FRACTIONAL GPU ALLOCATION TO CONTAINERS VIA MPS

1. **OBJECTIVE**
   To explore two approaches in allotting fractional GPU resources to Docker containers via MPS.

2. **BACKGROUND:**
   We explore two ways in which we may allocate GPU resources to Docker containers via MPS:
   a. Containerized Approach:
      In this approach, we start the MPS daemon in a “master” container, and connect other CUDA containers to it via the CUDA_MPS_PIPE_DIRECTORY environment variable.
      An overview of the steps to do the same, are as follows:
      - First, we create a Docker container specifically for running the MPS daemon. This container will have the necessary configurations and libraries to enable MPS functionality.
      - Within this container, we then start the MPS daemon, which manages the GPU resources and allows for resource sharing among multiple CUDA containers.
      - When creating CUDA containers that require GPU resources, we specify the CUDA_MPS_PIPE_DIRECTORY environment variable to point to the MPS daemon's pipe directory (/tmp/nvidia/mps). This allows the CUDA containers to communicate with the MPS daemon and request GPU resources.
      - ADVANTAGES:
        I. Isolation: Each CUDA container operates in its own isolated environment, separate from the MPS daemon and other containers. This isolation provides enhanced security and resource management, as issues in one container are less likely to affect others.
        II. Flexibility: With the containerized approach, you have more flexibility in allocating GPU resources to individual containers. You can adjust the resource allocation independently based on the specific requirements of each container.
        III. Scalability: It is relatively easy to scale the number of CUDA containers and adjust the resource allocation as

needed. You can spin up or down containers without affecting the global MPS daemon or other containers.

b. Non-containerized Approach:

In this approach, instead of running the MPS daemon within a specific container, we run it globally on the host machine or in a separate non-containerized environment. CUDA containers can then directly connect to the global MPS daemon.

An overview of the steps to do the same, are as follows:

- We start the MPS daemon globally on the host machine or in a separate non-containerized environment. This daemon manages the GPU resources and allows for resource sharing among CUDA containers running on the same host.
- When creating CUDA containers, we configure them to connect directly to the global MPS daemon, by specifying the necessary connection details (IP address or port) as environment variables or command-line parameters.
- ADVANTAGES:
  I. Simplified Setup: With the non-containerized approach, you eliminate the need to set up and manage an additional container specifically for the MPS daemon. This simplifies the overall setup process.
  II. Lower Overhead: By running the MPS daemon globally on the host machine or in a separate non-containerized environment, you may reduce resource overhead compared to running multiple instances of the MPS daemon within containers.
  III. Centralised Resource Management: With a global MPS daemon, you can have a centralised resource management system that oversees GPU allocations for all containers running on the host. This centralization simplifies resource management and monitoring.

3. SETUP DESCRIPTION

a. System Setup / Requirements:
   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 allocate GPU resources (via MPS) to a program “vect_add.cu” running within a CUDA container. We attempt to explore both approaches i.e the containerized as well as the non-containerized one.

4. 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. 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.

i. Containerized Approach:
Build "cuda_img" from "Dockerfile-cuda"
⇒ sudo docker build -t cuda-img -f Dockerfile-cuda .

Build "master_img" from "Dockerfile-master"
⇒ sudo docker build -t master-img -f Dockerfile-master .

Before running any containers set the GPU mode to “EXCLUSIVE” via MPS (but do not start the MPS Control Daemon):

⇒ sudo su -

⇒ export CUDA_VISIBLE_DEVICES="0"

⇒ nvidia-smi -i 0 -c EXCLUSIVE_PROCESS

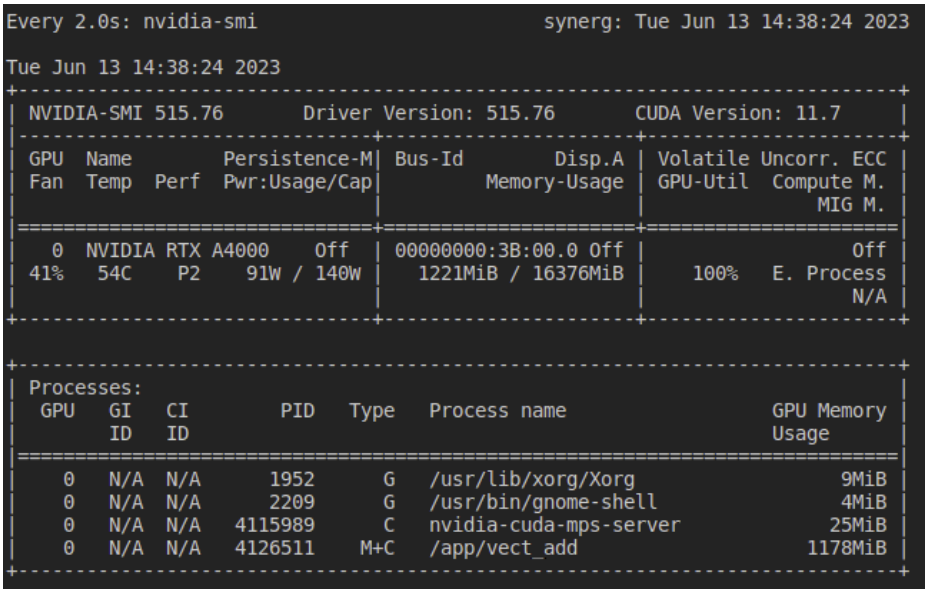
Run the master container first (in one terminal)

⇒ sudo docker run --gpus all --runtime=nvidia --ipc=host -v nvidia_mps:/tmp/nvidia-mps master-img

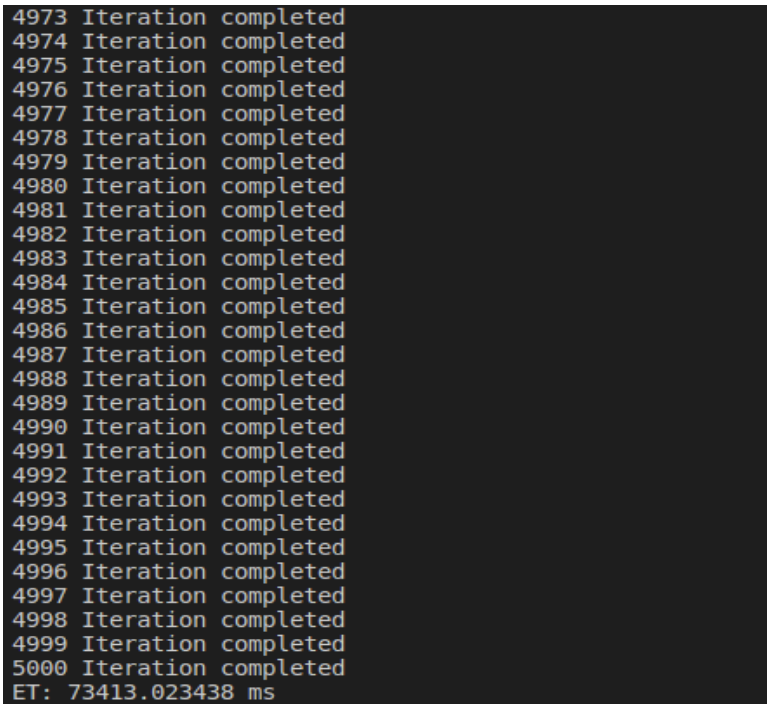
Run the CUDA container and allow it to connect to the master by specifying the path to the env variable (CUDA_MPS_PIPE_DIRECTORY in the dockerfile), besides the GPU allocation in terms of memory and cores (do this in a separate terminal):

⇒ sudo docker run --ipc=host --gpus all --runtime=nvidia -v nvidia_mps:/tmp/nvidia-mps --env CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=10 --env CUDA_MPS_PINNED_DEVICE_MEM_LIMIT='0=2G' cuda-img

Watch nvidia-smi and you will observe the nvidia-cuda-mps-server process spring up, once you launch the cuda container. You can also watch docker ps -a to get a list of all containers (running/stopped)



You can also observe that the CUDA program runs successfully (num iterations and overall execution time printed successfully)



ii. Non-Containerized Approach:

Under this approach, we drop the use of the master container to start our MPS Daemon and instead run it globally.

Firstly, start the MPS Daemon (use root privileges / sudo su -):

⇒ export CUDA_VISIBLE_DEVICES="0"

⇒ nvidia-smi -i 0 -c EXCLUSIVE_PROCESS

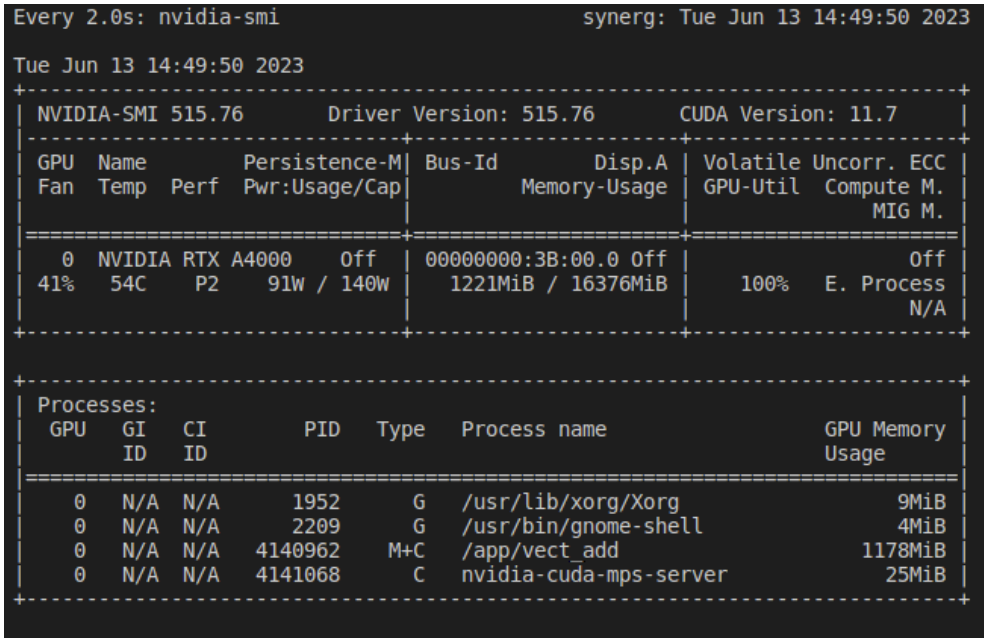
⇒ nvidia-cuda-mps-control -d

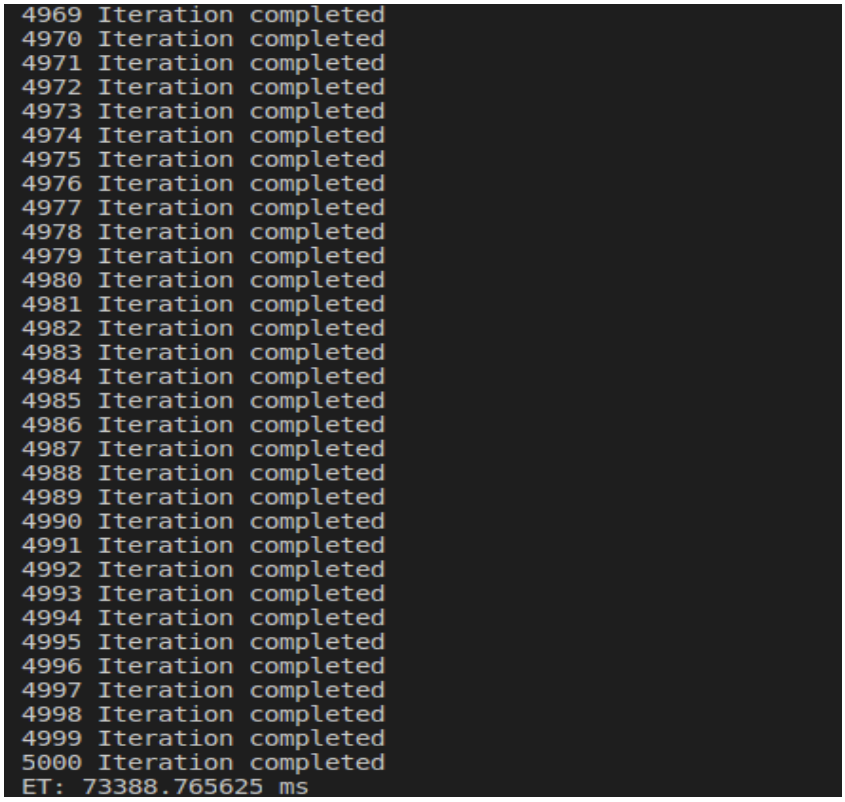
Next, in a separate terminal run the following:

⇒ export CUDA_MPS_PIPE_DIRECTORY=/tmp/nvidia-mps

⇒ sudo docker run --ipc=host --gpus all --runtime=nvidia -v nvidia_mps:/tmp/nvidia-mps --env CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=10 --env CUDA_MPS_PINNED_DEVICE_MEM_LIMIT='0=2G' cuda-img

We basically set the environment variable for the current terminal session followed by running our CUDA container. Once again, observe the nvidia-cuda-mps-server process spring up as you launch the cuda container. Ensure that the CUDA program runs successfully with the num iterations and execution time printed.





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>

5. 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