *meta_instruction: Optional[str] = None*, *eosys: Optional[str] = None*, *user: Optional[str] = None*, *eoh: Optional[str] = None*, *assistant: Optional[str] = None*, *eoa: Optional[str] = None*, *separator: Optional[str] = None*, *capability: Optional[Literal[completion, infilling, chat, python]] = None*, *stop_words: Optional[List[str]] = None*)

Parameters for chat template.

Parameters

- **model_name** (*str*) – the name of the deployed model. Determine which chat template will be applied. All the chat template names: *lmdeploy list*
- **system** (*str | None*) – begin of the system prompt
- **meta_instruction** (*str | None*) – system prompt
- **eosys** (*str | None*) – end of the system prompt
- **user** (*str | None*) – begin of the user prompt
- **eoh** (*str | None*) – end of the user prompt
- **assistant** (*str | None*) – begin of the assistant prompt
- **eoa** (*str | None*) – end of the assistant prompt
- **capability** – (‘completion’ | ‘infilling’ | ‘chat’ | ‘python’) = None

CHAPTER

TWENTYSIX

INDICES AND TABLES

- genindex
- search

INDEX