Documenting the journey

Tags

Figuring out the best way to serve and deploy LLMs:

Presentation Outline

▼ Which technologies I explored and why, and all the struggles with each stack

1. Huggingface `Transformers` library (directly served meta llama-3-8gb, using ai-stack GPUs)

- Ran on ai-stack GPU
- Did not use any front-end or interface, built a simple command line application on ai-stack, helped to understand `langchain` for RAG pipeline logic
- Used vanilla `transformers` library to serve LLM, did not use TGI server which might allow better performance

2. `llama-cpp` with `mistral` and `phi3` - `langchain` for RAG pipeline logic, `chainlit` for the front-end

- Was able to run locally on CPU
- Sped up quite a bit with GPU (after installing CUDA on local windows machine)
- Followed and adapted example code from chainlit cookbook
- Discovered that the framework simply does not support parallel inference requests and crashes when hosted in a flask server that took multiple requests at once

- Hence migrated to vLLM

3. `vLLM` with custom front-end

- Run on a VM (Ubuntu 22.04 LTS with A100 GPU)
- Extremely fast inference speeds, and the amount of GPU memory can be configured (CUDA Graphs)
- The code and RAG logic was established from `llama-cpp-python` already, and `langchain` had in-built vLLM support since its so capable, so all i really needed to do was to get vLLM installed and connected with the GPUst
    - This was very very painful, trying to install Nvidia GPU drivers on a Linux VM is a headache - everything via command line, and many many restrictions on permissions.
    - Tried to install on RHEL VM first but failed, and the entire process took up about 2 weeks, until i finally got it to work on the Ubuntu VM, and thats when the product really felt like it was at the bare minimum and ready to connect to the front-end.
- `vLLM` server is hosted in the same container but on a different port, and resource consumption also seems to be reasonable (though the VM is very powerful).
- The container was able to handle 10,000 POST requests (via cURL) at once, without even a big spike in resource usage (GPU mem remained, CPU spiked only by 1%)

The end product, how it works, and how it can be improved.

- How it works → Architecture diagram, explain and stuff
- Should we start, or end with a demo?
- Evaluation: `langsmith` and `ragas`
- How to improve:
    - Improve PDF Preprocessing

- Chat History Awareness (add diagram)

Working with Azure VMs

Ensure that the `.pem` file is in the same directory where the `ssh` or `scp` commands are executed. For now it's in my `~/Desktop/Developer` directory

▼ [FOCUS] Ubuntu + A100 GPU VM

▼ Installation

To use `vLLM` to serve up models on Ubuntu 22.04, first complete these 2 main installation steps.

1. Install Nvidia drivers for your GPU: Follow this guide
2. Install Python 3.10 - can use `sudo apt ...`
3. `pip install vLLM` within a clean virtual environment. It comes with CUDA 12.1 pre-compiled, so you shouldn't have to try to install CUDA before that. BUT the Nvidia drivers are necessary

run Ubuntu + A100 GPU VM (ensure that the `.pem` key is in the current working directory+)

```
ssh -i MP-chatbot-gpu-ubuntu_key.pem azureuser@20.195.28.2(
```

▼ Moving files between VM and Local

To copy a directory `LOCAL_DIR` contained within your current working directory to the VM

```
scp -i ../MP-chatbot-gpu-ubuntu_key.pem -r -O LOCAL_DIR/ az
```

- `-i`: for authentication, include the file path of the VM's `.pem` key.

- `-r`: Recursive flag to copy the entirety of the `LOCAL_DIR`'s content.
- `-O`: Legacy mode flag, in case you get the error `scp: failed to upload directory LOCAL_DIR to .`
- `azureuser@20.195.28.20:/home/azureuser/dev/`: The VM's host ip address, followed by the path of the VM's directory that should contain the `LOCAL_DIR`

▼ Development - vLLM and Flask Server

▼ 1. Start vllm openAI compatible server, using `phi-3-mini-4k-instruct` (this vLLM server can only host one model at a time)

```
export HF_HOME=/home/azureuser/dev/chatbot-server/mode
```

```
python -m vllm.entrypoints.openai.api_server --model
```

```
python3 -m vllm.entrypoints.openai.api_server --model
```

This vllm server will handle the concurrent inference of the model. I chose the port 8080 for it.

In the `chatbot-server` directory,

- the Flask server can be started easily by running `app.py`. I chose the port 8000 for it.
- the RAG and model serving logic is contained in `rag_utils.py`

▼ 2. Start Flask server which outsources model inference requests to the vllm openAI compatible server

```
python app.py
```

▼ 3. Test the Flask server using cURL or through the browser

```
curl --location 'http://localhost:8000/inference_stream'
    "message":  "Tell me everything you know about GEWCA
}'
```

This POST request reaches the Flask server, which then triggers model inference via the already running vLLM server (on port 8080). The reason for this is to allow the Flask server to host other MP SmartFeatures, while outsourcing the mechanics of the concurrent requests to the vLLM server (The flask server is unable to handle concurrent requests for model inference on its own).

Containerization:

Prior to attempting to containerize, ensure that:

1. Nvidia GPU drivers are installed for the current system
2. NVIDIA Container Toolkit has been installed
3. Docker/Podman has been installed and configured to make use of the nvidia container toolkit, as it allows the containers to access the GPUs of the system running the current container.

Useful Code Snippets:

```
sudo docker run --rm --runtime=nvidia --gpus all ubuntu nvidi
```

Building Container Image

```
docker build -f Dockerfile.combined -t flask-vllm:combined .
```

```
podman build -f Dockerfile.combined -t flask-vllm:combined .
```

Running the Container from the built image

▼ Explanation of flags

- `-v` volume mount the vdb directory from which the pipeline should retrieve information from for RAG
- `-p` publishes 8000 and 8080 ports of the VM which can be accessed via the outside world (allow inbound request rules have also been set up on these ports)
- `--rm` remove container after it is stopped.
- `--runtime=nvidia --gpus` to ensure that the container has access to the VM's GPUs when it is run

```
docker run -v ./vdb:/app/vdb -p 8000:8000 -p 8080:8080 -it --
```

```
podman run -v /var/lib/container-data/vectorize-output/vdb:/a
```

▼ Attempting to separate the containers

```
docker run --runtime nvidia --gpus all \
-v ~/dev/chatbot-server/models:/root/.cache/huggingface \
--env "HUGGING_FACE_HUB_TOKEN=hf_vwOtyhuFoipAaoGTutfbdCRFA(
-p 8080:8080   \
--ipc=host \
vllm/vllm-openai:latest --port 8080 --model microsoft/Phi-3
```

- Note that the combined approach is simpler and doesnt seem to consume much more resources. Also the sizes of the images are very similar.
- TODO: if we must run on separate containers, then some configuration using docker network is necessary - rebuilding of containers to redirect model inference requests will also be necessary

▼ **Possible errors**

```
[rank0]: safetensors_rust.SafetensorError: Error while dese
```

An error like this means that the downloaded tensors of the model have an issue with them. If this is the case, delete the entire `/models/hub` directory, and run the script locally again (ensure that `HF_HOME=chatbot-server/models`) to redownload them. Then, the containerization should work fine.

Attempting to containerize using podman

> The important thing to note in order to containerize this LLM application using podman is that (1) you must have an NVIDIA GPU driver installed, (2) Nvidia container toolkit installed and (3) a version of podman that supports CDI (podman ≥ v4.1.0). This is so that the GPU can be accessed to run vLLM workloads.

1. Followed instructions from askUbuntu and installed podman 4.6.2

   The manual instructions at the moment use a wrong url for the repo key. The following gives you `podman version 4.3.1`:

   - Run this script as `root`:

```
#!/bin/sh

ubuntu_version='22.04'
```

```
key_url="https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/unstable/xUbuntu_${ubuntu_version}/Release.key"
sources_url="https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/unstable/xUbuntu_${ubuntu_version}"

echo "deb $sources_url/ /" | tee /etc/apt/sources.list.d/devel:kubic:libcontainers:unstable.list
curl -fsSL $key_url | gpg --dearmor | tee /etc/apt/trusted.gpg.d/devel_kubic_libcontainers_unstable.gpg > /dev/null
apt update
apt install podman
```

- Podman `v4` is also available in the Ubuntu `23.10` repos
- Don't forget `chmod +x script.sh` - this has been moved to the dockerfile to ensure that the entrypoint command can be run everytime.

2. Followed instructions from Nvidia-container-toolkit (after installing it already)
    - Following "Procedure" and then "Running a Workload with CDI with this command:

```
podman run --rm --device nvidia.com/gpu=all --security-opt
```

- Should give this output:

```
GPU 0: NVIDIA A100 80GB PCIe (UUID: GPU-7b2a4ddc-2644-c0d8
```

Notes so far:

- Docker commands have to be run with `sudo`
- `--runtime=nvidia --gpus all` is what allows the container to access the GPUs, assuming that Nvidia Container Toolkit has been installed. The process would be different with podman.

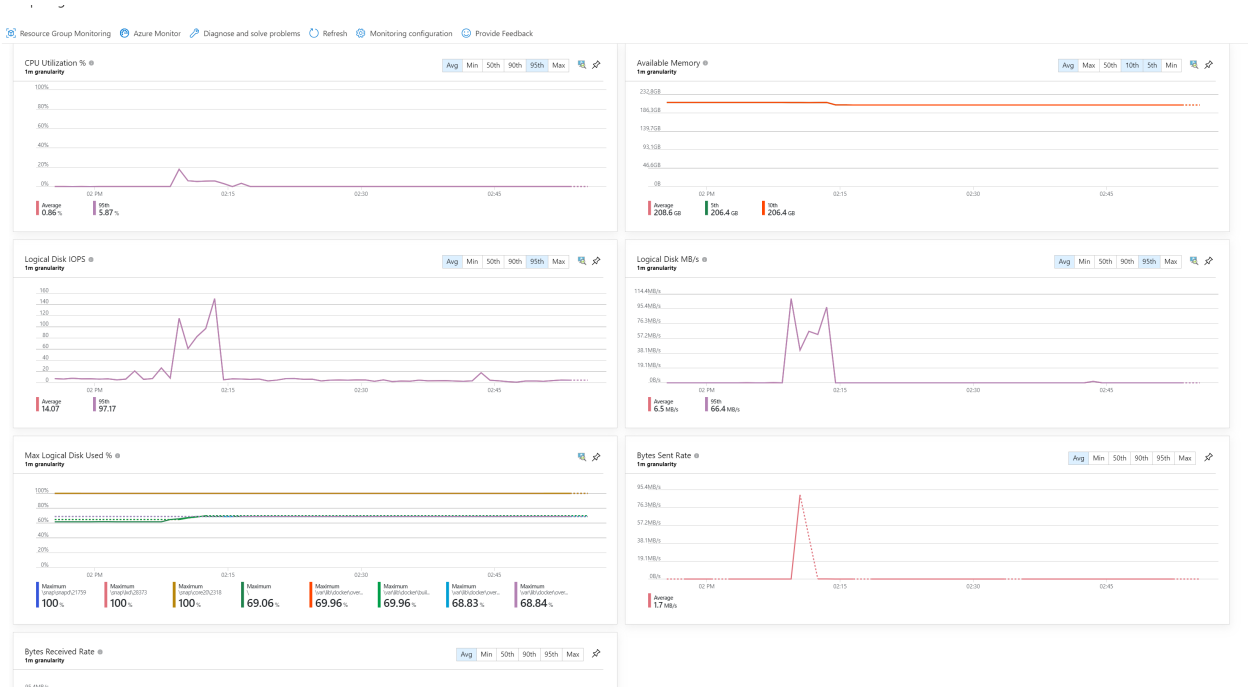
▼ *RHEL, T4 GPU VM - RHEL may not be the best linux-distribution to run vLLM, harder to install Nvidia drivers*

RHEL + T4 GPU VM

```
ssh -i MP-Chatbot-GPU_key.pem azureuser@13.88.44.107
```

Other Notes

Resource Utilization:



Seems like it's still manageable for now - seems to be using about 14GB of RAM (220-206)

▼ Running vLLM on CPU (very slow, ~30s inference on `llama3-8b`)

Load from web, meta-llama-3

```
docker run --rm --env "HF_TOKEN=hf_oUOtGDkfTGKALMdvPKQMZGBnkU
  --ipc=host \
  -p 8000:8000 \
  llm-serving:vllm-cpu \
  --model meta-llama/Meta-Llama-3-8B-Instruct
```

Trying to load with local mount:

```
docker run --env "HF_TOKEN=hf_oUOtGDkfTGKALMdvPKQMZGBnkUurwcT
```

Load container (that is able to run vllm on CPU) and mount volume to make code changse easier. working directory should contain `examples` directory.

```
docker run -it \
--rm \
--network=host \
--cpuset-cpus=<cpu-id-list, optional> \
--cpuset-mems=<memory-node, optional> \
-v ./examples://workspace/examples
vllm-cpu-env
```

▼ Misc useful code blocks:

Python Virtual Environment

```
python -m venv .venv
```

```
source .venv/Scripts/activate
```

Podman

Creating an image

```
podman build -t image_name:tag_name .
```

Running a container, mounting volume (*how to mount multiple volumes? vdb and pdf)*

```
podman run -v ./pdfs://home/user/app/pdfs image_name:tag_name
```

Sample run (mounted both the `./pdfs` and `./models` directories from local to the container (seems the local dir and container dir needs to have the same content otherwise there'll be an error):

```
podman run -v ./pdfs://app/pdfs:z -v ./vdb://app/vdb:z -v./ap
```

```
podman run -it --env "HF_TOKEN=hf_oUOtGDkfTGKALMdvPKQMZGBnkUu
```

Free up ports on linux

As the others have said, you'll have to kill all processes that are listening on that port. The easiest way to do that would be to use the `fuser(1)` command. For example, to see all of the processes listening for HTTP requests on port 80 (run as **root** or use `sudo`):

```
# fuser 80/tcp
```

If you want to kill them, then just add the `-k` option.

```
podman run --user 1001:1001 -v /var/lib/container-data/vector
```