vLLM

the vLLM Team

Jun 01, 2024

GETTING STARTED

vLLM is a fast and easy-to-use library for LLM inference and serving.

vLLM is fast with:

- State-of-the-art serving throughput
- Efficient management of attention key and value memory with **PagedAttention**
- Continuous batching of incoming requests
- Fast model execution with CUDA/HIP graph
- Quantization: GPTQ, AWQ, SqueezeLLM, FP8 KV Cache
- Optimized CUDA kernels

vLLM is flexible and easy to use with:

- Seamless integration with popular HuggingFace models
- High-throughput serving with various decoding algorithms, including *parallel sampling*, *beam search*, and more
- Tensor parallelism support for distributed inference
- Streaming outputs
- OpenAI-compatible API server
- Support NVIDIA GPUs and AMD GPUs
- (Experimental) Prefix caching support
- (Experimental) Multi-lora support

For more information, check out the following:

- vLLM announcing blog post (intro to PagedAttention)
- vLLM paper (SOSP 2023)
- How continuous batching enables 23x throughput in LLM inference while reducing p50 latency by Cade Daniel et al.
- *vLLM Meetups*.

CHAPTER

ONE

DOCUMENTATION

1.1 Installation

vLLM is a Python library that also contains pre-compiled C++ and CUDA (12.1) binaries.

1.1.1 Requirements

- OS: Linux
- Python: 3.8 – 3.11
- GPU: compute capability 7.0 or higher (e.g., V100, T4, RTX20xx, A100, L4, H100, etc.)

1.1.2 Install with pip

You can install vLLM using pip:

```
$ # (Recommended) Create a new conda environment.
$ conda create -n myenv python=3.9 -y
$ conda activate myenv

$ # Install vLLM with CUDA 12.1.
$ pip install vllm
```

Note: As of now, vLLM's binaries are compiled with CUDA 12.1 and public PyTorch release versions by default. We also provide vLLM binaries compiled with CUDA 11.8 and public PyTorch release versions:

```
$ # Install vLLM with CUDA 11.8.
$ export VLLM_VERSION=0.4.0
$ export PYTHON_VERSION=39
$ pip install https://github.com/vllm-project/vllm/releases/download/v${VLLM_VERSION}/
↪vllm-${VLLM_VERSION}+cu118-cp${PYTHON_VERSION}-cp${PYTHON_VERSION}-manylinux1_x86_64.
↪whl --extra-index-url https://download.pytorch.org/whl/cu118
```

In order to be performant, vLLM has to compile many cuda kernels. The compilation unfortunately introduces binary incompatibility with other CUDA versions and PyTorch versions, even for the same PyTorch version with different building configurations.

Therefore, it is recommended to install vLLM with a **fresh new** conda environment. If either you have a different CUDA version or you want to use an existing PyTorch installation, you need to build vLLM from source. See below for instructions.

1.1.3 Build from source

You can also build and install vLLM from source:

```
$ git clone https://github.com/vllm-project/vllm.git
$ cd vllm
$ # export VLLM_INSTALL_PUNICA_KERNELS=1 # optionally build for multi-LoRA capability
$ pip install -e .  # This may take 5-10 minutes.
```

Tip: Building from source requires quite a lot compilation. If you are building from source for multiple times, it is beneficial to cache the compilation results. For example, you can install ccache via either *conda install ccache* or *apt install ccache* . As long as *which ccache* command can find the *ccache* binary, it will be used automatically by the build system. After the first build, the subsequent builds will be much faster.

Tip: To avoid your system being overloaded, you can limit the number of compilation jobs to be run simultaneously, via the environment variable *MAX_JOBS*. For example:

```
$ export MAX_JOBS=6
$ pip install -e .
```

Tip: If you have trouble building vLLM, we recommend using the NVIDIA PyTorch Docker image.

```
$ # Use `--ipc=host` to make sure the shared memory is large enough.
$ docker run --gpus all -it --rm --ipc=host nvcr.io/nvidia/pytorch:23.10-py3
```

If you don't want to use docker, it is recommended to have a full installation of CUDA Toolkit. You can download and install it from the official website. After installation, set the environment variable *CUDA_HOME* to the installation path of CUDA Toolkit, and make sure that the *nvcc* compiler is in your *PATH*, e.g.:

```
$ export CUDA_HOME=/usr/local/cuda
$ export PATH="${CUDA_HOME}/bin:$PATH"
```

Here is a sanity check to verify that the CUDA Toolkit is correctly installed:

```
$ nvcc --version # verify that nvcc is in your PATH
$ ${CUDA_HOME}/bin/nvcc --version # verify that nvcc is in your CUDA_HOME
```

1.2 Installation with ROCm

vLLM supports AMD GPUs with ROCm 5.7 and 6.0.

1.2.1 Requirements

- OS: Linux
- Python: 3.8 – 3.11
- GPU: MI200s (gfx90a), MI300 (gfx942), Radeon RX 7900 series (gfx1100)
- ROCm 6.0 and ROCm 5.7

Installation options:

1. *Build from source with docker*
2. *Build from source*

1.2.2 Option 1: Build from source with docker (recommended)

You can build and install vLLM from source.

First, build a docker image from Dockerfile.rocm and launch a docker container from the image.

Dockerfile.rocm uses ROCm 6.0 by default, but also supports ROCm 5.7. It provides flexibility to customize the build of docker image using the following arguments:

- *BASE_IMAGE*: specifies the base image used when running `docker build`, specifically the PyTorch on ROCm base image. We have tested ROCm 5.7 and ROCm 6.0. The default is *rocm/pytorch:rocm6.0_ubuntu20.04_py3.9_pytorch_2.1.1*
- *BUILD_FA*: specifies whether to build CK flash-attention. The default is 1. For Radeon RX 7900 series (gfx1100), this should be set to 0 before flash-attention supports this target.
- *FX_GFX_ARCHS*: specifies the GFX architecture that is used to build CK flash-attention, for example, *gfx90a;gfx942* for MI200 and MI300. The default is *gfx90a;gfx942*
- *FA_BRANCH*: specifies the branch used to build the CK flash-attention in ROCm's flash-attention repo. The default is *ae7928c*
- *BUILD_TRITON*: specifies whether to build triton flash-attention. The default value is 1.

Their values can be passed in when running `docker build` with `--build-arg` options.

To build vllm on ROCm 6.0 for MI200 and MI300 series, you can use the default:

```
$ docker build -f Dockerfile.rocm -t vllm-rocm .
```

To build vllm on ROCm 6.0 for Radeon RX7900 series (gfx1100), you should specify `BUILD_FA` as below:

```
$ docker build --build-arg BUILD_FA="0" -f Dockerfile.rocm -t vllm-rocm .
```

To build docker image for vllm on ROCm 5.7, you can specify `BASE_IMAGE` as below:

```
$ docker build --build-arg BASE_IMAGE="rocm/pytorch:rocm5.7_ubuntu22.04_py3.10_pytorch_2.
↪0.1" \
  -f Dockerfile.rocm -t vllm-rocm .
```

To run the above docker image `vllm-rocm`, use the below command:

```
$ docker run -it \
   --network=host \
   --group-add=video \
   --ipc=host \
   --cap-add=SYS_PTRACE \
   --security-opt seccomp=unconfined \
   --device /dev/kfd \
   --device /dev/dri \
   -v <path/to/model>:/app/model \
   vllm-rocm \
   bash
```

Where the <*path/to/model*> is the location where the model is stored, for example, the weights for llama2 or llama3 models.

1.2.3 Option 2: Build from source

0. Install prerequisites (skip if you are already in an environment/docker with the following installed):

- ROCm
- Pytorch
- hipBLAS

For installing PyTorch, you can start from a fresh docker image, e.g, *rocm6.0.2_ubuntu22.04_py3.10_pytorch_2.1.2*, *rocm/pytorch:rocm6.0_ubuntu20.04_py3.9_pytorch_2.1.1*, *rocm/pytorch-nightly*.

Alternatively, you can install pytorch using pytorch wheels. You can check Pytorch installation guild in Pytorch Getting Started

For rocm6.0:

```
$ pip3 install torch --index-url https://download.pytorch.org/whl/rocm6.0
```

For rocm5.7:

```
$ pip install torch --index-url https://download.pytorch.org/whl/rocm5.7
```

1. Install Triton flash attention for ROCm

Install ROCm's Triton flash attention (the default triton-mlir branch) following the instructions from ROCm/triton

2. Optionally, if you choose to use CK flash attention, you can install flash attention for ROCm

Install ROCm's flash attention (v2.0.4) following the instructions from ROCm/flash-attention

Note:

- If you are using rocm5.7 with pytorch 2.1.0 onwards, you don't need to apply the *hipify_python.patch*. You can build the ROCm flash attention directly.
- If you fail to install *ROCm/flash-attention*, try cloning from the commit *6fd2f8e572805681cd67ef8596c7e2ce521ed3c6*.
- ROCm's Flash-attention-2 (v2.0.4) does not support sliding windows attention.

- You might need to downgrade the "ninja" version to 1.10 it is not used when compiling flash-attention-2 (e.g. *pip install ninja==1.10.2.4*)

3. Build vLLM.

```
$ cd vllm
$ pip install -U -r requirements-rocm.txt
$ python setup.py install # This may take 5-10 minutes. Currently, `pip install .`` does␣
↪not work for ROCm installation
```

Tip:

- You may need to turn on the `--enforce-eager` flag if you experience process hang when running the *benchmark_thoughput.py* script to test your installation.
- Triton flash attention is used by default. For benchmarking purposes, it is recommended to run a warm up step before collecting perf numbers.
- To use CK flash-attention, please use this flag `export VLLM_USE_FLASH_ATTN_TRITON=0` to turn off triton flash attention.
- The ROCm version of pytorch, ideally, should match the ROCm driver version.

1.3 Installation with Neuron

vLLM 0.3.3 onwards supports model inferencing and serving on AWS Trainium/Inferentia with Neuron SDK. At the moment Paged Attention is not supported in Neuron SDK, but naive continuous batching is supported in transformers-neuronx. Data types currently supported in Neuron SDK are FP16 and BF16.

1.3.1 Requirements

- OS: Linux
- Python: 3.8 – 3.11
- Accelerator: NeuronCore_v2 (in trn1/inf2 instances)
- Pytorch 2.0.1/2.1.1
- AWS Neuron SDK 2.16/2.17 (Verified on python 3.8)

Installation steps:

- *Build from source*
  - *Step 0. Launch Trn1/Inf2 instances*
  - *Step 1. Install drivers and tools*
  - *Step 2. Install transformers-neuronx and its dependencies*
  - *Step 3. Install vLLM from source*

1.3.2 Build from source

Following instructions are applicable to Neuron SDK 2.16 and beyond.

Step 0. Launch Trn1/Inf2 instances

Here are the steps to launch trn1/inf2 instances, in order to install PyTorch Neuron ("torch-neuronx") Setup on Ubuntu 22.04 LTS.

- Please follow the instructions at launch an Amazon EC2 Instance to launch an instance. When choosing the instance type at the EC2 console, please make sure to select the correct instance type.
- To get more information about instances sizes and pricing see: Trn1 web page, Inf2 web page
- Select Ubuntu Server 22.04 TLS AMI
- When launching a Trn1/Inf2, please adjust your primary EBS volume size to a minimum of 512GB.
- After launching the instance, follow the instructions in Connect to your instance to connect to the instance

Step 1. Install drivers and tools

The installation of drivers and tools wouldn't be necessary, if Deep Learning AMI Neuron is installed. In case the drivers and tools are not installed on the operating system, follow the steps below:

```
# Configure Linux for Neuron repository updates
. /etc/os-release
sudo tee /etc/apt/sources.list.d/neuron.list > /dev/null <<EOF
deb https://apt.repos.neuron.amazonaws.com ${VERSION_CODENAME} main
EOF
wget -qO - https://apt.repos.neuron.amazonaws.com/GPG-PUB-KEY-AMAZON-AWS-NEURON.PUB |␣
↪sudo apt-key add -

# Update OS packages
sudo apt-get update -y

# Install OS headers
sudo apt-get install linux-headers-$(uname -r) -y

# Install git
sudo apt-get install git -y

# install Neuron Driver
sudo apt-get install aws-neuronx-dkms=2.* -y

# Install Neuron Runtime
sudo apt-get install aws-neuronx-collectives=2.* -y
sudo apt-get install aws-neuronx-runtime-lib=2.* -y

# Install Neuron Tools
sudo apt-get install aws-neuronx-tools=2.* -y

# Add PATH
export PATH=/opt/aws/neuron/bin:$PATH
```

Step 2. Install transformers-neuronx and its dependencies

transformers-neuronx will be the backend to support inference on trn1/inf2 instances. Follow the steps below to install transformer-neuronx package and its dependencies.

```
# Install Python venv
sudo apt-get install -y python3.10-venv g++

# Create Python venv
python3.10 -m venv aws_neuron_venv_pytorch

# Activate Python venv
source aws_neuron_venv_pytorch/bin/activate

# Install Jupyter notebook kernel
pip install ipykernel
python3.10 -m ipykernel install --user --name aws_neuron_venv_pytorch --display-name
↪"Python (torch-neuronx)"
pip install jupyter notebook
pip install environment_kernels

# Set pip repository pointing to the Neuron repository
python -m pip config set global.extra-index-url https://pip.repos.neuron.amazonaws.com

# Install wget, awscli
python -m pip install wget
python -m pip install awscli

# Update Neuron Compiler and Framework
python -m pip install --upgrade neuronx-cc==2.* --pre torch-neuronx==2.1.* torchvision␣
↪transformers-neuronx
```

Step 3. Install vLLM from source

Once neuronx-cc and transformers-neuronx packages are installed, we will be able to install vllm as follows:

```
$ git clone https://github.com/vllm-project/vllm.git
$ cd vllm
$ pip install -U -r requirements-neuron.txt
$ pip install .
```

If neuron packages are detected correctly in the installation process, `vllm-0.3.0+neuron212` will be installed.

1.4 Installation with CPU

vLLM initially supports basic model inferencing and serving on x86 CPU platform, with data types FP32 and BF16.

Table of contents:

1.4.1 Requirements

- OS: Linux
- Compiler: gcc/g++>=12.3.0 (recommended)
- Instruction set architecture (ISA) requirement: AVX512 is required.

1.4.2 Quick start using Dockerfile

```
$ docker build -f Dockerfile.cpu -t vllm-cpu-env --shm-size=4g .
$ docker run -it \
             --rm \
             --network=host \
             --cpuset-cpus=<cpu-id-list, optional> \
             --cpuset-mems=<memory-node, optional> \
             vllm-cpu-env
```

1.4.3 Build from source

- First, install required compiler. We recommend to use `gcc/g++ >= 12.3.0` as the default compiler to avoid potential problems. For example, on Ubuntu 22.4, you can run:

```
$ sudo apt-get update  -y
$ sudo apt-get install -y gcc-12 g++-12
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-12 10 --slave /usr/
↪bin/g++ g++ /usr/bin/g++-12
```

- Second, install Python packages for vLLM CPU backend building:

```
$ pip install --upgrade pip
$ pip install wheel packaging ninja setuptools>=49.4.0 numpy
$ pip install -v -r requirements-cpu.txt --extra-index-url https://download.pytorch.org/
↪whl/cpu
```

- Finally, build and install vLLM CPU backend:

```
$ VLLM_TARGET_DEVICE=cpu python setup.py install
```

Note:

- BF16 is the default data type in the current CPU backend (that means the backend will cast FP16 to BF16), and is compatible will all CPUs with AVX512 ISA support.
- AVX512_BF16 is an extension ISA provides native BF16 data type conversion and vector product instructions, will brings some performance improvement compared with pure AVX512. The CPU backend build script will check the host CPU flags to determine whether to enable AVX512_BF16.
- If you want to force enable AVX512_BF16 for the cross-compilation, please set environment variable VLLM_CPU_AVX512BF16=1 before the building.

1.4.4 Performance tips

- vLLM CPU backend uses environment variable `VLLM_CPU_KVCACHE_SPACE` to specify the KV Cache size (e.g, `VLLM_CPU_KVCACHE_SPACE=40` means 40 GB space for KV cache), larger setting will allow vLLM running more requests in parallel. This parameter should be set based on the hardware configuration and memory management pattern of users.
- vLLM CPU backend uses OpenMP for thread-parallel computation. If you want the best performance on CPU, it will be very critical to isolate CPU cores for OpenMP threads with other thread pools (like web-service event-loop), to avoid CPU oversubscription.
- If using vLLM CPU backend on a bare-metal machine, it is recommended to disable the hyper-threading.
- If using vLLM CPU backend on a multi-socket machine with NUMA, be aware to set CPU cores and memory nodes, to avoid the remote memory node access. `numactl` is an useful tool for CPU core and memory binding on NUMA platform. Besides, `--cpuset-cpus` and `--cpuset-mems` arguments of `docker run` are also useful.

1.5 Quickstart

This guide shows how to use vLLM to:

- run offline batched inference on a dataset;
- build an API server for a large language model;
- start an OpenAI-compatible API server.

Be sure to complete the *installation instructions* before continuing with this guide.

Note: By default, vLLM downloads model from HuggingFace. If you would like to use models from ModelScope in the following examples, please set the environment variable:

```
export VLLM_USE_MODELSCOPE=True
```

1.5.1 Offline Batched Inference

We first show an example of using vLLM for offline batched inference on a dataset. In other words, we use vLLM to generate texts for a list of input prompts.

Import `LLM` and `SamplingParams` from vLLM. The `LLM` class is the main class for running offline inference with vLLM engine. The `SamplingParams` class specifies the parameters for the sampling process.

```
from vllm import LLM, SamplingParams
```

Define the list of input prompts and the sampling parameters for generation. The sampling temperature is set to 0.8 and the nucleus sampling probability is set to 0.95. For more information about the sampling parameters, refer to the class definition.

```
prompts = [
    "Hello, my name is",
    "The president of the United States is",
    "The capital of France is",
    "The future of AI is",
]
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)
```

Initialize vLLM's engine for offline inference with the `LLM` class and the OPT-125M model. The list of supported models can be found at *supported models*.

```
llm = LLM(model="facebook/opt-125m")
```

Call `llm.generate` to generate the outputs. It adds the input prompts to vLLM engine's waiting queue and executes the vLLM engine to generate the outputs with high throughput. The outputs are returned as a list of `RequestOutput` objects, which include all the output tokens.

```
outputs = llm.generate(prompts, sampling_params)

# Print the outputs.
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")
```

The code example can also be found in examples/offline_inference.py.

1.5.2 OpenAI-Compatible Server

vLLM can be deployed as a server that implements the OpenAI API protocol. This allows vLLM to be used as a drop-in replacement for applications using OpenAI API. By default, it starts the server at `http://localhost:8000`. You can specify the address with `--host` and `--port` arguments. The server currently hosts one model at a time (OPT-125M in the command below) and implements list models, create chat completion, and create completion endpoints. We are actively adding support for more endpoints.

Start the server:

```
$ python -m vllm.entrypoints.openai.api_server \
$     --model facebook/opt-125m
```

By default, the server uses a predefined chat template stored in the tokenizer. You can override this template by using the `--chat-template` argument:

```
$ python -m vllm.entrypoints.openai.api_server \
$     --model facebook/opt-125m \
$     --chat-template ./examples/template_chatml.jinja
```

This server can be queried in the same format as OpenAI API. For example, list the models:

```
$ curl http://localhost:8000/v1/models
```

You can pass in the argument `--api-key` or environment variable `VLLM_API_KEY` to enable the server to check for API key in the header.

Using OpenAI Completions API with vLLM

Query the model with input prompts:

```
$ curl http://localhost:8000/v1/completions \
$     -H "Content-Type: application/json" \
$     -d '{
$         "model": "facebook/opt-125m",
$         "prompt": "San Francisco is a",
$         "max_tokens": 7,
$         "temperature": 0
$     }'
```

Since this server is compatible with OpenAI API, you can use it as a drop-in replacement for any applications using OpenAI API. For example, another way to query the server is via the `openai` python package:

```python
from openai import OpenAI

# Modify OpenAI's API key and API base to use vLLM's API server.
openai_api_key = "EMPTY"
openai_api_base = "http://localhost:8000/v1"
client = OpenAI(
    api_key=openai_api_key,
    base_url=openai_api_base,
)
completion = client.completions.create(model="facebook/opt-125m",
                                      prompt="San Francisco is a")
print("Completion result:", completion)
```

For a more detailed client example, refer to examples/openai_completion_client.py.

Using OpenAI Chat API with vLLM

The vLLM server is designed to support the OpenAI Chat API, allowing you to engage in dynamic conversations with the model. The chat interface is a more interactive way to communicate with the model, allowing back-and-forth exchanges that can be stored in the chat history. This is useful for tasks that require context or more detailed explanations.

Querying the model using OpenAI Chat API:

You can use the create chat completion endpoint to communicate with the model in a chat-like interface:

```
$ curl http://localhost:8000/v1/chat/completions \
$     -H "Content-Type: application/json" \
$     -d '{
$         "model": "facebook/opt-125m",
$         "messages": [
$             {"role": "system", "content": "You are a helpful assistant."},
$             {"role": "user", "content": "Who won the world series in 2020?"}
$         ]
$     }'
```

Python Client Example:

Using the *openai* python package, you can also communicate with the model in a chat-like manner:

```
from openai import OpenAI
# Set OpenAI's API key and API base to use vLLM's API server.
openai_api_key = "EMPTY"
openai_api_base = "http://localhost:8000/v1"

client = OpenAI(
    api_key=openai_api_key,
    base_url=openai_api_base,
)

chat_response = client.chat.completions.create(
    model="facebook/opt-125m",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Tell me a joke."},
    ]
)
print("Chat response:", chat_response)
```

For more in-depth examples and advanced features of the chat API, you can refer to the official OpenAI documentation.

1.6 Examples

1.6.1 API Client

Source https://github.com/vllm-project/vllm/blob/main/examples/api_client.py.

```python
"""Example Python client for vllm.entrypoints.api_server"""

import argparse
import json
from typing import Iterable, List

import requests


def clear_line(n: int = 1) -> None:
    LINE_UP = '\033[1A'
    LINE_CLEAR = '\x1b[2K'
    for _ in range(n):
        print(LINE_UP, end=LINE_CLEAR, flush=True)


def post_http_request(prompt: str,
                      api_url: str,
                      n: int = 1,
                      stream: bool = False) -> requests.Response:
    headers = {"User-Agent": "Test Client"}
    pload = {
        "prompt": prompt,
        "n": n,
        "use_beam_search": True,
        "temperature": 0.0,
        "max_tokens": 16,
        "stream": stream,
    }
    response = requests.post(api_url, headers=headers, json=pload, stream=True)
    return response


def get_streaming_response(response: requests.Response) -> Iterable[List[str]]:
    for chunk in response.iter_lines(chunk_size=8192,
                                     decode_unicode=False,
                                     delimiter=b"\0"):
        if chunk:
            data = json.loads(chunk.decode("utf-8"))
            output = data["text"]
            yield output


def get_response(response: requests.Response) -> List[str]:
    data = json.loads(response.content)
    output = data["text"]
```

(continues on next page)

(continued from previous page)

```python
    return output


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--host", type=str, default="localhost")
    parser.add_argument("--port", type=int, default=8000)
    parser.add_argument("--n", type=int, default=4)
    parser.add_argument("--prompt", type=str, default="San Francisco is a")
    parser.add_argument("--stream", action="store_true")
    args = parser.parse_args()
    prompt = args.prompt
    api_url = f"http://{args.host}:{args.port}/generate"
    n = args.n
    stream = args.stream

    print(f"Prompt: {prompt!r}\n", flush=True)
    response = post_http_request(prompt, api_url, n, stream)

    if stream:
        num_printed_lines = 0
        for h in get_streaming_response(response):
            clear_line(num_printed_lines)
            num_printed_lines = 0
            for i, line in enumerate(h):
                num_printed_lines += 1
                print(f"Beam candidate {i}: {line!r}", flush=True)
    else:
        output = get_response(response)
        for i, line in enumerate(output):
            print(f"Beam candidate {i}: {line!r}", flush=True)
```

1.6.2 Aqlm Example

Source https://github.com/vllm-project/vllm/blob/main/examples/aqlm_example.py.

```python
import argparse

from vllm import LLM, SamplingParams


def main():

    parser = argparse.ArgumentParser(description='AQLM examples')

    parser.add_argument('--model',
                        '-m',
                        type=str,
                        default=None,
                        help='model path, as for HF')
    parser.add_argument('--choice',
```

(continues on next page)

(continued from previous page)

```python
                        '-c',
                        type=int,
                        default=0,
                        help='known good models by index, [0-4]')
    parser.add_argument('--tensor_parallel_size',
                        '-t',
                        type=int,
                        default=1,
                        help='tensor parallel size')

    args = parser.parse_args()

    models = [
        "ISTA-DASLab/Llama-2-7b-AQLM-2Bit-1x16-hf",
        "ISTA-DASLab/Llama-2-7b-AQLM-2Bit-2x8-hf",
        "ISTA-DASLab/Llama-2-13b-AQLM-2Bit-1x16-hf",
        "ISTA-DASLab/Mixtral-8x7b-AQLM-2Bit-1x16-hf",
        "BlackSamorez/TinyLlama-1_1B-Chat-v1_0-AQLM-2Bit-1x16-hf",
    ]

    model = LLM(args.model if args.model is not None else models[args.choice],
                tensor_parallel_size=args.tensor_parallel_size)

    sampling_params = SamplingParams(max_tokens=100, temperature=0)
    outputs = model.generate("Hello my name is",
                             sampling_params=sampling_params)
    print(outputs[0].outputs[0].text)


if __name__ == '__main__':
    main()
```

1.6.3 Gradio OpenAI Chatbot Webserver

Source https://github.com/vllm-project/vllm/blob/main/examples/gradio_openai_chatbot_webserver.py.

```python
import argparse

import gradio as gr
from openai import OpenAI

# Argument parser setup
parser = argparse.ArgumentParser(
    description='Chatbot Interface with Customizable Parameters')
parser.add_argument('--model-url',
                    type=str,
                    default='http://localhost:8000/v1',
                    help='Model URL')
parser.add_argument('-m',
                    '--model',
                    type=str,
```

(continues on next page)

(continued from previous page)

```
                    required=True,
                    help='Model name for the chatbot')
parser.add_argument('--temp',
                    type=float,
                    default=0.8,
                    help='Temperature for text generation')
parser.add_argument('--stop-token-ids',
                    type=str,
                    default='',
                    help='Comma-separated stop token IDs')
parser.add_argument("--host", type=str, default=None)
parser.add_argument("--port", type=int, default=8001)

# Parse the arguments
args = parser.parse_args()

# Set OpenAI's API key and API base to use vLLM's API server.
openai_api_key = "EMPTY"
openai_api_base = args.model_url

# Create an OpenAI client to interact with the API server
client = OpenAI(
    api_key=openai_api_key,
    base_url=openai_api_base,
)


def predict(message, history):
    # Convert chat history to OpenAI format
    history_openai_format = [{
        "role": "system",
        "content": "You are a great ai assistant."
    }]
    for human, assistant in history:
        history_openai_format.append({"role": "user", "content": human})
        history_openai_format.append({
            "role": "assistant",
            "content": assistant
        })
    history_openai_format.append({"role": "user", "content": message})

    # Create a chat completion request and send it to the API server
    stream = client.chat.completions.create(
        model=args.model,  # Model name to use
        messages=history_openai_format,  # Chat history
        temperature=args.temp,  # Temperature for text generation
        stream=True,  # Stream response
        extra_body={
            'repetition_penalty':
            1,
            'stop_token_ids': [
                int(id.strip()) for id in args.stop_token_ids.split(',')
```

(continues on next page)

(continued from previous page)

```python
                if id.strip()
            ] if args.stop_token_ids else []
        })

    # Read and return generated text from response stream
    partial_message = ""
    for chunk in stream:
        partial_message += (chunk.choices[0].delta.content or "")
        yield partial_message


# Create and launch a chat interface with Gradio
gr.ChatInterface(predict).queue().launch(server_name=args.host,
                                         server_port=args.port,
                                         share=True)
```

1.6.4 Gradio Webserver

Source https://github.com/vllm-project/vllm/blob/main/examples/gradio_webserver.py.

```python
import argparse
import json

import gradio as gr
import requests


def http_bot(prompt):
    headers = {"User-Agent": "vLLM Client"}
    pload = {
        "prompt": prompt,
        "stream": True,
        "max_tokens": 128,
    }
    response = requests.post(args.model_url,
                             headers=headers,
                             json=pload,
                             stream=True)

    for chunk in response.iter_lines(chunk_size=8192,
                                     decode_unicode=False,
                                     delimiter=b"\0"):
        if chunk:
            data = json.loads(chunk.decode("utf-8"))
            output = data["text"][0]
            yield output


def build_demo():
    with gr.Blocks() as demo:
        gr.Markdown("# vLLM text completion demo\n")
```

(continues on next page)

(continued from previous page)

```python
        inputbox = gr.Textbox(label="Input",
                              placeholder="Enter text and press ENTER")
        outputbox = gr.Textbox(label="Output",
                               placeholder="Generated result from the model")
        inputbox.submit(http_bot, [inputbox], [outputbox])
    return demo


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--host", type=str, default=None)
    parser.add_argument("--port", type=int, default=8001)
    parser.add_argument("--model-url",
                        type=str,
                        default="http://localhost:8000/generate")
    args = parser.parse_args()

    demo = build_demo()
    demo.queue().launch(server_name=args.host,
                        server_port=args.port,
                        share=True)
```

1.6.5 Llava Example

Source https://github.com/vllm-project/vllm/blob/main/examples/llava_example.py.

```python
import argparse
import os
import subprocess

import torch

from vllm import LLM
from vllm.sequence import MultiModalData

# The assets are located at `s3://air-example-data-2/vllm_opensource_llava/`.


def run_llava_pixel_values():
    llm = LLM(
        model="llava-hf/llava-1.5-7b-hf",
        image_input_type="pixel_values",
        image_token_id=32000,
        image_input_shape="1,3,336,336",
        image_feature_size=576,
    )

    prompt = "<image>" * 576 + (
        "\nUSER: What is the content of this image?\nASSISTANT:")

    # This should be provided by another online or offline component.
```

(continues on next page)

(continued from previous page)

```
    image = torch.load("images/stop_sign_pixel_values.pt")

    outputs = llm.generate({
        "prompt":
        prompt,
        "multi_modal_data":
        MultiModalData(type=MultiModalData.Type.IMAGE, data=image),
    })

    for o in outputs:
        generated_text = o.outputs[0].text
        print(generated_text)


def run_llava_image_features():
    llm = LLM(
        model="llava-hf/llava-1.5-7b-hf",
        image_input_type="image_features",
        image_token_id=32000,
        image_input_shape="1,576,1024",
        image_feature_size=576,
    )

    prompt = "<image>" * 576 + (
        "\nUSER: What is the content of this image?\nASSISTANT:")

    # This should be provided by another online or offline component.
    image = torch.load("images/stop_sign_image_features.pt")

    outputs = llm.generate({
        "prompt":
        prompt,
        "multi_modal_data":
        MultiModalData(type=MultiModalData.Type.IMAGE, data=image),
    })
    for o in outputs:
        generated_text = o.outputs[0].text
        print(generated_text)


def main(args):
    if args.type == "pixel_values":
        run_llava_pixel_values()
    else:
        run_llava_image_features()


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Demo on Llava")
    parser.add_argument("--type",
                        type=str,
                        choices=["pixel_values", "image_features"],
```

(continues on next page)

(continued from previous page)

```
                        default="pixel_values",
                        help="image input type")
    args = parser.parse_args()
    # Download from s3
    s3_bucket_path = "s3://air-example-data-2/vllm_opensource_llava/"
    local_directory = "images"

    # Make sure the local directory exists or create it
    os.makedirs(local_directory, exist_ok=True)

    # Use AWS CLI to sync the directory, assume anonymous access
    subprocess.check_call([
        "aws",
        "s3",
        "sync",
        s3_bucket_path,
        local_directory,
        "--no-sign-request",
    ])
    main(args)
```

1.6.6 LLM Engine Example

Source https://github.com/vllm-project/vllm/blob/main/examples/llm_engine_example.py.

```
import argparse
from typing import List, Tuple

from vllm import EngineArgs, LLMEngine, RequestOutput, SamplingParams


def create_test_prompts() -> List[Tuple[str, SamplingParams]]:
    """Create a list of test prompts with their sampling parameters."""
    return [
        ("A robot may not injure a human being",
         SamplingParams(temperature=0.0, logprobs=1, prompt_logprobs=1)),
        ("To be or not to be,",
         SamplingParams(temperature=0.8, top_k=5, presence_penalty=0.2)),
        ("What is the meaning of life?",
         SamplingParams(n=2,
                        best_of=5,
                        temperature=0.8,
                        top_p=0.95,
                        frequency_penalty=0.1)),
        ("It is only with the heart that one can see rightly",
         SamplingParams(n=3, best_of=3, use_beam_search=True,
                        temperature=0.0)),
    ]


def process_requests(engine: LLMEngine,
```

(continues on next page)

(continued from previous page)

```python
                        test_prompts: List[Tuple[str, SamplingParams]]):
    """Continuously process a list of prompts and handle the outputs."""
    request_id = 0

    while test_prompts or engine.has_unfinished_requests():
        if test_prompts:
            prompt, sampling_params = test_prompts.pop(0)
            engine.add_request(str(request_id), prompt, sampling_params)
            request_id += 1

        request_outputs: List[RequestOutput] = engine.step()

        for request_output in request_outputs:
            if request_output.finished:
                print(request_output)


def initialize_engine(args: argparse.Namespace) -> LLMEngine:
    """Initialize the LLMEngine from the command line arguments."""
    engine_args = EngineArgs.from_cli_args(args)
    return LLMEngine.from_engine_args(engine_args)


def main(args: argparse.Namespace):
    """Main function that sets up and runs the prompt processing."""
    engine = initialize_engine(args)
    test_prompts = create_test_prompts()
    process_requests(engine, test_prompts)


if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description='Demo on using the LLMEngine class directly')
    parser = EngineArgs.add_cli_args(parser)
    args = parser.parse_args()
    main(args)
```

1.6.7 MultiLoRA Inference

Source https://github.com/vllm-project/vllm/blob/main/examples/multilora_inference.py.

```python
"""
This example shows how to use the multi-LoRA functionality
for offline inference.

Requires HuggingFace credentials for access to Llama2.
"""

from typing import List, Optional, Tuple

from huggingface_hub import snapshot_download
```

(continues on next page)

(continued from previous page)

```python

from vllm import EngineArgs, LLMEngine, RequestOutput, SamplingParams
from vllm.lora.request import LoRARequest


def create_test_prompts(
        lora_path: str
) -> List[Tuple[str, SamplingParams, Optional[LoRARequest]]]:
    """Create a list of test prompts with their sampling parameters.

    2 requests for base model, 4 requests for the LoRA. We define 2
    different LoRA adapters (using the same model for demo purposes).
    Since we also set `max_loras=1`, the expectation is that the requests
    with the second LoRA adapter will be ran after all requests with the
    first adapter have finished.
    """
    return [
        ("A robot may not injure a human being",
         SamplingParams(temperature=0.0,
                        logprobs=1,
                        prompt_logprobs=1,
                        max_tokens=128), None),
        ("To be or not to be,",
         SamplingParams(temperature=0.8,
                        top_k=5,
                        presence_penalty=0.2,
                        max_tokens=128), None),
        (
            "[user] Write a SQL query to answer the question based on the table schema.\
↪n\n context: CREATE TABLE table_name_74 (icao VARCHAR, airport VARCHAR)\n\n question:␣
↪Name the ICAO for lilongwe international airport [/user] [assistant]",  # noqa: E501
            SamplingParams(temperature=0.0,
                           logprobs=1,
                           prompt_logprobs=1,
                           max_tokens=128,
                           stop_token_ids=[32003]),
            LoRARequest("sql-lora", 1, lora_path)),
        (
            "[user] Write a SQL query to answer the question based on the table schema.\
↪n\n context: CREATE TABLE table_name_11 (nationality VARCHAR, elector VARCHAR)\n\n␣
↪question: When Anchero Pantaleone was the elector what is under nationality? [/user]␣
↪[assistant]",  # noqa: E501
            SamplingParams(n=3,
                           best_of=3,
                           use_beam_search=True,
                           temperature=0,
                           max_tokens=128,
                           stop_token_ids=[32003]),
            LoRARequest("sql-lora", 1, lora_path)),
        (
            "[user] Write a SQL query to answer the question based on the table schema.\
↪n\n context: CREATE TABLE table_name_74 (icao VARCHAR, airport VARCHAR)\n\n question:␣
```

(continues on next page)

(continued from previous page)

```
↪Name the ICAO for lilongwe international airport [/user] [assistant]",  # noqa: E501
            SamplingParams(temperature=0.0,
                           logprobs=1,
                           prompt_logprobs=1,
                           max_tokens=128,
                           stop_token_ids=[32003]),
            LoRARequest("sql-lora2", 2, lora_path)),
        (
            "[user] Write a SQL query to answer the question based on the table schema.\
↪n\n context: CREATE TABLE table_name_11 (nationality VARCHAR, elector VARCHAR)\n\n␣
↪question: When Anchero Pantaleone was the elector what is under nationality? [/user]␣
↪[assistant]",  # noqa: E501
            SamplingParams(n=3,
                           best_of=3,
                           use_beam_search=True,
                           temperature=0,
                           max_tokens=128,
                           stop_token_ids=[32003]),
            LoRARequest("sql-lora", 1, lora_path)),
    ]


def process_requests(engine: LLMEngine,
                     test_prompts: List[Tuple[str, SamplingParams,
                                              Optional[LoRARequest]]]):
    """Continuously process a list of prompts and handle the outputs."""
    request_id = 0

    while test_prompts or engine.has_unfinished_requests():
        if test_prompts:
            prompt, sampling_params, lora_request = test_prompts.pop(0)
            engine.add_request(str(request_id),
                               prompt,
                               sampling_params,
                               lora_request=lora_request)
            request_id += 1

        request_outputs: List[RequestOutput] = engine.step()

        for request_output in request_outputs:
            if request_output.finished:
                print(request_output)


def initialize_engine() -> LLMEngine:
    """Initialize the LLMEngine."""
    # max_loras: controls the number of LoRAs that can be used in the same
    #   batch. Larger numbers will cause higher memory usage, as each LoRA
    #   slot requires its own preallocated tensor.
    # max_lora_rank: controls the maximum supported rank of all LoRAs. Larger
    #   numbers will cause higher memory usage. If you know that all LoRAs will
    #   use the same rank, it is recommended to set this as low as possible.
```

(continues on next page)

(continued from previous page)

```python
    # max_cpu_loras: controls the size of the CPU LoRA cache.
    engine_args = EngineArgs(model="meta-llama/Llama-2-7b-hf",
                             enable_lora=True,
                             max_loras=1,
                             max_lora_rank=8,
                             max_cpu_loras=2,
                             max_num_seqs=256)
    return LLMEngine.from_engine_args(engine_args)


def main():
    """Main function that sets up and runs the prompt processing."""
    engine = initialize_engine()
    lora_path = snapshot_download(repo_id="yard1/llama-2-7b-sql-lora-test")
    test_prompts = create_test_prompts(lora_path)
    process_requests(engine, test_prompts)


if __name__ == '__main__':
    main()
```

1.6.8 Offline Inference

Source https://github.com/vllm-project/vllm/blob/main/examples/offline_inference.py.

```python
from vllm import LLM, SamplingParams

# Sample prompts.
prompts = [
    "Hello, my name is",
    "The president of the United States is",
    "The capital of France is",
    "The future of AI is",
]
# Create a sampling params object.
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)

# Create an LLM.
llm = LLM(model="facebook/opt-125m")
# Generate texts from the prompts. The output is a list of RequestOutput objects
# that contain the prompt, generated text, and other information.
outputs = llm.generate(prompts, sampling_params)
# Print the outputs.
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")
```

1.6.9 Offline Inference Arctic

Source https://github.com/vllm-project/vllm/blob/main/examples/offline_inference_arctic.py.

```python
from vllm import LLM, SamplingParams

# Sample prompts.
prompts = [
    "Hello, my name is",
    "The president of the United States is",
    "The capital of France is",
    "The future of AI is",
]
# Create a sampling params object.
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)

# Create an LLM.
llm = LLM(model="snowflake/snowflake-arctic-instruct",
          quantization="deepspeedfp",
          tensor_parallel_size=8,
          trust_remote_code=True)
# Generate texts from the prompts. The output is a list of RequestOutput objects
# that contain the prompt, generated text, and other information.

outputs = llm.generate(prompts, sampling_params)
# Print the outputs.
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")
```

1.6.10 Offline Inference Distributed

Source https://github.com/vllm-project/vllm/blob/main/examples/offline_inference_distributed.py.

```python
"""
This example shows how to use Ray Data for running offline batch inference
distributively on a multi-nodes cluster.

Learn more about Ray Data in https://docs.ray.io/en/latest/data/data.html
"""

from typing import Dict

import numpy as np
import ray
from packaging.version import Version
from ray.util.scheduling_strategies import PlacementGroupSchedulingStrategy

from vllm import LLM, SamplingParams

assert Version(ray.__version__) >= Version(
```

(continues on next page)

(continued from previous page)

```python
    "2.22.0"), "Ray version must be at least 2.22.0"

# Create a sampling params object.
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)

# Set tensor parallelism per instance.
tensor_parallel_size = 1

# Set number of instances. Each instance will use tensor_parallel_size GPUs.
num_instances = 1


# Create a class to do batch inference.
class LLMPredictor:

    def __init__(self):
        # Create an LLM.
        self.llm = LLM(model="meta-llama/Llama-2-7b-chat-hf",
                       tensor_parallel_size=tensor_parallel_size)

    def __call__(self, batch: Dict[str, np.ndarray]) -> Dict[str, list]:
        # Generate texts from the prompts.
        # The output is a list of RequestOutput objects that contain the prompt,
        # generated text, and other information.
        outputs = self.llm.generate(batch["text"], sampling_params)
        prompt = []
        generated_text = []
        for output in outputs:
            prompt.append(output.prompt)
            generated_text.append(' '.join([o.text for o in output.outputs]))
        return {
            "prompt": prompt,
            "generated_text": generated_text,
        }


# Read one text file from S3. Ray Data supports reading multiple files
# from cloud storage (such as JSONL, Parquet, CSV, binary format).
ds = ray.data.read_text("s3://anonymous@air-example-data/prompts.txt")


# For tensor_parallel_size > 1, we need to create placement groups for vLLM
# to use. Every actor has to have its own placement group.
def scheduling_strategy_fn():
    # One bundle per tensor parallel worker
    pg = ray.util.placement_group(
        [{
            "GPU": 1,
            "CPU": 1
        }] * tensor_parallel_size,
        strategy="STRICT_PACK",
    )
```

(continues on next page)

(continued from previous page)

```
    return dict(scheduling_strategy=PlacementGroupSchedulingStrategy(
        pg, placement_group_capture_child_tasks=True))


resources_kwarg = {}
if tensor_parallel_size == 1:
    # For tensor_parallel_size == 1, we simply set num_gpus=1.
    resources_kwarg["num_gpus"] = 1
else:
    # Otherwise, we have to set num_gpus=0 and provide
    # a function that will create a placement group for
    # each instance.
    resources_kwarg["num_gpus"] = 0
    resources_kwarg["ray_remote_args_fn"] = scheduling_strategy_fn

# Apply batch inference for all input data.
ds = ds.map_batches(
    LLMPredictor,
    # Set the concurrency to the number of LLM instances.
    concurrency=num_instances,
    # Specify the batch size for inference.
    batch_size=32,
    **resources_kwarg,
)

# Peek first 10 results.
# NOTE: This is for local testing and debugging. For production use case,
# one should write full result out as shown below.
outputs = ds.take(limit=10)
for output in outputs:
    prompt = output["prompt"]
    generated_text = output["generated_text"]
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")

# Write inference output data out as Parquet files to S3.
# Multiple files would be written to the output destination,
# and each task would write one or more files separately.
#
# ds.write_parquet("s3://<your-output-bucket>")
```

1.6.11 Offline Inference Embedding

Source https://github.com/vllm-project/vllm/blob/main/examples/offline_inference_embedding.py.

```
from vllm import LLM

# Sample prompts.
prompts = [
    "Hello, my name is",
    "The president of the United States is",
    "The capital of France is",
```

(continues on next page)

(continued from previous page)

```python
    "The future of AI is",
]

# Create an LLM.
model = LLM(model="intfloat/e5-mistral-7b-instruct", enforce_eager=True)
# Generate embedding. The output is a list of EmbeddingRequestOutputs.
outputs = model.encode(prompts)
# Print the outputs.
for output in outputs:
    print(output.outputs.embedding)  # list of 4096 floats
```

1.6.12 Offline Inference Neuron

Source https://github.com/vllm-project/vllm/blob/main/examples/offline_inference_neuron.py.

```python
from vllm import LLM, SamplingParams

# Sample prompts.
prompts = [
    "Hello, my name is",
    "The president of the United States is",
    "The capital of France is",
    "The future of AI is",
]
# Create a sampling params object.
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)

# Create an LLM.
llm = LLM(
    model="TinyLlama/TinyLlama-1.1B-Chat-v1.0",
    max_num_seqs=8,
    # The max_model_len and block_size arguments are required to be same as
    # max sequence length when targeting neuron device.
    # Currently, this is a known limitation in continuous batching support
    # in transformers-neuronx.
    # TODO(liangfu): Support paged-attention in transformers-neuronx.
    max_model_len=128,
    block_size=128,
    # The device can be automatically detected when AWS Neuron SDK is installed.
    # The device argument can be either unspecified for automated detection,
    # or explicitly assigned.
    device="neuron",
    tensor_parallel_size=2)
# Generate texts from the prompts. The output is a list of RequestOutput objects
# that contain the prompt, generated text, and other information.
outputs = llm.generate(prompts, sampling_params)
# Print the outputs.
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")
```

1.6.13 Offline Inference With Prefix

Source https://github.com/vllm-project/vllm/blob/main/examples/offline_inference_with_prefix.py.

```python
from vllm import LLM, SamplingParams

prefix = (
    "You are an expert school principal, skilled in effectively managing "
    "faculty and staff. Draft 10-15 questions for a potential first grade "
    "Head Teacher for my K-12, all-girls', independent school that emphasizes "
    "community, joyful discovery, and life-long learning. The candidate is "
    "coming in for a first-round panel interview for a 8th grade Math "
    "teaching role. They have 5 years of previous teaching experience "
    "as an assistant teacher at a co-ed, public school with experience "
    "in middle school math teaching. Based on these information, fulfill "
    "the following paragraph: ")

# Sample prompts.
prompts = [
    "Hello, my name is",
    "The president of the United States is",
    "The capital of France is",
    "The future of AI is",
]
# Create a sampling params object.
sampling_params = SamplingParams(temperature=0.0)

# Create an LLM.
llm = LLM(model="facebook/opt-125m", enable_prefix_caching=True)

generating_prompts = [prefix + prompt for prompt in prompts]

# Generate texts from the prompts. The output is a list of RequestOutput objects
# that contain the prompt, generated text, and other information.
outputs = llm.generate(generating_prompts, sampling_params)
# Print the outputs.
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")

print("-" * 80)

# The llm.generate call will batch all prompts and send the batch at once
# if resources allow. The prefix will only be cached after the first batch
# is processed, so we need to call generate once to calculate the prefix
# and cache it.
outputs = llm.generate(generating_prompts[0], sampling_params)

# Subsequent batches can leverage the cached prefix
outputs = llm.generate(generating_prompts, sampling_params)

# Print the outputs. You should see the same outputs as before
```

(continues on next page)

(continued from previous page)

```python
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")
```

1.6.14 OpenAI Chat Completion Client

Source https://github.com/vllm-project/vllm/blob/main/examples/openai_chat_completion_client.py.

```python
from openai import OpenAI

# Modify OpenAI's API key and API base to use vLLM's API server.
openai_api_key = "EMPTY"
openai_api_base = "http://localhost:8000/v1"

client = OpenAI(
    # defaults to os.environ.get("OPENAI_API_KEY")
    api_key=openai_api_key,
    base_url=openai_api_base,
)

models = client.models.list()
model = models.data[0].id

chat_completion = client.chat.completions.create(
    messages=[{
        "role": "system",
        "content": "You are a helpful assistant."
    }, {
        "role": "user",
        "content": "Who won the world series in 2020?"
    }, {
        "role":
        "assistant",
        "content":
        "The Los Angeles Dodgers won the World Series in 2020."
    }, {
        "role": "user",
        "content": "Where was it played?"
    }],
    model=model,
)

print("Chat completion results:")
print(chat_completion)
```

1.6.15 OpenAI Completion Client

Source https://github.com/vllm-project/vllm/blob/main/examples/openai_completion_client.py.

```python
from openai import OpenAI

# Modify OpenAI's API key and API base to use vLLM's API server.
openai_api_key = "EMPTY"
openai_api_base = "http://localhost:8000/v1"

client = OpenAI(
    # defaults to os.environ.get("OPENAI_API_KEY")
    api_key=openai_api_key,
    base_url=openai_api_base,
)

models = client.models.list()
model = models.data[0].id

# Completion API
stream = False
completion = client.completions.create(
    model=model,
    prompt="A robot may not injure a human being",
    echo=False,
    n=2,
    stream=stream,
    logprobs=3)

print("Completion results:")
if stream:
    for c in completion:
        print(c)
else:
    print(completion)
```

1.6.16 OpenAI Embedding Client

Source https://github.com/vllm-project/vllm/blob/main/examples/openai_embedding_client.py.

```python
from openai import OpenAI

# Modify OpenAI's API key and API base to use vLLM's API server.
openai_api_key = "EMPTY"
openai_api_base = "http://localhost:8000/v1"

client = OpenAI(
    # defaults to os.environ.get("OPENAI_API_KEY")
    api_key=openai_api_key,
    base_url=openai_api_base,
)

```

(continues on next page)

(continued from previous page)

```python
models = client.models.list()
model = models.data[0].id

responses = client.embeddings.create(input=[
    "Hello my name is",
    "The best thing about vLLM is that it supports many different models"
],
                                     model=model)

for data in responses.data:
    print(data.embedding)  # list of float of len 4096
```

1.6.17 Save Sharded State

Source https://github.com/vllm-project/vllm/blob/main/examples/save_sharded_state.py.

```python
"""
Saves each worker's model state dict directly to a checkpoint, which enables a
fast load path for large tensor-parallel models where each worker only needs to
read its own shard rather than the entire checkpoint.

Example usage:

python save_sharded_state.py \
    --model /path/to/load \
    --quantization deepspeedfp \
    --tensor-parallel-size 8 \
    --output /path/to/save

Then, the model can be loaded with

llm = LLM(
    model="/path/to/save",
    load_format="sharded_state",
    quantization="deepspeedfp",
    tensor_parallel_size=8,
)
"""
import argparse
import dataclasses
import os
import shutil
from pathlib import Path

from vllm import LLM, EngineArgs

parser = argparse.ArgumentParser()
EngineArgs.add_cli_args(parser)
parser.add_argument("--output",
                    "-o",
                    required=True,
```

(continues on next page)

(continued from previous page)

```python
                    type=str,
                    help="path to output checkpoint")
parser.add_argument("--file-pattern",
                    type=str,
                    help="string pattern of saved filenames")
parser.add_argument("--max-file-size",
                    type=str,
                    default=5 * 1024**3,
                    help="max size (in bytes) of each safetensors file")


def main(args):
    engine_args = EngineArgs.from_cli_args(args)
    if engine_args.enable_lora:
        raise ValueError("Saving with enable_lora=True is not supported!")
    model_path = engine_args.model
    if not Path(model_path).is_dir():
        raise ValueError("model path must be a local directory")
    # Create LLM instance from arguments
    llm = LLM(**dataclasses.asdict(engine_args))
    # Prepare output directory
    Path(args.output).mkdir(exist_ok=True)
    # Dump worker states to output directory
    model_executor = llm.llm_engine.model_executor
    model_executor.save_sharded_state(path=args.output,
                                      pattern=args.file_pattern,
                                      max_size=args.max_file_size)
    # Copy metadata files to output directory
    for file in os.listdir(model_path):
        if os.path.splitext(file)[1] not in (".bin", ".pt", ".safetensors"):
            if os.path.isdir(os.path.join(model_path, file)):
                shutil.copytree(os.path.join(model_path, file),
                                os.path.join(args.output, file))
            else:
                shutil.copy(os.path.join(model_path, file), args.output)


if __name__ == "__main__":
    args = parser.parse_args()
    main(args)
```

1.6.18 Tensorize vLLM Model

Source https://github.com/vllm-project/vllm/blob/main/examples/tensorize_vllm_model.py.

```python
import argparse
import dataclasses
import json
import os
import uuid
from functools import partial
```

(continues on next page)

(continued from previous page)

```

from tensorizer import stream_io

from vllm import LLM
from vllm.distributed import (init_distributed_environment,
                              initialize_model_parallel)
from vllm.engine.arg_utils import EngineArgs
from vllm.engine.llm_engine import LLMEngine
from vllm.model_executor.model_loader.tensorizer import (TensorizerArgs,
                                                         TensorizerConfig,
                                                         serialize_vllm_model)

# yapf conflicts with isort for this docstring
# yapf: disable
"""
tensorize_vllm_model.py is a script that can be used to serialize and
deserialize vLLM models. These models can be loaded using tensorizer
to the GPU extremely quickly over an HTTP/HTTPS endpoint, an S3 endpoint,
or locally. Tensor encryption and decryption is also supported, although
libsodium must be installed to use it. Install vllm with tensorizer support
using `pip install vllm[tensorizer]`. To learn more about tensorizer, visit
https://github.com/coreweave/tensorizer

To serialize a model, install vLLM from source, then run something
like this from the root level of this repository:

python -m examples.tensorize_vllm_model \
   --model facebook/opt-125m \
   serialize \
   --serialized-directory s3://my-bucket \
   --suffix v1

Which downloads the model from HuggingFace, loads it into vLLM, serializes it,
and saves it to your S3 bucket. A local directory can also be used. This
assumes your S3 credentials are specified as environment variables
in the form of `S3_ACCESS_KEY_ID`, `S3_SECRET_ACCESS_KEY`, and
`S3_ENDPOINT_URL`. To provide S3 credentials directly, you can provide
`--s3-access-key-id` and `--s3-secret-access-key`, as well as `--s3-endpoint`
as CLI args to this script.

You can also encrypt the model weights with a randomly-generated key by
providing a `--keyfile` argument.

To deserialize a model, you can run something like this from the root
level of this repository:

python -m examples.tensorize_vllm_model \
   --model EleutherAI/gpt-j-6B \
   --dtype float16 \
   deserialize \
   --path-to-tensors s3://my-bucket/vllm/EleutherAI/gpt-j-6B/v1/model.tensors

```

(continues on next page)

(continued from previous page)

```
Which downloads the model tensors from your S3 bucket and deserializes them.

You can also provide a `--keyfile` argument to decrypt the model weights if
they were serialized with encryption.

For more information on the available arguments for serializing, run
`python -m examples.tensorize_vllm_model serialize --help`.

Or for deserializing:

`python -m examples.tensorize_vllm_model deserialize --help`.

Once a model is serialized, tensorizer can be invoked with the `LLM` class
directly to load models:

    llm = LLM(model="facebook/opt-125m",
              load_format="tensorizer",
              model_loader_extra_config=TensorizerConfig(
                    tensorizer_uri = path_to_tensors,
                    num_readers=3,
                    )
              )

A serialized model can be used during model loading for the vLLM OpenAI
inference server. `model_loader_extra_config` is exposed as the CLI arg
`--model-loader-extra-config`, and accepts a JSON string literal of the
TensorizerConfig arguments desired.

In order to see all of the available arguments usable to configure
loading with tensorizer that are given to `TensorizerConfig`, run:

`python -m examples.tensorize_vllm_model deserialize --help`

under the `tensorizer options` section. These can also be used for
deserialization in this example script, although `--tensorizer-uri` and
`--path-to-tensors` are functionally the same in this case.
"""


def parse_args():
    parser = argparse.ArgumentParser(
        description="An example script that can be used to serialize and "
        "deserialize vLLM models. These models "
        "can be loaded using tensorizer directly to the GPU "
        "extremely quickly. Tensor encryption and decryption is "
        "also supported, although libsodium must be installed to "
        "use it.")
    parser = EngineArgs.add_cli_args(parser)
    subparsers = parser.add_subparsers(dest='command')

    serialize_parser = subparsers.add_parser(
        'serialize', help="Serialize a model to `--serialized-directory`")
```

(continues on next page)

(continued from previous page)

```

    serialize_parser.add_argument(
        "--suffix",
        type=str,
        required=False,
        help=(
            "The suffix to append to the serialized model directory, which is "
            "used to construct the location of the serialized model tensors, "
            "e.g. if `--serialized-directory` is `s3://my-bucket/` and "
            "`--suffix` is `v1`, the serialized model tensors will be "
            "saved to "
            "`s3://my-bucket/vllm/EleutherAI/gpt-j-6B/v1/model.tensors`. "
            "If none is provided, a random UUID will be used."))
    serialize_parser.add_argument(
        "--serialized-directory",
        type=str,
        required=True,
        help="The directory to serialize the model to. "
        "This can be a local directory or S3 URI. The path to where the "
        "tensors are saved is a combination of the supplied `dir` and model "
        "reference ID. For instance, if `dir` is the serialized directory, "
        "and the model HuggingFace ID is `EleutherAI/gpt-j-6B`, tensors will "
        "be saved to `dir/vllm/EleutherAI/gpt-j-6B/suffix/model.tensors`, "
        "where `suffix` is given by `--suffix` or a random UUID if not "
        "provided.")

    serialize_parser.add_argument(
        "--keyfile",
        type=str,
        required=False,
        help=("Encrypt the model weights with a randomly-generated binary key,"
              " and save the key at this path"))

    deserialize_parser = subparsers.add_parser(
        'deserialize',
        help=("Deserialize a model from `--path-to-tensors`"
              " to verify it can be loaded and used."))

    deserialize_parser.add_argument(
        "--path-to-tensors",
        type=str,
        required=True,
        help="The local path or S3 URI to the model tensors to deserialize. ")

    deserialize_parser.add_argument(
        "--keyfile",
        type=str,
        required=False,
        help=("Path to a binary key to use to decrypt the model weights,"
              " if the model was serialized with encryption"))

    TensorizerArgs.add_cli_args(deserialize_parser)
```

(continues on next page)

(continued from previous page)

```

    return parser.parse_args()



def deserialize():
    llm = LLM(model=args.model,
              load_format="tensorizer",
              model_loader_extra_config=tensorizer_config
    )
    return llm



args = parse_args()

s3_access_key_id = (getattr(args, 's3_access_key_id', None)
                    or os.environ.get("S3_ACCESS_KEY_ID", None))
s3_secret_access_key = (getattr(args, 's3_secret_access_key', None)
                        or os.environ.get("S3_SECRET_ACCESS_KEY", None))
s3_endpoint = (getattr(args, 's3_endpoint', None)
               or os.environ.get("S3_ENDPOINT_URL", None))

credentials = {
    "s3_access_key_id": s3_access_key_id,
    "s3_secret_access_key": s3_secret_access_key,
    "s3_endpoint": s3_endpoint
}

_read_stream, _write_stream = (partial(
    stream_io.open_stream,
    mode=mode,
    s3_access_key_id=s3_access_key_id,
    s3_secret_access_key=s3_secret_access_key,
    s3_endpoint=s3_endpoint,
) for mode in ("rb", "wb+"))

model_ref = args.model

model_name = model_ref.split("/")[1]

os.environ["MASTER_ADDR"] = "127.0.0.1"
os.environ["MASTER_PORT"] = "8080"

init_distributed_environment(world_size=1, rank=0, local_rank=0)
initialize_model_parallel()

keyfile = args.keyfile if args.keyfile else None


if args.model_loader_extra_config:
    config = json.loads(args.model_loader_extra_config)
```

(continues on next page)

(continued from previous page)

```python
    tensorizer_args = TensorizerConfig(**config)._construct_tensorizer_args()
    tensorizer_args.tensorizer_uri = args.path_to_tensors
else:
    tensorizer_args = None

if args.command == "serialize":
    eng_args_dict = {f.name: getattr(args, f.name) for f in
                     dataclasses.fields(EngineArgs)}

    engine_args = EngineArgs.from_cli_args(argparse.Namespace(**eng_args_dict))
    engine = LLMEngine.from_engine_args(engine_args)

    input_dir = args.serialized_directory.rstrip('/')
    suffix = args.suffix if args.suffix else uuid.uuid4().hex
    base_path = f"{input_dir}/vllm/{model_ref}/{suffix}"
    model_path = f"{base_path}/model.tensors"
    tensorizer_config = TensorizerConfig(
        tensorizer_uri=model_path,
        **credentials)
    serialize_vllm_model(engine, tensorizer_config, keyfile)
elif args.command == "deserialize":
    if not tensorizer_args:
        tensorizer_config = TensorizerConfig(
            tensorizer_uri=args.path_to_tensors,
            encryption_keyfile = keyfile,
            **credentials
        )
    deserialize()
else:
    raise ValueError("Either serialize or deserialize must be specified.")
```

1.7 OpenAI Compatible Server

vLLM provides an HTTP server that implements OpenAI's Completions and Chat API.

You can start the server using Python, or using *Docker*:

```
python -m vllm.entrypoints.openai.api_server --model NousResearch/Meta-Llama-3-8B-
↪Instruct --dtype auto --api-key token-abc123
```

To call the server, you can use the official OpenAI Python client library, or any other HTTP client.

```python
from openai import OpenAI
client = OpenAI(
    base_url="http://localhost:8000/v1",
    api_key="token-abc123",
)

completion = client.chat.completions.create(
  model="NousResearch/Meta-Llama-3-8B-Instruct",
  messages=[
```

(continues on next page)

(continued from previous page)

```
    {"role": "user", "content": "Hello!"}
  ]
)

print(completion.choices[0].message)
```

1.7.1 API Reference

Please see the OpenAI API Reference for more information on the API. We support all parameters except:

- Chat: `tools`, and `tool_choice`.
- Completions: `suffix`.

1.7.2 Extra Parameters

vLLM supports a set of parameters that are not part of the OpenAI API. In order to use them, you can pass them as extra parameters in the OpenAI client. Or directly merge them into the JSON payload if you are using HTTP call directly.

```
completion = client.chat.completions.create(
  model="NousResearch/Meta-Llama-3-8B-Instruct",
  messages=[
    {"role": "user", "content": "Classify this sentiment: vLLM is wonderful!"}
  ],
  extra_body={
    "guided_choice": ["positive", "negative"]
  }
)
```

Extra Parameters for Chat API

The following *sampling parameters (click through to see documentation)* are supported.

```
    best_of: Optional[int] = None
    use_beam_search: Optional[bool] = False
    top_k: Optional[int] = -1
    min_p: Optional[float] = 0.0
    repetition_penalty: Optional[float] = 1.0
    length_penalty: Optional[float] = 1.0
    early_stopping: Optional[bool] = False
    ignore_eos: Optional[bool] = False
    min_tokens: Optional[int] = 0
    stop_token_ids: Optional[List[int]] = Field(default_factory=list)
    skip_special_tokens: Optional[bool] = True
    spaces_between_special_tokens: Optional[bool] = True
```

The following extra parameters are supported:

```
    echo: Optional[bool] = Field(
        default=False,
```

(continues on next page)

(continued from previous page)

```
        description=(
            "If true, the new message will be prepended with the last message "
            "if they belong to the same role."),
    )
    add_generation_prompt: Optional[bool] = Field(
        default=True,
        description=
        ("If true, the generation prompt will be added to the chat template. "
         "This is a parameter used by chat template in tokenizer config of the "
         "model."),
    )
    include_stop_str_in_output: Optional[bool] = Field(
        default=False,
        description=(
            "Whether to include the stop string in the output. "
            "This is only applied when the stop or stop_token_ids is set."),
    )
    guided_json: Optional[Union[str, dict, BaseModel]] = Field(
        default=None,
        description=("If specified, the output will follow the JSON schema."),
    )
    guided_regex: Optional[str] = Field(
        default=None,
        description=(
            "If specified, the output will follow the regex pattern."),
    )
    guided_choice: Optional[List[str]] = Field(
        default=None,
        description=(
            "If specified, the output will be exactly one of the choices."),
    )
    guided_grammar: Optional[str] = Field(
        default=None,
        description=(
            "If specified, the output will follow the context free grammar."),
    )
    guided_decoding_backend: Optional[str] = Field(
        default=None,
        description=(
            "If specified, will override the default guided decoding backend "
            "of the server for this specific request. If set, must be either "
            "'outlines' / 'lm-format-enforcer'"))
    guided_whitespace_pattern: Optional[str] = Field(
        default=None,
        description=(
            "If specified, will override the default whitespace pattern "
            "for guided json decoding."))
```

Extra Parameters for Completions API

The following *sampling parameters (click through to see documentation)* are supported.

```
    use_beam_search: Optional[bool] = False
    top_k: Optional[int] = -1
    min_p: Optional[float] = 0.0
    repetition_penalty: Optional[float] = 1.0
    length_penalty: Optional[float] = 1.0
    early_stopping: Optional[bool] = False
    stop_token_ids: Optional[List[int]] = Field(default_factory=list)
    ignore_eos: Optional[bool] = False
    min_tokens: Optional[int] = 0
    skip_special_tokens: Optional[bool] = True
    spaces_between_special_tokens: Optional[bool] = True
    truncate_prompt_tokens: Optional[Annotated[int, Field(ge=1)]] = None
```

The following extra parameters are supported:

```
    include_stop_str_in_output: Optional[bool] = Field(
        default=False,
        description=(
            "Whether to include the stop string in the output. "
            "This is only applied when the stop or stop_token_ids is set."),
    )
    response_format: Optional[ResponseFormat] = Field(
        default=None,
        description=
        ("Similar to chat completion, this parameter specifies the format of "
         "output. Only {'type': 'json_object'} or {'type': 'text' } is "
         "supported."),
    )
    guided_json: Optional[Union[str, dict, BaseModel]] = Field(
        default=None,
        description=("If specified, the output will follow the JSON schema."),
    )
    guided_regex: Optional[str] = Field(
        default=None,
        description=(
            "If specified, the output will follow the regex pattern."),
    )
    guided_choice: Optional[List[str]] = Field(
        default=None,
        description=(
            "If specified, the output will be exactly one of the choices."),
    )
    guided_grammar: Optional[str] = Field(
        default=None,
        description=(
            "If specified, the output will follow the context free grammar."),
    )
    guided_decoding_backend: Optional[str] = Field(
        default=None,
        description=(
```

(continues on next page)

(continued from previous page)

```
            "If specified, will override the default guided decoding backend "
            "of the server for this specific request. If set, must be one of "
            "'outlines' / 'lm-format-enforcer'"))
    guided_whitespace_pattern: Optional[str] = Field(
        default=None,
        description=(
            "If specified, will override the default whitespace pattern "
            "for guided json decoding."))
```

1.7.3 Chat Template

In order for the language model to support chat protocol, vLLM requires the model to include a chat template in its tokenizer configuration. The chat template is a Jinja2 template that specifies how are roles, messages, and other chat-specific tokens are encoded in the input.

An example chat template for `NousResearch/Meta-Llama-3-8B-Instruct` can be found here

Some models do not provide a chat template even though they are instruction/chat fine-tuned. For those model, you can manually specify their chat template in the `--chat-template` parameter with the file path to the chat template, or the template in string form. Without a chat template, the server will not be able to process chat and all chat requests will error.

```
python -m vllm.entrypoints.openai.api_server \
  --model ... \
  --chat-template ./path-to-chat-template.jinja
```

vLLM community provides a set of chat templates for popular models. You can find them in the examples directory here

1.7.4 Command line arguments for the server

vLLM OpenAI-Compatible RESTful API server.

```
usage: -m vllm.entrypoints.openai.api_server [-h] [--host HOST] [--port PORT]
                                             [--uvicorn-log-level {debug,info,warning,
↪error,critical,trace}]
                                             [--allow-credentials]
                                             [--allowed-origins ALLOWED_ORIGINS]
                                             [--allowed-methods ALLOWED_METHODS]
                                             [--allowed-headers ALLOWED_HEADERS]
                                             [--api-key API_KEY]
                                             [--lora-modules LORA_MODULES [LORA_MODULES .
↪..]]
                                             [--chat-template CHAT_TEMPLATE]
                                             [--response-role RESPONSE_ROLE]
                                             [--ssl-keyfile SSL_KEYFILE]
                                             [--ssl-certfile SSL_CERTFILE]
                                             [--ssl-ca-certs SSL_CA_CERTS]
                                             [--ssl-cert-reqs SSL_CERT_REQS]
```

(continues on next page)

(continued from previous page)

```
                                             [--root-path ROOT_PATH]
                                             [--middleware MIDDLEWARE]
                                             [--model MODEL]
                                             [--tokenizer TOKENIZER]
                                             [--skip-tokenizer-init]
                                             [--revision REVISION]
                                             [--code-revision CODE_REVISION]
                                             [--tokenizer-revision TOKENIZER_REVISION]
                                             [--tokenizer-mode {auto,slow}]
                                             [--trust-remote-code]
                                             [--download-dir DOWNLOAD_DIR]
                                             [--load-format {auto,pt,safetensors,npcache,
↪dummy,tensorizer}]
                                             [--dtype {auto,half,float16,bfloat16,float,
↪float32}]
                                             [--kv-cache-dtype {auto,fp8,fp8_e5m2,fp8_
↪e4m3}]
                                             [--quantization-param-path QUANTIZATION_
↪PARAM_PATH]
                                             [--max-model-len MAX_MODEL_LEN]
                                             [--guided-decoding-backend {outlines,lm-
↪format-enforcer}]
                                             [--distributed-executor-backend {ray,mp}]
                                             [--worker-use-ray]
                                             [--pipeline-parallel-size PIPELINE_PARALLEL_
↪SIZE]
                                             [--tensor-parallel-size TENSOR_PARALLEL_
↪SIZE]
                                             [--max-parallel-loading-workers MAX_
↪PARALLEL_LOADING_WORKERS]
                                             [--ray-workers-use-nsight]
                                             [--block-size {8,16,32}]
                                             [--enable-prefix-caching]
                                             [--disable-sliding-window]
                                             [--use-v2-block-manager]
                                             [--num-lookahead-slots NUM_LOOKAHEAD_SLOTS]
                                             [--seed SEED]
                                             [--swap-space SWAP_SPACE]
                                             [--gpu-memory-utilization GPU_MEMORY_
↪UTILIZATION]
                                             [--num-gpu-blocks-override NUM_GPU_BLOCKS_
↪OVERRIDE]
                                             [--max-num-batched-tokens MAX_NUM_BATCHED_
↪TOKENS]
                                             [--max-num-seqs MAX_NUM_SEQS]
                                             [--max-logprobs MAX_LOGPROBS]
                                             [--disable-log-stats]
                                             [--quantization {aqlm,awq,deepspeedfp,fp8,
↪marlin,gptq_marlin_24,gptq_marlin,gptq,squeezellm,sparseml,None}]
                                             [--rope-scaling ROPE_SCALING]
                                             [--enforce-eager]
                                             [--max-context-len-to-capture MAX_CONTEXT_
```

(continues on next page)

(continued from previous page)

```
↪LEN_TO_CAPTURE]
                                             [--max-seq-len-to-capture MAX_SEQ_LEN_TO_
↪CAPTURE]
                                             [--disable-custom-all-reduce]
                                             [--tokenizer-pool-size TOKENIZER_POOL_SIZE]
                                             [--tokenizer-pool-type TOKENIZER_POOL_TYPE]
                                             [--tokenizer-pool-extra-config TOKENIZER_
↪POOL_EXTRA_CONFIG]
                                             [--enable-lora]
                                             [--max-loras MAX_LORAS]
                                             [--max-lora-rank MAX_LORA_RANK]
                                             [--lora-extra-vocab-size LORA_EXTRA_VOCAB_
↪SIZE]
                                             [--lora-dtype {auto,float16,bfloat16,
↪float32}]
                                             [--long-lora-scaling-factors LONG_LORA_
↪SCALING_FACTORS]
                                             [--max-cpu-loras MAX_CPU_LORAS]
                                             [--fully-sharded-loras]
                                             [--device {auto,cuda,neuron,cpu}]
                                             [--image-input-type {pixel_values,image_
↪features}]
                                             [--image-token-id IMAGE_TOKEN_ID]
                                             [--image-input-shape IMAGE_INPUT_SHAPE]
                                             [--image-feature-size IMAGE_FEATURE_SIZE]
                                             [--scheduler-delay-factor SCHEDULER_DELAY_
↪FACTOR]
                                             [--enable-chunked-prefill]
                                             [--speculative-model SPECULATIVE_MODEL]
                                             [--num-speculative-tokens NUM_SPECULATIVE_
↪TOKENS]
                                             [--speculative-max-model-len SPECULATIVE_
↪MAX_MODEL_LEN]
                                             [--speculative-disable-by-batch-size␣
↪SPECULATIVE_DISABLE_BY_BATCH_SIZE]
                                             [--ngram-prompt-lookup-max NGRAM_PROMPT_
↪LOOKUP_MAX]
                                             [--ngram-prompt-lookup-min NGRAM_PROMPT_
↪LOOKUP_MIN]
                                             [--model-loader-extra-config MODEL_LOADER_
↪EXTRA_CONFIG]
                                             [--served-model-name SERVED_MODEL_NAME␣
↪[SERVED_MODEL_NAME ...]]
                                             [--engine-use-ray]
                                             [--disable-log-requests]
                                             [--max-log-len MAX_LOG_LEN]
```

Named Arguments

--host	host name
--port	port number Default: 8000
--uvicorn-log-level	Possible choices: debug, info, warning, error, critical, trace log level for uvicorn Default: "info"
--allow-credentials	allow credentials Default: False
--allowed-origins	allowed origins Default: ['*']
--allowed-methods	allowed methods Default: ['*']
--allowed-headers	allowed headers Default: ['*']
--api-key	If provided, the server will require this key to be presented in the header.
--lora-modules	LoRA module configurations in the format name=path. Multiple modules can be specified.
--chat-template	The file path to the chat template, or the template in single-line form for the specified model
--response-role	The role name to return if *request.add_generation_prompt=true*. Default: assistant
--ssl-keyfile	The file path to the SSL key file
--ssl-certfile	The file path to the SSL cert file
--ssl-ca-certs	The CA certificates file
--ssl-cert-reqs	Whether client certificate is required (see stdlib ssl module's) Default: 0
--root-path	FastAPI root_path when app is behind a path based routing proxy
--middleware	Additional ASGI middleware to apply to the app. We accept multiple –middleware arguments. The value should be an import path. If a function is provided, vLLM will add it to the server using @app.middleware('http'). If a class is provided, vLLM will add it to the server using app.add_middleware(). Default: []
--model	Name or path of the huggingface model to use. Default: "facebook/opt-125m"
--tokenizer	Name or path of the huggingface tokenizer to use.

--skip-tokenizer-init Skip initialization of tokenizer and detokenizer

Default: False

--revision 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.

--code-revision The specific revision to use for the model code on Hugging Face Hub. It can be a branch name, a tag name, or a commit id. If unspecified, will use the default version.

--tokenizer-revision The specific tokenizer version to use. It can be a branch name, a tag name, or a commit id. If unspecified, will use the default version.

--tokenizer-mode Possible choices: auto, slow

The tokenizer mode.

- "auto" will use the fast tokenizer if available.
- "slow" will always use the slow tokenizer.

Default: "auto"

--trust-remote-code Trust remote code from huggingface.

Default: False

--download-dir Directory to download and load the weights, default to the default cache dir of huggingface.

--load-format Possible choices: auto, pt, safetensors, npcache, dummy, tensorizer

The format of the model weights to load.

- "auto" will try to load the weights in the safetensors format and fall back to the pytorch bin format if safetensors format is not available.
- "pt" will load the weights in the pytorch bin format.
- "safetensors" will load the weights in the safetensors format.
- "npcache" will load the weights in pytorch format and store a numpy cache to speed up the loading.
- "dummy" will initialize the weights with random values, which is mainly for profiling.
- "tensorizer" will load the weights using tensorizer from CoreWeave. See the Tensorize vLLM Model script in the Examplessection for more information.

Default: "auto"

--dtype Possible choices: auto, half, float16, bfloat16, float, float32

Data type for model weights and activations.

- "auto" will use FP16 precision for FP32 and FP16 models, and BF16 precision for BF16 models.
- "half" for FP16. Recommended for AWQ quantization.
- "float16" is the same as "half".
- "bfloat16" for a balance between precision and range.
- "float" is shorthand for FP32 precision.

- "float32" for FP32 precision.

Default: "auto"

--kv-cache-dtype Possible choices: auto, fp8, fp8_e5m2, fp8_e4m3

Data type for kv cache storage. If "auto", will use model data type. CUDA 11.8+ supports fp8 (=fp8_e4m3) and fp8_e5m2. ROCm (AMD GPU) supports fp8 (=fp8_e4m3)

Default: "auto"

--quantization-param-path Path to the JSON file containing the KV cache scaling factors. This should generally be supplied, when KV cache dtype is FP8. Otherwise, KV cache scaling factors default to 1.0, which may cause accuracy issues. FP8_E5M2 (without scaling) is only supported on cuda versiongreater than 11.8. On ROCm (AMD GPU), FP8_E4M3 is instead supported for common inference criteria.

--max-model-len Model context length. If unspecified, will be automatically derived from the model config.

--guided-decoding-backend Possible choices: outlines, lm-format-enforcer

Which engine will be used for guided decoding (JSON schema / regex etc) by default. Currently support https://github.com/outlines-dev/outlines and https://github.com/noamgat/lm-format-enforcer. Can be overridden per request via guided_decoding_backend parameter.

Default: "outlines"

--distributed-executor-backend Possible choices: ray, mp

Backend to use for distributed serving. When more than 1 GPU is used, will be automatically set to "ray" if installed or "mp" (multiprocessing) otherwise.

--worker-use-ray Deprecated, use –distributed-executor-backend=ray.

Default: False

--pipeline-parallel-size, -pp Number of pipeline stages.

Default: 1

--tensor-parallel-size, -tp Number of tensor parallel replicas.

Default: 1

--max-parallel-loading-workers Load model sequentially in multiple batches, to avoid RAM OOM when using tensor parallel and large models.

--ray-workers-use-nsight If specified, use nsight to profile Ray workers.

Default: False

--block-size Possible choices: 8, 16, 32

Token block size for contiguous chunks of tokens.

Default: 16

--enable-prefix-caching Enables automatic prefix caching.

Default: False

--disable-sliding-window Disables sliding window, capping to sliding window size

Default: False

--use-v2-block-manager Use BlockSpaceMangerV2.

Default: False

--num-lookahead-slots Experimental scheduling config necessary for speculative decoding. This will be replaced by speculative config in the future; it is present to enable correctness tests until then.

Default: 0

--seed Random seed for operations.

Default: 0

--swap-space CPU swap space size (GiB) per GPU.

Default: 4

--gpu-memory-utilization The fraction of GPU memory to be used for the model executor, which can range from 0 to 1. For example, a value of 0.5 would imply 50% GPU memory utilization. If unspecified, will use the default value of 0.9.

Default: 0.9

--num-gpu-blocks-override If specified, ignore GPU profiling result and use this numberof GPU blocks. Used for testing preemption.

--max-num-batched-tokens Maximum number of batched tokens per iteration.

--max-num-seqs Maximum number of sequences per iteration.

Default: 256

--max-logprobs Max number of log probs to return logprobs is specified in SamplingParams.

Default: 5

--disable-log-stats Disable logging statistics.

Default: False

--quantization, -q Possible choices: aqlm, awq, deepspeedfp, fp8, marlin, gptq_marlin_24, gptq_marlin, gptq, squeezellm, sparseml, None

Method used to quantize the weights. If None, we first check the *quantization_config* attribute in the model config file. If that is None, we assume the model weights are not quantized and use *dtype* to determine the data type of the weights.

--rope-scaling RoPE scaling configuration in JSON format. For example, {“type”:”dynamic”,”factor”:2.0}

--enforce-eager Always use eager-mode PyTorch. If False, will use eager mode and CUDA graph in hybrid for maximal performance and flexibility.

Default: False

--max-context-len-to-capture Maximum context length covered by CUDA graphs. When a sequence has context length larger than this, we fall back to eager mode. (DEPRECATED. Use –max-seq-len-to-capture instead)

--max-seq-len-to-capture Maximum sequence length covered by CUDA graphs. When a sequence has context length larger than this, we fall back to eager mode.

Default: 8192

--disable-custom-all-reduce See ParallelConfig.

Default: False

--tokenizer-pool-size Size of tokenizer pool to use for asynchronous tokenization. If 0, will use synchronous tokenization.

Default: 0

--tokenizer-pool-type Type of tokenizer pool to use for asynchronous tokenization. Ignored if tokenizer_pool_size is 0.

Default: “ray”

--tokenizer-pool-extra-config Extra config for tokenizer pool. This should be a JSON string that will be parsed into a dictionary. Ignored if tokenizer_pool_size is 0.

--enable-lora If True, enable handling of LoRA adapters.

Default: False

--max-loras Max number of LoRAs in a single batch.

Default: 1

--max-lora-rank Max LoRA rank.

Default: 16

--lora-extra-vocab-size Maximum size of extra vocabulary that can be present in a LoRA adapter (added to the base model vocabulary).

Default: 256

--lora-dtype Possible choices: auto, float16, bfloat16, float32

Data type for LoRA. If auto, will default to base model dtype.

Default: “auto”

--long-lora-scaling-factors Specify multiple scaling factors (which can be different from base model scaling factor - see eg. Long LoRA) to allow for multiple LoRA adapters trained with those scaling factors to be used at the same time. If not specified, only adapters trained with the base model scaling factor are allowed.

--max-cpu-loras Maximum number of LoRAs to store in CPU memory. Must be >= than max_num_seqs. Defaults to max_num_seqs.

--fully-sharded-loras By default, only half of the LoRA computation is sharded with tensor parallelism. Enabling this will use the fully sharded layers. At high sequence length, max rank or tensor parallel size, this is likely faster.

Default: False

--device Possible choices: auto, cuda, neuron, cpu

Device type for vLLM execution.

Default: “auto”

--image-input-type Possible choices: pixel_values, image_features

The image input type passed into vLLM. Should be one of “pixel_values” or “image_features”.

--image-token-id Input id for image token.

--image-input-shape The biggest image input shape (worst for memory footprint) given an input type. Only used for vLLM's profile_run.

--image-feature-size The image feature size along the context dimension.

--scheduler-delay-factor Apply a delay (of delay factor multiplied by previousprompt latency) before scheduling next prompt.

Default: 0.0

--enable-chunked-prefill If set, the prefill requests can be chunked based on the max_num_batched_tokens.

Default: False

--speculative-model The name of the draft model to be used in speculative decoding.

--num-speculative-tokens The number of speculative tokens to sample from the draft model in speculative decoding.

--speculative-max-model-len The maximum sequence length supported by the draft model. Sequences over this length will skip speculation.

--speculative-disable-by-batch-size Disable speculative decoding for new incoming requests if the number of enqueue requests is larger than this value.

--ngram-prompt-lookup-max Max size of window for ngram prompt lookup in speculative decoding.

--ngram-prompt-lookup-min Min size of window for ngram prompt lookup in speculative decoding.

--model-loader-extra-config Extra config for model loader. This will be passed to the model loader corresponding to the chosen load_format. This should be a JSON string that will be parsed into a dictionary.

--served-model-name The model name(s) used in the API. If multiple names are provided, the server will respond to any of the provided names. The model name in the model field of a response will be the first name in this list. If not specified, the model name will be the same as the *–model* argument. Noted that this name(s)will also be used in *model_name* tag content of prometheus metrics, if multiple names provided, metricstag will take the first one.

--engine-use-ray Use Ray to start the LLM engine in a separate process as the server process.

Default: False

--disable-log-requests Disable logging requests.

Default: False

--max-log-len Max number of prompt characters or prompt ID numbers being printed in log.

Default: Unlimited

1.8 Deploying with Docker

vLLM offers official docker image for deployment. The image can be used to run OpenAI compatible server. The image is available on Docker Hub as vllm/vllm-openai.

```
$ docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
    -p 8000:8000 \
    --ipc=host \
    vllm/vllm-openai:latest \
    --model mistralai/Mistral-7B-v0.1
```

Note: You can either use the `ipc=host` flag or `--shm-size` flag to allow the container to access the host's shared memory. vLLM uses PyTorch, which uses shared memory to share data between processes under the hood, particularly for tensor parallel inference.

You can build and run vLLM from source via the provided dockerfile. To build vLLM:

```
$ DOCKER_BUILDKIT=1 docker build . --target vllm-openai --tag vllm/vllm-openai #␣
↪optionally specifies: --build-arg max_jobs=8 --build-arg nvcc_threads=2
```

Note: By default vLLM will build for all GPU types for widest distribution. If you are just building for the current GPU type the machine is running on, you can add the argument `--build-arg torch_cuda_arch_list=""` for vLLM to find the current GPU type and build for that.

To run vLLM:

```
$ docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    -p 8000:8000 \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
    vllm/vllm-openai <args...>
```

Note: **For `v0.4.1` and `v0.4.2` only** - the vLLM docker images under these versions are supposed to be run under the root user since a library under the root user's home directory, i.e. `/root/.config/vllm/nccl/cu12/libnccl.so.2.18.1` is required to be loaded during runtime. If you are running the container under a different user, you may need to first change the permissions of the library (and all the parent directories) to allow the user to access it, then run vLLM with environment variable `VLLM_NCCL_SO_PATH=/root/.config/vllm/nccl/cu12/libnccl.so.2.18.1` .

1.9 Distributed Inference and Serving

vLLM supports distributed tensor-parallel inference and serving. Currently, we support Megatron-LM's tensor parallel algorithm. We manage the distributed runtime with Ray. To run distributed inference, install Ray with:

```
$ pip install ray
```

To run multi-GPU inference with the `LLM` class, set the `tensor_parallel_size` argument to the number of GPUs you want to use. For example, to run inference on 4 GPUs:

```
from vllm import LLM
llm = LLM("facebook/opt-13b", tensor_parallel_size=4)
output = llm.generate("San Franciso is a")
```

To run multi-GPU serving, pass in the `--tensor-parallel-size` argument when starting the server. For example, to run API server on 4 GPUs:

```
$ python -m vllm.entrypoints.api_server \
$     --model facebook/opt-13b \
$     --tensor-parallel-size 4
```

To scale vLLM beyond a single machine, start a Ray runtime via CLI before running vLLM:

```
$ # On head node
$ ray start --head

$ # On worker nodes
$ ray start --address=<ray-head-address>
```

After that, you can run inference and serving on multiple machines by launching the vLLM process on the head node by setting `tensor_parallel_size` to the number of GPUs to be the total number of GPUs across all machines.

1.10 Production Metrics

vLLM exposes a number of metrics that can be used to monitor the health of the system. These metrics are exposed via the */metrics* endpoint on the vLLM OpenAI compatible API server.

The following metrics are exposed:

```
class Metrics:
    labelname_finish_reason = "finished_reason"

    def __init__(self, labelnames: List[str], max_model_len: int):
        # Unregister any existing vLLM collectors
        for collector in list(REGISTRY._collector_to_names):
            if hasattr(collector, "_name") and "vllm" in collector._name:
                REGISTRY.unregister(collector)

        # Config Information
        self.info_cache_config = Info(
            name='vllm:cache_config',
            documentation='information of cache_config')
```

(continues on next page)

(continued from previous page)

```
        # System stats
        #   Scheduler State
        self.gauge_scheduler_running = Gauge(
            name="vllm:num_requests_running",
            documentation="Number of requests currently running on GPU.",
            labelnames=labelnames)
        self.gauge_scheduler_waiting = Gauge(
            name="vllm:num_requests_waiting",
            documentation="Number of requests waiting to be processed.",
            labelnames=labelnames)
        self.gauge_scheduler_swapped = Gauge(
            name="vllm:num_requests_swapped",
            documentation="Number of requests swapped to CPU.",
            labelnames=labelnames)
        #   KV Cache Usage in %
        self.gauge_gpu_cache_usage = Gauge(
            name="vllm:gpu_cache_usage_perc",
            documentation="GPU KV-cache usage. 1 means 100 percent usage.",
            labelnames=labelnames)
        self.gauge_cpu_cache_usage = Gauge(
            name="vllm:cpu_cache_usage_perc",
            documentation="CPU KV-cache usage. 1 means 100 percent usage.",
            labelnames=labelnames)

        # Iteration stats
        self.counter_num_preemption = Counter(
            name="vllm:num_preemptions_total",
            documentation="Cumulative number of preemption from the engine.",
            labelnames=labelnames)
        self.counter_prompt_tokens = Counter(
            name="vllm:prompt_tokens_total",
            documentation="Number of prefill tokens processed.",
            labelnames=labelnames)
        self.counter_generation_tokens = Counter(
            name="vllm:generation_tokens_total",
            documentation="Number of generation tokens processed.",
            labelnames=labelnames)
        self.histogram_time_to_first_token = Histogram(
            name="vllm:time_to_first_token_seconds",
            documentation="Histogram of time to first token in seconds.",
            labelnames=labelnames,
            buckets=[
                0.001, 0.005, 0.01, 0.02, 0.04, 0.06, 0.08, 0.1, 0.25, 0.5,
                0.75, 1.0, 2.5, 5.0, 7.5, 10.0
            ])
        self.histogram_time_per_output_token = Histogram(
            name="vllm:time_per_output_token_seconds",
            documentation="Histogram of time per output token in seconds.",
            labelnames=labelnames,
            buckets=[
                0.01, 0.025, 0.05, 0.075, 0.1, 0.15, 0.2, 0.3, 0.4, 0.5, 0.75,
```

(continues on next page)

(continued from previous page)

```
                1.0, 2.5
            ])

        # Request stats
        #   Latency
        self.histogram_e2e_time_request = Histogram(
            name="vllm:e2e_request_latency_seconds",
            documentation="Histogram of end to end request latency in seconds.",
            labelnames=labelnames,
            buckets=[1.0, 2.5, 5.0, 10.0, 15.0, 20.0, 30.0, 40.0, 50.0, 60.0])
        #   Metadata
        self.histogram_num_prompt_tokens_request = Histogram(
            name="vllm:request_prompt_tokens",
            documentation="Number of prefill tokens processed.",
            labelnames=labelnames,
            buckets=build_1_2_5_buckets(max_model_len),
        )
        self.histogram_num_generation_tokens_request = Histogram(
            name="vllm:request_generation_tokens",
            documentation="Number of generation tokens processed.",
            labelnames=labelnames,
            buckets=build_1_2_5_buckets(max_model_len),
        )
        self.histogram_best_of_request = Histogram(
            name="vllm:request_params_best_of",
            documentation="Histogram of the best_of request parameter.",
            labelnames=labelnames,
            buckets=[1, 2, 5, 10, 20],
        )
        self.histogram_n_request = Histogram(
            name="vllm:request_params_n",
            documentation="Histogram of the n request parameter.",
            labelnames=labelnames,
            buckets=[1, 2, 5, 10, 20],
        )
        self.counter_request_success = Counter(
            name="vllm:request_success_total",
            documentation="Count of successfully processed requests.",
            labelnames=labelnames + [Metrics.labelname_finish_reason])

        # Deprecated in favor of vllm:prompt_tokens_total
        self.gauge_avg_prompt_throughput = Gauge(
            name="vllm:avg_prompt_throughput_toks_per_s",
            documentation="Average prefill throughput in tokens/s.",
            labelnames=labelnames,
        )
        # Deprecated in favor of vllm:generation_tokens_total
        self.gauge_avg_generation_throughput = Gauge(
            name="vllm:avg_generation_throughput_toks_per_s",
            documentation="Average generation throughput in tokens/s.",
            labelnames=labelnames,
        )
```

(continues on next page)

(continued from previous page)

```
```

1.11 Environment Variables

vLLM uses the following environment variables to configure the system:

```
environment_variables: Dict[str, Callable[[], Any]] = {

    # ================== Installation Time Env Vars ==================

    # Target device of vLLM, supporting [cuda (by default), rocm, neuron, cpu]
    "VLLM_TARGET_DEVICE":
    lambda: os.getenv("VLLM_TARGET_DEVICE", "cuda"),

    # Maximum number of compilation jobs to run in parallel.
    # By default this is the number of CPUs
    "MAX_JOBS":
    lambda: os.getenv("MAX_JOBS", None),

    # Number of threads to use for nvcc
    # By default this is 1.
    # If set, `MAX_JOBS` will be reduced to avoid oversubscribing the CPU.
    "NVCC_THREADS":
    lambda: os.getenv("NVCC_THREADS", None),

    # If set, vllm will build with Neuron support
    "VLLM_BUILD_WITH_NEURON":
    lambda: bool(os.environ.get("VLLM_BUILD_WITH_NEURON", False)),

    # If set, vllm will use precompiled binaries (*.so)
    "VLLM_USE_PRECOMPILED":
    lambda: bool(os.environ.get("VLLM_USE_PRECOMPILED")),

    # If set, vllm will install Punica kernels
    "VLLM_INSTALL_PUNICA_KERNELS":
    lambda: bool(int(os.getenv("VLLM_INSTALL_PUNICA_KERNELS", "0"))),

    # CMake build type
    # If not set, defaults to "Debug" or "RelWithDebInfo"
    # Available options: "Debug", "Release", "RelWithDebInfo"
    "CMAKE_BUILD_TYPE":
    lambda: os.getenv("CMAKE_BUILD_TYPE"),

    # If set, vllm will print verbose logs during installation
    "VERBOSE":
    lambda: bool(int(os.getenv('VERBOSE', '0'))),

    # Root directory for VLLM configuration files
```

(continues on next page)

(continued from previous page)

```
    # Note that this not only affects how vllm finds its configuration files
    # during runtime, but also affects how vllm installs its configuration
    # files during **installation**.
    "VLLM_CONFIG_ROOT":
    lambda: os.environ.get("VLLM_CONFIG_ROOT", None) or os.getenv(
        "XDG_CONFIG_HOME", None) or os.path.expanduser("~/.config"),

    # ================== Runtime Env Vars ==================

    # used in distributed environment to determine the master address
    'VLLM_HOST_IP':
    lambda: os.getenv('VLLM_HOST_IP', "") or os.getenv("HOST_IP", ""),

    # used in distributed environment to manually set the communication port
    # '0' is used to make mypy happy
    'VLLM_PORT':
    lambda: int(os.getenv('VLLM_PORT', '0'))
    if 'VLLM_PORT' in os.environ else None,

    # If true, will load models from ModelScope instead of Hugging Face Hub.
    # note that the value is true or false, not numbers
    "VLLM_USE_MODELSCOPE":
    lambda: os.environ.get("VLLM_USE_MODELSCOPE", "False").lower() == "true",

    # Instance id represents an instance of the VLLM. All processes in the same
    # instance should have the same instance id.
    "VLLM_INSTANCE_ID":
    lambda: os.environ.get("VLLM_INSTANCE_ID", None),

    # path to cudatoolkit home directory, under which should be bin, include,
    # and lib directories.
    "CUDA_HOME":
    lambda: os.environ.get("CUDA_HOME", None),

    # Path to the NCCL library file. It is needed because nccl>=2.19 brought
    # by PyTorch contains a bug: https://github.com/NVIDIA/nccl/issues/1234
    "VLLM_NCCL_SO_PATH":
    lambda: os.environ.get("VLLM_NCCL_SO_PATH", None),

    # when `VLLM_NCCL_SO_PATH` is not set, vllm will try to find the nccl
    # library file in the locations specified by `LD_LIBRARY_PATH`
    "LD_LIBRARY_PATH":
    lambda: os.environ.get("LD_LIBRARY_PATH", None),

    # flag to control if vllm should use triton flash attention
    "VLLM_USE_TRITON_FLASH_ATTN":
    lambda: (os.environ.get("VLLM_USE_TRITON_FLASH_ATTN", "True").lower() in
             ("true", "1")),

    # local rank of the process in the distributed setting, used to determine
    # the GPU device id
    "LOCAL_RANK":
```

(continues on next page)

(continued from previous page)

```
    lambda: int(os.environ.get("LOCAL_RANK", "0")),

    # used to control the visible devices in the distributed setting
    "CUDA_VISIBLE_DEVICES":
    lambda: os.environ.get("CUDA_VISIBLE_DEVICES", None),

    # timeout for each iteration in the engine
    "VLLM_ENGINE_ITERATION_TIMEOUT_S":
    lambda: int(os.environ.get("VLLM_ENGINE_ITERATION_TIMEOUT_S", "60")),

    # API key for VLLM API server
    "VLLM_API_KEY":
    lambda: os.environ.get("VLLM_API_KEY", None),

    # S3 access information, used for tensorizer to load model from S3
    "S3_ACCESS_KEY_ID":
    lambda: os.environ.get("S3_ACCESS_KEY_ID", None),
    "S3_SECRET_ACCESS_KEY":
    lambda: os.environ.get("S3_SECRET_ACCESS_KEY", None),
    "S3_ENDPOINT_URL":
    lambda: os.environ.get("S3_ENDPOINT_URL", None),

    # Usage stats collection
    "VLLM_USAGE_STATS_SERVER":
    lambda: os.environ.get("VLLM_USAGE_STATS_SERVER", "https://stats.vllm.ai"),
    "VLLM_NO_USAGE_STATS":
    lambda: os.environ.get("VLLM_NO_USAGE_STATS", "0") == "1",
    "VLLM_DO_NOT_TRACK":
    lambda: (os.environ.get("VLLM_DO_NOT_TRACK", None) or os.environ.get(
        "DO_NOT_TRACK", None) or "0") == "1",
    "VLLM_USAGE_SOURCE":
    lambda: os.environ.get("VLLM_USAGE_SOURCE", "production"),

    # Logging configuration
    # If set to 0, vllm will not configure logging
    # If set to 1, vllm will configure logging using the default configuration
    #    or the configuration file specified by VLLM_LOGGING_CONFIG_PATH
    "VLLM_CONFIGURE_LOGGING":
    lambda: int(os.getenv("VLLM_CONFIGURE_LOGGING", "1")),
    "VLLM_LOGGING_CONFIG_PATH":
    lambda: os.getenv("VLLM_LOGGING_CONFIG_PATH"),

    # this is used for configuring the default logging level
    "VLLM_LOGGING_LEVEL":
    lambda: os.getenv("VLLM_LOGGING_LEVEL", "INFO"),

    # Trace function calls
    # If set to 1, vllm will trace function calls
    # Useful for debugging
    "VLLM_TRACE_FUNCTION":
    lambda: int(os.getenv("VLLM_TRACE_FUNCTION", "0")),
```

(continues on next page)

(continued from previous page)

```
    # Backend for attention computation
    # Available options:
    # - "TORCH_SDPA": use torch.nn.MultiheadAttention
    # - "FLASH_ATTN": use FlashAttention
    # - "XFORMERS": use XFormers
    # - "ROCM_FLASH": use ROCmFlashAttention
    "VLLM_ATTENTION_BACKEND":
    lambda: os.getenv("VLLM_ATTENTION_BACKEND", None),

    # CPU key-value cache space
    # default is 4GB
    "VLLM_CPU_KVCACHE_SPACE":
    lambda: int(os.getenv("VLLM_CPU_KVCACHE_SPACE", "0")),

    # If the env var is set, it uses the Ray's compiled DAG API
    # which optimizes the control plane overhead.
    # Run vLLM with VLLM_USE_RAY_COMPILED_DAG=1 to enable it.
    "VLLM_USE_RAY_COMPILED_DAG":
    lambda: bool(os.getenv("VLLM_USE_RAY_COMPILED_DAG", 0)),

    # Use dedicated multiprocess context for workers.
    # Both spawn and fork work
    "VLLM_WORKER_MULTIPROC_METHOD":
    lambda: os.getenv("VLLM_WORKER_MULTIPROC_METHOD", "spawn"),
}
```

1.12 Usage Stats Collection

vLLM collects anonymous usage data by default to help the engineering team better understand which hardware and model configurations are widely used. This data allows them to prioritize their efforts on the most common workloads. The collected data is transparent, does not contain any sensitive information, and will be publicly released for the community's benefit.

1.12.1 What data is collected?

You can see the up to date list of data collected by vLLM in the usage_lib.py.

Here is an example as of v0.4.0:

```
{
  "uuid": "fbe880e9-084d-4cab-a395-8984c50f1109",
  "provider": "GCP",
  "num_cpu": 24,
  "cpu_type": "Intel(R) Xeon(R) CPU @ 2.20GHz",
  "cpu_family_model_stepping": "6,85,7",
  "total_memory": 101261135872,
  "architecture": "x86_64",
  "platform": "Linux-5.10.0-28-cloud-amd64-x86_64-with-glibc2.31",
  "gpu_count": 2,
```

(continues on next page)

(continued from previous page)

```
  "gpu_type": "NVIDIA L4",
  "gpu_memory_per_device": 23580639232,
  "model_architecture": "OPTForCausalLM",
  "vllm_version": "0.3.2+cu123",
  "context": "LLM_CLASS",
  "log_time": 1711663373492490000,
  "source": "production",
  "dtype": "torch.float16",
  "tensor_parallel_size": 1,
  "block_size": 16,
  "gpu_memory_utilization": 0.9,
  "quantization": null,
  "kv_cache_dtype": "auto",
  "enable_lora": false,
  "enable_prefix_caching": false,
  "enforce_eager": false,
  "disable_custom_all_reduce": true
}
```

You can preview the collected data by running the following command:

```
tail ~/.config/vllm/usage_stats.json
```

1.12.2 Opt-out of Usage Stats Collection

You can opt-out of usage stats collection by setting the VLLM_NO_USAGE_STATS or DO_NOT_TRACK environment variable, or by creating a ~/.config/vllm/do_not_track file:

```
# Any of the following methods can disable usage stats collection
export VLLM_NO_USAGE_STATS=1
export DO_NOT_TRACK=1
mkdir -p ~/.config/vllm && touch ~/.config/vllm/do_not_track
```

1.13 Integrations

1.13.1 Deploying and scaling up with SkyPilot

vLLM can be **run and scaled to multiple service replicas on clouds and Kubernetes** with SkyPilot, an open-source framework for running LLMs on any cloud. More examples for various open models, such as Llama-3, Mixtral, etc, can be found in SkyPilot AI gallery.

Prerequisites

- Go to the HuggingFace model page and request access to the model `meta-llama/Meta-Llama-3-8B-Instruct`.
- Check that you have installed SkyPilot (docs).
- Check that `sky check` shows clouds or Kubernetes are enabled.

```
pip install skypilot-nightly
sky check
```

Run on a single instance

See the vLLM SkyPilot YAML for serving, serving.yaml.

```
resources:
    accelerators: {L4, A10g, A10, L40, A40, A100, A100-80GB} # We can use cheaper␣
↪accelerators for 8B model.
    use_spot: True
    disk_size: 512  # Ensure model checkpoints can fit.
    disk_tier: best
    ports: 8081  # Expose to internet traffic.

envs:
    MODEL_NAME: meta-llama/Meta-Llama-3-8B-Instruct
    HF_TOKEN: <your-huggingface-token>  # Change to your own huggingface token, or use --
↪env to pass.

setup: |
    conda create -n vllm python=3.10 -y
    conda activate vllm

    pip install vllm==0.4.0.post1
    # Install Gradio for web UI.
    pip install gradio openai
    pip install flash-attn==2.5.7

run: |
    conda activate vllm
    echo 'Starting vllm api server...'
    python -u -m vllm.entrypoints.openai.api_server \
        --port 8081 \
        --model $MODEL_NAME \
        --trust-remote-code \
        --tensor-parallel-size $SKYPILOT_NUM_GPUS_PER_NODE \
        2>&1 | tee api_server.log &

    echo 'Waiting for vllm api server to start...'
    while ! `cat api_server.log | grep -q 'Uvicorn running on'`; do sleep 1; done

    echo 'Starting gradio server...'
    git clone https://github.com/vllm-project/vllm.git || true
```

(continues on next page)

(continued from previous page)

```
    python vllm/examples/gradio_openai_chatbot_webserver.py \
        -m $MODEL_NAME \
        --port 8811 \
        --model-url http://localhost:8081/v1 \
        --stop-token-ids 128009,128001
```

Start the serving the Llama-3 8B model on any of the candidate GPUs listed (L4, A10g, . . .):

```
HF_TOKEN="your-huggingface-token" sky launch serving.yaml --env HF_TOKEN
```

Check the output of the command. There will be a shareable gradio link (like the last line of the following). Open it in your browser to use the LLaMA model to do the text completion.

```
(task, pid=7431) Running on public URL: https://<gradio-hash>.gradio.live
```

Optional: Serve the 70B model instead of the default 8B and use more GPU:

```
HF_TOKEN="your-huggingface-token" sky launch serving.yaml --gpus A100:8 --env HF_TOKEN --
↪env MODEL_NAME=meta-llama/Meta-Llama-3-70B-Instruct
```

Scale up to multiple replicas

SkyPilot can scale up the service to multiple service replicas with built-in autoscaling, load-balancing and fault-tolerance. You can do it by adding a services section to the YAML file.

```
service:
    replicas: 2
    # An actual request for readiness probe.
    readiness_probe:
        path: /v1/chat/completions
        post_data:
        model: $MODEL_NAME
        messages:
            - role: user
            content: Hello! What is your name?
    max_tokens: 1
```

```
service:
    replicas: 2
    # An actual request for readiness probe.
    readiness_probe:
        path: /v1/chat/completions
        post_data:
        model: $MODEL_NAME
        messages:
            - role: user
            content: Hello! What is your name?
    max_tokens: 1

resources:
    accelerators: {L4, A10g, A10, L40, A40, A100, A100-80GB} # We can use cheaper␣
```

(continues on next page)

(continued from previous page)

```
↪accelerators for 8B model.
    use_spot: True
    disk_size: 512  # Ensure model checkpoints can fit.
    disk_tier: best
    ports: 8081  # Expose to internet traffic.

envs:
    MODEL_NAME: meta-llama/Meta-Llama-3-8B-Instruct
    HF_TOKEN: <your-huggingface-token>  # Change to your own huggingface token, or use --
↪env to pass.

setup: |
    conda create -n vllm python=3.10 -y
    conda activate vllm

    pip install vllm==0.4.0.post1
    # Install Gradio for web UI.
    pip install gradio openai
    pip install flash-attn==2.5.7

run: |
    conda activate vllm
    echo 'Starting vllm api server...'
    python -u -m vllm.entrypoints.openai.api_server \
        --port 8081 \
        --model $MODEL_NAME \
        --trust-remote-code \
        --tensor-parallel-size $SKYPILOT_NUM_GPUS_PER_NODE \
        2>&1 | tee api_server.log &

    echo 'Waiting for vllm api server to start...'
    while ! `cat api_server.log | grep -q 'Uvicorn running on'`; do sleep 1; done

    echo 'Starting gradio server...'
    git clone https://github.com/vllm-project/vllm.git || true
    python vllm/examples/gradio_openai_chatbot_webserver.py \
        -m $MODEL_NAME \
        --port 8811 \
        --model-url http://localhost:8081/v1 \
        --stop-token-ids 128009,128001
```

Start the serving the Llama-3 8B model on multiple replicas:

```
HF_TOKEN="your-huggingface-token" sky serve up -n vllm serving.yaml --env HF_TOKEN
```

Wait until the service is ready:

```
watch -n10 sky serve status vllm
```

```
Services
NAME  VERSION  UPTIME  STATUS  REPLICAS  ENDPOINT
vllm  1        35s     READY   2/2       xx.yy.zz.100:30001
```

(continues on next page)

(continued from previous page)

```
Service Replicas
SERVICE_NAME  ID  VERSION  IP            LAUNCHED     RESOURCES           STATUS  REGION
vllm          1   1        xx.yy.zz.121  18 mins ago  1x GCP({'L4': 1})  READY   us-east4
vllm          2   1        xx.yy.zz.245  18 mins ago  1x GCP({'L4': 1})  READY   us-east4
```

After the service is READY, you can find a single endpoint for the service and access the service with the endpoint:

```
ENDPOINT=$(sky serve status --endpoint 8081 vllm)
curl -L http://$ENDPOINT/v1/chat/completions \
    -H "Content-Type: application/json" \
    -d '{
        "model": "meta-llama/Meta-Llama-3-8B-Instruct",
        "messages": [
        {
            "role": "system",
            "content": "You are a helpful assistant."
        },
        {
            "role": "user",
            "content": "Who are you?"
        }
        ],
        "stop_token_ids": [128009,  128001]
    }'
```

To enable autoscaling, you could specify additional configs in *services*:

```
services:
    replica_policy:
        min_replicas: 0
        max_replicas: 3
    target_qps_per_replica: 2
```

This will scale the service up to when the QPS exceeds 2 for each replica.

Optional: Connect a GUI to the endpoint

It is also possible to access the Llama-3 service with a separate GUI frontend, so the user requests send to the GUI will be load-balanced across replicas.

```
envs:
    MODEL_NAME: meta-llama/Meta-Llama-3-70B-Instruct
    ENDPOINT: x.x.x.x:3031 # Address of the API server running vllm.

resources:
    cpus: 2

setup: |
    conda activate vllm
    if [ $? -ne 0 ]; then
        conda create -n vllm python=3.10 -y
```

(continues on next page)

(continued from previous page)

```
        conda activate vllm
    fi

    # Install Gradio for web UI.
    pip install gradio openai

run: |
    conda activate vllm
    export PATH=$PATH:/sbin
    WORKER_IP=$(hostname -I | cut -d' ' -f1)
    CONTROLLER_PORT=21001
    WORKER_PORT=21002

    echo 'Starting gradio server...'
    git clone https://github.com/vllm-project/vllm.git || true
    python vllm/examples/gradio_openai_chatbot_webserver.py \
        -m $MODEL_NAME \
        --port 8811 \
        --model-url http://$ENDPOINT/v1 \
        --stop-token-ids 128009,128001 | tee ~/gradio.log
```

1. Start the chat web UI:

```
sky launch -c gui ./gui.yaml --env ENDPOINT=$(sky serve status --endpoint vllm)
```

2. Then, we can access the GUI at the returned gradio link:

```
| INFO | stdout | Running on public URL: https://6141e84201ce0bb4ed.gradio.live
```

1.13.2 Deploying with KServe

vLLM can be deployed with KServe on Kubernetes for highly scalable distributed model serving.

Please see this guide for more details on using vLLM with KServe.

1.13.3 Deploying with NVIDIA Triton

The Triton Inference Server hosts a tutorial demonstrating how to quickly deploy a simple facebook/opt-125m model using vLLM. Please see Deploying a vLLM model in Triton for more details.

1.13.4 Deploying with BentoML

BentoML allows you to deploy a large language model (LLM) server with vLLM as the backend, which exposes OpenAI-compatible endpoints. You can serve the model locally or containerize it as an OCI-complicant image and deploy it on Kubernetes.

For details, see the tutorial vLLM inference in the BentoML documentation.

1.13.5 Deploying with LWS

LeaderWorkerSet (LWS) is a Kubernetes API that aims to address common deployment patterns of AI/ML inference workloads. A major use case is for multi-host/multi-node distributed inference.

vLLM can be deployed with LWS on Kubernetes for distributed model serving.

Please see this guide for more details on deploying vLLM on Kubernetes using LWS.

1.13.6 Deploying with dstack

vLLM can be run on a cloud based GPU machine with dstack, an open-source framework for running LLMs on any cloud. This tutorial assumes that you have already configured credentials, gateway, and GPU quotas on your cloud environment.

To install dstack client, run:

```
$ pip install "dstack[all]
$ dstack server
```

Next, to configure your dstack project, run:

```
$ mkdir -p vllm-dstack
$ cd vllm-dstack
$ dstack init
```

Next, to provision a VM instance with LLM of your choice(*NousResearch/Llama-2-7b-chat-hf* for this example), create the following *serve.dstack.yml* file for the dstack *Service*:

```
type: service

python: "3.11"
env:
    - MODEL=NousResearch/Llama-2-7b-chat-hf
port: 8000
resources:
    gpu: 24GB
commands:
    - pip install vllm
    - python -m vllm.entrypoints.openai.api_server --model $MODEL --port 8000
model:
    format: openai
    type: chat
    name: NousResearch/Llama-2-7b-chat-hf
```

Then, run the following CLI for provisioning:

```
$ dstack run . -f serve.dstack.yml

 Getting run plan...
 Configuration  serve.dstack.yml
 Project        deep-diver-main
 User           deep-diver
 Min resources  2..xCPU, 8GB.., 1xGPU (24GB)
 Max price      -
```

(continues on next page)

(continued from previous page)

```
 Max duration   -
 Spot policy    auto
 Retry policy   no

 #  BACKEND  REGION       INSTANCE       RESOURCES                          SPOT ␣
↪PRICE
 1  gcp   us-central1  g2-standard-4  4xCPU, 16GB, 1xL4 (24GB), 100GB (disk)  yes   $0.
↪223804
 2  gcp   us-east1     g2-standard-4  4xCPU, 16GB, 1xL4 (24GB), 100GB (disk)  yes   $0.
↪223804
 3  gcp   us-west1     g2-standard-4  4xCPU, 16GB, 1xL4 (24GB), 100GB (disk)  yes   $0.
↪223804
    ...
 Shown 3 of 193 offers, $5.876 max

Continue? [y/n]: y
 Submitting run...
 Launching spicy-treefrog-1 (pulling)
spicy-treefrog-1 provisioning completed (running)
Service is published at ...
```

After the provisioning, you can interact with the model by using the OpenAI SDK:

```
from openai import OpenAI

client = OpenAI(
    base_url="https://gateway.<gateway domain>",
    api_key="<YOUR-DSTACK-SERVER-ACCESS-TOKEN>"
)

completion = client.chat.completions.create(
    model="NousResearch/Llama-2-7b-chat-hf",
    messages=[
        {
            "role": "user",
            "content": "Compose a poem that explains the concept of recursion in␣
↪programming.",
        }
    ]
)

print(completion.choices[0].message.content)
```

Note: dstack automatically handles authentication on the gateway using dstack's tokens. Meanwhile, if you don't want to configure a gateway, you can provision dstack *Task* instead of *Service*. The *Task* is for development purpose only. If you want to know more about hands-on materials how to serve vLLM using dstack, check out this repository

1.13.7 Serving with Langchain

vLLM is also available via Langchain .

To install langchain, run

```
$ pip install langchain langchain_community -q
```

To run inference on a single or multiple GPUs, use VLLM class from langchain.

```
from langchain_community.llms import VLLM

llm = VLLM(model="mosaicml/mpt-7b",
           trust_remote_code=True,  # mandatory for hf models
           max_new_tokens=128,
           top_k=10,
           top_p=0.95,
           temperature=0.8,
           # tensor_parallel_size=... # for distributed inference
)

print(llm("What is the capital of France ?"))
```

Please refer to this Tutorial for more details.

1.14 Supported Models

vLLM supports a variety of generative Transformer models in HuggingFace Transformers. The following is the list of model architectures that are currently supported by vLLM. Alongside each architecture, we include some popular models that use it.

Architecture	Models	Example HuggingFace Models	*LoRA*
AquilaForCausalLM	Aquila & Aquila2	BAAI/Aquila-7B, BAAI/AquilaChat-7B, etc.	
ArcticForCausalLM	Arctic	Snowflake/snowflake-arctic-base, Snowflake/snowflake-arctic-instruct, etc.	
BaiChuanForCausalLM	Baichuan & Baichuan2	baichuan-inc/Baichuan2-13B-Chat, baichuan-inc/Baichuan-7B, etc.	
BloomForCausalLM	BLOOM, BLOOMZ, BLOOMChat	bigscience/bloom, bigscience/bloomz, etc.	
ChatGLMModel	ChatGLM	THUDM/chatglm2-6b, THUDM/chatglm3-6b, etc.	
CohereForCausalLM	Command-R	CohereForAI/c4ai-command-r-v01, etc.	
DbrxForCausalLM	DBRX	databricks/dbrx-base, databricks/dbrx-instruct, etc.	
DeciLMForCausalLM	DeciLM	Deci/DeciLM-7B, Deci/DeciLM-7B-instruct, etc.	
FalconForCausalLM	Falcon	tiiuae/falcon-7b, tiiuae/falcon-40b, tiiuae/falcon-rw-7b, etc.	
GemmaForCausalLM	Gemma	google/gemma-2b, google/gemma-7b, etc.	
GPT2LMHeadModel	GPT-2	gpt2, gpt2-xl, etc.	

continues on next page

Table 1 – continued from previous page

Architecture	Models	Example HuggingFace Models	LoRA
GPTBigCodeForCausalL	StarCoder, SantaCoder, WizardCoder	bigcode/starcoder, bigcode/gpt_bigcode-santacoder, WizardLM/WizardCoder-15B-V1.0, etc.	
GPTJForCausalLM	GPT-J	EleutherAI/gpt-j-6b, nomic-ai/gpt4all-j, etc.	
GPTNeoXForCausalLM	GPT-NeoX, Pythia, OpenAssistant, Dolly V2, StableLM	EleutherAI/gpt-neox-20b, EleutherAI/pythia-12b, OpenAssistant/oasst-sft-4-pythia-12b-epoch-3.5, databricks/dolly-v2-12b, stabilityai/stablelm-tuned-alpha-7b, etc.	
InternLMForCausalLM	InternLM	internlm/internlm-7b, internlm/internlm-chat-7b, etc.	
InternLM2ForCausalLM	InternLM2	internlm/internlm2-7b, internlm/internlm2-chat-7b, etc.	
JAISLMHeadModel	Jais	core42/jais-13b, core42/jais-13b-chat, core42/jais-30b-v3, core42/jais-30b-chat-v3, etc.	
LlamaForCausalLM	LLaMA, Llama 2, Meta Llama 3, Vicuna, Alpaca, Yi	meta-llama/Meta-Llama-3-8B-Instruct, meta-llama/Meta-Llama-3-70B-Instruct, meta-llama/Llama-2-13b-hf, meta-llama/Llama-2-70b-hf, openlm-research/open_llama_13b, lmsys/vicuna-13b-v1.3, 01-ai/Yi-6B, 01-ai/Yi-34B, etc.	
MiniCPMForCausalLM	MiniCPM	openbmb/MiniCPM-2B-sft-bf16, openbmb/MiniCPM-2B-dpo-bf16, etc.	
MistralForCausalLM	Mistral, Mistral-Instruct	mistralai/Mistral-7B-v0.1, mistralai/Mistral-7B-Instruct-v0.1, etc.	
MixtralForCausalLM	Mixtral-8x7B, Mixtral-8x7B-Instruct	mistralai/Mixtral-8x7B-v0.1, mistralai/Mixtral-8x7B-Instruct-v0.1, mistral-community/Mixtral-8x22B-v0.1, etc.	
MPTForCausalLM	MPT, MPT-Instruct, MPT-Chat, MPT-StoryWriter	mosaicml/mpt-7b, mosaicml/mpt-7b-storywriter, mosaicml/mpt-30b, etc.	
OLMoForCausalLM	OLMo	allenai/OLMo-1B-hf, allenai/OLMo-7B-hf, etc.	
OPTForCausalLM	OPT, OPT-IML	facebook/opt-66b, facebook/opt-iml-max-30b, etc.	
OrionForCausalLM	Orion	OrionStarAI/Orion-14B-Base, OrionStarAI/Orion-14B-Chat, etc.	
PhiForCausalLM	Phi	microsoft/phi-1_5, microsoft/phi-2, etc.	
Phi3ForCausalLM	Phi-3	microsoft/Phi-3-mini-4k-instruct, microsoft/Phi-3-mini-128k-instruct, etc.	
Phi3SmallForCausalLM	Phi-3-Small	microsoft/Phi-3-small-8k-instruct, microsoft/Phi-3-small-128k-instruct, etc.	
QWenLMHeadModel	Qwen	Qwen/Qwen-7B, Qwen/Qwen-7B-Chat, etc.	
Qwen2ForCausalLM	Qwen2	Qwen/Qwen2-beta-7B, Qwen/Qwen2-beta-7B-Chat, etc.	
Qwen2MoeForCausalLM	Qwen2MoE	Qwen/Qwen1.5-MoE-A2.7B, Qwen/Qwen1.5-MoE-A2.7B-Chat, etc.	

continues on next page

Table 1 – continued from previous page

Architecture	Models	Example HuggingFace Models	LoRA
StableLmForCausalLM	StableLM	stabilityai/stablelm-3b-4e1t/ , stabilityai/stablelm-base-alpha-7b-v2, etc.	
Starcoder2ForCausalL	Starcoder2	bigcode/starcoder2-3b, bigcode/starcoder2-7b, bigcode/starcoder2-15b, etc.	
XverseForCausalLM	Xverse	xverse/XVERSE-7B-Chat, xverse/XVERSE-13B-Chat, xverse/XVERSE-65B-Chat, etc.	

If your model uses one of the above model architectures, you can seamlessly run your model with vLLM. Otherwise, please refer to *Adding a New Model* for instructions on how to implement support for your model. Alternatively, you can raise an issue on our GitHub project.

Note: Currently, the ROCm version of vLLM supports Mistral and Mixtral only for context lengths up to 4096.

Tip: The easiest way to check if your model is supported is to run the program below:

```
from vllm import LLM

llm = LLM(model=...)  # Name or path of your model
output = llm.generate("Hello, my name is")
print(output)
```

If vLLM successfully generates text, it indicates that your model is supported.

Tip: To use models from ModelScope instead of HuggingFace Hub, set an environment variable:

```
$ export VLLM_USE_MODELSCOPE=True
```

And use with `trust_remote_code=True`.

```
from vllm import LLM

llm = LLM(model=..., revision=..., trust_remote_code=True)  # Name or path of your model
output = llm.generate("Hello, my name is")
print(output)
```

1.14.1 Model Support Policy

At vLLM, we are committed to facilitating the integration and support of third-party models within our ecosystem. Our approach is designed to balance the need for robustness and the practical limitations of supporting a wide range of models. Here's how we manage third-party model support:

1. **Community-Driven Support**: We encourage community contributions for adding new models. When a user requests support for a new model, we welcome pull requests (PRs) from the community. These contributions are evaluated primarily on the sensibility of the output they generate, rather than strict consistency with existing implementations such as those in transformers. **Call for contribution:** PRs coming directly from model vendors are greatly appreciated!
2. **Best-Effort Consistency**: While we aim to maintain a level of consistency between the models implemented in vLLM and other frameworks like transformers, complete alignment is not always feasible. Factors like acceleration techniques and the use of low-precision computations can introduce discrepancies. Our commitment is to ensure that the implemented models are functional and produce sensible results.
3. **Issue Resolution and Model Updates**: Users are encouraged to report any bugs or issues they encounter with third-party models. Proposed fixes should be submitted via PRs, with a clear explanation of the problem and the rationale behind the proposed solution. If a fix for one model impacts another, we rely on the community to highlight and address these cross-model dependencies. Note: for bugfix PRs, it is good etiquette to inform the original author to seek their feedback.
4. **Monitoring and Updates**: Users interested in specific models should monitor the commit history for those models (e.g., by tracking changes in the main/vllm/model_executor/models directory). This proactive approach helps users stay informed about updates and changes that may affect the models they use.
5. **Selective Focus**: Our resources are primarily directed towards models with significant user interest and impact. Models that are less frequently used may receive less attention, and we rely on the community to play a more active role in their upkeep and improvement.

Through this approach, vLLM fosters a collaborative environment where both the core development team and the broader community contribute to the robustness and diversity of the third-party models supported in our ecosystem.

Note that, as an inference engine, vLLM does not introduce new models. Therefore, all models supported by vLLM are third-party models in this regard.

We have the following levels of testing for models:

1. **Strict Consistency**: We compare the output of the model with the output of the model in the HuggingFace Transformers library under greedy decoding. This is the most stringent test. Please refer to test_models.py and test_big_models.py for the models that have passed this test.
2. **Output Sensibility**: We check if the output of the model is sensible and coherent, by measuring the perplexity of the output and checking for any obvious errors. This is a less stringent test.
3. **Runtime Functionality**: We check if the model can be loaded and run without errors. This is the least stringent test. Please refer to functionality tests and examples for the models that have passed this test.
4. **Community Feedback**: We rely on the community to provide feedback on the models. If a model is broken or not working as expected, we encourage users to raise issues to report it or open pull requests to fix it. The rest of the models fall under this category.

1.15 Adding a New Model

This document provides a high-level guide on integrating a HuggingFace Transformers model into vLLM.

Note: The complexity of adding a new model depends heavily on the model's architecture. The process is considerably straightforward if the model shares a similar architecture with an existing model in vLLM. However, for models that include new operators (e.g., a new attention mechanism), the process can be a bit more complex.

Tip: If you are encountering issues while integrating your model into vLLM, feel free to open an issue on our GitHub repository. We will be happy to help you out!

1.15.1 0. Fork the vLLM repository

Start by forking our GitHub repository and then *build it from source*. This gives you the ability to modify the codebase and test your model.

Tip: If you don't want to fork the repository and modify vLLM's codebase, please refer to the "Out-of-Tree Model Integration" section below.

1.15.2 1. Bring your model code

Clone the PyTorch model code from the HuggingFace Transformers repository and put it into the vllm/model_executor/models directory. For instance, vLLM's OPT model was adapted from the HuggingFace's modeling_opt.py file.

Warning: When copying the model code, make sure to review and adhere to the code's copyright and licensing terms.

1.15.3 2. Rewrite the `forward` methods

Next, you need to rewrite the `forward` methods of your model by following these steps:

1. Remove any unnecessary code, such as the code only used for training.
2. Change the input parameters:

```
def forward(
    self,
    input_ids: torch.Tensor,
-    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,
-    labels: Optional[torch.LongTensor] = None,
-    use_cache: Optional[bool] = None,
```

(continues on next page)

(continued from previous page)

```
-    output_attentions: Optional[bool] = None,
-    output_hidden_states: Optional[bool] = None,
-    return_dict: Optional[bool] = None,
-) -> Union[Tuple, CausalLMOutputWithPast]:
+    positions: torch.Tensor,
+    kv_caches: List[torch.Tensor],
+    attn_metadata: AttentionMetadata,
+) -> Optional[SamplerOutput]:
```

1. Update the code by considering that `input_ids` and `positions` are now flattened tensors.
2. Replace the attention operation with either `PagedAttention`, `PagedAttentionWithRoPE`, or `PagedAttentionWithALiBi` depending on the model's architecture.

Note: Currently, vLLM supports the basic multi-head attention mechanism and its variant with rotary positional embeddings. If your model employs a different attention mechanism, you will need to implement a new attention layer in vLLM.

1.15.4 3. (Optional) Implement tensor parallelism and quantization support

If your model is too large to fit into a single GPU, you can use tensor parallelism to manage it. To do this, substitute your model's linear and embedding layers with their tensor-parallel versions. For the embedding layer, you can simply replace `nn.Embedding` with `VocabParallelEmbedding`. For the output LM head, you can use `ParallelLMHead`. When it comes to the linear layers, we provide the following options to parallelize them:

- `ReplicatedLinear`: Replicates the inputs and weights across multiple GPUs. No memory saving.
- `RowParallelLinear`: The input tensor is partitioned along the hidden dimension. The weight matrix is partitioned along the rows (input dimension). An *all-reduce* operation is performed after the matrix multiplication to reduce the results. Typically used for the second FFN layer and the output linear transformation of the attention layer.
- `ColumnParallelLinear`: The input tensor is replicated. The weight matrix is partitioned along the columns (output dimension). The result is partitioned along the column dimension. Typically used for the first FFN layer and the separated QKV transformation of the attention layer in the original Transformer.
- `MergedColumnParallelLinear`: Column-parallel linear that merges multiple *ColumnParallelLinear* operators. Typically used for the first FFN layer with weighted activation functions (e.g., SiLU). This class handles the sharded weight loading logic of multiple weight matrices.
- `QKVParallelLinear`: Parallel linear layer for the query, key, and value projections of the multi-head and grouped-query attention mechanisms. When number of key/value heads are less than the world size, this class replicates the key/value heads properly. This class handles the weight loading and replication of the weight matrices.

Note that all the linear layers above take *linear_method* as an input. vLLM will set this parameter according to different quantization schemes to support weight quantization.

1.15.5 4. Implement the weight loading logic

You now need to implement the `load_weights` method in your `*ForCausalLM` class. This method should load the weights from the HuggingFace's checkpoint file and assign them to the corresponding layers in your model. Specifically, for *MergedColumnParallelLinear* and *QKVParallelLinear* layers, if the original model has separated weight matrices, you need to load the different parts separately.

1.15.6 5. Register your model

Finally, register your `*ForCausalLM` class to the `_MODELS` in vllm/model_executor/models/__init__.py.

1.15.7 6. Out-of-Tree Model Integration

We also provide a way to integrate a model without modifying the vLLM codebase. Step 2, 3, 4 are still required, but you can skip step 1 and 5.

Just add the following lines in your code:

```
from vllm import ModelRegistry
from your_code import YourModelForCausalLM
ModelRegistry.register_model("YourModelForCausalLM", YourModelForCausalLM)
```

If you are running api server with *python -m vllm.entrypoints.openai.api_server args*, you can wrap the entrypoint with the following code:

```
from vllm import ModelRegistry
from your_code import YourModelForCausalLM
ModelRegistry.register_model("YourModelForCausalLM", YourModelForCausalLM)
import runpy
runpy.run_module('vllm.entrypoints.openai.api_server', run_name='__main__')
```

Save the above code in a file and run it with *python your_file.py args*.

1.16 Engine Arguments

Below, you can find an explanation of every engine argument for vLLM:

```
usage: -m vllm.entrypoints.openai.api_server [-h] [--model MODEL]
                                             [--tokenizer TOKENIZER]
                                             [--skip-tokenizer-init]
                                             [--revision REVISION]
                                             [--code-revision CODE_REVISION]
                                             [--tokenizer-revision TOKENIZER_REVISION]
                                             [--tokenizer-mode {auto,slow}]
                                             [--trust-remote-code]
                                             [--download-dir DOWNLOAD_DIR]
                                             [--load-format {auto,pt,safetensors,npcache,
↪dummy,tensorizer}]
                                             [--dtype {auto,half,float16,bfloat16,float,
↪float32}]
                                             [--kv-cache-dtype {auto,fp8,fp8_e5m2,fp8_
```

(continues on next page)

(continued from previous page)

```
↪e4m3}]
                                             [--quantization-param-path QUANTIZATION_
↪PARAM_PATH]
                                             [--max-model-len MAX_MODEL_LEN]
                                             [--guided-decoding-backend {outlines,lm-
↪format-enforcer}]
                                             [--distributed-executor-backend {ray,mp}]
                                             [--worker-use-ray]
                                             [--pipeline-parallel-size PIPELINE_PARALLEL_
↪SIZE]
                                             [--tensor-parallel-size TENSOR_PARALLEL_
↪SIZE]
                                             [--max-parallel-loading-workers MAX_
↪PARALLEL_LOADING_WORKERS]
                                             [--ray-workers-use-nsight]
                                             [--block-size {8,16,32}]
                                             [--enable-prefix-caching]
                                             [--disable-sliding-window]
                                             [--use-v2-block-manager]
                                             [--num-lookahead-slots NUM_LOOKAHEAD_SLOTS]
                                             [--seed SEED]
                                             [--swap-space SWAP_SPACE]
                                             [--gpu-memory-utilization GPU_MEMORY_
↪UTILIZATION]
                                             [--num-gpu-blocks-override NUM_GPU_BLOCKS_
↪OVERRIDE]
                                             [--max-num-batched-tokens MAX_NUM_BATCHED_
↪TOKENS]
                                             [--max-num-seqs MAX_NUM_SEQS]
                                             [--max-logprobs MAX_LOGPROBS]
                                             [--disable-log-stats]
                                             [--quantization {aqlm,awq,deepspeedfp,fp8,
↪marlin,gptq_marlin_24,gptq_marlin,gptq,squeezellm,sparseml,None}]
                                             [--rope-scaling ROPE_SCALING]
                                             [--enforce-eager]
                                             [--max-context-len-to-capture MAX_CONTEXT_
↪LEN_TO_CAPTURE]
                                             [--max-seq-len-to-capture MAX_SEQ_LEN_TO_
↪CAPTURE]
                                             [--disable-custom-all-reduce]
                                             [--tokenizer-pool-size TOKENIZER_POOL_SIZE]
                                             [--tokenizer-pool-type TOKENIZER_POOL_TYPE]
                                             [--tokenizer-pool-extra-config TOKENIZER_
↪POOL_EXTRA_CONFIG]
                                             [--enable-lora]
                                             [--max-loras MAX_LORAS]
                                             [--max-lora-rank MAX_LORA_RANK]
                                             [--lora-extra-vocab-size LORA_EXTRA_VOCAB_
↪SIZE]
                                             [--lora-dtype {auto,float16,bfloat16,
↪float32}]
                                             [--long-lora-scaling-factors LONG_LORA_
```

(continues on next page)

(continued from previous page)

```
↪SCALING_FACTORS]
                                             [--max-cpu-loras MAX_CPU_LORAS]
                                             [--fully-sharded-loras]
                                             [--device {auto,cuda,neuron,cpu}]
                                             [--image-input-type {pixel_values,image_
↪features}]
                                             [--image-token-id IMAGE_TOKEN_ID]
                                             [--image-input-shape IMAGE_INPUT_SHAPE]
                                             [--image-feature-size IMAGE_FEATURE_SIZE]
                                             [--scheduler-delay-factor SCHEDULER_DELAY_
↪FACTOR]
                                             [--enable-chunked-prefill]
                                             [--speculative-model SPECULATIVE_MODEL]
                                             [--num-speculative-tokens NUM_SPECULATIVE_
↪TOKENS]
                                             [--speculative-max-model-len SPECULATIVE_
↪MAX_MODEL_LEN]
                                             [--speculative-disable-by-batch-size␣
↪SPECULATIVE_DISABLE_BY_BATCH_SIZE]
                                             [--ngram-prompt-lookup-max NGRAM_PROMPT_
↪LOOKUP_MAX]
                                             [--ngram-prompt-lookup-min NGRAM_PROMPT_
↪LOOKUP_MIN]
                                             [--model-loader-extra-config MODEL_LOADER_
↪EXTRA_CONFIG]
                                             [--served-model-name SERVED_MODEL_NAME␣
↪[SERVED_MODEL_NAME ...]]
```

1.16.1 Named Arguments

--model	Name or path of the huggingface model to use. Default: "facebook/opt-125m"
--tokenizer	Name or path of the huggingface tokenizer to use.
--skip-tokenizer-init	Skip initialization of tokenizer and detokenizer
--revision	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.
--code-revision	The specific revision to use for the model code on Hugging Face Hub. It can be a branch name, a tag name, or a commit id. If unspecified, will use the default version.
--tokenizer-revision	The specific tokenizer version to use. It can be a branch name, a tag name, or a commit id. If unspecified, will use the default version.
--tokenizer-mode	Possible choices: auto, slow The tokenizer mode. • "auto" will use the fast tokenizer if available. • "slow" will always use the slow tokenizer. Default: "auto"

--trust-remote-code Trust remote code from huggingface.

--download-dir Directory to download and load the weights, default to the default cache dir of huggingface.

--load-format Possible choices: auto, pt, safetensors, npcache, dummy, tensorizer

The format of the model weights to load.

- "auto" will try to load the weights in the safetensors format and fall back to the pytorch bin format if safetensors format is not available.
- "pt" will load the weights in the pytorch bin format.
- "safetensors" will load the weights in the safetensors format.
- "npcache" will load the weights in pytorch format and store a numpy cache to speed up the loading.
- "dummy" will initialize the weights with random values, which is mainly for profiling.
- "tensorizer" will load the weights using tensorizer from CoreWeave. See the Tensorize vLLM Model script in the Examplessection for more information.

Default: "auto"

--dtype Possible choices: auto, half, float16, bfloat16, float, float32

Data type for model weights and activations.

- "auto" will use FP16 precision for FP32 and FP16 models, and BF16 precision for BF16 models.
- "half" for FP16. Recommended for AWQ quantization.
- "float16" is the same as "half".
- "bfloat16" for a balance between precision and range.
- "float" is shorthand for FP32 precision.
- "float32" for FP32 precision.

Default: "auto"

--kv-cache-dtype Possible choices: auto, fp8, fp8_e5m2, fp8_e4m3

Data type for kv cache storage. If "auto", will use model data type. CUDA 11.8+ supports fp8 (=fp8_e4m3) and fp8_e5m2. ROCm (AMD GPU) supports fp8 (=fp8_e4m3)

Default: "auto"

--quantization-param-path Path to the JSON file containing the KV cache scaling factors. This should generally be supplied, when KV cache dtype is FP8. Otherwise, KV cache scaling factors default to 1.0, which may cause accuracy issues. FP8_E5M2 (without scaling) is only supported on cuda versiongreater than 11.8. On ROCm (AMD GPU), FP8_E4M3 is instead supported for common inference criteria.

--max-model-len Model context length. If unspecified, will be automatically derived from the model config.

--guided-decoding-backend Possible choices: outlines, lm-format-enforcer

Which engine will be used for guided decoding (JSON schema / regex etc) by default. Currently support https://github.com/outlines-dev/outlines and https:

//github.com/noamgat/lm-format-enforcer. Can be overridden per request via guided_decoding_backend parameter.

Default: “outlines”

--distributed-executor-backend Possible choices: ray, mp

Backend to use for distributed serving. When more than 1 GPU is used, will be automatically set to “ray” if installed or “mp” (multiprocessing) otherwise.

--worker-use-ray Deprecated, use –distributed-executor-backend=ray.

--pipeline-parallel-size, -pp Number of pipeline stages.

Default: 1

--tensor-parallel-size, -tp Number of tensor parallel replicas.

Default: 1

--max-parallel-loading-workers Load model sequentially in multiple batches, to avoid RAM OOM when using tensor parallel and large models.

--ray-workers-use-nsight If specified, use nsight to profile Ray workers.

--block-size Possible choices: 8, 16, 32

Token block size for contiguous chunks of tokens.

Default: 16

--enable-prefix-caching Enables automatic prefix caching.

--disable-sliding-window Disables sliding window, capping to sliding window size

--use-v2-block-manager Use BlockSpaceMangerV2.

--num-lookahead-slots Experimental scheduling config necessary for speculative decoding. This will be replaced by speculative config in the future; it is present to enable correctness tests until then.

Default: 0

--seed Random seed for operations.

Default: 0

--swap-space CPU swap space size (GiB) per GPU.

Default: 4

--gpu-memory-utilization The fraction of GPU memory to be used for the model executor, which can range from 0 to 1. For example, a value of 0.5 would imply 50% GPU memory utilization. If unspecified, will use the default value of 0.9.

Default: 0.9

--num-gpu-blocks-override If specified, ignore GPU profiling result and use this numberof GPU blocks. Used for testing preemption.

--max-num-batched-tokens Maximum number of batched tokens per iteration.

--max-num-seqs Maximum number of sequences per iteration.

Default: 256

--max-logprobs Max number of log probs to return logprobs is specified in SamplingParams.

Default: 5

--disable-log-stats Disable logging statistics.

--quantization, -q Possible choices: aqlm, awq, deepspeedfp, fp8, marlin, gptq_marlin_24, gptq_marlin, gptq, squeezellm, sparseml, None

Method used to quantize the weights. If None, we first check the *quantization_config* attribute in the model config file. If that is None, we assume the model weights are not quantized and use *dtype* to determine the data type of the weights.

--rope-scaling RoPE scaling configuration in JSON format. For example, {“type”:”dynamic”,”factor”:2.0}

--enforce-eager Always use eager-mode PyTorch. If False, will use eager mode and CUDA graph in hybrid for maximal performance and flexibility.

--max-context-len-to-capture Maximum context length covered by CUDA graphs. When a sequence has context length larger than this, we fall back to eager mode. (DEPRECATED. Use –max-seq-len-to-capture instead)

--max-seq-len-to-capture Maximum sequence length covered by CUDA graphs. When a sequence has context length larger than this, we fall back to eager mode.

Default: 8192

--disable-custom-all-reduce See ParallelConfig.

--tokenizer-pool-size Size of tokenizer pool to use for asynchronous tokenization. If 0, will use synchronous tokenization.

Default: 0

--tokenizer-pool-type Type of tokenizer pool to use for asynchronous tokenization. Ignored if tokenizer_pool_size is 0.

Default: “ray”

--tokenizer-pool-extra-config Extra config for tokenizer pool. This should be a JSON string that will be parsed into a dictionary. Ignored if tokenizer_pool_size is 0.

--enable-lora If True, enable handling of LoRA adapters.

--max-loras Max number of LoRAs in a single batch.

Default: 1

--max-lora-rank Max LoRA rank.

Default: 16

--lora-extra-vocab-size Maximum size of extra vocabulary that can be present in a LoRA adapter (added to the base model vocabulary).

Default: 256

--lora-dtype Possible choices: auto, float16, bfloat16, float32

Data type for LoRA. If auto, will default to base model dtype.

Default: “auto”

--long-lora-scaling-factors Specify multiple scaling factors (which can be different from base model scaling factor - see eg. Long LoRA) to allow for multiple LoRA adapters trained with those scaling factors to be used at the same time. If not specified, only adapters trained with the base model scaling factor are allowed.

--max-cpu-loras Maximum number of LoRAs to store in CPU memory. Must be >= than max_num_seqs. Defaults to max_num_seqs.

--fully-sharded-loras By default, only half of the LoRA computation is sharded with tensor parallelism. Enabling this will use the fully sharded layers. At high sequence length, max rank or tensor parallel size, this is likely faster.

--device Possible choices: auto, cuda, neuron, cpu

Device type for vLLM execution.

Default: "auto"

--image-input-type Possible choices: pixel_values, image_features

The image input type passed into vLLM. Should be one of "pixel_values" or "image_features".

--image-token-id Input id for image token.

--image-input-shape The biggest image input shape (worst for memory footprint) given an input type. Only used for vLLM's profile_run.

--image-feature-size The image feature size along the context dimension.

--scheduler-delay-factor Apply a delay (of delay factor multiplied by previousprompt latency) before scheduling next prompt.

Default: 0.0

--enable-chunked-prefill If set, the prefill requests can be chunked based on the max_num_batched_tokens.

--speculative-model The name of the draft model to be used in speculative decoding.

--num-speculative-tokens The number of speculative tokens to sample from the draft model in speculative decoding.

--speculative-max-model-len The maximum sequence length supported by the draft model. Sequences over this length will skip speculation.

--speculative-disable-by-batch-size Disable speculative decoding for new incoming requests if the number of enqueue requests is larger than this value.

--ngram-prompt-lookup-max Max size of window for ngram prompt lookup in speculative decoding.

--ngram-prompt-lookup-min Min size of window for ngram prompt lookup in speculative decoding.

--model-loader-extra-config Extra config for model loader. This will be passed to the model loader corresponding to the chosen load_format. This should be a JSON string that will be parsed into a dictionary.

--served-model-name The model name(s) used in the API. If multiple names are provided, the server will respond to any of the provided names. The model name in the model field of a response will be the first name in this list. If not specified, the model name will be the same as the *–model* argument. Noted that this name(s)will also be used in *model_name* tag content of prometheus metrics, if multiple names provided, metricstag will take the first one.

1.16.2 Async Engine Arguments

Below are the additional arguments related to the asynchronous engine:

```
usage: -m vllm.entrypoints.openai.api_server [-h] [--engine-use-ray]
                                             [--disable-log-requests]
                                             [--max-log-len MAX_LOG_LEN]
```

Named Arguments

--engine-use-ray Use Ray to start the LLM engine in a separate process as the server process.

--disable-log-requests Disable logging requests.

--max-log-len Max number of prompt characters or prompt ID numbers being printed in log.

Default: Unlimited

1.17 Using LoRA adapters

This document shows you how to use LoRA adapters with vLLM on top of a base model. Adapters can be efficiently served on a per request basis with minimal overhead. First we download the adapter(s) and save them locally with

```
from huggingface_hub import snapshot_download

sql_lora_path = snapshot_download(repo_id="yard1/llama-2-7b-sql-lora-test")
```

Then we instantiate the base model and pass in the `enable_lora=True` flag:

```
from vllm import LLM, SamplingParams
from vllm.lora.request import LoRARequest

llm = LLM(model="meta-llama/Llama-2-7b-hf", enable_lora=True)
```

We can now submit the prompts and call `llm.generate` with the `lora_request` parameter. The first parameter of `LoRARequest` is a human identifiable name, the second parameter is a globally unique ID for the adapter and the third parameter is the path to the LoRA adapter.

```
sampling_params = SamplingParams(
    temperature=0,
    max_tokens=256,
    stop=["[/assistant]"]
)

prompts = [
     "[user] Write a SQL query to answer the question based on the table schema.\n\n␣
↪context: CREATE TABLE table_name_74 (icao VARCHAR, airport VARCHAR)\n\n question: Name␣
↪the ICAO for lilongwe international airport [/user] [assistant]",
     "[user] Write a SQL query to answer the question based on the table schema.\n\n␣
↪context: CREATE TABLE table_name_11 (nationality VARCHAR, elector VARCHAR)\n\n␣
↪question: When Anchero Pantaleone was the elector what is under nationality? [/user]␣
↪[assistant]",
]
```

(continues on next page)

(continued from previous page)

```
outputs = llm.generate(
    prompts,
    sampling_params,
    lora_request=LoRARequest("sql_adapter", 1, sql_lora_path)
)
```

Check out examples/multilora_inference.py for an example of how to use LoRA adapters with the async engine and how to use more advanced configuration options.

1.17.1 Serving LoRA Adapters

LoRA adapted models can also be served with the Open-AI compatible vLLM server. To do so, we use `--lora-modules {name}={path} {name}={path}` to specify each LoRA module when we kickoff the server:

```
python -m vllm.entrypoints.openai.api_server \
    --model meta-llama/Llama-2-7b-hf \
    --enable-lora \
    --lora-modules sql-lora=~/.cache/huggingface/hub/models--yard1--llama-2-7b-sql-lora-
↪test/
```

The server entrypoint accepts all other LoRA configuration parameters (`max_loras`, `max_lora_rank`, `max_cpu_loras`, etc.), which will apply to all forthcoming requests. Upon querying the `/models` endpoint, we should see our LoRA along with its base model:

```
curl localhost:8000/v1/models | jq .
{
    "object": "list",
    "data": [
        {
            "id": "meta-llama/Llama-2-7b-hf",
            "object": "model",
            ...
        },
        {
            "id": "sql-lora",
            "object": "model",
            ...
        }
    ]
}
```

Requests can specify the LoRA adapter as if it were any other model via the `model` request parameter. The requests will be processed according to the server-wide LoRA configuration (i.e. in parallel with base model requests, and potentially other LoRA adapter requests if they were provided and `max_loras` is set high enough).

The following is an example request

```
curl http://localhost:8000/v1/completions \
    -H "Content-Type: application/json" \
    -d '{
        "model": "sql-lora",
```

(continues on next page)

(continued from previous page)

```
        "prompt": "San Francisco is a",
        "max_tokens": 7,
        "temperature": 0
    }' | jq
```

1.18 Performance and Tuning

1.18.1 Preemption

Due to the auto-regressive nature of transformer architecture, there are times when KV cache space is insufficient to handle all batched requests. The vLLM can preempt requests to free up KV cache space for other requests. Preempted requests are recomputed when sufficient KV cache space becomes available again. When this occurs, the following warning is printed:

` WARNING 05-09 00:49:33 scheduler.py:1057] Sequence group 0 is preempted by PreemptionMode.SWAP mode because there is not enough KV cache space. This can affect the end-to-end performance. Increase gpu_memory_utilization or tensor_parallel_size to provide more KV cache memory. total_cumulative_preemption_cnt=1 `

While this mechanism ensures system robustness, preemption and recomputation can adversely affect end-to-end latency. If you frequently encounter preemptions from the vLLM engine, consider the following actions:

- Increase *gpu_memory_utilization*. The vLLM pre-allocates GPU cache by using gpu_memory_utilization% of memory. By increasing this utilization, you can provide more KV cache space.
- Decrease *max_num_seqs* or *max_num_batched_tokens*. This can reduce the number of concurrent requests in a batch, thereby requiring less KV cache space.
- Increase *tensor_parallel_size*. This approach shards model weights, so each GPU has more memory available for KV cache.

You can also monitor the number of preemption requests through Prometheus metrics exposed by the vLLM. Additionally, you can log the cumulative number of preemption requests by setting disable_log_stats=False.

1.18.2 Chunked Prefill

vLLM supports an experimental feature chunked prefill. Chunked prefill allows to chunk large prefills into smaller chunks and batch them together with decode requests.

You can enable the feature by specifying `--enable-chunked-prefill` in the command line or setting `enable_chunked_prefill=True` in the LLM constructor.

```
llm = LLM(model="meta-llama/Llama-2-7b-hf", enable_chunked_prefill=True)
# Set max_num_batched_tokens to tune performance.
# NOTE: 512 is the default max_num_batched_tokens for chunked prefill.
# llm = LLM(model="meta-llama/Llama-2-7b-hf", enable_chunked_prefill=True, max_num_
↪batched_tokens=512)
```

By default, vLLM scheduler prioritizes prefills and doesn't batch prefill and decode to the same batch. This policy optimizes the TTFT (time to the first token), but incurs slower ITL (inter token latency) and inefficient GPU utilization.

Once chunked prefill is enabled, the policy is changed to prioritize decode requests. It batches all pending decode requests to the batch before scheduling any prefill. When there are available token_budget (`max_num_batched_tokens`), it schedules pending prefills. If a last pending prefill request cannot fit into `max_num_batched_tokens`, it chunks it.

This policy has two benefits:

- It improves ITL and generation decode because decode requests are prioritized.
- It helps achieve better GPU utilization by locating compute-bound (prefill) and memory-bound (decode) requests to the same batch.

You can tune the performance by changing `max_num_batched_tokens`. By default, it is set to 512, which has the best ITL on A100 in the initial benchmark (llama 70B and mixtral 8x22B). Smaller `max_num_batched_tokens` achieves better ITL because there are fewer prefills interrupting decodes. Higher `max_num_batched_tokens` achieves better TTFT as you can put more prefill to the batch.

- If `max_num_batched_tokens` is the same as `max_model_len`, that's almost the equivalent to the default scheduling policy (except that it still prioritizes decodes).
- Note that the default value (512) of `max_num_batched_tokens` is optimized for ITL, and it may have lower throughput than the default scheduler.

We recommend you set `max_num_batched_tokens > 2048` for throughput.

See related papers for more details (https://arxiv.org/pdf/2401.08671 or https://arxiv.org/pdf/2308.16369).

Please try out this feature and let us know your feedback via GitHub issues!

1.19 AutoAWQ

Warning: Please note that AWQ support in vLLM is under-optimized at the moment. We would recommend using the unquantized version of the model for better accuracy and higher throughput. Currently, you can use AWQ as a way to reduce memory footprint. As of now, it is more suitable for low latency inference with small number of concurrent requests. vLLM's AWQ implementation have lower throughput than unquantized version.

To create a new 4-bit quantized model, you can leverage AutoAWQ. Quantizing reduces the model's precision from FP16 to INT4 which effectively reduces the file size by ~70%. The main benefits are lower latency and memory usage.

You can quantize your own models by installing AutoAWQ or picking one of the 400+ models on Huggingface.

```
$ pip install autoawq
```

After installing AutoAWQ, you are ready to quantize a model. Here is an example of how to quantize Vicuna 7B v1.5:

```
from awq import AutoAWQForCausalLM
from transformers import AutoTokenizer

model_path = 'lmsys/vicuna-7b-v1.5'
quant_path = 'vicuna-7b-v1.5-awq'
quant_config = { "zero_point": True, "q_group_size": 128, "w_bit": 4, "version": "GEMM" }

# Load model
model = AutoAWQForCausalLM.from_pretrained(model_path, **{"low_cpu_mem_usage": True})
tokenizer = AutoTokenizer.from_pretrained(model_path, trust_remote_code=True)

# Quantize
model.quantize(tokenizer, quant_config=quant_config)

# Save quantized model
```

(continues on next page)

(continued from previous page)

```
model.save_quantized(quant_path)
tokenizer.save_pretrained(quant_path)
```

To run an AWQ model with vLLM, you can use TheBloke/Llama-2-7b-Chat-AWQ with the following command:

```
$ python examples/llm_engine_example.py --model TheBloke/Llama-2-7b-Chat-AWQ --
↪quantization awq
```

AWQ models are also supported directly through the LLM entrypoint:

```
from vllm import LLM, SamplingParams

# Sample prompts.
prompts = [
    "Hello, my name is",
    "The president of the United States is",
    "The capital of France is",
    "The future of AI is",
]
# Create a sampling params object.
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)

# Create an LLM.
llm = LLM(model="TheBloke/Llama-2-7b-Chat-AWQ", quantization="AWQ")
# Generate texts from the prompts. The output is a list of RequestOutput objects
# that contain the prompt, generated text, and other information.
outputs = llm.generate(prompts, sampling_params)
# Print the outputs.
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")
```

1.20 FP8 E5M2 KV Cache

The int8/int4 quantization scheme requires additional scale GPU memory storage, which reduces the expected GPU memory benefits. The FP8 data format retains 2~3 mantissa bits and can convert float/fp16/bflaot16 and fp8 to each other.

Here is an example of how to enable this feature:

```
from vllm import LLM, SamplingParams
# Sample prompts.
prompts = [
    "Hello, my name is",
    "The president of the United States is",
    "The capital of France is",
    "The future of AI is",
]
# Create a sampling params object.
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)
```

(continues on next page)

(continued from previous page)

```
# Create an LLM.
llm = LLM(model="facebook/opt-125m", kv_cache_dtype="fp8")
# Generate texts from the prompts. The output is a list of RequestOutput objects
# that contain the prompt, generated text, and other information.
outputs = llm.generate(prompts, sampling_params)
# Print the outputs.
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")
```

Note, current prefix caching doesn't work with FP8 KV cache enabled, forward_prefix kernel should handle different KV and cache type.

1.21 FP8 E4M3 KV Cache

Quantizing the KV cache to FP8 reduces its memory footprint. This increases the number of tokens that can be stored in the cache, improving throughput. OCP (Open Compute Project www.opencompute.org) specifies two common 8-bit floating point data formats: E5M2 (5 exponent bits and 2 mantissa bits) and E4M3FN (4 exponent bits and 3 mantissa bits), often shortened as E4M3. One benefit of the E4M3 format over E5M2 is that floating point numbers are represented in higher precision. However, the small dynamic range of FP8 E4M3 (±240.0 can be represented) typically necessitates the use of a higher-precision (typically FP32) scaling factor alongside each quantized tensor. For now, only per-tensor (scalar) scaling factors are supported. Development is ongoing to support scaling factors of a finer granularity (e.g. per-channel).

These scaling factors can be specified by passing an optional quantization param JSON to the LLM engine at load time. If this JSON is not specified, scaling factors default to 1.0. These scaling factors are typically obtained when running an unquantized model through a quantizer tool (e.g. AMD quantizer or NVIDIA AMMO).

To install AMMO (AlgorithMic Model Optimization):

```
$ pip install --no-cache-dir --extra-index-url https://pypi.nvidia.com nvidia-ammo
```

Studies have shown that FP8 E4M3 quantization typically only minimally degrades inference accuracy. The most recent silicon offerings e.g. AMD MI300, NVIDIA Hopper or later support native hardware conversion to and from fp32, fp16, bf16, etc. Thus, LLM inference is greatly accelerated with minimal accuracy loss.

Here is an example of how to enable this feature:

```
# two float8_e4m3fn kv cache scaling factor files are provided under tests/fp8_kv,␣
↪please refer to
# https://github.com/vllm-project/vllm/blob/main/examples/fp8/README.md to generate kv_
↪cache_scales.json of your own.

from vllm import LLM, SamplingParams
sampling_params = SamplingParams(temperature=1.3, top_p=0.8)
llm = LLM(model="meta-llama/Llama-2-7b-chat-hf",
          kv_cache_dtype="fp8",
          quantization_param_path="./tests/fp8_kv/llama2-7b-fp8-kv/kv_cache_scales.json")
prompt = "London is the capital of"
out = llm.generate(prompt, sampling_params)[0].outputs[0].text
print(out)
```

(continues on next page)

(continued from previous page)

```
# output w/ scaling factors:  England, the United Kingdom, and one of the world's leading␣
↪financial,
# output w/o scaling factors:  England, located in the southeastern part of the country.␣
↪It is known
```

Note, current prefix caching doesn't work with FP8 KV cache enabled, forward_prefix kernel should handle different KV and cache type.

1.22 Sampling Parameters

class vllm.SamplingParams(*n: int = 1, best_of: int | None = None, presence_penalty: float = 0.0, frequency_penalty: float = 0.0, repetition_penalty: float = 1.0, temperature: float = 1.0, top_p: float = 1.0, top_k: int = -1, min_p: float = 0.0, seed: int | None = None, use_beam_search: bool = False, length_penalty: float = 1.0, early_stopping: bool | str = False, stop: str | List[str] | None = None, stop_token_ids: List[int] | None = None, include_stop_str_in_output: bool = False, ignore_eos: bool = False, max_tokens: int | None = 16, min_tokens: int = 0, logprobs: int | None = None, prompt_logprobs: int | None = None, detokenize: bool = True, skip_special_tokens: bool = True, spaces_between_special_tokens: bool = True, logits_processors: List[Callable[[List[int], torch.Tensor], torch.Tensor] | Callable[[List[int], List[int], torch.Tensor], torch.Tensor]] | None = None, truncate_prompt_tokens: int[int] | None = None*)

Sampling parameters for text generation.

Overall, we follow the sampling parameters from the OpenAI text completion API (https://platform.openai.com/docs/api-reference/completions/create). In addition, we support beam search, which is not supported by OpenAI.

Parameters

- **n** – Number of output sequences to return for the given prompt.
- **best_of** – Number of output sequences that are generated from the prompt. From these *best_of* sequences, the top *n* sequences are returned. *best_of* must be greater than or equal to *n*. This is treated as the beam width when *use_beam_search* is True. By default, *best_of* is set to *n*.
- **presence_penalty** – Float that penalizes new tokens based on whether they appear in the generated text so far. Values > 0 encourage the model to use new tokens, while values < 0 encourage the model to repeat tokens.
- **frequency_penalty** – Float that penalizes new tokens based on their frequency in the generated text so far. Values > 0 encourage the model to use new tokens, while values < 0 encourage the model to repeat tokens.
- **repetition_penalty** – Float that penalizes new tokens based on whether they appear in the prompt and the generated text so far. Values > 1 encourage the model to use new tokens, while values < 1 encourage the model to repeat tokens.
- **temperature** – Float that controls the randomness of the sampling. Lower values make the model more deterministic, while higher values make the model more random. Zero means greedy sampling.
- **top_p** – Float that controls the cumulative probability of the top tokens to consider. Must be in (0, 1]. Set to 1 to consider all tokens.

- **`top_k`** – Integer that controls the number of top tokens to consider. Set to -1 to consider all tokens.
- **`min_p`** – Float that represents the minimum probability for a token to be considered, relative to the probability of the most likely token. Must be in [0, 1]. Set to 0 to disable this.
- **`seed`** – Random seed to use for the generation.
- **`use_beam_search`** – Whether to use beam search instead of sampling.
- **`length_penalty`** – Float that penalizes sequences based on their length. Used in beam search.
- **`early_stopping`** – Controls the stopping condition for beam search. It accepts the following values: *True*, where the generation stops as soon as there are *best_of* complete candidates; *False*, where an heuristic is applied and the generation stops when is it very unlikely to find better candidates; *"never"*, where the beam search procedure only stops when there cannot be better candidates (canonical beam search algorithm).
- **`stop`** – List of strings that stop the generation when they are generated. The returned output will not contain the stop strings.
- **`stop_token_ids`** – List of tokens that stop the generation when they are generated. The returned output will contain the stop tokens unless the stop tokens are special tokens.
- **`include_stop_str_in_output`** – Whether to include the stop strings in output text. Defaults to False.
- **`ignore_eos`** – Whether to ignore the EOS token and continue generating tokens after the EOS token is generated.
- **`max_tokens`** – Maximum number of tokens to generate per output sequence.
- **`min_tokens`** – Minimum number of tokens to generate per output sequence before EOS or stop_token_ids can be generated
- **`logprobs`** – Number of log probabilities to return per output token. Note that the implementation follows the OpenAI API: The return result includes the log probabilities on the *logprobs* most likely tokens, as well the chosen tokens. The API will always return the log probability of the sampled token, so there may be up to *logprobs+1* elements in the response.
- **`prompt_logprobs`** – Number of log probabilities to return per prompt token.
- **`detokenize`** – Whether to detokenize the output. Defaults to True.
- **`skip_special_tokens`** – Whether to skip special tokens in the output.
- **`spaces_between_special_tokens`** – Whether to add spaces between special tokens in the output. Defaults to True.
- **`logits_processors`** – List of functions that modify logits based on previously generated tokens, and optionally prompt tokens as a first argument.
- **`truncate_prompt_tokens`** – If set to an integer k, will use only the last k tokens from the prompt (i.e., left truncation). Defaults to None (i.e., no truncation).

`clone()` → *SamplingParams*

Deep copy excluding LogitsProcessor objects.

LogitsProcessor objects are excluded because they may contain an arbitrary, nontrivial amount of data. See https://github.com/vllm-project/vllm/issues/3087

`update_from_generation_config`(*generation_config: Dict[str, Any]*) → None

Update if there are non-default values from generation_config

1.23 Offline Inference

1.23.1 LLM Class

class vllm.**LLM**(*model: str*, *tokenizer: str | None = None*, *tokenizer_mode: str = 'auto'*, *skip_tokenizer_init: bool = False*, *trust_remote_code: bool = False*, *tensor_parallel_size: int = 1*, *dtype: str = 'auto'*, *quantization: str | None = None*, *revision: str | None = None*, *tokenizer_revision: str | None = None*, *seed: int = 0*, *gpu_memory_utilization: float = 0.9*, *swap_space: int = 4*, *enforce_eager: bool = False*, *max_context_len_to_capture: int | None = None*, *max_seq_len_to_capture: int = 8192*, *disable_custom_all_reduce: bool = False*, ***kwargs*)

An LLM for generating texts from given prompts and sampling parameters.

This class includes a tokenizer, a language model (possibly distributed across multiple GPUs), and GPU memory space allocated for intermediate states (aka KV cache). Given a batch of prompts and sampling parameters, this class generates texts from the model, using an intelligent batching mechanism and efficient memory management.

Parameters

- **model** – The name or path of a HuggingFace Transformers model.
- **tokenizer** – The name or path of a HuggingFace Transformers tokenizer.
- **tokenizer_mode** – The tokenizer mode. "auto" will use the fast tokenizer if available, and "slow" will always use the slow tokenizer.
- **skip_tokenizer_init** – If true, skip initialization of tokenizer and detokenizer. Expect valid prompt_token_ids and None for prompt from the input.
- **trust_remote_code** – Trust remote code (e.g., from HuggingFace) when downloading the model and tokenizer.
- **tensor_parallel_size** – The number of GPUs to use for distributed execution with tensor parallelism.
- **dtype** – The data type for the model weights and activations. Currently, we support *float32*, *float16*, and *bfloat16*. If *auto*, we use the *torch_dtype* attribute specified in the model config file. However, if the *torch_dtype* in the config is *float32*, we will use *float16* instead.
- **quantization** – The method used to quantize the model weights. Currently, we support "awq", "gptq", "squeezellm", and "fp8" (experimental). If None, we first check the *quantization_config* attribute in the model config file. If that is None, we assume the model weights are not quantized and use *dtype* to determine the data type of the weights.
- **revision** – The specific model version to use. It can be a branch name, a tag name, or a commit id.
- **tokenizer_revision** – The specific tokenizer version to use. It can be a branch name, a tag name, or a commit id.
- **seed** – The seed to initialize the random number generator for sampling.
- **gpu_memory_utilization** – The ratio (between 0 and 1) of GPU memory to reserve for the model weights, activations, and KV cache. Higher values will increase the KV cache size and thus improve the model's throughput. However, if the value is too high, it may cause out-of- memory (OOM) errors.
- **swap_space** – The size (GiB) of CPU memory per GPU to use as swap space. This can be used for temporarily storing the states of the requests when their *best_of* sampling parameters are larger than 1. If all requests will have *best_of=1*, you can safely set this to 0. Otherwise, too small values may cause out-of-memory (OOM) errors.

- **enforce_eager** – Whether to enforce eager execution. If True, we will disable CUDA graph and always execute the model in eager mode. If False, we will use CUDA graph and eager execution in hybrid.
- **max_context_len_to_capture** – Maximum context len covered by CUDA graphs. When a sequence has context length larger than this, we fall back to eager mode (DEPRECATED. Use *max_seq_len_to_capture* instead).
- **max_seq_len_to_capture** – Maximum sequence len covered by CUDA graphs. When a sequence has context length larger than this, we fall back to eager mode.
- **disable_custom_all_reduce** – See ParallelConfig
- ****kwargs** – Arguments for `EngineArgs`. (See *Engine Arguments*)

Note: This class is intended to be used for offline inference. For online serving, use the `AsyncLLMEngine` class instead.

DEPRECATE_LEGACY: ClassVar[bool] = False

A flag to toggle whether to deprecate the legacy generate/encode API.

encode(*prompts: str, pooling_params: PoolingParams | Sequence[PoolingParams] | None = None, prompt_token_ids: List[int] | None = None, use_tqdm: bool = True, lora_request: LoRARequest | None = None, multi_modal_data: MultiModalData | None = None*) → List[EmbeddingRequestOutput]

encode(*prompts: List[str], pooling_params: PoolingParams | Sequence[PoolingParams] | None = None, prompt_token_ids: List[List[int]] | None = None, use_tqdm: bool = True, lora_request: LoRARequest | None = None, multi_modal_data: MultiModalData | None = None*) → List[EmbeddingRequestOutput]

encode(*prompts: str | None = None, pooling_params: PoolingParams | Sequence[PoolingParams] | None = None, *, prompt_token_ids: List[int], use_tqdm: bool = True, lora_request: LoRARequest | None = None, multi_modal_data: MultiModalData | None = None*) → List[EmbeddingRequestOutput]

encode(*prompts: List[str] | None = None, pooling_params: PoolingParams | Sequence[PoolingParams] | None = None, *, prompt_token_ids: List[List[int]], use_tqdm: bool = True, lora_request: LoRARequest | None = None, multi_modal_data: MultiModalData | None = None*) → List[EmbeddingRequestOutput]

encode(*prompts: None, pooling_params: None, prompt_token_ids: List[int] | List[List[int]], use_tqdm: bool = True, lora_request: LoRARequest | None = None, multi_modal_data: MultiModalData | None = None*) → List[EmbeddingRequestOutput]

encode(*inputs: PromptStrictInputs | Sequence[PromptStrictInputs], /, *, pooling_params: PoolingParams | Sequence[PoolingParams] | None = None, use_tqdm: bool = True, lora_request: LoRARequest | None = None*) → List[EmbeddingRequestOutput]

Generates the completions for the input prompts.

This class automatically batches the given prompts, considering the memory constraint. For the best performance, put all of your prompts into a single list and pass it to this method.

Parameters

- **inputs** – The inputs to the LLM. You may pass a sequence of inputs for batch inference. See `PromptStrictInputs` for more details about the format of each input.
- **pooling_params** – The pooling parameters for pooling. If None, we use the default pooling parameters.
- **use_tqdm** – Whether to use tqdm to display the progress bar.

- **`lora_request`** – LoRA request to use for generation, if any.

Returns

A list of *EmbeddingRequestOutput* objects containing the generated embeddings in the same order as the input prompts.

Note: Using `prompts` and `prompt_token_ids` as keyword parameters is considered legacy and may be deprecated in the future. You should instead pass them via the `inputs` parameter.

`generate`(*prompts: str, sampling_params: SamplingParams | List[SamplingParams] | None = None, prompt_token_ids: List[int] | None = None, use_tqdm: bool = True, lora_request: LoRARequest | None = None, multi_modal_data: MultiModalData | None = None*) → List[RequestOutput]

`generate`(*prompts: List[str], sampling_params: SamplingParams | List[SamplingParams] | None = None, prompt_token_ids: List[List[int]] | None = None, use_tqdm: bool = True, lora_request: LoRARequest | None = None, multi_modal_data: MultiModalData | None = None*) → List[RequestOutput]

`generate`(*prompts: str | None = None, sampling_params: SamplingParams | List[SamplingParams] | None = None, *, prompt_token_ids: List[int], use_tqdm: bool = True, lora_request: LoRARequest | None = None, multi_modal_data: MultiModalData | None = None*) → List[RequestOutput]

`generate`(*prompts: List[str] | None = None, sampling_params: SamplingParams | List[SamplingParams] | None = None, *, prompt_token_ids: List[List[int]], use_tqdm: bool = True, lora_request: LoRARequest | None = None, multi_modal_data: MultiModalData | None = None*) → List[RequestOutput]

`generate`(*prompts: None, sampling_params: None, prompt_token_ids: List[int] | List[List[int]], use_tqdm: bool = True, lora_request: LoRARequest | None = None, multi_modal_data: MultiModalData | None = None*) → List[RequestOutput]

`generate`(*inputs: PromptStrictInputs | Sequence[PromptStrictInputs], /, *, sampling_params: SamplingParams | Sequence[SamplingParams] | None = None, use_tqdm: bool = True, lora_request: LoRARequest | None = None*) → List[RequestOutput]

Generates the completions for the input prompts.

This class automatically batches the given prompts, considering the memory constraint. For the best performance, put all of your prompts into a single list and pass it to this method.

Parameters

- **`inputs`** – A list of inputs to generate completions for.
- **`sampling_params`** – The sampling parameters for text generation. If None, we use the default sampling parameters. When it is a single value, it is applied to every prompt. When it is a list, the list must have the same length as the prompts and it is paired one by one with the prompt.
- **`use_tqdm`** – Whether to use tqdm to display the progress bar.
- **`lora_request`** – LoRA request to use for generation, if any.

Returns

A list of *RequestOutput* objects containing the generated completions in the same order as the input prompts.

Note: Using `prompts` and `prompt_token_ids` as keyword parameters is considered legacy and may be deprecated in the future. You should instead pass them via the `inputs` parameter.

1.23.2 LLM Inputs

`vllm.inputs.PromptStrictInputs`

The inputs to the LLM, which can take one of the following forms:

- A text prompt (`str` or *`TextPrompt`*)
- A tokenized prompt (*`TokensPrompt`*)

alias of `Union`[`str`, *`TextPrompt`*, *`TokensPrompt`*]

class `vllm.inputs.`**`TextPrompt`**`(`**args*, ***kwargs*`)`

Bases: `dict`

Schema for a text prompt.

`prompt: str`

The input text to be tokenized before passing to the model.

`multi_modal_data: typing_extensions.NotRequired[MultiModalData]`

Optional multi-modal data to pass to the model, if the model supports it.

class `vllm.inputs.`**`TokensPrompt`**`(`**args*, ***kwargs*`)`

Bases: `dict`

Schema for a tokenized prompt.

`prompt_token_ids: List[int]`

A list of token IDs to pass to the model.

`multi_modal_data: typing_extensions.NotRequired[MultiModalData]`

Optional multi-modal data to pass to the model, if the model supports it.

1.24 vLLM Engine

1.24.1 LLMEngine

class `vllm.`**`LLMEngine`**`(`*model_config: ModelConfig*, *cache_config: CacheConfig*, *parallel_config: ParallelConfig*, *scheduler_config: SchedulerConfig*, *device_config: DeviceConfig*, *load_config: LoadConfig*, *lora_config: LoRAConfig | None*, *vision_language_config: VisionLanguageConfig | None*, *speculative_config: SpeculativeConfig | None*, *decoding_config: DecodingConfig | None*, *executor_class: Type[ExecutorBase]*, *log_stats: bool*, *usage_context: UsageContext = UsageContext.ENGINE_CONTEXT*`)`

An LLM engine that receives requests and generates texts.

This is the main class for the vLLM engine. It receives requests from clients and generates texts from the LLM. It includes a tokenizer, a language model (possibly distributed across multiple GPUs), and GPU memory space allocated for intermediate states (aka KV cache). This class utilizes iteration-level scheduling and efficient memory management to maximize the serving throughput.

The *`LLM`* class wraps this class for offline batched inference and the *`AsyncLLMEngine`* class wraps this class for online serving.

The config arguments are derived from `EngineArgs`. (See *Engine Arguments*)

Parameters

- **`model_config`** – The configuration related to the LLM model.

- **cache_config** – The configuration related to the KV cache memory management.
- **parallel_config** – The configuration related to distributed execution.
- **scheduler_config** – The configuration related to the request scheduler.
- **device_config** – The configuration related to the device.
- **lora_config** (*Optional*) – The configuration related to serving multi-LoRA.
- **vision_language_config** (*Optional*) – The configuration related to vision language models.
- **speculative_config** (*Optional*) – The configuration related to speculative decoding.
- **executor_class** – The model executor class for managing distributed execution.
- **log_stats** – Whether to log statistics.
- **usage_context** – Specified entry point, used for usage info collection.

DO_VALIDATE_OUTPUT: ClassVar[bool] = False

A flag to toggle whether to validate the type of request output.

abort_request(*request_id: str | Iterable[str]*) → None

Aborts a request(s) with the given ID.

Parameters
request_id – The ID(s) of the request to abort.

Details:

- Refer to the `abort_seq_group()` from class `Scheduler`.

Example

```
>>> # initialize engine and add a request with request_id
>>> request_id = str(0)
>>> # abort the request
>>> engine.abort_request(request_id)
```

add_request(*request_id: str, inputs: str* | TextPrompt | TokensPrompt | *TextTokensPrompt, params:* SamplingParams | *PoolingParams, arrival_time: float* | *None = None, lora_request: LoRARequest* | *None = None*) → None

Add a request to the engine's request pool.

The request is added to the request pool and will be processed by the scheduler as *engine.step()* is called. The exact scheduling policy is determined by the scheduler.

Parameters

- **request_id** – The unique ID of the request.
- **inputs** – The inputs to the LLM. See `PromptInputs` for more details about the format of each input.
- **params** – Parameters for sampling or pooling. *SamplingParams* for text generation. `PoolingParams` for pooling.
- **arrival_time** – The arrival time of the request. If None, we use the current monotonic time.

Details:

- Set arrival_time to the current time if it is None.
- Set prompt_token_ids to the encoded prompt if it is None.
- Create *best_of* number of `Sequence` objects.
- Create a `SequenceGroup` object from the list of `Sequence`.
- Add the `SequenceGroup` object to the scheduler.

Example

```
>>> # initialize engine
>>> engine = LLMEngine.from_engine_args(engine_args)
>>> # set request arguments
>>> example_prompt = "Who is the president of the United States?"
>>> sampling_params = SamplingParams(temperature=0.0)
>>> request_id = 0
>>>
>>> # add the request to the engine
>>> engine.add_request(
>>>    str(request_id),
>>>    example_prompt,
>>>    SamplingParams(temperature=0.0))
>>> # continue the request processing
>>> ...
```

do_log_stats(*scheduler_outputs: SchedulerOutputs* | *None* = *None*, *model_output: List[SamplerOutput]* | *None* = *None*) → None

Forced log when no requests active.

classmethod from_engine_args(*engine_args: EngineArgs*, *usage_context: UsageContext* = *UsageContext.ENGINE_CONTEXT*) → *LLMEngine*

Creates an LLM engine from the engine arguments.

get_decoding_config() → DecodingConfig

Gets the decoding configuration.

get_model_config() → ModelConfig

Gets the model configuration.

get_num_unfinished_requests() → int

Gets the number of unfinished requests.

has_unfinished_requests() → bool

Returns True if there are unfinished requests.

step() → List[RequestOutput | EmbeddingRequestOutput]

Performs one decoding iteration and returns newly generated results.

Details:

- Step 1: Schedules the sequences to be executed in the next iteration and the token blocks to be swapped in/out/copy.
  - Depending on the scheduling policy, sequences may be *preempted/reordered*.

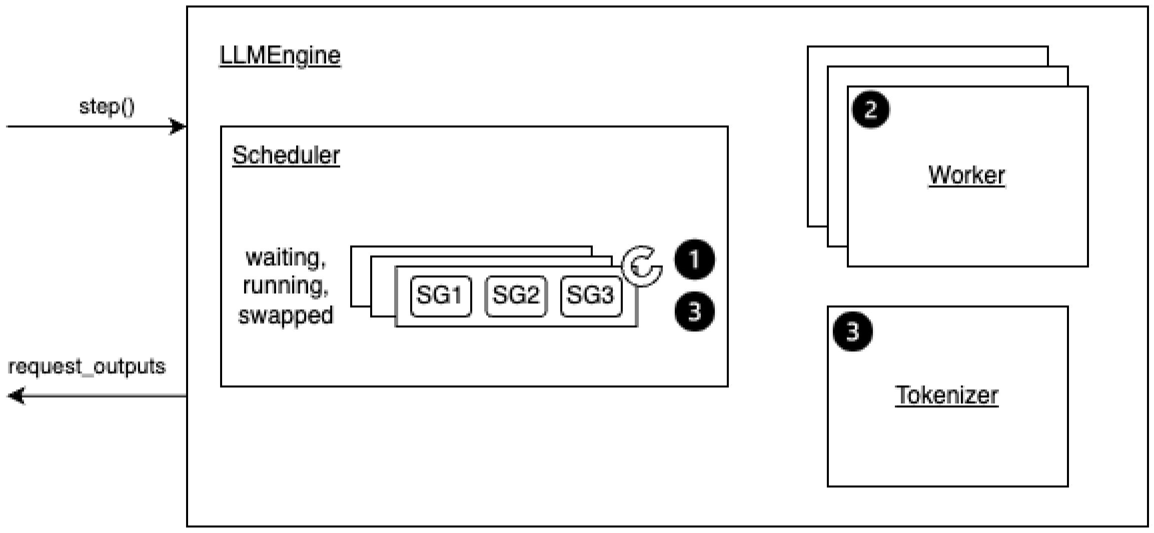


Fig. 1: Overview of the step function.

  - A Sequence Group (SG) refer to a group of sequences that are generated from the same prompt.
- Step 2: Calls the distributed executor to execute the model.
- Step 3: Processes the model output. This mainly includes:
  - Decodes the relevant outputs.
  - Updates the scheduled sequence groups with model outputs based on its *sampling parameters* (*use_beam_search* or not).
  - Frees the finished sequence groups.
- Finally, it creates and returns the newly generated results.

Example

```
>>> # Please see the example/ folder for more detailed examples.
>>>
>>> # initialize engine and request arguments
>>> engine = LLMEngine.from_engine_args(engine_args)
>>> example_inputs = [(0, "What is LLM?",
>>>    SamplingParams(temperature=0.0))]
>>>
>>> # Start the engine with an event loop
>>> while True:
>>>     if example_inputs:
>>>         req_id, prompt, sampling_params = example_inputs.pop(0)
>>>         engine.add_request(str(req_id),prompt,sampling_params)
>>>
>>>     # continue the request processing
>>>     request_outputs = engine.step()
>>>     for request_output in request_outputs:
>>>         if request_output.finished:
>>>             # return or show the request output
```

(continues on next page)

(continued from previous page)

```
>>>
>>>     if not (engine.has_unfinished_requests() or example_inputs):
>>>         break
```

1.24.2 AsyncLLMEngine

class vllm.**AsyncLLMEngine**(*worker_use_ray: bool*, *engine_use_ray: bool*, **args*, *log_requests: bool = True*, *max_log_len: int | None = None*, *start_engine_loop: bool = True*, ***kwargs*)

An asynchronous wrapper for *LLMEngine*.

This class is used to wrap the *LLMEngine* class to make it asynchronous. It uses asyncio to create a background loop that keeps processing incoming requests. The *LLMEngine* is kicked by the generate method when there are requests in the waiting queue. The generate method yields the outputs from the *LLMEngine* to the caller.

Parameters

- **worker_use_ray** – Whether to use Ray for model workers. Required for distributed execution. Should be the same as *parallel_config.worker_use_ray*.
- **engine_use_ray** – Whether to make LLMEngine a Ray actor. If so, the async frontend will be executed in a separate process as the model workers.
- **log_requests** – Whether to log the requests.
- **max_log_len** – Maximum number of prompt characters or prompt ID numbers being printed in log.
- **start_engine_loop** – If True, the background task to run the engine will be automatically started in the generate call.
- ***args** – Arguments for *LLMEngine*.
- ****kwargs** – Arguments for *LLMEngine*.

async abort(*request_id: str*) → None

Abort a request.

Abort a submitted request. If the request is finished or not found, this method will be a no-op.

Parameters
request_id – The unique id of the request.

async check_health() → None

Raises an error if engine is unhealthy.

async encode(*inputs: str* | TextPrompt | TokensPrompt | *TextTokensPrompt*, *pooling_params: PoolingParams*, *request_id: str*, *lora_request: LoRARequest | None = None*) → AsyncIterator[EmbeddingRequestOutput]

Generate outputs for a request from an embedding model.

Generate outputs for a request. This method is a coroutine. It adds the request into the waiting queue of the LLMEngine and streams the outputs from the LLMEngine to the caller.

Parameters

- **inputs** – The inputs to the LLM. See `PromptInputs` for more details about the format of each input.
- **pooling_params** – The pooling parameters of the request.

- **`request_id`** – The unique id of the request.
- **`lora_request`** – LoRA request to use for generation, if any.

Yields

The output *EmbeddingRequestOutput* objects from the LLMEngine for the request.

Details:

- If the engine is not running, start the background loop, which iteratively invokes `engine_step()` to process the waiting requests.
- Add the request to the engine's *RequestTracker*. On the next background loop, this request will be sent to the underlying engine. Also, a corresponding *AsyncStream* will be created.
- Wait for the request outputs from *AsyncStream* and yield them.

Example

```
>>> # Please refer to entrypoints/api_server.py for
>>> # the complete example.
>>>
>>> # initialize the engine and the example input
>>> engine = AsyncLLMEngine.from_engine_args(engine_args)
>>> example_input = {
>>>     "input": "What is LLM?",
>>>     "request_id": 0,
>>> }
>>>
>>> # start the generation
>>> results_generator = engine.encode(
>>>    example_input["input"],
>>>    PoolingParams(),
>>>    example_input["request_id"])
>>>
>>> # get the results
>>> final_output = None
>>> async for request_output in results_generator:
>>>     if await request.is_disconnected():
>>>         # Abort the request if the client disconnects.
>>>         await engine.abort(request_id)
>>>         # Return or raise an error
>>>         ...
>>>     final_output = request_output
>>>
>>> # Process and return the final output
>>> ...
```

async engine_step() → bool

Kick the engine to process the waiting requests.

Returns True if there are in-progress requests.

classmethod from_engine_args(*engine_args: AsyncEngineArgs*, *start_engine_loop: bool = True*, *usage_context: UsageContext = UsageContext.ENGINE_CONTEXT*) → *AsyncLLMEngine*

Creates an async LLM engine from the engine arguments.

async generate(*inputs: str | TextPrompt | TokensPrompt | TextTokensPrompt*, *sampling_params: SamplingParams*, *request_id: str*, *lora_request: LoRARequest | None = None*) → AsyncIterator[RequestOutput]

Generate outputs for a request.

Generate outputs for a request. This method is a coroutine. It adds the request into the waiting queue of the LLMEngine and streams the outputs from the LLMEngine to the caller.

Parameters

- **inputs** – The inputs to the LLM. See `PromptInputs` for more details about the format of each input.
- **sampling_params** – The sampling parameters of the request.
- **request_id** – The unique id of the request.
- **lora_request** – LoRA request to use for generation, if any.

Yields

The output *RequestOutput* objects from the LLMEngine for the request.

Details:

- If the engine is not running, start the background loop, which iteratively invokes `engine_step()` to process the waiting requests.
- Add the request to the engine's *RequestTracker*. On the next background loop, this request will be sent to the underlying engine. Also, a corresponding *AsyncStream* will be created.
- Wait for the request outputs from *AsyncStream* and yield them.

Example

```
>>> # Please refer to entrypoints/api_server.py for
>>> # the complete example.
>>>
>>> # initialize the engine and the example input
>>> engine = AsyncLLMEngine.from_engine_args(engine_args)
>>> example_input = {
>>>     "prompt": "What is LLM?",
>>>     "stream": False, # assume the non-streaming case
>>>     "temperature": 0.0,
>>>     "request_id": 0,
>>> }
>>>
>>> # start the generation
>>> results_generator = engine.generate(
>>>    example_input["prompt"],
>>>    SamplingParams(temperature=example_input["temperature"]),
>>>    example_input["request_id"])
>>>
>>> # get the results
>>> final_output = None
```

(continues on next page)

(continued from previous page)

```
>>> async for request_output in results_generator:
>>>     if await request.is_disconnected():
>>>         # Abort the request if the client disconnects.
>>>         await engine.abort(request_id)
>>>         # Return or raise an error
>>>         ...
>>>     final_output = request_output
>>>
>>> # Process and return the final output
>>> ...
```

`async get_decoding_config()` → DecodingConfig

Get the decoding configuration of the vLLM engine.

`async get_model_config()` → ModelConfig

Get the model configuration of the vLLM engine.

`start_background_loop()` → None

Start the background loop.

1.25 vLLM Paged Attention

- Currently, vLLM utilizes its own implementation of a multi-head query attention kernel (`csrc/attention/attention_kernels.cu`). This kernel is designed to be compatible with vLLM's paged KV caches, where the key and value cache are stored in separate blocks (note that this block concept differs from the GPU thread block. So in a later document, I will refer to vLLM paged attention block as "block", while refer to GPU thread block as "thread block").
- To achieve high performance, this kernel relies on a specially designed memory layout and access method, specifically when threads read data from global memory to shared memory. The purpose of this document is to provide a high-level explanation of the kernel implementation step by step, aiding those who wish to learn about the vLLM multi-head query attention kernel. After going through this document, users will likely have a better understanding and feel easier to follow the actual implementation.
- Please note that this document may not cover all details, such as how to calculate the correct index for the corresponding data or the dot multiplication implementation. However, after reading this document and becoming familiar with the high-level logic flow, it should be easier for you to read the actual code and understand the details.

1.25.1 Inputs

- The kernel function takes a list of arguments for the current thread to perform its assigned work. The three most important arguments are the input pointers `q`, `k_cache`, and `v_cache`, which point to query, key, and value data on global memory that need to be read and processed. The output pointer `out` points to global memory where the result should be written. These four pointers actually refer to multi-dimensional arrays, but each thread only accesses the portion of data assigned to it. I have omitted all other runtime parameters here for simplicity.

```
template<
typename scalar_t,
int HEAD_SIZE,
int BLOCK_SIZE,
```

(continues on next page)

(continued from previous page)

```
int NUM_THREADS,
int PARTITION_SIZE = 0>
__device__ void paged_attention_kernel(
... // Other side args.
const scalar_t* __restrict__ out,       // [num_seqs, num_heads, max_num_partitions,
↪ head_size]
const scalar_t* __restrict__ q,         // [num_seqs, num_heads, head_size]
const scalar_t* __restrict__ k_cache,   // [num_blocks, num_kv_heads, head_size/x,␣
↪block_size, x]
const scalar_t* __restrict__ v_cache,   // [num_blocks, num_kv_heads, head_size,␣
↪block_size]
... // Other side args.
)
```

- There are also a list of template arguments above the function signature that are determined during compilation time. `scalar_t` represents the data type of the query, key, and value data elements, such as FP16. `HEAD_SIZE` indicates the number of elements in each head. `BLOCK_SIZE` refers to the number of tokens in each block. `NUM_THREADS` denotes the number of threads in each thread block. `PARTITION_SIZE` represents the number of tensor parallel GPUs (For simplicity, we assume this is 0 and tensor parallel is disabled).
- With these arguments, we need to perform a sequence of preparations. This includes calculating the current head index, block index, and other necessary variables. However, for now, we can ignore these preparations and proceed directly to the actual calculations. It will be easier to understand them once we grasp the entire flow.

1.25.2 Concepts

- Just before we dive into the calculation flow, I want to describe a few concepts that are needed for later sections. However, you may skip this section and return later if you encounter any confusing terminologies.
- **Sequence**: A sequence represents a client request. For example, the data pointed to by `q` has a shape of `[num_seqs, num_heads, head_size]`. That represents there are total `num_seqs` of query sequence data are pointed by `q`. Since this kernel is a single query attention kernel, each sequence only has one query token. Hence, the `num_seqs` equals the total number of tokens that are processed in the batch.
- **Context**: The context consists of the generated tokens from the sequence. For instance, `["What", "is", "your"]` are the context tokens, and the input query token is `"name"`. The model might generate the token `"?"`.
- **Vec**: The vec is a list of elements that are fetched and calculated together. For query and key data, the vec size (`VEC_SIZE`) is determined so that each thread group can fetch and calculate 16 bytes of data at a time. For value data, the vec size (`V_VEC_SIZE`) is determined so that each thread can fetch and calculate 16 bytes of data at a time. For example, if the `scalar_t` is FP16 (2 bytes) and `THREAD_GROUP_SIZE` is 2, the `VEC_SIZE` will be 4, while the `V_VEC_SIZE` will be 8.
- **Thread group**: The thread group is a small group of threads(`THREAD_GROUP_SIZE`) that fetches and calculates one query token and one key token at a time. Each thread handles only a portion of the token data. The total number of elements processed by one thread group is referred as `x`. For example, if the thread group contains 2 threads and the head size is 8, then thread 0 handles the query and key elements at index 0, 2, 4, 6, while thread 1 handles the elements at index 1, 3, 5, 7.
- **Block**: The key and value cache data in vLLM are split into blocks. Each block stores data for a fixed number(`BLOCK_SIZE`) of tokens at one head. Each block may contain only a portion of the whole context tokens. For example, if the block size is 16 and the head size is 128, then for one head, one block can store 16 * 128 = 2048 elements.

- **Warp**: A warp is a group of 32 threads(WARP_SIZE) that execute simultaneously on a stream multiprocessor (SM). In this kernel, each warp processes the calculation between one query token and key tokens of one entire block at a time (it may process multiple blocks in multiple iterations). For example, if there are 4 warps and 6 blocks for one context, the assignment would be like warp 0 handles the 0th, 4th blocks, warp 1 handles the 1st, 5th blocks, warp 2 handles the 2nd block and warp 3 handles the 3rd block.
- **Thread block**: A thread block is a group of threads(NUM_THREADS) that can access the same shared memory. Each thread block contains multiple warps(NUM_WARPS), and in this kernel, each thread block processes the calculation between one query token and key tokens of a whole context.
- **Grid**: A grid is a collection of thread blocks and defines the shape of the collection. In this kernel, the shape is (num_heads, num_seqs, max_num_partitions). Therefore, each thread block only handles the calculation for one head, one sequence, and one partition.

1.25.3 Query

- This section will introduce how query data is stored in memory and fetched by each thread. As mentioned above, each thread group fetches one query token data, while each thread itself only handles a part of one query token data. Within each warp, every thread group will fetch the same query token data, but will multiply it with different key token data.

```
const scalar_t* q_ptr = q + seq_idx * q_stride + head_idx * HEAD_SIZE;
```

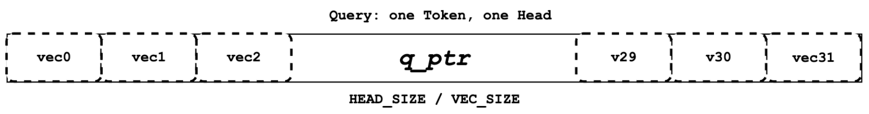


Fig. 2: Query data of one token at one head

- Each thread defines its own q_ptr which points to the assigned query token data on global memory. For example, if VEC_SIZE is 4 and HEAD_SIZE is 128, the q_ptr points to data that contains total of 128 elements divided into 128 / 4 = 32 vecs.

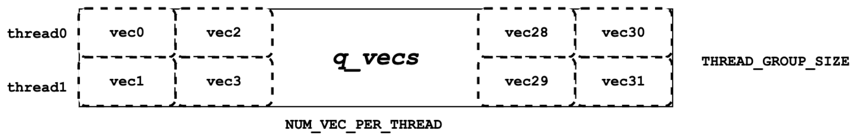


Fig. 3: q_vecs for one thread group

```
__shared__ Q_vec q_vecs[THREAD_GROUP_SIZE][NUM_VECS_PER_THREAD];
```

- Next, we need to read the global memory data pointed to by q_ptr into shared memory as q_vecs. It is important to note that each vecs is assigned to a different row. For example, if the THREAD_GROUP_SIZE is 2, thread 0 will handle the 0th row vecs, while thread 1 handles the 1st row vecs. By reading the query data in this way, neighboring threads like thread 0 and thread 1 can read neighbor memory, achieving the memory coalescing to improve performance.

1.25.4 Key

- Similar to the “Query” section, this section introduces memory layout and assignment for keys. While each thread group only handle one query token one kernel run, it may handle multiple key tokens across multiple iterations. Meanwhile, each warp will process multiple blocks of key tokens in multiple iterations, ensuring that all context tokens are processed by the entire thread group after the kernel run. In this context, “handle” refers to performing the dot multiplication between query data and key data.

```
const scalar_t* k_ptr = k_cache + physical_block_number * kv_block_stride
                    + kv_head_idx * kv_head_stride
                    + physical_block_offset * x;
```

- Unlike to `q_ptr`, `k_ptr` in each thread will point to different key token at different iterations. As shown above, that `k_ptr` points to key token data based on `k_cache` at assigned block, assigned head and assigned token.

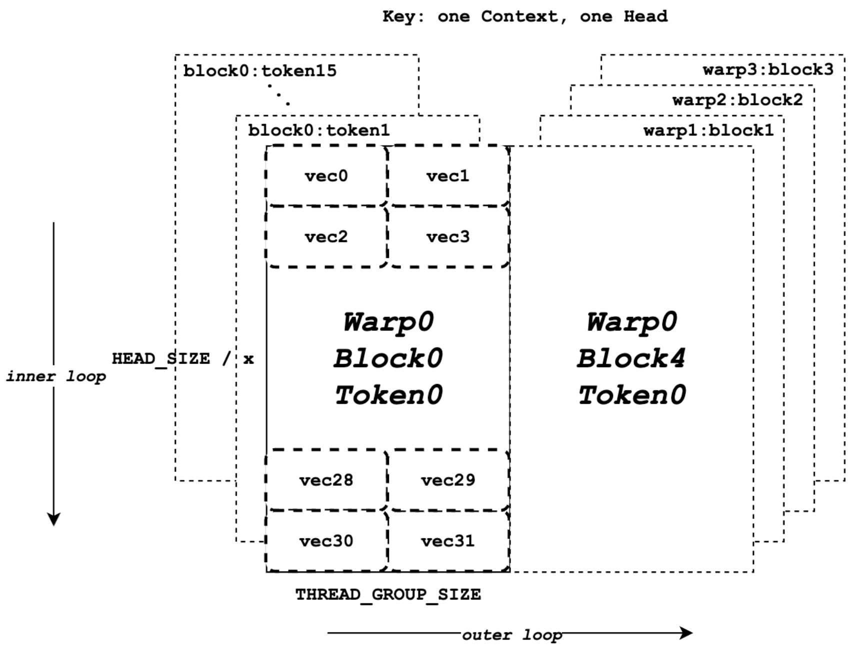


Fig. 4: Key data of all context tokens at one head

- The diagram above illustrates the memory layout for key data. It assumes that the `BLOCK_SIZE` is 16, `HEAD_SIZE` is 128, `x` is 8, `THREAD_GROUP_SIZE` is 2, and there are a total of 4 warps. Each rectangle represents all the elements for one key token at one head, which will be processed by one thread group. The left half shows the total 16 blocks of key token data for warp 0, while the right half represents the remaining key token data for other warps or iterations. Inside each rectangle, there are a total 32 vecs (128 elements for one token) that will be processed by 2 threads (one thread group) separately.

```
K_vec k_vecs[NUM_VECS_PER_THREAD]
```

- Next, we need to read the key token data from `k_ptr` and store them on register memory as `k_vecs`. We use register memory for `k_vecs` because it will only be accessed by one thread once, whereas `q_vecs` will be accessed by multiple threads multiple times. Each `k_vecs` will contain multiple vectors for later calculation. Each vec will be set at each inner iteration. The assignment of vecs allows neighboring threads in a warp to read neighboring memory together, which again promotes the memory coalescing. For instance, thread 0 will read vec 0, while thread 1 will read vec 1. In the next inner loop, thread 0 will read vec 2, while thread 1 will read vec

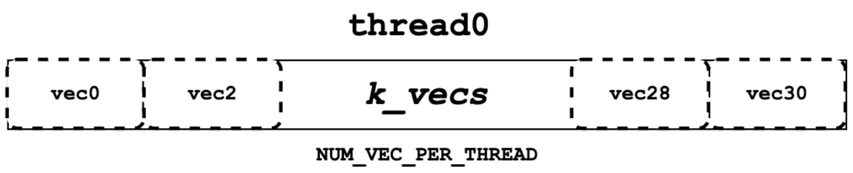


Fig. 5: k_vecs for one thread

3, and so on.

- You may still be a little confused about the overall flow. Don't worry, please keep reading the next "QK" section. It will illustrate the query and key calculation flow in a clearer and higher-level manner.

1.25.5 QK

- As shown the pseudo code below, before the entire for loop block, we fetch the query data for one token and store it in `q_vecs`. Then, in the outer for loop, we iterate through different `k_ptrs` that point to different tokens and prepare the `k_vecs` in the inner for loop. Finally, we perform the dot multiplication between the `q_vecs` and each `k_vecs`.

```
q_vecs = ...
for ... {
   k_ptr = ...
   for ... {
      k_vecs[i] = ...
   }
   ...
   float qk = scale * Qk_dot<scalar_t, THREAD_GROUP_SIZE>::dot(q_vecs[thread_group_
↪offset], k_vecs);
}
```

- As mentioned before, for each thread, it only fetches part of the query and key token data at a time. However, there will be a cross thread group reduction happen in the `Qk_dot<>::dot` . So `qk` returned here is not just between part of the query and key token dot multiplication, but actually a full result between entire query and key token data.

- For example, if the value of `HEAD_SIZE` is 128 and `THREAD_GROUP_SIZE` is 2, each thread's `k_vecs` will contain total 64 elements. However, the returned `qk` is actually the result of dot multiplication between 128 query elements and 128 key elements. If you want to learn more about the details of the dot multiplication and reduction, you may refer to the implementation of `Qk_dot<>::dot`. However, for the sake of simplicity, I will not cover it in this document.

1.25.6 Softmax

- Next, we need to calculate the normalized softmax for all `qk`s, as shown above, where each x represents a `qk`. To do this, we must obtain the reduced value of `qk_max`($m(x)$) and the `exp_sum`($\ell(x)$) of all `qk`s. The reduction should be performed across the entire thread block, encompassing results between the query token and all context key tokens.

$$m(x) := \max_i \quad x_i$$
$$f(x) := \left[\begin{array}{ccc} e^{x_1 - m(x)} & \ldots & e^{x_B - m(x)} \end{array}\right]$$
$$\ell(x) := \sum_i f(x)_i$$
$$\mathrm{softmax}(x) := \frac{f(x)}{\ell(x)}$$

`qk_max` and `logits`

- Just right after we get the `qk` result, we can set the temporary `logits` result with `qk` (In the end, the `logits` should store the normalized softmax result). Also we can compare and collect the `qk_max` for all `qk`s that are calculated by current thread group.

```
if (thread_group_offset == 0) {
   const bool mask = token_idx >= context_len;
   logits[token_idx - start_token_idx] = mask ? 0.f : qk;
   qk_max = mask ? qk_max : fmaxf(qk_max, qk);
}
```

- Please note that the `logits` here is on shared memory, so each thread group will set the fields for its own assigned context tokens. Overall, the size of logits should be number of context tokens.

```
for (int mask = WARP_SIZE / 2; mask >= THREAD_GROUP_SIZE; mask /= 2) {
    qk_max = fmaxf(qk_max, VLLM_SHFL_XOR_SYNC(qk_max, mask));
}

if (lane == 0) {
   red_smem[warp_idx] = qk_max;
}
```

- Then we need to get the reduced `qk_max` across each warp. The main idea is to make threads in warp to communicate with each other and get the final max `qk` .

```
for (int mask = NUM_WARPS / 2; mask >= 1; mask /= 2) {
    qk_max = fmaxf(qk_max, VLLM_SHFL_XOR_SYNC(qk_max, mask));
}
qk_max = VLLM_SHFL_SYNC(qk_max, 0);
```

- Finally, we can get the reduced `qk_max` from whole thread block by compare the `qk_max` from all warps in this thread block. Then we need to broadcast the final result to each thread.

exp_sum

- Similar to qk_max, we need to get the reduced sum value from the entire thread block too.

```
for (int i = thread_idx; i < num_tokens; i += NUM_THREADS) {
    float val = __expf(logits[i] - qk_max);
    logits[i] = val;
    exp_sum += val;
}
...
exp_sum = block_sum<NUM_WARPS>(&red_smem[NUM_WARPS], exp_sum);
```

- Firstly, sum all exp values from each thread group, and meanwhile, convert each entry of `logits` from `qk` to `exp(qk - qk_max)`. Please note, the `qk_max` here is already the max `qk` across the whole thread block. And then we can do reduction for `exp_sum` across whole thread block just like the `qk_max`.

```
const float inv_sum = __fdividef(1.f, exp_sum + 1e-6f);
for (int i = thread_idx; i < num_tokens; i += NUM_THREADS) {
   logits[i] *= inv_sum;
}
```

- Finally, with the reduced `qk_max` and `exp_sum`, we can obtain the final normalized softmax result as `logits`. This `logits` variable will be used for dot multiplication with the value data in later steps. Now, it should store the normalized softmax result of `qk` for all assigned context tokens.

1.25.7 Value

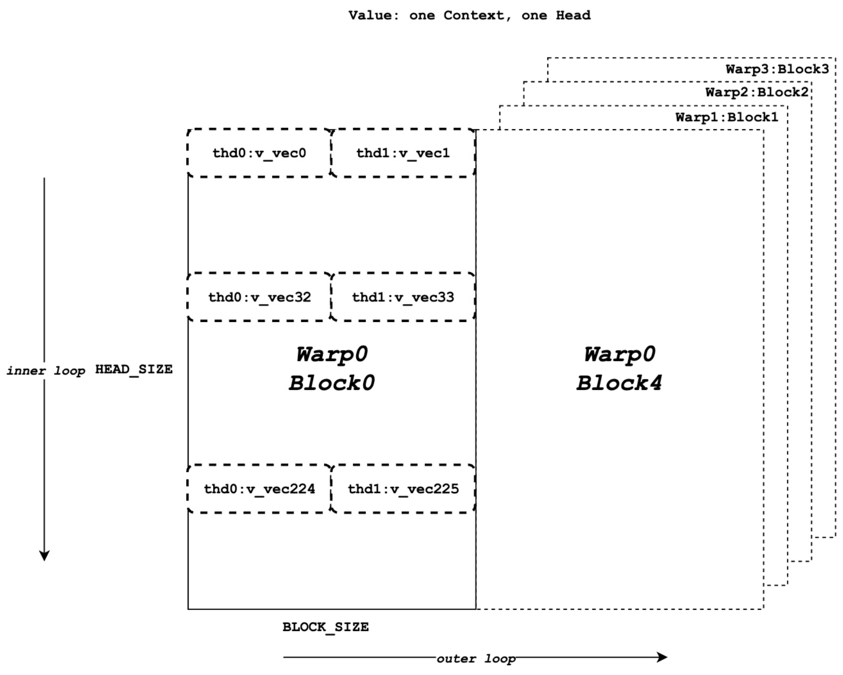


Fig. 6: Value data of all context tokens at one head

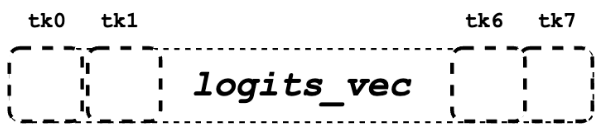


Fig. 7: `logits_vec` for one thread

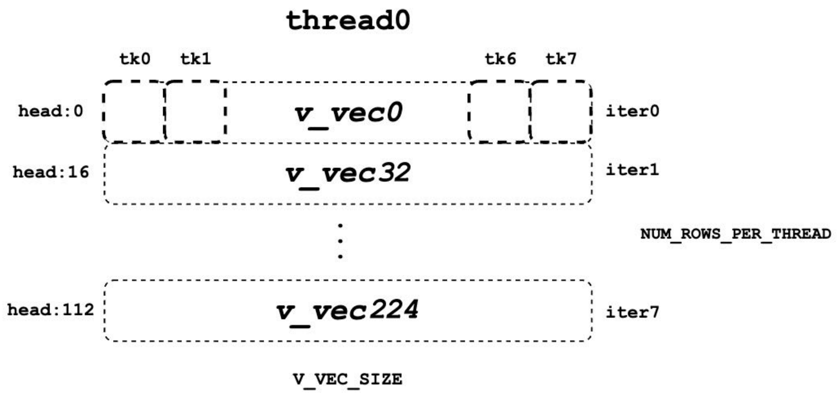


Fig. 8: List of `v_vec` for one thread

- Now we need to retrieve the value data and perform dot multiplication with `logits`. Unlike query and key, there is no thread group concept for value data. As shown in diagram, different from key token memory layout, elements from the same column correspond to the same value token. For one block of value data, there are `HEAD_SIZE` of rows and `BLOCK_SIZE` of columns that are split into multiple `v_vecs`.
- Each thread always fetches `V_VEC_SIZE` elements from the same `V_VEC_SIZE` of tokens at a time. As a result, a single thread retrieves multiple `v_vecs` from different rows and the same columns through multiple inner iterations. For each `v_vec`, it needs to be dot multiplied with the corresponding `logits_vec`, which is also `V_VEC_SIZE` elements from `logits`. Overall, with multiple inner iterations, each warp will process one block of value tokens. And with multiple outer iterations, the whole context value tokens are processd

```
float accs[NUM_ROWS_PER_THREAD];
for ... { // Iteration over different blocks.
    logits_vec = ...
    for ... { // Iteration over different rows.
        v_vec = ...
        ...
        accs[i] += dot(logits_vec, v_vec);
    }
}
```

- As shown in the above pseudo code, in the outer loop, similar to `k_ptr`, `logits_vec` iterates over different blocks and reads `V_VEC_SIZE` elements from `logits`. In the inner loop, each thread reads `V_VEC_SIZE` elements from the same tokens as a `v_vec` and performs dot multiplication. It is important to note that in each inner iteration, the thread fetches different head position elements for the same tokens. The dot result is then accumulated in `accs`. Therefore, each entry of `accs` is mapped to a head position assigned to the current thread.
- For example, if `BLOCK_SIZE` is 16 and `V_VEC_SIZE` is 8, each thread fetches 8 value elements for 8 tokens at a time. Each element is from different tokens at the same head position. If `HEAD_SIZE` is 128 and `WARP_SIZE` is 32, for each inner loop, a warp needs to fetch `WARP_SIZE * V_VEC_SIZE = 256` elements. This means there

are a total of 128 * 16 / 256 = 8 inner iterations for a warp to handle a whole block of value tokens. And each accs in each thread contains 8 elements that accumulated at 8 different head positions. For the thread 0, the accs variable will have 8 elements, which are 0th, 32th … 224th elements of a value head that are accumulated from all assigned 8 tokens.

1.25.8 LV

- Now, we need to perform reduction for accs within each warp. This process allows each thread to accumulate the accs for the assigned head positions of all tokens in one block.

```
for (int i = 0; i < NUM_ROWS_PER_THREAD; i++) {
   float acc = accs[i];
   for (int mask = NUM_V_VECS_PER_ROW / 2; mask >= 1; mask /= 2) {
      acc += VLLM_SHFL_XOR_SYNC(acc, mask);
   }
   accs[i] = acc;
}
```

- Next, we perform reduction for accs across all warps, allowing each thread to have the accumulation of accs for the assigned head positions of all context tokens. Please note that each accs in every thread only stores the accumulation for a portion of elements of the entire head for all context tokens. However, overall, all results for output have been calculated but are just stored in different thread register memory.

```
float* out_smem = reinterpret_cast<float*>(shared_mem);
for (int i = NUM_WARPS; i > 1; i /= 2) {
    // Upper warps write to shared memory.
    ...
        float* dst = &out_smem[(warp_idx - mid) * HEAD_SIZE];
        for (int i = 0; i < NUM_ROWS_PER_THREAD; i++) {
                ...
        dst[row_idx] = accs[i];
    }

    // Lower warps update the output.
        const float* src = &out_smem[warp_idx * HEAD_SIZE];
    for (int i = 0; i < NUM_ROWS_PER_THREAD; i++) {
                ...
        accs[i] += src[row_idx];
    }

        // Write out the accs.
}
```

1.25.9 Output

- Now we can write all of calculated result from local register memory to final output global memory.

```
scalar_t* out_ptr = out + seq_idx * num_heads * max_num_partitions * HEAD_SIZE
                + head_idx * max_num_partitions * HEAD_SIZE
                + partition_idx * HEAD_SIZE;
```

- First, we need to define the `out_ptr` variable, which points to the start address of the assigned sequence and assigned head.

```
for (int i = 0; i < NUM_ROWS_PER_THREAD; i++) {
const int row_idx = lane / NUM_V_VECS_PER_ROW + i * NUM_ROWS_PER_ITER;
if (row_idx < HEAD_SIZE && lane % NUM_V_VECS_PER_ROW == 0) {
    from_float(*(out_ptr + row_idx), accs[i]);
}
}
```

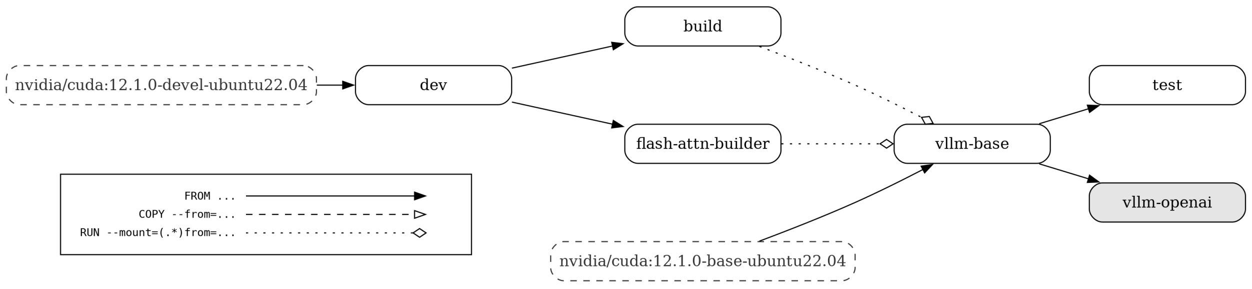
- Finally, we need to iterate over different assigned head positions and write out the corresponding accumulated result based on the `out_ptr`.

1.26 Dockerfile

See here for the main Dockerfile to construct the image for running an OpenAI compatible server with vLLM.

- Below is a visual representation of the multi-stage Dockerfile. The build graph contains the following nodes:
  - All build stages
  - The default build target (highlighted in grey)
  - External images (with dashed borders)

  The edges of the build graph represent:
  - FROM … dependencies (with a solid line and a full arrow head)
  - COPY –from=… dependencies (with a dashed line and an empty arrow head)
  - RUN –mount=(.*)from=… dependencies (with a dotted line and an empty diamond arrow head)



  Made using: https://github.com/patrickhoefler/dockerfilegraph

  Commands to regenerate the build graph (make sure to run it **from the `root` directory of the vLLM repository** where the dockerfile is present):

```
dockerfilegraph -o png --legend --dpi 200 --max-label-length 50 --filename␣
↪Dockerfile
```

or in case you want to run it directly with the docker image:

```
docker run \
   --rm \
   --user "$(id -u):$(id -g)" \
   --workdir /workspace \
   --volume "$(pwd)":/workspace \
   ghcr.io/patrickhoefler/dockerfilegraph:alpine \
   --output png \
   --dpi 200 \
   --max-label-length 50 \
   --filename Dockerfile \
   --legend
```

(To run it for a different file, you can pass in a different argument to the flag *–filename*.)

1.27 vLLM Meetups

We host regular meetups in San Francisco Bay Area every 2 months. We will share the project updates from the vLLM team and have guest speakers from the industry to share their experience and insights. Please find the materials of our previous meetups below:

- The third vLLM meetup, with Roblox, April 2nd 2024. [Slides]
- The second vLLM meetup, with IBM Research, January 31st 2024. [Slides] [Video (vLLM Update)] [Video (IBM Research & torch.compile)]
- The first vLLM meetup, with a16z, October 5th 2023. [Slides]

We are always looking for speakers and sponsors at San Francisco Bay Area and potentially other locations. If you are interested in speaking or sponsoring, please contact us at vllm-questions@lists.berkeley.edu.

1.28 Sponsors

vLLM is a community project. Our compute resources for development and testing are supported by the following organizations. Thank you for your support!

- a16z
- AMD
- Anyscale
- AWS
- Crusoe Cloud
- Databricks
- DeepInfra
- Dropbox
- Lambda Lab

- NVIDIA
- Replicate
- Roblox
- RunPod
- Trainy
- UC Berkeley
- UC San Diego

We also have an official fundraising venue through OpenCollective. We plan to use the fund to support the development, maintenance, and adoption of vLLM.

CHAPTER TWO

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

INDEX