vLLM

the vLLM Team

Jul 23, 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 and pipeline 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=3.10 -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=310
$ 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.

Note: vLLM also publishes a subset of wheels (Python 3.10, 3.11 with CUDA 12) for every commit since v0.5.3. You can download them with the following command:

```
$ export VLLM_VERSION=0.5.2 # vLLM's main branch version is currently set to latest␣
↪released tag
$ pip install https://vllm-wheels.s3.us-west-2.amazonaws.com/nightly/vllm-${VLLM_VERSION}
↪-cp38-abi3-manylinux1_x86_64.whl
$ # You can also access a specific commit
$ # export VLLM_COMMIT=...
$ # pip install https://vllm-wheels.s3.us-west-2.amazonaws.com/${VLLM_COMMIT}/vllm-$
↪{VLLM_VERSION}-cp38-abi3-manylinux1_x86_64.whl
```

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 6.1.

1.2.1 Requirements

- OS: Linux
- Python: 3.8 – 3.11
- GPU: MI200s (gfx90a), MI300 (gfx942), Radeon RX 7900 series (gfx1100)
- ROCm 6.1

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 6.1 by default, but also supports ROCm 5.7 and 6.0 in older vLLM branches. 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.
- *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 6.1 for MI200 and MI300 series, you can use the default:

```
$ DOCKER_BUILDKIT=1 docker build -f Dockerfile.rocm -t vllm-rocm .
```

To build vllm on ROCm 6.1 for Radeon RX7900 series (gfx1100), you should specify `BUILD_FA` as below:

```
$ DOCKER_BUILDKIT=1 docker build --build-arg BUILD_FA="0" -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, *rocm/pytorch:rocm6.1.2_ubuntu20.04_py3.9_pytorch_staging*, *rocm/pytorch-nightly*.

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

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 (v2.5.9.post1) following the instructions from ROCm/flash-attention Alternatively, wheels intended for vLLM use can be accessed under the releases.

Note:

- 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 develop # This may take 5-10 minutes. Currently, `pip install .`` does␣
↪not work for ROCm installation
```

Tip:

- Triton flash attention is used by default. For benchmarking purposes, it is recommended to run a warm up step before collecting perf numbers.
- Triton flash attention does not currently support sliding window attention. If using half precision, please use CK flash-attention for sliding window support.
- To use CK flash-attention or PyTorch naive attention, please use this flag `export VLLM_USE_TRITON_FLASH_ATTN=0` to turn off triton flash attention.
- The ROCm version of PyTorch, ideally, should match the ROCm driver version.

1.3 Installation with OpenVINO

vLLM powered by OpenVINO supports all LLM models from *vLLM supported models list* and can perform optimal model serving on all x86-64 CPUs with, at least, AVX2 support. OpenVINO vLLM backend supports the following advanced vLLM features:

- Prefix caching (`--enable-prefix-caching`)
- Chunked prefill (`--enable-chunked-prefill`)

Table of contents:

1.3.1 Requirements

- OS: Linux
- Instruction set architecture (ISA) requirement: at least AVX2.

1.3.2 Quick start using Dockerfile

```
$ docker build -f Dockerfile.openvino -t vllm-openvino-env .
$ docker run -it --rm vllm-openvino-env
```

1.3.3 Install from source

- First, install Python. For example, on Ubuntu 22.04, you can run:

```
$ sudo apt-get update  -y
$ sudo apt-get install python3
```

- Second, install prerequisites vLLM OpenVINO backend installation:

```
$ pip install --upgrade pip
$ pip install -r requirements-build.txt --extra-index-url https://download.pytorch.
↪org/whl/cpu
```

- Finally, install vLLM with OpenVINO backend:

```
$ PIP_PRE=1 PIP_EXTRA_INDEX_URL="https://download.pytorch.org/whl/cpu https://
↪storage.openvinotoolkit.org/simple/wheels/nightly/" VLLM_TARGET_DEVICE=openvino␣
↪python -m pip install -v .
```

1.3.4 Performance tips

vLLM OpenVINO backend uses the following environment variables to control behavior:

- `VLLM_OPENVINO_KVCACHE_SPACE` to specify the KV Cache size (e.g, `VLLM_OPENVINO_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_OPENVINO_CPU_KV_CACHE_PRECISION=u8` to control KV cache precision. By default, FP16 / BF16 is used depending on platform.
- `VLLM_OPENVINO_ENABLE_QUANTIZED_WEIGHTS=ON` to enable U8 weights compression during model loading stage. By default, compression is turned off.

To enable better TPOT / TTFT latency, you can use vLLM's chunked prefill feature (`--enable-chunked-prefill`). Based on the experiments, the recommended batch size is `256` (`--max-num-batched-tokens`)

OpenVINO best known configuration is:

```
$ VLLM_OPENVINO_KVCACHE_SPACE=100 VLLM_OPENVINO_CPU_KV_CACHE_PRECISION=u8 VLLM_OPENVINO_
↪ENABLE_QUANTIZED_WEIGHTS=ON \
    python3 vllm/benchmarks/benchmark_throughput.py --model meta-llama/Llama-2-7b-chat-
↪hf --dataset vllm/benchmarks/ShareGPT_V3_unfiltered_cleaned_split.json --enable-
↪chunked-prefill --max-num-batched-tokens 256
```

1.3.5 Limitations

- LoRA serving is not supported.
- Only LLM models are currently supported. LLaVa and encoder-decoder models are not currently enabled in vLLM OpenVINO integration.
- Tensor and pipeline parallelism are not currently enabled in vLLM integration.
- Speculative sampling is not tested within vLLM integration.

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 (optional, recommended)
- Instruction set architecture (ISA) requirement: AVX512 (optional, recommended)

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 recommended 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 Intel Extension for PyTorch

- Intel Extension for PyTorch (IPEX) extends PyTorch with up-to-date features optimizations for an extra performance boost on Intel hardware.
- IPEX after the `2.3.0` can be enabled in the CPU backend by default if it is installed.

1.4.5 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.
- We highly recommend to use TCMalloc for high performance memory allocation and better cache locality. For example, on Ubuntu 22.4, you can run:

```
$ sudo apt-get install libtcmalloc-minimal4 # install TCMalloc library
$ find / -name *libtcmalloc* # find the dynamic link library path
$ export LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libtcmalloc_minimal.so.4:$LD_PRELOAD #␣
```

(continues on next page)

(continued from previous page)

```
↪prepend the library to LD_PRELOAD
$ python examples/offline_inference.py # run vLLM
```

- 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 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.

1.5.1 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:

1.5.2 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.

1.6 Installation with TPU

vLLM supports Google Cloud TPUs using PyTorch XLA.

1.6.1 Requirements

- Google Cloud TPU VM (single host)
- TPU versions: v5e, v5p, v4
- Python: 3.10

Installation options:

1. *Build a docker image with Dockerfile*.
2. *Build from source*.

1.6.2 Build a docker image with `Dockerfile.tpu`

Dockerfile.tpu is provided to build a docker image with TPU support.

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

You can run the docker image with the following command:

```
$ # Make sure to add `--privileged --net host --shm-size=16G`.
$ docker run --privileged --net host --shm-size=16G -it vllm-tpu
```

1.6.3 Build from source

You can also build and install the TPU backend from source.

First, install the dependencies:

```
$ # (Recommended) Create a new conda environment.
$ conda create -n myenv python=3.10 -y
$ conda activate myenv

$ # Clean up the existing torch and torch-xla packages.
$ pip uninstall torch torch-xla -y

$ # Install PyTorch and PyTorch XLA.
$ export DATE="+20240713"
$ pip install https://storage.googleapis.com/pytorch-xla-releases/wheels/tpuvm/torch-
↪nightly${DATE}-cp310-cp310-linux_x86_64.whl
$ pip install https://storage.googleapis.com/pytorch-xla-releases/wheels/tpuvm/torch_xla-
↪nightly${DATE}-cp310-cp310-linux_x86_64.whl

$ # Install JAX and Pallas.
$ pip install torch_xla[tpu] -f https://storage.googleapis.com/libtpu-releases/index.html
$ pip install torch_xla[pallas] -f https://storage.googleapis.com/jax-releases/jax_
↪nightly_releases.html -f https://storage.googleapis.com/jax-releases/jaxlib_nightly_
```

(continues on next page)

(continued from previous page)

```
↪releases.html

$ # Install other build dependencies.
$ pip install packaging aiohttp
```

Next, build vLLM from source. This will only take a few seconds:

```
$ VLLM_TARGET_DEVICE="tpu" python setup.py develop
```

Tip: If you encounter the following error:

```
from torch._C import *  # noqa: F403
ImportError: libopenblas.so.0: cannot open shared object file: No such file or directory
```

Please install OpenBLAS with the following command:

```
$ sudo apt-get install libopenblas-base libopenmpi-dev libomp-dev
```

1.7 Installation with XPU

vLLM initially supports basic model inferencing and serving on Intel GPU platform.

Table of contents:

1.7.1 Requirements

- OS: Linux
- Supported Hardware: Intel Data Center GPU (Intel ARC GPU WIP)
- OneAPI requirements: oneAPI 2024.1

1.7.2 Quick start using Dockerfile

```
$ docker build -f Dockerfile.xpu -t vllm-xpu-env --shm-size=4g .
$ docker run -it \
             --rm \
             --network=host \
             --device /dev/dri \
             -v /dev/dri/by-path:/dev/dri/by-path \
             vllm-xpu-env
```

1.7.3 Build from source

- First, install required driver and intel OneAPI 2024.1 or later.
- Second, install Python packages for vLLM XPU backend building:

```
$ source /opt/intel/oneapi/setvars.sh
$ pip install --upgrade pip
$ pip install -v -r requirements-xpu.txt
```

- Finally, build and install vLLM XPU backend:

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

Note:

- FP16 is the default data type in the current XPU backend. The BF16 data type will be supported in the future.

1.8 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
```

1.8.1 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.

1.8.2 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:

```
$ vllm serve 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:

```
$ vllm serve 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:

```
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:

```python
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.

1.9 Debugging Tips

1.9.1 Debugging hang/crash issues

When an vLLM instance hangs or crashes, it is very difficult to debug the issue. But wait a minute, it is also possible that vLLM is doing something that indeed takes a long time:

- **Downloading a model**: Do you have the model already downloaded in your disk? If not, vLLM will download the model from the internet, which can take a long time. Be sure to check the internet connection. It would be better to download the model first using huggingface-cli and then use the local path to the model. This way, you can isolate the issue.
- **Loading the model from disk**: If the model is large, it can take a long time to load the model from disk. Please take care of the location you store the model. Some clusters have shared filesystems across nodes, e.g. distributed filesystem or network filesystem, which can be slow. It would be better to store the model in a local disk. In addition, please also watch the CPU memory usage. When the model is too large, it might take much CPU memory, which can slow down the operating system because it needs to frequently swap memory between the disk and the memory.
- **Tensor parallel inference**: If the model is too large to fit in a single GPU, you might want to use tensor parallelism to split the model across multiple GPUs. In that case, every process will read the whole model and split it into chunks, which makes the disk reading time even longer (proportional to the size of tensor parallelism). You can convert the model checkpoint to a sharded checkpoint using the provided script . The conversion process might take some time, but later you can load the sharded checkpoint much faster. The model loading time should remain constant regardless of the size of tensor parallelism.

If you have already taken care of the above issues, but the vLLM instance still hangs, with CPU and GPU utilization at near zero, it is likely that the vLLM instance is stuck somewhere. Here are some tips to help debug the issue:

- Set the environment variable `export VLLM_LOGGING_LEVEL=DEBUG` to turn on more logging.
- Set the environment variable `export CUDA_LAUNCH_BLOCKING=1` to know exactly which CUDA kernel is causing the trouble.

- Set the environment variable `export NCCL_DEBUG=TRACE` to turn on more logging for NCCL.
- Set the environment variable `export VLLM_TRACE_FUNCTION=1`. All the function calls in vLLM will be recorded. Inspect these log files, and tell which function crashes or hangs.

With more logging, hopefully you can find the root cause of the issue.

If it crashes, and the error trace shows somewhere around `self.graph.replay()` in `vllm/worker/model_runner.py`, it is a cuda error inside cudagraph. To know the particular cuda operation that causes the error, you can add `--enforce-eager` to the command line, or `enforce_eager=True` to the `LLM` class, to disable the cudagraph optimization. This way, you can locate the exact cuda operation that causes the error.

Here are some common issues that can cause hangs:

- **Incorrect network setup**: The vLLM instance cannot get the correct IP address if you have complicated network config. You can find the log such as `DEBUG 06-10 21:32:17 parallel_state.py:88] world_size=8 rank=0 local_rank=0 distributed_init_method=tcp://xxx.xxx.xxx.xxx:54641 backend=nccl`. The IP address should be the correct one. If not, override the IP address by setting the environment variable `export VLLM_HOST_IP=your_ip_address`. You might also need to set `export NCCL_SOCKET_IFNAME=your_network_interface` and `export GLOO_SOCKET_IFNAME=your_network_interface` to specify the network interface for the IP address.
- **Incorrect hardware/driver**: GPU/CPU communication cannot be established. You can run the following sanity check script to see if the GPU/CPU communication is working correctly.

```
import torch
import torch.distributed as dist
dist.init_process_group(backend="nccl")
local_rank = dist.get_rank() % torch.cuda.device_count()
data = torch.FloatTensor([1,] * 128).to(f"cuda:{local_rank}")
dist.all_reduce(data, op=dist.ReduceOp.SUM)
torch.cuda.synchronize()
value = data.mean().item()
world_size = dist.get_world_size()
assert value == world_size, f"Expected {world_size}, got {value}"

gloo_group = dist.new_group(ranks=list(range(world_size)), backend="gloo")
cpu_data = torch.FloatTensor([1,] * 128)
dist.all_reduce(cpu_data, op=dist.ReduceOp.SUM, group=gloo_group)
value = cpu_data.mean().item()
assert value == world_size, f"Expected {world_size}, got {value}"

print("sanity check is successful!")
```

Tip: Save the script as `test.py`.

If you are testing in a single-node, run it with `NCCL_DEBUG=TRACE torchrun --nproc-per-node=8 test.py`, adjust `--nproc-per-node` to the number of GPUs you want to use.

If you are testing with multi-nodes, run it with `NCCL_DEBUG=TRACE torchrun --nnodes 2 --nproc-per-node=2 --rdzv_backend=c10d --rdzv_endpoint=$MASTER_ADDR test.py`. Adjust `--nproc-per-node` and `--nnodes` according to your setup. Make sure `MASTER_ADDR`:

- is the correct IP address of the master node
- is reachable from all nodes
- is set before running the script.

If the script runs successfully, you should see the message `sanity check is successful!`.

If the problem persists, feel free to open an issue on GitHub, with a detailed description of the issue, your environment, and the logs.

Warning: After you find the root cause and solve the issue, remember to turn off all the debugging environment variables defined above, or simply start a new shell to avoid being affected by the debugging settings. If you don't do this, the system might be slow because many debugging functionalities are turned on.

1.10 Examples

1.10.1 API Client

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

```
"""Example Python client for `vllm.entrypoints.api_server`
NOTE: The API server is used only for demonstration and simple performance
benchmarks. It is not intended for production use.
For production use, we recommend `vllm serve` and the OpenAI client API.
"""

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)
```

(continues on next page)

(continued from previous page)

```
    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"]
    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)
```

1.10.2 Aqlm Example

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

```
from vllm import LLM, SamplingParams
from vllm.utils import FlexibleArgumentParser


def main():

    parser = FlexibleArgumentParser(description='AQLM examples')

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

1.10.3 Cpu Offload

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

```
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="meta-llama/Llama-2-13b-chat-hf", cpu_offload_gb=10)
# 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}")
```

1.10.4 Fuyu Example

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

```
import requests
from PIL import Image

from vllm import LLM, SamplingParams


def run_fuyu():
    llm = LLM(model="adept/fuyu-8b", max_model_len=4096)

    # single-image prompt
    prompt = "What is the highest life expectancy at of male?\n"
    url = "https://huggingface.co/adept/fuyu-8b/resolve/main/chart.png"
    image = Image.open(requests.get(url, stream=True).raw)
    sampling_params = SamplingParams(temperature=0, max_tokens=64)

    outputs = llm.generate(
        {
            "prompt": prompt,
            "multi_modal_data": {
                "image": image
            },
```

(continues on next page)

(continued from previous page)

```
        },
        sampling_params=sampling_params)

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


if __name__ == "__main__":
    run_fuyu()
```

1.10.5 Gradio OpenAI Chatbot Webserver

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

```
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,
                    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
```

(continues on next page)

(continued from previous page)

```
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(',')
                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)
```

1.10.6 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")
        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()
```

(continues on next page)

(continued from previous page)

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

1.10.7 Llava Example

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

```
from vllm import LLM
from vllm.assets.image import ImageAsset


def run_llava():
    llm = LLM(model="llava-hf/llava-1.5-7b-hf")

    prompt = "USER: <image>\nWhat is the content of this image?\nASSISTANT:"

    image = ImageAsset("stop_sign").pil_image

    outputs = llm.generate({
        "prompt": prompt,
        "multi_modal_data": {
            "image": image
        },
    })

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


if __name__ == "__main__":
    run_llava()
```

1.10.8 Llava Next Example

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

```
from io import BytesIO

import requests
from PIL import Image

from vllm import LLM, SamplingParams


def run_llava_next():
    llm = LLM(model="llava-hf/llava-v1.6-mistral-7b-hf", max_model_len=4096)

    prompt = "[INST] <image>\nWhat is shown in this image? [/INST]"
```

(continues on next page)

(continued from previous page)

```
    url = "https://h2o-release.s3.amazonaws.com/h2ogpt/bigben.jpg"
    image = Image.open(BytesIO(requests.get(url).content))
    sampling_params = SamplingParams(temperature=0.8,
                                     top_p=0.95,
                                     max_tokens=100)

    outputs = llm.generate(
        {
            "prompt": prompt,
            "multi_modal_data": {
                "image": image
            }
        },
        sampling_params=sampling_params)

    generated_text = ""
    for o in outputs:
        generated_text += o.outputs[0].text

    print(f"LLM output:{generated_text}")


if __name__ == "__main__":
    run_llava_next()
```

1.10.9 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
from vllm.utils import FlexibleArgumentParser


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,
```

(continues on next page)

(continued from previous page)

```
                        temperature=0.0)),
    ]


def process_requests(engine: LLMEngine,
                     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 = FlexibleArgumentParser(
        description='Demo on using the LLMEngine class directly')
    parser = EngineArgs.add_cli_args(parser)
    args = parser.parse_args()
    main(args)
```

1.10.10 Lora With Quantization Inference

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

```
"""
This example shows how to use LoRA with different quantization techniques
for offline inference.

Requires HuggingFace credentials for access.
"""

import gc
from typing import List, Optional, Tuple

import torch
from huggingface_hub import snapshot_download

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]]]:
    return [
        # this is an example of using quantization without LoRA
        ("My name is",
         SamplingParams(temperature=0.0,
                        logprobs=1,
                        prompt_logprobs=1,
                        max_tokens=128), None),
        # the next three examples use quantization with LoRA
        ("my name is",
         SamplingParams(temperature=0.0,
                        logprobs=1,
                        prompt_logprobs=1,
                        max_tokens=128),
         LoRARequest("lora-test-1", 1, lora_path)),
        ("The capital of USA is",
         SamplingParams(temperature=0.0,
                        logprobs=1,
                        prompt_logprobs=1,
                        max_tokens=128),
         LoRARequest("lora-test-2", 1, lora_path)),
        ("The capital of France is",
         SamplingParams(temperature=0.0,
                        logprobs=1,
                        prompt_logprobs=1,
                        max_tokens=128),
         LoRARequest("lora-test-3", 1, lora_path)),
    ]


```

(continues on next page)

(continued from previous page)

```
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("----------------------------------------------------------")
                print(f"Prompt: {request_output.prompt}")
                print(f"Output: {request_output.outputs[0].text}")


def initialize_engine(model: str, quantization: str,
                      lora_repo: Optional[str]) -> LLMEngine:
    """Initialize the LLMEngine."""

    if quantization == "bitsandbytes":
        # QLoRA (https://arxiv.org/abs/2305.14314) is a quantization technique.
        # It quantizes the model when loading, with some config info from the
        # LoRA adapter repo. So need to set the parameter of load_format and
        # qlora_adapter_name_or_path as below.
        engine_args = EngineArgs(
            model=model,
            quantization=quantization,
            qlora_adapter_name_or_path=lora_repo,
            load_format="bitsandbytes",
            enable_lora=True,
            max_lora_rank=64,
            # set it only in GPUs of limited memory
            enforce_eager=True)
    else:
        engine_args = EngineArgs(
            model=model,
            quantization=quantization,
            enable_lora=True,
            max_loras=4,
            # set it only in GPUs of limited memory
            enforce_eager=True)
    return LLMEngine.from_engine_args(engine_args)


```

(continues on next page)

(continued from previous page)

```
def main():
    """Main function that sets up and runs the prompt processing."""

    test_configs = [{
        "name": "qlora_inference_example",
        'model': "huggyllama/llama-7b",
        'quantization': "bitsandbytes",
        'lora_repo': 'timdettmers/qlora-flan-7b'
    }, {
        "name": "AWQ_inference_with_lora_example",
        'model': 'TheBloke/TinyLlama-1.1B-Chat-v0.3-AWQ',
        'quantization': "awq",
        'lora_repo': 'jashing/tinyllama-colorist-lora'
    }, {
        "name": "GPTQ_inference_with_lora_example",
        'model': 'TheBloke/TinyLlama-1.1B-Chat-v0.3-GPTQ',
        'quantization': "gptq",
        'lora_repo': 'jashing/tinyllama-colorist-lora'
    }]

    for test_config in test_configs:
        print(
            f"~~~~~~~~~~~~~~~~ Running: {test_config['name']} ~~~~~~~~~~~~~~~~"
        )
        engine = initialize_engine(test_config['model'],
                                   test_config['quantization'],
                                   test_config['lora_repo'])
        lora_path = snapshot_download(repo_id=test_config['lora_repo'])
        test_prompts = create_test_prompts(lora_path)
        process_requests(engine, test_prompts)

        # Clean up the GPU memory for the next test
        del engine
        gc.collect()
        torch.cuda.empty_cache()


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

1.10.11 MultiLoRA Inference

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

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

Requires HuggingFace credentials for access to Llama2.
"""

```

(continues on next page)

(continued from previous page)

```python
from typing import List, Optional, Tuple

from huggingface_hub import snapshot_download

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)),
```

(continues on next page)

(continued from previous page)

```
        (
            "[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-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.
```

(continues on next page)

(continued from previous page)

```
    # 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.
    # 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()
```

1.10.12 Offline Inference

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

```
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}")
```

1.10.13 Offline Inference Arctic

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

```
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}")
```

1.10.14 Offline Inference Distributed

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

```
"""
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 Any, Dict, List

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)

```
    "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: List[str] = []
        generated_text: List[str] = []
        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: Dict[str, Any] = {}
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>")
```

1.10.15 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)

```
    "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
```

1.10.16 Offline Inference Mlpspeculator

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

```
import gc
import time
from typing import List

from vllm import LLM, SamplingParams


def time_generation(llm: LLM, prompts: List[str],
                    sampling_params: SamplingParams):
    # Generate texts from the prompts. The output is a list of RequestOutput
    # objects that contain the prompt, generated text, and other information.
    # Warmup first
    llm.generate(prompts, sampling_params)
    llm.generate(prompts, sampling_params)
    start = time.time()
    outputs = llm.generate(prompts, sampling_params)
    end = time.time()
    print((end - start) / sum([len(o.outputs[0].token_ids) for o in outputs]))
    # Print the outputs.
    for output in outputs:
        generated_text = output.outputs[0].text
        print(f"text: {generated_text!r}")


if __name__ == "__main__":

    template = (
        "Below is an instruction that describes a task. Write a response "
        "that appropriately completes the request.\n\n### Instruction:\n{}"
        "\n\n### Response:\n")

    # Sample prompts.
    prompts = [
        "Write about the president of the United States.",
    ]
    prompts = [template.format(prompt) for prompt in prompts]
```

(continues on next page)

(continued from previous page)

```
    # Create a sampling params object.
    sampling_params = SamplingParams(temperature=0.0, max_tokens=200)

    # Create an LLM without spec decoding
    llm = LLM(model="meta-llama/Llama-2-13b-chat-hf")

    print("Without speculation")
    time_generation(llm, prompts, sampling_params)

    del llm
    gc.collect()

    # Create an LLM with spec decoding
    llm = LLM(
        model="meta-llama/Llama-2-13b-chat-hf",
        speculative_model="ibm-fms/llama-13b-accelerator",
        # These are currently required for MLPSpeculator decoding
        use_v2_block_manager=True,
    )

    print("With speculation")
    time_generation(llm, prompts, sampling_params)
```

1.10.17 Offline Inference Neuron

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

```
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.
```

(continues on next page)

(continued from previous page)

```
    # 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}")
```

1.10.18 Offline Inference Tpu

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

```
from vllm import LLM, SamplingParams

prompts = [
    "A robot may not injure a human being",
    "It is only with the heart that one can see rightly;",
    "The greatest glory in living lies not in never falling,",
]
answers = [
    " or, through inaction, allow a human being to come to harm.",
    " what is essential is invisible to the eye.",
    " but in rising every time we fall.",
]
N = 1
# Currently, top-p sampling is disabled. `top_p` should be 1.0.
sampling_params = SamplingParams(temperature=0.7,
                                 top_p=1.0,
                                 n=N,
                                 max_tokens=16)

# Set `enforce_eager=True` to avoid ahead-of-time compilation.
# In real workloads, `enforace_eager` should be `False`.
llm = LLM(model="google/gemma-2b", enforce_eager=True)
outputs = llm.generate(prompts, sampling_params)
for output, answer in zip(outputs, answers):
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")
    assert generated_text.startswith(answer)
```

1.10.19 Offline Inference With Prefix

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

```
from time import time

from vllm import LLM, SamplingParams

# Common prefix.
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",
]

generating_prompts = [prefix + prompt for prompt in prompts]

# Create a sampling params object.
sampling_params = SamplingParams(temperature=0.0)

# Create an LLM.
regular_llm = LLM(model="facebook/opt-125m", gpu_memory_utilization=0.4)

prefix_cached_llm = LLM(model="facebook/opt-125m",
                        enable_prefix_caching=True,
                        gpu_memory_utilization=0.4)
print("Results without `enable_prefix_caching`")

# Generate texts from the prompts. The output is a list of RequestOutput objects
# that contain the prompt, generated text, and other information.
start_time_regular = time()
outputs = regular_llm.generate(generating_prompts, sampling_params)
duration_regular = time() - start_time_regular

regular_generated_texts = []
# Print the outputs.
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    regular_generated_texts.append(generated_text)
```

(continues on next page)

(continued from previous page)

```
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")

print("-" * 80)

# Warmup so that the shared prompt's KV cache is computed.
prefix_cached_llm.generate(generating_prompts[0], sampling_params)

# Generate with prefix caching.
start_time_cached = time()
outputs = prefix_cached_llm.generate(generating_prompts, sampling_params)
duration_cached = time() - start_time_cached

print("Results with `enable_prefix_caching`")

cached_generated_texts = []
# Print the outputs. You should see the same outputs as before.
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    cached_generated_texts.append(generated_text)
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")

print("-" * 80)

# Compare the results and display the speedup
generated_same = all([
    regular_generated_texts[i] == cached_generated_texts[i]
    for i in range(len(prompts))
])
print(f"Generated answers are the same: {generated_same}")

speedup = round(duration_regular / duration_cached, 2)
print(f"Speed up of cached generation compared to the regular is: {speedup}")
```

1.10.20 OpenAI Chat Completion Client

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

```
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()
```

(continues on next page)

(continued from previous page)

```
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)
```

1.10.21 OpenAI Completion Client

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

```
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,
```

(continues on next page)

(continued from previous page)

```
    logprobs=3)

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

1.10.22 OpenAI Embedding Client

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

```
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

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
```

1.10.23 OpenAI Vision API Client

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

```
"""An example showing how to use vLLM to serve VLMs.

Launch the vLLM server with the following command:
vllm serve llava-hf/llava-1.5-7b-hf --chat-template template_llava.jinja
"""
import base64

import requests
from openai import OpenAI
```

(continues on next page)

(continued from previous page)

```

# 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

image_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/d/dd/Gfp-wisconsin-
↪madison-the-nature-boardwalk.jpg/2560px-Gfp-wisconsin-madison-the-nature-boardwalk.jpg"

# Use image url in the payload
chat_completion_from_url = client.chat.completions.create(
    messages=[{
        "role":
        "user",
        "content": [
            {
                "type": "text",
                "text": "What's in this image?"
            },
            {
                "type": "image_url",
                "image_url": {
                    "url": image_url
                },
            },
        ],
    }],
    model=model,
)

result = chat_completion_from_url.choices[0].message.content
print(f"Chat completion output:{result}")


# Use base64 encoded image in the payload
def encode_image_base64_from_url(image_url: str) -> str:
    """Encode an image retrieved from a remote url to base64 format."""

    with requests.get(image_url) as response:
        response.raise_for_status()
        result = base64.b64encode(response.content).decode('utf-8')

    return result

```

(continues on next page)

(continued from previous page)

```

image_base64 = encode_image_base64_from_url(image_url=image_url)
chat_completion_from_base64 = client.chat.completions.create(
    messages=[{
        "role":
        "user",
        "content": [
            {
                "type": "text",
                "text": "What's in this image?"
            },
            {
                "type": "image_url",
                "image_url": {
                    "url": f"{image_base64}"
                },
            },
        ],
    }],
    model=model,
)

result = chat_completion_from_base64.choices[0].message.content
print(f"Chat completion output:{result}")
```

1.10.24 Paligemma Example

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

```
from vllm import LLM
from vllm.assets.image import ImageAsset


def run_paligemma():
    llm = LLM(model="google/paligemma-3b-mix-224")

    prompt = "caption es"

    image = ImageAsset("stop_sign").pil_image

    outputs = llm.generate({
        "prompt": prompt,
        "multi_modal_data": {
            "image": image
        },
    })

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

```

(continues on next page)

(continued from previous page)

```

if __name__ == "__main__":
    run_paligemma()
```

1.10.25 Phi3V Example

Source https://github.com/vllm-project/vllm/blob/main/examples/phi3v_example.py.

```
from vllm import LLM, SamplingParams
from vllm.assets.image import ImageAsset


def run_phi3v():
    model_path = "microsoft/Phi-3-vision-128k-instruct"

    # Note: The default setting of max_num_seqs (256) and
    # max_model_len (128k) for this model may cause OOM.
    # You may lower either to run this example on lower-end GPUs.

    # In this example, we override max_num_seqs to 5 while
    # keeping the original context length of 128k.
    llm = LLM(
        model=model_path,
        trust_remote_code=True,
        max_num_seqs=5,
    )

    image = ImageAsset("cherry_blossom").pil_image

    # single-image prompt
    prompt = "<|user|>\n<|image_1|>\nWhat is the season?<|end|>\n<|assistant|>\n"  # noqa: E501
    sampling_params = SamplingParams(temperature=0, max_tokens=64)

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


if __name__ == "__main__":
    run_phi3v()
```

1.10.26 Save Sharded State

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

```
"""
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 dataclasses
import os
import shutil
from pathlib import Path

from vllm import LLM, EngineArgs
from vllm.utils import FlexibleArgumentParser

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

(continues on next page)

(continued from previous page)

```
        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)
```

1.10.27 Tensorize vLLM Model

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

```
import argparse
import dataclasses
import json
import os
import uuid

from vllm import LLM
from vllm.engine.arg_utils import EngineArgs
from vllm.model_executor.model_loader.tensorizer import (TensorizerArgs,
                                                         TensorizerConfig,
                                                         tensorize_vllm_model)
from vllm.utils import FlexibleArgumentParser

# 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
```

(continues on next page)

(continued from previous page)

```
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

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.

To support distributed tensor-parallel models, each model shard will be
serialized to a separate file. The tensorizer_uri is then specified as a string
template with a format specifier such as '%03d' that will be rendered with the
shard's rank. Sharded models serialized with this script will be named as
model-rank-%03d.tensors

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
```

(continues on next page)

(continued from previous page)

```
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 = FlexibleArgumentParser(
        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`")

    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."))
```

(continues on next page)

(continued from previous page)

```
    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)

    return parser.parse_args()



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

```

(continues on next page)

(continued from previous page)

```python

if __name__ == '__main__':
    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
    }

    model_ref = args.model

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

    keyfile = args.keyfile if args.keyfile else None

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

        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}"
        if engine_args.tensor_parallel_size > 1:
            model_path = f"{base_path}/model-rank-%03d.tensors"
        else:
            model_path = f"{base_path}/model.tensors"

        tensorizer_config = TensorizerConfig(
            tensorizer_uri=model_path,
            encryption_keyfile=keyfile,
            **credentials)

```

(continues on next page)

(continued from previous page)

```
        tensorize_vllm_model(engine_args, tensorizer_config)

    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.")
```

1.11 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*:

```
vllm serve 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.

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

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

1.11.1 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`.

vLLM also provides experimental support for OpenAI Vision API compatible inference. See more details in *Using VLMs*.

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

The following extra parameters are supported:

```
    echo: bool = Field(
        default=False,
        description=(
            "If true, the new message will be prepended with the last message "
            "if they belong to the same role."),
    )
    add_generation_prompt: 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."),
    )
    add_special_tokens: bool = Field(
        default=False,
        description=(
            "If true, special tokens (e.g. BOS) will be added to the prompt "
```

(continues on next page)

(continued from previous page)

```
            "on top of what is added by the chat template. "
            "For most models, the chat template takes care of adding the "
            "special tokens so this should be set to false (as is the "
            "default)."),
    )
    documents: Optional[List[Dict[str, str]]] = Field(
        default=None,
        description=
        ("A list of dicts representing documents that will be accessible to "
         "the model if it is performing RAG (retrieval-augmented generation)."
         " If the template does not support RAG, this argument will have no "
         "effect. We recommend that each document should be a dict containing "
         "\"title\" and \"text\" keys."),
    )
    chat_template: Optional[str] = Field(
        default=None,
        description=(
            "A Jinja template to use for this conversion. "
            "If this is not passed, the model's default chat template will be "
            "used instead."),
    )
    chat_template_kwargs: Optional[Dict[str, Any]] = Field(
        default=None,
        description=("Additional kwargs to pass to the template renderer. "
                     "Will be accessible by the chat template."),
    )
    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(
```

(continues on next page)

(continued from previous page)

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

The following extra parameters are supported:

```
    add_special_tokens: bool = Field(
        default=True,
        description=(
            "If true (the default), special tokens (e.g. BOS) will be added to "
            "the prompt."),
    )
    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=(
```

(continues on next page)

(continued from previous page)

```
            "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 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."))
```

1.11.3 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.

```
vllm serve <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

1.11.4 Command line arguments for the server

1.11.5 Tool calling in the chat completion API

vLLM supports only named function calling in the chat completion API. The `tool_choice` options `auto` and `required` are **not yet supported** but on the roadmap.

To use a named function you need to define the function in the `tools` parameter and call it in the `tool_choice` parameter.

It is the callers responsibility to prompt the model with the tool information, vLLM will not automatically manipulate the prompt. **This may change in the future.**

vLLM will use guided decoding to ensure the response matches the tool parameter object defined by the JSON schema in the `tools` parameter.

Please refer to the OpenAI API reference documentation for more information.

1.12 Deploying with Docker

vLLM offers an official Docker image for deployment. The image can be used to run OpenAI compatible server and 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` .

1.13 Distributed Inference and Serving

1.13.1 How to decide the distributed inference strategy?

Before going into the details of distributed inference and serving, let's first make it clear when to use distributed inference and what are the strategies available. The common practice is:

- **Single GPU (no distributed inference)**: If your model fits in a single GPU, you probably don't need to use distributed inference. Just use the single GPU to run the inference.
- **Single-Node Multi-GPU (tensor parallel inference)**: If your model is too large to fit in a single GPU, but it can fit in a single node with multiple GPUs, you can use tensor parallelism. The tensor parallel size is the number of GPUs you want to use. For example, if you have 4 GPUs in a single node, you can set the tensor parallel size to 4.
- **Multi-Node Multi-GPU (tensor parallel plus pipeline parallel inference)**: If your model is too large to fit in a single node, you can use tensor parallel together with pipeline parallelism. The tensor parallel size is the number of GPUs you want to use in each node, and the pipeline parallel size is the number of nodes you want to use. For example, if you have 16 GPUs in 2 nodes (8GPUs per node), you can set the tensor parallel size to 8 and the pipeline parallel size to 2.

In short, you should increase the number of GPUs and the number of nodes until you have enough GPU memory to hold the model. The tensor parallel size should be the number of GPUs in each node, and the pipeline parallel size should be the number of nodes.

After adding enough GPUs and nodes to hold the model, you can run vLLM first, which will print some logs like `# GPU blocks: 790`. Multiply the number by `16` (the block size), and you can get roughly the maximum number of tokens that can be served on the current configuration. If this number is not satisfying, e.g. you want higher throughput, you can further increase the number of GPUs or nodes, until the number of blocks is enough.

Note: There is one edge case: if the model fits in a single node with multiple GPUs, but the number of GPUs cannot divide the model size evenly, you can use pipeline parallelism, which splits the model along layers and supports uneven splits. In this case, the tensor parallel size should be 1 and the pipeline parallel size should be the number of GPUs.

1.13.2 Details for Distributed Inference and Serving

vLLM supports distributed tensor-parallel inference and serving. Currently, we support Megatron-LM's tensor parallel algorithm. We also support pipeline parallel as a beta feature for online serving. We manage the distributed runtime with either Ray or python native multiprocessing. Multiprocessing can be used when deploying on a single node, multi-node inferencing currently requires Ray.

Multiprocessing will be used by default when not running in a Ray placement group and if there are sufficient GPUs available on the same node for the configured `tensor_parallel_size`, otherwise Ray will be used. This default can be overridden via the `LLM` class `distributed-executor-backend` argument or `--distributed-executor-backend` API server argument. Set it to `mp` for multiprocessing or `ray` for Ray. It's not required for Ray to be installed for the multiprocessing case.

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:

```
$ vllm serve facebook/opt-13b \
$     --tensor-parallel-size 4
```

You can also additionally specify `--pipeline-parallel-size` to enable pipeline parallelism. For example, to run API server on 8 GPUs with pipeline parallelism and tensor parallelism:

```
$ vllm serve gpt2 \
$     --tensor-parallel-size 4 \
$     --pipeline-parallel-size 2
```

Note: Pipeline parallel is a beta feature. It is only supported for online serving as well as LLaMa, GPT2, and Mixtral style models.

1.13.3 Multi-Node Inference and Serving

If a single node does not have enough GPUs to hold the model, you can run the model using multiple nodes. It is important to make sure the execution environment is the same on all nodes, including the model path, the Python environment. The recommended way is to use docker images to ensure the same environment, and hide the heterogeneity of the host machines via mapping them into the same docker configuration.

The first step, is to start containers and organize them into a cluster. We have provided a helper script to start the cluster.

Pick a node as the head node, and run the following command:

```
$ bash run_cluster.sh \
$                   vllm/vllm-openai \
$                   ip_of_head_node \
$                   --head \
$                   /path/to/the/huggingface/home/in/this/node
```

On the rest of the worker nodes, run the following command:

```
$ bash run_cluster.sh \
$                   vllm/vllm-openai \
$                   ip_of_head_node \
$                   --worker \
$                   /path/to/the/huggingface/home/in/this/node
```

Then you get a ray cluster of containers. Note that you need to keep the shells running these commands alive to hold the cluster. Any shell disconnect will terminate the cluster.

Then, on any node, use `docker exec -it node /bin/bash` to enter the container, execute `ray status` to check the status of the Ray cluster. You should see the right number of nodes and GPUs.

After that, on any node, you can use vLLM as usual, just as you have all the GPUs on one node. The common practice is to set the tensor parallel size to the number of GPUs in each node, and the pipeline parallel size to the number of nodes. For example, if you have 16 GPUs in 2 nodes (8GPUs per node), you can set the tensor parallel size to 8 and the pipeline parallel size to 2:

```
$ vllm serve /path/to/the/model/in/the/container \
$     --tensor-parallel-size 8 \
$     --pipeline-parallel-size 2
```

You can also use tensor parallel without pipeline parallel, just set the tensor parallel size to the number of GPUs in the cluster. For example, if you have 16 GPUs in 2 nodes (8GPUs per node), you can set the tensor parallel size to 16:

```
$ vllm serve /path/to/the/model/in/the/container \
$     --tensor-parallel-size 16
```

To make tensor parallel performant, you should make sure the communication between nodes is efficient, e.g. using high-speed network cards like Infiniband. To correctly set up the cluster to use Infiniband, append additional arguments like `--privileged -e NCCL_IB_HCA=mlx5` to the `run_cluster.sh` script. Please contact your system administrator for more information on how to set up the flags. One way to confirm if the Infiniband is working is to run vLLM with `NCCL_DEBUG=TRACE` environment variable set, e.g. `NCCL_DEBUG=TRACE vllm serve ...` and check the logs for the NCCL version and the network used. If you find `[send] via NET/Socket` in the logs, it means NCCL uses raw TCP Socket, which is not efficient for cross-node tensor parallel. If you find `[send] via NET/IB/GDRDMA` in the logs, it means NCCL uses Infiniband with GPU-Direct RDMA, which is efficient.

Warning: After you start the Ray cluster, you'd better also check the GPU-GPU communication between nodes. It can be non-trivial to set up. Please refer to the sanity check script for more information.

Warning: Please make sure you downloaded the model to all the nodes (with the same path), or the model is downloaded to some distributed file system that is accessible by all nodes.

When you use huggingface repo id to refer to the model, you should append your huggingface token to the `run_cluster.sh` script, e.g. `-e HF_TOKEN=`. The recommended way is to download the model first, and then use the path to refer to the model.

1.14 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"
    _gauge_cls = prometheus_client.Gauge
    _counter_cls = prometheus_client.Counter
    _histogram_cls = prometheus_client.Histogram

    def __init__(self, labelnames: List[str], max_model_len: int):
        # Unregister any existing vLLM collectors
        self._unregister_vllm_metrics()

        # Config Information
        self._create_info_cache_config()
```

(continues on next page)

(continued from previous page)

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

        # Iteration stats
        self.counter_num_preemption = self._counter_cls(
            name="vllm:num_preemptions_total",
            documentation="Cumulative number of preemption from the engine.",
            labelnames=labelnames)
        self.counter_prompt_tokens = self._counter_cls(
            name="vllm:prompt_tokens_total",
            documentation="Number of prefill tokens processed.",
            labelnames=labelnames)
        self.counter_generation_tokens = self._counter_cls(
            name="vllm:generation_tokens_total",
            documentation="Number of generation tokens processed.",
            labelnames=labelnames)
        self.histogram_time_to_first_token = self._histogram_cls(
            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 = self._histogram_cls(
            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,
                1.0, 2.5
```

(continues on next page)

(continued from previous page)

```
            ])

        # Request stats
        #   Latency
        self.histogram_e2e_time_request = self._histogram_cls(
            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 = self._histogram_cls(
            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 = \
            self._histogram_cls(
                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 = self._histogram_cls(
            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 = self._histogram_cls(
            name="vllm:request_params_n",
            documentation="Histogram of the n request parameter.",
            labelnames=labelnames,
            buckets=[1, 2, 5, 10, 20],
        )
        self.counter_request_success = self._counter_cls(
            name="vllm:request_success_total",
            documentation="Count of successfully processed requests.",
            labelnames=labelnames + [Metrics.labelname_finish_reason])

        # Speculatie decoding stats
        self.gauge_spec_decode_draft_acceptance_rate = self._gauge_cls(
            name="vllm:spec_decode_draft_acceptance_rate",
            documentation="Speulative token acceptance rate.",
            labelnames=labelnames)
        self.gauge_spec_decode_efficiency = self._gauge_cls(
            name="vllm:spec_decode_efficiency",
            documentation="Speculative decoding system efficiency.",
            labelnames=labelnames)
        self.counter_spec_decode_num_accepted_tokens = (self._counter_cls(
            name="vllm:spec_decode_num_accepted_tokens_total",
            documentation="Number of accepted tokens.",
```

(continues on next page)

(continued from previous page)

```
            labelnames=labelnames))
        self.counter_spec_decode_num_draft_tokens = self._counter_cls(
            name="vllm:spec_decode_num_draft_tokens_total",
            documentation="Number of draft tokens.",
            labelnames=labelnames)
        self.counter_spec_decode_num_emitted_tokens = (self._counter_cls(
            name="vllm:spec_decode_num_emitted_tokens_total",
            documentation="Number of emitted tokens.",
            labelnames=labelnames))

        # Deprecated in favor of vllm:prompt_tokens_total
        self.gauge_avg_prompt_throughput = self._gauge_cls(
            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 = self._gauge_cls(
            name="vllm:avg_generation_throughput_toks_per_s",
            documentation="Average generation throughput in tokens/s.",
            labelnames=labelnames,
        )

    def _create_info_cache_config(self) -> None:
        # Config Information
        self.info_cache_config = prometheus_client.Info(
            name='vllm:cache_config',
            documentation='information of cache_config')

    def _unregister_vllm_metrics(self) -> None:
        for collector in list(prometheus_client.REGISTRY._collector_to_names):
            if hasattr(collector, "_name") and "vllm" in collector._name:
                prometheus_client.REGISTRY.unregister(collector)
```

1.15 Environment Variables

vLLM uses the following environment variables to configure the system:

Warning: Please note that `VLLM_PORT` and `VLLM_HOST_IP` set the port and ip for vLLM's **internal usage**. It is not the port and ip for the API server. If you use `--host $VLLM_HOST_IP` and `--port $VLLM_PORT` to start the API server, it will not work.

All environment variables used by vLLM are prefixed with `VLLM_`. **Special care should be taken for Kubernetes users**: please do not name the service as `vllm`, otherwise environment variables set by Kubernetes might conflict with vLLM's environment variables, because Kubernetes sets environment variables for each service with the capitalized service name as the prefix.

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

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

    # Target device of vLLM, supporting [cuda (by default),
    # rocm, neuron, cpu, openvino]
    "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 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
    # Defaults to `~/.config/vllm` unless `XDG_CONFIG_HOME` is set
    # 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.path.expanduser(
        os.getenv(
            "VLLM_CONFIG_ROOT",
            os.path.join(get_default_config_root(), "vllm"),
        )),

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

(continues on next page)

(continued from previous page)

```
    # Root directory for VLLM cache files
    # Defaults to `~/.cache/vllm` unless `XDG_CACHE_HOME` is set
    "VLLM_CACHE_ROOT":
    lambda: os.path.expanduser(
        os.getenv(
            "VLLM_CACHE_ROOT",
            os.path.join(get_default_cache_root(), "vllm"),
        )),

    # 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
    # Note: if VLLM_PORT is set, and some code asks for multiple ports, the
    # VLLM_PORT will be used as the first port, and the rest will be generated
    # by incrementing the VLLM_PORT value.
    # '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),

    # Interval in seconds to log a warning message when the ring buffer is full
    "VLLM_RINGBUFFER_WARNING_INTERVAL":
    lambda: int(os.environ.get("VLLM_RINGBUFFER_WARNING_INTERVAL", "60")),

    # 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
```

(continues on next page)

(continued from previous page)

```
    "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":
    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"),
```

(continues on next page)

(continued from previous page)

```
    # 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")),

    # Backend for attention computation
    # Available options:
    # - "TORCH_SDPA": use torch.nn.MultiheadAttention
    # - "FLASH_ATTN": use FlashAttention
    # - "XFORMERS": use XFormers
    # - "ROCM_FLASH": use ROCmFlashAttention
    # - "FLASHINFER": use flashinfer
    "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")),

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

    # OpenVINO KV cache precision
    # default is bf16 if natively supported by platform, otherwise f16
    # To enable KV cache compression, please, explicitly specify u8
    "VLLM_OPENVINO_CPU_KV_CACHE_PRECISION":
    lambda: os.getenv("VLLM_OPENVINO_CPU_KV_CACHE_PRECISION", None),

    # Enables weights compression during model export via HF Optimum
    # default is False
    "VLLM_OPENVINO_ENABLE_QUANTIZED_WEIGHTS":
    lambda: bool(os.getenv("VLLM_OPENVINO_ENABLE_QUANTIZED_WEIGHTS", False)),

    # If the env var is set, then all workers will execute as separate
    # processes from the engine, and we use the same mechanism to trigger
    # execution on all workers.
    # Run vLLM with VLLM_USE_RAY_SPMD_WORKER=1 to enable it.
    "VLLM_USE_RAY_SPMD_WORKER":
    lambda: bool(os.getenv("VLLM_USE_RAY_SPMD_WORKER", 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.
```

(continues on next page)

(continued from previous page)

```
    # Both spawn and fork work
    "VLLM_WORKER_MULTIPROC_METHOD":
    lambda: os.getenv("VLLM_WORKER_MULTIPROC_METHOD", "fork"),

    # Path to the cache for storing downloaded assets
    "VLLM_ASSETS_CACHE":
    lambda: os.path.expanduser(
        os.getenv(
            "VLLM_ASSETS_CACHE",
            os.path.join(get_default_cache_root(), "vllm", "assets"),
        )),

    # Timeout for fetching images when serving multimodal models
    # Default is 5 seconds
    "VLLM_IMAGE_FETCH_TIMEOUT":
    lambda: int(os.getenv("VLLM_IMAGE_FETCH_TIMEOUT", "5")),

    # Path to the XLA persistent cache directory.
    # Only used for XLA devices such as TPUs.
    "VLLM_XLA_CACHE_PATH":
    lambda: os.path.expanduser(
        os.getenv(
            "VLLM_ASSETS_CACHE",
            os.path.join(get_default_cache_root(), "vllm", "xla_cache"),
        )),
    "VLLM_FUSED_MOE_CHUNK_SIZE":
    lambda: int(os.getenv("VLLM_FUSED_MOE_CHUNK_SIZE", "65536")),

    # If set, vllm will skip the deprecation warnings.
    "VLLM_NO_DEPRECATION_WARNING":
    lambda: bool(int(os.getenv("VLLM_NO_DEPRECATION_WARNING", "0"))),
}
```

1.16 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.

1.16.1 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,
  "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
```

1.16.2 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
```

1.17 Integrations

1.17.1 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 \
```

(continues on next page)

(continued from previous page)

```
        --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 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
```

(continues on next page)

(continued from previous page)

```
        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␣
↪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

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
        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
```

1.17.2 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.

1.17.3 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.

1.17.4 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.

1.17.5 Deploying with Cerebrium

vLLM can be run on a cloud based GPU machine with Cerebrium, a serverless AI infrastructure platform that makes it easier for companies to build and deploy AI based applications.

To install the Cerebrium client, run:

```
$ pip install cerebrium
$ cerebrium login
```

Next, create your Cerebrium project, run:

```
$ cerebrium init vllm-project
```

Next, to install the required packages, add the following to your cerebrium.toml:

```
[cerebrium.deployment]
docker_base_image_url = "nvidia/cuda:12.1.1-runtime-ubuntu22.04"

[cerebrium.dependencies.pip]
vllm = "latest"
```

Next, let us add our code to handle inference for the LLM of your choice(*mistralai/Mistral-7B-Instruct-v0.1* for this example), add the following code to your main.py`:

```
from vllm import LLM, SamplingParams

llm = LLM(model="mistralai/Mistral-7B-Instruct-v0.1")

def run(prompts: list[str], temperature: float = 0.8, top_p: float = 0.95):

    sampling_params = SamplingParams(temperature=temperature, top_p=top_p)
    outputs = llm.generate(prompts, sampling_params)
```

(continues on next page)

(continued from previous page)

```
    # Print the outputs.
    results = []
    for output in outputs:
        prompt = output.prompt
        generated_text = output.outputs[0].text
        results.append({"prompt": prompt, "generated_text": generated_text})

    return {"results": results}
```

Then, run the following code to deploy it to the cloud

```
$ cerebrium deploy
```

If successful, you should be returned a CURL command that you can call inference against. Just remember to end the url with the function name you are calling (in our case /run)

```
curl -X POST https://api.cortex.cerebrium.ai/v4/p-xxxxxx/vllm/run \
 -H 'Content-Type: application/json' \
 -H 'Authorization: <JWT TOKEN>' \
 --data '{
   "prompts": [
     "Hello, my name is",
     "The president of the United States is",
     "The capital of France is",
     "The future of AI is"
   ]
 }'
```

You should get a response like:

```
{
    "run_id": "52911756-3066-9ae8-bcc9-d9129d1bd262",
    "result": {
        "result": [
            {
                "prompt": "Hello, my name is",
                "generated_text": " Sarah, and I'm a teacher. I teach elementary school␣
↪students. One of"
            },
            {
                "prompt": "The president of the United States is",
                "generated_text": " elected every four years. This is a democratic␣
↪system.\n\n5. What"
            },
            {
                "prompt": "The capital of France is",
                "generated_text": " Paris.\n"
            },
            {
                "prompt": "The future of AI is",
                "generated_text": " bright, but it's important to approach it with a␣
↪balanced and nuanced perspective."
            }
```

(continues on next page)

(continued from previous page)

```
        ]
    },
    "run_time_ms": 152.53663063049316
}
```

You now have an autoscaling endpoint where you only pay for the compute you use!

1.17.6 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.

1.17.7 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
    - vllm serve $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      -
 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

1.17.8 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.

1.18 Loading Models with CoreWeave's Tensorizer

vLLM supports loading models with CoreWeave's Tensorizer. vLLM model tensors that have been serialized to disk, an HTTP/HTTPS endpoint, or S3 endpoint can be deserialized at runtime extremely quickly directly to the GPU, resulting in significantly shorter Pod startup times and CPU memory usage. Tensor encryption is also supported.

For more information on CoreWeave's Tensorizer, please refer to CoreWeave's Tensorizer documentation. For more information on serializing a vLLM model, as well a general usage guide to using Tensorizer with vLLM, see the vLLM example script.

1.19 Frequently Asked Questions

Q: How can I serve multiple models on a single port using the OpenAI API?

A: Assuming that you're referring to using OpenAI compatible server to serve multiple models at once, that is not currently supported, you can run multiple instances of the server (each serving a different model) at the same time, and have another layer to route the incoming request to the correct server accordingly.

Q: Which model to use for offline inference embedding?

A: If you want to use an embedding model, try: https://huggingface.co/intfloat/e5-mistral-7b-instruct. Instead models, such as Llama-3-8b, Mistral-7B-Instruct-v0.3, are generation models rather than an embedding model

1.20 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.

1.20.1 Decoder-only Language Models

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.	
`Gemma2ForCausalLM`	Gemma2	`google/gemma-2-9b`, `google/gemma-2-27b`, etc.	
`GPT2LMHeadModel`	GPT-2	`gpt2`, `gpt2-xl`, etc.	
`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.	
`JambaForCausalLM`	Jamba	`ai21labs/Jamba-v0.1`, etc.	

continues on next page

Table 1 – continued from previous page

Architecture	Models	Example HuggingFace Models	LoRA
LlamaForCausalLM	Llama 3.1, Llama 3, Llama 2, LLaMA, Yi	meta-llama/Meta-Llama-3.1-405B-Instruct, meta-llama/Meta-Llama-3.1-70B, meta-llama/Meta-Llama-3-70B-Instruct, meta-llama/Llama-2-70b-hf, 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, microsoft/Phi-3-medium-128k-instruct, etc.	
Phi3SmallForCausalLM	Phi-3-Small	microsoft/Phi-3-small-8k-instruct, microsoft/Phi-3-small-128k-instruct, etc.	
PersimmonForCausalLM	Persimmon	adept/persimmon-8b-base, adept/persimmon-8b-chat, 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.	
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.	

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

1.20.2 Vision Language Models

Architecture	Models	Example HuggingFace Models	*LoRA*
`ChameleonForConditio`	Chameleon	`facebook/chameleon-7b` etc.	
`FuyuForCausalLM`	Fuyu	`adept/fuyu-8b` etc.	
`LlavaForConditionalG`	LLaVA-1.5	`llava-hf/llava-1.5-7b-hf`, `llava-hf/llava-1.5-13b-hf`, etc.	
`LlavaNextForConditio`	LLaVA-NeXT	`llava-hf/llava-v1.6-mistral-7b-hf`, `llava-hf/llava-v1.6-vicuna-7b-hf`, etc.	
`PaliGemmaForConditio`	PaliGemma	`google/paligemma-3b-pt-224`, `google/paligemma-3b-mix-224`, etc.	
`Phi3VForCausalLM`	Phi-3-Vision	`microsoft/Phi-3-vision-128k-instruct`, 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* and *Enabling Multimodal Inputs* for instructions on how to implement support for your model. Alternatively, you can raise an issue on our GitHub project.

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)
```

1.21 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.

1.22 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.

Note: By default, vLLM models do not support multi-modal inputs. To enable multi-modal support, please follow *this guide* after implementing the model here.

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!

1.22.1 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.

1.22.2 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.

1.22.3 2. Rewrite the `forward` methods

Next, you need to rewrite the `forward()` method 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,
```

(continues on next page)

(continued from previous page)

```
-     past_key_values: Optional[List[torch.FloatTensor]] = None,
-     inputs_embeds: Optional[torch.FloatTensor] = None,
-     labels: Optional[torch.LongTensor] = None,
-     use_cache: Optional[bool] = None,
-     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.

1.22.4 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 `torch.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.

1.22.5 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.

1.22.6 5. Register your model

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

1.22.7 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 `vllm serve <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>`.

1.23 Enabling Multimodal Inputs

This document walks you through the steps to extend a vLLM model so that it accepts *multi-modal* inputs.

See also:

Adding a New Model

1.23.1 1. Update the base vLLM model

It is assumed that you have already implemented the model in vLLM according to *these steps*. Further update the model as follows:

- Implement the `SupportsVision` interface.

```
+ from vllm.model_executor.models.interfaces import SupportsVision

- class YourModelForImage2Seq(nn.Module):
+ class YourModelForImage2Seq(nn.Module, SupportsVision):
```

Note: The model class does not have to be named `*ForCausalLM`. Check out the HuggingFace Transformers documentation for some examples.

- If you haven't already done so, reserve a keyword parameter in `forward()` for each input tensor that corresponds to a multi-modal input, as shown in the following example:

```
  def forward(
      self,
      input_ids: torch.Tensor,
      positions: torch.Tensor,
      kv_caches: List[torch.Tensor],
      attn_metadata: AttentionMetadata,
+     pixel_values: torch.Tensor,
  ) -> SamplerOutput:
```

1.23.2 2. Register input mappers

For each modality type that the model accepts as input, decorate the model class with `MULTIMODAL_REGISTRY.register_input_mapper`. This decorator accepts a function that maps multi-modal inputs to the keyword arguments you have previously defined in `forward()`.

```
  from vllm.model_executor.models.interfaces import SupportsVision
+ from vllm.multimodal import MULTIMODAL_REGISTRY

+ @MULTIMODAL_REGISTRY.register_image_input_mapper()
  class YourModelForImage2Seq(nn.Module, SupportsVision):
```

A default mapper is available for each modality in the core vLLM library. This input mapper will be used if you do not provide your own function.

See also:

Input Processing Pipeline

1.23.3 3. Register maximum number of multi-modal tokens

For each modality type that the model accepts as input, calculate the maximum possible number of tokens and register it via `INPUT_REGISTRY.register_dummy_data`.

```
  from vllm.inputs import INPUT_REGISTRY
  from vllm.model_executor.models.interfaces import SupportsVision
  from vllm.multimodal import MULTIMODAL_REGISTRY

  @MULTIMODAL_REGISTRY.register_image_input_mapper()
+ @MULTIMODAL_REGISTRY.register_max_image_tokens(<your_calculation>)
  @INPUT_REGISTRY.register_dummy_data(<your_dummy_data_factory>)
  class YourModelForImage2Seq(nn.Module, SupportsVision):
```

Here are some examples:

- Image inputs (static feature size): LLaVA-1.5 Model
- Image inputs (dynamic feature size): LLaVA-NeXT Model

See also:

Input Processing Pipeline

1.23.4 4. (Optional) Register dummy data

During startup, dummy data is passed to the vLLM model to allocate memory. This only consists of text input by default, which may not be applicable to multi-modal models. In such cases, you can define your own dummy data by registering a factory method via *INPUT_REGISTRY.register_dummy_data*.

```
  from vllm.inputs import INPUT_REGISTRY
  from vllm.model_executor.models.interfaces import SupportsVision
  from vllm.multimodal import MULTIMODAL_REGISTRY

  @MULTIMODAL_REGISTRY.register_image_input_mapper()
  @MULTIMODAL_REGISTRY.register_max_image_tokens(<your_calculation>)
+ @INPUT_REGISTRY.register_dummy_data(<your_dummy_data_factory>)
  class YourModelForImage2Seq(nn.Module, SupportsVision):
```

Note: The dummy data should have the maximum possible number of multi-modal tokens, as described in the previous step.

Here are some examples:

- Image inputs (static feature size): LLaVA-1.5 Model
- Image inputs (dynamic feature size): LLaVA-NeXT Model

See also:

Input Processing Pipeline

1.23.5 5. (Optional) Register input processor

Sometimes, there is a need to process inputs at the `LLMEngine` level before they are passed to the model executor. This is often due to the fact that unlike implementations in HuggingFace Transformers, the reshaping and/or expansion of multi-modal embeddings needs to take place outside model's `forward()` call. You can register input processors via *INPUT_REGISTRY.register_input_processor*.

```
  from vllm.inputs import INPUT_REGISTRY
  from vllm.model_executor.models.interfaces import SupportsVision
  from vllm.multimodal import MULTIMODAL_REGISTRY

  @MULTIMODAL_REGISTRY.register_image_input_mapper()
  @MULTIMODAL_REGISTRY.register_max_image_tokens(<your_calculation>)
  @INPUT_REGISTRY.register_dummy_data(<your_dummy_data_factory>)
+ @INPUT_REGISTRY.register_input_processor(<your_input_processor>)
  class YourModelForImage2Seq(nn.Module, SupportsVision):
```

A common use case of input processors is inserting placeholder tokens to leverage the vLLM framework for attention mask generation. Here are some examples:

- Insert static number of image tokens: LLaVA-1.5 Model
- Insert dynamic number of image tokens: LLaVA-NeXT Model

See also:

Input Processing Pipeline

1.24 Engine Arguments

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

```
usage: vllm serve [-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,
↪bitsandbytes}]
                  [--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] [--cpu-offload-gb CPU_OFFLOAD_GB]
                  [--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,fbgemm_fp8,marlin,gptq_
↪marlin_24,gptq_marlin,awq_marlin,gptq,squeezellm,compressed-tensors,bitsandbytes,None}]
                  [--rope-scaling ROPE_SCALING] [--rope-theta ROPE_THETA]
                  [--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_SCALING_FACTORS]
                  [--max-cpu-loras MAX_CPU_LORAS] [--fully-sharded-loras]
                  [--enable-prompt-adapter]
```

(continues on next page)

(continued from previous page)

```
                  [--max-prompt-adapters MAX_PROMPT_ADAPTERS]
                  [--max-prompt-adapter-token MAX_PROMPT_ADAPTER_TOKEN]
                  [--device {auto,cuda,neuron,cpu,openvino,tpu,xpu}]
                  [--scheduler-delay-factor SCHEDULER_DELAY_FACTOR]
                  [--enable-chunked-prefill [ENABLE_CHUNKED_PREFILL]]
                  [--speculative-model SPECULATIVE_MODEL]
                  [--num-speculative-tokens NUM_SPECULATIVE_TOKENS]
                  [--speculative-draft-tensor-parallel-size SPECULATIVE_DRAFT_TENSOR_
↪PARALLEL_SIZE]
                  [--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]
                  [--spec-decoding-acceptance-method {rejection_sampler,typical_
↪acceptance_sampler}]
                  [--typical-acceptance-sampler-posterior-threshold TYPICAL_ACCEPTANCE_
↪SAMPLER_POSTERIOR_THRESHOLD]
                  [--typical-acceptance-sampler-posterior-alpha TYPICAL_ACCEPTANCE_
↪SAMPLER_POSTERIOR_ALPHA]
                  [--disable-logprobs-during-spec-decoding DISABLE_LOGPROBS_DURING_SPEC_
↪DECODING]
                  [--model-loader-extra-config MODEL_LOADER_EXTRA_CONFIG]
                  [--ignore-patterns IGNORE_PATTERNS]
                  [--preemption-mode PREEMPTION_MODE]
                  [--served-model-name SERVED_MODEL_NAME [SERVED_MODEL_NAME ...]]
                  [--qlora-adapter-name-or-path QLORA_ADAPTER_NAME_OR_PATH]
                  [--otlp-traces-endpoint OTLP_TRACES_ENDPOINT]
```

1.24.1 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. If unspecified, model name or path will be used.

--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 Revision of the huggingface tokenizer 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, bitsandbytes

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 Examples section for more information.
- "bitsandbytes" will load the weights using bitsandbytes quantization.

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

--cpu-offload-gb The space in GiB to offload to CPU, per GPU. Default is 0, which means no offloading. Intuitively, this argument can be seen as a virtual way to increase the GPU memory size. For example, if you have one 24 GB GPU and set this to 10, virtually you can think of it as a 34 GB GPU. Then you can load a 13B model with BF16 weight,which requires at least 26GB GPU memory. Note that this requires fast CPU-GPU interconnect, as part of the model isloaded from CPU memory to GPU memory on the fly in each model forward pass.

Default: 0

--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: 20

--disable-log-stats Disable logging statistics.

--quantization, -q Possible choices: aqlm, awq, deepspeedfp, fp8, fbgemm_fp8, marlin, gptq_marlin_24, gptq_marlin, awq_marlin, gptq, squeezellm, compressed-tensors, bitsandbytes, 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}

--rope-theta RoPE theta. Use with *rope_scaling*. In some cases, changing the RoPE theta improves the performance of the scaled model.

--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.

--enable-prompt-adapter If True, enable handling of PromptAdapters.

--max-prompt-adapters Max number of PromptAdapters in a batch.

Default: 1

--max-prompt-adapter-token Max number of PromptAdapters tokens

Default: 0

--device Possible choices: auto, cuda, neuron, cpu, openvino, tpu, xpu

Device type for vLLM execution.

Default: "auto"

--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-draft-tensor-parallel-size, -spec-draft-tp Number of tensor parallel replicas for 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.

--spec-decoding-acceptance-method Possible choices: rejection_sampler, typical_acceptance_sampler

Specify the acceptance method to use during draft token verification in speculative decoding. Two types of acceptance routines are supported: 1) RejectionSampler which does not allow changing the acceptance rate of draft tokens, 2) TypicalAcceptanceSampler which is configurable, allowing for a higher acceptance rate at the cost of lower quality, and vice versa.

Default: "rejection_sampler"

--typical-acceptance-sampler-posterior-threshold Set the lower bound threshold for the posterior probability of a token to be accepted. This threshold is used by the TypicalAcceptanceSampler to make sampling decisions during speculative decoding. Defaults to 0.09

--typical-acceptance-sampler-posterior-alpha A scaling factor for the entropy-based threshold for token acceptance in the TypicalAcceptanceSampler. Typically defaults to sqrt of –typical-acceptance-sampler-posterior-threshold i.e. 0.3

--disable-logprobs-during-spec-decoding If set to True, token log probabilities are not returned during speculative decoding. If set to False, log probabilities are returned according to the settings in SamplingParams. If not specified, it defaults to True. Disabling log probabilities during speculative decoding reduces latency by skipping logprob calculation in proposal sampling, target sampling, and after accepted tokens are determined.

--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.

--ignore-patterns The pattern(s) to ignore when loading the model.Default to 'original/**/*' to avoid repeated loading of llama's checkpoints.

Default: []

--preemption-mode If 'recompute', the engine performs preemption by block swapping; If 'swap', the engine performs preemption by block swapping.

--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.

--qlora-adapter-name-or-path Name or path of the QLoRA adapter.

--otlp-traces-endpoint Target URL to which OpenTelemetry traces will be sent.

1.24.2 Async Engine Arguments

Below are the additional arguments related to the asynchronous engine:

```
usage: vllm serve [-h] [--engine-use-ray] [--disable-log-requests]
```

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.

1.25 Using LoRA adapters

This document shows you how to use LoRA adapters with vLLM on top of a base model.

LoRA adapters can be used with any vLLM model that implements `SupportsLoRA`.

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]",
]

outputs = llm.generate(
```

(continues on next page)

(continued from previous page)

```
    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.

1.25.1 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:

```
vllm serve meta-llama/Llama-2-7b-hf \
    --enable-lora \
    --lora-modules sql-lora=$HOME/.cache/huggingface/hub/models--yard1--llama-2-7b-sql-
↪lora-test/snapshots/0dfa347e8877a4d4ed19ee56c140fa518470028c/
```

Note: The commit ID *0dfa347e8877a4d4ed19ee56c140fa518470028c* may change over time. Please check the latest commit ID in your environment to ensure you are using the correct one.

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 '{
```

(continues on next page)

(continued from previous page)

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

1.26 Using VLMs

vLLM provides experimental support for Vision Language Models (VLMs). See the *list of supported VLMs here*. This document shows you how to run and serve these models using vLLM.

Important: We are actively iterating on VLM support. Expect breaking changes to VLM usage and development in upcoming releases without prior deprecation.

Currently, the support for vision language models on vLLM has the following limitations:

- Only single image input is supported per text prompt.

We are continuously improving user & developer experience for VLMs. Please open an issue on GitHub if you have any feedback or feature requests.

1.26.1 Offline Batched Inference

To initialize a VLM, the aforementioned arguments must be passed to the `LLM` class for instantiating the engine.

```
llm = LLM(model="llava-hf/llava-1.5-7b-hf")
```

Important: We have removed all vision language related CLI args in the `0.5.1` release. **This is a breaking change**, so please update your code to follow the above snippet. Specifically, `image_feature_size` is no longer required to be specified as we now calculate that internally for each model.

To pass an image to the model, note the following in *`vllm.inputs.PromptInputs`*:

- `prompt`: The prompt should follow the format that is documented on HuggingFace.
- `multi_modal_data`: This is a dictionary that follows the schema defined in `vllm.multimodal.MultiModalDataDict`.

```
# Refer to the HuggingFace repo for the correct format to use
prompt = "USER: <image>\nWhat is the content of this image?\nASSISTANT:"

# Load the image using PIL.Image
image = PIL.Image.open(...)

# Single prompt inference
outputs = llm.generate({
    "prompt": prompt,
    "multi_modal_data": {"image": image},
})
```

(continues on next page)

(continued from previous page)

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

# Batch inference
image_1 = PIL.Image.open(...)
image_2 = PIL.Image.open(...)
outputs = llm.generate(
    [
        {
            "prompt": "USER: <image>\nWhat is the content of this image?\nASSISTANT:",
            "multi_modal_data": {"image": image_1},
        },
        {
            "prompt": "USER: <image>\nWhat's the color of this image?\nASSISTANT:",
            "multi_modal_data": {"image": image_2},
        }
    ]
)

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

A code example can be found in examples/llava_example.py.

1.26.2 Online OpenAI Vision API Compatible Inference

You can serve vision language models with vLLM's HTTP server that is compatible with OpenAI Vision API.

Note: Currently, vLLM supports only **single** `image_url` input per `messages`. Support for multi-image inputs will be added in the future.

Below is an example on how to launch the same `llava-hf/llava-1.5-7b-hf` with vLLM API server.

Important: Since OpenAI Vision API is based on Chat API, a chat template is **required** to launch the API server if the model's tokenizer does not come with one. In this example, we use the HuggingFace Llava chat template that you can find in the example folder here.

```
vllm serve llava-hf/llava-1.5-7b-hf --chat-template template_llava.jinja
```

Important: We have removed all vision language related CLI args in the `0.5.1` release. **This is a breaking change**, so please update your code to follow the above snippet. Specifically, `image_feature_size` is no longer required to be specified as we now calculate that internally for each model.

To consume the server, you can use the OpenAI client like in the example below:

```
from openai import OpenAI
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="llava-hf/llava-1.5-7b-hf",
    messages=[{
        "role": "user",
        "content": [
            # NOTE: The prompt formatting with the image token `<image>` is not needed
            # since the prompt will be processed automatically by the API server.
            {"type": "text", "text": "What's in this image?"},
            {
                "type": "image_url",
                "image_url": {
                    "url": "https://upload.wikimedia.org/wikipedia/commons/thumb/d/dd/
↪Gfp-wisconsin-madison-the-nature-boardwalk.jpg/2560px-Gfp-wisconsin-madison-the-nature-
↪boardwalk.jpg",
                },
            },
        ],
    }],
)
print("Chat response:", chat_response)
```

A full code example can be found in examples/openai_vision_api_client.py.

Note: By default, the timeout for fetching images through http url is 5 seconds. You can override this by setting the environment variable:

```
export VLLM_IMAGE_FETCH_TIMEOUT=<timeout>
```

Note: There is no need to format the prompt in the API request since it will be handled by the server.

1.27 Speculative decoding in vLLM

Warning: Please note that speculative decoding in vLLM is not yet optimized and does not usually yield inter-token latency reductions for all prompt datasets or sampling parameters. The work to optimize it is ongoing and can be followed in this issue.

This document shows how to use Speculative Decoding with vLLM. Speculative decoding is a technique which improves inter-token latency in memory-bound LLM inference.

1.27.1 Speculating with a draft model

The following code configures vLLM to use speculative decoding with a draft model, speculating 5 tokens at a time.

```
from vllm import LLM, SamplingParams

prompts = [
    "The future of AI is",
]
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)

llm = LLM(
    model="facebook/opt-6.7b",
    tensor_parallel_size=1,
    speculative_model="facebook/opt-125m",
    num_speculative_tokens=5,
    use_v2_block_manager=True,
)
outputs = llm.generate(prompts, sampling_params)

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

1.27.2 Speculating by matching n-grams in the prompt

The following code configures vLLM to use speculative decoding where proposals are generated by matching n-grams in the prompt. For more information read this thread.

```
from vllm import LLM, SamplingParams

prompts = [
    "The future of AI is",
]
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)

llm = LLM(
    model="facebook/opt-6.7b",
    tensor_parallel_size=1,
    speculative_model="[ngram]",
    num_speculative_tokens=5,
    ngram_prompt_lookup_max=4,
    use_v2_block_manager=True,
)
outputs = llm.generate(prompts, sampling_params)

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

1.27.3 Resources for vLLM contributors

- A Hacker's Guide to Speculative Decoding in vLLM
- What is Lookahead Scheduling in vLLM?
- Information on batch expansion
- Dynamic speculative decoding

1.28 Performance and Tuning

1.28.1 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.

1.28.2 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!

1.29 Supported Hardware for Quantization Kernels

The table below shows the compatibility of various quantization implementations with different hardware platforms in vLLM:

Implementation	Volta	Turing	Ampere	Ada	Hopper	AMD GPU	Intel GPU	x86 CPU	AWS Inferentia	Google TPU
AQLM										
AWQ										
DeepSpeedFP										
FP8										
Marlin										
GPTQ										
SqueezeLLM										
bitsandbytes										

1.29.1 Notes:

- Volta refers to SM 7.0, Turing to SM 7.5, Ampere to SM 8.0/8.6, Ada to SM 8.9, and Hopper to SM 9.0.
- “” indicates that the quantization method is supported on the specified hardware.
- “” indicates that the quantization method is not supported on the specified hardware.

Please note that this compatibility chart may be subject to change as vLLM continues to evolve and expand its support for different hardware platforms and quantization methods.

For the most up-to-date information on hardware support and quantization methods, please check the quantization directory or consult with the vLLM development team.

1.30 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
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:

```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="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}")
```

1.31 FP8

vLLM supports FP8 (8-bit floating point) weight and activation quantization using hardware acceleration on GPUs such as Nvidia H100 and AMD MI300x. Currently, only Hopper and Ada Lovelace GPUs are officially supported for W8A8. Ampere GPUs are supported for W8A16 (weight-only FP8) utilizing Marlin kernels. Quantization of models with FP8 allows for a 2x reduction in model memory requirements and up to a 1.6x improvement in throughput with minimal impact on accuracy.

Please visit the HF collection of quantized FP8 checkpoints of popular LLMs ready to use with vLLM.

The FP8 types typically supported in hardware have two distinct representations, each useful in different scenarios:

- **E4M3**: Consists of 1 sign bit, 4 exponent bits, and 3 bits of mantissa. It can store values up to +/-448 and `nan`.
- **E5M2**: Consists of 1 sign bit, 5 exponent bits, and 2 bits of mantissa. It can store values up to +/-57344, +/- `inf`, and `nan`. The tradeoff for the increased dynamic range is lower precision of the stored values.

1.31.1 Quick Start with Online Dynamic Quantization

Dynamic quantization of an original precision BF16/FP16 model to FP8 can be achieved with vLLM without any calibration data required. You can enable the feature by specifying `--quantization="fp8"` in the command line or setting `quantization="fp8"` in the LLM constructor.

In this mode, all Linear modules (except for the final `lm_head`) have their weights quantized down to FP8_E4M3 precision with a per-tensor scale. Activations have their minimum and maximum values calculated during each forward pass to provide a dynamic per-tensor scale for high accuracy. As a result, latency improvements are limited in this mode.

```python
from vllm import LLM
model = LLM("facebook/opt-125m", quantization="fp8")
```

(continues on next page)

(continued from previous page)

```
# INFO 06-10 17:55:42 model_runner.py:157] Loading model weights took 0.1550 GB
result = model.generate("Hello, my name is")
```

Warning: Currently, we load the model at original precision before quantizing down to 8-bits, so you need enough memory to load the whole model.

1.31.2 Offline Quantization

For offline quantization to FP8, please install the AutoFP8 library.

```
git clone https://github.com/neuralmagic/AutoFP8.git
pip install -e AutoFP8
```

This package introduces the `AutoFP8ForCausalLM` and `BaseQuantizeConfig` objects for managing how your model will be compressed.

1.31.3 Offline Quantization with Dynamic Activation Scaling Factors

You can use AutoFP8 to produce checkpoints with their weights quantized to FP8 ahead of time and let vLLM handle calculating dynamic scales for the activations at runtime for maximum accuracy. You can enable this with the `activation_scheme="dynamic"` argument.

Warning: Please note that although this mode doesn't give you better performance, it reduces memory footprint compared to online quantization.

```
from auto_fp8 import AutoFP8ForCausalLM, BaseQuantizeConfig

pretrained_model_dir = "meta-llama/Meta-Llama-3-8B-Instruct"
quantized_model_dir = "Meta-Llama-3-8B-Instruct-FP8-Dynamic"

# Define quantization config with static activation scales
quantize_config = BaseQuantizeConfig(quant_method="fp8", activation_scheme="dynamic")
# For dynamic activation scales, there is no need for calbration examples
examples = []

# Load the model, quantize, and save checkpoint
model = AutoFP8ForCausalLM.from_pretrained(pretrained_model_dir, quantize_config)
model.quantize(examples)
model.save_quantized(quantized_model_dir)
```

In the output of the above script, you should be able to see the quantized Linear modules (FP8DynamicLinear) replaced in the model definition. Note that the `lm_head` Linear module at the end is currently skipped by default.

```
LlamaForCausalLM(
  (model): LlamaModel(
    (embed_tokens): Embedding(128256, 4096)
    (layers): ModuleList(
```

(continues on next page)

(continued from previous page)

```
      (0-31): 32 x LlamaDecoderLayer(
        (self_attn): LlamaSdpaAttention(
          (q_proj): FP8DynamicLinear()
          (k_proj): FP8DynamicLinear()
          (v_proj): FP8DynamicLinear()
          (o_proj): FP8DynamicLinear()
          (rotary_emb): LlamaRotaryEmbedding()
        )
        (mlp): LlamaMLP(
          (gate_proj): FP8DynamicLinear()
          (up_proj): FP8DynamicLinear()
          (down_proj): FP8DynamicLinear()
          (act_fn): SiLU()
        )
        (input_layernorm): LlamaRMSNorm()
        (post_attention_layernorm): LlamaRMSNorm()
      )
    )
    (norm): LlamaRMSNorm()
  )
  (lm_head): Linear(in_features=4096, out_features=128256, bias=False)
)
Saving the model to Meta-Llama-3-8B-Instruct-FP8-Dynamic
```

Your model checkpoint with quantized weights should be available at `Meta-Llama-3-8B-Instruct-FP8/`. We can see that the weights are smaller than the original BF16 precision.

```
ls -lh Meta-Llama-3-8B-Instruct-FP8-Dynamic/
total 8.5G
-rw-rw-r-- 1 user user  869 Jun  7 14:43 config.json
-rw-rw-r-- 1 user user  194 Jun  7 14:43 generation_config.json
-rw-rw-r-- 1 user user 4.7G Jun  7 14:43 model-00001-of-00002.safetensors
-rw-rw-r-- 1 user user 3.9G Jun  7 14:43 model-00002-of-00002.safetensors
-rw-rw-r-- 1 user user  43K Jun  7 14:43 model.safetensors.index.json
-rw-rw-r-- 1 user user  296 Jun  7 14:43 special_tokens_map.json
-rw-rw-r-- 1 user user  50K Jun  7 14:43 tokenizer_config.json
-rw-rw-r-- 1 user user 8.7M Jun  7 14:43 tokenizer.json
```

Finally, you can load the quantized model checkpoint directly in vLLM.

```
from vllm import LLM
model = LLM(model="Meta-Llama-3-8B-Instruct-FP8-Dynamic/")
# INFO 06-10 21:15:41 model_runner.py:159] Loading model weights took 8.4596 GB
result = model.generate("Hello, my name is")
```

1.31.4 Offline Quantization with Static Activation Scaling Factors

For the best inference performance, you can use AutoFP8 with calibration data to produce per-tensor static scales for both the weights and activations by enabling the `activation_scheme="static"` argument.

```
from datasets import load_dataset
from transformers import AutoTokenizer
from auto_fp8 import AutoFP8ForCausalLM, BaseQuantizeConfig

pretrained_model_dir = "meta-llama/Meta-Llama-3-8B-Instruct"
quantized_model_dir = "Meta-Llama-3-8B-Instruct-FP8"

tokenizer = AutoTokenizer.from_pretrained(pretrained_model_dir, use_fast=True)
tokenizer.pad_token = tokenizer.eos_token

# Load and tokenize 512 dataset samples for calibration of activation scales
ds = load_dataset("mgoin/ultrachat_2k", split="train_sft").select(range(512))
examples = [tokenizer.apply_chat_template(batch["messages"], tokenize=False) for batch␣
↪in ds]
examples = tokenizer(examples, padding=True, truncation=True, return_tensors="pt").to(
↪"cuda")

# Define quantization config with static activation scales
quantize_config = BaseQuantizeConfig(quant_method="fp8", activation_scheme="static")

# Load the model, quantize, and save checkpoint
model = AutoFP8ForCausalLM.from_pretrained(pretrained_model_dir, quantize_config)
model.quantize(examples)
model.save_quantized(quantized_model_dir)
```

Your model checkpoint with quantized weights and activations should be available at `Meta-Llama-3-8B-Instruct-FP8/`. Finally, you can load the quantized model checkpoint directly in vLLM.

```
from vllm import LLM
model = LLM(model="Meta-Llama-3-8B-Instruct-FP8/")
# INFO 06-10 21:15:41 model_runner.py:159] Loading model weights took 8.4596 GB
result = model.generate("Hello, my name is")
```

1.31.5 FP8 checkpoint structure explanation

Here we detail the structure for the FP8 checkpoints.

The following is necessary to be present in the model's `config.json`:

```
"quantization_config": {
    "quant_method": "fp8",
    "activation_scheme": "static" or "dynamic"
}
```

Each quantized layer in the state_dict will have these tensors:

- If the config has `"activation_scheme": "static"`:

```
model.layers.0.mlp.down_proj.weight                < F8_E4M3
model.layers.0.mlp.down_proj.input_scale           < F32
model.layers.0.mlp.down_proj.weight_scale          < F32
```

- If the config has `"activation_scheme": "dynamic"`:

```
model.layers.0.mlp.down_proj.weight                < F8_E4M3
model.layers.0.mlp.down_proj.weight_scale          < F32
```

Additionally, there can be FP8 kv-cache scaling factors contained within quantized checkpoints specified through the `.kv_scale` parameter present on the Attention Module, such as:

```
model.layers.0.self_attn.kv_scale                  < F32
```

1.32 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:

```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", 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.

1.33 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 ($\pm$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)

# 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.

1.34 Introduction

1.34.1 What is Automatic Prefix Caching

Automatic Prefix Caching (APC in short) caches the KV cache of existing queries, so that a new query can directly reuse the KV cache if it shares the same prefix with one of the existing queries, allowing the new query to skip the computation of the shared part.

Note: Technical details on how vLLM implements APC are in the next page.

1.34.2 Enabling APC in vLLM

Set `enable_prefix_caching=True` in vLLM engine to enable APC. Here is an example:

```
import time
from vllm import LLM, SamplingParams


# A prompt containing a large markdown table. The table is randomly generated by GPT-4.
LONG_PROMPT = "You are a helpful assistant in recognizes the content of tables in␣
↪markdown format. Here is a table as follows.\n# Table\n" + """
| ID  | Name          | Age | Occupation    | Country       | Email                |␣
↪Phone Number   | Address                       |
|-----|---------------|-----|---------------|---------------|----------------------|---
↪-------------|-------------------------------|
| 1   | John Doe      | 29  | Engineer      | USA           | john.doe@example.com   |␣
↪555-1234       | 123 Elm St, Springfield, IL   |
| 2   | Jane Smith    | 34  | Doctor        | Canada        | jane.smith@example.com |␣
↪555-5678       | 456 Oak St, Toronto, ON       |
| 3   | Alice Johnson | 27  | Teacher       | UK            | alice.j@example.com    |␣
↪555-8765       | 789 Pine St, London, UK       |
| 4   | Bob Brown     | 45  | Artist        | Australia     | bob.b@example.com      |␣
↪555-4321       | 321 Maple St, Sydney, NSW     |
| 5   | Carol White   | 31  | Scientist     | New Zealand   | carol.w@example.com    |␣
↪555-6789       | 654 Birch St, Wellington, NZ  |
| 6   | Dave Green    | 28  | Lawyer        | Ireland       | dave.g@example.com     |␣
↪555-3456       | 987 Cedar St, Dublin, IE      |
| 7   | Emma Black    | 40  | Musician      | USA           | emma.b@example.com     |␣
↪555-1111       | 246 Ash St, New York, NY      |
| 8   | Frank Blue    | 37  | Chef          | Canada        | frank.b@example.com    |␣
↪555-2222       | 135 Spruce St, Vancouver, BC  |
| 9   | Grace Yellow  | 50  | Engineer      | UK            | grace.y@example.com    |␣
↪555-3333       | 864 Fir St, Manchester, UK    |
| 10  | Henry Violet  | 32  | Artist        | Australia     | henry.v@example.com    |␣
↪555-4444       | 753 Willow St, Melbourne, VIC|
| 11  | Irene Orange  | 26  | Scientist     | New Zealand   | irene.o@example.com    |␣
↪555-5555       | 912 Poplar St, Auckland, NZ   |
| 12  | Jack Indigo   | 38  | Teacher       | Ireland       | jack.i@example.com     |␣
↪555-6666       | 159 Elm St, Cork, IE          |
| 13  | Karen Red     | 41  | Lawyer        | USA           | karen.r@example.com    |␣
```

(continues on next page)

(continued from previous page)

```
↪555-7777       | 357 Cedar St, Boston, MA     |
| 14  | Leo Brown     | 30  | Chef          | Canada        | leo.b@example.com      |␣
↪555-8888       | 246 Oak St, Calgary, AB      |
| 15  | Mia Green     | 33  | Musician      | UK            | mia.g@example.com      |␣
↪555-9999       | 975 Pine St, Edinburgh, UK   |
| 16  | Noah Yellow   | 29  | Doctor        | Australia     | noah.y@example.com     |␣
↪555-0000       | 864 Birch St, Brisbane, QLD  |
| 17  | Olivia Blue   | 35  | Engineer      | New Zealand   | olivia.b@example.com   |␣
↪555-1212       | 753 Maple St, Hamilton, NZ   |
| 18  | Peter Black   | 42  | Artist        | Ireland       | peter.b@example.com    |␣
↪555-3434       | 912 Fir St, Limerick, IE     |
| 19  | Quinn White   | 28  | Scientist     | USA           | quinn.w@example.com    |␣
↪555-5656       | 159 Willow St, Seattle, WA   |
| 20  | Rachel Red    | 31  | Teacher       | Canada        | rachel.r@example.com   |␣
↪555-7878       | 357 Poplar St, Ottawa, ON    |
| 21  | Steve Green   | 44  | Lawyer        | UK            | steve.g@example.com    |␣
↪555-9090       | 753 Elm St, Birmingham, UK   |
| 22  | Tina Blue     | 36  | Musician      | Australia     | tina.b@example.com     |␣
↪555-1213       | 864 Cedar St, Perth, WA      |
| 23  | Umar Black    | 39  | Chef          | New Zealand   | umar.b@example.com     |␣
↪555-3435       | 975 Spruce St, Christchurch, NZ|
| 24  | Victor Yellow | 43  | Engineer      | Ireland       | victor.y@example.com   |␣
↪555-5657       | 246 Willow St, Galway, IE    |
| 25  | Wendy Orange  | 27  | Artist        | USA           | wendy.o@example.com    |␣
↪555-7879       | 135 Elm St, Denver, CO       |
| 26  | Xavier Green  | 34  | Scientist     | Canada        | xavier.g@example.com   |␣
↪555-9091       | 357 Oak St, Montreal, QC     |
| 27  | Yara Red      | 41  | Teacher       | UK            | yara.r@example.com     |␣
↪555-1214       | 975 Pine St, Leeds, UK       |
| 28  | Zack Blue     | 30  | Lawyer        | Australia     | zack.b@example.com     |␣
↪555-3436       | 135 Birch St, Adelaide, SA   |
| 29  | Amy White     | 33  | Musician      | New Zealand   | amy.w@example.com      |␣
↪555-5658       | 159 Maple St, Wellington, NZ |
| 30  | Ben Black     | 38  | Chef          | Ireland       | ben.b@example.com      |␣
↪555-7870       | 246 Fir St, Waterford, IE    |
"""


def get_generation_time(llm, sampling_params, prompts):
    # time the generation
    start_time = time.time()
    output = llm.generate(prompts, sampling_params=sampling_params)
    end_time = time.time()
    # print the output and generation time
    print(f"Output: {output[0].outputs[0].text}")
    print(f"Generation time: {end_time - start_time} seconds.")


# set enable_prefix_caching=True to enable APC
llm = LLM(
    model='lmsys/longchat-13b-16k',
    enable_prefix_caching=True
```

(continues on next page)

(continued from previous page)

```
)

sampling_params = SamplingParams(temperature=0, max_tokens=100)

# Querying the age of John Doe
get_generation_time(
    llm,
    sampling_params,
    LONG_PROMPT + "Question: what is the age of John Doe? Your answer: The age of John␣
↪Doe is ",
)

# Querying the age of Zack Blue
# This query will be faster since vllm avoids computing the KV cache of LONG_PROMPT␣
↪again.
get_generation_time(
    llm,
    sampling_params,
    LONG_PROMPT + "Question: what is the age of Zack Blue? Your answer: The age of Zack␣
↪Blue is ",
)
```

1.34.3 Example workloads

We describe two example workloads, where APC can provide huge performance benefit:

- Long document query, where the user repeatedly queries the same long document (e.g. software manual or annual report) with different queries. In this case, instead of processing the long document again and again, APC allows vLLM to process this long document *only once*, and all future requests can avoid recomputing this long document by reusing its KV cache. This allows vLLM to serve future requests with much higher throughput and much lower latency.
- Multi-round conversation, where the user may chat with the application multiple times in the same chatting session. In this case, instead of processing the whole chatting history again and again, APC allows vLLM to reuse the processing results of the chat history across all future rounds of conversation, allowing vLLM to serve future requests with much higher throughput and much lower latency.

1.34.4 Limits

APC in general does not reduce the performance of vLLM. With that being said, APC only reduces the time of processing the queries (the prefilling phase) and does not reduce the time of generating new tokens (the decoding phase). So APC does not bring performance gain when vLLM spends most of the time generating answers to the queries (e.g. when the length of the answer is long), or new queries do not share the same prefix with any of existing queries (so that the computation cannot be reused).

1.35 Implementation

The core idea of PagedAttention is to partition the KV cache of each request into KV Blocks. Each block contains the attention keys and values for a fixed number of tokens. The PagedAttention algorithm allows these blocks to be stored in non-contiguous physical memory so that we can eliminate memory fragmentation by allocating the memory on demand.

To automatically cache the KV cache, we utilize the following key observation: Each KV block can be uniquely identified by the tokens within the block and the tokens in the prefix before the block.

```
                    Block 1                  Block 2                  Block 3
         [A gentle breeze stirred] [the leaves as children] [laughed in the distance]
Block 1: |<--- block tokens ---->|
Block 2: |<------- prefix ------>| |<--- block tokens --->|
Block 3: |<------------------ prefix -------------------->| |<--- block tokens ---->|
```

In the example above, the KV cache in the first block can be uniquely identified with the tokens "A gentle breeze stirred". The third block can be uniquely identified with the tokens in the block "laughed in the distance", along with the prefix tokens "A gentle breeze stirred the leaves as children". Therefore, we can build the following one-to-one mapping:

```
hash(prefix tokens + block tokens) <--> KV Block
```

With this mapping, we can add another indirection in vLLM's KV cache management. Previously, each sequence in vLLM maintained a mapping from their logical KV blocks to physical blocks. To achieve automatic caching of KV blocks, we map the logical KV blocks to their hash value and maintain a global hash table of all the physical blocks. In this way, all the KV blocks sharing the same hash value (e.g., shared prefix blocks across two requests) can be mapped to the same physical block and share the memory space.

This design achieves automatic prefix caching without the need of maintaining a tree structure among the KV blocks. More specifically, all of the blocks are independent of each other and can be allocated and freed by itself, which enables us to manages the KV cache as ordinary caches in operating system.

1.36 Generalized Caching Policy

Keeping all the KV blocks in a hash table enables vLLM to cache KV blocks from earlier requests to save memory and accelerate the computation of future requests. For example, if a new request shares the system prompt with the previous request, the KV cache of the shared prompt can directly be used for the new request without recomputation. However, the total KV cache space is limited and we have to decide which KV blocks to keep or evict when the cache is full.

Managing KV cache with a hash table allows us to implement flexible caching policies. As an example, in current vLLM, we implement the following eviction policy:

- When there are no free blocks left, we will evict a KV block with reference count (i.e., number of current requests using the block) equals 0.
- If there are multiple blocks with reference count equals to 0, we prioritize to evict the least recently used block (LRU).
- If there are multiple blocks whose last access time are the same, we prioritize the eviction of the block that is at the end of the longest prefix (i.e., has the maximum number of blocks before it).

Note that this eviction policy effectively implements the exact policy as in RadixAttention when applied to models with full attention, which prioritizes to evict reference count zero and least recent used leaf nodes in the prefix tree.

However, the hash-based KV cache management gives us the flexibility to handle more complicated serving scenarios and implement more complicated eviction policies beyond the policy above:

- Multi-LoRA serving. When serving requests for multiple LoRA adapters, we can simply let the hash of each KV block to also include the LoRA ID the request is querying for to enable caching for all adapters. In this way, we can jointly manage the KV blocks for different adapters, which simplifies the system implementation and improves the global cache hit rate and efficiency.
- Multi-modal models. When the user input includes more than just discrete tokens, we can use different hashing methods to handle the caching of inputs of different modalities. For example, perceptual hashing for images to cache similar input images.

1.37 Sampling Parameters

1.38 Offline Inference

1.38.1 LLM Class

1.38.2 LLM Inputs

vllm.inputs.**PromptInputs**

The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

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[MultiModalDataDict]

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[MultiModalDataDict]

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

1.39 vLLM Engine

1.39.1 LLMEngine

1.39.2 AsyncLLMEngine

1.40 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.

1.40.1 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,
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.

1.40.2 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.

1.40.3 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;
```

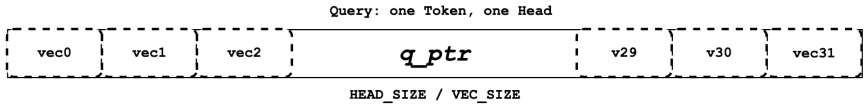


Fig. 1: Query data of one token at one head

- 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.

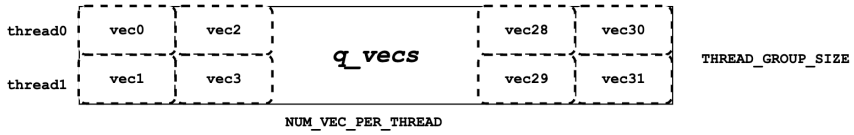


Fig. 2: `q_vecs` for one thread group

```
__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.

1.40.4 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.
- 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

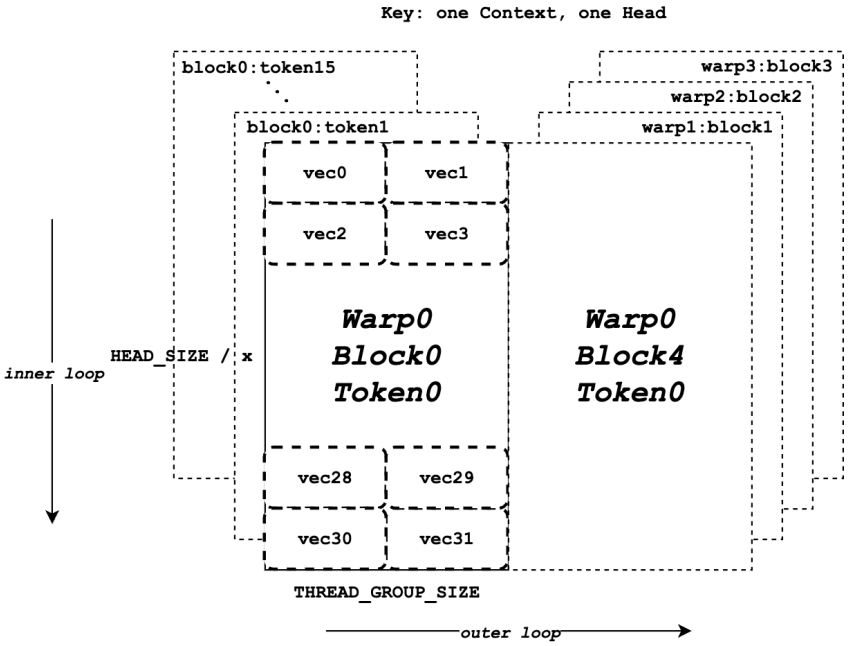


Fig. 3: Key data of all context tokens at one head

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.

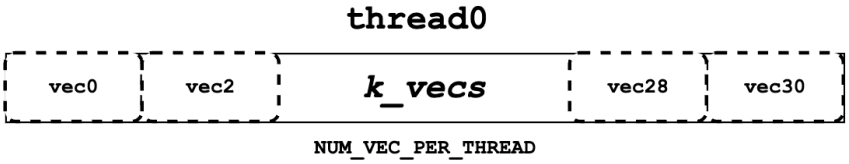


Fig. 4: `k_vecs` for one thread

```
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 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.

1.40.5 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.

1.40.6 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.

$$m(x) := \max_i \quad x_i$$

$$f(x) := \left[\; e^{x_1 - m(x)} \quad \dots \quad e^{x_B - m(x)} \; \right]$$

$$\ell(x) := \sum_i f(x)_i$$

$$\text{softmax}(x) := \frac{f(x)}{\ell(x)}$$

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 qks 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.

1.40.7 Value

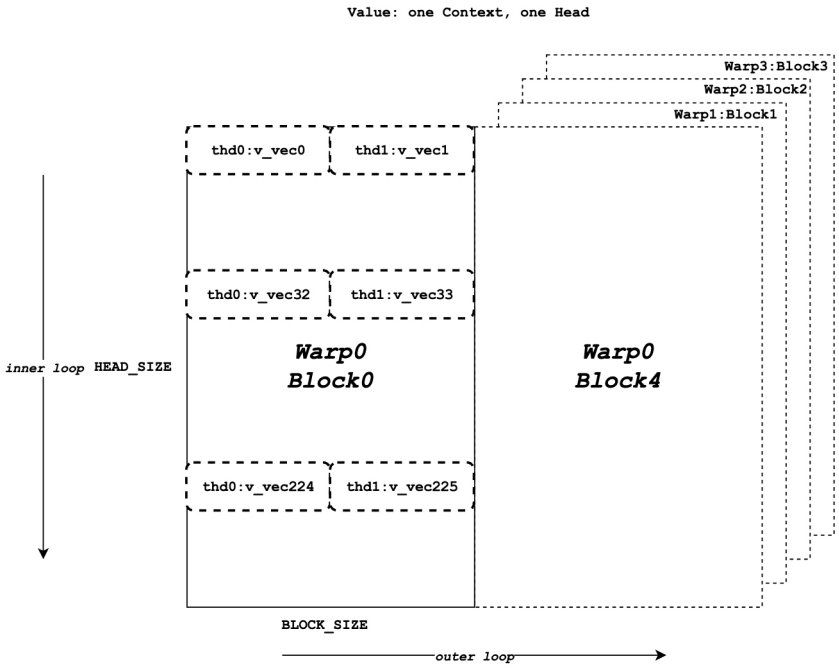


Fig. 5: Value data of all context tokens at one head

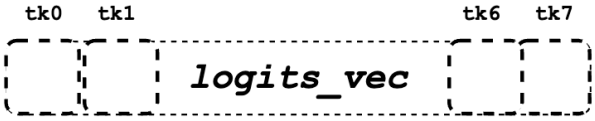


Fig. 6: `logits_vec` for one thread

- 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.
```

(continues on next page)

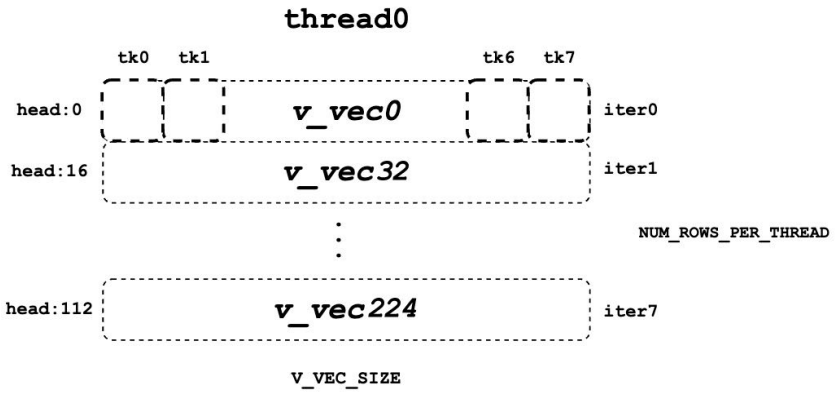


Fig. 7: List of v_vec for one thread

(continued from previous page)

```
    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.

1.40.8 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.
}
```

1.40.9 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`.

1.41 Input Processing

Each model can override parts of vLLM's *input processing pipeline* via *INPUT_REGISTRY* and MULTIMODAL_REGISTRY.

Currently, this mechanism is only utilized in *multi-modal* models for preprocessing multi-modal input data in addition to input prompt, but it can be extended to text-only language models when needed.

1.41.1 Guides

Input Processing Pipeline

1. Input data is passed to LLMEngine (or AsyncLLMEngine).
2. Tokenize the data if necessary.
3. Process the inputs using *INPUT_REGISTRY.process_input*.
    - For example, add placeholder tokens to reserve KV cache for multi-modal embeddings.
4. Send the processed inputs to ExecutorBase.
5. Distribute the inputs via WorkerBase to ModelRunnerBase.
6. If the data contains multi-modal data, convert it into keyword arguments using MULTIMODAL_REGISTRY.map_input.
    - For example, convert a PIL.Image.Image input to its pixel values for a vision language model.

1.41.2 Module Contents

LLM Engine Inputs

class vllm.inputs.**LLMInputs**(**args*, ***kwargs*)

Bases: dict

The inputs in LLMEngine before they are passed to the model executor.

multi_modal_data: typing_extensions.NotRequired[MultiModalDataDict | None]

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

prompt: typing_extensions.NotRequired[str | None]

The original prompt text corresponding to the token IDs, if available.

prompt_token_ids: List[int]

The token IDs of the prompt.

Registry

vllm.inputs.**INPUT_REGISTRY = <vllm.inputs.registry.InputRegistry object>**

The global `InputRegistry` which is used by `LLMEngine` to dispatch data processing according to the target model.

See also:

Input Processing Pipeline

vllm.inputs.registry.**DummyDataFactory**

Create dummy data to be inputted into the model.

Note: *InputProcessor* is not applied to the dummy data.

---

alias of `Callable[[InputContext, int], Tuple[SequenceData, Optional[MultiModalDataDict]]]`

class vllm.inputs.registry.**InputContext**(*model_config: ModelConfig*)

Contains information about the model which may be used to modify the inputs.

get_hf_config(*hf_config_type: Type[C]*) → C

Get the HuggingFace configuration (`transformers.PretrainedConfig`) of the model, additionally checking its type.

Raises

TypeError – If the model is not of the specified type.

get_multimodal_config() → MultiModalConfig

Get the multimodal configuration of the model.

Raises

ValueError – If the model is not multimodal.

model_config: ModelConfig

The configuration of the model.

vllm.inputs.registry.**InputProcessor**

Preprocess the inputs to the model.

alias of `Callable[[InputContext, LLMInputs], LLMInputs]`

class vllm.inputs.registry.**InputRegistry**

A registry to dispatch data processing according to the target model.

create_input_processor(*model_config: ModelConfig*)

Create an input processor (see *process_input()*) for a specific model.

dummy_data_for_profiling(*model_config: ModelConfig*, *seq_len: int*)

Create dummy data for profiling the memory usage of a model.

The model is identified by `model_config`.

See also:

Enabling Multimodal Inputs

process_input(*model_config: ModelConfig*, *inputs:* LLMInputs) → *LLMInputs*

Apply an input processor to an instance of model inputs.

The model is identified by `model_config`.

See also:

Input Processing Pipeline

register_dummy_data(*factory: Callable[[*InputContext*, int], Tuple[SequenceData, MultiModalDataDict | None]]*)

Register a dummy data factory to a model class.

During memory profiling, the provided function is invoked to create dummy data to be inputted into the model. The resulting memory usage should be an upper bound of what the model would use at inference time.

register_input_processor(*processor: Callable[[*InputContext, LLMInputs*], LLMInputs*])

Register an input processor to a model class.

The provided function is invoked on each input to the model. This happens before `map_input()`.

See also:

Input Processing Pipeline

1.42 Multi-Modality

vLLM provides experimental support for multi-modal models through the `vllm.multimodal` package.

Multi-modal inputs can be passed alongside text and token prompts to *supported models* via the `multi_modal_data` field in *`vllm.inputs.PromptInputs`*.

Currently, vLLM only has built-in support for image data. You can extend vLLM to process additional modalities by following *this guide*.

Looking to add your own multi-modal model? Please follow the instructions listed *here*.

1.42.1 Guides

Adding a Multimodal Plugin

This document teaches you how to add a new modality to vLLM.

Each modality in vLLM is represented by a `MultiModalPlugin` and registered to `MULTIMODAL_REGISTRY`. For vLLM to recognize a new modality type, you have to create a new plugin and then pass it to `register_plugin()`.

The remainder of this document details how to define custom `MultiModalPlugin` s.

Note: This article is a work in progress.

1.42.2 Module Contents

Registry

Base Classes

Image Classes

1.43 Dockerfile

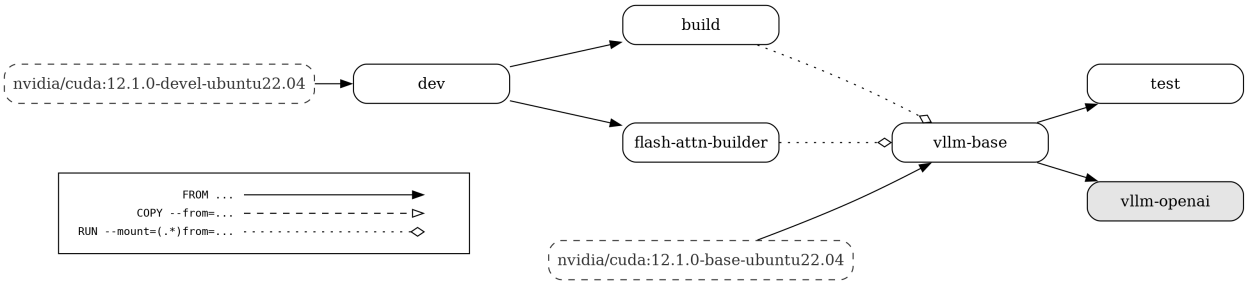
See here for the main Dockerfile to construct the image for running an OpenAI compatible server with vLLM. More information about deploying with Docker can be found here.

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 \
```

(continues on next page)

(continued from previous page)

```
  --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*.)

1.44 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 fourth vLLM meetup, with Cloudflare and BentoML, June 11th 2024. [Slides]
- 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.

1.45 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
- Google Cloud
- Lambda Lab
- NVIDIA
- Replicate
- Roblox
- RunPod
- Sequoia Capital
- Trainy

- UC Berkeley
- UC San Diego
- ZhenFund

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