Applied Containerization for Machine Learning in HPC

Research Computing
UNIVERSITY OF COLORADO BOULDER

Applied Containerization for Machine Learning in HPC

Date: October 21, 2025

Instructor: Mohal Khandelwal

- Website: www.rc.colorado.edu/rc
- Documentation: https://curc.readthedocs.io
- Helpdesk: rc-help@colorado.edu
- Survey: http://tinyurl.com/curc-survey18

Slides

https://github.com/ResearchComputing/applied_containerization_for_ml_short_course

Research Computing
UNIVERSITY OF COLORADO BOULDER

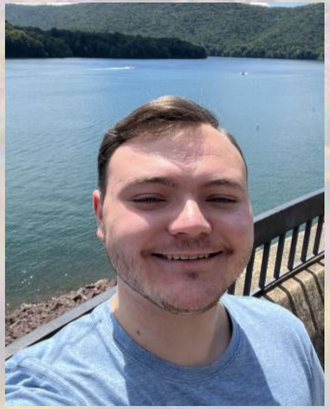
Meet the User Support Team

Layla
Freeborn

Brandon
Reyes

Andy
Monaghan

Michael
Schneider



John
Reiland

Dylan
Gottlieb

Mohal
Khandelwal

Ragan
Lee

Why Containers for ML in HPC?

- **Dependency Management**
  - Package complex ML software stacks with all dependencies.
  - Resolve version conflicts and avoid "dependency hell."
- **Portability**
  - Run the same environment across different HPC systems, local machines, or cloud platforms.
  - "Build once, run anywhere."
- **Scalability**
  - Easily scale workloads from a laptop to thousands of nodes.
  - Integrate seamlessly with schedulers like SLURM for large ML jobs.

Why Containers for ML in HPC? (cont.)

- **Reproducibility**
  - Capture the exact software environment for consistent, repeatable results.
- **Performance**
  - Near-native speed with minimal overhead; direct access to GPUs and high-speed interconnects.
  - Optimized images can boost job throughput and resource utilization.

What is a container?

- An isolated, encapsulated user-space instance.
- Runs on a shared OS kernel but has its own:
  - Filesystem
  - Processes
  - Network interfaces (can be configured)
- Container Image: A self-contained read-only file (or files) used to run the packaged application
- Container: A running instance of a container image

Containers vs. Virtual Machines (VMs)

Feature	Virtual Machine (VMs)	Containers
Isolation	Full (hardware/emulated)	Process/user-space
Guest OS	Full OS per VS	Shares host OS kernel
Setup	Slow (minutes)	Fast (seconds)
Resource Use	Heavy (more RAM/CPU)	Lightweight (minimal overhead)
Portability	Hypervisor-dependent	Build once, run anywhere
Use case	Legacy apps, OS-level isolation	ML, microservices, HPC, CI/CD

Key Components

- **Container Images**
  - Read-only templates used to create containers.
  - Contain application code, libraries, dependencies, and metadata.
  - Examples: Docker Hub images, SIF files (Apptainer).
- **Runtime Environments**
  - Software that runs containers (e.g., Apptainer, Docker).
  - Manages container lifecycle, isolation, and resource allocation.

Key Components (cont.)

- **Mount Points for Data (Bind Mounts)**
  - Mechanism to make host directories/files accessible inside the container.
  - Essential for accessing datasets, scripts, and output directories.
- **Resource Allocation**
  - Containers share host resources (CPU, memory, GPUs).
  - HPC schedulers (like SLURM) manage resource allocation for containerized jobs.

Apptainer (formerly Singularity)

- Apptainer is the direct successor to Singularity.
- Requires a Linux system
- Runs without requiring root access
- Single-file SIF format is easy to transport and share with others
- Can convert existing Docker images to Apptainer images
- Apptainer comes pre-installed on all Alpine compute nodes, so no need to load any specific software modules!

Using Apptainer on HPC

- View list of Apptainer commands.

```
$ apptainer --help
```

- Look at CURC's collection of pre-build containers.

```
$ echo $CURC_CONTAINER_DIR
$ ls $CURC_CONTAINER_DIR
```

- A Singularity Definition File (or “def file” for short) is a set of blueprints explaining how to build a custom container.
  - It includes specifics about the base OS, software to install, environment variables and other metadata.
  - https://apptainer.org/docs/user/1.0/definition_files.html

Using Apptainer on HPC

- By default, the cache directory for Apptainer builds is **/scratch/alpine/$USER**

```
$ echo $APPTAINER_CACHEDIR
```

- By default, only /home/$USER is available within any given container.
- To bind any additional folders/files to your container, use the -B flag in your apptainer run, exec, and shell commands:

```
$ apptainer run -B /source/directory:/target/directory sample-image.sif

$ apptainer run -B /projects/$USER,/pl/active,/scratch/alpine/$USER sample-image.sif
```

Apptainer Commands

Other useful Apptainer commands:

```
apptainer inspect        #See labels/environment vars, run scripts
apptainer pull           #pull an image from hub
apptainer exec           #Execute a command to your container
apptainer run            #Run your image as an executable
apptainer build          #Build a container
apptainer shell          #Access the command line of your container
```

Using images from a pre-built container

- You can fetch container images from container registries such as:
  - Docker Hub
  - NVIDIA NGC Catalog
- Let's walk through an example of using pre-built Tensorflow container to train a classification model

TensorFlow Example with Apptainer

```
1. Export paths
$ export IMAGES=/projects/$USER/containers/
$ export WORKDIR=/projects/$USER/containers/ml_work
$ mkdir -p $IMAGES $WORKDIR && cd $WORKDIR

2. Pull TensorFlow container (GPU version)
$ apptainer pull $IMAGES/tensorflow-2.15.0-cuda12.8.sif
docker://tensorflow/tensorflow:2.15.0-gpu

3. Inspect container metadata
$ apptainer inspect $IMAGES/tensorflow-2.15.0-cuda12.8.sif
```

TensorFlow Example (cont.)

- Bind mount your script directory.
- Create minst_tf.py that builds and trains a simple image classification model
- --nv: Enables NVIDIA GPU access.

```
4. Bind paths

$ apptainer exec -B $WORKDIR:/work,$IMAGES:/images $IMAGES/tensorflow-2.15.0-
cuda12.8.sif ls /work


5. Run the script

$ apptainer exec --nv -B $WORKDIR:$WORKDIR $IMAGES/ tensorflow-2.15.0-
cuda12.8.sif python3 $WORKDIR/mnist_tf.py
```

Example python script

```
cat >mnist_tf.py <<'PY'
import tensorflow as tf
mnist = tf.keras.datasets.mnist
(x_train, y_train), _ = mnist.load_data()
x_train = x_train / 255.0
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
model.fit(x_train, y_train, epochs=1)
print("Done training.")
PY
```

Access container interactively

- Shell into it

```
$ apptainer shell --nv $IMAGES/tensorflow-2.15.0-cuda12.8.sif
```

- Then run

```
$ python3
>>> import tensorflow as tf
>>> tf.__version__
>>> tf.config.list_physical_devices('GPU')
```

If it returns a list (e.g. [PhysicalDevice(name='/physical_device:GPU:0', ...)]), then TensorFlow successfully detected the GPU inside the container!

SLURM Integration

```
#!/bin/bash
#SBATCH --gres=gpu:1
#SBATCH --partition=aa100
#SBATCH --ntasks=4
#SBATCH --nodes=1
#SBATCH --qos=normal
#SBATCH --time=1:00:00
#SBATCH --job-name=tf-classify
#SBATCH --output=tf-classify.%j.out
#SBATCH --mail-type=ALL
#SBATCH --mail-user=<your email>


export IMAGES=/projects/$USER/containers/sif
export WORKDIR=/projects/$USER/containers/ml_work
mkdir -p $IMAGES $WORKDIR && cd $WORKDIR

apptainer exec --nv -B $WORKDIR:$WORKDIR $IMAGES/tensorflow-2.20.0.sif
python3 $WORKDIR/mnist_tf.py
```

Troubleshooting

- Check GPU access:
  - **`apptainer exec --nv $IMAGES/tensorflow-2.15.0-cuda12.8.sif python3 -c "import tensorflow as tf; print(tf.config.list_physical_devices('GPU'))"`**
- File system binding:
  - Ensure your data/scripts are accessible inside the container.
- Environment variables:
  - Pass with --env or set inside the container.
- Cache management:
  - apptainer cache list and apptainer cache clean
- Resource monitoring:
  - Use htop, nvidia-smi, or squeue to monitor jobs.

Pytorch Example with Apptainer

```
1. Export paths
$ export IMAGES=/projects/$USER/containers/
$ export WORKDIR=/projects/$USER/containers/ml_work
$ mkdir -p $IMAGES $WORKDIR && cd $WORKDIR

2. Pull TensorFlow container (GPU version)
$ apptainer pull $IMAGES/pytorch-2.9.0-cuda12.8.sif docker://pytorch/pytorch:2.9.0-
cuda12.8-cudnn9-runtime

3. Inspect container metadata
$ apptainer inspect $IMAGES/ pytorch-2.9.0-cuda12.8.sif
```

Pytorch Example (cont.)

- Bind mount your script directory.
- Create polyfit.py that builds and trains a small neural network

```
4. Bind paths

$ apptainer exec -B $WORKDIR:/work,$IMAGES:/images $IMAGES/pytorch-2.9.0-
cuda12.8.sif ls /work


5. Run the script

$ apptainer exec --bind $WORKDIR:$WORKDIR $IMAGES/pytorch-2.9.0-cuda12.8.sif
python3 $WORKDIR/polyfit.py
```

Sample python script

```
cat >polyfit.py <<'PY'
import torch
x = torch.linspace(-1, 1, 100).unsqueeze(1)
y = 3 * x ** 2 + 2 * x + 1 + 0.1 * torch.randn(x.size())
model = torch.nn.Sequential(torch.nn.Linear(1, 10), torch.nn.ReLU(),
torch.nn.Linear(10, 1))
loss_fn = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters())
for epoch in range(100): optimizer.zero_grad(); loss_fn(model(x),
y).backward(); optimizer.step()
print("Done training.")
PY
```

Best Practices: Container Management

- **Version Control & Tagging**
  - Tag containers with specific versions (e.g., myimage:1.0.0, myimage:latest-cuda11.8).
  - Store definition files (.def, Dockerfiles) in version control (Git).
- **Data Management**
  - Use bind mounts for large datasets to avoid including them in images.
  - Keep images small and focused on software environment.
- **Resource Allocation**
  - Match container resource needs to SLURM (or other scheduler) requests.
  - Avoid over-subscribing resources.

Any Questions

Thank you!

Survey and feedback

http://tinyurl.com/curc-survey18