DOCKERIZING WORKLOADS

1. OBJECTIVE

To understand Docker usage and the basic steps to dockerize any realistic application.

2. BACKGROUND

a. Need for Dockerization:

Serverless workloads are typically packaged and deployed as containers. Docker provides a lightweight and portable containerization platform that encapsulates the application and its dependencies, making it easy to package, distribute, and run serverless functions or applications.

b. Docker Components:

Docker components include the Docker Engine, Docker images, Docker containers, and Docker registries.

i. Docker Engine: The runtime that enables the creation and execution of Docker containers on a host machine.

ii. Docker Images: Lightweight, standalone, and executable packages that contain everything needed to run a piece of software, including the code, runtime, libraries, and dependencies.

iii. Docker Containers: Isolated and portable runtime instances created from Docker images, providing a consistent environment for running applications.

iv. Docker Registries: Repositories that store and distribute Docker images, allowing users to share and access container images from various sources.

3. SETUP DESCRIPTION

a. System Setup / Requirements:

Since we run our docker containers on GPUs via MPS we need added GPU support fulfilled via:

i. NVIDIA drivers and CUDA Toolkit

ii. MPS enabled GPU machine (hard requirement)

iii. Docker with NVIDIA runtime

b. Experimental Setup / Description:

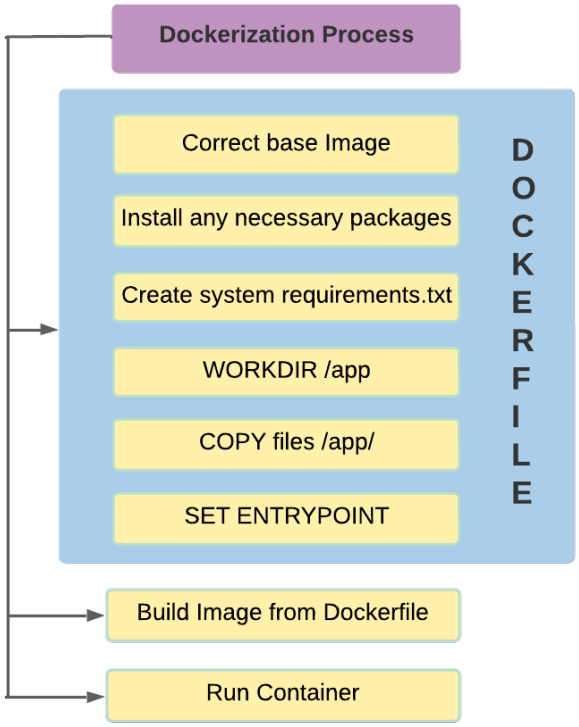
We wish to containerize four applications as follows:

i. Image Recognition: Container should accept image URLs as input and return the result

ii. Speech-to-Text: Container should accept URLs containing audio and return the transcribed text.
iii. Sentiment Analysis: Container should accept text as input and return the corresponding sentiment.
iv. Text Summarization: Container should accept text as input and return the corresponding summarised text. _

4. OVERVIEW OF THE DOCKERIZATION PROCESS

The figure below provides the basic steps to dockerize any application:



In a new dockerfile, we first select the correct base image (on the basis of the application) from DockerHub.
Next, we install any packages if necessary followed by creating a requirements.txt file (usually done by pipreqs, which we will see ahead)
Further on we create and set the workdirectory within the container to “app” and specify necessary commands to run via ENTRYPOINT or CMD, which are executed after the container launches.
We then build the container image from our dockerfile and launch the container.
This represents the basic steps which must be performed to dockerize any workload.

5. STEPS TO REPLICATE

a. Installation Dependencies:

i. NVIDIA Drivers and CUDA Toolkit Installation:

Install an appropriate version of the CUDA Toolkit (which comes in with an appropriate compatible version of NVIDIA Drivers) from NVIDIA's official website [1]. You may select the required OS, architecture, Distribution, Version and Installer Type for which you will get the download commands.

ii. MPS enabled GPU Machine:

Ensure that your machine has a MPS enabled NVIDIA GPU.

iii. Docker with NVIDIA runtime:

First, start off by installing Docker.

- Before you can install Docker Engine, you must first make sure that any conflicting packages are uninstalled.

  Uninstall the Docker Engine, CLI, containerd, and Docker Compose packages:

  ⇒ sudo apt-get purge docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin docker-ce-rootless-extras

  To delete all images, containers, and volumes:

  ⇒ sudo rm -rf /var/lib/docker

  ⇒ sudo rm -rf /var/lib/containerd

- Setup the Docker Repository

  ⇒ sudo apt-get update

  ⇒ sudo apt-get install ca-certificates curl gnupg

  ⇒ sudo install -m 0755 -d /etc/apt/keyrings

  ⇒ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

  ⇒ sudo chmod a+r /etc/apt/keyrings/docker.gpg

  ⇒ echo \
  "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
  "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

- Install Docker Engine
  ⇒ sudo apt-get update
  ⇒ sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin

- Verify Docker Installation:
  ⇒ sudo docker run hello-world

Follow the Docker installation process from Docker's official installation guide [2], in case of any queries.
Next, we will install the NVIDIA runtime into our installed Docker engine.

- Install the nvidia-container-toolkit package (and dependencies) after updating the package listing:
  ⇒ sudo apt-get update
  ⇒ sudo apt-get install -y nvidia-container-toolkit

- Configure the Docker daemon to recognize the NVIDIA Container Runtime:
  ⇒ sudo nvidia-ctk runtime configure --runtime=docker

- Restart the Docker daemon to complete the installation after setting the default runtime:
  ⇒ sudo systemctl restart docker

- At this point, a working setup can be tested by running a base CUDA container:
  ⇒ sudo docker run --rm --runtime=nvidia --gpus all nvidia/cuda:11.6.2-base-ubuntu20.04 nvidia-smi

The above steps for installation are consistent with the ones mentioned at NVIDIA's installation guide [3] for NVIDIA Docker.

b. Running Commands
   Now that we have our setup ready with the required installation dependencies, we can dive right into running the commands to achieve our objective.
   Let's discuss dockerizing our "Image Recognition Workload" (a similar approach can be followed for the rest of them)

i. Model / Variant selected: Resnet, PyTorch (there are files available for vgg16 and xception models in their respective directories)

ii. The code for the implemented ResNet Model (which is pre-trained) lies in “app.py”, found in “docker_workloads/image_rec/resnet/app.py”. For most of the work in section we will be concerned with this directory itself. The “app.py” creates a Flask application that exposes an API endpoint ("/predict") for performing image classification using a pre-trained ResNet50 model. When a POST request is received with a JSON payload containing the URL of an image, the code downloads the image, preprocesses it, passes it through the ResNet50 model, and returns the predicted class and score as a JSON response. The application runs on host 0.0.0.0 and port 5124, and the model utilizes GPU acceleration if available.

iii. The code within our “Dockerfile” defines the necessary actions for building the Docker image. The base image is based on the NVIDIA CUDA runtime with a specific version of CUDA and Ubuntu. The working directory is set to "/new_folder" in which the Flask application files are copied into. Dependencies are installed, including Python 3 and pip3. The "pipreqs" tool is installed and used to generate a requirements.txt file based on the imported app files. The requirements.txt file is then used to install the necessary dependencies. Port 5124 is exposed for the Flask application, and the container is instructed to run the Flask app using the specified command "python3 new_folder/app.py" when it starts.

iv. Now that we’ve understood the basic code concerned, we can finally look into the commands for implementing our objective:

- Start the MPS server via root privileges (sudo su -):
  ⇒ export CUDA_VISIBLE_DEVICES="0"
  ⇒ nvidia-smi -i 0 -c EXCLUSIVE_PROCESS
  ⇒ nvidia-cuda-mps-control -d

- Build container image “gpu-image-recognition-resnet”:
  Run this command in the same directory as your dockerfile.
  ⇒ sudo docker build -t gpu-image-recognition-resnet .

  If you have saved your Dockerfile by some other name instead of “Dockerfile”, say for example “Dockerfile_imgrec”, in that case run the following command:

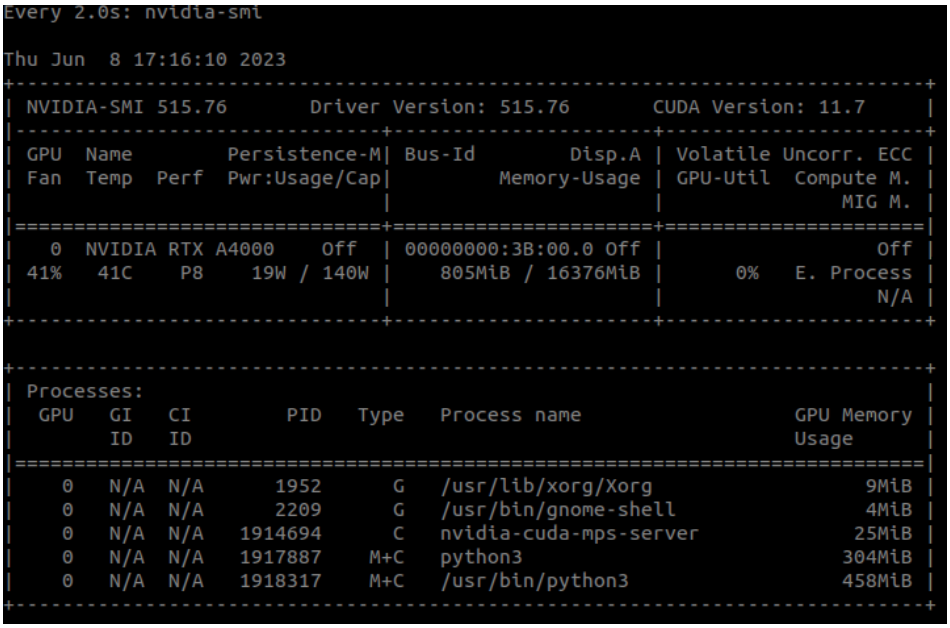
⇒ sudo docker build -f <Dockerfile_name> -t <image_name> .

- Monitor nvidia-smi and list of all containers:
  In a split terminal, run:
  ⇒ watch nvidia-smi
  ⇒ sudo docker watch ps -a

- Run the container from the built image:
  ⇒ sudo docker run -d --gpus all -p 5124:5124 --runtime=nvidia --ipc=host gpu-image-recognition-resnet

- Input: curl -X POST -H "Content-Type: application/json" -d '{"url": "http://images.cocodataset.org/val2017/000000039769.jpg"}' http://localhost:5124/predict

- Output: {
  "class": "tabby",
  "score": 8.045576095581055
  }
  Keep observing the nvidia-smi to watch how the python3, /usr/bin/python3 and nvidia-cuda-mps-sever processes spring up after providing the curl request.



You may repeat these steps for the remaining workloads. Find their particular codes and Dockerfiles within the respective directories. A readme.txt containing the build and run commands, has been attached for every workload's variant.
In addition a few more docker commands which might be useful are as follows:

To stop a running container:
⇒ sudo docker stop <container-id>

To remove a stopped container:
⇒ sudo docker rmi –force <container-id>

To view a container's logs in case it crashes or exits abruptly:
⇒ sudo docker logs <container-id>
If the container exits with code 139 it implies that a segmentation fault has occurred indicating you should share the the host's IPC with the container (--ipc=host in the docker command)

To view all docker images on the system:
⇒ sudo docker images

To execute commands within the container:
⇒ sudo docker exec -it <container-id> <command>

6. REFERENCES:

[1] NVIDIA's CUDA Toolkit Installation Guide:
https://developer.nvidia.com/cuda-downloads
[2] Docker Installation Guide:
https://docs.docker.com/engine/install/ubuntu/
[3] NVIDIA-Docker Installation Guide:
https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html