1. Ubuntu Server Installation on Bare Metal

Step 1 : Prepare Bootable USB

We need :

1. A windows system
2. A USB Drive
3. Rufus software
4. Ubuntu 22.04 server ISO image file (https://releases.ubuntu.com/22.04/)

Steps to create :

1. Insert USB in windows system
2. Open Rufus
3. Select the Ubuntu 22.04 ISO image file
4. Choose “GPT” partition scheme (UEFI) or “MBR”(BIOS)

Step 2: Boot Bare Metal Server Via USB

1. Plug USB into the Bare metal
2. Power on server
3. Press the key to open Boot Menu (F11, F9,ESC or DEL) depending on HPE model
4. Select USB device

Step 3: Install Ubuntu Server

1. Language and Keyboard : Choose English
2. Install Type : Choose Ubuntu Server —-------?(need GUI or not)
3. Network : Accept default DHCP or static IP if you want remote access —--------?
4. Storage Configuration: —--------?
   - Select Disk to install OS
   - Choose Custom if you want manual partitioning
   - Or just use guided with LVM (default)

   Note : Do not select the disk which is to be used as removable disk
5. User Setup:
   - Set Username and Password
   - Enable OpenSSH server for remote access
6. Updates :
   - Choose automatic updates

Step 3: First Boot

1. Remove the USB Drive
2. Reboot the server
3. Log in to the OS using credentials (username and password)

Step 4: Post Install Tasks

1. Sudo apt update && sudo apt upgrade -y
2. Sudo apt install net-tools htop curl git unzip -y

2. Install NVIDIA GPU Drivers on Ubuntu 22.04

Step 1: Identify Your GPU (GPU unknown most compatible GPU A100, A30 or V100)

Run : lspci | grep -i nvidia

Step 2: ensures you get tested drivers compatible with Docker + CUDA.

-sudo apt update && sudo apt install -y wget gnupg

-wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86_64/cuda-keyring_1.1-1_all.deb

sudo dpkg -i cuda-keyring_1.1-1_all.deb

sudo apt update

Step 3: Install the NVIDIA Driver (Recommended Version)

-sudo apt install -y nvidia-driver-550

(Note :Nvidia driver 500 is recommended for CUDA 12.2 and DeepStream 7+)

Step 4 : Reboot server

sudo reboot

Step 5: Verification of Installation:

nvidia-smi

You should see :
- your GPU list
- Driver version 550.XX
- CUDA version 12.2+

3. Install Full CUDA Toolkit

1. Run

   -sudo apt install -y cuda

   - Installs: nvcc (CUDA compiler)

2. Add cuda to the path :

   -echo ‘export PATH=/usr/local/cuda/bin:$PATH’ >> ~/.bashrc

   -echo ‘export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH’ >> ~/.bashrc

   -source ~/.bashrc

3. Verify installation :

   -nvcc –version

   (output: Cuda compilation tool , release 12.2 , V12.2.128)

4. Install Docker & NVIDIA Container Toolkit (CUDA 12.2 Compatible)

Step 1: Install Docker Engine:
- sudo apt update

```
- sudo apt install -y \
     ca-certificates \
     curl \
     gnupg \
    Lsb-release
```

Add GPG Key:

```
- sudo mkdir -p /etc/apt/keyrings

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

Add Docker repository:

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

Install Docker

```
  - sudo apt update

- sudo apt install -y docker-ce docker-ce-cli containerd.io
docker-buildx-plugin docker-compose-plugin
```

Enable and Start Docker :

```
- sudo systemctl enable docker

- sudo systemctl start docker
```

Test :

```
- docker -- version
```

Step 2: Install NVDIA Container Toolkit (for driver 550 , CUDA 12.2) - to enable –gpus all and CUDA access inside Docker containers

- distribution=$(. /etc/os-release; echo IDVERSION_ID)

-curl -s -L https://nvidia.github.io/libnvidia-container/gpgkey | sudo tee /etc/apt/keyrings/nvidia-container-toolkit.gpg > /dev/null

-curl -s -L https://nvidia.github.io/libnvidia-container/$distribution/libnvidia-container.list | \
sed 's|deb |deb [signed-by=/etc/apt/keyrings/nvidia-container-toolkit.gpg] |' | \
sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list > /dev/null

- sudo apt update

- sudo apt install -y nvidia-container-toolkit

- sudo nvidia-ctk runtime configure –runtime=docker

- sudo systemctl restart docker

Step 3: Test GPU Inside Docker

- docker run --rm --gpus all nvidia/cuda:12.2.0-runtime-ubuntu22.04 nvidia-smi

You should see:
GPU Name
Driver version: [550.XX](550.XX)
CUDA version : 12.2

Now next step will be divided into two tracks Because we need both docker setup and local python environment

5. Python environment and Libraries
Track 1 : Host-Level Python Environment (with Anaconda)

Step 1: Install Miniconda (Recommended)

- wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh

- bash Miniconda3-latest-Linux-x86_64.sh

Step 2: Verify conda installation

- conda –version

Step 3: Create Environment (CUDA compatible)

- conda create -n dl-env python=3.10 -y

- conda activate dl-env

Step 4: Install CUDA compatible Libraries

- pip install –upgrade pip

- pip install \
  torch torchvision torchaudio --index-url
  https://download.pytorch.org/whl/cu121 \
  tensorflow \
  jax[cuda12_pip] -f
  https://storage.googleapis.com/jax-releases/jax_cuda_releases.html \
  opencv-python-headless \
  transformers \
  xgboost \
  nltk \
  theano \
  wandb \
  chromadb \
  fastapi \
  uvicorn

Note : tensorflow will automatically use GPU if installed properly with driver and CUDA present (confirm with tf.config.list_physical_devices('GPU'))

Step 5: Confirm GPU Access

- python -c "import torch; print(torch.cuda.is_available())"

- python -c "import tensorflow as tf; print(tf.config.list_physical_devices('GPU'))"
  (output:
  True
  [PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
)

Track 2 : Docker Environment (Containerized Setup with GPU)

- mkdir dl-cuda-docker
- cd dl-cuda-docker
- nano Dockerfile
  Note for nano: press crlt+o to save, press Enter to confirm , press crlt+X to exit
- Put Docker file content as :

```
FROM nvidia/cuda:12.2.0-runtime-ubuntu22.04

# Install base utilities
RUN apt-get update && apt-get install -y \
    curl git wget unzip build-essential \
    python3 python3-pip python-is-python3

# Install Python dependencies
RUN pip install --upgrade pip


# 1. Install PyTorch from the custom PyTorch index
RUN pip install --prefer-binary --timeout 1000 --retries 5 \
```

```
    torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu121


# 2. Install everything else from PyPI
RUN pip install \
    tensorflow \
    jax[cuda12_pip] -f
https://storage.googleapis.com/jax-releases/jax_cuda_releases.htm
l \
    transformers \
    opencv-python-headless \
    wandb \
    nltk \
    xgboost \
    theano \
    chromadb \
    openai \
    fastapi \
    Uvicorn \
    theano-pymc

# DeepStream: Best installed on host or separate container with
NVIDIA support

# Create working directory
WORKDIR /workspace

CMD ["/bin/bash"]
```

- Build and Run image :
  - docker build -t dl-cuda-env .
  - docker run --rm -it --gpus all dl-cuda-env bash
- Inside the container , test commands :
  - python -c "import torch; print(torch.cuda.is_available())"
  - python -c "import tensorflow as tf; print(tf.config.list_physical_devices('GPU'))"

6. DCGM Export (NVIDIA GPU Monitoring)

Why:
- Exports GPU metrics (temperature, memory, usage) to Prometheus/Grafana

Where To install:
- As a **container** (preferred) — works cleanly with NVIDIA drivers inside Docker

Installation Steps:
- docker run -d --gpus all --restart unless-stopped \
  --name dcgm-exporter \
  -p 9400:9400 \
  nvcr.io/nvidia/k8s/dcgm-exporter:3.1.7-3.1.5-ubuntu22.04

Access GPU metrics at this :
- http://<server-ip>:9400/metrices

Test with :
- curl http://localhost:9400/metrics | grep gpu

7. MIG Profile Creation & Automation (for A100/A30)

Note this MIG setup is only for configuration :
- Ubuntu 22.04
- NVIDIA driver 550
- CUDA 12.2
- Targeting data center GPUs like A100 , A30 , or H100 (only these support MIG)

MIG: Multi-Instance GPU allows you to split one physical GPU (like A100) into smaller, isolated logical GPU instances (e.g., 3x 10GB GPUs from a 40GB A100).

Prerequisite:
- MIG-capable GPU (A100, A30, H100)
- NVIDIA driver ≥ 450.80.02(we are using Driver 550)
- CUDA ≥ 11(we are CUDA 12.2)
- Nvidia-smi CLI utility

Step 1: Enable MIG on Each GPU
- nvidia-smi -L (check GPU)
- sudo nvidia-smi -i 0 -mig 1 (enable MIG mode (e.g., on GPU 0))
- sudo nvidia-smi --gpu-reset -i 0 (Reboot or reset GPU)
- nvidia-smi -i 0 --query-gpu=mig.mode.current --format=csv (test output : enabled)

Step 2: Create MIG Instances (Manually First)

We will divide GPU 0 into profiles

Example : Create **1x 20GB + 2x 10GB** MIG instances

Commands to demonstrate example :

- sudo nvidia-smi mig-create-gpu-instance --gpu-instance-profile=3g.20gb -i 0
- sudo nvidia-smi mig-create-compute-instance -gi 0 -ci 0

- sudo nvidia-smi mig-create-gpu-instance --gpu-instance-profile=1g.10gb -i 0
- sudo nvidia-smi mig-create-compute-instance -gi 1 -ci 1

- sudo nvidia-smi mig-create-gpu-instance --gpu-instance-profile=1g.10gb -i 0
- sudo nvidia-smi mig-create-compute-instance -gi 2 -ci 2

Note : Each -gi ID maps to an instance . you can validate it with:

- nvidia-smi

Step 3: Automate with script + input .txt file

Note : Automation will be done according to requirements —------?

But for example requirement is just creating the instances with txt file

So for that lets have example .

Example mig-setup.txt :

```
# MIG Configuration: A100 on GPU 0
3g.20gb
1g.10gb
1g.10gb
```

Example setup-mig.sh

```
#!/bin/bash
GPU_ID=0

# Enable MIG mode
sudo nvidia-smi -i $GPU_ID -mig 1
```

```
sleep 2
sudo nvidia-smi --gpu-reset -i $GPU_ID
sleep 2

# Read profiles from file
INDEX=0
while read profile; do
  [[ "$profile" == \#* ]] && continue
  sudo nvidia-smi mig-create-gpu-instance
--gpu-instance-profile=$profile -i $GPU_ID
  sudo nvidia-smi mig-create-compute-instance -gi $INDEX -ci $INDEX
  INDEX=$((INDEX + 1))
done < mig-setup.txt
```

Make this executable:
- chmod +x setup-mig.sh ./setup-mig.sh

Note : These GUP instances are not persistent in nature so these will vanish after reboot of server for persistence you need to configure nvidia-persistenced or have to call this script while booting the system via systemd or rc.local (this can be done as per requirements —------?)

Step 4: Assigning GPU instances to docker containers

- docker run --gpus '"device=0:0"' nvidia/cuda:12.2.0-runtime-ubuntu22.04 nvidia-smi

8.DeepStream SDK Setup (For AI Video Analytics)

DeepStream is NVIDIA's end to end video analytics SDK
Version Description:

- DeepStream 6.4
- Ubuntu 22.04
- Driver > 525 (we are using 550)
- CUDA 12.2

DeepStream Installation on Host :

1. Add NVIDIA GPG Key

- wget
  https://nvidia.github.io/deepstream/deepstream-6.4/ubuntu22.04/nvidia-deepstream-6.4_6.4.0-1_amd64.deb
2. Install .deb Package
   - sudo apt install ./nvidia-deepstream-6.4_6.4.0-1_amd64.deb

3. Install dependencies:
   - sudo apt-get install -y \
     libgstreamer1.0-dev \
     libgstreamer-plugins-base1.0-dev \
     gstreamer1.0-plugins-good \
     gstreamer1.0-plugins-bad \
     gstreamer1.0-plugins-ugly \
     gstreamer1.0-libav
4. Test Example App:
   - cd
     /opt/nvidia/deepstream/deepstream-6.4/sources/apps/sample_apps/deepstream-test1
     make
     ./deepstream-test1 <video-file-path>

   DeepStream Installation on Docker :
   - docker pull nvcr.io/nvidia/deepstream:6.4-triton-multiarch
   - docker run --rm -it --gpus all \
     --entrypoint bash \
     nvcr.io/nvidia/deepstream:6.4-triton-multiarch
   - Inside a container run sample pipeline
     deepstream-app -c
     /opt/nvidia/deepstream/deepstream/samples/configs/deepstream-app/source1_usb_dec_infer_resnet_int8.txt

9.NVIDIA TensorRT

TensorRT is NVIDIA’s SDK for optimizing deep learning inference. It converts models (TF, ONNX, PyTorch → ONNX) into fast, GPU-efficient runtime.

- compatibility:
  TensorRT 10.0+ supports CUDA 12.X
  Ubuntu 22.04 is official supported

Works with Driver 550

- Install TensorRT on Host:
  sudo apt -y tensorrt
  Or
  wget
  https://developer.download.nvidia.com/compute/machine-learning/repos/ubuntu2204/x86_64/tensorrt-local-repo-ubuntu2204-10.0.1-cuda-12.2_1.0-1_amd64.deb
  sudo dpkg -i tensorrt-local-repo-*.deb
  sudo apt-key add /var/nv-tensorrt/*.pub
  sudo apt-get update
  sudo apt-get install -y tensorrt
- Docker TensorRT Container:
  docker pull nvcr.io/nvidia/tensorrt:24.05-py3
  docker run --rm -it --gpus all nvcr.io/nvidia/tensorrt:24.05-py3
  Inside container :
  python3 -c "import tensorrt as trt; print(trt.__version__)"

10. RAPIDS

RAPIDS is NVIDIA's GPU-accelerated data science stack (like pandas, NumPy, XGBoost) for dataframes, graphs, and ML — but GPU-native.

- Compatibility :
  RAPIDS 24.04 supports CUDA 12.2
  Supports Python 3.10 , Ubuntu 22.04
- Host Setup(conda recommended)
  conda create -n rapids-12 python=3.10 -y
  conda activate rapids-12
  conda install -c rapidsai -c nvidia -c conda-forge \
  rapids=24.04 \
  python=3.10 \
  cudatoolkit=12.2
- RAPIDS Docker Image
  docker pull
  rapidsai/rapidsai-core:24.04-cuda12.2-runtime-ubuntu22.04-py3.10
  docker run --rm -it --gpus all
  rapidsai/rapidsai-core:24.04-cuda12.2-runtime-ubuntu22.04-py3.10
  Test:
  python3 -c "import cudf; df = cudf.DataFrame({'a': [1, 2]}); print(df)"

11. OpenNN(C++ Neutral Network Library)

Install on Host:

- Install tools

  sudo apt install git cmake build-essential

- Clone repo

  git clone https://github.com/Artelnics/OpenNN.git

  Cd OpenNN

- Build with CMake

  mkdir build

  cd build

  cmake ..

  make -j$(nproc)

Note : You now have the OpenNN library compiled in `./build`. You can link it in C++ apps or wrap it into Python with PyBind11 (according to requirement——-?).

12. CROMA DB setup (Vector DB)

Install on host:

- pip install chromadb
- chromadb run --path ./chroma_store (To enable persistent storage)

Install on Docker : Already in our main docker image

-python3 -c "import chromadb; print(chromadb.__version__)" (check by this)

13.Weights and Biases (wandb)

wandb (Weights & Biases) is a tool to track machine learning experiments, logs, metrics, visualizations, model versions, and more. Commonly used with:

- PyTorch
- TensorFlow
- Keras

- HuggingFace Transformers

It integrates seamlessly with GPU training environments — both host and container.

Compatibility :

- Python 3.10+
- CUDA (auto detect GPU)
- Ubuntu 22.04
- Docker

Host Installation (via pip or conda)

- conda activate your_env_name
- pip install wandb
  Or
- pip install wandb(for global installation)

Docker Installation

- Our docker already had wandb

Initial Setup before using wandb

- Wandb login
  (this opens a URL where you sign in and paste the token back into terminal)

Note : wandb logs stores at path ~/.wandb/ for host

14.Theano

Compatibility :

- Ubuntu 22.04
- Python 3.10
- CUDA 12.2 - not compatible fully
- Docker

Install on host (anaconda Recomended)

- conda create -n theano-env python=3.10 -y
- conda activate theano-env
- pip install theano-pymc

Enabiling CUDA with theano

- Edit the .theanorc file (create if missing)
- Nano `/. theanorc\
- Add
  [global]
  device = cuda
  floatX = float32

  [nvcc]
  fastmath = True

  [cuda]
  root = /usr/local/cuda
  Docker steup:
    - Already configured
      python -c "import theano; print(theano.config.device)" - check with this command

15.XGBoost

- Host Installation
  For CPU only version
    - pip install xboost
  To enable GPU :
    - Use prebuild wheel (limited versions)
    - Build from source (recommended for CUDA 12.2 compatibility)
  Installing XGBoost with GPU (conda easiest)
    - conda install -c conda-forge py-xgboost cudatoolkit=12.2

      Or

    - conda install -c conda-forge compilers cmake git
    - git clone --recursive https://github.com/dmlc/xgboost
    - cd xgboost
    - mkdir build && cd build
    - cmake .. -DUSE_CUDA=ON
    - make -j$(nproc)
    - cd ../python-package
    - pip install .
- Docker Setup
  Already done
  But for full gpu support build inside docker file

```
RUN apt-get update && apt-get install -y git cmake build-essential
RUN git clone --recursive https://github.com/dmlc/xgboost
WORKDIR xgboost/build
RUN cmake .. -DUSE_CUDA=ON && make -j$(nproc)
WORKDIR ../python-package
RUN pip install .
```

16.NLTK (Natural Language Toolkit (NLP library))

Note: It's CPU-based, so doesn't need CUDA/GPU support.

- Host installation
  pip install nltk
- Docker installation
  Already installed in docker image
  But just add this in your docker file so that container has necessary corpora and doesn't require downloading at runtime.,
    - RUN python3 -m nltk.downloader punkt wordnet averaged_perceptron_tagger

17.Transformers

- CUDA compatibility via pytorch >= 2.1
- Host Installation
  pip install transformers
  pip install install accelerate optimum
- Docker Installation
  Already done
  But can add this in docker file to give better GPU usage
  RUN pip install accelerate optimum

18.TorchVision

- CUDA compatibility via cu121 build

- Host installation
  pip install torchvision --index-url https://download.pytorch.org/whl/cu121
- Docker installation
  Already done

19.DL4J (DeepLearning4J)- JAVA Based

- Compatibility
  - Ubuntu 22.04
  - CUDA 12.2 - limited (most stable with 11.x-12.0)
  - Docker
- Host Installation
  - sudo apt update
  - sudo apt install openjdk-17-jdk
  - setup Maven or Gradle Project
    Example pom.xml dependency

```
  <dependency>
<groupId>org.deeplearning4j</groupId>
  <artifactId>deeplearning4j-core</artifactId>
  <version>1.0.0-beta7</version>
</dependency>
```

For cuda support

```
<dependency>
  <groupId>org.nd4j</groupId>
  <artifactId>nd4j-cuda-12.0-platform</artifactId>
  <version>1.0.0-beta7</version>
</dependency>
```

- Docker Installation
  - Docker File

```
FROM openjdk:17

RUN apt-get update && apt-get install -y maven

COPY . /app
```

```
WORKDIR /app
RUN mvn install
```

Include your pom.xl and java code in /app

Example Java code

```
MultiLayerConfiguration config = new
NeuralNetConfiguration.Builder()
    .updater(new Adam(0.01))
    .list()
    .layer(new
DenseLayer.Builder().nIn(784).nOut(1000).activation(Activation.RE
LU).build())
    .layer(new
OutputLayer.Builder().nIn(1000).nOut(10).activation(Activation.SO
FTMAX).build())
    .build();

MultiLayerNetwork model = new MultiLayerNetwork(config);
model.init();
```

20.JAX

- Host Installation(inside conda or python venv)
  -pip install --upgrade "jax[cuda12_pip]" -f https://storage.googleapis.com/jax-releases/jax_cuda_releases.html
- Docker installation
  Already done

21. System Auto Start Scripting(at Boot)

Currently on start we can do these things : ——---- as per requirements

- GPU & MIG setup (if applicable)
- Docker containers (e.g., DeepStream, custom ML image)
- Background services (e.g., ChromaDB server, Jupyter)
- Logging/monitoring agents (e.g., DCGM Exporter)
- Host Python script

- Example boot up script
  - Create a shell script (boot.sh)

```
#!/bin/bash

# 1. Optional: Set up MIG profiles
nvidia-smi mig -cgi 19,19,19 -C 0

# 2. Start your main Docker container
docker start your_container_name

# 3. Optional: Start background service like ChromaDB
nohup chromadb run --host 0.0.0.0 --port 8000 &

# 4. Optional: Start Jupyter or custom Python script
# nohup jupyter lab --no-browser --port=8888 &
# nohup python3 your_script.py &

# 5. Log timestamp
echo "Boot script executed at $(date)" >>
/var/log/boot_script.log
```

- Give execution permission
  chmod +x /opt/boot.sh
- Register It as a Systemd Service
  sudo nano /etc/systemd/system/boot-init.service -(create service file)

```
[Unit]
Description=Custom Boot Startup Tasks
After=network.target docker.service

[Service]
Type=simple
ExecStart=/opt/boot.sh
RemainAfterExit=yes

[Install]
WantedBy=multi-user.target
```

- Enable and test
  sudo systemctl daemon-reexec
  sudo systemctl enable boot-init.service
  sudo systemctl start boot-init.service
  sudo systemctl status boot-init.service
  sudo reboot(reboot)
  cat /var/log/boot_script.log(Check logs after reboot)