Dockerfile (Droid SLAM)

Tags docker droid-slam guide

Droid-SLAM docker

Guide to writing a good docker file

https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Writing the dockerfile

1. **Pick a base image**

Keeping the `base` image almost same as my system. Since Droid-SLAM is already working on my environment, so it is intuitive to keep use the same environment for the `container`.

```
FROM nvidia/cuda:11.8.0-devel-ubuntu22.04
```

Refer → https://hub.docker.com/r/nvidia/cuda/tags

2. **Metadata and LABELS**

Adding some labels to the `image`. *Metadata* helps to organize images by project, record licensing information, to aid in automation, or for other reasons.

```
# Metadata
LABEL maintainer="rajahaseeb147@gmail.com" \
    project-name="Droid-SLAM" \
    com.droid-slam.version="0.0.1-alpha" \
    com.droid-slam.realease-data="2023.01.20"
```

3. **Set arguments and environment variables**

Next, we set some arguments like here in this case `DEBIAN_FRONTEND` mode. We will be using a `noninteractive` front-end. You use this mode when you need *zero interaction while installing or upgrading the system via apt*. It accepts the default answer for all questions. It might mail an error message to the root user, but that's it all.

Otherwise, it is totally silent and humble, a perfect front-end for automatic installs. One can use such modes in Dockerfile, shell scripts, cloud-init scripts, and more.

We also need to set `NVIDIA_DRIVER_CAPABILITIES` to control which driver features are exposed to the container.

```
# Set arguments ane env variables
ARG DEBIAN_FRONTEND=noninteractive

ENV NVIDIA_VISIBLE_DIVICES all
ENV NVIDIA_DRIVER_CAPABILITIES graphics,utility,compute
```

Also check → https://vsupalov.com/docker-arg-vs-env/

4. **Nvidia keys**

Next we fetch the updated nvidia signing keys.

> Nvidia posted:
>
> To best ensure the security and reliability of our RPM and Debian package repositories, NVIDIA is updating and rotating the signing keys used by the `apt`, `dnf/yum`, and `zypper` package managers beginning April 27, 2022.
>
> If you don't update your repository signing keys, expect package management errors when attempting to access or install packages from CUDA repositories.

```
# Fetch nvidia signing keys
RUN apt-key del 7fa2af80
RUN apt-key adv --fetch-keys https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86_64/3bf863cc.pub
```

CUDA applications built using older NGC base containers may contain outdated repository keys. If you build Docker containers using these images as a base and update the package manager or install additional NVIDIA packages as part of your Dockerfile, these commands may fail as they would on a non-container system. To work around this, integrate the earlier commands into the Dockerfile you use to build the container.

Refer → https://developer.nvidia.com/blog/updating-the-cuda-linux-gpg-repository-key/

5. **Install base packages and utilities**

Next we install some debian base packages and command line utilities. To reduce complexity, dependencies, file sizes, and build times, avoid installing extra or unnecessary packages just because they might be “nice to have.”

```
# Install ubuntu base packages
RUN apt-get update && apt install -y --no-install-recommends \
    software-properties-common \
    dbus-x11 \
    libglvnd0 \
    libgl1 \
    libglx0 \
    libegl1 \
    libxext6 \
    libx11-6 \
    libgl1-mesa-dev \
    libglew-dev

RUN apt-get update && apt install -y --no-install-recommends \
    build-essential \
    libboost-all-dev \
    libsm6 \
    libxext6 \
    libxrender-dev \
    ninja-build

# Installing required utilities
RUN apt-get update && apt-get install -y --no-install-recommends \
    curl \
    gdown \
    git \
    ssh \
    unzip \
    vim \
    wget \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*
```

6. **Set working directory**

It is usually a good idea to set a working directory to install stuff and clone repositories. We can use `WORKDIR` command for it inside the docker file. It can also help you keep track of your directory locations.

> 💡 Avoid proliferating instructions like `RUN cd … && do-something`, which are hard to read, troubleshoot, and maintain.

```
# Set working directory
RUN mkdir -p /root/
WORKDIR /root/
```

7. **Install miniconda**

Next we are going to install **miniconda** inside our container. We can also do with pip but it depends on you. I use both.

Also sometimes we can make use of `environment.yml` file for which we need conda installation.

> 💡 The primary reason of using it over pip or pipenv is to install Python packages that have *non-Python dependencies (*written in C, Fortran or some other compilable language). Conda does this dependency resolution and source installation quite well. In fact chances are, if your pip installation is failing (for non-permissions reasons), Conda can probably help you.

```
# Set user to root to avoid permission fails
USER root

# Install miniconda
RUN wget \
    https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh && \
    bash Miniconda3-latest-Linux-x86_64.sh -b && \
    rm -f Miniconda3-latest-Linux-x86_64.sh

ENV PATH="/root/miniconda3/bin:${PATH}"
```

8. **Install application dependencies**

Next we install our main application dependencies which in my case is Droid SLAM.

```
# Install Droid SLAM dependencies
RUN conda install python=3.9

RUN conda install pytorch==1.10.1 torchvision==0.11.2 torchaudio==0.10.1 cudatoolkit=11.3 -c pytorch -c conda-forge

RUN pip install gdown matplotlib open3d opencv-python torch-scatter tensorboard scipy tqdm pyyaml

RUN conda install -c conda-forge suitesparse
```

9. **Clone the repository**

Now we have to clone all the required repositories. But before that, set the user to `root to avoid any permission issues.

```
# Clone repository
WORKDIR /root
RUN git clone --recursive https://github.com/pytholic/DROID-SLAM.git
```

10. **Install thirdparty dependencies**

```
# Install extensions
RUN cd DROID-SLAM && \
    python setup.py install

# Cleanup
RUN rm -rf DROID_SLAM
```

Here I am removing the cloned repository folder at the end. The reason to do that is that we don't want to develop inside the container. We just want the container to have all the dependencies. Later we will clone on the host machine and mount our local drive in the container and we can access it from within the container.

You can learn more about it at this link.

11. **Set command or entrypoint**

Finally we set the commands that we want to run by default when we start our container. We can use `CMD` or `ENTRYPOINT` for it. Another practice is to use them together. There is some nuance between the two which you can learn about just by googling it.

In my case I will just start bash CLI.

```
# Set entry commands
ENTRYPOINT ["/bin/bash"]
```

Notes

- If you are gonna use the cloned repository for development purposes, then it should not be inside your docker container. It should be on your `host` machine and then you can mount it to your docker container (Refer → https://docs.docker.com/storage/volumes/)
- In case the repository you want to clone is private, you will have to copy `ssh_keys` from host machine to the container. Something like:

```
# Make ssh dir
RUN mkdir /root/.ssh/

# Copy over private key, and set permissions
ADD id_rsa /root/.ssh/id_rsa
RUN chmod 700 /root/.ssh/id_rsaContinuing dockerfile.
```

- In case you face the following error

```
libGL error: MESA-LOADER: failed to open swrast
```

It's related to the following lines. Make sure you add them.

```
ENV NVIDIA_VISIBLE_DIVICES all
ENV NVIDIA_DRIVER_CAPABILITIES graphics,utility,compute
```

- In this example, there are some packages and libraries that you might not need like 3D rendering libraries, so make sure to install only those which you need.

Building image

```
docker build -t pytholic/droid-slam -f Dockerfile .
```

Running container

I wrote a small bash script to run the container.

```
#!/bin/bash

# start sharing xhost
xhost +local:root

# run docker
docker run \
  --name droid-slam \
  --gpus all \
  --privileged \
  -v /tmp/.X11-unix:/tmp/.X11-unix:rw \
  -v $HOME/.Xauthority:$HOME/.Xauthority \
  -v $HOME/projects/uav_mapping/DROID-SLAM:/root/DROID-SLAM \
  -e DISPLAY=$DISPLAY \
  -e XAUTHORITY=$HOME/.Xauthority \
  -e QT_X11_NO_MITSHM=1 \
  -it pytholic/droid-slam
```

I will mention some of the important things here.

-v → This flag is used to bind volumes

```
-v $HOME/projects/uav_mapping/DROID-SLAM:/root/DROID-SLAM \
```

Here I am mounting my local `dev` repo to the container so that I can access it within the container.

```
-v /tmp/.X11-unix:/tmp/.X11-unix:rw \
-v $HOME/.Xauthority:$HOME/.Xauthority \
-e XAUTHORITY=$HOME/.Xauthority
```

The above two commands are only useful if you need `X11` forwarding i.e. you need visualization. Here we need to give `.Xauthority` access to the docker.

Once you run the container, you can always start it.

```
docker start -ai <container>
```